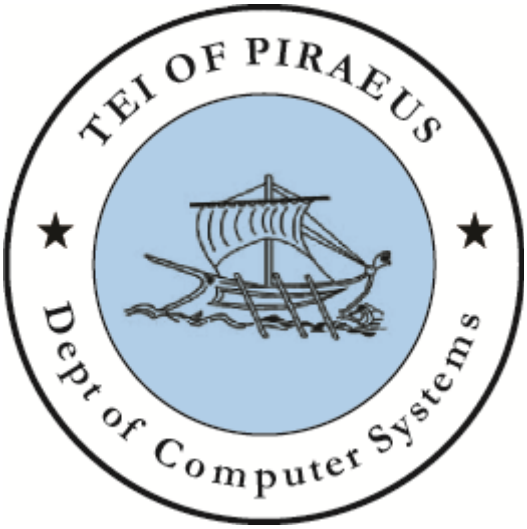


ΑΕΙ ΠΕΙΡΑΙΑ Τ.Τ.

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ Τ.Ε.

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάπτυξη Game Engine σε C++

Ιωάννης Γ. Παναγιωτόπουλος

Εισηγητής: Πρεζεράκος Γεώργιος, Καθηγητής

ΑΘΗΝΑ
ΙΑΝΟΥΑΡΙΟΣ 2017

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάπτυξη Game Engine σε C++

Ιωάννης Γ. Παναγιωτόπουλος
Α.Μ. 39572

Εισηγητής:

Πρεζεράκος Γεώργιος, Καθηγητής

Εξεταστική Επιτροπή:

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΠΤΥΧΙΑΚΗΣ ΕΡΓΑΣΙΑΣ

Ο/Η κάτωθι υπογεγραμμένος Παναγιωτόπουλος Ιωάννης, του Γεώργιου, με αριθμό μητρώου 39572 φοιτητής του Τμήματος Μηχανικών Η/Υ Συστημάτων Τ.Ε. του Α.Ε.Ι. Πειραιά Τ.Τ. πριν αναλάβω την εκπόνηση της Πτυχιακής Εργασίας μου, δηλώνω ότι ενημερώθηκα για τα παρακάτω:

«Η Πτυχιακή Εργασία (Π.Ε.) αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο του συγγραφέα, όσο και του Ιδρύματος και θα πρέπει να έχει μοναδικό χαρακτήρα και πρωτότυπο περιεχόμενο.

Απαγορεύεται αυστηρά οποιοδήποτε κομμάτι κειμένου της να εμφανίζεται αυτούσιο ή μεταφρασμένο από κάποια άλλη δημοσιευμένη πηγή. Κάθε τέτοια πράξη αποτελεί προϊόν λογοκλοπής και εγείρει θέμα Ηθικής Τάξης για τα πνευματικά δικαιώματα του άλλου συγγραφέα. Αποκλειστικός υπεύθυνος είναι ο συγγραφέας της Π.Ε., ο οποίος φέρει και την ευθύνη των συνεπειών, ποινικών και άλλων, αυτής της πράξης.

Πέραν των όποιων ποινικών ευθυνών του συγγραφέα σε περίπτωση που το Ίδρυμα του έχει απονείμει Πτυχίο, αυτό ανακαλείται με απόφαση της Συνέλευσης του Τμήματος. Η Συνέλευση του Τμήματος με νέα απόφασης της, μετά από αίτηση του ενδιαφερόμενου, του αναθέτει εκ νέου την εκπόνηση της Π.Ε. με άλλο θέμα και διαφορετικό επιβλέποντα καθηγητή. Η εκπόνηση της εν λόγω Π.Ε. πρέπει να ολοκληρωθεί εντός τουλάχιστον ενός ημερολογιακού 6μήνου από την ημερομηνία ανάθεσης της. Κατά τα λοιπά εφαρμόζονται τα προβλεπόμενα στο άρθρο 18, παρ. 5 του ισχύοντος Εσωτερικού Κανονισμού.»

ΠΕΡΙΛΗΨΗ

Το παρών έγγραφο αποτελεί την πιο δυνατή προσπάθεια ανάλυσης της λειτουργίας της Engine. Το έγγραφο λοιπόν χωρίζεται σε 3 μεγάλα κομμάτια.

1. **Εισαγωγή**

Ξεκινάω με μια περιγραφή της, καθώς και την απαρίθμηση των επιμέρους Modules που την απαρτίζουν. Εξηγούμε κάποια βασικά πράγματα γύρω από το Game Development και λίγες λειτουργίες των κύριων βιβλιοθηκών. Επίσης παραθέτω κάποια βήματα για την δημιουργία ενός περιβάλλοντος για εργασία πάνω στην Engine.

2. **Περιγραφή της διαδικασίας δημιουργίας ARPG παιγνίων**

Σε αυτό το κομμάτι ξεκινάω να περιγράφω βήμα-βήμα το πώς φτιάχνεται ένα παιχνίδι από την αρχή. Σε κάθε βήμα, αναλύονται οι δομές δεδομένων που δέχεται η Engine καθώς και η συμπεριφορές τους. Επίσης γίνονται παρουσιάσεις χρήσης κάποιων εργαλείων που βοηθάνε στην διαδικασία αυτή. Στο τέλος του κομματιού αυτού ο αναγνώστης έχει τις απαραίτητες γνώσεις να ξεκινήσει να φτιάξει ένα απλό παιχνίδι.

3. **Ανάλυση του δυαδικού πυρήνα**

Σε αυτό το κομμάτι αρχίζω να αναλύω τις λειτουργίες και τις συμπεριφορές των επιμέρους κομματιών του πυρήνα της Engine. Γίνεται εκτενή παράθεση κώδικα C++ και τύπων γραμμικής άλγεβρας. Ο σκοπός αυτής της ενότητας είναι να γίνει πλήρης τεκμηρίωση μέρους του κώδικα που θεωρείται απαραίτητο για κάποιον που θέλει να επεκτείνει την Engine από την πλευρά του πυρήνα.

Abstract

This document is the best possible way to analyze the operation of the Game Engine that I developed. This document is split into 3 big parts:

1. **Introduction**

This part contains an introduction to the Game Engine, and a brief mention to the modules that consists it. We explain some fundamental things around Game development, and part of operation of the libraries used. Furthermore, there a small tutorial on how to set up a brief development environment for the Game Engine

2. **ARPG Game development procedure**

In this part, we describe the procedure of creating an ARPG Game from the start step by step. In every step, we describe the data structures that the Engine uses as well as their behavior. Also, there is a brief introduction on each tool used to create the data structures. At the end of this part, someone can begin creating his/her own simple Game.

3. **Binary Core Analysis**

In this part, we analyze the core of the Engine and each part of it. There is an extender quotation of C++ code and Linear Algebra equations. The purpose of this part, is to be the most possible documentation cover-up of the Code, which is necessary for creating features from the Core-side of the Engine.

Επιστημονική περιοχή: Ανάπτυξη εφαρμογών σε C++

Λέξεις κλειδιά: Παιχνίδι, πρόγραμμα, περιγραφή δεδομένων, Game Engine

ΠΕΡΙΕΧΟΜΕΝΑ

Περιεχόμενα

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

Ευρετήριο εικόνων

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

Ευρετήριο πινάκων

ΣΥΝΤΟΜΟΓΡΑΦΙΕΣ

RPG: Role Playing Game

ARPG: Action Role Playing Game

NPC: Non-Player Controled

SZLL: Standard Zeta Lua Library

GUI: Graphical User Interface

GPU: Graphics Process Unit

VRAM: Video Random Access Memory

ΜΕΡΟΣ ΠΡΩΤΟ: ΕΙΣΑΓΩΓΗ

1 ΕΙΣΑΓΩΓΗ

1.1 Τι είναι ARPG Engine

ARPG Engine είναι μια “μηχανή” για την ανάπτυξη ARPG παιχνιδιών. Τώρα κάποιος θα ρωτήσει “Τι είναι ARPG παιχνίδια;” ή “Τι είναι η μηχανή;” και πολύ καλά θα κάνει!

Τα αρχικά ARPG προέρχονται από τις αγγλικές λέξεις: Action Role Playing Game. Σε ελεύθερη απόδοση αυτό σημαίνει: Παιχνίδι ρόλων και δράσης. Τα παιχνίδια ρόλων είναι παιχνίδια όπου ο χρήστης καλείται να πάρει κάποιον ρόλο ενός χαρακτήρα του παιχνιδιού και να κάνει κάποια πράγματα που του ανατίθενται. Οι κανόνες και ο τρόπος παιχνιδιού μπορεί να διαφέρει από παιχνίδι σε παιχνίδι. Μεγάλο μέρος του παιχνιδιού (και σε μερικά παιχνίδια αποτελεί και την συντριπτική πλειοψηφία του παιχνιδιού) παίζει η ιστορία που κρύβεται πίσω από το παιχνίδι. Αυτή η ιστορία επηρεάζει και παρακινεί τον παίχτη να “βαδίσει” μέσα στο παιχνίδι και να αναπτυχθεί. Τα RPG ξεκίνησαν σαν επιτραπέζια και σήμερα είναι επί τον πλείστων ηλεκτρονικά (Video Games). Με αυτά θα ασχοληθούμε μόνο.

Τα ηλεκτρονικά RPG παιχνίδια έχουν αρκετές κατηγορίες που ανάλογα την κατηγορία εμφανίζει κάποια χαρακτηριστικά. Σχεδόν όλα μοιράζονται κάποια χαρακτηριστικά. Στα περισσότερα ο παίχτης χρειάζεται να “ανεβάσει” τον χαρακτήρα του επίπεδα ώστε να πετύχει κάποιο στόχο. Ο χαρακτήρας βρίσκεται σε ένα κόσμο και συνήθως κινείται ελεύθερα. Μπορεί να έρθει σε επαφή με άλλους χαρακτήρες του παιχνιδιού (NPC), να σκοτώσει εχθρούς και να διεκπεραιώσει αποστολές. Ανάλογα με τη κατηγορία του παιχνιδιού, κάποια από αυτά ίσως διαφέρουν. Πχ: Στα JRPG (Japanese RPG) ο χαρακτήρας συνήθως είναι ακίνητος στην μάχη και γίνεται σε σύστημα γύρων σαν το σκάκι. Κάθε πλευρά έχει τον γύρο του για να κάνει ότι είναι να κάνει. Στα δυτικά παιχνίδια αυτό δεν υπάρχει. Και οι δυο πλευρές μπορούν να επιτεθούν ανά πάσα στιγμή χωρίς να περιμένουν τον γύρο τους.

Τα Action RPG είναι μια κατηγορία που μοιάζει πολύ με το δυτικό τρόπο παιχνιδιού που προαναφέραμε. Εδώ ο παίχτης έχει τον απόλυτο έλεγχο του χαρακτήρα του παιχνιδιού και επικεντρώνεται στην κυρίως δράση του παιχνιδιού. Ο χρήστης συνήθως καλείται να “καλλιεργήσει” τον χαρακτήρα του όσο πιο καλά γίνεται μακροπρόθεσμα ώστε να έχει το καλύτερο αποτέλεσμα βραχυπρόθεσμα.

Το Project όμως δεν λέγετε ARPG Game, άλλα ARPG Game “Engine”, και αυτό θα αναλύσουμε. Προς το παρών ας εξηγήσουμε το Game Engine ή στα ελληνικά “Μηχανή παιχνιδιών”. Για λόγους συντομίας θα αναφερόμαστε σε αυτή σαν “Μηχανή” από εδώ και πέρα. Μια μηχανή παιχνιδιών μπορεί να παίζει από ελάχιστο ρόλο έως και μεγάλο ρόλο σε ένα παιχνίδι. Συνήθως παρέχουν λειτουργίες για την εμφάνιση γραφικών στην οθόνη, το παίξιμο των ήχων, τον εντοπισμό των συγκρούσεων, την εμφάνιση του κόσμου, τον υπολογισμό των σκιών κτλ. Κάποιες άλλες παρέχουν πιο υψηλόβαθμες λειτουργίες όπως δημιουργία βασικών χαρακτήρων ή ακόμα πιο υψηλόβαθμες όπως μηχανές για να φτιάχνεις συγκεκριμένο είδους παιχνιδιών όπως είναι και η εν λόγο.

1.2 Σκοπός

Ο σκοπός της εν λόγο μηχανής είναι κυρίως να παρέχει σε επερχόμενους προγραμματιστές παιχνιδιών (Game Developers) μια απλή λύση για να μπορούν να φτιάχνουν -σχετικά πάντα- εύκολα δισδιάστατα παιχνίδια ARPG. Στόχος είναι να απαλλάξουμε τον προγραμματιστή από δύσκολες λειτουργίες του προγραμματισμού παιχνιδιών και να τον αφήσουμε να εργαστεί κυρίως πάνω στο περιεχόμενο του παιχνιδιού. Έτσι απελευθερώνουμε τους προγραμματιστές από την υποχρέωση να μάθουν λειτουργίες χαμηλού επιπέδου που απαιτούν μαθηματικά και γνώσεις αρχιτεκτονικής του συστήματος που προγραμματίζει. Με αυτό τον τρόπο ο προγραμματιστής αφιερώνει όλο τον χρόνο στο περιεχόμενο.

Η ολική απαλλαγή από τις πολύπλοκες λειτουργίες είναι δύσκολο να επιτευχθεί. Αυτό γιατί οι απαιτήσεις συνεχώς αλλάζουν και δεν μπορεί να προβλεφθεί κάθε σενάριο. Αν κάποιος προγραμματιστής χρειάζεται να φτιάξει κάτι που δεν παρέχεται από την μηχανή απευθείας, τότε καλείτε να σκαρφιστεί κάποια τεχνική ή ακόμα και να επεξεργαστεί τον πυρήνα για να μπορέσει να το φτιάξει. Όπως και να έχει, αν κάποιο Project έχει κοινό που χρειάζεται κάτι που λείπει από την μηχανή, τότε ο προγραμματιστής ή η ομάδα θα το προσθέσει σε μελλοντική ενημέρωση. Το ίδιο ισχύει και για την υπάρχουσα μηχανή. Αν κάτι θεωρηθεί ότι λείπει, τότε θα προστεθεί όσο πιο σύντομα.

Η εν λόγω μηχανή έχει σκοπό να χρησιμοποιηθεί σαν βιβλιοθήκη πάνω σε άλλο Project. Ο προγραμματιστής θα πρέπει να την συνδέσει (Link) στο εκτελέσιμο του (Executable).

1.3 Άδεια χρήσης

Ο κώδικας της μηχανής είναι ανοικτός και διαθέσιμος υπό την άδεια MIT, που παρατίθεται παρακάτω.

```
The MIT License

Copyright (c) 2014 Ioannis G. Panagiotopoulos (AKA Klapeto)

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated
documentation files (the "Software"), to deal in the Software without
restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software,
and to permit persons to whom the Software is furnished to do so, subject
to the following conditions.

The above copyright notice and this permission notice shall be included
in all copies or substantial
portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE
AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

Η MIT άδεια επιτρέπει ότι επιτρέπουν και οι άλλες άδειες ανοικτού λογισμικού (Open-Source). Η εξαίρεση με τις άδειες GPL είναι ότι αν συνδέσεις την μηχανή στο πρόγραμμα σου, δεν θα έχεις την υποχρέωση να το διανείμεις σαν ανοικτό λογισμικό υπό την ίδια άδεια.

Αυτό έγινε με την ιδέα ότι κάποιος μπορεί να φτιάξει ένα παιχνίδι για την μηχανή και να μπορεί να το πουλήσει, πουλώντας το περιεχόμενο μόνο και διανέμοντας το μαζί με την μηχανή σαν βιβλιοθήκη.

1.4 Ονομασία

Όπως κάθε πρόγραμμα ανοικτού λογισμικού έτσι και αυτή η μηχανή, θα πρέπει να φέρει ένα όνομα. Το κωδικός-όνομα που επιλέχτηκε για την μηχανή είναι “Μηχανή Ζήτα” ή στα Αγγλικά “Zeta Engine”. Το όνομα προήλθε από το ελληνικό γράμμα Ζ.

1.5 Πλατφόρμα/Αρχιτεκτονική

Στην θεωρία η μηχανή απαρτίζεται από κομμάτια και βιβλιοθήκες που είναι Cross-Platform(είναι συμβατές με πολλές πλατφόρμες). Αυτό στην θεωρία την κάνει Cross-Platform σε συνδυασμό την συγγραφή της σε C++. Στην πραγματικότητα αυτό είναι πιο δύσκολο από ότι φαίνεται. Σε πλατφόρμες GNU/Linux η μηχανή θα τρέξει με το κατάλληλο Build Configuration. Σε περιβάλλοντα Windows χρειάζεται πιο πολύ προσπάθεια. Αυτό έχει να κάνει με το ότι είναι δύσκολο να χτίσεις(Build) προγράμματα στα Windows αν δεν είναι σε Projects του Visual Studio. Στο μέλλον η μηχανή θα γίνει μεταφερθεί (Port ή μεταφορά κώδικα σε πλατφόρμα) σε Visual Studio.

Η μηχανή έχει δοκιμαστεί και λειτούργησε στα παρακάτω συστήματα:

- GNU/Linux Ubuntu 14.04 x86-64 (Unity Interface)
- GNU/Linux Mint 17.1 x86-64 (Cinnamon Interface)
- GNU/Linux Lubuntu x86-64 (LXDE Interface)
- GNU/Linux Fedora 21 x86-64 (Gnome Interface [X11])

Να σημειωθεί ότι η μηχανή χρησιμοποιεί τον X11 (Xorg) (Παραθυρικό περιβάλλον) σε GNU/Linux συστήματα. Δεν έχει δοκιμαστεί σε συστήματα με Wayland ή Cannonical Mir, αλλά εάν τα εν λόγω παραθυρικά περιβάλλοντα είναι συμβατά προς με παλιές εκδόσεις (Backwards Compatible) του X11 τότε ίσως δεν υπάρχει πρόβλημα. Πέρα από αυτό αν χρησιμοποιηθεί SDK που το υποστηρίζει (πχ SDL 2.x), τότε το πρόβλημα εξαλείφεται. Προς το παρόν το SDK που χρησιμοποιείται (Allegro5) υποστηρίζει μόνο X11 στα GNU/Linux.

Όσο αφορά την αρχιτεκτονική, αυτό που πρέπει να τονιστεί είναι ότι η μηχανή παρέχεται σε μορφή πηγαίου κώδικα (Source Code) που σημαίνει ότι δεν είναι χτισμένη σε κάποια συγκεκριμένη αρχιτεκτονική υπολογιστών. Ο προγραμματιστής παίρνει τον κώδικα και τον χτίζει στην αρχιτεκτονική του συστήματός του ή του συστήματος που θέλει. Παρόλο αυτά, κατά την συγγραφή της χρησιμοποιήθηκε η αρχιτεκτονική X86-64 γιατί είναι η πιο διαδεδομένη στους υπολογιστές καθώς και από τις πιο γρήγορες όσο αφορά το υλικό που την υποστηρίζει.

1.6 Γλώσσα υλοποίησης

Ο πυρήνας(Core) της μηχανής είναι γραμμένος σε C++. Αυτό έγινε γιατί χρειάζεται μια γρήγορη γλώσσα. Πολλές γλώσσες χρειάζονται έναν interpreter για να τρέξουν. Δηλαδή ο κώδικας μεταφράζεται κατά την εκτέλεση. Αυτό σε εφαρμογές πραγματικού

χρόνου(Real Time) όπως η μηχανή θα επιφέρει καθυστερήσεις που μπορεί να είναι εμφανές στον χρήστη, ιδιαίτερα αν η μηχανή τρέχει σε παλιό υλικό. Η γλώσσα C μεταγλωττίζεται απευθείας σε γλώσσα μηχανής. Αυτό κάνει τα προγράμματα που γράφτηκαν με αυτή να τρέχουν όσο πιο γρήγορα γίνεται. Επειδή όμως η απλή C δεν βολεύει στην συγγραφή της μηχανής, χρειαζόμαστε την αντικειμενοστραφή εκδοχή της γλώσσας, την C++. Πέρα από αυτό, η πλειοψηφία των βιβλιοθηκών που υπάρχουν είναι γραμμένες σε C/C++. Έτσι δεν χρειάζεται να εφαρμοστούν δεσμευτές γλωσσών (Language Binders) που θα προσθέσουν επιπλέον βάρος(Overhead) σε όλες τις κλήσεις. Με την προϋπόθεση ότι θα πρέπει να παρέχει στον παίχτη σταθερή απόδοση, η ταχύτητα του πυρήνα είναι σημαντικό θέμα. Τα αντικειμενοστραφή στοιχεία της C++ σε συνδυασμό με την ταχύτητα της C είναι τέλειος συνδυασμός.

Το πρότυπο της C++ που χρησιμοποιείται είναι το C++11. Παρέχει πολλά ενδιαφέροντα νέα στοιχεία στην γλώσσα και χρησιμοποιούνται συνέχεια στην μηχανή.

Πέρα από τον πυρήνα, η μηχανή αποτελείται επίσης από την “Στάνταρ Βιβλιοθήκη Lua της Μηχανής Ζήτα” (Standard Zeta Lua Library, SZLL). Η SZLL είναι γραμμένη σε γλώσσα Lua. Η Lua είναι μια γλώσσα σεναρίων (Scripting Language) που είναι σχετικά απλή αλλά παρέχει μηχανισμούς για πιο προηγμένα πράγματα όπως αντικειμενοστραφή προγραμματισμό. Είναι πιο κοντά στην ανθρώπινη γλώσσα σε σχέση με την C++ και δεν χρειάζεται ρητές μετατροπές (Explicit casting) για να πάρεις το αποτέλεσμα που θες. Πέρα από αυτό, όπως προαναφέραμε είναι γλώσσα σεναρίων που σημαίνει ότι το πρόγραμμα γράφεται σε απλό κείμενο και ύστερα το μεταγλωττίζει κατευθείαν η Lua Engine όταν τρέξει το παιχνίδι. Κάποιος θα αναρωτηθεί γιατί επιλέξαμε μια interpreted γλώσσα εδώ, κάτι που έρχεται σε αντίθεση με τα προηγούμενα. Η απάντηση είναι απλή. Η C++ χρησιμοποιείται για τον πυρήνα της μηχανής που χρειάζεται την ταχύτητα και η Lua για τα πιο “ελαφρά” κομμάτια που δεν θα επιφέρουν καθυστερήσεις και η σαφήνεια/αναγνωσιμότητα του κώδικα έχει προτεραιότητα.

1.7 Βιβλιοθήκες

Η μηχανή χρησιμοποιεί κάποιες βιβλιοθήκες για πετύχει τον στόχο της. Οι βιβλιοθήκες που συνδέονται απευθείας είναι:

- **Allegro 5:** Μια βιβλιοθήκη Multimedia SDK. Η μηχανή μπορεί να προγραμματιστεί με διάφορες άλλα αυτή είναι η προεπιλεγμένη. Περισσότερα παρακάτω.

- **CEGUI:** Η βιβλιοθήκη GUI που χρησιμοποιεί η μηχανή. Ανάμεσα στις διάφορες βιβλιοθήκες GUI, η CEGUI είναι πιο εύκολη όσο αφορά την προσάρτησή της σε Game Engines. Περισσότερα παρακάτω.
- **Libxml++:** Πρόκειται για το C++ Wrapper της libxml2. Είναι μια βιβλιοθήκη XML Parsing και Validation με XSD Schemas. Η Libxml2 είναι από τις λίγες βιβλιοθήκες που κάνουν την δουλεία τους γρήγορα και με λίγους πόρους.
- **Lua5.1:** Είναι ο στάνταρ Interpreter της Γλωσσάς Lua. Η βιβλιοθήκη του Interpreter μπορεί να προσαρτηθεί(Embed) σε προγράμματα C/C++ για να τρέξουν Lua Scripts.
- **Zlib:** Είναι μια βιβλιοθήκη συμπίεσης-αποσυμπίεσης δεδομένων με τον αλγόριθμο zlib που είναι μια παραδοχή του gzip. Με κλήσεις της βιβλιοθήκης αυτής, η μηχανή μετατρέπει τα συμπιεσμένα δεδομένα σε δεδομένα που μπορεί να αναπαράγει.
- **Tolua++:** Είναι μια βιβλιοθήκη που έχει βοηθητικές κλήσεις για μετατροπή αντικειμένων της C++ σε αντικείμενα της Lua και αντίστροφα.

Κάποιες άλλες βιβλιοθήκες που συνδέονται αυτομάτως από τις παραπάνω ή από την C++ όπως πχ:

- **stdc++:** Η στάνταρ βιβλιοθήκη της C++.
- **pthread:** Η βιβλιοθήκη των Posix Threads στα Linux. Η μηχανή χρησιμοποιεί αυτή την βιβλιοθήκη έμμεσα μέσω wrappers της stdc++.
- **Glib:** Βοηθητικές κλήσεις που χρησιμοποιούνται από την libxml++ κυρίως.
- **Glx:** Βιβλιοθήκη για τα γραφικά στην οθόνη.
- **X11:** Η βιβλιοθήκη του Xserver για την δημιουργία παραθύρων και λήψη συμβάντων από τον χρήστη.
- κτλ.

1.8 Μεταγλώττιση και Εγκατάσταση

Για να μπορέσετε να δοκιμάσετε και να χρησιμοποιήσετε την μηχανή, πρέπει να την χτίσετε (Build) και να την εγκαταστήσετε στο σύστημά σας. Πριν γίνει αυτό, πρέπει να έχετε στο σύστημά σας εγκατεστημένες τις βιβλιοθήκες που προαναφέραμε στο προηγούμενο κεφάλαιο (1.7) μαζί με τα headers τους (devel εκδόσεις). Πχ για διανομές βασισμένες σε Debian θα μπορούσατε να εκτελέσετε σε τερματικό τις παρακάτω εντολές:

Πίνακας 1.8.1: Πρότυπο εντολής εγκατάστασης σε Debian-Based διανομές

```
sudo apt-get install allegro5-dev allegro5-image-dev ...
```

Σε Debian διανομές η βιβλιοθήκη CEGUI δεν υπάρχει στην επαρκή έκδοση, για αυτό και θα πρέπει να την κατεβάσετε και να την εγκαταστήσετε μηχανικά. Ο τρόπος δεν μπορεί να περιγράφει εδώ γιατί ξεφεύγει από τους στόχους του εγγράφου.

Εφόσον όλες οι βιβλιοθήκες έχουν εγκατασταθεί και τα εργαλεία GNU/Autotools είναι διαθέσιμα, μπορείτε να κατεβάσετε τον πηγαίο κώδικα της μηχανής για να ξεκινήσει το χτίσιμο:

Πίνακας 1.8.2: Διεύθυνση του πηγαίου κώδικα της Μηχανή

```
http://sourceforge.net/projects/eftihia/
```

Αφού κατεβάσετε το συμπιεσμένο αρχείο και το αποσυμπιέσετε, με το τερματικό “πλοηγηθείτε στο φάκελο”. Ύστερα οι παρακάτω εντολές θα ξεκινήσουν το χτίσιμο:

Πίνακας 1.8.3: Εντολές τερματικού για χτίσιμο της Μηχανή

```
$ ./configure
$ make
```

Την εντολή “make” μπορείτε να την εκτελέσετε μαζί με το όρισμα “-jx” όπου “x” είναι ο αριθμός των παράλληλων εργασιών που να εκτελεστούν. Αυτό θα ανοίξει “x” διεργασίες που θα μεταγλωττίζουν, επιταχύνοντας την διαδικασία κατά “x” φορές. Συνήθως το “x” είναι ο αριθμός των πυρήνων του επεξεργαστή που χτίζεται η μηχανή.

Κατά την διάρκεια του χτισίματος το τερματικό θα εμφανίζει πάρα πολλά ακαταλαβίστικα σύμβολα. Αυτό σημαίνει ότι όλα πάνε καλά. Αν αυτό διακοπεί απότομα και στο τέλος έχει “Error” τότε αναφέρετε το σφάλμα στο site της μηχανής.

Όταν τελειώσει το χτίσιμο τότε μπορείτε να εγκαταστήσετε την μηχανή στο σύστημα:

Πίνακας 1.8.4: Εντολές τερματικού για χτίσιμο της Μηχανή

```
$ sudo make install
```

Μόλις τελειώσει και αυτό, όλα είναι έτοιμα. Για να δοκιμάσετε το Demo της μηχανής πηγαίνετε στον φάκελο Demo μέσα στον φάκελο που κάναμε αποσυμπίεση:

Πίνακας 1.8.5: Εντολές εκτέλεσης του Demo

```
$ cd ./Demo
$ ./DemoExample.sh
```

Αν όλα έχουν πάει καλά, τότε το Demo παιχνίδι της μηχανής θα ξεκινήσει. Λάβετε υπόψιν ότι το Demo δεν εγκαθίσταται στο σύστημα. Αν διαγράψετε τους φακέλους που χτίσατε την μηχανή θα το χάσετε και θα πρέπει να το ξαναχτίσετε μετά προκειμένου να το ξανά τρέξετε.

2 ΠΕΡΙΓΡΑΦΗ ΤΗΣ ENGINE

2.1 Ο πυρήνας

Ο πυρήνας της μηχανής είπαμε ότι έχει γραφτεί σε C++. Στην ουσία είναι ότι θα μεταγλωττιστεί σε δυαδική μορφή(Binary). Ο πυρήνας χωρίζεται σε 2 κομμάτια: το περιβάλλον C++ (C++ Environment) και το περιβάλλον Lua(Lua Environment). Το περιβάλλον C++ περιέχει όλες τις λειτουργίες που “αποκρύπτονται” από τον χρήστη όπως πχ: τις λειτουργίες εντοπισμού συγκρούσεων. Από αυτές τις πολύπλοκες λειτουργίες στην ουσία αποκρύπτεται ο τρόπος που γίνονται και στον χρήστη εμφανίζονται τα αποτελέσματα. Κάποιες από αυτές είναι ελέγξιμες από τον χρήστη. Πχ: ο χρήστης δίνει εντολή στον “παίχτη” να πάει σε ένα σημείο. Ο χρήστης νοιάζεται μόνο για το σημείο που θα πάει ο παίχτης παρά για το τι θα συμβεί μέσα στον πυρήνα για να κινηθεί ο χαρακτήρας στην οθόνη. Για αυτο χρήστης εργάζεται μόνο μέσα στο περιβάλλον Lua. Το περιβάλλον Lua είναι το περιβάλλον που εκτελούνται τα Lua Scripts. Είναι αυτόνομο αλλά η C++ μπορεί να έχει πρόσβαση σε αυτό. Σε αυτό το πεδίο ο χρήστης γράφει τα Scripts του όπως θέλει και το πεδίο C++ τα εκτελεί. Το πεδίο C++ “εξάγει” στο περιβάλλον Lua ένα API(Application programming interface) που ο χρήστης το χρησιμοποιεί για να δίνει εντολές στο περιβάλλον C++. Για παράδειγμα:

Πίνακας 2.1.1: Μέρος του αρχείου “ChickensOut.lua”

```
self.InteractListener = Zeta.LuaListener(function(User1,User2)
   if (User1:getName() == "Κότα") then
      local fear = require('Demo.Abilities.Effects.Fear')
      if (User1:canReceiveEffect(fear)) then
         User1:setTarget(User2)
         ...
         User1:addEffect(fear:getNewEffect(User1,1))
      end
   end
end ,Zeta.WorldEvent.Type.Interact)
```

Ο παραπάνω κώδικας έχει αρκετές κλήσεις που συνδέουν το περιβάλλον της Lua με το περιβάλλον της C++:

Πίνακας 2.1.2: Επεξηγήσεις κλήσεων του κώδικα του Πίνακα 2.1.1

`Zeta.LuaListener`	Το σύμβολο “LuaListener” είναι κλάση του Πυρήνα C++. Δημιουργείται ένα αντικείμενο LuaListener της C++, και αποθηκεύεται στο περιβάλλον της Lua

`User1,User2`	Αντικείμενα C++ Τύπου Zeta::Lifeform

`User1:getName()`
`User1:canReceiveEffect(fear)`
`User1:setTarget(User2)`
`User1:addEffect(...)`
	Κλήσεις μεθόδων της C++ της Κλάσης Zeta::Lifeform. Σε κάθε τέτοια κλήση, εκτελείται κώδικας C++ από πίσω και επιστρέφονται τα αποτελέσματα πίσω στην Lua.

`Zeta.WorldEvent.Type.Interact`	Enumeration(απαρίθμηση) της C++ που έχει μετατραπεί σε σύμβολο της Lua για περισσότερη σαφήνεια.

Ο παραπάνω πίνακας δίνει μερικούς από τους τρόπους επικοινωνίας C++/Lua. Δεν δίνονται περαιτέρω εξηγήσεις γιατί αυτό είναι κομμάτι του 2ου μέρους του εγγράφου.

Οι κυρίως τρόποι επικοινωνίας είναι:

- Κλήσεις συναρτήσεων C++
- Δημιουργία αντικειμένων C++ μέσα στο περιβάλλον Lua
- Σύνδεση διάφορων συμβάντων με κλήσεις κώδικα Lua

Οι συνδέσεις αυτές γίνονται με παραγωγή κώδικα C++ μέσω του τροποποιημένου εργαλείου “tolua++Z”. Το εργαλείο αυτό έχει τροποποιηθεί για να μπορεί να δέχεται κώδικα C++11 και να μπορεί να εξάγει κώδικα “doclua” που βοηθάει στην τεκμηρίωση του API της Μηχανής. Το εργαλείο αυτό είναι προς το παρών διαθέσιμο μόνο ως εκτελέσιμο μαζί με το Demo της μηχανής. Ο κώδικας θα δημοσιευτεί αργότερα όταν φτάσει σε Stable φάση.

Δεν είναι όλες οι κλήσεις που παρέχονται από την μηχανή ασφαλές για να χρησιμοποιούνται ασύστολα. Για παράδειγμα, όταν αποθηκεύεις έναν πόρο (Resource) μέσα στην Lua τότε ο πυρήνας χάνει την κυριότητα του πόρου και αφήνεται στον προγραμματιστή να ελέγξει το πότε πρέπει να καταστραφεί. Αυτό είναι μια λειτουργία που απαιτεί πιο προηγμένες γνώσεις και πρέπει να χρησιμοποιείται από άτομα που ξέρουν πολύ καλά τι κάνουν, αλλιώς θα έχουν προβλήματα με την μνήμη. Ευτυχώς, παρέχονται από την μηχανή πιο ασφαλές εναλλακτικές που θα πρέπει να είναι η πρώτη επιλογή και σε περίπτωση που δεν γίνεται αλλιώς να χρησιμοποιείται η μη ασφαλής επιλογή.

Για παράδειγμα, στο παραπάνω παράδειγμα του αρχείου “ChickensOut.lua”, πρέπει όταν καλεστεί αυτή η συνάρτηση να προστεθεί το “Effect.Fear” στον στόχο. Για να γίνει αυτό πρέπει να υπάρχει ένα EffectClass που δημιουργεί το Effect αυτό στον στόχο. Το EffectClass πρέπει να φορτωθεί από πριν αρχίσει το παιχνίδι γιατί αν φορτωθεί όταν χρειαστεί ίσως προκαλέσει “πάγωμα” (Hang) μέχρι να τελειώσει το

φόρτωσή του και το παιχνίδι να διαταχτεί ενοχλητικά. Για αυτό και όταν φορτωθεί το EffectClass του Effect.Fear δημιουργείται από την Lua και αποθηκεύεται σε καθολική μεταβλητή της Lua για να είναι πάντα διαθέσιμο. Δηλαδή ο πόρος παύει να είναι στην κατοχή του πυρήνα. Για να διαγραφεί τώρα ο πόρος πρέπει να το κάνει ο χρήστης όποτε νομίζει ότι πρέπει. Αυτό φαίνεται απλό άλλα στην πραγματικότητα δεν είναι καθόλου για έναν προγραμματιστή που δεν έχει ξανά ασχοληθεί με ρητή συλλογή απορριμμάτων (Explicit Garbage Collection). Η μηχανή παρέχει έναν μηχανισμό για την αποφυγή αυτού άλλα δεν βοηθάει πάντα. Σε μελλοντικές ενημερώσεις της μηχανής αυτά τα προβλήματα θα εξαλειφθούν.

Ο πυρήνας δίνει πολύ έμφαση στην σταθερότητά του. Η προσπάθεια που έγινε ήταν να τρέξει ο πυρήνας ακόμα και υπό αντίξοες συνθήκες. Τι εννοείται με τον όρο “αντίξοες συνθήκες”; Όταν στον πυρήνα εισάγουμε μη έγκυρα δεδομένα, υπό κανονικές συνθήκες θα τερμάτιζε απρόσμενα επειδή πήγε να επεξεργαστεί κάτι που δεν του είχε νόημα. Για παράδειγμα, αν του δώσουμε να φορτώσει έναν χάρτη και η πρώτη εγγραφή του XML αρχείου δεν είναι <map> (δεν είναι έγκυρο (Valid) για την μηχανή), ο πυρήνας θα πάει να το ψάξει και θα βρει στο τέλος NULL. Δουλεύοντας πάνω σε null Pointer στην C++, σηκώνει (Raises) σήμα Segmentation Fault που τερματίζει το πρόγραμμα.

Όμως αυτό δεν συμβαίνει. Το τι συμβαίνει θα δοθεί παραθέτοντας περιγραφικά τα βήματα φόρτωσης ενός χάρτη.

1. Δίνεται η εντολή στο WorldManager για να αλλάξει ο χάρτης σε “./Maps/Village1.tmx”
2. Ο WorldManager ζητάει από τον MapLoader να φορτώσει τον χάρτη.
3. Ο MapLoader φορτώνει τον χάρτη. Όσο φορτώνει ελέγχει:
   a) Αν το αρχείο υπάρχει
   b) Αν είναι έγκυρο
   c) Αν τα αρχεία που συνδέει ο χάρτης υπάρχουν και είναι έγκυρα.
4. Ο χάρτης ζητάει από τον ResourcesContext τους πόρους που χρειάζεται (πχ Images). Αν κάτι δεν υπάρχει ή είναι άκυρο, τότε ο ResourcesContext δίνει ένα έγκυρο και χρησιμοποιήσιμο NullReference.
5. Αν κάποιες από τις προϋποθέσεις του βήματος 3 αποτύχουν, τότε ο χάρτης δίνει σήμα σφάλματος στον MapLoader και αυτός τον εκμηδενίζει και αναφέρει το σφάλμα στο αρχείο καταγραφής.

6. Ο εκμηδενισμένος χάρτης είναι απόλυτα χρησιμοποιήσιμος και δεν φέρει προβλήματα. Παρόλο αυτά για τον παίχτη δεν είναι αυτό που περίμενε.

Στην παραπάνω περίπτωση αν υπάρξει σφάλμα κατά την φόρτωση τότε η μηχανή θα συνεχίσει αλλά ο χάρτης που θα παίξει ο χρήστης θα είναι ένα μια εικόνα από σκουπίδια λόγω ότι ενέργησαν τα NullReferences. Περισσότερα στο 3ο μέρος.

Αυτή η ιδιότητα του πυρήνα τον αποτρέπει από το να καταρρέει σε σφάλματα, να συνεχίζει την εκτέλεση χωρίς πρόβλημα αν το σφάλμα δεν φαίνεται άμεσα αλλά και να εντοπίζονται κατευθείαν τα σφάλματα λόγο αρχείων.

Πέρα από τον μηχανισμό σφαλμάτων, ο πυρήνας παρέχει ένα μηχανισμό αυτοσχέδιου συλλογής απορριμμάτων(Garbage Collection). Με λίγα λόγια, αυτός ο μηχανισμός δίνει τους πόρους όπου ζητηθούν, εγγυάται την μοναδικότητα των πόρων, και τους καταστρέφει όταν δεν χρειάζονται πια. Περισσότερα στο 3ο μέρος.

Πέρα από αυτά ο πυρήνας ασχολείται και με τα παρακάτω:

- Συγκρούσεις αντικειμένων
- Διαχείριση των Animations
- Βασικές λειτουργίες RPG
- Σύνδεση με τα Game SDK

2.2 Standard Zeta Lua Library

Η Standard Zeta Lua Library ή SZLL είναι μια συλλογή από σενάρια Lua(Lua Scripts) που ο σκοπός τους είναι να κάνουν την ανάπτυξη των παιχνιδιών πιο εύκολη, κλείνοντας μέσα τους ακόμα περισσότερες λειτουργίες. Είναι έτσι φτιαγμένα σαν κλάσεις-αντικείμενα για μεγαλύτερη ευκολία. Επίσης παρέχει τεκμηρίωση του API του πυρήνα μέσω DocLua. Κάποιες από τις λειτουργίες που παρέχει η SZLL είναι:

- Δημιουργία προκαθορισμένων δομών RPG (πχ ActiveAbility, OffAnimation)
- Δομές δεδομένων τύπου containers (πχ Double Ended Queue)
- Δομή για δημιουργία κλάσεων Lua
- Αποτύπωση πινάκων Lua σε αρχεία και διάβασμα από αυτά

Η χρήση της SZLL δεν είναι απαραίτητη, όμως για μικρά παιχνίδια βοηθάει πολύ. Για μεγαλύτερα παιχνίδια συνήθως πρέπει να φτιάχνεται από την αρχή μια εξειδικευμένη βιβλιοθήκη.

2.3 XML Schemas

Τα XML Schemas (XSD) είναι αρχεία παρόμοια με τα XML που καθορίζουν το πώς θα πρέπει να είναι κάποια XML αρχεία. Αν ένα αρχείο XML περάσει τον έλεγχο εγκυρότητας (Validation) που του γίνεται πάνω σε ένα XML Schema, τότε αυτό το XML είναι εγγυημένο ότι θα έχει τα δεδομένα και εγγραφές που περιγράφει το Schema. Πχ ένα XML Schema καθορίζει ότι ένα XML αρχείο πρέπει να έχει στην αρχή ένα και μόνο ένα στοιχείο <Lifeform> το οποίο πρέπει να έχει ένα Attribute “Name” που είναι τύπου String.

Βάσει αυτών των κανόνων η μηχανή ελέγχει για την εγκυρότητα των XML αρχείων που της φορτώνονται και έτσι αποτρέπει τα σφάλματα.

Τα XML Schemas της μηχανής είναι εξωτερικά αρχεία. Αυτό γίνεται για να μην μεγαλώνει η βιβλιοθήκη σε μέγεθος και να μπορεί εύκολα να διορθώνεται σε περίπτωση παράλειψης ή προσθήκης νέου στοιχείου. Αυτό σημαίνει ότι δεν πρέπει να πειράζονται από κανένα που δεν ξέρει τι κάνει αλλιώς θα έχει άσχημα αποτελέσματα.

3 Η ΒΙΒΛΙΟΘΗΚΗ ALLEGRO5

Η μηχανή κανονικά είναι ευέλικτη ως το “ποιος” θα κάνει τις δουλειές χαμηλού επιπέδου. Και εδώ μιλάμε για τις βιβλιοθήκες πολυμέσων (Multimedia Libraries). Αυτές οι βιβλιοθήκες παρέχουν βασικές λειτουργίες για την δημιουργία διαδραστικών εφαρμογών (πχ σε εφαρμογές με GUI από πίσω κρύβεται κάποια Multimedia Library). Ο προγραμματιστής δίνει κάποια εντολή σε αυτές και αυτές με την σειρά τους εκτελούν όλες τις πολύπλοκες λειτουργίες από πίσω που χρειάζονται για παραχθεί το αποτέλεσμα που χρειάζεται. Πχ ο προγραμματιστής δίνει την εντολή να εμφανιστεί μια εικόνα στο σημείο 232,536 με περιστροφή κατά 38 μοίρες. Η βιβλιοθήκη από πίσω θα κάνει όλους τους απαραίτητους μετασχηματισμούς στην εικόνα και ύστερα θα την εμφανίσει. Αυτή η δουλειά δεν είναι απλή για αυτό και την κάνει η Multimedia Library. Οι βιβλιοθήκες αυτές συνήθως ασχολούνται με:

- Εμφάνιση εικόνων στην οθόνη
- Αναπαραγωγή ήχων
- Φόρτωση πόρων
- Διαχείριση χρονικών μετρητών
- Σύνδεση με εξωτερικά συμβάντα
- Ενσωμάτωση εικονικού συστήματος αρχείων
- Δικτύωση
- κ.α.

Όλα αυτά τα χρησιμοποιεί ο προγραμματιστής για να φτιάξει την Game Engine.

Όπως προαναφέραμε, η εν λόγω μηχανή είναι ευέλικτη σε αυτό το θέμα. Παρέχει Wrappers που κλείνουν εσωτερικά άλλες κλήσεις. Αυτό της επιτρέπει να ενσωματώνει διαφορετικές Multimedia Libraries παράγοντας πάντα τα ίδια αποτελέσματα.

Η μηχανή αναπτύχθηκε με ενσωμάτωση της βιβλιοθήκης Allegro5 όπου είναι και η προκαθορισμένη για το χτίσιμο της. Η Allegro5 είναι γραμμένη σε C και παρέχει ότι χρειάζεται η μηχανή και με το παραπάνω. Είναι φτιαγμένη σε Modules που σημαίνει ότι συνδέεις στο πρόγραμμα σου μόνο τις λειτουργίες που χρειάζεσαι. Είναι cross-platform και οι λειτουργίες της για εμφάνιση εικόνων στην οθόνη είναι επιταχυνόμενες από το υλικό μέσω OpenGL ή Direct-X για Linux και Windows αντίστοιχα. Από τα modules της Allegro5 η μηχανή χρησιμοποιεί:

- Το κύριο (allegro5), για μετρητές, σύνδεση με συμβάντα, και εμφάνιση εικόνων

- Το allegro_image για φόρτωση εικόνων διαφόρων μορφών
- Το allegro_audio και allegro_acodec για την αναπαραγωγή ήχων
- Το allegro_primitives για την εμφάνιση διαφόρων σχημάτων

Άλλες παρόμοιες βιβλιοθήκες με την Allegro5 είναι:

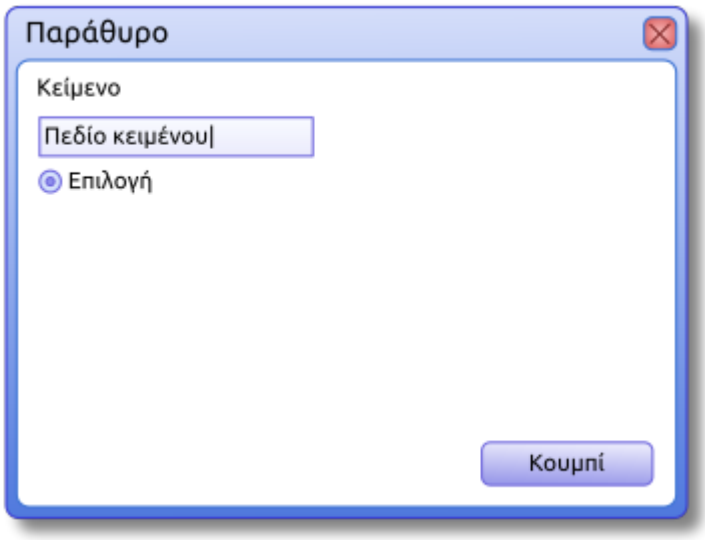
- SDL
- SFML

Όλες παράγουν τα ίδια αποτελέσματα με την σωστή χρήση.

Ο λόγος που επιλέχτηκε η Allegro5 για το προδιαγεγραμμένο SDK είναι θέμα κυρίως εξοικείωσης που υπάρχει με αυτή. Πέρα από αυτό, ήταν η μόνη που παρείχε Hardware Acceleration για τα γραφικά μέσω OpenGL την εποχή που είχε αρχίσει η συγγραφή της μηχανής. Αυτό τότε θεωρήθηκε καταλυτικός παράγοντας στην επιλογή.

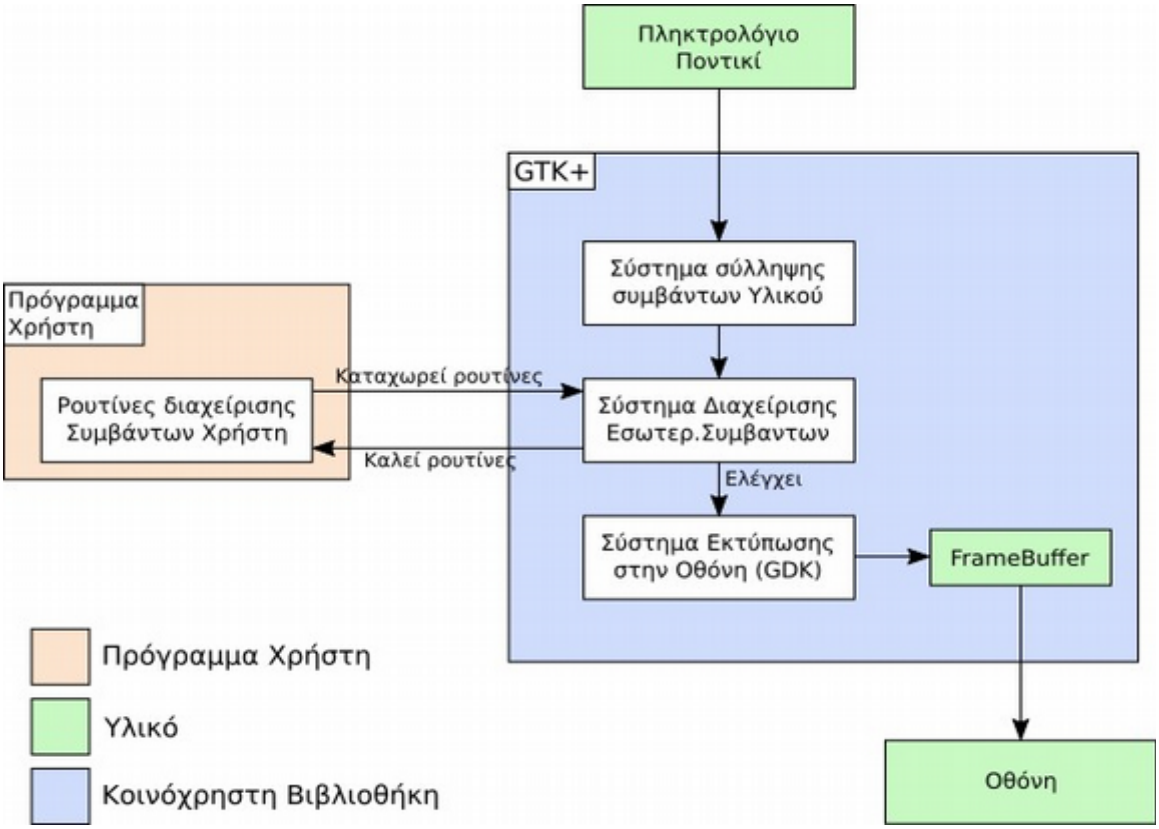
4 Η ΒΙΒΛΙΟΘΗΚΗ CEGUI

Τα παιχνίδια RPG βασίζονται πάρα πολύ στην διεπαφή παιχνιδιού-χρήστη. Για αυτό και είναι απαραίτητο το παιχνίδι να ανταλλάσσει πληροφορίες με τον χρήστη ανά πάσα στιγμή. Σε αυτό βοηθάει η γραφική διεπαφή χρήστη (Graphical User Interface – GUI). GUI έχουμε χρησιμοποιήσει όλοι. Πρόκειται για τα παράθυρα, τα κουμπιά, τα κείμενα, οι επιλογές που μας εμφανίζονται γραφικά και μας παρέχουν εύκολη διεπαφή με το πρόγραμμα



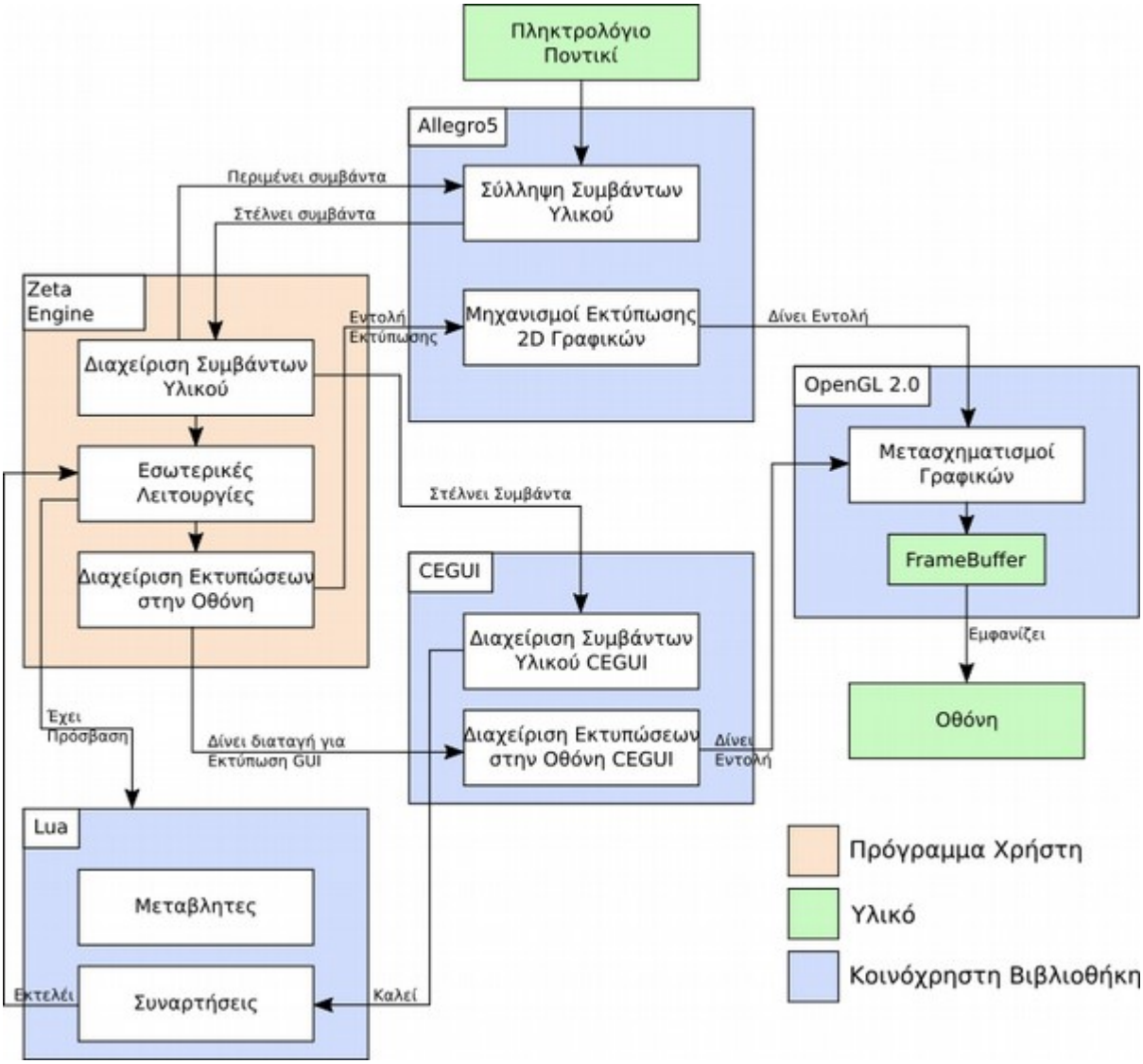
Εικόνα 4.1: Παράδειγμα Γραφικής Διεπαφής Χρήστη

Στο παραπάνω σχήμα φαίνεται ένα παράθυρο που όλοι έχουμε συναντήσει. Το παράθυρο και όλα τα στοιχεία που έχει μέσα είναι προϊόν μιας βιβλιοθήκης GUI.Ο αρχικός στόχος για GUI στην μηχανή ήταν να έχει το δικό του. Όμως αυτό θα μπορούσε να ήταν ένα άλλο μεγάλο Project που θα έπαιρνε πολύ χρόνο. Για αυτό και προς το παρών η μηχανή χρησιμοποιεί την βιβλιοθήκη GUI CEGUI. Αυτή η βιβλιοθήκη σε αντίθεση με τις άλλες (πχ GTK+, wxWidgets) αφήνει στον προγραμματιστή να της δίνονται τα συμβάντα, να εμφανίζεται όποτε θέλει αυτός, και να έχει τον πλήρη έλεγχο γύρω από αυτά. Για παράδειγμα, στη βιβλιοθήκη GTK+ ο προγραμματιστής χρειάζεται να ρυθμίσει μόνο πως θα φαίνεται το GUI και τι συμπεριφορές θα έχει κάθε στοιχείο. Τα συμβάντα του υλικού και το πότε θα εμφανίσει στην οθόνη αφήνεται αποκλειστικά στην βιβλιοθήκη.



Εικόνα 4.2: Παράδειγμα υλοποίησης προγράμματος σε GTK+

Στην CEGUI πρέπει ο προγραμματιστής να έχει φτιάξει από πριν την αρχικοποίηση ένα OpenGL Context, που πάνω σε αυτό η CEGUI θα εκτυπώνει τα παράθυρα της όταν της δοθεί η εντολή. Το OpenGL Context το φτιάχνει για εμάς η Allegro5 και το μοιράζεται με την CEGUI μέσω της μηχανής. Η CEGUI είναι η μόνη που αφήνει σε εμάς να φτιάξουμε το Context για αυτό και επιλέχτηκε ανάμεσα στις άλλες καθώς θέλουμε να σχεδιάζονται τα πάντα σε έναν FrameBuffer και όχι σε ξεχωριστούς.



Εικόνα 4.3: Σχέσεις μεταξύ βιβλιοθηκών στην Game Engine

Στο παραπάνω διάγραμμα φαίνεται ότι η μηχανή είναι μεσολαβητής μεταξύ της Allegro5 και της CEGUI. Η μηχανή είναι αυτή που αποφασίζει τι η CEGUI θα λάβει και πότε θα τα λάβει. Αντίστοιχα η CEGUI με την σειρά της και όπως έχει ρυθμιστεί μέσω της Lua, μπορεί να καλέσει κλίσεις της μηχανής που θα αλλάξουν κάτι στο παιχνίδι.

Η CEGUI χρησιμοποιεί XML αρχεία για αναπαράσταση των παραθύρων και των πεδίων. Τα στυλ είναι και αυτά ρυθμιζόμενα από XML αρχεία και εικόνες.

Κάθε στοιχείο στην CEGUI θεωρείται παράθυρο. Ο τύπος του ορίζεται από ένα String που συνδέεται με δομές XML που του προσδιορίζουν την εμφάνιση και τα περαιτέρω στοιχεία που μπορεί να έχει μέσα. Οι συμπεριφορές όπως προαναφέραμε

καθορίζονται από κλήσεις C ή Lua και Python μέσω των Wrappers που παρέχει. Για περισσότερα με το CEGUI πηγαίνετε στο Online Documentation.

5 Η ΓΛΩΣΣΑ LUA

Η μηχανή πέρα από την περιγραφή δεδομένων, χρειάζεται και λειτουργικά δεδομένα που μπορούν εύκολα να αλλάξουν χωρίς να χρειάζεται να μεταγλωττίζεις συνέχεια το πρόγραμμα. Για αυτό χρειάστηκε να μπει μια δεύτερη γλώσσα στο προσκήνιο μια γλώσσα σεναρίων (Scripting Language).

Η γλώσσα σεναρίων συνήθως είναι interpreted, που σημαίνει ότι ο κώδικας μεταγλωττίζεται και εκτελείται κατά την φόρτωση του αρχείου που τον περιέχει. Αυτό σημαίνει ότι ο κώδικας γράφεται σε αρχεία κειμένου και μπορείς να τον αλλάξεις μετά από κάθε εκτέλεση χωρίς να πειράξεις το εκτελέσιμο αρχείο σου.

Με την προϋπόθεση ότι η γλώσσα θα πρέπει να είναι απλή, εύκολη, γρήγορη, και να έχει καλή συνεργασία με την γλώσσα-οικοδεσπότη (Host Language) επιλέχτηκε η Lua. Τα δεδομένα της είναι εύκολα προσβάσιμα από την C++ μέσω του VM της. Πέρα από αυτό, στην επιλογή της συνέβαλε ότι είναι και πολύ δημοφιλής στις Game Engine. Ένα παράδειγμα είναι το παιχνίδι World Of Warcraft που χρησιμοποιεί την Lua για την αποθήκευση δεδομένων και ορισμού συμπεριφοράς των Addon. Επειδή το παιχνίδι δέχεται Lua Scripts, ένας παίχτης μπορεί να γράψει ένα Addon και να το χρησιμοποιήσει κατευθείαν. Εδώ στην μηχανή γίνεται κάτι αντίστοιχο αλλά σε πιο προηγμένο επίπεδο.

Η Lua είναι μια γλώσσα απλή αλλά με προοπτικές για πιο προηγμένα πράγματα. Δεν έχει κλάσεις για την παραγωγή αντικειμενοστραφούς κώδικα αλλά έχει απλούστερες δομές δεδομένων που μπορεί να εξομοιώσει τον αντικειμενοστραφή προγραμματισμό σε πολύ ικανοποιητικό επίπεδο.

Η Lua έχει που παρακάτω τύπους δεδομένων:

- Συμβολοσειρές (Strings)
- Αριθμοί (Numbers)
- Συναρτήσεις (Functions)
- Πίνακες (Tables)
- Τίποτα (nil)

Οι τύποι δεδομένων δεν καθορίζονται ρητά αλλά ανάλογα τι είδους δεδομένα που εισάγονται σε κάθε μεταβλητή αλλάζει και ο τύπος της.

Κάθε συνάρτηση της Lua μπορεί να επιστρέψει κάτι, ή ακόμα και πολλά πράγματα μαζί. Τα αρχεία της Lua είναι και αυτά σαν συναρτήσεις και μπορούν να επιστρέψουν πράγματα σε αυτό που τα κάλεσε.

Παρακάτω παρατίθεται βασικός κώδικας σε Lua με επεξηγήσεις.

Πίνακας 5.1: Παράδειγμα Lua με συναρτήσεις

```
function foo(n)     -- Καθορίζεται μια συνάρτηση με μια παράμετρο n
   local x = 5      -- Καθορίζεται μια τοπική μεταβλητη x και πέρνει τιμή 5
   print(x * n)     -- Εκτυπώνεται το αποτέλεσμα του x * n
end                 -- Τέλος συνάρτησης. Το x χάνεται εδώ γιατί είναι local

local function bar(x,y) --Τοπική συνάρτηση.Όταν τελειώσει το αρχείο χάνετε
  return x,y            -- Επιστρέφει στον καλών, το x και y
end

-- Εξωτερικός Κώδικας: Εκτελείτε όταν εκτελεστεί το αρχείο
print("Γειά!")
foo(6)          -- Καλείτε η foo που καθορίσαμε πριν
local a,b = bar(3,6) -- Καλείτε η bar και τα αποτελέσματα μπαίνουν στα a,b
```

Πίνακας 5.2: Παράδειγμά Lua με πίνακες

```
local table = {           -- Καθορίζουμε τοπικό πίνακα
   field1 = "Γειά!!",     -- Του ορίζουμε το πεδίο με όνομα "field1"
   numField = 6,          -- Του ορίζουμε το πεδίο με όνομα "numField"
   funcField = function(x)  -- Του ορίζουμε το πεδίο με όνομα "funcField"
      print(x)              -- Κώδικας του πεδίου "funcField"
   end,
   tablefield = {      -- το Πεδίο "tablefield" είναι και αυτός πινακας
      g = "Haha!"
   }
}

table.field1 = "Το άλλαξα!"   -- Αλλαγή του πεδίου "field1"
table["field1"] = "Το άλλαξα πάλι!"   -- Αλλαγή του πεδίου "field1"
table.funcField("Μυνημα")  -- Κλήση του πεδίου "funcField"
table.funcField = 4        -- Το Πεδίο "funcField" είναι πλεόν ο αριθμός 4
table.tableField.g = "3G"  -- Αλλαγή πεδίου εμφωλευμένου πίνακα
table.newField = "New!!"   -- Δλημιουργία νέου πεδίου και απόδωση τιμής
table.numField = nil       -- Διαγαφή του πεδίου "numField"
```

Αξίζει να σημειωθεί ότι στην Lua υπάρχει Garbage Collector. Εάν κάτι του αποδοθεί η τιμή `nil` και τα δεδομένα που είχε μέσα δεν έχουν αναφορές σε άλλες μεταβλητές, τότε αυτά τα δεδομένα θα καταστραφούν στον επόμενο κύκλο του Garbage Collector.

Η Lua είναι γραμμένη σε C. Έτσι η σύνδεση με την C/C++ είναι εύκολη. Για την ακρίβεια, μέσω του προγράμματος C ο προγραμματιστής φτιάχνει ένα περιβάλλον Lua και με διάφορες συναρτήσεις μπορεί να ελεγχθεί ανά πάσα στιγμή.

Μέσω της C ο προγραμματιστής μπορεί να έχει πρόσβαση σε όλες τις μεταβλητές και συναρτήσεις της Lua, να καλέσει οποιαδήποτε από αυτές και να λάβει τα αποτελέσματά τους. Επίσης μπορεί να αντιστοιχίσει κάποια συνάρτηση της C με ένα σύμβολο της Lua και μέσω αυτού του συμβόλου να καλείτε μέσω της Lua. Σε συνδυασμό με το εργαλείο “tolua++” μπορούν να αντιστοιχιστούν και κλάσεις της C++. Για περισσότερα πάνω στην Lua ανατρέξτε το εγχειρίδιο χρήστη της Lua στο διαδίκτυο.

6 ΠΡΟΑΠΑΙΤΟΥΜΕΝΕΣ ΓΝΩΣΕΙΣ

6.1 Patterns που χρησιμοποιούνται

Στην μηχανή χρησιμοποιούνται διάφορα αντικειμενοστραφή πρότυπα προγραμματισμού (OOP Patterns). Τα πρότυπα αυτά είναι τεχνικές προγραμματισμού που βοηθάνε στην καλύτερη λειτουργικότητα και ποιότητα του προγράμματος. Μερικά από αυτά που χρησιμοποιούνται πρέπει να εξηγηθούν για να μπορεί να γίνει πιο κατανοητή η λειτουργία της μηχανής αργότερα.

1. **Singleton**

Κάποια αντικείμενα πρέπει να δημιουργούνται μόνο μια φορά καθ' όλη την εκτέλεση του προγράμματος. Πέρα από αυτό, το μοναδικό αυτό αντικείμενο πρέπει να είναι προσβάσιμο καθολικά. Αυτό το εγγυάται το πρότυπο Singleton. Αυτό επιτυγχάνεται με την αφαίρεση των Constructors μιας κλάσης και δημιουργία μιας καθολικής συνάρτησης που θα επιστρέφει πάντα το ίδιο αντικείμενο.

2. **Object Pool**

Κατά την διάρκεια του παιχνιδιού χρειάζεται να δημιουργούνται και να καταστρέφονται αντικείμενα. Μερικά από αυτά τα αντικείμενα εάν είναι μεγάλα τότε ίσως υπάρξουν καθυστερήσεις κατά την δημιουργία και καταστροφή. Αυτό είναι απαράδεκτο για τον χρήστη. Και εδώ έρχεται να λύσει το πρόβλημα η τεχνική Object Pooling. Σε αυτή την τεχνική υπάρχει μια ουρά από προ-κατασκευασμένα αντικείμενα απολύτως ρυθμιζόμενα για κάθε ανάγκη. Όταν κάτι χρειαστεί να δημιουργήσει ένα νέο αντικείμενο τότε δεν το δημιουργεί άλλα το ζητάει από την δεξαμενή αντικειμένων. Το αντικείμενο ρυθμίζεται σαν να είχε κατασκευαστεί. Όταν το αντικείμενο δεν χρειάζεται πια δεν καταστρέφεται άλλα ξαναγυρίζει πίσω στην δεξαμενή για μελλοντική χρήση. Στην δεξαμενή φροντίζεται να υπάρχει αρκετό απόθεμα από αντικείμενα σε περιπτώσεις που ζητηθούν πάρα πολλά μαζί. Σε πιο προηγμένες υλοποιήσεις, εάν το απόθεμα εξαντληθεί τότε δημιουργείται ένα νέο αντικείμενο το οποίο θα αποδοθεί στην αίτηση και θα μεγαλώσει το απόθεμα κατά ένα. Η υλοποίηση αυτής της τεχνικής εξαρτάται από την εκάστοτε χρήση της.

3. **Builder**

Στην μηχανή υπάρχει η ανάγκη για να υπάρξουν γενικές κλάσεις που θα παράξουν αντικείμενα με τα ίδια χαρακτηριστικά όποτε αυτό ζητηθεί. Ανάλογα με το τι δεδο-

μένα εισόδου θα δοθούν στην γενική κλάση, η ίδια κλάση μπορεί να παράξει διαφορετικές οικογένειες δεδομένων. Για παράδειγμα,

Πίνακας 6.1.1: Βοηθητικός Κώδικας Lua για το παράδειγμα Builder

```
Ability1 = {
   ...
   Passive = false,
   ...
}
```

Εδώ εάν ο παραπάνω πίνακας εισαχθεί στην γενική κλάση “AbilityClass”, τότε όταν της ζητηθεί να δημιουργήσει ένα “Ability” αυτή θα κατασκευάσει ένα “ActiveAbility” επειδή η τιμή του πεδίου “Passive” είναι “false”. Αν ήταν “true” τότε θα έφτιαχνε ένα “PassiveAbility”. Με την αλλαγή ενός πεδίου στον πίνακα εισαγωγής αλλάζει όλη η λειτουργία της γενικής κλάσης. Περισσότερα γύρω από αυτή την λειτουργία στο 3ο μέρος.

4. **NullObject**

Όπως είχαμε προαναφέρει, για να εγγυηθεί η λειτουργία της μηχανής σε περίπτωση που κάποιος πόρος δεν είναι διαθέσιμος, θα πρέπει να παρέχουμε κάτι για να αντικαταστήσει τον ελλιπές πόρο ώστε να μην υπάρχουν καθυστερήσεις και καταρρεύσεις του προγράμματος κατά την εκτέλεση. Αυτήν την δουλειά την κάνει το NullObject. Πρόκειται για ένα αντικείμενο καθολικά προσβάσιμο που έχει την πλήρη λειτουργία ενός τύπου πόρου που προσδιορίζεται και δίνεται σε μια λειτουργία όταν ο πόρος που ζητήθηκε δεν είναι διαθέσιμος για οποιοδήποτε λόγο.

6.2 Βασικές γνώσεις Game Development

α) Γενικά και ο Κύριος Βρόγχος

Σε ένα πρόγραμμα οι εντολές εκτελούνται η μία μετά την άλλη και όταν φτάσει στην τελευταία εντολή το πρόγραμμα τερματίζει. Συνήθως η πορεία και ποιες εντολές θα εκτελεστούν καθορίζονται με τα δεδομένα εισόδου του προγράμματος. Κατά την διάρκεια και το τέλος του προγράμματος παράγονται τα αποτελέσματα εξόδου. Το πρόγραμμα έχει πεπερασμένο χρόνο εκτέλεσης που για ίδια δεδομένα θα κάνει περίπου ίδιο χρόνο και πορεία.

Σε εφαρμογές με GUI το πρόγραμμα είναι σε μια μόνιμη αναμονή για εξωτερικά συμβάντα χρήστη (πχ πάτημα ενός κουμπιού) και ανάλογα το συμβάν εκτελούνται κάποιες ενέργειες.

Στα παιχνίδια η κατάσταση είναι παρόμοια με τις εφαρμογές GUI αλλά το πρόγραμμα δεν θα περιμένει για πάντα κάποιο συμβάν χρήστη για να κάνει κάτι. Είναι ρυθμισμένο να κάνει κάποιες ενέργειες ανά τακτά χρονικά διαστήματα είτε ο χρήστης δώσει κάποια εντολή είτε όχι. Αυτές οι ενέργειες κάθε φορά που εκτελούνται αποτελούν ένα πλαίσιο ή frame του παιχνιδιού. Στα frame εκτελείται η γενική λογική του παιχνιδιού. Πχ κινήσεις, συγκρούσεις, τεχνητή νοημοσύνη κτλ. Το τι θα γίνει στο frame, εξαρτάται από το παιχνίδι αποκλειστικά και από τι συμβάντα χρήστη έγιναν πριν ξεκινήσει το frame. Για παράδειγμα, μπορεί ο χρήστης να κάνει κλικ σε ένα σημείο για πάει ο χαρακτήρας. Ο χαρακτήρας όμως θα αρχίσει να κινείται μόλις ξεκινήσει το επόμενο frame διότι εκεί υπάρχουν οι λειτουργίες που τον κάνουν να κινείται.

Συνήθως αφού έχουν ολοκληρωθεί όλες οι ενέργειες ενός frame, ακολουθεί η “Σχεδίαση” του παιχνιδιού. Εκεί το παιχνίδι αναλαμβάνει να εμφανίσει ότι είναι να εμφανιστεί στην οθόνη. Αφού ολοκληρωθεί και αυτό τότε το πρόγραμμα είναι έτοιμο να δεχθεί άλλο συμβάν ή να κάνει το επόμενο frame.

Τα παιχνίδια είναι ένα σύνολο από μεταβλητές και αντικείμενα που μεταβάλλονται κατά την διάρκεια της εκτέλεσης μέσω των συμβάντων χρήστη ή των frames. Κάθε φορά που έρχεται η σειρά να σχεδιαστεί το παιχνίδι στην οθόνη, τα πάντα “παγώνουν” και εμφανίζεται η κατάσταση των μεταβλητών και των αντικειμένων εκείνη την δεδομένη στιγμή.

Όλα αυτά γίνονται στην πράξη μέσω του κύριου βρόγχου (Main Loop) του παιχνιδιού. Αυτός ο βρόγχος είναι στην ουσία μια ατέρμονος βρόγχος. Η εκτέλεση βγαίνει από εκεί μόνο όταν το πρόγραμμα πρέπει να τερματίσει. Αυτό συνήθως ελέγχεται μέσω μιας μεταβλητής Boolean. Πχ:

Πίνακας 6.2.1: Παράδειγμά ελεγχόμενης ατέρμονης βρόχου

```
bool running = true;
while (running) {
   ...
   if (...) {
      running = false;
   }
   ...
}
```

Οι αρμοδιότητες του κύριου βρόγχου είναι συνήθως να ελέγχει για συμβάντα υλικού, να διανέμει αυτά τα συμβάντα σε ότι αντικείμενα τα χρειάζονται, να εκτελούν τα frames και να σχεδιάζουν στην οθόνη. Όλα αυτά γίνονται με προτεραιότητα. Μεγαλύτερη προτεραιότητα έχουν να διανεμηθούν τα συμβάντα υλικού. Αυτό γιατί κάθε φορά που ο χρήστης δίνει μια εντολή θέλει να έχει άμεση ανταπόκριση με το παιχνίδι και το frame να μην εκτελεστεί με παλαιά δεδομένα (πχ παλιά θέση του ποντικιού). Επόμενο στην σειρά έχει το frame. Τελευταίο είναι η σχεδίαση του παιχνιδιού. Αυτό έρχεται πάντα μετά το frame γιατί δεν θα είχε νόημα να εμφανίσουμε στην οθόνη κάτι που δεν έχει επεξεργαστεί ακόμα, με άλλα λόγια κάτι που έχει ακόμα τις παλιές τιμές.

Η υλοποίηση του κύριου βρόγχου μέσα στην μηχανή εξαρτάται από την εκάστοτε βιβλιοθήκη SDK που χρησιμοποιείται. Για εκπαιδευτικούς λόγους θα παραθέσουμε την θεωρητική υλοποίηση της με την Allegro5.

Με την Allegro5, υπάρχει μια ουρά συμβάντων που έχει ένα δικό της νήμα που προσθέτει συμβάντα όποτε αυτά φτάσουν. Το κύριο νήμα όπου βρίσκεται ο κύριος βρόγχος, ελέγχει σε κάθε κύκλο αν υπάρχουν συμβάντα στην ουρά. Αν υπάρχουν, τότε το βγάζει από την ουρά και το επεξεργάζεται ανάλογα. Για να παραχθούν frames, πρέπει να οριστεί κάθε πότε θα γίνεται ένα frame. Αυτό γίνεται μέσω ενός χρονομέτρου (Timer) που ρυθμίζεται να στέλνει ένα συμβάν χρονομέτρου στην ουρά συμβάντων. Έτσι ο κύριος βρόγχος θα ξέρει ότι όταν του έρθει συμβάν χρονομέτρου, θα πρέπει να κάνει ένα frame.

Ο ρυθμός που έρχονται τα συμβάντα για να εκτελεστούν frames ίσως διαφέρει από τον ρυθμό που θα εκτελεστούν στην πραγματικότητα. Αυτό γίνεται γιατί οι λειτουργίες του frame μπορεί να είναι πολλές και επίπονες για τον επεξεργαστή και να μην προλάβει να τελειώσει το frame πριν έρθει η εντολή για το επόμενο. Αυτό σπανίζει γιατί πλέον οι επεξεργαστές είναι πολύ ταχύτατοι και δεν έχουν τέτοιο πρόβλημα. Το πιο συχνό πρόβλημα εμφανίζεται λόγω ότι το frame συνοδεύεται από την σχεδίαση στην οθόνη. Η σχεδίαση είναι βαριά δουλειά και δεν την κάνει ο επεξεργαστής αλλά αφήνεται στην κάρτα γραφικών (GPU). Καθώς τα γραφικά γίνονται όλο και πιο ρεαλιστικά, ο φόρτος της GPU γίνεται πιο πολύς και φτάνει σε σημείο να παίρνει πολύ ώρα μέχρι να εκτελέσει όλες τις εντολές σχεδίασης. Ο ρυθμός που τα frames εκτελούνται και σχεδιάζονται στην οθόνη μετριέται σε Frames/sec ή frames per Second ή FPS. Ο ρυθμός που τα συμβάντα για frame στέλνονται μπορεί να είναι κλειδωμένος σε έναν σταθερό ρυθμό πχ (60 FPS) ή να είναι αόριστος και να γίνεται όσο πιο γρήγορα γίνε-

ται. Το πρώτο το κάνουμε μέσω του χρονομέτρου όπως προαναφέραμε, ενώ στο τελευταίο απλά ελέγχουμε αν η ουρά είναι άδεια και όσο ισχύει αυτό κάνουμε συνέχεια νέα frames (δεν έχουμε συμβάντα για frame).

Κατά την σχεδίαση του frame, το πρόγραμμα δίνει εντολές σχεδίασης και μετασχηματισμού των γραφικών και των μοντέλων που υπάρχουν στο παιχνίδι. Αυτό την σήμερον ημέρα γίνεται στην GPU. Ανάλογα τις διαστάσεις του παιχνιδιού (3D ή 2D), το τι συμβαίνει κατά την σχεδίαση του παιχνιδιού διαφέρει. Σε τρισδιάστατα παιχνίδια οι φάσεις περνάνε είναι συνήθως οι ακόλουθες:

1. Μετασχηματισμοί των μοντέλων
2. Μετασχηματισμός σε σχέση με το πλαίσιο όρασης (Camera)
3. Εφαρμογή φωτισμών και σκιών
4. Προβολή πάνω στο δισδιάστατο επίπεδο (Framebuffer)
5. Εφαρμογή υφών (Textures)
6. Εμφάνιση Framebuffer

Σε δισδιάστατα παιχνίδια υπάρχουν λιγότερες λειτουργίες, χάριν του ότι δεν υπάρχουν μοντέλα, φωτισμοί και σκιές. Υπάρχουν μόνο textures (ή αλλιώς στα δισδιάστατα παιχνίδια λέγονται και sprites), τα οποία απλά μετασχηματίζονται και σχεδιάζονται σε συγκεκριμένη σειρά πάνω στην ενδιάμεση μνήμη πλαισίου (Framebuffer).

β) Η Ενδιάμεση Μνήμη Πλαισίου ή Framebuffer

Τι είναι όμως ο framebuffer; Ο framebuffer είναι ένα μέρος της μνήμης της GPU που αποθηκεύεται η εικόνα που θα εμφανιστεί σε κάθε καρέ της οθόνης. Είναι στην ουσία ένας μεγάλος τρισδιάστατος πίνακας που έχει διαστάσεις όσο η ανάλυση της οθόνης επί 3, που είναι τα κανάλια των χρωμάτων RGB (Βλ. σχ. 6.2.1). Το κανάλι της διαφάνειας (Alpha Channel) συνήθως δεν υπάρχει καθώς πίσω από τον framebuffer δεν υπάρχει κάτι. Πχ σε κώδικα C θα μπορούσε να αναπαρασταθεί ως:

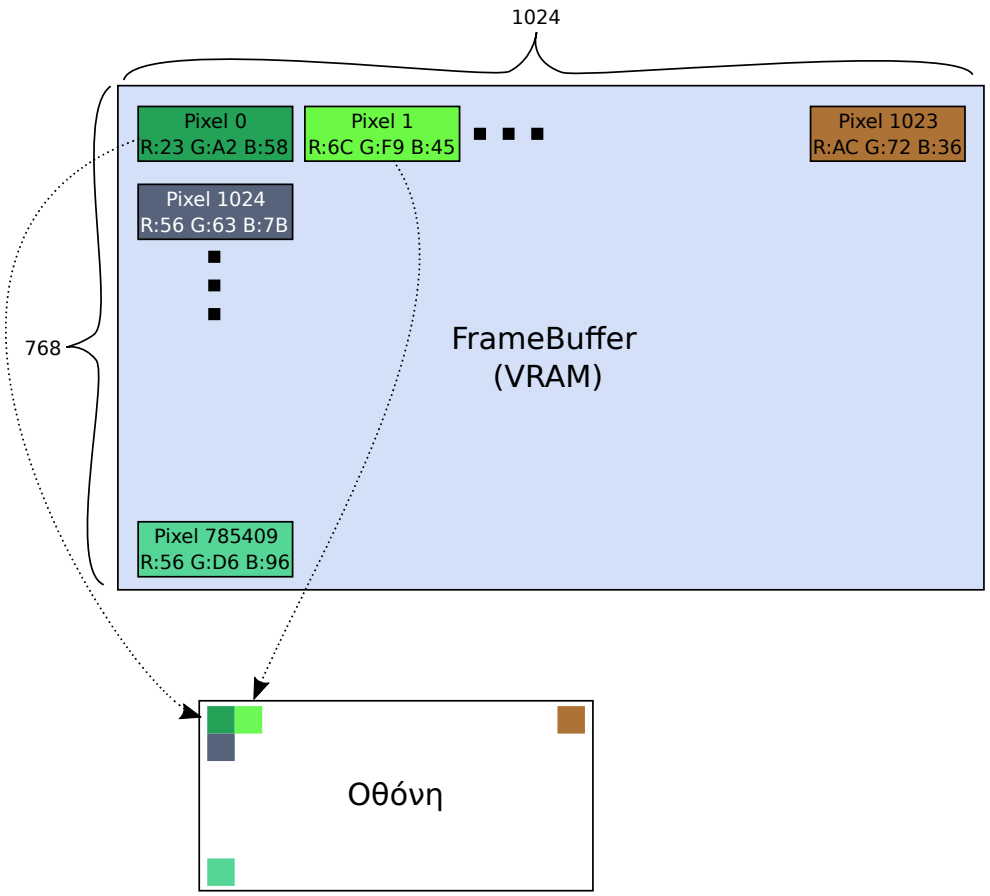
Πίνακας 6.2.2: Παράδειγμα ορισμού FrameBuffer σε κώδικα C

```c
#define SCREEN_WIDTH 1024
#define SCREEN_HEIGHT 768

unsigned char framebuffer[SCREEN_WIDTH][SCREEN_HEIGHT][3];
```

Τα περιεχόμενα του framebuffer τα βλέπουμε στην οθόνη όποια στιγμή και να κοιτάξουμε. Ότι βλέπουμε (παράθυρα, Επιφάνεια Εργασίας, ποντίκι, κτλ.) είναι το περιεχόμενο αυτού του πίνακα στην μνήμη της GPU ή αλλιώς VRAM.

Ο framebuffer επανεμφανίζεται σε κάθε καρέ οθόνης. Ο ρυθμός που γίνεται αυτό εξαρτάται από την ρύθμιση του ρυθμού ανανέωσης της οθόνης. Πχ αν η οθόνη λειτουργεί στα 60 Hz τότε ο framebuffer επανεμφανίζεται 60 φορές το δευτερόλεπτο.

Συνήθως ο framebuffer “καθαρίζει” και ανασχεδιάζεται σε κάθε καρέ. Το καθάρισμα αυτό γίνεται συνήθως με γέμισμα ολόκληρου του framebuffer με ένα χρώμα, συνήθως μαύρο. Μετά αρχίζει να σχεδιάζεται από τους επιμέρους framebuffers που απαρτίζουν το σύστημα. Αυτοί οι framebuffers είναι από τα προγράμματα που τρέχουν στο λειτουργικό σύστημα. Όταν ο κύριος framebuffer πρέπει να επανεμφανιστεί, παίρνει όλους τους επιμέρους και τους αποτυπώνει πάνω του με σειρά και τοποθεσία ανάλογα του προγράμματος που το διαχειρίζεται αυτό (πχ Ο Window Manager που τρέχει μαζί με τον X Server).



Εικόνα 6.2.1: Αναπαράσταση του Κύριου Framebuffer

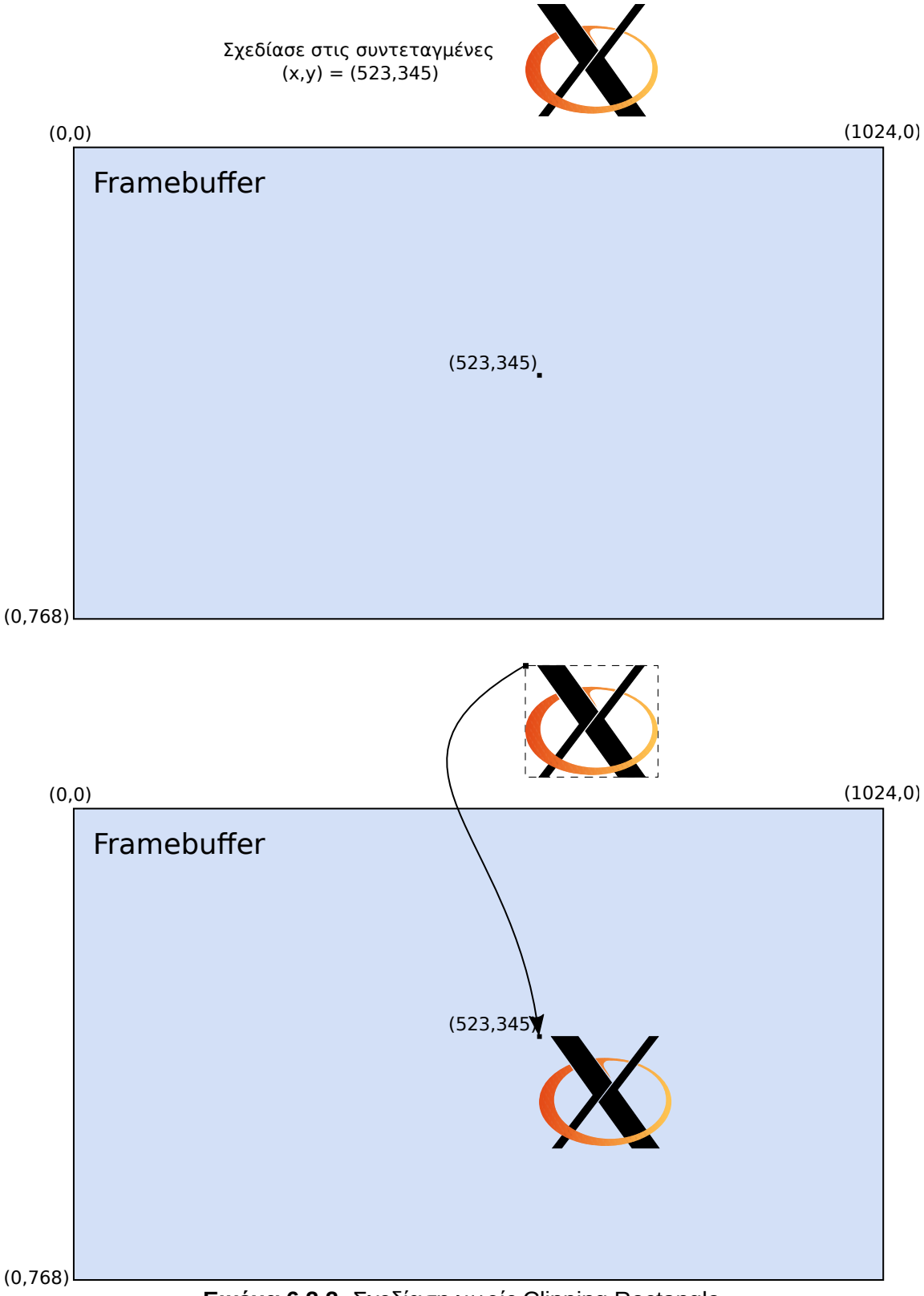
Για παράδειγμα, κάθε παράθυρο σε ένα παραθυρικό περιβάλλον (πχ X Window System, Windows Desktop), υπάρχει ένας framebuffer πίσω από αυτό προκειμένου να μπορεί να εμφανίζει τα γραφικά που θέλει αυτό το παράθυρο. Το πρόγραμμα ασχολείται με το τι θα σχεδιάσει πάνω στον δικό του framebuffer και ο Window Manager ασχολείται που θα τον εμφανίσει πάνω στον κύριο. Αξίζει να σημειωθεί ότι οι επιμέρους framebuffers μπορούν να έχουν κανάλι διαφάνειας (Alpha Channel) στα χρώματα τους.

Ότι σχεδιάζεται πάνω στον επιμέρους framebuffer δεν είναι τίποτα άλλο από εικόνες ή αλλιώς bitmaps. Τα bitmaps είναι σαν τους framebuffers, μόνο που μπορεί να αποθηκεύονται και στην κύρια μνήμη αν δεν υπάρχει αρκετός χώρος στην μνήμη της κάρτας γραφικών (GPU). Αυτό συμβαίνει σπάνια και σε περιπτώσεις που το παιχνίδι είναι πολύ απαιτητικό. Η αποθήκευση στην κύρια μνήμη πρέπει να αποφεύγεται γιατί πρέπει να χρησιμοποιείται μόνο για δεδομένα προγράμματος, καθώς η αποθήκευση πολλών εικόνων σε συνδυασμό με τα δεδομένα των προγραμμάτων που τρέχουν, μπορεί να την γεμίσει εύκολα. Έτσι η τεχνική που εφαρμόζεται σε περιπτώσεις που η VRAM γεμίσει είναι να αποδεσμεύονται οι εικόνες που δεν χρησιμοποιούνται ενεργά εκείνη την στιγμή από το παιχνίδι (πχ δεν φαίνονται πουθενά στο πλάνο) και να φορτώνονται αυτές που χρειάζονται. Αυτό βέβαια επειδή συνεπάγεται διάβασμα από τον σκληρό δίσκο, τότε θα υπάρξει δραματική πτώση στην απόδοση του παιχνιδιού λόγω ότι πριν γίνει ένα frame πρέπει να διαβαστεί η εικόνα από τον σκληρό δίσκο που είναι αργή δουλειά.

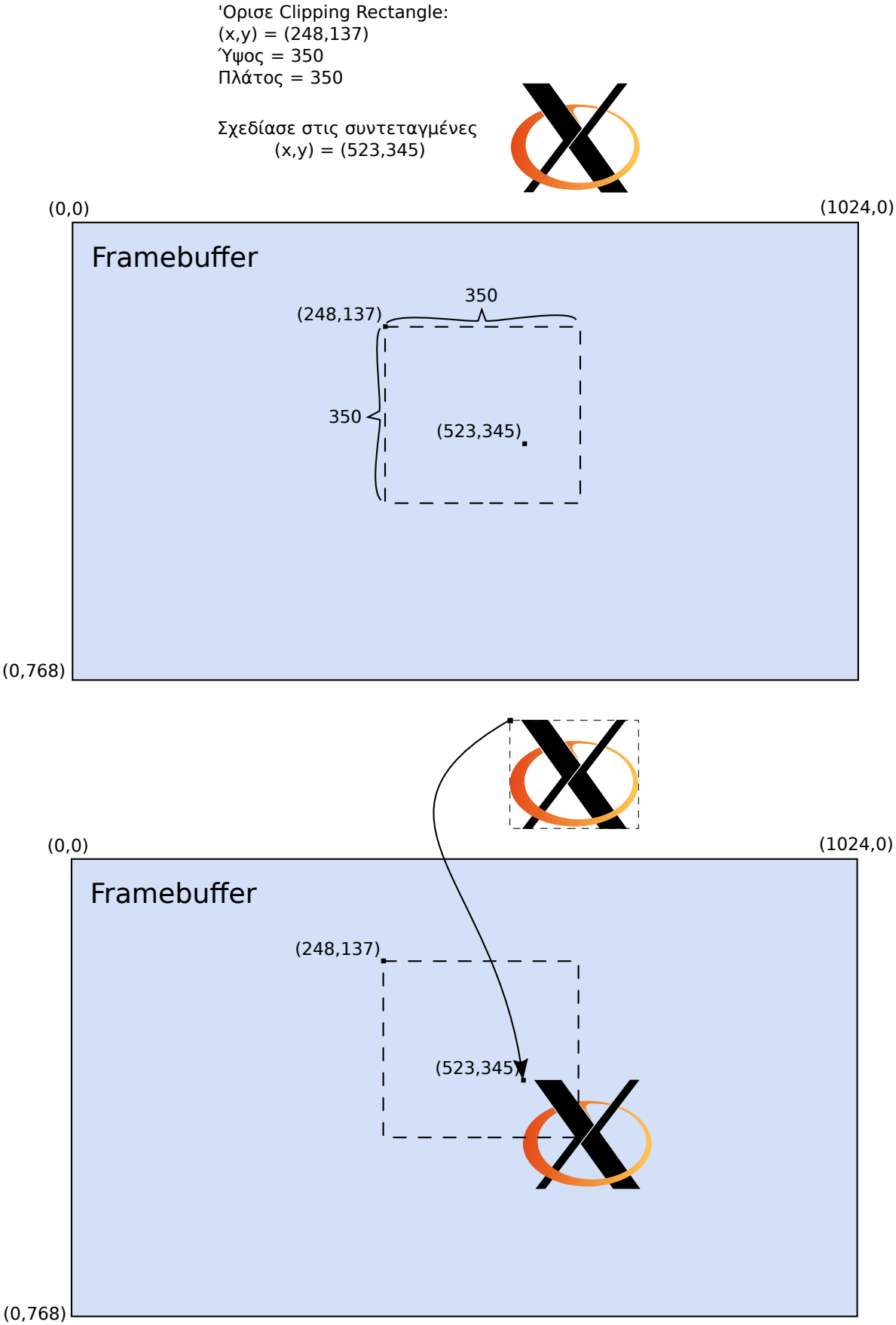
γ) Περικοπή Σχεδίασης

Οι εικόνες σχεδιάζονται στον επιμέρους framebuffer σε τοποθεσίες και μεγέθη που καθορίζει το πρόγραμμα. Εάν κάποια εικόνα σχεδιαστεί εκτός ορίων του framebuffer, δεν υπάρχει πρόβλημα γιατί ο framebuffer περνάει από περικοπή (Clipping). Το Clipping, εφαρμόζει ένα τετράγωνο πάνω στον framebuffer και ότι εξέχει από το τετράγωνο θα αφαιρεθεί, αφήνοντας στο τέλος μια εικόνα στις διαστάσεις του τετραγώνου. Πολλές φορές η περικοπή εφαρμόζεται στον framebuffer συνεχόμενα, με την έννοια ότι όσο η περικοπή θα είναι ενεργή, όσες σχεδιάσεις γίνουν πάνω στον framebuffer, δεν θα εξέχουν εκτός του τετραγώνου περικοπής (Clipping Rectangle). Για να γίνει πιο κατανοητό το πως γίνεται η περικοπή στα παρακάτω σχήματα, πρέπει να γίνουν κάποιες διευκρινίσεις για σύστημα συντεταγμένων της οθόνης. Στην οθόνη το σημείο (0,0) είναι στην πάνω αριστερή γωνία. Δεξιά από αυτό είναι τα

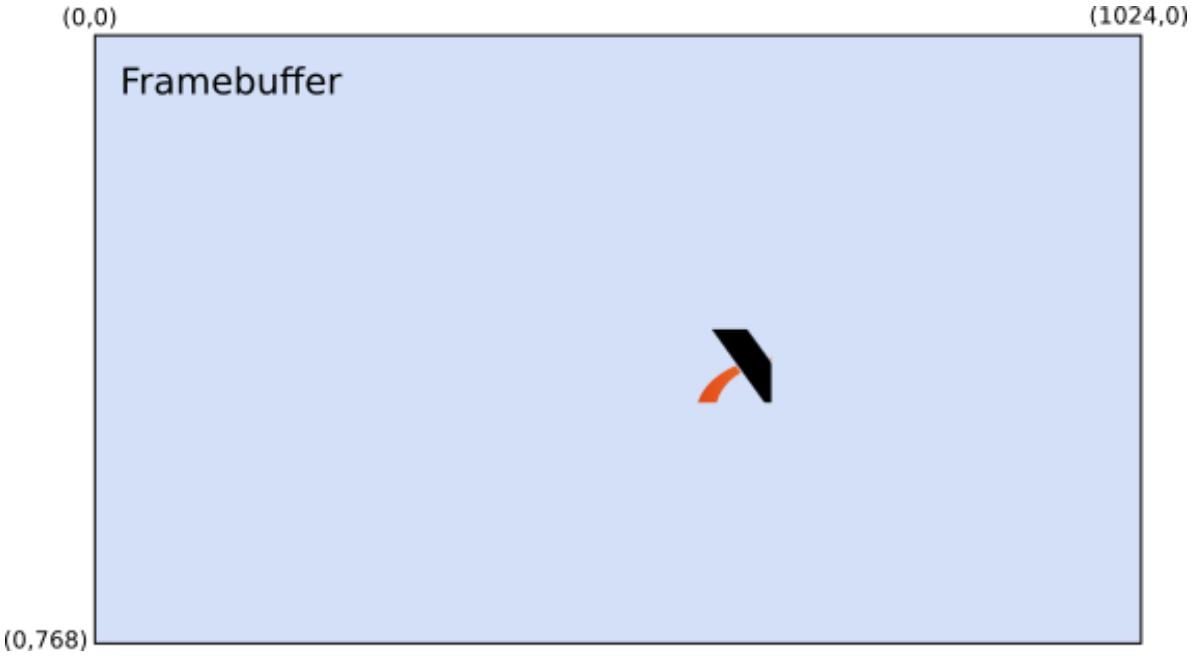
θετικά X, αριστερά τα αρνητικά X, κάτω τα θετικά Y και πάνω τα αρνητικά Y. Όταν λέμε ότι μια εικόνα θα σχεδιαστεί στις συντεταγμένες (x,y) τότε το πάνω αριστερό άκρο της εικόνας θα βρίσκεται στο (x,y) και θα εκτείνεται από εκεί και πέρα προς τα δεξιά και κάτω.



Εικόνα 6.2.2: Σχεδίαση χωρίς Clipping Rectangle



Εικόνα 6.2.3: Σχεδίαση με Clipping Rectangle



Εικόνα 6.2.4: Αποτέλεσμα Σχεδίασης 6.2.3

Όπως είδαμε στα προηγούμενα σχήματα, κατά την σχεδίαση με ορισμένο Clipping Rectangle το αποτέλεσμα της σχεδίασης είναι μόνο το μέρος που είναι εντός του τετραγώνου. Το τετράγωνο αυτό μπορεί να αλλάζει συνεχώς ή ακόμα και να καταργείται και να ξανά-ορίζεται όποτε χρειάζεται.

δ) Συνένωση Εικόνων με κανάλια Διαφάνειας

Όπως είχαμε προαναφέρει, στους επιμέρους framebuffers το τι θα σχεδιαστεί έχει να κάνει με το πρόγραμμα που τον δημιούργησε. Πχ ένα πρόγραμμα με GUI θα σχεδιάσει τα στοιχεία ελέγχου όπως κουμπιά, κείμενα, κτλ. Σε ένα παιχνίδι θα σχεδιαστούν οι εικόνες από τον κόσμο του παιχνιδιού, την εικόνα του παίχτη κτλ. Και στις δυο περιπτώσεις αυτό που θα αποτυπωθεί πάνω στον framebuffer θα είναι μια εικόνα Bitmap. Δεν μπορείς πχ να δώσεις εντολή σχεδίασε στον framebuffer το “κουμπί” ούτε τον “παίχτη”. Αυτό θα πρέπει να το κάνει ο προγραμματιστής.

Δεδομένου ότι οι εικόνες έχουν κανάλι Alpha και ότι μπορούν να αποτυπωθούν στον framebuffer σε οποιοδήποτε σημείο, ακόμα και αν πριν έχει αποτυπωθεί άλλη εικόνα εκεί, πρέπει το παραγόμενο αποτέλεσμα της συνένωσης εικόνων να είναι σωστό. Για παράδειγμα, όταν μια μαύρη εικόνα με διαφάνεια 0,5 (50%) αποτυπωθεί πάνω από μια άσπρη εικόνα, τότε ο framebuffer σε αυτό το σημείο πρέπει να έχει γκρι. Αν δεν υπήρχε η διαφάνεια τότε απλά το παραγόμενο χρώμα θα ήταν αυτό της εικόνας που αποτυπώθηκε πάνω, δηλαδή το μαύρο. Όμως σήμερα η διαφάνεια

υπάρχει παντού και έτσι πρέπει να παίρνουμε τα σωστά αποτελέσματα είτε με διαφάνεια είτε όχι. Για να γίνει αυτό πρέπει όταν γίνεται αποτύπωση πάνω σε μια εικόνα ή τον framebuffer, να εφαρμοστεί ένας τύπος που να παίρνει ως μεταβλητή τις διαφάνειες των εικόνων και να παράγει το σωστό χρώμα για κάθε κανάλι. Ο τύπος για το παραγόμενο χρώμα ενός καναλιού είναι ο παρακάτω:

$$C_0 = \frac{1}{\alpha_0}\left[C_f \alpha_f + C_b \alpha_b (1-\alpha_f)\right] (6.2.1)$$

όπου:

C_f η τιμή του χρώματος της επάνω εικόνας (που αποτυπώνεται)

α_f η τιμή της διαφάνειας της επάνω εικόνας

C_b η τιμή του χρώματος της κάτω εικόνας (πάνω σε αυτή αποτυπώνεται η άλλη)

α_b η τιμή της διαφάνειας της κάτω εικόνας

α_0 η τιμή της παραγόμενης διαφάνειας που δίνεται από τον τύπο:

$$\alpha_0 = \alpha_f + \alpha_b (1-\alpha_f) (6.2.2)$$

Σε κάθε pixel εφαρμόζεται σε κάθε κανάλι χρώματος το τύπος του C_0 και τον τύπο του α_0 για την τελική διαφάνεια του pixel.

Σύμφωνα με αυτούς τους τύπους, για να αποτυπωθούν πολλές εικόνες στον framebuffer, πρέπει να οριστεί μια σειρά με την οποία μία-μία θα σχεδιάζονται. Με άλλα λόγια σε κάθε εικόνα πρέπει να αποδοθεί ένας δείκτης Z (Z-Index) που σύμφωνα με αυτόν τον δείκτη θα αποτυπώνονται πάνω στον framebuffer οι εικόνες με συγκεκριμένη σειρά. Αυτός ο δείκτης και η υλοποίηση του εξαρτάται από το πρόγραμμα που γράφει ο προγραμματιστής.

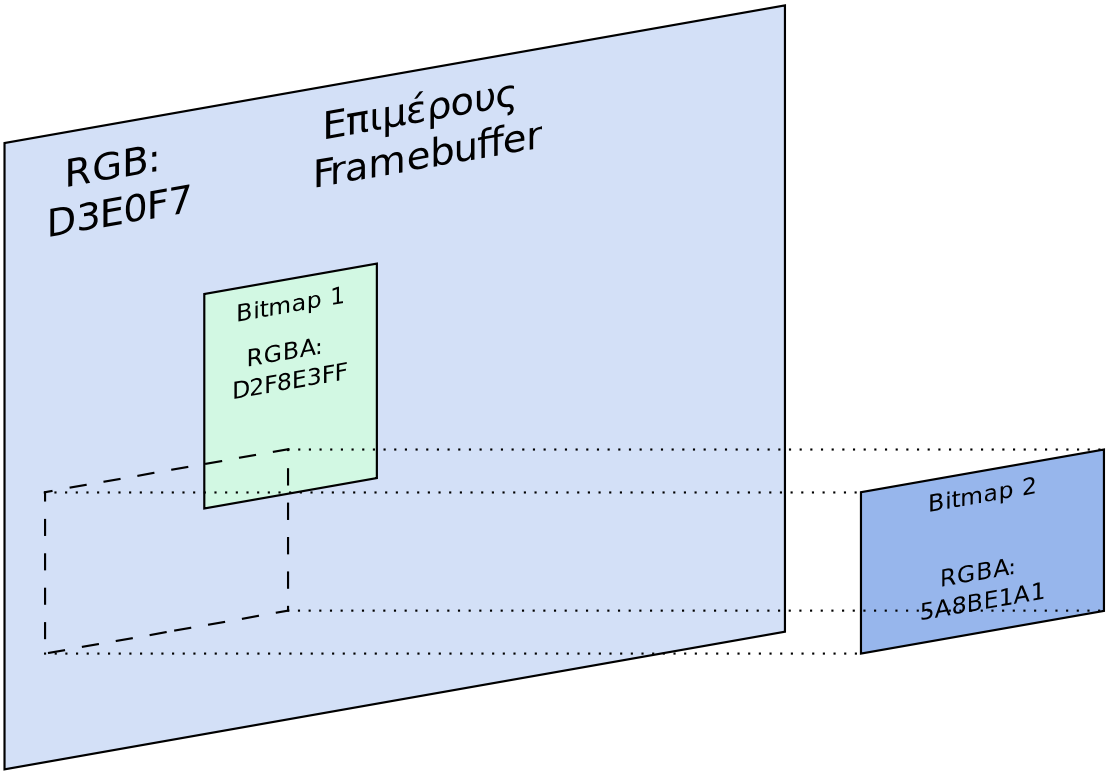
ε) Παράδειγμα Συνένωσης Εικόνων

Έστω ότι έχουμε έναν framebuffer στο πρόγραμμά μας και έχουμε βάλει σε όλα τα pixel του το χρώμα RGBA = 0xD3E0F7FF (R = D3, G = E0, B = F7, Alpha = FF). Στην ουσία είναι μια εικόνα που έχει παντού αυτό το χρώμα. Με Alpha = FF σημαίνει ότι είναι πλήρης αδιαφανής άρα δεν μας ενδιαφέρει τι έχει πίσω της.

Έστω ότι πάνω στον framebuffer αποτυπώνουμε την εικόνα “Bitmap 1” που έχει παντού το χρώμα RGBA = 0xD2F8E3FF. Κατά την αποτύπωση εφαρμόζονται οι τύποι που προαναφέραμε και τα αποτελέσματα μπαίνουν στον framebuffer. Προσοχή! Τα νέα Pixel δεν μπαίνουν πάνω από τα παλιά αλλά στα pixel της περιοχής του

framebuffer που αποτυπώνεται η εικόνα αλλάζουν οι τιμές των καναλιών τους και παίρνουν τις τιμές των αποτελεσμάτων. Τα Pixel της εικόνας σε καμία περίπτωση δεν αλλάζουν. Πάντα τα pixel του framebuffer αλλάζουν με νέες τιμές.

Έστω ότι θέλουμε να αποτυπώσουμε την εικόνα “Bitmap 2” που έχει παντού RGBA = 0x5ABE1A1. Αυτή η εικόνα δεν είναι πλήρης αδιαφανής (A = A1) και πρέπει να αποτυπωθεί σε σημείο όπου θα πέσει σε σημείο που είχε αποτυπωθεί η “Bitmap1”.

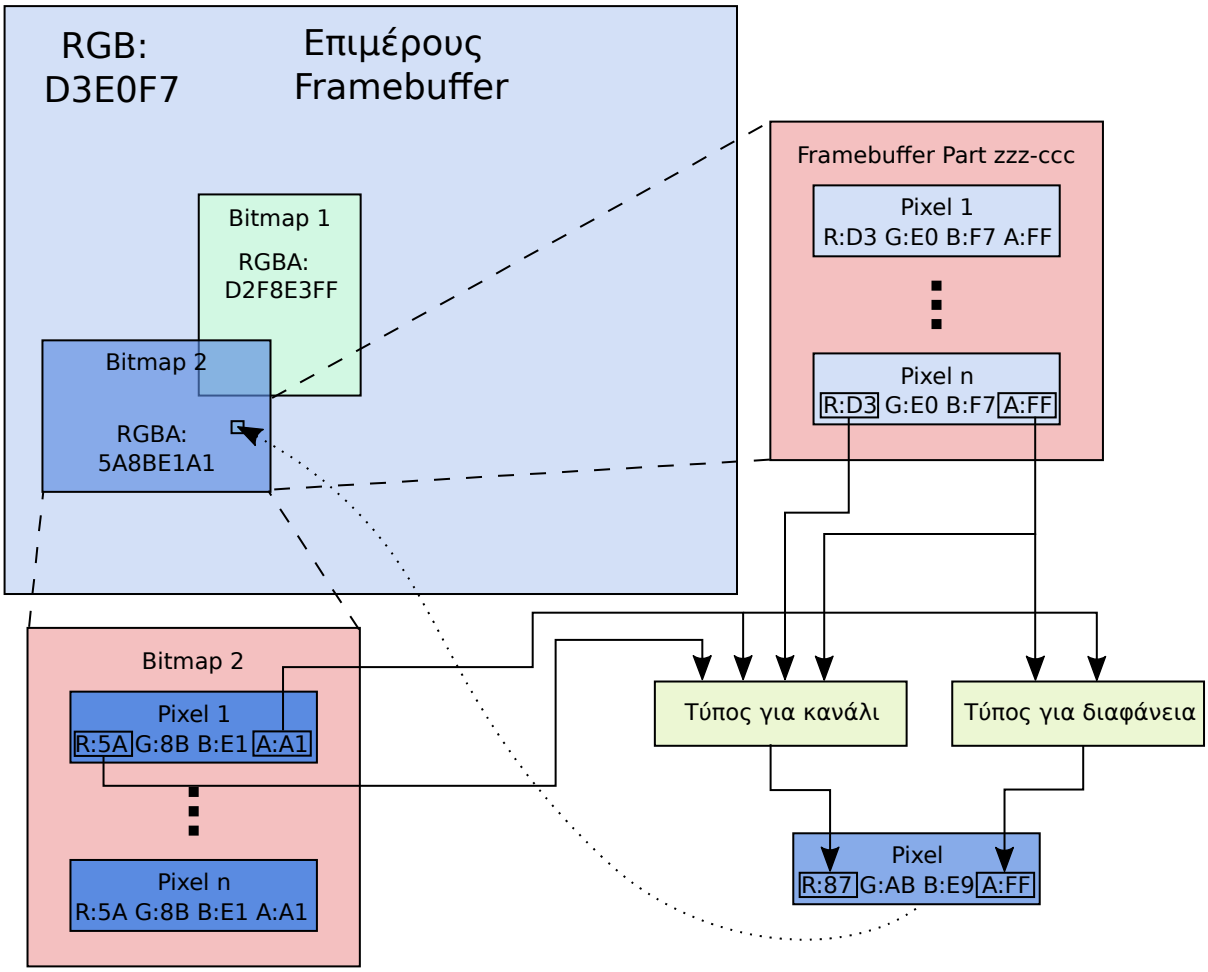


Εικόνα 6.2.5: Παράδειγμα σχεδίασης της Bitmap 2 στον Framebuffer

Όπως είπαμε, δεν πρέπει να μας επηρεάζει που στον framebuffer “Φαίνεται” η “Bitmap 1” και πέφτει στο σημείο που θα σχεδιαστεί η “Bitmap 2”. Απλά τα Pixel του framebuffer έχουν σε αυτό το σημείο τιμές που έχει ως αποτέλεσμα να φαίνεται η “Bitmap 1”. Τίποτα διαφορετικό δεν γίνεται από πριν. Παίρνουμε όλα τα pixel στο σημείο του framebuffer και της εικόνας “Bitmap 2” και εφαρμόζουμε τους τύπους για να πάρουμε τις τιμές που θα πρέπει να βάλουμε σε αυτά τα Pixel του.

Συνεχίζοντας την ίδια δουλειά, η κάρτα γραφικών (συνήθως) κάνει την ίδια διαδικασία για όλα τα Pixel της εικόνας που αποτυπώνεται (“Bitmap 2”). Όταν ολοκληρωθεί αυτή η διαδικασία, ο framebuffer θα μοιάζει όπως στο παραπάνω σχήμα, δηλαδή θα

φαίνονται και οι δυο εικόνες με την “Bitmap 2” να φαίνεται πάνω στην “Bitmap 1” λόγο της σειράς που τις σχεδιάσαμε.



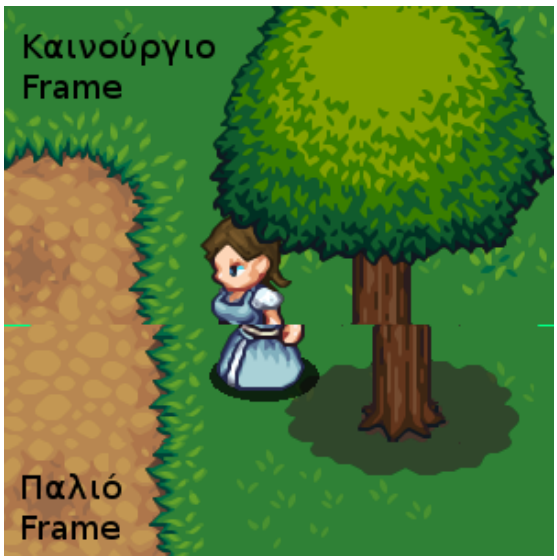
Εικόνα 6.2.6: Σχεδίαση της "Bitmap 2" σε σημείο του Framebuffer που έχει δεδομένα της "Bitmap 1"

στ) Πολλαπλοί Framebuffers, Page Flipping και V-Sync

Έστω ότι ένα πρόγραμμα έχει έναν framebuffer. Όταν σχεδιάζεται και αδειάζεται από το πρόγραμμα, υπάρχουν περιπτώσεις που η κάρτα γραφικών θα τον δείξει στην οθόνη σε ακατάλληλη στιγμή. Τέτοιες περιπτώσεις μπορεί να είναι:

- Ο framebuffer βρίσκεται στην φάση του αδειάσματος. Εκεί το πρόγραμμα είχε ξεκινήσει να αδειάζει τον framebuffer του γεμίζοντας τον με κάποιο χρώμα και η οθόνη τον έδειξε πάνω που ήταν μισό-άδειος ή τελείως άδειος. Εκεί φαίνεται τότε στο πρόγραμμα ένα Frame που από την μέση και πάνω είναι ένα χρώμα, και από κάτω κανονικό ή δεν δείχνει τίποτα.

- Ο framebuffer βρίσκεται στην φάση της σχεδίασης. Σε αυτή την περίπτωση, το πρόγραμμα έχει αρχίσει ήδη να σχεδιάζει το επόμενο Frame, αλλά η οθόνη τον έδειξε όταν δεν είχε τελειώσει με συνέπεια να εμφανιστούν στον χρήστη μισά από ότι θα έπρεπε. Σε περιπτώσεις που ο framebuffer δεν καθαρίζεται, άλλα γεμίζει από πάνω κατευθείαν, τότε το frame θα δίνει μισό παλιό και μισό καινούργιο. Τότε παρατηρείται το φαινόμενο σπασίματος της εικόνας (Screen Tearing).



Εικόνα 6.2.7: Εξομοίωση του φαινομένου Screen Tearing

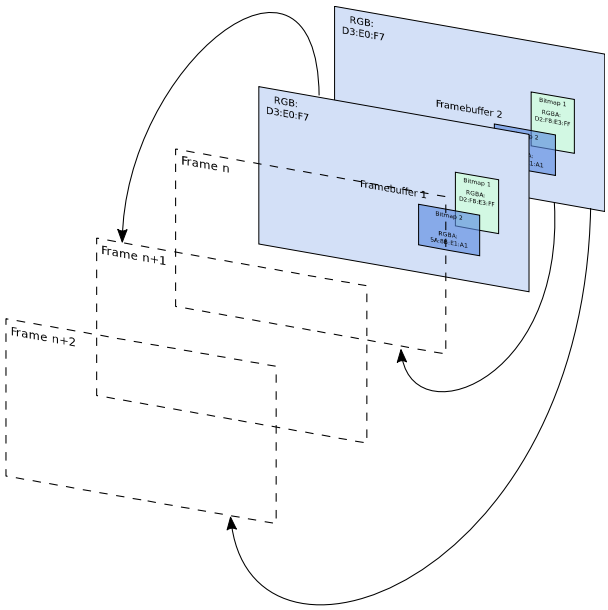
Οι παραπάνω περιπτώσεις μπορεί να φαίνονται σπάνιες αλλά δεν είναι. Όταν οι ρυθμοί ανανέωσης της οθόνης σήμερα ξεπερνούν τα 60hz, τότε τα φαινόμενα αυτά είναι πολύ συχνά σε βαθμό που γίνεται πολύ ενοχλητικό για τον χρήστη. Ειδικά σε περίπτωση που η εικόνα έχει γρήγορες κινήσεις, το φαινόμενο Tearing είναι έντονο.

Υπάρχουν κάποιες λύσεις που βοηθάνε στο να μειωθεί ή να εξαλειφθεί το φαινόμενο αυτό. Καμιά δεν είναι τέλεια. Όλες οι υπάρχουσες έχουν τα μειονεκτήματα και τα πλεονεκτήματα τους.

- **Double Buffering.** Εδώ το πρόγραμμα έχει τουλάχιστον 2 framebuffers, έναν εξωτερικό που φαίνεται (Front Buffer) που βρίσκεται στην VRAM, και έναν από πίσω που σχεδιάζονται (Back Buffer) που βρίσκεται στην κύρια μνήμη RAM. Όταν έρθει η ώρα για σχεδίαση, το πρόγραμμα παίρνει τον Back Buffer και τον σχεδιάζει. Αφού τον σχεδιάσει, τον αντιγράφει ολόκληρο στον Front Buffer και αυτός διαβάζεται από την οθόνη όποτε χρειαστεί. Το πλεονέκτημα εδώ είναι ότι πάντα στην οθόνη θα εμφανίζεται κάτι ολοκληρωμένο, γιατί όταν

γίνει η αντιγραφή από τον Back Buffer θα έχει τελειώσει η σχεδίαση εκεί. Το φαινόμενο του τρέμουλου εξαλείφεται αλλά το φαινόμενο Tearing δεν εξαλείφεται τελείως. Ακόμα παρατηρούνται μισά-μισά frame σε γρήγορες κινήσεις όπως στο σχήμα 6.2.6 γιατί η εμφάνιση του Front Buffer μπορεί να γίνει την ώρα που γίνεται η αντιγραφή.

- **Double Buffering με Page Flipping.** Η λειτουργία εδώ είναι παρόμοια με το Double Buffering, αλλά εδώ και οι δυο Buffers βρίσκονται στην VRAM. Οι Buffers εναλλάσσουν θέσεις μπρος-πίσω, έτσι σε κάθε Frame μία ο ένας εμφανίζεται στην οθόνη μία ο άλλος. Όσο ο ένας είναι διαθέσιμος να διαβαστεί από την οθόνη, ο άλλος σχεδιάζεται. Όταν έρθει το επόμενο Frame, ανταλλάζουν θέση ο Front Buffer με τον Back, και ο Front πλέον σχεδιάζεται ενώ ο Back είναι διαθέσιμος για εμφάνιση. Αυτό επαναλαμβάνεται συνέχεια. Τα πλεονεκτήματα εδώ είναι ότι και οι δυο Framebuffers βρίσκονται στο στην VRAM και έτσι απλά αλλάζουν θέση και δεν χρειάζεται να αντιγραφούν ο ένας στον άλλον. Αυτό επιτυγχάνει να αυξήσει την απόδοση των Frames γιατί δεν γίνεται αντιγραφή δεδομένων που είναι αρκετά βαριά δουλειά όταν δεν γίνεται μεταξύ δεδομένων της VRAM. Το Tearing εμφανίζεται και εδώ αλλά αν συνδυαστεί με το V-Sync, τότε εξαλείφεται τελείως. Το μειονέκτημα εδώ είναι ότι στην VRAM πρέπει να δεσμευτεί διπλάσια μνήμη για τον Framebuffer. Αυτό με τις σημερινές GPU που έχουν μεγάλη μνήμη VRAM, δεν είναι πρόβλημα, άλλα παλιότερα θα ήταν.

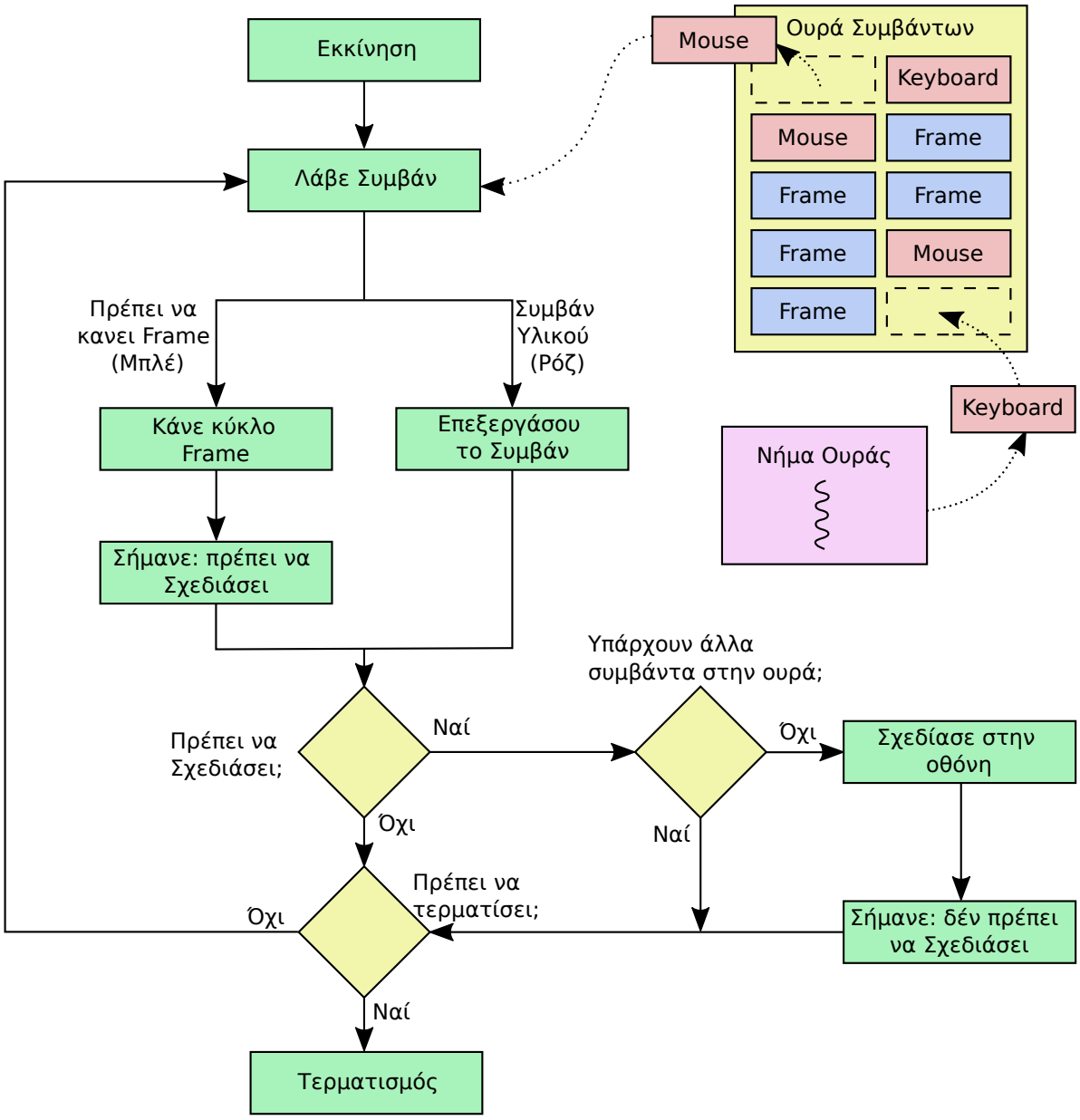


Εικόνα 6.2.8: Αναπαράσταση του Page Flipping

- **V-Sync.** Πρόκειται για τον κάθετο συγχρονισμό από όπου και προέρχεται η λέξη (Vertical Synchronization). Εδώ το Frame σχεδιάζεται και γίνεται διαθέσιμο για εμφάνιση περιμένοντας μέχρις ότου να σημάνει η οθόνη ότι θα κάνει ανανέωση. Μόλις η οθόνη γίνει έτοιμη, το Frame εμφανίζεται και το πρόγραμμα συνεχίζει την εκτέλεση του. Το πλεονέκτημα εδώ είναι ότι το Screen Tearing εξαλείφεται εντελώς γιατί σε κάθε ανανέωση της οθόνης, το Frame είναι έτοιμο και διαθέσιμο για εμφάνιση. Υπάρχουν όμως δύο μειονεκτήματα.
  - Ο ρυθμός διεξαγωγής των Frame (FPS) κλειδώνεται αυτόματα στον ρυθμό ανανέωσης της οθόνης. Ακόμα και το παιχνίδι να μπορεί να τρέξει πολύ πιο γρήγορα, θα τρέξει όσο πάει η οθόνη, λόγω ότι πολύ απλά συγχρονίζεται με αυτή.
  - Λόγω ότι το πρόγραμμα περιμένει την οθόνη για να συνεχίσει, στο διάστημα αυτό το πρόγραμμα δεν κάνει τίποτα. Αυτό σημαίνει ότι τα συμβάντα που θα μπούνε στην ουρά για επεξεργασία θα πρέπει να περιμένουν την οθόνη να εμφανίσει, και έτσι δεν επεξεργάζονται εγκαίρως. Έτσι δημιουργείται μια καθυστέρηση στην ανταπόκριση του προγράμματος με τις εντολές του χρήστη, γνωστό και ως Input Lag. Για παράδειγμα, ο χρήστης έκανε κλικ με το ποντίκι να κινηθεί ο χαρακτήρας, άλλα αυτός ξεκίνησε 200ms αργότερα. Μπορεί ο χρόνος αυτός να φαίνεται μικρός στο νούμερο, άλλα σε πραγματικού χρόνου (Real Time) εφαρμογές όπως τα παιχνίδια ARPG, η καθυστέρηση είναι αισθητή. Αυτό είναι ιδιαιτέρα ενοχλητικό και ίσως μερικές φορές επηρεάσει το παιχνίδι του χρήστη σε βαθμό που δεν μπορεί να το ολοκληρώσει.
- **Επερχόμενες Τεχνολογίες.** Η VESA έχει βάλει στο Standard του Display Port 1.2 το πρότυπο Adaptive-Sync. Αυτή η τεχνολογία παρέχει στις οθόνες προσαρμόσιμο ρυθμό ανανέωσης που ορίζεται την εκάστοτε στιγμή από την GPU. Με άλλα λόγια ο ρυθμός ανανέωσης της οθόνης συγχρονίζεται στα FPS του παιχνιδιού. Το φαινόμενο Screen Tearing εξαλείφεται χωρίς να επηρεάζεται το παιχνίδι. Το μόνο μειονέκτημα αυτού είναι ότι χρειάζεται οθόνη και GPU που το υποστηρίζει.

ζ) Ο Κύριος Βρόγχος της Zeta Engine

Ο κύριος βρόγχος της μηχανής είναι ο πιο κοινός βρόγχος για δισδιάστατες εφαρμογές. Υπάρχουν τουλάχιστον 2 νήματα, ένα κύριο και ένα για να λαμβάνει τα συμβάντα. Το νήμα για τα συμβάντα γεμίζει μια ουρά με αυτά και το κύριο νήμα έχει την αρμοδιότητα να την αδειάζει, κάνοντας ότι πρέπει για κάθε συμβάν. Τα Frames είναι και αυτά χρονικά συμβάντα. Όταν δεν υπάρχει άλλο συμβάν για εκπλήρωση, τότε σχεδιάζεται ένα Frame.



Εικόνα 6.2.9: Ο Κύριος βρόγχος της Μηχανής με την Allegro5

6.3 Νήματα και παράλληλος προγραμματισμός

Όταν εκτελείται ένα πρόγραμμα, οι εντολές του εκτελούνται η μια μετά την άλλη σειριακά. Πολλές φορές όμως το ίδιο το πρόγραμμα χρειάζεται να κάνει μια δεύτερη δουλειά παράλληλα χωρίς να σταματήσει την υπάρχουσα. Για αυτές τις περιπτώσεις υπάρχουν τα νήματα.

Τα νήματα είναι κομμάτια του προγράμματος που για το πρόγραμμα φαίνονται να τρέχουν παράλληλα με το κύριο νήμα. Στην πραγματικότητα δεν είναι απαιτητό να γίνεται αυτό. Τα νήματα εκτελούνται από το λειτουργικό σύστημα με τέτοιο τρόπο, που ακόμα και σε ένα μονοπύρηνο σύστημα (Με έναν επεξεργαστή) θα φαίνεται ότι εκτελούνται παράλληλα χωρίς καθυστερήσεις. Αυτό γίνεται με εκμετάλλευση της απίστευτης ταχύτητας των επεξεργαστών για χρονοπρογραμματισμό στον πιο βέλτιστο βαθμό.

Πέρα από τις παράλληλες δουλειές, τα νήματα χρησιμοποιούνται για παράλληλη επεξεργασία δεδομένων. Όταν για παράδειγμα έχουμε ένα μεγάλο κομμάτι δεδομένων που χρειάζεται να επεξεργαστεί με συγκεκριμένο τρόπο τότε μπορούμε να σπάσουμε την δουλειά αυτή σε μικρότερες με νήματα. Κάθε νήμα θα πάρει ένα κομμάτι από τα δεδομένα και θα το επεξεργαστεί. Σε συστήματα με πολλούς επεξεργαστές αυτό θα μειώσει τον χρόνο ολοκλήρωσης της δουλειάς απίστευτα.

Τα νήματα μπορεί να φαίνονται πολύ βολικά άλλα δεν μπορούν να χρησιμοποιούνται χωρίς αναγκαίο λόγο. Αυτό γιατί το να ανοίξεις νήματα δεν είναι “Ελαφριά” δουλειά για το πρόγραμμα. Ακόμα και να ήταν γρήγορο να τα ανοίξεις, για να επωφεληθεί το πρόγραμμα αυτά πρέπει το ίδιο το πρόγραμμα να μπορεί να παραλληλοποιηθεί. Αν δεν μπορεί, τότε δεν θα υπάρξει βελτίωση, αντίθετα θα μειωθεί η απόδοσή του λόγω του ότι τα νήματα θα εναλλάσσουν την θέση τους στον επεξεργαστή (Context Switch) κατά την εκτέλεση, κάτι που είναι επίσης βαριά δουλειά.

Πέρα από αυτά, τα νήματα χρειάζονται συγχρονισμό μεταξύ τους. Για παράδειγμα, όταν ανοίγουμε ένα νήμα για να κάνει μια δουλειά και χρειαζόμαστε το αποτέλεσμα του, θα χρειαστεί να περιμένουμε να τελειώσει στο κύριο νήμα και να πάρουμε το αποτέλεσμα.

Οι δυσκολίες των νημάτων δεν σταματάνε εκεί. Όταν τα νήματα χρησιμοποιούν κοινόχρηστες μεταβλητές (μεταβλητές που έχουν πρόσβαση και τις αλλάζουν ταυτόχρονα πολλά νήματα), τότε υπάρχουν πολύ συχνά τα φαινόμενα **Race Condition**.

Race Conditions

Έστω ότι 2 νήματα έχουν την αρμοδιότητα να αυξάνουν μια μεταβλητή. Σε κάθε αύξηση της μεταβλητής, θα πρέπει να διαβαστεί από την μνήμη, να αυξηθεί και να εγγραφεί πίσω στην μνήμη. Το πρώτο νήμα διαβάζει “4”. Μετά το δεύτερο διαβάζει “4”. Και τα δυο νήματα την αυξάνουν στην τοπική τους μεταβλητή και έχουν “5” στην προσωρινή τους μνήμη. Σε επόμενη φάση και τα 2 νήματα βάζουν στην κοινόχρηστη το “5” κάτι που είναι λάθος. Θα έπρεπε η κοινόχρηστη να είχε “6”. Ας δούμε τι γίνεται στην πράξη. Παρακάτω δίνεται κώδικας για την παραπάνω λειτουργία. Δεν είναι παράλληλος άλλα δεν θέλουμε να επικεντρωθούμε για τώρα σε αυτό.

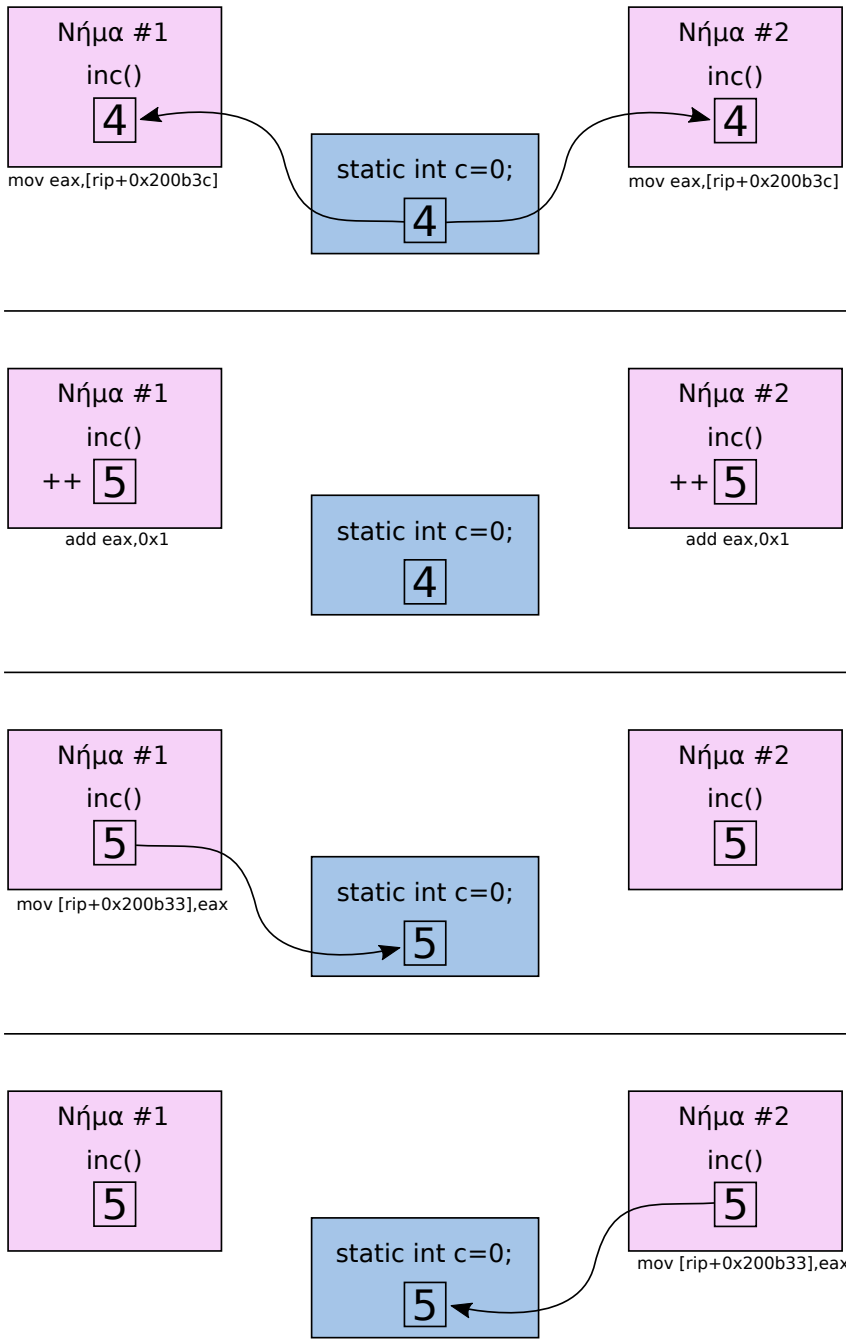
Πίνακας 6.3.1: Παράδειγμα για το Race Condition σε C

```c
static int c = 0;

void inc() {
    ++c;
}
```

Πίνακας 6.3.2: Η inc() του πίνακα 6.3.1 μεταγλωττισμένη σε x86-64 Assembly με σύνταξη Intel

```
inc():
   push rbp            # Αποθήκευσε τον Base Pointer στην Στοίβα
   mov rbp,rsp   # θέσε νέο Base Pointer απο τον τρέχουσο Stack Pointer
   mov eax,DWORD PTR [rip+0x200b3c]            # Βαλε στον eax το c
   add eax,0x1                              # Πρόσθεσε στον eax +1
   mov DWORD PTR [rip+0x200b33],eax   # Βάλε στο c το περιεχόμενο του eax
   pop rbp       # Ανάκτησε τον Base Pointer απο την στοίβα
   ret           # Επέστρεψε
```

Όπως βλέπουμε, η διαδικασία αύξησης της `c` γίνεται σε 3 εντολές. Αν ένα νήμα βρεθεί στον ίδιο κώδικα μαζί με ένα άλλο και εκτελέσει την εντολή διαβάσματος πριν το άλλο νήμα εκτελέσει την εντολή εγγραφής τότε θα έχουμε Race Condition.



Εικόνα 6.3.1: Σχηματική αναπαράσταση του Race Condition

Για να πάρουμε σωστά αποτελέσματα, υπάρχει η ανάγκη να εκτελεστεί ένα μέρος του κώδικα από μόνο ένα νήμα την φορά. Σε αυτό βοηθάνε τα Mutex.

Τα Mutex είναι αντικείμενα-κλειδαριές που εγγυούνται ότι θα κλειδωθούν και θα ξεκλειδωθούν μόνο από ένα νήμα την φορά. Έτσι όταν ένα νήμα κλειδώσει ένα Mutex τότε θα πρέπει να το ξεκλειδώσει προκειμένου να ξανά κλειδωθεί (είτε από το ίδιο το νήμα είτε από άλλο). Με αυτά τα χαρακτηριστικά, μπορούμε να κλειδώσουμε κομ-

μάτια κώδικα για να εκτελεστούν από μόνο ένα νήμα την φορά. Αυτό το κομμάτι κώδικα λέγετε κρίσιμο σημείο (Critical Section).

Αυτό επιτυγχάνεται με μηχανισμούς που όταν ένα νήμα δοκιμάσει να κλειδώσει ένα Mutex που είναι ήδη κλειδωμένο, τότε θα περιμένει εκεί μέχρι να ξεκλειδώσει. Όταν ξεκλειδώσει και του δοθεί η σειρά να το κλειδώσει, τότε το κλειδώνει και συνεχίζει την εκτέλεση.

Αν ένα νήμα περιμένει να ξεκλειδώσει ένα Mutex που για κάποιο λόγο δεν ξεκλειδώνει ποτέ, τότε το νήμα θα παγώσει για πάντα εκεί. Επειδή τα άλλα νήματα του προγράμματος θα περιμένουν αυτό να τελειώσει, τότε παγώνουν και τα υπόλοιπα με συνέπεια να παγώνει όλο το πρόγραμμα. Αυτή η κατάσταση λέγεται αδιέξοδο (Deadlock).

Deadlock μπορεί να επιτευχθεί εύκολα για παραδείγματος χάρη, όταν ένα νήμα προσπαθεί να κλειδώσει ένα Mutex που το έχει κλειδώσει το ίδιο το νήμα προηγουμένως. Παρόλο που το Mutex έχει κλειδωθεί από το ίδιο το νήμα δεν μπορεί να το ξανά κλειδώσει γιατί έτσι ακριβώς λειτουργεί- πρέπει να ξεκλειδώσει πρώτα για να κλειδωθεί ξανά. Αυτό είναι συχνό πρόβλημα σε αναδρομικές συναρτήσεις.

Αυτό το πρόβλημα λύνεται με μια άλλη δομή Mutex που λέγεται αναδρομικό Mutex (Recursive Mutex). Αυτό το είδος Mutex δίνει την δυνατότητα να κλειδωθεί πολλές φορές χωρίς να ξεκλειδωθεί αλλά μόνο από ένα νήμα. Για να μπορεί να κλειδωθεί από άλλο νήμα, θα πρέπει να ξεκλειδωθεί από το νήμα που το κλείδωσε όσες φορές το κλείδωσε. Τα αναδρομικά Mutex είναι πολύπλοκα, προσθέτουν επιπλέον καθυστερήσεις στο κλείδωμα τούς και συνήθως πρέπει να αποφεύγονται και να γίνεται απλοποίηση του προγράμματος με απλά Mutex.

Ο κώδικας που είναι φτιαγμένος ώστε να φέρει τα ίδια αποτελέσματα τόσο σε μονο-νηματικό περιβάλλον, τόσο και σε πολυ-νηματικό, λέγεται Thread-Safe κώδικας.

Το πρότυπο 2011 της C++ προσφέρει φορητά νήματα στην stdc++. Ένα παράδειγμα παρατίθεται παρακάτω.

Πίνακας 6.3.3: Παράδειγμα Thread-Safe Κώδικας σε C++11

```
#include <thread>
#include <iostream>
#include <mutex>

// Ορίζουμε την κοινή μεταβλητή μας
static int c = 0;
// Ορίζουμε το Mutex μας
static std::mutex my_mutex;

void inc() {
     // Φτιάχνουμε μια κλειδαριά με το my_mutex
     std::unique_lock<std::mutex> my_lock(my_mutex);

     // Ξεκλειδώνουμε το Mutex γιατι κλειδώνει αυτόματα
     // όταν φτιάχνουμε την κλειδαριά. Αν προσπαθήσουμε να την
     // κλειδώσουμε ενώ είναι ίδη κλειδωμένη απο το ίδιο το νήμα,
     // θα καταλήξουμε σε αδιέξοδο (Deadlock)
     my_lock.unlock();

     for (int i = 0; i < 100000; i++) {
           // Κλειδώνουμε το Mutex
           my_lock.lock();
           // Εκτελούμε τον Κρίσιμο κώδικα
           ++c;
           // Ξεκλειδώνουμε το Mutex
           my_lock.unlock();
     }
}

int main() {
     // Ανοίγουμε ένα νημα που καλεί την inc()
     std::thread th = std::thread(&inc);

     inc(); // Καλούμε την inc() απο αυτό το νήμα

     th.join();  // Περιμένουμε μέχρι να τελειώσει το νήμα που ανοίξαμε

     std::cout << c << std::endl;  // Εμφανίζουμε το αποτελεσμα

     return 0;
}
```

Στον παραπάνω κώδικα, το `std::unique_lock<>` δημιουργείται όταν γίνει η κλήση της `inc()` και προσπαθεί να κλειδώσει αυτόματα. Μέσα στην `for`, κλειδώνουμε το Mutex, αυξάνουμε την `c`, και το ξεκλειδώνουμε για να μπορεί το άλλο νήμα να την αυξήσει. Η χρήση του `std::unique_lock<>` είναι λίγο άστοχη εδώ άλλα ήθελα να δείξω πως το κρίσιμο σημείο περικλείεται από κλείδωμα και ξεκλείδωμα του Mutex. Το Template `std::lock_guard<>` θα βόλευε πιο πολύ. Αυτό το Template κατά την κλήση του κατασκευαστή (Constructor) περιμένει μέχρι να κλειδώσει τον Mutex που του δίνεται ως όρισμα. Μόλις κλειδώσει συνεχίζει την εκτέλεση στο μπλοκ κώδικα (κώδικας

μέσα σε { }) που κατασκευάστηκε. Όταν η εκτέλεση βγει από το μπλοκ που ορίστηκε, όπως όλες οι προσωρινές μεταβλητές και αντικείμενα που ορίστηκαν εκεί, θα καταστραφούν. Όταν το `std::lock_guard<>` καταστραφεί, θα ξεκλειδώσει το Mutex που του είχε δοθεί όταν κατασκευάστηκε. Με το `std::lock_guard<>` η υλοποίηση της **`inc`**`()` γίνεται πιο απλή και δεν κινδυνεύουμε από Deadlock όπως στην προηγούμενη.

Πίνακας 6.3.4: Απλοποίηση της inc()

```
void inc() {
     for (int i = 0; i < 100000; i++) {
           std::lock_guard<std::mutex> lock(my_mutex);
           ++c;
     }
}
```

Condition Variables και Spurious Wakes

Στο παράλληλο προγραμματισμό με νήματα υπάρχουν κάποιες φορές που χρειάζεται ένα νήμα να περιμένει να δεχτεί ένα σήμα από άλλο νήμα για να συνεχίσει την δουλειά του. Αυτό γίνεται με τα Condition Variables. Αυτά είναι κάποια κοινόχρηστα αντικείμενα που κάνουν κάποιο νήμα να περιμένει μέχρι αυτή η μεταβλητή να δώσει σήμα να συνεχίσει. Το σήμα αυτό προέρχεται από άλλο νήμα που δίνει αυτή την εντολή στην μεταβλητή να “Ξυπνήσει” (Wakeup) όσα νήματα “Κοιμούνται” (Sleep) από αυτήν την.

Τα Condition Variables χρειάζονται ένα κλειδωμένο από το νήμα Mutex για να λειτουργήσουν σωστά. Όταν αρχίσει να περιμένει τότε ξεκλειδώνει αυτόν τον Mutex. Όταν θα ξυπνήσει θα κλειδώσει τον Mutex και θα συνεχίσει.

Το ξύπνημα του νήματος δεν είναι πάντα προβλέψιμο. Υπάρχουν πολλές περιπτώσεις που το νήμα θα ξυπνήσει και θα συνεχίσει ενώ δεν του έχει δοθεί τέτοιο σήμα. Αυτά τα ξυπνήματα λέγονται Spurious Wakeups.

Τέτοιες περιπτώσεις εννοείται ότι δεν είναι επιθυμητές. Για να αποφευχθούν θα πρέπει να εισάγουμε μια ακόμη μεταβλητή που θα ελέγχεται κάθε φορά που ξυπνάει το νήμα. Όταν θα πρέπει να ξυπνήσει ένα νήμα θα πρέπει πέρα από το σήμα που θα δοθεί στο Condition Variable, να αλλάξει και η δευτερεύουσα μεταβλητή για να συνεχίσει το κοιμώμενο νήμα.

Πίνακας 6.3.5: Παράδειγμα σωστού Wakeup

```
#include <thread>
#include <condition_variable>

static std::mutex mutex;
static std::condition_variable var;
static bool threadShouldWake = false;

void threadFunc1() {
     // Πρέπει να έχουμε κλειδωμένο τον Mutex πριν
     // μπούμε σε wait
     std::unique_lock<std::mutex> lock(mutex);

     // Όσο η δευτερεύουσα μεταβλητή δεν έχει αλλάξει
     while (!threadShouldWake) {
           // θα περιμένουμε
           var.wait(lock);
     }

     // Επαναφέρουμε την δευτερεύουσα μεταβλητή αφού
     // το νήμα ξυπνήσει πραγματικά
     threadShouldWake = false;
}

void threadFunc2() {
     // Κλειδώνουμε τον Mutex πριν κάνουμε αλλαγή
     std::unique_lock<std::mutex> lock(mutex);
     // θέτουμε την δευτερεύουσα μεταβλητή
     // για να ξέρει ότι το νήμα όντως πρέπει
     // να ξυπνήσει
     threadShouldWake = true;
     var.notify_all();
}
```

6.4 Τεχνικές Βελτιστοποίησης (Optimization)

C++

Η C++ είναι μια μεταγλωτιζόμενη γλώσσα προγραμματισμού, που σημαίνει ότι ο κώδικας μπορεί να αντιστοιχηθεί σε τελική γλώσσα μηχανής 1:1. Αυτό επιτρέπει στον προγραμματιστή να εφαρμόσει κάποιες τεχνικές για να βελτιστοποιήσει τον κώδικα ως προς την ταχύτητα και την κατανάλωση πόρων που χρησιμοποιεί. Ο Compiler αναλαμβάνει να κάνει τα Optimizations επί το πλείστων, αλλά μερικές φορές μπορούμε να βοηθήσουμε ακόμα περισσότερο με το να γράψουμε καλύτερο κώδικα μέσω κάποιων τεχνικών.

Οι περισσότερες τεχνικές που εφαρμόστηκαν στην μηχανή έχουν τις ρίζες τους από τα εγχειρίδια του Agner Fog πάνω στην βελτιστοποίηση στην C++. http://agner.org/optimize/

Lua

Η Lua σε αντίθεση με την C++ είναι Interpreted γλώσσα. Ο κώδικας που γράφεται εδώ μπορεί να έχει σημαντικές διαφορές από την μία μορφή στην άλλη. Εδώ ο Interpreter δεν μπορεί να κάνει τα Optimizations που θα έκανε ο Compiler γιατί ο κώδικας δεν είναι σταθερός, από την άποψη ότι τον μεταφράζει κατά την εκτέλεση του προγράμματος, και έτσι δεν ξέρει τι να περιμένει. Για αυτό και δεν μπορεί να τον αλλάξει για να τον κάνει πιο γρήγορο. Πχ.

Πίνακας 6.4.1: Παράδειγμα Optimization σε C++

```
static inline constexpr int func() {
     return 5;
}

int main() {

     for (int i = 0; i < 10000; ++i) {
           std::cout << func() << std::endl;
     }

     return 0;
}
```

Στον παραπάνω κώδικα ο Optimizer στην `for` θα αντικαταστήσει την κλήση της `func()` και θα βάλει στην θέση της το 5. Αυτό θα το κάνει γιατί ο Compiler κατάλαβε κατά την μεταγλώττιση την απλότητα της συνάρτησης και την έκανε inline. Το Inlining βοηθάει να εξαλειφτεί η καθυστέρηση (Overhead) από την κλήση της συνάρτησης με το να αντιγράψει τον κώδικα εκεί που καλείται. Σε βρόγχους και σε κρίσιμα σημεία του προγράμματος βοηθάει πολύ στην ταχύτητα γιατί εξαλείφονται οι εντολές πασαρίσματος των ορισμάτων και επιστροφής της τιμής. Το μειονέκτημα εδώ είναι ότι αυξάνεται το μέγεθος του δυαδικού αρχείου.

Στον παραπάνω βρόγχο, το κέρδος στην ταχύτητα ίσως γίνει αισθητό για μεγάλες τιμές των επαναλήψεων. Ο αντίστοιχος κώδικας στην Lua δεν θα έκανε Inline την συνάρτηση, αντίθετα θα την καλούσε συνέχεια παρόλο που το αποτέλεσμα της δεν θα άλλαζε ποτέ. Αυτό γιατί τα σύμβολα της Lua μπορούν αλλάξουν το τι αναπαριστάνουν ανά πάσα στιγμή. Για αυτό βαριές κλήσεις συναρτήσεων που δεν θα αλλάξουν το αποτέλεσμα όσες φόρες και να κληθεί, μπορούμε να τις αποθηκεύσουμε σε μια προσωρινή μεταβλητή και να χρησιμοποιήσουμε αυτή. Πχ το προηγούμενο πρόγραμμα θα μπορούσαμε να το βελτιστοποιήσουμε στην Lua όπως παρακάτω:

Πίνακας 6.4.2: Τεχνική Optimization σε Lua με Caching

```
local function func()
     return 5
end

local num = func()

for i=0,100000 do
     print(num)
end
```

Η παραπάνω τεχνική ισχύει για όλες τις Scripting γλώσσες που χρησιμοποιούν Interpreter. Πέρα από την παραπάνω, υπάρχουν και άλλες τεχνικές. Οι περισσότερες όμως εξαρτιούνται από το εκάστοτε πρόγραμμα που γράφεται. Πχ η παραπάνω θα επιβάρυνε το πρόγραμμα αντί να το ελαφρύνει αν η `func()` καλούνταν πολύ λίγες φορές (< 10). Αυτό γιατί η διαδικασία δημιουργίας και καταστροφής μιας μεταβλητής είναι πιο βαριά από την βαρύτητα 10 κλήσεων.

Η Lua λειτουργεί με Bytecode. Μετά την φάση στης λεκτικής και συντακτικής ανάλυσης, δημιουργεί δυαδικό κώδικα (Bytecode) που είναι στην ουσία αριθμοί που στον πυρήνα της Lua που αναπαριστούν εντολές σαν την X86 Assembly. Ο Bytecode γίνεται ο κώδικας που θα εκτελεστεί όποτε πρέπει. Η Lua επιτρέπει με το εργαλείο τής να μεταφραστούν τα Scripts της σε Bytecode από πριν την εκτέλεση του προγράμματος φορτώνοντας τα κατευθείαν. Έτσι παραλείπεται η φάση της λεκτικής και συντακτικής ανάλυσης και φορτώνεται πιο γρήγορα το Script.

Πέρα από την πρώιμη μεταγλώττιση, μπορεί να χρησιμοποιηθεί ο Just In Time Compiler της Lua (LuaJIT). Ο LuaJIT μεταγλωττίζει τον Bytecode σε γλώσσα μηχανής κατά την εκτέλεση. Αυτό του επιτρέπει να κάνει κάποια Optimizations (εφόσον του επιτραπεί από τον χρήστη) όσο το μεταγλωττίζει. Ανάλογα των περιπτώσεων, το πρόγραμμα στην Lua μπορεί να κερδίσει πολύ ταχύτητα. Το μειονέκτημα εδώ είναι ότι μπορεί να υπάρξουν καθυστερήσεις κατά την μεταγλώττιση. Οι καθυστερήσεις αυτές μπορούν να μειωθούν αν γίνει πρώιμη μεταγλώττιση. Παρόλο αυτά αν το πρόγραμμα κάνει πολλές μεταγλωττίσεις κατά την εκτέλεση, οι καθυστερήσεις ίσως να είναι ακόμα εμφανές.

Στην μηχανή ο απλός Interpreter είναι παραπάνω από γρήγορος και δεν χρειάζεται να χρησιμοποιούμε τον JIT. Αν παρόλο αυτά χρειαστεί, τότε είναι εύκολο να αντικαταστήσουμε την Lua5.1 με τον LuaJIT. Απλά κάνουμε τον πυρήνα Build από την αρχή

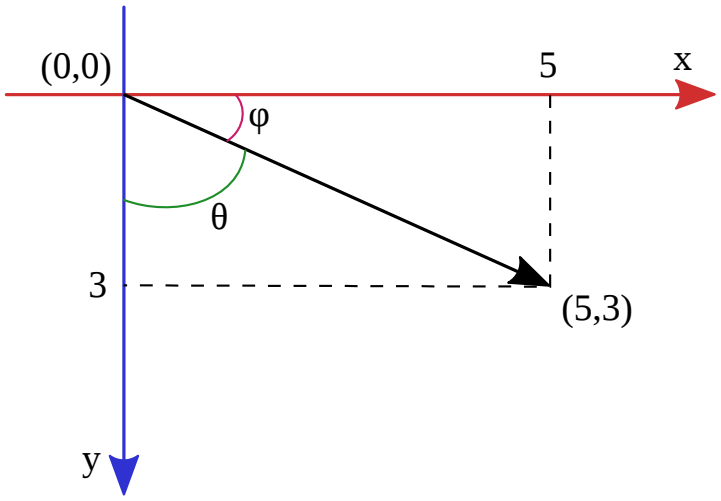
και να συνδέσουμε τον LuaJIT αντί για Lua5.1. Δεν χρειάζεται να πειραχτεί καθόλου κώδικας.

6.5 Διανυσματική άλγεβρα

Στις κατώτερες λειτουργίες της μηχανής γίνεται χρήση διανυσμάτων σε δισδιάστατο επίπεδο. Τα διανύσματα είναι στοιχεία της γραμμικής άλγεβρας που έχουν κατεύθυνση και μέτρο. Τα διανύσματα στο 2D επίπεδο αναπαριστάνονται από ένα ζεύγος αριθμών που τοποθετείται πάνω σε ένα σύστημα αξόνων x, y. Στην μηχανή για λόγω ότι η σύμβαση είναι έτσι, στο σύστημα αξόνων (y) τα αρνητικά είναι προς τα πάνω και τα θετικά προς τα κάτω. Για παράδειγμα, ένα διάνυσμα:

$$(x, y) = (5, 3) \quad (6.5.1)$$

Αναπαρίσταται στο επίπεδο της μηχανής:



Εικόνα 6.5.1: Αναπαράσταση του διανύσματος (6.1)

Το μέτρο του βρίσκεται παίρνοντας τον τύπο του πυθαγόρειου θεωρήματος:

$$|(x, y)| = \sqrt{x^2 + y^2} \quad (6.5.2)$$

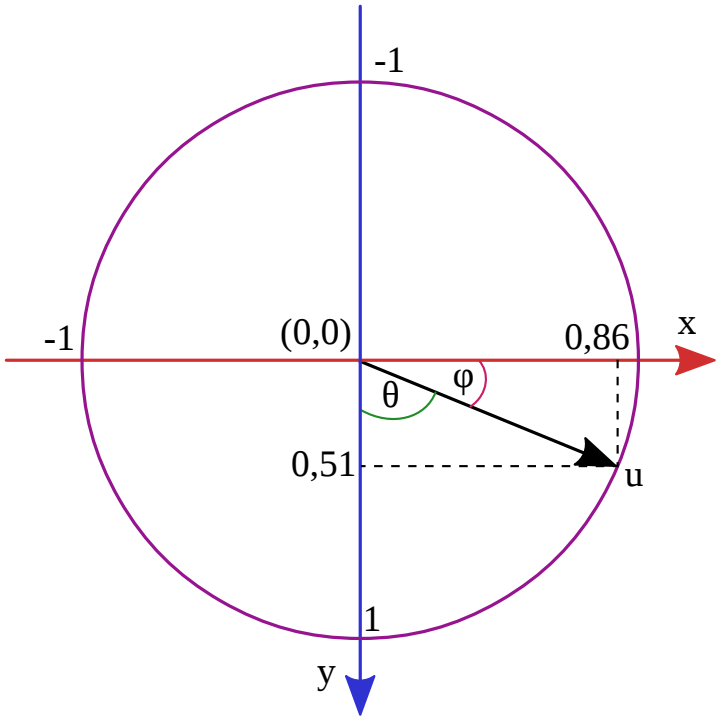
Πολλές φορές χρειάζεται να από το διάνυσμα να πάρουμε το μοναδιαίο του (που έχει μέτρο 1). Αυτό βοηθάει στον εύκολο προσδιορισμό των γωνιών κατεύθυνσης. Ο τύπος που μας το δίνει αυτό το διάνυσμα είναι:

$$u = (x_u, y_u) \quad (6.5.3)$$

όπου:

$$x_u = \frac{x}{|(x,y)|} = \cos(\varphi) = \sin(\theta)$$
$$y_u = \frac{y}{|(x,y)|} = \cos(\theta) = \sin(\varphi)$$

Το παραγόμενο αυτό διάνυσμα είναι ένα σημείο στον μοναδιαίο κύκλο, όπου τα (x, y) μπορούν να εκφράσουν τις γωνίες φ, θ μέσω των τριγωνομετρικών συναρτήσεων.



Εικόνα 6.5.2: Το μοναδιαίο διάνυσμα του (6.5.1)

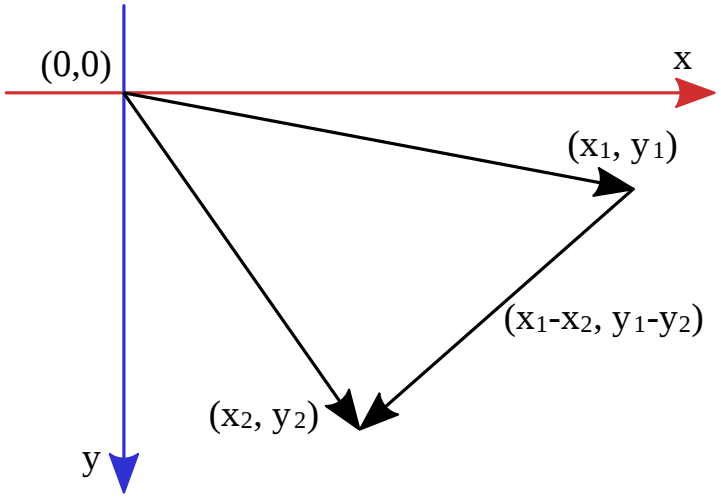
Οι πράξεις στα διανύσματα γίνονται στα επιμέρους (x,y) πχ:

$$(x_1, y_1)(x_2, y_2) = (x_1 x_2, y_1 y_2)$$
$$(x, y)m = (mx, my)$$
$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$
$$(x, y) + m = (x + m, y + m)$$

Τα διανύσματα χρησιμοποιούνται και ως απλά σημεία στο επίπεδο. Για αυτό πολλές φορές χρειάζεται να βρούμε την απόσταση μεταξύ αυτών των σημείων. Για να βρεθεί αυτό, πρέπει να αφαιρέσουμε τα διανύσματα και να βρούμε το μέτρο του διανύσματος που προέκυψε από την διαφορά. Ο παρακάτω τύπος δίνει την απόσταση 2 σημείων στο επίπεδο.

$$\left|\left((x_{1,}y_1),(x_{2,}y_2)\right)\right|=\sqrt{\left(x_1-x_2\right)^2+\left(y_1-y_2\right)^2}\,(6.5.4)$$

Η διαφορά δύο διανυσμάτων δεν βοηθάει μόνο στον υπολογισμό της απόστασης. Στην μηχανή βοηθάει και στον υπολογισμό της κατεύθυνσης ενός υπάρχοντος σημείου προς ένα άλλο. Για παράδειγμα, έστω ότι θέλουμε ένα σημείο $(x_{1,}y_1)$ να κατευθυνθεί στο σημείο $(x_{2,}y_2)$. Η αφαίρεση αυτών θα μας δώσει:



Εικόνα 6.5.3: Αναπαράσταση Διαφοράς διανυσμάτων

Από διάνυσμα που προέκυψε από την διαφορά, αν μετατραπεί σε μοναδιαίο τότε είναι πολύ χρήσιμο για υπολογιστεί η κατεύθυνση και η επόμενη θέση του σημείου σε κάθε Frame της μηχανής. Περισσότερα στο 3ο μέρος.

6.6 Μηχανές πεπερασμένων καταστάσεων

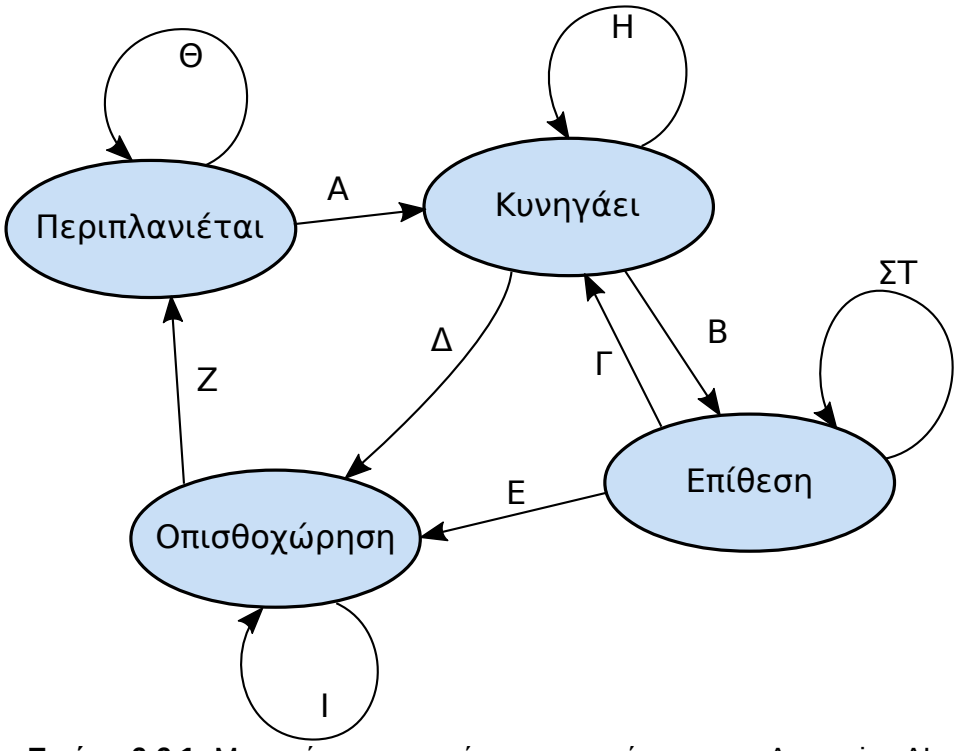
Σε κάποια μεγάλα προγράμματα, ανάλογα τι δεδομένα εισάγονται κάθε στιγμή, κάποιες οντότητες χρειάζεται να λειτουργήσουν διαφορετικά και να αλλάζουν κατάσταση. Σε αυτό χρησιμεύουν οι μηχανές πεπερασμένων καταστάσεων (Finite State Machines). Για να καταλάβουμε πιο καλά τι είναι αυτά, θα δώσω ένα παράδειγμα από την μηχανή.

Έστω ότι έχουμε έναν εχθρό. Αυτόν τον εχθρό θέλουμε να τον ρυθμίζουμε έτσι ώστε:

- Όταν δεν κάνει τίποτα, να περιπλανιέται γύρω από το σημείο που εμφανίστηκε

- Όταν ο παίχτης πλησιάσει σε κάποια απόσταση και ο εχθρός βρίσκεται σε κατάσταση περιπλάνησης, να αρχίσει να τον κυνηγάει (Α), αλλιώς να συνεχίσει την περιπλάνηση
- Όταν πλησιάσει τον παίχτη σε ικανοποιητική απόσταση, να μπει σε κατάσταση επίθεσης και να επιτεθεί (Β), αλλιώς να συνεχίσει το κυνήγι (Η)
- Αφού του επιτεθεί, να ελέγχει αν υπάρχει πάλι η ικανοποιητική απόσταση. Αν ναι να επιτεθεί πάλι (ΣΤ), αλλιώς να τον κυνηγήσει (Γ)
- Αν όσο κυνηγάει ή τελειώσει την επίθεση ο εχθρός έχει απομακρυνθεί πολύ από την αρχική του θέση, να αρχίσει να οπισθοχωρήσει (Δ) (Ε)
- Όσο οπισθοχωρεί, αν έχει φτάσει στο σημείο που ξεκίνησε να μπει σε κατάσταση περιπλάνησης (Ζ), αλλιώς να συνεχίσει να οπισθοχωρεί (Θ)

Αυτή η συμπεριφορά χαρακτηρίζεται από καταστάσεις που εναλλάσσει ο εχθρός. Ανάλογα την κατάσταση που βρίσκεται κάνει άλλες λειτουργίες. Λάβετε υπόψιν ότι από οποιαδήποτε κατάσταση δεν μπορεί να πάει σε οποιαδήποτε άλλη. Αυτό ορίζεται από τους κανόνες που θέτουμε. Αυτούς τους κανόνες και τις εναλλαγές στις καταστάσεις τις φτιάχνουμε αρχικά σε ένα διάγραμμα για να μας είναι πιο εύκολο στην υλοποίηση. Ένα διάγραμμα για την παραπάνω μηχανή πεπερασμένων καταστάσεων είναι το ακόλουθο.



Εικόνα 6.6.1: Μηχανή πεπερασμένων καταστάσεων του Agressive AI

Σε κώδικα C++ μια πολύ απλή υλοποίηση μπορεί να γίνει με ένα Enumeration Flag και μια `switch` που θα έχει κώδικα για κάθε κατάσταση ξεχωριστά. Για παράδειγμα:

Πίνακας 6.6.1: Παράδειγμα σε C++ για Μηχανή Πεπερασμένων Καταστάσεων

```
enum class State {
    Wandering, Following, Attacking, FallingBack
};

void func(State state, float distance, ...) {
    switch (state) {
    case State::Wandering:
          ...
          if (distance < 15.0f) {
                state = State::Following;
          }
          break;
    case State::Following:
          ...
          break;
    case State::Attacking:
          ...
          break;
    case State::FallingBack:
          ...
          break;
    default:
          ...
          break;
    }
}
```

Στην μηχανή, οι μηχανές αυτές χρησιμοποιούνται κυρίως για την τεχνητή νοημοσύνη, τις καταστάσεις των όντων και καταστάσεις αποστολών.

7 ΥΠΟΣΤΗΡΙΖΟΜΕΝΑ ΕΡΓΑΛΕΙΑ

Η μηχανή έχει φτιαχτεί να χρησιμοποιεί δομές δεδομένων που είναι συμβατές με κάποια εξωτερικά εργαλεία έτσι ώστε να γίνεται πιο εύκολη η δημιουργία των παιχνιδιών. Αυτά είναι τα ακόλουθα:

- **Tiled Map Editor**

Πρόκειται για ένα εργαλείο που κάνει την δημιουργία των χαρτών πολύ εύκολη. Επιτρέπει την δημιουργία Tilesheets από εικόνες και εξαγωγή των δεδομένων αυτών σε XML αρχεία. Οι χάρτες φτιάχνονται σε φάση WYSIWYG (What You See Is What You Get). Δημιουργείς τον χάρτη στο πρόγραμμα και μετά όταν φορτωθεί από την μηχανή θα φαίνεται ακριβώς όπως τον φτιάξαμε μαζί με ότι άλλο προσθέσαμε (Εχθροί, αντικείμενα κτλ.). Τα δεδομένα των χαρτών εξάγονται και αυτά σε XML και τα δεδομένα των στρώσεων (Layers) συμπιέζονται σε μορφή zlib κωδικοποιημένα σε μορφή Base64. Το εργαλείο αυτό επιλέχτηκε γιατί μας παρείχε όλες αυτές τις ευκολίες, κάτι που χωρίς αυτό η δημιουργία των χαρτών θα ήταν πολύ χρονοβόρα. Περισσότερα στο επόμενο μέρος.

- **Dark Function Editor**

Το εργαλείο αυτό βοηθάει στο να φτιαχτούν εύκολα τα Spritesheets και τα Animations των αντικειμένων-όντων του παιχνιδιού. Τα Spritesheets φτιάχνονται από υπάρχουσες εικόνες όπου ορίζουμε τα τετράγωνα που περιβάλουν κάθε καρέ της εικόνας. Αφού γίνει αυτό, τότε από αυτές τις επιμέρους εικόνες φτιάχνουμε τα Animations. Τα δεδομένα και των Spritesheets και Animations εξάγονται σε XML μορφή. Αυτό το εργαλείο επιλέχτηκε γιατί μπορούμε εύκολα και γρήγορα να παράγουμε τα Animations με δεδομένα ακριβώς που ζητάμε μέσα στην μηχανή. Ήταν το μόνο εργαλείο που μπορούσε να ταιριάξει τις απαιτήσεις της μηχανής. Περισσότερα στο επόμενο μέρος.

- **CEED Editor**

Πρόκειται για το εργαλείο που παρέχεται από την βιβλιοθήκη CEGUI. Βοηθάει στην δημιουργία παραθύρων WYSIWYG, αλλά ως εκεί προς το παρών. Τα πιο δύσκολα κομμάτια του CEGUI (όπως τα Look'n'Feel) θα πρέπει να γίνουν με μηχανικό τρόπο. Το εργαλείο αυτό είναι κυρίως βοηθητικό για την CEGUI και είναι η μόνη επιλογή αν θέλουμε να επιταχύνουμε την δημιουργία του GUI.

ΜΕΡΟΣ ΔΕΥΤΕΡΟ: ΠΕΡΙΓΡΑΦΗ ΤΗΣ ΔΙΑΔΙΚΑΣΙΑΣ ΔΗΜΙΟΥΡΓΙΑΣ ARPG ΠΑΙΓΝΙΩΝ

1 ΑΠΑΡΙΘΜΗΣΗ ΒΗΜΑΤΩΝ – ΔΟΜΩΝ ΔΕΔΟΜΕΝΩΝ

Σε αυτό το μέρος θα ασχοληθούμε με το πως μπορούμε να φτιάξουμε τα κομμάτια ενός παιχνιδιού για την μηχανή. Τα κομμάτια αυτά είναι στην ουσία δεδομένα, αναγνώσιμα από τον άνθρωπο, που ενώνονται από την μηχανή και παράγουν το αποτέλεσμα που περιγράφουν. Συνήθως ακολουθείται μια διαδικασία για την δημιουργία του παιχνιδιού. Η πιο απλή και βέλτιστη είναι η παρακάτω:

1. Δημιουργία του βασικού γραφικού περιβάλλοντος (GUI) και σύνδεση του με την μηχανή.
2. Δημιουργία του παίχτη (ή πρότυπο παίχτη)
3. Δημιουργία χαρτών
4. Δημιουργία έτερων όντων
5. Δημιουργία ικανοτήτων
6. Δημιουργία έτερου περιεχομένου (Αποστολές, συμβάντα κόσμου κτλ.)

Εννοείται ότι δεν χρειάζεται οπωσδήποτε να ακολουθηθεί η σειρά που περιγράφεται για να φτιαχτεί ένα παιχνίδι. Απλά έτσι γίνεται πιο απλή και βλέπουμε τα αποτελέσματα αμέσως μόλις φτιάξουμε κάτι.

Τα κομμάτια του παιχνιδιού περιγράφονται από δομές δεδομένων που αντλούν τα δεδομένα από εξωτερικά αρχεία. Αυτές οι δομές είναι οι:

- Αρχεία που ορίζουν την Εμφάνιση της Διεπαφής Χρήστη (GUI)
- Αρχεία που ορίζουν Κλάσεις Ζωής (Lifeform Classes)
- Αρχεία που ορίζουν Κλάσεις Animation (Animation Classes)
- Αρχεία που ορίζουν Ομάδες από Animations (Animation Packs)
- Αρχεία που ορίζουν Ομάδες από Sprites (Spritesheets)
- Αρχεία που ορίζουν Σχέσεις μεταξύ Κλάσεων Ζωής (Factions)
- Αρχεία που ορίζουν Χάρτες (Maps)
- Αρχεία που ορίζουν Ομάδες από Tiles (Tilesets)
- Δομές που ορίζουν Κλάσεις Ικανοτήτων (Ability Classes)
- Δομές που ορίζουν Κλάσεις Βλημάτων (Projectile Classes)
- κ.α.

Όλα αυτά αν συνδυαστούν, μπορούν να φτιάξουν ένα παιχνίδι ARPG από την αρχή ως το τέλος.

2 ΔΙΕΠΑΦΗ ΧΡΗΣΤΗ ΜΕ ΤΟ CEGUI

2.1 Το εργαλείο CEED

Η βιβλιοθήκη CEGUI παρέχει ένα εργαλείο για να δημιουργηθούν GUI παράθυρα WYSIWYG. Παρέχει δυνατότητα να φτιάξει ολόκληρο το GUI Project μαζί με τους προκαθορισμένους καταλόγους. Πέρα από τα παράθυρα, μπορεί κάποιος να φτιάξει και τα Imagesets του GUI, εισάγοντας μια εικόνα και σπάζοντας την σε κομμάτια.

Το πώς φτιάχνονται τα GUI όπως προαναφέραμε στο προηγούμενο κομμάτι, δεν είναι αρμοδιότητα της μηχανής. Αυτό γιατί οι δομές δεδομένων που γράφονται για το GUI δεν τις “αγγίζει” η μηχανή καθόλου σε σχέση με τα άλλα εργαλεία όπου οι δομές καθορίζουν την λειτουργία της. Για την μηχανή το CEGUI απλά έχει κάτι να σχεδιάσει κάθε φορά που πρέπει να σχεδιαστεί ένα Frame. Για λεπτομέρειες για την παραγωγή GUI ανατρέξτε στο επίσημο site του CEGUI http://cegui.org.uk/.

Για το μέρος αυτό, θα θεωρήσουμε ότι έχουμε φτιάξει το GUI του Demo. Τα αρχεία του GUI βρίσκονται στο “./Demo/EftihiaGUI”.

2.2 Χαρτογράφηση σε Lua Script

Το GUI ελέγχεται αποκλειστικά από Lua Scripts. Αυτό σημαίνει ότι η μηχανή δεν θα φορτώσει ούτε θα εμφανίσει ποτέ τίποτα από μόνη της. Αυτά πρέπει να τα κάνει ο προγραμματιστής του παιχνιδιού με τα Scripts που θα φτιάξει.

Έτσι αρχικά πρέπει να φορτώσουμε το GUI στην CEGUI και ύστερα να αποθηκεύσουμε αναφορές από τα επιμέρους στοιχεία σε μεταβλητές της Lua για να μπορούμε εύκολα και γρήγορα να έχουμε πρόσβαση σε αυτά. Το πιο συνετό είναι να φτιάξουμε ένα Lua Module (Lua Script που επιστρέφει έναν πίνακα με δεδομένα) για κάθε “οθόνη” που θα έχει το παιχνίδι (πχ ένα για το Κύριο Μενού, για την Οθόνη Φορτώματος κτλ.). Ας δούμε πως χαρτογραφούμε το κύριο μενού.

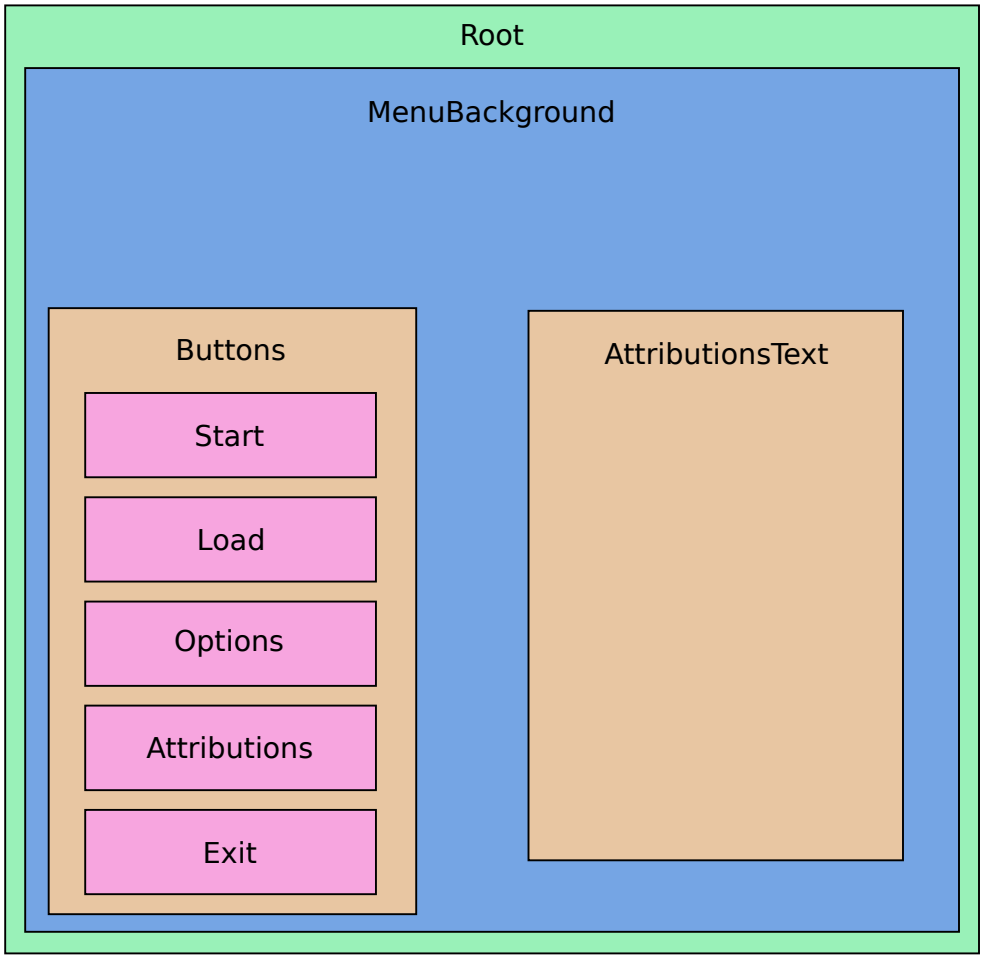
Πίνακας 2.2.1: Κώδικας του αρχείου "Demo/GUI/init.lua"

```
-- Δημιουργούμε ένα κενό πίνακα που θα επιστρέψουμε
local GUI = {}

-- Φορτώνουμε σε προσωρινές μεταβλητές τα υποσυστήματα του
-- CEGUI που θα χρησιμοποιήσουμε εδώ
local guiSystem = CEGUI.System:getSingleton()
local winMgr = CEGUI.WindowManager:getSingleton()

-- Λέμε στο CEGUI σε ποια Resource Groups θα βρεί ότι χρειάζεται
```

```
CEGUI.Scheme:setDefaultResourceGroup("schemes")
CEGUI.ImageManager:setImagesetDefaultResourceGroup("imagesets")
CEGUI.Font:setDefaultResourceGroup("fonts")
CEGUI.WidgetLookManager:setDefaultResourceGroup("looknfeel")
CEGUI.WindowManager:setDefaultResourceGroup("layouts")

-- Φορτώνουμε το GUI που φτιάξαμε
CEGUI.SchemeManager:getSingleton():createFromFile("Eftihia.scheme");

-- Φτιάχνουμε το Κύριο (Αδιαφανές) παράθυρο
GUI.Root = winMgr:createWindow("DefaultWindow","root")
GUI.WindowManager = winMgr
-- Ορίζουμε το Κύριο Παράθυρο και το ToolTip μας
guiSystem:getDefaultGUIContext():setRootWindow(GUI.Root)
guiSystem:getDefaultGUIContext():setDefaultTooltipType("Eftihia/Tooltip")
-- Επιστρέφουμε τα δεδομένα μας
return GUI
```

Στον παραπάνω κώδικα, φτιάξαμε το αρχείο φορτώματος του “GUI” κυριολεκτικά και μεταφορικά. Στην Lua, τα Modules λειτουργούν σαν τα πακέτα της Java. Για παράδειγμα, όταν θέλουμε να φορτώσουμε το αρχείο “Dir1/Dir2/Scr.lua”, στην Lua θα το φορτώσουμε με το `local scr = require('Dir1.Dir2.Scr')`, δηλαδή αντί για “/” χρησιμοποιούμε τελείες. Μερικές φορές χρειάζεται να φορτώσουμε το `local scr = require('Dir1.Dir2')` που είναι κατάλογος. Αυτό γίνεται με το να προσθέσουμε στον κατάλογο Dir2 ένα αρχείο Lua "init.lua" όπως στο παράδειγμα. Εκμεταλλευόμενοι αυτό, το χρησιμοποιούμε για να αρχικοποιήσουμε το CEGUI και να φορτώσουμε τα αρχεία του GUI που φτιάξαμε. Στο Module επιστρέφουμε τον πίνακα `local GUI = {}` που του ορίζουμε τα πεδία: `GUI.Root` και `GUI.WindowManager`. Στο μέλλον όταν θα κάνουμε `local GUI = require('Demo.GUI')`, θα ξέρουμε ότι ο πίνακας GUI θα έχει τα εν λόγω πεδία με τις τιμές που του βάλαμε στο Script. Αξίζει να σημειωθεί ότι με το φόρτωμα με την `require('...')`, το Script θα εκτελεστεί μόνο μια φορά. Μετά από αυτό, κάθε φορά που θα ζητείται το εν λόγω Script θα δίνεται από την Cache. Αυτό βολεύει για να μην γίνονται συνέχεια αρχικοποιήσεις και να έχουμε τα ίδια δεδομένα πάντα (σε περιπτώσεις που τα πεδία έχουν Pointers όπως στο παράδειγμα).

Στην συνέχεια, θα πρέπει να υλοποιήσουμε το Module του κύριου μενού, όπου θα φορτώνει και θα χαρτογραφεί το αρχείο "MainMenu.layout". Όπως και κάθε Layout (Οθόνες), τα παράθυρα ταξινομούνται σε δενδροειδή ιεραρχία, με πρώτο παράθυρο το Root. Το "MainMenu.layout" το έχουμε φτιάξει σε ιεραρχία όπως παρακάτω:



Εικόνα 2.2.1: Αναπαράσταση Ιεραρχίας του "MainMenu.layout"

Το παράθυρο Root είναι κοινό για όλα τα Layouts που θα φορτώσουμε. Στην ουσία θα είναι όλα παιδιά του. Από το Root εμείς θα αρχίσουμε να χαρτογραφούμε το MainMenu σε πίνακα Lua.

Πίνακας 2.2.2: Μέρος Α: του αρχείου "Demo/GUI/MainMenu.lua"

```
-- Φορτώνουμε σε προσώρινές μεταβλητές ότι χρειαζόμαστε
-- για να κάνουμε πιο γρήγορο το πρόγραμμα
local GUI = require('Demo.GUI')
local Root = GUI.Root
local toBtn = CEGUI.toPushButton
-- Φορτώνουμε το "MainMenu.layout" και το προσθέτουμε στο Root σαν παιδί
Root:addChild(GUI.WindowManager:loadLayoutFromFile("MainMenu.layout"))

-- Αφου το φορτώσαμε, φτιάχνουμε το Χάρτη-Πίνακα του Layout
local MainMenu = {
     this = Root:getChild("MenuBackground"),
     Buttons = {
           this = Root:getChild("MenuBackground/Buttons"),
           Start = toBtn(Root:getChild("MenuBackground/Buttons/Start")),
           Options =
```

```
toBtn(Root:getChild("MenuBackground/Buttons/Options")),
            Load = toBtn(Root:getChild("MenuBackground/Buttons/Load")),
            Attributions =
toBtn(Root:getChild("MenuBackground/Buttons/Attributions")),
            Exit = toBtn(Root:getChild("MenuBackground/Buttons/Exit"))
        },
        AttriText = Root:getChild("MenuBackground/AttributionsText")
}
```

Κάθε παράθυρο στην CEGUI μπορεί να περιέχει άλλα παράθυρα σαν παιδιά. Όταν γίνεται αυτό, βάζουμε ένα πεδίο `this` που περιέχει αυτό το παράθυρο, όπως και στο `MainMenu`, και `Buttons`. Κάθε παράθυρο που βάζουμε σε πεδία το ζητάμε από το Root, από όπου μπορούμε να ζητήσουμε οποιοδήποτε παράθυρο του έχει προστεθεί σαν παιδί. Όπως βλέπουμε αυτό γίνεται σαν διαδρομές αρχείων. Πχ η έκφραση: `Root:getChild("MenuBackground/Buttons/Start")`, σημαίνει ότι θα πάρει το παράθυρο “Start” που βρίσκεται μέσα στο παράθυρο “Buttons”, το οποίο βρίσκεται στο παράθυρο “MenuBackground”, το οποίο βρίσκεται στο παράθυρο “Root”. Αυτό θα ακολουθήσει αυτή την διαδρομή για να βρει αυτό το παράθυρο. Αν το παράθυρο αυτό ζητείται έτσι πολλές φορές όσο τρέχει το πρόγραμμα δεν είναι καθόλου αποδοτικό. Και αυτό γιατί γίνονται πολλές αναζητήσεις σε δυαδικά δένδρα(Binary Trees) μέσα στην CEGUI. Για αυτό και τα χαρτογραφούμε σε πίνακες της Lua που είναι πιο γρήγορα και δεν χρειάζεται να μεταβαίνουμε από την Lua στην μηχανή πολλές φορές. Για παράδειγμα, όταν θα έπρεπε να χρησιμοποιήσουμε το “Start” οπουδήποτε μέσα στο πρόγραμμα χωρίς την χαρτογράφηση θα έπρεπε να εκτελέσουμε την:

Πίνακας 2.2.3: Ανάκτηση του Μη-Χαρτογραφημένου Παραθύρου "Start"

```
local Start =
CEGUI.toPushButton(require('Demo.GUI').Root:getChild("MenuBackground/But
tons/Start"))
```

Η παραπάνω κλήση εάν καλείται πολλές φορές μέσα στο δευτερόλεπτο, θα προκαλέσει καθυστερήσεις χωρίς λόγο. Αυτό γιατί η παραπάνω κλήση περιέχει: 5 Lua Indexings (Ότι καθολικό “.” και “:”, θεωρείται αναζήτηση σε πίνακα Lua), 2 εξωτερικές κλήσεις βιβλιοθήκης (`toPushButton`, και `getChild`), μαζί με τις μεταβιβάσεις τους στην C++ και πίσω και 3 αναζητήσεις σε δυαδικά δένδρα της CEGUI (`"MenuBackground/Buttons/Start"`). Αν το παράθυρο “Start” δεν διαγραφτεί ή αντικατασταθεί από κάποιο άλλο κατά την διάρκεια το προγράμματος, τότε η παραπάνω κλήση θα επιστρέφει πάντα τον ίδιο Pointer! Στο πρόγραμμά μας όμως ξέρουμε καλά

ότι τα παράθυρά μας θα παραμένουν καθ' όλη την διάρκεια του προγράμματος (με ελάχιστες εξαιρέσεις) και δεν πρόκειται να αλλάξει η δομή του GUI. Αυτό σημαίνει ότι η παραπάνω κλήση είναι χάσιμο χρόνου (και γραψίματος). Για αυτό και αποθηκεύουμε αυτόν τον Pointer σε μεταβλητή της Lua από όπου θα βρίσκουμε το παράθυρό μας εκεί. Με την χαρτογράφηση του MainMenu, το παράθυρο “Start” μπορούμε να το πάρουμε ως εξής:

Πίνακας 2.2.4: Ανάκτηση του Χαρτογραφημένου Παραθύρου "Start"

```
local Start = require('Demo.GUI.MainMenu').Buttons.Start
```

Όπως βλέπουμε, η παραπάνω έκφραση είναι πιο απλή, κατανοητή, αλλά και πιο γρήγορη από την προηγούμενη. Αυτό γιατί το αποτέλεσμα της προηγούμενης έχει αποθηκευτεί στο `Buttons.Start` και απλά αντιγράφεται στο `local Start`.

Κάτι που δεν εξηγήσαμε, είναι το **`CEGUI`**`.toPushButton`. Στην CEGUI, τα πάντα είναι παράθυρα (Windows), από τα παράθυρα έως και τα κουμπιά και οι μπάρες για την κύλιση. Κάθε τύπος εξειδικεύεται σε υπό-κλάση του Window. Έτσι όταν ζητάμε ένα παράθυρο από την CEGUI, αυτή δεν ξέρει πώς το φτιάξαμε και έτσι μας το επιστρέφει σαν γενικό παράθυρο ακόμα και αν αυτό είναι ένα κουμπί. Εμείς απλά το μετατρέπουμε σε εξειδικευμένο για να αποκτήσουμε πρόσβαση στις επιπλέον λειτουργίες του. Στο παραπάνω, η μετατροπή σε κουμπί γίνεται με την κλήση της **`CEGUI`**`.toPushButton`.

2.3 Καθορισμός συμπεριφορών

Μέχρι στιγμής είδαμε πως να κάνουμε εύκολη την πρόσβαση στα παράθυρα του GUI. Τώρα όμως πρέπει να δούμε πώς θα τα ρυθμίσουμε έτσι ώστε να αντιδρούν όταν δεχτούν κάποιο συμβάν, πχ όταν πατήσουμε ένα κουμπί να εμφανίσει κάτι. Αυτά γίνονται με το να καταχωρίσουμε συναρτήσεις της Lua στην CEGUI. Για παράδειγμα:

Πίνακας 2.3.1: Μέρος Β: του αρχείου "Demo/GUI/MainMenu.lua"

```
-- Συνάρτηση που πρέπει να καλεστεί όταν πατηθεί το Start
function MainMenu.Buttons.Start_onMouseClick(event)
    -- Φόρτωσε το Layout για την επιλογή χαρακτήρων
    local char = require('Demo.GUI.CharSelect')
    -- Εμφάνισε την οθόνη της επιλογής
    char.this:show()
    char.this:activate()
```

```
    -- Κρύψε τα κουμπιά του Κύριου μενου
    MainMenu.Buttons.this:hide()
end
...
local Buttons = MainMenu.Buttons
-- Κάνε το Start όταν του γίνει "MouseClick" να καλέσει αυτή την
συνάρτηση
Buttons.Start:subscribeEvent("MouseClick",Buttons.Start_onMouseClick)
```

Κάθε παράθυρο μπορεί να ρυθμιστεί για οποιοδήποτε συμβάν μπορεί να λάβει. Για περαιτέρω πληροφορίες για τα συμβάντα που μπορούν να λάβουν τα παράθυρα, συμβουλευτείτε την τεκμηρίωση (Documentation) της CEGUI.

3 ΔΗΜΙΟΥΡΓΙΑ ΌΝΤΩΝ

3.1 Τα XML αρχεία LifeformClass

Τα αρχεία LifeformClass είναι XML αρχεία που περιγράφουν μια κατηγορία ζωντανής οντότητας. Στην ουσία είναι κλάσεις-καλούπια που μπορούν να φτιάξουν ζωντανά αντικείμενα που έχουν τις ιδιότητες που περιγράφει αυτό το αρχείο. Για παράδειγμα, μια LifeformClass μπορεί να ορίζει ένα Npc. Μια άλλη να ορίζει μια κατηγορία εχθρού, πχ “Φίδι Κολοβό Επιπέδου: 68-72”. Ακόμα και ο παίχτης ορίζεται από ένα LifeformClass.

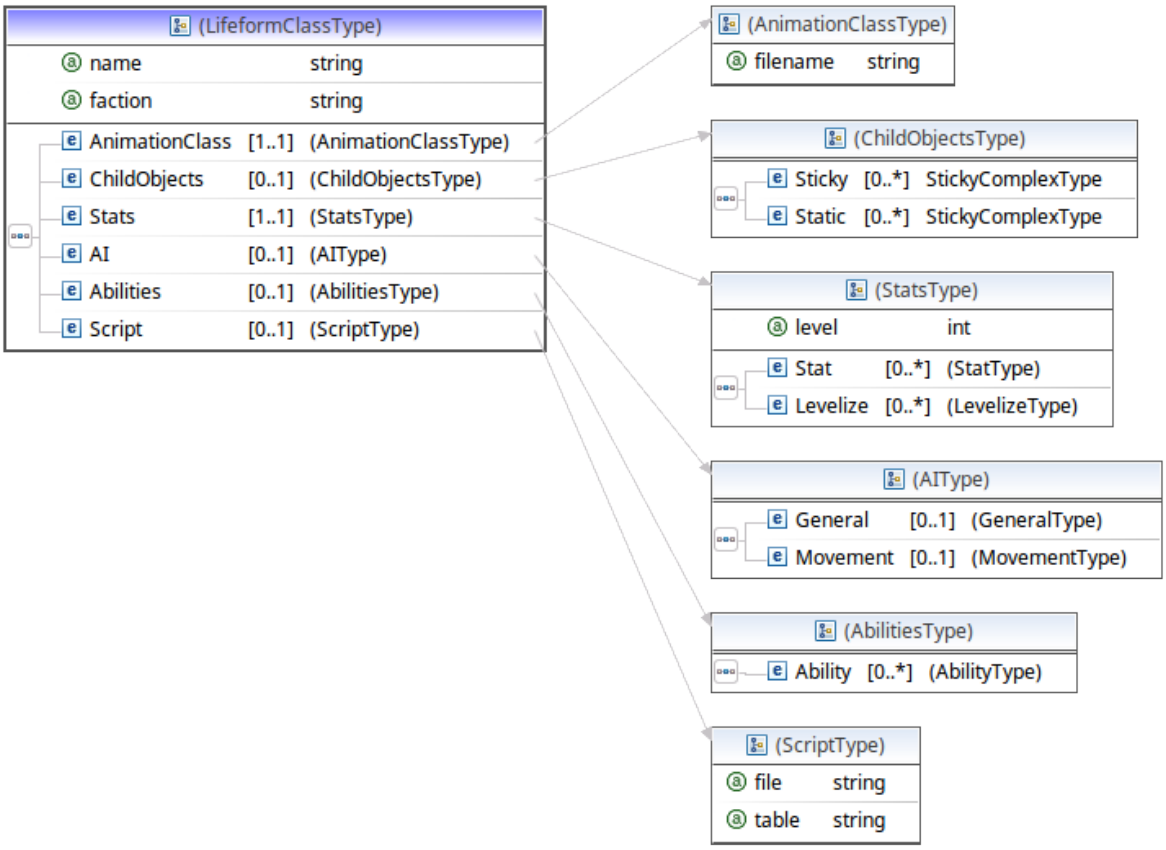
Τα Lifeform είναι μοναδικές οντότητες σε έναν χάρτη. Κάθε Lifeform έχει μια αναφορά σε ένα LifeformClass που του ορίζει τα πάντα. Τα LifeformClass είναι κοινόχρηστα. Δηλαδή μπορούμε να φτιάξουμε πολλά “Φίδια Κολοβά” στον ίδιο χάρτη και να μοιράζονται το ίδιο LifeformClass. Παρόλο που το μοιράζονται, τα φίδια είναι ανεξάρτητα και κινούνται ελεύθερα μέσα στους κανόνες που ορίζει αυτό το LifeformClass.

Οι πληροφορίες που παρέχει αυτή η δομή ποικίλουν και είναι οι εξής:

- Όνομα που μοιράζεται
- Ομάδα (Faction)
- Κατηγορία Animations
- Αντικείμενα-Παιδιά
- Στατιστικά-Δυνάμεις
- Τεχνητή Νοημοσύνη
- Τεχνικές-Επιθέσεις
- Επιπρόσθετες πληροφορίες που παρέχονται σε Lua Script

Οι παραπάνω πληροφορίες δίνονται με λεπτομέρειες μέσα στο XML αρχείο. Για να το καταλάβουμε καλύτερα πάμε να αναλύσουμε την δομή του XML αρχείου μιας LifeformClass μέσω του XSD Schema του.

- **LifeformClass**



Εικόνα 3.1.1: LifeformClass XSD Schema Μέρος Α

Το παραπάνω διάγραμμα δείχνει τι πρέπει να έχει το αρχικό Node του αρχείου. Όπως βλέπουμε, πρέπει να έχει όνομα “LifeformClass” και να έχει απαραίτητα Attributes τα “name” και “faction”. Πιο ειδικά:

- **name:** Έχει το προκαθορισμένο όνομα που θα έχει το Lifeform που θα παραχθεί από αυτό το LifeformClass. Όπως και με όλα τα πεδία, το όνομα αυτό μπορεί να αλλάξει αργότερα κατά προτίμηση.
- **faction:** Έχει το όνομα της ομάδας (Faction) που θα ανήκουν τα Lifeform που θα παραχθούν. Αυτό το όνομα θα πρέπει να ορίζεται στο αρχείο των Factions, αλλιώς θα θεωρηθεί ουδέτερο (Neutral).

Για παράδειγμα, ένα LifeformClass μπορεί να ξεκινάει:

Πίνακας 3.1.1: Παράδειγμα LifeformClass

```
<LifeformClass name="Φίδι" faction="Enemies">
    ...
</LifeformClass>
```

Τα περιεχόμενα του LifeformClass ορίζονται με αυστηρή σειρά όπως στο διάγραμμα 3.1.1. Πιο ειδικά:

Στοιχείο 1ο: AnimationClass (Απαραίτητο)

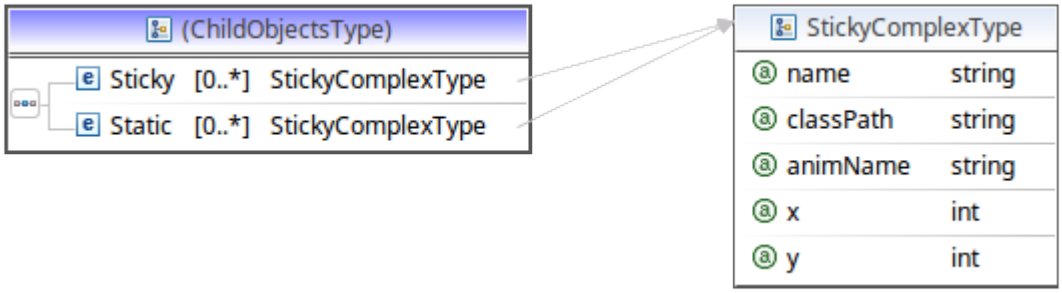
Αυτό το στοιχείο έχει ένα απαραίτητο Attribute με όνομα “filename”. Το “filename” πρέπει να έχει μια διαδρομή αρχείου σε ένα αρχείο AnimationClass, από όπου και το Lifeform θα πάρει μορφή. Περισσότερα για το AnimationClass παρακάτω.

Πίνακας 3.1.2: Παράδειγμα LifeformClass: AnimationClass

```
<AnimationClass filename="Demo/AnimationClasses/Snake.xml" />
```

Στοιχείο 2ο: ChildObjects (Προαιρετικό)

Αυτό το στοιχείο ορίζει τι αντικείμενα-παιδιά θα έχουν τα Lifeform που θα παραχθούν.



Εικόνα 3.1.2: Ορισμός του τύπου ChildObjects

Όπως βλέπουμε, η ομάδα “ChildObjects” μπορεί να έχει προαιρετικά πολλά στοιχεία “Sticky” και ύστερα προαιρετικά πολλά στοιχεία “Static”. Τα στοιχεία “Sticky” είναι αντικείμενα που ακολουθούν το γονιό-Lifeform. Τα στοιχεία “Static” δεν κινούνται αλλά μένουν εκεί που δημιουργήθηκαν μαζί με το γονιό. Ακόμα και αν ο γονιός κινηθεί το StaticChild θα μείνει ακίνητο.

Και τα δύο είδη ChildObject μοιράζονται την ίδια δομή στο LifeformClass:

- **name(απαραίτητο):** Είναι Attribute που θα ορίζει το όνομα για αυτό το ChildObject. Αυτό το όνομα μπορεί να χρησιμοποιείται για να έχουμε πρόσβα-

ση σε αυτό μέσω του γονιού. Πρέπει να είναι μοναδικό. Αν οριστεί άλλο με το ίδιο όνομα, το τελευταίο θα αντικαταστήσει το προηγούμενο.

- **classPath(απαραίτητο):** Είναι μια διαδρομή αρχείου που θα έχει το AnimationClass που θα ορίζει την εμφάνιση του.
- **animName(απαραίτητο):** Είναι ένα όνομα από ένα Animation που ορίζεται στο AnimationClass του αρχείου “classPath”. Αυτό το Animation θα γίνει αυτό που θα εμφανιστεί από το ChildObject.
- **x, y(απαραίτητα):** Είναι ακέραιοι αριθμοί που ορίζουν τα Offsets που θα εμφανιστεί (Spawn) το ChildObject σε σχέση με την θέση γονιού.

Για παράδειγμα, παρακάτω βλέπουμε ότι προσθέτουμε ένα Sticky ChildObject το οποίο θέλουμε να το χρησιμοποιήσουμε σαν “Θαυμαστικό” που θα εμφανίζεται όταν το NPC έχει κάποια διαθέσιμη αποστολή:

Πίνακας 3.1.3: Παράδειγμα LifeformClass: ChildObjects

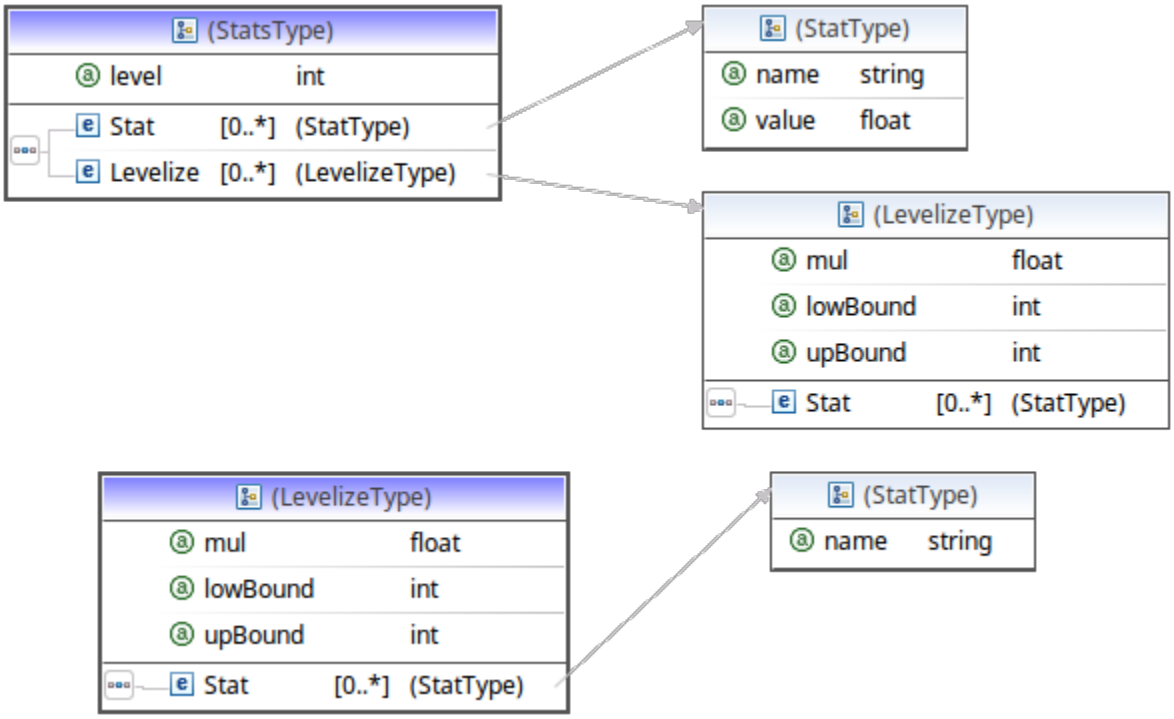
```
<ChildObjects>
     <Sticky name="mark" classPath="Demo/Sprites/Mark/Mark.xml"
           animName="Default" x="0" y="-50" />
</ChildObjects>
```

Στοιχείο 3ο: Stats (Απαραίτητο)

Σε αυτό το στοιχείο ορίζονται όλα τα στατιστικά που θα έχουν τα Lifeform που θα παραχθούν. Τα στατιστικά είναι αριθμοί που ορίζουν πόσο “Δυνατό” είναι το Lifeform. Για παράδειγμα, ορίζουν πόση ζωή έχουν, πόση επίθεση, ακόμα και την ταχύτητα κίνησής τους.

Επειδή όμως τα LifeformClass έχουν τον κύριο σκοπό να φτιάχνουν μαζικά Lifeforms, εάν όλα τα Lifeform που παραχθούν από μια LifeformClass έχουν τα ίδια στατιστικά τότε οι μάχες μετά εν λόγο Lifeforms μπορεί να γίνουν πολύ προβλέψιμες και βαρετές. Για αυτό και παρέχονται μηχανισμοί τυχαιοποίησης των στατιστικών εντός ορίων.

Ας δούμε την δομή:



Εικόνα 3.1.3: Ορισμός των τύπων Stats, Stat(S,L), και Levelize

Η δομή Stats έχει ένα απαραίτητο Attribute “level” που ορίζει το επίπεδο των Lifeform που θα παραχθούν. Ύστερα, μπορεί να έχει πολλές δομές Stat και μετά μπορεί να έχει πολλές δομές Levelize.

Η δομές Stat της δομής Stats πρέπει να έχουν από ένα όνομα “name” και μια τιμή “value”. Όλες αυτές οι δομές θα πάνε και θα βάλουν σε όλα τα Lifeforms που θα παραχθούν στο στατιστικό “name” ακριβώς την τιμή “value”. Για παράδειγμα:

Πίνακας 3.1.4: Παράδειγμα LifeformClass: Stats

```
<Stats level="1">
    <Stat name="Speed" value="250" />
    ...
</Stats>
```

Εδώ, στα Lifeform που θα φτιαχτούν θα έχουν επίπεδο 1 και στο στατιστικό “Speed” την τιμή 250.

Οι δομές Levelize έρχονται μετά τα Stat και έχουν τα εξής Attributes:

- **mul(προαιρετικό):** Δεκαδικός αριθμός που θα πολλαπλασιαστούν τα “lowBound” και “upBound” (αφού τους προστεθεί η τιμή του “level”). Εάν δεν οριστεί, τότε παίρνει αυτόματα την τιμή (1).

- **lowBound(προαιρετικό):** Ακέραιος αριθμός που αντιπροσωπεύει το κατώτατο όριο που θα πρέπει να πάρουν τα ακόλουθα Stats. Η τιμή του “lowBound” θα προστεθεί στο “level” και θα πολλαπλασιαστεί με το “mul”. Η τιμή που θα βγει ως αποτέλεσμα, θα είναι η κατώτατη δυνατή τιμή που θα πρέπει να πάρουν αυτά τα Stats. Εάν δεν οριστεί του δίνεται αυτόματα η τιμή (-1).
- **upBound(προαιρετικό):** Αντίστοιχα με το “lowBound” μόνο που η τιμή που θα βγει από τις πράξεις θα είναι η ανώτατη δυνατή τιμή που θα μπορούν να πάρουν τα Stats. Εάν δεν οριστεί, του δίνεται αυτόματα η τιμή (+1).

Οι δομές “Stat” που βρίσκονται μέσα στο Levelize, έχουν μόνο ένα Attribute “name” που είναι το Stat που θα πρέπει να μπει αυτόματα η τιμή.

Σύμφωνα με τις τιμές των “lowBound” και “upBound” που πάρουν εσωτερικά στην μηχανή, θα μπουν τυχαίες τιμές σε όλα τα Stat που ορίζονται μέσα σε αυτό το Levelize. Πιο ειδικά, ο τύπος που εφαρμόζεται για κάθε Stat είναι ο παρακάτω:

$$stat[name] = RAND(\alpha, \beta)(\alpha < \beta)(3.1.1)$$

Όπου: $\alpha = |level + lowBound| * mul$, $\beta = |level + upBound| * mul$

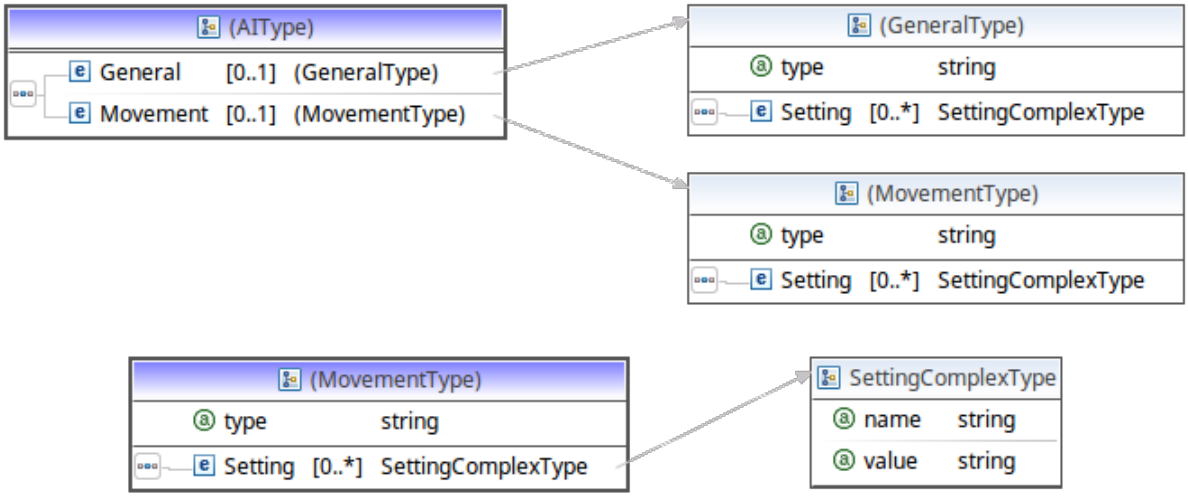
Αξίζει να σημειωθεί ότι τα Stats εφαρμόζονται όπως στην σειρά στο XML αρχείο. Για παράδειγμα, αν στην αρχή βάλουμε κάποια τιμή στο Stat “Speed” και παρακάτω το ξανά-ορίσουμε, η τιμή που θα πάρει θα είναι η τελευταία. Όπως επίσης αν το βάλουμε παρακάτω σε δομή Levelize, τότε η τιμή θα παρθεί από τον τύπο και όχι ότι ορίσαμε παραπάνω. Το ίδιο συμβαίνει αν ορίσαμε σε Levelize ένα Stat και το ξανά-ορίσαμε σε παρακάτω Levelize. Το τελευταίο θα επικρατήσει.

Πίνακας 3.1.5: Ολοκληρωμένο παράδειγμα LifeformClass: Stats

```
<Stats level="1">
     <Stat name="Speed" value="250" />
     <Stat name="HP5" value="1" />
     <Stat name="MP5" value="5" />
     <Levelize lowBound="+2" mul="100" upBound="+3">
           <Stat name="MaxHP" />
           <Stat name="MaxMana" />
     </Levelize>
     <Levelize lowBound="+2" mul="10" upBound="+3">
           <Stat name="Patk" />
           <Stat name="Mdef" />
           <Stat name="Matk" />
           <Stat name="Pdef" />
     </Levelize>
</Stats>
```

Στοιχείο 4ο: AI (Προαιρετικό)

Σε αυτό το στοιχείο, ορίζεται η γενική και κινητική συμπεριφορά των παραγόμενων Lifeforms.



Εικόνα 3.1.4: Ορισμός των τύπων AI, Setting

Σε αυτό το στοιχείο ρυθμίζεται η τεχνητή νοημοσύνη που θα έχουν τα Lifeforms. Υπάρχουν δύο προαιρετικές ρυθμίσεις: General για την ρύθμιση της γενικής συμπεριφοράς και Movement για την ρύθμιση της κινητικής. Πιο ειδικά και τα δύο αν οριστούν πρέπει να έχουν:

- **type(απαραίτητο):** Πρέπει να πάρει διακριτές τιμές. Για το General, μπορεί να πάρει “Aggressive” ή “Passive”. Για το Movement μπορεί να πάρει “Wandering” ή “Idle”.

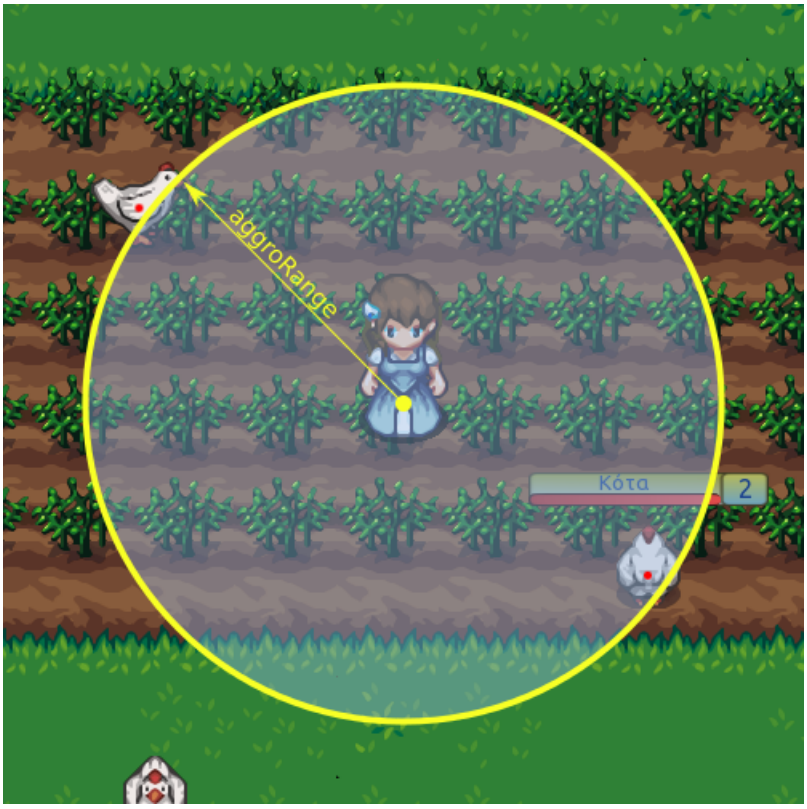
Όταν οριστεί κάτι από δυο, πρέπει ανάλογα το τι “type” δόθηκε να οριστούν “Settings” μέσα με συγκεκριμένα ονόματα. Υπάρχουν οι εξής περιπτώσεις:

1. **General type=”Aggressive”**

Το General με αυτή την ρύθμιση θα κάνει τα Lifeforms να είναι επιθετικά. Θα ελέγχουν μια περιοχή γύρω τους αν υπάρχει εχθρός (Lifeform που είναι σε Faction με εχθρότητα σε σχέση με το Faction που έχει το Lifeform που ελέγχει). Αν βρεθεί εχθρός μέσα σε αυτή την περιοχή, τότε αρχίζουν να τον κυνηγάνε και να του επιτίθενται με ότι Abilities έχουν στην διάθεσή τους. Αν απομακρυνθούν πολύ από την αρχική τους θέση, σταματάνε την μάχη και οπισθοχωρούν.

- **aggroRange:** Ορίζει την ελάχιστη απόσταση που θα πρέπει να βρεθεί εχθρός σε σχέση με το Lifeform για να αρχίσει να του επιτίθεται.

- **driftRange:** Ορίζει την μέγιστη απόσταση που θα μπορεί απομακρυνθεί το Lifeform από την αρχική του θέση πριν αρχίσει να οπισθοχωρεί.
- Τόσο το aggroRange και το driftRange υπολογίζονται στο παιχνίδι σε κυκλική ακτίνα γύρω από το σημείο που ελέγχονται. Με άλλα λόγια, η περιοχή τους είναι ένας κύκλος με ακτίνα “aggroRange” και κέντρο το σημείο που βρίσκονται εκείνη την στιγμή. Στην περίπτωση του “driftRange” είναι το ίδιο, μόνο που το κέντρο είναι το αρχικό σημείο εκκίνησης του κυνηγητού.
- Τα Lifeforms που ελέγχονται για αν είναι εντός της περιοχής, ελέγχονται ως προς το σημείο που βρίσκονται και όχι για κάποιο ορθογώνιο που περιέχουν (πχ ορθογώνιο σύγκρουσης ή ορθογώνιο στόχευσης). Για παράδειγμα, στην παρακάτω εικόνα τα κόκκινα σημεία ελέγχονται.



Εικόνα 3.1.5: Παράδειγμα περιοχής AggroRange

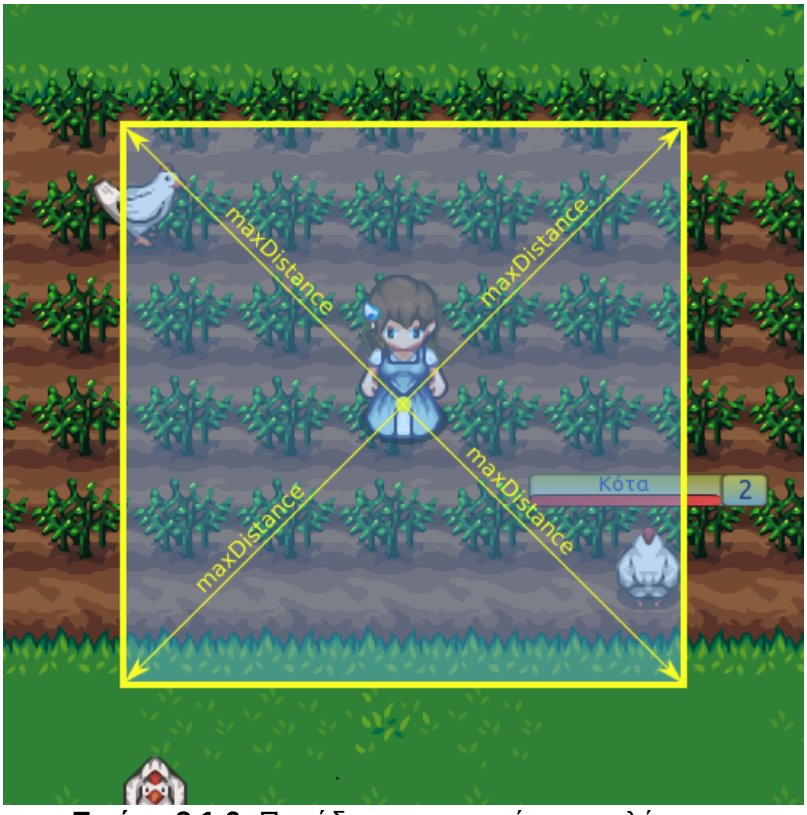
2. **General type=”Passive”**

Εδώ τα Lifeforms δεν κάνουν κάτι αν έρθουν σε επαφή με εχθρούς. Η διαχείριση του αφήνεται ολοκληρωτικά στο Movement. Δεν χρειάζεται Settings εδώ.

3. **Movement type=”Wandering”**

Τα Lifeforms με αυτή την ρύθμιση θα κινούνται ελεύθερα μεταξύ ορίων γύρω από το σημείο που εμφανίστηκαν. Οι κινήσεις που θα κάνουν θα είναι τυχαίες άλλα δεν θα ξεπερνάνε κάποια όρια που ορίζονται. Οι κινήσεις μπορεί να ρυθμιστούν να μην είναι συνεχόμενες άλλα να υπάρχουν στάσιμα κενά ενδιάμεσα με τυχαίους χρόνους αναμονής. Πέρα από αυτό, μπορεί να ρυθμιστεί σε αυτή την αναμονή να εκτελείται κάποια τυχαία ενέργεια, κυρίως Animation. Τα απαραίτητα Settings για αυτό είναι:

- **maxWaitTime:** Ο μέγιστος χρόνος σε δευτερόλεπτα (μπορεί να είναι και δεκαδικός) που μπορεί να περάσει πριν το Lifeform πρέπει να κινηθεί πάλι.
- **minWaitTime:** Αντίστοιχα με πάνω, ορίζεται ο ελάχιστος χρόνος που πρέπει να περάσει προτού να κινηθεί πάλι το Lifeform.
- **maxDistance:** Είναι η μέγιστη απόσταση που θα μπορεί να απομακρυνθεί το Lifeform από το σημείο που εμφανίστηκε όσο περιπλανιέται. Εδώ σε αντίθεση με το “aggroRange” και “driftRange”, οι αποστάσεις καθορίζονται σε τετράγωνο όπου οι διαγώνιες έχουν από το κέντρο απόσταση “maxDistance”. Για παράδειγμα:



Εικόνα 3.1.6: Παράδειγμα περιοχής περιπλάνησης

- **idleAction:** Περιέχει ένα string που πρέπει να αντιστοιχεί σε κάποιο Animation που έχει το AnimationClass που έχει οριστεί. Αυτό το Animation θα το κάνει το Lifeform κάθε φορά που τυχαίνει μέσα στην αναμονή μεταξύ των περιπλανήσεων.
- Αν δεν οριστεί, δεν θα κάνει τίποτα. Αν δεν οριστεί κάποια από τρεις πρώτες ρυθμίσεις (maxWaitTime, minWaitTime, maxDistance), τότε γίνεται επαναφορά και των τριών με τιμές:

minWaitTime=0 δεπτερόλεπτα

maxWaitTime=5 δευτερόλεπτα

maxDistance=500

Το **idleAction** θα αγνοηθεί σε αυτή την περίπτωση.

4. **Movement type=”Idle”**

Σε αυτή την περίπτωση, το Lifeform θα παραμένει ακίνητο όσο δεν έχει να κάνει κάτι (πχ να κυνηγήσει κάποιο εχθρό). Δεν ορίζονται ιδιότητες εδώ.

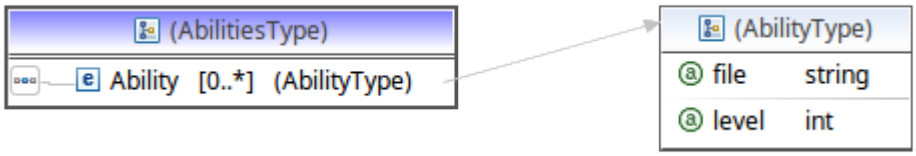
Πίνακας 3.1.6: Παράδειγμα Lifeform: AI

```
<AI>
    <General type="Aggressive">
        <Setting name="aggroRange" value="300" />
        <Setting name="driftRange" value="600" />
    </General>
    <Movement type="Wandering">
        <Setting name="maxDistance" value="400" />
        <Setting name="minWaitTime" value="1" />
        <Setting name="maxWaitTime" value="4" />
        <Setting name="IdleAction" value="Eat" />
    </Movement>
</AI>
```

Όπως είχαμε προαναφέρει, το AI είναι προαιρετικό. Αν δεν οριστεί καθόλου, τότε το AI του Lifeform θα γίνει General=Passive και Movement=Idle.

Στοιχείο 5ο: Abilities (Προαιρετικό)

Σε αυτό το στοιχείο μπορεί να προστεθεί μια λίστα από Abilities που θα έχει το Lifeform.



Εικόνα 3.1.7: Ορισμός του Ability

Τα Abilities που ορίζονται θα τα έχουν όλα τα Lifeforms που θα έχουν αυτό το LifeformClass. Στο στοιχείο Ability ορίζονται τα ορίσματα:

1. **file:** Η διεύθυνση του Lua αρχείου που έχει το AbilityClass.
2. **level:** Το επίπεδο που θα έχει το Ability.

Πίνακας 3.1.7: Παράδειγμα Lifeform: Abilities

```
<Abilities>
    <Ability file="Demo/Abilities/Strike.lua" level="1" />
</Abilities>
```

Περισσότερα για τα Abilities, στο εν λόγω κεφάλαιο.

Στοιχείο 6ο: Script

Το στοιχείο Script είναι παρωχημένο κομμάτι της μηχανής. Αγνοείται εντελώς από την μηχανή άλλα κρατείται στο XSD Schema για μελλοντικές πιθανές χρήσεις (Κυρίως Debuging).

Με αυτά τα στοιχεία, μπορεί να κατασκευαστεί ο κύριος κορμός ενός LifeformClass. Με αυτά τα στοιχεία μπορεί να χρησιμοποιηθεί στην πράξη. Τώρα όμως πάμε να δούμε πως ορίζουμε την εμφάνιση των Lifeform.

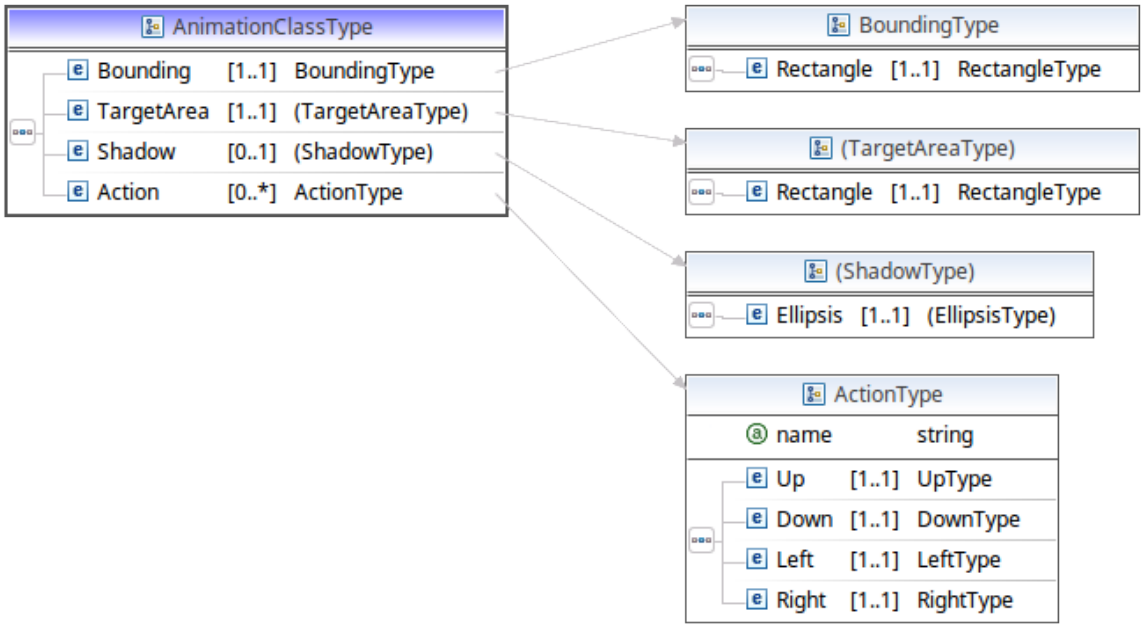
3.2 Τα XML αρχεία AnimationClass

Τα AnimationClasses είναι δομές δεδομένων που προσδιορίζουν την ολική εμφάνιση ενός Lifeform. Στην ουσία είναι πολλές ομάδες από Animations που χρησιμοποιούνται κάτω από το ειδικές περιπτώσεις. Για παράδειγμα, μια ομάδα Animation που πρέπει να έχει ένα AnimationClass είναι τα Animations για την κίνηση. Άλλη μια είναι για την στασιμότητα (Όταν δεν κινείται) κ.α. Οι ομάδες αυτές ονομάζονται “Actions”.

Κάθε Action πρέπει να έχει 4 “δείκτες” σε Animation για κάθε από τις 4 κατευθύνσεις, πάνω, κάτω αριστερά και δεξιά. Αυτοί οι δείκτες δεν είναι τίποτα άλλο από διαδρομές σε αρχεία στον δίσκο που περιέχουν τα Animation που ζητούνται.

Πέρα από τα Actions, τα AnimationClass ορίζουν και κάποια επιπλέον πράγματα που πρέπει να καθορίζονται προκειμένου να υπάρχει σωστή συμπεριφορά του αντικειμένου που τα χρησιμοποιούν. Αυτά είναι:

- **Bounding Rectangle** (Περιοχή πάνω στο Αντικείμενο/Lifeform που θα ελέγχεται για συγκρούσεις)
- **Target Rectangle** (Περιοχή πάνω στο Αντικείμενο/Lifeform που θα χρησιμοποιείται για να μπορεί να επιλεγεί)
- **Shadow Elipsis** (Περιοχή πάνω στο Αντικείμενο/Lifeform που θα εμφανίζεται σαν σκιά)



Εικόνα 3.2.1: AnimationClass XSD Schema

Στο παραπάνω διάγραμμα περιγράφεται η δομή ενός AnimationClass. Πριν πάμε να περιγράψουμε τα πρώτα στοιχεία (Bounding, TargetArea, Shadow), είναι απαραίτητο να αναλύσουμε τα Actions που θα δώσουν την μορφή στο Lifeform που θέλουμε να φτιάξουμε.

Έστω ότι θέλουμε να φτιάξουμε την Μαριάννα. Θα φτιάξουμε τις βασικές ενέργειες (Actions) που θα κάνει όπως Κίνηση και στασιμότητα. Η μηχανή για κάθε από τις

ενέργειες χρειάζεται 4 Animations για κάθε κατεύθυνση. Πώς θα φτιάξουμε τα Animations;

Τα Animations είναι στην ουσία διαδοχικές εικόνες που εναλλάσσονται γρήγορα η μία μετά την άλλη. Αυτές οι εικόνες λέγονται και Sprites. Όμως αυτές οι εικόνες είναι αρχεία που φορτώνονται στην VRAM; Αν ήταν μεμονωμένα αρχεία, η φόρτωση και η εναλλαγή θα ήταν αργή. Για αυτό και όλες τις εικόνες που χρησιμοποιούνται σε ένα Animation τις βάζουμε μαζί σε μια μεγάλη εικόνα που λέγετε SpriteSheet.

Εικόνα 3.2.2: Μέρος από Spritesheet

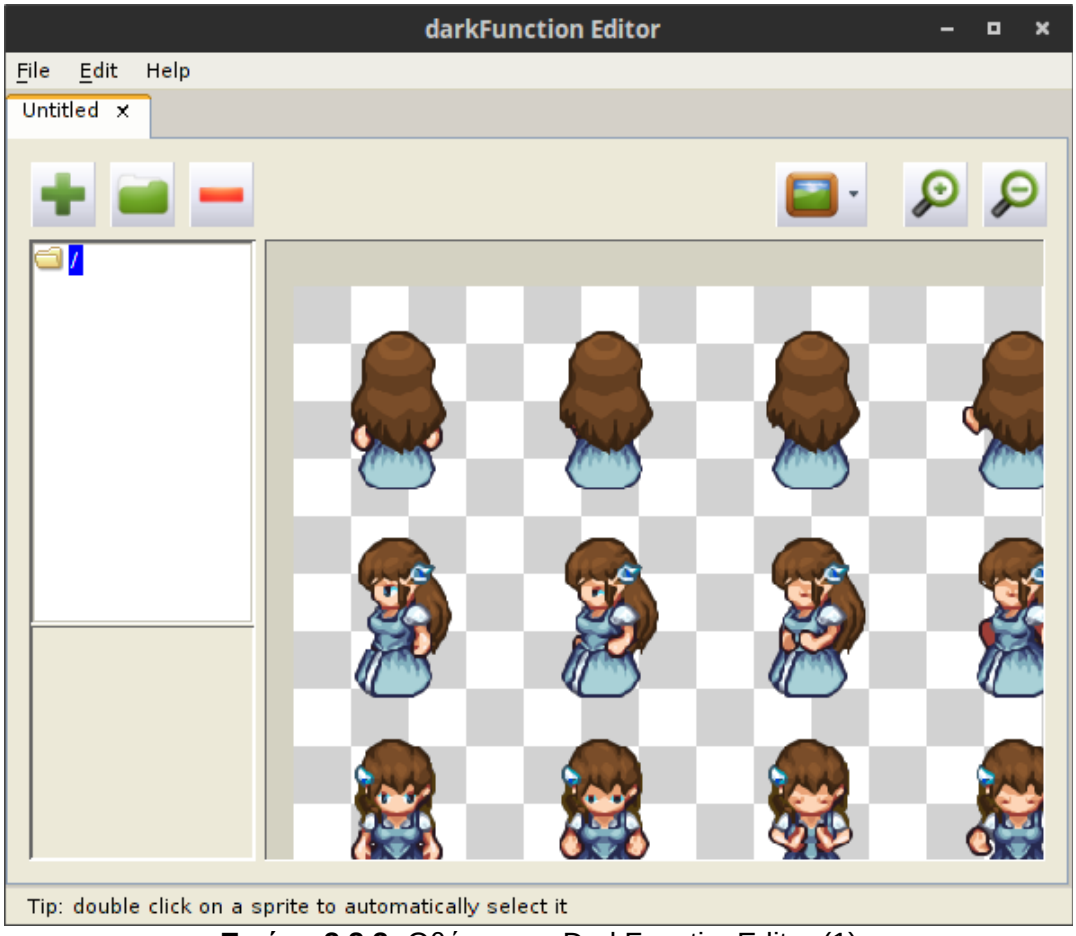
Παραπάνω βλέπουμε ένα μέρος από το SpriteSheet που θα χρησιμοποιήσουμε για να φτιάξουμε την Μαριάννα. Περιέχει όλα τα Sprites των 4 κινήσεων. Πχ στην πρώτη σειρά έχει την κίνηση προς τα πάνω. Έτσι όπως είναι στην μηχανή δεν λέει κάτι για τα Animations. Πρέπει να της πούμε ποιες εικόνες είναι για ποια Animation. Αρχικά πρέπει να της δώσουμε κάποιες πληροφορίες για το Spritesheet, όπως που

βρίσκεται κάθε Sprite και τι όνομα έχει. Σε αυτό μας βοηθάει το εξωτερικό εργαλείο που λέγεται Dark Function Editor.

a) Το εργαλείο Dark Function Editor

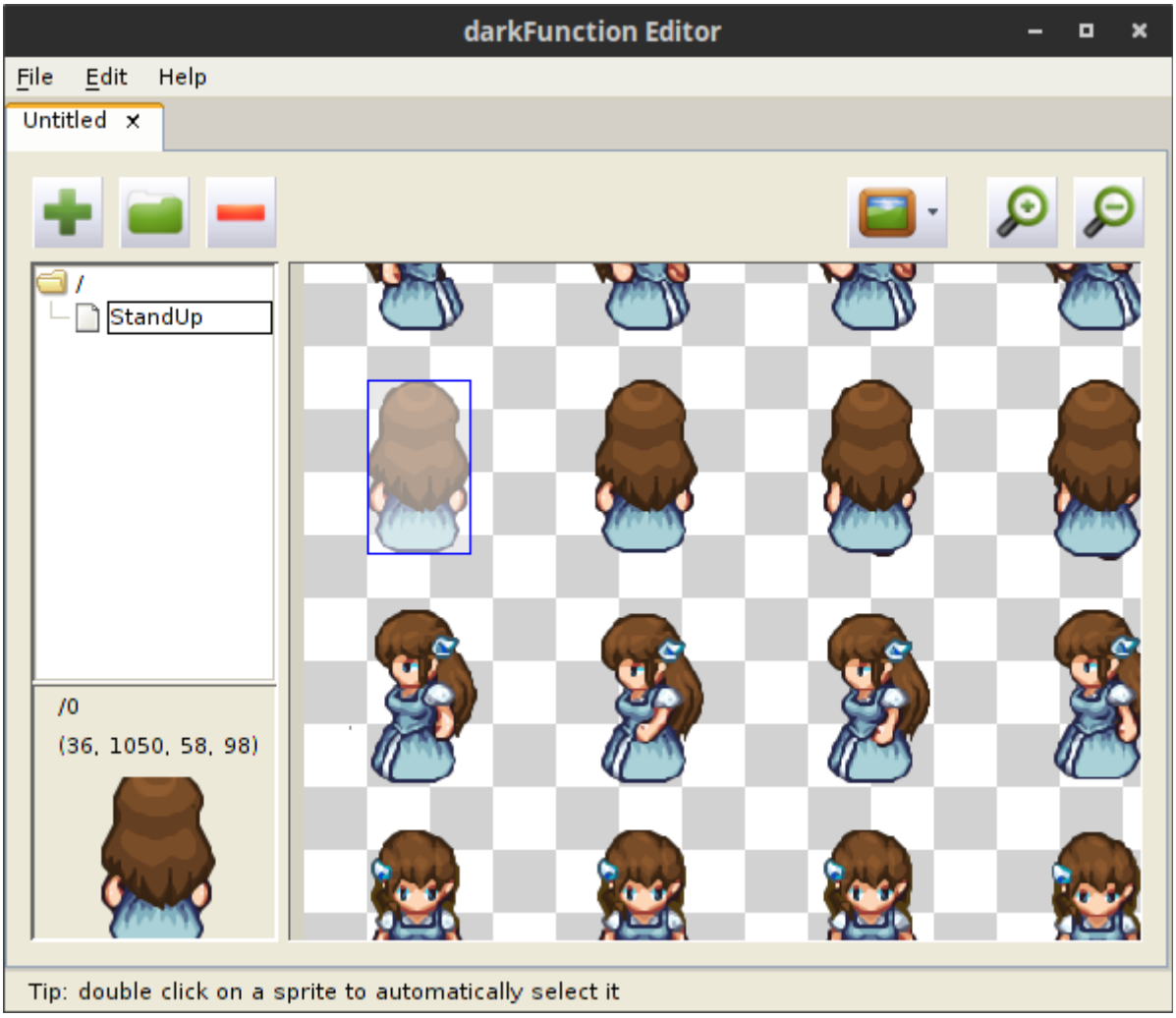
Το εργαλείο Dark Function Editor είναι ένα εργαλείο που μας βοηθάει να παράγουμε εύκολα πληροφορίες για Spritesheets καθώς και Animations. Δέχεται σαν είσοδο ένα Spritesheet και εμείς μέσα από το γραφικό του περιβάλλον παράγουμε τους ορισμούς των Sprites και τα Animation. Στο τέλος μας δίνει 2 αρχεία XML, ένα που έχει πληροφορίες για τα Sprites και ένα που μας λέει ποια Animation έχει αυτό το Spritesheet και πως να τα εμφανίσει η μηχανή. Το εργαλείο παρέχεται από τον Samuel Taylor υπό την άδεια GPL3 (http://darkfunction.com/editor/).

Πάμε να φτιάξουμε τα Animations της Μαριάννας. Αφού ανοίξουμε το πρόγραμμα πάμε File->New->Spritesheet και πατάμε “Define Sprites” γιατί έχουμε ήδη έτοιμη την ενιαία εικόνα. Διαφορετικά αν έχουμε πολλές εικόνες, το εργαλείο μας δίνει την δυνατότητα να τις ενώσουμε σε μια ενιαία. Επιλέγουμε την εικόνα μας (Sprites/Female_NPC_3/Female_NPC_3.png) και μας βγάζει στην παρακάτω οθόνη.



Εικόνα 3.2.3: Οθόνη απο DarkFunctionEditor (1)

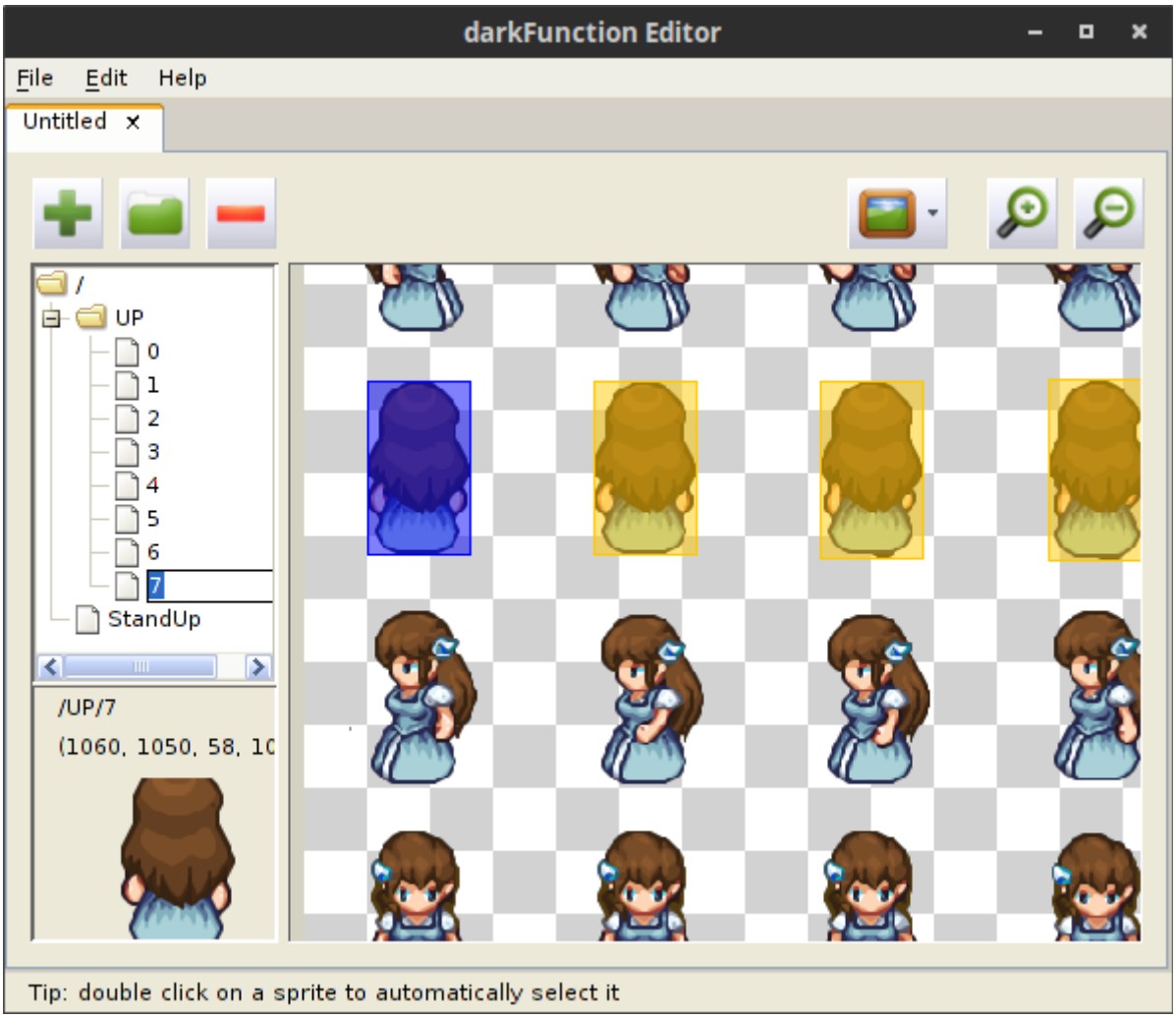
Μας εμφάνισε την εικόνα μας. Εδώ λοιπόν αν πατήσουμε πάνω στην εικόνα δεξί κλικ, θα κινηθούμε πάνω σε αυτή. Έτσι ψάχνουμε τα Sprites που έχουν να κάνουν με την κίνηση (είναι οι σειρές 9-12). Αφού τις βρούμε, τώρα μπορούμε να “αποκόψουμε” τα Sprites και να τους δώσουμε όνομα. Κάνουμε διπλό κλικ πάνω στην πρώτη εικόνα και αμέσως ένα τετράγωνο την περιτριγυρίζει και μια εγγραφή δημιουργείται δίπλα στην οποία πρέπει να βάλουμε το όνομά της. Τις βάζουμε το αντίστοιχο όνομα.



Εικόνα 3.2.4: Οθόνη απο DarkFunctionEditor (2)

Το εργαλείο μας δίνει τη δυνατότητα να φτιάξουμε φακέλους σαν κατηγορίες. Αυτό βοηθάει στην οργάνωση των Sprites ώστε να μπορούμε να φτιάξουμε πιο εύκολα τα Animation. Πατώντας τον πράσινο φάκελο θα μας φτιάξει ένα φάκελο και θα του δώσουμε ένα όνομα, πχ (UP). Εκεί για παράδειγμα θα βάλουμε με διπλό κλικ ότι Sprite έχει να κάνει με την κίνηση προς τα πάνω. Καλό θα ήταν να τα βάλουμε με την

σειρά και να αφήνουμε το πρόγραμμα να βάζει μόνο του αριθμήσεις στο όνομα, ώστε να ξέρουμε την σειρά μετά.

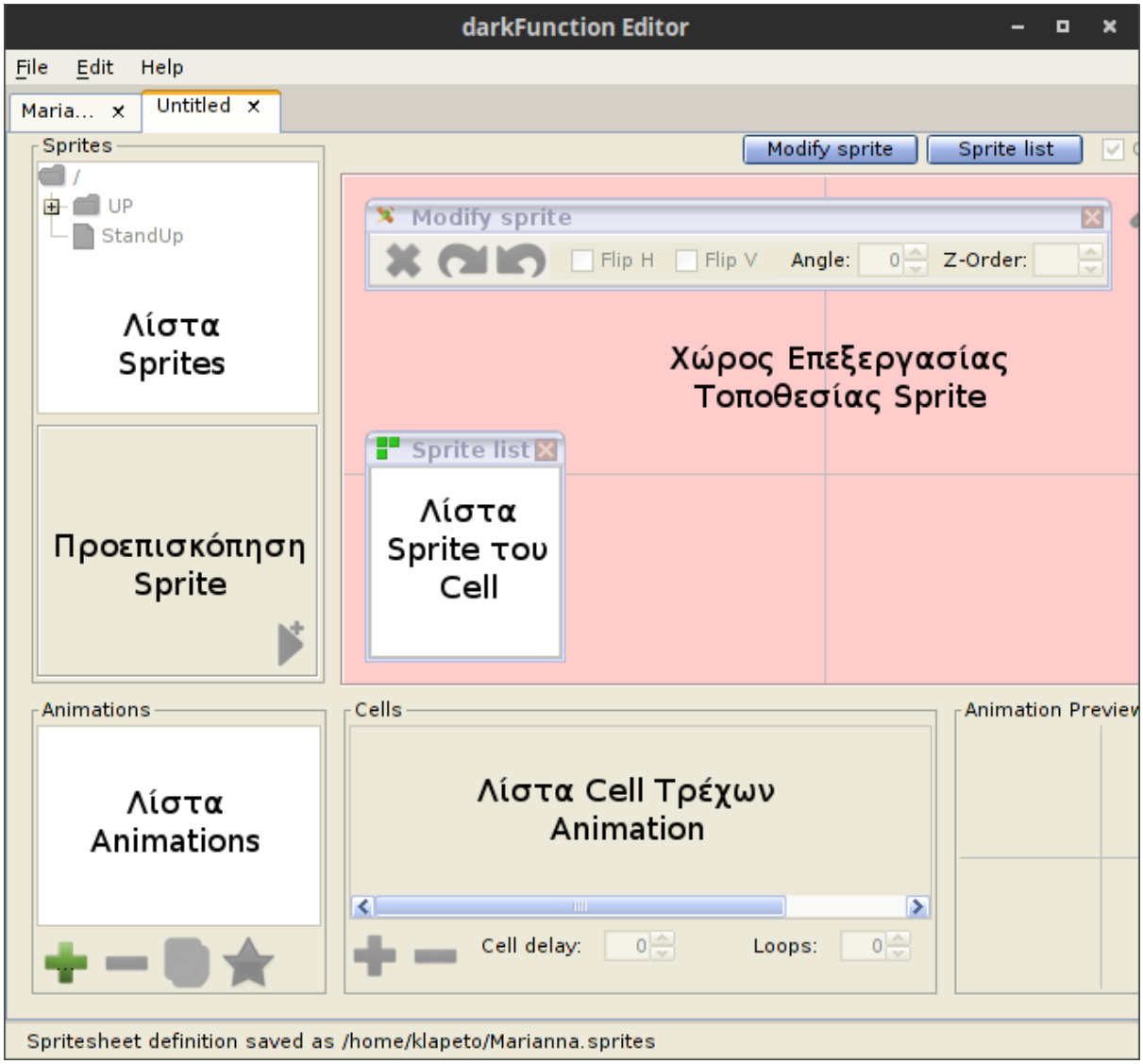


Εικόνα 3.2.5: Οθόνη απο DarkFunctionEditor (3)

Αφού τελειώσουμε και με τις 4 κινήσεις, είμαστε έτοιμοι να τις φτιάξουμε σε Animation. Όπως είμαστε, κάνουμε File->Save και το αποθηκεύουμε “Mariana.sprites”. Ύστερα File->New->AnimationSet. Πρέπει να επιλέξουμε ένα Spritesheet. Το πρόγραμμα έχει επιλέξει από μόνο του το τελευταίο που φτιάξαμε.

Αφού το ανοίξουμε, το πρόγραμμα μας βάζει στην λειτουργία για δημιουργία Animation. Αριστερά μας δείχνει την λίστα με τα Sprites που φτιάξαμε προηγουμένως. Από κάτω είναι μια προεπισκόπηση του Sprite που επιλέγουμε. Από κάτω από αυτό είναι η λίστα με τα Animation που θα δημιουργήσουμε. Δίπλα είναι η λίστα με τα Frames του τρέχοντος Animation. Δίπλα είναι προεπισκόπηση του Animation. Πάνω είναι ο χώρος που θα κινούμε τα Sprites στην σωστή θέση. Αυτός ο χώρος εί-

ναι πολύ σημαντικός γιατί αντιπροσωπεύει το Lifeform που θα εμφανίζεται έτσι. Το παράθυρο “Modify Sprite” είναι για επεξεργαζόμαστε το εκάστοτε Sprite σε κάθε Cell.



Εικόνα 3.2.6: Οθόνη απο DarkFunctionEditor (4)

Ας ξεκινήσουμε. Πατώντας τον πράσινο σταυρό κάτω από την Λίστα των Animation, μας προσθέτει ένα στοιχείο που μπορούμε να του αλλάξουμε όνομα με διπλό κλικ. Ας του δώσουμε “WalkUp” για να φτιάξουμε την κίνηση όταν κοιτάζει πάνω.

Δίπλα στην λίστα με τα Cell εμφανίστηκε ένα πράσινο κουτί (Cell). Το Cell αυτό είναι ένα Frame του Animation. Το Cell έχει χρονική διάρκεια και λίστα από Sprites. Ο χρόνος του ρυθμίζεται από το κουτί “Cell Delay”. Η τιμή που θα δοθεί εδώ, στην μηχανή θα είναι σε Centi Seconds (s*10e-2). Εφόσον χρησιμοποιούμε το πρόγραμμα

δεν μας πολύ ενδιαφέρει αυτό καθώς θα το πάμε με το μάτι, όμως για αρχή καλή τιμή είναι το 10. Επειδή τα Sprites του Animation είναι 8, τότε πατώντας το σταυρό κάτω από την λίστα των Cell μας προσθέτει ένα επιπλέον Cell και παίρνει αρχικό χρόνο τον ίδιο χρόνο με το προηγούμενο. Το κάνουμε μέχρι να έχουμε 8 Cells.

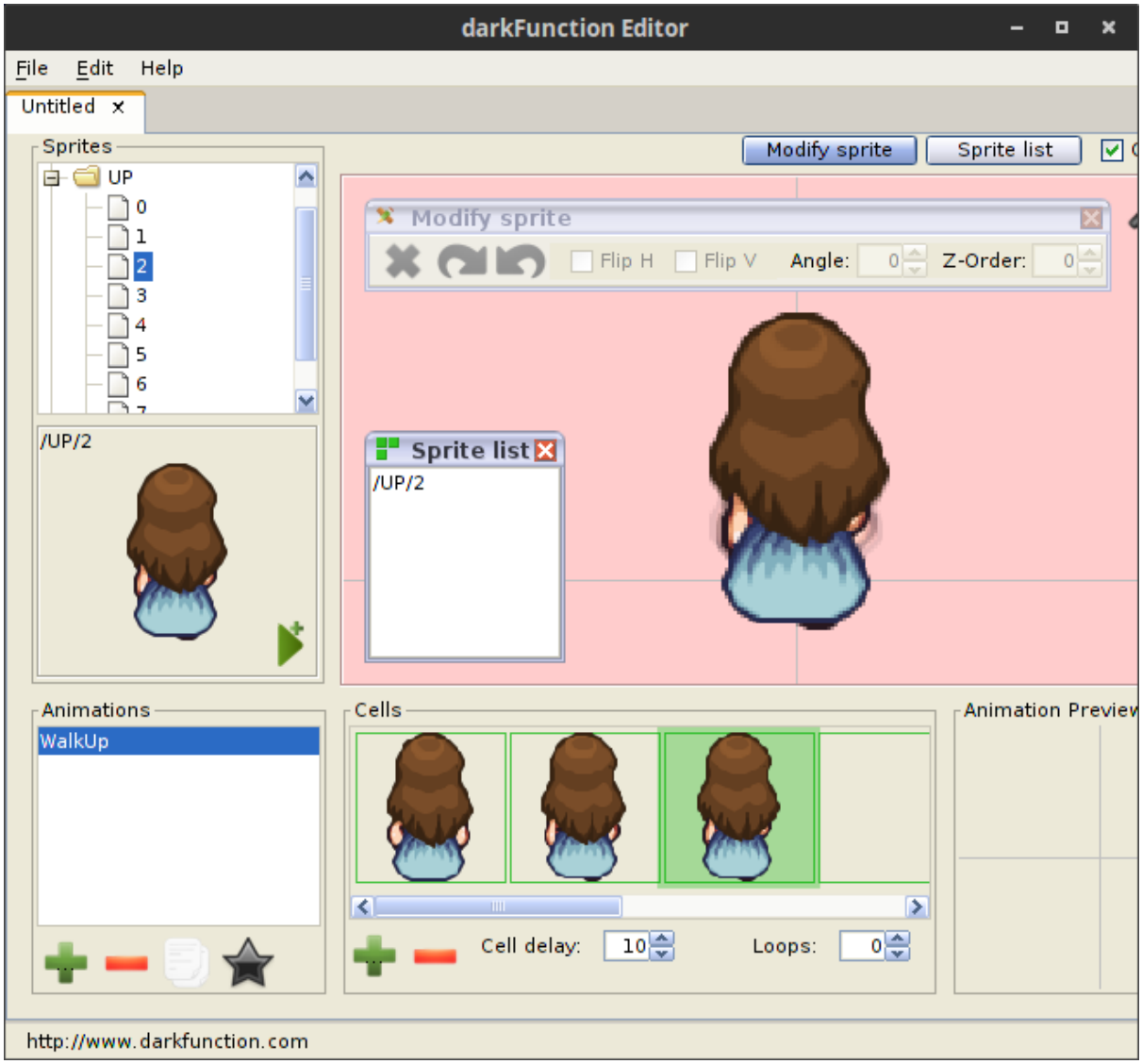
Από τη λίστα επιλέγουμε το πρώτο Cell. Τώρα πρέπει να του βάλουμε το Sprite. Από τη λίστα με τα Sprites, πάμε στον φάκελο UP και επιλέγουμε το πρώτο Sprite που είναι το “0” (ή όπως αλλιώς το ονομάσατε όταν φτιάχνατε το SpriteSheet). Με διπλό κλικ μπαίνει στο Cell και εμφανίζεται στον χώρο επεξεργασίας την τοποθεσίας.

Το Sprite εμφανίστηκε στο κέντρο του σταυρού. Τι είναι όμως αυτός ο σταυρός και τι αναπαριστά; Το κέντρο αυτό του σταυρού δεν θα είναι τίποτα άλλο από τις συντεταγμένες του Lifeform στην μηχανή. Έτσι όπως είναι, όταν θα εμφανίσουμε μια Μαριάννα, η εικόνα της θα εμφανιστεί στο κέντρο των συντεταγμένων αυτών. Χρειάζεται να αλλάξει αυτό;

Η απάντηση είναι ναι. Αυτό γιατί μπορεί να φαίνεται εντάξει στην τελική, αλλά όταν θα αρχίζει να κινείται θα έχει απόκλιση από εκεί που θέλουμε να πάει. Στην πραγματικότητα θα βρίσκεται εκεί που της είπαμε άλλα δεν θα φαίνεται έτσι. Το κέντρο της βρίσκεται στο σημείο, το ίδιο και η εικόνα της γιατί έτσι την ρυθμίσαμε.

Σε αυτά τα παιχνίδια, ο χαρακτήρας φαίνεται σαν να είναι όρθιος. Για αυτό και όταν τον στέλνουμε σε ένα σημείο, θέλουμε τα πόδια του να βρίσκονται εκεί. Η μηχανή όταν κινεί τα Lifeform, μετακινεί το σημείο τους (κέντρο) εκεί που θέλουμε. Έτσι αν θέλουμε τα πόδια της να στέκονται πάντα πάνω από το σημείο που πρέπει να βρίσκεται, πρέπει να της βάλουμε το κέντρο στα πόδια της. Δεν θα αλλάξουμε το σημείο αλλά το που εμφανίζεται η εικόνα της. Αυτό θα κάνουμε με την επεξεργασία της τοποθεσίας. Έχοντας το κέντρο του σταυρού σαν το μέρος που θα είναι το κέντρο του Lifeform, θα μετακινήσουμε το Sprite, είτε με το ποντίκι είτε με τα βελάκια του πληκτρολογίου. Το κεντράρουμε οριζοντίως.

Ώρα να βάλουμε το δεύτερο. Αφού επιλέξουμε το δεύτερο Cell, εισάγουμε το δεύτερο Sprite και το τοποθετούμε στην θέση που είναι το προηγούμενο. Προσέχουμε να είναι κεντραρισμένο ώστε να μην είναι ανώμαλη η κίνηση του Animation.



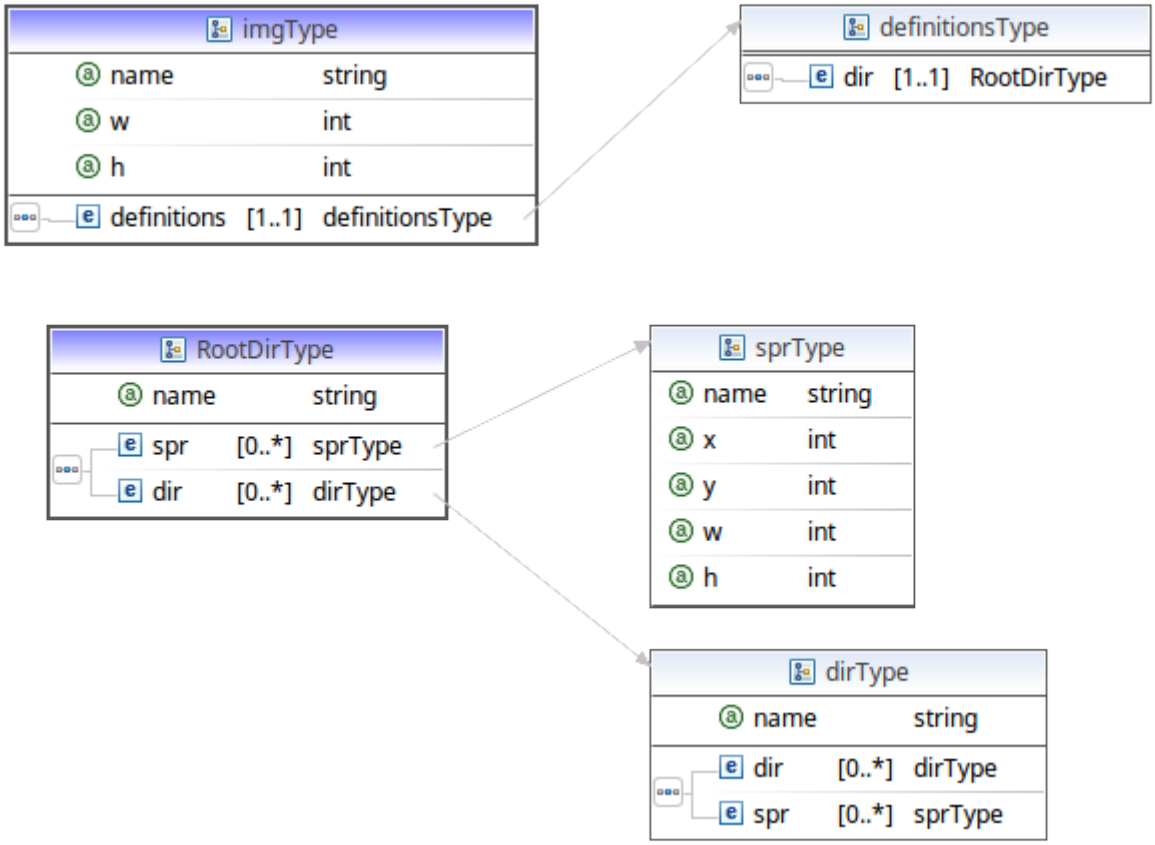
Εικόνα 3.2.7: Οθόνη απο DarkFunctionEditor (5)

Το εργαλείο μας δίνει μια ιδέα για το που βρίσκεται το προηγούμενο Sprite, κάνοντας την ζωή μας πιο εύκολη. Αφού τελειώσουμε με όλα τα Cells, πατάμε το Looping καθώς θα είναι επαναλαμβανόμενο Animation και πατάμε το πράσινο κουμπί (Play) στο παράθυρο του “Animation Preview” για να ελέγξουμε αν είναι εντάξει. Αν όχι, ρυθμίζουμε κάποιο Cell που απέχει σε απόσταση ή χρόνο μέχρι να είναι τέλειο.

Τώρα μπορούμε να φτιάξουμε και τις άλλες κινήσεις. Αφού τις φτιάξουμε, αποθηκεύουμε το αρχείο με κατάληξη “*.anim”.

b) Τα XML αρχεία Sprites

Ας πάμε να αναλύσουμε τα αρχεία που παράγει το εργαλείο και μεταφράζει σε Animations η μηχανή.



Εικόνα 3.2.8: Ορισμός του DarkFunction (*.sprites) XML

Το δένδρο του αρχείου έχει ως εξής:

- **<img> (Μοναδικό)** Το πρώτο στοιχείο. Πρέπει να έχει
  - **name (απαραίτητο):** Η διαδρομή του αρχείου της εικόνας του Spritesheet
  - **w (απαραίτητο):** Το πλάτος της εικόνας
  - **h (απαραίτητο):** Το ύψος της εικόνας
  - **<definitions> (Μοναδικό)**
    - **<dir> (1 και πάνω)** Αναπαριστά έναν φάκελο. Το πρώτο είναι απαραίτητο.
      - **name (απαραίτητο):** το όνομα του φακέλου
      - **<dir> (0 και πάνω):** Μπορεί να έχει εμφωλευμένο <dir>
      - **<spr> (0 και πάνω)** Ορισμός ενός Sprite
        - **name (απαραίτητο):** το όνομα του Sprite
        - **x (απαραίτητο):** Η συντεταγμένη x στην εικόνα που θα αρχίσει η περικοπή

- **y (απαραίτητο):** Η συντεταγμένη y στην εικόνα που θα αρχίσει η περικοπή
- **w (απαραίτητο):** Το πλάτος της περικοπής
- **h (απαραίτητο):** Το ύψος της περικοπής

Πίνακας 3.2.1: Παράδειγμα αρχείου "*.sprites"

```
<?xml version="1.0"?>
<!-- Generated by darkFunction Editor (www.darkfunction.com) -->
<img name="Female_NPC_3.png" w="1664" h="2688">
     <definitions>
          <dir name="/">
               <dir name="UP">
                    <spr name="0" x="164" y="1050" w="58" h="98" />
                    <spr name="1" x="292" y="1050" w="58" h="100" />
                    <spr name="2" x="420" y="1050" w="58" h="102" />
                    <spr name="3" x="548" y="1050" w="58" h="100" />
                    <spr name="4" x="676" y="1050" w="58" h="98" />
                    <spr name="5" x="804" y="1050" w="58" h="100" />
                    <spr name="6" x="932" y="1050" w="58" h="102" />
                    <spr name="7" x="1060" y="1050" w="58" h="100" />
               </dir>
               <spr name="StandUp" x="36" y="1050" w="58" h="98" />
          </dir>
     </definitions>
</img>
```

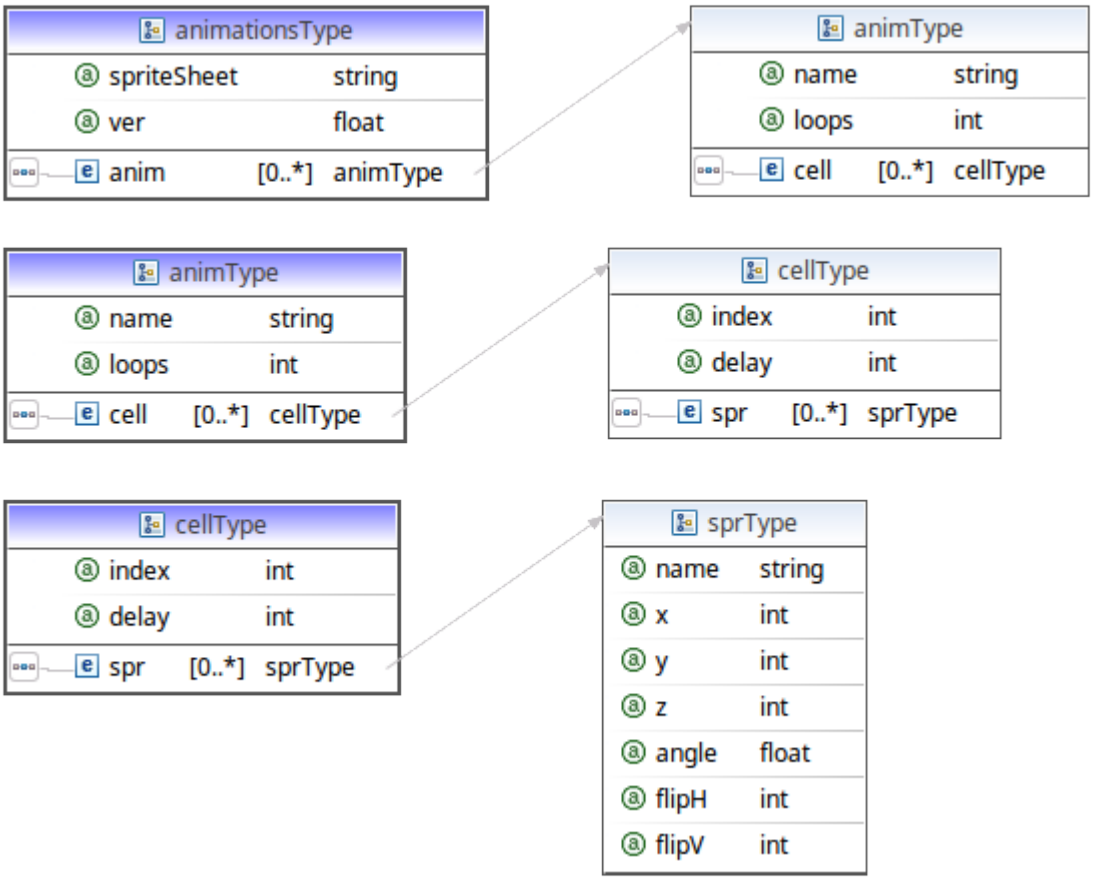
Στο παραπάνω παράδειγμα βλέπουμε το αρχείο που παρήγαγε το εργαλείο από το παράδειγμα χρήσης του στο Spritesheet. Όπως βλέπουμε, τα στοιχεία <img> <definitions> και <dir name=”/”> πρέπει να υπάρχουν πάντα. Μετά το πρώτο <dir> υπάρχουν τα δεδομένα που θα επεξεργαστεί η μηχανή. Τα Sprites μπαίνουν σε νοητό δένδρο όπως τα φτιάχναμε με φακέλους στο εργαλείο. Όταν η μηχανή τελειώσει την επεξεργασία, το Spritesheet θα έχει τα παρακάτω Sprites με ονόματα.

- “/StandUp”
- “/UP/0”
- “/UP/1”
- κ.ο.κ

Εφόσον υπάρχει το DarkFunction Editor δεν χρειάζεται ποτέ να φτιάξουμε αυτά τα αρχεία χειροκίνητα, καθώς τα κάνουμε εύκολα και χωρίς σφάλματα από εκεί. Απλά χρειάζεται να τεκμηριωθεί επειδή πρόκειται για δομή δεδομένων της μηχανής.

c) Τα XML αρχεία Animations

Τα αρχεία *.anim είναι XML αρχεία που ορίζουν ομάδες από Animation. Στην μηχανή, κάθε αρχείο αποτελεί ένα “πακέτο” με animation (AnimationPack). Από τα πακέτα αυτά, το AnimationClass αντλεί τα Animations που θα χρησιμοποιήσει. Ας δούμε όμως πως δομούνται.

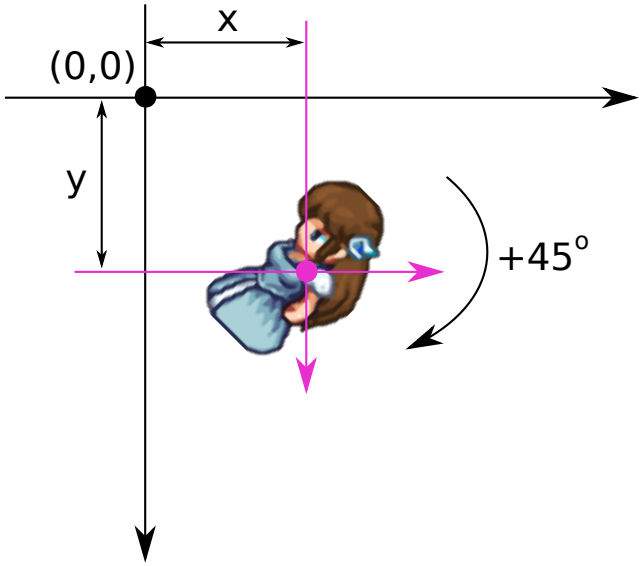


Εικόνα 3.2.9: Ορισμός του *.anim XML

Το δένδρο του αρχείου έχει ως εξής:

- **<animations> (Μοναδικό)** Το πρώτο στοιχείο. Έχει τα εξής
  - **spriteSheet (απαραίτητο):** Η διαδρομή του αρχείου *.sprites που θα χρησιμοποιηθεί.
  - **ver (απαραίτητο):** Έκδοση του Dark Function Editor που το έφτιαξε.
  - **<anim> (1 και πάνω)** Ορίζει ένα Animation.
    - **name (απαραίτητο):** Το όνομα του Animation. Αυτό το όνομα θα χρησιμοποιείται για να γίνει αναφορά σε αυτό το Animation.

- **loops (απαραίτητο):** 1 = Επαναλαμβάνεται, 0 = Δεν επαναλαμβάνεται (το Animation). Στην περίπτωση που επαναλαμβάνεται, όταν εμφανιστούν όλα τα Frames (Cells) του animation, ξεκινάει από την αρχή.
- **<cell> (1 και πάνω)** Ορίζει ένα Frame του Animation
  - **index (απαραίτητο):** Ορίζει την σειρά του Cell. Η σειρά πάει αύξουσα από το 0.
  - **delay (απαραίτητο):** Ορίζει την χρονική διάρκεια του Cell σε centiseconds.
  - **<spr> (1 και πάνω)** Ορίζει ένα Sprite στο Frame
    - **name (απαραίτητο):** Το όνομα του Sprite που θα χρησιμοποιηθεί. Αυτό το όνομα πρέπει να υπάρχει στο Spritesheet που συνδέσαμε στην αρχή του αρχείου. Έχει την μορφή διαδρομής πχ: “/UP/0”.
    - **x, y (απαραίτητο):** Είναι οι συντεταγμένες Offset που θα εμφανιστεί το Sprite σε σχέση με το κέντρο του Lifeform. Οι συντεταγμένες έχουν το σύστημα αξόνων όπως της οθόνης, δηλαδή στον άξονα του x δεξιά είναι τα θετικά και στον άξονα του y κάτω είναι τα θετικά.
    - **z (απαραίτητο):** Πρόκειται για το z-index. Αυτό είναι ένας αριθμός που καθορίζει τη σειρά που θα σχεδιαστεί το Sprite σε σχέση με τα άλλα Sprites που έχει το Cell. Sprites με μεγαλύτερες τιμές θα εμφανιστούν μπροστά από Sprites με μικρότερες.
    - **angle (προαιρετικό):** Είναι η γωνία που θα σχεδιαστεί το Sprite. Το Sprite σχεδιάζεται στο κέντρο της εικόνας του, σε απόσταση (x,y) από το κέντρο του Lifeform και με γωνία “angle” σε σχέση με το κέντρο του Sprite. Οι γωνίες είναι σε μοίρες και μπορούν να έχουν δεκαδικά ψηφία. Η περιστροφή γίνεται δεξιόστροφα για θετικές τιμές και αριστερόστροφα για αρνητικές. Υπόψιν ότι οι τιμές μπορούν να ξεπερνάνε τις 360 μοίρες απλά το αποτέλεσμα θα είναι ενδιάμεσα 0-360 μοίρες. Στην παρακάτω εικόνα δείχνεται ένα παράδειγμα εφαρμογής της γωνίας καθώς και των Offset. Η προκαθορισμένη γωνία αν δεν οριστεί είναι 0 μοίρες.



Εικόνα 3.2.10: Παράδειγμα εφαρμογής Angle

- **flipH, flipV (προαιρετικό):** Αυτές οι τιμές ορίζουν εάν το Sprite θα αντιστραφεί οριζοντίως (flipH) ή καθέτως (flipV). Για τιμές “1” θα εφαρμοστεί η αντιστροφή. Για τιμές “0” και οτιδήποτε άλλο, θα μείνει ως έχει.
-

Αρχικό Sprite	flipH = 1
	
flipV = 1	flipH = 1 flipV = 1
	

Εικόνα 3.2.11: Πίνακας αποτελεσμάτων των Flip Flags

Με το παραπάνω τελειώνει η περιγραφή των *.anim XML Files. Περιεκτικά: κάθε αρχείο αντιμετωπίζεται από την μηχανή σαν πακέτο από Animations. Κάθε Animation έχει πολλά Cells (Frames) που το καθένα μπορεί να έχει πολλά Sprites.

Πίνακας 3.2.2: Παράδειγμα αρχείου Animations

```
<?xml version="1.0"?>
<!-- Generated by darkFunction Editor (www.darkfunction.com) -->
<animations spriteSheet="Female_NPC_3.sprites" ver="1.2">
  <anim name="Walk_Down" loops="1">
    <cell index="0" delay="10">
      <spr name="/Walk_Down/1" x="0" y="-30" z="0"/>
    </cell>
    <cell index="1" delay="10">
      <spr name="/Walk_Down/2" x="1" y="-30" z="0"/>
    </cell>
    <cell index="2" delay="10">
      <spr name="/Walk_Down/3" x="0" y="-30" z="0"/>
    </cell>
    <cell index="3" delay="10">
      <spr name="/Walk_Down/4" x="0" y="-30" z="0"/>
    </cell>
    <cell index="4" delay="10">
      <spr name="/Walk_Down/5" x="0" y="-30" z="0"/>
    </cell>
    <cell index="5" delay="10">
      <spr name="/Walk_Down/6" x="-1" y="-30" z="0"/>
    </cell>
    <cell index="6" delay="10">
      <spr name="/Walk_Down/7" x="-1" y="-30" z="0"/>
    </cell>
    <cell index="7" delay="10">
      <spr name="/Walk_Down/8" x="0" y="-30" z="0"/>
    </cell>
  </anim>
</animations>
```

- **Actions**

Τώρα πλέον αφού φτιάξαμε τα Animations της Μαριάννας, πρέπει να ορίσουμε ποιά Animation πρέπει να εφαρμόζονται σε κάποια Actions.

Όπως είχαμε προαναφέρει, τα Actions είναι κάποιες ομάδες από Animations που ορίζουν Animation για κάθε κατεύθυνση. Μια ομάδα πχ μπορεί να έχει τα Animation για να εμφανίζει κάποιου είδους χτυπήματος που κάνει ο χαρακτήρας. Αυτό βοηθάει όταν θα θέλουμε να αλλάξουμε Action στον χαρακτήρα (πχ από εκεί που κινείται να χτυπήσει κάτι), να μην βαλθούμε να βρούμε εμείς ποιο Animation να εφαρμοστεί ανάλογα με την κατεύθυνση άλλα να το κάνει η μηχανή με τα δεδομένα που της δί-

νουμε μέσω των Action αυτών. Έτσι για παράδειγμα όταν θα θέλουμε να αλλάξουμε το Animation σε ένα Lifeform μέσω της Lua κάνουμε:

Πίνακας 3.2.3: Παράδειγμα αλλαγής Animation μέσω της Lua

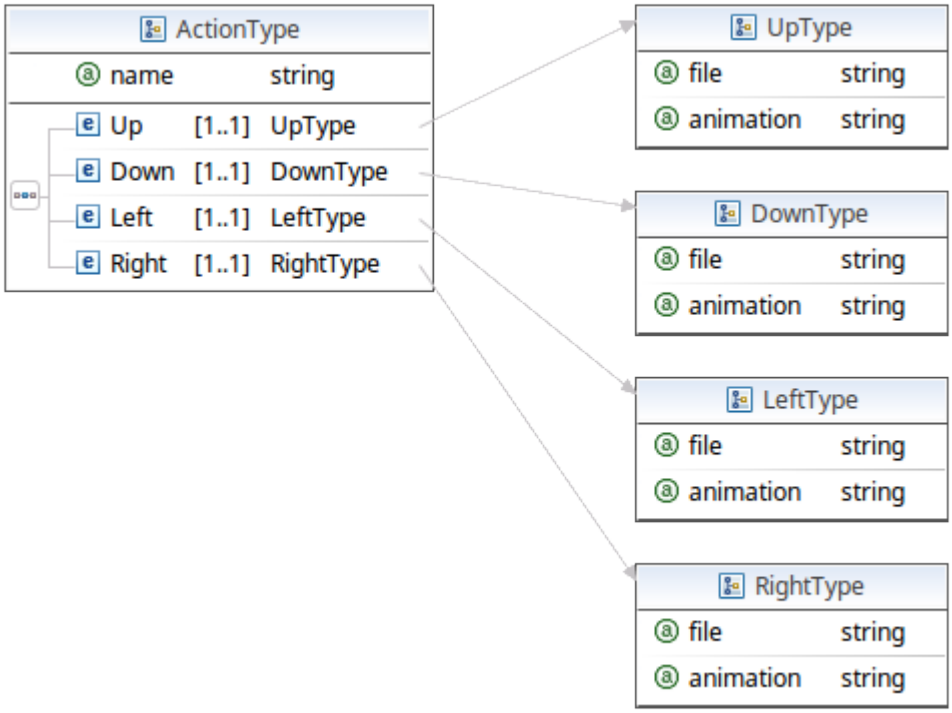
```
local Marianna = Zeta.WorldManager:getPlayer()
Marianna:getAnimationHandler():setAnimation("Slash")
```

Με την προϋπόθεση ότι έχουμε ορίσει ένα Action με όνομα “Slash” στο Animation Class που έχει η Μαριάννα, τότε ο παραπάνω κώδικας θα πάει και αυτομάτως θα βάλει το κατάλληλο Animation “Slash” που χρειάζεται για την κατεύθυνση που κοιτάει η Μαριάννα εκείνη την στιγμή. Αξίζει να σημειωθεί ότι αν δεν υπάρχει το Action που ζητηθεί, θα μπει στην θέση του το καθολικό “Idle_Down” ή ότι άλλο οριστεί σαν Fallback Animation το οποίο δεν είναι τίποτα άλλο από κάποιο καθολικό Animation από καθολικό Action.

Τι είναι όμως το καθολικό Action; Είναι κάποια Actions που τα έχουν όλα τα Lifeform ανεξαρτήτου Class. Τα Actions αυτά υπάρχουν στο καθένα αλλά αλλάζουν τα Animation που έχουν μέσα ανάλογα με το τι έχουμε βάλει εμείς μέσω του Animation Class. Τα καθολικά Actions είναι τα παρακάτω:

- **“__Movement__”** Που έχει τα 4 Animations για την κίνηση
- **“__Idle__”** Που έχει τα 4 Animations για την Στασιμότητα
- **“__Death__”** Που έχει τα 4 Animations για την διαδικασία θανάτου
- **“__Dead__”** Που έχει τα 4 Animations για νεκρή στάση

Αυτά τα ονόματα από Actions αρχίζουν και τελειώνουν με 2 κάτω παύλες για να διαφέρουν από τα άλλα μη καθολικά Actions. Αυτά τα Actions εναλλάσσονται μεταξύ τους σε κάποιο Lifeform όποτε κρίνει η μηχανή απαραίτητο. Πχ, όταν κινηθεί κάποιο Lifeform, δεν χρειάζεται να του πούμε εμείς να εφαρμόσει το “__Movement__” στο Lifeform. Θα το κάνει μόνη της η μηχανή. Το ίδιο ισχύει και για την στασιμότητα, τον θάνατο και όταν είναι νεκρό.



Εικόνα 3.2.12: Ορισμός του Action

Ας πάμε να ορίσουμε ένα βασικό Action της Μαριάννας, την στασιμότητα. Όπως είπαμε η στασιμότητα πρέπει να έχει το προκαθορισμένο όνομα “__Idle__”. Έτσι:

<Action>

- **name (απαραίτητο):** Πρέπει να έχει το όνομα του Action. Με αυτό το όνομα θα αναφερόμαστε σε αυτό.
- **<Up> <Down> <Left> <Right> (απαραίτητα, με την σωστή σειρά):** Το καθένα από αυτά συνδέει ένα Animation από ένα Animation Pack (*.anim) για την κατεύθυνση που περιγράφουν.
  - **file (απαραίτητο):** Έχει την διαδρομή του αρχείου (*.anim) που έχει το Animation που θέλουμε.
  - **animation (απαραίτητο):** Το όνομα του animation που θέλουμε να συνδέσουμε από το αρχείο που δείχνει το “file”.

Με τα παραπάνω μπορούμε να φτιάξουμε το Action που θέλουμε.

Πίνακας 3.2.4: Παράδειγμα ορισμού Action

```
<Action name="__Idle__">
    <Up file="Demo/Marianna.anim" animation="Stand_Up" />
    <Down file="Demo/Marianna.anim" animation="Stand_Down" />
    <Left file="Demo/Marianna.anim" animation="Stand_Left" />
    <Right file="Demo/Marianna.anim" animation="Stand_Right" />
```

```
</Action>
```

Για την στάση προς τα πάνω έχουμε φτιάξει το Animation “Stand_Up” που είναι στο αρχείο “Demo/Marianna.anim”. Με αυτές τις πληροφορίες γεμίζουμε τα πεδία όπως παραπάνω. Με τον ίδιο τρόπο μπορούμε να φτιάξουμε και την κίνηση “__Movement__”.

Όπως βλέπουμε, τα Animation που ζητάμε βρίσκονται στο ίδιο αρχείο. Είναι καλό αυτό γιατί αποφεύγεται η φόρτωση πολλών αρχείων από την μηχανή και γενικά είναι καλή εξάσκηση να δημιουργείται σε ένα “*.anim” όλα τα Animations ενός AnimationClass. Βέβαια αυτό σημαίνει ότι δεν μας εμποδίζει κανείς να φέρουμε Animation από άλλο αρχείο. Για αυτό τον λόγο, για κάθε κατεύθυνση ενός Action μας δίνεται αυτή η επιλογή.

Τα μη καθολικά Actions είναι για χρήσεις του προγραμματιστή του παιχνιδιού. Η μηχανή δεν θα αλλάξει σε αυτά τα Actions από μόνη της εκτός από ορισμένες εξαιρέσεις όπως τα Abilities. Δεν υπάρχει όριο στον αριθμό των Actions που μπορεί να έχει ένα Animation Class.

- **Bounding**

Μέχρι στιγμής έχουμε ένα Lifeform που μπορεί να υπάρξει κανονικά μέσα στο παιχνίδι και να κινηθεί όπως θέλουμε. Όμως έτσι όπως είναι δεν θα μπορεί να συγκρουστεί με τίποτα στο κόσμο. Για αυτό και πρέπει να του ορίσουμε το Bounding Rectangle (Ορθογώνιο Σύγκρουσης).

Το Bounding Rectangle είναι ένα νοητό ορθογώνιο πάνω στο Lifeform (ή και εκτός), που ελέγχεται σε κάθε κίνηση για σύγκρουση με τα άλλα ορθογώνια στον κόσμο. Εάν συγκρουστεί με κάποιο άλλο κατά την κίνηση, τότε η κίνηση θα μεταβληθεί ώστε στα 2 ορθογώνια να μην συμπίπτουν οι περιοχές τους ποτέ. Έτσι δημιουργούμε συγκρούσεις στον κόσμο.

Εάν δεν υπήρχαν αυτά, τότε ο οι χαρακτήρες θα περνούσαν ο ένας μέσα από τον άλλον και μέσα από άψυχα αντικείμενα όπως ένα δένδρο. Τα ορθογώνια αυτά μας βοηθάνε να αποτραπούν αυτά τα συμβάντα.

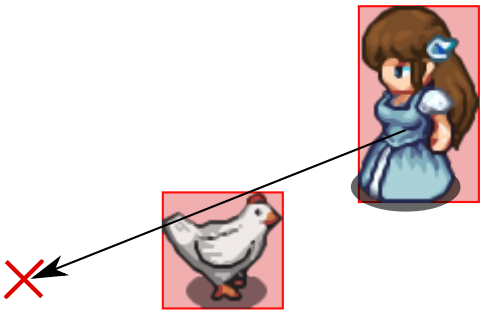
Δεν μπορεί κάθε AnimationClass να έχει ίδιο μέγεθος και τοποθεσία του Bounding Rectangle με άλλο Animation Class. Πρέπει να ταιριάζει με το τι εμφανίζει το Animation Class. Για παράδειγμα, ένα Τέρας με μέγεθος Sprite 200x300 pixel, το ορθογώνιο του δεν μπορεί να είναι μεγέθους αντίστοιχου του ενός μικρού ανθρώπου

μεγέθους Sprite 36x96 pixel. Αυτό γιατί στις συγκρούσεις θα συμπεριφέρεται σαν κάτι μικρό ενώ στην οθόνη θα φαίνεται τεράστιο. Για παράδειγμα, το τέρας του παραδείγματος με το Bounding Rectangle ενός ανθρώπου, θα έχει μέγεθος όσο ένα φορτηγό στην οθόνη. Όταν πάει να περάσει ανάμεσα από δυο κτήρια που μόνο ένας άνθρωπος χωράει, το τέρας παρόλο που δεν φαίνεται να χωράει στο κενό, θα περάσει γιατί το ορθογώνιο του χωράει να περάσει. Έτσι βλέπουμε ένα θηρίο να περνάει το στενό δρομάκι. Για αυτό και πρέπει να προσαρμόζουμε το κάθε ορθογώνιο στις ανάγκες του Animation. Αυτό το κάνει το στοιχείο Bounding του Animation Class.

Ο ορισμός του Bounding Rectangle γίνεται αντίστοιχα με Offsets όπως μετά Sprites. Το κέντρο του Lifeform θεωρείται το (0,0). Από εκεί με βάση το κύριο Animation προσαρμόζουμε το μέγεθος και την τοποθεσία που θέλουμε στο Bounding Rectangle. Ας το δούμε με το παράδειγμα μας.

Τα Sprites στο Demo φαίνονται σαν να είναι όρθια. Έτσι δεν θα ήταν συνετό να βάλουμε το Bounding Rectangle να καλύπτει όλο το Sprite. Στην ουσία θέλουμε τα πόδια του χαρακτήρα να συγκρούονται με αντικείμενα στο κόσμο και όχι όλος ο χαρακτήρας. Αυτό γιατί δεν θα μπορούμε να εμφανίσουμε κάποιο χαρακτήρα όταν βρίσκεται από “πίσω” γιατί τα ορθογώνια τους θα συγκρούονται και δεν θα αφήσει η μηχανή ο χαρακτήρας να κινηθεί από πίσω από κάτι άλλο.

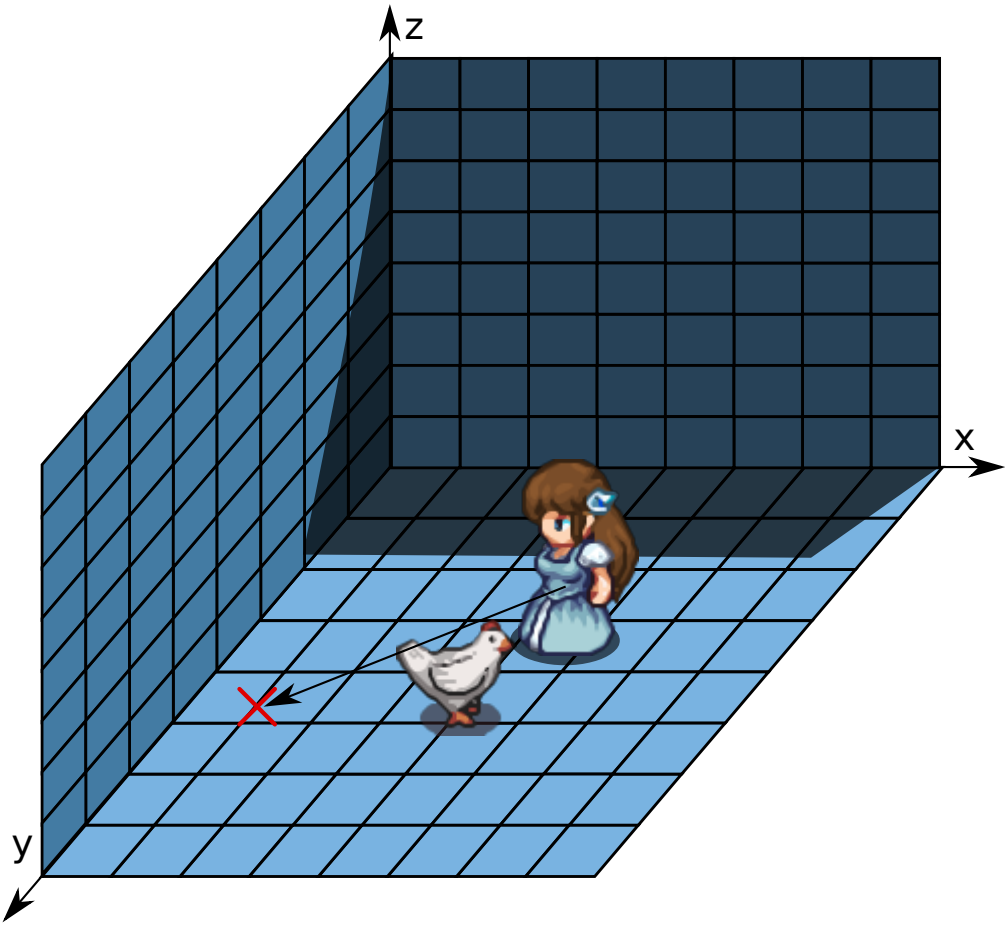
Για παράδειγμα έστω ότι έχουμε φτιάξει την κότα και την Μαριάννα με τα ορθογώνια να καλύπτουν πλήρως τα Sprites τους. Έστω η Μαριάννα βρίσκεται σε ένα σημείο και θέλει να κινηθεί όπως στην παρακάτω εικόνα.



Εικόνα 3.2.13: Παράδειγμα με πλήρης Bounding Rectangles

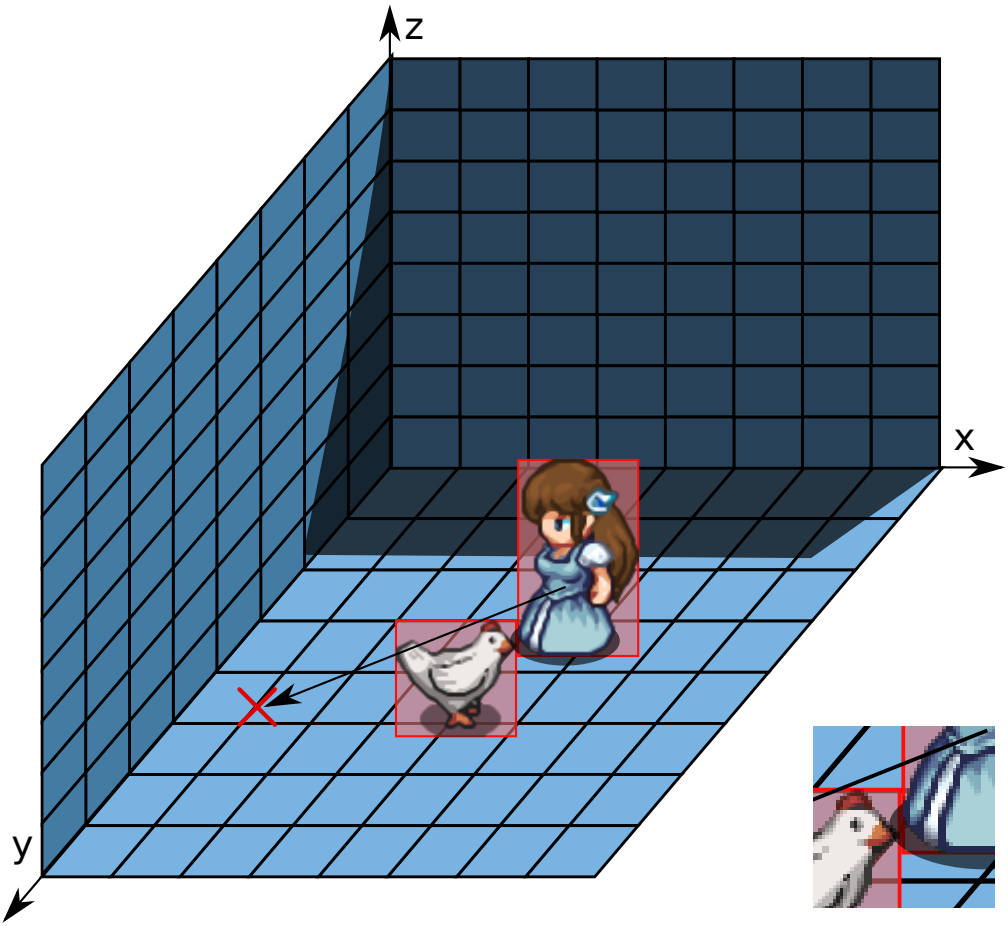
Μπορεί στην εικόνα να μην φαίνεται, αλλά στο παιχνίδι θα φαίνονται σαν να είναι όρθιοι και η κότα και η Μαριάννα. Στην ουσία είναι ψευδό-τρισδιάστατα τα γραφικά

αυτά. Θα το νιώσουμε πιο εύκολα με την όραση μας στην παρακάτω εικόνα όπου μεταφέρουμε την παραπάνω σκηνή σε τρισδιάστατο επίπεδο.



Εικόνα 3.2.14: Μεταφορά της σχεδίασης 3.2.13 σε 3D Επίπεδο

Όπως βλέπουμε στην παραπάνω σκηνή, η Μαριάννα προσπάθησε να πάει στο Χ αλλά για κάποιο περίεργο λόγο ενώ προχωρούσε κανονικά, σταμάτησε εκεί που βρίσκεται τώρα (στην πραγματικότητα δεν θα γίνει ακριβώς αυτό άλλα για χάρη του παραδείγματος θα σταματήσει). Όπως καταλαβαίνουμε, η Μαριάννα θα έπρεπε να κινηθεί κανονικά και να περάσει πίσω από την κότα καταλήγοντας στο Χ. Ο λόγος φαίνεται στην παρακάτω εικόνα που επανεμφανίζουμε τα ορθογώνια συγκρούσεων.

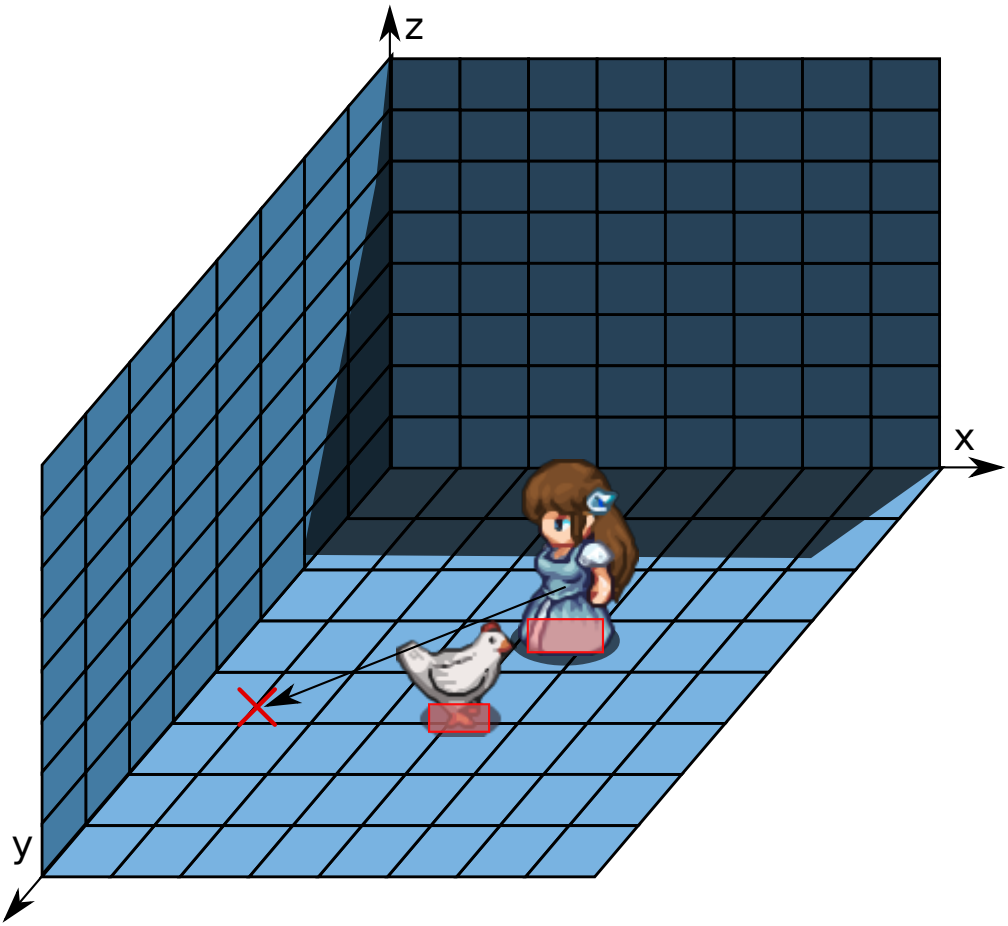


Εικόνα 3.2.15: Η Σχεδίαση 3.2.14 με εμφανή τα Ορθογώνια Συγκρούσεων

Παραπάνω βλέπουμε ότι τα ορθογώνια συγκρούσεων ακουμπάνε. Η μηχανή σε καμιά περίπτωση δεν θα αφήσει να περάσει το ένα μέσα στο άλλο. Έτσι η κίνηση διακόπτεται. Αυτό δεν είναι καθόλου ωραίο και ο χρήστης του παιχνιδιού θα προβληματιστεί με την παραλογή συμπεριφορά (επειδή κάτι “αόρατο” να αποτρέπει την κίνηση ενώ ευθύνεται η κότα). Πώς θα περάσει η Μαριάννα πίσω από την κότα και να γίνει σωστή συμπεριφορά των συγκρούσεων;

Στην μηχανή δεν υπάρχει η διάσταση Z, που σημαίνει ότι ελέγχει μόνο το (x,y) για συγκρούσεις. Στην περίπτωσή μας το (x,y) είναι το πάτωμα όπου οι χαρακτήρες βρίσκονται όρθιοι. Τι βρίσκεται σε επαφή με το πάτωμα; Τα πόδια των χαρακτήρων και αυτό πρέπει να ορίσουμε σαν περιοχή ελέγχου συγκρούσεων του χαρακτήρα ή αλλιώς το Bounding Rectangle.

Φτιάχνοντας με αυτή την λογική τα Bounding Rectangles, παίρνουμε το παρακάτω αποτέλεσμα.

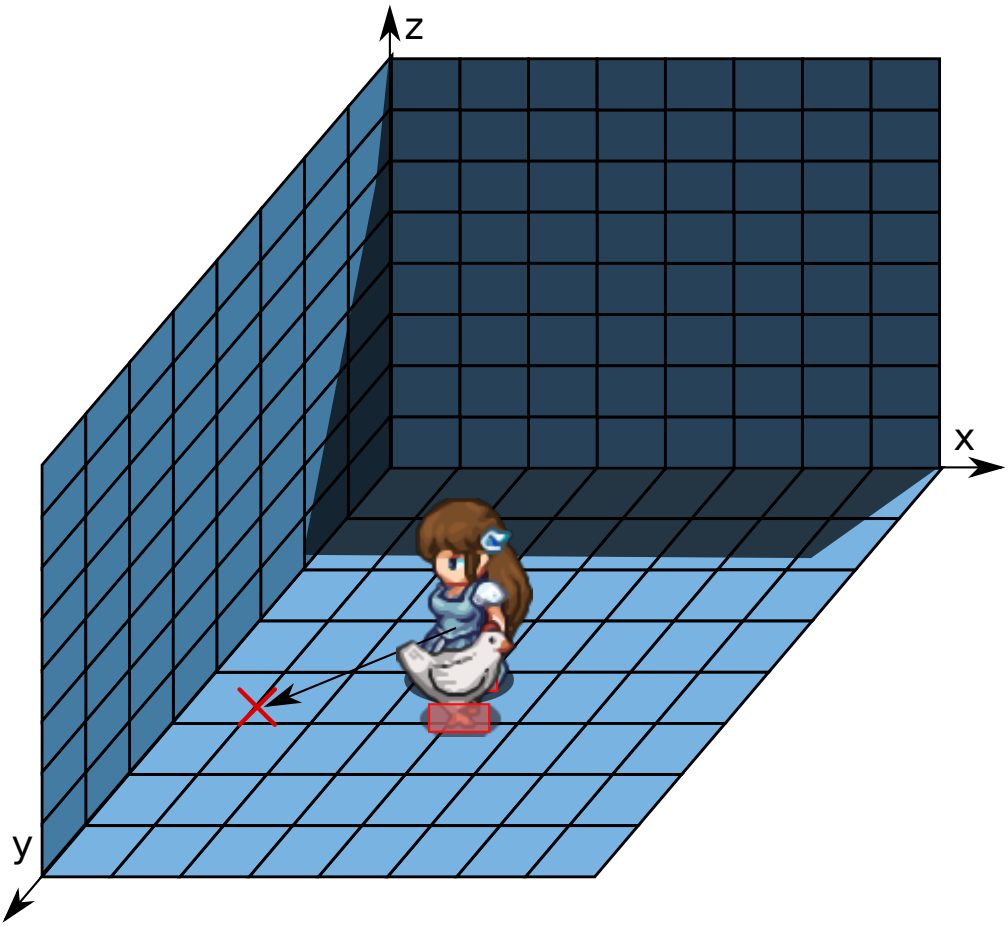


Εικόνα 3.2.16: Διορθωμένα Ορθογώνια Συγκρούσεων της Σχεδίασης 3.2.15

Τοποθετήσαμε και μικρύναμε τα ορθογώνια ώστε να καλύπτουν την περιοχή των ποδιών του κάθε χαρακτήρα. Προσέξτε ότι το ορθογώνιο της κότας είναι μικρότερο από της Μαριάννας γιατί διαφέρουν σε μέγεθος οι χαρακτήρες. Αυτό γιατί όπως είπαμε πρέπει παρόλο αυτά να τα προσαρμόζουμε και στην σιλουέτα του χαρακτήρα.

Τα ποδαράκια της κότας είναι πολύ μικρά αλλά δεν μπορούμε να της βάλουμε τόσο μικρό Ορθογώνιο. Αυτό γιατί όταν θα συγκρούεται πχ με την Μαριάννα, τότε θα μπαίνει υπερβολικά μέσα στο Sprite της κότας κάτι που δεν είναι επιθυμητό. Για αυτό και το κάνουμε λίγο μακρύτερο, όσο περίπου το μήκος της κότας. Έτσι οι συγκρούσεις θα είναι εντάξει και δεν θα έχουμε περίεργες παρενέργειες με τα Sprites.

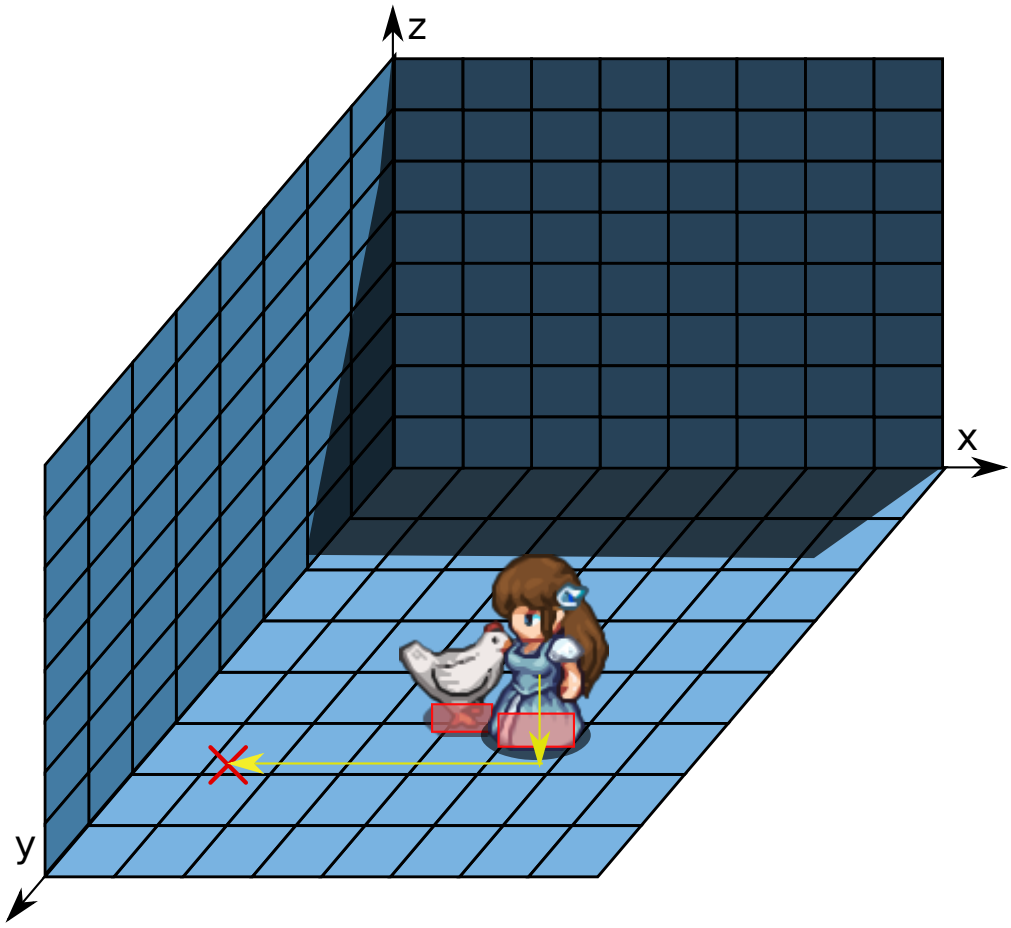
Με τα παραπάνω Ορθογώνια, το αποτέλεσμα της κίνησης θα είναι σωστό όπως βλέπουμε παρακάτω.



Εικόνα 3.2.17: Αποτέλεσμα χρήσης σωστών Ορθογωνίων Συγκρούσεων

Όπως βλέπουμε, η Μαριάννα κινήθηκε πίσω από την κότα όπως έπρεπε και στο τέλος θα καταλήξει στο X. Η μηχανή φροντίζει να εμφανιστεί πίσω από την κότα για αυτό δεν ασχολούμαστε εμείς.

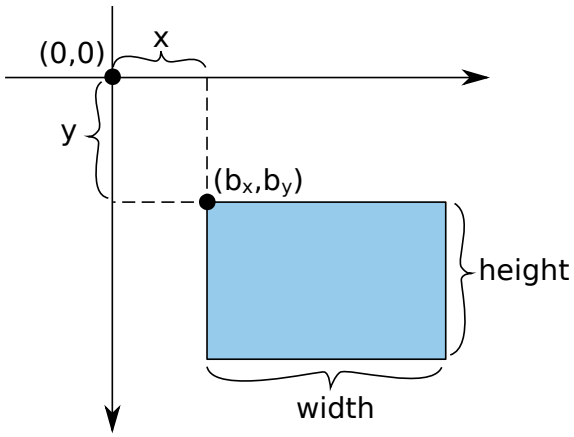
Παρόλο τα σωστά ορθογώνια, εάν η κλήση της κίνησης είναι μικρή η σύγκρουση είναι αναπόφευκτη. Όμως η μηχανή παρέχει μηχανισμό “Ψεύδο-Pathfinding” για μικρές αποστάσεις. Αυτό σημαίνει ότι κατά την σύγκρουση δεν θα σταματήσει η Μαριάννα, αλλά θα κινηθεί στο μήκος του κάθετου άξονα μέχρι να βρεθεί στο ύψος του X. Εάν κατά την κίνηση αυτή η σύγκρουση πάψει να υπάρχει (δηλαδή πάει αρκετά κάτω από τη κότα ώστε να μην συγκρούονται τα ορθογώνια), θα κινηθεί πάλι ελεύθερα προς στο X. Εάν η σύγκρουση συνεχίζει να υπάρχει όταν φτάσει στο ύψος του X, τότε σταματάει τελείως. Αυτό γίνεται και αντίστροφα όταν κινείται κάτω, με την διαφορά ότι τότε κινείται πάνω στον άξονα του x μέχρι να ελευθερωθεί ή να σταματήσει.



Εικόνα 3.2.18: Παράδειγμα Ψευτό- Pathfinding

Αφού κατανοήσαμε το πώς πρέπει να φτιάξουμε τα ορθογώνια, τώρα πρέπει να δούμε πώς θα τα τοποθετήσουμε πάνω στο Sprite.

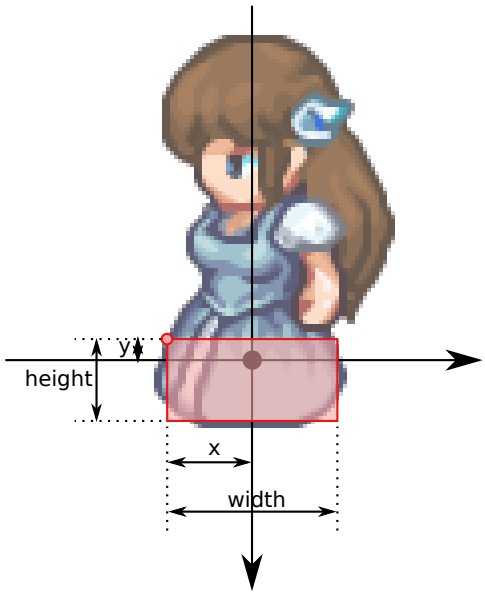
Όπως είπαμε, η τοποθέτηση γίνεται αντίστοιχα με αυτή των Sprites, δηλαδή με Offsets. Το κέντρο του Lifeform είναι το σημείο (0,0). Από εκεί μετράμε pixels που θα απέχει η πάνω αριστερά γωνία του Ορθογωνίου.



Εικόνα 3.2.19: Πρότυπο τοποθέτησης των Ορθογωνίων

Με την παραπάνω εικόνα, μπορούμε να τοποθετήσουμε το ορθογώνιο στην Μαριάννα. Πρέπει να ξέρουμε στο περίπου που θα είναι το κέντρο του Lifeform στο Sprite της. Αυτό μπορούμε να το δούμε όταν επεξεργαζόμαστε τα Animations της στο Dark Function Editor.

Για να το καταλάβουμε λίγο καλύτερα, ας δούμε την παρακάτω εικόνα.



Εικόνα 3.2.20: Παράδειγμα τοποθέτησης ορθογωνίου

Βλέπουμε πώς θα βάλουμε το ορθογώνιο στην Μαριάννα. Με οποιοδήποτε τρόπο επιλέξουμε να το υπολογίσουμε, βρίσκουμε τις επιθυμητές τιμές που είναι πχ

- x = -19
- y = -7
- width = 38
- height = 30

Τώρα με αυτές τις πληροφορίες μπορούμε να πάμε να συμπληρώσουμε την γραμμή “Bounding” από το αρχείο του AnimationClass.

Πίνακας 3.2.5: Παράδειγμα της γραμμής Bounding του AnimationClass

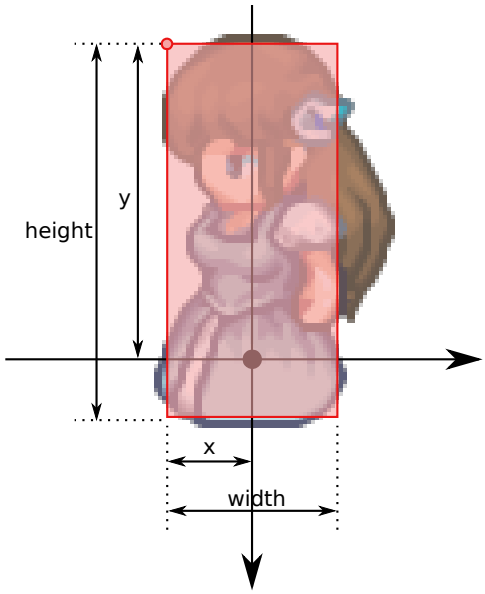
```
<Bounding>
    <Rectangle x="-19" y="-7" width="38" height="30" />
</Bounding>
```

Όπως ορίζει το XSD Schema του AnimationClass, πρέπει να υπάρχει το <Bounding> με μια εσωτερική <Rectangle> με πεδία που έχουν τα δεδομένα που θέλουμε.

- **Target Area**

Το TargetArea είναι και αυτό ένα ορθογώνιο σαν το Bounding, μόνο που χρησιμοποιείται για άλλους σκοπούς. Αυτό το ορθογώνιο δεν “συγκρούεται” με τα άλλα ορθογώνια αλλά χρησιμοποιείται για να ξέρει η μηχανή την “Ενεργό” περιοχή του Lifeform. Η μηχανή πχ, όταν θέλει να ελέγξει αν υπάρχει ένα Lifeform σε ένα σημείο, δεν θα ελέγξει αν υπάρχει κάποιο pixel του Sprite του σε αυτό το σημείο αλλά την ενεργό περιοχή. Για αυτό συνήθως αυτό το ορθογώνιο καλύπτει σχεδόν όλη την φιγούρα του Lifeform ή τουλάχιστον όσο χρειάζεται. Στο Demo, αυτή η περιοχή χρησιμοποιείται για να επιλέγουμε Lifeforms (Target).

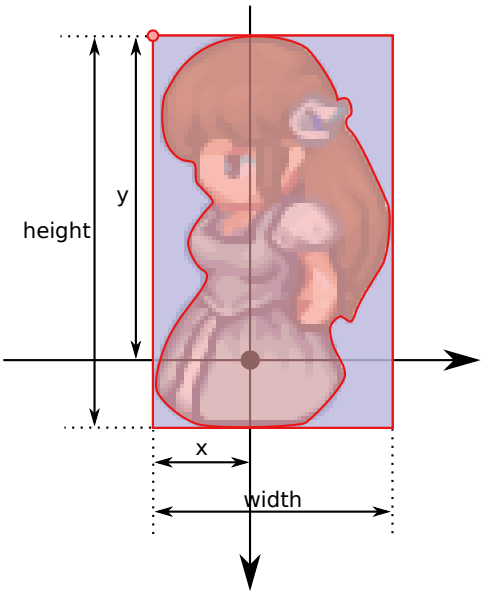
Εφόσον δεν αλληλοεπιδρά με τίποτα άλλο στον κόσμο, δεν χρειάζεται να είμαστε ιδιαίτερα προσεχτικοί για το που το τοποθετούμε πάνω στο Lifeform. Έτσι στο παράδειγμα μας που θα το χρησιμοποιήσουμε σαν επιλογέα, θα το φτιάξουμε κάπως έτσι:



Εικόνα 3.2.21: Παράδειγμα τοποθέτησης TargetArea

Επειδή θέλουμε όταν πατήσουμε πάνω στη Μαριάννα να σηκώσει η μηχανή συμβάν, τότε χρειάζεται να την καλύψουμε όλη με αυτό και έτσι φτιάχνουμε το ορθογώνιο μεγάλο.

Βλέπουμε ότι το ορθογώνιο δεν καλύπτει τελείως κάθε Pixel της. Μεγάλο μέρος από τα μαλλιά της είναι εκτός όπως και διάφορα μικρά σημεία. Εδώ έχει γίνει ένας συμβιβασμός για να έχουμε σωστή συμπεριφορά και όσο πιο δυνατή κάλυψη του χαρακτήρα. Ας δούμε παρακάτω το ορθογώνιο αν προσπαθούσαμε να καλύψουμε όλα τα Pixel της Μαριάννας.



Εικόνα 3.2.22: Παράδειγμα τοποθέτησης TargetArea με πλήρης κάλυψη

Το ορθογώνιο έχει αυξηθεί τόσο σε πλάτος όσο και ύψος. Τώρα καλύπτει κάθε Pixel της Μαριάννας και αν η μηχανή ελέγξει οπουδήποτε πάνω της (στο Demo το έχουμε ρυθμίσει όταν κάνουμε κλικ στον κόσμο να ελέγχεται αν υπάρχει Lifeform εκεί), θα μας δώσει την Μαριάννα. Όμως μαζί με την Μαριάννα καλύψαμε και πολύ κενό χώρο γύρω της όπως βλέπουμε με γαλάζιο φόντο πίσω από το ορθογώνιο. Αυτός ο κενός χώρος είναι ενεργός για την μηχανή και όταν γίνει κλικ εκεί, τότε θα μας επιστρέψει την Μαριάννα.

Εάν το κλικ πέσει κοντά στην Μαριάννα τότε δεν θα παρατηρηθεί τίποτα περίεργο. Αν όμως γίνει πχ, κοντά στην πάνω δεξιά γωνία, τότε ο παίχτης θα παρατηρήσει κάτι περίεργο καθώς δεν κλίκαρε την Μαριάννα άλλα το συμβάν συνέβη γιατί το σημείο που κλίκαρε καλύπτονταν από το ορθογώνιο. Για αυτό κρίνεται απαραίτητο να περιοριστεί ο κενός χώρος.

Ένας τρόπος είναι να περιορίσουμε το ορθογώνιο στον χώρο που θα κλικάρει ο χρήστης στις περισσότερες περιπτώσεις. Σε αυτή την περίπτωση είναι το σώμα της Μαριάννας. Έτσι μικραίνοντας το ορθογώνιο, καλύπτοντας μόνο τον κορμό, τα πόδια και το κεφάλι, έχουμε ένα ικανοποιητικό ποσοστό κάλυψης όπως στην εικόνα 3.2.21.

Τα μαλλιά στο πλάι της είναι ακάλυπτα άλλα η πιθανότητα να γίνει εκεί κλικ είναι μικρή. Υπάρχει ένας κενός χώρος αριστερά της δίπλα από το κεφάλι και το στήθος της, άλλα δεν μας πειράζει αυτό γιατί αν ένα κλικ επιστρέψει την Μαριάννα, ο χρήστης δεν θα παραξενευτεί ιδιαίτερα καθώς είναι πολύ κοντά στο σώμα της. Πέρα από αυτό, τα Sprites είναι μικρότερα από ότι φαίνονται εδώ και έτσι τα κενά είναι πιο απίθανα να χτυπηθούν από ότι φαίνεται.

Κάπως έτσι γίνονται τα TargetArea. Η φιλοσοφία όπως είπαμε είναι ότι αυτά δίνουν στην μηχανή μια ιδέα για το ουσιαστικό μέγεθος του Lifeform.

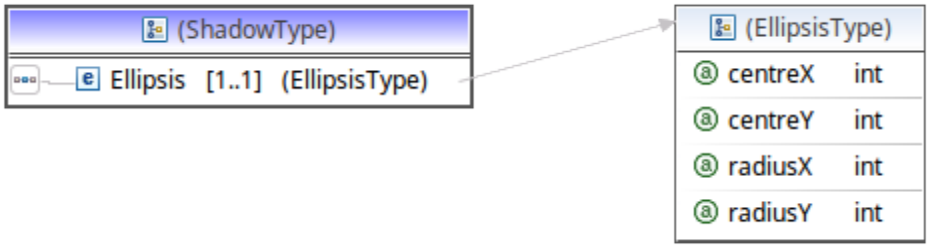
Στο AnimationClass το ορίζουμε όπως και το Bounding Rectangle.

Πίνακας 3.2.6: Παράδειγμα της γραμμής TargetArea του AnimationClass

```
<TargetArea>
    <Rectangle x="-24" y="-75" width="48" height="98" />
</TargetArea>
```

- **Shadow**

Το πεδίο Shadow είναι προαιρετικό και αν οριστεί δημιουργεί μια ελλειπτική σκιά σε σημείο και μέγεθος που ορίζονται από αυτό.



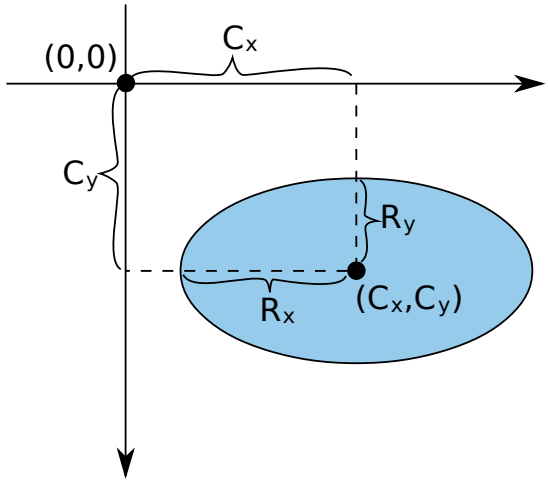
Εικόνα 3.2.23: Ορισμός του Shadow

Το πεδίο Shadow εφόσον οριστεί, πρέπει να έχει ένα πεδίο Ellipsis. Σε αυτό με την σειρά του πρέπει να έχει τις παρακάτω ιδιότητες.

- **centreX:** Η offset συντεταγμένη X του κέντρου της έλλειψης σε σχέση με το κέντρο του Lifeform.

- **centreY:** Η offset συντεταγμένη Y του κέντρου της έλλειψης σε σχέση με το κέντρο του Lifeform.
- **radiusX:** Το πλάτος της έλλειψης από το κέντρο.
- **radiusY:** Το ύψος της έλλειψης από το κέντρο.

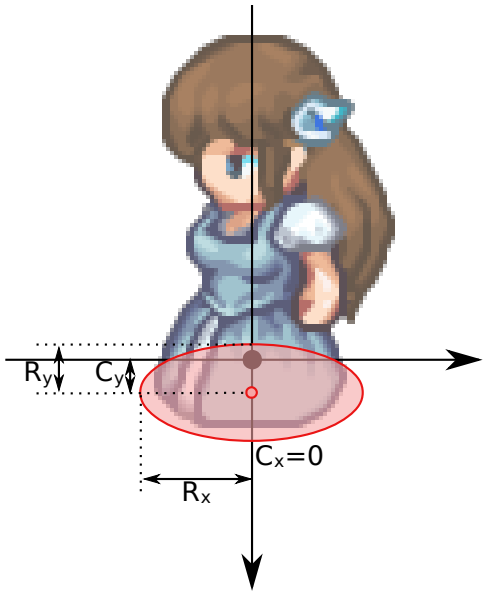
Για καταλάβουμε καλύτερα το πως λειτουργούν αυτές οι συντεταγμένες ας δούμε το παρακάτω σχήμα.



Εικόνα 3.2.24: Αναπαράσταση της έλλειψης στους άξονες

Όπως βλέπουμε, λειτουργεί περίπου όπως το Rectangle μόνο που από το κέντρο εφαρμόζονται το πλάτος και το ύψος.

Σαν σκιά πρέπει να βρίσκεται κάτω από το Lifeform άλλα όπως είπαμε το Lifeform φαίνεται σαν είναι όρθιο. Έτσι πρέπει να βάλουμε την σκιά κάπως έτσι:



Εικόνα 3.2.25: Παράδειγμα εφαρμογής του Shadow στην Μαριάννα

Η μηχανή φτιάχνει το σχήμα αυτό που της ορίζουμε και το εμφανίζει μαζί με το Lifeform σε κάθε Action. Πρέπει να σημειωθεί ότι η σκιά πάντα εμφανίζεται πίσω από το Sprite του Lifeform. Το χρώμα και η διαφάνεια είναι προς το παρών προδιαγεγραμμένα: RGBA = 0x000000BA (R=0, G=0, B=0, Alpha=186).

Το αποτέλεσμα της παραπάνω εφαρμογής θα είναι:

Εικόνα 3.2.26: Αποτέλεσμα εφαρμογής Shadow

Η γραμμή που θα μας δώσει το αποτέλεσμα αυτό είναι:

Πίνακας 3.2.7: Παράδειγμα ορισμού του Shadow

```
<Shadow>
     <Ellipsis centreX="0" centreY="10" radiusX="28" radiusY="15" />
</Shadow>
```

Αξίζει να σημειωθεί ότι το σχήμα αυτό δεν αντιδρά με κανένα άλλο αντικείμενο στον κόσμο, ακόμα και με το ίδιο το Lifeform που το έχει. Το μόνο που κάνει από μόνο του, είναι να βρίσκεται πάντα στο σωστό σημείο που του έχουμε ορίσει. Αυτό δεν σημαίνει ότι δεν μπορούμε εμείς να το επεξεργαστούμε κατά την εκτέλεση του παιχνιδιού.

3.3 Animation Effects

Η μηχανή δίνει την δυνατότητα για καθολικά Animations τα οποία μπορούν να χρησιμοποιηθούν ανά πάσα στιγμή από οτιδήποτε όσες φορές χρειαστεί. Αυτά λέγονται Animation Effects.

Τα Animations αυτά φτιάχνονται όπως όλα τα άλλα Animations, μόνο που δεν χρειάζονται AnimationClass για να εφαρμοστούν στην πράξη. Το μόνο που χρειάζεται ο προγραμματιστής να φτιάξει για αυτά είναι τα αρχεία τα Sprites (*.sprites) και τα αρχεία που ορίζουν τα Animations (*.anim). Ύστερα μέσω της Lua, φορτώνει τα αρχεία anim στο καθολικό Container, ονοματίζοντας το καθένα από αυτά για να μην υπάρχουν εμπλοκές(conflicts) μεταξύ των ονομάτων των Animation (Namespace). Αφού φορτωθούν, τότε μπορούν τα Animation αυτά να ζητηθούν ανά πάσα στιγμή και να ανατεθούν σε κάποιο Lifeform.

Όταν ανατίθεται ένα Animation FX σε ένα Lifeform, πρέπει να του ορίσουμε Offsets για το που θα εμφανιστεί το Animation σε σχέση με τις συντεταγμένες του Lifeform. Πέρα από αυτό, πρέπει να ορίσουμε αν θα φαίνεται μπροστά ή πίσω από το Sprite του Lifeform.

Από την στιγμή που θα αναθέσουμε ένα Animation FX σε ένα Lifeform, στο επόμενο Frame θα εμφανιστεί κανονικά εκεί που του είπαμε. Το Animation θα τρέξει κανονικά μέχρι να φτάσει στο τέλος του (τελευταίο Cell). Όταν σταματήσει θα πάψει να εμφανίζεται και θα αφαιρεθεί αυτομάτως από το Lifeform που το είχε. Στην περίπτωση που το Animation έχει το Flag “loop” (δηλαδή να επαναλαμβάνεται συνέχεια) τότε δεν θα σταματήσει ποτέ και θα εμφανίζεται συνέχεια. Για να το σταματήσουμε σε μια τέτοια περίπτωση, θα πρέπει να το ανακτήσουμε από το Lifeform που το έχει μέσω του ονόματος του και να του πούμε να σταματήσει. Αφού του το πούμε αυτό, τα υπόλοιπα γίνονται αυτόματα σαν να μην επαναλαμβανόταν.

Ας πάμε να δούμε τον κώδικα για την φόρτωση ενός Animation Pack (*.anim).

Πίνακας 3.3.1: Παράδειγμα φόρτωσης AnimationFX

```
local AnimationFX = Zeta.AnimationEffectsManager:getInstance()
AnimationFX:addAnimationPack("Demo/Sprites/FX1.anim","FX")
```

Αρχικά ανακαλούμε το μοναδικό αντικείμενο “AnimationEffectsManager” που είναι στην ουσία το Container που περιέχει όλα τα FX που θα φορτώσουμε. Ύστερα καλούμε την συνάρτηση “addAnimationPack()” για να προσθέσουμε το αρχείο που θέλουμε.

Η κλήση αυτή είναι εσωτερική της μηχανής από την μεριά της C++ και ορίζεται ως εξής.

Πίνακας 3.3.2: Ορισμός της κλήσης addAnimationPack()

```
addAnimationPack(path, name)
```

Τα ορίσματα έχουν ως εξής:

- **path (String)** = Η διαδρομή του αρχείου που περιέχει τα Animations (*.anim)
- **name (String)** = Το όνομα που να δοθεί σε αυτή την ομάδα Animations

Αφού ολοκληρωθεί η κλήση της, τότε τα Animations θα είναι έτοιμα να αποδοθούν σε Lifeforms. Υπόψιν ότι η κλήση φορτώνει αμέσως το αρχείο, που σημαίνει ότι αν είναι μεγάλο ίσως να παρατηρηθεί κάποια καθυστέρηση, για αυτό και πρέπει να αποφεύγεται να καλείται εν ώρα παιχνιδιού.

Εάν τώρα κάπου μέσα στο παιχνίδι θέλουμε να εμφανίσουμε το Animation “Hit” από την ομάδα “FX” που φορτώσαμε, αρκεί να τραβήξουμε το Animation με το όνομα του από τον Container “AnimationEffectsManager” και να το προσθέσουμε στο Lifeform που θέλουμε, σαν OffAnimation με την κλήση “addOffAnimation”, πχ:

Πίνακας 3.3.3: Παράδειγμα απόδοσης OffAnimation (AnimationFX)

```
local AnimCont = Zeta.AnimationEffectsManager:getInstance()
local Animation = AnimCont:getAnimationFX("Hit","FX")

Victim:addOffAnimation(Animation,0,0,Zeta.AnimationHandler.QueuePlace.Front)
```

Η κλήση “getAnimationFX” παίρνει 2 ορίσματα και επιστρέφει το Animation που ζητάμε. Η κλήση “addOffAnimation” είναι της κλάσης Lifeform και ορίζεται ως εξής:

Πίνακας 3.3.4: Ορισμός της addOffAnimation()

```
addOffAnimation(Animation, dx, dy, QueuePlace)
```

Όπου:

- **Animation (Zeta::Animation)** = Το Animation αντικείμενο που θέλουμε να εμφανίσουμε στο Lifeform
- **dx (float)** = Η απόσταση στον άξονα X από το κέντρο του Lifeform που θα εμφανιστεί το Animation
- **dx (float)** = Η απόσταση στον άξονα Y από το κέντρο του Lifeform που θα εμφανιστεί το Animation

- **QueuePlace (Enumeration)** = Αν θα εμφανιστεί μπροστά ή πίσω από το Lifeform. Παίρνει τις παρακάτω τιμές:
  - **Zeta.AnimationHandler.QueuePlace.Front** = Θα εμφανιστεί μπροστά από το Sprite του Lifeform
  - **Zeta.AnimationHandler.QueuePlace.Back** = Θα εμφανιστεί πίσω από το Sprite του Lifeform

Από την στιγμή που θα επιστρέψει η κλήση αυτή, στο επόμενο Frame το Lifeform που θα έχει αυτό το Animation θα το εμφανίσει μέχρι να τελειώσει ή να το σταματήσουμε εμείς.

Τα AnimationFX έχουν διάφορες χρήσεις. Δεν είναι συνδεδεμένα με τίποτα, για αυτό και μπορεί το ίδιο Animation να αποδοθεί σε πολλά Lifeform ταυτόχρονα χωρίς να επηρεάζει το ένα το άλλο. Η ανάθεση μπορεί να γίνει ανά πάσα στιγμή και μπορεί να θεωρείται ότι έχει μηδαμινό χρόνο καθυστέρησης σε αντίθεση με την φόρτωση που μπορεί να καθυστερήσει. Για αυτό και η φόρτωση θα πρέπει να γίνεται κατά τις “φορτώσεις” του παιχνιδιού, όπου ο χρήστης έτσι κι αλλιώς περιμένει.

Πρέπει να σημειωθεί ότι αν ζητηθεί από το AnimationEffectsManager μια ομάδα ή ένα Animation που δεν υπάρχει (ή δεν έχει φορτωθεί), τότε θα επιστρέψει το NullAnimation, που θα δείξει κατά πάσα πιθανότητα σκουπίδια προκειμένου να εντοπιστεί το πρόβλημα χωρίς να καταρρεύσει το παιχνίδι. Επίσης, αν δεν δοθεί όνομα στην ομάδα κατά την φόρτωση της, τότε παίρνει αυτόματα το όνομα της διαδρομής του αρχείου.

3.4 Καθορισμός Ομάδων αντιπαλότητας

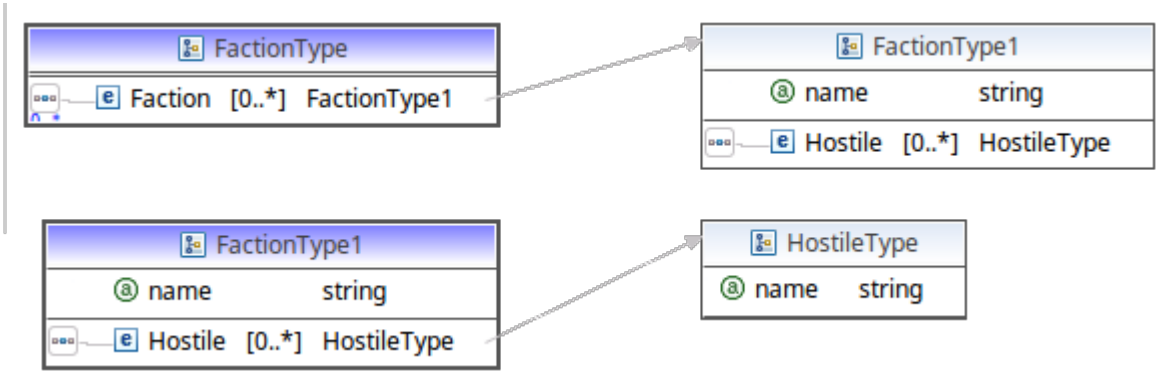
Στην μηχανή, πρέπει κάθε Lifeform να ανήκει σε μια ομάδα (Faction). Οι ομάδες αυτές καθορίζουν την συμπεριφορά τους με Lifeforms που ανήκουν σε άλλες ομάδες. Προς το παρόν οι σχέσεις που μπορούν να υπάρχουν είναι οι εξής:

- **Ουδέτερη:** Το Lifeform δεν επιτίθεται σε Lifeforms που είναι σε ομάδα που του είναι ουδέτερη.
- **Εχθρική:** Το Lifeform θα επιτεθεί σε Lifeforms που ανήκουν σε ομάδα που του είναι εχθρική.

Κάθε ομάδα χρειάζεται να ορίσει ρητά το πώς θα συμπεριφέρεται προς μια άλλη. Αν δεν οριστεί τότε γίνεται αυτομάτως ουδέτερη.

Επίσης πρέπει να ειπωθεί ότι αν η ομάδα A είναι εχθρική προς την ομάδα B, τότε η B δεν γίνεται αυτομάτως εχθρική προς την A. Ένα άτομο από την A θα επιτεθεί σε άτομο του B άλλα άτομο από την B δεν θα επιτεθεί στο A.

Ας πάμε να ορίσουμε κάποιες ομάδες. Οι ομάδες μπορούν να οριστούν όλες σε ένα αρχείο και κάθε Lifeform να ανατίθεται σε ομάδα μέσω του LifeformClass.



Εικόνα 3.4.1: Ορισμός του Factions XSD

Το αρχείο που ορίζει τις ομάδες ορίζεται ως:

<Factions>

- **<Faction>** : Ορίζει μια ομάδα.
  - **name (Απαραίτητο):** Το όνομα της ομάδας
  - **<Hostile> (0 ή περισσότερα):** Θέτει μια ομάδα που θα είναι εχθρική.
    - **name (Απαραίτητο):** Το όνομα της ομάδας που θα είναι εχθρική.

Ας φτιάξουμε ένα πολύ βασικό παράδειγμα. Θέλουμε να έχουμε 2 βασικές ομάδες: τους σύμμαχους και τους εχθρούς.

Πίνακας 3.4.1: Παράδειγμα ορισμού Ομάδων

```
<?xml version="1.0" encoding="UTF-8"?>
<Factions>
	<Faction name="Allies">
		<Hostile name="Enemies" />
	</Faction>
	<Faction name="Enemies">
		<Hostile name="Allies" />
	</Faction>
</Factions>
```

Εδώ ορίσαμε την ομάδα “Allies” που θα επιτίθενται σε άτομα της ομάδας “Enemies”, όπως και ορίσαμε την ομάδα “Enemies” να επιτίθενται στην “Allies”. Τώρα αν θέλουμε να αναθέσουμε μια LifeformClass σε μια ομάδα, το κάνουμε όπως είχαμε πει.

Πίνακας 3.4.2: Παράδειγμα ανάθεσης σε ομάδα

```
...
<LifeformClass name="Φίδι" faction="Enemies">
...
```

Να σημειωθεί ότι υπάρχει και η ομάδα “Neutral” που δημιουργείται αυτόματα και σε αυτή την ομάδα μπαίνουν όλα τα LifeformClass που δεν τους την ορίζουμε ρητά. Τα Lifeform που ανήκουν σε αυτή την ομάδα δεν επιτίθενται σε τίποτα.

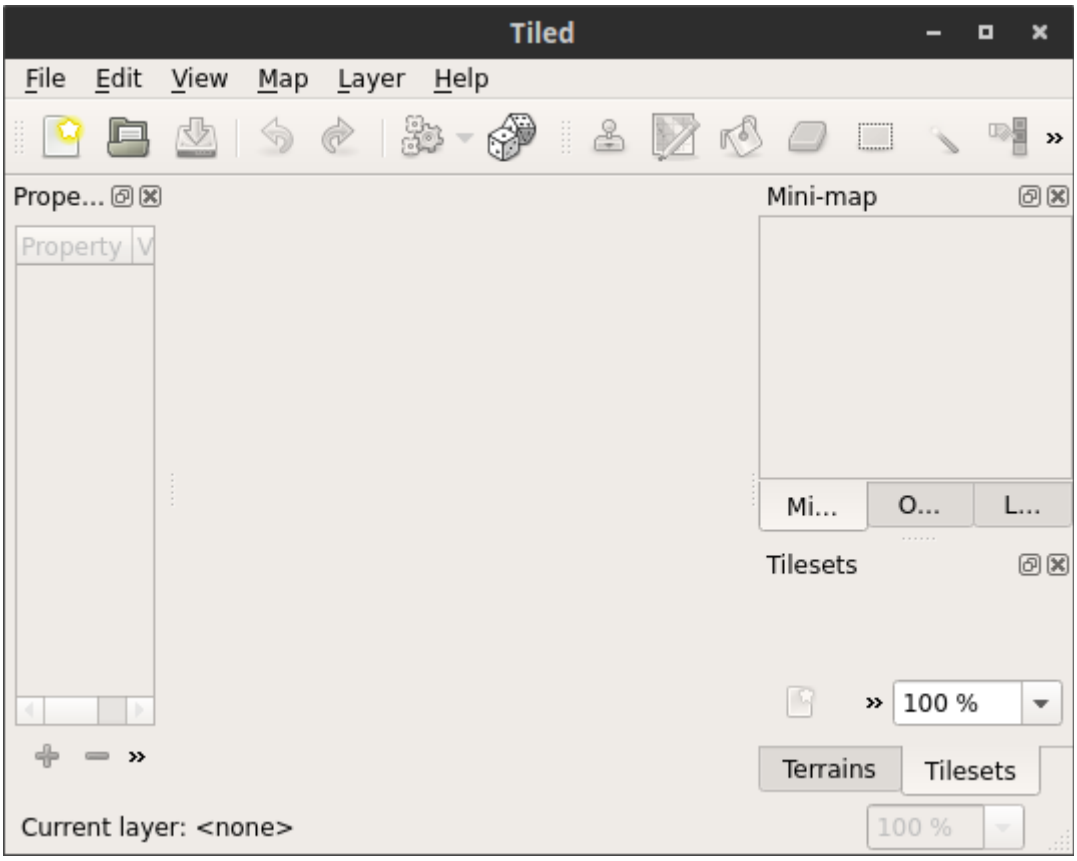
4 ΔΗΜΙΟΥΡΓΙΑ ΧΑΡΤΩΝ

Μέχρι στιγμής έχουμε δει πως φτιάχνουμε ζωντανές οντότητες (Lifeforms). Όμως αυτές οι οντότητες χρειάζεται να μπουν σε κάποιο “κόσμο”. Μπορούμε να παίξουμε μόνο με τις οντότητες άλλα θα υπάρχουν όλα σε ένα μαύρο κόσμο. Αυτό όμως δεν είναι συνετό. Χρειάζεται να δημιουργήσουμε κόσμους. Αυτοί οι κόσμοι λέγονται χάρτες.

4.1 Το εργαλείο Tiled

Η μηχανή χρησιμοποιεί αρχεία XML για της ορίσουμε τους χάρτες. Το να φτιάξουμε τους χάρτες γράφοντας αυτό τα αρχεία από την αρχή είναι απίστευτα επίπονο. Έτσι η μηχανή έχει προσαρμοστεί στα δεδομένα που παράγει το εργαλείο “Tiled” του Thorbjørn Lindeijer.

Το εργαλείο αυτό μας δίνει την δυνατότητα να φτιάξουμε χάρτες και πίνακες πλακιδίων (Tilesets) γραφικά και εύκολα. Στο τέλος μας δίνει την δυνατότητα να το εξάγουμε σε διάφορους τύπους αρχείων όπως και XML. Το εργαλείο αυτό παρέχει πολλές λειτουργίες που είναι χρήσιμες στην μηχανή που θα τις δούμε παρακάτω.



Εικόνα 4.1.1: Παράθυρο του εργαλείου Tiled

Αριστερά έχουμε την λίστα από τις ιδιότητες του αντικειμένου που επιλέξαμε. Στο κέντρο θα έχουμε τον χάρτη που φτιάχνουμε. Δεξιά έχουμε πάνω την μικρογραφία του χάρτη μας και τις λίστες από στρώσεις (Layers) και αντικειμένων (Objects). Κάτω από εκεί έχουμε το παράθυρο των Tilesets και των εδαφών (Terrains). Ας πάμε να φτιάξουμε ένα δωμάτιο.

4.2 Πίνακες πλακιδίων (Tilesets)

Οι χάρτες είναι μια σειρά από μικρές εικόνες που εμφανίζονται η μια δίπλα στην άλλη σε έναν πίνακα. Αυτές οι εικόνες λέγονται πλακίδια (Tiles). Αυτά τα Tiles, προέρχονται από μια μεγάλη εικόνα που έχει πολλά διαφορετικά Tiles, κάτι σαν το Spritesheet και λέγεται πίνακας πλακιδίων ή Tileset. Χωρίς Tileset ένας χάρτης είναι άχρηστος γιατί δεν έχει τα γραφικά που θέλει να δείξει. Για αυτό είναι απαραίτητη η δημιουργία του.

Ένα Tileset μπορεί να είναι εξωτερικό ή ενσωματωμένο στο αρχείο του χάρτη. Η μηχανή απορρίπτει τα ενσωματωμένα για οικονομία πόρων, καθώς κάθε ενσωματωμένο Tileset δεν μπορεί να αναφερθεί σε άλλους χάρτες και χάνεται όταν ο χάρτης αλλάξει.

Το Tileset αποτελείται από 2 αρχεία, την εικόνα και το αρχείο XML που ορίζονται ιδιότητες των Tiles και δεδομένα που χρησιμοποιεί το Tiled. Το Tileset φορτώνεται αυτόματα από τον χάρτη που φορτώνουμε καθώς έχει αναφορά σε αυτό. Τότε η μηχανή σπάει την εικόνα του σε μικρότερες μεγέθους που ορίζεται από το αρχείο XML του Tileset. Οι εικόνες αυτές σπάνε δίπλα-δίπλα στην εικόνα, δηλαδή αν η εικόνα είναι διαστάσεων 2048x1024 Pixel και το μέγεθος του Tile που ορίζουμε είναι 64x64 pixel, τότε θα φτιαχτούν:

$$n_{tiles} = \frac{\frac{width_{image}}{width_{tile}} * height_{image}}{height_{tile}} = \frac{\frac{2048}{64} * 1024}{64} = 32 * 16 = 512 \quad (4.1.1)$$

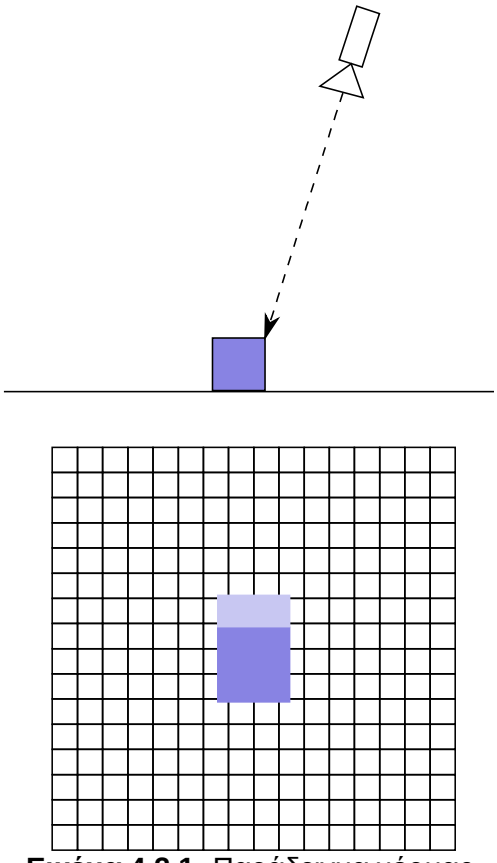
Με άλλα λόγια, όλη η εικόνα θα σπάσει σε τετραγωνάκια που έχουν μέγεθος του Tile. Κάθε Tile που δημιουργείται του δίνεται μια εσωτερική αναφορά και όταν το Tileset που περιέχει αυτό το Tile συνδέεται με έναν χάρτη, του δίνεται και μια μοναδική καθολική αναφορά στον χάρτη. Για αυτό, κάθε Tile πρέπει να είναι μοναδικό και να μην έχει τα ίδια γραφικά με ένα άλλο, για να αποφύγουμε πλεονασμούς δεδο-

μένων και πόρων. Το Tileset πρέπει να το βλέπουμε σαν μια παλέτα από χαρακτηριστικά του χάρτη όσο αφορά την εμφάνιση.

Ας πάμε να ξεκινήσουμε να φτιάξουμε ένα χάρτη. Για αρχή θα φτιάξουμε κάτι μικρό, ένα δωμάτιο.

Στο εργαλείο Tiled, πηγαίνουμε File->New και μας εμφανίζει ένα παράθυρο διαλόγου για να επιλέξουμε κάποιες ιδιότητες του χάρτη μας. Οι επιλογές μας είναι:

- **Orientation:** Το πως θα φαίνεται ο χάρτης. Στα δισδιάστατα γραφικά υπάρχουν 2 νόρμες: το Top-Down (Orthogonal) και το Isometric. Στα Top-Down ο κόσμος φαίνεται σαν η κάμερα να κοιτάει κάθετα στο έδαφος (όχι απαραίτητα 90 μοίρες, συνήθως λίγο λιγότερο).
-

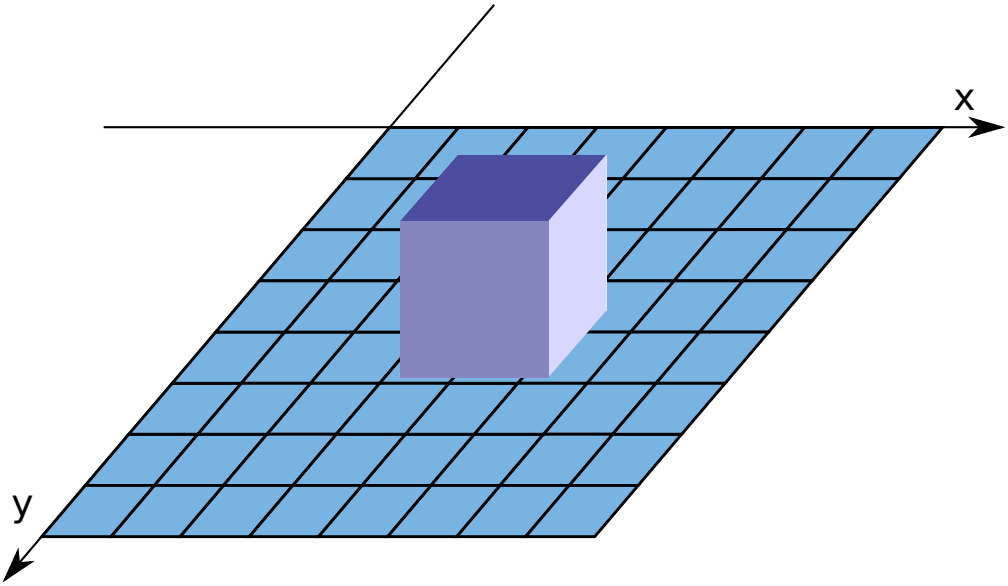


Εικόνα 4.2.1: Παράδειγμα νόρμας Top-Down

Σε αυτή την τεχνική, τα Tiles χρειάζονται μόνο πλάτος, μήκος και συνήθως είναι τετράγωνα.

Στην τεχνική “Isometric”, τα tiles σχεδιάζονται σαν να είναι ψευδό τρισδιάστατα. Η κάμερα τοποθετείται όπως στο Top-Down αλλά λίγο πλάγια, δίνοντας μια αίσθηση

τρισδιάστατου. Τα tiles σε αυτή την τεχνική χρειάζονται επιπλέον δεδομένα για την μορφή και την τοποθεσία τους στον κόσμο για να λειτουργήσουν σωστά.



Εικόνα 4.2.2: Παράδειγμα εμφάνισης Isometric

Στην παραπάνω εικόνα, ο κύβος μοιάζει να είναι τρισδιάστατος αλλά δεν είναι. Το Tile σχεδιάστηκε να φαίνεται έτσι.

Η μηχανή δεν υποστηρίζει Isometric Tiles και για αυτό θα ασχοληθούμε μόνο με τετράγωνα Top-Down Tiles. Για αυτό και η επιλογή μας εδώ είναι το “Orthogonal”

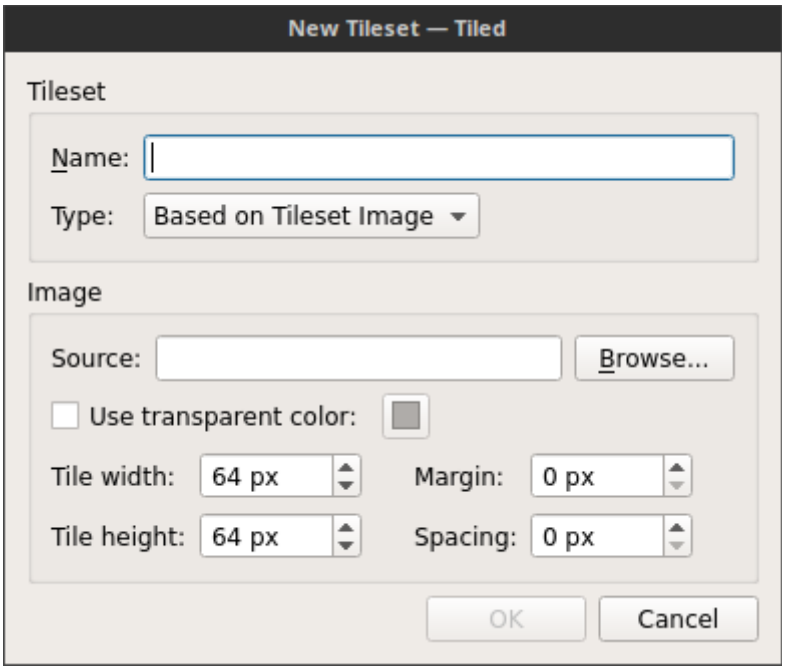
- **Tile Layer Format:** Εδώ ορίζουμε τι μορφή θα έχουν τα δεδομένα στο XML αρχείο όσον αφορά τα Layers. Τα Layers είναι μεγάλοι πίνακες (όσο και το μέγεθος του χάρτη) που περιέχουν αναφορές σε Tiles. Θα δούμε παρακάτω περισσότερα. Αυτό που μας νοιάζει τώρα είναι να ορίσουμε τι μορφή έχουν. Επειδή αυτοί οι πίνακες είναι μεγάλοι, πρέπει να τους συμπιέσουμε για να μικρύνει το αρχείο του χάρτη. Η μηχανή υποστηρίζει συμπίεση Zlib με κωδικοποίηση Base64 η οποία είναι και η προεπιλεγμένη μορφή.
- **Tile Render Order:** Στοιχείο που η μηχανή αγνοεί εντελώς και ότι τιμή και να δοθεί δεν θα αλλάξει κάτι.
- **Map Size:** Ορίζουμε το μέγεθος του χάρτη σε αριθμό Tiles σε μήκος και πλάτος. Ανάλογα το μέγεθος των Tiles, θα πάρει και ανάλογο μέγεθος ο χάρτης σε Pixels. Ας ορίζουμε αρχικά 20x20 Tiles για το δωμάτιο που θα φτιάξουμε.

- **Tile Size:** Ορίζουμε το μέγεθος των Tiles που θα έχει ο χάρτης. Για το Demo μας τα Tiles που έχουμε είναι μεγέθους 64x64.

Πατώντας τώρα OK μας βγάζει σε ένα κενό γκρι πίνακα διαστάσεων όσο είχαμε ορίσει στο Map Size.

Ακόμα δεν μπορούμε να σχεδιάσουμε τον χάρτη γιατί δεν έχουμε τι να βάλουμε μέσα. Χρειαζόμαστε είτε να συνδέσουμε ένα υπάρχων Tileset, είτε να φτιάξουμε ένα. Με την υπόθεση ότι δεν έχουμε Tileset, πάμε να φτιάξουμε ένα κάνοντας το μεμονωμένο αρχείο για να μπορούμε να συνδέσουμε και σε άλλους χάρτες χωρίς να το φτιάξουμε από την αρχή.

Επιλέγουμε από το μενού Map->New Tileset. Μας εμφανίζει ένα νέο παράθυρο διαλόγου.



Εικόνα 4.2.3: Παράθυρο δημιουργίας Tileset

Δίνουμε ένα όνομα που θέλουμε και επιλέγουμε την επιλογή: “Based on Tileset Image” καθώς θα έχουμε μια μεγάλη εικόνα που θα σπάσουμε σε μικρές.

Στο Source βάζουμε το αρχείο της εικόνας. Στο Tile width/height, βάζουμε τα μεγέθη των Tile που έχουμε. Τα πεδία Margin και Spacing τα αγνοεί η μηχανή.

Πατώντας OK το Tileset είναι έτοιμο για χρήση αλλά προς το παρών είναι ενσωματωμένο στον χάρτη μας, πράγμα που δεν θέλουμε. Πατώντας το “Export Tileset As” μας αφήνει να το εξάγουμε σε αρχείο. Το αποθηκεύουμε σε μέρος που είναι και η ει-

κόνα του μαζί για λόγους σαφήνειας. Τώρα το Tileset είναι προσβάσιμο και από άλλους χάρτες χωρίς να το ξαναφτιάξουμε. Όμως χρειάζεται κάτι ακόμα που θα δούμε παρακάτω.

a) *Ο χάρτης των Στατικών Συγκρούσεων*

Τα Tilesets χρειάζεται να φέρουν κάποιες πληροφορίες για τα Tiles που έχουν. Μια σημαντική πληροφορία είναι το τι είδους σύγκρουση προκαλεί στον κόσμο. Αυτό το ορίζουμε μέσω μιας ιδιότητας που βάζουμε σε κάθε Tile, την ιδιότητα “Collision".

Αυτή η ιδιότητα παίρνει σαν τιμή έναν αριθμό από το 0 (0x0) έως το 15 (0xF). Το 0 αντιπροσωπεύει την μηδενική σύγκρουση και είναι το προκαθορισμένο εφόσον δεν οριστεί η ιδιότητα. Η τιμή 15 αντιπροσωπεύει την πλήρη σύγκρουση με όλο το Tile. Και οι 15 διαφορετικές περιπτώσεις περιγράφονται από την παρακάτω σχεδίαση.

7 (0x7)	1 (0x1)	3 (0x3)	2 (0x2)	11 (0xB)
9 (0x9)	5 (0x5)	15 (0xF)	10 (0xA)	6 (0x6)
13 (0xD)	4 (0x4)	12 (0xC)	8 (0x8)	14 (0xE)

Εικόνα 4.2.4: Πίνακας στατικών συγκρούσεων

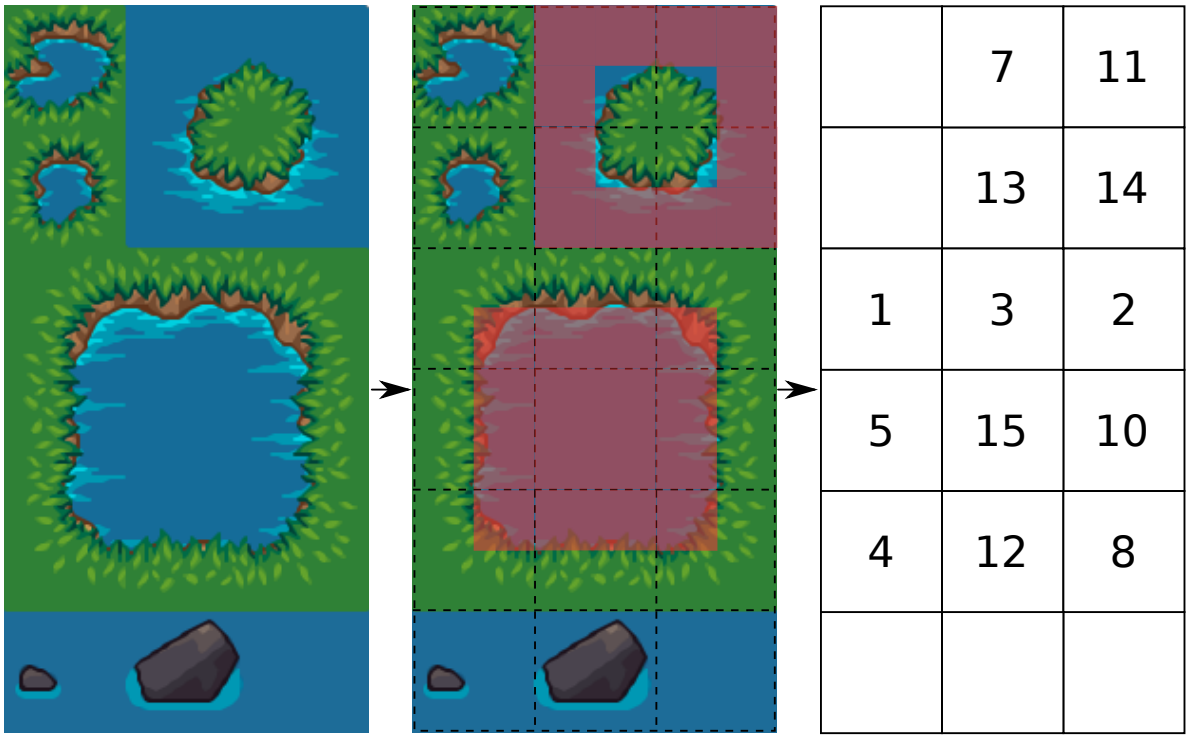
Όπως βλέπουμε, κάθε Tile χωρίζεται σε 4 υπό-τετράγωνα για έλεγχο σύγκρουσης. Στον παραπάνω πίνακα, κάθε τετράγωνο αναπαριστά μια περίπτωση από τις 15 που έχει σύγκρουση. Τα μπλε ορθογώνια (υπό-τετράγωνα) μέσα σε κάθε τετράγωνο αναπαριστούν το ποια επιφάνεια του Tile θα είναι συμπαγής και δεν θα μπορεί να περάσει τίποτα από εκεί. Ο αριθμός μέσα στο τετράγωνο είναι ο αριθμός που πρέπει να βάλουμε στην ιδιότητα “Collision” για να καλύψουμε την επιφάνεια που θέλουμε. Η δεκαεξαδική τιμή από κάτω είναι ο ίδιος αριθμός που χρησιμοποιείται εσωτερικά

στην μηχανή γιατί χρησιμοποιεί Bitfields για τα Collisions όπως θα δούμε στο επόμενο μέρος.

Οι τιμές αυτές όπως είπαμε θα ανατεθούν σε Tiles στο Titleset που φτιάχνουμε. Έτσι δεν θα χρειάζεται κάθε φορά που βάζουμε Tiles στον χάρτη να βάζουμε τις τιμές από την αρχή.

Επιστρέφουμε στο Tiled. Το Tileset μας είναι έτοιμο άλλα δεν έχει ακόμα τις πληροφορίες για τις συγκρούσεις. Πατώντας ένα Tile κάτω δεξιά που είναι το Tileset μας, αριστερά στην οθόνη μας βγάζει τις ιδιότητές του. Πατώντας κάτω το +, μπορούμε να προσθέσουμε μια δικιά μας ιδιότητα. Αφού ορίσουμε το όνομα της ιδιότητας (Collision), μπορούμε να βάλουμε και την τιμή της. Υπόψιν ότι μπορούμε να βάλουμε πολλές διαφορετικές ιδιότητες στο Tile. Η μηχανή θα τις πάρει όλες αλλά μόνο την ιδιότητα “Collision” θα χρησιμοποιήσει εσωτερικά. Οι υπόλοιπες αφήνονται για τον προγραμματιστή.

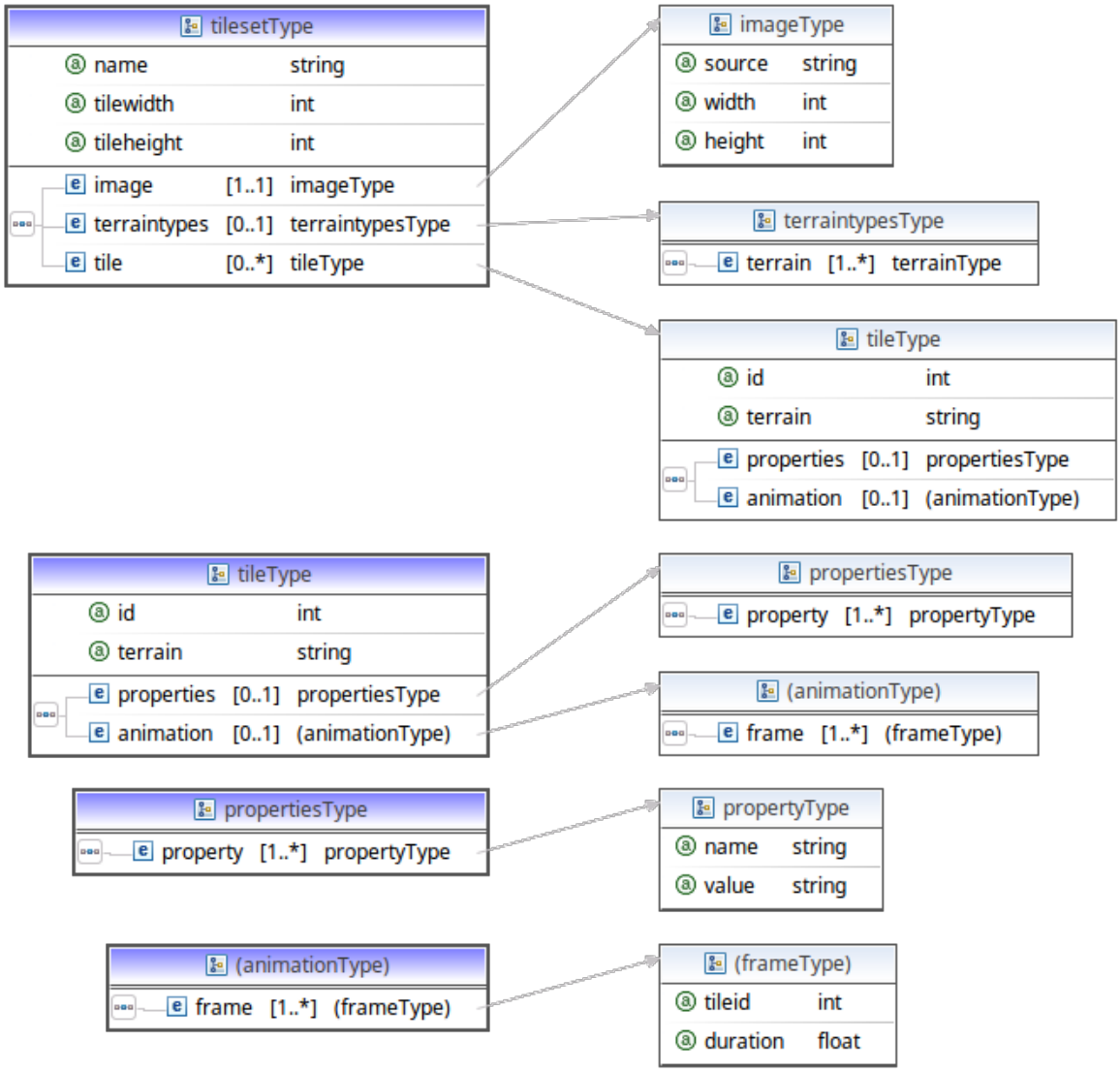
Τώρα τι τιμή να βάλουμε στην Collision; Όπως είπαμε, η τιμή της Collision ορίζει την περιοχή του Tile που θα είναι συμπαγής και δεν θα μπορούν να περάσουν μέσα Lifeforms. Για παράδειγμα δεν θέλουμε τα Lifeforms να περπατάνε στο νερό, για αυτό και θα κάνουμε συμπαγές τις άκρες που αρχίζει το νερό.



Εικόνα 4.2.5: Παράδειγμα αντιστοίχησης τιμών Collision

Στο Spritesheet παραπάνω βλέπουμε τι αριθμούς θα βάλουμε προκειμένου να πετύχουμε αυτό που θέλουμε. Τώρα κάθε φορά που θα βάζουμε ένα Tile στον χάρτη, θα παίρνει αυτόματα την τιμή του Collision και η μηχανή θα συμπεριφέρεται αντιστοίχως. Πρέπει να θυμίσουμε ότι σε ό,τι Tile δεν βάλουμε τιμή στο Collision θα πάρει αυτόματα την τιμή 0 και τα Lifeform θα περνάνε από μέσα του. Έτσι μειώνεται ο αριθμός των Tile που πρέπει να βάλουμε τιμή στο Collision.

Είδαμε πως φτιάχνουμε ένα Tileset μέσω του Tiled. Όμως η μηχανή δέχεται XML αρχεία που έχουν μορφή ίδια με αυτή που εκδίδει το Tiled. Αυτό σημαίνει ότι δεν είναι απαραίτητο να χρησιμοποιήσουμε το Tiled για να φτιάξουμε Tilesets. Μπορούμε να γράψουμε το αρχείο XML από την αρχή με το χέρι σαν να το είχε φτιάξει το Tiled.



Εικόνα 4.2.6: Ορισμός του Tileset XSD

<tileset> Το πρώτο στοιχείο του αρχείου. Περιέχει:

name (Απαραίτητο): Το όνομα του Tileset. Χρησιμοποιείται κυρίως από το Tiled.

tileWidth (Απαραίτητο): Το μέγεθος των Tile σε πλάτος.

tileHeight (Απαραίτητο): Το μέγεθος των Tile σε ύψος.

<image> (Απαραίτητο 1): Το στοιχείο που ορίζει την εικόνα που θα χρησιμοποιηθεί. Περιέχει:

source (Απαραίτητο): Η διαδρομή του αρχείου της εικόνας.

width (Απαραίτητο): Το μέγεθος της εικόνας σε πλάτος.

height (Απαραίτητο): Το μέγεθος της εικόνας σε ύψος.

<terraintypes> (Προαιρετικό): Προαιρετικό στοιχείο που το αγνοεί τελείως η μηχανή. Δεν χρειάζεται να γραφτεί με το χέρι. Χρησιμοποιείται μόνο από το Tiled.

<tiled> (0 ή περισσότερα): Περιγράφει κάποιες ιδιότητες για κάποιο συγκεκριμένο Tile. Περιέχει:

id (Απαραίτητο): Το id του tile που θα ισχύσουν οι ιδιότητες. Είναι ένας αριθμός που αντιπροσωπεύει ένα Tile. Αυτός ο αριθμός είναι αύξων και ξεκινάει από πάνω αριστερά, πηγαίνοντας δεξιά και ύστερα από κάτω αριστερά πάλι.

terrain (Προαιρετικό): Στοιχείο που αγνοεί η μηχανή και χρησιμοποιείται από το Tiled και μόνο.

<properties> (Προαιρετικό): Καθορίζει την ομάδα των ιδιοτήτων του Tile.

<property> (1 ή περισσότερα): Ορίζει μια ιδιότητα.

name (Απαραίτητο): Ορίζει το όνομα της ιδιότητας.

value (Απαραίτητο): Ορίζει την τιμή της ιδιότητας.

<animation> (Προαιρετικό): Ορίζει το Animation του Tile εφόσον υπάρχει.

<frame> (1 ή περισσότερα): Ορίζει ένα Frame.

tileid (Απαραίτητο): Το tile που περιέχει το frame.

duration (Απαραίτητο): Η διάρκεια του Frame σε msec.

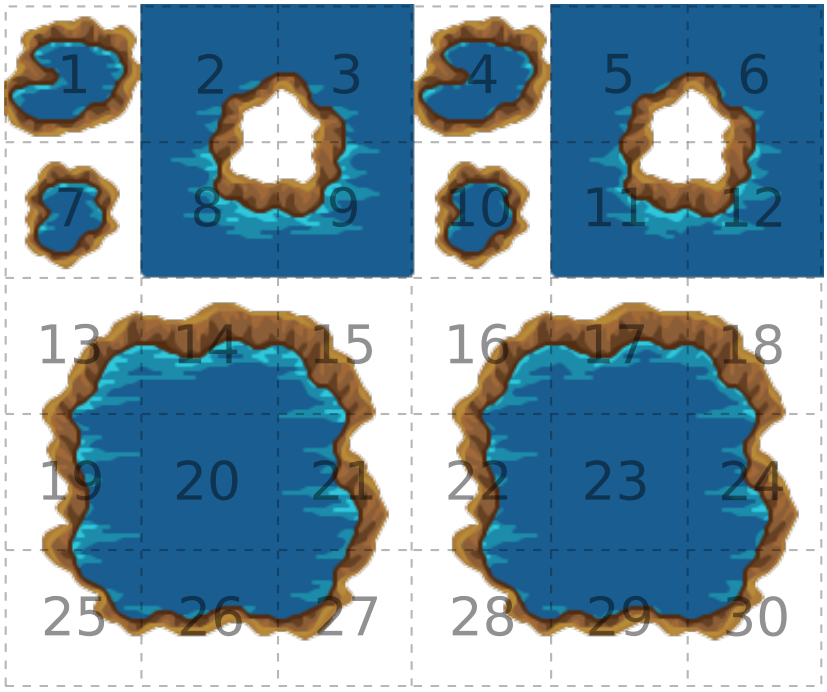
Αυτό που δεν έχουμε αναφέρει είναι για τα Animation. Κάποια Tiles μπορούμε να τα κάνουμε Animated. Αυτό γίνεται είτε μέσω του XML όπως παραπάνω, είτε μέσω του Tiled.

Για να κάνουμε Animated Tiles, χρειαζόμαστε όπως και στα Sprites πολλά διαφορετικά Tiles για κάθε Frame. Τα Tiles πρέπει να βρίσκονται στο ίδιο Tileset όπως παρακάτω.

Ορίζουμε ποιο Tile θα αντιπροσωπεύει το Animation και μετά παραθέτουμε τα Frames όπως παραπάνω, μέσω του XML (στοιχείο <Animation>) ή μέσω του Tiled (μενού Animation).

Ένα που χρειάζεται να δούμε είναι η ιδιότητα Relatives. Σε αυτή την ιδιότητα ορίζονται με τους αριθμούς τους τα έτερα Tiles(χωρισμένα με κόμμα) τα οποία πρέπει να συγχρονίζονται με αυτό. Αυτό έχει να κάνει μόνο στην περίπτωση που το Tile είναι Animated και για σωστό αποτέλεσμα πρέπει όλα τα συγγενικά Tiles να έχουν αυτή την ιδιότητα με την ίδια τιμή.

Όταν τελειώσουμε να φτιάχνουμε το Tileset, τότε το εξάγουμε σε αρχείο και το εισάγουμε ξανά από το μενού Map->Add External Tileset.



Εικόνα 4.2.7: Παράδειγμα Tileset με Animation

4.3 Στρώσεις (Layers)

Σε έναν χάρτη, αυτό που βλέπουμε είναι πολλές στρώσεις η μία πάνω στην άλλη. Μια στρώση μπορεί να περιέχει Tiles σχετικά με κάτι και από πάνω να φαίνεται κάτι άλλο που ανήκει σε μια άλλη στρώση.

Οι στρώσεις τοποθετούνται σε στοίβα. Τα Tiles της πιο κάτω θα σχεδιαστούν πρώτα πάνω στον Framebuffer. Ύστερα τα Tiles της από πάνω της θα σχεδιαστούν πάνω στον Framebuffer χωρίς να καθαριστεί. Έτσι αν κάποιο Tile της από πάνω καλύψει ένα Tile της από κάτω, τότε δεν θα φαίνεται ή θα φαίνεται ένα μέρος του ανάλογα πόσο το κάλυψε η πάνω στρώση. Το ίδιο θα συνέβαινε και με την πάνω αν υπήρχε και τρίτη στρώση από πάνω της.

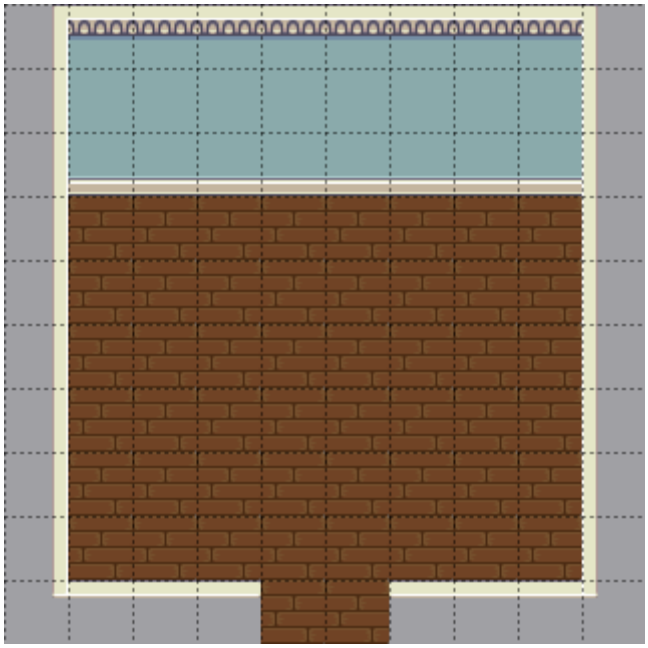
Με αυτόν τον τρόπο πετυχαίνουμε προσαρμοσμένες εμφανίσεις αντικειμένων στον χάρτη και μας επιτρέπει να ορίσουμε κάποια αντικείμενα όσον αφορά την θέση τους με τα Lifeforms, είτε στατικά είτε δυναμικά όπως θα δούμε παρακάτω.

Έχουμε από πριν στο Tiled ένα κενό χάρτη και έτοιμο ένα Tileset για αρχίσουμε να βάζουμε Tiles. Στην καρτέλα Layers βλέπουμε ότι το πρόγραμμα έχει ήδη βάλει ένα Layer, οπότε μπορούμε να βάλουμε τα Tiles μας.

Πριν το κάνουμε αυτό, θα ήταν καλό να δώσουμε ένα όνομα στο Layer για να ξέρουμε αργότερα τι έχουμε βάλει εκεί. Επιλέγοντας το Layer, αριστερά εμφανίζονται οι ιδιότητες του από όπου και του δίνουμε το όνομα “Floor” καθώς και θα φτιάξουμε το πάτωμα του δωματίου πρώτα. Επιλέγουμε ένα Tile από το Tileset που θα αντιπροσωπεύει το πάτωμα και γεμίζουμε τον χώρο που θα καλύψει. Υπόψιν ότι πρέπει να γεμίσουμε το πάτωμα με κάτι που έχει 0 στο Collision, δηλαδή δεν έχει καθόλου σύγκρουση γιατί οι χαρακτήρες θα πρέπει να κινούνται πάνω στο πάτωμα.

Αφού φτιάξαμε το πάτωμα, ας βάλουμε τοίχους. Επιλέγουμε τον τοίχο. (Έχετε στον νου ότι μπορούμε να επιλέξουμε πολλά Tiles μαζί από το Tileset και να τα τοποθετήσουμε όπως τα βλέπουμε εκεί πάνω στον χάρτη). Τώρα όμως όταν πάμε να βάλουμε τον τοίχο κάπου, τα Tiles του πατώματος χάνονται και φαίνεται άσχημο. Δεν θέλουμε να χαθούν αυτά τα Tiles αλλά ο τοίχος να ζωγραφιστεί από πάνω. Για αυτό και χρειαζόμαστε ένα νέο Layer που θα εμφανίζεται από πάνω από το Layer “Floor”.

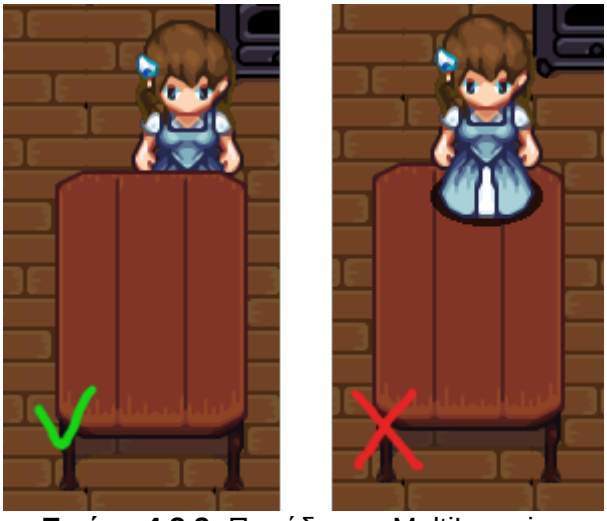
Δημιουργούμε ένα νέο “Tile Layer” από το πρώτο κουμπί κάτω από την λίστα των Layers. Του δίνουμε όνομα “Walls” και συνεχίζουμε να βάζουμε τους τοίχους όπως θέλαμε χωρίς πρόβλημα όπως πχ παρακάτω:



Εικόνα 4.3.1: Παράδειγμα χάρτη 01

Ο χάρτης αν και ελλιπής, είναι εντάξει για να χρησιμοποιηθεί στο παιχνίδι. Υπάρχει όμως κάτι που πρέπει να σκεφτούμε. Την σειρά που θα σχεδιαστούν τα Layers σε σχέση με τα Lifeforms που θα υπάρξουν μέσα σε αυτόν. Για παράδειγμα οι τοίχοι και το πάτωμα ξέρουμε ότι εμφανίζονται πάντα πίσω από τα Lifeforms, για αυτό και θα σχεδιάζονται πρώτα αυτά και μετά τα Lifeforms. Τι γίνεται όμως αν κάτι πρέπει να σχεδιαστεί μπροστά από τα Lifeforms;

Ας τοποθετήσουμε ένα τραπεζάκι μέσα στον χώρο. Σε ένα καινούργιο Layer “Objects01” βάζουμε ένα τραπεζάκι. Στο δικό μου Tileset, έχω κάνει το πάνω μέρος του τραπεζιού να μην έχει σύγκρουση για να μπορεί ο παίχτης να πηγαίνει από κάτω και φαίνεται ωραία όπως παρακάτω.



Εικόνα 4.3.2: Παράδειγμα MultiLayering

Αριστερά έχουμε αυτό που θέλουμε να πετύχουμε και δεξιά αυτό που θα πετύχουμε όπως το έχουμε τώρα. Για να λύσουμε αυτό το πρόβλημα πρέπει να σπάσουμε το τραπέζι σε 2 Layers, σε ένα που θα εμφανίζεται πίσω από τον χαρακτήρα (το μπροστινό μέρος) και σε ένα που θα εμφανίζεται από πίσω (το πίσω μέρος). Για αυτό και φτιάχνουμε ένα νέο Layer “Objects_Front”. Εκεί βάζουμε το μέρος του τραπεζιού που θέλουμε να εμφανίζεται μπροστά από τον χαρακτήρα. Όμως αυτό δεν φτάνει. Πρέπει να πούμε στην μηχανή ότι αυτό το Layer θα εμφανίζεται μπροστά από τα Lifeform. Αυτό γίνεται από την ιδιότητα του χάρτη “Layer Priority”

Η ιδιότητα αυτή είναι μια προσαρμοσμένη ιδιότητα που βάζουμε στο αρχείο του χάρτη για να δώσουμε στην μηχανή την σειρά που πρέπει να σχεδιαστούν τα Layers του χάρτη. Πέρα από αυτό επίσης λέει και την σειρά ενδιάμεσα στα Layers που πρέπει να σχεδιαστούν τα Lifeforms και τα Objects. Η γενική μορφή της ιδιότητας είναι:

$$BackLayer_1, BackLayer_n, \cdots, \text{\_OBJ\_}, FrontLayer_1, FrontLayer_n, \cdots \quad (4.3.1)$$

Αρχικ μπαίνουν τα ονόματα των Layers που θα σχεδιαστούν πρώτα χωρισμένα με κόμματα, ύστερα το καθολικό σύμβολο “_OBJ_” που αναπαριστά ότι σε αυτή την θέση θα σχεδιαστούν τα Lifeforms και ύστερα Layers που θέλουμε να σχεδιαστούν μπροστά από τα Lifeforms. Πρέπει να τονιστεί ότι η σειρά αυτή είναι αυστηρή και με αυτό που θα γράψουμε εκεί, έτσι θα γίνει και στην μηχανή αγνοώντας το πως το φτιάξαμε στο Tiled. Επίσης αν κάνουμε κάποιο λάθος σε κάποιο όνομα Layer, τότε αυτό το Layer δεν θα εντοπιστεί από την μηχανή και θα μπει στην σειρά που θα σχεδιαστούν πίσω, για αυτό και πρέπει να είμαστε ιδιαίτερα προσεχτικοί όταν το γράφουμε για να πάρουμε τα σωστά αποτελέσματα.

Αυτή την ιδιότητα την γράφουμε στο μενού Map->Map Properties. Εκεί προσθέτουμε μια ιδιότητα “Layer Priority”. Τώρα πρέπει να δώσουμε την τιμή. Όπως είπαμε το πάτωμα, οι τοίχοι καθώς και το μπροστινό μέρος του τραπεζιού θα εμφανιστούν πίσω από τα Lifeforms. Το μόνο που θα εμφανιστεί μπροστά είναι το πίσω μέρος του τραπεζιού. Έτσι αυτό που πρέπει να γράψουμε στην ιδιότητα είναι:

Πίνακας 4.3.1: Παράδειγμα ιδιότητας "Layer Priority"

```
Floor,Walls,Objects01,_OBJ_,Objects_Front
```

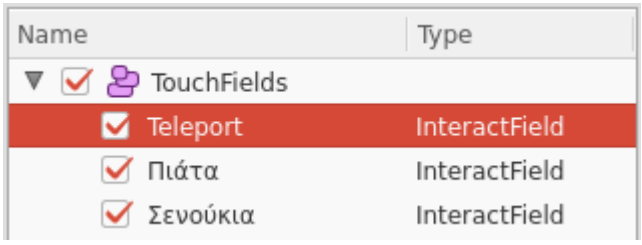
Αποθηκεύοντας τον χάρτη μας, είναι πλέον έτοιμος να φορτωθεί από την μηχανή.

4.4 Αντικείμενα Interact

Η μηχανή μας επιτρέπει να προσθέσουμε στον χάρτη “Δυναμικά” στοιχεία που ανάλογα αν κάτι συγκρουστεί μαζί του ή ο παίχτης κάνει κλικ πάνω τους, να εκτελεστεί κάποιος κώδικας Lua. Αυτά τα αντικείμενα είναι τα InteractFields. Αυτά τα αντικείμενα είναι αόρατα (δηλαδή δεν έχουν Sprite ή Tile) άλλα μπορούν να έχουν σύγκρουση εάν χρειάζεται.

Για να βάλουμε ένα αντικείμενο Interact, πρέπει να έχουμε στον χάρτη μας ένα ObjectLayer. Για να το φτιάξουμε, πρέπει στο Tiled να πάμε στην καρτέλα “Objects” (δίπλα από την Layers) και να πατήσουμε το κουμπί “Add Object Layer”. Τώρα πατώντας το κουμπί πάνω στην γραμμή εργαλείων (κάτω από το κύριο μενού) με το μπλε τετράγωνο, μπορούμε με το αριστερό κλικ να σχεδιάσουμε τετράγωνα μέσα στον χάρτη. Όταν φτιάχνουμε ένα τετράγωνο στο ObjectLayer που φτιάξαμε, προστίθεται μια κενή γραμμή. Αυτή η γραμμή αντιπροσωπεύει το αντικείμενο που φτιάξαμε. Η γραμμή είναι κενή γιατί το αντικείμενο δεν έχει όνομα και τύπο όπως βλέπουμε.

Για κάθε τετράγωνο που φτιάχνουμε, θα ήταν καλό να του δώσουμε ένα όνομα για να ξέρουμε τι είναι. Αυτό γίνεται πατώντας διπλό κλικ στην κενή γραμμή που αντιπροσωπεύει το αντικείμενο στην μεριά του ονόματος. Το να δώσουμε ένα όνομα στο αντικείμενο δεν είναι απαραίτητο για την μηχανή καθώς το αγνοεί. Αυτό που είναι απαραίτητο για την μηχανή είναι να του δώσουμε τον τύπο του. Τον τύπο τον γράφουμε όπως και το όνομα εκεί που είναι κενό. Για τα αντικείμενα Interact, πρέπει να δώσουμε “InteractField”.



Εικόνα 4.4.1: Παράδειγμα εισαγωγή Object

Τώρα που ορίσαμε το αντικείμενο είναι τελείως άχρηστο χωρίς να του δώσουμε να ξέρει τι να κάνει. Τα InteractFields όπως είπαμε μπορούν να εκτελέσουν κώδικα Lua

είτε όταν συγκρουστεί κάποιο Lifeform μαζί του, είτε όταν ο παίχτης το κάνει κλικ, είτε και τα δύο. Για να βάλουμε το τι κώδικα να τρέξει σε κάποια περίπτωση, αρκεί να του θέσουμε τις αντίστοιχες ιδιότητες όπως και με τα άλλα στοιχεία, στα Custom Properties αριστερά στο Tiled. Οι ιδιότητες είναι:

- **onCollision:** Στην τιμή αυτής της ιδιότητας βάζουμε τον κώδικα Lua που θέλουμε να τρέξουμε σε περίπτωση που κάποιο Lifeform συγκρουστεί με αυτό. Εάν δεν οριστεί, τότε δεν θα γίνει τίποτα σε περίπτωση σύγκρουσης.
- **onClick:** Στην τιμή αυτής της ιδιότητας βάζουμε τον κώδικα Lua που θέλουμε να τρέξουμε σε περίπτωση που ο παίχτης κάνει κλικ πάνω σε αυτό το αντικείμενο. Όπως και με την παραπάνω αν δεν οριστεί δεν θα γίνει τίποτα σε περίπτωση κλικ.
- **Collidable:** Παίρνει τιμές 0 ή 1. Σε περίπτωση που δοθεί η τιμή 1, τότε το αντικείμενο θα γίνει συμπαγές και δεν θα μπορούν να περάσουν μέσα Lifeforms, κάτι σαν αόρατος τοίχος. Σε κάθε περίπτωση τα συμβάντα σύγκρουσης καλούνται. Εάν δεν οριστεί τότε ορίζεται αυτόματα η τιμή 0.

4.5 Αντικείμενα Tile

Υπάρχουν κάποιες περιπτώσεις που ο απλός διαχωρισμός των Tiles ενός αντικειμένου σε πολλά Layers (όπως στην περίπτωση του τραπεζιού), δεν αρκεί επειδή τα Lifeforms έχουν αρκετά μεγάλα Sprites σε σχέση με το αντικείμενο που θέλουμε να δείξουμε. Για παράδειγμα αν το τραπέζι μας ήταν μικρό σε μήκος, με την προηγούμενη τεχνική θα παίρναμε το παρακάτω αποτέλεσμα σε αυτή την περίπτωση.

Εικόνα 4.5.1: Παράδειγμα αποτυχημένου MultiLayering

Στην παραπάνω εικόνα, φτιάξαμε το τραπεζάκι μικρότερο και βάλαμε το πάνω μέρος του σε Layer που θα σχεδιαστεί πάνω από τα Lifeforms. Όμως όπως βλέπουμε, όταν ο χαρακτήρας μας πάει πολύ κοντά στο τραπέζι, τότε το πάνω μέρος του τραπεζιού σχεδιάζεται πάνω στον χαρακτήρα όπως το ρυθμίσαμε. Εννοείται ότι αυτό δεν πρέπει να μείνει έτσι γιατί η πιθανότητα να γίνει αυτό δεν είναι μικρή. Δυστυχώς το κλασικό σπάσιμο όπως είδαμε δεν λειτουργεί γιατί ο χαρακτήρας είναι πολύ ψηλός και φτάνει το Layer που σχεδιάζεται από πάνω του. Χρειαζόμαστε κάτι δυναμικό που θα λαμβάνει υπόψη την τοποθεσία των χαρακτήρων και θα σχεδιάζεται αντίστοιχα είτε πίσω του είτε μπροστά. Για αυτό η μηχανή παρέχει τα TileObjects.

Τα TileObjects είναι αντικείμενα όπως τα InteractFields μόνο που δεν σηκώνουν κανένα συμβάν (onCollision κτλ) και έχουν εμφανή μορφή. Είναι σαν απλά Tiles μόνο που για την μηχανή είναι αντικείμενα όπως τα InteractFields και τα Lifeforms. Τα TileObjects ως προς το πως σχεδιάζονται, συμπεριφέρονται σαν να είναι Lifeforms. Δηλαδή η μηχανή τα βάζει ταξινομεί ως το ύψος που βρίσκεται το καθένα και ανάλογα τα σχεδιάζει μπροστά ή πίσω. Δεν είναι συμπαγής και τα Lifeforms περνάνε από μέσα τους.

Για να βάλουμε ένα TileObject, γίνεται όπως και με τα InteractFields. Χρειαζόμαστε ένα ObjectLayer που θα τα βάλουμε. Ύστερα επιλέγουμε το Tile που θέλουμε να έχει το TileObject και από την γραμμή εργαλείων επιλέγουμε το “Insert Tile (T)”. Τώρα όπου κάνουμε κλικ θα μπει το Tile. Επειδή όμως είναι αντικείμενο, δεν μπαίνει στοιχισμένο με τα άλλα Tiles. Έτσι χρειάζεται να έχουμε πατημένο το Ctrl όταν κάνουμε κλικ για να μπει στοιχισμένο εκεί που θέλουμε. Αφού τα βάλουμε, τότε βλέπουμε τι έχουμε βάλει στο ObjectLayer με κενές γραμμές. Εδώ καλύτερα να μην δώσουμε ονόματα για να μην παρεμβαίνουν μέσα στον χάρτη. Παρόλο αυτά πρέπει να δώσουμε τον τύπο. Εδώ τύπο θα δώσουμε “Static”. Και εδώ τελείωσαν όλα με αυτό το κομμάτι. Δεν χρειάζεται να δώσουμε ιδιότητες καθώς δεν παίρνει καμία. Πρέπει να έχετε στον νου ότι επειδή είναι αντικείμενα και αυτά, θα σχεδιαστούν στην σειρά το “_OBJ_” του Layer Priority.

Εικόνα 4.5.2:
Αποτέλεσμα χρήσης TileObjects

Ένα τελευταίο πράγμα που πρέπει να έχετε στον νου είναι ότι τα TileObjects δεν πρέπει να γίνονται κατάχρηση και να χρησιμοποιούνται όταν δεν μπορεί να βρεθεί λύση με χρήση απλών TileLayers. Αυτό μην σας γίνει σαν απαγορευτικός κανόνας, απλά να σκέφτεστε ότι και αυτά παίρνουν μέρος στην λογική του παιχνιδιού σε αντίθεση με τα απλά Tiles, οπότε η κατάχρηση θα επιβαρύνει το παιχνίδι χωρίς λόγο. Αυτό ισχύει για όλα τα αντικείμενα, απλά με τα TileObjects είναι πολύ πιο εύκολο να παρασυρθεί κανείς.

4.6 Αντικείμενα NPC

Μέχρι στιγμής είδαμε πως βάζουμε στον χάρτη στατικά αντικείμενα και το περιβάλλον. Τώρα ας δούμε πως βάζουμε άλλα Lifeforms και ας ξεκινήσουμε από τα NPC (Non Player Controled). Θεωρητικά και οι εχθροί θεωρούνται NPC αλλά εδώ θα τους διαχωρίσουμε στους άκακους(NPC) και τους κακούς(Enemy). Τα NPC μπορούμε να τα βάλουμε μέσω κλήσεων από Lua στην μηχανή άλλα και πιο εύκολα από το Tiled. Εδώ θα δούμε πως τα βάζουμε από το Tiled καθώς μέσω της Lua είναι πολύ απλό (μια κλήση).

Για να μπορούμε να βάλουμε μέσω του Tiled, χρειάζεται να χτίσουμε ένα υπόβαθρο στην Lua. Συνήθως τα άκακα NPC είναι μοναδικά στον κόσμο σε αντίθεση με τους εχθρούς. Ανάλογα το τι θέλουμε να φτιάξουμε (Άκακο ή εχθρό), χρειάζεται να φτιάξουμε και διαφορετικό υπόβαθρό. Για να ξεδιαλύνουμε λίγο τα πράγματα, έτσι όπως είναι η μηχανή κάθε LifeformClass χρειάζεται και ένα αρχείο Lua που επιστρέφει έναν πίνακα με επιπλέον δεδομένα γύρω από το LifeformClass για να μπορούμε εύκολα μέσω του Tiled να βάζουμε τα NPC και τους εχθρούς.

Για αυτό τον λόγο υπάρχει η SZLL μαζί με την μηχανή για να παρέχει τα βασικά αρχεία Lua που χρειάζονται να τα παίρνει ο προγραμματιστής και να εμπλουτίζει στα μέτρα του. Θα τα δούμε και αυτά άλλα ας δούμε τι δεδομένα χρειάζεται ο πίνακας.

Ας φτιάξουμε την Μαρία Μπρίλιου. Έχουμε ήδη φτιάξει το AnimationClass της και έχουμε έτοιμο το LifeformClass. Ακόμα και έτσι μπορούμε να την βάλουμε στο παιχνίδι κανονικά μέσω της Lua. Εφόσον τα έχουμε όλα έτοιμα ας φτιάξουμε το αρχείο Lua της Μαρίας.

Φτιάχνουμε έναν φάκελο “Npc/Maria” (δεν υπάρχει δέσμευση σε αυτό) και βάζουμε εκεί το XML του LifeformClass της. Φτιάχνουμε στον ίδιο φάκελο ένα αρχείο “init.lua”. Το αρχείο πρέπει να έχει την ελάχιστη παρακάτω μορφή:

Πίνακας 4.6.1: Παράδειγμα αρχείου Lua Npc

```
local Maria = {
     Class = "Npc/Maria/Class.xml",
     Animations = {}
}

return Maria
```

Όπως είπαμε το αρχείο μόλις εκτελεστεί πρέπει να επιστρέψει έναν πίνακα με κάποια δεδομένα. Τα πεδία του πίνακα που λαμβάνει υπόψη η μηχανή είναι:

- **Class:** Απαραίτητο πεδίο που πρέπει να έχει την συμβολοσειρά με την διαδρομή του αρχείου του XML του LifeformClass του NPC.
- **Animations:** Προαιρετικός πίνακας που θα περιέχει τα OffAnimations. Θα τα δούμε παρακάτω. Ακόμα και να παραλειφθεί δεν θα υπάρχει πρόβλημα.

Αυτό το αρχείο πλέον αποτελεί το Module του NPC. Με αυτό το Module, η χρήση του NPC μέσα στην Lua γίνεται πιο εύκολη καθώς μπορούμε να το καλέσουμε με μια κλήση της “require” και να εξασφαλίσουμε την μοναδικότητα του Module σε όλο το παιχνίδι.

Τώρα το NPC είναι έτοιμο να το προσθέσουμε μέσω του Tiled. Όπως και με τα άλλα αντικείμενα χρειάζεται ένα ObjectLayer. Προσθέτουμε ένα μικρό τετραγωνάκι εκεί που θέλουμε να είναι το NPC. Του δίνουμε το όνομα του στην κενή γραμμή και στον τύπο βάζουμε “Npc”. Τώρα πρέπει να βάλουμε μια ιδιότητα σε αυτό το αντικείμενο.

- **Module:** Πρέπει να του δώσουμε το Module που θα φορτώσει για αυτό το NPC. Εδώ ακολουθείται η νοοτροπία της Lua. Δηλαδή δεν θα δώσουμε απευ-

θείας την διαδρομή του αρχείου άλλα θα το δώσουμε σε μορφή πακέτων με τελείες, δηλαδή στο παράδειγμα μας θα δώσουμε “Npc.Maria”.

Τώρα αν φορτώσουμε τον χάρτη, η Μαρία θα εμφανιστεί εκεί που την βάλαμε στο Tiled με ότι έχει ορίσει το LifeformClass.

4.7 Αντικείμενα Enemy

Πέρα από τα άκακα NPC, υπάρχουν και οι εχθροί που χρειάζονται διαφορετικά δεδομένα στο Module τους. Στα Enemy, χρειάζεται το Module να φιλοξενήσει το LifeformClass του εχθρού. Αυτό σημαίνει ότι η μοναδικότητα του Module είναι σημαντική. Ας υποθέσουμε ότι έχουμε ένα LifeformClass για ένα φίδι στον φάκελο “Enemy/Snake”. Εκεί φτιάχνουμε το αρχείο “init.lua”. Η μορφή του πρέπει να είναι:

Πίνακας 4.7.1: Παράδειγμα αρχείου Lua Enemy

```lua
local Snake = {
     Class = Zeta.LifeformClass('Enemy/Snake/Class.xml'),
     RespawnTime = 30,
     VanishTime = 10
}

Snake.Class:setTable(Snake)

return Snake
```

Όπως βλέπουμε δεν είναι απλά όπως στο NPC. Εδώ έχουμε δύο κλήσεις προς τον πυρήνα της μηχανής. Πιο συγκεκριμένα τα πεδία:

- **Class:** Απαραίτητο πεδίο. Εδώ πρέπει να βάλουμε ένα αντικείμενο LifeformClass που έχει φορτώσει το XML αρχείο του LifeformClass που θέλουμε. Το αντικείμενο αυτό είναι του πυρήνα της μηχανής. Με την κλήση “Zeta.LifeformClass(...)” καλούμε τον Constructor μέσα στην C++ και επιστρέφουμε το αντικείμενο μέσα στο πεδίο Class. Αυτό μας δίνει πολλές δυνατότητες όπως θα δούμε παρακάτω.
- **RespawnTime:** Προαιρετικό πεδίο που θέτουμε τον χρόνο που θα ξαναζωντανέψει ο εχθρός αφού πεθάνει. Ο χρόνος είναι σε δευτερόλεπτα. Αν δεν οριστεί, ορίζεται αυτόματα σε 30 δευτερόλεπτα.
- **VanishTime:** Προαιρετικό πεδίο που ορίζει τον χρόνο που το Sprite του νεκρού εχθρού θα εξαφανιστεί από τον κόσμο αφού πεθάνει. Ο εχθρός εξαφανίζεται μόνο δεν διαγράφεται, που σημαίνει ότι όταν έρθει η ώρα να ξαναζωντα-

νέψει, θα επανέλθει κανονικά. Αν δεν οριστεί θα οριστεί αυτόματα σε 10 δευτερόλεπτα.

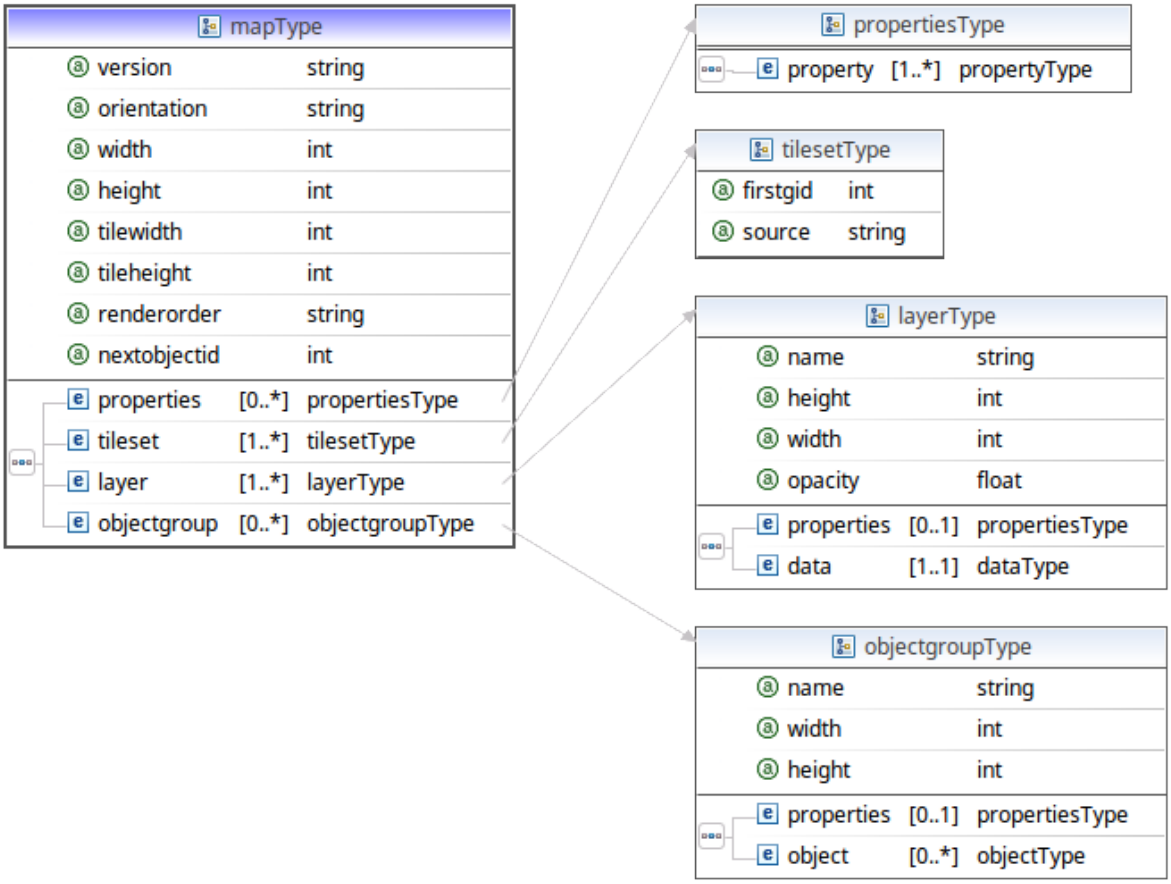
Η κλήση “Snake.Class:setTable(...)” είναι εσωτερική κλήση της μηχανής, που ορίζει στο LifeformClass ποιο είναι το Module του. Είναι προαιρετικό αλλά καλό θα ήταν να γίνεται.

Με αυτά τα στοιχεία, το Module είναι έτοιμο για χρήση. Η χρήση από εδώ και πέρα είναι η ίδια με του NPC, μόνο που όταν φτιάχνουμε ένα αντικείμενο στο Tiled δεν βάζουμε “Npc” στον τύπο αλλά “Enemy”.

Με τον ίδιο τρόπου μπορούμε να προσθέσουμε πολλά ίδια Enemy στον χάρτη. Στο παιχνίδι το καθένα από αυτά θα είναι ξεχωριστή οντότητα.

4.8 Σύνοψη του Map.xsd

Το XML Schema που χρησιμοποιεί η μηχανή για την επαλήθευση των χαρτών, είναι προσαρμοσμένο στα μέτρα της μηχανής. Αυτό σημαίνει ότι υπάρχουν κανόνες ως προς το τι χάρτες μπορείς να φτιάξεις με το Tiled. Για παράδειγμα δεν επιτρέπεται να βάλεις ενσωματωμένα Tilesets στον χάρτη κάτι που το Tiled επιτρέπει. Για αυτό παρακάτω θα δούμε τι κανόνες υπάρχουν γύρω από τους χάρτες αναλύοντας το XSD Schema.



Εικόνα 4.8.1: Ορισμός του Map XSD 1

Αρχικά να πούμε ότι υπάρχουν πολλά στοιχεία στο XSD Schema που αγνοούνται από την μηχανή και χρησιμεύουν κυρίως στο Tiled. Επειδή το Tiled τα εξάγει αυτά τα στοιχεία έτσι κι αλλιώς, χρειάζεται και η μηχανή να τα δέχεται χωρίς να απορρίπτει τον χάρτη.

<map>: Το πρώτο στοιχείο. Περιέχει:

version (Απαραίτητο): Στοιχείο του tiled που το αγνοεί η μηχανή.

orientation (Απαραίτητο): Ορίζει τι τύπος χάρτης είναι όπως είχαμε πει όταν αρχίσαμε να τον φτιάχνουμε στο Tiled. Εδώ η τιμή πρέπει να είναι απαραίτητα “orthogonal”, αλλιώς η μηχανή τον απορρίπτει.

height/width (Απαραίτητο): Ορίζονται το πλάτος και ύψος του χάρτη σε Tiles

tilewidth/tileheight (Απαραίτητο): Ορίζεται το πλάτος και ύψος των Tile του χάρτη σε Pixels.

renderorder (Απαραίτητο): Στοιχείο του tiled που το αγνοεί η μηχανή.

nextobjectid (Απαραίτητο): Στοιχείο του tiled που το αγνοεί η μηχανή.

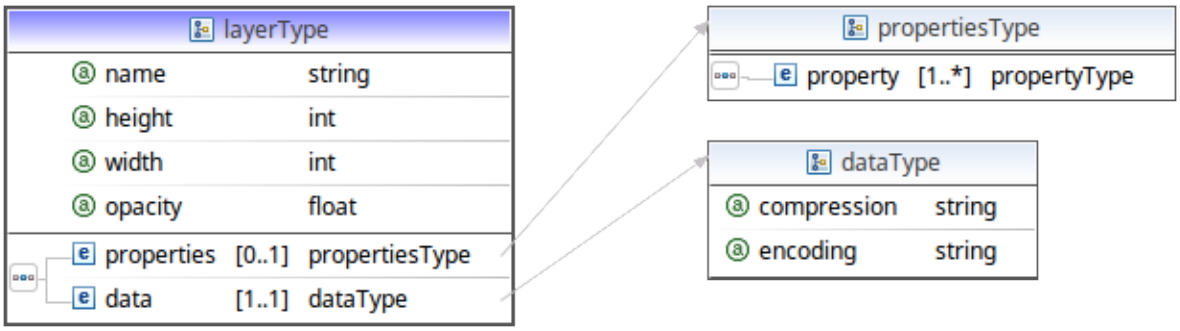
<properties> (0 ή περισσότερα): Ορίζει τις προσαρμοσμένες ιδιότητες του χάρτη. Οι ιδιότητες ορίζονται όπως και στα άλλα στοιχεία.

<tileset> (1 ή περισσότερα): Ορίζει ένα εξωτερικό Tileset. Περιέχει:

firstgid (Απαραίτητο): Ορίζει για το Tileset το καθολικό ID που θα έχει το πρώτο του Tile στον χάρτη. Στην ουσία όλα τα Tileset που συνδέονται στον χάρτη ενώνονται το ένα μετά το άλλο. Με άλλα λόγια αν υπάρχει δεύτερο Tileset, το firstgid που θα έχει είναι ο αριθμός των Tile του πρώτου + 1. Αυτό προστίθεται από μόνο του στο Tiled.

source (Απαραίτητο): Εδώ ορίζεται η διαδρομή του αρχείου του Tileset που θέλουμε να συνδέσουμε στον χάρτη.

<layer> (1 ή περισσότερα): Ορίζει ένα Layer.



Εικόνα 4.8.2: Ορισμός του Layer XSD

name (Απαραίτητο): Ορίζει το όνομα του Layer.

height/width (Απαραίτητο): Ορίζει το ύψος/πλατός σε Tiles

opacity (Απαραίτητο): Ορίζει την διαφάνεια του Layer. Η μηχανή το αγνοεί (για αυτή την έκδοση).

<properties> (Προαιρετικό): Ορίζει τις ιδιότητες του Layer, με τρόπο όπως και τα άλλα στοιχεία του Map.

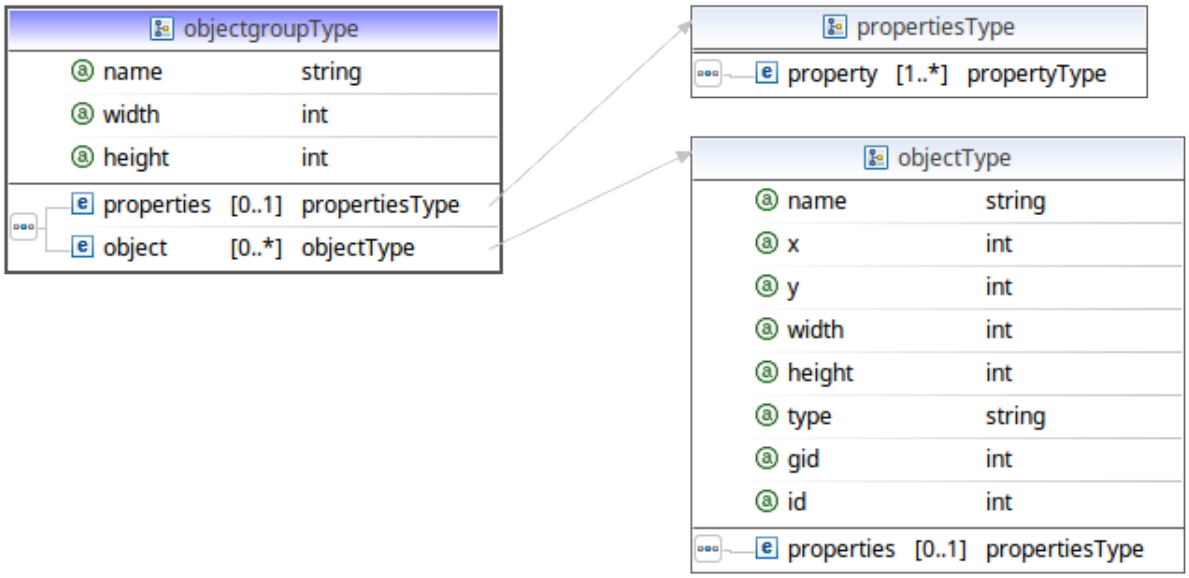
<data> (Απαραίτητο): Ορίζει τα δεδομένα του Layer. Περιέχει:

compression (Απαραίτητο): Ορίζει το είδος της συμπίεσης των δεδομένων. Η μόνη επιτρεπτή τιμή εδώ είναι “zlib”.

encoding (Απαραίτητο): Ορίζει το είδος της κωδικοποίησης των συμπιεσμένων δεδομένων. Η μόνη τιμή που επιτρέπεται εδώ από την μηχανή είναι η “base64”.

Εμφωλευμένα δεδομένα (Απαραίτητο): Μέσα στο στοιχείο <data> υπάρχουν τα κωδικοποιημένα και συμπιεσμένα δεδομένα.

<objectgroup> (0 ή περισσότερα): Ορίζει ένα ObjectLayer.



Εικόνα 4.8.3: Ορισμός του ObjectGroup XSD

name (Απαραίτητο): Ορίζει το όνομα του ObjectLayer.

width/height (Απαραίτητο): Ορίζει το πλάτος/ύψος του Layer.

<properties> (Προαιρετικό): Ορίζει τις ιδιότητες του Layer.

<object> (0 ή περισσότερα): Ορίζει ένα αντικείμενο. Περιέχει:

name (Απαραίτητο): Ορίζει το όνομα του αντικειμένου. Αγνοείται από την μηχανή.

x/y (Απαραίτητο): Ορίζει τις συντεταγμένες του αντικειμένου στον χάρτη.

width/height (Προαιρετικό): Ορίζει τις διαστάσεις του αντικειμένου σε Pixels.

type (Προαιρετικό): Ορίζει τον τύπο του αντικειμένου. Για την μηχανή δεκτές τιμές είναι αυτές που προαναφέραμε (Static, Npc, Enemy, InteractField). Οτιδήποτε άλλο το αγνοεί.

id (Προαιρετικό): Στοιχείο του tiled που το αγνοεί η μηχανή.

gid (Προαιρετικό): Ορίζεται το Tile που έχει μορφή το Object. Αυτό το πεδίο έχει μόνο νόημα εφόσον το αντικείμενο είναι τύπου “Static”.

<properties> (Προαιρετικό): Ορίζονται οι προσαρμοσμένες ιδιότητες του αντικειμένου.

Εδώ τελειώνει η ανάλυση των δεδομένων του χάρτη. Το πιο συνετό είναι να φτιάξει κάποιος έναν χάρτη με το εργαλείο Tiled παρά να πάρει να αρχίσει να γράφει το αρχείο XML από την αρχή. Για αυτό και δόθηκαν οι οδηγίες στην αρχή για το Tiled.

5 ΔΗΜΙΟΥΡΓΙΑ ΙΚΑΝΟΤΗΤΩΝ

Μέχρι στιγμής έχουμε δει πως φτιάχνουμε LifeformClasses, να φτιάχνουμε χάρτες και να βάζουμε μέσα NPC και εχθρούς. Καθώς όμως η μηχανή επικεντρώνεται σε παιχνίδια ARPG, είναι αναγκαίο τα Lifeforms να μπορούν να επιτεθούν το ίδιο και ο παίχτης. Για να γίνει αυτό, χρειάζεται να δημιουργήσουμε ικανότητες (Abilities) και να τα αποδώσουμε στα Lifeform και στον παίχτη.

Οι ικανότητες χωρίζονται σε τρεις κατηγορίες:

- **Ενεργές Ικανότητες (Active Abilities):** Είναι ικανότητες που ενεργοποιούνται κατά βούληση του Lifeform και έχουν συνήθως στιγμιαία διάρκεια. Οι ενεργές ικανότητες έχουν πολλές ιδιότητες:
  - **Χρόνος φόρτωσης (Cast Time)**, όπου είναι ο χρόνος που χρειάζεται η ικανότητα να φορτώσει πριν βγει το αποτέλεσμα της. Όσο η ικανότητα φορτώνει, το Lifeform που το κάνει δεν μπορεί να κάνει κάτι άλλο ούτε να κινηθεί. Κατά την διάρκεια της φόρτωσης μπορεί να αλλάξει το Animation του Lifeform που την χρησιμοποιεί, καθώς και όταν τελειώσει η φόρτωση.
  - **Χρόνος επαναφόρτισης (Cooldown Time),** που είναι ο χρόνος που χρειάζεται να περάσει μέχρι να μπορεί να ξαναχρησιμοποιηθεί η ικανότητα πάλι.
  - **Κόστος πόρων (Mana Cost),** που ορίζει πόση μαγική ενέργεια απαιτείται για να εκτελεστεί η ικανότητα. Αν εκτελεστεί επιτυχώς, η μαγική ενέργεια θα μειωθεί κατά αυτό το ποσό.
  - **Animation για φόρτωση (Casting Animation),** που ορίζεται ποιο Animation πρέπει να κάνει το Lifeform όσο φορτώνεται η ικανότητα.
  - **Ήχος για φόρτωση (Casting Sound),** που ορίζεται ο ήχος που θα παιχτεί όταν θα αρχίσει να φορτώνεται η ικανότητα.
  - **Animation για απελευθέρωση (Release Animation),** που ορίζεται ποιο Animation πρέπει να κάνει το Lifeform όταν τελειώσει το φόρτωμα.
  - **Ήχος για απελευθέρωση (Release Sound),** που ορίζεται ο ήχος που θα παιχτεί όταν τελειώσει το φόρτωμα της ικανότητας.
  - **Εμβέλεια (Range),** πού ορίζεται η ελάχιστη απόσταση που χρειάζεται να απέχει ο εκτελεστής από τον στόχο του προκειμένου να μπορέσει να εκτελεστεί η ικανότητα.

- **Παθητικές Ικανότητες (Passive Abilities):** Είναι οι ικανότητες που δεν εκτελούνται κατά βούληση άλλα παραμένουν ενεργές όσο υπάρχουν στο Lifeform. Εδώ το μόνο που υπάρχει είναι το τι θα γίνει όταν αποδοθεί στο Lifeform και τι θα γίνει όταν πάψει να το έχει το Lifeform.
- **Αναζωογονήσεις (Regenerations):** Είναι κάτι αντίστοιχο με τις παθητικές ικανότητες, μόνο που πέρα από το τι θα γίνει κατά την εισαγωγή και εξαγωγή, πρέπει να οριστεί και τι θα γίνεται σε κάθε χτύπο που κάνει (Tick). Στην ουσία έχει εσωτερικά ένα ρολόι που κάθε τόσο εκτελεί κάτι πάνω στο Lifeform.

Πέρα από τους τρεις κύριους τύπους ικανοτήτων, υπάρχουν κάποια στοιχεία που είναι κομμάτια των τριών παραπάνω.

- **Επιδράσεις (Effects):** Οι επιδράσεις είναι καταστάσεις που αποκτούν τα Lifeforms από διάφορες ικανότητες. Μπορεί να είναι αορίστου χρόνου, παροδικές (Durable) και τύπου αναζωογόνησης (Over Time). Χρησιμοποιούνται από τις ενεργές και παθητικές ικανότητες.
- **Σωματίδια (Projectiles):** Είναι αντικείμενα που μπορεί εκτοξεύονται όταν ολοκληρωθεί κάποια ενεργή ικανότητα. Αυτά μπορεί να είναι:
  - **Κατευθυντήρια (Directional):** Τους ορίζεται μια κατεύθυνση να πάνε και πηγαίνουν προς αυτή την κατεύθυνση μέχρι να φτάσει στο τέλος η ζωή τους.
  - **Ακολουθητικά (Seeking):** Τους ορίζεται ένας στόχος και πηγαίνουν κατά πάνω του. Ακόμα και αν αρχίσει να κινείται ο στόχος θα τον ακολουθήσει.

Για να φτιάξουμε μια ικανότητα, χρειαζόμαστε να φτιάξουμε το AbilityClass της. Το AbilityClass δεν είναι τίποτα άλλο από ένα αρχείο Module της Lua που περιέχει δεδομένα γύρω από την ικανότητα. Σε αντίθεση όμως με τα Module των NPC που είδαμε παραπάνω, εδώ χρειαζόμαστε να ορίσουμε πολλές φορές και συναρτήσεις που θα τις καλεί η μηχανή όποτε χρειαστεί.

5.1 Οι πίνακες AbilityClass

Ας πάμε να δούμε πως θα φτιάξουμε μια πολύ απλή ικανότητα που θα ο χρήστης θα χτυπάει από κοντά τον αντίπαλο.

Ας υποθέσουμε ότι τα Lifeform που θα χρησιμοποιούν αυτού του είδους την ικανότητα έχουν τα απαραίτητα Animations στα AnimationClasses τους και έχουμε έναν ήχο για όταν η ικανότητα έχει επιτυχία. Έχουμε ήδη πει κάποια πράγματα γύρω από

τις ενεργές ικανότητες, τώρα ας δούμε λίγα πράγματα παραπάνω και πως τις φτιάχνουμε.

Πίνακας 5.1.1: Παράδειγμα πίνακα ενός απλού Active Ability

```
local Strike = {
     Name = "Strike",
     Levels = 5,
     CoolDown = 0.5,
     ManaCost = 0,
     CastTime = 0,
     Range = 80,
     Type = Zeta.AbilityClass.Type.Active,
     ReleaseAnimation = "Slash",
     ReleaseSound = "Abilities/Sounds/swing.ogg",
     Effects = {},
     Tables = {},
     Projectiles = {}
}

function Strike.onApply(Ability, Caster, Target)
     Target:offsetHP(-(Ability:getLevel() * 100))
end

Strike.Internal = Zeta.AbilityClass(Strike)

return Strike
```

Όπως βλέπουμε, ο πίνακας έχει αρκετά πράγματα και έχουμε και μια συνάρτηση. Τα πεδία δεν είναι όλα εδώ αλλά μόνο αυτά που χρειάζονται. Αργότερα όταν θα φτιάξουμε μια ικανότητα με Σωματίδιο/Βλήμα, θα έχουμε πολλά περισσότερα. Όλα τα πεδία είναι προαιρετικά και αν δεν οριστεί κάτι, ορίζεται κάτι προκαθορισμένο.

- **Name:** Ορίζει το όνομα της ικανότητας. Είναι ιδιαίτερα σημαντικό γιατί αυτό το όνομα θα χρησιμοποιούμε για να πούμε ποια ικανότητα θέλουμε το Lifeform να χρησιμοποιήσει. Αν δεν οριστεί, ορίζεται σαν όνομα “Lua Generated Ability Class”. Προσοχή σε αυτό, καθώς αν δημιουργηθούν δύο ίδια ονόματα ικανοτήτων στο ίδιο Lifeform, τότε θα υπάρχει ασάφεια ως προς το τι θα χρησιμοποιηθεί όταν επικαλεστεί. Για αυτό και πρέπει να ορίζεται.
- **Levels:** Ορίζει το πόσα επίπεδα θα έχει η ικανότητα. Κάτι που δεν είπαμε είναι ότι οι ικανότητες έχουν επίπεδα. Αυτό δεν επηρεάζει τη μηχανή αλλά πιο πολύ υπάρχει για να βοηθάει τους προγραμματιστές παιχνιδιών. Διαφορετικά Lifeform μπορεί να έχουν διαφορετικό επίπεδο ικανότητας και να κάνει άλλα πράγματα ανάλογα το επίπεδο. Αν δεν οριστεί, ορίζεται το (1).

- **CoolDown:** Ορίζει τον χρόνο που χρειάζεται για να περάσει μέχρι να μπορεί το Lifeform να ξαναχρησιμοποιήσει την ικανότητα. Ορίζεται σε δευτερόλεπτα. Αν δεν οριστεί θεωρείται το (1.0) δευτερόλεπτο.
- **ManaCost:** Ορίζεται το πόσο Mana χρειάζεται για να χρησιμοποιηθεί η ικανότητα. Εάν χρησιμοποιηθεί, τότε η τιμή αυτή αφαιρείται από το Mana που απομένει στο Lifeform. Αν δεν οριστεί, θεωρείται το (0.0).
- **CastTime:** Ορίζεται το πόσο χρόνο σε δευτερόλεπτα χρειάζεται να φορτώσει η ικανότητα πριν βγει το αποτέλεσμα. Πιο ειδικά, είναι ο χρόνος που σπαταλιέται από την στιγμή που ενεργοποιηθεί η ικανότητα μέχρι να καλεστεί η συνάρτηση “onApply”. Πιο πολλά έχουμε πει παραπάνω. Αν δεν οριστεί, θεωρείται το (0.0) δευτερόλεπτα.
- **Range:** Ορίζουμε την ελάχιστη απόσταση που απαιτείται για να εκτελεστεί η ικανότητα. Εδώ οι μονάδες απόστασης εξαρτιούνται από το μέγεθος των γραφικών που χρησιμοποιούνται. Αν δεν οριστεί, θεωρείται το (80.0).
- **Type:** Ορίζεται ο τύπος. Στην ουσία εδώ θα πούμε στην μηχανή ποιος από τους τρεις τύπους ικανοτήτων είναι η ικανότητα και θα ψάξει τα κατάλληλα πεδία. Οι τρεις τιμές που μπορεί να πάρει είναι:
  - **Zeta.AbilityClass.Type.Active** για ενεργές ικανότητες
  - **Zeta.AbilityClass.Type.Passive** για παθητικές ικανότητες
  - **Zeta.AbilityClass.Type.Regeneration** για αναζωογονήσεις
- **ReleaseAnimation:** Ορίζεται το όνομα του Animation που θα αλλάξει το Lifeform που κάνει την ικανότητα. Αυτό το Animation θα εμφανιστεί αφού τελειώσει το φόρτωμα (Casting). Αν το Animation δεν το έχει το Lifeform, τότε δεν αλλάζει σε τίποτα. Το ίδιο γίνεται και αν δεν οριστεί το πεδίο αυτό.
- **ReleaseSound:** Περιέχει την διαδρομή του αρχείου με τον ήχο που θα παιχτεί αφού τελειώσει το φόρτωμα (Casting). Αν δεν οριστεί, δεν θα παίξει τίποτα. Υπόψη ότι η μηχανή δεν θα σταματήσει τον ήχο αν τελειώσει η ικανότητα. Για αυτό να αποφεύγονται μεγάλοι (σε διάρκεια) ήχοι για να αποφευχθούν οι περίεργες καταστάσεις (ήχος που παίζει ακόμα ας έχει τελειώσει η ικανότητα).
- **Effects:** Πίνακας που περιέχει Effect Tables. Θα δούμε παρακάτω για αυτά.
- **Tables:** Λοιποί πίνακες για χρήση του προγραμματιστή.
- **Projectiles:** Πίνακας που περιέχει Projectile Tables. Θα δούμε παρακάτω για αυτά.

Είδαμε τα πεδία του πίνακα. Όμως ο πίνακας έχει ακόμα ένα πεδίο: την συνάρτηση “onApply”.

Σε όλες τις ικανότητες ανεξαρτήτου τύπου, υπάρχουν τρεις συναρτήσεις: η **onApply**, η **onRemove**, και η **onLevelChange**. Ανάλογα τον τύπο της ικανότητας καλούνται και την αντίστοιχη στιγμή που πρέπει. Στην περίπτωση της ενεργής ικανότητας, η **onApply** θα καλεστεί μόλις αρχίσει το Casting. Η **onRemove** δεν θα καλεστεί καθόλου. Η **onLevelChange** καλείται κάθε φορά που αλλάζει το επίπεδο της ικανότητας.

Όταν καλείται οποιαδήποτε από τις δυο, η συνάρτηση φέρει τρία ορίσματα:

- **Ability (Zeta.Ability):** Είναι το εσωτερικό αντικείμενο της ικανότητας που έχει το Lifeform που την εκτέλεσε. Αυτό εξάγει το API που ορίζει η μηχανή γύρω από το αντικείμενο. Υπόψη ότι αν είναι ενεργή η ικανότητα, μπορούμε εύκολα να το μετατρέψουμε σε “Zeta.ActiveAbility”, για να πάρουμε μεγαλύτερο φάσμα των μεθόδων.
- **Caster (Zeta.Lifeform):** Είναι το εσωτερικό αντικείμενο του εκτελεστή της ικανότητας. Όπως και με όλα τα εσωτερικά αντικείμενα, έτσι και εδώ υπάρχει API στην Lua που χειριζόμαστε το αντικείμενο που θα δούμε παρακάτω.
- **Target (Zeta.Lifeform):** Είναι το εσωτερικό αντικείμενο του στόχου που πηγαίνει η ικανότητα.

Στην προτελευταία γραμμή δημιουργούμε το εσωτερικό αντικείμενο AbilityClass της μηχανής και το βάζουμε στο πεδίο “Internal” προκειμένου να μπορούμε να το αποδίδουμε χωρίς να το δημιουργούμε συνέχεια.

Με αυτά έχουμε έτοιμο το Module της ικανότητας για να την αποδώσουμε σε Lifeforms. Όποτε καλέσουμε το αρχείο στην Lua με την “require” θα έχουμε στο Internal το αντικείμενο έτοιμο να το αποδώσουμε σε Lifeform.

Για τις παθητικές ικανότητες, αυτά που χρειάζονται είναι:

- **Name**
- **Levels**
- **Effects**
- **onApply()**
- **onRemove()**

Η onApply() θα καλεστεί όταν ένα Lifeform αποκτήσει αυτή την ικανότητα και τον onRemove() όταν την χάσει. Πέρα από αυτό, τα Effects αφήνονται για τον προγραμματιστή.

Στις αναζωογονήσεις τα πράγματα είναι λίγο πιο σύνθετα από ότι στις παθητικές ικανότητες. Εδώ χρειάζονται:

- **Name**
- **Levels**
- **Attribute:** Εδώ ορίζεται το όνομα του στατιστικού που θα επηρεάζει η ικανότητα. Αν δεν οριστεί τότε θα οριστεί αυτόματα το “HP”.
- **MaxValue:** Εδώ ορίζεται το όνομα του στατιστικού που περιέχει την μέγιστη τιμή που μπορεί να πάρει η μεταβαλλόμενη τιμή. Αν δεν οριστεί, θεωρείται το “MaxHP”.
- **Modifier:** Ορίζεται το όνομα του στατιστικού που περιέχει την τιμή που θα μεταβάλλει το στατιστικό που ορίζει το “Attribute”, σε κάθε χτύπο του ρολογιού της ικανότητας.
- **TriggerEvery:** Ορίζεται το κάθε πότε θα μεταβάλλεται το “Attribute”. Στην ουσία ορίζουμε το ρολόι της ικανότητας. Η τιμή είναι σε δευτερόλεπτα. Αν δεν οριστεί, ορίζεται αυτόματα σε 5.0 δευτερόλεπτα.

Οι συναρτήσεις “onApply()” και “onRemove()” δεν χρειάζεται να οριστούν, καθώς δεν θα καλεστούν ποτέ.

Για να δώσουμε ένα παράδειγμα εδώ, αν φτιάξουμε μια αναζωογόνηση και δεν ορίσουμε τίποτα, τότε η ικανότητα αυτή θα έχει:

- **Attribute = HP**
- **MaxValue = MaxHP**
- **Modifier = HP5**
- **TriggerEvery = 5.0s**

Αν υποθέσουμε ότι το HP5=10, τότε αυτή η ικανότητα κάθε 5.0 δευτερόλεπτα θα προσθέτει την τιμή του HP5 στο HP μέχρι να φτιάσει την τιμή του MaxHP. Όταν φτάσει στην τιμή αυτή, θα σταματήσει. Αν μειωθεί το HP κάτω του MaxHP θα ενεργοποιηθεί ξανά.

5.2 Οι πίνακες EffectClass

Όπως είδαμε, οι ενεργές ικανότητες έχουν στιγμιαία δράση. Μόλις ενεργοποιηθεί η ικανότητα καλείται η onApply() και τέλος. Η onRemove() δεν καλείται ποτέ. Πολλές φορές όμως χρειάζεται αφού ολοκληρωθεί η ικανότητα να αφήσουμε μια επίδραση πάνω στον στόχο που να κρατήσει για λίγη ώρα ή και για πάντα(!). Οι ενεργές ικανότητες από μόνες τους όπως βλέπουμε δεν μπορούν να μας βοηθήσουν σε αυτό. Χρειαζόμαστε τις επιδράσεις ή αλλιώς Effects.

Οι επιδράσεις είναι καταστάσεις που επιδρούν πάνω σε Lifeforms συνήθως από αποτέλεσμα μια ικανότητας. Οι επιδράσεις μπορεί να είναι παροδικές, παροδικές με περιοδικό σκανδαλισμό και αορίστου χρόνου. Ο χρόνος που θα επιδράσει είναι απολυτά ελεγχόμενος μέσω της Lua και μπορεί ανά πάσα στιγμή να διακοπεί. Αν διακοπεί μια επίδραση, στο επόμενο Frame θα χαθεί από το Lifeform.

Οι επιδράσεις μπορεί να είναι μοναδικές σε κάθε Lifeform αλλά μπορεί και να στοιβάζονται, με Lifeforms να μπορούν να λάβουν την ίδια επίδραση πολλές φορές και ας μην έχει τελειώσει η προηγούμενη, με το συνολικό αποτέλεσα να αθροίζεται (ανάλογα το τι κάνει η επίδραση).

Όπως και με τις ικανότητες, οι επιδράσεις ορίζονται από πίνακες Lua. Ανάλογα με τα πεδία που ορίσουμε στον πίνακα, η μηχανή θα φτιάξει μια EffectClass αντίστοιχα με τις τρεις περιπτώσεις παραπάνω. Ας δούμε τώρα ένα παράδειγμα.

Πίνακας 5.2.1: Παράδειγμα πίνακα Effect

```
local UltraMode = {
     Name = "Ultra Mode",
     Levels = 1,
     Durable = true,
     OverTime = true,
     UpTime = 300,
     Stackable = false,
     TickEvery = 1.0,
}

function UltraMode.onApply(Effect,Owner,Level,Source)
     Owner:addAttributeModifier("PAtk", "UltraMode", 1.5,
Zeta.Modifier.Type.Scalar)
end

function UltraMode.onRemove(Effect,Owner,Level,Source)
     Owner:removeAttributeModifier("UltraMode")
end

function UltraMode.onTick(Effect,Owner,Level,Source)
     Owner:offsetHP(Owner:getAttributeValue("MaxHP") * 0.1)
end
return Zeta.EffectClass(UltraMode)
```

Εδώ έχουμε την περίπτωση που η επίδραση είναι παροδική και με περιοδικούς σκανδαλισμούς. Πιο αναλυτικά:

- **Name:** Ορίζει το όνομα της επίδρασης. Αν δεν οριστεί θα οριστεί αυτόματα το “No Named EffectClass”.
- **Levels:** Όπως και με τις ικανότητες, εδώ ορίζονται το πόσα επίπεδα θα έχει η επίδραση.
- **Durable:** Θέτουμε αν η επίδραση θα είναι πεπερασμένη. Παίρνει τιμές true ή false. Αν δεν οριστεί θεωρείται το false.
- **OverTime:** Θέτουμε αν η επίδραση θα προκαλεί περιοδικούς σκανδαλισμούς κατά την διάρκεια της. Αν ναι, τότε θέτουμε true αλλιώς false. Αν δεν οριστεί θεωρείται το false.
- **UpTime:** Εφόσον η επίδραση είναι παροδική, τότε εδώ πρέπει να ορίσουμε την διάρκειά της σε δευτερόλεπτα. Αν δεν την ορίσουμε θα μπει αυτόματα η τιμή 1.0 δευτερόλεπτα.
- **Stackable:** Εδώ ορίζουμε αν η επίδραση θα μπορεί να αποδοθεί σε Lifeform ακόμα και αν την έχει ήδη. Για παράδειγμα αν το επιτρέπαμε στο παράδειγμα μας, τότε αν ένα Lifeform έπαιρνε δυο φορές την επίδραση αυτή, η κλήση onApply() καλούταν 2 φορές χωρίς να έχει προηγηθεί η onRemove(). Το αποτέλεσμα θα ήταν το Patk του Lifeform να αυξηθεί 1.5*1.5=2.25 φορές που σε ορισμένες περιπτώσεις αυτό θα το έκανε υπερβολικά δυνατό. Για αυτό με αυτήν την σημαία το ελέγχουμε. Δέχεται τιμές true ή false. Αν δεν οριστεί ορίζεται αυτόματα η τιμή false.
- **TickEvery:** Εφόσον η επίδραση προκαλεί σκανδαλισμούς (έχει true στο πεδίο OverTime), τότε πρέπει να ορίσουμε και αυτό το πεδίο. Εδώ ορίζουμε κάθε πόσα δευτερόλεπτα θα καλείται η συνάρτηση OnTick().

Πέρα από τα πεδία, υπάρχει η ανάγκη να οριστούν και οι συναρτήσεις onApply(), onRemove(), και onTick() αν είναι OverTime. Η onApply() καλείται σε όλες τις περιπτώσεις όταν αποδοθεί η επίδραση σε Lifeform. Ομοίως και η onRemove() καλείται όταν αφαιρεθεί η επίδραση από το Lifeform. Η onTick() καλείται κάθε TickEvery δευτερόλεπτα αν έχει οριστεί η σημαία OverTime. Όλες οι συναρτήσεις των επιδράσεων παίρνουν τα παρακάτω ορίσματα:

- **Effect (Zeta.Effect):** Εσωτερικό αντικείμενο που είναι το Effect που προκάλεσε την κλήση. Ανάλογα του είδους την επίδρασης αυτό το όρισμα μπορεί να είναι είτε Zeta.Effect, Zeta.DurableEffect, ή Zeta.OverTimeEffect.
- **Owner (Zeta.Lifeform):** Εσωτερικό αντικείμενο που είναι το Lifeform που έχει την επίδραση και καλέστηκε η συνάρτηση.
- **Level (Number):** Αριθμός που ορίζει το επίπεδο της επίδρασης που επιδρά.
- **Source (Zeta.Ability):** Είναι η ικανότητα που προκάλεσε την επίδραση. Μπορεί να γίνει Cast σε ότι εξειδικευμένη μορφή ικανότητας εφόσον ξέρουμε τι ικανότητα είναι. Προσοχή! Αυτό το όρισμα μπορεί να μην έχει τιμή (nil) αν η επίδραση δεν προήλθε από ικανότητα ή αν η επίδραση αποδόθηκε μέσω της ActiveAbilityClass.

Αφού ορίσουμε τις συναρτήσεις, στο τέλος επιστρέφουμε ένα εσωτερικό αντικείμενο Zeta.EffectClass που παίρνει όρισμα τον πίνακα και το κατασκευάζει. Αυτό χρειάζεται αν θέλουμε να το προσθέσουμε απευθείας σε ένα Lifeform. Αν θέλουμε να το ενσωματώσουμε σε μια ικανότητα, τότε θα πρέπει να βάλουμε τον πίνακα αυτό μέσα στον πίνακα Effects του πίνακα της ικανότητας, όπως θα δούμε παρακάτω.

5.3 Οι πίνακες ProjectileClass

Πολλές φορές κάποιες ενεργές ικανότητες που φτιάχνουμε, χρειάζεται να εκτοξευτεί ένα βλήμα που όταν ακουμπήσει κάποιο στόχο να γίνουν κάποια πράγματα. Η μηχανή δίνει αυτή την δυνατότητα μέσω των ProjectileClass.

Τα ProjectileClass είναι πίνακες αντίστοιχους με αυτούς των ικανοτήτων και των επιδράσεων που ορίζουν ένα βλήμα και την συμπεριφορά. Λόγου αυτού, οι πίνακες των βλημάτων είναι πιο πολύπλοκοι από ότι έχουμε δει μέχρι στιγμής διότι απαιτεί πιο πολλές συναρτήσεις και επιπλέον πίνακες για να λειτουργήσει σωστά.

Υπάρχουν τρεις κατηγορίες βλημάτων.

- **Απλό βλήμα:** Σε αυτή την κατηγορία βλήματος, δίνεται μια συντεταγμένη στον χάρτη και θα αρχίσει να κινείται προς αυτή την συντεταγμένη. Όταν φτάσει στην συντεταγμένη αυτή, τότε καλείται μια συνάρτηση που ορίζεται από τον προγραμματιστή. Αν το βλήμα βγει εκτός οθόνης, καταστρέφεται.
- **Κατευθυντήριο βλήμα 4 θέσεων:** Αυτό το βλήμα μόλις δημιουργείται, αρχίσει να κινείται προς την κατεύθυνση που κοιτάει το Lifeform που το εκτόξευσε. Αν το βλήμα βγει εκτός οθόνης, καταστρέφεται.

- **Ακολουθητικό βλήμα:** Αυτό το βλήμα παίρνει τον στόχο που έχει το Lifeform όταν δημιουργείται και το ακολουθάει οπού πάει. Όταν συγκρουστεί μαζί του, καλείται η αντίστοιχη συνάρτηση.

Σε κάθε μία περίπτωση όταν το βλήμα συγκρουστεί με κάτι, θα καλεστεί η αντίστοιχη συνάρτηση Lua που ορίζεται από τον προγραμματιστή.

Για να φτιάξουμε ένα ProjectileClass, πρέπει να φτιάξουμε τον πίνακα του.

Πίνακας 5.3.1: Παράδειγμα Projectile Lua πίνακα

```
local EnergyBall = {
     Name = "EnergyBall",
     Type = Zeta.Projectile.Type.Seeking,
     AnimationClass = "Demo/AnimationClasses/AuraSphere.xml",
     Speed = 1200,
     DirectionalRotate = true,
     Offsets = {
           Up = { x=0, y=-35 },
           Down = { x=0, y=20 },
           Left = { x=-35, y=1 },
           Right = { x=35, y=1  }
     },
     Tables = {
     }
}

function EnergyBall.onRelease(Projectile)

end

function EnergyBall.onCollision(Projectile, Other)
     Other:offsetHP(Other:getAttributeValue("MaxHP") * 0.25)
end

function EnergyBall.onDestinationReach(Projectile)

end
```

Εδώ βλέπουμε τα πεδία και τις συναρτήσεις που ορίζουμε στους πίνακες ProjectileClass. Πιο ειδικά:

- **Name:** Ορίζεται το όνομα του βλήματος. Το όνομα είναι προαιρετικό. Αν δεν οριστεί ορίζεται το “Unamed Projectile”.
- **AnimationClass:** Ορίζεται το AnimationClass που θα έχει το βλήμα. Εδώ το AnimationClass είναι ίδιο με αυτό των Lifeforms άλλα διαφέρει το τι αναπαριστά το κάθε Action. Πιο συγκεκριμένα:
  - Το πεδίο **Bounding** διατηρεί την λειτουργία του όπως και στα Lifeforms. Είναι το τετράγωνο του βλήματος που θα ελέγχεται για συγκρούσεις.

- Το πεδίο **TargetArea** πρέπει να ορίσει όλες τις τιμές σε 0. Θα αγνοηθεί από την μηχανή άλλα για να μην το απορρίψει πρέπει να δηλωθεί έστω και 0.
- Το πεδίο **Shadow** θα αγνοηθεί.
- Τα προδιαγεγραμμένα Actions αλλάζουν λειτουργία εδώ.
  - Το action __**Movement**__ πρέπει να περιέχει το Animation του βλήματος κατά την κίνηση του.
  - Το action __**Idle**__ πρέπει να περιέχει το Animation που θα έχει το βλήμα όταν έχει φορτίσει και περιμένει να εκτοξευτεί (αφού έχει φορτίσει και μένει ακίνητο).
  - Το action __**Dead**__ πρέπει να περιέχει το Animation που θα έχει το βλήμα όταν φορτίζει (από την στιγμή που δημιουργηθεί, μέχρι να τελειώσει αυτό το Animation όπου θα αλλάξει αυτόματα στο “__Idle__”).
  - Το action __**Death**__ πρέπει να περιέχει το Animation που θα έχει το βλήμα όταν αρχίζει να καταστρέφεται.
- Για να εξηγήσουμε λίγο πως λειτουργούν τα Animation εδώ, όταν ένα Lifeform εκτελέσει μια ικανότητα και δημιουργηθεί το Projectile, τότε αν η ικανότητα έχει φόρτωση (Casting) θα εφαρμοστεί το Animation __**Dead**__. Αν η φόρτωση κρατήσει πιο πολύ από ότι διαρκεί το Animation __**Dead**__, τότε μόλις τελειώσει η διάρκεια του animation αυτού, θα το αντικαταστήσει το Animation __**Idle**__ μέχρι να τελειώσει η φόρτωση της ικανότητας. Όταν το βλήμα εκτοξευτεί, τότε όσο το βλήμα κινείται θα έχει το Animation __**Movement**__. Όταν καταστραφεί, τότε πριν καταστραφεί τελείως θα ολοκληρώσει μια φορά το Animation __**Death**__.
- Οτιδήποτε άλλο Action, θα αγνοηθεί.
- Αν κάποιο από τα προδιαγεγραμμένα Actions δεν οριστεί, θα οριστεί κάτι προεπιλεγμένο (συνήθως σκουπίδια). Για αυτό καλό είναι να ορίζονται όλα ακόμα και έχουν το ίδιο Animation.

- **Speed:** Ορίζει την ταχύτητα του βλήματος. Η ταχύτητα μπορεί να ελεγχθεί μέσα από δοκιμές. Αν δεν οριστεί ορίζεται το 4.0.
- **DirectionalRotate:** Ορίζεται αν το Animation θα περιστρέφεται ανάλογα την κλήση που έχει το βλήμα κατά την κίνηση του. Σε αυτή την περίπτωση δεν χρειάζεται να ορίσουμε τα Animation για όλες τις κατευθύνσεις αλλά μόνο για

την δεξιά, μειώνοντας πολύ την δουλειά που χρειάζεται για το βλήμα. Για αυτό και προτείνεται να χρησιμοποιείται έναντι της προεπιλογής. Πρέπει να τονιστεί ότι μόνο το Animation περιστρέφεται, το ορθογώνιο σύγκρουσης μένει όπως έχει.

Εικόνα 5.3.1: Παράδειγμα DirectionalRotate

- **Offsets:** Πρόκειται για έναν πίνακα που περιέχει 4 πίνακες (Up, Down, Right, Left). Αυτοί οι επιμέρους πίνακες περιέχουν τιμές Offset για (x,y). Όταν δημιουργείται ένα βλήμα, τοποθετείται στις συντεταγμένες του Lifeform που το έφτιαξε. Αυτό το κάνει να φαίνεται σαν να βγαίνει από μέσα του. Πολλές φορές δεν μας αρέσει αυτό και θέλουμε να φαίνεται ότι βγαίνει μπροστά του. Σε αυτό μας βοηθάνε τα Offsets. Οι αριθμοί αυτοί εφαρμόζονται στο βλήμα όταν φτιαχτεί ανάλογα που κοιτάει το Lifeform που το δημιουργεί. Για παράδειγμα, στην παραπάνω εικόνα η Μαριάννα κοιτάζει δεξιά και στο βλήμα της εφαρμόστηκαν τα Offsets που ορίζει ο πίνακας **Right**. Οι αριθμοί που έχει αυτός ο πίνακας πήγαν και προστέθηκαν στις συντεταγμένες του βλήματος και έτσι μετακινήθηκε λίγο δεξιά γιατί το x=35, που σημαίνει ότι η συντεταγμένη x του βλήματος αυξήθηκε κατά 35 και έτσι πήγε πιο δεξιά, όπως και λίγο κάτω (λόγο του y=1).
- **Tables:** Προσαρμοσμένος πίνακας για χρήση του προγραμματιστή. Αγνοείται.
- **onRelease():** Η συνάρτηση αυτή καλείται μόλις αρχίσει να κινείται το βλήμα. Παίρνει όρισμα το

    - **Projectile:** που είναι εσωτερικό αντικείμενο που αντιπροσωπεύει το βλήμα που ξεκίνησε την πορεία του.
- **onCollision():** Η συνάρτηση αυτή θα καλεστεί κάθε φορά που το βλήμα συγκρούεται με ένα Lifeform. Παίρνει ορίσματα:
    - **Projectile:** που είναι εσωτερικό αντικείμενο που αντιπροσωπεύει το βλήμα που συγκρούστηκε.
    - **Other:** που είναι εσωτερικό αντικείμενο Lifeform που συγκρούστηκε με το βλήμα.
- **onDestinationReach():** Η συνάρτηση αυτή καλείται ανάλογα το είδους του βλήματος:
    - Αν είναι απλό βλήμα, τότε καλείται όταν φτάσει στις συντεταγμένες που του ορίσαμε.
    - Αν είναι βλήμα 4 θέσεων, τότε καλείται όταν βγει εκτός οθόνης, λίγο πριν καταστραφεί.
    - Αν είναι ακολουθητικό βλήμα, τότε θα καλεστεί είτε όταν συγκρουστεί με τον στόχο, είτε όταν τον φτάσει και δεν είχε σύγκρουση (πολύ σπάνιο φαινόμενο). Στην ουσία καλείται μαζί με την **onCollision()** όταν συγκρουστεί με τον στόχο.

Με αυτά τελειώνουμε όσο αφορά τα βλήματα. Να υπενθυμίσουμε ότι αυτοί οι πίνακες κανονικά είναι να τοποθετούνται στον πίνακα **Projectiles** του πίνακα μιας ενεργής ικανότητας. Από μόνοι τους δεν κάνουν τίποτα.

5.4 Ολοκληρωμένο παράδειγμά

Μέχρι στιγμής έχουμε πει επιμέρους τα κομμάτια μια ικανότητας αλλά δεν τα έχουμε χρησιμοποιήσει μαζί. Παρακάτω θα δούμε ένα παράδειγμα από το Demo της μηχανής που περιέχει μια ενεργή ικανότητα με ένα βλήμα που εκτοξεύεται ακολουθώντας τον στόχο και του προκαλεί αρχική ζημιά ενώ παράλληλα του εφαρμόζει ζημιά με την πάροδο του χρόνου.

Πίνακας 5.4.1: Ολοκληρωμένο παράδειγμα ικανότητας

```
local AuraSphere = {
	Name = "Αιθέρια Σφαίρα",
	Levels = 5,
	CoolDown = 2.5,
	ManaCost = 50.0,
	CastTime = 2.5,
	Range = 600,
	Type = Zeta.AbilityClass.Type.Active,
	CastAnimation = "Cast",
	ReleaseAnimation = "Release",
	ReleaseSound = "Demo/Abilities/Sounds/shoot.ogg",
	CastSound = "Demo/Abilities/Sounds/charge.ogg",
	Tables = {
		DOTDamage = {20,50,80}
	}
}

AuraSphere.Effects = {
	DOT = {
		Levels = 3,
		Durable = true,
		OverTime = true,
		UpTime = 5,
		Stackable = false,
		TickEvery = 1,
		onTick = function(Effect,Owner,Level,Source)
			Owner:offsetHP(-(AuraSphere.Tables.DOTDamage[Level]))
		end
	}
}

AuraSphere.Projectiles = {
	Sphere = {
		Name = "Sphere",
		Type = Zeta.Projectile.Type.Seeking,
		AnimationClass = "Demo/AnimationClasses/AuraSphere.xml",
		Speed = 1200,
		DirectionalRotate = true,
		Offsets = {
			Up = { x=0, y=-35 },
			Down = { x=0, y=20 },
			Left = { x=-35, y=1 },
			Right = { x=35, y=1  }
		},
		onCollision = function(Projectile, Other)
			Other:offsetHP(Other:getAttributeValue("MaxHP") * 0.25)
			Projectile:getParentAbility():applyEffect("DOT",Other,1)
		end
	}
}

function AuraSphere.onApply(Ability,Caster,Target)
	Ability:invokeProjectile("Sphere",0,0)
end

AuraSphere.Internal = Zeta.AbilityClass(AuraSphere)
return AuraSphere
```

Εδώ σπάσαμε τους ορισμό της ικανότητας, της επίδρασης και του βλήματος για χάρη σαφήνειας. Όπως βλέπουμε, σε κάθε περίπτωση ορίσαμε μόνο τις συναρτήσεις που χρειαζόμαστε. Στο τέλος δημιουργούμε την AbilityClass και την αποθηκεύουμε στο πεδίο **Internal.** Στο τέλος επιστρέφουμε το Module για να μπορούμε να το ανακαλέσουμε μέσω της require.

Για να μπορέσουμε να την αποδώσουμε σε ένα LifeformClass, μπορούμε να το κάνουμε με τον παρακάτω κώδικα:

Πίνακας 5.4.2: Παράδειγμα απόδοσης Ικανότητας σε Lifeform

```
Class = Zeta.LifeformClass(Path)
Class:setTable(self)
Class:addAbility(require('Demo.Abilities.AuraSphere').Internal,1)
```

Εδώ ενώ φτιάχνουμε μια LifeformClass από το Path, του αποδίδουμε τον πίνακα και ύστερα του αποδίδουμε την ικανότητα. Την ικανότητα την παίρνουμε από το Module που φτιάξαμε, από το πεδίο Internal που βάλαμε το AbilityClass. Τώρα ότι Lifeform φτιαχτεί από αυτό το LifeformClass, θα έχει αυτή την ικανότητα.

Μπορούμε να προσθέσουμε ικανότητες και μεμονωμένα σε κάποια Lifeforms χωρίς να πειράξουμε την κλάση τους. Αυτό γίνεται με αντίστοιχη κλήση στο Lifeform που θέλουμε.

6 ΔΗΜΙΟΥΡΓΙΑ ΑΠΟΣΤΟΛΩΝ

Η μηχανή παρέχει κάποιους μηχανισμούς που μας βοηθάνε να φτιάξουμε αποστολές (Quests). Οι μηχανισμοί αυτοί βοηθάνε απλά, δεν φτιάχνουν τις αποστολές. Αυτό αφήνεται στον προγραμματιστή καθώς οι απαιτήσεις διαφέρουν εδώ από παιχνίδι σε παιχνίδι.

Οι μηχανισμοί αυτοί είναι:

- Κανάλι Κύριων Συμβάντων Κόσμου (World Event Channel)
- Ακροατές Συμβάντων Lua (Lua Event Listeners)

Το καθένα από αυτά παρέχει βασικές λειτουργίες για έλεγχο γύρω από το τι συμβαίνει στον κόσμο μέσω Lua Callbacks.

6.1 Τα World Event Channels

Όλα τα συμβάντα που συμβαίνουν στο παιχνίδι περνάνε από ένα κομβικό σημείο της μηχανής που τα διαχειρίζεται, τον **WorldManager**. Αυτός κάνει και άλλες δουλειές πέρα από την διαχείριση των συμβάντων αλλά μόνο αυτό θα μας απασχολήσει για τώρα.

Ο WorldManager έχει 7 κύρια κανάλια συμβάντων:

- **LoadBegin:** Αυτό το συμβάν σηκώνεται πριν ξεκινήσει η φόρτωση ενός χάρτη.
- **LoadEnd:** Αυτό το συμβάν σηκώνεται όταν τελειώσει η φόρτωση ενός χάρτη.
- **FrameBegin:** Καλείται πριν ξεκινήσει ένα Frame (Όχι Draw Frame).
- **FrameEnd:** Καλείται όταν ολοκληρωθεί ένα Frame.
- **LifeformDeath:** Καλείται κάθε φορά που ένα Lifeform πεθαίνει.
- **LifeformShow:** Καλείται κάθε φορά που ένα Lifeform εμφανίζεται (είτε από κλήση της **show()** είτε επειδή εμφανίστηκε στην οθόνη)
- **LifeformHide:** Καλείται κάθε φορά που ένα Lifeform εξαφανίζεται (είτε χάνεται από την οθόνη).

Σε κάθε ένα από αυτά τα συμβάντα, μπορεί να αποδοθεί μια συνάρτηση Lua να καλείται κάθε φορά που συμβαίνει. Η απόδοση αυτή γίνεται μέσω του WorldManager Interface στην Lua. Για παράδειγμα:

Πίνακας 6.1.1: Παράδειγμα απόδοσης Callback στον WorldManager

```
local mgr = Zeta.WorldManager:getInstance()
mgr:setCallback(Zeta.WorldManager.Callback.LoadEnd,onLoadEnd)
```

6.2 Τα αντικείμενα LuaListener

Πέρα από τα γενικά συμβάντα που ορίζονται στον WorldManager, υπάρχουν και πιο εξειδικευμένα για τον κόσμο που συνδέονται μέσω των αντικειμένων LuaListener.

Τα αντικείμενα LuaListener είναι εσωτερικά αντικείμενα της μηχανής τα οποία δέχονται μια συνάρτηση Lua και ένα τύπο **Κοσμικού Συμβάντος (WorldEvent)**. Όταν δημιουργείται αυτό το αντικείμενο ενεργοποιείται κατευθείαν και καταχωρεί τον εαυτό του στον WorldManager ώστε όταν γίνει κάποιο συμβάν τύπου αυτού που του δόθηκε, να ενημερωθεί και να καλέσει την συνάρτηση που έχει.

Υπάρχουν 7 τύποι Κοσμικών Συμβάντων. Όλα έχουν κοινό τύπο:

- **Συμβάν Ζημιάς (Damage Event):** Αυτό το συμβάν πρέπει να σηκώνεται όταν ένα Lifeform δέχεται ζημιά. Αυτό το συμβάν δεν σηκώνεται από μόνο του μέσω της μηχανής αλλά θα πρέπει ο προγραμματιστής να το σηκώσει μέσα στο παιχνίδι του όποτε θεωρεί αυτός ότι πρέπει. Η μηχανή δεν επηρεάζει το παιχνίδι από αυτά τα συμβάντα αλλά αφήνεται στον προγραμματιστή. Ο ορισμός γίνεται ως παρακάτω:

Πίνακας 6.2.1: Παράδειγμα δημιουργίας DamageEvent

```
local event = Zeta.WorldEvent()
event:setAsDamageEvent(Victim,Dealer,Amount)
event:broadcast()
```

  Εδώ φτιάχνουμε το συμβάν, ύστερα το μετατρέπουμε σε DamageEvent και του δίνουμε τα ορίσματα που πρέπει. Η κλήση **broadcast()** θα ενημερώσει αλλά τα LuaListener που ακούν τα DamageEvents, ώστε να καλέσουν την συνάρτηση τους. Οι συναρτήσεις Lua πρέπει να δέχονται τα ίδια ορίσματα που δέχεται η **setAsDamageEvent()** γιατί με αυτά θα καλεστεί.
- **Συμβάν Θανάτου (Death Event):** Αυτό το συμβάν σηκώνεται αυτόματα όταν κάποιο Lifeform πεθάνει. Παρόλο ο προγραμματιστής μπορεί να σηκώσει και δικά του τέτοια συμβάντα όποτε κρίνει, το κάνει όπως παραπάνω με την μόνη αλλαγή ότι καλείται με την συνάρτηση **setAsDeathEvent(Victim)**.

- **Συμβάν Αλληλεπίδρασης (Interact Event):** Αυτό το συμβάν πρέπει να σηκώνεται όποτε υπάρχει αλληλεπίδραση μεταξύ Lifeform. Αυτό αφήνεται στον προγραμματιστή για πότε θα σηκωθεί το συμβάν. Ορίζεται από την συνάρτηση **setAsInteractEvent(Lifeform1, Lifeform2)**.
- **Συμβάν Χρήσης Ενεργής Ικανότητας (Ability Use Event):** Αυτό το συμβάν σηκώνεται αυτόματα όταν γίνει χρήση κάποιας ενεργής ικανότητας από οποιοδήποτε Lifeform. Όπως και με όλα τα συμβάντα μπορεί και ο προγραμματιστής να το σηκώσει όποτε θέλει. Ορίζεται στην Lua με την συνάρτηση: **setAsAbilityUseEvent(User, Ability)**.
- **Συμβάν Εξόδου (World Exit Event):** Ειδικό συμβάν που σκοπός του είναι να σηκώνεται όταν ο παίχτης αλλάξει χάρτη. Πρέπει να σηκώνεται από τον προγραμματιστή.
- **Συμβάν Σύγκρουσης (Collision Event):** Αυτό το συμβάν σηκώνεται αυτόματα όταν γίνει κάποια σύγκρουση μεταξύ Lifeform και άλλου ενεργού αντικειμένου (άλλο Lifeform ή InteractField). Μπορεί να σηκωθεί θέτοντας το World Event με την συνάρτηση **setAsAbilityUseEvent(Lifeform, Object)**.
- **Προσαρμοσμένο Συμβάν (Custom Event):** Εδώ ορίζονται προσαρμοσμένα κανάλια στον WorldManager για αποκλειστική χρήση του προγραμματιστή. Δίνεται ένα όνομα στο κανάλι και ένας πίνακας Lua για τα ορίσματα. Από εκεί και πέρα όταν σηκωθεί προσαρμοσμένο συμβάν στο ίδιο κανάλι, θα λειτουργήσει κανονικά. Για παράδειγμα:

Πίνακας 6.2.2: Παράδειγμα Προσαρμοσμένου Συμβάντος

```
local Listener = Zeta.LuaListener("QuestCompleted",function(Quest)

...

end)

...

local event = Zeta.WorldEvent()
event:setAsCustomEvent("QuestCompleted", { Quest })
event:broadcast()
```

Παραπάνω, βλέπουμε την δημιουργία ενός αντικειμένου LuaListener με ορισμούς μια συνάρτηση που έχει όρισμα έναν πίνακα (“Quest”) και δεύτερο όρισμα το κανάλι

(“QuestCompleted”). Παρακάτω βλέπουμε την δημιουργία του προσαρμοσμένου WorldEvent και την διάδοσή του μεσώ της broadcast().

Έχουμε δει πως φτιάχνουμε WorldEvents αλλά λίγα για τα LuaListeners. Τα LuaListeners δημιουργούνται στην Lua αποκλειστικά. Κατά την δημιουργία τους πρέπει να τους ορίσουμε το κανάλι που θα ακούσουν και την συνάρτηση που θα καλέσουν όταν δοθεί στο κανάλι ένα Event. Υπάρχουν δυο τρόποι. Τον ένα τον είδαμε παραπάνω στο παράδειγμα 6.2.2. Οι δυο μορφές είναι:

Πίνακας 6.2.3: Ορισμός κλήσεων LuaListener

```
local Listener = Zeta.LuaListener("Channel",func)
local Listener = Zeta.LuaListener(func,EventType)
```

Στην πρώτη μορφή όπως είπαμε χρησιμοποιείται όταν θέλουμε να ακούσουμε κάποιο προσαρμοσμένο κανάλι. Με την δημιουργία του LuaListener, δημιουργείται και το κανάλι.

Στην δεύτερη περίπτωση το αντικείμενο θα ακούσει σε κάποιο από τα προκαθορισμένα συμβάντα της μηχανής. Το πιο συμβάν θα ακούσει, καθορίζεται από την τιμή του ορίσματος **EventType**. Οι τιμές που μπορεί να πάρει το όρισμα καθορίζονται από το παρακάτω Enumeration:

- **Zeta.WorldEvent.Type.AbilityUse**
- **Zeta.WorldEvent.Type.Collision**
- **Zeta.WorldEvent.Type.Custom**
- **Zeta.WorldEvent.Type.Damage**
- **Zeta.WorldEvent.Type.Death**
- **Zeta.WorldEvent.Type.Interact**
- **Zeta.WorldEvent.Type.Nothing**
- **Zeta.WorldEvent.Type.WorldExit**

Το τι αντιπροσωπεύουν αυτές οι τιμές το είδαμε στην αρχή του κεφαλαίου. Οι τιμές:

- **Zeta.WorldEvent.Type.Custom**
- **Zeta.WorldEvent.Type.Nothing**

Δεν χρησιμοποιούνται και έχουν νόημα μόνο για την μηχανή.

Όταν το αντικείμενο δημιουργηθεί, ενεργοποιείται και μπαίνει αυτόματα στην λίστα των Listeners στον WorldManager. Αν ο χρήστης θέλει να βγάλει προσωρινά το αντι-

κείμενο από τους Listeners και να το επαναφέρει αργότερα, τότε υπάρχουν οι παρακάτω κλήσεις στο αντικείμενο:

- **activate():** Ενεργοποιεί το αντικείμενο και μπαίνει αρχίζει να ακούει. Εκτελείται αυτόματα κατά την δημιουργία.
- **deactivate():** Απενεργοποιεί το αντικείμενο και παύει να ακούει στο κανάλι που έχει μέχρι να ενεργοποιηθεί ξανά.

Πρέπει να τονιστεί (αν και προφανές) ότι όταν το αντικείμενο καταστρέφεται παύει να ακούει. Επίσης αν το αντικείμενο είχε κατασκευάσει κανάλι, το κανάλι δεν καταστρέφεται.

7 ΣΥΝΔΕΟΝΤΑΣ ΤΑ ΚΟΜΜΑΤΙΑ

Ήρθε η ώρα να βάλουμε όλα τα παραπάνω σε λειτουργία. Όπως έχουμε πει στην πολύ αρχή, η μηχανή είναι βιβλιοθήκη της C++. Για να λειτουργήσει χρειάζεται να γράψουμε ένα απλό πρόγραμμα που θα κάνει απαραίτητες κλήσεις στην μηχανή. Το πρόγραμμα του Demo είναι ότι μπορεί να χρειαστεί ένας προγραμματιστής, εκτός αν χρειαστεί κάτι πολύ εξειδικευμένο. Για αυτό θα χρησιμοποιήσουμε το Demo.

Το πρόγραμμα έχει κάποιες Hard Coded διαδρομές αρχείων που διαβάζει. Για αυτό θα πρέπει να βάλουμε τα αρχεία που διαβάζει εκεί που πρέπει. Πιο ειδικά:

- Ο φάκελος **XMLFiles** θα τον έχουμε όπως είναι στον φάκελο που θα έχουμε τα αρχεία του παιχνιδιού (Demo).
- Το αρχείο **Boot.lua** που θα έχει τις αρχικές εντολές που θα ξεκινάει το παιχνίδι.
- Το αρχείο ρυθμίσεων του CEGUI, το **CEGUI_Settings.xml** που έχει τις ρυθμίσεις για το που θα βρει τα αρχεία το CEGUI.
- Το αρχείο των ομάδων αντιπαλότητας **Factions.xml**.

Με αυτά τα αρχεία μέσα στον φάκελο Demo, μπορούμε να τρέξουμε το πρόγραμμα του Demo χωρίς να περάσουμε ορίσματα. Αν παρόλο αυτά θέλουμε να ορίσουμε κάτι δικό μας, αρκεί να ορίσουμε τα παρακάτω ορίσματα:

- **--fps=x**: Όπου x είναι ο αριθμός των καρέ ανά δευτερόλεπτο που θέλουμε να τρέξει το παιχνίδι.
- **--datafolder=path:** Εδώ ορίζουμε τον φάκελο που βρίσκονται τα δεδομένα του παιχνιδιού. Εκεί θα βρει τον φάκελο **XMLFiles**.
- **--bootfile=file:** Εδώ ορίζεται η διαδρομή του αρχείου Lua που θα εκτελεστεί αμέσως μόλις τελειώσει η μηχανή την αρχικοποίηση της και μπορεί να τρέξει το παιχνίδι.
- **--ceguiconf=file:** Εδώ ορίζουμε το XML αρχείο που έχει τις ρυθμίσεις του CEGUI.
- **--gameconf=file:** Εδώ ορίζουμε το αρχείο που έχει τις ρυθμίσεις του παιχνιδιού (Ανάλυση οθόνης, ένταση ήχου, κτλ.) Επειδή αυτό το κομμάτι είναι ημιτελές την στιγμή που γράφεται αυτό, δεν έχει γίνει τεκμηρίωση και η μηχανή μπορεί να το αγνοήσει.

- **--factionfile=file:** Ορίζεται το αρχείο XML που έχει όλα τα Factions (Ομάδες αντιπαλότητας).
- **--help:** Με την εντολή αυτή εμφανίζεται το παραπάνω κείμενο στα Αγγλικά, καθώς και οι προεπιλεγμένες τιμές των ορισμάτων. Αν δοθεί αυτό το όρισμα, τότε δεν θα τρέξει το παιχνίδι.

Με αυτά ορίσματα μπορούμε να προσαρμόσουμε το πρόγραμμα του Demo στα μέτρα μας. Όμως αν θέλουμε πραγματικό έλεγχο, τότε θα χρειαστεί να φτιάξουμε δικό μας πρόγραμμα.

Το παιχνίδι ξεκινάει μόλις τρέξει το **Boot.lua**. Από εκεί και πέρα ο έλεγχος δίνεται στον προγραμματιστή. Αυτός θα επιλέξει τι θα κάνει αυτό, τι θα καλέσει κτλ. Συνήθως (όπως και στο Demo), στο αρχείο αυτό απλά εμφανίζουμε το κύριο Μενού και μετά αυτό ανάλογα τι κλήσεις Lua έχουν καταχωρηθεί στα συμβάντα του CEGUI, θα κάνει κάποιες ενέργειες. Όμως αυτό μπορεί να διαφέρει από παιχνίδι και παιχνίδι. Όποτε θεωρηθεί ότι πρέπει να ξεκινήσει το παιχνίδι, πρέπει να φορτωθεί στον παίχτη μια LifeformClass.

Ο παίχτης είναι μοναδικός, δημιουργείται μια φορά και δεν καταστρέφεται μέχρι να κλείσει το πρόγραμμα. Ακόμα και αν το παιχνίδι έχει σταματήσει και βρίσκεται πχ στο κύριο Μενού, ο παίχτης υπάρχει και αν ξεκινήσει το παιχνίδι ξανά χωρίς να τον αλλάξουμε, θα είναι όπως τον αφήσαμε πριν σταματήσουμε το παιχνίδι. Αυτό που γίνεται είναι να αλλάζει LifeformClass όποτε χρειαστεί.

Αφού φτιάξουμε τον παίχτη όπως θέλουμε, μετά σειρά έχει να φορτώσουμε τον χάρτη που θέλουμε. Αυτό γίνεται μέσω του **WorldManager**. Η κλήση **changeMap(Path)** δέχεται μια διαδρομή αρχείου σε αρχείο tmx (που φτιάχνουμε με το Tiled) και αρχίζει να το φορτώνει. Η φόρτωση είναι βαριά διαδικασία και μπορεί να παγώνει το παιχνίδι, για αυτό και είναι συνετό να διακοπούμε την ροή του παιχνιδιού (pause), να εμφανίζουμε μια οθόνη φόρτωσης (Loading Screen) και ύστερα να ξεκινάμε την φόρτωση. Αφού τελειώσει η φόρτωση (μπορούμε να το ξέρουμε μέσω του συμβάν LoadEnd του WorldManager), τότε μπορούμε να συνεχίσουμε την ροή του παιχνιδιού. Ένα πράγμα που πρέπει να προσέχουμε και να κάνουμε είναι το που θα βρίσκεται ο παίχτης όταν φορτωθεί ο χάρτης. Όπως είπαμε, ο παίχτης διατηρείται και έτσι όταν φορτωθεί ο χάρτης θα βρίσκεται στις συντεταγμένες που είχε στον προηγούμενο. Αυτό ίσως έχει ως αποτέλεσμα ο παίχτης να εμφανιστεί σε ακατάλληλο μέρος στον νέο χάρτη (πχ μέσα σε νερό ή δάσος και να μην μπορεί να κινηθεί). Για

αυτό θα πρέπει αφού τελειώσει η φόρτωση, να μετακινήσουμε τον παίχτη σε επιθυμητές συντεταγμένες (Spawn location). Αυτό γίνεται με την κλήση που έχουν όλα τα Lifeform: **teleportToCordinates(x,y)**. Από εκεί και πέρα το παιχνίδι κυλάει κανονικά όπως έχει προγραμματιστεί.

Αυτά όσο αφορά την σύνδεση. Έχετε στον νου ότι η σύνδεση που είδαμε εδώ είναι η πολύ βασική και η πραγματική θα γίνει από τον προγραμματιστή.

8 ΡΕΠΕΡΤΟΡΙΟ ΕΝΤΟΛΩΝ ΤΗΣ ENGINE ΣΤΗΝ LUA

Εδώ αναλύσουμε περιγραφικά το API της μηχανής που εξάγει στο περιβάλλον της Lua. Στο περιβάλλον Lua υπάρχουν δυο χώροι ονομάτων (Namespaces) με API, το ένα είναι της CEGUI και το άλλο της μηχανής. Στο API της μηχανής, κάθε τύπος και κλήση βρίσκεται στον καθολικό πίνακα **Zeta.** Παρακάτω όταν κάτι βρίσκεται πιο δεξιά, τότε σημαίνει ότι είναι πεδίο του πιο πάνω αριστερά. Πχ στο παρακάτω, το **Success** είναι πεδίο του **Result**, που είναι πεδίο του **Ability**, που είναι πεδίο του **Zeta**. Με άλλα λόγια να το χρησιμοποιήσουμε πρέπει να γράψουμε στην Lua: **Zeta.Ability.Result.Success.** Όπου υπάρχει η λέξη **Καλούμενο,** τότε σημαίνει ότι καλώντας το όνομα της κλάσης σαν συνάρτηση δημιουργεί ένα τέτοιο αντικείμενο (Constructor) πχ.

Πίνακας 8.1: Παράδειγμα κλήσης από Όνομα

```
local Ability = Zeta.AbilityClass(Table)
```

Κάποιες κλάσεις είναι Singleton που σημαίνει ότι μόνο ένα αντικείμενο από αυτές δημιουργείται σε όλο το πρόγραμμα. Για να φέρουμε αυτό το αντικείμενο, αυτές οι κλάσεις έχουν μια Static μέθοδο (συνάρτηση) που καλείται χωρίς αντικείμενο από το όνομα της κλάσης. Στην Lua αυτό γίνει έτσι:

Πίνακας 8.2: Παράδειγμα κλήσης Κλάσης Singleton στην Lua

```
local manager = Zeta.AnimationEffectsManager:getInstance()
```

8.1 Zeta.Ability

Abstract τύπος που κληρονομείται από τις εξειδικευμένες ικανότητες **ActiveAbility, PassiveAbility, Regeneration.** Περιέχει:

- **Result:** Enumeration που επιστρέφει η κλήση **invokeAbility()** της κλάσης Lifeform. Αυτό το αποτέλεσμα είναι το τι συνέβη όταν προσπαθήσαμε να χρησιμοποιήσουμε αυτή την ικανότητα. Έχει τα πεδία:
  - **Success:** Επιστρέφεται όταν η ικανότητα είχε επιτυχία.

    - **onCooldown:** Επιστρέφεται όταν η ικανότητα είχε αποτυχία λόγω ότι χρησιμοποιήθηκε σύντομα και δεν πέρασε το απαιτούμενο χρονικό διάστημα για να ξανά μπορέσει να χρησιμοποιηθεί.
    - **NoTarget:** Επιστρέφεται όταν η ικανότητα είχε αποτυχία λόγω ότι το Lifeform δεν είχε στόχο.
    - **NoMana**: Επιστρέφεται όταν η ικανότητα είχε αποτυχία λόγω ότι το Lifeform δεν είχε αρκετό Mana.
    - **OutOfRange:** Επιστρέφεται όταν η ικανότητα είχε αποτυχία λόγω ότι το Lifeform ήταν πιο μακριά από τον στόχο του από το ελάχιστο που απαιτεί η ικανότητα.
    - **TargetIsDead:** Επιστρέφεται όταν η ικανότητα είχε αποτυχία λόγω ότι ο στόχος είναι νεκρός.
    - **TargetIsNotHostile:** Επιστρέφεται όταν η ικανότητα είχε αποτυχία λόγω ότι ο στόχος δεν βρίσκεται στην λίστα των εχθρικών Lifeform για τον χρήστη της ικανότητας που ορίζεται στο αρχείο Factions.
    - **InvalidAbility:** Επιστρέφεται όταν η ικανότητα είχε αποτυχία λόγω ότι η ικανότητα που ζητήθηκε δεν την έχει το Lifeform.
- **getLevel():** Επιστρέφει το επίπεδο της ικανότητας.
- **setLevel(Level):** Ορίζει το επίπεδο της ικανότητας.
    - **Level(Integer):** Το επιθυμητό επίπεδο.
- **getOwner():** Επιστρέφει το Lifeform που έχει αυτή την ικανότητα.
    - **Επιστρέφει -> Zeta.Lifeform**

8.2 Zeta.AbilityClass

Κλάση που δημιουργεί AbilityClasses.

- **Type (Enumeration):** Απαρίθμηση για τον προσδιορισμό του τύπου Ability που θα παράγει αυτή η AbilityClass
    - **Active:** Ενεργή
    - **Passive:** Παθητική
    - **Regeneration:** Αναζωογόνηση
- **getLevels():** Επιστρέφει τον αριθμό των επίπεδων που θα έχουν οι ικανότητες.

    - **Επιστρέφει -> Integer**
- **getAbilityName():** Επιστρέφει το όνομα που θα έχουν οι ικανότητες και η ίδια η κλάση.
    - **Επιστρέφει -> String**
- **getType():** Επιστρέφει τον τύπο των ικανοτήτων που θα κατασκευάσει η κλάση.
    - **Επιστρέφει -> Type (Enumeration)**
- **getLuaTable():** Επιστέφει τον πίνακα Lua από όπου φτιάχτηκε αυτή η κλάση.
    - **Επιστρέφει -> Lua Table**
- **onApply(Ability, Caster, Target):** Καλεί την συνάρτηση **onApply()** που ορίστηκε στον πίνακα Lua της κλάσης με τα ορίσματα που ορίστηκαν.
    - **Ability(Zeta.Ability):** Η ικανότητα που το προκάλεσε.
    - **Caster(Zeta.Lifeform):** Το Lifeform που το προκάλεσε.
    - **Target(Zeta.Lifeform):** Ο στόχος της ικανότητας.
- **onRemove(Ability, Caster, Target):** Καλεί την συνάρτηση **onRemove()** που ορίστηκε στον πίνακα Lua της κλάσης με τα ορίσματα που ορίστηκαν.
    - **Ability(Zeta.Ability):** Η ικανότητα που το προκάλεσε.
    - **Caster(Zeta.Lifeform):** Το Lifeform που το προκάλεσε.
    - **Target(Zeta.Lifeform):** Ο στόχος της ικανότητας.
- **Καλούμενο(Table):** Δημιουργείται και επιστρέφεται μια AbilityClass από τον πίνακα Lua που δόθηκε σαν όρισμα.
    - **Table(Lua Table):** Ο πίνακας Lua που περιέχει τα δεδομένα για την δημιουργία.
    - **Επιστρέφει -> Zeta.AbilityClass**

8.3 Zeta.ActiveAbility

Αντικείμενο ενεργής ικανότητας που δημιουργήθηκε από AbilityClass και βρίσκεται σε Lifeform. Κληρονομεί όλες τις κλήσεις του **Zeta.Ability.**

- **getClass():** Επιστρέφει την **ActiveAbilityClass** που κατασκεύασε αυτήν την ικανότητα.
    - **Επιστρέφει -> Zeta.ActiveAbilityClass**

- **invokeProjectile(Name, x, y):** Κατασκευάζει ένα βλήμα από την κλάση **Name** και του βάζει επιπρόσθετα τα **x,y** σαν offsets. Η κλάση του βλήματος πρέπει να έχει οριστεί στον πίνακα του AbilityClass που δημιουργήθηκε η ικανότητα. Το βλήμα θα συμπεριφερθεί ανάλογα την κλάση του.
  - **Name(String):** Το όνομα της κλάσης του βλήματος που θα φτιαχτεί
  - **x(Integer):** Αριθμός που θα προστεθεί στην συντεταγμένη **x** του βλήματος.
  - **y(Integer):** Αριθμός που θα προστεθεί στην συντεταγμένη **y** του βλήματος.
- **getRemainingCooldown():** Επιστρέφει έναν αριθμό που είναι ο χρόνος που απομένει μέχρι να επαναφορτίσει η ικανότητα σε δευτερόλεπτα.
  - **Επιστρέφει -> Float**
- **getCastTime():** Επιστρέφει έναν αριθμό που είναι ο χρόνος που χρειάζεται να φορτώσει η ικανότητα όταν έχει χρησιμοποιηθεί, σε δευτερόλεπτα.
  - **Επιστρέφει -> Float**
- **setCastTime(Float):** Θέτει τον χρόνο σε δευτερόλεπτα που χρειάζεται η ικανότητα για να φορτώσει.
  - **Time(Float):** Ο χρόνος σε δευτερόλεπτα.
- **getCoolDown():** Επιστρέφει τον χρόνο που χρειάζεται η ικανότητα να ξανά φορτίσει αφού χρησιμοποιήθηκε, σε δευτερόλεπτα.
  - **Επιστρέφει -> Float**
- **setCoolDown(Time):** Θέτει τον χρόνο που χρειάζεται η ικανότητα να ξανά φορτίσει.
  - **Time(Float):** Ο χρόνος σε δευτερόλεπτα
- **getManaCost():** Επιστρέφει το ελάχιστο Mana που χρειάζεται το Lifeform προκειμένου να χρησιμοποιήσει την ικανότητα.
  - **Επιστρέφει -> Float**
- **setManaCost(Cost):** Ορίζει το κόστος Mana για αυτή την ικανότητα.
  - **Cost(Αριθμός):** Το κόστος σε μονάδες.
- **getRange():** Επιστρέφει την εμβέλεια της ικανότητας σε μονάδες
  - **Επιστρέφει -> Float**
- **setRange(Range):** Ορίζει την εμβέλεια της ικανότητας σε μονάδες.
  - **Range(Float):** Η εμβέλεια σε μονάδες.

- **applyEffect(Name, Target, Level):** Εφαρμόζει την επίδραση που ορίζεται από την κλάση **Name** στο Lifeform **Target**, με επίπεδο **Level**. Η επίδραση θα κατασκευαστεί από κλάση που ορίστηκε στον πίνακα **Effects** του πίνακα Lua της **ActiveAbilityClass** που κατασκευάστηκε αυτή η ικανότητα.
  - **Name(String):** Το όνομα της κλάσης της επίδρασης
  - **Target(Zeta.Lifeform):** Ο στόχος της επίδρασης
  - **Level(Integer):** Το επίπεδο της επίδρασης.

8.4 Zeta.ActiveAbilityClass

Εξειδικευμένο αντικείμενο AbilityClass για ενεργές ικανότητες. Κληρονομεί όλες τις μεθόδους του **Zeta.AbilityClass**. Επιστρέφεται από ενεργή ικανότητα (Zeta.ActiveAbility) όταν καλεστεί η **getClass().**

- **applyEffect(Name, Target, Level, Ability):** Εφαρμόζει την επίδραση που ορίζεται από την κλάση **Name** στο Lifeform **Target**, με επίπεδο **Level** από την ικανότητα **Ability**. Η επίδραση θα κατασκευαστεί από κλάση που ορίστηκε στον πίνακα **Effects** του πίνακα Lua του AbilityClass.
  - **Name(String):** Το όνομα της κλάσης της επίδρασης
  - **Target(Zeta.Lifeform):** Ο στόχος της επίδρασης
  - **Level(Integer):** Το επίπεδο της επίδρασης.
  - **Ability(Zeta.Ability):** Η ικανότητα που το προκάλεσε.

8.5 Zeta.Animation

Ορίζει ένα εσωτερικό αντικείμενο Animation. Αυτά τα αντικείμενα δεν δημιουργούνται από στην Lua αλλά μόνο εσωτερικά στην μηχανή και μπορούν να τροποποιηθούν στην Lua μέσω αυτής της κλάσης.

- **getName():** Επιστρέφει το όνομα του Animation.
  - **Επιστρέφει -> String**
- **isLooping():** Επιστρέφει **true** αν το Animation επαναλαμβάνεται μετά την λήξη του, αλλιώς **false.**
  - **Επιστρέφει -> Boolean**
- **getNumFrames():** Επιστρέφει τον αριθμό των Frames που έχει το Animation.
  - **Επιστρέφει -> Integer**

8.6 Zeta.AnimationEffectsManager

Είναι ένα μοναδικό αντικείμενο στο παιχνίδι (Singleton) που φιλοξενεί όλα τα καθολικά Animation Effects που μπορούν να αποδοθούν σε Lifeforms.

- **getAnimationFX(Name, AnimationPack):** Επιστέφει το Animation που έχει όνομα **Name** από το ομάδα **AnimationPack**. Τα AnimationPack είναι κάθε αρχείο *.anim που φορτώνεται στο AnimationEffectsManager. Ανάλογα πως φορτώθηκε, μπορεί να είναι είτε το όνομα που του δόθηκε κατά την φόρτωση, είτε η διαδρομή του αρχείου του. Το Animation που επιστρέφεται πρέπει να αποδοθεί σε Lifeform.
  - **Name(String):** Το όνομα του Animation όπως ορίζεται μέσα στο αρχείο *.anim.
  - **AnimationPack(String):** Το όνομα της ομάδας που βρίσκεται το Animation.
- **addAnimationPack(Path, Name):** Φορτώνει το αρχείο *.anim που βρίσκεται στο **Path** σαν μια ομάδα και του δίνει το όνομα **Name**. Το όρισμα **Name** αν δεν οριστεί, τότε η ομάδα θα πάρει όνομα το **Path**. Αυτή η κλήση πρέπει να αποφεύγεται κατά την διάρκεια της ροής του παιχνιδιού γιατί η φόρτωση μπορεί να προκαλέσει παγώματα.
  - **Path(String):** Η διαδρομή του αρχείου που είναι*.anim που θα φορτωθεί.
  - **Name(String)(Προαιρετικό):** Το όνομα που θα δοθεί στην ομάδα που θα φορτωθεί. Αν δεν οριστεί τότε δίνεται το **Path**.
- **getInstance():** Επιστρέφει το μοναδικό αντικείμενο **AnimationEffectsManager**. Αυτή η κλήση είναι Static, που σημαίνει δεν καλείται από αντικείμενο, αλλά κατευθείαν από το όνομα της κλάσης.
  - **Επιστρέφει -> Zeta.AnimationEffectsManager**

8.7 Zeta.AnimationHandler

Τα αντικείμενα αυτά διαχειρίζονται τα Animations μέσα σε Lifeforms και Objects (Projectiles, κτλ).

- **QueuePlace (Enumeration):** Αυτή η απαρίθμηση είναι για να προσδιορίσουμε που θα εμφανίζεται κάποιο εξωτερικό Animation (OffAnimation) σε σχέση με το κύριο.
  - **Front:** Το OffAnimation θα εμφανίζεται μπροστά από το κύριο.

    - **Back:** Το OffAnimation θα εμφανίζεται πίσω από το κύριο.
- **setAnimation(Name):** Ορίζει το κύριο Animation να είναι αυτό με το όνομα **Name**. Αυτό το όνομα πρέπει να υπάρχει στο AnimationClass. Αν δεν υπάρχει το Animation αυτό, τότε η μηχανή βάζει το προδιαγεγραμμένο Animation ακινησίας προς τα κάτω.
    - **Name(String):** Το όνομα του Animation που θα μπει στο κύριο.
- **setAnimationClass(Path):** Ορίζει το AnimationClass που ορίζεται στο αρχείο **Path.** Αν έχει οριστεί ήδη μια AnimationClass, τότε αφαιρείται η πρώτη. Η κλήση μπορεί να προκαλέσει πάγωμα προσωρινό στο παιχνίδι αν το AnimationClass δεν έχει φορτωθεί από άλλο AnimationHandler προηγουμένως. Αυτό γιατί το αρχείο θα φορτωθεί από την αρχή, για αυτό και καλό είναι να αποφεύγεται η κλήση κατά την ροή του παιχνιδιού ή να προ φορτώνεται.
    - **Path(String):** Η διαδρομή του αρχείου του AnimationClass που θα φορτωθεί.
- **getMainAnimationPlayer():** Επιστρέφει το αντικείμενο **AnimationPlayer** που έχει το κύριο Animation.
    - **Επιστρέφει -> Zeta.AnimationPlayer**
- **addOffAnimation(Animation, dx, dy, QueuePlace):** Προσθέτει το **Animation** σαν **OffAnimation** και το εμφανίζει με offsets **dx,dy** είτε μπροστά από το κύριο, είτε πίσω ανάλογα την τιμή του **QueuePlace.** Αν το **Animation** δεν επαναλαμβάνεται, τότε όταν τελειώσει αφαιρείται από το AnimationHandler.
    - **Animation(Zeta.Animation):** Το Animation που θα προστεθεί.
    - **dx(Float):** Το Offset που θα προστεθεί στο x του Animation σε σχέση με το κύριο όταν θα σχεδιαστούν.
    - **dy(Float):** Το Offset που θα προστεθεί στο y του Animation σε σχέση με το κύριο όταν θα σχεδιαστούν.
    - **QueuePlace(Enumeration):** Η απαρίθμηση QueuePlace για τον προσδιορισμό της σειράς που θα τοποθετηθεί το Animation.
- **getOffAnimation(Name, QueuePlace):** Επιστρέφει το OffAnimation με όνομα **Name** που βρίσκεται στην σειρά **QueuePlace.** Αν το Animation που ζητείται δεν υπάρχει, τότε θα επιστρέψει **NullOffAnimation.**

- **Name(String):** Το όνομα του Animation που θα ζητηθεί.
- **QueuePlace(Enumeration):** Σε ποια σειρά το προσθέσαμε (είτε μπροστά είτε πίσω)
- **Επιστρέφει -> Zeta.OffAnimation**

8.8 Zeta.AnimationPlayer

Αντικείμενο που δέχεται ένα **Animation** και το αναπαράγει και το σχεδιάζει όποτε ζητηθεί. Με το αντικείμενο αυτό μπορούμε να το ελέγξουμε.

- **isPlaying():** Επιστρέφει **true** αν το Animation παίζει εκείνη την ώρα που καλείται, αλλιώς **false**.
  - **Επιστρέφει -> Boolean**
- **isVisible():** Επιστρέφει **true** αν το Animation εμφανίζεται εκείνη την ώρα που καλείται, αλλιώς **false**. Αυτό δεν έχει να κάνει με το αν εμφανίζεται στην οθόνη αλλά αν θα πρέπει να εμφανίζεται γενικά.
  - **Επιστρέφει -> Boolean**
- **setVisible(Visible):** Θέτει αν το Animation θα πρέπει να σχεδιάζεται όποτε του ζητηθεί.
  - **Visible(Boolean):** Αν δοθεί **true,** τότε θα σχεδιάζεται το Animation, αλλιώς όχι.
- **hide():** Εξαφανίζει το Animation κάνοντάς το να μην σχεδιάζεται. Είναι ισοδύναμο με το **setVisible(false).**
- **show():** Εμφανίζει το Animation αν είχε εξαφανιστεί με κάποια εντολή (πχ **hide()**). Είναι ισοδύναμη με την **setVisible(true).**
- **play():** Ξεκινάει να παίζει το Animation. Δεν συνεπάγεται ότι θα το εμφανίσει αν το έχει εξαφανίσει ο προγραμματιστής πχ με την **hide().**
- **stop():** Σταματάει την αναπαραγωγή του Animation και το επαναφέρει στο αρχικό Frame. Δεν εξαφανίζει το Animation.
- **pause():** Διακόπτει την αναπαραγωγή του Animation παγώνοντας το στο Frame που βρίσκεται εκείνη την ώρα.
- **reset():** Επαναφέρει το Animation στο αρχικό Frame. Δεν το διακόπτει.

8.9 Zeta.AudioContext

Είναι μια Abstract κλάση που ασχολείται με τον ήχο και την μουσική που παίζει.

- **getMainGain():** Επιστρέφει έναν αριθμό από 0.0 έως 1.0 που αναπαριστά την κύρια ένταση του ήχου του παιχνιδιού. Το 1.0 είναι 100% και το 0.0 το 0%
  - **Επιστρέφει -> Float**
- **setMainGain(MainGain):** Θέτει την κύρια ένταση του ήχου του παιχνιδιού.
  - **MainGain(Float):** Ένας αριθμός από 0.0 έως 1.0 που θα αναπαριστά την ένταση. Το 1.0 είναι 100% και το 0.0 το 0%.
- **setMusic(Path):** Θέτει το αρχείο **Path** ως την κύρια μουσική του παιχνιδιού.
  - **Path(String):** Η διαδρομή του αρχείου της μουσικής
- **start():** Ξεκινάει την κύρια μουσική να παίζει, εφόσον έχει οριστεί με την **setMusic()**.
- **stop():** Σταματάει να παίζει την κύρια μουσική εφόσον παίζει.

8.10 Zeta.CEGUIChild

Τα αντικείμενα **CEGUIChild** είναι αντικείμενα που δέχονται ένα αντικείμενο τύπου **CEGUI.Window** και προστίθενται σαν **ChildObject** σε άλλα **Objects** ή **Lifeforms.** Όταν προστεθεί το αντικείμενο σε ένα άλλο, τότε αυτό το ακολουθάει συνέχεια. Το αντικείμενο παιδί παίρνει μαζί του και το **CEGUI.Window** και το μετακινεί μαζί του. Κληρονομεί όλες τις μεθόδους από την **Zeta.ChildObject.**

- **setOffesets(dx, dy):** Ορίζει τα offsets που θα προστεθούν στις απόλυτες συντεταγμένες του αντικείμενου **CEGUI.Window** που έχει το παιδί αυτό καθώς και στο παιδί το ίδιο.
  - **dx(Float):** Το offset που θα προστεθεί στο x του παιδιού και στου **CEGUI.Window**
  - **dy(Float):** Το offset που θα προστεθεί στο y του παιδιού και στου **CEGUI.Window**
- **isDestroy():** Επιστρέφει **true** αν το αντικείμενο **CEGUI.Window** θα καταστραφεί όταν καταστραφεί και το παιδί, αλλιώς **false**.
  - **Επιστρέφει -> Boolean**
- **setDestroy(Destroy):** Θέτει αν το αντικείμενο **CEGUI.Window** θα πρέπει να καταστραφεί όταν καταστραφεί και το παιδί ή όχι.

- **Destroy(Boolean): true** αν θα πρέπει να καταστραφεί το αντικείμενο **CEGUI.Window** μαζί με το παιδί, αλλιώς **false**

- **getOffsetX():** Επιστρέφει την τιμή που θα εφαρμόζεται ως Offset στην συντεταγμένη x.
  - **Επιστρέφει -> Float**
- **setOffsetX(Offset):** Θέτει το Offset για την συντεταγμένη x.
  - **Offset(Float):** Την τιμή Offset που θα δοθεί.
- **getOffsetY():** Επιστρέφει την τιμή που θα εφαρμόζεται ως Offset στην συντεταγμένη y.
  - **Επιστρέφει -> Float**
- **setOffsetY(Offset):** Θέτει το Offset για την συντεταγμένη y.
  - **Offset(Float):** Την τιμή Offset που θα δοθεί.
- **getWindow():** Επιστρέφει το **CEGUI.Window** που έχει αποδοθεί στο παιδί αυτό.
  - **Επιστρέφει -> CEGUI.Window**
- **setWindow(Window):** Θέτει το **CEGUI.Window** για αυτό το παιδί.
  - **Window(CEGUI.Window):** Το Window που θα αποδοθεί.
- **Καλούμενο(Window, Destroy):** Κατασκευάζει και επιστρέφει ένα παιδί και του ορίζει να έχει το **Window** σαν **CEGUI.Window** και την τιμή **Destroy** για αν θα έχουν την ίδια μοίρα παιδί και παράθυρο.
  - **Window(CEGUI.Window):** Το παράθυρο που θα έχει το παιδί.
  - **Destroy(Boolean): true** για αν το **CEGUI.Window** θα πρέπει να καταστραφεί μαζί με το παιδί, αλλιώς **false.**

8.11 Zeta.ChildObject

Abstract κλάση που αντιπροσωπεύει ένα αντικείμενο παιδί ενός **Object,** ή **Lifeform.** Ανάλογα τον τύπο τους, τα παιδιά έχουν διαφορετική συμπεριφορά.

- **getAnimationHandler():** Επιστρέφει το αντικείμενο **Zeta.AnimationHandler** που έχει αυτό το παιδί. Το αντικείμενο αυτό είναι αρμόδιο για τα Animations του παιδιού
  - **Επιστρέφει -> Zeta.AnimationHandler**

- **getParent():** Επιστέφει το **Zeta.Object** που είναι ο πατέρας του παιδιού, ή **nil** αν δεν έχει πατέρα.
  - **Επιστρέφει -> Zeta.Object ή nil**
- **hasParent():** Επιστέφει **true** αν το παιδί αυτό έχει αντικείμενο πατέρα, αλλιώς **false.**
  - **Επιστρέφει -> Boolean**
- **setParent(Parent):** Θέτει τον πατέρα του παιδιού.
  - **Parent(Zeta.Object):** Ο πατέρας που θα αποδοθεί στο παιδί.

8.12 Zeta.ConsoleManager

Singleton κλάση που ασχολείται με εκτέλεση εντολών από String. Χρησιμοποιείται κυρίως για γραμμή εντολών μέσα στο παιχνίδι.

- **excecuteLine(Line):** Εκτελεί την εντολή με τα ορίσματα της που έχει το όρισμα **Line.**
  - **Line(String):** Η εντολή που θα εκτελεστεί
- **getInstance():** Επιστρέφει το μοναδικό αντικείμενο της κλάσης. Κλήση Static.
  - **Επιστρέφει -> Zeta.ConsoleManager**

8.13 Zeta.DurableEffect

Τα αντικείμενα αυτά δημιουργούνται σε Lifeforms και προκαλούν παροδική επίδραση. Μετά το τέλος της επίδρασης, καταστρέφονται από τους κατόχους τους στο επόμενο Frame. Κληρονομούν όλες τις μεθόδους το **Zeta.Effect.**

- **getRemainingTime():** Επιστρέφει τον χρόνο που απομένει μέχρι να τελειώσει αυτή η επίδραση σε δευτερόλεπτα.
  - **Επιστρέφει -> Float**
- **getUpTime():** Επιστέφει τον συνολικό χρόνο που κρατάει η επίδραση σε δευτερόλεπτα.
  - **Επιστρέφει -> Float**
- **setUpTime(UpTime):** Θέτει τον χρόνο που κρατάει η επίδραση σε δευτερόλεπτα.
  - **UpTime(Float):** Ο χρόνος σε δευτερόλεπτα.
- **reset():** Επαναφέρει τον χρόνο που διένυσε η επίδραση στο 0.

8.14 Zeta.Effect

Τα αντικείμενα αυτά δημιουργούνται σε Lifeforms, και προκαλούν επίδραση αορίστου χρόνου.

- **getSource():** Επιστρέφει την ενεργή ικανότητα που προκάλεσε την επίδραση. Αν δεν την προκάλεσε ενεργή ικανότητα, τότε επιστρέφει **nil.**
  - **Επιστρέφει -> Zeta.ActiveAbility**
- **isFinished():** Επιστρέφει **true** αν η επίδραση έχει τελειώσει, αλλιώς **false.**
  - **Επιστρέφει -> Boolean**
- **end():** Τερματίζει την επίδραση.

8.15 Zeta.EffectClass

Είναι αντικείμενα που ορίζουν μια EffectClass που ανάλογα πως έχει ρυθμιστεί παράγει Effects διαφόρων τύπων.

- **getUpTime():** Επιστρέφει τον συνολικό χρόνο που θα κρατήσουν τα Effects σε δευτερόλεπτα (Εάν είναι με πεπερασμένη διάρκεια).
  - **Επιστρέφει -> Float**
- **getTickEvery():** Επιστρέφει κάθε πόσα δευτερόλεπτα θα καλείται η Callback για το tick(Εάν το EffectClass ορίζει OverTime).
  - **Επιστρέφει -> Float**
- **getNewEffect(Target, Level):** Κατασκευάζει και επιστρέφει κάποια κατηγορία **Zeta.Effect** ανάλογα το πως έχει ρυθμιστεί αυτή η EffectClass, με τα ορίσματα **Target** και **Level.** Προσοχή!! Στο Lifeform **Target** δεν αποδίδεται το **Effect** που κατασκευάζεται! Πρέπει να το κάνει ο προγραμματιστής.
  - **Target(Zeta.Lifeform):** Το Lifeform που θα έχει το Effect. Δεν το προσθέτει στο Lifeform. Πρέπει να γίνει μηχανικά.
  - **Level(Integer):** Το επίπεδο της επίδρασης.
  - **Επιστρέφει -> Zeta.Effect**
- **Καλούμενο(Table):** Κατασκευάζει και επιστρέφει μια Zeta.EffectClass ρυθμισμένη από τον πίνακα Lua **Table.**
  - **Table(Lua Table):** Ο πίνακας Lua που θα ρυθμίσει τη EffectClass.
  - **Επιστρέφει -> Zeta.EffectClass**

8.16 Zeta.Enemy

Εξειδικευμένο αντικείμενο Lifeform που όταν πεθαίνει, μετά από ένα δοσμένο χρονικό διάστημα αναγεννιέται.

- **Καλούμενο(LifeformClass, x, y, RespawnTime, VanishTime):** Δημιουργεί και επιστρέφει ένα **Zeta.Enemy** κλάσης **LifeformClass** στις συντεταγμένες **x,y.** Όταν το Lifeform πεθάνει τότε όταν περάσουν **VanishTime** δευτερόλεπτα, το πτώμα του θα εξαφανιστεί από τον κόσμο. Όταν περάσουν **RespawnTime** δευτερόλεπτα από τον θάνατό του, τότε θα αναγεννηθεί.
  - **LifeformClass(String):** Η διαδρομή του αρχείου XML LifeformClass που θα είναι το Lifeform.
  - **x(Float):** Η συντεταγμένη x που θα γεννηθεί το Lifeform.
  - **y(Float):** Η συντεταγμένη y που θα γεννηθεί το Lifeform.
  - **RespawnTime(Float):** Ο χρόνος σε δευτερόλεπτα που θα περάσει μέχρι να αναγεννηθεί. Αν δεν οριστεί ορίζεται το 30s.
  - **VanishTime(Float):** Ο χρόνος σε δευτερόλεπτα που θα περάσει μέχρι το πτώμα να εξαφανιστεί.
  - **Επιστρέφει -> Zeta.Enemy**

8.17 Zeta.InteractField

Αντικείμενο που αντιπροσωπεύει ένα InteractField. Κληρονομεί όλες τις μεθόδους του **Zeta.Object**.

8.18 Zeta.InteractObject

Αντικείμενο αντίστοιχο του InteractField, μόνο που μπορεί να έχει και Animations. Κληρονομεί όλες τις μεθόδους του **Zeta.Object**.

- **setOnClick(Function):** Θέτει την συνάρτηση Lua **Function** να καλεστεί όταν το αντικείμενο το κλικάρει ο παίχτης.
  - **Function(Lua Function):** Η συνάρτηση που θα καλεστεί.
- **setOnCollide(Function):** Θέτει την συνάρτηση Lua **Function** να καλεστεί όταν το αντικείμενο συγκρουστεί από κάποιο άλλο.
  - **Function(Lua Function):** Η συνάρτηση που θα καλεστεί.

- **setAnimation(Name):** Θέτει στο αντικείμενο το Animation με όνομα **Name.** Προσοχή, αυτή η συνάρτηση έχει διαφορετική συμπεριφορά με αυτή της **Lifeform**. Εδώ το αντικείμενο δεν έχει κατεύθυνση που κοιτάει.
  - **Name(String):** Το όνομα του Animation που θα αποδοθεί.
- **Καλούμενο(AnimationClass, x, y):** Κατασκευάζει και επιστρέφει ένα **Zeta.InteractObject**, του αποδίδει το **AnimationClass** και του τοποθετεί στις συντεταγμένες **x,y.**
  - **AnimationClass(String):** Η διαδρομή του αρχείου του AnimationClass που θα έχει το αντικείμενο.
  - **x(Float):** Η συντεταγμένη x που θα τοποθετηθεί το αντικείμενο.
  - **y(Float):** Η συντεταγμένη y που θα τοποθετηθεί το αντικείμενο.
  - **Επιστρέφει -> Zeta.InteractObject**

8.19 Zeta.Lifeform

Τα αντικείμενα αυτά είναι από τα πιο σημαντικά στο παιχνίδι. Η κλάση αυτή εξάγει μεγάλο μέρος του API. Κληρονομεί όλες τις μεθόδους της **Zeta.Object.** Κάθε Lifeform κληρονομεί τις παρακάτω μεθόδους.

- **moveToPosition(x, y):** Κάνει το Lifeform να αρχίσει να κινείται προς στις συντεταγμένες **x, y.** Το Lifeform θα τηρήσει κανονικά τις συμβάσεις όπως συγκρούσεις, επιρροές κτλ. Για αυτό δεν είναι σίγουρο ότι θα φτάσει εκεί (πχ μπορεί να βρει τοίχο). Η ταχύτητα που θα κινηθεί καθορίζεται από το Attribute **Speed**.
  - **x(Float):** Η συντεταγμένη x που θα κινηθεί το Lifeform.
  - **y(Float):** Η συντεταγμένη y που θα κινηθεί το Lifeform.
- **moveToPosition(Vector):** Ίδια με την **moveToPosition(x, y)** μόνο που δέχεται σαν όρισμα ένα αντικείμενο **Zeta.Vector2.**
  - **Vector(Zeta.Vector2):** Το αντικείμενο που περιέχει την συντεταγμένη που θα πάει το Lifeform.
- **getState():** Επιστρέφει ένα αντικείμενο **Zeta.LifeformState** που αντιπροσωπεύει την κατάσταση του Lifeform.
  - **Επιστρέφει -> Zeta.LifeformState**

- **stopMoving():** Κάνει το Lifeform να σταματήσει να κινείται αν κινούνταν. Αυτή η κλήση δεν ακινητοποιεί απλά το Lifeform αλλά θέτει και το κατάλληλο Animation αδράνειας.
- **setClass(Path):** Αντικαθιστά την υπάρχουσα LifeformClass με αυτή που ορίζει **Path.** Το Lifeform θα υποστεί ολική επαναφορά (Αναγεννιέται, κτλ).
  - **Path(String):** Η διαδρομή του αρχείου του LifeformClass που θα αποδοθεί.
- **setAnimationClass(Path):** Αντικαθιστά την υπάρχουσα AnimationClass με αυτή που ορίζει **Path.** Το Lifeform **δεν** επαναφέρεται.
  - **Path(String):** Η διαδρομή του αρχείου του AnimationClass που θα αποδοθεί.
- **setActionAnimation(Name):** Θέτει το κατάλληλο Animation για το Action **Name,** ανάλογα σε τι κατεύθυνση κοιτάζει το Lifeform εκείνη την στιγμή. Αυτά ορίζονται στο AnimationClass που έχει το Lifeform.
  - **Name(String):** Το όνομα του Action που θα πάρει τα Animation.
- **addOffAnimation(Animation, Dx, Dy, QueuePlace):** Προσθέτει το **Animation** στην λίστα **QueuePlace** των OffAnimations για αυτό το Lifeform. Το OffAnimation θα σχεδιαστεί αφού του προστεθούν τα **Dx, Dy** στις συνταγμένες. Έχει ίδιο αποτέλεσμα με την **AnimationHandler.addOffAnimation().**
  - **Animation(Zeta.Animation):** Το Animation που θα προστεθεί. Το Animation αυτό πρέπει να είναι ανεξάρτητο και όχι δεσμευμένο. Συνήθως προέρχεται από το **Zeta.AnimationEffectsManager**.
  - **Dx(Float):** Το offset που θα προστεθεί στην συντεταγμένη x του Animation πριν σχεδιαστεί.
  - **Dy(Float):** Το offset που θα προστεθεί στην συντεταγμένη y του Animation πριν σχεδιαστεί.
  - **QueuePlace(Enumeration):** Σε ποια σειρά θα τοποθετηθεί το OffAnimation. Αυτό Enumeration είναι είδους **Zeta.AnimationHandler.QueuePlace**
- **getAttributeValue(Name):** Επιστρέφει την τιμή που έχει το Attribute **Name.** Προσοχή! Αν δεν υπάρχει το Attribute αυτό, δημιουργείται και παίρνει την τιμή 0.
  - **Επιστρέφει -> Float**

- **setAttributeBaseValue(Name, Value):** Θέτει την τιμή-βάση του Attribute **Name.** Η βάση αυτή είναι η τιμή που πάνω της επιδρούν όλα τα Modifiers.
  - **Name(String):** Το όνομα του Attribute που θα αλλάξει.
  - **Value(Float):** Η τιμή που θα πάρει η βάση.
- **addAttributeModifier(AttributeName, ModifierName, Value, Type):** Δημιουργεί ένα Modifier τύπου **Type,** με τιμή **Value** και το προσθέτει στην λίστα του Attribute **AttributeName** σε εγγραφή με όνομα **ModifierName.** Αργότερα το Modifier αυτό μπορεί να ανακαλεστεί μέσω του Attribute που προστέθηκε (**AttributeName**) και το όνομα του (**ModifierName**).
  - **AttributeName(String):** Το όνομα του Attribute που θα προστεθεί το Modifier.
  - **ModifierName(String):** Το όνομα που θα πάρει το Modifier μέσα στην λίστα του Attribute.
  - **Value(Float):** Η τιμή που θα έχει το Modifier.
  - **Type(Enumeration):** Καθορίζει την συμπεριφορά του Modifier, εάν η τιμή του θα πολλαπλασιαστεί στο Attribute ή θα προστεθεί. Η τιμές του Enumeration αυτού ορίζονται στο **Zeta.Modifier.Type.**
- **getAnimationHandler():** Επιστρέφει τον AnimationHandler που έχει το Lifeform. Αυτό διαχειρίζεται όλα τα Animation του Lifeform.
  - **Επιστρέφει -> Zeta.AnimationHandler**
- **removeAttributeModifier(AttributeName, ModifierName):** Αφαιρεί το Modifier με όνομα **ModifierName** από το Attribute με όνομα **AttributeName.** Αυτό το Modifier πρέπει να έχει προστεθεί προηγουμένως με την **addAttributeModifier()** αλλιώς δεν θα έχει αποτέλεσμα.
  - **AttributeName(String):** Το όνομα του Attribute που βρίσκεται το Modifier
  - **ModifierName(String):** Το όνομα του Modifier που θα αφαιρεθεί.
- **offsetMana(Value):** Προσθέτει την τιμή **Value** στο Attribute με όνομα **Mana.**
  - **Value(Float):** Η τιμή που θα προστεθεί στο Attribute.
- **offsetHP(Value):** Προσθέτει την τιμή **Value** στο Attribute με όνομα **HP.**
  - **Value(Float):** Η τιμή που θα προστεθεί στο Attribute.
- **setMana(Value):** Θέτει την τιμή-βάση του Attribute με όνομα **Mana** σε τιμή **Value.**

    - **Value(Float):** Η τιμή που θα αποδοθεί στην βάση του Attribute.
- **setHP(Value):** Θέτει την τιμή-βάση του Attribute με όνομα **HP** σε τιμή **Value.**
    - **Value(Float):** Η τιμή που θα αποδοθεί στην βάση του Attribute.
- **isHostileWith(Lifeform):** Επιστρέφει **true** αν αυτό το Lifeform είναι εχθρικό προς το **Lifeform,** αλλιώς **false.**
    - **Lifeform(Zeta.Lifeform):** Το άλλο Lifeform που θα ελεγχθεί.
    - **Επιστρέφει -> Boolean**
- **addEffect(Effect):** Προσθέτει την επιρροή **Effect** στο Lifeform.
    - **Effect(Zeta.Effect):** Η επιρροή που θα προστεθεί.
- **canReceiveEffect(EffectClass):** Επιστρέφει **true** αν το Lifeform τηρεί τις προϋποθέσεις για να λάβει Effect που δημιουργεί η **EffectClass,** αλλιώς **false.**
    - **EffectClass(Zeta.EffectClass):** Το EffectClass που θα ελεγχθεί.
    - **Επιστρέφει -> Boolean**
- **isInCombat():** Επιστρέφει **true** αν το Lifeform βρίσκεται σε μάχη**,** αλλιώς **false.**
    - **Επιστρέφει -> Boolean**
- **setInCombat(Value):** Θέτει αν το Lifeform βρίσκεται σε μάχη. Αυτή η τιμή επηρεάζει το AI.
    - **Value(Boolean):** true για κατάσταση μάχης
- **resetState():** Επαναφέρει την κατάσταση του Lifeform σε αδράνεια και κατεύθυνση προς τα κάτω.
- **IsAlive():** Επιστρέφει **true** αν το Lifeform είναι ζωντανό**,** αλλιώς **false.**
    - **Επιστρέφει -> Boolean**
- **getLevel():** Επιστρέφει το επίπεδο του Lifeform.
    - **Επιστρέφει -> Integer**
- **hasTarget():** Επιστρέφει **true** αν το Lifeform έχει κάποιο στόχο**,** αλλιώς **false.**
    - **Επιστρέφει -> Boolean**
- **getLuaTable():** Επιστρέφει τον πίνακα Lua του Lifeform εάν έχει.
    - **Επιστρέφει -> Lua Table**

- **setLuaTable(LuaTable):** Θέτει τον πίνακα Lua για αυτό το Lifeform. Προσοχή! Αυτός ο πίνακας δεν επηρεάζει το Lifeform και είναι για χρήση του προγραμματιστή.
  - **LuaTable(Lua Table):** Ο πίνακας Lua που θα αποδοθεί.
- **addAbility(AbilityClass, Level):** Δημιουργεί και προσθέτει στην λίστα των διαθέσιμων ικανοτήτων μια ικανότητα που δημιουργεί η **AbilityClass** με επίπεδο **Level.**
  - **AbilityClass(Zeta.AbilityClass):** Η AbilityClass που θα χρησιμοποιηθεί.
  - **Level(Integer):** Το επίπεδο της ικανότητας που θα προστεθεί.
- **setTarget(Lifeform):** Θέτει τον στόχο για αυτό το Lifeform. Μπορεί να αφαιρεθεί ο στόχος θέτοντας τον στόχο **nil.**
  - **Lifeform(Zeta.Lifeform):** Το Lifeform που θα γίνει στόχος ή **nil** για τίποτα.
- **getActiveAbility(Name):** Επιστρέφει την ενεργή ικανότητα με όνομα **Name** εφόσον την έχει το Lifeform, αλλιώς επιστρέφει **nil.**
  - **Name(String):** Το όνομα της ικανότητας που θα επιστραφεί.
  - **Επιστρέφει -> Zeta.ActiveAbility**
- **getPassiveAbility(Name):** Επιστρέφει την παθητική ικανότητα με όνομα **Name** εφόσον την έχει το Lifeform, αλλιώς επιστρέφει **nil.**
  - **Name(String):** Το όνομα της ικανότητας που θα επιστραφεί.
  - **Επιστρέφει -> Zeta.PassiveAbility**
- **getRegeneration(Name):** Επιστρέφει την αναζωογόνηση με όνομα **Name** εφόσον την έχει το Lifeform, αλλιώς επιστρέφει **nil.**
  - **Name(String):** Το όνομα της ικανότητας που θα επιστραφεί.
  - **Επιστρέφει -> Zeta.Regeneration**
- **getTarget():** Επιστρέφει τον στόχο που έχει αυτό το Lifeform ή **nil** αν δεν έχει στόχο.
  - **Επιστρέφει -> Zeta.Lifeform**
- **teleportToCordinates(x, y):** Τηλεμεταφέρει το Lifeform στις συντεταγμένες **x,y.** Αυτή η κλήση δεν κάνει το Lifeform να κινηθεί εκεί, άλλα μεταφέρεται εκεί στιγμιαία και άνευ όρων.
  - **x(Float):** Η συντεταγμένη χ που θα μεταφερθεί το Lifeform.
  - **y(Float):** Η συντεταγμένη y που θα μεταφερθεί το Lifeform.

- **die():** Σκοτώνει ακαριαία το Lifeform. Όλες οι απαραίτητες ενέργειες θα γίνουν όσο αφορά τον θάνατο (αλλαγή Animation, σήκωμα WorldEvent κτλ.).
- **face(x, y):** Κάνει το Lifeform να κοιτάξει προς την κατεύθυνση που βρίσκονται οι συντεταγμένες **x, y.**
  - **x(Float):** Η συντεταγμένη x που θα κοιτάξει το Lifeform.
  - **y(Float):** Η συντεταγμένη y που θα κοιτάξει το Lifeform.
- **face(Vector):** Αντίστοιχη με την **face(x, y)** μόνο που δέχεται ένα **Zeta.Vector2.**
  - **Vector(Zeta.Vector2):** Το διάνυσμα που περιέχει την συντεταγμένη που θα κοιτάξει το Lifeform.
- **face(Object):** Κάνει το Lifeform να κοιτάξει το αντικείμενο Object.
  - **Object(Zeta.Object):** Το αντικείμενο που θα κοιτάξει το Lifeform
- **Καλούμενο:** Κατασκευάζει και επιστρέφει ένα κενό Lifeform. Αυτό το Lifeform δεν έχει LifeformClass ούτε AnimationClass.
  - **Επιστρέφει -> Zeta.Lifeorm**
- **Καλούμενο(LifeformClassPath, x, y):** Κατασκευάζει και επιστρέφει ένα Lifeform με LifeformClass από την διαδρομή του αρχείου **LifeformClassPath** και το τοποθετεί στις συντεταγμένες **x,y.**
  - **Επιστρέφει -> Zeta.Lifeorm**
- **Καλούμενο(LifeformClass, x, y):** Κατασκευάζει και επιστρέφει ένα Lifeform με LifeformClass το **LifeformClass** και το τοποθετεί στις συντεταγμένες **x,y.**
  - **Επιστρέφει -> Zeta.Lifeform**

8.20 Zeta.LifeformClass

Τα αντικείμενα LifeformClass διαχειρίζονται τις κλάσεις των Lifeform και μπορεί να δημιουργηθούν μέσα στην Lua. Αυτές οι κλάσεις θέτουν το “καλούπι” για Lifeforms. Όταν αποδοθούν σε Lifeform, πολλά στοιχεία του Lifeform θα αλλάξουν βάση αυτής της κλάσης.

- **getLifeformName():** Επιστέφει το όνομα που θα πάρει το Lifeform
  - **Επιστρέφει -> String**
- **getAnimationClassName():** Επιστρέφει το όνομα του AnimationClass που θα έχει το Lifeform. Συνήθως είναι μια διαδρομή στο XML αρχείο.

- **Επιστρέφει -> String**

- **getLevel():** Επιστρέφει το επίπεδο που θα έχει το Lifeform.
  - **Επιστρέφει -> Integer**
- **setTable(Table):** Θέτει τον πίνακα Lua **Table** που θα έχουν τα Lifeform. Αυτός ο πίνακας δεν επηρεάζει το Lifeform, αλλά αφήνεται στον προγραμματιστή τι θα τον κάνει. Να σημειωθεί ότι τα Lifeform έχουν και μοναδικό ξέχωρο πίνακα.
  - **Table(Lua Table):** Τον πίνακα Lua που θα αποδοθεί
- **setOnClick(Function):** Θέτει την συνάρτηση **Function** να καλεστεί όταν το Lifeform θα το κλικάρει ο παίχτης.
  - **Function(Lua Function):** Η συνάρτηση που να καλεστεί.
- **setOnCollision(Function):** Θέτει την συνάρτηση **Function** να καλεστεί όταν το Lifeform συγκρουστεί με άλλο.
  - **Function(Lua Function):** Η συνάρτηση που να καλεστεί.
- **levelizeStats(LevelMultiplier):** Θέτει όλα τα Attributes (με εξαίρεση το Speed) που θα έχουν τα Lifeform με τυχαίες τιμές βάση του τύπου:

$$Attribute = RAND\left(Level * LevelMultiplier, (Level+1) * LevelMultiplier\right) \quad (5.1.1)$$

  - **LevelMultiplier(Float):** Η τιμή που θα μπει στον τύπο.
- **levelizeStats(Multiplier, UpperBound, LowerBound):** Θέτει όλα τα Attributes (με εξαίρεση το Speed) που θα έχουν τα Lifeform με τυχαίες τιμές βάση του τύπου **3.1.1**(σελ 77) όπου

mul=Multiplier (Float)

upBound=UpperBound (Integer)

lowBound=LowerBound (Integer)

- **addAbility(AbilityClass, Level):** Προσθέτει την ικανότητα που ορίζει το **AbilityClass** στις διαθέσιμες που θα έχουν τα Lifeform με αυτό το LifeformClass. Η ικανότητα θα έχει επίπεδο **Level.**
  - **AbilityClass(Zeta.AbilityClass):** Η AbilityClass της ικανότητας.
  - **Level(Integer):** Το επίπεδο που θα έχει η ικανότητα.
- **Καλούμενο(Path):** Κατασκευάζει και επιστρέφει ένα LifeformClass από το αρχείο **Path.**
  - **Path(String):** Η διαδρομή του αρχείου του LifeformClass
  - **Επιστρέφει -> Zeta.LifeformClass**

8.21 Zeta.LifeformState

Αυτά είναι βοηθητικά αντικείμενα που περιέχουν την κατάσταση κάποιου Lifeform. Η κατάσταση σπάει σε κατεύθυνση που κοιτάει και ενέργεια που κάνει εκείνη την στιγμή.

- **Direction(Enumeration):** Απαρίθμηση για τον προσδιορισμό της κατεύθυνσης.
  - **Down**
  - **Up**
  - **Left**
  - **Right**
- **Action(Enumeration):** Απαρίθμηση για τον προσδιορισμό της ενέργειας.
  - **Idle**
  - **Moving**
  - **Dying**
  - **Dead**
  - **Casting**
- **set(CombinedState):** Θέτει την κατάσταση από τον αριθμό **CombinedState.** Αυτός ο αριθμός πρέπει να προέρχεται από λογική πράξη **OR** ενός **Direction** και ενός **Action.** Επίσης μπορεί να αποδοθεί μόνο **Direction** ή **Action** χωρίς συνδυασμό. Λάθος συνδυασμός θα αγνοηθεί (πχ Direction OR Direction).
  - **CombinedState(Integer):** Ο συνδυασμένος αριθμός που αναπαριστά την κατάσταση.
- **getCombinedState():** Επιστρέφει έναν αριθμό που είναι ο συνδυασμός του **Direction** και του **Action.**
  - **Επιστρέφει -> Integer**
- **setAction(Action):** Θέτει την ενέργεια της κατάστασης.
  - **Action(Enumeration):** Την ενέργεια που θα αποδοθεί
- **setDirection(Direction):** Θέτει την κατεύθυνση της κατάστασης.
  - **Direction(Enumeration):** Την κατεύθυνση που θα αποδοθεί
- **getAction():** Επιστρέφει την ενέργεια της κατάστασης.
  - **Επιστρέφει -> Enumeration**
- **getDirection():** Επιστρέφει την κατεύθυνση της κατάστασης.

- **Επιστρέφει -> Enumeration**

- **Καλουμενο:** Κατασκευάζει και επιστρέφει μια προσωρινή κατάσταση με τιμές **Direction.Down** και **Action.Idle.**
  - **Επιστρέφει -> Zeta.LifeformState**

8.22 Zeta.Logger

Αντικείμενο Singleton που χρησιμοποιείται για να γίνουν εγγραφές στο Log File της μηχανής.

- **getInstance():** Επιστρέφει το μοναδικό αντικείμενο της κλάσης.
  - **Επιστρέφει -> Zeta.Logger**
- **write(Message):** Γράφει το **Message** στο αρχείο.
  - **Message(String):** Το κείμενο που θα γραφτεί.

8.23 Zeta.MainLoop

Αντικείμενο Singleton που χρησιμοποιείται για να έχουμε πρόσβαση στον ρυθμό FPS.

- **getInstance():** Επιστρέφει το μοναδικό αντικείμενο της κλάσης.
  - **Επιστρέφει -> Zeta.MainLoop**
- **getCurrentFPS():** Επιστρέφει τον τρέχων ρυθμό FPS που τρέχει η μηχανή.
  - **Επιστρέφει -> Float**

8.24 Zeta.Map

Αντικείμενο που διαχειρίζεται τον τρέχων χάρτη.

- **addObject(Object, ToBeDeleted):** Προσθέτει το **Object** στον χάρτη. Αν η τιμή του **ToBeDeleted** είναι **true**, τότε με την αλλαγή του χάρτη θα καταστραφεί. Η ανάκτηση του μετά την προσθήκη δεν είναι εφικτή μέσω του χάρτη.
  - **Object(Zeta.Object):** Το αντικείμενο που θα προστεθεί.
  - **ToBeDeleted(Boolean):** Εάν θα καταστραφεί με την αλλαγή του χάρτη ή όχι.
- **insertObject(Object, ToBeDeleted):** Προσθέτει το **Object** στον χάρτη. Αν η τιμή του **ToBeDeleted** είναι **true**, τότε με την αλλαγή του χάρτη θα καταστρα-

φεί. Η διαφορά με την παραπάνω είναι ότι μπορεί να αφαιρεθεί πριν την αλλαγή χάρτη.
    - **Object(Zeta.Object):** Το αντικείμενο που θα προστεθεί.
    - **ToBeDeleted(Boolean):** Εάν θα καταστραφεί με την αλλαγή του χάρτη ή όχι.
- **removeObject(Object):** Αφαιρεί το αντικείμενο από τον χάρτη εφόσον έχει προστεθεί με την **insertObject().** Αν πρέπει να καταστραφεί, θα γίνει.
    - **Object(Zeta.Object):** Το αντικείμενο που θα αφαιρεθεί.

8.25 Zeta.Npc

Βοηθητικό αντικείμενο ίδιο με τα Lifeform.

- **Καλούμενο(LifeformClass, x, y, LuaTable):** Κατασκευάζει και επιστρέφει ένα Lifeform με LifeformClass το **LifeformClass** και το τοποθετεί στις συντεταγμένες **x,y.** Του αποδίδεται ο πίνακας **LuaTable**
    - **Επιστρέφει -> Zeta.Lifeform**

8.26 Zeta.Object

Γενικά αντικείμενα που είναι στατικά (δεν έχουν ζωή). Από αυτή την κλάση κληρονομούν όλες τις μεθόδους τους οι πιο εξειδικευμένες κλάσεις (όπως Lifeform).

- **getBounding():** Επιστρέφει το ορθογώνιο σύγκρουσης του αντικειμένου.
    - **Επιστρέφει -> Zeta.Rectangle**
- **setBounding(Rectangle):** Θέτει το ορθογώνιο σύγκρουσης του αντικειμένου.
    - **Rectangle(Zeta.Rectangle):** Το ορθογώνιο που θα αποδοθεί
- **getTargetArea():** Επιστρέφει το ορθογώνιο στόχευσης του αντικειμένου.
    - **Επιστρέφει -> Zeta.Rectangle**
- **setTargetArea(Rectangle):** Θέτει το ορθογώνιο στόχευσης του αντικειμένου.
    - **Rectangle(Zeta.Rectangle):** Το ορθογώνιο που θα αποδοθεί
- **getPosition():** Επιστρέφει το διάνυσμα των συντεταγμένων που βρίσκεται το αντικείμενο.
    - **Επιστρέφει -> Zeta.Vector2**
- **setPosition(x, y):** Θέτει την τοποθεσία που βρίσκεται το αντικείμενο.
    - **x(Float):** Η συντεταγμένη x που θα μετακινηθεί το αντικείμενο

    - **y(Float):** Η συντεταγμένη y που θα μετακινηθεί το αντικείμενο
- **setPosition(Vector):** Ίδια με την παραπάνω, μόνο που δέχεται ένα διάνυσμα συντεταγμένων.
    - **Vector(Zeta.Vector2):** Το διάνυσμα των συντεταγμένων που θα μετακινηθεί το αντικείμενο
- **getName():** Επιστρέφει το όνομα του αντικειμένου.
    - **Επιστρέφει -> String**
- **getDistance(Object):** Επιστρέφει την απόσταση μεταξύ αυτού του αντικειμένου και του **Object**. Η απόσταση μετριέται σε Pixel και γίνεται μεταξύ των κέντρων των αντικειμένων.
    - **Object(Zeta.Object):** Το άλλο αντικείμενο
    - **Επιστρέφει -> Float**
- **addChildObject(Name, Object):** Προσθέτει το **Object** σαν αντικείμενο παιδί στο υπάρχων με όνομα **Name**. Το αν το παιδί αυτό θα καταστραφεί μαζί με τον γονέα καθορίζεται από το παιδί. Στο παιδί αλλάζει το όνομα που είχε πριν και παίρνει το **Name.** Αν το όνομα **Name** υπάρχει ήδη στην λίστα των παιδιών του αντικειμένου που το δέχεται και το Flag καταστροφής είναι **true**, τότε θα καταστρέψει το παιδί αυτό και θα παραβλέψει την προσθήκη.
    - **Name(String):** Το όνομα που θα πάρει το παιδί
    - **Object(ChildObject):** Το αντικείμενο παιδί που θα προστεθεί
- **addChildObject(Dx, Dy, Name, Class, Animation):** Δημιουργεί και προσθέτει ένα αντικείμενο παιδί με όνομα **Name** και AnimationClass **Class.** Στο αντικείμενο παιδί θα αποδοθεί το Animation με όνομα **Animation** εφόσον υπάρχει στο AnimationClass που ορίστηκε μέσω του **Class.** Στο αντικείμενο παιδί ορίζονται οι συντεταγμένες αυτές του κέντρου του γονιού και τους προστίθενται τα offsets **Dx, Dy**.
    - **Dx(Float):** Το Offset x που θα προστεθεί στις συντεταγμένες του παιδιού
    - **Dy(Float):** Το Offset y που θα προστεθεί στις συντεταγμένες του παιδιού
    - **Name(String):** Το όνομα που θα πάρει το παιδί
    - **Class(String):** Η διαδρομή του αρχείου του AnimationClass
    - **Animation(String):** Το όνομα του Animation που πρέπει να πάρει

- **removeChildObject(Name):** Αφαιρεί το αντικείμενο παιδί με όνομα **Name.** Αν το παιδί πρέπει να καταστραφεί, θα καταστραφεί.
  - **Name(String):** Το όνομα του παιδιού που πρέπει να αφαιρεθεί.
- **getRenderPosition():** Επιστρέφει ένα διάνυσμα που είναι το οι συντεταγμένες του αντικειμένου στην οθόνη σύμφωνα με την κύρια κάμερα.
  - **Επιστρέφει -> Zeta.Vector2**
- **isCollideAble():** Επιστρέφει **true** αν το αντικείμενο μπορεί να συγκρουστεί με άλλα, αλλιώς **false.**
  - **Επιστρέφει -> Boolean**
- **setCoolideAble(Value):** Θέτει αν το αντικείμενο θα μπορεί να συγκρουστεί με άλλα.
  - **Value(Boolean): true** για να μπορεί να συγκρουστεί
- **IsTriggeringCollisionEvents():** Επιστρέφει **true** αν το αντικείμενο μπορεί να σηκώσει συμβάντα συγκρούσεων. Αυτά τα συμβάντα σηκώνονται όταν το αντικείμενο συγκρουστεί με άλλο.
  - **Επιστρέφει -> Boolean**
- **setTriggeringCollisionEvents(Value):** Θέτει αν το αντικείμενο θα σηκώνει συμβάντα σύγκρουσης.
  - **Value(Boolean): true** για να μπορεί να σηκώσει συμβάντα
- **isVisible():** Επιστρέφει **true** αν το αντικείμενο εμφανίζεται (δεν έχει εξαφανιστεί μέσω άλλης κλήσης).
  - **Επιστρέφει -> Boolean**
- **setVisible(Value):** Εμφανίζει ή εξαφανίζει το αντικείμενο ανάλογα την τιμή **Value.**
  - **Value(Boolean): true** για να μπορεί να εμφανίζεται
- **show():** Εμφανίζει το αντικείμενο (εφόσον έχει εξαφανιστεί)
- **hide():** Εξαφανίζει το αντικείμενο (δεν το καταστρέφει)
- **getChildObject(Name):** Επιστρέφει το αντικείμενο παιδί με όνομα **Name.** Αν αυτό το παιδί δεν υπάρχει, επιστρέφει **nil.**
  - **Name(String):** Το όνομα του αντικειμένου που θα επιστραφεί
  - **Επιστρέφει -> Zeta.ChildObject**

8.27 Zeta.OffAnimation

Τα αντικείμενα αυτά είναι τα OffAnimations που μπορεί να πάρουν τα αντικείμενα. Κληρονομεί όλες τις μεθόδους της κλάσης **Zeta.AnimationPlayer.**

- **getDx():** Επιστρέφει το offset dx που θα έχει το Offanimation σε σχέση με το αντικείμενο που θα το έχει.
  - **Επιστρέφει -> Float**
- **setDx(Value):** Θέτει το offset dx που θα έχει το Offanimation σε σχέση με το αντικείμενο που θα το έχει.
  - **Value(Float):** Το offset dx που θα πάρει
- **getDy():** Επιστρέφει το offset dy που θα έχει το Offanimation σε σχέση με το αντικείμενο που θα το έχει.
  - **Επιστρέφει -> Float**
- **setDy(Value):** Θέτει το offset dy που θα έχει το Offanimation σε σχέση με το αντικείμενο που θα το έχει.
  - **Value(Float):** Το offset dy που θα πάρει
- **setOffsets(Dx, Dy):** Θέτει τα Offsets Dx, Dy που θα έχει το Offanimation σε σχέση με το αντικείμενο που θα το έχει.
  - **Dx(Float):** Το Offset dx
  - **Dy(Float):** Το Offset dy

8.28 Zeta.OverTimeEffect

Αυτά τα αντικείμενα διαχειρίζονται παροδικές επιδράσεις με περιοδικούς σκανδαλισμούς. Κληρονομούν όλες τις ιδιότητες του Zeta.DurableEffect.

- **getTickEvery():** Επιστρέφει τον χρόνο σε δευτερόλεπτα που είναι η περίοδος που κάνει σκανδαλισμό η επιρροή.
  - **Επιστρέφει -> Float**
- **setTickEvery(Value):** Θέτει τον χρόνο σε δευτερόλεπτα που θα είναι η περίοδος που κάνει σκανδαλισμό η επιρροή.
  - **Value(Float):** Ο χρόνος σε δευτερόλεπτα

8.29 Zeta.Player

Αυτό το αντικείμενο διαχειρίζεται τον παίχτη. Ο παίχτης είναι μοναδικός κατά την όλη εκτέλεση του προγράμματος. Κληρονομεί όλες τις μεθόδους της κλάσης **Zeta.Lifeform.**

- **setView(View):** Θέτει την κάμερα για τον παίχτη.
  - **View(Zeta.View):** Η κάμερα που θα αποδοθεί
- **moveToWorldPosition(x, y):** Κάνει τον παίχτη να μετακινηθεί στις συντεταγμένες **x,y** της οθόνης. Οι συντεταγμένες αυτές θα μετατραπούν από συντεταγμένες οθόνης σε συντεταγμένες κόσμου και ο παίχτης θα κινηθεί εκεί.
  - **x(Float):** Η συντεταγμένη οθόνης x
  - **y(Float):** Η συντεταγμένη οθόνης y
- **targetLifeformPosition(x, y):** Μετατρέπει τις συντεταγμένες οθόνης **x,y** σε συντεταγμένες κόσμου και ύστερα ελέγχει αν υπάρχει κάποιο Lifeform σε αυτές τις συντεταγμένες. Αν υπάρχει, τότε γίνεται στόχος του παίχτη αλλιώς αφαιρείται όποιος στόχος υπήρχε πριν.
  - **x(Float):** Η συντεταγμένη οθόνης x
  - **y(Float):** Η συντεταγμένη οθόνης y
- **setLevelUpCallback(Function):** Θέτει την **Function** να καλείται κάθε φορά που ο παίχτης ανεβαίνει επίπεδο.
  - **LuaFunction(Lua Function):** Η συνάρτηση που θα καλεστεί
- **setTargetIndicator(ChildName):** Θέτει το αντικείμενο-παιδί σαν αντικείμενο στόχος για την επιθυμητή τοποθεσία του παίχτη. Πιο ειδικά, αυτό το αντικείμενο παιδί εμφανίζεται κάθε φορά που καλείται η **moveToWorldPosition()** στο σημείο που ορίστηκε. Όταν ο παίχτης σταματήσει να κινείται, τότε αυτό το αντικείμενο εξαφανίζεται. Χρησιμεύει για να γίνεται εμφανές που θα πάει ο παίχτης. Το αντικείμενο παιδί πρέπει να έχει προστεθεί στον παίχτη πριν καλεστεί αυτή η συνάρτηση. Αν δεν καλεστεί αυτή η συνάρτηση, τότε δεν θα εμφανίζεται τίποτα στις συντεταγμένες.
  - **ChildName(String):** Το όνομα του παιδιού που θα γίνει αντικείμενο τοποθεσίας
- **getXp():** Επιστρέφει το συνολικό XP που έχει ο παίχτης.
  - **Επιστρέφει -> Float**

- **setXp(Value):** Θέτει το συνολικό XP που έχει ο παίχτης.
  - **Value(Float):** Το XP που θα αποδοθεί
- **offsetXP(Value):** Προσθέτει την τιμή **Value** στο συνολικό XP του παίχτη.
  - **Value(Float):** Το XP που θα προστεθεί
- **setXpToNextLevel(Value):** Θέτει το XP που χρειάζεται ο παίχτης για να ανέβει επίπεδο.
  - **Value(Float):** Το XP που θα χρειαστεί
- **getXpToNextLevel(Value):** Επιστρέφει το XP που χρειάζεται ο παίχτης για να ανέβει επίπεδο.
  - **Επιστρέφει -> Float**

8.30 Zeta.Projectile

Αντικείμενα που διαχειρίζονται γενικά βλήματα. Κληρονομούν όλες τις μεθόδους από την κλάση **Zeta.Object.**

- **clearCollidedObjects():** Καθαρίζει την εσωτερική λίστα με τα αντικείμενα που συγκρούστηκε το βλήμα.
- **destroy():** Καταστρέφει το βλήμα εκτελώντας όλες τις απαραίτητες ενέργειες.
- **setTargetLocation(x, y):** Θέτει τις συντεταγμένες **x,y** ως τον στόχο που θα κατευθυνθεί το βλήμα.
  - **x(Float):** Η συντεταγμένη x
  - **y(Float):** Η συντεταγμένη y
- **getParentAbility():** Επιστρέφει την ικανότητα που προκάλεσε το βλήμα. Μπορεί να επιστρέψει **nil** αν δεν το προκάλεσε ικανότητα.
  - **Επιστρέφει -> Zeta.ActiveAbility**

8.31 Zeta.Rectangle

Τα αντικείμενα αυτά ορίζουν ένα ορθογώνιο. Τα ορθογώνια χρησιμοποιούνται σε πολλά σημεία (όπως ορθογώνιο σύγκρουσης).

- **getX():** Επιστρέφει την συντεταγμένη x της τοποθεσίας του ορθογωνίου που είναι η πάνω αριστερή γωνία του.
  - **Επιστρέφει -> Float**

- **getY():** Επιστρέφει την συντεταγμένη y της τοποθεσίας του ορθογωνίου που είναι η πάνω αριστερή γωνία του.
  - **Επιστρέφει -> Float**
- **setPosition(x, y):** Θέτει τις συντεταγμένες του ορθογωνίου που είναι η πάνω αριστερή γωνία του.
  - **x(Float):** Η συντεταγμένη x
  - **y(Float):** Η συντεταγμένη y
- **offsetPosition(Dx, Dy):** Προσθέτει τα Offsets **Dx, Dy** στην τοποθεσία του ορθογωνίου.
  - **Dx(Float):** Το Offset για την συντεταγμένη x
  - **Dy(Float):** Το Offset για την συντεταγμένη y
- **getWidth():** Επιστρέφει το πλάτος του ορθογωνίου.
  - **Επιστρέφει -> Integer**
- **getHeight():** Επιστρέφει το ύψος του ορθογωνίου.
  - **Επιστρέφει -> Integer**
- **setWidth(Value):** Θέτει το πλάτος του ορθογωνίου.
  - **Vaule(Float):** Το πλάτος που θα πάρει
- **setHeight(Value):** Θέτει το ύψος του ορθογωνίου.

Vaule(Float): Το ύψος που θα πάρει

- **setSize(Width, Height):** Θέτει το μέγεθος του ορθογωνίου.
  - **Width(Float):** Το πλάτος
  - **Height(Float):** Το ύψος
- **overlapsPoint(x, y):** Επιστρέφει **true** αν το σημείο **x, y** βρίσκεται εντός του χώρου του ορθογωνίου, αλλιώς **false.**
  - **x(Float):** Η συντεταγμένη x
  - **y(Float):** Η συντεταγμένη y
  - **Επιστρέφει -> Boolean**
- **overlapsPoint(Vector):** Επιστρέφει **true** αν το σημείο **Vector** βρίσκεται εντός του χώρου του ορθογωνίου, αλλιώς **false.**
  - **Vector(Zeta.Vector2):** Το διάνυσμα που έχει τις συντεταγμένες
  - **Επιστρέφει -> Boolean**

- **overlapsRectangle(Rectangle):** Επιστρέφει **true** αν το ορθογώνιο **Rectangle** βρίσκεται εντός του χώρου του ορθογωνίου είτε ολόκληρο είτε μερικώς, αλλιώς **false.**
  - **Rectangle(Zeta.Rectangle):** Το άλλο ορθογώνιο
  - **Επιστρέφει -> Boolean**
- **Καλούμενο():** Κατασκευάζει και επιστρέφει ένα ορθογώνιο στο σημείο (0, 0) με πλάτος και ύψος 0.
  - **Επιστρέφει -> Zeta.Rectangle**
- **Καλούμενο(x, y, Width, Height):** Κατασκευάζει και επιστρέφει ένα ορθογώνιο στο σημείο **(x, y)** με πλάτος **Width** και ύψος **Height**.
  - **x(Float):** Οι συντεταγμένη x που θα κατασκευαστεί το ορθογώνιο
  - **y(Float):** Οι συντεταγμένη y που θα κατασκευαστεί το ορθογώνιο
  - **Width(Integer):** Το πλάτος του ορθογωνίου
  - **Height(Integer):** Το ύψος του ορθογωνίου
  - **Επιστρέφει -> Zeta.Rectangle**
- **Καλούμενο(Vector, Width, Height):** Κατασκευάζει και επιστρέφει ένα ορθογώνιο στο σημείο που θέτει το διάνυσμα **Vector** με πλάτος **Width** και ύψος **Height**.
  - **Vector(Zeta.Vector2):** Οι συντεταγμένη που θα κατασκευαστεί το ορθογώνιο
  - **Width(Integer):** Το πλάτος του ορθογωνίου
  - **Height(Integer):** Το ύψος του ορθογωνίου
  - **Επιστρέφει -> Zeta.Rectangle**
  -

8.32 Zeta.SeekingProjectile

Εξειδικευμένο είδος βλήματος. Κληρονομεί όλες τις μεθόδους της κλάσης **Zeta.Projectile.**

- **setTarget(Lifeform):** Θέτει το Lifeform που θα κατευθύνεται το βλήμα.
  - **Lifeform(Zeta.Lifeform):** Το Lifeform που θα πάει το βλήμα.

8.33 Zeta.Settings

Αυτή η κλάση Singleton διαχειρίζεται ρυθμίσεις του παιχνιδιού όπως η ανάλυση της οθόνης κτλ. Στα παρακάτω, η δομή δεδομένων **Zeta.Display.Resolution** είναι μια απλή δομή με πεδία **height** και **width.**

- **getInstance():** Επιστρέφει το μοναδικό αντικείμενο της κλάσης.
  - **Επιστρέφει -> Zeta.Settings**
- **getCurrentResolution():** Επιστρέφει την τρέχουσα ανάλυση που παίζει το παιχνίδι.
  - **Επιστρέφει -> Zeta.Display.Resolution**
- **getFullscreen():** Επιστρέφει **true** αν το παιχνίδι τρέχει σε πλήρη οθόνη.
  - **Επιστρέφει -> Boolean**
- **setFullscreen(Value):** Θέτει αν το παιχνίδι θα τρέχει σε πλήρη οθόνη.
  - **Value(Boolean): true** για πλήρη οθόνη
- **setResolution(Resolution):** Θέτει την ανάλυση της οθόνης σε **Resolution.**
  - **Resolution(Zeta.Display.Resolution):** Η δομή δεδομένων που περιέχει την ανάλυση.
- **getResolutionsTable():** Επιστρέφει έναν πίνακα Lua που περιέχει όλες τις διαθέσιμες αναλύσεις οθόνης για το σύστημα που τρέχει το παιχνίδι. Ο πίνακας αυτός είναι String Key Indexed (hashtable με κλειδιά String).
  - **Επιστρέφει -> LuaTable**
- **loadFile(Path):** Φορτώνει ένα αρχείο ρυθμίσεων και το εφαρμόζει.
  - **Path(String):** Η διαδρομή του αρχείου που έχει τις ρυθμίσεις.

8.34 Zeta.System

Αυτή η κλάση Singleton διαχειρίζεται το σύστημα της μηχανής, όμως μέσω της Lua μπορούμε να κάνουμε περιορισμένα πράγματα για λόγους ευκολίας.

- **getInstance():** Επιστρέφει το μοναδικό αντικείμενο της κλάσης.
  - **Επιστρέφει -> Zeta.System**
- **getAudioContext():** Επιστρέφει το αντικείμενο διαχείρισης του ήχου.
  - **Επιστρέφει -> Zeta.AudioContext**
- **shutdown():** Τερματίζει την λειτουργία του προγράμματος κάνοντας όλους τους απαραίτητους καθαρισμούς.

- **abort(Message):** Τερματίζει την λειτουργία του προγράμματος κάνοντας όλους τους απαραίτητους καθαρισμούς και γράφοντας το **Message** στο LogFile.
  - **Message(String):** Το μήνυμα που θα γραφτεί

8.35 Zeta.Timer

Αυτά τα αντικείμενα κατασκευάζονται σαν χρονισμένη κλήση Lua. Με άλλα λόγια αυτά τα αντικείμενα μόλις ενεργοποιηθούν, μετά από χρονικό διάστημα που τους ορίζει ο προγραμματιστής, θα καλέσουν μια συνάρτηση Lua. Το αντικείμενο μετά μπορεί ο προγραμματιστής να το επαναφέρει ώστε να το ξανακάνει.

- **getMaxTime():** Επιστρέφει τον χρόνο σε δευτερόλεπτα που θα περάσει από την εκκίνηση του μετρητή μέχρι να καλέσει την συνάρτηση και να σταματήσει.
  - **Επιστρέφει -> Float**
- **setMaxTime(Value):** Θέτει τον χρόνο σε δευτερόλεπτα που θα περάσει από την εκκίνηση του μετρητή μέχρι να καλέσει την συνάρτηση και να σταματήσει.
  - **Value(Float):** Ο χρόνος σε δευτερόλεπτα
- **getRemainingTime():** Επιστρέφει τον χρόνο σε δευτερόλεπτα που απομένει μέχρι να καλεστεί η κλήση Lua.
  - **Επιστρέφει -> Float**
- **setRemainingTime():** Θέτει τον χρόνο σε δευτερόλεπτα που απομένει μέχρι να καλεστεί η κλήση Lua. Ο χρόνος αυτός μπορεί να πάρει σε ότι τιμή χρειάζεται ακόμα και μεγαλύτερη της μέγιστης που ορίστηκε.
  - **Value(Float):** Ο χρόνος σε δευτερόλεπτα
- **isOver():** Επιστρέφει **true** αν ο μετρητής **δεν** τρέχει.
  - **Επιστρέφει -> Boolean**
- **isRunning():** Επιστρέφει **true** αν ο μετρητής τρέχει.
  - **Επιστρέφει -> Boolean**
- **start():** Ξεκινάει τον μετρητή.
- **pause():** Διακόπτει τον μετρητή στο σημείο που βρίσκεται.
- **reset():** Επαναφέρει τον χρόνο που απομένει στον αρχικό χρόνο.
- **setCallback(Function):** Θέτει την συνάρτηση **Function** σαν την συνάρτηση που θα καλεστεί όταν τελειώσει ο μετρητής.

- **Function(Lua Function):** Η συνάρτηση που θα καλεστεί

- **Καλούμενο(MaxTime):** Κατασκευάζει και επιστρέφει έναν μετρητή με χρόνο για την κλήση **MaxTime.** Ο μετρητής δεν ξεκινάει κατά την κατασκευή, αλλά θα πρέπει να το κάνει ο προγραμματιστής όποτε πρέπει.
  - **Επιστρέφει -> Zeta.Timer**

8.36 Zeta.Vector2

Αυτά τα αντικείμενα ορίζουν διανύσματα. Τα διανύσματα χρησιμεύουν όταν θέλουμε να ορίζουμε μια συντεταγμένη ή κατεύθυνση. Έχουν μια συντεταγμένη x και y.

- **getX():** Επιστέφει το x του διανύσματος.
  - **Επιστρέφει -> Float**
- **setX(Value):** Θέτει το x του διανύσματος.
  - **Value(Float):** Η τιμή που θα αποδοθεί
- **getY():** Επιστέφει το y του διανύσματος.
  - **Επιστρέφει -> Float**
- **setY(Value):** Θέτει το y του διανύσματος.
  - **Value(Float):** Η τιμή που θα αποδοθεί
- **set(x, y):** Θέτει τα x και y.
  - **x(Float):** Η τιμή x που θα αποδοθεί
  - **y(Float):** Η τιμή y που θα αποδοθεί
- **offset(Dx, Dy):** Προσθέτει τα offsets **Dx, Dy** στα x και y αντίστοιχα του διανύσματος.
  - **Dx(Float):** Η τιμή που προστεθεί στο x
  - **Dy(Float):** Η τιμή που προστεθεί στο y
- **getLength():** Επιστρέφει το μήκος του διανύσματος.
  - **Επιστρέφει -> Float**
- **getDistance(Vector):** Επιστρέφει την απόσταση μεταξύ αυτού του διανύσματος και του **Vector.**
  - **Vector(Zeta.Vector2):** Το άλλο διάνυσμα.
  - **Επιστρέφει -> Float**
- **normalize():** Μετατρέπει το διάνυσμα σε μοναδιαίο (με μήκος 1). Αυτό χρησιμεύει κυρίως για τον προσδιορισμό της κατεύθυνσης

- **Καλούμενο():** Κατασκευάζει και επιστρέφει ένα διάνυσμα στο (0,0)
  - **Επιστρέφει -> Float**
- **Καλούμενο(x, y):** Κατασκευάζει και επιστρέφει ένα διάνυσμα στο **(x,y)**
  - **Επιστρέφει -> Float**
- **Πράξεις:** Στα αντικείμενα αυτά μπορούν να γίνουν πράξεις μεταξύ τους όπως παρακάτω.
  - **Vector + Vector = Vector**
  - **Vector - Vector = Vector**
  - **Vector * Float = Vector**

8.37 Zeta.View

Αυτά τα αντικείμενα ορίζουν μια κάμερα στον κόσμο. Συνήθως η κάμερα είναι μοναδική. Λειτουργεί σαν ορθογώνιο.

- **setPosition(x, y):** Θέτει τις συντεταγμένες της κάμερας που είναι η πάνω αριστερά γωνία του.
  - **x(Float):** Η συντεταγμένη x
  - **y(Float):** Η συντεταγμένη y
- **offsetPosition(Dx, Dy):** Προσθέτει τα Offsets **Dx, Dy** στην τοποθεσία της κάμερας.
  - **Dx(Float):** Το Offset για την συντεταγμένη x
  - **Dy(Float):** Το Offset για την συντεταγμένη y

- **resize(Width, Height):** Αλλάζει το μέγεθος της κάμερας σε **Width x Height**
  - **Width(Integer):** Το νέο πλάτος
  - **Height(Integer):** Το νέο ύψος
- **getX():** Επιστρέφει την συντεταγμένη x που βρίσκεται η κάμερα στον κόσμο.
  - **Επιστρέφει -> Float**
- **getY():** Επιστρέφει την συντεταγμένη y που βρίσκεται η κάμερα στον κόσμο.
  - **Επιστρέφει -> Float**
- **getWidth():** Επιστρέφει το πλάτος της κάμερας.
  - **Επιστρέφει -> Integer**
- **getHeight():** Επιστρέφει το ύψος της κάμερας.

- **Επιστρέφει -> Integer**

- **isInView(Object):** Επιστρέφει **true** αν το **Object** βρίσκεται εντός κάμερας, αλλιώς **false.**
  - **Επιστρέφει -> Boolean**

8.38 Zeta.WorldEvent

Αυτά τα αντικείμενα ορίζουν συμβάντα κόσμου. Ο προγραμματιστής τα χρησιμοποιεί για να σηκώσει συμβάντα που δεν σηκώνονται από την μηχανή όποτε κρίνει αυτός.

- **Type(Enumeration):** Ορίζει τον τύπο του συμβάντος.
  - **Damage**
  - **Death**
  - **Interact**
  - **AbilityUse**
  - **WorldExit**
  - **Collision**
  - **Custom**
  - **Nothing**
- **setAsDamageEvent(Victim, Dealer, Amount):** Μετατρέπει το συμβάν σε συμβάν ζημιάς.
  - **Victim(Zeta.Lifeform)**: Ποιο Lifeform δέχεται την ζημιά
  - **Dealer(Zeta.Lifeform):** Ποιο Lifeform κάνει την ζημιά
  - **Amount(Unsinged Integer):** Το ποσό της ζημιάς
- **setAsDeathEvent(Victim):** Μετατρέπει το συμβάν σε συμβάν θανάτου.
  - **Victim(Zeta.Lifeform):** Ποιο Lifeform πέθανε
- **setAsInteractEvent(Lifeform1, Lifeform2):** Μετατρέπει το συμβάν σε συμβάν διάδρασης.
  - **Lifeform1(Zeta.Lifeform):** Το Lifeform1
  - **Lifeform2(Zeta.Lifeform):** Το Lifeform2
- **setAsAbilityUseEvent(User, Ability):** Μετατρέπει το συμβάν σε συμβάν χρήσης ικανότητας.
  - **User(Zeta.Lifeform):** Το Lifeform που χρησιμοποιεί την ικανότητα

    - **Ability(Zeta.Ability):** Η ικανότητα που χρησιμοποιήθηκε
- **setAsWorldExitEvent(Invoker):** Μετατρέπει το συμβάν σε συμβάν εξόδου κόσμου.
    - **Invoker(Zeta.Lifeform):** Το Lifeform που βγήκε εκτός
- **setAsCustomEvent(Channel, Table):** Μετατρέπει το συμβάν σε προσαρμοσμένο.
    - **Channel(String):** Το κανάλι που θα μεταδοθεί το συμβάν.
    - **Table(Lua Table):** Ο πίνακας Lua που θα περαστεί μαζί με το συμβάν όταν μεταδοθεί.
- **broadcast():** Μεταδίδει το συμβάν στο κανάλι που χρειάζεται.
- **Καλούμενο():** Κατασκευάζει και επιστρέφει ένα συμβάν. Το συμβάν αυτό είναι τύπου **Nothing** και δεν θα κάνει τίποτα. Πρέπει να μετατραπεί σε εξειδικευμένο με τις παραπάνω κλήσεις πριν καλεστεί η **broadcast().**
    - **Επιστρέφει -> Zeta.WorldEvent**

8.39 Zeta.WorldManager

Αυτό το αντικείμενο Singleton διαχειρίζεται την ροή του παιχνιδιού και τα συμβάντα που συμβαίνουν στον κόσμο.

- **Callback(Enumeration):** Ορίζει τύπους κλήσεων Lua που γίνονται από το αντικείμενο αυτό.
    - **ChangeMap**
    - **LoadBegin**
    - **LoadEnd**
    - **FrameBegin**
    - **FrameEnd**
    - **LifeformDeath**
    - **LifeformShow**
    - **LifeformHide**
- **getInstance():** Επιστρέφει το μοναδικό αντικείμενο της κλάσης.
    - **Επιστρέφει -> Zeta.WorldManager**

- **changeMap(Path):** Φορτώνει και αλλάζει τον χάρτη που υπάρχει την στιγμή που καλείται. Η φόρτωση γίνεται αμέσως και θα προκαλέσει πάγωμα του παιχνιδιού μέχρι να τελειώσει, για αυτό προσοχή πότε καλείται.
  - **Path(String):** Η διαδρομή του αρχείου χάρτη που θα φορτωθεί
- **setCallback(Type, Function):** Ορίζει την συνάρτηση Lua **Function** να καλείται κάθε φορά που σηκώνεται ένα συμβάν τύπου **Type.**
  - **Type(Enumeration):** Ο τύπος του συμβάντος που θα καταχωρηθεί η συνάρτηση.
  - **Function(Lua Function):** Η συνάρτηση που θα καλείται
- **raiseEvent(Event):** Μεταδίδει το συμβάν **Event.**
  - **Event(Zeta.WorldEvent):** Το συμβάν που θα μεταδοθεί
- **getView():** Επιστρέφει την κάμερα του κόσμου.
  - **Επιστρέφει -> Zeta.View**
- **show():** Εμφανίζει τον κόσμο (Χάρτη, παίχτη, Lifeforms κτλ)
- **hide():** Εξαφανίζει τον κόσμο (δεν τον πειράζει, απλά δεν τον δείχνει στην οθόνη). Υπόψη ότι ο κόσμος συνεχίζει να τρέχει και δεν διακόπτεται η λειτουργία του.
- **pause():** Διακόπτει την λειτουργία του κόσμου παγώνοντας τον.
- **resume():** Συνεχίζει την λειτουργία του κόσμου (εφόσον έχει διακοπεί).
- **getCurrentMap():** Επιστρέφει τον χάρτη που τρέχει την στιγμή που καλείται.
  - **Επιστρέφει -> Zeta.Map**
- **getPlayer():** Επιστρέφει τον παίχτη.
  - **Επιστρέφει -> Zeta.Player**

9 ΣΥΝΕΧΙΖΟΝΤΑΣ

Συνδυάζοντας όλα αυτά που περιγράφτηκαν στο μέρος αυτό, μπορεί πλέον κάποιος να φτιάξει ένα παιχνίδι μόνο με αρχεία XML και Lua Scripts.

Εάν κάποιος θέλει πιο εξεζητημένα πράγματα θα χρειαστεί να επεξεργαστεί τον κώδικα της μηχανής.

10 ΤΟ DEMO

Ας πάμε να δούμε λίγο περιεκτικά το τι περιέχει το Demo της μηχανής. Το Demo αυτό αναπτύχτηκε παράλληλα με την μηχανή και χρησιμοποιούνταν παράλληλα για Debugging. Κάθε φορά που δημιουργούνταν ένα νέο χαρακτηριστικό, τότε το έβαζα στο Demo και έκανα Tests. Μπορούμε να πούμε ότι το Demo είναι μια εκδοχή που έχουμε τα περισσότερα χαρακτηριστικά της μηχανής σε λειτουργική κατάσταση.

Όπως ξέρουμε, το παιχνίδι ξεκινάει με την εκτέλεση του Boot.lua. Εδώ αυτό που κάνει είναι να εμφανίσει το κύριο μενού. Από εκεί επιλέγοντας το “Εκκίνηση” ξεκινάμε ένα νέο παιχνίδι και για αυτό και μας βγάζει σε οθόνη επιλογής χαρακτήρα. Για το Demo κατασκευάσαμε μόνο έναν.

Διαλέγοντας την, ξεκινάμε το παιχνίδι πατώντας το “Ξεκίνα!”. Από εκεί εμφανίζεται μια οθόνη που μας λέει να περιμένουμε μέχρι να φορτώσει το παιχνίδι. Όταν τελειώσει η φόρτωση, τότε εμφανίζεται το κουμπί “Συνέχεια” και πατώντας το ξεκινάει το παιχνίδι.

Εικόνα 10.1: ScreenShot1

Στο παραπάνω ScreenShot βλέπουμε την σκηνή που έχουμε όταν ξεκινήσουμε το παιχνίδι. Τα παράθυρα τα εμφανίσαμε εμείς. Την εμφάνιση την έχουμε ρυθμίσει να γίνεται μέσω πλήκτρων του πληκτρολογίου. Στο πρώτο παράθυρο βλέπουμε τα στατιστικά του παίχτη μας. Στο δεύτερο βλέπουμε το δένδρο των ικανοτήτων. Κάτω έχουμε την μπάρα που μπαίνουν οι ενεργές ικανότητες προκειμένου να μπορούμε να τις χρησιμοποιήσουμε μέσω των πλήκτρων των γραμμάτων που έχουν πάνω τους. Σέρνοντας μια ενεργή ικανότητα από το δένδρο, μπορούμε να την βάλουμε σε κάποιο γράμμα της μπάρας. Από τότε μπορούμε να την χρησιμοποιήσουμε όποτε πατήσουμε το πλήκτρο γράμματος του πληκτρολογίου που έχει πάνω. Μπορούμε να της αλλάξουμε και θέση, απλά σέρνοντας την σε κουτί με άλλο γράμμα. Δεξιά της μπάρας έχουμε στατιστικά που δεν έχουν τιμή. Αυτά θα δείχνουν τα στατιστικά που έχει ο στόχος μας και εφόσον δεν έχουμε, δεν δείχνουν τίποτα.

Μπορούμε να κινηθούμε παντού κάνοντας δεξί κλικ στο έδαφος που θέλουμε να πάμε. Αν είμαστε κοντά σε άλλο Lifeform, μπορούμε να διαδραστούμε με αυτό κάνοντας πάνω του αριστερό κλικ.

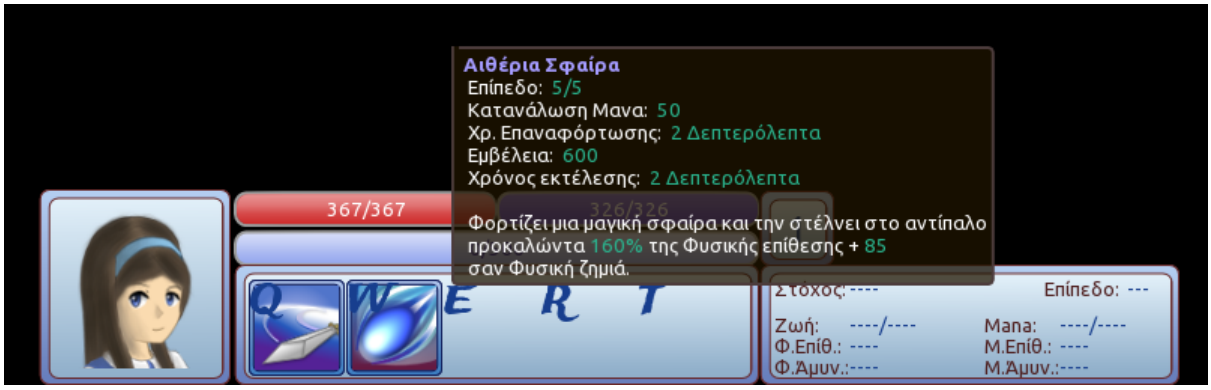
Εικόνα 10.2: Screenshot2

Όταν πατάμε το NPC Μαρία Μπρίλιου, μας βγάζει ένα παράθυρο διαλόγου. Αυτό μας λέει τι έχει να μας πει το NPC, καθώς και από κάτω μας δίνονται τι απαντήσεις μπορούμε να δώσουμε. Σε κάθε απάντηση το NPC θα μας πει κάτι άλλο. Πάνω από το κεφάλι του NPC έχει ένα θαυμαστικό. Αυτό σημαίνει ότι έχει αποστολή να μας δώσει. Έτσι αν πατήσουμε τον διάλογο που κάνει η τρίτη απάντηση που θα δώσουμε, ξεκινάει η αποστολή.

Η αποστολή είναι απλή. Πρέπει να διώξουμε τις κότες έξω από το σπίτι. Αυτό το κάνουμε πηγαίνοντας κοντά τους και κάνοντας δεξί κλικ. Όταν τις κλικάρουμε, τότε τις "τρομοκρατούμε" και αρχίζουν να πηγαίνουν αντίθετα από την κατεύθυνση που έχουμε προς αυτές για λίγο χρονικό διάστημα. Αν τις καταφέρουμε να φτάσουν στην έξοδο, βγαίνουν από το δωμάτιο και ολοκληρώνεται η αποστολή.

Εικόνα 10.3: Screenshot3

Αν κατευθυνθούμε προς την έξοδο, θα φύγουμε από το δωμάτιο και θα αρχίσει να φορτώνεται ο νέος χάρτης που είναι έξω από το σπίτι. Βλέπουμε δίπλα ένα άλλο NPC και παντού κότες να γυρνάνε και να τρώνε. Αν κατευθυνθούμε προς τα κάτω πέρα από τα χωράφια, θα δούμε πολλά φίδια. Αυτά είναι επιθετικά αλλά είναι καλή ευκαιρία να δοκιμάσουμε τις ικανότητές μας. Μέχρι στιγμής έχουμε μόνο την ικανότητα που είναι αυτή που είναι ήδη ξεκλείδωτη, η “Χτύπημα Σπαθιού”. Για λόγους του Demo, μπορούμε να ξεκλειδώσουμε και την επόμενη την “Αιθέρια Σφαίρα”. Κάνοντας δεξί κλικ πάνω στην κλειδωμένη ικανότητα στο δένδρο ικανοτήτων (Το δένδρο εμφανίζεται με το πλήκτρο A), την ξεκλειδώνουμε. Κάνοντας επιπλέον κλικ πάνω της, τις ανεβάζουμε επίπεδα μέχρι το μέγιστο που μπορεί να πάρει. Σέρνοντας την στην μπάρα, μπορούμε να την αναθέσουμε σε κάποιο πλήκτρο για χρήση.



Εικόνα 10.4: Screenshot4

Η πρώτη ικανότητα έχει μικρή εμβέλεια και χρειάζεται να γίνει από κοντά. Η δεύτερη είναι επίθεση από μακριά που εξαπολύει μια σφαίρα που ακολουθεί τον στόχο μέχρι να τον τραυματίσει. Έχοντας τα φίδια σαν στόχο μπορούμε να τις δοκιμάσουμε. Τα φίδια είναι εχθροί που σημαίνει ότι όταν πεθάνουν, ξανά ζωντανεύουν μετά από ένα χρονικό διάστημα.

Εικόνα 10.5: Screenshot5

Αν μας τραυματίσει ένα φίδι, τότε βλέπουμε ότι η ζωή που μας αφαίρεσε αναπληρώνεται σιγά-σιγά. Σε αυτό βοηθάει η αναζωογόνηση που έχει ο χαρακτήρας μας. Το ίδιο γίνεται και για το Mana που σπαταλιέται.

Εδώ τελειώνει η περιγραφή του Demo. Μπορεί κάποιος να το περιεργαστεί και μόνος του αφού χτίσει την μηχανή και τρέξει το πρόγραμμα του Demo.

ΜΕΡΟΣ ΤΡΙΤΟ: ΓΕΝΙΚΗ ΑΝΑΛΥΣΗ ΤΟΥ ΔΥΑΔΙΚΟΥ ΠΥΡΗΝΑ

Εδώ τώρα θα αναλύσουμε πως λειτουργεί η μηχανή εσωτερικά στην έκδοση που βρίσκεται τώρα (1.0.3). Θα πάρουμε κομμάτι-κομμάτι και θα γίνει ανάλυση στα σημαντικότερα επιμέρους κομμάτια. Ο σκοπός είναι να δοθεί μια ιδέα για το πως λειτουργεί για να μπορούν οι προγραμματιστές να την τροποποιήσουν.

1 ΣΥΣΤΗΜΑ ΕΥΕΛΙΚΤΟΥ SDK

Η μηχανή προσφέρει ένα σύστημα για την σύνδεση με το υλικό/λειτουργικό που μπορεί να προσαρμοστεί. Η λειτουργία βασίζεται σε Interfaces που χρειάζεται να υλοποιηθούν από τον προγραμματιστή ώστε να κάνουν αυτά που ορίζουν. Η μηχανή δεν την νοιάζει πως θα γίνουν από πίσω. Αυτό που χρειάζεται για να λειτουργήσει σωστά είναι ότι όταν ένα Interface υλοποιηθεί, πρέπει να φέρει το αποτέλεσμα που ορίζει αλλιώς θα υπάρχει πρόβλημα. Για παράδειγμα όταν το Interface σου ορίζει μια συνάρτηση ότι πρέπει να σχεδιάζει ένα ορθογώνιο στο (x,y) και η υλοποίηση της συνάρτησης δεν το κάνει, τότε η μηχανή προφανώς δεν λειτουργήσει σωστά.

1.1 Η Abstract Κλάση Display

Η κλάση αυτή περιέχει όλες τις απαραίτητες συναρτήσεις που χρειάζεται να υλοποιηθούν από κλάσεις που θα περιέχουν το αντικείμενο του παραθύρου της οθόνης. Τα αντικείμενα θα πρέπει να κατασκευάζουν ένα OpenGL context. Οι συναρτήσεις που πρέπει να υλοποιηθούν είναι:

Πίνακας 1.1.1: Display Interface

```
class Display {
public:
     virtual int getWidth() const=0;
     virtual int getHeight() const=0;
     virtual void resize(int width, int height)=0;
     virtual void setFullScreen(bool value)=0;
     virtual std::vector<Resolution> getAvailableResolutions() const=0;
     virtual void switchFramebuffer()=0;
     Display();
     virtual ~Display();
};
```

Πιο αναλυτικά:

- `int getWidth() const`

Η κλήση αυτή πρέπει να επιστρέφει το πλάτος του OpenGL Context που κατασκευάστηκε από την κλάση αυτή. Πρέπει να επιστραφεί η τρέχουσα τιμή και όχι η αρχική.

- `int getHeight() const`

Η κλήση αυτή πρέπει να επιστρέφει το ύψος του OpenGL Context που κατασκευάστηκε από την κλάση αυτή. Πρέπει να επιστραφεί η τρέχουσα τιμή και όχι η αρχική.

- `void resize(int width, int height)`

Η κλήση αυτή πρέπει να αλλάζει το μέγεθος του OpenGL Context που δημιούργησε το αντικείμενο σε **width x height**.

- `void setFullScreen(bool value)`

Η κλήση αυτή πρέπει να εναλλάσσει το OpenGL Context από παράθυρο σε πλήρη οθόνη ανάλογα την τιμή του **value.** Για **true** θα πρέπει το Context να μετατραπεί σε πλήρη οθόνη. Για **false** θα πρέπει να μετατραπεί σε παράθυρο.

- `std::vector<Resolution> getAvailableResolutions() const`

Αυτή η συνάρτηση πρέπει να επιστρέψει ένα `std::vector<Resolution>` με αντικείμενα `Resolution`. Αυτός ο πίνακας πρέπει να περιέχει όλες τις διαθέσιμες αναλύσεις οθόνης για το Context που περιέχει το αντικείμενο. Ο ορισμός της κλάσης `Resolution` είναι στο Display.hpp.

- `void switchFramebuffer()`

Η συνάρτηση αυτή πρέπει να εναλλάσσει τον Front και τον Back Buffer του Context σε περιπτώσεις που χρησιμοποιείται Double ή Triple Buffering. **Σημαντικό:** πριν την εναλλαγή, ο Buffer που φαίνεται πρέπει να καθαρίζεται από τα περιεχόμενα του σε χρώμα RGB(0,0,0).

- `Display()`

Ο κατασκευαστής της κλάσης πρέπει να κατασκευάζει ένα OpenGL Context σε παράθυρο και να το αποθηκεύει μέσα στην κλάση αυτή για περαιτέρω προσαρμογή μέσω του Interface. Το παράθυρο πρέπει να είναι από την αρχή σε πλήρη οθόνη

(εκτός περιπτώσεων Debuging) και στην προεπιλεγμένη ανάλυση της οθόνης του χρήστη.

- `virtual ~Display()`

Ο καταστροφέας της κλάσης πρέπει να καταστρέφει και το OpenGL Context μαζί με το παράθυρο του X-server.

Κάποιες διευκρινήσεις ως προς τις κλάσεις και τα αντικείμενα. Από το Interface αυτό μπορούν να δημιουργηθούν πολλές κλάσεις αλλά κατά την εκτέλεση μόνο ένα αντικείμενο πρέπει να δημιουργηθεί. Για αυτό φροντίζει η κλάση System. Το αντικείμενο όπως είπαμε πρέπει να φτιάχνει ένα παράθυρο στον X-Server με OpenGL Context και να αποθηκεύει κάποια στοιχειώδες δομή για την περαιτέρω διαχείριση του. Ένα παράδειγμα είναι η κλάση AllegroDisplay.

1.2 Η Abstract Κλάση Bitmap

Το Interface αυτό φροντίζει για κάποιες βασικές κλήσεις που χρειάζεται η μηχανή για τον σχεδιασμό και την επεξεργασία των Textures στο OpenGL Context. Τα αντικείμενα που παράγονται από τις κλάσεις που θα το υλοποιήσουν, πρέπει να περιέχουν εσωτερικά μια δομή (πχ Pointer) για την αποθήκευση του Texture. Το αντικείμενο πρέπει να θεωρείται σαν μοναδικό Resource και δεν πρέπει να αντιγράφεται ούτε να παραποιείται πέρα από όταν ζητείται ρητά μέσω των κλήσεων του Interface.

Πίνακας 1.2.1: Bitmap Interface

```
class Bitmap: public Drawable, public Resource {
public:
     Bitmap(const std::string& path);
     Bitmap(int width, int height);
     virtual void draw(Float x, Float y, Float rotation, Float scale,
                  FlipFlag flip = Normal) const;
     virtual void drawAtCentre(Float x, Float y, Float rotation,
                  Float scale,
                  FlipFlag flip = Normal) const;
     virtual Bitmap& createSubBitmap(int x, int y, int width,
                  int height) const;
     virtual void eraseSubBitmap(Bitmap& bitmap) const;
     virtual void reserveSubBitmaps(size_t number) const;
     virtual ~Bitmap();
};
```

Η κλάση κληρονομεί από την **Drawable** και την **Resource**, που σημαίνει ότι θα πρέπει να οριστεί το πεδίο **name** και μια συνάρτηση από την **Drawable**.

- `Bitmap(const std::string& path)`

Αυτός ο κατασκευαστής πρέπει να φορτώνει την εικόνα που βρίσκεται στο **path** σαν Texture και να το αποθηκεύσει εσωτερικά. Θα πρέπει να οριστούν τα πεδία **width, height.** Πρέπει να οριστεί το πεδίο **name=path.**

- `Bitmap(int width, int height)`

Αυτός ο κατασκευαστής πρέπει να κατασκευάζει ένα κενό Texture και να το αποθηκεύσει εσωτερικά.

- `void draw(Float x, Float y, Float rotation, Float scale, FlipFlag flip = Normal) const`

Η κλήση αυτή πρέπει να σχεδιάσει το Texture που έχει εσωτερικά το αντικείμενο στο σημείο **(x,y)** της οθόνης με κλίση **rotation** μοίρες**, κλίμακα scale** και αντεστραμμένο σύμφωνα με την τιμή του Enumeration **flip.** Η συντεταγμένη **(x,y)** θα πρέπει να βρίσκεται στην πάνω αριστερή γωνία του texture όταν σχεδιαστεί. Η κλίση **rotation** ξεκινάει από 0 όπου δεν θα περιστρέφει καθόλου και αυξάνεται ανάλογα την κλίση σύμφωνα με το μέρος δεύτερο. Το όρισμα **scale** ορίζει το πόσο θα αυξηθεί ή θα μειωθεί το μέγεθος του Texture όταν θα σχεδιαστεί. Για τιμή 1.0 θα μείνει ως έχει. Για μεγαλύτερες τιμές, θα αυξηθεί και για μικρότερες θα μειωθεί. Το Enumeration **flip** ορίζει το πως θα αντιστραφεί το Texture όταν θα σχεδιαστεί. Παίρνει τιμές από το **FlipFlag** Enumeration που βρίσκεται στο Header File του Interface.

- `void drawAtCentre(Float x, Float y, Float rotation, Float scale, FlipFlag flip = Normal) const`

Ίδια με την παραπάνω, με την μόνη διαφορά ότι το Texture θα πρέπει να σχεδιαστεί έτσι ώστε το **(x,y)** να βρίσκεται στο κέντρο του Texture και όχι στην πάνω αριστερή γωνία.

- `Bitmap& createSubBitmap(int x, int y, int width, int height) const`

Δημιουργεί και αποθηκεύει εσωτερικά μια εικόνα παιδί από το μητρικό Texture και το επιστρέφει. Η εικόνα παιδί θα προέρχεται από την περικοπή της μητρικής σύμφωνα με ένα ορθογώνιο που ορίζει η συνάρτηση. Πιο ειδικά, η πάνω αριστερή γωνία της μητρικής εικόνας είναι το (0,0) για το ορθογώνιο. Τα ορίσματα **x,y** ορίζουν

που θα βρίσκεται το ορθογώνιο σύμφωνα με αυτή την γωνία αυτή. Τα ορίσματα **width, height** ορίζουν το πλάτος και το ύψος του ορθογωνίου αντίστοιχα. Αφού δημιουργηθεί το παιδί, αποθηκεύεται σε λίστα ικανή να ανακτηθεί μέσω του Pointer που επιστρέφεται. Στο τέλος πρέπει να επιστραφεί το παιδί που δημιουργήθηκε.

- `void eraseSubBitmap(Bitmap& bitmap) const`

Αφαιρεί και διαγράφει την εικόνα-παιδί **bitmap** από την λίστα παιδιών.

- `void reserveSubBitmaps(size_t number) const`

Δεσμεύει χώρο στην μνήμη για τουλάχιστον **number** εικόνες-παιδιά. Ανάλογα την υλοποίηση του Interface, διαφέρει το τι θα χρειαστεί να δεσμευτεί. Συνήθως δεσμεύεται χώρος στην λίστα.

- `virtual ~Bitmap()`

Ο καταστροφέας πρέπει να αποδεσμεύει την εικόνα από την μνήμη καθώς και όλες τις εικόνες παιδιά.

- `void draw(Float x, Float y, Float rotation = 0.0f, Float scale = 1.0f) const`

Αυτή η κλήση προέρχεται από την **Drawable**. Κάνει την ίδια δουλειά με την πρώτη, μόνο που δεν δέχεται το όρισμα **flip.**

1.3 Η Abstract Κλάση Event

Σε αντίθεση με τα συμβάντα που έχουμε δει μέχρι τώρα, εδώ η κλάση Event έχει να κάνει με συμβάντα υλικού. Οι υλοποιήσεις αυτού του Interface χρειάζεται να ορίσουν μόνο τον κατασκευαστή όπου θα μετατρέπει το αντικείμενο σε εξειδικευμένο συμβάν. Η κλάση Event έχει ένα Union που περιέχει τρία διαφορετικά συμβάντα τα οποία ανάλογα για τον σκοπό που θα χρησιμοποιηθεί το συμβάν, θα οριστούν τα πεδία τους.

Ο κατασκευαστής θα δέχεται κάποιο τρίτο αντικείμενο (εκτός μηχανής) συμβάντος και θα κατασκευάζει ειδικά το εξειδικευμένο Event. Για αυτό, η κλάση Event παρέχει κάποιες βοηθητικές συναρτήσεις που προσαρμόζουν το αντικείμενο ανάλογα τον τύπο του Event.

Πίνακας 1.3.1: Περιεκτική αναπαράσταση της κλάσης Event

```
class Event {
public:
     struct DisplayEvent {
          int x, y;
          int width, height;
     };

     struct KeyboardEvent {
          KeyboardKey key;
          int unichar;
          unsigned int modifiers;
     };
     struct MouseEvent {
          int x, y, wheel;
          MouseButton button;
          float pressure;
          int dx, dy;
     };

     EventType type;

     union {
          DisplayEvent display;
          KeyboardEvent keyboard;
          MouseEvent mouse;
     };

     void toNoEvent();

     void toDisplayEvent(EventType type, int x, int y, int width,
                                                       int height);

     void toMouseEvent(EventType type, int x, int y, int wheel,
                MouseButton button, float pressure, int dx = 0,
                                                       int dy = 0);

     void toKeyboardEvent(EventType type, KeyboardKey key, int unichar,
                unsigned int modifiers);

};
```

Η κλάση έχει το Union με τα 3 πεδία: display, keyboard, mouse. Πέρα από αυτό έχει και το πεδίο Event που πρέπει να θέτεται ανάλογα το συμβάν. Οι τιμές αυτού το Enumeration παίρνονται από το ZetaConfig.hpp

Πίνακας 1.3.2: Το EventType Enumeration

```
enum class EventType {
     Window_Close,
     Window_Resize,
     Window_Lost_Focus,
```

```
        Window_Got_Focus,
        Mouse_Button_Down,
        Mouse_Button_Up,
        Mouse_Move,
        Keyboard_Key_Down,
        Keyboard_Key_Up,
        Keyboard_Char_Pressed,
        Ignore
};
```

Έστω ότι έχουμε ένα συμβάν από το SDK. Έχουμε φτιάξει την κλάση που κληρονομεί την Event και έχει έναν κατασκευαστή που δέχεται το συμβάν από το SDK. Στον κατασκευαστή τώρα πρέπει να καταλάβουμε τι συμβάν είναι. Ανάλογα την περίπτωση θα καλεστεί και η ανάλογη συνάρτηση:

- **Συμβάντα οθόνης:** Συμβάντα όπως κλείσιμο του παραθύρου, μετασχηματισμός του παραθύρου, κρύψιμο/εμφάνιση του παραθύρου κτλ., πρέπει να μετατρέψουν το συμβάν σε αντίστοιχο οθόνης καλώντας:

```
void toDisplayEvent(EventType type, int x, int y, int width, int height)
```

  - **type:** Ανάλογα τον τύπο του συμβάντος πρέπει να δοθεί κάποια τιμή από τον πίνακα 1.3.2.
  - **x, y:** Εάν το συμβάν έχει να κάνει με μετακίνηση του παραθύρου στην οθόνη, τότε πρέπει να οριστούν οι νέες συντεταγμένες. Εάν το συμβάν δεν έχει να κάνει με αυτό, μπορούν να οριστούν σε 0, που θα αγνοηθούν.
  - **width, height:** Εάν το συμβάν είχε να κάνει με μετασχηματισμό του μεγέθους του παραθύρου, τότε πρέπει τα ορίσματα αυτά να πάρουν το νέο μέγεθος. Εάν το συμβάν δεν έχει να κάνει με αυτό, μπορούν να οριστούν σε 0, που θα αγνοηθούν.
- **Συμβάν Πληκτρολογίου:** Συμβάντα όπως το πάτημα ή ελευθέρωση ενός πλήκτρου του πληκτρολογίου ή το πάτημα ενός πλήκτρου χαρακτήρα, πρέπει να μετατρέψουν το αντικείμενο σε συμβάν πληκτρολογίου μέσω της συνάρτησης:
- ```
  void toKeyboardEvent(EventType type, KeyboardKey key, int unichar,
                       unsigned int modifiers)
  ```
  - **type:** Ανάλογα τον τύπο του συμβάντος πρέπει να δοθεί κάποια τιμή από τον πίνακα 1.3.2.
  - **key:** Πρέπει να δοθεί ποιο πλήκτρο προκάλεσε το συμβάν. Οι τιμές που μπορεί να πάρει αυτό το Enumeration υπάρχουν στο αρχείο ZetaConfig.hpp.
  ```

    - **unichar:** Σε περίπτωση που πατήθηκε κάποιος χαρακτήρας στο πληκτρολόγιο, τότε πρέπει να βάλουμε σε αυτό το όρισμα τον χαρακτήρα. Ο χαρακτήρας πρέπει να είναι σε κωδικοποίηση UTF-8.
    - **modifiers:** Σε περίπτωση που μαζί με το πλήκτρο πατήθηκε κάποιο πλήκτρο εναλλαγής όπως πχ Shift + CTRL, τότε εδώ πρέπει να δοθεί το Bitfield που θα έχει όλα τα αυτά τα πλήκτρα που πατήθηκαν σε συνδυασμό. Οι τιμές που μπορούν να συνδυαστούν είναι στο Enumeration **KeyModifiers** στο αρχείο ZetaConfig.hpp.
- **Συμβάν ποντικιού:** Συμβάντα όπως το πάτημα/ελευθέρωση ενός πλήκτρου του ποντικιού και κούνημα του ποντικιού, πρέπει να μετατρέψουν το συμβάν μέσω της συνάρτησης:
- ```
  void toMouseEvent(EventType type, int x, int y, int wheel,
                    MouseButton button, float pressure, int dx, int dy)
  ```
    - **type:** Ανάλογα τον τύπο του συμβάντος πρέπει να δοθεί κάποια τιμή από τον πίνακα 1.3.2.
    - **x, y:** Σε περίπτωση που το συμβάν ήταν η κίνηση του ποντικιού, εδώ πρέπει να δοθεί η νέα συντεταγμένη του δείκτη.
    - **wheel:** Σε περίπτωση που η ροδέλα του ποντικιού κινήθηκε, εδώ θα δοθεί μια τιμή +- πόσο κινήθηκε.
    - **button:** Εδώ ορίζεται ποιο πλήκτρο του ποντικιού πατήθηκε. Η τιμή του Enumeration υπάρχει στο αρχείο ZetaConfig.hpp.
    - **pressure:** Για συσκευές με αισθητήρα πίεσης. Η μηχανή το αγνοεί προς το παρών.
    - **dx, dy:** Προαιρετικές τιμές που δίνονται σε περίπτωση που είναι διαθέσιμη η μεταβολή της θέσης του ποντικιού.

Με τις παραπάνω συναρτήσεις μπορεί να μετατραπεί το αντικείμενο σε συμβάν κατανοητό από την μηχανή. Εάν υπάρξει κάποιο συμβάν που δεν χρειάζεται η μηχανή, μπορεί να μετατραπεί σε κενό μέσω της κλήσης **toNoEvent().** Η μηχανή θα το αγνοήσει.

Όταν η κλάση υλοποιηθεί, θα πρέπει να δημιουργηθεί ένας κατασκευαστής που θα δέχεται το αντικείμενο του SDK που περιέχει το συμβάν και θα μετατρέπει το κατασκευασμένο αντικείμενο στο σωστό μέσω των παραπάνω κλήσεων.
  ```

1.4 Η Abstract Κλάση LoopRunner

Ο κύριος βρόγχος της μηχανής είναι δεδομένος πως λειτουργεί. Όμως κάποια στοιχεία που συνδέονται με το SDK πρέπει να διασπαστούν στην υλοποίηση για να μπορεί να γίνει ευέλικτο. Για παράδειγμα, το πως θα λάβει τα συμβάντα από το υλικό δεν το κάνει ο κύριος κώδικας της μηχανής αλλά το SDK.

Η κλάση LoopRunnger, φροντίζει ώστε να υλοποιηθεί ο βρόχος όπως ταιριάζει στο εκάστοτε SDK και να καλεί κάποιες εσωτερικές συναρτήσεις όποτε πρέπει.

Πίνακας 1.4.1: Η κλάση LoopRunner

```
class LoopRunner {
public:
     virtual void start()=0;
     virtual int getTargetFps() const=0;
     virtual void setTargetFps(int targetFps)=0;
     void setLoop(GeneralLoop* loop);
     GeneralLoop* getLoop() const;
     LoopRunner();
     virtual ~LoopRunner();
protected:
     void render();
     void update();
     void handleEvents(Event& event);
     bool isLoopRunning();
};
```

Βλέπουμε ότι υπάρχουν κάποιες συναρτήσεις προς υλοποίηση και κάποιες έτοιμες. Οι έτοιμες είναι για να καλούνται όταν χρειάζεται. Η πιο σημαντική είναι η start(). Στην υλοποίηση αυτής της συνάρτησης θα γίνει όλος ο βρόγχος. Στην ουσία μόλις καλεστεί αυτή, η εκτέλεση δεν θα βγει από αυτή όσο η μηχανή θα τρέχει.

- `void start()`

Εδώ πρέπει να υλοποιεί μια ελεγχόμενη ατέρμονη βρόγχος. Μέσα στον βρόγχο πρέπει να υπάρχει ένα περιβάλλον που θα μπορεί να περιμένει για συμβάντα εισόδου (υλικού) και να εκτελεί την **update()** συγχρονισμένα x φορές το δευτερόλεπτο. Γενικά πρέπει να υπάρχει ένα αντικείμενο που θα περιμένει για συμβάντα εισόδου και χρονικά συμβάντα προκειμένου η μηχανή να συνεχίσει και να εμφανίσει. Πιο εκτενή ανάλυση του βρόγχου έχει γίνει στο πρώτο μέρος. Γενικά ο βρόγχος χωρίζεται σε 3 στάδια.

  - **Στάδιο 1:** Εδώ η εκτέλεση παγώνει μέχρι να έρθει κάποιο συμβάν υλικού από τον χρήστη (είσοδος) ή να πρέπει να γίνει Frame. Ο κανόνας για τα

Frame είναι ότι πρέπει να γίνονται τόσα όσο ορίζει η μεταβλητή **targetFps** ανά δευτερόλεπτο**.** Δηλαδή πρέπει να υπάρχει κάτι σαν χρονόμετρο, που κάθε 1/**targetFps** δευτερόλεπτα να σημάνει ότι πρέπει να γίνει Frame. Το Frame στην ουσία πρέπει να είναι και αυτό ένα συμβάν που απλά θα κάνει τον βρόγχο να παίρνει άλλη πορεία από ότι θα έπαιρνε σε περίπτωση εισόδου χρήστη.

- **Στάδιο 2:** Εδώ ερχόμαστε όταν η εκτέλεση συνεχίζει, δηλαδή όταν σταματάει να περιμένει διότι έλαβε κάποιο συμβάν ή πρέπει να γίνει Frame. Σε κάθε περίπτωση, το συμβάν πρέπει να μετατραπεί σε αντικείμενο συμβάντος της μηχανής από την κλάση Event που αναλύσαμε παραπάνω. Όταν γίνει αυτό, τότε πρέπει να καλεστεί η **handleEvent()** με όρισμα αυτό το αντικείμενο. Εάν το συμβάν είναι για Frame, τότε πρέπει να εκτελεστεί και η **update().** Αφού γίνει αυτό, πρέπει να σημαίνουμε ότι πρέπει να σχεδιάσει η μηχανή. Συνήθως γίνεται με ένα Flag.
- **Στάδιο 3:** Εδώ φτάνουμε όταν έχουμε σημάνει ότι πρέπει να σχεδιάσουμε. Το ότι το Flag για σχεδίαση είναι true δεν σημαίνει ότι μπορούμε να σχεδιάσουμε. Στο μεσοδιάστημα που είχε καλεστεί η **update()**, μπορεί είχε υπάρξει συμβάν εισόδου. Το συμβάν εισόδου έχει προτεραιότητα από την σχεδίαση. Εάν προβούμε σε σχεδίαση παρόλο αυτά, τότε ο χρήστης θα νιώσει μια καθυστέρηση στην ενέργεια που έκανε γιατί προηγήθηκε η σχεδίαση που μπορεί να πάρει χρόνο. Για αυτό πριν κάνουμε την σχεδίαση πρέπει να ελέγξουμε αυτή την περίπτωση. Αν όντως υπάρχει συμβάν πριν την σχεδίαση, τότε αναβάλουμε την σχεδίαση και συνεχίσουμε το βρόγχο στον επόμενο κύκλο προκειμένου να το εξυπηρετήσουμε. Αυτό πρέπει να γίνεται όσο υπάρχουν ακόμα συμβάντα για εξυπηρέτηση. Όταν βρεθούμε σε κατάσταση που δεν έχουμε συμβάντα για εξυπηρέτηση, τότε μπορούμε να προβούμε σε σχεδίαση. Προσοχή! Η σχεδίαση πρέπει να γίνει εφόσον έχει σημάνει ότι πρέπει να γίνει. Όταν οι προϋποθέσεις το κάνουν εφικτό, τότε η σχεδίαση μπορεί να γίνει καλώντας την **render().** Αφού ολοκληρωθεί η σχεδίαση τότε πρέπει να καθαρίσουμε το flag για την σχεδίαση για να μην ξαναγίνει πριν την ώρα του (γίνεται μετά από κάθε Frame).

Όπως καταλαβαίνουμε, το **Στάδιο 1** υπάρχει σε κάθε κύκλο του βρόγχου και το **Στάδιο 2** εν μέρει, ανάλογα το αν θα γίνει Frame. Το **Στάδιο 3** εξαρτάται από το αν καλύπτονται οι προϋποθέσεις για σχεδίαση. Ο βρόγχος πρέπει να τρέχει όσο η συνάρτηση **isLoopRunnig()** επιστρέφει **true**. Στο τέλος της συνάρτησης μετά το βρόγχο, πρέπει να καθαρίζεται το περιβάλλον που φτιάχτηκε για τον βρόγχο, καθώς κατά 99% η μηχανή θα τερματίσει όταν η **isLoopRunning()** επιστρέψει **false**.

- `int getTargetFps() const`

Η συνάρτηση αυτή πρέπει να επιστρέψει τον ακέραιο που αντιπροσωπεύει τον ρυθμό που θα εκτελεστούν τα καρέ στον βρόχο. Ο αριθμός αυτός δεν πρέπει να είναι ο τρέχον ρυθμός καρέ (FPS) αλλά ο στόχος.

- `void setTargetFps(int targetFps)`

Ο στόχος της συνάρτησης είναι να ορίσει τον στόχο του ρυθμού Frame που πρέπει να τρέξει ο βρόγχος. Εφόσον είναι εφικτό, πρέπει να γίνεται καθώς ο βρόγχος τρέχει.

  - **targetFPS:** Ο στόχος του ρυθμού Frame που πρέπει να επιβληθεί.

- `LoopRunner()`

Ο default κατασκευαστής του αντικειμένου πρέπει να υπάρχει και να το προετοιμάζει για την κλήση της **start().**

1.5 Η Abstract Κλάση ShapeRenderer

Αυτό το Interface κάνει εφικτή την σχεδίαση διάφορων σχημάτων όταν απαιτηθεί. Πρέπει να θεωρείται ότι οι κλήσεις αυτές θα καλούνται πάντα όταν υπάρχει ένα OpenGL context. Δεν θα καλεστούν ποτέ πριν.

Πίνακας 1.5.1: Η κλάση ShapeRenderer

```
class ShapeRenderer {
public:
     virtual void holdBitmapDraw(bool vall)=0;

     virtual void drawEllipsis(float center_x, float center_y,
                               float radius_x, float radius_y,
                               const Color& borderC,
                               float thickness)=0;

     virtual void drawFilledEllipsis(float center_x, float center_y,
                 float radius_x, float radius_y, const Color& fillC)=0;

     virtual void drawRectangle(float a_x, float a_y, float b_x,
```

```
                        float b_y, const Color& borderC,
                        float thickness)=0;

     virtual void drawFilledRectangle(float a_x, float a_y, float b_x,
           float b_y, const Color& fillC)=0;

     virtual void drawLine(float x1, float y1, float x2, float y2,
                 const Color& fillC, float thickness)=0;

     ShapeRenderer();
     virtual ~ShapeRenderer();
};
```

- `void holdBitmapDraw(bool vall)`

Η συνάρτηση αυτή πρέπει να ενεργοποιεί/απενεργοποιεί μια κατάσταση αποδοτικής σχεδίασης για textures που είναι κομμάτια (sub-bitmaps) ενός μεγαλύτερου. Πιο ειδικά, κάποια SDK παρέχουν κάποιες τεχνικές βελτιστοποίησης σε περιπτώσεις όπως την παραπάνω. Εάν υπάρχουν, τότε πρέπει να ενεργοποιηθούν όταν η συνάρτηση καλεστεί με το **val=true** και να απενεργοποιηθούν όταν καλεστεί με **val=false.** Το τι βελτιστοποιήσεις θα γίνουν αφήνεται στο SDK και τον προγραμματιστή του Interface. Πρέπει να τονιστεί ότι η συνάρτηση αυτή θα καλεστεί για ενεργοποίηση όταν θα σχεδιαστούν πολλές υπό-εικόνες από μια μεγαλύτερή και μέχρι να καλεστεί ξανά για απενεργοποίηση. Εάν το SDK δεν παρέχει κάτι τέτοιο τότε η υλοποίηση δεν θα κάνει τίποτα.

- `void drawEllipsis(float center_x, float center_y, float radius_x, float radius_y, const Color& borderC, float thickness)`

Σχεδιάζει μια έλλειψη χωρίς γέμισμα (μόνο το περίγραμμα) με κέντρο τις συντεταγμένες οθόνης **(center_x, center_y)**, με ακτίνα στον άξονα x **radius_x** και στον άξονα y **radius_y.** Το χρώμα του περιγράμματος ορίζεται από το **borderC** και το πάχος από το **thickness.** Το εσωτερικό της έλλειψης πρέπει να είναι διαφανές.

- `void drawFilledEllipsis(float center_x, float center_y, float radius_x, float radius_y, const Color& fillC)`

Ίδια με την παραπάνω, μόνο που θα σχεδιάζει την έλλειψη με γεμισμένο το εσωτερικό με χρώμα **fillC**.

- `void drawRectangle(float a_x, float a_y, float b_x, float b_y, const Color& borderC, float thickness)`

Σχεδιάζει ένα ορθογώνιο χωρίς γέμισμα. Εδώ τα ορθογώνια διαφέρουν στο πως ορίζονται. Τα ορθογώνια έχουν 2 σημεία, το σημείο που βρίσκεται στην πάνω-αρι-

στερή γωνία **(a_x, a_y)**, και το σημείο που βρίσκεται στην κάτω-δεξιά γωνία **(b_x, b_y)**. Το χρώμα του περιγράμματος ορίζεται από το **borderC** και το πάχος από το **thickness.** Το εσωτερικό του ορθογωνίου πρέπει να είναι διαφανές.

- `void drawFilledRectangle(float a_x, float a_y, float b_x, float b_y, const Color& fillC)`

Ίδια με την παραπάνω, μόνο που θα σχεδιάζει το ορθογώνιο με γεμισμένο το εσωτερικό με χρώμα **fillC**.

- `void drawLine(float x1, float y1, float x2, float y2, const Color& fillC, float thickness)`

Σχεδιάζει μια γραμμή που ορίζεται από τα σημεία **(x1, y1)** και **(x2, y2).** Το χρώμα της γραμμής ορίζεται από το **fillC** και το πάχος από το **thickness.**

Ο κατασκευαστής συνήθως δεν χρειάζεται να κάνει τίποτα καθώς το Interface χρειάζεται για τις συναρτήσεις μόνο. Θα μπορούσαν να γίνουν Static αλλά η μηχανή χρειάζεται και το αντικείμενο εσωτερικά για να μπορεί να το δίνει όπου χρειάζεται.

1.6 Οι κλάσεις Sound και SoundInstance

Οι κλάσεις Sound και SoundInstance είναι στενά συνδεδεμένες και έχουν να κάνουν με τους ήχους που θα παίζει η μηχανή. Τα Sound είναι αντικείμενα που φιλοξενούν εσωτερικά τον πόρο του ήχου. Τα SoundInstance είναι αντικείμενα που δημιουργεί η Sound για να δίνει όπου χρειάζεται να παίξει ο ήχος. Η ανάγκη αυτής της δεύτερης είναι επειδή ο πόρος του ήχου είναι κοινόχρηστος μέσα στην μηχανή. Εάν κάθε κομμάτι της μηχανής που τον έχει ήθελε να τον ελέγξει, τότε θα υπήρχε συνωστισμός και ο ήχος μία θα έπαιζε και σε τυχαία στιγμή θα σταματούσε επειδή κάποιο άλλο κομμάτι της μηχανής ήθελε να σταματήσει. Η SounInstance είναι ειδικά αντικείμενα που επιτρέπουν τον ίδιο πόρο να παιχτεί πολλές φορές. Κάθε φορά που ζητείται ο ήχος αυτός, τότε του δίνεται ένα αντικείμενο SoundInstance που είναι σαν αντίγραφο του ήχου που μπορεί να το παίξει και να το σταματήσει όποτε θέλει το κομμάτι.

Πίνακας 1.6.1: Η κλάση Sound

```
class Sound: public Resource {
public:
     virtual SoundInstance& createNewInstance() const=0;
     Sound();
     Sound(const std::string& name);
```

```
    virtual ~Sound();
};
```

- `SoundInstance& createNewInstance() const`

Δημιουργεί και επιστρέφει ένα SoundInstance από το αρχείο που φιλοξενεί αυτό το αντικείμενο.

- `Sound()`

Ο προεπιλεγμένος κατασκευαστής πρέπει να κατασκευάζει ένα κενό ήχο που να μπορεί να παράγει SoundInstances.

- `Sound(const std::string& name)`

Ο κατασκευαστής με ένα όρισμα std::string πρέπει να φορτώνει το αρχείο ήχου από την διαδρομή **name** και να το φιλοξενεί εσωτερικά.

- `virtual ~Sound()`

Ο καταστροφέας πρέπει να καταστρέφει και το εσωτερικό αντικείμενο που φιλοξενεί.

Επειδή η κλάση Sound είναι Resouce, όταν η μηχανή θα καλεί τους κατασκευαστές θα ελέγχει για Exceptions. Για αυτό αν κάτι πάει στραβά κατά την κατασκευή η φόρτωση, τότε μπορεί να γίνει throw κάποιο Exception της μηχανής για να ειδοποιηθεί ο πυρήνας για το συμβάν.

Πίνακας 1.6.2: Η κλάση SoundInstance

```
class SoundInstance: public LuaPushable, public NonCopyable {
public:

    virtual void play()=0;
    virtual void stop()=0;
    virtual void setLoop(bool value)=0;
    virtual void setGain(float value)=0;
    virtual void setPan(float value)=0;

    virtual ~SoundInstance();
protected:
    const Sound *parentSound;
    float gain;
    float pan;

    bool loops;
};
```

Όπως είπαμε, τα αντικείμενα της κλάσης SoundInstance πρέπει να συμπεριφέρονται σαν αντίγραφα του ήχου (Sound) που τα παράγει. Αυτά όπου δοθούν θα έχουν έλεγχο του δικού τους αντικείμενου του ήχου.

- `void play()`

Αρχίζει να παίζει τον ήχο

- `void stop()`

Σταματάει να παίζει τον ήχο

- `void setLoop(bool value)`

Θέτει τον ήχο να επαναλαμβάνεται ή να μην αναλόγως την τιμή του **value.** Για **true** θα επαναλαμβάνεται. Η τιμή αυτή πρέπει να αποδοθεί στο μέλος **loops.**

- `void setGain(bool value)`

Θέτει την ένταση στον ήχο να είναι **value.** Η τιμή αυτή πρέπει να αποδοθεί στο μέλος **gain.**

- `void setPan(bool value)`

Θέτει την τοποθεσία του ήχου να είναι **value.** Η τοποθεσία έχει να κάνει με που ακούγεται ο ήχος, είτε πιο κοντά στο δεξί κανάλι είτε κεντρικά είτε αριστερά. Η τιμή **value** πρέπει να παίρνει τιμές από -1,0 μέχρι 1,0, όπου -1=Αριστερά, 0=Κέντρο, 1=Δεξιά. Η τιμή αυτή πρέπει να αποδοθεί στο μέλος **pan.**

- `virtual ~SoundInstance()`

Ο καταστροφέας πρέπει να καταστρέφει ότι έχει παραχθεί με την δημιουργία του αντικειμένου.

1.7 Η abstract κλάση AudioContext

Πέρα από τις παραπάνω κλάσεις, η μηχανή χρειάζεται κάτι επιπλέον για ελέγχει καθολικά όλους τους ήχους που παίζουν. Σε αυτό βοηθάει η κλάση AudioContext.

Πίνακας 1.7.1: Η κλάση AudioContext

```
class AudioContext {
public:
     virtual float getMainGain() const=0;
     virtual void setMainGain(float mainGain)=0;
     AudioContext();
};
```

- `float getMainGain() const`

Η συνάρτηση αυτή πρέπει να επιστρέφει την καθολική ένταση του συστήματος ήχου (Master Gain).

- `void setMainGain(float mainGain)`

Η συνάρτηση αυτή πρέπει να θέτει την καθολική ένταση (Master Gain) στην τιμή **mainGain.**

2 ΤΟ ΓΕΝΙΚΟ ΣΥΣΤΗΜΑ

Η ραχοκοκαλιά της μηχανής βρίσκεται στο Singleton αντικείμενο System. Το αντικείμενο αυτό φιλοξενεί όλα τα σημαντικά κομμάτια που χρειάζεται να λειτουργήσει η μηχανή. Αυτά τα αντικείμενα μπορούν ανακτηθούν μόνο μέσω της System.

Κατά την δημιουργία του System, ο προγραμματιστής πρέπει να της δώσει αυτά τα αντικείμενα. Συνήθως αυτά τα αντικείμενα είναι υλοποιήσεις των παραπάνω Interface.

Πίνακας 2.1: Η κλάση System

```
class System: public LuaPushable {
public:
     void start();

     RenderingContext& getRenderingContext();
     ResourceContext& getResourcesManager();
     AudioContext& getAudioContext();
     ShapeRenderer& getShapeRenderer();

     static System& getInstance();

     static System& create(RenderingContext* renderingContext,
                 ResourceContext* resManager, AudioContext* audio,
                 ShapeRenderer *renderer) {
           if (instance != nullptr) {
                 delete instance;
           }
           instance = new System(resManager, renderingContext, audio,
                                                 renderer);
           return *instance;
     }

     [[ noreturn ]]
     void shutdown();

     [[ noreturn ]]
     void abort(const std::string &error);

     void pushToLua(lua_State* lstate);
private:
     System(ResourceContext* resManager,
                       RenderingContext* renderingContext,
                       AudioContext *audio, ShapeRenderer *renderer);
     ~System();

     ResourceContext* resManager;
     RenderingContext* renderingContext;
     AudioContext* audio;
     ShapeRenderer* renderer;

     static System* instance;
};
```

Όπως βλέπουμε μεγάλο μέρος των συναρτήσεων είναι για την ανάκτηση των Modules του System. Πέρα από αυτές έχουμε και κάποιες που κάνουν περαιτέρω λειτουργίες.

- **start():** Ο σκοπός της συνάρτησης είναι να ξεκινήσει τον κύριο βρόγχο του παιχνιδιού. Στην ουσία καλεί την **MainLoop::start()** που θα δούμε παρακάτω. Πρέπει να τονιστεί ότι η συνάρτηση δεν θα επιστρέψει μέχρι να λήξει το παιχνίδι.
- **getInstance():** Κλασική συνάρτηση που έχουν όλες οι κλάσεις Singleton. Επιστρέφει το μοναδικό αντικείμενο της κλάσης.
- **create(...):** Σε αντίθεση με την πλειοψηφία των κλάσεων Singleton που το αντικείμενο δημιουργείται με την πρώτη κλήση της getInstance(), εδώ η System θέλει ειδική μεταχείριση. Το μοναδικό αντικείμενό της πρέπει πρώτα να δημιουργηθεί με την **create(...)** και ύστερα να ζητείται με την **getInstance().** Τα ορίσματα που παίρνει είναι τα Modules που θα φιλοξενήσει το System. Όπως βλέπουμε στον κώδικα, αν υπάρχει ήδη το αντικείμενο System τότε το καταστρέφει και το δημιουργεί εκ νέου με τα νέα ορίσματα. Στο τέλος επιστρέφει το αντικείμενο.
- **shutdown(), abort(error):** Και οι δυο συναρτήσεις κάνουν το ίδιο πράγμα, με την διαφορά ότι η **abort(error)** γράφει το κείμενο **error** στο Log File. Οι συναρτήσεις αυτές τερματίζουν την λειτουργία του προγράμματος αφού προηγηθεί ένας καθαρισμός. Ο καθαρισμός πάει με την εξής αυστηρή σειρά:
    1. Διαγραφή του τρέχοντος χάρτη και όλων των NPC/Enemies που δημιουργήθηκαν κατά την φόρτωσή του. Για τα NPC/Enemies που τοποθετήθηκαν μέσω της Lua η καταστροφή τους εξαρτάται από το Flag της καταστροφής που είχαν κατά την προσθήκη.
    2. Διαγραφή του παίχτη. Η μνήμη του παίχτη δεν αποδεσμεύεται, αλλά ότι δεδομένα είχε το αντικείμενο του επαναφέρονται στις αρχικές τιμές (Σαν ένα κενό Lifeform).
    3. Καταστροφή του CEGUI.
    4. Καταστροφή του AudioContext.
    5. Καταστροφή του ResourceContext. Εδώ όλα τα αρχεία που έχουν φορτωθεί θα καταστραφούν και κάθε αναφορά προς αυτά θα πάψει να ισχύει. Για αυτό και η καταστροφή του γίνεται στα σε αυτή την σειρά.

6. Καταστροφή του RenderingContext.
7. Καταστροφή του ShapeRenderer.

Αφού γίνουν οι παραπάνω καθαρισμοί, τότε το πρόγραμμα τερματίζει. Πρέπει να τονιστεί ότι ο καθαρισμός αυτός δεν είναι ολοκληρωτικός. Με άλλα λόγια δεν καταστρέφονται τα πάντα που έχουν δημιουργηθεί κατά την εκτέλεση, παρά μόνο αυτά που δημιούργησε η μηχανή. Αντικείμενα της μηχανής που δημιουργήθηκαν εκτός αυτής (πχ ένα AbilityClass που φτιάχτηκε μέσα στην Lua και αποθηκεύτηκε εκεί) δεν θα καταστραφούν λόγω ότι η μηχανή δεν ξέρει που βρίσκονται στην μνήμη. Έτσι κι αλλιώς, με τον τερματισμό του προγράμματος η LuaEngine καταστρέφεται και αυτή και μαζί όλα αυτά τα αντικείμενα. Η καταστροφή τους θα είναι “σκληρή”, σημαίνοντας ότι δεν θα καλεστούν οι Destructors και δεν θα αναζητήσουν αναφορές σε άλλα Modules.

- **pushToLua(lstate):** Συνάρτηση που πρέπει να υλοποιηθεί για να μπορεί το αντικείμενο να μπει κατά απαίτηση στο περιβάλλον της Lua.

Με τα παραπάνω είδαμε το τι κάνει η κλάση System περιεκτικά. Τώρα πρέπει να δούμε τι κάνουν τα επιμέρους Module που φιλοξενεί μέσα της.

2.1 Σύγχρονα και Ασύγχρονα Contextes

Πριν προχωρήσουμε στην ανάλυση των επιμέρους Modules, θα πρέπει να δούμε πως λειτουργούν κάποια άλλα μέλη των Modules. Ένα σημαντικό μέλος είναι η κλάση **Context**.

Όταν η κλάση **Context** κληρονομείται, τότε η κλάση που την κληρονόμησε μπορεί να εκτελέσει κάποιες λειτουργίες. Οι λειτουργίες αυτές εξαρτιούνται από τις υλοποιήσεις μιας άλλης κλάσης, της **ContextOperation.** Το **Context** έχει μια ουρά από **ContextOperations** που θα εκτελεστούν με σειρά FIFO, δηλαδή ο πρώτος που θα μπει στην ουρά θα είναι ο πρώτος που θα εκτελεστεί.

Που βοηθάει αυτό; Πριν αποφασιστεί να παγώσει η ανάπτυξη της μηχανής για γραφτεί αυτό το κείμενο, υπήρχε ένας στόχος που θα επέτρεπε την αξιοποίηση πολλών πυρήνων επεξεργαστή για διάφορες λειτουργίες μέσω νημάτων (Multithreading). Προς το παρών αυτό έχει παγώσει (μέχρι να ξεκινήσει πάλι η ανάπτυξη της μηχανής), όμως η βάση υπάρχει μέσω των σύγχρονων και ασύγχρονων Contextes.

Ένα Context θεωρείται σύγχρονο όταν συμβαδίζει με το κύριο νήμα, με άλλα λόγια δεν είναι ξέχωρο. Οι ενέργειες που θα εκτελεστούν από το Context θα είναι σε σειρά

με τις άλλες εντολές που βρίσκονται πριν την κλήση για την εκτέλεση των ενεργειών, σαν μια απλή κλήση συνάρτησης.

Σε αντίθεση με τα σύγχρονα, τα ασύγχρονα Contextes τρέχουν σε μοναδικό νήμα μέσω της βοηθητικής κλάσης **AsynchronousWorker**. Εφόσον δεν έχει υλοποιηθεί ακόμα, δεν θα αναλυθούν αυτές οι κλάσεις και θα επικεντρωθούμε στις σύγχρονες.

Αρχικά πρέπει να δούμε πως λειτουργεί η κλάση **Context.**

Πίνακας 2.1.1: Η Κλάση Context

```
template<typename ContextType>
class Context {
public:

	virtual void addOperationToQueue(ContextOperation<ContextType>&
operation) {
		operation.setHandled(false);
		operations.push(&operation);
	}

	virtual void executeOperations() {
		while (operations.size() > 0) {
			ContextOperation<ContextType> *operation =
								operations.front();
			operations.pop();
			operation->handle(*static_cast<ContextType*>(this));
			operation->setHandled(true);
			operation->notifyWaiters();
			if (operation->isToBeDeleted()) {
				delete operation;
			}
		}
	}

	Context() {

	}
	virtual ~Context() {
		executeOperations();
	}
protected:
	std::queue<ContextOperation<ContextType>*> operations;
};
```

Όπως βλέπουμε πρόκειται για template. Δέχεται τον τύπο του Context όταν θα κληρονομηθεί όπως θα δούμε παρακάτω. Έχει το πεδίο **opertaions** που είναι τύπου **std::queue<>** που δέχεται αντικείμενα τύπου **ContextOpertaion<ContextType>*.** Αυτό το πεδίο είναι η ουρά που χρειάζεται η κλάση για να λειτουργήσει.

Η συνάρτηση **addOperationToQueue(...)** δέχεται σαν όρισμα μια λειτουργία για να την προσθέσει στην ουρά. Πριν την προσθέσει, θέτει το flag της ότι έχει εκτελεστεί σε false για παν ενδεχόμενο. Ύστερα την προσθέτει στην ουρά.

Η συνάρτηση **executeOperations()** εκτελεί όλες τις ενέργειες στην σειρά μέσω της μεθόδου **handle(),** που δέχεται σαν όρισμα το **Context** την κάλεσε. Ύστερα θέτει την ενέργεια ότι ολοκληρώθηκε μέσω της **setHandled(true)**. Αφού γίνει και αυτό, τότε ενημερώνουμε τα νήματα που περιμένουν αυτή την διεργασία να ολοκληρωθεί, ότι όντως ολοκληρώθηκε μέσω της **notifyWaiters()**. Περισσότερα με αυτό παρακάτω. Στο τέλος, εάν η ενέργεια πρέπει να καταστραφεί μετά την εκτέλεσή της, τότε γίνεται αυτό ανάλογα την τιμή που θα επιστρέψει η **isToBeDeleted()**.

Κάτι που πρέπει να ειπωθεί είναι ότι όταν το Context καταστραφεί για τον x λόγο, τότε όλες οι ενέργειες πρέπει να εκτελεστούν, όχι μόνο για την σαφήνεια της λειτουργίας της κλάσης αλλά και για την αποφυγή τυχών Memory Leaks (πχ ενέργειες που χρειάζονται καταστροφή).

Μια κλάση που συνδέεται στενά με την **Context** είναι όπως είδαμε τη **ContextOperation.** Σε αυτή την abstract κλάση, ο προγραμματιστής καλείται να υλοποιήσει μια συνάρτηση που θα καλεστεί όταν εκτελέσει αυτή την ενέργεια το Context που θα την φιλοξενήσει.

Πίνακας 2.1.2: Η κλάση ContextOperation

```
template<typename ContextType>
class ContextOperation {
public:
     virtual void handle(ContextType& context)=0;

     bool isHandled() const;
     void setHandled(bool handled) {
           std::unique_lock<std::mutex> lk(mutex);
           this->handled = handled;
     }

     bool isToBeDeleted() const;

     void setToBeDeleted(bool toBeDeleted) {
           this->toBeDeleted = toBeDeleted;
     }

     void waitToBeCompleted() {
           std::unique_lock<std::mutex> lk(mutex);
           while (!handled)
                 done.wait(lk);
     }
```

```
        void notifyWaiters() {
              std::unique_lock<std::mutex> lk(mutex);
              done.notify_all();
        }

        ContextOperation();
        ContextOperation(bool toBeDeleted);
        virtual ~ContextOperation();
    protected:
        std::mutex mutex;
        std::condition_variable done;
        bool handled;
        bool toBeDeleted;
    };
```

Πρόκειται για template που δέχεται το είδος του Context σαν όρισμα για να το περάσει στον ορισμό της συνάρτησης **handle()**. Το είδος του Context δεν είναι τίποτα άλλο από κάποια κλάση που κληρονομεί την κλάση **Context** όπως είδαμε παραπάνω.

Επειδή τα αντικείμενα αυτά ίσως χρειαστούν να τροποποιούνται από πολλά νήματα, όπως βλέπουμε υπάρχει ένα **std::mutex** και μια **std::contdition_variable.**

Η συνάρτηση **handle(...)** δέχεται σαν όρισμα το Context που την κάλεσε. Ο προγραμματιστής στην κλάση που θα υλοποιήσει, πρέπει να βάλει μέσα σε αυτή τον κώδικα που θέλει να τρέξει το Context, καθώς αυτή η συνάρτηση θα καλεστεί όταν έρθει η ώρα.

Η **isHandled()** και **isToBeDeleted()** απλά επιστρέφουν τις τιμές των αντίστοιχων πεδίων χωρίς περαιτέρω ελέγχους.

Στη συνάρτηση **setHandled(...)**, ο σκοπός της είναι απλά να θέσει την τιμή στο πεδίο **handled**. Όμως επειδή πρόκειται για εγγραφή σε πολύ-νηματικό περιβάλλον, πρέπει να κλειδώσουμε τον κώδικα αυτόν για να αποτρέψουμε να τον εκτελέσει ταυτόχρονα άλλο νήμα για να αποφύγουμε το φαινόμενο Race Condition. Σε αυτό βοηθάει η **std::unique_lock**. Αυτό το αντικείμενο δέχεται στον Constructor του ένα mutex για να κλειδώσει. Εάν ο mutex είναι ήδη κλειδωμένος, τότε περιμένει μέχρι να μπορέσει να τον κλειδώσει. Όταν το καταφέρει, τότε το Construction του τελειώνει και συνεχίζει η εκτέλεση στο block που ορίστηκε μέχρι να βγει από αυτό το block. Όταν βγει από αυτό το block, τότε το αντικείμενο αυτό καταστρέφεται και ξεκλειδώνει το Mutex που κλείδωσε. Με αυτό τον τρόπο εγγυόμαστε ότι από την στιγμή που κατασκευάστηκε, δεν θα έχουμε άλλα νήματα στον παρακάτω κώδικα μέχρι το τέλος του Block.

Ο σκοπός της **waitToBeCompleted()** είναι να την καλεί το νήμα που δημιούργησε το αντικείμενο και να περιμένει μέχρι να το Context να εκτελέσει την ενέργεια πριν βγει από την συνάρτηση. Αυτό επιτυγχάνεται με τη Condition Variable **done**. Πρέπει να καλέσουμε την μέθοδο **wait()** της **done** για να κάνουμε το νήμα να περιμένει. Η **wait()** χρειάζεται ένα κλειδωμένο **std::unique_lock** για αυτό και κατασκευάζουμε ένα στην αρχή και περιμένουμε να κλειδωθεί. Όταν κλειδωθεί, τότε καλούμε την **wait()** με όρισμα το **std::unique_lock**. Αυτή η κλήση το ξεκλειδώνει και κάνει το νήμα που την κάλεσε να περιμένει μέχρι να κάποιο άλλο νήμα να ξυπνήσει την **done** και να συνεχίσει όπου και θα βγει από το την συνάρτηση. Όπως βλέπουμε, την **wait()** την έχουμε βάλει μέσα σε μια while() που ελέγχουμε το μέλος **handled**. Όσο αυτό είναι false, τότε θα καλείται η **wait()** μέχρι να γίνει true. Τι νόημα έχει αυτό; Η ανάγκη να ελέγχουμε το μέλος **handled** κάθε φορά που ξυπνάει η **wait()** υπάρχει γιατί αυτή η συνάρτηση είναι λίγο απρόβλεπτη για το πότε θα ξυπνήσει και θα πάψει να περιμένει. Εάν όσο περιμένει το νήμα το πρόγραμμα λάβει κάποιο σήμα (είτε από το λειτουργικό είτε από άλλη διεργασία), τότε υπάρχει πιθανότητα να ξυπνήσει πριν την ώρα της και να φέρει ανεπιθύμητα αποτελέσματα. Αυτό το φαινόμενο λέγεται και **Spurious Wake**. Για αυτό υπάρχει η ανάγκη να ελέγχεται μια επιπλέον μεταβλητή πριν βγούμε από την συνάρτηση για να ξέρουμε ότι όντως η **wait()** ξύπνησε από το πρόγραμμα και όχι από άλλο λόγο. Εδώ ξέρουμε ότι το μέλος **handled** θα αλλάξει μόνο μέσω της **setHandled(...)** που την καλεί το Context αφότου εκτέλεσε την ενέργεια. Έτσι ξέρουμε ότι όταν πάρει την τιμή true και η **wait()** ξυπνήσει, θα πρέπει να σταματήσουμε να περιμένουμε. Προσοχή!! Εάν αυτή η συνάρτηση καλεστεί από το ίδιο νήμα που βρίσκεται το Context θα γίνει αδιέξοδο(**Deadlock**) γιατί θα περιμένει τον εαυτό του να ξυπνήσει την **wait()** κάτι που είναι αδύνατον.

Η συνάρτηση **notifyWaiters()** καλείται από το Context όταν ολοκληρώσει την ενέργεια και ξυπνάει όλα τα νήματα που περιμένουν μέσω της **waitToBeCompleted()**. Αρχικά κλειδώνουμε το Mutex που μοιράζονται όλα τα νήματα, καθώς τα άλλα νήματα το έχουν ξεκλειδώσει μέσω της **wait()**. Ύστερα καλούμε την **notify_all()** της **done** για να ξυπνήσουμε όλα τα νήματα. Προσοχή! Εάν καλεστεί χωρίς προηγουμένως να έχει αποδοθεί στο μέλος **handled** η τιμή true, τότε αυτή η κλήση δεν θα έχει αποτέλεσμα στα νήματα που περιμένουν μέσω της **waitToBeCompleted()** για τους λόγους που είπαμε παραπάνω.

Ο απλός κατασκευαστής θέτει false σε **handled** και **toBeDeleted** ενώ ο δεύτερος θέτει την τιμή **toBeDeleted** σε αυτό που ορίζει το όρισμα. Ο καταστροφέας δεν κάνει τίποτα.

Με όλα τα παραπάνω, ο προγραμματιστής μπορεί να φτιάξει κάποιο είδος Context με το να κληρονομήσει την **Context** σε κάποια κλάση. Για αυτό το συγκεκριμένο Context μπορεί να κατασκευάσει λειτουργίες με το να κληρονομήσει την ContextOperations με όρισμα template το Context που θα τις πάρει και υλοποιώντας την **handle(...)**. Ένα παράδειγμα για το πως κατασκευάζονται αυτές οι κλάσεις δίνεται παρακάτω.

Πίνακας 2.1.3: Παράδειγμα δομής κλάσεων Context, ContextOperation

```
class ParadeigmaContext: public Context<ParadeigmaContext> {
     /* Ορισμός κλάσης */
};

class ParadeigmaOperation: public ContextOperation<ParadeigmaContext>{

     void handle(ParadeigmaContext& cntx){
           /* Κώδικας */
     }

};
```

2.2 Το RenderingContext

Το RenderingContext είναι ένα αντικείμενο που περιέχει το αντικείμενο του OpenGL context. Όπως είπαμε στο προηγούμενο κεφάλαιο, το αντικείμενο του SDK που περιέχει το OpenGL Context κλείνεται εσωτερικά μέσα σε μια κλάση που κληρονομεί την **Display**. Αυτή η κλάση δεν πρέπει να χρησιμοποιείται κατευθείαν για να κατασκευαστεί το OpenGL Context αλλά πρέπει να αφήσει την μηχανή να το κάνει μέσω της abstract κλάσης **RenderingContext.** Εκεί υπάρχει μια συνάρτηση που χρειάζεται υλοποίηση, για αυτό και καλείται να κληρονομηθεί η κλάση.

Πίνακας 2.2.1: Η κλάση RenderingContext

```
class RenderingContext: public Context<RenderingContext> {
public:

     virtual Display& getDisplay()=0;

     RenderingContext() {

     }
```

```
     virtual ~RenderingContext() {

     }
};
```

Όπως βλέπουμε, είναι ένα πολύ απλό Interface. Η ανάγκη για την υλοποίηση μιας τόσο απλής συνάρτησης, της **getDisplay(),** πηγάζει από τις διαφορετικές υλοποιήσεις και λειτουργίες του εκάστοτε SDK που χρησιμοποιείται. Η λειτουργία αυτής της συνάρτησης είναι να επιστρέφει το αντικείμενο Display που δημιούργησε το **RenderingContext**. Για παράδειγμα θα δούμε την υλοποίηση στην μηχανή.

Πίνακας 2.2.2: Η κλάση SynchronousRenderingContext

```
template<typename DisplayT>
class SynchronousRenderingContext: public RenderingContext {
public:
     Display& getDisplay() {
          return display;
     }

     void addOperationToQueue(ContextOperation<RenderingContext>&
                                                          operation) {
          std::lock_guard<std::mutex> lock(mutex);
          Context::addOperationToQueue(operation);
     }
     void executeOperations() {
          std::lock_guard<std::mutex> lock(mutex);
          Context::executeOperations();
     }

     SynchronousRenderingContext() :
                RenderingContext() {

     }
     ~SynchronousRenderingContext() {

     }
private:
     std::mutex mutex;
     DisplayT display;
};
```

Η κλάση πέρα από το ότι κληρονομεί την RenderingContext, γίνεται και Template δεχόμενο τον τύπο του αντικειμένου **Display**. Για να θυμηθούμε, η κλάση **Display** υλοποιείται από τον προγραμματιστή για να την προσαρμόσει στο SDK που χρησιμοποιεί. Έτσι στο όρισμα του Template θα πρέπει να μπει αυτή η καινούργια κλάση που έφτιαξε για να μπορέσει η **SynchronousRenderingContext** να κατασκευάσει

ένα αντικείμενο και μέσω αυτού ένα OpenGL Context. Αυτό γίνεται μέσω του μέλους **display** που παίρνει τον τύπο από το όρισμα του template. Έτσι όταν κατασκευαστεί το RenderingContext θα κατασκευαστεί και ένα αντικείμενο **DisplayT**.

Η υλοποίηση της **getDisplay()** εδώ είναι απλή: απλά επιστρέφει το αντικείμενο **Display** που κατασκευάστηκε. Πέρα από αυτή, βλέπουμε ότι κάποιες συναρτήσεις γίνονται Override όπως η **addOperationToQueue(...)** και η **exectureOperations()**. Το μόνο που προστίθεται σε αυτές είναι το κλείδωμα του mutex πριν την κλήση της μητρικής συνάρτησης. Αυτό γίνεται για να πετύχουμε Thread-Safety γιατί η κλάση Context μπορεί να χρησιμοποιηθεί σε πολύ-νηματικό περιβάλλον.

Τώρα όταν κατασκευαστεί ένα **SynchronousRenderingContext**, θα κατασκευαστεί και ένα Display ανάλογο του ορίσματος του template. Για αυτό και δεν πρέπει να κατασκευάζεται χωρίς λόγο, παρά μόνο όταν είναι να το περάσουμε σαν όρισμα στην **create(...)** της κλάσης **System** στην αρχή του προγράμματος.

2.3 Το ResourceContext

Το ResourceContext είναι ένα Interface που οι υλοποιήσεις του καλούνται να φορτώσουν και να φιλοξενήσουν κυρίως αρχεία. Υπάρχει ένα μεγάλο σύστημα από πίσω που λειτουργεί σαν Garbage Collector. Δηλαδή κάθε πώρος υπάρχει στην μνήμη μόνο αν κάποιο μέρος της μηχανής έχει αναφορά σε αυτό. Αν χαθεί κάθε αναφορά τότε διαγράφεται. Το σύστημα αποτελείται από διάφορα κομμάτια.

Οι κλάσεις των Πόρων

Κάθε τύπος πόρου έχει μια κλάση που φορτώνει και χειρίζεται τον πόρο. Όλες αυτές οι κλάσεις έχουν κοινή κλάση που κληρονομούν, την κλάση **Resource**.

Πίνακας 2.3.1: Η κλάση Resource

```
class Resource {
public:
	const std::string& getName() const {
		return name;
	}

	void setName(const std::string& name) {
		this->name = name;
	}

	Resource(const std::string& name) :
			name(name) {
```

```
	}
	virtual ~Resource() {

	}
protected:
	std::string name;
};
```

Η κλάση είναι απλή και έχει μόνο ένα μέλος, που είναι το όνομα του πόρου. Το όνομα αυτό συνήθως είναι η διαδρομή του αρχείου που καλείται να αναθέσουν οι κλάσεις που θα την κληρονομήσουν.

Πίνακας 2.3.2: Η κλάση File

```
class File: public Resource {
public:

	char* getData() const {
		return data;
	}

	size_t getSize() const {
		return size;
	}

	File();
	File(const std::string& path);
	~File();
protected:
	size_t size;
	mutable char *data;
};
```

Η κλάση **File** έχει την αρμοδιότητα να φορτώσει ένα αρχείο στην μνήμη. Αυτό που αξίζει να δούμε είναι η διαδικασία φόρτωσης του αρχείου. Η φόρτωση γίνεται με τον Constructor **File(const std::string&).**

Πίνακας 2.3.3: Κώδικας φόρτωσης αρχείου

```
File::File(const std::string& path) :
		Resource(path) {
	if (FileLoader::fileExists(path.c_str())) {
		FileLoader::FileStat fl = FileLoader::Load(path.data());
		data = fl.data;
		size = fl.size;
	} else {
		throw Exception(path, "File does not exist");
	}
}
```

Εδώ βλέπουμε αρχικά ότι ελέγχουμε μέσω του Interface **FileLoader** εάν το αρχείο υπάρχει. Αν υπάρχει τότε το φορτώνουμε μέσω της συνάρτησης **Load(...)**. Εδώ πρέπει να δούμε τι κάνει το **FileLoader**, για να καταλάβουμε τι επιστρέφει.

Πίνακας 2.3.4: Ο ορισμός του FileLoader

```
class FileLoader {
public:
	typedef struct FileStat {
		char *data;
		size_t size;
	};
	static FileStat Load(const char *path) {
		FILE* fp = fopen(path, "r");
		FileStat fl;
		if (fp != nullptr) {
			fseek(fp, 0, SEEK_END);
			fl.size = ftell(fp);
			fseek(fp, 0, SEEK_SET);
			fl.data = new char[fl.size + 2];
			fread(fl.data, sizeof(char), fl.size, fp);
			fl.data[fl.size] = 0;
			fclose(fp);
		} else {
			fl.data = nullptr;
			fl.size = 0;
		}
		return fl;
	}

	static bool fileExists(const char *path) {
		if (access(path, F_OK) != -1) {
			return true;
		} else {
			return false;
		}
	}
};
```

Βλέπουμε τον ορισμό του **FileStat** που είναι μια δομή με έναν Pointer στα δεδομένα του αρχείου και το μέγεθος του.

Στην Load, αρχικά φορτώνουμε το αρχείο. Αν φορτώθηκε εντάξει, τότε πηγαίνουμε στο τέλος του με την **fseek(fp, 0, SEEK_END)**. Από εκεί μπορούμε να βρούμε το μέγεθος του αρχείου με την **ftell(fp)** που επιστρέφει σε ποιο σημείο βρίσκεται ο δείκτης του αρχείου όπου και το βάζουμε στο πεδίο **size** του struct που θα επιστρέψουμε. Ύστερα επαναφέρουμε τον δείκτη στην αρχή του αρχείου. Με το μέγεθος που βρήκαμε, δεσμεύουμε μνήμη για να βάλουμε τα δεδομένα του αρχείου, τόση όσο είναι το μέγεθος. Ύστερα διαβάζουμε το αρχείο και το βάζουμε στο πεδίο **data** του

struct. Στο τέλος του πίνακα των δεδομένων του αρχείου, βάζουμε 0 για λόγους σαφήνειας. Εάν η **fread(...)** επιστρέψει NULL, τότε επιστρέφουμε το struct με NULL στον δείκτη δεδομένων και 0 στο μέγεθος.

Η **fileExists(...)** ελέγχει εάν το αρχείο υπάρχει μέσω της **access()**. Στην περίπτωση που επιστρέψει -1 σημαίνει ότι δεν γίνεται να διαβαστεί το αρχείο (λόγο δικαιωμάτων) ή δεν υπάρχει.

Μέσω του **FileLoader** φορτώνεται το αρχείο μέσα στο αντικείμενο της File. Γιατί όμως χρησιμοποιούμε ξέχωρη κλάση για την φόρτωση θα ρωτήσει κάποιος. Ο λόγος είναι για λόγους ευελιξίας.

Η κλάση **AnimationPack** έχει ως αρμοδιότητα να φορτώνει αρχεία *.anim, να τα κάνει parse και να φιλοξενεί μέσα της όλα τα Animation του αρχείου.

Πίνακας 2.3.5: Η κλάση AnimationPack

```
class AnimationPack: public Resource {
public:

     const Animation& getAnimation(const std::string& name) const;
     const SpriteSheet& getSpriteSheet() const;
     std::vector<std::string> listAnimationNames() const;

     AnimationPack();
     AnimationPack(const std::string& path);
     ~AnimationPack();
protected:
     const SpriteSheet *spriteSheet;
     ZSmallMap<std::string, Animation> animations;
};
```

Η **getAnimation(...)** μας επιστρέφει το Animation με το όνομα **name**. Αν δεν υπάρχει αυτό το όνομα, τότε επιστρέφεται ένα **NullAnimation** που είναι ένα κοινόχρηστο αντικείμενο που λειτουργεί πλήρως σαν Animation.

Η **getSpriteSheet()** επιστρέφει το SpriteSheet που έχει αναφορά το αρχείο αυτό.

Η **listAnimationNames()** επιστρέφει ένα **std::vector<>** με τα ονόματα όλων των διαθέσιμων Animations του αντικειμένου αυτού.

Ο απλός κατασκευαστής κατασκευάζει ένα κενό **AnimationPack** με Spritesheet ένα **NullSpritesheet** και κανένα Animation.

Ο δεύτερος κατασκευαστής είναι το ζουμί της κλάσης. Εδώ φορτώνεται το αρχείο που θέτει το **path** και δημιουργεί την λίστα των Animations μέσα στο αντικείμενο.

Πίνακας 2.3.6: Η κλήση AnimationPack::AnimationPack(const std::string&)

```
AnimationPack::AnimationPack(const std::string& path) :
          Resource(path) {
     // Ask the RESCON for the file the path says.
     auto& resMan = System::getInstance().getResourcesManager();
     xmlpp::DomParser parser;

     try {
          File animFile(path);

          // Parse the file
          parser.parse_memory_raw((unsigned char*) (animFile.getData()),
                    animFile.getSize());
          // Validate the XML through the pre-parsed XSD Schema
          XMLSchemaValidator::getInstance().getValidator(

     XMLSchemaValidator::ResourceValidator::Animations).validate(
                    parser.get_document());
     } catch (const std::exception &ex) {
          // If execution comes here, then maybe file is corrupt
          throw Exception(path, ex.what());
     }

     xmlpp::Element *root = parser.get_document()->get_root_node();

     // Get the Referenced SpriteSheet from RESCON
     spriteSheet = &resMan.getSpriteSheet(
               StringOperations::translateUnixPath(path,
               root->get_attribute_value("spriteSheet").c_str()));
     // Execute the XPATH Query to find all <anims>
     xmlpp::NodeSet anims = root->find("//anim");

     // For all the anims
     for (auto node : anims) {
          animations[
          static_cast<xmlpp::Element*>(node)->get_attribute_value(
                    "name").c_str()].set(node, *this);
     }
}
```

Αρχικά ζητάμε από το σύστημα το **ResourceContext** (λέγεται και ResourceManager). Το αρχείο όπως είπαμε στο προηγούμενο μέρος είναι ένα XML αρχείο με αυστηρή μορφή. Έτσι θα χρησιμοποιήσουμε τον **xmlpp::DomParser** για να κάνουμε parse το αρχείο. Μέσα στην **try** κατασκευάζουμε ένα αρχείο **File** και του φορτώνουμε το αρχείο που λέει το όρισμα της συνάρτησης για να το έχουμε στην μνήμη του **animFile**. Ύστερα καλούμε την **parse_memory_raw()** της βιβλιοθήκης libxml++ για να μας κάνει Parse τα δεδομένα της μνήμης του animFile. Αφού γίνει αυτό, τότε πρέπει να ελέγξουμε αν το αρχείο είναι έγκυρο σύμφωνα με το XSD Schema που ορίζει η μηχανή. Το αντικείμενο Singleton **XMLSchemaValidator** περιέχει φορτωμένα όλα τα XSD Schemas της μηχανής και μπορούμε να ανακτήσουμε

όποιο θέλουμε για να ελέγξουμε κάποιο XML αρχείο. Έτσι ζητάμε το XSD για τα Animations και κάνουμε τον έλεγχο μέσω της **validate()**. Αν κάτι πάει στραβά μέχρι στιγμής, τότε γίνεται throw ένα exception για να ειδοποιηθεί αυτός που φορτώνει το αρχείο. Οι περιπτώσεις που μπορεί να γίνει αυτό είναι είτε να μην υπάρχει το αρχείο, είτε το αρχείο να είναι κακώς μορφοποιημένο ή να μην είναι έγκυρο για το XSD.

Αφού περάσει αυτό το στάδιο, τότε από το δομημένο δένδρο που έφτιαξε ο Parser παίρνουμε το attribute “spriteSheet” και ζητάμε από τον ResourceManager να μας φέρει έναν Pointer από το αρχείο που ορίζει το attribute αυτό. Η συνάρτηση **translateUnixPath(...)** δέχεται μια διαδρομή αρχείου και την μετατρέπει σε απόλυτη. Δηλαδή αν η διαδρομή περιέχει τις ειδικές διαδρομές “.” ή “..” τότε τις μετατρέπει στις απόλυτες που αντιστοιχούν.

Αφού πάρουμε τον Pointer στο spritesheet, τότε μέσω ενός XPATH Querry βρίσκουμε όλα τα στοιχεία <anim>. Αφού τα λάβουμε, τότε για καθένα από αυτά κατασκευάζουμε ένα αντικείμενο **Animation** στο πίνακα και του βάζουμε δεδομένα μέσω της **set(...)**. Το πώς λειτουργεί αυτό θα το δούμε σε άλλο κεφάλαιο. Περιεκτικά θα πούμε ότι παίρνει το στοιχείο <anim> και ανάλογα τα δεδομένα που έχει κατασκευάζει το **Animation.**

Πίνακας 2.3.7: AnimationPack::getAnimation(const std::string&) και Καταστροφέας

```
const Animation& AnimationPack::getAnimation(const std::string& name)
                                                                  const {
     // Find the animation with that name
     auto itr = animations.find(name);
     if (itr != animations.end()) {
           // if exists in the pack, return it
           return itr->second;
     } else {
           // if it does not exist in the pack return the NullAnimation
           return NullReference<Animation>::getNull();
     }
}

AnimationPack::~AnimationPack() {
     // Finaly, release the Spritesheet
     ResourceContext* resources =
&System::getInstance().getResourcesManager();
     if (resources != nullptr) {
           resources->releaseResource(*spriteSheet);
     }
}
```

Ενδεικτικά θα δούμε τις παραπάνω δυο συναρτήσεις για να τις έχουμε για αναφορά γιατί αυτό το κώδικα θα υπάρχει αρκετά σε όλες τις κλάσεις των πόρων.

Η **getAnimation()** ψάχνει στο Hashtable το **name**. Αν το βρει τότε επιστρέφει το **Animation** με αυτό το **name**. Αν δεν το βρει τότε σημαίνει ότι έγινε κάποιο λάθος, καθώς δεν υπάρχει σε αυτό το αρχείο αυτό το Animation και έτσι επιστρέφει το **NullAnimation**.

Ο καταστροφέας αρχικά καλεί τον **ResourcesManager** και ελέγχει αν υπάρχει ο Pointer του. Αν υπάρχει τότε του λέει ότι δεν χρειάζεται πλέον το Spritesheet που είχε ζητήσει όταν κατασκευάστηκε. Ο λόγος που ελέγχουμε αν υπάρχει ο Pointer του ResroucesManager είναι επειδή μπορεί κατά την φάση αυτή που καταστρέφεται ο πόρος, να γίνεται τερματισμός του προγράμματος και να έχει καταστραφεί ο ResourcesManager.

Η κλάση **AnimationClass** ευθύνεται για την φόρτωση των αρχείων XML AnimationClass και δόμηση των επιμέρους κομματιών που ορίζει.

Πίνακας 2.3.8: Η κλάση AnimationClass

```
class AnimationClass: public Resource {
public:
     const Animation& getAnimation(const std::string& name,
                 LifeformState::CombinedState fallbackAnimation =
                             LifeformState::Action::Idle
                             | LifeformState::Direction::Down) const;

     const Animation& getAnimation(LifeformState::CombinedState state)
                                                                  const;

     const Rectangle& getBoundBox() const;
     const Rectangle& getTargetBox() const;
     const Ellipsis& getShadow() const;
     bool hasShadow() const;

     AnimationClass();
     AnimationClass(const std::string& path);
     virtual ~AnimationClass();
protected:
     ZSmallMap<std::string, const Animation*> animations;
     Vector2D<const Animation*> primAnimations;
     Rectangle boundBox;
     Rectangle targetBox;
     Ellipsis shadow;
     const Animation* nullAnimation;
     bool has_Shadow;

     void retrivePrimitives();
};
```

Εδώ θα αναλύσουμε κυρίως λειτουργία στην θεωρία καθώς ο λειτουργικός κώδικας είναι λίγο μεγάλος. Θα ξεκινήσουμε με την φόρτωση που γίνεται με τον κατασκευαστή που δέχεται την διαδρομή του αρχείου σαν όρισμα.

Αρχικά όλα τα PrimitiveAnimations (Animations για τις βασικές κινήσεις και καταστάσεις) αρχικοποιούνται. Τα Primitives υπάρχουν σαν δείκτες σε Animations μέσα στην δομή **Zeta::Vector2D** που λειτουργεί σαν δισδιάστατος δυναμικός πίνακας. Ο λόγος που τοποθετούνται σε δισδιάστατο πίνακα είναι λόγω ότι κάθε στοιχείο αντιστοιχεί σε κάποιο συνδυασμό της κατάστασης που ζητείται. Θα δούμε περισσότερα παρακάτω. Η αρχικοποίηση γίνεται με ανάθεση στους δείκτες σε δείκτη του **NullAnimation**. Έτσι κάθε Primitive έχει κάτι να δείξει στην χειρότερη περίπτωση, ακόμα και αν αυτό δεν έχει κάποιο νόημα για τον χρήστη (σκουπίδια).

Σε δεύτερη φάση, γίνεται Parse και επικύρωση του αρχείου όπως είδαμε στο προηγούμενο παράδειγμα. Ύστερα από το αρχείο που έγινε Parse, κατασκευάζουμε τα μέλη **boundBox, targetBox, shadow** σύμφωνα με τα δεδομένα που έχουν τα αντίστοιχα στοιχεία στο XML αρχείο. Εάν η σκιά υπάρχει, τότε το μέλος **has_Shadow** παίρνει την τιμή **true**. Το μέλος **shadow** θέτεται να σχεδιάζει το Background και παίρνει το χρώμα του γεμίσματος RGBA=0x000000BA.

Σε τρίτη φάση, συλλέγονται όλα τα στοιχεία **<Action>** του XML αρχείου και για καθένα από αυτά, για κάθε εσωτερικό στοιχείο (που συνήθως είναι οι 4 κατευθύνσεις) γίνεται μια εγγραφή στο Hashtable **animations** με όνομα κλειδιού το όνομα του **<Action>+"/"+** το όνομα του παιδιού. Στην εγγραφή αυτή βάζουμε το Animation που λέει το παιδί αυτό(attribute "animation") από το AnimationPack που ορίζει πάλι αυτό(attribute "file").

Σε τελικό στάδιο καλείται η **retrievePrimitives()**. Αυτή η συνάρτηση αναζητεί στο μέλος **animations** για Animations με όνομα που ταιριάζει σε Primitive Animation και παίρνει τον δείκτη και τον αντιγράφει στο **primAnimations** σε κατάλληλη θέση. Για παράδειγμα:

Πίνακας 2.3.9: Μέρος της συνάρτησης AnimationClass::retrievePrimitives()

```
void AnimationClass::retrivePrimitives() {
// Get the Primitives for Movement

     auto itr = animations.find("__Movement__/Up");
     if (itr != animations.end()) {
           primAnimations((LifeformState::Direction::Up - 1),
                       (LifeformState::Action::Moving >> 4) - 1) =
                                                      itr->second;
     }
```

```
     itr = animations.find("__Movement__/Down");
     if (itr != animations.end()) {
          primAnimations((LifeformState::Direction::Down - 1),
                         (LifeformState::Action::Moving >> 4) - 1) =
                                                                  itr-
>second;
     }
[...]
```

Όπως βλέπουμε, όλα τα Animations έχουν όνομα της μορφής: “ActionName/Direction”. Τα Actions με τα παρακάτω ονόματα θεωρούνται Primitive:

- __Movement__
- __Idle__
- __Death__
- __Dead__

Τα παραπάνω εντοπίζονται από την συνάρτηση και τοποθετούνται στην κατάλληλη θέση στον πίνακα **primAnimations**.

Οι επόμενες συναρτήσεις που χρήζουν ανάλυσης είναι οι συναρτήσεις **getAnimation(...)**. Η πρώτη που δέχεται όρισμα ένα **std::string** και έναν **Integer** έχει ως αρμοδιότητα να επιστρέψει το Animation με όνομα **name**. Σε περίπτωση που το Animation αυτό δεν υπάρχει, τότε επιστρέφεται το Primitive Animation που είναι ο συνδυασμός της κατάστασης και της κατεύθυνσης(**LifeformState**) που ορίζεται στο **fallbackAnimation**. Για αυτούς τους συνδυασμούς θα δούμε σε άλλο κεφάλαιο.

Η πιο απλή έκδοση της συνάρτησης δέχεται μόνο το **LifeformState(Integer)**. Σε αυτή την περίπτωση θα επιστρέψει το Primitive Animation που ορίζει.

Το τελευταίο που πρέπει να εξηγήσουμε εδώ είναι ο καταστροφέας. Εδώ δεν καταστρέφουμε τίποτα γιατί ότι Animation έχουμε πάρει το έχουμε πάρει σαν αναφορά από διάφορα AnimationPacks. Έτσι το μόνο που έχουμε να κάνουμε είναι για κάθε Animation να πούμε στον ResourcesManager ότι δεν χρειαζόμαστε πλέον το AnimationPack που το περιέχει.

Η τελευταία κλάση που θα δούμε είναι η **SpriteSheet**. Αυτή αναλαμβάνει να φορτώνει αρχεία XML *.sprites και να κατασκευάζει εσωτερικά τα μεμονωμένα Sprites από την μητρική εικόνα.

Πίνακας 2.3.10: Η κλάση Spritesheet

```
class SpriteSheet: public Resource {
public:

     const Bitmap& getSprite(const std::string& path) const;

     const Bitmap& getImage() const;

     SpriteSheet();
     SpriteSheet(const std::string& path);
     ~SpriteSheet();
protected:
     ZSmallMap<std::string, Bitmap*> sprites;
     const Bitmap *image;
private:
     void handleNode(const std::string& curName,
                                    xmlpp::Element* curNode);
};
```

Η λειτουργία της φόρτωσης είναι ανάλογη των παραπάνω, δηλαδή μέσω του κατασκευαστή με το όρισμα **std::string**. Αρχικά το αρχείο XML γίνεται Parse και επικυρώνεται μέσω του αντίστοιχου XSD Schema. Ύστερα φορτώνεται η μητρική εικόνα από την διαδρομή που ορίζει το στοιχείο **<img>**. Μετά καλείται η αναδρομική συνάρτηση **handleNode(...)**.

Πίνακας 2.3.11: Η συνάρτηση SpriteSheet::handleNode(const std::string&, xmlpp::Element*)

```
void SpriteSheet::handleNode(const std::string& curName,
      xmlpp::Element* curNode) {

 // For all the <spr... /> Nodes that exist in this <dir>
 for (xmlpp::Element* nd =
      static_cast<xmlpp::Element*>(curNode->get_first_child("spr"));
            nd != nullptr;
            nd = static_cast<xmlpp::Element*>(nd->get_next_sibling())) {

      // Create a new subbitmap with the attributes the <spr> defines
      // The key name is full path so far + the <spr> "name" attribute
      sprites[curName + "/" + nd->get_attribute_value("name")] =
                  &image->createSubBitmap(
                              strtol(
                  nd->get_attribute_value("x").c_str(), nullptr, 0),
                              strtol(
                  nd->get_attribute_value("y").c_str(), nullptr, 0),
                              strtol(
                  nd->get_attribute_value("w").c_str(), nullptr, 0),
                              strtol(
                  nd->get_attribute_value("h").c_str(), nullptr, 0));
 }

 // For all the <spr... /> Nodes that exist in this <dir>
```

```
  for (xmlpp::Element* nd =
       static_cast<xmlpp::Element*>(curNode->get_first_child("dir"));
             nd != nullptr;
             nd = static_cast<xmlpp::Element*>(nd->get_next_sibling())) {

       // Call the same function for this <dir> and with the new name
       handleNode(curName + "/" + nd->get_attribute_value("name"), nd);
  }
}
```

Η ανάγκη αυτής της συνάρτησης υπάρχει για να σχηματιστεί το όνομα του Sprite. Αυτό γιατί το εργαλείο DarkFunction τα ταξινομεί σε δένδρο φακέλων. Για παράδειγμα:

Πίνακας 2.3.12: Μέρος αρχείο *.sprites

```
       ...
       <dir name="/">
             <dir name="Cast_Up">
                   <spr name="0" x="36" y="28" w="58" h="98" />
                   <spr name="1" x="164" y="28" w="58" h="98" />
                   ...
```

Στο παραπάνω μέρος του αρχείου, το πρώτο στοιχείο **<spr>** θα πρέπει να πάρει όνομα μέσα στο SpriteSheet “/Cast_Up/0”.

Θεωρούμε ότι στο όρισμα της συνάρτησης **curName** έχουμε την διαδρομή μέχρι και το <dir> **curNode**. Το μέλος κάθε διαδρομής καθορίζεται από την τιμή του attribute **name** κάθε στοιχείου. Έτσι αρχικά ελέγχουμε αν υπάρχουν στοιχεία <spr> για να κατασκευάσουμε τα sprites που ορίζουν. Αυτό το κάνουμε με την πρώτη for(). Τα casting που κάνουμε μέσα στην for είναι απολύτως ασφαλή γιατί η συνάρτηση θα καλείται πάντα αφού το αρχείο έχει περάσει επιτυχώς την επικύρωση με το XSD. Έτσι για κάθε στοιχείο <spr> κατασκευάζουμε ένα Sub-Bitmap από τα δεδομένα που έχει και το βάζουμε στο **sprites** με κλειδί την πλήρη διαδρομή που έχουμε μέχρι στιγμής + το όνομα του <spr> (name attribute).

Αφού κατασκευάσαμε ότι τυχών Sprites υπάρχουν κάτω από το συγκεκριμένο <dir>, αρχίζουμε να ψάχνουμε για άλλα στοιχεία <dir> με την δεύτερη for(). Εκεί για κάθε στοιχείο καλούμε εκ νέου την **handleNode(...)** με ορίσματα το πλήρες όνομα μέχρι τώρα + το όνομα του στοιχείου που έχουμε τώρα στην for().

Η αρχική κλήση της **handleNode(...)** καλείται με κενή συμβολοσειρά για όνομα και το πρώτο πρώτο <dir> στην ιεραρχία.

Και εδώ τελειώνουμε με τις επιμέρους κλάσεις πόρων. Υπάρχουν και άλλες κλάσεις που κληρονομούν την Resource αλλά καλύτερα ταιριάζουν να αναλυθούν σε άλλα κεφάλαια.

Το Template SharedResource

Το Template αυτό έχει ως στόχο να κάνει έναν πόρο κοινόχρηστο. Έχει ένα σύστημα μετρήματος των αναφορών που συνδυασμό με το **ResourcesContext** κάνουν την ουσιαστική δουλειά του χειρισμού των πόρων. Η συγκεκριμένη λειτουργεί σαν κέλυφος του πόρου για το **ResourcesContext**, το οποίο χειρίζεται αυτά αντί του πόρου του ίδιου.

Το Template δέχεται σαν όρισμα την κλάση που θα χειρίζεται τον εκάστοτε πόρο. Η κλάση όπως είναι φυσικό, πρέπει να κληρονομεί την Resource. Το Template κληρονομεί την **NonCopyable** για να αποφευχθεί η κατά λάθος αντιγραφή του πόρου και την εξασφάλιση της μοναδικότητας μέσα στην μηχανή. Αυτό γίνεται μέσω της διαγραφής των κατασκευαστών αντιγραφής μέσα στην **NonCopyable.**

Πίνακας 2.3.13: Το Template SharedResource

```
template<typename RType>
class SharedResource: public NonCopyable {
public:
     bool isValid() const;
     void setValid(bool valid);
     int getReferenceCount() const;
     const RType& getInstance() const;
     void setInstance(RType& instance);

     void load(const std::string& path) {
           // If there is already an Instance, destroy it
           if (instance != nullptr) {
                 delete instance;
           }
           try {
                 // try to costruct and Load the Resource
                 instance = new RType(path);
           } catch (Exception& ex) {
                 // If any Engine related Exception was thrown
                 // Destroy the Instance if it actually was created
                 delete instance;
                 // Log the Error
                 Logger::getInstance().write(
                             "During loading: " + path + " Reason: " +
                                         ex.reason(),
                                         Logger::MessageType::Error);
                 // Invalidate the resource
                 valid = false;
                 // Clear the Instance Pointer
```

```
                instance = nullptr;
                return;
            } catch (std::exception& ex) {
                // If any General related Exception was thrown
                // Destroy the Instance if it actually was created
                delete instance;
                // Log the Error
                Logger::getInstance().write(
                            "During loading: " + path + " Reason: "
                                        + std::string(ex.what()),
                            Logger::MessageType::Error);
                // Invalidate the resource
                valid = false;
                // Clear the Instance Pointer
                instance = nullptr;
                return;
            }
            // If the resource was loaded successfully
            // Set the Name of the Resource to the Name
            instance->setName(path);
            // Validate the resource
            valid = true;
        }

        void increaseReference();
        void decreaseReference();

        SharedResource();
        SharedResource(const std::string& path);
        SharedResource(RType& resource);
        ~SharedResource();
    private:
        RType* instance;
        int referenceCount;
        bool valid;
    };
```

Υπάρχει ένας δείκτης στον ουσιαστικό πόρο, ένας αριθμός των αναφορών προς τον πόρο και ένα flag για την εγκυρότητα. Ο αριθμός των αναφορών χειρίζεται από τον **ResourcesContext** αποκλειστικά. Εδώ η μόνη συνάρτηση που αξίζει να δούμε είναι η **load(...)** καθώς οι άλλες απλά επιστρέφουν και δίνουν τιμές στα μέλη. Αυτές θα τις δούμε αργότερα.

Η **load(...)** δέχεται σαν όρισμα την διαδρομή του αρχείου που θα φορτωθεί. Αρχικά ελέγχει αν υπάρχει ήδη κάτι στην μνήμη του δείκτη. Αν υπάρχει, τότε καταστρέφει τον πόρο που υπάρχει εκεί. Κατασκευάζει τον πόρο καλώντας τον κατασκευαστή της κλάσης(που ορίστηκε στο template) με όρισμα std::string και του περνάει την διαδρομή του αρχείου. Σύμφωνα με την πολιτική της μηχανής γύρω από τις κλάσεις που κληρονομούν την Resource, πρέπει ο κατασκευαστής που δέχεται ένα std::string να φορτώνει το αρχείο που λέει αυτό το όρισμα. Έτσι κατασκευάζεται και ο δείκτης της

μνήμης του μπαίνει στο **instance.** Αν κάτι πήγε στραβά και ο κατασκευαστής έκανε throw κάποιο excpetion, τότε καταστρέφεται ο πόρος, αναφέρεται στο Log File και ακυρώνεται. Αν όλα πάνε καλά, τότε του θέτεται σαν όνομα η διαδρομή του αρχείου και γίνεται επικύρωση για να μπορεί να χρησιμοποιηθεί.

Το ResourceContext

Ήρθε η ώρα να αναλύσουμε τον πυρήνα του συστήματος διαχείρισης πόρων. Ο πυρήνας έχει την αρμοδιότητα να φορτώνει τους πόρους και να δίνει αναφορές όπου ζητούνται. Πρέπει να εγγυάται την μοναδικότητά τους και να κρατούνται στην μνήμη μόνο όσο χρειάζονται.

Ο πυρήνας λειτουργεί ως εξής. Όταν ζητείται ένας πόρος, τότε αναζητείται στους πίνακες των πόρων που είναι φορτωμένοι. Αν υπάρχει εκεί, τότε δεν φορτώνει τίποτα, επιστρέφει μια αναφορά και αυξάνει τον αριθμό των αναφορών για αυτό τον πόρο. Αν δεν υπάρχει, τότε τον φορτώνει, αυξάνει τον αριθμό των αναφορών (στην περίπτωση αυτή γίνεται 1) και επιστρέφει την αναφορά. Σε περίπτωση που ένα κομμάτι της μηχανής δεν χρειάζεται έναν πόρο, τότε ειδοποιεί τον πυρήνα και αυτός μειώνει την αναφορά του πόρου. Αν η αναφορά του πόρου φτάσει 0(ή λιγότερο), τότε καταστρέφει τον πόρο και αποδεσμεύει την μνήμη του. Ύστερα αν ο πόρος ξανά αναζητηθεί, τότε θα φορτωθεί εκ νέου.

Πίνακας 2.3.14: Το Interface ResourceContext

```
class ResourceContext: public Context<ResourceContext> {
public:

     virtual SoundInstance& getSound(const std::string& path)=0;
     virtual const Bitmap& getBitmap(const std::string& path)=0;
     virtual const AnimationClass& getAnimationClass(const std::string&
                                                          path)=0;
     virtual const AnimationPack& getAnimationPack(const std::string&
                                                          path)=0;
     virtual const File& getFile(const std::string& path)=0;
     virtual const SpriteSheet& getSpriteSheet(const std::string&
                                                    path)=0;
     virtual const AbilityClass& getAbilityClass(const std::string&
                                                    path)=0;
     virtual const LifeformClass& getLifeformClass(const std::string&
                                                    path)=0;

     virtual void releaseResource(const Bitmap& resource)=0;
     virtual void releaseResource(const Sound& resource)=0;
     virtual void releaseResource(const AnimationClass& resource)=0;
     virtual void releaseResource(const File& resource)=0;
```

```
      virtual void releaseResource(const SpriteSheet& resource)=0;
      virtual void releaseResource(const AnimationPack& resource)=0;
      virtual void releaseResource(const AbilityClass& resource)=0;
      virtual void releaseResource(const LifeformClass& resource)=0;

      virtual const Bitmap& getNullBitmap()=0;
      virtual const Sound& getNullSound()=0;

      ResourceContext() {

      }
      virtual ~ResourceContext() {

      }
};
```

Το Interface παρέχει για κάθε είδους πόρου, μια συνάρτηση παραλαβής αναφοράς (getX) και μια συνάρτηση αποδέσμευσης αναφοράς (releaseResource). Λόγω ότι τα Bitmaps και τα Sounds έχουν διαφορετικό τρόπο φόρτωσης από SDK σε SDK, υπάρχει η ανάγκη για ξεχωριστή φόρτωση των NullBitmap και NullSound.

Η υλοποίηση που χρησιμοποιεί η μηχανή την δεδομένη στιγμή είναι της κλάσης **SynchronousResourceContext**. Αυτό έχει ως λειτουργία να κάνει ότι είπαμε παραπάνω, συν ότι πρέπει να το κάνει σε περιβάλλον Thread-Safe. Έτσι ότι γίνεται, γίνεται σε κλειδωμένο κώδικα. Η κλάση φτιάχτηκε να μπορούν πολλά νήματα να φορτώνουν διαφορετικά **είδη** πόρων. Προσοχή! Αυτό δεν σημαίνει ότι μπορούν να φορτωθούν το ίδιο είδος πόρων ταυτόχρονα, παρά μόνο διαφορετικά είδη. Πχ δεν μπορούν να φορτώσουν 2 νήματα ταυτόχρονα 2 **AnimationClass.**

Πίνακας 2.3.15: Το Template SynchronousResourceContext

```
template<typename BitmapT, typename SoundT>
class SynchronousResourceContext: public ResourceContext {
public:

      SoundInstance& getSound(const std::string& path) {
            return getResource(path, sounds,
      soundMutex).createNewInstance();
      }

      const Bitmap& getBitmap(const std::string& path) {
            return getBitmapImpl(path);
      }

      const AnimationClass& getAnimationClass(const std::string& path) {
            return getResource(path, animationClasses,
      animationClassMutex);
      }
```

```
	const File& getFile(const std::string& path);
	const SpriteSheet& getSpriteSheet(const std::string& path);
	const AnimationPack& getAnimationPack(const std::string& path);
	const AbilityClass& getAbilityClass(const std::string& path);
	const LifeformClass& getLifeformClass(const std::string& path);

	void releaseResource(const Bitmap& value) {
		releaseMappedResource(value.getName(), bitmaps, bitmapMutex);
	}

	void releaseResource(const Sound& value);
	void releaseResource(const AnimationClass& value);
	void releaseResource(const File& value);
	void releaseResource(const SpriteSheet& value);
	void releaseResource(const AnimationPack& value);
	void releaseResource(const AbilityClass& value);
	void releaseResource(const LifeformClass& value);

	const Bitmap& getNullBitmap() {
		return NullReference<BitmapT>::getNull();
	}
	const Sound& getNullSound() {
		return NullReference<SoundT>::getNull();
	}

	SynchronousResourceContext();
	~SynchronousResourceContext();

private:
	std::mutex bitmapMutex;
	std::mutex soundMutex;
	std::mutex animationClassMutex;
	std::mutex fileMutex;
	std::mutex spriteSheetMutex;
	std::mutex animationPackMutex;
	std::mutex abilityClassMutex;
	std::mutex lifeformClassMutex;

	std::unordered_map<std::string, SharedResource<BitmapT>> bitmaps;
	std::unordered_map<std::string, SharedResource<SoundT>> sounds;
	std::unordered_map<std::string, SharedResource<AnimationClass>>
animationClasses;
	std::unordered_map<std::string, SharedResource<File>> files;
	std::unordered_map<std::string, SharedResource<SpriteSheet>>
spriteSheets;
	std::unordered_map<std::string, SharedResource<AnimationPack>>
animationPacks;
	std::unordered_map<std::string, SharedResource<AbilityClass>>
abilityClasses;
	std::unordered_map<std::string, SharedResource<LifeformClass>>
lifeformClasses;

	template<typename RType>
	const RType& getResource(const std::string& path,
		std::unordered_map<std::string, SharedResource<RType>>&
map,
		std::mutex& mutex);

	template<typename RType>
```

```
    void releaseMappedResource(const std::string& path,
                std::unordered_map<std::string, SharedResource<RType>>&
                                                                    map,
                std::mutex& mutex);

    const Bitmap& getBitmapImpl(const std::string& path);
};
```

Το Template έχει τεράστια δομή καθώς όλος ο λειτουργικός κώδικας του βρίσκεται στο Header αρχείο και αυτό γιατί είναι Template. Κάθε συνάρτηση **getX(...)** έχει αρμοδιότητα να επιστρέφει μια αναφορά σε αυτό που ορίζει. Ο κώδικας σε κάθε μια είναι ο ίδιος, με εξαιρέσεις την **getSound(...)**, **getBitmap(...)** όπως βλέπουμε στον πίνακα 2.3.16. Οι υπόλοιπες καλούν την συνάρτηση **getResource(...)** με ορίσματα την διαδρομή του αρχείου που ζητείται, το Hastable που ανήκει και το Mutex που θα χρησιμοποιηθεί για να κλειδώσει ο κώδικας, όπως στην συνάρτηση **getAnimationClass(...)**. Αντίστοιχα οι συναρτήσεις **releaseResource(...)**, καλούν την **releaseMappedResource(...)** με τα αντίστοιχα ορίσματα για να αποδεσμεύσουν έναν πόρο.

Η ανάγκη της ύπαρξης των **getNullBitmap()** και **getNullSound()** είναι γιατί οι υλοποιήσεις τους εξαρτιούνται από τα ορίσματα του Template που θα κατασκευαστεί το **SynchronousResourceContext**. Για αυτό και δεν μπορεί να ανακτηθεί μια αναφορά σε **NullBitmap** ή **NullSound** καθώς δεν υπάρχουν υλοποιήσεις των κλάσεων **Bitmap** και **Sound** μέσα στον πυρήνα της μηχανής.

Ο κατασκευαστής δεν κάνει τίποτα. Ο καταστροφέας καθαρίζει όλα τα Hashtables και καταστρέφει όλους τους πόρους.

Η πραγματική λειτουργία του Template βρίσκεται στις τελευταίες συναρτήσεις. Η συνάρτηση - Template **getResource(...)** ευθύνεται για να επιστραφεί ο πόρος που βρίσκεται στο **path** που θα πρέπει να ανήκει στο hashtable **map** και να κλειδωθεί ο κώδικας με το **mutex**. Ο πόρος θα φορτωθεί σαν είδος Resource **Rtype**.

Πίνακας 2.3.16: Η συνάρτηση SynchronousResourceContext::getResource(...)

```
template<typename RType>
    const RType& getResource(const std::string& path,
          std::unordered_map<std::string, SharedResource<RType>>& map,
                std::mutex& mutex) {
          // Lock the mutex so no other Thread will try to load with
          // the same signature function
          std::unique_lock<std::mutex> lock(mutex);
          // Find the filename in the referenced Hashtable
          auto itr = map.find(path);
```

```
            if (itr != map.end()) {
                  // If we found it, then check if the resource is valid
                  if (itr->second.isValid()) {
                        // If is valid, increase it's reference count
                        itr->second.increaseReference();
                        // Return a reference of the Resource
                        return itr->second.getInstance();
                  } else {
                  // If is not valid, return a Null resource reference
                        return NullReference<RType>::getNull();
                  }
            } else {
                  // If the Filename does not exist in the Hashtable
                  // Check it the file exists on the disk
                  if (FileLoader::fileExists(path.c_str())) {
                  // If the file exist, construct a SharedResource on the
                        // Hashtable with the path as key and
                        // get a reference to it
                        SharedResource<RType> &res = map[path];
                        // Load to this SharedResource the Filename
                        res.load(path);
                        // Check the Valid flag to check if it loaded fine
                        if (res.isValid()) {
                        // If it loaded OK, the increase its reference
                              res.increaseReference();
                              // Return a reference to the Resource
                              return res.getInstance();
                        } else {
                              // If the file did not load for any reason
                              // Return a Null Resource reference
                              return NullReference<RType>::getNull();
                        }
                  } else {
                        // If the file does not exist, then Log an error
                        Logger::getInstance().write(
                                    "During loading: " + path
                                          + " Reason: File does not
exist",
                                    Logger::MessageType::Error);
                  // Construct an Empty SharedResource on the Hashtable
                        // With key the path, and invalidate it to prevent
                        // further future futile tries to load it
                        map[path].setValid(false);
                        // Return a Null Resource reference
                        return NullReference<RType>::getNull();
                  }
            }
      }
```

Αρχικά κλειδώνουμε το Mutex που υπάρχει στο όρισμα της συνάρτησης για να μην συνεχίσει κάποιο άλλο νήμα που καλέσει την ίδια συνάρτηση. Ψάχνουμε το **path** μέσα στο **map** να δούμε αν υπάρχει. Εάν υπάρχει, τότε σημαίνει ότι μάλλον έχει ήδη φορτωθεί και δεν χρειάζεται να το ξανά φορτώσουμε, αρκεί να επιστρέψουμε μια αναφορά στον ήδη φορτωμένο πόρο. Έτσι ελέγχουμε αν είναι εντάξει για χρήση

μέσω της κλήσης **isValid()**. Εάν επιστρέψει **true**, τότε είναι εντάξει για χρήση και αυξάνουμε τον αριθμό των αναφορών του για να ξέρουμε ότι έχουμε δώσει ακόμη μια αναφορά. Τελικά επιστρέφουμε τον πόρο. Αν η **isValid()** επιστρέψει **false**, τότε σημαίνει ότι είχε γίνει προσπάθεια στο παρελθόν να φορτωθεί ο πόρος **path** σαν **RType** και είχε αποτυχία. Έτσι ξέρουμε ότι δεν έχει νόημα να ξαναδοκιμάσουμε και επιστρέφουμε ένα **NullReferemce<RType>** γιατί πρέπει να επιστρέψουμε κάτι που μπορεί να χρησιμοποιηθεί οπωσδήποτε για να αποφύγουμε τα Segmentation Faults. Ακόμα και αν αυτό που επιστρέψουμε δεν έχει νόημα για τον χρήστη, αυτός θα καταλάβει ότι κάτι πάει στραβά και θα κληθεί να διορθώσει το πρόβλημα ή να το αναφέρει.

Εάν το **path** δεν βρεθεί στο **map**, τότε σημαίνει ότι ο πόρος δεν είναι φορτωμένος, έτσι θα δοκιμάσουμε να τον φορτώσουμε. Αρχικά ελέγχουμε εάν το αρχείο υπάρχει όντως στον δίσκο μέσω της **FileLoader::fileExists(...)**. Εάν υπάρχει, τότε κατασκευάζουμε το **SharedResource<RType>** που θα φιλοξενήσει τον πόρο. Αυτό το κατασκευάζουμε με το να ζητήσουμε στο **map** να μας επιστρέψει μια αναφορά στο αντικείμενο με κλειδί **path**. Ακόμα και αν δεν υπάρχει αυτό το αντικείμενο, το κατασκευάζει με τον Default Constructor και επιστρέφει μια αναφορά που την βάζουμε στην μεταβλητή **res**. Αυτό τώρα είναι κενό και δεν μπορεί να χρησιμοποιηθεί πριν φορτώσουμε κάτι με την **load(...)**. Ζητάμε από το **res** να φορτώσει το **path**. Αν φορτώθηκε εντάξει, τότε η **isValid()** θα επιστρέψει **true** και θα μπορέσουμε να το επιστρέψουμε. Πριν το κάνουμε πρέπει να αυξήσουμε τον αριθμό των αναφορών του. Αν τελικά δεν φορτώθηκε σωστά, τότε επιστρέφουμε ένα **NullReference**.

Αν το αρχείο που αναζητούμε δεν υπάρχει στον δίσκο, τότε πρέπει να το αναφέρουμε στο Log File και να φροντίσουμε να μην ξανά κάνουμε την ίδια δουλειά για αυτό το **path**. Έτσι κατασκευάζουμε το **SharedResource** στο **map** με κλειδί το **path**. Όπως είπαμε, αυτό είναι κενό και δεν μπορεί να χρησιμοποιηθεί. Για αυτό θέτουμε το flag **valid** σε **false**. Έτσι όταν ξανά ζητηθεί αυτός ο πόρος, τότε θα βρεθεί στο **map** αλλά η **isValid()** θα επιστρέψει **false** και έτσι θα επιστραφεί το **NullReference**. Το ίδιο κάνουμε και τώρα.

Εδώ τελειώνει η ανάλυση της **getResource(...)** που χρησιμοποιείται για όλους του πόρους εκτός από τα **Bitmap**. Για αυτή την περίπτωση, χρησιμοποιείται μια τροποποιημένη έκδοση της συνάρτησης, την **getBitmapImpl(...)**.

Πίνακας 2.3.17: Η κλήση SynchronousResourceContext::getBitmapImpl()

```
const Bitmap& getBitmapImpl(const std::string& path) {
	std::unique_lock<std::mutex> lock(bitmapMutex);
	auto itr = bitmaps.find(path);
	if (itr != bitmaps.end()) {
		if (itr->second.isValid()) {
			itr->second.increaseReference();
			return itr->second.getInstance();
		} else {
			return NullReference<BitmapT>::getNull();
		}
	} else {
		// If it doesn't exist in the Hashtable
		// Check if this running thread is the Rendering Thread
		if (Display::isRenderingThread()) {
			if (FileLoader::fileExists(path.c_str())) {
				SharedResource<BitmapT> &res = bitmaps[path];
				res.load(path);
				if (res.isValid()) {
					res.increaseReference();
					return res.getInstance();
				} else {
					return NullReference<BitmapT>::getNull();
				}
			} else {
				Logger::getInstance().write(
						"During loading: " + path
							+ " Reason: File does not
exist",
							Logger::MessageType::Error);
				bitmaps[path].setValid(false);
				return NullReference<BitmapT>::getNull();
			}
		} else {
			// If this thread is not the Rendering Thread, then
		// we need to request from the rendering thread to load it
		// for us. So we unlock the mutex, so the Other thread
			// can load it (it will call the same function) and will
			// deadlock on the beginning where it locks the mutex)
			lock.unlock();
			// We construct a BitmapRequest with The filename
			BitmapLoadRequest request(path);
			// Add the request to the Rendering Context's Queue
			// to execute

System::getInstance().getRenderingContext().addOperationToQueue(request);
			// Wait for the Operation to be completed by the
			// Rendering Thread.
			request.waitToBeCompleted();
			// Since the Other Thread loaded it, it means that it
			// called this function too,  but there the
		// Display::isRenderingThread() returned true, and did all
		// the associated stuff to the hashtable, like increasing the
		// reference count, so we don't need to do it. So we just
			// return the Bitmap we got.
			return request.getBitmap();
		}
	}
```

```
}
```

Εδώ το μεγαλύτερο μέρος της συνάρτησης είναι ίδιο με την προηγούμενη, με λίγες αλλαγές. Οι αλλαγές βρίσκονται στο κομμάτι που εκτελείται όταν δεν έχει ήδη φορτωθεί το Bitmap και πρέπει να φορτωθεί.

Η συνάρτηση φτιάχτηκε για λειτουργεί ακόμα και αν εκτελεστεί σε ξεχωριστό νήμα. Επειδή όμως όπως λειτουργεί το OpenGL, δεν μπορούμε να φορτώσουμε τα Bitmaps στην VRAM αν δεν γίνει η φόρτωση από το νήμα που φτιάχτηκε το OpenGL. Για αυτό χρειάζεται να ελέγχουμε αν η συνάρτηση κλήθηκε από αυτό το νήμα. Αυτό γίνεται από την static συνάρτηση της κλάσης **Display** την **isRenderingThread()** που επιστρέφει **true** μόνο αν καλεστεί από το νήμα που κατασκευάστηκε το **Display**. Αν όντως είναι το νήμα που έχει το OpenGL Context, τότε μπορούμε να συνεχίσουμε την φόρτωση σαν έναν απλό πόρο.

Εάν η εκτέλεση δεν βρίσκεται σε αυτό νήμα, τότε χρειάζεται να ζητήσουμε από το Rendering να φορτώσει για αυτό το δικό μας το Bitmap. Αυτό γίνεται από το **ContextOperation** που υπάρχει εσωτερικά, το **BitmapLoadRequest**.

Πίνακας 2.3.18: Η κλάση BitmapLoadRequest

```
class BitmapLoadRequest: public ContextOperation<RenderingContext> {
public:
     const std::string& getPath() const;
     const Bitmap& getBitmap() const;

     void handle(RenderingContext& context) {
           bitmap =
     &System::getInstance().getResourcesManager().getBitmap(
                              path);
     }

     BitmapLoadRequest(const std::string& path) :
                 ContextOperation(), path(path), bitmap(nullptr) {
     }

     ~BitmapLoadRequest() {
     }
private:
     std::string path;
     const Bitmap *bitmap;
};
```

Όπως βλέπουμε, στην συνάρτηση **handle()** που θα εκτελεστεί από το άλλο νήμα, έχει μια απλή κλήση για ανάκτηση του Bitmap με το ίδιο path. Στην ουσία, το άλλο νήμα θα καλέσει την ίδια συνάρτηση που βρισκόμαστε τώρα (getBitmapImpl()) αλλά

σε αυτή την περίπτωση η κλήση **isRenderingThread()** θα επιστρέψει **true** γιατί εκτελείται από το νήμα που φτιάχτηκε το Display. Έτσι θα φορτωθεί σαν κανονικός πόρος και στο πεδίο **bitmap** του **BitmapLoadRequest** θα μπει η φορτωμένη εικόνα που θα μπορεί να την ανακτήσει το νήμα που του ζητήθηκε μέσω της **getBitmap()**. Για να γίνει όλη αυτή η διαδικασία, πρέπει να κατασκευάσουμε ένα αντικείμενο **BitmapLoadRequest** με διαδρομή αρχείου το **path** και να το προσθέσουμε στην ουρά των λειτουργιών που θα εκτελέσει το Rendering νήμα. Ύστερα πρέπει να περιμένουμε μέχρι να το εκτελέσει το Rendering νήμα και στο τέλος να πάρουμε την εικόνα.

Υπάρχει όμως ένα θέμα. Όταν καλεστεί η **getBitmapImpl()** από το Rendering νήμα, το πρώτο πράγμα που θα κάνει είναι να κλειδώσει το **bitmapMutex**. Επειδή όμως αυτό έχει ήδη κλειδωθεί από το άλλο νήμα, τότε το Rendering θα περιμένει το δικό μας να το ξεκλειδώσει για να συνεχίσει. Όμως εμείς περιμένουμε το Rendering να τελειώσει. Έτσι το ένα νήμα περιμένει το άλλο και θα έχουμε ένα Deadlock. Για αυτό πριν αρχίσουμε να περιμένουμε, πρέπει να ξεκλειδώσουμε το mutex στο νήμα μας, για να αφήσουμε το Rendering να συνεχίσει. Έτσι πριν κάνουμε οτιδήποτε, το ξεκλειδώνουμε μέσω την **lock.unlock()**, ύστερα συνεχίζουμε με την κατασκευή του request, την προσθήκη του στο Rendering νήμα και την αναμονή για ολοκλήρωση.

Βλέπουμε ότι στο τέλος απλά επιστρέφουμε το Bitmap που φόρτωσε το Rendering νήμα, χωρίς να κάνουμε ότι κάνανε στην περίπτωση που ήμασταν το Rendering νήμα. Αυτό γιατί όντως τα έκανε το Rendering νήμα που πήρε άλλο δρόμο όταν καλέστηκε ή **isRenderingThread()**. Για αυτό ξέρουμε ότι το Bitmap που έχουμε έχει φορτωθεί σωστά και έχει μπει στο **bitmapMap** με σωστό αριθμό αναφορών.

Η συνάρτηση **releaseMappedResource(...)** έχει σκοπό να αποδεσμεύσει έναν πόρο από ένα **map**. Η αποδέσμευση δεν σημαίνει ότι και θα καταστραφεί ο πόρος, απλά σημαίνει ότι κάποιο κομμάτι της μηχανής που το ζήτησε, δεν το χρειάζεται πια και πρέπει ο **ResourcesContext** να γίνει γνώστης αυτού.

Πίνακας 2.3.19: Η συνάρτηση SynchronousResourceContext::releaseMappedResource(...)

```
template<typename RType>
void releaseMappedResource(const std::string& path,
           std::unordered_map<std::string, SharedResource<RType>>& map,
           std::mutex& mutex) {
    // Lock the mutex so no other Thread will try to load with
    // the same signature function
```

```
        std::unique_lock<std::mutex> lock(mutex);
        // Find the filename in the referenced Hashtable
        auto itr = map.find(path);
        if (itr != map.end()) {
              // If it exists in the HashTable, check if it is valid
              if (itr->second.isValid()) {
                    // If it is valid, decrease its reference Count
                    itr->second.decreaseReference();
                    // If the reference count reached 0 or below
                    if (itr->second.getReferenceCount() <= 0) {
                          // Log a Message of the Resource deallocation
                          Logger::getInstance().write("Releasing Resource: "
                                                                  + path);
              // Destroy the resource and erase its record on the hashtable
                          map.erase(itr);
                    }
              }
        }
  }
```

Αρχικά κλειδώνουμε τον Mutex που μας παρέχεται. Όταν γίνει αυτό, αναζητούμε το **path** στο **map.** Εφόσον υπάρχει, τότε ελέγχουμε αν είναι έγκυρο μεσώ της **isValid()**. Αυτό μας επιβεβαιώνει ότι όντως είναι πόρος που φορτώθηκε στην μνήμη και όχι κάποιος που υπάρχει για να αποφεύγουμε την φόρτωση από αυτό το **path**. Τότε μειώνουμε τον αριθμό των αναφορών για αυτόν το πόρο. Εάν οι αναφορές προς αυτόν τον πόρο έχουν φτάσει στο 0, τότε σημαίνει ότι κανένα κομμάτι της μηχανής δεν το χρειάζεται πλέον και έτσι μπορούμε να τον καταστρέψουμε. Αν τελικά τον καταστρέψουμε, τότε το αναφέρουμε στο Log File και ύστερα αφαιρούμε την εγγραφή του από το **map**. Αφαίρεση αυτή, θα προκαλέσει την καταστροφή του **SharedResrouce** που υπάρχει εκεί και όταν καλεστεί ο καταστροφέας, θα καταστραφεί και ο πραγματικός πόρος.

Εδώ τελειώνει η ανάλυση του πυρήνα του **ResourcesContext**. Είδαμε πως λειτουργεί το όλο σύστημα και πως μοιράζονται οι πόροι στα υποσυστήματα που τα ζητούν. Όμως υπάρχει και κάτι που δεν έχουμε αναλύσει και το έχουμε δει πολλές φορές. Αυτό είναι τα **NullReference**.

NullReference

Στην φιλοσοφία της μηχανής είναι να λειτουργήσει ακόμα και στα πιο κρίσιμα σφάλματα. Συνήθως όταν σε ένα πρόγραμμα γίνει ένα πολύ σοβαρό σφάλμα, τότε η εκτέλεση του τερματίζει ακαριαία και εμφανίζει ένα μήνυμα λάθους. Εδώ τα πράγματα είναι λίγο διαφορετικά. Σφάλματα που έχουν να κάνουν με Segmentation Faults, προκαλούν το πρόγραμμα να τερματίζει εκείνη την στιγμή που συνέβη, χωρίς να δο-

θεί μια ευκαιρία να αναφερθεί το σφάλμα. Συνήθως αυτά τα σφάλματα προέρχονται από λάθος αναφορές. Πέρα από αυτό, συχνό φαινόμενο είναι να προσπαθήσουμε να γράψουμε/διαβάσουμε σε Null διεύθυνση (0x0). Τέτοιες περιπτώσεις θα είχαμε συνέχεια εάν μέσα στην μηχανή ζητάγαμε αρχεία που δεν υπάρχουν. Σε τέτοια περίπτωση, η μηχανή θα πήγαινε να χρησιμοποιήσει διεύθυνση Null, με αποτέλεσμα να τερμάτιζε χωρίς ειδοποιήσει και να γράψει κάποιο σφάλμα. Έτσι ο εντοπισμός του προβλήματος (το αρχείο που δεν υπάρχει), θα ήταν πολύ δύσκολη δουλειά. Για αυτό και επινοήθηκε το **NullReference**. Πρόκειται για απλούς πόρους, καθολικά προσβάσιμους για χρήση αντί των πόρων που φορτώνονται. Για παράδειγμα εάν κάποιος πόρος είχε πρόβλημα κατά την φόρτωση του και δεν μπορεί να χρησιμοποιηθεί, είναι φυσικό ότι δεν μπορεί να δοθεί σε αυτόν που τον ζήτησε, γιατί το κομμάτι που τον ζήτησε θεωρεί δεδομένο ότι ο **ResourceContext** θα του τον επιστρέψει και θα λειτουργεί κανονικά. Όμως δεν μπορούμε να του επιστρέψουμε αυτό που ζήτησε, για αυτό και του επιστρέφουμε έναν απλό πόρο **NullReference** που τηρεί όλες τις προδιαγραφές για αυτόν τον τύπο πόρου που του ζήτησε. Το κομμάτι που τον ζήτησε δεν καταλαβαίνει ότι δεν είναι αυτό που ήθελε γιατί το θεωρεί δεδομένο και έτσι συνεχίζει να χρησιμοποιεί κανονικά το **NullReference** σαν να ήταν αυτό που ζήτησε. Επειδή όμως το **NullReference** περιέχει σκουπίδια, όταν θα το δει ο χρήστης θα καταλάβει ότι κάτι πάει στραβά και θα αρχίσει να ψάχνει στο Log File για μηνύματα λάθους που έγγραψε ο **ResourcesContext** όταν υπήρξε σφάλμα κατά την φόρτωση του πραγματικού πόρου. Έτσι το Debugging γίνεται πολύ πιο εύκολο.

Πίνακας 2.3.20: Το Template NullReference

```
template<typename type>
class NullReference {
public:

	static type& getNull() {
		static type nullInstance;
		return nullInstance;
	}

	NullReference() {
	}

	virtual ~NullReference() {
	}
};
```

Ο τρόπος λειτουργίας του είναι πολύ απλός. Προϋποθέτει ότι ο πόρος μπορεί να κατασκευαστεί εντός της μηχανής χωρίς να φορτωθεί αρχείο. Για αυτό απαιτείται σε κάθε κλάση που κληρονομεί την **Resource,** να έχει έναν Default Contructor που να κατασκευάζει ένα απλό χρησιμοποιήσιμο πόρο. Έτσι κατά την κλήση της **getNull()** κατασκευάζεται ένα static αντικείμενο από την κλάση που ορίστηκε σαν όρισμα στο Template. Επειδή είναι static θα επιμένει καθ' όλη την εκτέλεση του προγράμματος και θα κατασκευαστεί μόνο μια φορά. Έτσι όταν καλείται αυτή η συνάρτηση, απλά θα επιστρέφεται αυτό το απλό αντικείμενο.

Ο λόγος που ορίστηκε μέσα στην συνάρτηση αντί του σώματος του Template, είναι ότι υπάρχει μια ιδιαιτερότητα με αυτό τον τρόπο που ορίστηκε στο πρότυπο της C++11. Ο κανόνας λέει ότι οι static μεταβλητές εντός συνάρτησης πρέπει να αρχικοποιούνται μόνο από ένα νήμα. Έτσι ξέρουμε ότι η αρχικοποίηση θα είναι thread-safe χωρίς περαιτέρω κώδικα, καθώς αυτό το αναλαμβάνει ο compiler.

Εδώ τελειώνει η ανάλυση του **ResourceContext**. Υπάρχουν και άλλες κλάσεις που συνδέονται με αυτό αλλά ταιριάζουν πιο πολύ σε παρακάτω κεφάλαια.

2.4 Η Lua Engine

Μεγάλο μέρος της μηχανής δέχεται εντολές μέσω scripts της Lua. Για αυτό και πρέπει να υπάρχει ένα εσωτερικό σύστημα διαχείρισης του Lua Environment (ή LuaE για συντομία). Το περιβάλλον Lua είναι μοναδικό τόσο στην μηχανή τόσο και στο CEGUI. Δηλαδή υπάρχει μόνο ένα που το μοιράζονται όλα το κομμάτια για να μπορούν να έχουν πρόσβαση το ένα στο άλλο και μέσω της Lua.

Πέρα από τον διαμοιρασμό του περιβάλλοντος Lua, η κλάση **LuaEngine** είναι υπεύθυνη για την σαφή αναπαράσταση των δεδομένων Lua εντός του περιβάλλοντος C++ της μηχανής. Κάθε μεταβλητή της Lua μπορεί να υπάρχει εντός της μηχανής σαν αναφορά μέσω ενός αριθμού. Σε αυτό βοηθά η βιβλιοθήκη της Lua που παρέχει αυτόν τον μηχανισμό. Η **LuaEngine** εκμεταλλεύεται αυτόν τον μηχανισμό για να παρέχει εύκολη πρόσβαση στο περιβάλλον της Lua οπουδήποτε στην μηχανή.

Καταχώριση τύπων και η Abstract κλάση LuaPushable

Πολλοί τύποι αντικειμένων της μηχανής μπορούν να περαστούν στο περιβάλλον Lua. Για να μπορέσει μια κλάση να το κάνει αυτό, πρέπει έχει καταχωριστεί σαν τύπος στο περιβάλλον Lua και να κληρονομήσει την κλάση **LuaPushable**. Η καταχώριση γίνεται μέσω μιας δευτερεύουσας βιβλιοθήκης, της tolua++.

Η tolua++ έχει ένα εργαλείο που δέχεται αρχεία που περιγράφουν τις κλάσεις που θέλουμε να περάσουμε στο περιβάλλον Lua και παράγει κώδικα C που το κάνει αυτό μέσω κλήσεων στην βιβλιοθήκη tolua++. Το εργαλείο αυτό έχει τροποποιηθεί για να δέχεται και κώδικα C++11 και η τροποποιημένη μορφή του βρίσκεται σε δυαδική μορφή εντός του κώδικα.

Εάν θέλουμε να καταχωρίσουμε μια νέα κλάση στο περιβάλλον Lua, τότε πρέπει να κατασκευάζομε το αρχείο .pkg όπως ορίζει το documentation της tolua++ και να το βάλουμε στον φάκελο libzeta/Zeta/Core/toLua_Binders. Έπειτα πρέπει να γράψουμε μια νέα γραμμή στο αρχείο Zeta.pkg με το όνομα του νέου αρχείου. Στο τέλος εκτελούμε το script ./rebuild.sh το οποίο θα παράγει τον κώδικα και το documentation για το eclipse.

Αυτό θα κάνει την κλάση να είναι λειτουργική εντός της Lua. Εάν θέλουμε να μπορούμε να βάζουμε αντικείμενα της μέσω της C++ στην Lua, θα πρέπει η κλάση μας να κληρονομήσει επιπλέον την κλάση **LuaPushable** και να υλοποιήσει την συνάρτηση **pushToLua(...)**.

Πίνακας 2.4.1: Η κλάση LuaPushable

```
class LuaPushable {
public:

     virtual void pushToLua(lua_State *lstate)=0;

     constexpr LuaPushable() {

     }
     virtual ~LuaPushable() {

     }
};
```

Το σώμα της συνάρτησης είναι πάντα το ίδιο για κάθε κλάση:

Πίνακας 2.4.2: Παράδειγμα της συνάρτησης ::pushToLua(lua_State)

```
void System::pushToLua(lua_State* lstate) {
     tolua_pushusertype(lstate, this, "Zeta::System");
}
```

Η ανάγκη για να υπάρχει σε κάθε κλάση που το υλοποιεί είναι το τελευταίο όρισμα της **tolua_pushusertype(...)**. Αυτή η συμβολοσειρά πρέπει να είναι το όνομα όπως

καταχωρίστηκε στην Lua μέσω της tolua++. Συνήθως είναι το πλήρες όνομα της C++ όπως παραπάνω. Το πρώτο όρισμα είναι πάντα το lua_State που μας παρέχεται, ενώ το δεύτερο είναι το αντικείμενο που θα βάλουμε στο περιβάλλον της Lua.

Το κέλυφος των τύπων Lua

Όλοι οι τύποι της Lua έχουν μια κλάση Wrapper εντός της μηχανής για να μπορούν οι προγραμματιστές εύκολα να παίζουν με αυτά χωρίς να χρειαστεί να μάθουν βαθύτερη γνώση της βιβλιοθήκης Lua. Οι απλοί τύποι όπως string και number έχουν απλές κλάσεις. Όλες αυτές οι κλάσεις κληρονομούν την **LuaValue** που κληρονομεί την **LuaPushable**.

Πίνακας 2.4.3: Η κλάση LuaValue

```
class LuaValue: public LuaPushable {
public:

     enum class Type {
           Number, String, Boolean, Table, Function, UserData, Nil, None,
                                Other
     };

     virtual bool setFromStack(lua_State* lstate, int stackIndex)=0;

     constexpr LuaValue() {

     }

     virtual ~LuaValue() {

     }
};
```

Αυτή η κλάση-interface είναι βάση για όλους τους τύπους της Lua και βοηθάει σε πιο πολύπλοκους τύπους όπως θα δούμε παρακάτω. Ορίζει την συνάρτηση **setfromStack()** που οι κλάσεις που θα την υλοποιήσουν θα πρέπει να αναθέτουν από την τιμή στην στοίβα της Lua που βρίσκεται στον δείκτη **stackIndex**. Το όρισμα **lstate** έχει να κάνει με το από ποιο Lua State θα παρθεί η τιμή. Η κλήση έχει να κάνει με το όλο σύστημα της Lua και συνήθως κάποιος που τροποποιήσει την μηχανή δεν χρειάζεται να ασχοληθεί με αυτή. Η συνάρτηση πρέπει να επιστρέφει true αν η ανάκτηση της τιμής ήταν επιτυχής. Συνήθως η επιτυχημένη ανάκτηση είναι όταν ο τύπος της τιμής που δείχνει η στοίβα είναι ίδιος με αυτό που υλοποιεί η κλάση. Εάν δεν είναι, τότε δεν πρέπει να γίνεται απόδοση της τιμής και να επιστρέφεται false.

Πέρα από την συνάρτηση αυτή, ορίζεται και το Enumeration **Type**. Αυτό απαριθμεί όλους τους τύπους Lua (εκτός από Thread και Light User Data). Ο σκοπός αυτού είναι για να προσδιορίζεται ο τύπος του LuaValue σε περίπτωση που δοθεί μέσω αυτής της κλάσης και όχι μέσω κλάσης που το κληρονομεί, όπως θα δούμε παρακάτω.

Όπως είπαμε, οι τύποι της Lua, οι περισσότεροι έχουν μια κλάση Wrapper που κληρονομούν την LuaValue και λειτουργούν γύρω από αυτό. Ο κύριος σκοπός τους είναι να ορίζουν μια τιμή Lua εντός του C++ περιβάλλοντος και να μπορεί να μεταφερθεί εύκολα στο περιβάλλον Lua. Μια τέτοια κλάση είναι η παρακάτω.

Πίνακας 2.4.4: Η κλάση LuaNumber

```
class LuaNumber: public LuaValue {
public:

     void pushToLua(lua_State* lstate);
     bool setFromStack(lua_State* lstate, int stackIndex);

     void setValue(Float value) {
           this->value = value;
     }

     Float getValue() const {
           return value;
     }

     constexpr LuaNumber() :
                 value(0.0f) {

     }

     constexpr LuaNumber(Float value) :
                 value(value) {

     }

     ~LuaNumber() {
     }
private:
     Float value;
};
```

Τέτοιοι απλοί τύποι, απλά φιλοξενούν την τιμή και την βάζουν στην Lua μεσώ της **pushToLua(...)**. Αυτό ισχύει για τους τύπους, String, Number και Boolean. Για πιο σύνθετους τύπους Lua, τα πράγματα γίνονται πιο πολύπλοκα.

Στους τύπους Table και Function, εκεί χρειάζονται κλάσεις που είναι πιο λειτουργικοί. Για αυτό και χρησιμοποιούν αναφορές προς αυτές τις μεταβλητές για να τις τροποποιήσουν. Επειδή όμως οι αναφορές δεν μπορούν να χρησιμοποιούνται αυθαίρε-

τα χωρίς επίβλεψη, χρειάζεται ένα σύστημα βιβλίου-καταχωρίσεων, όπως στην και με τους πόρους στο **ResourceContext**. Εδώ ο πόρος είναι η αναφορά σε μεταβλητή της Lua.

Όπως είπαμε, οι αναφορές σε Lua είναι αριθμοί που μας δίνει το περιβάλλον Lua για να μπορούμε εύκολα να έχουμε πρόσβαση σε μεταβλητές. Με αυτούς τους αριθμούς μπορούμε εύκολα να ανακτήσουμε την μεταβλητή χωρίς να ψάχνουμε μέσω του ονόματός της.

Για λόγους σαφήνειας και για έχουμε μια σειρά όπως είπαμε στην προ-προηγούμενη παράγραφο, οι αναφορές δίνονται αυστηρά μόνο της κλάσης **LuaEngine**. Αυτό γίνεται σε συνεργασία με την κλάση **LuaReferenced**. Αυτή την κληρονομούν όλες οι κλάσεις που χρειάζονται αναφορές σε μεταβλητές Lua και τα υπόλοιπα τα αναλαμβάνει αυτή.

Πίνακας 2.4.5: Η κλάση LuaReferenced

```
class LuaReferenced: public LuaValue {
public:

     const std::string& getName() const;
     void setName(const std::string& name);
     int getLuaReference() const;

     virtual bool setFromStack(lua_State* lstate, int stackIndex);
     virtual void setLuaReference(int reference);

     bool isValid() const;
     void pushToLua(lua_State* lstate);

     LuaReferenced();
     LuaReferenced(const std::string& luaName);
     LuaReferenced(int stackIndex);
     LuaReferenced(LuaReferenced&& other);
     LuaReferenced(const LuaReferenced& other);
     LuaReferenced& operator=(const LuaReferenced& other);
     LuaReferenced& operator=(LuaReferenced&& other);
     virtual ~LuaReferenced();
protected:
     std::string name;
     int luaReference;
};
```

Ο βασικός σκοπός της είναι να εξασφαλίζει ότι κατά την αντιγραφή ή μετακίνηση των αντικειμένων των κλάσεων που την κληρονομούν, ότι δεν θα υπάρχουν αντιγραφές των αναφορών Lua η απαράδεκτες διαγραφές αναφορών, λόγω ότι κάποιο αντίγραφο καταστράφηκε.

Οι τρεις πρώτες συναρτήσεις είναι αυτό-εξηγούμενες. Η συνάρτηση **setFromStack(...)** υλοποιεί την ανάκτηση της αναφοράς από την στοίβα της Lua στον δείκτη στοίβας **stackIndex**. Η ανάκτηση γίνεται μέσω της **LuaEngine**. Αν υπάρχει ήδη μια αναφορά στο μέλος luaReference, τότε μειώνεται ο αριθμός των αναφορών για αυτό το Lua Reference. Αντίστοιχα η **setLuaReference(...)** αντιγράφει το **reference** και να αυξάνει τον αριθμό αναφορών για αυτή την Lua αναφορά.

Πίνακας 2.4.6: Οι συναρτήσεις LuaReferenced::set(...) και ::setLuaReference(...)

```
bool LuaReferenced::setFromStack(lua_State* lstate, int stackIndex) {
	// Check if the member has already a reference
	if (luaReference != LUA_NOREF) {
		// If it does, release it
		LuaEngine::getInstance().decreaseReference(luaReference);
	}
	// get the reference
	luaReference = LuaEngine::getInstance().getReference(stackIndex);
	// Check if we got the reference (We might got a stackIndex pointing
	// to nothing)
	if (luaReference != LUA_NOREF) {
		return true;
	} else {
		return false;
	}
}

void LuaReferenced::setLuaReference(int reference) {
	// Check if the member has already a reference
	if (luaReference != LUA_NOREF) {
		// If it does, release it
		LuaEngine::getInstance().decreaseReference(luaReference);
	}
	// Increase this reference count
	LuaEngine::getInstance().increaseReference(reference);
	// copy the reference
	luaReference = reference;
}
```

Η συνάρτηση **isValid(...)** απλά ελέγχει αν το μέλος **luaReference** έχει κάποιον αριθμό. Αυτό που αξίζει να δούμε περαιτέρω είναι τον κατασκευαστή αντιγραφής (Copy-Constructor) και κατασκευαστή μετακίνησης (Move Constructor).

Πίνακας 2.4.7: Οι συναρτήσεις LuaReferenced(const&) και LuaReferenced(&&)

```
LuaReferenced::LuaReferenced(LuaReferenced&& other) :
	name(std::move(other.name)), luaReference(other.luaReference) {
	// When moving the object, invalidate the other reference
	// to prevent releasing it
	other.luaReference = LUA_NOREF;
}
```

```
LuaReferenced::LuaReferenced(const LuaReferenced& other) {
    // When copying, copy everything except the luaReference
    name = other.name;
    luaReference = other.luaReference;
    // Increase the Reference Count for this reference
    LuaEngine::getInstance().increaseReference(luaReference);
}
```

Στην πρώτη περίπτωση (Move Constructor), μετακινούμε το όνομα στο νέο αντικείμενο και αντιγράφουμε τον αριθμό της αναφοράς. Αυτό που χρειάζεται να κάνουμε είναι να αφαιρέσουμε την αναφορά τελείως από το άλλο αντικείμενο για αποφύγουμε τον καταστροφέα της κλάσης να μειώσει τον αριθμό των αναφορών της, καθώς δεν αντιγράφουμε το αντικείμενο αλλά το μετακινούμε σε άλλη μνήμη.

Στην δεύτερη περίπτωση (Copy Constructor), αντιγράφουμε όλα τα μέλη και στο τέλος φροντίζουμε να αυξήσουμε τον αριθμό των αναφορών καθώς πλέον υπάρχουν δύο αντικείμενα που έχουν την ίδια αναφορά.

Οι δυο παραπάνω παράγραφοι ισχύουν και για τους αντίστοιχους τελεστές =.

Ας δούμε όμως τους τύπους που χρησιμοποιούν αυτή την κλάση. Οι τύποι αυτοί είναι οι **LuaTable** και **LuaFunctor**.

Η κλάση LuaTable, ίσως είναι η πιο πολύπλοκη και λειτουργική κλάση του συστήματος Lua. Ο σκοπός της είναι να παρέχει όσο το δυνατόν πλήρη λειτουργικότητα, παρέχοντας λειτουργίες δημιουργίας, πρόσβασης και παραποίησης πινάκων Lua μέσω του εύκολου αυτού Wrapper.

Πίνακας 2.4.8: Η κλάση LuaTable

```
class LuaTable: public LuaReferenced {
public:

    template<typename ValueT>
    bool getField(const LuaValue& key, ValueT& value) const {
          if (isValid()) {
                saveStack();
                pushTable();
                const_cast<LuaValue&>(key).pushToLua(lstate);
                getTableFieldFromStack();
                bool ret = setValueFromStack(value);
                resetStack();
                return ret;
          } else {
                return false;
          }
    }
```

```
	int getFieldReference(const std::string& name) const;
	int getFieldReference(int index) const;

	void setField(const LuaValue& key, const LuaValue& value) {
		if (isValid()) {
			saveStack();
			pushTable();
			const_cast<LuaValue&>(key).pushToLua(lstate);
			const_cast<LuaValue&>(value).pushToLua(lstate);
			setTableFieldFromStack();
			resetStack();
		}
	}

	void niliseField(const LuaValue& key) {
		setField(key, LuaNil());
	}

	void forEach(
		const std::function<
		void(const LuaString&, const LuaValue&, LuaValue::Type)>&
									func);

	void forEachI(
		const std::function<void(const LuaValue&, LuaValue::Type)>&
									func);

	bool setFromStack(lua_State* lstate, int stackIndex);
	void setLuaReference(int reference);

	void printContents() const;

	void create(int reserve = 0);

	LuaTable();
	LuaTable(int stackIndex);
	~LuaTable();
private:
	lua_State* lstate;
	mutable int stackPointer;

	void saveStack() const;
	void resetStack() const;

	void pushTable() const;

	void setTableFieldFromStack();
	void getTableFieldFromStack() const;

	bool setValueFromStack(std::string& value) const;
	bool setValueFromStack(Float& value) const;
	bool setValueFromStack(int& value) const;
	bool setValueFromStack(bool& value) const;
	bool setValueFromStack(LuaTable& value) const;
	bool setValueFromStack(LuaFunctor& value) const;
};
```

Η συνάρτηση **getField(...)** έχει σκοπό να αναθέσει στην μεταβλητή **value** την τιμή που βρίσκεται στο πεδίο του πίνακα Lua με όνομα **key**. Το όνομα του πεδίου στους πίνακες lua μπορεί να είναι οποιαδήποτε τιμή Lua. Έτσι το θέτουμε σε γενικό τύπο **LuaValue**. Αρχικά ελέγχουμε μέσω της **isValid()** εάν έχουμε αναφορά σε κάποιον πίνακα. Εφόσον έχουμε, αποθηκεύουμε τον δείκτη της στοίβας Lua για να τον επαναφέρουμε όταν τελειώσουμε. Αυτό γίνεται μέσω της **saveStack()** και το κάνουμε για να σιγουρέψουμε ότι η στοίβα Lua θα είναι όπως την παραλάβαμε όταν θα βγούμε από την συνάρτηση. Στην επόμενη φάση, βάζουμε τον πίνακα Lua που έχουμε αναφορά στην αρχή της στοίβας. Από πάνω του βάζουμε το κλειδί που είναι το πεδίο του πίνακα. Καλούμε την **getTableFieldFromStack()** η οποία βάζει στην αρχή στης στοίβας την τιμή που υπάρχει στο κλειδί που θέσαμε. Μετά καλείται η **setValueFromStack()**. Αυτή η συνάρτηση όπως βλέπουμε είναι Overloaded, που σημαίνει ότι υπάρχει πολλές φορές με το ίδιο όνομα και διαφορετικό signature. Αυτό γιατί το template όρισμα **value**, μπορεί να πάρει πολλούς διαφορετικούς τύπους, που ανάλογα τον τύπο πρέπει να γίνουν διαφορετικοί έλεγχοι. Για παράδειγμα, εάν θέλουμε να βάλουμε την τιμή σε μια μεταβλητή float, θα πρέπει να δούμε αν το πεδίο αυτό στον πίνακα είναι όντως float. Αυτό γιατί η C++ είναι static typing και έρχεται σε σύγκρουση με το dynamic typing της Lua, που μπορεί να ελέγχεται ο τύπος της μεταβλητής κατά την εκτέλεση, καθώς και να αλλάζει. Για παράδειγμα, στην C++ ορίζονται πίνακες ο οποίοι θα έχουν αυστηρά μόνο έναν τύπο δεδομένων, ενώ στην Lua οι πίνακες μπορούν να έχουν διάφορους. Για αυτό και υπάρχει η ανάγκη για έλεγχο πριν την ανάθεση, καθώς η μεταβλητή που μας ήρθε με αναφορά για να βάλουμε την τιμή, να μην ταιριάζει με αυτό που βρίσκεται στον πίνακα, πχ να πρέπει να θέσουμε έναν ακέραιο Int και στον πίνακα να έχουμε string τιμή στο κλειδί που μας δόθηκε.

Πίνακας 2.4.9: Παραδείγματα της συνάρτησης setValueFromStack()

```
bool LuaTable::setValueFromStack(bool& value) const {
    // If the type of the item on the top of the stack is correct,
    // then assign and return true
    if (lua_isboolean(lstate, -1)) {
        value = lua_toboolean(lstate, -1);
        return true;
    } else {
        return false;
    }
}

bool LuaTable::setValueFromStack(LuaTable& value) const {
    // If the type of the item on the top of the stack is correct,
    // then assign and return true
```

```
        if (lua_istable(lstate, -1)) {
                value.setFromStack(lstate, -1);
                return true;
        } else {
                return false;
        }
}
```

Στα παραπάνω δυο παραδείγματα, βλέπουμε τις εξειδικευμένες περιπτώσεις της συνάρτησης. Όταν πχ πρέπει βάλουμε τιμή σε τύπου **bool** ελέγχουμε αν είναι η τιμή στην στοίβα είναι boolean. Αν είναι όντως, την παίρνουμε και επιστρέφουμε true. Αν δεν είναι, τότε επιστρέφουμε false για να δείξουμε ότι δεν έγινε απόδοση τιμής στην μεταβλητή. Το ίδιο γίνεται και στην περίπτωση του **LuaTable**, όπου αναθέτουμε την τιμή μέσω της ειδικής συνάρτησης.

Επιστρέφοντας στην **getField(...)**, αφού η **setValueFromStack(...)** επιστρέψει, αποθηκεύουμε το αποτέλεσμα της προσωρινά, για να το επιστρέψουμε σε αυτόν που κάλεσε την **getField(...)**. Ο λόγος που δεν επιστρέφουμε κατευθείαν, είναι γιατί χρειάζεται πριν επιστρέψουμε, να επαναφέρουμε την στοίβα στην κατάσταση που ήταν όταν μπήκαμε στην συνάρτηση. Αυτό γίνεται με την **resetStack()** που επαναφέρει την στοίβα να δείχνει στον **stackPointer** που αποθηκεύσαμε στην αρχή. Τελικά, επιστρέφουμε το αποτέλεσμα της **setValueFromStack(...)**.

Οι συναρτήσεις **getFieldReference(...)** επιστρέφουν μια αναφορά Lua σε κάποιο πεδίο του πίνακα. Ανάλογα το όρισμα της, το βάζει σαν κλειδί και αν υπάρχει κάτι, τότε ζητάει μια αναφορά σε αυτό από την **LuaEngine** και την επιστρέφει.

Η **setField(...)** έχει σκοπό να θέσει την τιμή **value** σε ένα πεδίο του πίνακα με όνομα **key**. Το όνομα αυτό μπορεί να είναι οτιδήποτε τιμή Lua. Η λειτουργία της είναι αντίστοιχη με της **getField(...)**. “Σπρώχνονται” αρχικά το κλειδί και ύστερα η τιμή στην στοίβα. Μετά καλείται η **setFieldFromStack()** όπου θέτει table[**key**]=**value** με τις αντίστοιχες τιμές στην στοίβα. Αντίστοιχα η **niliseField()** θέτει table[**key**]=nil δηλαδή διαγράφει το πεδίο του πίνακα.

Από τις πιο χρήσιμες συναρτήσεις, η **forEach(...)**, καλεί για όλα τα στοιχεία του πίνακα την συνάρτηση lambda που παρέχεται σαν όρισμα. Η συνάρτηση αυτή θα πρέπει να έχει ορίσματα το κλειδί, την τιμή και τον τύπο του κλειδιού. Πχ:

Πίνακας 2.4.10: Παράδειγμα κλήσης της LuaTable::forEach()

```
LuaTable projectiles(-1);

projectiles.forEach(
   [](const LuaString& key, const LuaValue& value, LuaValue::Type type) {
          if (type == LuaValue::Type::Table) {
                ...
          }
    });
```

Το πρώτο όρισμα είναι το κλειδί και είναι πάντα **const LuaString&**. Ακόμα και αν το κλειδί στον πίνακα είναι διαφορετικό (πχ αριθμός), τότε μετατρέπεται σε string. Η τιμή value είναι πάντα **const LuaValue&** και ο τύπος του προσδιορίζεται από το τρίτο όρισμα το type.

Αντίστοιχα λειτουργεί και η **forEachI()**, μόνο που εδώ ο πίνακας αναζητείται με κλειδιά ακέραια και ξεκινάει από το 0 έως τέλος. Δηλαδή η συνάρτηση lambda θα καλεστεί για τα στοιχεία table[0], table[1], table[2] κοκ. Για αυτό και το όρισμα της lambda θα παίρνει μόνο την τιμή του στοιχείου και τον τύπο του.

Πίνακας 2.4.11: Η συνάρτηση LuaTable::forEach()

```
void LuaTable::forEach(
          const std::function<
                void(const LuaString&, const LuaValue&,
                                                LuaValue::Type)>& func) {
    // Check if the reference is valid
    if (luaReference != LUA_NOREF) {
          // Preserve the stack
          saveStack();
          // Push the table on the stack
          lua_getref(lstate, luaReference);
          // Push a nil to make lua_next() iterate
          lua_pushnil(lstate);
          // Iterate through all items with this loop
          while (lua_next(lstate, -2)) {
                // 2nd stage stack preservation
                int stack = lua_gettop(lstate) - 1;
          // duplicate the key on stack to prevent the original to be
                // cast to string if it is a number
                lua_pushvalue(lstate, -2);
                switch (lua_type(lstate, -2)) {
                case LUA_TNUMBER:
                      func(LuaString(lua_tostring(lstate, -1)),
                                  LuaNumber(lua_tonumber(lstate, -2)),
                                  LuaValue::Type::Number);
                      break;
                case LUA_TSTRING:
                      func(LuaString(lua_tostring(lstate, -1)),
                                  LuaString(lua_tostring(lstate, -2)),
```

```
                                    LuaValue::Type::String);
                        break;
                case LUA_TBOOLEAN:
                        func(LuaString(lua_tostring(lstate, -1)),
                                    LuaBoolean(lua_toboolean(lstate, -2)),
                                    LuaValue::Type::Boolean);
                        break;
                case LUA_TTABLE:
                func(LuaString(lua_tostring(lstate, -1)), LuaTable(-2),
                                    LuaValue::Type::Table);
                        break;
                case LUA_TFUNCTION:
                        func(LuaString(lua_tostring(lstate, -1)),
                                    LuaFunctor(-2),
                                    LuaValue::Type::Function);
                        break;
                default:
                        break;
                }
                // 2nd stage Reset (removes the duplicate key and value)
                lua_settop(lstate, stack);
            }
            // Final reset
            resetStack();
        }
}
```

Μέσω του API της lua μπορούμε να ξεκινήσουμε να παίρνουμε όλα τα στοιχεία στον πίνακα. Κάθε φορά που καλείται η **lua_next()**, παίρνει το κλειδί από την στοίβα και φέρνει το επόμενο στοιχείο μαζί με το κλειδί του. Τότε εμείς ελέγχουμε τον τύπο του στοιχείου, καλούμε την συνάρτηση που μας παρέχει το όρισμα με τον αντίστοιχο τύπο και όρισμα της **LuaValue**. Με αντίστοιχο τρόπο λειτουργεί και η **forEachI()**, αλλά αλλάζει ο τρόπος που γίνεται κύκλος στον πίνακα.

Η **printContents()** εκτυπώνει στον stdout όλα τα περιεχόμενα του πίνακα μαζί με τα κλειδιά τους.

Η **create(...)** δημιουργεί και παίρνει αναφορά στον καινούργιο πίνακα. Η τιμή του **reserve** έχει να κάνει με πόσα στοιχεία θα δεσμευτούν στον νέο πίνακα. Αυτό βοηθάει όταν ξέρουμε περίπου πόσα στοιχεία θα πάρει ο πίνακας από πριν.

Η κλάση **LuaFunctor** χρησιμοποιείται για να έχουμε αναφορές Lua σε συναρτήσεις της Lua για να τις καλούμε γρήγορα χωρίς να τις αναζητούμε με το όνομά τους.

Πίνακας 2.4.12: Η κλάση LuaFunctor

```
class LuaFunctor: public LuaReferenced {
public:

     void operator()(std::initializer_list<LuaPushable*> argList) const;
     void operator()() const {
           operator ()( { } );
     }

     bool setFromStack(lua_State* lstate, int stackIndex);
     void set(const std::string& script);
     void setLuaReference(int reference);

     LuaFunctor();
     LuaFunctor(int stackIndex, const std::string& name = "");
     LuaFunctor(const std::string& script);
     ~LuaFunctor();
};
```

Ο τρόπος που αποδίδεται η αναφορά είναι αντίστοιχος του **LuaTable**, μόνο που ελέγχεται αν το στοιχείο που θα γίνει αναφορά είναι συνάρτηση Lua.

Ο κατασκευαστής **LuaFunctor(int, const std::string&)** παίρνει αναφορά από την συνάρτηση στην στοίβα, στον δείκτη **stackIndex** και δίνει το όνομα εφόσον οριστεί. Το όνομα είναι το μέλος **name** της κλάσης **LuaReferenced**. Ο κατασκευαστής **LuaFunctor(const std::string&)** και η συνάρτηση **set(...)** φορτώνουν τον κώδικα Lua που ορίζει το όρισμα **script** και το κάνει σαν αναφορά σε συνάρτηση Lua. Προσοχή! Το **script** δεν είναι διαδρομή σε αρχείο αλλά string που έχει μέσα τον κώδικα.

Οι λειτουργικές συναρτήσεις εδώ είναι οι παραδοχές **operator() ()**. Η απλή συνάρτηση χωρίς όρισμα, απλά καλεί την συνάρτηση που έχομε αναφορά χωρίς ορίσματα. Η παραδοχή με το όρισμα **argList**, δέχεται ένα **std::initializer_list<>** από **LuaPushable** τα οποία θα γίνουν ορίσματα στην συνάρτηση lua όταν καλεστεί.

Πίνακας 2.4.13: Η συνάρτηση LuaFunction::operator ()(std::initializer_list<>)

```
void LuaFunctor::operator ()(
           std::initializer_list<LuaPushable*> argList) const {
     // Check if we have a valid reference
     if (luaReference != LUA_NOREF) {
           lua_State* lstate = LuaEngine::getInstance().getState();
           // Push the function o stack
           lua_getref(lstate, luaReference);
           // Initialize the argument count to 0
           int argC = 0;
           // For all the arguments on the argList
           for (auto arg : argList) {
```

```
                // Check if it not a Null
                if (arg != nullptr) {
                      // If it is not a null, then push it to the stack
                      arg->pushToLua(lstate);
                      // Increase the argument count
                      ++argC;
                }
          }
          // Once we pushed all the arguments on the Lua stack,
          // call the referenced function with all the arguments
          if (lua_pcall(lstate, argC, 0, 0)) {
          // If there was an error while calling the function, then
                // report it!
                Logger::getInstance().write(
                            "During calling Lua Function: " + name
                                  + " Reason:"
                            + std::string(lua_tostring(lstate, -1)));
                // Remove the error message from stack
                lua_pop(lstate, 1);
          }
     }
}
```

Πριν καλέσουμε την συνάρτηση Lua μέσω της **lua_pcall(...)** πρέπει να βάλουμε όλα τα ορίσματα που μας δόθηκαν με την **argList**. Καθώς όλα αυτά είναι τύπου **LuaPushable**, αυτό γίνεται καλώντας την **pushToLua(...)**. Όμως επειδή πρέπει η **lua_pcall(...)** να ξέρει πόσα είναι τα ορίσματα στην στοίβα, έχουμε έναν μετρητή **argC** που τον αυξάνουμε κάθε φορά που βάζουμε ένα όρισμα στην στοίβα. Στο τέλος καλούμε την συνάρτηση με τον αριθμό των ορισμάτων που βάλαμε.

Ο Πυρήνας LuaEngine

Το κύριο κομμάτι του συστήματος Lua είναι η κλάση Singleton **LuaEngine**. Εκεί φιλοξενείται το καθολικό περιβάλλον της Lua το **lua_State** μέσω του οποίου γίνονται όλες οι ενέργειες. Πέρα από αυτό, έχει και ως αρμοδιότητα να κάνει τον επιτηρητή για τις αναφορές σε στοιχεία Lua.

Πίνακας 2.4.14: Η κλάση LuaEngine

```
class LuaEngine: public SingleInstanced<LuaEngine> {
public:

     lua_State* getState();

     void includePath(std::string path);

     int getReference(int index);
     int getReference(const std::string& fullName);
```

```
    void increaseReference(int reference);
    void decreaseReference(int reference);

    void pushToStackValue(const std::string& fullName);
    void registerFunction(const std::string &name, lua_CFunction func);
    void excecuteScriptFile(const std::string& path,
                                            bool force = false);

    void performGarbageCollection();

    ~LuaEngine();
private:
    LuaEngine();

    ZSmallSet<std::string> excecutedScripts;
    ZMap<int, int> sharedReferences;

    lua_State *lstate;

    void createState();
    void destroyState();
    void executeScriptImpl(const std::string& path);
    int getReferenceImpl(const std::string& fullName);

    friend class SingleInstanced;
};
```

Η συνάρτηση **includePath(...)** δέχεται σαν όρισμα μια διαδρομή και την προσθέτει στην λίστα αναζήτησης της συνάρτησης **require()** της Lua. Αυτό που κάνει από πίσω είναι να τρέξει έναν Lua κώδικα που να παραποιεί τον πίνακα που περιέχει τις διαδρομές που ψάχνει η **require()**.

Οι συναρτήσεις **getReference(...)** επιστρέφουν μια αναφορά στο στοιχείο Lua που δείχνει το όρισμα της. Πριν επιστραφεί η αναφορά, πρέπει να δημιουργηθεί η αναφορά του στο **sharedReferences** ή στο **namedReferences**, για να ξέρουμε ότι την δώσαμε αυτή την αναφορά.

Η διαφορά αυτών των δυο συναρτήσεων είναι στο από που παίρνουν την αναφορά. Στην περίπτωση με **int**, η αναφορά παίρνεται από στην στοίβα Lua στον δείκτη που δείχνει το **index**.

Πίνακας 2.4.15: Η συνάρτηση LuaEngine::getReference(int)

```
int LuaEngine::getReference(int index) {
    // Check if there is a value there
    if (!lua_isnoneornil(lstate, index)) {
          // Duplicate the Value on the Stack to preserve it
          lua_pushvalue(lstate, index);
          // Get a reference from the value on top (duplicated value)
          int ret = luaL_ref(lstate, LUA_REGISTRYINDEX);
```

```
            // Register the Reference on the References Table
            sharedReferences[ret] = 1;
            // Return the Reference
            return ret;
      } else {
            return LUA_NOREF;
      }
}
```

Αρχικά ελέγχουμε αν το στοιχείο που δείχνει το **index** είναι έγκυρο. Εφόσον είναι, τότε διπλασιάζουμε το στοιχείο και το βάζουμε πάνω στην στοίβα. Αυτό το κάνουμε γιατί η συνάρτηση **luaL_ref(...)** που καλούμε παρακάτω το αφαιρεί και επειδή ο καλών ίσως το χρειάζεται μετά την κλήση, τότε το διπλασιάζουμε για να μην παρέμβουμε με το στοιχείο που είχε αυτός. Η **luaL_ref(...)** μας δίνει τον αριθμό της αναφοράς και βγάζει το στοιχείο που είναι πάνω στη στοίβα. Στην συνέχεια δημιουργούμε μια εγγραφή στο **sharedReferences** με κλειδί τον αριθμό της αναφοράς και βάζουμε τιμή 1 καθώς είναι η πρώτη φορά που παίρνουμε αναφορά από αυτό το στοιχείο. Το ότι είναι η πρώτη φορά δεν το ξέρουμε σίγουρα, γιατί το στοιχείο βρίσκεται στην στοίβα και είναι αδύνατον να ξέρουμε αν έχουμε δώσει αναφορά για αυτό στο παρελθόν. Στο τέλος επιστρέφουμε την αναφορά.

Το ίδιο κάνει και η δεύτερη μορφή της συνάρτησης, μόνο που το στοιχείο το παίρνει κατευθείαν από το περιβάλλον της Lua. Το string όρισμα που δέχεται είναι το όνομα του στοιχείου μέσα στην Lua. Το όνομα μπορεί να είναι και εμφωλευμένο μέσα σε πίνακες. Για παράδειγμα όταν της δώσουμε το όρισμα “Table1.Table2.SPC” θα πάρει αναφορά στο στοιχείο SPC. Αυτό για να γίνει, χρειάζεται να σπάσουμε το string σε tokens και να αρχίσουμε να ψάχνουμε μέσα από αυτά τους πίνακες. Πχ στο παραπάνω παράδειγμα, θα πρέπει να πάρουμε τον πίνακα Table1, ύστερα από τον Table1 να πάρουμε το πεδίο Table2 και από εκεί να πάρουμε το SPC. Αφού τελειώσουμε με όλα τα tokens, τότε παίρνουμε την αναφορά από το τελευταίο, που είναι το SPC.

Η συνάρτηση λειτουργεί καλώντας την **getReferenceImp(const std::string&)** παίρνοντας την αναφορά από αυτή και ύστερα βάζοντας την στο **sharedReferences**.

Πίνακας 2.4.16: Η συνάρτηση LuaEngine::getReferenceImpl(const std::string&)

```
int LuaEngine::getReferenceImpl(const std::string& fullName) {
     std::list<std::string> tokens;
     // Break the String into tokens separated by '.'
     StringOperations::getTokens(fullName, '.', tokens);
     // Get the First token
     std::string table = *tokens.begin();
```

```
	// Remove the taken token from List to prevent requesting it again
	tokens.erase(tokens.begin());
	// Save the Lua Stack Pointer
	int stackTop = lua_gettop(lstate);
	// Initialize the return value
	int ret = LUA_NOREF;
	// Get the first Token name from Global
	lua_getglobal(lstate, table.c_str());
	if (!lua_isnoneornil(lstate, -1)) {
		// For the rest tokens
		for (auto &token : tokens) {
			// If the item at the top of the Stack is a Lua Table
			if (lua_istable(lstate, -1)) {
				// Put that table on the top of the stack
				lua_getfield(lstate, -1, token.c_str());
			} else {
			// If it's not a Lua Table, then we found the Last item
				break;
			}
		}
		// Check if the value is valid
		if (!lua_isnoneornil(lstate, -1)) {
		// Get a reference to to item on the top of the stack that is
			// the last item retrieved from the previous for()
			ret = luaL_ref(lstate, LUA_REGISTRYINDEX);
		}
	}
	// Reset the Lua Stack
	lua_settop(lstate, stackTop);
	// Return the Lua reference
	return ret;
}
```

Αρχικά παίρνουμε την λίστα από τα tokens μεσώ της **StringOperations::getTokens()** (θα την δούμε αργότερα). Από τα tokens παίρνουμε το πρώτο που είναι το πρώτο στοιχείο. Αυτό το στοιχείο θα παρθεί από τον global πίνακα της Lua, για αυτό και θέλει ειδική μεταχείριση. Όταν πάρουμε το πρώτο στοιχείο στην στοίβα, τότε το ελέγχουμε αν είναι έγκυρο και εφόσον είναι, τότε παίρνουμε διαδοχικά κάθε πεδίο που είναι εμφωλευμένο και ελέγχουμε αν είναι πίνακας. Αν δεν είναι τότε σημαίνει ότι φτάσαμε (πιθανόν) στο τέλος και αυτό το στοιχείο είναι το στοιχείο που θέλει ο καλών αναφορά. Ελέγχουμε αν το στοιχείο είναι έγκυρο και εφόσον είναι, παίρνουμε αναφορά σε αυτό και την αποθηκεύουμε στην μεταβλητή που θα επιστρέψουμε. Αν κάτι δεν πάει καλά, πχ το πρώτο στοιχείο ή κάποιο πεδίο δεν υπάρχει, τότε δεν θα αποδοθεί τιμή στην μεταβλητή που θα επιστραφεί και θα επιστραφεί μηδενική αναφορά. Επίσης αν κάποιο πεδίο που σύμφωνα με το string είναι πίνακας και τελικά ήταν απλό πεδίο, τότε θα επιστραφεί αναφορά σε αυτό το πεδίο.

Οι συναρτήσεις **increaseReference(...)** και **decreaseReference(...)** λειτουργούν όπως και στον **ResourceContext** με την διαφορά ότι εδώ ο πόρος είναι μια αναφορά σε Lua.

Η **performGargbageCollection(...)** καλεί μια ειδική συνάρτηση της Lua που καθαρίζει την άχρηστη μνήμη. Στην ουσία βλέπει ποια κομμάτια μνήμης δεν έχουν αναφορές σε σύμβολα και τα διαγράφει.

2.5 Ο XMLSchemaValidator

Η μηχανή χρησιμοποιεί πολλά διαφορετικά είδη αρχείων XML. Για να ξέρει ότι αυτά τα αρχεία είναι έγκυρα για ανάκτηση δεδομένων, χρειάζεται να το επιβεβαιώσει μέσω των XSD Schemas. Αυτά φορτώνονται σε Validators που στεγάζονται μέσα στον **XMLSchemaValidator**. Από εκεί κάθε κομμάτι της μηχανής μπορεί να έχει πρόσβαση σε αυτά.

Πίνακας 2.5.1: Η κλάση XMLSchemaValidator

```
class XMLSchemaValidator: public SingleInstanced<XMLSchemaValidator> {
public:

     enum class ResourceValidator {
          Sprites,
          Animations,
          Map,
          Tileset,
          AnimationClass,
          LifeformClass,
          Faction
     };

     xmlpp::SchemaValidator& getValidator(ResourceValidator validator);

     void setSchemaFile(const std::string& path,
                                        ResourceValidator validator);

     ~XMLSchemaValidator();
private:
     XMLSchemaValidator();
     xmlpp::SchemaValidator* validators;

     friend class SingleInstanced;
};
```

Η κλάση έχει εσωτερικά αντικείμενα **SchemaValidator** της libxml++. Ο αριθμός τους καθορίζεται από το Enumertaion **ResourceValidator**. Μέσω της **setSchemaFile(...)**, φορτώνονται XSD αρχεία στα Validators. Με την

getValidator(...) επιστρέφονται οι Validators μέσω του ορίσματος **validator** που παίρνει τιμή από το Enumeration **ResourceValidator**. Η φόρτωση γίνεται με κλήσεις στην libxml++ για αυτό και δεν γίνεται περαιτέρω ανάλυση. Αυτό που πρέπει να καταλάβουμε είναι ότι η κλάση λειτουργεί σαν Singleton Container, δηλαδή ένα Container αντικειμένων προσβάσιμο από παντού. Τέτοιου είδους κλάσεις υπάρχουν και αργότερα, για αυτό και θα υπάρξουν αναφορές εδώ.

2.6 Η MainLoop

Ο κύριος βρόγχος της μηχανής σπάει σε δυο μέρη: την υλοποίηση και το αντικείμενο που τον τρέχει. Η υλοποίηση είναι ένα αντικείμενο της **GeneralLoop** ή ένα αντικείμενο κλάσης που την κληρονομεί. Το αντικείμενο που τρέχει τον κύριο βρόγχο είναι ένα αντικείμενο μιας κλάσης που κληρονομεί την Abstract κλάση **LoopRunner**.

Η **LoopRunner** υπάρχει την εξειδικευμένη υλοποίηση του κύριου βρόγχου για κάθε SDK. Για παράδειγμα εάν χρησιμοποιούμε την Allegro5 τότε θα χρειαστεί να φτιάξουμε άλλη κλάση, σε σχέση με το αν χρησιμοποιούσαμε την SFML. Αυτή παρέχει ένα Interface για υλοποίηση, που βάση αυτού θα τρέξει η υλοποίηση του **GeneralLoop**. Η **LoopRunner** δέχεται ένα αντικείμενο **GeneralLoop** και το τρέχει ανάλογα την υλοποίηση που έχει μέσω του SDK.

Πίνακας 2.6.1: Η κλάση LoopRunner

```
class LoopRunner {
public:

	virtual void start()=0;
	virtual int getTargetFps() const=0;
	virtual void setTargetFps(int targetFps)=0;

	void setLoop(GeneralLoop* loop) {
		this->loop = loop;
	}

	GeneralLoop* getLoop() const {
		return loop;
	}

	LoopRunner() :
			loop(nullptr) {

	}

	virtual ~LoopRunner() {

	}
```

```
private:
     GeneralLoop *loop;
protected:
     void render();
     void update();
     void handleEvents(Event& event);
     bool isLoopRunning();
};
```

Η υλοποίηση της **start()** έχει σκοπό να εκτελεί τον κύριο βρόχο ανάλογα το SDK. Ο βρόγχος πρέπει να εκτελείται όσο η συνάρτηση **isLoopRunning()** επιστρέφει true. Ο βρόγχος πρέπει να ελέγχει για συμβάντα σε πρώτη φάση. Εκεί θεωρητικά πρέπει να περιμένει για συμβάν. Όταν συμβεί, τότε το συμβάν πρέπει να μετατραπεί σε συμβάν της μηχανής (**Zeta::Event**) και να καλεστεί με αυτό η **handleEvents()**. Ο βρόγχος πρέπει απαραίτητα κάθε x χρονικά διαστήματα να καλεί την **update()** και ύστερα την **render()**. Το χρονικό διάστημα ορίζεται μέσω της **setTargetFps(...)**. Σε αυτή ορίζεται το πόσες φορές το δευτερόλεπτο θα καλούνται οι **update()** και η **render()**. Ένα παράδειγμα των συναρτήσεων παρατίθεται παρακάτω.

Πίνακας 2.6.2: Παράδειγμα της LoopRunner::start()

```
void AllegroLoop::start() {
     ALLEGRO_EVENT_QUEUE *event_queue = al_create_event_queue();
     al_register_event_source(event_queue,
al_get_keyboard_event_source());
     al_register_event_source(event_queue, al_get_mouse_event_source());
     al_register_event_source(event_queue,
                 al_get_timer_event_source(frame_timer));

     bool redraw = true;
     al_start_timer(frame_timer);
     while (isLoopRunning()) {
           ALLEGRO_EVENT al_event;
           al_wait_for_event(event_queue, &al_event);

           AllegroEvent Z_event(al_event);
           handleEvents(Z_event);
           if (al_event.type == ALLEGRO_EVENT_TIMER) {
                 update();
                 redraw = true;
           }
           if (redraw && al_is_event_queue_empty(event_queue)) {
                 redraw = false;
                 render();
           }
     }
     al_stop_timer(frame_timer);
     al_destroy_event_queue(event_queue);
}
```

```
void AllegroLoop::setTargetFps(int targetFps) {
     this->targetFps = targetFps;
     al_set_timer_speed(frame_timer, 1.0 /
static_cast<double>(targetFps));
}
```

Το παραπάνω παράδειγμα είναι για την Allegro5. Όπως βλέπουμε, έχουμε ένα σύστημα που μας δίνει τα εξωτερικά συμβάντα και το έχουμε ρυθμίσει να μας δίνει και ένα χρονικό συμβάν κάθε 1.0 / targetFPS δευτερόλεπτα για να καλούμε τις συναρτήσεις των **update()** και **render()**. Αρχικά η while() περιμένει για συμβάν και όταν έρθει το μετατρέπει σε συμβάν της μηχανής μέσω της κλάσης **AllegroEvent** που κληρονομεί την **Event**. Ύστερα καλείται η **handleEvents()** με το συμβάν που μετατράπηκε. Μετά ελέγχεται αν είναι συμβάν για frame και αν είναι καλείται η **update()** και μετά θέτεται το flag **redraw** για να καλεστεί η **render()**. Αυτή δεν μπορεί να καλεστεί εάν υπάρχουν ακόμα συμβάντα για να δοθούν στην **handleEvents()**. Αυτό γίνεται όπως είπαμε σε προηγούμενα κεφάλαια, για να μην υπάρχει καθυστέρηση στην αντα-πόκριση σε συμβάν χρήστη. Η κλήση **redraw()** έχει αρκετό Latency (καθυστέρηση), για αυτό και δεν πρέπει να περιμένουμε μέχρι να τελειώσει αυτή για να δει ο χρήστης αποτέλεσμα.

Επιστρέφοντας στην **LoopRunner**, οι κλήσεις **handleEvents(...), update(), render()** καλούν τις αντίστοιχες από το αντικείμενο **GeneralLoop** εφόσον υπάρχει ο Pointer στο μέλος **loop**.

Όπως είπαμε, η **GeneralLoop** είναι το αντικείμενο που θα μπει στην “υποδοχή” του **LoopRunner** και όταν θα τρέξει αυτό, θα καλέσει τις συναρτήσεις **update(), handleEvents()**, και **draw()** του αντικειμένου αυτού.

Πίνακας 2.6.3: Η κλάση GeneralLoop

```
class GeneralLoop {
public:
     typedef std::list<Updateable*>::iterator UpdateIndex;
     typedef std::list<Drawable*>::iterator DrawIndex;
     typedef std::list<SystemEventListener*>::iterator ListenerIndex;

     UpdateIndex addItemToBeUpdated(Updateable* item) {
           if (item != nullptr) {
                 updateableItems.push_back(item);

           }
           return updateableItems.end();
     }
```

```
	UpdateIndex insertItemToBeUpdated(const UpdateIndex& itr,
			Updateable* item) {
		if (item != nullptr) {
			return updateableItems.insert(itr, item);
		}
		return updateableItems.end();
	}
	void removeItemToBeUpdated(const UpdateIndex& itr) {
		updateableItems.erase(itr);
	}

	DrawIndex addItemToBeDrawn(Drawable* item);
	DrawIndex insertItemToBeDrawn(const DrawIndex& itr, Drawable* item);
	void removeItemToBeDrawn(const DrawIndex& itr);

	ListenerIndex addSystemListener(SystemEventListener* item);
	ListenerIndex insertSystemListener(const ListenerIndex& itr,
			SystemEventListener* item);
	void removeSystemListener(const ListenerIndex& itr);

	void start() {
		running = true;
	}

	void stop() {
		running = false;
	}

	inline bool isLoopRunning() const {
		return running;
	}

	float getCurrentFPS() const;

	void update();
	void draw();
	void handleEvent(Event& event);

	GeneralLoop();
	virtual ~GeneralLoop();
private:
	typedef std::chrono::duration<float, std::chrono::seconds::period>
fpTime;

	std::list<Updateable*> updateableItems;
	std::list<Drawable*> drawableItems;
	std::list<SystemEventListener*> eventListeners;
	std::chrono::high_resolution_clock::time_point ltime;
	RenderingContext* renderingContext;
	float elapsed;
	float curFps;
	bool running;
};
```

Η κλάση βλέπουμε ότι έχει τρεις διαφορετικές λίστες. Η κάθε μια από αυτή θα έχει αντικείμενα που θα χρησιμοποιηθούν όταν καλεστούν οι αντίστοιχες συναρτήσεις.

Δηλαδή στα αντικείμενα στην λίστα **updateableItems**, θα καλεστεί για καθένα από αυτά η συνάρτηση μέλος **Updateable::update()** όταν καλεστεί η **update()** της **GeneralLoop**. Το ίδιο ισχύει και για τα αντίστοιχα αντικείμενα της **drawableItems**, που θα σχεδιαστούν όταν καλεστεί η **draw()**, όπως και με τα **SystemEventListener** που θα τους διανεμηθεί το συμβάν όταν καλεστεί η **handleEvent(...)**.

Οι υπόλοιπες συναρτήσεις είναι απλές. Οι **addxxx(...)** προσθέτουν το αντικείμενο στο τέλος της ουράς για την λίστα που αντιπροσωπεύουν και επιστρέφουν έναν list iterator για την τοποθεσία που προστέθηκε. Το ίδιο κάνουν και οι **insertxxx(...)** με την διαφορά ότι τοποθετούν το αντικείμενο στην θέση που ορίζεται από τον Iterator πού δίνεται σαν όρισμα.

Οι **start()** και **stop()**, απλά θέτουν την μεταβλητή για να επικοινωνεί με τον **LoopRunner** και να ελέγχει πότε θα ξεκινήσει και θα σταματήσει.

Από αυτές αυτό που αξίζει να δούμε είναι η **update()**.

Πίνακας 2.6.4: Η συνάρτηση GeneralLoop::update()

```
void GeneralLoop::update() {
	// Get the current Time point
	auto begin = std::chrono::high_resolution_clock::now();
	// get The difference between the Last Time and now
	elapsed = fpTime(begin - ltime).count();
	// For all the items on the Update List
	for (auto updateable : updateableItems) {
		try {
			// try and update the item
			updateable->update(elapsed);
		} catch (Exception& ex) {
			// If Zeta related excetion happened, report it
			Logger::getInstance().write(...);
		} catch (CEGUI::Exception& e) {
			// If CEGUI related excetion happened, report it
			Logger::getInstance().write(...);
		} catch (...) {
			// If anything else happened, report it too!
			Logger::getInstance().write(...);
		}
	}
	// Calculate the Frame Rate
	curFps = 1.0f / elapsed;
	// Set the new Last Time to be the now
	ltime = begin;
}
```

Η συνάρτηση αυτή, πέρα από την κλήση του μέλους **update()** των αντικειμένων, φροντίζει να υπολογίσει τον χρόνο που μεσολάβησε από την προηγούμενη κλήση της. Αυτό γίνεται παίρνοντας την χρονική στιγμή που καλέστηκε η συνάρτηση και

αφαιρώντας την από την προηγούμενη χρονική στιγμή που καλέστηκε. Έτσι έχουμε τον χρόνο που πέρασε από την προηγούμενη. Αυτό είναι αναγκαίο, γιατί η συνάρτηση μέλος **update()** χρειάζεται να το ξέρει αυτό, γιατί μερικά αντικείμενα χρειάζονται αυτό το χρονικό διάστημα για να υπολογίζουν χρόνους. Πέρα από αυτό, υπολογίζεται και το Frame Rate. Στο τέλος αποθηκεύουμε την χρονική στιγμή που καλέστηκε για να το ξέρει η επόμενη κλήση.

Μια κλήση ακόμα που πρέπει να δούμε είναι η **draw()**.

Πίνακας 2.6.5: Η συνάρτηση GeneralLoop::draw()

```
void GeneralLoop::draw() {
     // For all the Items on the Draw List
     for (auto drawable : drawableItems) {
           // Draw with no offsets
           drawable->draw(0.0f, 0.0f);
     }
     // Execute any operations on the RenderingContext
     renderingContext->executeOperations();
     // Flip the Framebuffer
     renderingContext->getDisplay().switchFramebuffer();
}
```

Εδώ βλέπουμε ότι αφού σχεδιαστούν όλα τα αντικείμενα, τότε δίνεται η εντολή στο **renderingContext** να εκτελέσει όλες τις λειτουργίες του και να φέρουμε τον πίσω Frame Buffer μπροστά καθώς σε αυτόν σχεδιάσαμε. Έτσι σε αυτό που πήγαμε πίσω θα σχεδιάσουμε όταν ξανά καλεστεί η **draw()**.

Όπως είδαμε, η κλάση αυτή παράγει αντικείμενο που θα δοθεί στον σε έναν **LoopRunner** για να το τρέξει. Πριν το τρέξει, θα πρέπει να γεμίσουμε τις λίστες του αντικειμένου με στοιχεία. Για να ήμαστε σίγουροι, κατασκευάσαμε μια κλάση που κληρονομεί την **GeneralLoop** και στον κατασκευαστή της γεμίζουμε τις λίστες με τα απαραίτητα αντικείμενα. Πέρα από αυτό, θέλουμε αυτό το αντικείμενο να είναι καθολικά προσβάσιμο και να μπορεί να μπει στο περιβάλλον Lua. Έτσι έχουμε την κλάση **MainLoop**.

Πίνακας 2.6.6: Η κλάση MainLoop

```
class MainLoop: public GeneralLoop,
           public SingleInstanced<MainLoop>,
           public LuaPushable {
public:
     void pushToLua(lua_State* lstate);
private:
```

```
        MainLoop();
        ~MainLoop() {

        }
        GUIManager* guiMgr;
        WorldManager* worldMgr;

        friend class SingleInstanced;
};
```

Τα δυο αντικείμενα που πρέπει να μπουν αναγκαστικά στις λίστες για σχεδίαση και ενημέρωση (update) είναι το **GUIManager** που έχει να κάνει με το CEGUI (θα δούμε παρακάτω) και το **WorldManager** που έχει να κάνει με όλο το παιχνίδι. Το **GUIManager** πρέπει να μπει και στην λίστα των **eventListeners** καθώς πρέπει να ακούει και τα συμβάντα χρήστη το GUI.

Καθώς η κλάση κληρονομεί την **LuaPushable** είναι εφικτή για να μπει στην Lua. Η **SingleInstanced** την κάνει singleton.

Τώρα χρειάζεται μια υλοποίηση της **LoopRunner** από ένα SDK για να τρέξει. Για παράδειγμα στο Demo, η Loop αρχίζει με τον παρακάτω κώδικα.

Πίνακας 2.6.7: Παράδειγμα εκκίνησης της MainLoop

```
Zeta::AllegroLoop loop(fps);
loop.setLoop(&Zeta::MainLoop::getInstance());
loop.start();
```

3 ΤΟ ΣΥΣΤΗΜΑ ANIMATION

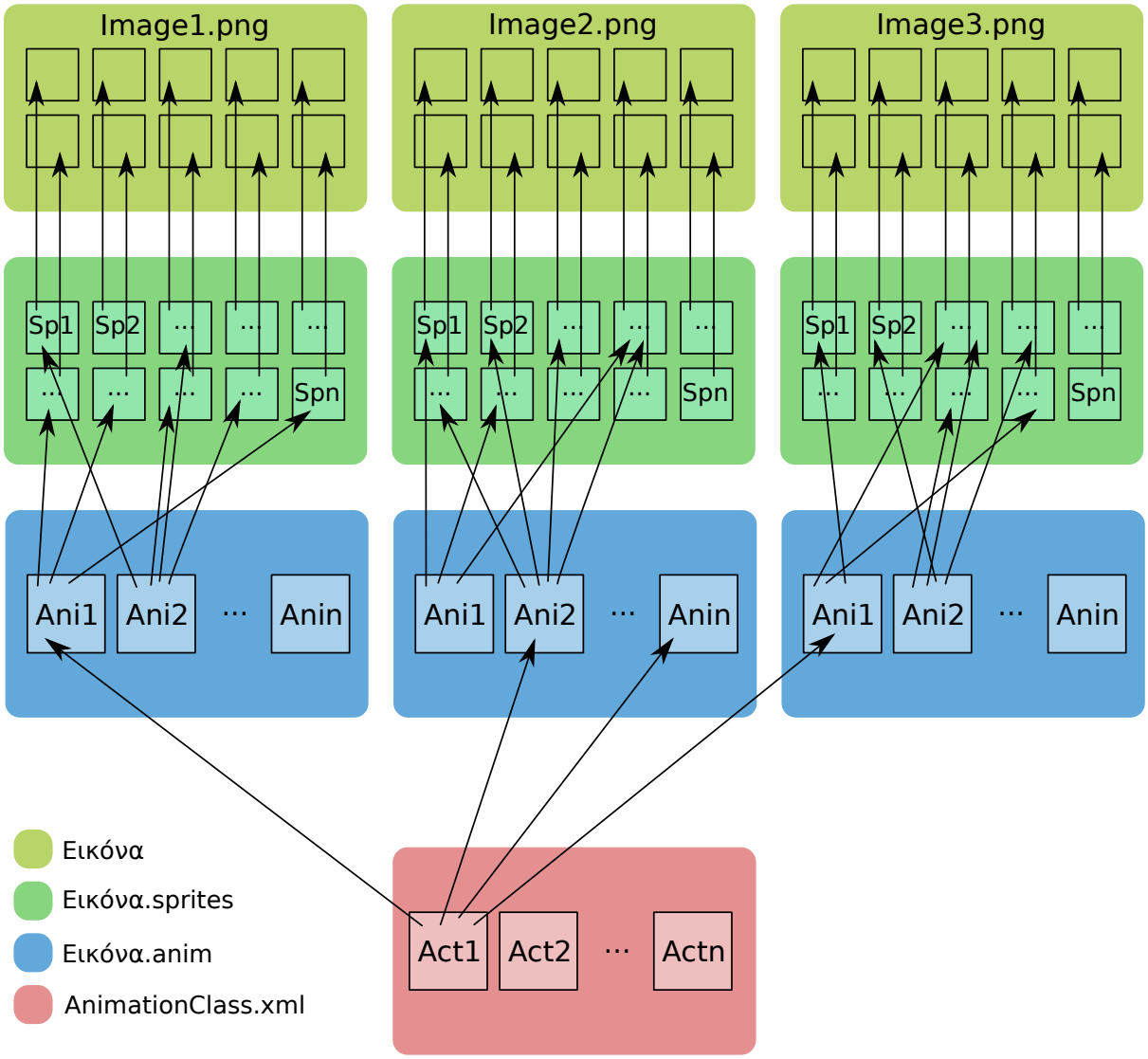
Το σύστημα Animation βασίζεται γύρω από πληροφορίες σε αρχεία XML και αρχεία εικόνας. Μια εικόνα με ένα αρχείο *.sprites ορίζει τα επιμέρους sprites σε μια μεγάλη εικόνα που περιέχει πολλά. Όπως είπαμε σε προηγούμενο κεφάλαιο, τα sprites που είναι συγγενικά πρέπει να τοποθετούνται στην ίδια μεγάλη εικόνα. Αυτή η μεγάλη εικόνα που τα περιέχει λέγεται **Spritesheet**. Από αυτή την εικόνα κατασκευάζουμε το αρχείο *.sprites που ορίζει σε τις συντεταγμένες, το μέγεθος και το όνομα του κάθε sprite. Η μηχανή από το αρχείο φορτώνει την μεγάλη εικόνα και την σπάει σε μικρότερα sprites που ορίζονται απο αυτό. Αυτά τα sprites για την μηχανή θεωρούνται μια ξεχωριστή εικόνα, παρόλο που στην VRAM μοιράζονται την ίδια με άλλα συγγενικά.

Από το αρχείο *.sprites, ύστερα κατασκευάζεται το αρχείο *.anim που ορίζει τα Animations. Αυτό το αρχείο ορίζει για κάθε animation το όνομα του, τα Cells τα ο ποια είναι τα Frames του Animation και τα sprites που έχει κάθε Cell. Έτσι με αυτό το αρχείο έχουμε μια δέσμη Animations από ένα Spritesheet. Αυτό εσωτερικά στην μηχανή λέγετε **AnimationPack**.

Από πολλά AnimationPack κατασκευάζονται αρχεία που ορίζουν τα Animations που θα έχει κάποιο Lifeform. Εκεί ορίζονται ονόματα από ενέργειες που θα κάνει το Lifefeorm, τα Animations που θα έχουν αυτές οι ενέργειες για κάθε κατεύθυνση και σε ποιο AnimationPack βρίσκονται. Αυτά τα αρχεία λέγονται **AnimationClass**.

3.1 Τα αντικείμενα Animation

Η ανάλυση ξεκινάει από το πιο μικρό στοιχείο και πηγαίνει στα μεγαλύτερα. Εδώ το μικρότερο στοιχείο είναι η εικόνα και το Sprite. Η εικόνα όπως είπαμε θα περιέχει πολλά spites. Λόγο της κοινόχρηστης φύση της και δεδομένου ότι είναι αρχείο, η εικόνα θα αναπαρίσταται στην Engine από την κλάση Bitmap που είδαμε σε προηγούμενο κεφάλαιο. Τα Sprites είναι αντικείμενα που πρέπει να μπορούν να σχεδιαστούν, για αυτό και πρέπει να έχουν εσωτερικά ένα Bitmap. Επειδή όμως δεν μπορούμε να αντιγράψουμε την εικόνα στην μνήμη (θα ήταν μεγάλη σπατάλη πόρων), η κλάση Bitmap μας δίνει την δυνατότητα των SubBitmaps ή υπό-εικόνων. Αυτή την δυνατότητα χρησιμοποιούμε στην κλάση Sprite.



Εικόνα 3.1.1: Δένδρο ιεραρχίας του συστήματος Animation

Στο παραπάνω διάγραμμα βλέπουμε πως συσχετίζονται τα διάφορα κομμάτια του συστήματος Animation. Μέχρι στιγμής έχουμε δει την μητρική εικόνα(κίτρινο) και πώς από αυτή παίρνουμε τα Sprites μέσω του αρχείου SpriteSheet *.sprites. Τα αντικείμενα SpriteSheet αντιπροσωπεύουν αυτά τα αρχεία και παρέχουν στα άλλα κομμάτια αναφορές στα επιμέρους Sprites. Πέρα από αυτά τα αρχεία, είπαμε ότι υπάρχουν και τα *.anim που αντιπροσωπεύονται από την κλάση AnimationPack (μπλε).

Πίνακας 3.1.1: Η κλάση AnimationPack

```
class AnimationPack: public Resource {
public:
     const Animation& getAnimation(const std::string& name) const;
     const SpriteSheet& getSpriteSheet() const;
     std::vector<std::string> listAnimationNames() const;
```

```
     AnimationPack();
     AnimationPack(const std::string& path);
     ~AnimationPack();
protected:
     ZSmallMap<std::string, Animation> animations;
     const SpriteSheet *spriteSheet;
};
```

Η AnimationPack ευθύνεται για την δημιουργία και φιλοξενία των Animation από τα αρχεία *.anim. Είναι και αυτό πόρος και έτσι έχουμε τους δυο κατασκευαστές, έναν για κενό AnimationPack και έναν για την φόρτωση από αρχείο. Τα Animations φιλοξενούνται στο Hashtable **animations**.

Πίνακας 3.1.2: Ο κατασκευαστής AnimationPack::AnimationPack(const std::string&)

```
AnimationPack::AnimationPack(const std::string& path) :
          Resource(path) {
     // Ask the RESCON for the file the path says.
     auto& resMan = System::getInstance().getResourcesManager();

     /*
      * Κώδικας για Parse Και Validation
      */

     xmlpp::Element *root = parser.get_document()->get_root_node();

     // Get the Referenced SpriteSheet from RESCON
     spriteSheet = &resMan.getSpriteSheet(
                 StringOperations::translateUnixPath(path,
                 root->get_attribute_value("spriteSheet").c_str()));
     // Execute the XPATH Query to find all <anims>
     xmlpp::NodeSet anims = root->find("//anim");

     // For all the anims
     for (auto node : anims) {
     animations[static_cast<xmlpp::Element*>(node)->get_attribute_value(
                       "name").c_str()].set(node, *this);
     }
}
```

Όπως με κάθε αρχείο XML, αρχικά κάνουμε Parse και Validate με το αντίστοιχο XSD Schema. Αν περάσουν όλοι οι έλεγχοι, τότε ζητάμε από τον ResourcesContext το SpriteSheet που μας λέει το αρχείο. Ο ResourcesContext θα μας δώσει το SpriteSheet φορτωμένο όπως είπαμε παραπάνω. Ύστερα αναζητάμε όλα τα στοιχεία <anim>. Κάθε τέτοιο στοιχείο στο αρχείο ορίζει ένα Animation. Έτσι κατασκευάζουμε ένα αντικείμενο Animation στο **animations** και το κάνουμε να φορτωθεί το animation

από τις πληροφορίες του xml node που έχουμε. Ας δούμε πως λειτουργεί η κλάση **Animation**.

Πίνακας 3.1.3: Η κλάση Animation

```
class Animation {
public:
	const std::string& getName() const;
	void setName(const std::string& name);
	bool isLooping() const;
	const AnimationPack& getAnimationPack() const;

	inline int getNumFrames() const {
		return cells.size();
	}

	inline const Cell& getFrame(unsigned int index) const {
		return cells[index];
	}

	void set(xmlpp::Node *node, const AnimationPack &parent);

	Animation();
	Animation(xmlpp::Node *node, const AnimationPack &parent);
	virtual ~Animation();
protected:
	std::string name;
	std::vector<Cell> cells;
	const AnimationPack *parentPack;
	bool loops;
};
```

Οι πρώτες τέσσερις συναρτήσεις επιστρέφουν/θέτουν τα μέλη που περιγράφουν. Βλέπουμε ότι το Animation έχει όνομα και έναν πίνακα από αντικείμενα **Cell** που είναι τα Frames του Animation.

Πίνακας 3.1.4: Η συνάρτηση Animation::set()

```
void Animation::set(xmlpp::Node* node, const AnimationPack& parent) {
	this->cells.clear();
	xmlpp::Element* anim = static_cast<xmlpp::Element*>(node);
	// Get the name of the Animation
	name = anim->get_attribute_value("name").c_str();

	// Get whether the animation loops
	if (strtol(anim→get_attribute_value("loops").c_str(),
                                                    nullptr, 0) == 0) {
		loops = false;
	} else {
		loops = true;
	}
```

```
	// Get all the <cell> children
	xmlpp::NodeSet cells = anim->find("./cell");

	// For all the cells
	for (auto nd : cells) {
		// cast it to Element to gain access to its Attributes
		xmlpp::Element *cellNode = static_cast<xmlpp::Element*>(nd);

		// Create a new Cell with the stated delay the XML describes
		Cell tmpCell(
			strtol(cellNode->get_attribute_value("delay").c_str(),
					nullptr,
					0) / 100.0f);

		tmpCell.clearSprites();
		// Get all the <spr> children
		xmlpp::NodeSet sprites = cellNode->find("./spr");

		// For all the sprites
		for (auto spr : sprites) {
			tmpCell.addSprite(spr, parent.getSpriteSheet());
		}
		this->cells.push_back(std::move(tmpCell));
	}

	parentPack = &parent;
}
```

Η **set(...)** οφείλεται για την φόρτωση του Animation. Θεωρούμε ότι στο **node** έχουμε ένα έγκυρο στοιχείο <anim>. Επειδή μπορεί να έχει καλεστεί αφού έχει φορτωθεί ήδη ένα Animation στο αντικείμενο, χρειάζεται να καθαρίσουμε τον πίνακα **cells** από τα υπάρχων Cells. Ύστερα παίρνουμε το όνομα από το attribute **name**. Ύστερα ελέγχουμε την τιμή του attribute **loops** και θέτουμε την αντίστοιχη τιμή στο μέλος. Σε επόμενη φάση αναζητάμε όλα τα εμφωλευμένα στοιχεία <cell>. Για καθένα από αυτά δημιουργούμε ένα αντικείμενο **Cell**.

Πίνακας 3.1.5: Η κλάση Cell

```
class Cell: public Drawable {
public:
	void draw(Float x, Float y, Float rotation = 0.0f, Float scale =
			1.0f) const;
	void addSprite(const Sprite& spr);
	void addSprite(xmlpp::Node *sprNode, const SpriteSheet& spriteSheet)
	{
		addSprite(Sprite(sprNode, spriteSheet));
	}

	inline Float getDelay() const {
		return delay;
	}
```

```
	void clearSprites() {
		sprites.clear();
	}

	Cell() : delay(1) {
		sprites.push_back(Sprite());
	}
	Cell(Float delay) :
			delay(delay) {
		sprites.push_back(Sprite());
	}
	Cell(Float delay, const Sprite& sprite) :
			delay(delay) {
		sprites.push_back(sprite);
	}
	~Cell() {
	}
protected:
	std::list<Sprite> sprites;
	Float delay;
};
```

Όπως βλέπουμε, τα αντικείμενα **Cell** μπορούν να σχεδιαστούν καθώς η κλάση κληρονομεί την **Drawable**. Όλοι οι κατασκευαστές θέτουν τις τιμές όπως ορίζονται, καθώς και προσθέτουν ένα κενό **Sprite** στην λίστα των **sprites** που έχει το **Cell**. Η βασική προσθήκη ενός **Sprite** γίνεται μέσω την συνάρτησης **addSprite(const Sprite& spr)**.

Πίνακας 3.1.6: Η συνάρτηση Cell::addSprite(const Sprite& spr)

```
void Cell::addSprite(const Sprite& spr) {
	// We have to see where it will be inserted
	auto itr = sprites.begin();
	// for all sprites we already have added
	for (; itr != sprites.end(); ++itr) {
		// check if the Z value is greater that ours
		if ((*itr).getZ() > spr.getZ()) {
			// if it is, then we have to put it there
			break;
		}
	}
	// insert the sprite on the iterator we found
	sprites.insert(itr, spr);
}
```

Τα Sprites μέσα σε ένα Cell, ταξινομούνται με βάση το πιο θα σχεδιαστεί μπροστά, πιο θα είναι πίσω από αυτό, κοκ. Αυτή η ταξινόμηση γίνεται με βάση μια τιμή που δίνεται σε κάθε Sprite μέσα στο Cell, την τιμή Z. Sprites με μικρές τιμές Z σχεδιάζονται μπροστά και αυτά με μεγαλύτερες σχεδιάζονται πίσω από αυτά. Για να μπορέσουμε

να προσθέσουμε ταξινομημένο το Sprite, εφαρμόζουμε μια τεχνική αλγορίθμου Insert Sort. Στην ουσία συγκρίνουμε την τιμή μας με την τιμή του εκάστοτε Sprite που υπάρχει ήδη και αν βρούμε κάτι μεγαλύτερο, τότε το τοποθετούμε πριν από αυτό. Η λειτουργία αυτή επιτυγχάνεται με το να αποθηκεύουμε τον iterator και να τερματίσουμε τον βρόγχο όταν πληροί η προϋπόθεση. Όταν τερματιστεί ο βρόγχος, τότε έχουμε στον iterator την θέση που θα βάλουμε το Sprite.

Η **draw(...)** απλά καλεί την **draw(...)** για κάθε sprite στην λίστα **sprites** στην σειρά, για να σχεδιαστούν σωστά. Η σειρά είναι σωστή, καθώς αυτό το φροντίζει η **addSprite()**.

Είδαμε την **addSprite(const Sprite&)** αλλά η συνάρτηση **Animation::set()** καλεί την δεύτερη μορφή της την **addSprite(xmlpp::Node*, const SpriteSheet&)**. Όπως βλέπουμε στον κώδικα, αυτή απλά καλεί την προηγούμενη αφού έχει κατασκευάσει ένα προσωρινό **Sprite**.

Πίνακας 3.1.7: Η κλάση Sprite

```
class Sprite: public Drawable {
public:
     inline int getZ() const {
           return z;
     }

     void draw(Float x, Float y, Float rotation = 0.0f,
                 Float scale = 1.0f) const;

     Sprite();
     Sprite(xmlpp::Node *node, const SpriteSheet& spriteSheet);
     ~Sprite();
protected:
     const Bitmap *image;
     int x;
     int y;
     int z;
     float angle;
     Bitmap::FlipFlag flip;
};
```

Τα αντικείμενα αυτά είναι σχεδιάσιμα, καθώς έχουν την τελική υπό-εικόνα **image** που θα σχεδιαστεί. Έχουν τιμές για Offsets σε άξονες x,y για να σχεδιαστεί σωστά στο Animation. Περισσότερες πληροφορίες γύρω από αυτά, έχουμε δει στο δεύτερο μέρος. Το Sprite κατασκευάζεται και φορτώνεται από τον δεύτερο κατασκευαστή.

Πίνακας 3.1.8: Η συνάρτηση Sprite::Sprite(xmlpp::Node*, const SpriteSheet&)

```
Sprite::Sprite(xmlpp::Node* node, const SpriteSheet& spriteSheet) :
          Sprite() {
     if (node != nullptr) {
          // Cast to Element to get the Attributes
          xmlpp::Element *spr = static_cast<xmlpp::Element*>(node);
          // Get the fields
          x = strtol(spr->get_attribute_value("x").c_str(), nullptr, 0);
          y = strtol(spr->get_attribute_value("y").c_str(), nullptr, 0);
          z = strtol(spr->get_attribute_value("z").c_str(), nullptr, 0);
          angle = strtof(spr->get_attribute_value("angle").c_str(),
                                    nullptr);

     // OR the return number of strtol so we can get the final flip Flag
          flip =
                static_cast<Bitmap::FlipFlag>((strtol(
          spr->get_attribute_value("flipH").c_str(), nullptr, 0) << 1)
                | strtol(spr->get_attribute_value("flipV").c_str(),
                                                  nullptr, 0));
          image = &spriteSheet.getSprite(
                      spr->get_attribute_value("name").c_str());
     }
}
```

Αρχικά παίρνουμε τα στοιχεία x,y,z από το xml στοιχείο. Το μέλος flip είναι Bitfield. Το πρώτο Bit ορίζει εάν το Sprite θα είναι αντεστραμμένο κάθετα. Το δεύτερο Bit καθορίζει αν θα είναι αντεστραμμένο οριζοντίως. Για να πάρουμε την τελική τιμή, κάνουμε πράξη OR ανάμεσα στις δυο τιμές που μας δίνει το στοιχείο. Σε τελική φάση παίρνουμε την υπό-εικόνα από το SpriteSheet που μας δόθηκε και το όνομα που ορίζει το attribute **name** του στοιχείου xml που μας δόθηκε.

Τώρα ήρθε η ώρα να βάλουμε όλες αυτές τις τιμές σε εφαρμογή. Η εφαρμογή τους δεν είναι τίποτα άλλο από την σχεδίαση τους μέσω της **draw()**.

Πίνακας 3.1.9: Η συνάρτηση Sprite::draw()

```
void Sprite::draw(Float x, Float y, Float rotation, Float scale) const {
     image->drawAtCentre(x + (this->x * scale), y + (this->y * scale),
                (angle * 0.0174532925f) + rotation, scale, flip);
}
```

Η κλήση **draw()** προέρχεται από την abstract κλάση Drawable που κάνει όλες τις κλάσεις που την κληρονομούν να μπορούν να σχεδιαστούν στην οθόνη. Τα ορίσματα που δέχεται η συνάρτηση είναι τα **x,y** που είναι τιμές Offset που πρέπει να εφαρμοστούν. Η τιμή **rotation** είναι η κλίση που πρέπει να εφαρμοστεί στην σχεδίαση. Η

τιμή **scale** είναι η κλίμακα που πρέπει να εφαρμοστεί στην σχεδίαση. Πχ για τιμή 1.3, η σχεδίαση θα μεγεθυνθεί κατά 30% κοκ. Η κλήση αυτή πρέπει περνάει τα ορίσματα της αθροιστικά σε οποιαδήποτε κλήση της **draw()** γίνει εσωτερικά.

Εδώ τα **x,y** είναι εκεί που πρέπει να σχεδιάσουμε καθώς δεν έχουμε δικές μας συντεταγμένες. Η σχεδίαση θα γίνει στο κέντρο, όπως ορίζει το πρωτόκολλο της μηχανής. Στα offset **x,y** πρέπει να αθροίσουμε τα εσωτερικά Offsets. Για να γίνει αυτό σωστά, επειδή σχεδιάζουμε στο κέντρο και με πιθανή κλίμακα, πρέπει να εφαρμόσουμε την κλίμακα στο εσωτερικό Offset πριν το προσθέσουμε. Η γωνία που μας δίνεται από το αρχείο xml για την περιστροφή είναι σε μοίρες. Όμως η σχεδίαση απαιτεί την γωνία σε ακτίνια. Έτσι την μετατρέπουμε σε ακτίνια πολλαπλασιάζοντας την με 0,0174... αθροίζοντας την υπάρχουσα γωνία που μας δίνεται (**rotation**). Η κλίμακα και η αντιστροφή περνιούνται όπως έχουν.

Με την παραπάνω κλάση κατασκευάζεται το αντικείμενο Sprite στην συνάρτηση **Cell::addSprite()**. Το κατασκευασμένο Sprite τοποθετείται στην σωστή σειρά και επιστρέφει στην **Animation::set() (3.1.8)** που τοποθετούνται τα Sprites μέσα στο **tmpCell**. Αφού μπουν όλα τα Sprites, τότε το Cell μπαίνει στην λίστα των Cells του Animation. Τελικά αυτή επιστρέφει στον Constructor **AnimationPack**. Εκεί για κάθε στοιχείο <anim> κατασκευάζει το αντίστοιχο Animation και στο τέλος το AnimationPack έχει φορτωθεί ολοκληρωτικά.

Το τελευταίο κομμάτι του δένδρου των Animation είναι η κλάση **AnimationClass**. Όπως ξέρουμε, ένα AnimationClass είναι μια ομάδα από Animations από διάφορα AnimationPacks. Τα Animations εσωτερικά ομαδοποιούνται σε ενέργειες **(Actions)** που θα κάνει το **Lifeform** που θα έχει αυτό το AnimationClass. Κάθε ομάδα έχει 4 Animations, ένα για τις 4 κατευθύνσεις. Σε κάθε κατεύθυνση ορίζεται το όνομα του Animation που θα χρησιμοποιήσει το **Lifeform** όταν κάνει αυτό το Action, καθώς και σε ποιο AnimationPack βρίσκεται.

Κάθε Animation έχει μοναδικό όνομα εντός του AnimationClass, που ορίζεται από το όνομα της ενέργειας και την κατεύθυνση. Η μορφή είναι: “Action/Direction”. Σε κάθε AnimationClass ορίζονται και κάποιες προδιαγεγραμμένες ενέργειες που είναι οι βασικές που έχει κάθε **Lifeform**. Αυτές έχουν σταθερά ονόματα και μπαίνουν σε ειδικό κομμάτι μνήμης εντός του αντικειμένου. Τα Animation που βρίσκονται σε αυτές τις ομάδες λέγονται και **Primitive Animations**.

Πέρα από τα Animations, τα αντικείμενα αυτά στεγάζουν και άλλα αντικείμενα στενά συνδεδεμένα με τα Animations. Αυτά είναι το ορθογώνιο σύγκρουσης, το ορθογώνιο στόχευσης και η έλλειψη της σκιάς.

Τα Animations ανακτώνται από την **getAnimation(...)** είτε με την έκδοση με το **std::string** είτε με τον αριθμό **LifeformState**. Η πρώτη αναζητά το string στο **animations** και αν υπάρχει, τότε επιστρέφει το Animation που δείχνει. Αν δεν υπάρχει, τότε επιστρέφει το **fallback** Animation μέσω της άλλης έκδοσης της **getAnimation()**.

Πίνακας 3.1.10: Η συνάρτηση AnimationClass::getAnimation(int)

```
const Animation& AnimationClass::getAnimation(
           LifeformState::CombinedState state) const {
     // Check if the state is valid
     if ((state & 0xF) <= LifeformState::maxDirectionValue
                 && (state & 0xF) > 0
                 && (state >> 4) <= LifeformState::maxActionValue
                 && (state >> 4) > 0) {
           // If it is valid, then return the animation from the table
           return *primAnimations((state & 0xF) - 1, (state >> 4) - 1);
     }
     return *nullAnimation;
}
```

Το **LifeformState** είναι ένας ακέραιος που τα Bits του χρησιμοποιούνται για αποθήκευση της κατεύθυνσης και του Action που κάνει εκείνη την στιγμή το Lifeform. Τα 4 πρώτα Bits έχουν να κάνουν με την κατεύθυνση και τα 4 επόμενα με το Action. Σπάζοντας αυτόν τον αριθμό στους 2 αυτούς τους αριθμούς που αποτελείται, μας δίνει που βρίσκεται το Animation στον πίνακα **primAnimations** για αυτόν τον συνδυασμό. Για αυτό και πρέπει να ελέγχουμε ότι αυτοί οι αριθμοί βρίσκονται εντός ορίων του πίνακα. Το σπάσιμο γίνεται με λογικές πράξεις πάνω στον συνδυασμένο αριθμό. Κάνοντας AND κρατάμε τα Bits της κατεύθυνσης. Όταν θέλουμε να πάρουμε το Action, τότε απλά ολισθένουμε τον αριθμό κατά 4 για να πάνε τα Bits του Action στην αρχή. Αυτό σβήνει αυτόματα τα Bits της κατεύθυνσης. Έτσι παίρνουμε τους 2 αριθμούς και εφόσον είναι εντός ορίων, επιστρέφουμε το **primAnimation** που δείχνουν. Αν είναι εκτός ορίων, τότε επιστρέφουμε το **NullAnimation**.

3.2 Τα αντικείμενα AnimationPlayer

Μέχρι στιγμής έχουμε δει πως φορτώνονται τα αντικείμενα Animation και όλα τα συναφή τους. Αυτά απλά περιέχουν τις πληροφορίες και τα δεδομένα και είναι παθητικά. Με άλλα λόγια αυτά τα Animation δεν μπορούν να παιχτούν από μόνα τους. Αυτό γιατί τα Animation είναι κοινόχρηστα σε διάφορα Lifeform και δεν μπορούν να φιλοξενήσουν εσωτερικά δεδομένα για την αναπαραγωγή καθώς αυτά θα διαφέρουν από Lifeform σε Lifeform. Για αυτό υπάρχουν τα αντικείμενα **AnimationPlayer**.

Τα AnimationPlayer κάνουν την δουλειά που δεν μπορούν να κάνουν τα Animation. Σε αντίθεση με τα κοινόχρηστα Animation, τα αντικείμενα AnimationPlayer μπορούν να υπάρχουν σε πλειάδες. Η αρμοδιότητα τους είναι να δέχονται ένα Animation και να το αναπαράγουν και να το σχεδιάζουν όποτε πρέπει.

Πίνακας 3.2.1: Η κλάση AnimationPlayer

```
class AnimationPlayer: public Drawable, public Updateable {
public:

	bool isPlaying() const {
		return playing;
	}
	bool isVisible() const {
		return visible;
	}
	void setVisible(bool visible) {
		this->visible = visible;
	}
	const Animation& getAnimation() const {
		return *animation;
	}
	void hide() {
		visible = false;
	}
	void show() {
		visible = true;
	}
	void play() {
		playing = true;
	}
	void stop() {
		playing = false;
		frameTimer = 0;
		frameIndex = 0;
	}
	void pause() {
		playing = false;
	}
	void reset() {
		frameTimer = 0;
		frameIndex = 0;
	}
```

```
    void setAnimation(const Animation& animation);
    virtual void draw(Float x, Float y, Float rotation = 0.0f, Float
scale =
                1.0f) const;

    void update(Float elapsedTime);
    AnimationPlayer();


    AnimationPlayer(const Animation& animation) :
                animation(&animation), frameIndex(0), frameTimer(0),
                      playing(true), visible(true) {

    }
    virtual ~AnimationPlayer() {

    }
private:
    const Animation* animation;
    int frameIndex;
    Float frameTimer;
    bool playing;
    bool visible;
};
```

Η κλάση κληρονομεί την **Drawable** κάνοντας την σχεδιάσιμη μέσω της **draw(...)** και την **Updateable** κάνοντας την να μπορεί να ενημερωθεί μέσω της **update(...)**.

Στην κλάση υπάρχουν 2 μεταβλητές που ελέγχουν το αντικείμενο. Το μέλος **visible** είναι το flag μέσω του οποίου όταν καλεστεί η **draw(...)** θα ξέρουμε αν θα πρέπει να σχεδιάσουμε το Animation ή όχι. Το μέλος **playing** είναι το flag μέσω του οποίου όταν καλεστεί η **update(...)** θα ξέρουμε αν θα πρέπει να ενημερώσουμε το Animation. Όπως βλέπουμε, οι πρώτες συναρτήσεις ασχολούνται με την μεταποίηση αυτών των μελών.

Η λειτουργία της αναπαραγωγής βασίζεται στις κλήσεις της **update(...)**. Αυτή η συνάρτηση πάντα όταν καλείται, παίρνει σαν όρισμα τον χρόνο που πέρασε σε δευτερόλεπτα από την προηγούμενη κλήση της. Ξέροντας αυτόν τον χρόνο κάθε φορά, μπορούμε με ακρίβεια να εναλλάσσουμε τα Cells (Frames) του Animation όποτε πρέπει. Όταν ξεκινάει η αναπαραγωγή του Animation, παίρνεται το πρώτο του Frame και η τιμή του **delay** του μπαίνει στο μέλος **frameTimer**. Αυτό το μέλος πάντα έχει μέσα του τον χρόνο σε δευτερόλεπτα που απομένει μέχρι να πρέπει να αλλάξει το Frame του Animation και να μπει το επόμενο. Η αρχική τιμή του πρώτου Frame μπαίνει όταν τεθεί το Animation στο AnimationPlayer. Θέλουμε όταν περάσει το **delay** του frame που έχουμε να αλλάξει στο επόμενο. Έτσι όταν καλεστεί η

update(...), ο χρόνος που υπάρχει στο **frameTimer** θα μειώνεται με τιμή που είναι ο χρόνος που έχει όρισμα η συνάρτηση. Έτσι η τιμή του **frameTimer** θα έχει πάντα τον χρόνο που απομένει για το frame που έχουμε. Όταν αυτή η τιμή φτάσει στο 0, τότε ξέρουμε ότι πρέπει να αλλάξουμε στο επόμενο frame.

Πίνακας 3.2.2: Η συνάρτηση AnimationPlayer::update()

```
void AnimationPlayer::update(Float elapsedTime) {
    // Check if the animation is playing. We should only update it if
    // it is playing
    if (playing) {
          // subtract the time elapsed from the frame's remaining time
          frameTimer -= elapsedTime;
          // check if the remaining time has reached 0
          if (frameTimer <= 0.0f) {
             // if there is no remaining time, then we should switch to
           // Animation's next frame. Before we do that, we should check
                // if we reached beyond the last frame.
                if (++frameIndex >= animation->getNumFrames()) {
                      // If we passed the last frame, check if
                      // the animation loops
                      if (!animation->isLooping()) {
                            // If it does not loop, then stop it
                            playing = false;
                         // we should decrease the frameIndex because it
                         // points beyond the last frame. So it should
                            // point to last.
                            --frameIndex;
                      } else {
                           // if the animation should loop, then set the
                            // frameIndex to the first frame
                            frameIndex = 0;
                      }
                }
                // set the frame remaining time to the new frame's delay
                frameTimer = animation->getFrame(frameIndex).getDelay();
          }
    }
}
```

Το frame που έχουμε είναι στην ουσία η τιμή του μέλους **frameIndex**. Αυτός ο αριθμός είναι δείκτης του πίνακα των frames(Cells) του Animation που έχουμε. Έτσι για να πάμε στο επόμενο frame απλά τον αυξάνουμε. Πρέπει να προσέξουμε να δούμε αν υπερβήκαμε τον αριθμό των Frame που έχει το Animation με αυτή την αύξηση. Αν έγινε αυτό, τότε σημαίνει ότι ολοκληρώσαμε το τελευταίο Frame του Animation, που άμεσα σημαίνει ότι το Animation τελείωσε. Στην περίπτωση αυτή πρέπει να δούμε αν θα σταματήσουμε το Animation ή θα το επαναλάβουμε από την αρχή. Αυτό καθορίζεται από αν το Animation έχει το flag **loops** true. Η τιμή αυτή

εξαρτάται από το πώς φτιάχτηκε το αρχείο του Animation και δεν αλλάζει. Δηλαδή αν επαναληφθεί, τότε θα επαναλαμβάνεται συνέχεια μέχρι να σταματήσει ρητά μέσω της κλήσης **stop()** ή **pause()** που θα αποτρέψουν το Animation να ενημερωθεί περαιτέρω. Αν το Animation πρέπει να επαναληφθεί, τότε το **frameIndex** γίνεται 0 για να δείξει στο πρώτο Frame. Αν δεν πρέπει να επαναληφθεί, τότε κάνουμε το **frameIndex** να δίνει στο τελευταίο frame που παίχτηκε και παγώνουμε το Animation με το να το αποτρέπουμε να ενημερωθεί. Η αποτροπή αυτή γίνεται θέτοντας το μέλος **playing** σε false. Αφού τελικά έχουμε την τιμή του επόμενου frame στο μέλος **frameIndex** τότε θέτουμε τον χρόνο που θα διανύσει το νέο Frame στο μέλος **frameTimer** από την τιμή **delay** του νέου Frame.

Με την παραπάνω λειτουργία υλοποιείται η ενημέρωση του Animation μέσω της **update(...)**. Πέρα από αυτό η κλάση όπως είπαμε έχει αρμοδιότητα και να σχεδιάζει το Animation που έχει. Αυτό γίνεται μέσω της **draw(...)**.

Πίνακας 3.2.3: Η συνάρτηση AnimationPlayer::draw()

```
void AnimationPlayer::draw(Float x, Float y, Float rotation,
          Float scale) const {
     // Check if the animation should be drawn
     if (visible) {
          //  Draw the Animation's frame pointed by the frameIndex
          animation->getFrame(frameIndex).draw(x, y, rotation, scale);
     }
}
```

Η προϋπόθεση για να γίνει η σχεδίαση είναι να έχει το μέλος **visible** τιμή true. Αυτό για λόγους ελέγχου μέσω των πρώτων συναρτήσεων της κλάσης. Εφόσον πρέπει να σχεδιάσουμε τότε παίρνουμε από το Animation που έχουμε, το Frame που δείχνει το **frameIndex** και το σχεδιάζουμε περνώντας όλα τα ορίσματα που μας δόθηκαν χωρίς να χρειάζεται να τα μεταποιήσουμε.

3.3 Τα αντικείμενα AnimationHandler

Ένα Lifeform χρειάζεται κάτι παραπάνω από έναν απλό **AnimationPlayer** για διαχειριστούν τα Animations τους. Χρειαζόμαστε να μπορούμε να έχουμε παραπάνω από ένα Animation και να μπορούμε να ξέρουμε σε τι σειρά θα εμφανιστούν. Σε αυτά βοηθάει η κλάση **AnimationHandler**.

Πίνακας 3.3.1: Η κλάση AnimationHandler

```
class AnimationHandler: public Drawable, public Updateable, public
LuaPushable {
public:
     enum class QueuePlace {
          Front, Back
     };

     const AnimationClass& getAnimationClass() const;
     bool isReleaseResources() const;
     void setReleaseResources(bool releaseResources);
     AnimationPlayer& getMainAnimationPlayer() const;

     void setAnimation(const std::string& name,
                LifeformState::CombinedState nonExist =
                                   LifeformState::Action::Idle
                              | LifeformState::Direction::Down);

     void setAnimation(LifeformState::CombinedState name);
     void update(Float elapsedTime);
     void draw(Float x, Float y, Float rotation = 0.0f,
                Float scale = 1.0f) const;
     void setAnimationClass(const AnimationClass& animClass);
     void setAnimationClass(const std::string& path);
     void addOffAnimation(const Animation& animation, Float dx = 0.0f,
                Float dy = 0.0f, QueuePlace queue = QueuePlace::Front);
     OffAnimation& getOffAnimation(const std::string& name,
                              QueuePlace queue = QueuePlace::Front);

     void pushToLua(lua_State* lstate);
     AnimationHandler();
     AnimationHandler(const AnimationClass& animClass);
     AnimationHandler(const std::string& classPath,
                const std::string& animation = "");
     ~AnimationHandler();
private:
     ZSmallMap<std::string, OffAnimation> frontalAnimations;
     ZSmallMap<std::string, OffAnimation> backAnimations;
     mutable AnimationPlayer mainAnimation;
     const AnimationClass *animClass;
     bool releaseResources;
};
```

Ο πυρήνας της κλάσης είναι το μέλος **mainAnimation** που είναι ένα **AnimationPlayer** που θα φιλοξενεί το κύριο **Animation** του Lifeform. Μετά υπάρχουν οι ομάδες(**queues**) των Animations που θα σχεδιαστούν μπροστά (**frontalAnimations**) και αυτών που θα σχεδιαστούν πίσω (**backAnimations**). Επίσης εδώ φιλοξενείται και η **AnimationClass** που θα χρησιμοποιεί το Lifeform.

Οι συναρτήσεις **setAnimation(...)** είναι απλά wrappers των αντίστοιχων της AnimationPlayer και καλούν τις αντίστοιχες για το μέλος **mainAnimation**. Η

update(...) καλεί την αντίστοιχη για το κύριο Animation καθώς και τα OffAnimation. Η σειρά που θα γίνει δεν έχει σημασία. Αυτό που έχει σημασία είναι εάν κάποιο από τα OffAnimations τελείωσε και δεν επαναλαμβάνεται. Αν γίνει αυτό, τότε πρέπει να αφαιρεθεί από την ομάδα.

Πίνακας 3.3.2: Η συνάρτηση AnimationHandler::update()

```
void AnimationHandler::update(Float elapsedTime) {
     // Update the main Animation
     mainAnimation.update(elapsedTime);
     // for all the front Animations
     for (auto itr = frontalAnimations.begin();
                 itr != frontalAnimations.end();
                 ) {
           // Update the Animation
           itr->second.update(elapsedTime);
           // Check if it is still playing
           if (itr->second.isPlaying()) {
                 // If it is still playing, then get the next one
                 ++itr;
           } else {
              // if it does not playing, then remove it, since it is over
                 itr = frontalAnimations.erase(itr);
           }
     }
     // for all the front Animations
     // Το ίδιο με το παραπάνω ...
}
```

Στην **draw(...)** γίνεται το ίδιο, μόνο που εδώ η σειρά έχει σημασία. Τα **backAnimations** πρέπει να φαίνονται πίσω από το κύριο για αυτό και θα πρέπει να σχεδιαστούν πρώτα. Ύστερα σχεδιάζεται το κύριο. Μετά τα **frontalAnimations** καθώς πρέπει να φαίνονται μπροστά από το κύριο.

Η επόμενη συνάρτηση που θα δούμε είναι η **addOffAnimation(...)**. Αυτή έχει ως σκοπό να δημιουργήσει ένα αντικείμενο **OffAnimation** από το Animation που μας δίνεται και να το τοποθετήσει στην αντίστοιχη ομάδα σχεδίασης αφού εφαρμοστούν τα Offsets.

Πίνακας 3.3.3: Η συνάρτηση AnimationHandler::addOffAnimation()

```
void AnimationHandler::addOffAnimation(const Animation& animation,
                              Float dx, Float dy, QueuePlace queue) {
     // Construct an Empty OffAnimation
     OffAnimation anim;
     // Set the Animation of the OffAnimation
     anim.setAnimation(animation);
     // Set the offsets
```

```
        anim.setOffsets(dx, dy);

        // Check which queue we should add it
        switch (queue) {
        case QueuePlace::Front:
              frontalAnimations[animation.getName()] = std::move(anim);
              break;
        case QueuePlace::Back:
              backAnimations[animation.getName()] = std::move(anim);
              break;
        }
}
```

Με τον τρόπο που ελέγχεται για ποια ομάδα να προστεθεί το OffAnimation, γίνεται η αναζήτηση στην συνάρτηση **getOffAnimation(...)**. Εκεί αν το όνομα που αναζητείται δεν υπάρχει στην ομάδα που ψάχνεται, τότε επιστρέφεται το **NullAnimation**.

Το flag **releaseResources** ελέγχεται αν είναι true κάθε φορά που αλλάζει το **AnimationClass**. Αν είναι true, τότε το παλιό θα αποδεσμευτεί μέσω του **ResourcesContext**. Ο ίδιος έλεγχος γίνεται και στον καταστροφέα.

3.4 Τα αντικείμενα OffAnimation

Στο προηγούμενο κεφάλαιο είδαμε ότι τα Animation που θα χρησιμοποιηθούν ως δευτερεύοντα στο κύριο, δημιουργούν αντικείμενα **OffAnimation** όπου και περνιούνται εκεί. Στην πραγματικότητα είναι **AnimationPlayer**s με μερικά επιπλέον δεδομένα. Αυτά τα επιπλέον δεδομένα είναι κάποια Offsets που θα εφαρμοστούν κατά την σχεδίαση. Η ανάγκη των Offset αυτών είναι για να μετακινούμε κάθε δευτερεύον Animation σε σχέση με το κύριο.

Πίνακας 3.4.1: Η κλάση OffAnimation

```
class OffAnimation: public AnimationPlayer {
public:
     Float getDx() const
     void setDx(Float dx);
     Float getDy() const;
     void setDy(Float dy);
     void setOffsets(Float dx, Float dy);

     void draw(Float x, Float y, Float rotation = 0.0f,
                 Float scale = 1.0f) const {
           AnimationPlayer::draw(x + dx, y + dy, rotation, scale);
     }

     OffAnimation() :
                 AnimationPlayer(), dx(0.0f), dy(0.0f) {
```

```
    }

    OffAnimation(const Animation& animation) :
                AnimationPlayer(animation), dx(0.0f), dy(0.0f) {
    }

    ~OffAnimation() {

    }

private:
    Float dx;
    Float dy;
};
```

Όπως βλέπουμε, η κλάση κληρονομεί την **AnimationPlayer** και κάνει Override την **draw(...)**. Στην νέα συνάρτηση καλούμε την μητρική προσθέτοντας τα επιπλέον Offset στα offset που έχουμε πάρει από τον καλών για να έχουμε το αποτέλεσμα που θέλουμε.

4 ΤΟ ΣΥΣΤΗΜΑ ΧΑΡΤΩΝ (MAP)

4.1 Ο χάρτης Map

Τώρα θα δούμε το μεγαλύτερο κομμάτι του λειτουργικού κομματιού του πυρήνα της μηχανής, τον χάρτη. Ο χάρτης είναι ο κόσμος που διαδραματίζονται τα πάντα. Είναι μια δομή που περιέχει εσωτερικά όλα τα δεδομένα εμφάνισης του κόσμου, στατικά αντικείμενα και Lifeforms. Πέρα από την φιλοξενία αυτών τον δεδομένων, ο χάρτης αναλαμβάνει να ενημερώσει (**update**) τα αντικείμενα όποτε χρειάζεται και να τους δώσει διαταγή να σχεδιαστούν όπου χρειάζεται. Για αυτό και ο χάρτης είναι από τις πιο πολύπλοκες δομές της μηχανής.

Πίνακας 4.1.1: Η κλάση Map

```
class Map: public Resource, public Updateable, public LuaPushable {
public:

     const PropertyList& getProperties() const;
     unsigned int getHeight() const;
     unsigned int getTileHeight() const;
     unsigned int getTileWidth() const;
     unsigned int getWidth() const;
     void setView(View &camera);
     const Tile& getTileByGID(unsigned int gid) const;
     bool isRectangleColliding(const Rectangle& rect, Object *owner,
           Rectangle::Surface surface = Rectangle::Surface::Every) const;

     void setCordinateColliding(Float x, Float y, bool val);
     Object* getObjectAtPosition(Float x, Float y,
                                       Object* ignoreThis = nullptr);

     std::vector<Object*> getAllObjectsAtPosition(Float x, Float y,
                 Object* ignoreThis = nullptr);
     const ZSmallSet<Object*>& getVisibleObjects() const;
     void addTileToUpdate(Tile* tile) const;
     void load(const std::string& path);
     void nullise();
     void clearCachedTilesets();
     void addObject(Object *obj, bool toBeDeleted);
     void insertObject(Object *obj, bool toBeDeleted);
     void removeObject(Object *obj);
     void draw(const View& camera) const;
     void update(Float elapsedTime);
     void pushToLua(lua_State* lstate);
     Map();
     Map(const std::string& path);
     ~Map();

protected:
     ZSmallMap<std::string, SharedResource<Tileset>> cachedTilesets;
     std::vector<MapTileset> tilesets;
```

```
        ZSmallMap<std::string, Layer*> layers;
        std::vector<Layer*> upperLayers;
        std::vector<Layer*> lowerLayers;

        std::list<ObjectPair> objects;
        std::list<Object*> visibles;
        ZSmallSet<Object*> inView;
        mutable ZSmallSet<Tile*> tilesToBeUpdated;
        ZSmallMap<Object*, std::list<ObjectPair>::iterator> insertedObjects;
        std::vector<std::list<ObjectPair>::iterator> toBeDeleted;

        Vector2D<int> collisions;
        PropertyList properties;
        StaticQuadtree quadtree;

        View *camera;

        unsigned int width;
        unsigned int height;

        unsigned int tileWidth;
        unsigned int tileHeight;

        unsigned int subWidth;
        unsigned int subHeight;
        unsigned int subTileWidth;
        unsigned int subTileHeight;
        int tilesetLifeSpan;

        void getProperties(xmlpp::Node* rootNode);
        void getTilesets(xmlpp::Node* rootNode);
        void getLayers(xmlpp::Node* rootNode);
        void parsePriorities(xmlpp::Node* rootNode);
        void getInteractFields(xmlpp::Node* rootNode);
        void getStaticObjects(xmlpp::Node* rootNode);
        void getNpcsAndEnemies(xmlpp::Node* rootNode);
        void loadCollisions();
        void clear();

};
```

Το βασικό στοιχείο της κλάσης είναι τα **Layers**. Αυτά είναι πίνακες από αναφορές σε **Tiles** που βρίσκονται σε **Tilesets**. Τα Tilesets αποθηκεύονται στον πίνακα **tilesets** που περιέχει στοιχεία **MapTileset** που θα δούμε παρακάτω. Όλα τα Layers βρίσκονται στο HashTable **layers** συνδεδεμένα με το όνομα που τους δόθηκε στο αρχείο. Τα αντικείμενα του χάρτη αποθηκεύονται στο **objects**, εκ τον οποίων αυτά που βρίσκονται εντός οθόνης θα υπάρχουν και στο **inView**. Τα ίδια στοιχεία που θα βρίσκονται εκεί, θα βρίσκονται και στον **visibles** με την διαφορά ότι θα είναι ταξινομημένα με βάση την συντεταγμένη **y** που έχουν. Η λίστα των **tilesToBeUpdated** θα έχει τα Tiles που θα έχουν Animation και πρέπει να ενημερωθούν κάθε φορά που θα καλείται η **update()** στον χάρτη. Ο δισδιάστατος πίνακας **collisions** περιέχει κάθε

στατική σύγκρουση στον χάρτη. Τα υπόλοιπα μέλη περιέχουν δεδομένα που χρησιμοποιεί ο χάρτης.

Η περαιτέρω ανάλυση της κλάσης θα γίνεται σταδιακά. Αρχικά θα αναλύσουμε την φόρτωση.

4.2 Η φόρτωση

Η φόρτωση γίνεται μέσω της συνάρτησης **load(...)**.

Πίνακας 4.2.1: Η συνάρτηση Map::load()

```
void Map::load(const std::string& path) {

    // Clear the map
    clear();
    // set the name from the file path
    name = path;
    // set the camera
    camera = &WorldManager::getInstance().getView();
    // insert the player on the inView
    inView.insert(&WorldManager::getInstance().getPlayer());

    /*
    * Κώδικας για φορτωση του XML
    */

    // If execution gets here, then the File is OK for iteration
    xmlpp::Element *root = parser.get_document()->get_root_node();
    // Get the map dimensions
    width = strTo10(root->get_attribute_value("width").c_str());
    height = strTo10(root->get_attribute_value("height").c_str());
    // Get the primary tile dimensions
    tileWidth = strTo10(root->get_attribute_value("tilewidth").c_str());
    tileHeight = strTo10(
                      root->get_attribute_value("tileheight").c_str());
    subTileWidth = tileWidth >> 1;
    subTileHeight = tileHeight >> 1;

    // Set the Quad Tree bounds
    quadtree.setBounds(0, 0, width * tileWidth, height * tileHeight);
    // The names say it all
    getProperties(root);
    getTilesets(root);
    getLayers(root);
    parsePriorities(root);
    getInteractFields(root);
    getStaticObjects(root);
    getNpcsAndEnemies(root);
    loadCollisions();
}
```

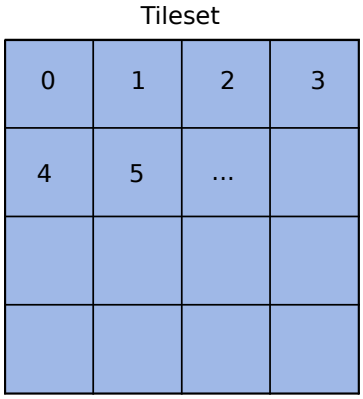
Αρχικά καθαρίζουμε τον χάρτη μέσω της **clear()**. Αυτή καθαρίζει κάθε μέλος της Map και φορτώνει τις default τιμές σε αυτά. Αυτό χρειάζεται γιατί η load() μπορεί να καλεστεί αφού έχει ήδη φορτωθεί ένας χάρτης. Στην συνέχεια παίρνουμε την κάμερα και τον παίχτη. Το επόμενο είναι να κάνουμε την συναφή δουλειά να φορτώσουμε το αρχείο, να το κάνουμε Parse και Validate. Ύστερα παίρνουμε τα βασικά στοιχεία του χάρτη από το πρώτο στοιχείο του αρχείου (width, height κτλ). Στην συνέχεια θέτουμε τα όρια του Quadtree μας. Πρέπει να είναι όσο είναι και ο χάρτης και να αρχίζει εκεί που αρχίζει ο χάρτης δηλαδή στο (0,0).

Μετά σε σειρά ανακτώνται μέσω των συναρτήσεων τα δεδομένα. Από αυτά θα αναλύσουμε την ανάκτηση των Layers, Tilesets και όλων των αντικειμένων.

α) Τα Tilesets

Η συνάρτηση **getTilesets()** έχει σκοπό να φορτώσει κάθε Tileset που έχει αναφορά στο αρχείο του χάρτη, καθώς και να τα κρατήσει σε cache. Τα Tilesets που γίνονται αναφορά μπαίνουν σε νοητό ενιαίο Tileset όπου κάθε Tile παίρνει ένα μοναδικό αριθμό GID (Global ID), μέσω του οποίου γίνεται προσβάσιμο από τα Layers. Με άλλα λόγια, εάν έχουμε 2 Tilesets, το A με 30 Tiles και το B με 50, τότε το πρώτο Tile του A θα πάρει GID=1, τα ενδιάμεσα του θα πάρουν τιμές από 1 μέχρι 30 και ύστερα θα ξεκινάνε τα Tiles του B με το πρώτο του να έχει GID=31 κοκ. Για αυτό και έχουμε την βοηθητική κλάση **MapTileset**, όπου φιλοξενεί το Tileset καθώς και το GID του πρώτου του Tile.

Όπως και με τα **SpriteSheets**, έτσι και τα **Tilesets** έχουν ανάλογη λειτουργία, μόνο που κάθε **Tile** αναπαρίσταται με ένα ID. Αυτός ο αριθμός ορίζει την ταυτότητα του και το που βρίσκεται πάνω στο Tileset. Τα Tiles πάνω στο Tileset είναι διαδοχικά ID οριζοντίως, μέχρι το τέλος και μετά συνεχίζει στην κάτω γραμμή κοκ.



Εικόνα 4.2.1: Παράδειγμα απόδοσης ID σε Tileset

Με τον παραπάνω τρόπο σπάει η μητρική εικόνα του Tileset σε Tiles και τους αποδίδεται εικονικά το ID. Εικονικά το λέμε γιατί δεν δίνουμε στην πραγματικότητα την τιμή καθώς είναι πάντα στάνταρ τι τιμή θα έχει κάθε Tile εντός του Tileset.

Πίνακας 4.2.2: Η κλάση Tileset

```
class Tileset: public Resource {
public:
     const Tile& getTile(unsigned int id) const;
     const Tile* getMapTile(unsigned int id) const;
     void load(const std::string& path);
     Tileset();
     Tileset(const std::string& path);
     virtual ~Tileset();
protected:
     std::vector<Tile> tiles;
     const Bitmap *img;
     int tileWidth;
     int tileHeight;
     unsigned int numTiles;
};
```

Όλα τα Tiles βρίσκονται σε πίνακα και η ανάκτηση τους γίνεται με το ID τους μέσω των συναρτήσεων **getTile(...)** και **getMapTile(...)**. Η διαφορά αυτών των δυο είναι ότι στην πρώτη αν το ID δεν είναι έγκυρο, τότε θα επιστραφεί **NullTile** ενώ στην δεύτερη θα επιστραφεί NULL pointer. Η ανάκτηση ενός Tile μέσω αυτών των συναρτήσεων γίνεται με το ID που είναι δείκτης εντός του πίνακα **tiles**. Τα tiles μέσα σε αυτόν αντιστοιχούν όπως στο σχεδιάγραμμα 4.2.1. Δηλαδή στο σχεδιάγραμμα αυτό, για να πάρουμε το tile που βρίσκεται στην δεύτερη γραμμή και πρώτη στήλη, πρέπει να καλέσουμε την συνάρτηση με ID 4.

Αυτό μου μας νοιάζει να δούμε εδώ είναι η φόρτωση του Tileset. Αυτό γίνεται μέσω της **load(...)**. Η φόρτωση αναλαμβάνει πέρα από το σπάσιμο της μητρικής εικόνας σε Tile, να δημιουργήσει τυχών Tiles που έχουν Animation και να τους ορίσει τι είδους σύγκρουση έχουν. Τα Animated Tiles, μπορούν να έχουν μια ιδιότητα που τους ορίζει συγγενικά Animated Tiles.

Ο λόγος ύπαρξης των συγγενικών Animated Tiles, είναι ότι πολλά Animated Tiles είναι συνήθως μαζί με κάποια άλλα που μαζί ολοκληρώνουν ένα κομμάτι, πχ ξηράς. Μια κλασική περίπτωση είναι της ξηράς που καταλήγει σε θάλασσα όπως είδαμε στο δεύτερο μέρος. Εκεί είχαμε πολλά Tiles που σχημάτιζαν κάτι ενιαίο. Ο λόγος που ήταν έτσι μέσα στο Tileset ήταν γιατί στην μορφή αυτή μπορεί εύκολα να γίνει κατανοητό που πρέπει να τοποθετείται κάθε Tile και να το φτιάξουμε σαν Terrain στο ερ-

γαλείο Tiled. Τα Animated Tiles λόγω ότι είναι Animated, χρειάζεται να γίνονται **update(...)** από τον Map, όποτε πρέπει. Αυτό γίνεται μόνο όταν το Tile βρίσκεται εντός της οθόνης. Έτσι αν έχουμε κινηθεί και εμφανίστηκε ένα μικρό μέρος της όχθης στο τελευταίο Frame, τότε στο επόμενο Frame τα Tiles της όχθης θα ενημερωθούν και θα τρέξει το Animation κανονικά. Αυτό θα γίνει μόνο για τα Tiles που εμφανίστηκαν. Αν όμως κινηθούμε και εμφανιστούν Tiles που πριν δεν υπήρχαν, πχ κάποιο Tile της γωνίας της όχθης που πριν η όχθη δεν είχε γωνίες, τότε αυτό το Tile της γωνίας θα ενημερωθεί στο επόμενο Frame. Εδώ υπάρχει ένα πρόβλημα. Αυτό το Tile θα βρίσκεται πάντα πίσω όσο αφορά το Animation σε σχέση με τα άλλα που εμφανίστηκαν νωρίτερα. Έτσι το Animation της γωνίας θα δεν θα είναι σε συγχρονισμό με τα άλλα όπως θα έπρεπε και θα φαίνεται άσχημο στον χρήστη. Για αυτό και χρειάζεται να υπάρχει αυτός ο συγχρονισμός σε αυτά τα Tiles που τον χρειάζονται για να φαίνονται καλά. Σε αυτό βοηθάει η ιδιότητα των συγγενικών Tiles. Αυτή η ιδιότητα μπαίνει σε κάθε Animated Tile που χρειάζεται να είναι συγχρονισμένο με άλλα. Όλα τα Tiles που πρέπει να είναι συγχρονισμένα μεταξύ τους πρέπει να την φέρουν και να έχει την ίδια τιμή, η οποία είναι όλα τα ID των Tile που πρέπει να συγχρονιστούν, χωρισμένα από κόμματα.

Ας πάμε να δούμε τα σημαντικότερα σημεία της συνάρτησης της φόρτωσης.

Πίνακας 4.2.3: Η συνάρτηση Tileset::load() μέρος 1

```
void Tileset::load(const std::string& path) {
    /*
    * Κώδικας Parsing, Validation, ανάκτησης πεδίων tileWidth,
    * tileHeight και φόρτωσης μητρικής εικόνας απο Resources Context */
    // Calculate the tiles that exist in every axis
    int numTiles_W = img->getWidth() / tileWidth;
    int numTiles_H = img->getHeight() / tileHeight;
    // Calculate the number of tiles
    numTiles = numTiles_W * numTiles_H;
    // Reserver the subBitmaps
    img->reserveSubBitmaps(numTiles + 1);
    tiles.resize(numTiles + 2);

    unsigned int index = 0;
    // for all the Tiles on both axis
    for (int i = 0; i < numTiles_H; ++i) {
          for (int j = 0; j < numTiles_W; ++j) {
                // Get the tile from the calculated coordinates
                tiles[index].set(
                      img->createSubBitmap(j * tileWidth,
                                           i * tileHeight,
                                       tileWidth, tileHeight), index);
                ++index;
          }
    }
```

Στο πρώτο μέρος της συνάρτησης, κάνουμε ότι κάνουμε με κάθε αρχείο XML, parse, Validate και αφού περάσουμε το στάδιο του Validate, παίρνουμε τα βασικά δεδομένα από τα Attributes. Στην συνέχεια υπολογίζουμε πόσα tiles θα έχει κάθε διάσταση. Αυτό θα το κάνουμε για να μπορούμε να ξέρουμε σε ποια σημεία θα σπάσουμε την εικόνα σε Tiles καθώς αυτά είναι τοποθετημένα σε μορφή δισδιάστατου πίνακα. Μέσα στην επανάληψη, θέτουμε στο αντικείμενο **Tile** στον δείκτη **index** μια εικόνα που την παίρνουμε σπάζοντας την μητρική μέσω της **createSubBitmap(...)** και θέτοντας τις συντεταγμένες και το μέγεθος του ορθογωνίου που θα πάρει τα δεδομένα. Στο τέλος αυξάνουμε το **index** για να θέσουμε το επόμενο Tile στην επόμενη επανάληψη.

Ας πάνε εν συντομία να δούμε την κλάση **Tile**.

Πίνακας 4.2.4: Η κλάση Tile

```
class Tile: public Drawable, public Updateable {
public:
     enum Collision {
           None = 0x0,
           N = 0x3,
           E = 0xA,
           S = 0xC,
           W = 0x5,
           NW = 0x1,
           NE = 0x2,
           SE = 0x8,
           SW = 0x4,
           NxW = 0x7,
           NxE = 0xB,
           SxE = 0xE,
           SxW = 0xD,
           NWxSE = 0x9,
           SWxNE = 0x6,
           Full = 0xF
     };

     struct TileFrame {
           Tile* tile;
           Float uptime;
     };

     inline unsigned int getID() const;
     void setCollision(Collision coll);
     inline Collision getCollision() const;
     void addFrame(TileFrame& frame);
     void addRelativeTile(Tile& other);
     void setAnimated(bool value);
     inline bool isAnimated() const;
     inline const std::vector<Tile*>& getRelativeTiles() const;
     void draw(Float x, Float y, Float rotation = 0.0f,
                 Float scale = 1.0f) const;
```

```
	void set(const Bitmap& bmp, unsigned int id) {
		img = &bmp;
		this->id = id;
	}

	inline void update(Float elapsedTime) {
		frameCounter -= elapsedTime;
		if (frameCounter <= 0.0f) {
			if (++frameIndex >= frames.size()) {
				frameIndex = 0;
			}
			frameCounter = frames[frameIndex].uptime;
		}
	}

	Tile();
	Tile(const Bitmap& bmp, unsigned int id);
	virtual ~Tile();
protected:
	std::vector<TileFrame> frames;
	std::vector<Tile*> relativeTiles;
	const Bitmap *img;
	unsigned int id;
	Collision collision_value;

	unsigned int frameIndex;
	Float frameCounter;
	bool animated;

	friend class Tileset;
};
```

Η κλάση είναι απλή. Περιέχει αρκετά μέλη πεδία τα οποία κυρίως φιλοξενούνται. Το βασικό στοιχείο εδώ είναι το μέλος **img** όπου θα είναι η εικόνα που θα σχεδιάζει το Tile όταν του ζητηθεί. Πέρα από αυτό, εξίσου σημαντικό είναι και τα πεδία **id** και **collision_value**. Το πρώτο απλά είναι το ID του Tile που έχει εντός του Tileset και το δεύτερο φιλοξενεί την τιμή της σύγκρουσης που θα έχει το Tile στον κόσμο. Οι τιμές που μπορεί να πάρει το στοιχείο αυτό είναι από το Enumeration **Collision**. Για αυτό θα δούμε παρακάτω.

Τα υπόλοιπα πεδία έχουν να κάνουν στην περίπτωση που το Tile είναι Animated. Τα πεδία **frames** και **relativeTiles** γεμίζουν από άλλα στοιχεία της μηχανής. Τα πεδία **frameIndex** και **frameCounter** έχουν να κάνουν με την συνάρτηση **update(...)** όπου και ενημερώνουν το Animation (σε περίπτωση που υπάρχει).

Η συνάρτηση **set(...)** που χρησιμοποιούμε κατά την φόρτωση όπως βλέπουμε κάνει απλά ανάθεση τιμών. Για αυτό και επιστρέφοντας πίσω στην **load(...)** που εξηγούσαμε πριν, στην επανάληψη απλά θέτουμε την εικόνα που αποσπάσαμε και το id στο Tile που έχουμε.

Στο επόμενο στάδιο της φόρτωσης, παίρνουμε όλα τα Tiles που έχουν Animation και τους τα φορτώνουμε από εδώ.

Πίνακας 4.2.5: Η συνάρτηση Tileset::load() μέρος 2

```
// Find all the animation nodes
xmlpp::NodeSet props = root->find("//animation");

// for all the Nodes
for (auto node : props) {

     // Get the tile that has this ID
     Tile& tile = tiles[strTo10(
                 node->get_parent()->get_attribute_value("id").c_str())];

     // Set that this Tile is animated
     tile.setAnimated(true);

     // Get all the frame nodes
     xmlpp::NodeSet frameNodes = node->find("./frame");

     // For all the frame nodes
     for (auto frameNode : frameNodes) {
       xmlpp::Element* element = static_cast<xmlpp::Element*>(frameNode);

          // Create a Tile frame
          Tile::TileFrame frame;

          // Get the tile that this Frame will have
          frame.tile = &tiles[strTo10(
                      element->get_attribute_value("tileid").c_str())];
          // Get the delay of this frame
          frame.uptime = strTo10(
                      element->get_attribute_value("duration").c_str())
                                  / 1000.0f;

          // Add the frame to the tile
          tile.addFrame(frame);
     }
}
```

Αρχικά παίρνουμε όλα τα στοιχεία <animation> καθώς έτσι θα ξέρουμε ακριβώς ποια στοιχεία θα επεξεργαστούμε. Ύστερα για όλα αυτά τα στοιχεία, παίρνουμε το **Tile** που ορίζει το στοιχείο που έχουμε και του θέτουμε ότι θα είναι Animated. Ύστερα πρέπει να βρούμε όλα τα στοιχεία-παιδιά <frame> για να του κατασκευάσουμε τα Frames. Αφού τα βρούμε, τότε για καθένα από αυτά κατασκευάζουμε το Frame, παίρνοντας το Tile που θα έχει το Frame και τον χρόνο που θα διαρκέσει. Ο χρόνος που θέτει το εργαλείο Tiled είναι σε msec, για αυτό και τον διαιρούμε προς 1000 για να πάρουμε τον χρόνο σε δευτερόλεπτα που χρησιμοποιεί η μηχανή. Στο τέλος αφού

έχουμε έτοιμο το Frame και το προσθέτουμε την λίστα του Tile. Το ίδιο κάνουμε και για τα υπόλοιπα.

Στην τελευταία φάση της φόρτωσης, χρειάζεται να πάρουμε την τιμή της σύγκρουσης που θα έχει κάθε Tile, καθώς και να συνδέσουμε τα Tiles που χρειάζονται να είναι συγχρονισμένα λόγω ότι είναι συγγενικά (μόνο αυτά που είναι Animated).

Πίνακας 4.2.6: Η συνάρτηση Tileset::load() μέρος 3

```
// get tall the properties Nodes
props = root->find("//properties");
// For all the nodes
for (auto node : props) {
     // Get the Tile that is referenced
     Tile& tile = tiles[strTo10(
                 node->get_parent()->get_attribute_value("id").c_str())];

     // Get the Collision Property
     xmlpp::NodeSet propsNodes =
                       node->find("./property[@name='Collision']");

     // If the property is found
     if (propsNodes.size() > 0) {

           // Get the collision Value
           int coll =
                 strTo10(
                       static_cast<xmlpp::Element*>(
                 propsNodes[0])->get_attribute_value("value").c_str());
           tile.setCollision((Tile::Collision) coll);
     }

     // Check if the tile is animated
     if (tile.isAnimated()) {

           // Check for Relative tiles
           propsNodes = node->find("./property[@name='Relatives']");

           // Check if there is any relatives
           if (propsNodes.size() > 0) {

                 // We have to tokenize the value
                 std::list<std::string> tokens;

                 // Tokenize with the , value
                 StringOperations::getTokens(
     static_cast<xmlpp::Element*>(
           propsNodes[0])->get_attribute_value("value"), ',', tokens);
                 // For all the Tokens
                 for (auto& token : tokens) {
                       // Add to the Tile the Relative
                       unsigned int index = strTo10(token.c_str());
                       if (index <= tiles.size()) {
                             tile.addRelativeTile(tiles[index]);
                       }
```

```
                }
            }
        }
```

Όπως και πριν, παίρνουμε το Tile που αναφέρει το στοιχείο. Αναζητούμε εάν έχει κάποιο στοιχείο με τιμή “Collision” στο attribute **name** και εφόσον έχει, τότε θέτουμε την τιμή του σαν την τιμή της σύγκρουσης του Tile.

Εάν το Tile που έχουμε είναι Animated, τότε ελέγχουμε αν έχει ιδιότητα με όνομα Relatives. Αν έχει τότε παίρνουμε την τιμή της ιδιότητας αυτής και την σπάμε σε Tokens χωρισμένα από κόμματα. Οι τιμές που θα μας έρθουν είναι τα ID των Tile που θα έχει αυτό το Tile σαν συγγενικά. Έτσι παίρνουμε αυτά τα Tile και προσθέτουμε τους δείκτες του στην λίστα των συγγενικών Tile, αφού πρώτα ελέγχουμε ότι το ID είναι έγκυρο.

Μέχρι στιγμής έχουμε δει πως φορτώνεται ένα Tileset. Κατά την φόρτωση του χάρτη, όπως έχουμε δει τα Tilesets που αναφέρονται φορτώνονται μέσω της συνάρτησης **getTilesets(...)**. Ο σκοπός εδώ είναι να δημιουργήσουμε το ενιαίο Tileset από όλα τα εξωτερικά Tilesets που αναφέρονται. Πέρα από αυτό πρέπει να κρατάμε τα φορτωμένα Tilesets σε cache.

Πίνακας 4.2.7: Η συνάρτηση Map::getTilesets()

```
void Map::getTilesets(xmlpp::Node* rootNode) {
    auto nodes = rootNode->find("./tileset");
    // for all the tilesets
    for (auto node : nodes) {
        // Cast to Element for getting access to Attributes
        xmlpp::Element *tilesetNope =
    static_cast<xmlpp::Element*>(node);

        // Get a temporary Map Tileset
        MapTileset tl;

        // Translate The path of the tileset
        std::string tilesetName = StringOperations::translateUnixPath(
            this->name, tilesetNope->get_attribute_value("source"));

        // Check if we already loaded this tileset before
        auto itr = cachedTilesets.find(tilesetName);

        if (itr != cachedTilesets.end()) {
            // If we loaded before, then get this reference
            tl.setTileset(&itr->second.getInstance());
        } else {
            // If we haven't loaded then we should now
            SharedResource<Tileset> &tmpTl =
    cachedTilesets[tilesetName];
```

```
                // Try and load the tileset
                tmpTl.load(tilesetName);
                if (tmpTl.isValid()) {
                      // Get the reference to the MapTileset
                      tl.setTileset(&tmpTl.getInstance());
                      tmpTl.increaseReference();
                } else {
                      // Nullise the Tileset

    tmpTl.setInstance(NullReference<Tileset>::getNull());
                }
          }
          // Get The first GID
          tl.setFirstGid(
                strTo10(
                tilesetNope->get_attribute_value("firstgid").c_str()));

          // Push the Tileset with the GID
          tilesets.push_back(std::move(tl));
    }

    // We have to cycle through the cached tilesets to remove those
    // that haven't been used for 2 loads
    for (auto itr = cachedTilesets.begin(); itr !=
cachedTilesets.end();) {
          // Check if the tileset is valid (eg. not a NullTileset)
          if (itr->second.isValid()) {
                // Decrease the reference count
                itr->second.decreaseReference();
                // Check if we need to remove it
                if (itr->second.getReferenceCount() < tilesetLifeSpan) {
                      // Report the removal
                      Logger::getInstance().write(
                                  "Released Tileset: "
                            + itr->second.getInstance().getName());
                      // Remove and take the iterator to the next item
                      itr = cachedTilesets.erase(itr);
                      // avoid increasing the iterator
                      continue;
                }
          }
          // Get the next item
          ++itr;
    }
}
```

Αρχικά ψάχνουμε τα στοιχεία <tileset>. Για καθένα από αυτά, πρέπει να δημιουργήσουμε ένα αντικείμενο **MapTileset** που απλά φιλοξενεί τον Pointer του Tileset και το GID. Ο pointer στο Tileset που θα βάλουμε, πρέπει να δούμε αν το έχουμε στο Hashtable **cachedTilesets** και αν είναι εκεί τότε τον αναθέτουμε και προσθέτουμε το αντικείμενο **MapTileset** στον πίνακα **tilesets** που θα είναι ο πίνακας από όπου τα Layers θα έχουν πρόσβαση στα διαθέσιμα Tilesets.

Στο μέλος **cachedTilesets** χρειάζεται να αποθηκεύουμε τα Tilesets που φορτώνουμε και να κρατάμε ένα “βιβλίο” αναφορών όπως και με το ResourcesContext. Θα μπορούσαμε να το κάνουμε κατευθείαν με αυτό, όμως έτσι όπως είναι το σύστημα, κάθε φορά που θα φορτωνόταν ένας χάρτης θα αποδεσμευόταν το Tileset, θα έφευγε από την μνήμη καθώς μόνο ένας χάρτης το χρειάζεται (έναν χάρτη πάντα έχουμε, δεν δείχνουμε 2 χάρτες την στιγμή) και έτσι αν ο επόμενος χάρτης το χρειαζόταν, θα έπρεπε να ξαναφορτωθεί. Για το αποφύγουμε αυτό, το αποθηκεύουμε εσωτερικά στον χάρτη. Έτσι πρέπει να κρατάμε εδώ τον αριθμό των αναφορών, που εδώ λειτουργεί διαφορετικά. Ο αριθμός αυτός λειτουργεί σαν “απομένουσα ζωή” του Tileset. Κάθε φορά που φορτώνεται ένα Tileset, του δίνεται ζωή 2 φορτώσεις πριν καταστραφεί (αυτή η τιμή εξαρτάται από το μέλος tilesetLifeSpan και η προεπιλεγμένη τιμή είναι 2). Αν σε επόμενη φόρτωση το ίδιο Tileset ζητηθεί, τότε θα υπάρχει στο μέλος **cachedTilesets** και θα παραταθεί η ζωή του για μια φόρτωση. Αν όμως στην επόμενη φόρτωση δεν ζητηθεί καθόλου, τότε η απομένουσα ζωή του θα μειωθεί, καθώς μειώνονται οι ζωές όλων των Tileset που υπάρχουν στο **cachedTilesets** σε κάθε φόρτωση. Αν αυτό το Tileset δεν ζητηθεί από χάρτη δεύτερη φορά στην σειρά, τότε η απομένουσα ζωή του φτάνει στο -3 και αυτό σημαίνει ότι σε αυτή την φόρτωση που γίνεται, θα καταστραφεί. Αυτό γίνεται στην τελευταία επανάληψη της συνάρτησης.

β) Τα Layers

Στην επόμενη φάση της φόρτωσης του χάρτη, γίνεται η φόρτωση των Layers. Τα Layers (Στρώσεις) είναι πίνακες δεικτών σε Tiles, που έχουν μέγεθος όσο το μέγεθος του χάρτη σε Tiles. Οι δείκτες αυτοί πρέπει να παρθούν από τα διαθέσιμα Tilesets που έχουν φορτωθεί προηγουμένως στο μέλος **tilesets**.

Πίνακας 4.2.8: Η συνάρτηση Map::getLayers()

```
void Map::getLayers(xmlpp::Node* rootNode) {
     // Find all the Layers
     auto nodes = rootNode->find("./layer");

     // For all the Layers
     for (auto node : nodes) {
           // Cast to Element for getting access to Attributes
           xmlpp::Element *layerNode =
static_cast<xmlpp::Element*>(node);

           // Create a new Layer by passing the node to the constructor
           // as well as the referenced tilesets and the *this (as the
           //  'Father' of the Layer)
```

```
            Layer *lay = new Layer(layerNode, tilesets, *this);
            layers[layerNode->get_attribute_value("name")] = lay;
        }
    }
```

Όπως βλέπουμε εδώ ο κώδικας είναι απλός. Απλά για κάθε στοιχείο <layer> κατασκευάζουμε ένα αντικείμενο **Layer** και το βάζουμε στο hashtable **layers**. Όλος ο λειτουργικός κώδικας της φόρτωσης του Layer βρίσκεται εντός της κλάσης του.

Πίνακας 4.2.9: Η κλάση Layer

```
class Layer {
public:
    void draw(float dx, float dy, int numTilesX, int numTilesY) const;

    const Tile* getTileAt(unsigned int x, unsigned int y) const;

    Layer();
    Layer(xmlpp::Element *layerNode, const std::vector<MapTileset>
                                            &tilesets,Map &parrent);
    ~Layer();
protected:

    Vector2D<const Tile*> tiles;
    Map* parent;
    unsigned int width;
    unsigned int height;

    int tileWidth;
    int tileHeight;

    void nullise();
};
```

Αρχικά θα δούμε την φόρτωση, όπου γίνεται στον δεύτερο κατασκευαστή.

Πίνακας 4.2.10: Ο κατασκευαστής Layer(xmlpp::Element*, const std::vector<MapTileset>&, Map&)

```
Layer::Layer(xmlpp::Element *layerNode, const
std::vector<MapTileset>&tilesets,
            Map &parrent) :
            parent(&parrent) {

    // Get the tile dimensions
    tileWidth = parrent.getTileWidth();
    tileHeight = parrent.getTileHeight();

    // Get the dimensions of the Layer
    width = strTo10(layerNode->get_attribute_value("width").c_str());
    height = strTo10(layerNode->get_attribute_value("height").c_str());
```

```
	// Resize the Tiles table to the Layer Dimensions
	tiles.resize(height, width);

	// Get the Layer Raw Data node
	layerNode = static_cast<xmlpp::Element*>(
	                                    layerNode->find("./data")[0]);

	// The pointer that will contain all the Layer's GID reference
	// Numbers
	unsigned int *gids = nullptr;

	{
		// Get the Data String
		auto data = layerNode->get_child_text()->get_content();
		// We have to clean the RAW data from spaces and line breaks
		int i = 0;
		// For all the character on the string
		for (auto c : data) {
			// check if the char is a whitespace
			if (c == ' ' || c == '\n' || c == '\t') {
				// if it is, then increase the i
				++i;
			} else {
				// if it is not, then we reached the actual data
				break;
			}
		}

		// clean the data from the initial whitespace
		data = data.substr(i);

		// Decode the data from the Base64
		const std::string &decoded = base64_decode(data);

		// get The size of the GID will have (size * 4 bytes (int))
		uLongf size = (width * height) << 2;

		// Allocate memory for the GIDS
		gids = new unsigned int[size];

		// try to uncompress the Decoded data
		if (uncompress((Bytef*) gids, &size,
				(const Bytef*) decoded.data(),
				decoded.size()) != Z_OK) {
			// If something happened report it
			Logger::getInstance().write(
		"During uncompressing the Decoded Data from a Layer of the
Map: "
					+ parrent.getName(),
Logger::MessageType::Error);
			// Delete the allocated data
			delete[] gids;
			// return
			return;
		}
	}

	// If everything was alright when uncompressing
	// we have to get the referenced tiles
```

```
      for (unsigned int i = 0; i < height; ++i) {
            for (unsigned int j = 0; j < width; ++j) {
                  // Calculate the Index of the tile on the Table
                  int index = i * width + j;
                  // check if the GID there has a reference
                  if (gids[index] == 0) {
                        // if it does not, then do not assign anything
                        continue;
                  }

                  // Remove the flags of the Tile
                  gids[index] &= ~(0x80000000u | 0x40000000u
                                          | 0x20000000u);

                  // We have to find in which Tileset this GID residents
                  // We have to iterate The tilesets in reverse, to get
                  // first the Tileset with the larger First GID
                  for (auto itr = tilesets.rbegin();
                              itr != tilesets.rend(); ++itr) {
                  // Check if this Tileset's First GID is bellow the
                  // Tile's GID
                        if ((*itr).getFirstGid() <= gids[index]) {
                              // If it does, then assign the Tile to the
                              // Table
                              tiles(j, i) =
                                    (*itr).getTileset()->getMapTile(
                                          gids[index]
                                          - (*itr).getFirstGid());
                              // We need to go no further since we found
                              // it
                              break;
                        }
                  }

            }
      }
      // deallocate the data
      delete[] gids;
}
```

Αρχικά παίρνουμε τα βασικά στοιχεία από το στοιχείο <layer>. Αφού έχουμε τις διαστάσεις του Layer, τότε μπορούμε να μετασχηματίσουμε τον πίνακα **tiles** στο σωστό μέγεθος. Αυτός ο πίνακας είναι ο δισδιάστατος πίνακας που θα αναπαριστά το Layer και θα φιλοξενήσει τους δείκτες στα Tiles. Κατά τον μετασχηματισμό του, όλοι οι Pointers γίνονται Null. Αφού το κάνουμε αυτό, παίρνουμε τα καθαρά δεδομένα του Layer από το στοιχείο <data>.

Τα καθαρά δεδομένα είναι ένας μονοδιάστατος πίνακας από αριθμούς GID. Μπορεί να είναι στην σειρά, αλλά δεδομένα αυτά αναπαριστούν το Layer σε δισδιάστατο επίπεδο και για αυτό θα χρειαστεί παρακάτω να υπολογίζουμε την τοποθεσία κάποιου Tile σε αυτό. Πέρα από αυτό, τα δεδομένα αυτά είναι συμπιεσμένα με την

zlib και αυτά τα συμπιεσμένα δεδομένα είναι κωδικοποιημένα με την κωδικοποίηση Base64. Πριν προβούμε σε αυτές τις ενέργειες, πρέπει να καθαρίσουμε τα δεδομένα από τυχών κενά, tabs και νέες γραμμές που πιθανόν υπάρχουν πριν τα δεδομένα γιατί ίσως παρεμβάλουν με αυτά. Αφού καθαρίσουμε τα δεδομένα, τότε τα αποκωδικοποιούμε από Base64. Στην συνέχεια δεσμεύουμε μνήμη για να βάλουμε τα GIDS όταν τα αποσυμπιέσουμε. Με την **uncompress(...)** της βιβλιοθήκης zlib, αποσυμπιέζουμε τα δεδομένα που αποκωδικοποιήσαμε και ελέγχουμε αν έγινε η εργασία χωρίς σφάλματα. Αν υπήρχαν σφάλματα, τότε αποδεσμεύουμε την μνήμη και επιστρέφουμε χωρίς να βάλουμε περαιτέρω δεδομένα. Σε αυτή την φάση, το Layer είναι απολύτως χρησιμοποιήσιμο, καθώς έχει αρχικοποιηθεί για αυτό και απλά επιστρέφουμε.

Εφόσον ήμαστε εντάξει με τα δεδομένα, τότε μπορούμε να τα διαβάζουμε και να αναθέσουμε Tiles όπου ορίζεται. Αυτό γίνεται με την τελευταία επανάληψη. Αυτή η επανάληψη γίνεται για κάθε στοιχείο του δισδιάστατου πίνακα **tiles**. Εδώ, πρέπει να δούμε τι θα βάλουμε σε καθένα από αυτά. Αυτό το ορίζουν τα δεδομένα που αποσυμπιέσαμε. Όπως είπαμε, αυτά τα δεδομένα είναι ένας δισδιάστατος πίνακας που έχει μετατραπεί σε μονοδιάστατο. Έτσι ξέρουμε ότι τα πρώτα **width** στοιχεία είναι η πρώτη γραμμή του πίνακα, τα επόμενα **width** στοιχεία η δεύτερη κοκ. Με αυτή την λογική υπολογίζουμε το **index** που θα χρησιμοποιήσουμε για να πάρουμε την αναφορά από τα αποσυμπιεσμένα δεδομένα. Με αυτόν τον δείκτη παίρνουμε την τιμή από το **gids** για το και ελέγχουμε αν είναι 0. Αν είναι, τότε σημαίνει αυτό το Tile του Layer δεν έχει κανένα Tile και έτσι πάμε στο επόμενο. Αν δεν είναι, τότε σημαίνει ότι έχει κάποιο GID που αναφέρεται σε κάποιο Tile σε κάποιο από τα Tilesets που έχει ο χάρτης. Πριν το ψάξουμε, πρέπει να αφαιρέσουμε από την τιμή κάποια Flags που ενσωματώνει το Tiled. Αυτό το κάνουμε με την λογική πράξη **AND** όπως βλέπουμε. Με την πράξη αυτή “διαλέγουμε” ποια Bits θα κρατήσουμε από την τιμή. Επειδή όμως εδώ θέλουμε να κρατήσουμε τα πάντα εκτός από τα Bits των Flags, συνδυάζουμε τις τιμές των Flags με την **OR** και την αντιστρέφουμε για να πάρουμε όλα τα Bits εκτός αυτών.

Αφού έχουμε καθαρή την τιμή του GID, τότε αρχίζουμε να το ψάχνουμε στα διαθέσιμα Tilesets. Η αναζήτηση γίνεται αντίστροφα από το Tileset με την μεγαλύτερη τιμή **firstGID** πρώτα. Αυτό το κάνουμε γιατί όταν βρούμε το πρώτο **firstGid** πού είναι μικρότερο από το GID που έχουμε, τότε θα ξέρουμε ότι αυτό το Tileset το έχει αυτό το Tile που ψάχνουμε, καθώς για να ανήκει ένα GID σε ένα Tileset, πρέπει να βρί-

σκεται μεταξύ των ορίων του **firstGid** αυτού και του επόμενου Tileset στην σειρά. Αφού βρούμε σε ποιο Tileset βρίσκεται, πρέπει να ανακτήσουμε το Tile αυτό από μέσα του. Τα Tiles όμως μέσα στο Tileset δεν ορίζονται από το GID τους αλλά από το ID που έχουν εσωτερικά. Δηλαδή για να πάρουμε το πρώτο, πρέπει να ζητήσουμε το 0 κοκ. Όπως είχαμε αναφέρει παλιότερα, όταν γίνεται η ανάθεση firstGID στα Tilesets μπαίνουν στην σειρά, δηλαδή πχ το πρώτο Tile του δεύτερου Tileset στην σειρά, θα έχει GID το firstGID του Tileset, το δεύτερο θα είναι firstGID+1 κοκ. Συνοπτικά είναι ο παρακάτω τύπος.

$$GID = firstGID + ID \Leftrightarrow ID = GID - firstGID \quad (4.2.1)$$

Λύνοντας τον ως προς το ID, παίρνουμε το τι πρέπει να κάνουμε για να πάρουμε το Tile από το Tileset, δηλαδή την αφαίρεση. Αφού αναθέσουμε το Tile που πήραμε, τότε σταματάμε την επανάληψη για να βρούμε το Tileset και πηγαίνουμε στο επόμενο Tile του Layer. Αφού τελειώσουμε με όλα τα Tiles, αποδεσμεύουμε την μνήμη των GID καθώς δεν τα χρειαζόμαστε άλλο.

Είδαμε μέχρι στιγμής την φόρτωση του Layer. Αυτό που αξίζει να δούμε επιπλέον από την κλάση Layer, είναι η διαδικασία σχεδίασης του Layer, που γίνεται μέσω της συνάρτησης **draw(...)**.

Πίνακας 4.2.11: Η συνάρτηση Layer::draw()

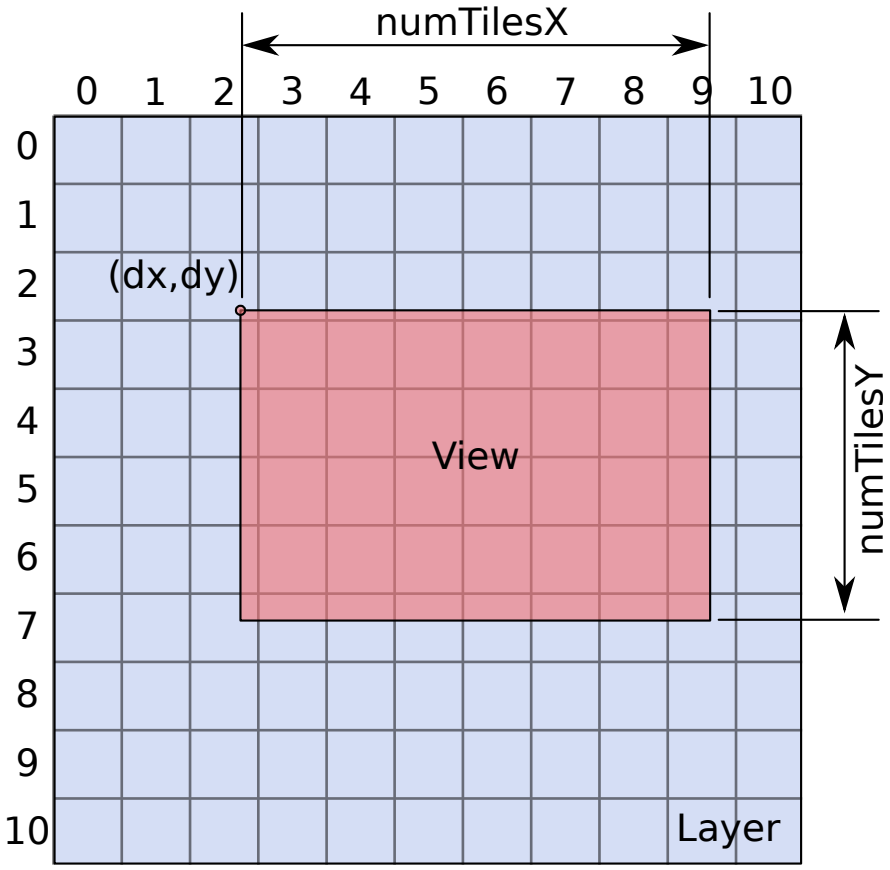
```
void Layer::draw(float dx, float dy, int numTilesX, int numTilesY) const
{

     // Calculate the start Tile Coordinates (Top Left Tile)
     int startTileX = (dx / tileWidth);
     int startTileY = (dy / tileHeight);
     // Calculate the End Tile coordinates (Bottom Right Tile)
     unsigned int endTileX = startTileX-- + numTilesX;
     unsigned int endTileY = startTileY-- + numTilesY;

     // Check if the Coordinates are valid
     // and if they are not, correct them
     if (startTileX < 0) {
           startTileX = 0;
     }
     if (startTileY < 0) {
           startTileY = 0;
     }
     if (endTileX >= width) {
           if (width > 0) {
                 endTileX = width - 1;
           } else {
                 endTileX = 0;
           }
     }
```

```
	if (endTileY >= height) {
		if (height > 0) {
			endTileY = height - 1;
		} else {
			endTileY = 0;
		}
	}
	// Get the Renderer and apply the Hold Optimization
	auto& renderer = System::getInstance().getShapeRenderer();
	renderer.holdBitmapDraw(true);
	// for all the Tiles we calculated
	for (unsigned int i = startTileY; i <= endTileY; ++i) {
		for (unsigned int j = startTileX; j <= endTileX; ++j) {
			// Get the tile at the coordinates
			const Tile* tile = tiles(j, i);
			// Check if the tile exists
			if (tile != nullptr) {
				// if it exists, then check if it animated
				if (tile->isAnimated()) {
					// if it is animated, let the Parrent Map
					// that this tile must be updated
				parent->addTileToUpdate(const_cast<Tile*>(tile));
				}
				// Draw the Tile on the appropriate coordinates
				tile->draw(std::ceil(((j * tileWidth) – dx)
						+ 0.5f),
					std::ceil(((i * tileHeight) - dy)
					+ 0.5f));
			}
		}
	}
	// Stop the optimization
	renderer.holdBitmapDraw(false);
}
```

Το Signature της συνάρτησης **draw(...)** είναι πολύ διαφορετικό σε σχέση με αυτά που έχουμε δει μέχρι στιγμής. Εδώ, τα ορίσματα που δέχεται είναι οι συντεταγμένες πάνω στο Layer και το πόσα Tiles θα σχεδιάσει σε κάθε άξονα. Στην ουσία ορίζεται ένα ορθογώνιο πάνω στον χάρτη όπου πρέπει να πάρουμε τα Tiles που βρίσκονται μέσα σε αυτό από το Layer και να τα σχεδιάζουμε.

Επειδή όμως εδώ έχουμε να κάνουμε με διακριτά Tiles μέσα στον δισδιάστατο πίνακα του Layer, πρέπει να μετατρέψουμε τις απόλυτες συντεταγμένες που μας παρέχονται σε διακριτές. Δηλαδή πρέπει να δούμε οι συντεταγμένες dx,dy, σε ποιο στοιχείο πέφτουν πάνω στον δισδιάστατο πίνακα και να ξεκινήσουμε από εκεί να σχεδιάζουμε όλα τα Tiles που ορίζονται από τα άλλα 2 ορίσματα. Αυτό γίνεται διαιρώντας τις απόλυτες συντεταγμένες με το μέγεθος του Tile. Για να βρούμε το τελευταίο tile που θα σχεδιάσουμε, θα πρέπει να προσθέσουμε τον αριθμό των Tiles που θα σχεδιάσουμε σε κάθε διάσταση. Στην επόμενη φάση, πρέπει να ελέγχουμε αν οι τιμές

αυτές που υπολογίσαμε, είναι εντός ορίων του πίνακα του Layer. Αν είναι εκτός, τότε τις διορθώνουμε με τις πιο κοντινές επιτρεπτές τιμές.



Εικόνα 4.2.2: Παράδειγμα προσδιορισμού των τιμών στην Layer::draw()

Στο παραπάνω παράδειγμα, οι υπολογισμοί μας θα μας δώσουν:

startTileX = 2

startTileY = 2

endTileX = 9

endTileY = 7

Αφού έχουμε ελέγξει τις τιμές και τις έχουμε διορθώσει, τότε μπορούμε να ξεκινήσουμε την σχεδίαση των Tiles. Η επανάληψη είναι εμφωλευμένη και ξεκινάει από τα όρια που υπολογίσαμε για κάθε διάσταση. Σε κάθε επανάληψη, στις μεταβλητές **(i, j)** έχουμε τους δείκτες στον πίνακα **tiles** του Layer. Έτσι μπορούμε να πάρουμε το Tile που βρίσκεται εκεί. Πριν κάνουμε οποιαδήποτε ενέργεια, πρέπει να ελέγξουμε αν όντως υπάρχει Tile εκεί, ελέγχοντας τον Pointer που υπάρχει στον πίνακα. Αν υπάρχει τότε συνεχίζουμε, αλλιώς πάμε στο επόμενο Tile που πρέπει να σχεδιάσουμε.

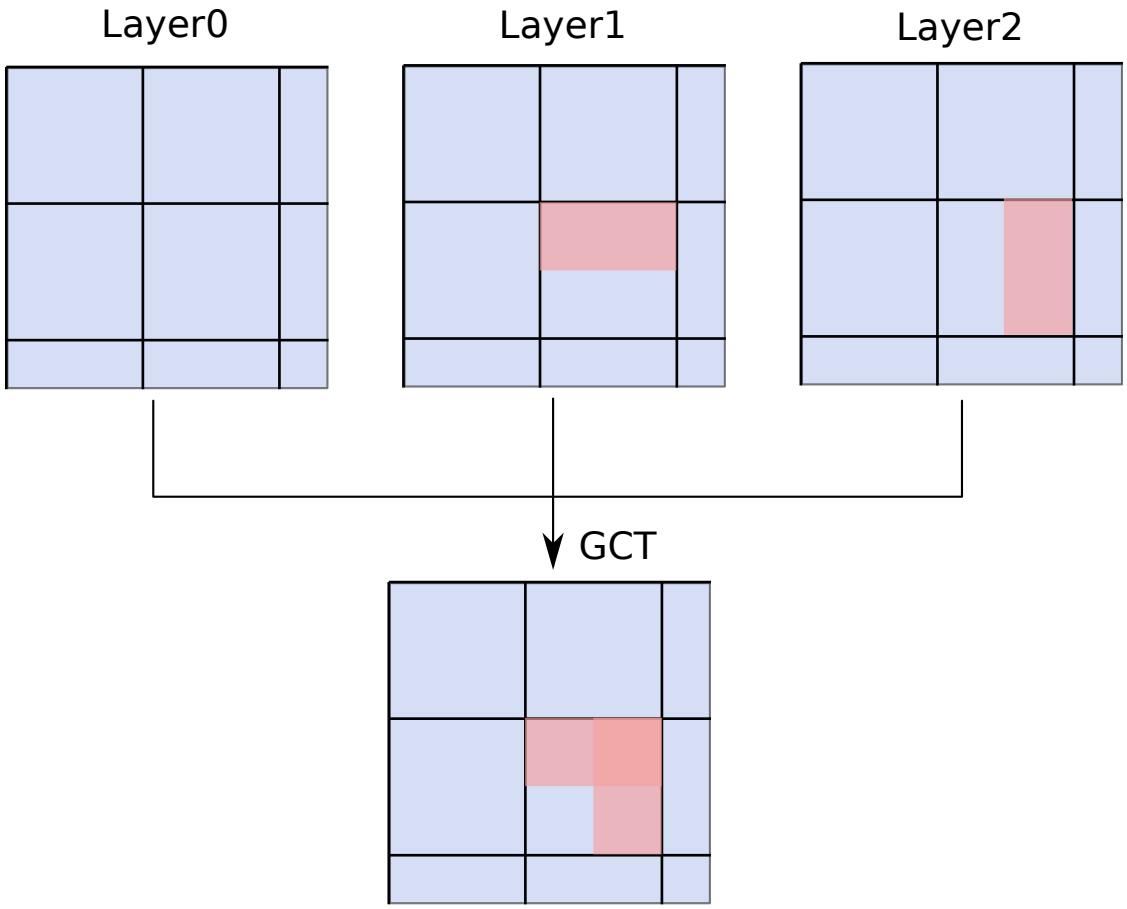
Έχοντας ένα έγκυρο Tile, ελέγχουμε αν είναι Animated. Εφόσον είναι, τότε πρέπει να πούμε στο χάρτη που έχει αυτό το Layer, να προσθέσει το Tile στην λίστα των Tiles που θα πρέπει να ενημερωθούν σε κάθε Frame. Εδώ απλά ελέγχουμε και το ζητάμε στον χάρτη, δεν κάνουμε τίποτα περαιτέρω με αυτό καθώς από εκεί και πέρα ασχολείται το **Map**. Στην επόμενη φάση πρέπει να σχεδιάσουμε το Tile που έχουμε στο σωστό σημείο πάνω στην οθόνη. Αυτό γίνεται με την κλήση **draw(...)**. Πρέπει να υπολογίσουμε όμως που θα σχεδιαστεί. Ξέρουμε το πού βρίσκεται το Tile μέσα στον πίνακα μέσω των μεταβλητών **(i, j)**. Για βρούμε της απόλυτες συντεταγμένες του πάνω στην οθόνη, αρκεί να πολλαπλασιάσουμε τα **(i, j)** επί τις διαστάσεις του Tile. Αν όμως κάνουμε μόνο αυτό, τότε το Tile θα σχεδιαστεί σαν το View να βρίσκεται στο **(0, 0)** και υπάρχει πολύ μεγάλη πιθανότητα να σχεδιαστεί εκτός οθόνης, κάτι που είναι λάθος. Για αυτό και πρέπει να κάνουμε Offset όλη την σχεδίασή του σε σχέση με τα **(dx, dy)** που μας δόθηκαν. Αυτές οι μεταβλητές μας ήρθαν από το κύριο **View** που είναι η κάμερα του παιχνιδιού. Στην ουσία είναι οι συντεταγμένες της κάμερας την στιγμή που γίνεται η σχεδίαση. Εφόσον τα Tiles που σχεδιάζουμε είναι μόνο αυτά που βρίσκονται εντός της κάμερας (χάρη των υπολογισμών που κάναμε στην αρχή) πρέπει να κάνουμε και την σχεδίαση αναγκαστικά offset για να φανεί το αποτέλεσμα που θέλουμε.

γ) Στατικές Συγκρούσεις

Στα πράγματα που πρέπει να δούμε όσο αφορά την φόρτωση είναι οι στατικές συγκρούσεις. Οι στατικές συγκρούσεις είναι οι συγκρούσεις που παθαίνουν τα Lifeforms με διάφορα Tiles του κόσμου. Τα Tiles μπορούν να έχουν μια ιδιότητα που τους προσδιορίζει την σύγκρουσή τους. Είναι ένας αριθμός που αντιπροσωπεύει ένα σχήμα εσωτερικά στην μηχανή, που σύμφωνα με αυτό θα ελέγχονται τα Lifeforms για συγκρούσεις. Οι τιμές αυτές αποδίδονται στο Tileset και όχι στο Layer. Έτσι γίνεται πιο γενικό και εύκολο στο να χτιστούν οι χάρτες. Κάθε Tile σπάει σε 4 υπο-Tiles τα οποία αυτά μπορεί είτε να είναι συγκρούσιμο, είτε όχι. Σύμφωνα με το ποια είναι συγκρούσιμα, ο αριθμός αυτός προδιαγεγραμμένος. Αυτό γιατί η μηχανή κάνει υπολογισμούς βάση αυτού που θα δούμε παρακάτω.

Ο χάρτης έχει έναν δισδιάστατο πίνακα από ακεραίους αριθμούς που αντιπροσωπεύουν τις καθολικές συγκρούσεις. Λόγω ότι τα Tiles σπάνε σε 4 όσο αφορά τις συγκρούσεις, έτσι και ο πίνακας θα έχει μέγεθος όσο το μέγεθος του χάρτη σε Tiles x2 σε κάθε διάσταση. Αυτό που πρέπει να έχει ο πίνακας αυτός είναι σε κάθε στοιχείο

του να περιέχει συγχωνευμένες τις συγκρούσεις από κάθε Tile που βρίσκεται στο σημείο αυτό σε όλα τα Layers που έχει ο χάρτης. Για παράδειγμα, έστω οτι έχουμε 3 Layers. Σε ένα στοιχείο του καθολικού πίνακα στατικών συγκρούσεων, το πρώτο Layer δεν έχει σύγκρουση, το δεύτερο έχει όλη την πάνω πλευρά και το τρίτο την δεξιά πλευρά. Το τελικό στοιχείο του καθολικού πίνακα πρέπει να είναι συγχωνευμένες αυτές οι τιμές, δηλαδή να έχει ένα Γ με την γωνία του δεξιά.



Εικόνα 4.2.3: Παράδειγμα συγχώνευσης Collision Tiles

Για τον λόγο αυτό, πρέπει να λάβουμε υπόψη το Tile σε αυτό το σημείο από όλα τα Layers που έχουμε.

Πίνακας 4.2.12: Η συνάρτηση Map::loadCollisions()

```
void Map::loadCollisions() {
     // Calculate the Sub values
     subWidth = width << 1;
     subHeight = height << 1;

     // Clear the table
```

```
      collisions.clear();

      // resize the table to the new size
      collisions.resize(subHeight, subWidth);

      // For every layer we have
      for (auto layer : layers) {

            // for every tile the layer has:
            // i,j has the Tile coordinate on the Layer
            // a,b has the sub-tile coordinate on the collisions Table
            for (unsigned int i = 0, a = 0; i < height; ++i, a += 2) {
                  for (unsigned int j = 0, b = 0; j < width; ++j, b += 2) {

            // Get the Tile at the coordinate and check if it exists there
                        if (const Tile *tl = layer.second->getTileAt(j, i)) {

                                    // if it exists, then get its collision and
                                    // according to the value assign the value
                                    // on the collision table
                                    switch (tl->getCollision()) {
                                    case Tile::Collision::N:
                                          collisions(b, a + 1) = true;
                                          collisions(b + 1, a + 1) = true;
                                          break;
                                    case Tile::Collision::S:
                                          collisions(b + 1, a) = true;
                                          collisions(b, a) = true;
                                          break;
                                    case Tile::Collision::W:
                                          collisions(b + 1, a) = true;
                                          collisions(b + 1, a + 1) = true;
                                          break;
                                    ...
```

Στον παραπάνω κώδικα για κάθε Layer που έχουμε τρέχουμε μια επανάληψη για όλα τα στοιχεία του δισδιάστατου πίνακα. Η επανάληψη γίνεται με τον αριθμό των Tiles παρά των υπο-Tiles, αυτό γιατί σε κάθε Tile ξέρουμε τι θα βάλουμε στα 4 υπό-Tiles της περιοχής. Έτσι πέρα από τα **(i, j)** του πίνακα, θα έχουμε και τα **(a, b)** που θα φιλοξενούν τους δείκτες του **collisions** που είναι διπλάσιος αυτού των Layers. Αφού πάρουμε το Tile που δείχνουν οι δείκτες μας από το Layer που έχουμε, ελέγχουμε αν υπάρχει όντως (ο Pointer δεν είναι Null). Εφόσον έχουμε όντως έγκυρο Tile, τότε προχωράμε να αναθέσουμε τιμές στα υπό-Tiles που υπάρχουν στην περιοχή. Η ανάθεση γίνεται βάση της τιμής Collision που έχει το Tile. Όπως είπαμε η τιμή αυτή αναπαριστά κάποιον συνδυασμό των “ενεργών” υπό-Tiles του Tile. Τα ποια υπό-Tiles πρέπει να ενεργοποιηθούν, παίρνεται από τον παρακάτω πίνακα που έχουμε ξαναδεί σε προηγούμενο κεφάλαιο.

N+W 7 (0x7)	SE 8 (0x8)	S 12 (0xC)	SW 4 (0x4)	N+E 11 (0xB)
NW+SE 9 (0x9)	E 10 (0xA)	FULL 15 (0xF)	W 5 (0x5)	NE+SW 6 (0x6)
S+W 13 (0xD)	NE 2 (0x2)	N 3 (0x3)	NW 1 (0x1)	S+E 14 (0xE)

Εικόνα 4.2.4: Ο Πίνακας Αναφοράς Στατικών Συγκρούσεων

Σύμφωνα με τον παραπάνω πίνακα, θέτουμε τις τιμές μέσα στην συνάρτηση. Όπως βλέπουμε, έχουμε και συνδυασμένες τιμές. Οι τιμές αυτές του πίνακα αποδόθηκαν με τέτοιο τρόπο, ώστε να μπορούν να συνδυαστούν με μια OR και να έχουμε σωστό αποτέλεσμα. Πχ αν κάνουμε OR στην τιμή του S(0xC) και του W(0x5) μας δίνει 0xD που είναι η τιμή του S+W. Συνδυάζοντας και τα σχήματά τους, βλέπουμε ότι είναι σωστός ο συνδυασμός τόσο στον αριθμό, όσο και στο σχήμα.

Όταν οι επαναλήψεις τελειώσουν, τότε θα έχουμε ολοκληρωμένο τον πίνακα στατικών συγκρούσεων, που θα χρησιμοποιείται όσο ο χάρτης τρέχει.

Εδώ τελειώνουν τα κυριότερα μέρη της φόρτωσης του χάρτη. Μερικά από τα πράγματα που παραλείφθηκαν είναι ο διαχωρισμός των Layers σε άνω και κάτω και η δημιουργία των αντικειμένων. Όσο αφορά τα αντικείμενα, θα δούμε περισσότερα παρακάτω.

4.3 Το τετραδικό δένδρο (Quadtree)

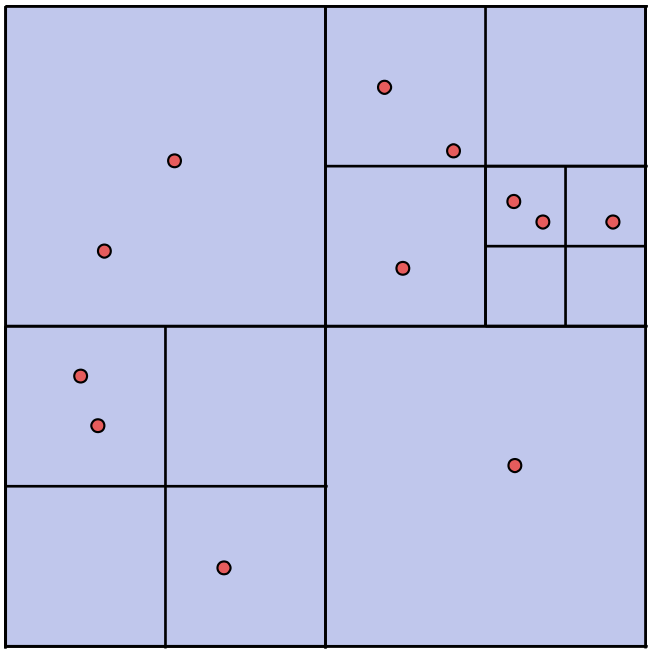
Στον χάρτη υπάρχουν τα αντικείμενα που είναι δυναμικά. Αυτά τα αντικείμενα είναι Lifeforms, StaticObjects, InteractFields κτλ. Όλα τα δυναμικά αντικείμενα μπορούν να ενημερωθούν μέσω της **update(...)**. Πέρα από αυτό, τα αντικείμενα αυτά ελέγχονται και για συγκρούσεις. Οι συγκρούσεις ελέγχονται από κάθε Lifeform που κινείται σε κάθε Frame. Για να είναι σωστός ο έλεγχος των συγκρούσεων, πρέπει να ελεγχθεί το

Lifeform με όλα τα άλλα δυναμικά αντικείμενα αν συγκρούεται. Εν συντομία, κάθε Lifeform θα ελέγξει αν συγκρούεται με άλλο. Αυτό έχει τετραγωνική πολυπλοκότητα (O(N^2)). Για παράδειγμα, αν έχουμε 10 Lifeforms που κινούνται, τότε θα γίνουν:

$$testsCount(n) = n^2$$
$$testsCount(10) = 10^2 = 100$$

Δηλαδή για 10 αντικείμενα θα γίνουν 100 έλεγχοι. Αυτό είναι πέρα από συνετό και πρέπει να γίνει κάτι να ελαττωθεί. Μια τεχνική που εφαρμόστηκε ήταν να ελέγχονται μόνο όταν κάτι κινείται. Όμως αυτό δεν έφτανε, γιατί σε περιπτώσεις που όλα τα αντικείμενα θα κινούνταν, πάλι θα είμαστε τον μέγιστο αριθμό ελέγχων. Για αυτό και εφαρμόστηκε ένα επιπλέον μέτρο, το τετραδικό δένδρο.

Το τετραδικό δένδρο είναι μια τεχνική διάσπασης του δισδιάστατου χώρου σε 4 μικρότερα, ανάλογα κάποια μεταβλητή πολυπλοκότητας. Ανάλογα την πολυπλοκότητα κάποιας περιοχής, ίσως χρειαστεί να διασπαστεί περαιτέρω κάποια περιοχή που έχει ήδη διασπαστεί. Στην δική μας περίπτωση η μεταβλητή πολυπλοκότητας είναι ο αριθμός των αντικειμένων στον χάρτη. Δηλαδή θα πρέπει να διασπάσουμε την περιοχή ανάλογα τον αριθμό των αντικειμένων που υπάρχουν εκεί. Για παράδειγμα αν θέλουμε σε μια περιοχή να υπάρχουν το πολύ 2 αντικείμενα, τότε θα έχουμε ένα δένδρο όπως στο παρακάτω παράδειγμα.



Εικόνα 4.3.1: Παράδειγμα QuadTree

Εδώ έχουμε 13 αντικείμενα διάσπαρτα στον χώρο. Ο χώρος αυτός μπορούμε να θεωρήσουμε ότι είναι ο όλος ο χάρτης. Είπαμε ότι θέλουμε το πολύ 2 αντικείμενα σε ένα κομμάτι. Έτσι αρχικά διασπάμε τον χώρο σε 4 κομμάτια. Κάνοντας αυτό, έχουμε 2 αντικείμενα στον πάνω πάνω αριστερά αλλά πολύ παραπάνω στο κάτω αριστερά και στο πάνω δεξιά. Για τηρήσουμε τον κανόνα, διασπάμε και αυτά τα κομμάτια σε 4 ώστε να το πετύχουμε. Στο κάτω αριστερά το πετύχαμε άλλα όχι στο πάνω δεξιά. Για αυτό και διασπάμε την υπό-περιοχή του που έχει παραπάνω για το πετύχουμε. Αυτό μπορούμε να το κάνουμε συνεχόμενα μέχρι να έχουμε το αποτέλεσμα που θέλουμε ή να φτάσουμε σε κάποιο όριο βάθους του δένδρου.

Σπάζοντας τον χώρο έτσι, έχουμε σε κάθε υπό-περιοχή το πολύ 2 αντικείμενα. Έτσι όταν πάμε να ελέγξουμε τις συγκρούσεις για κάποιο αντικείμενο, θα πάρουμε μόνο τα αντικείμενα που βρίσκονται στην περιοχή που βρίσκεται. Έτσι στο παράδειγμα αυτό θα γίνουν το πολύ 2 έλεγχοι για κάθε αντικείμενο, πολύ λιγότεροι από τις 13 που θα κάναμε.

Στην μηχανή, χρησιμοποιούμε μια λιγότερο δυναμική έκδοση του τετραδικού δένδρου. Οι διασπάσεις των περιοχών δεν γίνονται δυναμικά όποτε χρειάζεται, αλλά γίνεται από την κατασκευή του δένδρου και απενεργοποιούνται οι υπό-περιοχές. Όποτε χρειάζεται να γίνει διάσπαση, ενεργοποιούνται οι υπό-περιοχές που χρειάζονται. Αυτή η προσαρμοσμένη έκδοση του δένδρου φτιάχτηκε για λόγους Optimization, προκειμένου να αποφύγουμε τις πολλές malloc() κατά τις διασπάσεις, γιατί το δένδρο κατασκευάζει και καταστρέφει όλες τις περιοχές του σε κάθε Frame.

Πίνακας 4.3.1: Η κλάση StaticQuadtree

```
class StaticQuadtree: public Drawable {
public:
     int insert(Object* obj);
     std::vector<Object*> getObjectsAt(Float x, Float y) const;
     std::vector<Object*> getObjectsAt(const Rectangle& rect) const;

     void draw(Float x, Float y, Float rotation = 0.0f,
                Float scale = 1.0f) const;

     void clear();
     bool isLeaf() const;
     void setLeaf(bool leaf);

     void setBounds(Float x, Float y, Float width, Float height);
     void offset(Float dx, Float dy);
     void resetPosition();
     void setSize(Float width, Float height);
     void setPosition(Float x, Float y);
```

```
      Float getWidth() const;
      Float getHeight() const;

      StaticQuadtree();
      StaticQuadtree(StaticQuadtree* parent, int level);
      ~StaticQuadtree();
private:
      enum Node {
            NW, NE, SW, SE
      };
      enum Insert_Out {
            No_Contact = -2, Child_Handles = -1, Parent_Handles = 0
      };

      std::vector<Object*> objects;

      StaticQuadtree *childs[4];
      StaticQuadtree *parent;

      Rectangle bounds;

      int level;
      const int maxLevel = 5;
      const unsigned int maxObjects = 10;

      bool leaf;

      void subDivide();
      bool isInside(Object *obj);
      bool touches(Object *obj);
};
```

Η κλάση είναι αναδρομική, που σημαίνει ότι εσωτερικά έχει αντικείμενα της ίδιας κλάσης. Κάθε υπό-περιοχή θα την λέμε παιδί και θα είναι και αυτό **StaticQuadtree**. Επειδή το δένδρο θα χωρίσει περιοχή, χρειάζεται έναν χώρο που ορίζεται από το ορθογώνιο **bounds**. Το **level** ορίζει το βάθος του τρέχοντος δένδρου. Το **maxLevel** και **maxObjects** είναι κοινόχρηστα σε όλα τα παιδιά και τους θέτει τα όρια. Το **leaf** ορίζει αν τα παιδιά είναι ενεργά (δηλαδή αν πρέπει να θεωρείται διασπασμένο). Το **objects** περιέχει τα αντικείμενα αυτής της περιοχής.

Πίνακας 4.3.2: Οι κατασκευαστές της StaticQuadtree

```
StaticQuadtree::StaticQuadtree() :
            parent(nullptr), level(0), leaf(true) {
      childs[NW] = new StaticQuadtree(this, 1);
      childs[NE] = new StaticQuadtree(this, 1);
      childs[SW] = new StaticQuadtree(this, 1);
      childs[SE] = new StaticQuadtree(this, 1);

}
```

```
StaticQuadtree::StaticQuadtree(StaticQuadtree* parent, int level) :
          parent(parent), level(level), leaf(true) {
     // Check if the level is below the max allowed
     if (level < maxLevel) {
          // Create the children on the next level
          childs[NW] = new StaticQuadtree(this, level + 1);
          childs[NE] = new StaticQuadtree(this, level + 1);
          childs[SW] = new StaticQuadtree(this, level + 1);
          childs[SE] = new StaticQuadtree(this, level + 1);
     } else {
          // if we reached the max level, create nothing
          childs[NW] = nullptr;
          childs[NE] = nullptr;
          childs[SW] = nullptr;
          childs[SE] = nullptr;
     }
}
```

Όπως είπαμε, κατά την κατασκευή του πρώτου δένδρου κατασκευάζονται και όλα τα παιδιά. Έτσι πρέπει να κατασκευάσουμε τα παιδιά, τα οποία παιδιά να κατασκευάσουν τα παιδιά τους κοκ, μέχρι να φτάσουμε στο μέγιστο βάθος που ορίζεται από το **maxLevel**. Το αρχικό δένδρο κατασκευάζεται από τον απλό κατασκευαστή που δεν δέχεται ορίσματα. Αυτός κατασκευάζει τα 4 παιδιά μέσω του δεύτερου κατασκευαστή. Τα παιδιά ελέγχουν το επίπεδο που τους δόθηκε από τον πατέρα. Αν μπορούν να δημιουργήσουν παιδιά, τότε τα δημιουργούν και τους δίνεται το επόμενο επίπεδο βάθους. Αν τα παιδιά φτάσουν στο μέγιστο βάθος, τότε δεν δημιουργείται τίποτα. Αυτό που πρέπει να τονιστεί είναι ότι σε κάθε κατασκευή, το flag **leaf** ορίζεται σε true, που σημαίνει ότι τα παιδιά τους απενεργοποιούνται αυτόματα.

Η επόμενη συνάρτηση που θα δούμε εδώ είναι η **insert(...)**. Αυτή έχει αρμοδιότητα να εισάγει ένα αντικείμενο στο δένδρο αυτό. Δεν εγγυάται ότι το αντικείμενο θα εισαχθεί στο δένδρο, καθώς υπάρχουν προϋποθέσεις. Για αυτό και η συνάρτηση επιστρέφει ένα αποτέλεσμα της εισαγωγής. Η λειτουργία της είναι να δοκιμάσει να βάλει το αντικείμενο μέσα στον πίνακα της. Αν όμως υπάρχουν ήδη πολλά αντικείμενα μέσα στον πίνακα, τότε πρέπει να γίνει διάσπαση και να τοποθετηθεί σε ένα από τα παιδιά. Αυτό θα γίνει μόνο εφόσον το αντικείμενο χωράει εξολοκλήρου μέσα στο παιδί. Αν πχ εξέχει από το παιδί, τότε πρέπει ο πατέρας να το πάρει παρόλο αυτά. Υπάρχει όμως και η περίπτωση το αντικείμενο να μην βρίσκεται καν εντός της περιοχής του πατέρα, που σημαίνει ότι πρέπει να το αναλάβει ο πατέρας του πατέρα.

Πίνακας 4.3.3: Η συνάρτηση StaticQuadtree::insert(...)

```
int StaticQuadtree::insert(Object* obj) {
	// Check if the object is overlapping with this node
	if (bounds.overlapsRectangle(obj->getBounding())) {
		// If it overlaps, then check if it is wholly inside
		if (bounds.overlapsWholeRectangle(obj->getBounding())) {
			// if it it's wholly inside, then add it
			objects.push_back(obj);
		} else {
			// if it is not wholly inside, check if we have a parent
			if (parent != nullptr) {
				// if we have a parent, then he should add it
				return Insert_Out::Parent_Handles;
			} else {
				// if we haven't a parent, then we will add it
				objects.push_back(obj);
			}
		}
	} else {
		// if it does not overlap at all, then let the caller know
		return No_Contact;
	}
	// Here we will get if we added the object
	// We have to check if we reached the max object count
	// for this node
	if (objects.size() > maxObjects) {
		// If we have reached the limits, then we need to subdivide
		// the node to 4
		if (leaf) {
			// if we are leaf, then check if we can subdivide
			if (level < maxLevel) {
				// if we can then do it
				subDivide();
			} else {
				// if we can't subdivide, then pass it to the
				// parent
				// (although we have the item?? it makes no sense)
				return Insert_Out::Parent_Handles;
			}
		}
		// since we subdivided, we have to insert the objects to the
		// new
		for (auto itr = objects.begin(); itr != objects.end();) {
			// for all sub-nodes
			for (int i = 0; i < 4; ++i) {
				// try to insert and get the outcome
				int res = childs[i]->insert(*itr);
				// check the result
				if (res == Insert_Out::Parent_Handles) {
					// if we should handle it, to the next
					// object since we already handle it
					++itr;
					break;
				} else if (res == Insert_Out::Child_Handles) {
					// if the child handle it, then erase the
					// object from this node
					itr = objects.erase(itr);
					break;
```

```
                        }
                    }
                }
            }
            // Let the parrent that we handled the object
            return Insert_Out::Child_Handles;
        }
```

Αρχικά ελέγχουμε αν το αντικείμενο ακουμπάει την περιοχή μας. Εφόσον ακουμπάει, πρέπει να δούμε αν είναι τελείως μέσα στην περιοχή (δεν εξέχει). Εφόσον είναι, τότε μπορούμε να το βάλουμε, αλλιώς πρέπει να το αναλάβει ο πατέρας. Αν δεν υπάρχει πατέρας, τότε δεν υπάρχει άλλη επιλογή από το να το δεχτούμε εμείς. Αν δεν ακουμπάει καν στην περιοχή μας, τότε πρέπει να ενημερώσουμε τον πατέρα μέσω της τιμής **No_Contact**. Εφόσον το αντικείμενο έχει μπει στον πίνακα, τότε πρέπει να ελέγξουμε αν έχουμε περάσει τον μέγιστο αριθμό αντικειμένων για ένα Quadtree. Εφόσον δεν τον έχουμε περάσει, τότε επιστρέφουμε ότι το αναλάβαμε μέσω της τιμής **Child_Handles**. Αν όμως έχουμε περάσει τον αριθμό, τότε πρέπει να διασπάσουμε την περιοχή και να βάλουμε τα αντικείμενα μας στις υπό-περιοχές που ταιριάζουν. Πριν κάνουμε την διάσπαση, πρέπει να ελέγξουμε αν μπορούμε να διασπάσουμε. Για να μπορέσουμε, πρέπει να μην έχουμε υπερβεί το μέγιστο βάθος. Εφόσον μπορούμε, τότε το κάνουμε και πρέπει να πάρουμε τα αντικείμενα που έχουμε να τα βάλουμε στα 4 παιδιά. Για κάθε περιοχή, δοκιμάζουμε να τοποθετήσουμε το αντικείμενο καλώντας την ίδια συνάρτηση **insert(...)**. Όπως ξέρουμε μέχρι στιγμής, 3 είναι τα πιθανά αποτελέσματα, είτε να το αναλάβει το παιδί, είτε να μην το ακουμπάει καν είτε να πρέπει να το αναλάβουμε εμείς σαν πατέρας. Αν δεν ακουμπάει καν, τότε δοκιμάζουμε το επόμενο παιδί μέχρι να πετύχουμε το παιδί που θα το αναλάβει. Αν το αναλάβει το παιδί, τότε πρέπει να το διαγράψουμε από τον πίνακα του πατέρα γιατί τώρα θα το έχει το παιδί στον δικό του πίνακα. Αν πρέπει να το αναλάβουμε εμείς, τότε πάμε στο επόμενο αντικείμενο καθώς το έχουμε ήδη μέσα στον πίνακα μας.

Πίνακας 4.3.4: Η συνάρτηση StaticQuadtree::subDivide()

```
void StaticQuadtree::subDivide() {
    // We have to place the children
    Float x = bounds.getX(), y = bounds.getY();
    Float half_w = bounds.getWidth() / 2, half_h =
                                            bounds.getHeight() / 2;

    // Top Left
```

```
        childs[NW]->setLeaf(true);
        childs[NW]->setBounds(x, y, half_w, half_h);

        // Top Right
        childs[NE]->setLeaf(true);
        childs[NE]->setBounds(x + half_w, y, half_w, half_h);

        // Bottom Left
        childs[SW]->setLeaf(true);
        childs[SW]->setBounds(x, y + half_h, half_w, half_h);

        // Bottom Right
        childs[SE]->setLeaf(true);
        childs[SE]->setBounds(x + half_w, y + half_h, half_w, half_h);

        // Since we have children, we are no longer leaf
        leaf = false;
    }
```

Στην συνάρτηση της διάσπασης, αυτό που χρειάζεται να κάνουμε είναι να θέσουμε τις διαστάσεις και την τοποθεσία του κάθε παιδιού σε σχέση με τον πατέρα. Και τα 4 παιδιά θα βρίσκονται εντός του πατέρα αλλά θα μοιράζονται τον χώρο του δια του 4. Εφόσον ενεργοποιούμε τα παιδιά, τότε χρειάζεται να κάνουμε τον πατέρα να μην είναι **leaf** θέτοντάς το σε false.

Η τελευταία συνάρτηση που θα δούμε εδώ, είναι η ανάκτηση των αντικειμένων που υπάρχουν στο Quadtree. Πρέπει όταν μας δίνεται μια τοποθεσία για αναζήτηση, να επιστρέψουμε όλα τα αντικείμενα που υπάρχουν εντός του παιδιού που βρίσκεται αυτή η τοποθεσία. Όμως αυτό δεν φτάνει απλά. Πρέπει να επιστρέψουμε και τα παιδιά που έχουν οι διαδοχικοί πατεράδες. Αυτό γιατί όπως είδαμε, όταν αντικείμενο προσπαθούμε να το βάλουμε σε ένα παιδί και δεν χωράει ολόκληρο μέσα, τότε δεν το αναλαμβάνει το παιδί, αλλά ο πατέρας του.

Πίνακας 4.3.5: Η συνάρτηση StaticQuadtree::getObjectsAt(Float, Float)

```
std::vector<Object*> StaticQuadtree::getObjectsAt(Float x, Float y) const
{
    // we should collect all the objects from here and the children
    std::vector<Object*> returnOBJS = objects, childOBJS;
    // check if this node is leaf
    if (!leaf) {
        // if it is not, then get the objects from the child that
        // overlaps
        if (childs[NW]->bounds.overlapsPoint(x, y)) {
            childOBJS = childs[NW]->getObjectsAt(x, y);
        } else if (childs[NE]->bounds.overlapsPoint(x, y)) {
            childOBJS = childs[NE]->getObjectsAt(x, y);
        } else if (childs[SW]->bounds.overlapsPoint(x, y)) {
            childOBJS = childs[SW]->getObjectsAt(x, y);
```

```
            } else if (childs[SE]->bounds.overlapsPoint(x, y)) {
                  childOBJS = childs[SE]->getObjectsAt(x, y);
            }
      } else {
            // if it is leaf, then return the objects we have
            return objects;
      }

      // merge the table to get all the objects
      returnOBJS.insert(returnOBJS.end(), childOBJS.begin(),
childOBJS.end());
      // return the objects
      return returnOBJS;
}
```

Τα αντικείμενα που θα επιστρέψουμε είναι σίγουρα αυτά που περιέχει το υπάρχων Quadtree, συν ότι βρούμε από τα παιδιά. Εφόσον έχουμε παιδιά, τότε ψάχνουμε σε ποιο από τα παιδιά βρίσκεται η περιοχή που μας παρέχεται. Εφόσον την βρούμε, τότε παίρνουμε τα αντικείμενα που έχει αυτό το παιδί και τα προσθέτουμε στην λίστα των αντικειμένων που θα επιστρέψουμε στον καλών. Πρέπει να τονίσουμε ότι η συνάρτηση είναι αναδρομική, που σημαίνει ότι για να πάρουμε τα αντικείμενα των παιδιών μας, θα καλέσουμε την ίδια σε αυτά και αυτά με την σειρά τους την ίδια στα παιδιά τους κοκ. Έτσι ο αρχικός καλών θα έχει τα αντικείμενα που πρέπει σε οποιοδήποτε βάθος και αν πέσει η περιοχή που μας παρείχε.

Η συνάρτηση **clear()** καθαρίζει στο τρέχων δένδρο τον πίνακα με τα αντικείμενα χωρίς να τα καταστρέψει. Πέρα από αυτό, καθαρίζει και τα παιδιά του απενεργοποιώντας τα.

4.4 Λειτουργία Frame

Ο χάρτης καθώς περιέχει όλα τα αντικείμενα του κόσμου, χρειάζεται να αναλάβει και το πως θα ενημερωθούν αυτά τα αντικείμενα. Για λόγους ταχύτητας, δεν ενημερώνονται όλα τα αντικείμενα. Θα πρέπει να ενημερωθούν μόνο αυτά που θα έχουν αλληλεπίδραση με τον παίχτη, δηλαδή αυτά που φαίνονται στην οθόνη. Για να ξέρουμε όμως ποια αντικείμενα βρίσκονται εντός οθόνης, πρέπει να γίνουν έλεγχοι. Πέρα από αυτό, χρειάζονται να ενημερωθούν και Tiles τα οποία είναι Animated που θα εμφανιστούν στην οθόνη. Όλα αυτά γίνονται στην λειτουργία Frame του χάρτη ή αλλιώς στην συνάρτηση **update(...)**.

Πίνακας 4.4.1: Η συνάρτηση Map::update()

```
void Map::update(Float elapsedTime) {
	quadtree.clear();
	visibles.clear();

	// Update all the Animated Tiles
	for (auto tile : tilesToBeUpdated) {
		tile->update(elapsedTime);
	}

	// For all the items that are to be removed
	for (auto& itr : toBeDeleted) {
		// Remove it from the inView if it is there
		inView.erase(itr->getObject());
		// if we must destroy it too
		if (itr->isToBeDeleted()) {
			// destroy it
			delete itr->getObject();
		}
		// remove it from the objects list
		objects.erase(itr);
	}

	// clear the objects to be removed
	toBeDeleted.clear();
	// Reset the offsets of the camera
	camera->resetOffests();

	Player* player = &WorldManager::getInstance().getPlayer();

	// Put the player on the visibles list
	visibles.push_back(player);

	// for all objects we have
	for (auto& objPair : objects) {
		// Get the actual object
		Object *obj = objPair.getObject();

		// Check if it is in view
		if (camera->isInView(*obj)) {
			// Check if it is visible
			if (!obj->isVisible()) {

				// if it is not visible, then it means
				// that it just showed up on the camera
				// and we must show it
				obj->show();
			}

	// we must calculate the order of the object according to other
objects
			auto itr = visibles.begin();
			// for all the objects
			for (; itr != visibles.end(); ++itr) {

				// check if the Y coord is below the others
				if (obj->getPosition().getY() <
```

```
                                        (*itr)->getPosition().getY()) {
                                // If it is, then we must put it there
                                break;
                        }
                }

                // insert the object on the right position to be
                // rendered
                visibles.insert(itr, obj);
                // insert it on the inView too
                inView.insert(obj);
        } else {
                // if it is not on the view, check if we already
                // have hide it
                if (obj->isVisible()) {

                        // if it is not hidden, hide it
                        obj->hide();
                }
                // remove it from the inView if it is there.
                inView.erase(obj);
        }
    }
    // All the objects on the screen should be checked for collisions
    // so we insert them on the quadtree
    for (auto obj : visibles) {
        // we should only add then if the object can collide
        if (obj->isCollideAble()
                || obj->isTriggeringCollisionEvents()) {
                quadtree.insert(obj);
        }
    }

    // All the objects on screen should be updated
    for (auto obj : visibles) {
        obj->update(elapsedTime);
    }
}
```

Κάθε φορά που κάνουμε ένα Frame, το **quadtree** και ο πίνακας **visibles** πρέπει να καθαριστούν γιατί έχουν δεδομένα από το προηγούμενο Frame που πιθανόν να μην είναι έγκυρα πλέον. Για αυτό αυτά τα 2 θα τα ξαναγεμίζουμε σε κάθε Frame. Σε πρώτη φάση, πρέπει να ενημερώσουμε τα Tiles που είναι Animated και φαίνονται στην οθόνη. Αυτά βρίσκονται στον πίνακα **tilesToBeUpdated**. Αυτός ο πίνακας γεμίζει από τα **Layers** όταν σχεδιάζονται όπως είδαμε στο κεφάλαιο 4.2.1.b. Όταν προστίθεται ένα Tile στον πίνακα τότε προστίθενται και τα συγγενικά του, για αυτό και σε αυτή την φάση ξέρουμε ότι ο πίνακας έχει όλα τα Tiles που χρειάζονται ενημέρωση.

Σε επόμενη φάση πρέπει να αφαιρέσουμε τα αντικείμενα που χρειάζονται αφαίρεση. Αυτά τα αντικείμενα βρίσκονται στον πίνακα **toBeDeleted**. Σε αυτόν τον πίνακα μπαίνουν iterators για την λίστα Objects. Κάθε φορά που χρειάζεται να αφαιρεθεί ένα

αντικείμενο από τον κόσμο, τότε προστίθεται ο iterator του στον πίνακα **toBeDeleted**. Αρχικά στην επανάληψη για την αφαίρεση, το αφαιρούμε από τον πίνακα **inView**. Ο πίνακας **inView** είναι ένα κοντινό αντίγραφο του **visibles** που πρέπει να περιέχει όλα τα αντικείμενα που βρίσκονται εντός οθόνης. Η διαφορά του **inView** πέρα από την δομή, είναι ότι δεν καθαρίζεται σε κάθε Frame όπως ο **visibles**. Αυτό γιατί κάποια στοιχεία της μηχανής χρειάζονται τα εμφανές αντικείμενα σε χρόνο που ο **visibles** είναι ακόμα άδειος. Για αυτό και χρειάζεται ρητή αφαίρεση του αντικειμένου κάθε φορά που αφαιρούμε ένα από τον χάρτη. Πρέπει να τονιστεί ότι ο πίνακας αυτός είναι Set (hashtable χωρίς key) που σημαίνει ότι ακόμα και αν δεν υπάρχει το αντικείμενο μέσα σε αυτόν, δεν θα υπάρχει πρόβλημα να καλεστεί η συνάρτηση **erase(...)**, απλώς δεν θα κάνει τίποτα. Αφού το αφαιρέσουμε, τότε ελέγχουμε αν χρειάζεται να το καταστρέψουμε κιόλας. Αφού κάνουμε και αυτή την ενέργεια, τότε το αφαιρούμε και από την λίστα **objects** μέσω του iterator που έχουμε. Αφού αφαιρέσαμε όλα αυτά τα αντικείμενα, χρειάζεται να καθαρίσουμε τον πίνακα **toBeDeleted**. Το επόμενο που χρειάζεται να κάνουμε είναι να επαναφέρουμε τα Offsets της κάμερας. Η κύρια κάμερα αποθηκεύει τις τιμές που έγινε offset κάθε φορά που την μετακινούν. Επειδή σε κάθε Frame αυτό θα αλλάζει, χρειάζεται να τις μηδενίσουμε. Το επόμενο που χρειάζεται είναι να πάρουμε τον παίχτη και να τον βάλουμε στον πίνακα με τα **visibles** που είναι σίγουρο ότι θα βρίσκεται σε αυτόν, καθώς ο παίχτης βρίσκεται πάντα στην οθόνη.

Στην επόμενη επανάληψη θα γίνει η διαλογή και ενημέρωση των αντικειμένων. Με την έννοια διαλογή θεωρείται η διαλογή των ποιόν θα ενημερωθούν και ποια όχι. Όπως είπαμε αυτή η διαλογή γίνεται με βάση αν το αντικείμενο βρίσκεται εντός οθόνης. Έτσι αρχικά πρέπει να ελέγξουμε αν το αντικείμενο “συγκρούεται” με το ορθογώνιο της κάμερας. Αν συγκρούεται, τότε σημαίνει ότι βρίσκεται εντός οθόνης. Μερικά αντικείμενα έχουν κάποια callback συνάρτηση για όταν εμφανιστούν. Δηλαδή όταν εμφανιστούν πρέπει να καλεστεί αυτή η συνάρτηση. Για αυτό και πρέπει να δούμε αν χρειάζεται να την καλέσουμε. Αυτή η συνάρτηση πρέπει να καλείται μόνο αν το αντικείμενο εμφανίστηκε και προηγουμένως ήταν εξαφανισμένο, για αυτό και πρέπει να ελέγξουμε αν ήταν πριν εμφανισμένο πριν την καλέσουμε. Εφόσον πρέπει να το καλέσουμε, τότε το κάνουμε μέσω της **show()**. Η show δεν καλεί απλά την συνάρτηση (εφόσον υπάρχει αυτή) αλλά εμφανίζει και το αντικείμενο, για αυτό και είναι απαραίτητη η κλήση της είτε υπάρχει η callback είτε όχι.

Σε επόμενη φάση, πρέπει να βάλουμε το αντικείμενο αυτό στην λίστα των αντικειμένων στην οθόνη **visibles**. Το αντικείμενο δεν μπορεί να μπει απλά στην λίστα, άλλα σε συγκεκριμένη σειρά. Αυτό γιατί τα αντικείμενα στον κόσμο πρέπει να σχεδιάζονται σε σειρά με βάση την θέση τους στον άξονα y. Γιατί όμως αυτό; Γιατί θέλουμε ένα αντικείμενο που βρίσκεται μπροστά από ένα άλλο να σχεδιάζεται δεύτερο για να μπορεί να φαίνεται έτσι. Προκειμένου να το πετύχουμε αυτό, τα αντικείμενα με μικρότερο y πρέπει να μπαίνουν πρώτα στην λίστα και αυτά με το μεγαλύτερο τελευταία. Έτσι και εδώ πρέπει να το βάλουμε στην σωστή σειρά. Εδώ υλοποιούμε ακόμη μια φορά την insert Sort. Για κάθε αντικείμενο που υπάρχει ήδη στην λίστα των **visibles** ελέγχουμε αν θα πρέπει να το βάλουμε σε αυτή την σειρά. Όταν βρούμε την σειρά, το τοποθετούμε εκεί καθώς και στο set **inView**.

Αν το αντικείμενο που έχουμε δεν βρίσκεται τελικά στην οθόνη, τότε θα πρέπει να κάνουμε το αντίστροφο από ότι κάναμε όταν θα το εμφανίζαμε. Επειδή υπάρχουν αντικείμενα με callback όταν εξαφανίζονται, θα πρέπει να καλεστεί αυτή η callback. Μέσω της **hide()** γίνεται η εξαφάνιση και η κλήση. Τέλος, θα πρέπει να αφαιρέσουμε το αντικείμενο από το **inView** εφόσον υπάρχει.

Πριν γίνει η ενημέρωση των αντικειμένων, θα πρέπει να τα εισάγουμε στο τετραδικό δένδρο του χάρτη. Η εισαγωγή τους γίνεται μόνο εφόσον μπορούν να προκαλέσουν συγκρούσεις ή να προκαλούν συμβάντα σύγκρουσης. Η εισαγωγή τους πρέπει να γίνει πριν την ενημέρωση γιατί στην αρχή της συνάρτησης είχαμε καθαρίσει το δένδρο και είναι άδειο. Έτσι αν κάποιο αντικείμενο ζητήσει να δει που συγκρούεται πριν γεμίσει το δένδρο με τα αντικείμενα, τότε θα παίρνει πάντα ότι δεν συγκρούεται πουθενά, κάτι που είναι λάθος. Αφού μπουν τα απαραίτητα αντικείμενα στο τετραδικό δένδρο, τότε μπορούμε να κάνουμε την τελευταία επανάληψη που είναι η πραγματική ενημέρωση των αντικειμένων.

Η κλήση αυτή ολοκληρώνει ένα Frame. Στην ουσία εδώ γίνεται όλη η δουλειά του παιχνιδιού και μετά το πέρας της κλήσης μπορεί να γίνει μια σχεδίαση.

4.5 Λειτουργία Draw

Η δεύτερη πιο σημαντική λειτουργία του χάρτη είναι η λειτουργία σχεδίασης η Draw. Σε αυτή την λειτουργία, πρέπει να σχεδιαστούν όλα τα Layers στην σωστή σειρά και ενδιάμεσα να σχεδιαστούν τα αντικείμενα που είναι να σχεδιαστούν.

Πίνακας 4.5.1: Η συνάρτηση Map::draw()

```
void Map::draw(const View& camera) const {
     // Get the number of tiles we need to draw on both axis
     int numTilesWidth = std::ceil((camera.getWidth() / tileWidth)
                 + 0.5f) + 1;
     int numTilesHeight = std::ceil((camera.getHeight() / tileHeight)
                 + 0.5f) + 1;

     // Get the camera location
     Float x = camera.getX();
     Float y = camera.getY();

     // clear the list of the Animated tiles
     tilesToBeUpdated.clear();

     // Draw all lower the layers
     for (const Layer* layer : lowerLayers) {
           layer->draw(x, y, numTilesWidth, numTilesHeight);
     }

     // draw all the visible objects
     for (auto obj : visibles) {
           obj->draw(x, y);
     }

     // draw all the upperLayers
     for (const Layer* layer : upperLayers) {
           layer->draw(x, y, numTilesWidth, numTilesHeight);
     }
}
```

Όπως είδαμε σε προηγούμενο κεφάλαιο, η συνάρτηση **draw(...)** της κλάσης **Layer** απαιτεί συντεταγμένες πάνω στο Layer και τον αριθμό των Tiles που θα σχεδιάσει σε κάθε άξονα. Για αυτό και χρειάζεται να υπολογίσουμε αυτούς τους αριθμούς πριν ξεκινήσουμε την σχεδίαση τους. Ο αριθμός αυτός πρέπει να καλύπτει τις διαστάσεις της κάμερας. Για αυτό και διαιρούμε το μέγεθος της με το μέγεθος των Tiles και προσθέτουμε 1 για να είμαστε σίγουροι ότι δεν θα έχουμε κενά.

Πριν κάνουμε οποιαδήποτε σχεδίαση, χρειάζεται να καθαρίσουμε τον πίνακα με τα Animated Tiles, τον **tilesToBeUpdated**. Αυτό γιατί κάθε φορά που σχεδιάζουμε ένα Layer, αυτό εντοπίζει ποια Tiles του είναι Animated και τα προσθέτει σε αυτόν τον πίνακα, όπως είδαμε στο αντίστοιχο κεφάλαιο. Αφού το κάνουμε και αυτό είμαστε έτοιμοι για τις σχεδιάσεις. Κατά την φόρτωση, τα Layers χωρίζονται σε “μπροστινά” και “πίσω” σύμφωνα με την τιμή του Property “Layer Priority” του χάρτη. Έτσι ξέρουμε ότι τα Layers του πίνακα **lowerLayers** πρέπει να σχεδιαστούν πρώτα, ύστερα όλα τα εμφανή αντικείμενα και τελευταία τα **upperLayer**.

Η συνάρτηση έχει και άλλο κώδικα για την σχεδίαση των συγκρούσεων των Tiles, αλλά επειδή υπάρχει για λόγους Debugging δεν αναφέρεται.

4.6 Εντοπισμός συγκρούσεων

Στην μηχανή, ο εντοπισμός συγκρούσεων μεταξύ αντικείμενων καθώς και μεταξύ των στατικών Tiles του κόσμου, είναι διασπασμένο μεταξύ 2 κομματιών, του χάρτη και των Lifeforms. Ο χάρτης όπως ξέρουμε περιέχει όλα τα δεδομένα, ακόμα και τα Lifeforms. Όμως επειδή τα Lifeforms είναι ελεύθερα, ο χάρτης δεν μπορεί να ξέρει πότε αυτά θα θέλουν να ελέγξουν για συγκρούσεις και έτσι αυτός αφήνεται να δέχεται αιτήματα για να ελέγξει όποτε χρειαστεί. Αυτό που κάνει ο χάρτης από την πλευρά του είναι να δέχεται ένα ορθογώνιο ή μια συντεταγμένη και να ελέγχει αν συγκρούεται με κάτι στον κόσμο εκείνη την στιγμή. Η βασική συνάρτηση που χρησιμοποιείται από τα Lifeforms είναι η **isRectangleColliding(...)**.

Πίνακας 4.6.1: Η συνάρτηση Map::isRectangleColliding() (Μέρος 1ο)

```
bool Map::isRectangleColliding(const Rectangle& rect, Object *owner,
          Rectangle::Surface surface) const {
     // Get a list of objects around the rectangle
     std::vector<Object*> objs = quadtree.getObjectsAt(rect);
     // get a collided flag
     bool collided = false;
     // for all the objects around the rectangle
     for (auto obj : objs) {
          // check if it is not the same person, lol
          if (obj != owner) {
               // check if the Rectangles overlap
               if (obj->getBounding().overlapsRectangle(rect)) {
                    // check if the other is collidable
                    if (obj->isCollideAble()) {
                         // if it is, then bingo!
                         collided = true;
                    }
                    // Trigger Collision Events
                    obj->onCollidedWith(owner);
                    owner->onCollidedWith(obj);
               }
          }
     }
     // If we collided with something
     if (collided) {
          // return true
          return true;
     }
```

Η συνάρτηση δέχεται σαν ορίσματα το ορθογώνιο που θα ελεγχθεί, το αντικείμενο που το έχει, καθώς και ένα προαιρετικό όρισμα για την πλευρά του ορθογωνίου που θα πρέπει να ελεγχθεί. Αν δεν οριστεί το όρισμα αυτό, παίρνει αυτόματα τιμή **Every**.

Αρχικά ζητάμε από το QuadTree μας φέρει τα αντικείμενα γύρω από το ορθογώνιο αυτό. Ο πίνακας που επιστρέφει περιέχει όλα τα αντικείμενα που πιθανόν να συγκρούεται το ορθογώνιο. Αν συγκρούεται με κάτι, αυτό δεν σημαίνει ότι πρέπει να επιστρέψουμε κατευθείαν true, καθώς πρέπει να δούμε με τι άλλο συγκρούεται. Ο λόγος που πρέπει να γίνει αυτό είναι γιατί υπάρχουν Callbacks για τις συγκρούσεις που πρέπει να καλούνται κάθε φορά που συγκρούεται ένα αντικείμενο με ένα άλλο. Στην επανάληψη για όλα τα αντικείμενα που πήραμε από το QuadTree, ελέγχουμε ότι αντικείμενο που έχουμε δεν είναι ο κάτοχος του ορθογωνίου. Αυτό γιατί δεν έχει νόημα να ελέγξουμε αν το αντικείμενο συγκρούεται με τον εαυτό του, καθώς αυτό θα μας φέρει πάντα αποτέλεσμα true. Έτσι αποφεύγουμε να ελέγξουμε τον κάτοχο, συγκρίνοντας τους Pointers. Αφού περάσουμε αυτόν τον έλεγχο, τότε ελέγχουμε αν το ορθογώνιο μας συγκρούεται με το αντικείμενο που έχουμε σε αυτή την επανάληψη. Αν συγκρούεται, τότε θέτουμε το Flag **collided** σε true και καλούμε τις CallBacks και για τα 2 αντικείμενα. Μετά το πέρας της επανάληψης θα έχουν καλεστεί οι Callbacks για όλα τα αντικείμενα που συγκρούστηκαν με το ορθογώνιο και το flag θα έχει την τιμή που πρέπει. Εάν αυτή έχει true, τότε δεν χρειάζεται να συνεχίσουμε περαιτέρω και επιστρέφουμε true.

Αν μετά το τέλος δεν έχουμε συγκρουστεί με τίποτα, τότε πρέπει να ελέγξουμε αν το ορθογώνιο συγκρούεται με κάποιο Tile του χάρτη. Αυτό γίνεται στο δεύτερο σκέλος της συνάρτησης.

Πίνακας 4.6.2: Η συνάρτηση Map::isRectangleColliding() (Μέρος 2ο)

```
     std::vector<Vector2> points;

// check which surface of our rectangle to check and add the appropriate
points
     switch (surface) {
     case Rectangle::Surface::Top:
           points.push_back(Vector2(rect.getX(), rect.getY()));
           points.push_back(
                       Vector2(rect.getX() + rect.getWidth(),
                                                   rect.getY()));
           break;
     case Rectangle::Surface::Left:
           points.push_back(Vector2(rect.getX(), rect.getY()));
           points.push_back(
```

```
                        Vector2(rect.getX(), rect.getY() +
                                                rect.getHeight()));
                break;
        case Rectangle::Surface::Right:
                points.push_back(
                        Vector2(rect.getX() + rect.getWidth(),
                                                        rect.getY()));
                points.push_back(
                        Vector2(rect.getX() + rect.getWidth(),
                                        rect.getY() +
                                                rect.getHeight()));
                break;
        case Rectangle::Surface::Bottom:
                points.push_back(
                        Vector2(rect.getX(), rect.getY() +
                                                rect.getHeight()));
                points.push_back(
                        Vector2(rect.getX() + rect.getWidth(),
                                        rect.getY() + rect.getHeight()));
                break;
        case Rectangle::Surface::Every:
                points.push_back(Vector2(rect.getX(), rect.getY()));
                points.push_back(
                        Vector2(rect.getX() + rect.getWidth(),
                                                        rect.getY()));
                points.push_back(
                        Vector2(rect.getX(), rect.getY() +
                                                rect.getHeight()));
                points.push_back(
                                Vector2(rect.getX() + rect.getWidth(),
                                        rect.getY() + rect.getHeight()));
                break;
        }

        // An another flag
        bool collided = false;

        // For all the points we have
        for (Vector2& point : points) {

                // Calculate the coordinates on the Sub-tile table
                unsigned int eX = point.getX() / subTileWidth;
                unsigned int eY = point.getY() / subTileHeight;

                // Check if the coordinates are within the bounds
                if (eX < subWidth && eY < subHeight) {
                        // merge the Collision value
                                collided |= collisions(eX, eY);
                }
        }
        // return the result
        return collided;
}
```

Στην περίπτωση που πρέπει να ελέγξουμε αν υπάρχει σύγκρουση με κάποιο Tile, τότε λαμβάνουμε υπόψη και την πλευρά του ορθογωνίου που θα ελέγξουμε σε αντί-

θεση με προηγουμένως. Ανάλογα την πλευρά που μας δίνεται να ελέγξουμε, προσθέτουμε τις γωνίες που περιέχει αυτή η πλευρά σε έναν πίνακα. Για παράδειγμα αν πρέπει να ελεγχθεί η πάνω πλευρά, τότε πρέπει να βάλουμε την πάνω αριστερή και δεξιά γωνία του ορθογωνίου στον πίνακα. Αυτό το κάνουμε γιατί ο έλεγχος δεν γίνεται με βάση το ίδιο ορθογώνιο αλλά με βάση των 4 σημείων που ορίζεται ο ορθογώνιο. Έτσι αφού έχουμε βάλει όλα τα σημεία που πρέπει, αρχίζουμε για κάθε σημείο να ελέγχουμε αν σε αυτό το σημείο στον χάρτη υπάρχει υπό-Tile που έχει σύγκρουση. Την τιμή σε αυτό το σημείο την κάνουμε OR και στο τέλος επιστρέφουμε την τιμή που έχει αυτό το flag.

Βάση της παραπάνω συνάρτησης λειτουργεί και η παραδοχή της για έλεγχο ενός σημείου παρά ενός ορθογωνίου. Η συνάρτηση είναι το 30% της συνολικής λειτουργίας του ελέγχου συγκρούσεων. Το υπόλοιπο τμήμα βρίσκεται εντός του κώδικα κίνησης του Lifeform που θα δούμε παρακάτω.

5 ΤΑ ΥΠΟΣΥΣΤΗΜΑΤΑ RPG

Σε αυτό το κομμάτι θα δούμε τις κλάσεις και τα αντικείμενα που στοιχίζουν το παιχνίδι σαν είδος RPG. Δηλαδή θα δούμε πως λειτουργούν αυτά τα κομμάτια προκειμένου ο χρήστης να έχει την εμπειρία ενός τέτοιου παιχνιδιού. Όπως είδαμε στην αρχή-αρχή, τα παιχνίδια τέτοιου είδους στρέφονται γύρω από έναν χαρακτήρα που “αναπτύσσεται” καθ' όλη την διάρκεια του παιχνιδιού. Η ανάπτυξη αυτή γίνεται μέσω διάφορων ενεργειών και αλληλεπιδράσεων με άλλους χαρακτήρες που ελέγχονται από το παιχνίδι. Η αλληλεπιδράσεις αυτές μπορεί να είναι είτε φιλικές είτε εχθρικές. Αυτό που μας νοιάζει στην τελική είναι ο πυρήνας όλου αυτού του συστήματος που είναι οι χαρακτήρες ή αλλιώς τα **Lifeforms**.

5.1 Η Abstract Κλάση Object

Η κλάση **Lifeform** είναι η βάση των χαρακτήρων του παιχνιδιού. Αυτή χρησιμοποιείται από ότι είναι έμβιο όν στο παιχνίδι. Όμως στο παιχνίδι υπάρχουν και αντικείμενα που δεν είναι έμβια. Για αυτό και χρειάζεται μια δομή για αυτά. Σε αυτή την δομή βασίζεται η **Lifeform** όπου και θα δίνει την “ζωή” σε αυτά που την κληρονομούν. Η κλάση αυτή που βασίζονται όλα τα αντικείμενα είτε έμβια είτε μη, είναι η **Object**.

Πίνακας 5.1.1: Η κλάση Object

```
class Object: public Drawable, public Updateable, public LuaPushable {
public:
     const Rectangle& getBounding() const;
     void setBounding(const Rectangle& bounding);
     const Rectangle& getTargetArea() const;
     const Vector2& getPosition() const;
     virtual void setPosition(Float x, Float y);
     const std::string& getName() const;
     void setName(const std::string& name);
     const PropertyList& getCustomProperties() const;

     Float getDistance(const Object& other) const;

     inline bool isCollideAble() const;
     void setCoolideAble(bool value);
     bool isTriggeringCollisionEvents() const;
     void setTriggeringCollisionEvents(bool value);
     inline bool isVisible() const;
     void setVisible(bool visible);

     inline void show() {
           visible = true;
```

```
            onShow();
      }

      inline void hide() {
            visible = false;
            onHide();
      }

      virtual void onCollidedWith(Object *other)=0;
      virtual void onClick(Object *other)=0;

      virtual void onShow() {

      }
      virtual void onHide() {

      }

      void addChildObject(const std::string& name, ChildObject *child);
      void addChildObject(Float x, Float y, const std::string& name,
                  const std::string& classPath,
                                    const std::string& animName);
      void removeChildObject(const std::string& name);
      ChildObject* getChildObject(const std::string& name);

      virtual void offsetPosition(Float dx, Float dy);
      void offsetShapes(Float dx, Float dy);
      void addChildToBeOffseted(const std::string& name);
      void removeChildToBeOffseted(const std::string& name);
      Vector2 getRenderPosition() const;

      virtual void accept(Visitor& visitor)=0;

      Object();
      Object(Float x, Float y);
      Object(Float x, Float y, int width, int height);
      virtual ~Object();
 protected:
      PropertyList customProperties;
      ZSmallMap<std::string, ChildObject*> childs;
      ZSmallMap<std::string, ChildObject*> childsToBeOffseted;
      std::vector<Shape*> shapesToBeOffseted;
      std::string name;
      Vector2 position;
      Rectangle bounding;
      Rectangle targetArea;
      bool collideAble;
      bool triggersCollisionEvents;
      bool visible;

      virtual void addShapesToBeOffseted();
 };
```

Σαν βάση, αυτή η κλάση έχει αρκετή λειτουργία σαν αποθηκευτικός χώρος που είναι κοινόχρηστος σε όλα τα αντικείμενα, έμβια και μη. Κάθε αντικείμενο έχει 3 απαραίτητα στοιχεία. Το ένα είναι η συντεταγμένες τους (**position**), το δεύτερο είναι η πε-

ριοχή που καλύπτει το αντικείμενο και ορίζεται από το ορθογώνιο **bounding** και η περιοχή που χρησιμοποιείται για την στόχευση του από τον παίχτη που ορίζεται από το **targetArea**. Υπάρχουν και τα δευτερεύοντα στοιχεία. Υπάρχει το όνομα του αντικειμένου **name**, καθώς και κάποιες προσαρμοσμένες ιδιότητες που ορίζονται από τον προγραμματιστή στο αντικείμενο **customProperties**.

Κάθε αντικείμενο μπορεί να συγκρουστεί στον κόσμο. Αυτό είναι ελέγξιμο από το μέλος **collideAble**. Αν αυτό το μέλος έχει τιμή true, τότε όταν στον κόσμο κάτι πάει να πέσει πάνω του, θα σταματήσει. Αν έχει false, τότε πρέπει να περάσει από μέσα του. Ανεξάρτητα αν το αντικείμενο θα προκαλεί εμφανές συγκρούσεις, μπορεί να προκαλέσει συμβάντα συγκρούσεων βάση της τιμής **triggersCollisionEvents**. Για να το καταλάβουμε καλύτερα αυτό πρέπει να δούμε ένα παράδειγμα. Πχ, μπορεί να έχουμε ένα αντικείμενο που το έχουμε σαν “πλακάκι-κουμπί” που όταν πατήσει ο παίχτης πάνω, πρέπει να του συμβεί κάτι. Αν κάνουμε το πλακάκι συγκρούσιμο μέσω του **collideAble=true**, τότε δεν θα μπορεί να πατήσει πάνω του αλλά να πάει τριγύρω του, κάτι που δεν είναι αυτό που θέλουμε. Για αυτό και θέλουμε ο παίχτης να περάσει από πάνω του σαν να μην είναι συγκρούσιμο αλλά να προκαλέσει συμβάν σύγκρουσης. Σε αυτό βοηθάει η διάσπαση που έχει γίνει. Θέτοντας **collideAble=false** και **triggersCollisionEvents=true** το πετυχαίνουμε αυτό.

Το μέλος **visible** ορίζει αν το αντικείμενο θα σχεδιαστεί όποτε του δοθεί η εντολή μέσω της **draw(...)**. Εάν αυτό έχει τιμή false, τότε κατά την draw() δεν πρέπει να γίνει η σχεδίαση. Η εναλλαγή της τιμής αυτής γίνεται μέσω των συναρτήσεων **setVisible(...)**, **hide()** και **show()**. Όπως βλέπουμε υπάρχει μια ιδιαιτερότητα σε αυτές τις συναρτήσεις πέρα από απλά την αλλαγή της τιμής. Ανάλογα τι τιμή θα δοθεί πρέπει να καλεστεί η **onShow()** ή **onHide()**. Οι τελευταίες είναι Callback που καλούνται όταν το αντικείμενο εμφανίζεται ή εξαφανίζεται με αυτόν τον τρόπο. Εδώ αυτές είναι κενές αλλά μπορούν να υλοποιηθούν από κλάσεις που κληρονομούν την Object.

Η κλάση φιλοξενεί και αντικείμενα παιδιά. Τα παιδιά είναι αντικείμενα παρόμοια με τα Objects μόνο που έχουν και Animation. Τα παιδιά αυτά αποθηκεύονται στον πίνακα **childs**. Ένα παιδί μπορεί να προστεθεί είτε αφού έχει κατασκευαστεί εκτός του αντικειμένου πατέρα, είτε να κατασκευαστεί εντός αυτού μέσω των 2 συναρτήσεων **addChildObject(...)**. Τα αντικείμενα αυτά όταν προστίθενται, μπαίνουν αυτόματα και στον πίνακα **childsToBeOffseted**. Ο πίνακας αυτός περιέχει όλα τα παιδιά που θα γίνουν offset μαζί με τον γονιό όταν γίνει αυτός offset. Τα παιδιά μπορούν να προ-

σθαφαιρούνται στον πίνακα αυτόν μέσω των συναρτήσεων **addChildeToBeOffseted()** και **removeChildToBeOffseted()**. Πέρα από τα παιδιά, τα σχήματα του αντικειμένου πρέπει να γίνουν Offset μαζί με το αντικείμενο. Τα σχήματα είναι όλα σχήματα που πάνε μαζί με το αντικείμενο, όπως στην περίπτωση του Object είναι τα ορθογώνια **bounding** και **targetArea** που πρέπει να ακολουθούν το αντικείμενο σε κάθε μετακίνηση που μπορεί να υποστεί. Αυτός ο πίνακας είναι ανοικτός, δηλαδή αν κάποια κλάση που θα κληρονομήσει την Object προσθέσει και άλλο σχήμα που πρέπει να ακολουθεί, τότε μπορεί να το προσθέσει κάνοντας Override την **addShapesToBeOffseted()**.

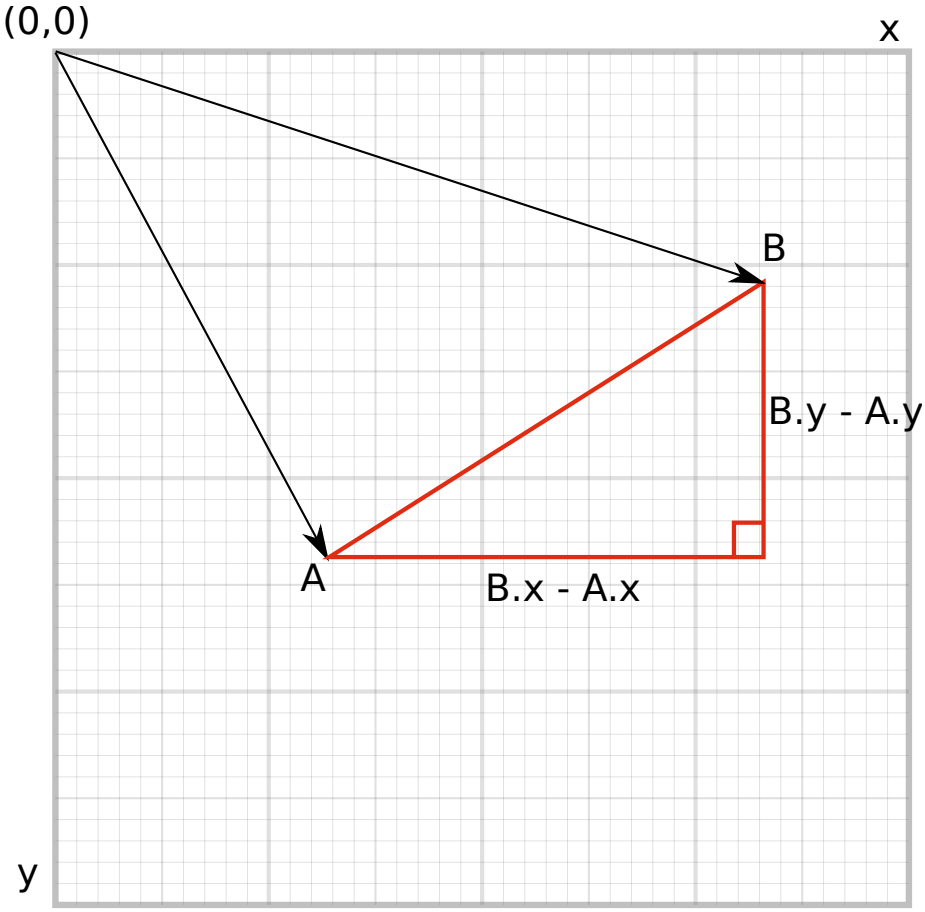
Πριν συνεχίσουμε, θα πρέπει να δούμε πως λειτουργούν κάποιες πολύ βασικές κλάσεις, την **Vector2** και **Rectangle**.

Πίνακας 5.1.2: Η κλάση Vector2

```
class Vector2: public LuaPushable {
public:
     inline void offset(Float dx, Float dy);
     inline void offset(const Vector2 other);

     Vector2 operator+(const Vector2& other) const;
     Vector2 operator-(const Vector2& other) const;
     Vector2 operator*(Float m) const;
     Vector2& operator*=(Float m);

     Float getLength() const {
           return std::sqrt((x * x) + (y * y));
     }

     Float getDistance(const Vector2& other) const {
           Float dx = x - other.x;
           Float dy = y - other.y;
           return std::sqrt(dx * dx + dy * dy);
     }

     void normalize() {
           Float len = getLength();
           x /= len;
           y /= len;
     }

     void round() {
           x = std::ceil(x + 0.5f);
           y = std::ceil(y + 0.5f);
     }

     Vector2();
     Vector2(Float x, Float y);
     ~Vector2();

     Float x;
```

```
    Float y;
};
```

Η κλάση **Vector2** ορίζει ένα διάνυσμα (x,y). Η πιο συχνή χρήση του είναι σαν συντεταγμένες πάνω στον κόσμο και πιο σπάνια σαν κατεύθυνση. Υπάρχουν οι απλές πράξεις γύρω από το διάνυσμα, καθώς και πιο σύνθετες. Η συνάρτηση **getDistance** υπολογίζει και επιστρέφει την απόσταση από δύο διανύσματα. Ο υπολογισμός γίνεται μέσω του πυθαγορείου θεωρήματος.



Εικόνα 5.1.1: Παράδειγμα απόστασης διανυσμάτων

Σύμφωνα με αυτό, στο παραπάνω παράδειγμα η απόσταση των σημείων (A,B) θα είναι:

$$Dist = \sqrt{(B.x - A.x)^2 + (B.y - A.y)^2}$$

και σύμφωνα με αυτόν έχουμε τον κώδικα της συνάρτησης. Για την παραδοχή της συνάρτησης **getLength()** θέλουμε να βρούμε το μήκος του διανύσματος. Εκεί θεωρούμε το δεύτερο διάνυσμα το (0,0) και απλοποιώντας έχουμε μόνο μια ρίζα.

Η συνάρτηση **normalize()** μετατρέπει το διάνυσμα σε μοναδιαίο. Αυτό θα μας βοηθήσει αργότερα για να υπολογίζουμε την κατεύθυνση ενός διανύσματος.

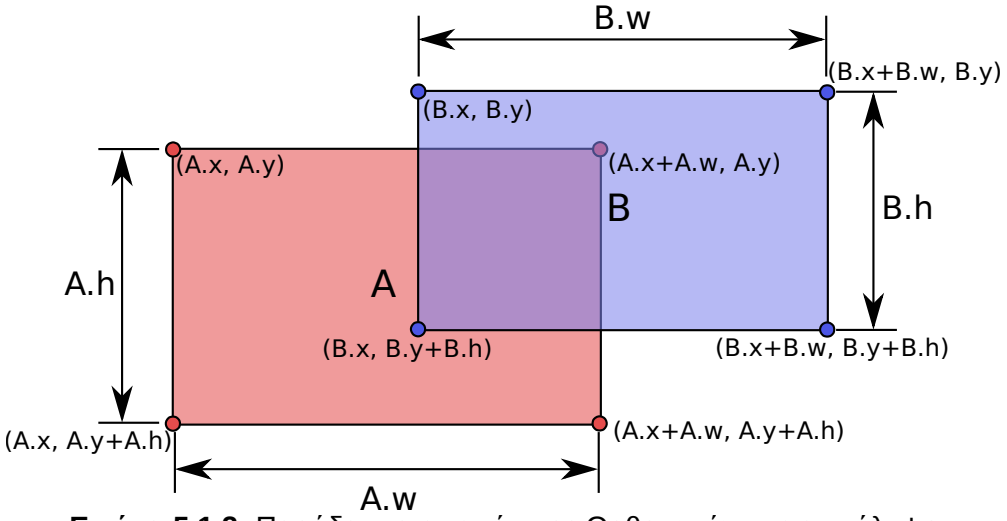
Η επόμενη κλάση που θα δούμε είναι η **Rectangle**. Η **Rectangle** ορίζει ένα ορθογώνιο από ένα σημείο και τις διαστάσεις. Για αυτό τον λόγο κληρονομεί την **Vector2**. Αυτό γίνεται μέσω της **Shape** που παρέχει μια πιο γενική βάση για όλα τα σχήματα που χρησιμοποιεί η μηχανή. Ο λόγος που θα γίνει ανάλυση μόνο στην **Rectangle** είναι επειδή αυτή χρησιμοποιείται πιο πολύ από όλες και χρησιμεύει στον εντοπισμό συγκρούσεων όπως θα δούμε σε παρακάτω κεφάλαιο. Για αυτό και έχει κάποιες πολύπλοκες συναρτήσεις για τον εντοπισμό εάν υπάρχει επικάλυψη μεταξύ ορθογωνίων (Overlap).

Πίνακας 5.1.3: Η κλάση Rectangle

```
class Rectangle: public Shape, public LuaPushable {
public:
	enum class Surface {
		Top, Left, Right, Bottom, Every
	};

	inline Float getWidth() const;
	inline Float getHeight() const;
	void setHeight(Float height);
	void setWidth(Float width);
	void setSize(Float width, Float height);

	inline bool overlapsPoint(Float x, Float y) const {
		return (x >= position.x) && (y >= position.y)
				&& ((position.x + width) >= x)
				 && ((position.y + height) >= y);
	}

	inline bool overlapsPoint(const Vector2& point) const {
		return (point.x > position.x) && (point.y > position.y)
				&& ((position.x + width) > point.y)
				&& ((position.y + height) > point.y);
	}

	inline bool overlapsRectangle(const Rectangle& other) const {
		return (other.position.x + other.width) > position.x
				&& other.position.x < (position.x + width)
				&& (other.position.y + other.height) > position.y
				&& other.position.y < (position.y + height);
	}

	inline bool overlapsWholeRectangle(const Rectangle& other) const {
		return (position.x < other.position.x)
		&& (position.y < other.position.y)
		&& ((position.x + width) > (other.position.x + other.width))
		&& ((position.y + height) > (other.position.y +other.height));
	}
```

```
     virtual void draw(Float x, Float y, Float rotation = 0.0f, Float
scale =
                  1.0f) const;

     Rectangle();
     Rectangle(const Vector2& location, Float width, Float height);
     Rectangle(Float x, Float y, Float width, Float height;
     ~Rectangle();
protected:
     Float width;
     Float height;

};
```

Οι συναρτήσεις που θα επικεντρωθούμε εδώ είναι οι **overlapsx(...)**. Θα δούμε αρχικά την πιο χρησιμοποιημένη που είναι η **overlapsRectangle(...)**. Αυτή πρέπει να ελέγξει αν το άλλο ορθογώνιο που παρέχεται μέσω του ορίσματος επικαλύπτει έστω και λίγο το δικό μας (**this**). Ο τρόπος που γίνεται ο έλεγχος είναι μέσω συγκρίσεων των 4 σημείων του ορθογωνίου σε σχέση με 4 σημεία του άλλου. Μπορεί στα αντικείμενα των ορθογωνίων να έχουμε μόνο τις συντεταγμένες της πάνω αριστερά γωνίας και τις διαστάσεις αλλά αυτά είναι υπεραρκετά για να υπολογίσουμε τις συντεταγμένες των άλλων 3 σημείων όπως βλέπουμε στο παρακάτω σχήμα.



Εικόνα 5.1.2: Παράδειγμα απεικόνισης Ορθογωνίων με επικάλυψη

Για να καταλάβουμε καλύτερα το παράδειγμα μαζί με τον κώδικα, το ορθογώνιο A είναι το **this** και το B είναι το **other**. Αυτό που κάνουμε είναι να προσπαθήσουμε να επιβεβαιώσουμε την κατάσταση που βρίσκεται στο σχήμα όπου έχουμε επικάλυψη.

Όπως πάντα, θεωρούμε ότι τα θετικά για τον άξονα **x** αυξάνονται προς τα δεξιά και για τον **y** προς τα κάτω. Διαβάζοντας το σχήμα έχουμε:

- Το **B.x+B.w** έχει μεγαλύτερη τιμή από το **A.x**
- Το **B.x** έχει μικρότερη τιμή από το **A.x+A.w**

Αν ισχύουν τα 2 παραπάνω τότε σημαίνει ότι στον άξονα των **x** τα 2 ορθογώνια επικαλύπτονται. Αυτό όμως δεν σημαίνει ότι επικαλύπτονται και στον άξονα των **y**, για αυτό και πρέπει να ελέγξουμε και εκεί. Στον άξονα των **y** έχουμε:

- Το **B.y+B.h** έχει μεγαλύτερη τιμή από το **A.y**
- Το **B.y** έχει μικρότερη τιμή από το **A.y+A.h**

Αν ισχύουν τα 2 παραπάνω τότε σημαίνει ότι στον άξονα των **y** έχουμε επικάλυψη στα 2 ορθογώνια. Αν έχουμε επικάλυψη και στους 2 άξονες τότε τα ορθογώνια όντως επικαλύπτονται.

Ένα ερώτημα που θέτεται είναι γιατί επιλέξαμε τα συγκεκριμένα σημεία να ελέγξουμε και δεν πήραμε άλλα στους αντίστοιχους άξονες. Ο λόγος είναι ότι μόνο για αυτά τα σημεία ισχύουν οι έλεγχοι ανεξάρτητα το που βρίσκονται τα ορθογώνια. Αυτό που εννοούμε είναι ότι αν παίρναμε άλλα σημεία και το Β ορθογώνιο βρισκόταν αριστερά του Α τότε κάποιος έλεγχος δεν θα ίσχυε. Ακόμα και αν τα ορθογώνια όντως επικαλύπτονταν η συνάρτηση θα επέστρεφε false λόγου αυτού. Ένας τρόπος για να αποδείξουμε ότι αυτά τα σημεία θα μας φέρουν τα σωστά αποτελέσματα σε κάθε περίπτωση, είναι να καταγράψουμε τι ισχύει σε κάθε άξονα για κάθε από τις 4 περιπτώσεις (το πού βρίσκεται το Β σε σχέση με το Α). Στο τέλος παίρνοντας τις κοινές συνθήκες που ισχύουν, παίρνουμε τους παραπάνω 4 ελέγχους.

Στην περίπτωση που πρέπει να ελέγξουμε ένα σημείο αν βρίσκεται εντός του ορθογωνίου μέσω της **overlapsPoint(...)**, τα πράγματα είναι πιο απλά καθώς δεν έχουμε να αντιμετωπίσουμε περιπτώσεις για το που βρίσκεται το σημείο σε σχέση με το ορθογώνιο. Η περίπτωση είναι μία: είτε είναι μέσα είτε όχι. Αυτό που ελέγχουμε σε κάθε άξονα είναι αν το σημείο βρίσκεται μεταξύ των σημείων που ορίζεται από την πλευρά του ορθογωνίου. Πχ αν η τιμή του **y** του σημείου βρίσκεται μεταξύ τις τιμές **A.y** και **A.y+A.h**. Πιο γενικά για βρίσκεται εντός πρέπει:

$$A.y<y<(A.y+A.h)$$
$$A.x<x<(A.x+A.w)$$

Περίπου το ίδιο ισχύει και για την περίπτωση που θέλουμε να ελέγξουμε αν ένα ορθογώνιο βρίσκεται ολόκληρο εντός μέσω της **overlapsWholeRectangle(...)**. Σε

αυτή την περίπτωση ελέγχουμε σε κάθε άξονα αν τα σημεία του άλλου ορθογωνίου βρίσκονται ενδιάμεσα του δικού μας.

5.2 Η κλάση Lifeform

Τώρα θα δούμε το πιο σημαντικό κομμάτι του RPG μέρους της μηχανής, την κλάση **Lifeform**. Αυτή η κλάση κληρονομεί την **Object** και στην ουσία προσθέτει πάνω της την “ζωή”. Με αυτό εννοούμε ότι τα αντικείμενα της μπορούν να έχουν βούληση, να κινούνται και ζουν με τα άλλα αντίστοιχα. Το τελευταίο σημαίνει ότι κάθε Lifeform ανήκει κάπου και έχει σχέσεις είτε φιλίας είτε εχθρικές με κάποια άλλα. Για αυτούς τους λόγους η κλάση αυτή είναι η πιο πολύπλοκη σε όλη την μηχανή.

Τα κυριότερα κομμάτια της κλάσης είναι: ο μηχανισμός των Animation, ο μηχανισμός των καταστάσεων, ο μηχανισμός της κίνησης, τα AI, η ομάδα που ανήκει και οι λίστες των Attributes και Abilities. Όλα αυτά συγκροτούν ένα ολοκληρωμένο Lifeform.

Παρακάτω θα δούμε τον ορισμό της κλάσης αλλά δεν θα βρίσκονται όλες οι συναρτήσεις μέλη της κλάσης παρά οι πιο σημαντικές, λόγω ότι η κλάση είναι μεγάλη.

Πίνακας 5.2.1: Η κλάση Lifeform

```
class Lifeform: public Object {
public:

     virtual void update(Float elapsedTime);
     virtual void draw(Float x, Float y, Float rotation = 0.0f,
                                              Float scale = 1.0f) const;

     virtual void onCollidedWith(Object *other);
     virtual void onClick(Object *other);
     virtual void onShow();
     virtual void onHide();
     void moveToPosition(Float x, Float y);
     void startMovingToDirection(const Vector2& pos);
     void stopMoving();
     void setClass(const std::string& path);
     void setClass(const LifeformClass& lifeClass);
     void setAnimationClass(const std::string& path);
     void setActionAnimation(const std::string& name);
     bool isAnimationFinished() const;
     void addOffAnimation(const Animation& animation, Float dx = 0.0f,
           Float dy = 0.0f, AnimationHandler::QueuePlace queue =
                 AnimationHandler::QueuePlace::Front);
     void resetState();
     bool canMove() const {
           return (state.getAction() != LifeformState::Action::Casting);
     }
     inline bool isAlive() const {
           return state.getAction() != LifeformState::Action::Dead
                 && state.getAction() != LifeformState::Action::Dying;
```

```
    }
    Attribute& getAttribute(const std::string& name);
    void addAttributeModifier(const std::string& attributeName,
                              Modifier *mod);
    void addAttributeModifier(const std::string& attributeName,
                const std::string& modifierName, Float value,
                enum Modifier::Type type);
    void removeAttributeModifier(const std::string& attributeName,
                const std::string& modifierName);
    void addEffect(Effect *eff);
    bool canReceiveEffect(const EffectClass& effClass) const;
    bool isHostileWith(Lifeform* other) const;
    Ability::Result invokeAbility(const std::string& name) const;
    void addAbility(const std::string& abilityClass, int level);
    void addAbility(const AbilityClass& abilityClass, int level);
    virtual void teleportToCordinates(Float x, Float y);
    virtual void die();
    virtual void resurect();
    void setCastingState();
    void hardReset();
    void fullResetState();
    void resetShapes();
    void face(const Vector2 &position);
    bool isFacing(Object *other) const;

    Lifeform();
    Lifeform(const std::string& lifeClass, Float x, Float y);
    Lifeform(const LifeformClass& lifeClass, Float x, Float y);
    virtual ~Lifeform();
protected:

    mutable ZMap<std::string, Attribute> attributes;
    ZSmallMap<std::string, Modifier*> modifiers;
    ZSmallMap<std::string, ActiveAbility*> activeAbilities;
    ZSmallMap<std::string, PassiveAbility*> passiveAbilities;
    ZSmallMap<std::string, Regeneration*> regenerations;
    std::list<Effect*> effects;
    ZSmallSet<std::string> nonStackableEffects;
    AnimationHandler animHandler;
    const LifeformClass *lifeformClass;
    Ellipsis shadow;
    Vector2 targetPosition;
    Vector2 direction;
    LuaTable table;
    LuaFunctor onCollisionF;
    LuaFunctor onClickF;
    Lifeform *target;
    Behaviour *generalAI;
    Behaviour *movementAI;
    const Faction* faction;
    int level;
    LifeformState state;
    LifeformState prevState;
    bool inCombat;
    bool dirty;

    void setIdleState();
    void setDeadState();
    void setMovingState();
```

```
      void setState(const LifeformState& st);
      void setState(LifeformState::CombinedState newState);
      virtual void updateMovement(Float elapsedTime);
      void updateEffects(Float elapsedTime);
      void updateAbilities(Float elapsedTime);
      virtual void addShapesToBeOffseted();
      void calculateDirection();

      friend class LifeformClass;
};
```

a) Λειτουργίες

Το βασικό στοιχείο του Lifeform είναι η κατάστασή του **LifeformState**. Η κατάσταση είναι ένας αριθμός που ορίζει το τι κάνει το Lifeform και σε ποια κατεύθυνση κοιτάει. Αυτά τα δυο ενσωματώνονται σε έναν αριθμό εσωτερικά του **LifeformState**. Εδώ βλέπουμε ότι έχουμε 2 μέλη τύπου LifeformState, το **state** και το **prevState**. Το πρώτο περιέχει την κατάσταση που έχει το Lifeform αυτή την στιγμή και το άλλο την προηγούμενη κατάσταση. Κάθε φορά που μπαίνει μια καινούργια κατάσταση στο **state** η παλιά του πηγαίνει στο **prevState**. Αυτό θα μας χρησιμεύει αργότερα όταν θα χρειαστεί να επαναφέρουμε την προηγούμενη κατάσταση του Lifeform. Όπως είπαμε μέχρι στιγμής, η κατάσταση είναι ένα Bitfield που περιέχει την κατάσταση και την κατεύθυνση.

Πίνακας 5.2.2: Η κλάση LifeformState

```
class LifeformState {
public:
      typedef int CombinedState;

      enum Direction {
            Down = 1, Up, Left, Right
      };
      enum Action {
            Idle = (1 << 4),
            Moving = (2 << 4),
            Dying = (3 << 4),
            Dead = (4 << 4),
            Casting = (5 << 4)
      };

      void set(int combinedState) {
            if ((combinedState & 0xF) > 0
                        && (combinedState & 0xF) <= Direction::Right) {
                  m_direction = (Direction) (combinedState & 0xF);
            }
            if ((combinedState & 0xF0) >= Action::Idle
                        && (combinedState & 0xF0) <= Action::Casting) {
```

```
                m_action = (Action) (combinedState & 0xF0);
            }
        }

        CombinedState getCombinedState() const {
            return m_direction | m_action;
        }

        void setAction(Action action);
        void setDirection(Direction direction);
        inline Action getAction() const;
        inline Direction getDirection() const;
        void operator=(int combinedState);

        LifeformState();
        LifeformState(int combinedState);
        ~LifeformState();
    private:
        Direction m_direction;
        Action m_action;
    };
```

Οι καταστάσεις που μπορεί να έχει ένα Lifeform είναι η ακινησία, να κινείται, να πεθαίνει, να είναι νεκρό ή να εκτελεί κάποιο Ability. Οι κατευθύνσεις που μπορεί να έχει είναι γνωστές. Αυτές οι τιμές είναι έτσι ώστε να μπορούν να συνδυαστούν σε έναν αριθμό που να αντιπροσωπεύει αυτόν τον συνδυασμό και να μπορεί να παρθούν μετά τα επιμέρους κομμάτια μέσω λογικών πράξεων, όπως βλέπουμε στην συνάρτηση **set(...)**.

Οι τρόποι που γίνεται η αλλαγή της κατάστασης είναι μέσω ειδικών συναρτήσεων που έχουμε όπως πχ **setCastingState()** που συνήθως χρειάζονται να κάνουν και κάτι άλλο πέρα από την αλλαγή της κατάστασης (συνήθως να αλλάξουν το Animation). Ας δούμε το παράδειγμα της **fullResetState()** που ο σκοπός της είναι να επαναφέρει την κατάσταση του Lifeform στην προηγούμενη.

Πίνακας 5.2.3: Η συνάρτηση Lifeform::fullResetState()

```
void Lifeform::fullResetState() {
    // Change the State according to the Previous state
    switch (prevState.getAction()) {
    case LifeformState::Action::Idle:
        setIdleState();
        break;
    case LifeformState::Action::Moving:
        setMovingState();
        break;
    default:
        setIdleState();
        break;
```

```
	}
}
```

Στην παραπάνω ελέγχουμε την τιμή της προηγούμενης και ανάλογα καλούμε την συνάρτηση που πρέπει. Βλέπουμε όμως ότι δεν μας ενδιαφέρουν όλες οι καταστάσεις παρά μόνο αυτές που έχει νόημα να θέσουμε. Πχ αν πριν εκτελούσε κάποιο Ability, δεν θα είχε νόημα να βάλουμε το Lifeform σε αυτή την κατάσταση καθώς το Ability έχει τελειώσει.

Τώρα ας δούμε πως ενημερώνεται το Lifeform μέσω της **update(...)**.

Πίνακας 5.2.4: Η συνάρτηση Lifeform::update()

```
void Lifeform::update(Float elapsedTime) {
	// If this Lifeform is alive
	if (isAlive()) {

		// Update the AI Handlers if defined
		if (generalAI != nullptr) {
			generalAI->update(elapsedTime);
		}

		if (movementAI != nullptr) {
			movementAI->update(elapsedTime);
		}

		// Check if moving and update it
		updateMovement(elapsedTime);
		// Update the effects this Lifeform has (durations, etc)
		updateEffects(elapsedTime);
		// Update the abilities (cooldown, etc)
		updateAbilities(elapsedTime);

		for (auto& child : childs) {
			child.second->update(elapsedTime);
		}
	} else {
		// else if is not alive

		// if is Dying, then this means the die() function called
		// and the death animation is invoked. When the death
		// animation is finished, we should set the according dead
		// state
		if (state.getAction() == LifeformState::Action::Dying
			&& !animHandler.getMainAnimationPlayer().isPlaying()) {
			// set the acording animations/states to Dead
			setDeadState();
		}
	}
	// Update the animations
	animHandler.update(elapsedTime);
}
```

Η συνάρτηση αυτή αναλαμβάνει όλη την λογική του Lifeform. Αρχικά ελέγχουμε αν το Lifeform είναι ζωντανό. Εφόσον είναι, τότε χρειάζεται να ενημερώσουμε τα αντικείμενα του γενικού AI και του AI κίνησης εφόσον υπάρχουν. Η ύπαρξη τους εξαρτάται από το LifeformClass που αποδίδεται στο Lifeform. Μετά από αυτά καλούμε τις συναρτήσεις ενημέρωσης της κίνησης, των επιδράσεων, των ικανοτήτων και τελικά όλων των αντικειμένων παιδιών. Το πως λειτουργούν αυτές οι συναρτήσεις θα δούμε παρακάτω.

Αν το αντικείμενο δεν είναι ζωντανό, τότε μάλλον είτε πεθαίνει είτε είναι νεκρό. Όταν σε ένα Lifeform η ζωή φτάσει στο < 0 τότε μπαίνει στην κατάσταση **Dying**. Σε αυτή την κατάσταση το Lifeform παίρνει το Primitive Animation **Dying**. Θεωρείται νεκρό αλλά αυτό το είναι το τελευταίο Animation που κάνει πριν μπει στην κατάσταση **Dead** και πάρει το Primitive Animation **Dead**. Για αυτό και όσο η κατάσταση είναι **Dying** πρέπει να ελέγχουμε πότε το Animation που τρέχει αυτή την στιγμή θα τελειώσει. Όταν αυτό τελειώσει, τότε σημαίνει ότι το Dying Animation τελείωσε και μπορούμε να μπούμε στην **Dead** state και Animation.

b) Κίνηση

Τώρα θα δούμε τι κάνει η **updateMovement()**. Όπως είπαμε τα Lifeforms μπορούνε να κινούνται στον κόσμο. Για να κινηθούν, πρέπει να τους οριστεί η τοποθεσία που πρέπει να πάνε. Αυτή βρίσκεται στο μέλος **targetPosition**. Η κίνηση μπορεί να γίνει μέσω της συνάρτησης **moveToPosition(...)**. Αυτή καλείται να θέσει την τοποθεσία στόχο που πρέπει να πάει το Lifeform καθώς και να υπολογιστεί η κατεύθυνση που πρέπει να πάρει. Πέρα από αυτό, επειδή το Lifeform θα αρχίσει να κινείται, θα πρέπει να αλλάξει η κατάσταση του και να μπει στο **Moving**.

Πίνακας 5.2.5: Η συνάρτηση Lifeform::moveToPosition()

```
void Lifeform::moveToPosition(Float x, Float y) {
    if (canMove()) {
          // Set the targetPosition
          targetPosition.set(x, y);
          // Calculate the Normalized Direction
          calculateDirection();

          // What state we have now?
          setMovingState();
    } else if (state.getAction() == LifeformState::Action::Casting) {
          // XXX: Hack to reduce the lag felt when you cast from idle
          // state and a move was chosen while casting was occurring,
          // resulting staying idle rather than moving to the desired
```

```
            // location. Its not a bug, but the player will not like it
            prevState = LifeformState::Action::Moving;
            // Set the targetPosition
            targetPosition.set(x, y);
            // Calculate the Normalized Direction
            calculateDirection();
        }
    }
```

Καλώντας αυτήν την συνάρτηση δίνουμε εντολή στο Lifeform να αρχίσει να κινείται προς τις συντεταγμένες μέχρι να φτάσει εκεί. Αρχικά ελέγχουμε αν μπορούμε να κινηθούμε. Ο μόνος απαγορευτικός παράγοντας που θα μας αποτρέψει να κινηθούμε, είναι το Lifeform να βρίσκεται σε κατάσταση εκτέλεσης κάποιου Ability. Η σύμβαση της μηχανής είναι ότι ένα Lifeform θα πρέπει να τελειώσει την εκτέλεση ενός Ability πριν κάνει κάποια άλλη ενέργεια που του δόθηκε να κάνει κατά την διάρκεια. Για αυτό και η **canMove()** ελέγχει την κατάσταση του Lifeform και εφόσον δεν είναι **Casting** επιστρέφει true αν μπορούμε να κινηθούμε. Εφόσον μπορούμε, τότε θέτουμε στο μέλος **targetPosition** το σημείο που πρέπει να πάει το Lifeform. Στην συνέχεια πρέπει να υπολογίσουμε την κατεύθυνση που πρέπει να αρχίσει να κινείται το Lifeform. Αυτό γίνεται με την **calculateDirection()**.

Πίνακας 5.2.6: Η συνάρτηση Lifeform::calculateDirection()

```
void Lifeform::calculateDirection() {
    // Get the difference between the targetPosition and current
    // Position
    direction = targetPosition - position;
    // Normalize the vector to get the direction
    direction.normalize();
}
```

Αυτό που γίνεται εδώ είναι γνωστό. Θέτουμε στο μέλος **direction** την διαφορά της τοποθεσίας στόχου και της τωρινής τοποθεσίας του Lifeform. Κάνοντας **normalize()** το διάνυσμα που πήραμε σαν αποτέλεσμα, μας δίνεται ένα μοναδιαίο που θεωρείται η κατεύθυνση που πρέπει να κινηθεί το Lifeform για να πάει στο **targetPosition**. Τα μέλη του **x,y** του **direction** θα έχουν δεκαδικές τιμές από -1.0 έως 1.0. Αυτές οι τιμές είναι το τι πρέπει να προσθέσουμε στο **x,y** της θέσης του Lifeform (**position**) προκειμένου να μετατοπιστεί προς αυτή την κατεύθυνση.

Αφού υπολογίσουμε και το **direction**, τότε μπορούμε να θέσουμε και την κατάσταση του Lifeform σε **Moving** μέσω της **setMovingState()**.

Πίνακας 5.2.7: Η συνάρτηση setMovingState()

```
void Lifeform::setMovingState() {
    bool animationShouldChange = false;

    {
        // get the absolute value of the direction on both axis
        Float xValue = std::abs(direction.getX());
        Float yValue = std::abs(direction.getY());

        // The animation to change
        LifeformState::Direction animDir = state.getDirection();
        // Check if we need to use a Vertical or Horizontal animation
        // we actually see where we lean more, on the x axis or the y
        // axis
        if (xValue > yValue) {
            // If we need a horizontal, check if we need right or
            // left
            if (direction.getX() > 0.0f) {
                animDir = LifeformState::Direction::Right;
            } else {
                animDir = LifeformState::Direction::Left;
            }
        } else {
            // if we need vertical check if we need up or down
            if (direction.getY() > 0.0f) {
                animDir = LifeformState::Direction::Down;
            } else {
                animDir = LifeformState::Direction::Up;
            }
        }
        // If state is not already to the one we calculated
        if (state.getDirection() != animDir) {
            // Set the direction to the correct one
            state = animDir;
            // Set to change the Animation
            animationShouldChange = true;
        }
    }
    if (state.getAction() != LifeformState::Action::Moving
            || animationShouldChange) {
        // Set that we are moving now
        setState(LifeformState::Action::Moving);
        animHandler.setAnimation(state.getCombinedState());
    }
    // Start the animations (if not already started)
    animHandler.getMainAnimationPlayer().play();
}
```

Τα πράγματα όπως βλέπουμε ξεφεύγουν από απλή αλλαγή της κατάστασης. Συνήθως κάθε φορά που αλλάζουμε μια κατάσταση, χρειάζεται να αλλάξουμε και το Animation στο αντίστοιχο που αντιπροσωπεύει την κατάσταση. Εδώ όμως για την κίνηση έχουμε 4 διαφορετικά για τις 4 διαφορετικές κατευθύνσεις. Για αυτό και πρέπει να επιλέξουμε πιο από τα 4 θα αλλάξουμε. Αυτό που πρέπει είναι όταν το Lifeform κι-

νείται προς τα δεξιά, να μπει το Animation που κινείται προς τα δεξιά κοκ. Όμως το Lifeform δεν κάνει 4 απλές κινήσεις αλλά κινείται βάση κάποιας κατεύθυνσης. Δηλαδή μπορεί να κινηθεί προς κάτω-δεξιά. Αν γίνει αυτό, τότε τι Animation θα βάλουμε; Θα πρέπει να διαλέξουμε ανάμεσα στο Animation που κινείται δεξιά και σε αυτό που κινείται κάτω. Η διαλογή θα γίνει βάση το που κινείται πιο κοντά. Δηλαδή αν το Lifeform κινείται πιο δεξιά από ότι κάτω, τότε φυσικό είναι να διαλέξουμε να βάλουμε το Animation που κινείται δεξιά, κοκ. Για αυτό και πρέπει να διαλέξουμε βάση των τιμών της κατεύθυνσης που κινείται το Lifeform (**direction**).

Αρχικά παίρνουμε τις τιμές της κατεύθυνσης και στους δυο άξονες και υπολογίζουμε τις απόλυτες τιμές. Συγκρίνοντας τις τιμές αυτές βρίσκουμε σε ποιόν άξονα κινούμαστε πιο κοντά. Μόλις βρούμε ποιόν άξονα θα χρησιμοποιήσουμε για το Animation, τότε πρέπει να βρούμε σε ποια φορά. Δηλαδή αν κινούμαστε κοντά στον κάθετο άξονα πρέπει να ελέγξουμε αν το y της κατεύθυνσης είναι αρνητικό η θετικό. Αν είναι θετικό κινούμαστε προς τα κάτω και αν είναι αρνητικό προς τα πάνω. Αντίστοιχα ισχύει και για τον οριζόντιο άξονα. Εφόσον ξεκαθαρίσουμε ποιο Animation θα χρησιμοποιήσουμε, τότε πρέπει να δούμε αν θα χρειαστεί να το αλλάξουμε. Θα το αλλάξουμε μόνο αν δεν το έχουμε ήδη στο Lifeform. Αυτό γιατί κάθε φορά που αλλάζουμε ένα Animation, γίνεται reset στο πρώτο του Frame. Έτσι ακόμα και αν αλλάζουμε στο ίδιο Animation κάθε φορά που διαλέγουμε να κινηθούμε, το Animation θα γίνεται reset στην αρχή του δίνοντας μια άσχημη εικόνα στον χρήστη. Εάν χρειαστεί να το αλλάξουμε, τότε αλλάζουμε την κατεύθυνση στην νέα και θέτουμε το flag για να το αλλάξουμε.

Σε επόμενη φάση χρειάζεται να ελέγξουμε αν πρέπει να αλλάξουμε και την κατάσταση πέρα από την κατεύθυνση. Εάν η κατάσταση μας δεν είναι **Moving** ή πρέπει να αλλάξουμε το Animation, τότε αλλάζουμε την κατάσταση και θέτουμε το νέο Animation από την νέα κατάσταση. Στο τέλος παίζουμε το Animation εάν δεν έχει ήδη ξεκινήσει από προηγούμενο Frame.

Είδαμε τι γίνεται στην **moveToPosition(...)** όταν μπορούμε να κινηθούμε. Όταν δεν μπορούμε όμως πρέπει να κάνουμε κάποιον επιπλέον έλεγχο. Υπάρχει μια περίπτωση που το Lifeform δεν μπορεί να κινηθεί λόγω ότι βρίσκεται σε κατάσταση **Casting**. Η σύμβαση της μηχανής θέτει ότι το Lifeform μένει ακίνητο όσο βρίσκεται σε αυτή την κατάσταση. Υπό κανονικές συνθήκες ότι εντολή δοθεί όσο βρίσκεται σε αυτή την κατάσταση θα αγνοηθεί. Πολλές φορές όμως η ακινησία κάνει ευάλωτο τον παίχτη και χρειάζεται μόλις τελειώσει το Casting να κινηθεί αμέσως σε ασφαλές σημείο. Σε

τέτοιες περιπτώσεις ο χρήστης δίνει την εντολή για την κίνηση πάρα πολλές φορές (Spam) όσο βρίσκεται σε αυτή την κατάσταση προκειμένου ο παίχτης να φύγει αμέσως μόλις τελειώσει. Συνήθως ο χρήστης κλικάρει την τοποθεσία πολλές φόρες μέχρι να κινηθεί. Εδώ υπάρχει ένα πρόβλημα. Όπως είπαμε υπό κανονικές συνθήκες οι εντολές κίνησης αγνοούνται όσο η **canMove()** επιστρέφει false (δηλαδή είναι σε κατάσταση Casting ή νεκρό). Αυτές οι εντολές κίνησης δεν αποθηκεύονται. Δηλαδή όταν τελειώσει το Casting το τελευταίο κλικ για κίνηση που έκανε ο χρήστης δεν θα ληφθεί υπόψη. Ο χρήστης όπως είπαμε θα συνεχίσει να κάνει κλικ μέχρι να κινηθεί ο παίχτης του. Αυτό σημαίνει ότι αφού τελειώσει το Casting, μέχρι ο χρήστης να κάνει το επόμενο κλικ, ο χαρακτήρας θα μείνει ακίνητος. Αυτό δίνει μια δυσάρεστη αίσθηση καθυστέρησης στην ανταπόκριση του χαρακτήρα. Δεν πρόκειται για Bug, απλά ο χρήστης έχει την απαίτηση ο χαρακτήρας του να κινηθεί αμέσως μετά το Casting.

Για αυτό επινοήθηκε το εν λόγο hack που ελέγχει αν η κατάσταση που απέτρεψε την κίνηση είναι το Casting. Ξέρουμε ότι κάθε φορά που ο παίχτης τελειώνει Casting κάνει επαναφορά στην προηγούμενη κατάσταση που είχε πριν μπει σε κατάσταση Casting. Εκμεταλλευόμενοι αυτό, παραποιούμε την προηγούμενη κατάσταση και την θέτουμε να είναι **Moving** ακόμα και αν μην ήταν. Θέτουμε το **targetPosition** στις συντεταγμένες που όρισε ο χρήστης και υπολογίζουμε την κατεύθυνση. Όλο αυτό θα έχει ως αποτέλεσμα όταν γίνει η επαναφορά στο προηγούμενο State, ο παίχτης να μπει σε κατάσταση κίνησης με τα δεδομένα που ορίσαμε, ακόμα και αν δεν καλεστεί η **moveToPosition()** όπως θα δούμε παρακάτω.

Αφού ολοκληρωθεί η **moveToPosition()**, στο επόμενο Frame θα λειτουργήσει στη συνάρτηση της **update()** που είδαμε, η **updateMovement()**.

Πίνακας 5.2.8: Η συνάρτηση Lifeform::updateMovement()

```
void Lifeform::updateMovement(Float elapsedTime) {
    // If movement is enabled (Eg. Not casting, or has target location)
    if (state.getAction() == LifeformState::Action::Moving) {
          // Get the speed attribute and multiply it by the elapsed
          // time. We multiply it to ensure that the speed of the
          // movement will  remain constant on FPS changes
          Float speed = attributes["Speed"].getValue() * elapsedTime;
          // If we are near the target location
          if (position.getDistance(targetPosition) <= (speed + 0.1f)) {
                // Stop Moving and set the associated primitive
                // animations
                setIdleState();
          } else {
```

```
                // Else if we are far from target Location

                // Calculate the offsets to add to get the next Position
                // (direction) has the Normalized vector from the
                // currentPosition to target Position So multiplying
                // each delta with speed gives us the according offsets
                // to add to current x,y to retrieve the the next frame
                // position
                Float off_x = direction.getX() * speed;
                Float off_y = direction.getY() * speed;

                bool moving = true;

                // Retrieve the current Map
                Map& curMap =
    WorldManager::getInstance().getCurrentMap();

                {
                      // Get a copy of Bound Rectangle
                      Rectangle rect = bounding;

                      // Offset the copy to the future position on X
                      // Axis
                      rect.offsetPosition(off_x, 0);

                      // Check if the future bounding will collide with
                      // anything on X axis (cause we offset on with
                      // off_x)
                      if (curMap.isRectangleColliding(rect, this,
                         off_x > 0 ?
                                     Rectangle::Surface::Right :
                                     Rectangle::Surface::Left)) {
                            // If it does collide, the reset the off_x,
                            // meaning that we will not move on X axis
                            off_x = 0;

                            // the distance from targetPosition on Y
                            // axis
                            float dy = targetPosition.getY()
                                        - (position.getY() + off_y);
                            // if the distance is very near
                            if (std::abs(dy) < (speed + 0.1f)) {
                                  // Stop moving
                                  moving = false;
                            }
                      }
                }
                {
                      // Get a copy of Bound Rectangle
                      Rectangle rect = bounding;

                      // Offset the copy to the future position on Y
                      // Axis
                      rect.offsetPosition(0, off_y);

                      // Check if the future bounding will collide with
                      // anything on Y axis (cause we offset on wit
                      // off_y)
                      if (curMap.isRectangleColliding(rect, this,
```

```
                                    off_y > 0 ?
                                          Rectangle::Surface::Bottom :
                                          Rectangle::Surface::Top)) {

                                    // If it does collide, the reset the off_y,
                                    // meaning that we will not move on Y axis
                                    off_y = 0;

                                    // the distance from targetPosition on Y
                                    // axis
                                    float dx = targetPosition.getX()
                                                - (position.getX() + off_x);

                                    // if the distance is very near
                                    if (std::abs(dx) < (speed + 0.1f)) {
                                          // We should stop moving
                                          moving = false;
                                    }
                              }
                        }

                        // If we are still moving and any of the offsets are not
                        // 0
                        if (moving && (off_x != 0 || off_y != 0)) {
                              // this means we can actually move

                              // add the offsets to position, bounding and
                              // TargetArea
                              offsetPosition(off_x, off_y);

                              // If we faced an obstacle, one of the offsets
                              // must be 0 depending on which side we faced it.
                              // In such case the initial direction will miss
                              // the targetPosition because we didn't move
                              // completely as intended. So we have to
                              // recalculate the direction so the next frame
                              // will move to correct position (See paper for
                              // further explanation)
                              if (off_x == 0 || off_y == 0) {
                                    calculateDirection();
                                    // set the according moving animations
                                    setMovingState();
                              }
                        } else {
                              // else we can not move
                              // set the according idle Animations
                              setIdleState();
                        }

                  }
            }
      }
```

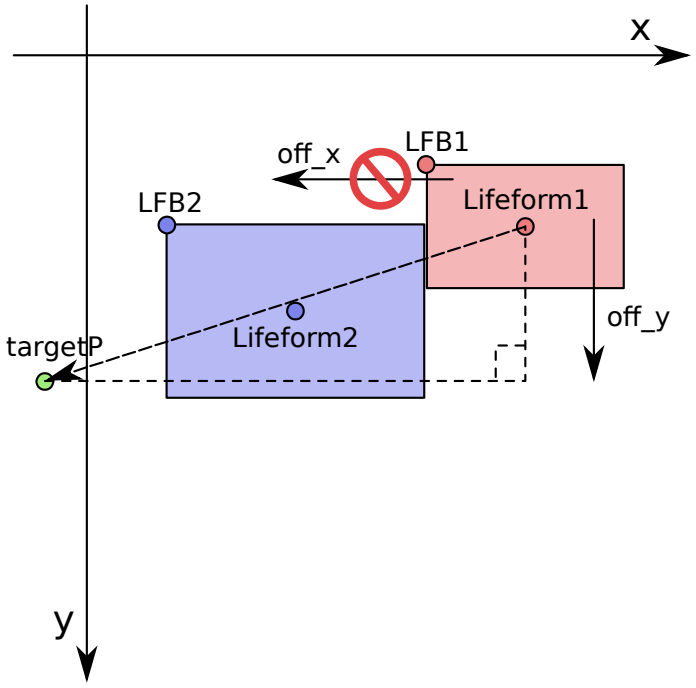
Αρχικά η συνάρτηση ελέγχει αν όντως βρισκόμαστε σε κατάσταση **Moving**. Εφόσον ισχύει αυτό συνεχίζουμε. Η κίνηση του Lifeform χαρακτηρίζεται από το Attribute “Speed”. Αυτή η τιμή ορίζει την ταχύτητα που θα κινηθεί το Lifeform και

βάση αυτού θα υπολογίσουμε τις τελικές συντεταγμένες για αυτό το Frame. Αν όμως το παιχνίδι τρέχει σε διαφορετικά FPS, τότε η κίνηση αυτή θα είναι ανώμαλη όσο αφορά την ταχύτητα που κινείται το Lifeform. Δηλαδή αν τρέχουμε σε 10FPS η τοποθεσία του Lifeform θα γίνεται Offset βάση της ταχύτητας 10 φορές το δευτερόλεπτο. Αν τα FPS είναι 60 τότε αυτό θα γίνεται 6 φορές πιο γρήγορα που σημαίνει ότι ο χαρακτήρας θα κινείται πιο γρήγορα όταν το παιχνίδι τρέχει σε 60FPS από ότι όταν τρέχει με λιγότερα. Αυτό είναι απαράδεκτο, για αυτό και λαμβάνουμε υπόψη αυτό τον παράγοντα πολλαπλασιάζοντας την ταχύτητα με το **elapsedTime** που είναι ο χρόνος που πέρασε από το προηγούμενο Frame.

Αφού υπολογίσουμε την ταχύτητα για αυτό το Frame, πρέπει να δούμε αν βρισκόμαστε πολύ κοντά στις συντεταγμένες που πρέπει να πάμε. Αν ισχύει αυτό τότε σημαίνει ότι φτάσαμε και χρειάζεται να σταματήσουμε να κινούμαστε μέσω της **setIdleState()**. Αυτή απλά θέτει το State σε **Idle** και αναθέτει το σχετικό Animation.

Εφόσον πρέπει να κινηθούμε, υπολογίζουμε τι πρέπει να προσθέσουμε στο **position** προκειμένου να κινηθούμε σωστά προς στο **direction**. Αυτό γίνεται πολλαπλασιάζοντας τις τιμές του **direction** επί την υπολογισμένη ταχύτητα. Πριν προβούμε όμως στην μετατόπιση πρέπει να κάνουμε ελέγχους αν μπορούμε να την κάνουμε. Θα κάνουμε ελέγχους στους 2 άξονες εάν έχουμε συγκρούσεις και εφόσον δεν έχουμε συνεχίζουμε.

Αυτό που θα κάνουμε είναι για κάθε άξονα θα δημιουργήσουμε ένα αντίγραφο του ορθογωνίου σύγκρουσης. Αυτό θα το μετατοπίσουμε κατά το offset που υπολογίσαμε για αυτόν το άξονα. Στην συνέχεια θα ελέγξουμε μέσω του χάρτη αν το ορθογώνιο αυτό συγκρούεται με κάτι. Για τον έλεγχο αυτό παίρνουμε συγκεκριμένες πλευρές καθώς ελέγχουμε για μόνο έναν άξονα. Δηλαδή στην πρώτη περίπτωση που έχουμε εδώ (άξονας x) θα πρέπει να διαλέξουμε μεταξύ της αριστερής και της δεξιάς πλευράς. Αυτό εξαρτάται από την τιμή του offset που μετακινήσαμε. Αν έχει θετική τιμή τότε κινηθήκαμε δεξιά και τότε πρέπει να ελέγξουμε αν η δεξιά πλευρά του ορθογωνίου θα συγκρουστεί κοκ. Αν τελικά συγκρούεται με αυτή την κίνηση, τότε σημαίνει ότι δεν θα την κάνουμε για αυτό και θέτουμε το offsetX σε 0. Αυτό δεν σημαίνει ότι δεν θα κινηθούμε καθόλου αλλά δεν θα κινηθούμε στον άξονα των x. Στον άξονα των y θα πρέπει να κάνουμε περαιτέρω ελέγχους πέρα από το αν υπάρχει και εκεί σύγκρουση.



Εικόνα 5.2.1: Παράδειγμα ελέγχου κρούσεων

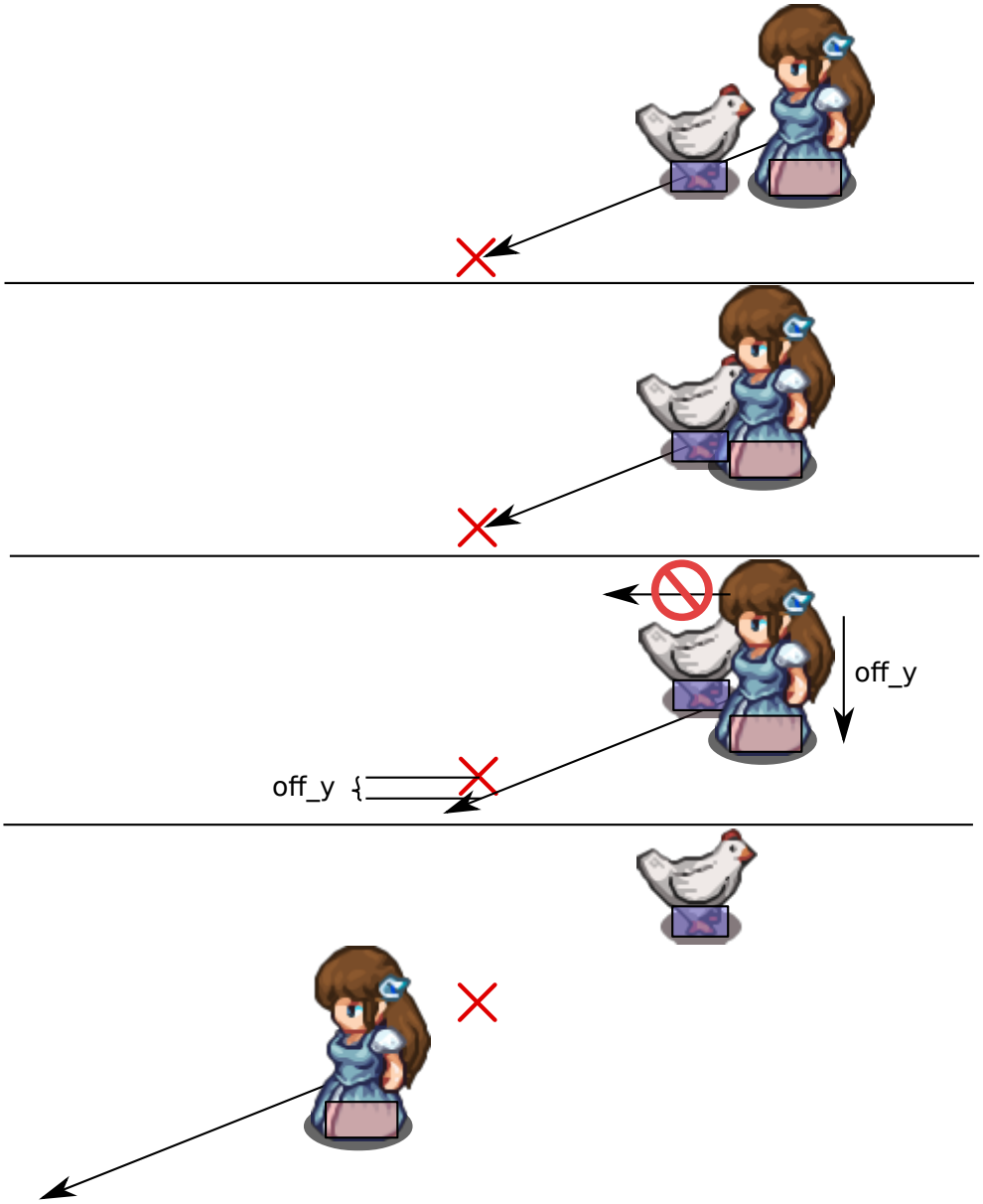
Οι παραπάνω έλεγχοι και ο τρόπος που κάνουμε την κίνηση είναι για υλοποιήσουμε έναν τρόπο ψευδο-pathFinding με απλό τρόπο. Δεν απαιτούμε κανονικό PathFinding απλά να πλησιάσει όσο γίνεται τον στόχο σε περίπτωση σύγκρουσης.

Όπως βλέπουμε στο προηγούμενο σχεδιάγραμμα, το Lifeform1 κινώντας προς το **targetP**, συγκρούεται με το Lifeform2. Σε αυτή την φάση έχουμε ελέγξει στον άξονα των x για συγκρούσεις και βρήκαμε ότι έχουμε σύγκρουση. Έτσι δεν πρόκειται να κινηθούμε σε αυτόν και το off_x το κάνουμε μηδέν. Ξέροντας ότι δεν θα φτάσουμε ποτέ στον στόχο όσο αφορά τον άξονα του x, κοιτάμε μήπως έχουμε φτάσει στον στόχο στον άξονα των y. Εάν φτάσαμε τότε σταματάμε την κίνηση καθώς φτάσαμε όσο πιο κοντά γίνεται.

Σε επόμενη φάση κάνουμε ότι κάναμε για τον άξονα του x για τον άξονα του y. Αυτό γιατί μπορεί να μας αποτράπηκε η κίνηση στον άξονα του x αλλά στον y μέχρι στιγμής θα κινηθούμε κατά off_y (εκτός και αν έχουμε φτάσει πριν και θέσαμε **moving=false**). Επειδή μπορεί και εκεί να συγκρουόμαστε πρέπει να κάνουμε ακριβώς τα ίδια.

Μετά ελέγχουμε αν πρέπει να κάνουμε κάποια κίνηση βάση του flag **moving**. Πέρα από αυτό ελέγχουμε αν τα Offsets είναι μηδέν. Αν είναι μηδέν τότε σημαίνει ότι δεν πρέπει να κάνουμε κίνηση καθώς έχουμε σύγκρουση και στους 2 άξονες και πρέπει να σταματήσουμε. Εφόσον πρέπει να κινηθούμε, εφαρμόζουμε τα offset που έχουμε.

Έχετε στον νου ότι αν ένα από τα 2 offsets είναι μηδέν δεν μας πειράζει καθώς είναι και αυτό κίνηση. Στο τέλος ελέγχουμε αν κάποιο offset ήταν μηδέν. Αν όντως κάποιο είναι μηδέν τότε χρειάζεται να ξανά υπολογίσουμε την κατεύθυνση που έχουμε. Αυτό γιατί το offset που είχαμε 0 μας απέτρεψε από να κινηθούμε σωστά προς την κατεύθυνση που είχαμε. Αυτό σημαίνει ότι η κατεύθυνση παραμένει η ίδια και έτσι η πορεία που θα ακολουθήσει το Lifeform στα επόμενα Frame θα έχει απόκλιση(Offset). Η απόκλιση αυτή μπορεί να γίνει αθροιστική σε κάθε Frame ανάλογα πόσα Frames χρειάζεται να κινηθεί σε έναν μόνο άξονα μέχρι να συνεχίσει κανονικά την πορεία. Η κίνηση αυτή θα κάνει το Lifeform να περάσει δίπλα από τον στόχο κατά αυτό το αθροιστικό Offset. Όπως ξέρουμε, το Lifeform σταματάει την κίνηση του όταν συγκρουστεί και στους 2 άξονες ή φτάσει στον στόχο σε έναν άξονα ενώ στον άλλον συγκρούεται ή φτάσει στον στόχο του πολύ κοντά (δηλαδή τελικά φτάσει στον στόχο του). Όπως είδαμε, σε κάθε Frame που γίνεται η κίνηση ελέγχουμε αν ήμαστε πολύ κοντά στον στόχο προκειμένου να σταματήσουμε. Στην περίπτωση αυτή, όταν περάσουμε δίπλα από τον στόχο, ανάλογα το πόσο μεγάλο ήταν το Offset, ίσως να μην έρθει αρκετά κοντά προκειμένου να σταματήσει, με αποτέλεσμα να συνεχίσει την πορεία του μέχρι να μπορέσει να έρθει κοντά, πράγμα που δεν θα γίνει ποτέ καθώς η κατεύθυνση της κίνησης αλλάζει μόνο όταν αποδοθεί νέος στόχος. Έτσι το Lifeform κινείται για πάντα προς την κατεύθυνση αυτή.



Εικόνα 5.2.2: Παράδειγμα Offset κατεύθυνσης

Προκειμένου να το λύσουμε αυτό, ξανά υπολογίζουμε την κατεύθυνση μας προς τον στόχο σε περίπτωση που κάποιο Offset είναι μηδέν. Επειδή θα έχουμε νέα κατεύθυνση θα πρέπει να υπολογίζουμε πάλι το τι Animation για κίνηση χρειαζόμαστε (σε σπάνιες περιπτώσεις, η κατεύθυνση μπορεί να έχει αλλάξει δραματικά). Για αυτό και καλούμε εκ νέου την **setMovingState()** προκειμένου να εξασφαλίσουμε σωστό Animation.

Αυτά σε περίπτωση που κινηθούμε. Σε περίπτωση τα Offset η το flag μας το αποτρέψουν, τότε σταματάμε την κίνηση μέσω της **setIdleState()** που θέτει το state σε **Idle** και τα απαραίτητα Animation για ακινησία.

c) Attributes και Modifiers

Κάποια από τα σημαντικά στοιχεία των Lifeform είναι τα **Attributes** ή αλλιώς στατιστικά. Αυτά είναι τιμές-ιδιότητες του Lifeform που ορίζουν την κατάσταση του Lifeform όσο αφορά το παιχνίδι σε RPG επίπεδο. Τα περισσότερα στατιστικά αφήνονται για τον προγραμματιστή και μπορεί να είναι άπειρα σε αριθμό και δημιουργούνται δυναμικά. Κάποια είναι στάνταρ και χρησιμοποιούνται και από την μηχανή. Αυτά είναι τα παρακάτω:

- **HP**
- **MaxHP**
- **Mana**
- **MaxMana**
- **Speed**

Τα Attributes έχουν μια βασική τιμή την **BaseValue** η οποία δύσκολα αλλάζει. Η κανονική τιμή που χρησιμοποιείται είναι η υπολογισμένη τιμή **CalculatedValue**. Γιατί όμως έτσι; Στα RPG πολλές φορές κάποιο Lifeform δέχεται κάποια επιρροή που πρέπει να το ανεβάσει κάποιο Attribute. Όμως αυτή η επιρροή δεν είναι μόνιμη και πρέπει όταν περάσει κάποιος χρόνος να φύγει και το Attribute να επανέλθει στην προηγούμενη κατάσταση. Όμως η προηγούμενη κατάσταση του μπορεί να ήταν επηρεασμένη από άλλες επιρροές που δεν υπάρχουν πια. Τι γίνεται τότε; Πως θα βρούμε την σωστή τιμή που θα πρέπει να βάλουμε στο Attribute; Για αυτό και έχουμε 2 τιμές, την Base και την Calculated. Πέρα από αυτά, προκειμένου να αλλάξει η τιμή του Attribute θα γίνεται αυστηρά μέσω αντικειμένων τύπου **Modifier**. Αυτά τα αντικείμενα στοιβάζονται σε μια λίστα και κάθε φορά που χρειάζεται, υπολογίζουν ένα-ένα την νέα τιμή βάση της προηγούμενης υπολογισμένης.

Πίνακας 5.2.9: Η κλάση Attribute

```
class Attribute {
public:

	void calculate();
	void addModifier(Modifier *mod);
	void removeModifier(Modifier *mod);
	void removeModifierAndDelete(Modifier *mod);

	AttributeValue getValue() const {
		return calculatedValue;
	}
```

```
	AttributeValue getBaseValue() const {
		return baseValue;
	}

	void applyToBaseFlat(AttributeValue x);
	void applyToBaseMul(AttributeValue x);
	void setBase(AttributeValue value);

	void setCalculatedValue(AttributeValue value)
	void removeAllModifiers(bool deleteThem);

	Attribute();
	Attribute(AttributeValue baseValue);
	Attribute(const Attribute& other);
	virtual ~Attribute();
protected:

	struct ModifierComparator {
		bool operator()(const Modifier* mod1, const Modifier* mod2);
	};

	std::set<Modifier*, ModifierComparator> modifiers;

	AttributeValue baseValue;
	AttributeValue calculatedValue;

	void reapplyOrders();
};
```

Στην κλάση έχουμε το set των **Modifier** με προσαρμοσμένο συγκριτή κλειδιού. Αυτός ο συγκριτής επιστρέφει true ανάλογα της σύγκρισης των μελών **order** των **Modifier** που συγκρίνονται. Αυτό το κάνουμε γιατί κανονικά το std::set<> χρησιμοποιεί σύγκριση τον Pointer για να ταξινομήσει τα στοιχεία μέσα του. Κάτι τέτοιο μπορεί να φέρει ανεπιθύμητα αποτελέσματα για αυτό και επιβάλουμε δικιά μας μέθοδο σύγκρισης. Πριν συνεχίσουμε ας πάμε να δούμε λίγο την κλάση **Modifier**.

Πίνακας 5.2.10: Η κλάση Modifier

```
class Modifier {
public:
	enum class Type {
		Flat, Scalar
	};

	int getOrder() const;

	virtual void calculate(AttributeValue &value) const=0;

	Modifier(AttributeValue modValue);
	virtual ~Modifier();
protected:
	AttributeValue modValue;
```

```
private:
	int order;

	void setOrder(int order) {
		this->order = order;
	}
	friend class Attribute;
};
```

Τα μόνα μέλη που υπάρχουν εδώ είναι η τιμή που θα παραποιήσει το Attribute **modValue** καθώς και η τιμή που αναφέραμε προηγουμένως που θα παίξει ρόλο στην σειρά η **order**. Η υλοποιήσεις της κλάσης θα πρέπει να υλοποιήσουν την **calculate(...)** όπου θα δέχονται μια αναφορά στην υπολογισμένη τιμή του Attribute και θα την παραποιούν όπως πρέπει. Η μηχανή έχει 2 υλοποιήσεις, την **FlatModifier** και την **MulModifier**. Η πρώτη απλά προσθέτει την **modValue** στην τιμή **value** ενώ η δεύτερη την πολλαπλασιάζει.

Γυρνώντας πίσω στην Attribute, προκειμένου να προστεθεί ένα **Modifier** πρέπει να έχει κατασκευαστεί βάση κάποιας υλοποίησης και να προστεθεί μέσω της **addModifier(...)**.

Πίνακας 5.2.11: Η συνάρτηση Attribute::addModifier()

```
void Attribute::addModifier(Modifier* mod) {
	// Check if the modifier is real
	if (mod != nullptr) {
		// we have to check if there is any other modifier in the list
		if (modifiers.size() > 0) {
			// if there is, then we need to set order to this one
			// according to the last added on the list
			mod->setOrder((*modifiers.rbegin())->getOrder() + 1);
		} else {
			// If there is none then we are first on the line
			mod->setOrder(1);
		}
		// insert the modifier
		modifiers.insert(mod);
		// check if the first modifier has reached large order number
		// (usually happens when massive amount of Modifiers are
		// applied and removed fast)
		if ((*modifiers.begin())->getOrder() > 100) {
			// sanitize the order numbers to prevent overflows
			reapplyOrders();
		}
	}
	// recalculate
	calculate();
}
```

Αρχικά ελέγχουμε αν το Modifier υπάρχει. Εφόσον υπάρχει τότε ελέγχουμε αν έχουμε και άλλα μέσα στην λίστα. Αυτό γιατί θα πρέπει να πάρουμε το **order** από το τελευταίο που προσθέσαμε και να βάλουμε στο νέο +1. Εάν δεν έχουμε κάποιο ήδη στην λίστα, δεν μπορούμε να το κάνουμε αυτό και έτσι του θέτουμε 1 καθώς είναι το πρώτο. Στην συνέχεια το τοποθετούμε στην λίστα. Μετά πρέπει να κάνουμε έναν έλεγχο για το πρώτο στοιχείο της λίστας. Υπάρχει μια πιθανότητα όταν προστίθενται πολλά Modifiers στην λίστα, κάποιο Modifier που ήταν πάνω πάνω να καταλήξει να είναι πρώτο και να έχει μεγάλη τιμή λόγω ότι προστέθηκαν πολλά μαζί. Έτσι από εκεί και πέρα τα επόμενα Modifiers θα έχουν Order μεγαλύτερο από του πρώτου. Αν είναι μεγάλη τιμή, τότε υπάρχει μια πιθανότητα να γίνει order overflow και να μην γίνει σωστή ταξινόμηση. Προκειμένου να το αποφύγουμε αυτό, κάνουμε αυτό τον έλεγχο και εφόσον έχουμε πάρει κάποια μεγάλη τιμή Order στο πρώτο, κάνουμε εκ νέου ανάθεση τιμών στα Modifiers μέσω της **reapplyOrders()**.

Πίνακας 5.2.12: Η συνάρτηση Attribute::reapplyOrders()

```
void Attribute::reapplyOrders() {
     int order = 1;
     for (auto mod : modifiers) {
           mod->setOrder(order++);
     }
}
```

Στην ουσία απλά ξεκινάμε πάλι την αρίθμηση. Παρόλο που οι αριθμοί είναι περίεργοι, η σειρά μέσα στην λίστα είναι σωστή και έτσι απλά θέτουμε τους νέους αριθμούς.

Πρέπει να τονιστεί ότι η πιθανότητα να χρειαστεί αυτή η λύση είναι απειροελάχιστη αλλά προκειμένου να εξαλείψουμε την πιθανότητα του προβλήματος, την ορίζουμε.

Αυτό που δεν είδαμε είναι το πως γίνεται ο υπολογισμός. Όπως είπαμε, η παραποίηση ενός Attribute γίνεται μέσω των Modifiers. Αυτά πρέπει να επηρεάζουν βάση την σειρά που προστίθενται. Αυτό γιατί η σειρά του υπολογισμού επηρεάζει το τελικό αποτέλεσμα, πχ

- base=10
- Προσθήκη modifier για +2 -> 10+2 = 12
- Προσθήκη modifier για x2 -> 12*2= 24

Αν στο παραπάνω αλλάξουμε την σειρά των 2 υπολογισμών θα πάρουμε 22 αντί για 24, κάτι που είναι λάθος καθώς ο υπολογισμός έπρεπε να γίνει βάση της σειράς

προσθήκης. Αυτό εξασφαλίζει το σύστημα order που εξηγήσαμε προηγουμένως. Για αυτό και στον υπολογισμό έχουμε τα στοιχεία με την σωστή σειρά στο set **modifiers**.

Πίνακας 5.2.13: Η συνάρτηση Attribute::calculate()

```
void Attribute::calculate() {
     // reset the calculated value
     calculatedValue = baseValue;
     // for all modifiers
     for (auto mod : modifiers) {
           // calculate the value
           mod->calculate(calculatedValue);
     }
}
```

Μέχρι στιγμής είδαμε τα Attributes σαν μεμονωμένα αντικείμενα μαζί με τα Modifiers τους. Όμως από μόνα τους δεν έχουν νόημα. Αυτά καθώς και τα Modifiers περιέχονται σε maps εντός του Lifeform. Το καθένα από αυτά έχει ένα μοναδικό όνομα που μέσου αυτού μπορούν να τροποποιηθούν και να αφαιρεθούν (τα Modifiers μόνο). Ο ορισμός των Attributes σε ένα Lifeform γίνεται όταν του αποδίδεται μια **LifeformClass** όπως θα δούμε παρακάτω. Για την προσθήκη ενός Modifier κάνουμε την εξής δουλειά.

Πίνακας 5.2.14: Η συνάρτηση Lifeform::addAttributeModifier(...)

```
void Lifeform::addAttributeModifier(const std::string& attributeName,
           const std::string& modifierName, Float value,
                                              Modifier::Type type) {

     // Try finding this modifierName in the modifiers container
     auto itr = modifiers.find(modifierName);

     // if this modifier is not present
     if (itr == modifiers.end()) {

           // Get the Attribute reference with the Arg's name
           Attribute &att = attributes[attributeName];

           // Here depending on Modifier type, we add the according
           // Modifier
           Modifier *mod;
           switch (type) {
           case Modifier::Type::Flat:
                 mod = new FlatModifier(value);
                 break;
           case Modifier::Type::Scalar:
                 mod = new MulModifier(value);
                 break;
           default:
```

```
                mod = new FlatModifier(value);
                Logger::getInstance().write(
                        "Illegal Enumeration at Lifeform:" + name
                                + " on Function Call:
                                        addAttributeModifier()."
                        "Reason: Possibly from a Lua call with a
number/Variable argument rather than one of the pre-Defined ones(eg.
Zeta.ModifierType.Flat). Defaulting to Flat...",
                        Logger::MessageType::Warning);
                break;
            }

            att.addModifier(mod);

            modifiers[modifierName] = mod;

            // set dirty
            dirty = true;
        }
    }
```

Στην παραπάνω μορφή της συνάρτησης, το Modifier κατασκευάζεται εντός και προστίθεται. Αρχικά ελέγχουμε το Map **modifiers** εάν έχει ξανά προστεθεί Modifier με το ίδιο όνομα. Αν όχι, τότε βάση του τύπου που μας δίνεται το κατασκευάζουμε και το βάζουμε στο Attribute που μας δίνεται από το όνομα στο όρισμα. Στην συνέχεια προσθέτουμε το Modifier στο **modifiers** προκειμένου να ξέρουμε ότι το έχουμε πάρει αυτό. Ύστερα θέτουμε το **dirty** flag σε true. Αυτό το Flag το θέτουμε πάντα όταν έχει αλλάξει ένα Attribute για να βοηθήσει το GUI να καταλάβει ότι πρέπει να αλλάξει τιμές για αυτό το Lifeform εάν δείχνει κάτι.

5.3 Η κλάση LifeformClass

Μέχρι στιγμής έχουμε δει πολύ βασικά πράγματα γύρω από τα Lifeforms. Όμως αυτό που δίνει πραγματική υπόσταση σε αυτό είναι το **LifeformClass** που έχει. Σε πολύ προηγούμενα κεφάλαια είδαμε τι περιέχει μια LifeformClass για ένα Lifeform. Τώρα θα δούμε πως αποδίδεται σε ένα Lifeform.

Πίνακας 5.3.1: Η κλάση LifeformClass

```
class LifeformClass: public Resource {
public:

    const std::string& getLifeformName() const;
    void getStats(ZMap<std::string, Attribute>& stats) const;
    void getAbilities(Lifeform *owner) const;
    void getChildObjets(Lifeform *owner) const;
    const std::string& getAnimationClassName() const;
```

```
	int getLevel() const;
	const LuaTable& getTable() const;
	const LuaFunctor& getOnClick() const;
	const LuaFunctor& getOnCollision() const;
	const Faction& getFaction() const;

	void setTable(lua_Object table);
	void setOnClick(lua_Object func);
	void setOnCollision(lua_Object func);

	Behaviour* getGeneralBehaviour(Lifeform& owner) const;
	Behaviour* getMovementBehaviour(Lifeform& owner) const;

	void levelizeStats(float levelMultiplier = 1.0f);
	void levelizeStats(float multiplier, int lowerBound, int
upperBound);
	void addAbility(const AbilityClass& abClass, int level);

	LifeformClass();
	LifeformClass(const std::string& path);
	virtual ~LifeformClass();
private:

	typedef struct {
		std::string abClass;
		int level;
	} LifeformAbility;

	typedef struct {
		const AbilityClass* abClass;
		int level;
	} LuaAbility;

	mutable ZSmallMap<std::string, LifeformAbility> abilities;
	mutable std::vector<LuaAbility> luaAbilities;
	mutable ZSmallMap<std::string, std::uniform_int_distribution<int>>
stats;
	std::vector<ChildParams> childParameters;
	std::string lifeform_name;
	std::string animClass;

	LuaTable table;
	LuaFunctor onCollisionF;
	LuaFunctor onClickF;
	const Faction* faction;
	BehaviourClass* genericAI;
	BehaviourClass* movementAI;
	int level;

};
```

Εδώ τα περισσότερα μέλη έχουν τα δεδομένα που θα έχουν όλα τα Lifeform που δεχτούν αυτή την LifeformClass. Τα περισσότερα αντιγράφονται μέσα στο Lifeform που την δέχεται και κάποια ανακτώνται από εδώ. Η φόρτωση είναι παρόμοια με τα

υπόλοιπα αρχεία XML για αυτό και θα την παραλείψουμε επικεντρώνοντας στα μέλη και πως λειτουργούν.

Το μέλος **abilities** περιέχει αντικείμενα **LifeformAbility** που περιέχουν ένα string με την διαδρομή του Ability και το επίπεδο του Ability. Όταν στο Lifeform καλεστεί η **getAbilities(...)** θα καλεστεί για καθένα από αυτά αντικείμενα το **addAbility(...)** του Lifeform.

Πίνακας 5.3.2: Οι συναρτήσεις Lifeform::addAbilitiy(x,x)

```
void Lifeform::addAbility(const std::string& abilityClass, int level) {
	const AbilityClass& cls =
		System::getInstance().getResourcesManager().getAbilityClass(
					abilityClass);
	addAbility(cls, level);
}

void Lifeform::addAbility(const AbilityClass& abilityClass, int level) {
	// Check if the ability exists
	{
	auto itr = activeAbilities.find(abilityClass.getAbilityName());
		if (itr != activeAbilities.end()) {
			// If exists, just set the new level
			itr->second->setLevel(level);
			return;
		}
	}
	{
	auto itr = passiveAbilities.find(abilityClass.getAbilityName());
		if (itr != passiveAbilities.end()) {
			// If exists, just set the new level
			itr->second->setLevel(level);
			return;
		}
	}
	{
		auto itr = regenerations.find(abilityClass.getAbilityName());
		if (itr != regenerations.end()) {
			// If exists, just set the new level
			itr->second->setLevel(level);
			return;
		}
	}
	// Else if it does not exist
	// Generate a new ability
	Ability* ab = abilityClass.getNewAbility(this, level);

	if (ab != nullptr) {
		switch (abilityClass.getType()) {
		case AbilityClass::Type::Active:
			activeAbilities[abilityClass.getAbilityName()] =
					static_cast<ActiveAbility*>(ab);
			break;
		case AbilityClass::Type::Passive:
			passiveAbilities[abilityClass.getAbilityName()] =
```

```
                        static_cast<PassiveAbility*>(ab);
                break;
            case AbilityClass::Type::Regeneration:
                regenerations[abilityClass.getAbilityName()] =
                        static_cast<Regeneration*>(ab);
                break;
            }
        }
    }
```

Στην πρώτη εκδοχή της συνάρτησης ζητάμε μέσω του string που μας δόθηκε το **AbilityClass** και μετά καλούμε την δεύτερη μορφή της συνάρτησης. Εκεί ελέγχουμε στους 3 διαφορετικούς πίνακες εάν υπάρχει αυτό το Ability ήδη και αν υπάρχει απλά θέτουμε το νέο επίπεδο και δεν κάνουμε τίποτα περαιτέρω. Αν δεν υπάρχει πουθενά όμως, τότε μπορούμε να την προσθέσουμε. Δημιουργούμε ένα νέο αντικείμενο Ability μέσω της AbilityClass που πήραμε. Από εκεί ανάλογα τον τύπο της, το προσθέτουμε τον αντίστοιχο πίνακα του Lifeform.

Ο πίνακας **luaAbilities** γεμίζει χειροκίνητα αφού έχει κατασκευαστεί το LifeformClass, μέσω της **addAbility(...)** της LifeformClass. Αυτή συνήθως καλείται μέσω της Lua.

Το επόμενο μέλος είναι ο πίνακας **stats**. Αυτός γεμίζει με ομοιογενής μηχανές αριθμών (uniform Distribution). Τα όρια αυτών των μηχανών ορίζονται μέσω του αρχείου XML όπως είδαμε στα αρχικά κεφάλαια. Αυτές οι τιμές γίνονται Levelize μέσω των **levelizeStats(...)** όπου τελειοποιούνται τα άκρα των μηχανών. Όταν το Lifeform χρειαστεί να πάρει τα Attributes του, καλεί την **getStats(...)** με όρισμα τον πίνακα των Attributes του.

Πίνακας 5.3.3: Η συνάρτηση LifeformClass::getStats()

```
void LifeformClass::getStats(ZMap<std::string, Attribute>& stats) const {
    std::default_random_engine generator(
        std::chrono::system_clock::now().time_since_epoch().count());
    for (auto stat : this->stats) {
        stats[stat.first] = stat.second(generator);
    }
    stats["MaxHP"] = stats["HP"];
    stats["MaxMana"] = stats["Mana"];
}
```

Εδώ χρειάζεται κάθε φορά που καλούμε την συνάρτηση να κάνουμε seed τον generator που θα μας φέρει τον τυχαίο αριθμό μεταξύ των ορίων των μηχανών. Για κάθε στοιχείο του πίνακα **stats** του LifeformClass δημιουργούμε στον πίνακα που

μας παρέχεται ένα Attibute και του θέτουμε το base (μέσω του Operator =) την τιμή που θα μας φέρει η μηχανή τυχαίων αριθμών για αυτό το Attribute. Στο τέλος θέτουμε τα βασικά Attributes HP και Mana να είναι “γεμάτα”.

Το μέλος **childParameters** περιέχει αντικείμενα με όλα τα απαραίτητα στοιχεία για την δημιουργία των παιδιών. Αυτά τα στοιχεία είναι:

- Όνομα (name)
- Διαδρομή AnimationClass (animationClass)
- Όνομα Animation (animationName)
- Ο τύπος (type)
- Τα offset (dx,dy)

Βάση αυτών, κατασκευάζονται τα αντικείμενα παιδιά όπως είδαμε κατά την ανάλυση της κλάσης **Object**.

Το μέλος **animClass** περιέχει την διαδρομή του αρχείου του **AnimationClass** που θα χρησιμοποιήσει το Lifeform.

Το **table** περιέχει τον πίνακα Lua που θα περιέχει επιπλέον δεδομένα για τον προγραμματιστή. Θέτεται χειροκίνητα μεσώ της Lua. Με τον ίδιο τρόπο θέτονται και οι Callbacks **onCollisionF** και **onClickF** που θα καλούνται όταν το Lifeform συγκρουστεί με κάτι ή ο παίχτης το κλικάρει αντίστοιχα.

Τα μέλη τύπου **BehaviourClass** περιέχουν τις υλοποιήσεις του AI που πρέπει να έχει το Lifeform. Περισσότερα για αυτό παρακάτω.

Στο τέλος έχουμε το επίπεδο των Lifeform που θα θέτονται από την LifeformClass.

Όπως ξέρουμε, οι LifeformClass σε ένα Lifeform είναι κάτι σαν εξάρτημα. Δηλαδή όταν θέτουμε μια LifeformClass σε ένα Lifeform δεν δένεται απαραίτητα με αυτό. Μπορούμε εύκολα να αλλάξουμε την LifeformClass ανά πάσα στιγμή μέσω της **setClass(...)**.

Πίνακας 5.3.4: Η συνάρτηση Lifeform::setClass(const LifeformClass&)

```
void Lifeform::setClass(const LifeformClass& lifeClass) {
     for (auto child : childs) {
           delete child.second;
     }
     childs.clear();
     childsToBeOffseted.clear();
     if (lifeformClass != nullptr) {
           System::getInstance().getResourcesManager().releaseResource(
                       *lifeformClass);
     }
```

```
     setState(LifeformState::Action::Idle |
LifeformState::Direction::Down);
     // Get the stated Lifeform Class from RESCON
     lifeformClass = &lifeClass;
     // Get the level
     level = lifeformClass->getLevel();
     // Remove all attributes
     attributes.clear();
     // Get the new Attributes
     lifeformClass->getStats(attributes);
     // Get the AnimationClass
     animHandler.setAnimationClass(
                 lifeformClass->getAnimationClassName());
     // Get the faction
     faction = &lifeformClass->getFaction();
     // get the Lifeform name
     name = lifeformClass->getLifeformName();
     // Get the AI handlers
     generalAI = lifeformClass->getGeneralBehaviour(*this);
     movementAI = lifeformClass->getMovementBehaviour(*this);
     // Reset all shapes to AnimationClass defaults
     resetShapes();
     // Offset the shapes to the current position
     offsetShapes(position.getX(), position.getY());
     for (auto ab : activeAbilities) {
          delete ab.second;
     }
     activeAbilities.clear();
     for (auto ab : passiveAbilities) {
          delete ab.second;
     }
     passiveAbilities.clear();
     for (auto ab : regenerations) {
          delete ab.second;
     }
     regenerations.clear();
     // get the new Abilities
     lifeformClass->getAbilities(this);
     // get the Childs
     lifeformClass->getChildObjets(this);
     // Get the Lua table
     table = lifeformClass->getTable();
     // Get the onClick Lua Callback
     onClickF = lifeformClass->getOnClick();

     // Get the onCollision Lua Callback
     onCollisionF = lifeformClass->getOnCollision();

     // Finaly start the animations
     animHandler.getMainAnimationPlayer().play();
}
```

Μέσω αυτής της συνάρτησης παίρνει την τελική μορφή το Lifeform όταν κατασκευάζεται. Επειδή όμως η συνάρτηση μπορεί να καλείται και για να αντικαταστήσει το LifeformClass αφού έχει κατασκευαστεί και ζήσει, χρειάζεται να καθαρίσουμε το Lifeform πριν πάρουμε τα νέα δεδομένα. Αρχικά καταστρέφουμε όλα τα αντικείμενα

παιδιά καθώς και τα βγάζουμε από τις λίστες offset. Μετά ελέγχουμε αν έχουμε ήδη κάποιο LifeformClass φορτωμένο εδώ. Εάν έχουμε τότε το αποδεσμεύουμε καθώς θα πάρουμε ένα άλλο. Στην συνέχεια επαναφέρουμε την κατάσταση του Lifeform σε ακινησία προς τα κάτω. Αποδίδουμε την νέα LifeformClass στο αντίστοιχο μέλος και παίρνουμε από αυτή το νέο επίπεδο που θα έχουμε. Στην συνέχεια παίρνουμε τα νέα Attributes αφού καθαρίσουμε τα παλιά. Επόμενο είναι να βάλουμε στο AnimationHandler την νέα AnimationClass. Αυτός δέχεται την διαδρομή της νέας από το LifeformClass και την φορτώνει. Μετά παίρνουμε το Faction και το όνομα του Lifeform. Στην συνέχεια κατασκευάζουμε τα αντικείμενα AI της κίνησης και της συμπεριφοράς από τις υλοποιήσεις που έχουμε μέσα στο LifefromClass. Αυτό που χρειάζεται μετά είναι να επαναφέρουμε τα σχήματα του Lifeform. Όπως και με τους άλλους πίνακες, έτσι και με τους πίνακες των Abilities τους καθαρίζουμε πριν λάβουμε τα νέα. Μετά κατασκευάζουμε τα αντικείμενα παιδιά και τέλος παίρνουμε αναφορά προς τον πίνακα Lua και των callback για σύγκρουση και κλικ.

Μετά την κλήση αυτής της συνάρτησης, το Lifeform είναι έτοιμο για χρήση. Για την ακρίβεια αν προστεθεί μέσα στον χάρτη, τότε ανάλογα το AI που του αποδόθηκε, θα αρχίσει να υπάρχει και κανονικά.

5.4 Abilities και Effects

a) Abilties

Τώρα θα πάμε να δούμε το πως λειτουργούν οι ικανότητες (Abilties). Οι ικανότητες όπως ξέρουμε χωρίζονται σε **ActiveAbilities, PassiveAbilities,** και **Regenerations**. Αυτές οι κλάσεις προέρχονται από την κοινή κλάση-βάση **Ability**. Δεν την κληρονομούν κατευθείαν άλλα ενδιάμεσα υπάρχει το template **ClassifiedAbilty** που προσθέτει την κλάση της ικανότητας.

Τα Abilties κατασκευάζονται από **AbilityClass**. Από αυτή εξειδικεύονται οι **ActiveAbilityClass**, **PassiveAbilityClass**, **RegenerationAbilityClass**, οι οποίες αναλαμβάνουν να κατασκευάσουν κάποιο Ability. Ο όλος μηχανισμός φαίνεται πολύπλοκος αλλά θα εξηγηθεί αναλυτικά παρακάτω.

Πίνακας 5.4.1: Η κλάση AbilityClass

```
class AbilityClass: public Resource, public LuaPushable {
public:

     enum class Type {
           Active, Passive, Regeneration
     };

     int getLevels() const;
     const std::string& getAbilityName() const;
     bool isPassive() const;
     Type getType() const;
     const AbilityClass* getParent() const;
     const LuaTable& getTable() const;
     lua_Object getLuaTable() const;

     void onApply(Ability *invoker, Lifeform *caster, Lifeform *target)
                                                                  const;
     void onRemove(Ability *invoker, Lifeform *caster, Lifeform *target)
                                                                  const;
     void onLevelChange(Ability *invoker) const;

     virtual Ability* getNewAbility(Lifeform *caster, int level) const;

     virtual void pushToLua(lua_State* lstate);

     AbilityClass();
     AbilityClass(const std::string& path);
     AbilityClass(lua_Object table);
     AbilityClass(AbilityClass& parent, LuaTable& table);
     virtual ~AbilityClass();
protected:
     std::string abilityName;
     mutable LuaTable table;
     LuaFunctor onApplyF;
     LuaFunctor onRemoveF;
     LuaFunctor onLevelChangeF;
     AbilityClass* implementation;
     AbilityClass* parent;
     int levels;
     Type type;

     void getData(LuaTable& table);
     void nullise();
};
```

Η φόρτωση ενός AbilityClass όπως είδαμε στα πρώτα κεφάλαια, γίνεται μέσω πίνακα Lua. Ο πίνακας αυτός μπορεί να οριστεί απευθείας σαν όρισμα στον κατασκευαστή ή να καλεστεί ο κατασκευαστής με την διαδρομή του αρχείου Lua που επιστρέφει τον πίνακα που θα παρθούν τα δεδομένα.

Βλέπουμε στα μέλη της κλάσης ότι έχουμε 2 τα οποία είναι ίδια με αυτή, τα μέλη **implementation** και **parent**. Αυτά υπάρχουν λόγω του τρόπου που λειτουργεί η

κλάση. Πριν φορτωθεί ένα αρχείο, δεν ξέρουμε τι είδους εξειδικευμένη AbilityClass να χρησιμοποιήσουμε. Για αυτό και ξεκινάμε την φόρτωση με την βασική και αφού βρούμε τον τύπο, τότε δημιουργούμε μια εξειδικευμένη μέσα στο μέλος **implementation**. Για να το καταλάβουμε καλύτερα, ας δούμε έναν κατασκευαστή.

Πίνακας 5.4.2: Η συνάρτηση AbiltyClass::AbilityClass(lua_Object)

```
AbilityClass::AbilityClass(lua_Object table) :
    Resource("Lua Generated Ability Class"),
    implementation(nullptr),
    parent(nullptr), levels(1) {
    try {
          this->table.setFromStack(
                LuaEngine::getInstance().getState(), table);
          getData(this->table);
          switch (type) {
          case Type::Active:
                implementation =
                      new ActiveAbilityClass(*this, this->table);
                break;
          case Type::Passive:
                implementation =
                      new PassiveAbilityClass(*this, this->table);
                break;
          case Type::Regeneration:
                implementation =
                      new RegenerationAbilityClass(*this, this->table);
                break;
          default:
                implementation =
                      new ActiveAbilityClass(*this, this->table);
                break;
          }
    } catch (Exception& ex) {
          Logger::getInstance().write(
                      "During getting table for ability class: Reason: "
                                  + ex.reason());
          nullise();
    }
}
```

Εδώ αρχικά φορτώνουμε τον πίνακα Lua και παίρνουμε τα βασικά δεδομένα μέσω της **getData()**. Αυτή γεμίζει τα μέλη **onApplyF, onRemoveF, onLevelChangeF, level** και **type**, με τις αντίστοιχες callback και δεδομένα. Μετά ανάλογα την τιμή που έχει το μέλος **type**, πρέπει να δημιουργήσουμε μια εξειδίκευση στο μέλος **implementation**. Για παράδειγμα ας δούμε την περίπτωση της **ActiveAbilityClass**.

Πίνακας 5.4.3: Η κλάση ActiveAbilityClass

```
class ActiveAbilityClass: public AbilityClass {
public:

     Float getCooldown() const;
     Float getManaCost() const;
     Float getCastTime() const;
     Float getRange() const;
     const std::string& getCastAnimation() const;
     const std::string& getReleaseAnimation() const;
     const std::string& getCastSound() const;
     const std::string& getReleaseSound() const;

     const ProjectileClass* getProjectileClass(const std::string& name)
                                                                  const;

     Ability* getNewAbility(Lifeform *caster, int level) const;

     const ZMap<std::string, ProjectileClass>& getProjectileClasses()
                                                                  const;
     const ZMap<std::string, EffectClass>& getEffectClasses() const;

     void applyEffect(const std::string& name, Lifeform *target,
                 int level, ActiveAbility* ability = nullptr) const;

     ActiveAbilityClass(AbilityClass& parent, LuaTable& table);
     ~ActiveAbilityClass();
private:
     ZMap<std::string, EffectClass> effects;
     ZMap<std::string, ProjectileClass> projectiles;
     std::string castAnimation;
     std::string releaseAnimation;
     std::string castSound;
     std::string releaseSound;

     Float range;
     Float manaCost;
     Float castTime;
     Float cooldown;
};
```

Κάθε **ActiveAbilityClass** έχει τα πολύ βασικά στοιχεία που χρειάζεται καθώς και μπορεί να έχει **effects** και **projectiles** που μπορεί να χρησιμοποιήσει. Για αυτά θα δούμε παρακάτω. Σε αυτή την κλάση θα δούμε αρκετές επιπλέον λειτουργίες που κάνει σε σχέση με την **AbilityClass**. Πάμε να δούμε το πώς παίρνει τα δεδομένα της όταν κατασκευαστεί από την **AbilityClass**.

Πίνακας 5.4.4: Η συνάρτηση ActiveAbilityClass::ActiveAbilitycClass()

```
ActiveAbilityClass::ActiveAbilityClass(AbilityClass& parent, LuaTable&
table) :
     AbilityClass(parent, table), range(80.0f), manaCost(0.0f),
     castTime(0.0f), cooldown(1.0f) {

     table.getField(LuaString("CoolDown"), cooldown);
     table.getField(LuaString("ManaCost"), manaCost);
     ...
     {
     LuaTable effects;
     if (table.getField(LuaString("Effects"), effects)) {
           effects.forEach(
                 [this](const LuaString& key, const LuaValue& value,
                                                  LuaValue::Type type) {
                       if (type==LuaValue::Type::Table) {
                             this->effects[key.getValue()] = std::move(
                                   EffectClass(
                                   static_cast<const LuaTable&>(value)));
                       }
                 });
           }
     }
     {
     LuaTable projectiles;
     if (table.getField(LuaString("Projectiles"), projectiles)) {
           projectiles.forEach(
                 [this](const LuaString& key, const LuaValue& value,
                                                  LuaValue::Type type) {
                       if (type==LuaValue::Type::Table) {
                             this->projectiles[key.getValue()].load(
                                   static_cast<const LuaTable&>(value));
                       }
                 });
           }
     }
}
```

Στην αρχή παίρνουμε όλα τα δεδομένα που χρειαζόμαστε μέσω του πίνακα. Μετά πρέπει να πάρουμε τους εμφωλευμένους πίνακες για να κατασκευαστούν τα **EffectClass** και τα **ProjectileClass**. Αυτό γίνεται μέσω της συνάρτησης Lambda που καλούμε για κάθε στοιχείο του εμφωλευμένου πίνακα. Ελέγχουμε αν το στοιχείο είναι και αυτό πίνακας και αν είναι τότε συνεχίζουμε με την κατασκευή και την απόδοση στα αντίστοιχα Hashtables.

Όπως είδαμε στην κλάση **Lifeform** προκειμένου να πάρουμε ένα **Ability** από μια **AbilityClass** χρειάζεται να καλέσουμε από αυτή την **getNewAbility(...)**. Όταν την καλούμε από την βασική **AbilityClass,** αυτή επιστρέφει ότι επιστρέψει η εξειδικευμένη στο μέλος **implementation**. Για παράδειγμα η **ActiveAbilityClass** κάνει Override την συνάρτηση και επιστρέφει ένα σωστό αντικείμενο.

Πίνακας 5.4.5: Η συνάρτηση ActiveAbility::getNewAbility()

```
Ability* ActiveAbilityClass::getNewAbility(Lifeform* caster, int level)
                                                                const {
     return new ActiveAbility(*this, caster, level);
}
```

Όμως τι είναι το αντικείμενο αυτό; Είναι ένα αντικείμενο που μπορεί να χρησιμοποιήσει ένα Lifeform. Η κλάση **ActiveAbility** είναι μια από τις τελευταίες κλάσεις του δένδρου κληρονομικότητας της κλάσης **Ability**.

Πίνακας 5.4.6: Η κλάση Ability

```
class Ability: public LuaPushable {
public:
     enum class Result {
           Success,
           onCooldown,
           NoTarget,
           NoMana,
           OutOfRange,
           TargetIsDead,
           TargetIsNotHostile,
           InvalidAbility,
           Other
     };
     int getLevel() const;
     void setLevel(int level) {
           this->level = level;
           onLevelChange();
     }
     Lifeform& getOwner();
     void setOwner(Lifeform *owner);
     Ability(Lifeform *owner, int level);
     virtual ~Ability();
protected:
     Ability();
     Lifeform *owner;
     int level;

     virtual void onLevelChange()=0;
};
```

Όπως είπαμε, τα αντικείμενα της οικογένειας Ability είναι αυτά που χρησιμοποιούν τα Lifeform και κατασκευάζονται από εξειδικευμένες **AbilityClass**. Για αυτό και χρειάζεται να ξέρουν ποιος είναι ο ιδιοκτήτης τους και να τον έχουν στο μέλος **owner**. Το καθένα από αυτά είναι ανεξάρτητα από τα άλλα, ακόμα και αν έχουν κοινό **AbilityClass**, για αυτό και κάποια δεδομένα πρέπει να υπάρχουν και εδώ όπως το

level. Τέτοια δεδομένα μπορεί να αλλάξουν κατά την διάρκεια του παιχνιδιού για αυτό και χρειάζεται το καθένα να έχει ένα αντίγραφο.

Βλέπουμε επίσης ότι έχουμε μια Pure virtual συνάρτηση, την **onLevelChange()**. Θα αναρωτηθεί κανείς γιατί την έχουμε εδώ, καθώς η συνάρτηση αυτή θα καλείται από την **AbilityClass**. Το θέμα είναι ότι εδώ δεν έχουμε κανένα μέλος που να λέει κάτι για την AbilityClass του Ability. Αυτό γιατί αυτή η κλάση είναι πολύ αφηρημένη και οι εξειδικευμένες που θα την κληρονομήσουν θα χρειάζεται να έχουν το AbilityClass. Έτσι χρειάζεται μια ενδιάμεση κλάση που θα εισάγει το εξειδικευμένο AbilityClass στην δομή.

Πίνακας 5.4.7: Το Template ClassifiedAbility

```
template<typename AbClass>
class ClassifiedAbility: public Ability {
public:
     const AbClass& getClass() const {
          return abilityClass;
     }
     ClassifiedAbility(const AbClass& abilityClass, Lifeform *owner,
                                                          int level);
     virtual ~ClassifiedAbility() {
          ResourceContext* resources =
                         &System::getInstance().getResourcesManager();
          if (resources != nullptr) {
               resources->releaseResource(abilityClass);
          }
     }
protected:
     const AbClass& abilityClass;
};
```

Το template βλέπουμε κληρονομεί την Ability και ορίζει το όρισμα του template σαν το AbilityClass καθώς και παρέχει και συνάρτηση για την ανάκτηση του. Στον καταστροφέα φροντίζει να αποδεσμεύσει το AbilityClass και υπάρχει μόνο ένας κατασκευαστής που δέχεται το AbilityClass, το Lifeform που θα το έχει και το επίπεδο.

Τώρα θα δούμε την πιο πολύπλοκη εκδοχή των Abilities, την **ActiveAbility**.

Πίνακας 5.4.8: Η κλάση ActiveAbility

```
class ActiveAbility: public ClassifiedAbility<ActiveAbilityClass>,
          public Updateable {
public:

     enum class State {
          Ready, onCoodown, Casting, Inactive
     };
```

```
      Result invoke();
      void update(Float elapsedTime);

      Float getRemainingCooldown() const {
            return state == State::onCoodown ? frameCounter : 0;
      }
      bool isCasting() const;
      bool isReady() const;
      bool isOnCooldown() const;
      bool isDone() const;
      bool canBeInvoked() const;
      bool isThereEnoughMana() const;
      Float getCastTime() const;
      void setCastTime(Float castTime);
      Float getCoolDown() const;
      void setCoolDown(Float coolDown);
      Float getManaCost() const;
      void setManaCost(Float manaCost);
      Float getRange() const;
      void setRange(Float range);

      void stopCasting();

      void invokeProjectile(const std::string& name, Float x, Float y);
      void applyEffect(const std::string& name, Lifeform *target,
                                                            int level);

      void accept(Visitor& visitor);

      void pushToLua(lua_State* lstate);

      ActiveAbility(const ActiveAbilityClass& abilityClass,
                                          Lifeform *owner, int level);
      virtual ~ActiveAbility();
protected:

      ZSmallMap<std::string, Projectile*> availableProjectiles;
      ZSmallSet<Projectile*> projectiles;

      SoundInstance *castingSound;
      SoundInstance *releaseSound;

      Lifeform *target;
      Attribute &ownerMana;

      Float manaCost;
      Float coolDown;
      Float castTime;
      Float range;

      Float frameCounter;
      State state;

      bool hasCastingAnimation;
      bool hasReleaseAnimation;
      bool hasCastingSound;
      bool hasReleaseSound;
      bool animationChanged;
      bool isCastingSoundPlaying;
```

```
    bool finished;

    virtual void release();
    virtual Result checkConditions();
    virtual void cast();
    void onLevelChange();
};
```

Όπως ξέρουμε τα αντικείμενα **ActiveAbility** είναι ξεχωριστές οντότητες που υπάρχουν στα Lifeforms και κληρονομούν τα δεδομένα τους από τα **ActiveAbilityClass**. Όμως αυτές οι οντότητες πρέπει να μπορούν να αλλάξουν ανάλογα το τι συμβαίνει στο Lifeform που τις έχει, για αυτό και χρειάζεται να αντιγράψουν αυτά τα δεδομένα εντός. Έτσι έχουμε τα μέλη **manaCost, coolDown, castTime** και **range**. Πέρα από αυτά έχουμε τα διαθέσιμα **Projectiles** στον πίνακα **availableProjectiles** εκ των οποίων αυτά που θα ενεργοποιηθούν θα μπουν και στον **projectiles** όπως θα δούμε.

Στα μέλη **castingSound** και **releaseSound** θα μπουν οι ήχοι που θα παιχτούν στις στιγμές αυτές εφόσον ορίζονται από το **ActiveAbilityClass**. Αντίστοιχα θα οριστούν τα flags **has(x)**, που ανάλογα αν ορίζονται οι πόροι αυτοί παίρνουν τιμή true. Αυτές οι τιμές μπαίνουν κατά την κατασκευή του ActiveAbility. Υπάρχει μια αναφορά στο Mana του ιδιοκτήτη και τον στόχο της ικανότητας.

Οι ActiveAbilities είναι οι μόνες που χρειάζονται να “φορτώσουν” πριν εκτελεστούν τελείως. Αυτό λέγεται και Casting όπως έχουμε προαναφέρει. Για αυτό και η λειτουργία τους χαρακτηρίζεται από 4 διαφορετικά στάδια που απαριθμούνται από το Enumeration **State**. Ένα επιπρόσθετο είναι το flag **finished**. Όταν δίνεται η εντολή να εκτελεστεί η ικανότητα, τότε η κλάση Lifeform καλεί την **invoke()**.

Πίνακας 5.4.9: Η συνάρτηση ActiveAbility::invoke()

```
Ability::Result ActiveAbility::invoke() {
    if (state == State::Ready) {
        if (ownerMana.getValue() >= manaCost) {
            return checkConditions();
        } else {
            return Ability::Result::NoMana;
        }
    } else {
        return Ability::Result::onCooldown;
    }
}
```

Για να μπορεί να εκτελεστεί επιτυχώς η ικανότητα, χρειάζεται νια γίνουν μια σειρά από ελέγχους. Ο πρώτος έλεγχος είναι αν η κατάσταση του Ability είναι σε **Ready**. Αν δεν είναι σε αυτή τότε σημαίνει ότι η ικανότητα έχει ξαναχρησιμοποιηθεί σύντομα και βρίσκεται σε κατάσταση **onCooldown**. Ο δεύτερος έλεγχος που χρειάζεται να κάνουμε είναι ότι υπάρχει αρκετό Mana για να χρησιμοποιηθεί η ικανότητα. Εφόσον υπάρχει, τότε καλούμε την **checkConditions()** που κάνει τους περαιτέρω ελέγχους.

Πίνακας 5.4.10: Η συνάρτηση ActiveAbility::checkConditions()

```
Ability::Result ActiveAbility::checkConditions() {
	target = owner->getTarget();
	if (target != nullptr) {
		if (owner->isHostileWith(target)) {
			if (target->isAlive()) {
				if (owner->getDistance(*target) <= range) {
					cast();
					return Result::Success;
				} else {
					return Result::OutOfRange;
				}
			} else {
				return Result::TargetIsDead;
			}
		} else {
			return Result::TargetIsNotHostile;
		}
	} else {
		return Result::NoTarget;
	}
}
```

Αρχικά ελέγχουμε αν ο ιδιοκτήτης έχει έναν στόχο. Πάντα τα ActiveAbilities χρειάζονται έναν στόχο για να εκτελεστούν πάνω του. Για αυτό και συνεχίζουμε μόνο εφόσον υπάρχει στόχος και εφόσον είναι εχθρικός προς την ομάδα μας συνεχίζουμε με το να ελέγξουμε αν είναι ζωντανός. Στο τέλος ελέγχουμε αν είμαστε σε ικανοποιητική απόσταση από τον στόχο και καλούμε την **cast()**.

Πίνακας 5.4.11: Η συνάρτηση ActiveAbility::cast()

```
void ActiveAbility::cast() {
	finished = false;
	owner->face(target);
	if (hasCastingAnimation) {
		owner->setActionAnimation(abilityClass.getCastAnimation());
	}
	owner->setCastingState();
	if (hasCastingSound) {
		castingSound->play();
	}
```

```
	abilityClass.onApply(this, owner, target);
	for (auto projectile : projectiles) {
		projectile->charge();
	}
	state = State::Casting;
	WorldEvent ev;
	ev.setAsAbilityUseEvent(owner, this);
	ev.broadcast();
}
```

Καλώντας την **cast()** σημάνουμε ότι η ικανότητα χρησιμοποιήθηκε με επιτυχία και μπαίνει στην κατάσταση **Casting**. Για αυτό και επαναφέρουμε το flag **finished** σε false. Εξαναγκάσουμε τον ιδιοκτήτη να αλλάξει κατεύθυνση και να κοιτάξει τον στόχο μέσω της **face(...)** για λόγους εμφάνισης. Στην συνέχεια ελέγχουμε αν η ικανότητα έχει κάποιο ειδικό Animation στο AnimationClass του ιδιοκτήτη και εφόσον υπάρχει, τότε κάνουμε τον ιδιοκτήτη να αλλάξει σε αυτό και να μπει σε κατάσταση **Casting**. Το ίδιο κάνουμε και για τον ήχο, που όμως ο ήχος θα παιχτεί από το ActiveAbility. Μετά καλούμε την **onApply** Lua Callback. Σε αυτή την callback οι προγραμματιστές μπορούν να ενεργοποιήσουν ότι Pojectiles θέλουν καθώς μετά κάνουμε όλα τα Projectiles που έχουν ενεργοποιηθεί (μέσω της Lua καλώντας την invokeProjectile) να γίνουν **charge()**. Θέτουμε την κατάσταση του Ability σε **Casting** και σηκώνουμε ένα συμβάν χρήσης ικανότητας.

Μετά την κλήση αυτής της συνάρτησης, δεν γίνεται τίποτα άλλο. Επειδή η ικανότητα έχει διαρκής λειτουργία (Updateable) τα υπόλοιπα τα αναλαμβάνει η **update()**.

Πίνακας 5.4.12: Η συνάρτηση ActiveAbility::update()

```
void ActiveAbility::update(Float elapsedTime) {
	switch (state) {
	case State::Casting:
		frameCounter -= elapsedTime;
		if (frameCounter <= 0) {
			release();
			frameCounter = coolDown;
			state = State::onCoodown;
		}
		break;
	case State::onCoodown:
		frameCounter -= elapsedTime;
		if (frameCounter <= 0) {
			state = State::Ready;
			frameCounter = castTime;
		}
		break;
	default:
		break;
```

```
	}
	{
		Map& curMap = WorldManager::getInstance().getCurrentMap();
		for (auto itr = projectiles.begin();
				itr != projectiles.end();) {
			if (!(*itr)->isToBeDeleted()) {
				++itr;
			} else {
				curMap.removeObject((*itr));
				itr = projectiles.erase(itr);
			}
		}
	}
	if (animationChanged && owner->isAnimationFinished()) {
		owner->fullResetState();
		animationChanged = false;
		finished = true;
	}
}
```

Εδώ αρχικά ελέγχουμε τι κατάσταση έχουμε. Αν είμαστε την **Casting** τότε πρέπει να μετρήσουμε τον χρόνο που θα κάνουμε **Casting**. Πρέπει να ήμαστε σε αυτή την κατάσταση για **castTime** χρόνο. Ο χρόνος αυτός έχει μπει προηγουμένως στο μέλος **frameCounter**. Έτσι μειώνοντας τον χρόνο που περνάει μετά από κάθε κλήση της συνάρτησης, τον μετράμε αποτελεσματικά και όταν αυτό το μέλος γίνει 0 ή μικρότερο, τότε σημαίνει ότι το **Casting** τελείωσε και πρέπει να εξαπολύσουμε την ικανότητα. Αυτό γίνεται με την **release()**.

Πίνακας 5.4.13: Η συνάρτηση ActiveAbility::release()

```
void ActiveAbility::release() {
	if (hasReleaseAnimation) {
		owner->setActionAnimation(abilityClass.getReleaseAnimation());
	}
	if (hasReleaseSound) {
		releaseSound->play();
	}
	if (hasCastingSound) {
		castingSound->stop();
	}
	owner->offsetMana(-manaCost);
	for (auto projectile : projectiles) {
		projectile->release();
	}
	state = State::onCoodown;
	animationChanged = true;
}
```

Όπως και με την **cast()** έτσι και εδώ ελέγχουμε αν έχουμε Animation για αυτή την κατάσταση και ήχο. Αν έχουμε τα θέτουμε σε λειτουργία και σταματάμε αυτά της

προηγούμενης κατάστασης αν παίζουν ακόμα. Μετά αφαιρούμε το ποσό του mana που χρειάζεται για να εκτελεστεί η ικανότητα και καλούμε την **release()** για όλα τα ενεργά projectiles. Τελικά αλλάζουμε την κατάσταση σε **onCooldown** και θέτουμε το **animationChanged** σε true.

Πίσω στην **update()**, σε επόμενο Frame η εκτέλεση θα μπει στην switch στο σημείο που είναι για το **onCooldown**. Εκεί πρέπει να μετρήσουμε πάλι τον χρόνο που χρειάζεται μέχρι να βάλουμε την κατάσταση σε **Ready** για να μπορεί να επαναχρησιμοποιηθεί η ικανότητα. Επίσης θέτουμε τον χρόνο του **frameCounter** στο **castTime** όπως είχαμε κάνει και πριν που το θέσαμε σε **coolDown**.

Ανεξάρτητα σε ποια κατάσταση βρίσκεται το Ability, τα Projectiles που έχουν ενεργοποιηθεί πρέπει να ενημερώνονται πάντα. Για αυτό φροντίζει ο χάρτης που τα προσθέτουμε σε αυτόν. Όμως εμείς εδώ πρέπει να φροντίζουμε για το πόσο θα πρέπει να ζουν αυτά τα Projectiles. Έτσι ελέγχουμε αν τα βλήματα έχουν χαρακτηριστεί για καταστροφή και εφόσον είναι έτσι, τότε τα αφαιρούμε από τον χάρτη και από τον πίνακα των ενεργών Projectiles. Περισσότερα για αυτά παρακάτω.

Ένα τελευταίο πράγμα που γίνεται σε κάθε Frame είναι ο έλεγχος το αν στον ιδιοκτήτη έχουμε αλλάξει το Animation πρόσφατα. Αυτό το κάνουμε μέσω του flag **animationChanged**. Κάθε φορά που αλλάζουμε το Animation το θέτουμε σε true όπως είδαμε. Αν εδώ είναι true, τότε ελέγχουμε αν έχει τελειώσει το Animation που του αλλάξαμε. Αν τελείωσε, τότε πρέπει να επαναφέρουμε τον ιδιοκτήτη στο κανονικό του Animation που είναι Idle. Αυτό το κάνουμε με την συνάρτηση **fullResetState()**. Τέλος επαναφέρουμε το flag και θέτουμε ότι το Ability τελείωσε όλη του την διάρκεια μεσώ του flag **finished**.

b) Effects

Όσο αναλύαμε τα Abilities, ήρθαμε σε επαφή με κάποια αντικείμενα τύπου **Effects**. Αυτά τα αντικείμενα διαχειρίζονται τις επιδράσεις που μπορεί να δημιουργήσει μια ικανότητα όπως είχαμε πει στα πρώτα κεφάλαια.

Όπως και με τα **Abilities** έτσι και εδώ, υπάρχουν **EffectClass** που τα οποία κατασκευάζουν τα τελικά **Effect** που θα χρησιμοποιηθούν από κάποιο Lifeform.

Πίνακας 5.4.14: Η κλάση EffectClass

```
class EffectClass {
public:

     const std::string& getName() const;
     const LuaFunctor& getOnApply() const;
     const LuaFunctor& getOnRemove() const;
     const LuaFunctor& getOnTick() const;
     Float getUpTime() const;
     Float getTickEvery() const;
     bool isStackable() const;

     Effect* getNewEffect(Lifeform *target, int level) const;

     EffectClass();
     EffectClass(const LuaTable& table);
     EffectClass(lua_Object table);
     ~EffectClass();
private:
     std::string name;

     LuaFunctor onApply;
     LuaFunctor onRemove;
     LuaFunctor onTick;

     Float uptime;
     Float tickEvery;

     bool durable;
     bool overTime;
     bool stackable;
};
```

Όπως είδαμε στην φόρτωση του **ActiveAbilityClass**, τα **EffectClass** φορτώνονται μόνο εκεί και από πίνακα. Μέσω αυτού του πίνακα γεμίζουν όλα τα μέλη όταν το αντικείμενο κατασκευάζεται. Από εκεί και πέρα τα αντικείμενα **Effect** κατασκευάζονται από την συνάρτηση **getNewEffect(...)**.

Πίνακας 5.4.15: Η συνάρτηση EffectClass::getNewEffect()

```
Effect* EffectClass::getNewEffect(Lifeform* target, int level) const {
     if (overTime) {
          return new OverTimeEffect(*this, target, level);
     } else if (durable) {
          return new DurableEffect(*this, target, level);
     } else {
          return new Effect(*this, target, level);
     }
}
```

Εδώ βλέπουμε ότι ανάλογα τον τύπο του Effect κατασκευάζουμε διαφορετικό αντικείμενο που επιστρέφουμε. Αυτό γιατί υπάρχει διαφορετική συμπεριφορά ανάλογα τον τύπο του. Οι τύποι **DurableEffect** και **OverTimeEffect** είναι επιδράσεις που έχουν πεπερασμένη διάρκεια σε αντίθεση με τον απλό τύπο που έχει διάρκεια μέχρι να αφαιρεθεί ρητά από το Lifeform που το έχει.

Πίνακας 5.4.16: Η κλάση DurableEffect

```
class DurableEffect: public Effect {
public:
	Float getRemainingTime() const {
		return frameCounter;
	}

	Float getUpTime() const;
	void setUpTime(Float upTime);

	void reset();
	virtual void update(Float elapsedTime);
	DurableEffect(const EffectClass& effCls, Lifeform *target,
                                                        int level);
	virtual ~DurableEffect();
protected:
	enum class EffectState {
		Begin, Running, End
	};
	Float frameCounter;
	Float upTime;

	EffectState state;
};
```

Η αρμοδιότητα της παραπάνω κλάσης είναι η επίδραση να είναι παροδική, σε χρόνο που ορίζεται από το **upTime**. Πρέπει στην αρχή να καλεί την onApply() και στο τέλος την onRemove().

Πίνακας 5.4.17: Η συνάρτηση DurableEffect::update()

```
void DurableEffect::update(Float elapsedTime) {
	switch (state) {
	case EffectState::Running:
		frameCounter -= elapsedTime;
		if (frameCounter <= 0) {
			state = EffectState::End;
		}
		break;
	case EffectState::Begin: {
		LuaNumber lvl(level);
		effClass->getOnApply()( { this, owner, &lvl, source });
	}
```

```
            state = EffectState::Running;
            break;
      case EffectState::End: {
            LuaNumber lvl(level);
            effClass->getOnRemove()( { this, owner, &lvl, source });
      }
            finished = true;
            break;
      }
}
```

Βλέπουμε ότι και εδώ έχουμε καταστάσεις. Αυτές οι καταστάσεις είναι η αρχή (**Begin**), η διάρκεια (**Running**) και το τέλος (**End**). Η κατάσταση κατά την κατασκευή είναι η Begin, έτσι στο πρώτο Frame που θα καλεστεί η συνάρτηση θα καλέσει την **onApply()** και θα θέσει την κατάσταση σε **Running**. Έτσι στα επόμενα Frame θα κυλίσει ο χρόνος κανονικά μέχρις ότου να τελειώσει όπου θα τεθεί η κατάσταση σε End και στο επόμενο Frame θα καλεστεί η **onRemove()**. Ο τρόπος που το υλοποιήσαμε έτσι κάνει την διάρκεια του Effect να έχει 2 επιπλέον Frame στην κανονική του διάρκεια. Αυτό το κάνουμε για λόγους σαφήνειας εσωτερικά στο Lifeform.

Η διαφορά του **DurableEffect** και **OverTimeEffect** είναι μόνο στην κατάσταση που κυλάει ο χρόνος. Εκεί υπάρχει ένας δεύτερος μετρητής που μετράει κάθε πότε θα πρέπει να καλεστεί η **onTick()**.

Όπως ξέρουμε τα Effects μπαίνουν στα Lifeforms μέσω της αντίστοιχης συνάρτησης. Μόνο κατασκευασμένα αντικείμενα Effect μπορούν να προστεθούν, έτσι η προσθήκη τους γίνεται ρητά μέσω συναρτήσεων Lua (πχ στην onApply() κάποιου ActiveAbility). Τόσο η κατασκευή και η απόδοση γίνεται μέσω της **applyEffect(...)** της **ActiveAbilityClass** που περιέχει όλα τα διαθέσιμα **EffectClass**.

Πίνακας 5.4.18: Η συνάρτηση ActiveAbilityClass::applyEffect()

```
void ActiveAbilityClass::applyEffect(const std::string& name, Lifeform*
target,
            int level, ActiveAbility* ability) const {
      if (target != nullptr) {
            auto itr = effects.find(name);
            if (itr != effects.end()) {
                  Effect* eff = itr->second.getNewEffect(target, level);
                  eff->setSource(ability);
                  target->addEffect(eff);
            } else {
                  Logger::getInstance().write(
                              "At Ability " + name
                                          + ". Called \'applyEffect\' with
                                                      invalid name",
                              Logger::MessageType::LuaError);
            }
```

```
      } else {
            Logger::getInstance().write(
                        "At Ability " + name
                                    + ". Called \'applyEffect\' with NULL
                                                Target",
                        Logger::MessageType::LuaError);
      }
}
```

Εδώ, αρχικά ελέγχουμε αν ο στόχος του Effect υπάρχει. Εφόσον υπάρχει τότε ψάχνουμε τον πίνακα **effects** για το **EffectClass** με το όνομα που μας δόθηκε (υπόψη τα ονόματα των EffectClass δίνονται βάση του ονόματος του πίνακα που είχε τα δεδομένα του). Εφόσον το βρούμε και αυτό, τότε κατασκευάζουμε αυτό το Effect, του δίνουμε σαν προέλευση το **ActiveAbilty** που έχει και τελικά το προσθέτουμε στον στόχο.

Στην μεριά του Lifeform γίνονται κάποιοι περαιτέρω έλεγχοι πριν προστεθεί πραγματικά το Effect.

Πίνακας 5.4.19: Η συνάρτηση Lifeform::addEffect()

```
void Lifeform::addEffect(Effect* eff) {
      // if eff is not null
      if (eff != nullptr) {
            if (!eff->getClass().isStackable()) {
                  auto itr = nonStackableEffects.find(
                                          eff->getClass().getName());
                  if (itr != nonStackableEffects.end()) {
                        delete eff;
                        return;
                  } else {
                        nonStackableEffects.insert(
                                          eff->getClass().getName());
                  }
            }
            effects.push_back(eff);
      }
}
```

Αυτό που ελέγχουμε επιπλέον είναι αν το Effect είναι Stackable, δηλαδή αν θα πρέπει να προστεθεί αν υπάρχει ήδη. Αν δεν είναι stackable, τότε το ψάχνουμε στο Set **nonStackableEffects**. Εκεί υπάρχουν τα ονόματα των Effects που πρέπει να είναι μοναδικά. Έτσι αν το βρούμε, τότε καταστρέφουμε αυτό που μας δόθηκε και επιστρέφουμε. Αν όμως δεν το βρούμε τότε σημαίνει ότι δεν το έχουμε ξανά λάβει αυτό το Effect και πρέπει να το προσθέσουμε στο Set καθώς και στον πίνακα των **effects**.

Το τελευταίο πράγμα που θα δούμε εδώ είναι το πως διαχειρίζεται τα Effects το Lifeform. Αυτό γίνεται μέσω της **updateEffects()** που καλείται όπως είδαμε κάθε φορά που καλείται η **update()** και είναι ζωντανό το Lifeform.

Πίνακας 5.4.20: Η συνάρτηση Lifeform::updateEffects()

```
void Lifeform::updateEffects(Float elapsedTime) {
    // for all the effects in the effects list
    for (auto itr = effects.begin(); itr != effects.end();) {
          // Update the effect
          (*itr)->update(elapsedTime);
          // If effect isFinished (duration is over, etc)
          if ((*itr)->isFinished()) {
                auto nonS = nonStackableEffects.find(
                                    (*itr)->getClass().getName());
                if (nonS != nonStackableEffects.end()) {
                      nonStackableEffects.erase(nonS);
                }

                // delete the effect
                delete *itr;
                // erase it's pointer from the list to
                // avoid segmentation faults. We get the next iterator
                // from erasing it
                itr = effects.erase(itr);
          } else {
                // if the effect is not Finished, the just carry on
                ++itr;
          }
    }
}
```

Στον πίνακα **effects** έχουμε όλα τα Effects που είναι ενεργά αυτή την στιγμή στο Lifeform. Έτσι χρειάζεται να καλέσουμε την **update()** για καθένα από αυτά. Αφού την καλέσουμε για κάποιο, τότε πρέπει να ελέγξουμε αν μετά την κλήση αυτή το Effect τελείωσε την δουλειά του (πχ στην περίπτωση του DurableEffect η δουλεία του τελειώνει όταν τελειώσει η διάρκειά του). Αν έχει τελειώσει, τότε σημαίνει ότι θα το αφαιρέσουμε και θα το καταστρέψουμε. Σε αυτή την περίπτωση ελέγχουμε αν υπάρχει στον πίνακα **nonStackableEffects** προκειμένου να το αφαιρέσουμε και να μπορεί να ξαναμπεί στο μέλλον αν χρειαστεί. Τελικά το καταστρέφουμε και το αφαιρούμε από τον πίνακα.

5.5 Τα βλήματα (Projectiles)

Πολλές φορές κάποια ActiveAbilities χρειάζονται να εκτοξεύσουν ένα βλήμα όπως πχ μια σφαίρα ενέργειας και όταν αυτό ακουμπήσει τον στόχο να του προκαλέσει κάτι. Αυτό επιτυγχάνεται από τα διάφορα είδη βλημάτων που παρέχονται. Όπως και με ότι συσχετίζεται με τα Abilities έτσι και εδώ τα βλήματα (**Projectiles**) κατασκευάζονται από αντικείμενα **ProjectileClass** όπου παίρνουν τα δεδομένα τους από τους ειδικούς πίνακες Lua μέσα στον πίνακα του **AbilityClass** όπως είδαμε στην κατασκευή του.

Πίνακας 5.5.1: Η κλάση ProjectileClass

```
class ProjectileClass {
public:

     enum class SpawnOffset {
          Up = 0, Right, Left, Down
     };

     enum class ProjectileType {
          Normal, Directional, Seeking
     };

     const LuaFunctor& getOnCollision() const;
     void setOnCollision(const LuaFunctor& onCollisionF);
     const LuaFunctor& getOnDestinationReach() const;
     void setOnDestinationReach(const LuaFunctor& onDestinationReachF);
     const LuaFunctor& getOnRelease() const;
     void setOnRelease(const LuaFunctor& onRelease);
     const AnimationClass& getAnimationClass() const;
     Float getSpeed() const;
     bool isDirectionalRotate() const;
     const Vector2& getOffset(SpawnOffset offset) const;

     void load(const LuaTable& table);

     void onCollision(Projectile *invoker, Object *obj) const;
     void onRelease(Projectile *invoker) const;
     void onDestinationReach(Projectile *invoker) const;

     Projectile* create(ActiveAbility *parent) const;

     ProjectileClass();
     ProjectileClass(const LuaTable& table);
     virtual ~ProjectileClass();
private:

     Vector2 spawnOffsets[4];
     std::string name;

     const AnimationClass *animClass;
     Float speed;
     LuaFunctor onReleaseF;
```

```
      LuaFunctor onCollisionF;
      LuaFunctor onDestinationReachF;
      ProjectileType type;
      bool directionalRotate;

};
```

Τα βλήματα όπως και με τα άλλα αντικείμενα, έχουν ένα όνομα (μοιράζεται). Πέρα από αυτό έχουν και μια **AnimationClass** από όπου θα τροφοδοτούνται με τα Animation που χρειάζονται. Τα Animation θα είναι αντίστοιχα με αυτά του Lifeform, δηλαδή όταν θα κινείται αριστερά θα πάει να αναζητήσει το ίδιο όνομα Animation που θα αναζητούσε αν ήταν Lifeform.

Η κλάση παράγει ανάλογα τον τύπο **type** αντικείμενα είτε Projectile, DirectionalProjectile είτε SeekingProjectile που θα τα δούμε αυτά παρακάτω. Αυτό που θέλουμε να δούμε εδώ είναι τα μέλη. Βλέπουμε ότι έχουμε 4 **Vector2** που θα περιέχουν τα Offsets που θα προστεθούν στην τοποθεσία του βλήματος μόλις δημιουργηθεί. Όταν δημιουργείται ένα Projectile, του δίνεται η τοποθεσία του ιδιοκτήτη που είναι το Lifeform που το προκάλεσε. Όμως πολλές φορές το να εμφανιστεί ακριβώς εκεί που βρίσκεται ο ιδιοκτήτης δεν είναι επιθυμητό, για αυτό και έχουμε αυτά τα 4 Offsets που προστίθενται στην τοποθεσία αυτή. Είναι 4 για κάθε μία από τις 4 κατευθύνσεις προκειμένου να μπορούμε να έχουμε διαφορετικά. Πέρα από αυτό, έχουμε και τις συναρτήσεις Lua που θα καλεστούν. Η **onReleaseF** θα πρέπει να καλεστεί όταν καλεστεί η **release()** από το projectile. Η **onCollisionF** θα πρέπει να καλείται κάθε φορά που το βλήμα συγκρούεται με κάτι. Τέλος η **onDestinationReachF** θα πρέπει να καλεστεί όταν θεωρείται ότι το βλήμα έφτασε στον στόχο του. Υπάρχει και ένα τελευταίο μέλος, το **directionalRotate**. Αυτό ανάλογα την τιμή του, θα πρέπει το Projectile να κάνει περιστροφή του Animation ανάλογα την κατεύθυνση που έχει το Projectile. Περισσότερα για αυτό παρακάτω.

Πίνακας 5.5.2: Η κλάση Projectile

```
class Projectile: public Object {
public:

      enum class ProjectileState {
            Charging, Moving, Dying, Dead, Inactive
      };

      void update(Float elapsedTime);
      virtual void onCollidedWith(Object *other);
      void draw(Float x, Float y, Float rotation = 0.0f,
```

```
                Float scale = 1.0f) const;
    void accept(Visitor& visitor);
    void clearCollidedObjects() {
          collidedObjects.clear();
    }
    void onClick(Object *other) {
    }
    void destroy();
    void charge();
    void release();

    void abort();
    void applyOffsets();

    bool isToBeDeleted() const {
          return (state == ProjectileState::Dead);
    }

    void setTargetLocation(Float x, Float y);
    ActiveAbility* getParentAbility();

    virtual void resetState() {
    }

    Projectile(const ProjectileClass *cls, ActiveAbility *parent,
                Float x,
                Float y);
    virtual ~Projectile();
protected:
    enum class AnimationDirection {
          Right, Left, Up, Down
    };
    ZSet<Object*> collidedObjects;
    AnimationHandler animation;
    const ProjectileClass *projectileClass;
    ActiveAbility *parentAbility;
    View *camera;
    Vector2 targetPosition;
    Vector2 direction;
    ProjectileState state;
    LifeformState animDir;
    Float rotation;

    virtual void move(Float elapsedTime);
    void calculateAnimationDirection(LifeformState::Action action,
                bool changeAnimation = false);
};
```

Η παραπάνω κλάση είναι η βασική όλων των Projectiles. Όπως βλέπουμε κληρονομεί την **Object** για να μπορεί να προστεθεί εντός του χάρτη και να μπορεί να ενημερωθεί. Υπάρχει το set με τα αντικείμενα που έχει συγκρουστεί το βλήμα και λέγεται **collidedObjects**. Όπως και με όλα τα αντικείμενα που έχουν Animation, έτσι και εδώ υπάρχει ένας **AnimationHandler** προκειμένου να ασχολείται με αυτά. Έχουμε δείκτη προς το **ProjectileClass** καθώς και το **ActiveAbility** που έχει συγγένεια το βλήμα.

Επίσης υπάρχει και δείκτης προς την κύρια κάμερα του παιχνιδιού προκειμένου να έχουμε γρήγορα πρόσβαση σε αυτή. Η κάμερα θα μας βοηθήσει ώστε να ξέρουμε αν το βλήμα βγήκε εκτός οθόνης προκειμένου να το καταστρέψουμε. Έχουμε και τα 2 διανύσματα που θα μας βοηθήσουν για την κίνηση του βλήματος, το σημείο που θα πρέπει να πάει το βλήμα (**targetPosition**) και την κατεύθυνση **direction**. Η κατάσταση του βλήματος ορίζεται από το **state**, η κατεύθυνση του Animation από το **animDir** και η περιστροφή του από το **rotation**.

Τα βλήματα ορίζονται από τις 3 βασικές καταστάσεις που έχουν: την κατάσταση φόρτωσης (**Charge**), την κατάσταση κίνησής και την καταστροφή τους. Όταν ένα βλήμα δημιουργείται, τότε μπαίνει σε κατάσταση **Charge**. Συνήθως αυτή η κατάσταση ταυτίζεται με την κατάσταση **Cast** του **ActiveAbility** που το δημιουργεί. Έτσι όσο το ActiveAbility βρίσκεται σε Cast τότε το Βλήμα βρίσκεται σε κατάσταση **Charge**. Σε αυτή την κατάσταση μπαίνει το βλήμα μέσω της συνάρτησης **charge()**.

Πίνακας 5.5.3: Η συνάρτηση Projectile::charge()

```
void Projectile::charge() {
     animation.setAnimation(
                 animDir.getDirection() | LifeformState::Action::Dead);
     animation.getMainAnimationPlayer().play();
     state = ProjectileState::Charging;
}
```

Ανάλογα την κατεύθυνση που έχει υπολογιστεί προς τον στόχο, θέτουμε το Animation από το Action **Dead**. Η σύμβαση λέει ότι για τα Projectile το Action Dead θα έχει τα Animation για το Charge. Έτσι το θέτουμε και τελικά θέτουμε την κατάσταση σε **Charging**. Το επόμενο που θα γίνει είναι το **update()** στην επόμενη κλήση του.

Πίνακας 5.5.4: Η συνάρτηση Projectile::update()

```
void Projectile::update(Float elapsedTime) {
     switch (state) {
     case ProjectileState::Charging:
           if (!animation.getMainAnimationPlayer().isPlaying()) {
                 animation.setAnimation(
                             animDir.getDirection()
                                         | LifeformState::Action::Idle);
                 animation.getMainAnimationPlayer().play();
           }
           break;
     case ProjectileState::Moving:
           move(elapsedTime);
```

```
        WorldManager::getInstance().getCurrentMap().isRectangleColliding(
                        bounding, this);
                break;
        case ProjectileState::Dying:
                if (!animation.getMainAnimationPlayer().isPlaying()) {
                        visible = false;
                        state = ProjectileState::Dead;
                }
                break;
        case ProjectileState::Dead:
        case ProjectileState::Inactive:
                break;
        }
        animation.update(elapsedTime);
}
```

Στην κατάσταση **Charging** αυτό που θα γίνει εδώ είναι ότι θα ελέγξουμε αν το Animation που κάνει charge έχει τελειώσει. Αν έχει τελειώσει τότε πρέπει να βάλουμε το Animation που θα μπαίνει σε αυτή την περίπτωση που βάση σύμβασης είναι στο Action Idle.

Όταν τελειώσει το Casting του Ability, τότε όπως είδαμε σε όλα τα ενεργά βλήματα καλείται η **release()**. Με αυτή την κλήση το βλήμα θα πρέπει να αρχίσει να κάνει το ταξίδι του προς τον στόχο του. Η κλήση αυτή γίνεται όπως και με την **charge()** με αλλαγή στο σωστό Animation και θέτοντας την κατάσταση του βλήματος σε **Moving**. Πέρα από αυτό καλείται και η **onRelease()** από την ProjectileClass του. Αφού αλλάξαμε την κατάσταση, στο επόμενο Frame στην **update()** θα καλεστεί η **move()**.

Πίνακας 5.5.5: Η συνάρτηση Projectile::move()

```
void Projectile::move(Float elapsedTime) {
        Float speed = projectileClass->getSpeed() * elapsedTime;
        offsetPosition(direction.getX() * speed, direction.getY() * speed);
        if (projectileClass->isDirectionalRotate()) {
                rotation = std::atan2(direction.getY(),
                                                direction.getX()) * 57.2957795f;
        } else {
                calculateAnimationDirection(LifeformState::Action::Moving);
        }
        if (!camera->isInView(*this)) {
                state = ProjectileState::Dead;
        }
        if (position.getDistance(targetPosition) < (speed + 0.1f)) {
                setPosition(targetPosition.getX(), targetPosition.getY());
                projectileClass->onDestinationReach(this);
        }
}
```

Όπως και με κάθε κίνηση, υπολογίζουμε την πραγματική ταχύτητα που πρέπει να έχουμε βάση του ρυθμού που τρέχει το παιχνίδι. Στην συνέχεια μετακινούμε απευθείας το βλήμα στην επόμενη θέση χωρίς να ελέγξουμε για συγκρούσεις. Αυτό γιατί τα βλήματα δεν μπορούν να σταματήσουν από συγκρούσεις με άλλα αντικείμενα. Μετά ελέγχουμε το flag **directionalRotate**. Αν είναι true, τότε πρέπει να κάνουμε rotate το Animation της κίνησης που έχουμε αυτή την στιγμή βάση της κατεύθυνσης που έχουμε. Αυτό γίνεται θεωρώντας ότι το Animation κοιτάζει δεξιά. Έτσι αν η κατεύθυνση είναι πάνω αριστερά, θα περιστρέφει με τέτοια κλίση ώστε να κοιτάζει προς αυτή την κατεύθυνση. Για αυτό και πρέπει να υπολογίσουμε το **rotation** που θα εφαρμόσουμε όταν το Projectile γίνει **draw()**. Αυτό γίνεται μέσω της τοξοεφαπτομένης της κατεύθυνσης που μας επιστρέφει την κλίση σε μοίρες. Όμως δεν χρειάζεται μόνο αυτό, καθώς η περιστροφή γίνεται σε ακτίνια, για αυτό και πολλαπλασιάζουμε τις μοίρες που πήραμε επί 57.2957795f για πάρουμε το σωστό αποτέλεσμα σε ακτίνια.

Αν δεν χρειάζεται να γίνει αυτό, τότε απλά υπολογίζουμε το Animation για την κατεύθυνση που έχουμε. Αυτό το κάνουμε συνέχεια γιατί μπορεί η κατεύθυνση να αλλάξει κατά την διάρκεια. Μετά ελέγχουμε αν το βλήμα είναι εντός οθόνης και αν δεν είναι τότε του βάζουμε την κατάσταση σε Dead προκειμένου να αφαιρεθεί. Τέλος ελέγχουμε αν φτάσαμε στην τοποθεσία-στόχο. Αν φτάσαμε τότε διορθώνουμε τις συντεταγμένες μας και καλούμε την **onDestinationReach()**.

Πίσω στην **update()** αφού τελείωσε η **move()** τότε κάνουμε κάτι περίεργο θα έλεγε κανείς. Καλούμε την **isRectangleColliding()** για το βλήμα χωρίς να ελέγχουμε το τι επιστρέφει. Αυτό το κάνουμε προκειμένου να ελέξει ο χάρτης για συγκρούσεις και να καλέσει τις callback εφόσον γίνονται.

Στη κατάσταση **Dying** μπαίνει το βλήμα όταν καλεστεί η **destroy()** και σε αυτή την περίπτωση όπως βλέπουμε εμφανίζει ένα τελευταίο Animation πριν εξαφανιστεί και μπει στην κατάσταση **Dead**. Μετά το ActiveAbility που έχει το βλήμα, θα ελέγξει την κατάσταση του και εφόσον είναι Dead, θα το αφαιρέσει από τον κόσμο.

Πέρα από τα απλά βλήματα που τους θέτεται μια συντεταγμένη για να κινηθούν, υπάρχουν και κλάσεις για δημιουργία βλημάτων με πιο εξειδικευμένες κινήσεις. Για παράδειγμα στην περίπτωση του **DirectionalProjectile**, θέλουμε το βλήμα ανάλογα την κατεύθυνση που κοιτάζει ο ιδιοκτήτης, να πάρει την ίδια το βλήμα και να κινηθεί προς αυτή. Αυτό γίνεται θέτοντας την κατεύθυνση που θα κινηθεί το βλήμα προς κάποια από τις 4.

Πίνακας 5.5.6: Η συνάρτηση DirectionalProjectile::calculateTargetPosition()

```
void DirectionalProjectile::calculateTargetPosition() {
     switch (parentAbility->getOwner().getState().getDirection()) {
     case LifeformState::Direction::Down:
           direction.set(0.0f, 1.0f);
           break;
     case LifeformState::Direction::Up:
           direction.set(0.0f, -1.0f);
           break;
     case LifeformState::Direction::Left:
           direction.set(-1.0f, 0.0f);
           break;
     case LifeformState::Direction::Right:
           direction.set(1.0f, 0.0f);
           break;
     default:
           break;
     }
}
```

Όπως βλέπουμε εδώ, απλά θέτουμε την κατεύθυνση της κίνησης και όχι τις συντεταγμένες. Έτσι το βλήμα θα κινηθεί προς κάποια από αυτές μέχρι να βγει εκτός οθόνης όπου θα αλλάξει κατάσταση σε **Dead**.

Στην περίπτωση του **SeekingProjectile** θα πρέπει το βλήμα να κατευθύνεται στον στόχο μέχρι να τον πετύχει. Αυτό επιτυγχάνεται με το να υπολογίζουμε σε κάθε Frame την κατεύθυνση σε σχέση με την θέση που έχει ο στόχος εκείνη την στιγμή. Πέρα από αυτό, τα υπόλοιπα είναι ίδια για την κίνηση.

5.6 Τεχνητή νοημοσύνη (Artificial Intelligence)

Για να μπορεί ο κόσμος να είναι πιο “ζωντανός”, έχει δημιουργηθεί ένα σύστημα απλής τεχνητής νοημοσύνης. Αυτό περιλαμβάνει ένα σύνολο συμπεριφορών που μπορούν να έχουν τα Lifeforms. Οι συμπεριφορές αυτές διαχωρίζονται σε 2 κατηγορίες: την γενική συμπεριφορά **(GeneralBehaviour)** και συμπεριφορά κίνησης **(Movement Behaviour)**. Το πρώτο ασχολείται με την συμπεριφορά με άλλα Lifeforms και το δεύτερο ασχολείται με τις κινήσεις που θα κάνει το Lifeform.

a) Παθητική Συμπεριφορά

Στην παθητική συμπεριφορά το Lifeform αδιαφορεί για τα άλλα Lifeform που υπάρχουν τριγύρω. Δεν κάνει στην ουσία τίποτα αλλά είναι μια καλή ευκαιρία να δούμε την βάση όλου του συστήματος. Όπως και με τα άλλα κομμάτια του υποσυστήματος

RPG, έτσι και εδώ υπάρχει μια κλάση που λειτουργεί σαν μηχανή παραγωγής αντικειμένων **Behaviour**.

Πίνακας 5.6.1: Η κλάση BehaviourClass

```
class BehaviourClass {
public:
     virtual Behaviour* getNewBehaviour(Lifeform& owner) const=0;

     BehaviourClass() {

     }
     virtual ~BehaviourClass() {

     }
};
```

Η παραπάνω abstract κλάση είναι η βάση για όλα τις υπό-κλάσεις που θα εξειδικεύουν την λειτουργία τους, όπως η **PassiveBehaviourClass** η οποία υλοποιεί απλά την **getNewBehaviour()**.

Πίνακας 5.6.2: Η συνάρτηση PassiveBehaviourClass::getNewBehaviour()

```
Behaviour* PassiveBehaviourClass::getNewBehaviour(Lifeform& owner) const
{
     return new PassiveBehaviour(owner);
}
```

Η συνάρτηση πρέπει να επιστρέφει ένα αντικείμενο **Behaviour** που θα χρησιμοποιεί το Lifeform για συμπεριφορά. Το αντικείμενο είναι εξειδικευμένο ανάλογα την λειτουργία.

Πίνακας 5.6.3: Η κλάση Behaviour

```
class Behaviour:public Updateable {
public:

     virtual void onDeath()=0;

     Behaviour(Lifeform &owner) :
                 owner(&owner) {

     }
     virtual ~Behaviour() {

     }
protected:
     Lifeform *owner;
};
```

Όλες οι υπό κλάσεις της Behaviour πρέπει να έχουν σαν μέλος το Lifeform που θα “ελέγχουν” και να υλοποιήσουν την **onDeath()** που θα καλεί το Lifeform όταν πεθαίνει και πρέπει να το ξέρει το **Behaviour**. Πέρα από αυτό, οι υπό κλάσεις πρέπει να υλοποιήσουν την **update()** που προέρχεται από την **Updateable** όπου εκεί θα γίνεται ο έλεγχος του Lifeform. Στην περίπτωση του **PassiveBehaviour** οι υλοποιήσεις αυτές θα είναι κενές καθώς το Lifeform δεν θα κάνει τίποτα.

b) Επιθετική Συμπεριφορά

Η δεύτερη μορφή γενικής συμπεριφοράς είναι η επιθετική ή **AgressiveBehaiour**. Σε αυτή την περίπτωση το Lifeform ελέγχει τριγύρω του για Lifeforms που βρίσκονται σε ομάδα αντιπαλότητας με την δικιά του και εφόσον είναι, τότε αρχίζει να επιτίθεται στο πρώτο που θα βρει.

Οι ομάδες αντιπαλότητας ορίζονται από τα αντικείμενα **Faction**. Αυτά τα αντικείμενα είναι κοινόχρηστα στα Lifeforms και ορίζουν μια λίστα εχθρικών **Faction**.

Πίνακας 5.6.4: Η κλάση Faction

```
class Faction {
public:

	bool isHostile(const std::string& faction) const {
		return hostile.find(faction) != hostile.end();
	}

	void setHostile(const std::string& faction) {
		hostile.insert(faction);
	}

	bool isHostile(const Faction& other) const {
		return isHostile(other.getName());
	}

	const std::string& getName() const;
	void setName(const std::string& name);

	Faction() :
			name("Neutral") {

	}
	~Faction();
private:
	std::string name;
	ZSet<std::string> hostile;
};
```

Κάθε Faction έχει ένα όνομα που το χαρακτηρίζει και μια λίστα από εχθρικά ονόματα Faction. Βάση αυτού τα Lifeforms ελέγχουν αν είναι εχθρικά με κάποια άλλα. Όλα τα αντικείμενα Faction βρίσκονται στον **FactionManager** από όπου και τα Lifeforms παίρνουν το Faction βάση του **LifeformClass** του.

Η επιθετική συμπεριφορά χαρακτηρίζεται από την ομάδα του Lifeform, την εμβέλεια που μπορεί να ελέγξει και την μέγιστη απόσταση που μπορεί να κινηθεί.

Πίνακας 5.6.5: Η κλάση AggressiveBehaviour

```
class AggressiveBehaviour: public Behaviour {
public:
     void update(Float elapsedTime);
     void onDeath();
     AggressiveBehaviour(Lifeform &owner, Float agroRange = 300,
                 Float driftRange = 600);
     virtual ~AggressiveBehaviour();
private:
     enum class State {
          Scanning, Attacking, FallBack, Moving, Casting
     };
     Vector2 initialPosition;
     Lifeform *target;
     ActiveAbility *curAbility;
     Float agroRange;
     Float driftRange;
     State state;

     void scan();
     void attack();
     void getAbility();
};
```

Όπως είναι αναμενόμενο, η κλάση λειτουργεί βάση καταστάσεων. Πιο ειδικά, το μέλος **initialPosition** υπάρχει για να αποθηκεύουμε την αρχική τοποθεσία που έχει το Lifeform σε περίπτωση που χρειαστεί να κινηθεί. Αυτό γιατί μπορεί προκειμένου να επιτεθεί, να πρέπει να πλησιάσει τον εχθρό και όταν τελειώσει την δουλειά του η απομακρυνθεί πολύ από αυτή την τοποθεσία, να πρέπει να εγκαταλείψει την επίθεση και να επιστρέψει σε αυτή. Το **target** θα περιέχει τον στόχο που θα επιτεθούμε όταν βρούμε κάποιον. Το **curAbility** θα περιέχει την ικανότητα που θα χρησιμοποιήσουμε στον στόχο.

Όπως είπαμε προηγουμένως, η όλη δουλειά στις συμπεριφορές γίνεται στην υλοποίηση της **update()**.

Πίνακας 5.6.6: Η συνάρτηση AggressiveBehaviour::update()

```
void AggressiveBehaviour::update(Float elapsedTime) {
            switch (state) {
            case State::Scanning:
                  scan();
                  break;
            case State::Attacking:
                  if (target->isAlive()) {
                        if (curAbility == nullptr) {
                              getAbility();
                        }
                        attack();
                  } else {
                        owner->moveToPosition(initialPosition);
                        state = State::FallBack;
                        target = nullptr;
                        break;
                  }
                  break;
            case State::Casting:
                  if (curAbility->isDone()) {
                        getAbility();
                        state = State::Attacking;
                  }
                  break;
            case State::Moving:
                  if (target->isAlive()) {
                        if (owner->getDistance(*target)
                                    < curAbility->getClass().getRange()) {
                              state = State::Attacking;
                              attack();
                        } else {
                              owner->moveToPosition(
                                          target->getPosition());
                        }
                  } else {
                        owner->moveToPosition(initialPosition);
                        state = State::FallBack;
                        target = nullptr;
                        break;
                  }
                  if (owner->getDistance(initialPosition) > driftRange) {
                        owner->moveToPosition(initialPosition);
                        state = State::FallBack;
                  }
                  break;
            case State::FallBack:
                  if (!owner->isMoving()) {
                        state = State::Scanning;
                        owner->setInCombat(false);
                  }
                  break;
            }
}
```

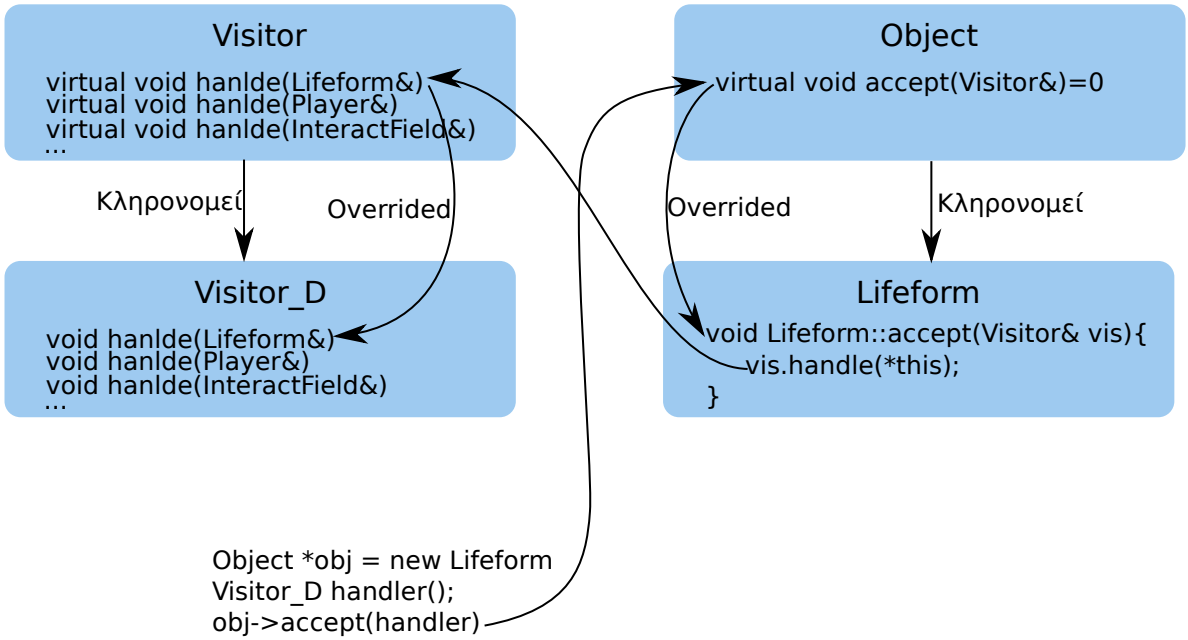
Κατά την κατασκευή του αντικειμένου, η κατάσταση που αποδίδεται είναι η **Scanning**. Σε αυτή την κατάσταση το Lifeform πρέπει να ελέγξει τριγύρω του για πιθανό στόχο μέσω της **scan()**.

Πίνακας 5.6.7: Η συνάρτηση AggressiveBehaviour::scan()

```
void AggressiveBehaviour::scan() {
class ObjHandler: public Visitor {
public:

     void handle(Lifeform& lf) {
           if (&lf != owner->owner && lf.isAlive()) {
                 if (
                 owner->owner->getFaction().isHostile(lf.getFaction())) {
                       if (owner->owner->getDistance(lf) <=
                                                      owner->agroRange) {
                             valid = true;
                       }
                 }
           }
     }
     void handle(Player& lf) {
           if (&lf != owner->owner && lf.isAlive()) {
                 if (
                 owner->owner->getFaction().isHostile(lf.getFaction())) {
                       if (owner->owner->getDistance(lf) <=
                                                      owner->agroRange) {
                             valid = true;
                       }
                 }
           }
     }

     bool isValid() const {
           return valid;
     }

     ObjHandler(AggressiveBehaviour* owner) :
                 owner(owner), valid(false) {

     }
private:
     AggressiveBehaviour *owner;
     bool valid;
} handler(this);

     if (owner->getActiveAbilities().size() > 0) {
           auto& visibles =

     WorldManager::getInstance().getCurrentMap().getVisibleObjects();
           for (auto obj : visibles) {
                 obj->accept(handler);
                 if (handler.isValid()) {
                       target = static_cast<Lifeform*>(obj);
                       state = State::Attacking;
                       owner->setInCombat(true);
                       owner->setTarget(target);
```

```
                        break;
                    }
                }
            }
        }
```

Μέσα στην συνάρτηση έχουμε μια εμφωλευμένη κλάση, την **ObjectHandler** και ένα αντικείμενο από αυτή, το **handler**. Ο λόγος που την έχουμε εμφωλευμένη εδώ είναι γιατί μόνο σε αυτή την συνάρτηση χρειάζεται και πουθενά αλλού στην μηχανή. Αυτή η κλάση μας χρησιμεύει για να κάνουμε Double Dispatching μέσω του Visitor Pattern που χρησιμοποιεί. Στην ουσία θα εκτελέσει διαφορετικό κώδικα για κάθε είδους αντικειμένου που έχουν κοινή κλάση-βάση. Επειδή μετά θα ζητήσουμε από τον χάρτη τα αντικείμενα που είναι εμφανή στην οθόνη, αυτός θα μας φέρει όλα τα αντικείμενα, είτε είναι Lifeform είτε άψυχα Objects. Για αυτό οι Pointers που θα έχουμε θα είναι τύπου **Object** και αυτό δεν μας μας λέει αν το αντικείμενο είναι Lifeform ή κάτι άλλο. Εμείς όμως θέλουμε να ελέγξουμε μόνο τα αντικείμενα που είναι Lifeform ή Player, για αυτό και χρησιμοποιούμε αυτό το Pattern που χρησιμεύει εδώ. Η κλάση βάση **Visitor** έχει κενές υλοποιήσεις της **handle()** με ορίσματα τα πιθανά εξειδικευμένα αντικείμενα της Object. Κάθε κλάση παιδί της Object έχει μια υλοποίηση της **accept()** (είναι virtual) που δέχεται σαν όρισμα ένα **Visitor**. Μέσα σε αυτές τις υλοποιήσεις η **accept()** καλεί την **handle()** του Visitor με όρισμα τον εαυτό του, δηλαδή το **this**. Επειδή εντός του κώδικα της υλοποίησης το **this** ξέρουμε τι τύπος είναι, τότε καλείται η εξειδικευμένη **handle()** της Visitor. Δηλαδή η υλοποίηση της **accept()** στην κλάση Lifeform θα καλέσει την **handle()** που έχει όρισμα ένα Lifeform κοκ. Έτσι όταν θα καλέσουμε την **accept()** μέσω της **Object**, καθώς αυτή η συνάρτηση είναι pure virtual, θα καλεστεί η υλοποίηση που έχει το παιδί και αυτό με την σειρά του θα καλέσει την εκδοχή της **handle()** που του του αντιστοιχεί. Εάν καλέσουμε την **accept()** με όρισμα ένα αντικείμενο από κλάση που κληρονόμησε την **Visitor**, τότε κατά την κλήση της **handle()** θα καλεστεί η **handle()** της κλάσης παιδί, όπως και στην περίπτωση της κλήσης της **accept()** μέσω του Object. Αυτή η τεχνική λέγεται Double Dispatch. Με αυτή την τεχνική μπορούμε να κάνουμε ένα αντικείμενο να εκτελέσει κάποιες λειτουργίες εσωτερικά που κανονικά δεν γίνεται γιατί δεν γράφτηκε έτσι. Εδώ το χρησιμοποιούμε όπως είπαμε για να διακρίνουμε τι είδος είναι το αντικείμενο που έχουμε μέσω της υπερ-κλάσης.



Εικόνα 5.6.1: Αναπαράσταση του Visitor Pattern

Αρχικά ελέγχουμε αν έχουμε κάποια ικανότητα για να χρησιμοποιήσουμε. Αν δεν έχουμε ικανότητες τότε δεν μπορούμε να επιτεθούμε και θα ήταν ανούσιο να συνεχίσουμε. Σε επόμενη φάση παίρνουμε από τον χάρτη όλα τα αντικείμενα που είναι στην οθόνη. Μέσω του **handler**, ελέγχουμε το Lifeform αν είναι εχθρικό και είναι εντός εμβέλειας του **agroRange**. Εφόσον τηρούνται αυτά τότε θέτεται το flag **valid** σε true. Πίσω στην επανάληψη, ελέγχουμε αυτό το flag και εφόσον είναι true, αυτό σημαίνει ότι βρήκαμε έναν στόχο να επιτεθούμε. Έτσι τον παίρνουμε, θέτουμε την κατάσταση σε **Attacking** και θέτουμε τον ιδιοκτήτη σε κατάσταση **Combat**.

Στην **update()** στο επόμενο frame, θα μπούμε στην Attacking και θα ελέγξουμε αν ο στόχος είναι ζωντανός. Αν είναι τότε ελέγχουμε αν έχουμε επιλέξει ποια ικανότητα θα χρησιμοποιήσουμε για την επίθεση. Αν δεν έχουμε βρει ικανότητα, τότε διαλέγουμε μία μέσω της **getAbility()**.

Πίνακας 5.6.8: Η συνάρτηση AggressiveBehaviour::getAbility()

```
void AggressiveBehaviour::getAbility() {
     auto& abilities = owner->getActiveAbilities();

     for (auto abi : abilities) {
           if (abi.second->canBeInvoked()) {
                 curAbility = abi.second;
                 return;
           }
     }
     if (abilities.size() > 0) {
```

```
            for (auto abi : abilities) {
                  if (abi.second->isThereEnoughMana()) {
                        curAbility = abi.second;
                        return;
                  }
            }
      } else {
            curAbility = nullptr;
      }
}
```

Εδώ παίρνουμε την λίστα με όλες τις ActiveAbilities του χρήστη. Για κάθε μία αυτό αυτές ελέγχουμε αν μπορούμε να την χρησιμοποιήσουμε. Αν μπορούμε, τότε το αποδίδουμε στο μέλος **curAbility** και επιστρέφουμε. Αν δεν βρεθεί ούτε μία, τότε ελέγχουμε το μέγεθος της λίστας για να δούμε ότι έχουμε έστω και ένα ActiveAbility. Το ότι πριν δεν βρήκαμε κάποιο διαθέσιμο αυτό σημαίνει ότι δεν έχουμε αρκετό Mana ή όλα βρίσκονται σε κατάσταση **Cooldown**. Για αυτό και εδώ θα ψάξουμε να βρούμε κάποιο που να μπορούμε να χρησιμοποιήσουμε όταν τελειώσει το **Cooldown** του και έτσι ψάχνουμε το πρώτο που έχουμε αρκετό Mana για να το χρησιμοποιήσουμε. Αν παρόλο αυτά πάλι δεν βρούμε κάτι, τότε μάλλον θα δεν θα χρησιμοποιήσουμε ActiveAbility.

Πίσω στην **update()** αφού έχουμε θεωρήσει ότι πήραμε κάποιο Ability, τότε καλούμε την **attack()** προκειμένου να επιτεθούμε.

Πίνακας 5.6.9: Η συνάρτηση AggressiveBehaviour::getAbility()

```
void AggressiveBehaviour::attack() {
      if (target->isAlive()) {
            if (curAbility != nullptr) {
                  switch (curAbility->invoke()) {
                  case Ability::Result::OutOfRange:
                        state = State::Moving;
                        initialPosition = owner->getPosition();
                        break;
                  case Ability::Result::Success:
                        state = State::Casting;
                        break;
                  default:
                        state = State::Moving;
                        initialPosition = owner->getPosition();
                        getAbility();
                        break;
                  }
            }
      } else {
            state = State::FallBack;
            owner->setInCombat(false);
      }
}
```

Ελέγχουμε αν ο στόχος είναι ζωντανός. Για δεύτερη φορά ελέγχουμε αν έχουμε το Ability, καθώς μπορεί να μην έχουμε κανένα διαθέσιμο όπως είπαμε προηγουμένως. Εφόσον υπάρχει, τότε το καλούμε και ελέγχουμε το αποτέλεσμα. Αν δεν είμαστε εντός εμβέλειας, τότε χρειάζεται να κινηθούμε πιο κοντά στον στόχο και έτσι θέτουμε την κατάσταση σε **Moving**. Αν είχε επιτυχία, τότε μπαίνουμε σε κατάσταση **Casting**. Σε οποιαδήποτε άλλη κατάσταση, θα κινηθούμε εξίσου προς τον στόχο και θα ψάξουμε νέα ικανότητα. Αν ο στόχος έχει πεθάνει, τότε μπαίνουμε σε κατάσταση **FallBack** για να επιστρέψουμε πίσω και βγαίνουμε από κατάσταση Combat.

Πίσω στην **update()**, έχουμε αρκετές περιπτώσεις που μπορεί να βρισκόμαστε. Αν είμαστε σε **Casting** τότε περιμένουμε μέχρι να τελειώσει και ψάχνουμε νέα ικανότητα για νέα επίθεση. Αν χρειάζεται να κινηθούμε, τότε ελέγχουμε ξανά αν ο στόχος είναι ζωντανός. Ο λόγος που ελέγχουμε συνέχεια αυτό είναι γιατί ο στόχος μπορεί να πέθανε από τρίτους λόγους. Εφόσον είναι ζωντανός, ελέγχουμε αν η απόσταση μας με τον στόχο είναι μικρότερη της εμβέλειας της ικανότητας. Αν είναι, τότε σημαίνει ότι μπορούμε να την χρησιμοποιήσουμε και έτσι μπαίνουμε πάλι σε κατάσταση **Attacking** καθώς και καλούμε την **attack()** προκειμένου να χρησιμοποιήσουμε την ικανότητα αμέσως. Αν ο στόχος τελικά δεν είναι ζωντανός, τότε χρειάζεται να εγκαταλείψουμε την επίθεση και να γυρίσουμε πίσω στην θέση που είχαμε πριν αρχίσουμε την επίθεση. Δίνουμε την εντολή στον ιδιοκτήτη, θέτουμε την κατάσταση σε **Fallback** και αφαιρούμε τον στόχο. Σε κάθε περίπτωση, πρέπει να ελέγξουμε αν έχουμε απομακρυνθεί πολύ από την αρχική θέση. Αν έγινε αυτό τότε εγκαταλείπουμε και επιστρέφουμε πίσω. Αν είμαστε σε κατάσταση **Fallback** τότε δεν κάνουμε τίποτα μέχρι ο ιδιοκτήτης να επιστρέψει στην αρχική θέση και μόλις φτάσει, επαναφέρουμε την κατάσταση σε **Scanning** προκειμένου να αναζητήσει καινούργιο στόχο.

c) Ακινησία

Τώρα θα δούμε τις συμπεριφορές κινήσεων. Αυτές οι συμπεριφορές λειτουργούν όταν το Lifeform δεν έχει κάτι άλλο να κάνει, δηλαδή το GeneralBehaviour είναι ανενεργό. Αυτό γίνεται μέσω του flag **combat** του Lifeform. Όταν αυτό είναι true, τότε σημαίνει ότι αναλαμβάνει το General και όταν είναι false το MovingBehaviour.

Η πιο απλή μορφή κίνησης είναι η ακινησία. Σε αυτή την περίπτωση το Lifeform είναι ακίνητο όσο επιδρά το MovingBehaviour. Η δομή του είναι η **IdleMovement** και λειτουργικά δεν κάνει τίποτα, απλά υπάρχει για λόγους σαφήνειας. Συνήθως χρησι-

μοποιείται για NPCs αλλά καλύτερα είναι να αποφεύγεται τελείως καθώς μόνο επιπρόσθετο βάρος θα είναι στα αντικείμενα Lifefrorm καθώς χωρίς καθόλου AI κάνει ακριβώς το ίδιο πράγμα.

d) Περιφορά με τυχαία πράξη

Η πιο “ζωντανή” μορφή κίνησης είναι η περιφορά. Σε αυτή, το Lifeform κινείται εντός μιας περιοχής σε τυχαίες (εντός ορίων) χρονικές στιγμές. Κατά την περίοδο που το Lifeform περιμένει μέχρι κάνει την επόμενη κίνηση, μπορεί κάνει κάποιο Action σε επίσης τυχαία στιγμή. Αυτό το είδος κίνησης ορίζεται από την κλάση **WanderingMovementClass**.

Πίνακας 5.6.10: Η κλάση WanderingMovmentClass

```
class WanderingMovementClass: public BehaviourClass {
public:
     Behaviour* getNewBehaviour(Lifeform& owner) const;
     WanderingMovementClass(Float minWaitTime, Float maxWaitTime,
                  Float maxDistance, const std::string& idleAction = "");
     ~WanderingMovementClass();
private:
     std::string idleAction;
     Float minWaitTime;
     Float maxWaitTime;
     Float maxDistance;
};
```

Το μέλος **idleAction** περιέχει το όνομα του Action που θα κάνει το Lifeform κατά την αναμονή. Μπορεί να μην οριστεί αν δεν θέλουμε να κάνει κάτι. Το **minWaitTime** και **maxWaitTime** ορίζουν τα όρια μεταξύ θα επιλέγεται ο χρόνος που θα περιμένει το Lifeform κατά την αναμονή. Το **maxDistance** ορίζει την μέγιστη απόσταση που θα μπορεί να απομακρυνθεί το Lifeform από το σημείο που δημιουργήθηκε. Η **getNewBehaviour()** κατασκευάζει αντικείμενα **WanderingMovement**.

Πίνακας 5.6.11: Η κλάση WanderingMovement

```
class WanderingMovement: public Behaviour {
public:

     const std::string& getIdleAction() const;
     void setIdleAction(const std::string& idleAction);
     void update(Float elapsedTime);

     void onDeath() {
     }
```

```
    WanderingMovement(const std::string& action, Lifeform &owner,
                Float minWaitTime = 1.0f, Float maxWaitTime = 4.0f,
                Float maxDistance = 500);
    ~WanderingMovement();
private:
    std::string idleAction;
    std::default_random_engine generator;
    std::uniform_real_distribution<Float> timeRandomizer;
    std::uniform_real_distribution<Float> actionRandomizer;
    std::uniform_int_distribution<int> xBounds;
    std::uniform_int_distribution<int> yBounds;

    Float waitTime;
    Float actionWaitTime;
};
```

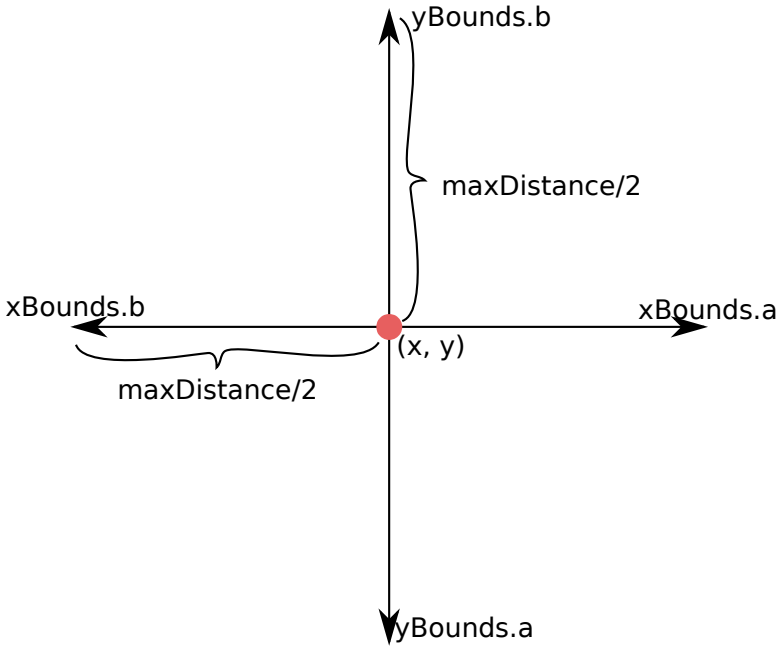
Έχουμε ένα αντίγραφο του ονόματος του Action που θα κάνει το Lifeform καθώς θα πρέπει να μπορεί και να αλλάζει κατά την διάρκεια. Έχουμε επίσης τις μηχανές τυχαίων αριθμών για τον χρόνο αναμονής κίνησης **(timeRandomizer)**, χρόνο μέχρι εκτέλεση κίνησης **(actionRandomizer)** και ορίων περιοχής κίνησης **(xBounds, yBounds)**.

Η λειτουργία γίνεται στην **update()** αλλά θα πρέπει να δούμε πως γίνεται η αρχικοποίηση πρώτα κατά την κατασκευή του αντικειμένου.

Πίνακας 5.6.12: Η συνάρτηση WanderingMovement::WanderingMovement(...)

```
WanderingMovement::WanderingMovement(const std::string& action, Lifeform
&owner, Float minWaitTime, Float maxWaitTime, Float maxDistance) {
    if (minWaitTime < maxWaitTime) {
        timeRandomizer =
std::uniform_real_distribution<Float>(minWaitTime, maxWaitTime);
        actionRandomizer =
    std::uniform_real_distribution<Float>(minWaitTime, maxWaitTime);
    } else {
        timeRandomizer =
                    std::uniform_real_distribution<Float>(1.0f, 4.0f);
        actionRandomizer =
                    std::uniform_real_distribution<Float>(1.0f, 4.0f);
    }
    generator.seed(reinterpret_cast<intptr_t>(this));
    waitTime = timeRandomizer(generator);
    actionWaitTime = actionRandomizer(generator);
    const Vector2& position = owner.getPosition();
    xBounds = std::uniform_int_distribution<int>(
                position.getX() - maxDistance / 2,
                position.getX() + maxDistance / 2);
    yBounds = std::uniform_int_distribution<int>(
                position.getY() - maxDistance / 2,
                position.getY() + maxDistance / 2);

}
```

Αρχικά ελέγχουμε αν οι χρόνοι που μας δόθηκαν είναι έγκυροι. Δηλαδή δεν μπορεί ο ελάχιστος χρόνος να είναι μεγαλύτερος του μέγιστου. Εφόσον είναι έγκυροι τότε συνεχίζουμε να τους θέσουμε όρια στις μηχανές τυχαίων των αναμονών. Αν δεν είναι έγκυρα, τότε θέτουμε δικούς μας χρόνους. Σε επόμενη φάση βάζουμε σπόρο στον **generator** με το τέχνασμα τον δείκτη **this**. Αρχικοποιούμε τους χρόνους από τις μηχανές τυχαίων. Στην συνέχεια πρέπει να θέσουμε τα όρια της περιοχής της κίνησης. Αυτή η περιοχή είναι ένα ορθογώνιο στο κέντρο που βρίσκεται αυτή την στιγμή το Lifeform. Για αυτό θέτουμε τα όρια στον άξονα του x και y. Αυτές οι μηχανές θα μας δίνουν από εδώ και πέρα συντεταγμένες εντός της περιοχής αυτής.



Εικόνα 5.6.2: Αναπαράσταση της περιοχής κίνησης

Ας πάμε να δούμε τώρα την **update()** που κάνει την λειτουργική δουλειά.

Πίνακας 5.6.13: Η συνάρτηση WanderMovement::update()

```
void WanderingMovement::update(Float elapsedTime) {
     if (owner->isAlive() && !owner->isMoving() && !owner->isInCombat())
     {
           waitTime -= elapsedTime;
           if (owner->isAnimationFinished()) {
                 if (waitTime <= 0) {
                       waitTime = timeRandomizer(generator);
                       owner->moveToPosition(xBounds(generator),
                                             yBounds(generator));
```

```
                } else if (idleAction.size() > 0) {
                      actionWaitTime -= elapsedTime;
                      if (actionWaitTime <= 0) {
                            actionWaitTime =
                                        actionRandomizer(generator);
                            owner->setActionAnimation(idleAction);
                      } else {
                            owner->stopMoving();
                      }
                }
          }
     }
}
```

Ελέγχουμε αν ο ιδιοκτήτης είναι ζωντανός, δεν κινείται και δεν βρίσκεται σε μάχη (δηλαδή το GeneralBehaviour δεν το έχει αναλάβει). Εφόσον ισχύουν αυτά, τότε μπορούμε να συνεχίσουμε. Μειώνουμε τον χρόνο που θα πρέπει να περιμένουμε μέχρι να κάνουμε την επόμενη τυχαία κίνηση **(waitTime)**. Σε επόμενη φάση ελέγχουμε αν το Animation που έχει αυτή την στιγμή το Lifeform έχει τελειώσει. Στην ουσία εδώ ελέγχουμε αν το Action που του είπαμε να κάνει τελείωσε. Αν το Animation τελείωσε, τότε ελέγχουμε αν πέρασε ο χρόνος που πρέπει να περιμένουμε για κίνηση. Αν πρέπει να κάνουμε κίνηση, τότε παίρνουμε νέο χρόνο για επόμενη κίνηση και κινούμαστε τυχαία στον χώρο που ορίσαμε μέσω των μηχανών τυχαίων συντεταγμένων **xBounds, yBounds**. Αν τελικά ο χρόνος δεν τελείωσε, τότε ελέγχουμε αν μας έχει οριστεί **idleAction** για να κάνουμε όσο περιμένουμε. Αν έχει οριστεί, τότε μειώνουμε τον χρόνο αναμονής για Action και ελέγχουμε αν πέρασε ο χρόνος. Αν πέρασε, τότε όπως και με την κίνηση παίρνουμε νέο χρόνο για Action και κάνουμε το Lifeform να εκτελέσει το Action. Αν δεν πέρασε ο χρόνος τότε σταματάμε να κινούμαστε αν κινούμασταν.

6 ΠΡΟΒΛΗΜΑΤΑ ΠΟΥ ΒΡΕΘΗΚΑΝ

Κατά την ανάπτυξη της μηχανής, βρέθηκα μπροστά σε κάποια προβλήματα. Κάποια από αυτά ήταν προβλήματα που αφορούσαν τον κώδικα και κάποια ήταν σχεδιαστικά

Ένα μεγάλο πρόβλημα που αντιμετώπισα και με πήγε πίσω στο χρόνο, ήταν αυτό του Heap Corruption. Η ιστορία έχει ως εξής εδώ. Όσο η μηχανή αναπτύσσονταν και λειτουργούσε, όλα ήταν καλά. Όμως μετά από καιρό, η μηχανή τερμάτιζε ακαριαία με λόγο κάποιου προβλήματος Memory Allocation. Ψάχνοντας το πρόβλημα, δεν μπορούσα να εντοπίσω πουθενά την προέλευσή του καθώς κάθε φορά που γινόταν, το πρόγραμμα τερμάτιζε σε τυχαίο σημείο του. Το Debuging δεν βοήθησε καθόλου. Αναζήτησα τον πιο πρόσφατο κώδικα να δω που έχω κάνει λάθος, αλλά χωρίς αποτέλεσμα. Ήταν τόσο τυχαίο και χωρίς προέλευση (έτσι φαινόταν), που κόντεψα να πιστέψω στην μαγεία! Ύστερα από ένα μήνα εντατικού ψαξίματος, το πρόβλημα βρέθηκε χάρις στο εργαλείο Valgrind το οποίο εντόπισε ένα Invalid Write.

Το πρόβλημα υπήρχε σε κώδικα που είχα γράψει στην αρχή-αρχή της ανάπτυξης, για την ακρίβεια στην κλάση File. Εκεί όπως έχουμε δει, στο τέλος των δεδομένων του αρχείου βάζαμε ένα 0. Στην έκδοση που προκαλούσε το πρόβλημα, το 0 το έβαζα σε θέση που ήταν εκτός του ορίου του πίνακα. Για την ακρίβεια πήγαινε στο όριο +1. Επειδή τα δεδομένα αυτά μπήκαν σε μνήμη που μας προήλθε από malloc(), τότε η εγγραφή αυτή πήγαινε και έγγραφε σε Header που άνηκε σε άλλο Block μνήμης της malloc(). Αυτό είχε ως αποτέλεσμα το Header αυτό να γίνεται corrupt και όταν πάει να γίνει free(), να προκαλεί κατάρρευση.

Θα μπορούσε αυτός ο κώδικας να προκαλέσει προβλήματα από την αρχή που γράφτηκε αλλά δεν έγινε παρά πολύ μετά. Αυτό γιατί όταν το πρόγραμμα είναι μικρό, δεν είναι σίγουρο ότι μετά την μνήμη που μας δόθηκε θα υπάρχει Header άλλου Block. Όμως καθώς το πρόγραμμα μεγάλωνε και οι malloc() αυξανόταν σε αριθμό, η πιθανότητα να βρεθεί εκεί Header αυξανόταν, ώσπου γινόταν σίγουρο και προκαλούσε κατάρρευση σε κάθε εκτέλεση. Το πρόβλημα λύθηκε διορθώνοντας τον κώδικα και γράφοντας το 0 σε σωστή θέση εντός του Block.

Ένα άλλο μεγάλο πρόβλημα που είχα, ήταν ο εντοπισμός των συγκρούσεων. Αρχικά ότι δοκίμαζα, δεν λειτουργούσε όπως θα ήθελα. Στην αρχή σκέφτηκα τα ορθογώνια να ελέγχονται και όταν υπάρχει σύγκρουση, να μην γίνει κίνηση. Όμως αυτό σημαίνει ότι αν συγκρουστεί κάτι, τότε του απέτρεπε κάθε κίνηση, ακόμα και να απο-

μακρυνθεί. Κάτι άλλο που σκέφτηκα ήταν με διακριτές θέσεις. Οι διακριτές θέσεις λειτουργούσαν, αλλά ήταν εκτός στόχου της μηχανής, για αυτό και ξαναγύρισα στο πρώτο μήπως το διορθώσω. Το κατάφερα αλλά πάλι η κίνηση ήταν λίγο περίεργη, καθώς όταν ένα Lifeform ακουμπούσε κάτι, τότε σταματούσε σαν να είχε “κόλλα” κάτω. Αυτό που ήθελα ήταν να ολισθαίνει μέχρι να φτάσει στο σημείο σε κάποιο άξονα. Για αυτό μου ήρθε η ιδέα των “μελλοντικών” συγκρούσεων όπου και υλοποίησα στο Lifeform. Όπως είδαμε, απλά “εξομοιώνουμε” την κίνηση σε κάθε άξονα και ελέγχουμε αν μπορούμε να κινηθούμε σε καθένα ξεχωριστά.

Ένα άλλο πρόβλημα ήταν η πολυπλοκότητα του πυρήνα. Αρχικά, σε κάθε Frame καλούνταν η update() για κάθε αντικείμενο στον χάρτη. Αυτό όμως προκαλούσε καθυστερήσεις αν τα αντικείμενα ήταν πολλά σε αριθμό. Για αυτό και περιόρισα τον αριθμό των αντικειμένων που θα γίνουν Update σε αυτά που φαίνονται στην οθόνη. Αυτό δεν έφτανε για την περίπτωση που τα Lifeforms ήταν πολλά στην οθόνη και έπρεπε να ελέγξουν για συγκρούσεις. Όπως είχαμε δει σε προηγούμενο κεφάλαιο, αν ελέγχαμε κάθε Lifeform εναντίον όλων των άλλων, θα είχαμε μεγάλη πολυπλοκότητα και το παιχνίδι θα καθυστερούσε πολύ αισθητά. Για αυτό και χρειάστηκε να εισάγω την τεχνική του Quadtree όπου περιόρισα πολύ τους ελέγχους.

Ένα τελευταίο πρόβλημα που με ταλαιπώρησε λίγο ήταν αυτό των Attributes και των Modifiers. Όπως είχαμε πει, τα Modifiers πρέπει να μπαινοβγαίνουν στα Attributes και να μπορεί να διατηρηθεί η προηγούμενη τιμή χωρίς να επηρεάσει τα άλλα. Αρχικά τα Modifiers ήταν απλοί αριθμοί, όμως έτσι δεν λειτουργούσε, και κατέληξα να έχει κάθε Attribute μια στοίβα από αντικείμενα Modifiers που θα πρέπει να υπολογίζουν την τελική τιμή.

7 ΣΥΓΚΡΙΣΗ ΜΕ ΑΝΤΙΣΤΟΙΧΗ ΜΗΧΑΝΗ

Υπάρχουν πολλές δισδιάστατες μηχανές RPG αλλά μέχρι στιγμής μόνο μηχανή Flare είναι η πιο κοντινή όσο αφορά την λειτουργία. Η Flare είναι και αυτή μια μηχανή για παραγωγή ARPG παιχνιδιών. Την έφτιαξε ο Clint Bellanger και την έχει δημοσιεύσει υπό την άδεια GPL.

Σε αντίθεση με την μηχανή μας, στην Flare τα πάντα ορίζονται από παθητικά αρχεία κειμένου. Δεν ορίζονται πουθενά Scripts, παρά μόνο περιγράφονται μέσω των αρχείων αυτών τα πάντα στο παιχνίδι. Το πλεονέκτημα εδώ της μηχανής μας είναι ότι μέσω των Scripts, το παιχνίδι μπορεί να γίνει πιο διαδραστικό. Από την άλλη, στην Flare αυτό δίνει ένα πλεονέκτημα ότι όποιος θέλει να φτιάξει παιχνίδι, δεν είναι απαραίτητο να ξέρει βασικό προγραμματισμό που λίγο πολύ χρειάζεται σε εμάς.

Ένα πλεονέκτημα της Flare και μειονέκτημα σε εμάς είναι ότι η Flare έχει δικό της σύστημα GUI. Αυτό της παρέχει απόλυτο έλεγχο πάνω του και δεν χρειάζεται να βασίζεται σε τρίτη βιβλιοθήκη που βασιζόμαστε εμείς.

Ένα πλεονέκτημα της δικιά μας μηχανής είναι ότι μέσω των Scripts δίνει την πρόσβαση σε λειτουργίες χαμηλού επιπέδου για πιο εξειδικευμένα χαρακτηριστικά. Στην Flare πρέπει να γίνει επεξεργασία του κώδικα πυρήνα για να αλλάξει κάτι.

Πρέπει να τονιστεί ότι η Flare κατασκευάστηκε αρχικά σαν παιχνίδι, για αυτό και πολλά χαρακτηριστικά είναι Hardcoded σε σχέση με την δικιά μας που φτιάχτηκε από την αρχή σαν Engine.

8 ΠΑΡΑΤΗΡΗΣΕΙΣ

Εδώ ήρθε το τέλος της ανάλυσης της μηχανής. Πριν κλείσουμε, θα πρέπει να γίνουν κάποιες αναφορές σε κάποιες παρατηρήσεις που έγιναν κατά την συγγραφή αυτού του έγγραφου.

Η μηχανή αναπτύχθηκε σε γρήγορους ρυθμούς προκειμένου να υπάρχει ένα αποτέλεσμα όσο πιο σύντομα γίνεται. Υπήρχε μια συνέπεια όσο αφορά την συγγραφή του κώδικα αλλά πολλές φορές έγιναν κάποια λάθη λόγο ταχύτητας που εντοπίστηκαν τώρα. Συχνό φαινόμενο ήταν να γραφτεί κώδικας που δεν χρησίμευε πουθενά ή υπήρχε μια διακλάδωση if που έλεγχε μια μεταβλητή που θα ήταν πάντα true.

Οι πιο πολλές περιπτώσεις ήταν να γραφτεί κώδικας που ναι μεν έφερνε το αποτέλεσμα που ήθελα αλλά όχι με τον βέλτιστο τρόπο. Αυτές τις περιπτώσεις τις εντόπιζα και άλλαζα τον κώδικα με μια πολύ πιο βέλτιστη έκδοση. Πχ

Πίνακας 8.1: Παράδειγμα αστοχίας κώδικα

```
if (std::sqrt(dx*dx) < (speed + 0.1f)) {
     // We should stop moving
     moving = false;
}
```

Ο παραπάνω κώδικας τον είχα γράψει σε ένα μέρος ελέγχου συγκρούσεων σε επίπεδο Lifeform. Με την std::sqrt(dx*dx) αυτό που ήθελα να κάνω είναι να πάρω το dx σε θετικό, ανεξάρτητα το τι πρόσημο είχε. Αυτό θα ήταν αποτελεσματικό, αλλά θα ήταν πολύ αργό για τον σκοπό του και μπορεί να αντικατασταθεί με την std::abs(dx) που θα μας φέρει την απόλυτη τιμή και θα έχει το ίδιο αποτέλεσμα.

Το παραπάνω παράδειγμα ήταν η πιο ανόητη περίπτωση που βρέθηκε. Υπήρχαν και περιπτώσεις που ο κώδικας ήθελε αλλαγή ως προς την δομή του. Ένα τέτοιο παράδειγμα ήταν η φόρτωση του χάρτη. Αρχικά όλη η φόρτωση ήταν σε μια συνάρτηση. Αυτό έκανε τον κώδικα πολύ δυσανάγνωστο και αδόμητο και έτσι την έσπασα σε πολλές συναρτήσεις που η κάθε μία φορτώνει ένα κομμάτι του χάρτη.

Υπήρχαν και πιο πολύπλοκες περιπτώσεις βελτιστοποίησης που χρειαζόταν σκέψη για να γίνουν. Αυτή η περίπτωση ήταν της κλάσης **LuaTable**. Λόγο της φύσης της Lua, οι μεταβλητές της δεν είχαν ρητό τύπο. Για αυτό και η αρχική υλοποίηση της κλάσης είχε πολλαπλές επαναλήψεις το ίδιου κώδικα με την μόνη αλλαγή ένα string. Αυτή η επανάληψη του κώδικα την θεώρησα ανεπίτρεπτη και έτσι προχώρησα σε ρι-

ζική αλλαγή της κλάσης και της φύσης της. Για παράδειγμα στην πρώτη μορφή υπήρχε μια **getField()** για κάθε τύπο μεταβλητής της Lua. Αυτό το άλλαξα και από 5 συναρτήσεις με τον ίδιο κώδικα, έγινε μία με Template που δεν επαναλαμβάνει τον κώδικα.

Πέρα από τις παρατηρήσεις στον κώδικα, υπήρχαν και παρατηρήσεις που έχουν να κάνουν με την όλη σχεδίαση της μηχανής. Η ανάπτυξή της είχε ξεκινήσει με το να έχει κάποιες δεδομένες λειτουργίες. Για παράδειγμα τα συμβάντα του κόσμου αρχικά ήταν τα 5 που οριζόταν από την μηχανή. Αυτό αργότερα κατάλαβα ότι θα περιόριζε πολύ τους προγραμματιστές που θα ήθελαν να φτιάξουν ένα πιο πολύπλοκο παιχνίδι. Έτσι πρόσθεσα τα προσαρμοσμένα συμβάντα. Αυτό ήταν μια εύκολη περίπτωση αλλαγής. Υπήρχαν και περιπτώσεις που η αλλαγή δεν ήταν εύκολη. Για παράδειγμα στην περίπτωση των Abilities, έτσι όπως τα έφτιαξα ήταν περιοριστικά ως προς τον προγραμματιστή. Όταν το κατάλαβα αυτό ήταν αργά και δεν μπορούσα να τα αλλάξω γιατί ήταν ήδη σε προχωρημένο στάδιο και για την αλλαγή τους θα έπρεπε να κάνω αλλαγές και σε άλλα κομμάτια της μηχανής. Για αυτό και προχώρησα μόνο σε μικρές αλλαγές.

Με όλα τα παραπάνω κατέληξα σε ένα συμπέρασμα που θα έπρεπε να ξέρω όταν ξεκίνησα την ανάπτυξη της μηχανής: χρειάζεται λιγότερο Hardcoding και περισσότερες επιλογές Callback από πολλά συμβάντα για να μπορείς να έχεις πολλές ελευθερίες να αναπτύξεις κάτι. Η μηχανή θα έπρεπε να είχε διαθέσιμες Callback για τον προγραμματιστή για οτιδήποτε συνέβαινε εντός της και όχι για περιορισμένο αριθμό συμβάντων.

9 ΒΙΒΛΙΟΓΡΑΦΙΑ

1. **Η Γλώσσα Προγραμματισμού C** – Dennis Ritchie, Brian Kernighan
2. **C++ Reference** - http://en.cppreference.com/w/cpp
3. **Allegro 5.0 reference manual** - https://www.allegro.cc/manual/5/
4. **Lua 5.1 Reference Manual** - https://www.lua.org/manual/5.1/
5. **tolua++** - https://github.com/LuaDist/toluapp
6. **Crazy Eddie's GUI System Mk-2: Developer Documentation** - http://static.cegui.org.uk/docs/0.8.6/
7. **libxml++ - An XML Parser for C++** - https://developer.gnome.org/libxml++-tutorial/stable/
8. **Zlib 1.2.8 manual** - http://www.zlib.net/manual.html
9. **Tiled Documentation** - http://doc.mapeditor.org/
10. **Dark Function Editor** - http://darkfunction.com/editor/
11. **Use Quadtrees to Detect Likely Collisions in 2D Space** - http://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374
12. **Liberated Pixel Cup** - http://lpc.opengameart.org/
13. **2D Computer Graphics** - https://en.wikipedia.org/wiki/2D_computer_graphics
14. **Framebuffer** - https://en.wikipedia.org/wiki/Framebuffer
15. **Alpha Compositing** - https://en.wikipedia.org/wiki/Alpha_compositing
16. **Screen Tearing** - https://en.wikipedia.org/wiki/Screen_tearing
17. **Optimizing Software in C++** - Agner Fog http://www.agner.org/optimize/optimizing_cpp.pdf
18. **Vector Space** - https://en.wikipedia.org/wiki/Vector_space