

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ



Τμήμα Πληροφορικής

ΤΕΧΝΟΛΟΓΙΕΣ ΑΝΑΠΤΥΞΗΣ ΗΛΕΚΤΡΟΝΙΚΩΝ ΠΑΙΧΝΙΔΙΩΝ

ΑΝΑΠΤΥΞΗ 2D ΠΑΙΧΝΙΔΙΟΥ ΜΕ SFML

-

CREATING A 2D GAME WITH SFML

Όνομα: Αριστοτέλης Ματακιάς - Α.Μ: Π19100

Email Επικοινωνίας: mataktelis01@gmail.com

Επιβλέπων καθηγητής: Θεμιστοκλής Παναγιωτόπουλος

Μάρτιος 2023



Περιεχόμενα

1 Εισαγωγή	2
2 Περιγραφή του προβλήματος	3
2.1 Κριτήρια της εκφώνησης	3
2.2 Παρατηρήσεις	4
3 Ανάλυση	5
4 Σχεδίαση	6
5 Υλοποίηση	7
5.1 Προ-επισκόπηση	7
5.2 Βασικοί ορισμοί	7
5.2.1 Header files και Implementation files	7
5.2.2 Κλίση ενός header file	9
5.2.3 Μεταγλώττιση κατά τμήματα - Separate Compilation	10
5.2.4 Εργαλείο Make	11
5.2.5 Εργαλείο CMake	12
6 Πηγαίος κώδικας	15
6.1 Βασικά αρχεία και διαγράμματα κλάσεων	17
6.2 Διαχείριση εισόδου από τον χρήστη - EventManager	17
6.2.1 Χρήση του Event της SFML	17
6.2.2 Εισαγωγή στην κλάση EventManager	19
6.3 Καταστάσεις του προγράμματος - BaseState και StateManager	21
6.4 Χάρτες	23
7 Εκτέλεση	26
7.1 Build	26
7.2 Gameplay	27
8 Συμπεράσματα και παρατηρήσεις	30
8.1 Κάλυψη κριτηρίων	30
8.2 Σημεία προς βελτίωση	31
8.3 Συμπεράσματα	32
9 Βιβλιογραφία	33
9.1 Πηγές	33
9.2 Assets	33

1 Εισαγωγή

Το παρόν αρχείο αποτελεί την τεκμηρίωση για την εργασία του μαθήματος **Τεχνολογίες Ανάπτυξης Ηλεκτρονικών Παιχνιδιών**. Στόχος της εργασίας είναι η ανάπτυξη ενός βιντεοπαιχνιδιού. Η γλώσσα προγραμματισμού που επιλέχθηκε είναι η **C++**. Περισσότερες λεπτομέρειες για την υλοποίηση θα παρουσιαστούν σε επόμενες ενότητες.

Ο λόγος για τον οποίο δεν χρησιμοποιήθηκαν τα εργαλεία που παρουσιάστηκαν στο μάθημα (**C#** και **Unity**) είναι ότι παρόλο που επιτρέπουν στον χρήστη να δημιουργήσει εύκολα και γρήγορα ένα παιχνίδι, ο χρήστης δεν γνωρίζει ακριβώς πώς το παιχνίδι λειτουργεί. Μηχανές όπως η **Unity** παρέχουν ένα έτοιμο περιβάλλον ανάπτυξης πάνω στο οποίο ο χρήστης θα αναπτύξει αυτό που θέλει. Το περιβάλλον αυτό διαχειρίζεται όλες τις λειτουργίες του προγράμματος και απαλλάσσει τον χρήστη από πολλές λεπτομέρειες υλοποίησης, οι οποίες για κάποιους θεωρούνται δύσκολες. Αντίθετα όταν ο ίδιος ο χρήστης δημιουργεί το περιβάλλον, κάτι που μπορεί να επιτευχθεί με γλώσσες χαμηλότερου επιπέδου όπως η **C** και η **C++**, μπορεί να το προσαρμόσει ακριβώς στις ανάγκες του. Σαφώς αυτό σημαίνει ότι απαιτείται περισσότερος χρόνος για την ανάπτυξη του παιχνιδιού, καθώς πρώτα πρέπει να φτιαχτεί το περιβάλλον, στο οποίο συνήθως αναφερόμαστε ως μπχανή (**engine**) του παιχνιδιού. Μπορεί η δημιουργία μίας μπχανής από το μηδέν να ακούγεται ως μία δύσκολη και επίπονη δουλεία αλλά στη πραγματικότητα είναι μία αρκετά δημιουργική διαδικασία για κάθε προγραμματιστή που ασχολείται είτε επαγγελματικά είτε ερασιτεχνικά με την ανάπτυξη βιντεοπαιχνιδιών.

Η εργασία εστιάζει κυρίως στην Υλοποίηση του παιχνιδιού, το οποίο είναι 2D. Επιπλέον οι βιβλιοθήκες και τα περιεχόμενα (**Assets**) του παιχνιδιού είναι όλα δωρεάν. Ο πηγαίος κώδικας του παιχνιδιού βρίσκεται στα παραδοτέα της εργασίας αλλά και στο **Github**. Είναι σημαντικό να αναφερθεί ότι ο κώδικας του παιχνιδιού βασίστηκε στο βιβλίο **SFML Game Development By Example** του οποίου ο κώδικας είναι διαθέσιμος και αποτελεί έναν πολύ καλό οδηγό για εισαγωγή στον προγραμματισμό βιντεοπαιχνιδιών. Η εργασία βασίστηκε στο κεφάλαιο 7 του βιβλίου στο οποίο ενώ δεν περιέχονται όλα σχεδιαστικά πρότυπα (**design patterns**) του βιβλίου, εξακολουθεί να είναι επαρκές για τη δημιουργία ενός ολοκληρωμένου 2D παιχνιδιού.



2 Περιγραφή του προβλήματος

2.1 Κριτήρια της εκφώνησης

Σκοπός της εργασίας είναι η δημιουργία ενός βιντεοπαιχνιδιού και δίνονται τα παρακάτω κριτήρια τα οποία θα πρέπει στην ιδανική περίπτωση να πληρούνται:

- **Αληθοφάνεια** Ο χώρος σας θα πρέπει να "πείθει" τον χρήστη, να μη δημιουργεί την εντύπωση πως παραβιάζει τους φυσικούς νόμους και, γενικά, να παρέχει τη ζητούμενη εμπειρία χωρίς να βασίζεται σε σχεδιαστικές ή αισθητικές ακρότητες.
- **Περιεχόμενο** Θα χρησιμοποιήσετε περιεχόμενο που θα θυμίζει τον φυσικό, "πραγματικό" κόσμο για το σύνολο του οποίου θα αναφέρετε πηγές (μέσα στον εικονικό κόσμο, χρησιμοποιώντας κατάλληλα μέσα απεικόνισης πληροφορίας).
- **Πληρότητα** Ο χώρος θα είναι ολοκληρωμένος σαν συστατικό εφαρμογής εικονικής πραγματικότητας: θα εμφανίζεται στο χρήστη ως ένας τρισδιάστατος κόσμος, θα έχει λειτουργικότητα σε διάφορα σημεία ανάλογα με το τι αναπαριστά και το τι εξυπηρετεί, θα εξελίσσεται με το πέρασμα του χρόνου χάρη σε κινούμενα στοιχεία και στοιχεία με λειτουργικότητα, θα έχει φωτισμό και άλλα διακοσμητικά στοιχεία, και γενικά θα παρέχει μία ολοκληρωμένη, πολυτροπική, δυναμική εμπειρία αναπαράστασης χωρίς αδικαιολόγητα κενά και σημεία ασυνέχειας ή ασυνέπειας.
- **Σχεδιασμός** Ο σχεδιασμός του χώρου σας θα χαρακτηρίζεται από ακρίβεια και λεπτομέρεια. Η αυξημένη σχεδιαστική/δομική πολυπλοκότητα δεν αποτελεί στοιχείο το οποίο θα αξιολογηθεί απαραίτητα θετικά.
- **Αισθητική** Από άποψη αισθητικής, ο χώρος σας θα πρέπει να είναι ευπαρουσίαστος και να μην χαρακτηρίζεται από αισθητικές ακρότητες. Επίσης, θα πρέπει να "προσκαλεί" τον χρήστη και να του κεντρίζει την προσοχή.
- **Πρωτοτυπία** Η δομή, ο σχεδιασμός, η αισθητική και η λειτουργικότητα του χώρου σας θα αντανακλούν την έκταση στην οποία διερευνήσατε τις σχεδιαστικές δυνατότητες της πλατφόρμας, τις δυνατότητες της οικείας γλώσσας, καθώς και διάφορα σχεδιαστικά πρότυπα και πρακτικές υλοποίησης.
- **Χρηστικότητα** Ο χώρος θα προορίζεται για χρήση από ανθρώπους και για την κάλυψη υπαρκτών αναγκών τους. (π.χ. δυνατότητα μετακίνησης και επίσκεψης διαφόρων σημείων του τρισδιάστατου κόσμου)
- **Κίνηση (animation)** Η κατασκευή σας θα περιέχει αναπαραστάσεις κινούμενων στοιχείων, ώστε να δημιουργεί στον χρήστη την εντύπωση ότι συμμετέχει σε ένα δυναμικό, εξελισσόμενο με το χρόνο, κόσμο. Τέτοια στοιχεία μπορεί να είναι, για παράδειγμα, "άψυχα" αντικείμενα, αντικείμενα που μετακινούνται σε προκαθορισμένες διαδρομές, ζώα/ρομπότ, κ.ά. Τονίζεται ότι ο όρος κίνηση χρησιμοποιείται εδώ όχι με την έννοια της μετακίνησης, αλλά της οποιασδήποτε μεταβολής της τιμής κάποιου χαρακτηριστικού ενός αντικειμένου. Έτσι, ως κινούμενο συστατικό νοείται και ένα αντικείμενο που αλλάζει χρώμα με την πάροδο του χρόνου (π.χ., ένα φανάρι ρύθμισης κυκλοφορίας, κ.ά.), ένα αντικείμενο που αλλάζει διαστάσεις, κ.ά.
- **Λειτουργικότητα (functionality)** Ανάλογα με το τι αναπαριστά, η κατασκευή σας θα περιέχει συστατικά με λειτουργικότητα, δηλαδή, συστατικά πάνω στα οποία θα μπορεί να επιδράσει ο χρήστης ή, γενικότερα, συστατικά τα οποία αντιδρούν με συγκεκριμένο τρόπο σε συγκεκριμένες ενέργειες του χρήστη. Παραδείγματα αντικειμένων



με λειτουργικότητα είναι μία πόρτα η οποία ανοίγει και κλείνει όταν ο χρήστης επιδρά σε ένα διακόπτη, ένας μηχανισμός που ξεκινά και σταματά όταν ο χρήστης επιδρά σε ένα μοχλό, ένα ρομπότ ή ένας συναγερμός που ενεργοποιείται όταν ο χρήστης πλησιάζει κάποιο σημείο του εικονικού κόσμου ή βρεθεί μέσα σε κάποια περιοχή του, ένα φαδιόφωνο του οποίου την ένταση ο χρήστης προσαρμόζει με κάποιο χειριστήριο, μία συσκευή τηλεόρασης την οποία ο χρήστης ενεργοποιεί και απενεργοποιεί, κ.ά.

- **Ανάπτυξη** Η λειτουργικότητα των διαφόρων αντικειμένων θα υλοποιηθεί στην οικεία γλώσσα (C#/Unity 3D). Ο κώδικας που θα συντάξετε θα είναι καλά σχεδιασμένος, θα ενσωματώνει βέλτιστες τεχνικές υλοποίησης, θα είναι ευπαρουσίαστος και, το σημαντικότερο, θα είναι συνολικά, αναλυτικά και κατανοητά σχολιασμένος.

2.2 Παρατηρήσεις

Το παιχνίδι που θα παρουσιαστεί θα είναι δισδιάστατο (2D) και όχι τρισδιάστατο (3D) όπως αναφέρεται στα κριτήρια, χωρίς να σημαίνει αυτό ότι τα κριτήρια αλλάζουν σημαντικά. Πιο συγκεκριμένα το παιχνίδι θα είναι ένα απλό 2D platformer συγκρίσιμο σε μεγάλο βαθμό με το κλασικό παιχνίδι **Super Mario Bros.** (Εικόνα 1).



Εικόνα 1: Super Mario Bros. (NES)

Ο παίκτης θα μετακινεί με τα τυπικά πλήκτρα **W**, **A**, **S**, **D** όπως χαρακτήρα σε ένα περιβάλλον φτιαγμένο από **tiles**. Στο περιβάλλον αυτό θα υπάρχουν "εχθροί" τους οποίους θα μπορεί ο παίκτης να προσπερνά ή να πολεμάει. Σκοπός του παίκτη είναι να φτάσει στο τέλος του χάρτη.



3 Ανάλυση

Στα πλαίσια της εργασίας είχαν οριστεί οι παρακάτω στόχοι:

- Χρήση δωρεάν και Open Source περιβάλλοντος για την ανάπτυξη του παιχνιδιού**
Η γλώσσα προγραμματισμού είναι η C++ και βασικό όρόλ στην ανάπτυξη του παιχνιδιού έπαιξε η βιβλιοθήκη SFML. Τα αρχικά σημαίνουν **Simple and Fast Multimedia Library** και είναι μια cross-platform βιβλιοθήκη που παρέχει απλά API για τη δημιουργία παιχνιδιών και εφαρμογών. Η SFML μπορεί να χρησιμοποιηθεί τόσο για την ανάπτυξη μεγάλης κλίμακας εμπορικών παιχνιδιών, όσο και για μικρότερες ομάδες και χομπίστες που είναι αρχάριοι με την C++.

- Χρήση από δωρεάν assets**

Τα assets είναι ένας ευρύς όρος που χρησιμοποιείται για να περιγράψει κάθε μεμονωμένο στοιχείο που υπάρχει σε ένα παιχνίδι. Αυτά μπορεί να περιλαμβάνουν έργα τέχνης, τρισδιάστατα μοντέλα, GUIs, μουσική, ακόμη και scripts για πράγματα όπως η φυσική.

Η δημιουργία game assets αποτελεί ένα έργο σχεδίασης. Ιδανικά, ο δημιουργός ενός παιχνιδιού φτιάχνει ο ίδιος τα assets για το παιχνίδι του, ή στη περίπτωση μίας μεγαλύτερης ομάδας που ασχολείται με την ανάπτυξη ενός παιχνιδιού, θα υπάρχουν άτομα που θα εξειδικεύονται αποκλειστικά σε τέτοια θέματα σχεδίασης. Τα assets είναι αυτά που φαίνονται άμεσα σε αυτόν που βλέπει το παιχνίδι και για πολλούς θεωρείται πώς είναι στοιχειώδης μέρος του ίδιου του παιχνιδιού. Υπάρχουν πολλές πλατφόρμες οι οποίες προσφέρουν assets και για να μην υπάρχουν θέματα αδειών θα χρησιμοποιηθούν αποκλειστικά και μόνο δωρεάν assets. Με αυτό τον τρόπο θα μπορεί ο πλήρης πηγαίος κώδικας του project να είναι ελεύθερα διαθέσιμος.

- Χρήση Git και Github**

Η χρήση source control στο development είναι πολύ σημαντική για κάθε είδους λογισμικού. Όλος ο πηγαίος κώδικας και τα assets βρίσκονται σε ένα Github repository το οποίο μάλιστα είναι public. Σε αυτό βρίσκονται όλα τα commits και μπορεί οποιοσδήποτε να πραγματοποιήσει αλλαγές, να προτείνει βελτιώσεις ή ακόμα να χρησιμοποιήσει τον κώδικα για την δημιουργία του δικού του παιχνιδιού. Ο σύνδεσμος του repository είναι ο παρακάτω:

<https://github.com/mataktelis11/Simple-SFML-2D-Game>

- Χρήση command line εργαλείων για το build του κώδικα και όχι μέσω κάποιου IDE/GUI**

Η εξάρτηση από κάποιο GUI based IDE μπορεί πολλές φορές να κάνει το development πιο περίπλοκο παρά εύκολο. Επίσης πολλές φορές κρύβονται σημαντικές πληροφορίες για το πώς ακριβώς γίνεται build το τελικό εκτελέσιμο. Χρησιμοποιώντας αποκλειστικά command line εργαλεία (όπως το CMake) και έναν απλό text editor (αντί για ένα IDE) μπορούμε να έχουμε πλήρη έλεγχο για όλο το project και επίσης γίνεται πιο εύκολο να διαμοιραστεί καθώς αποτελείται μόνο από αρχεία κώδικα και αρχεία ήχου/εικόνων (assets).

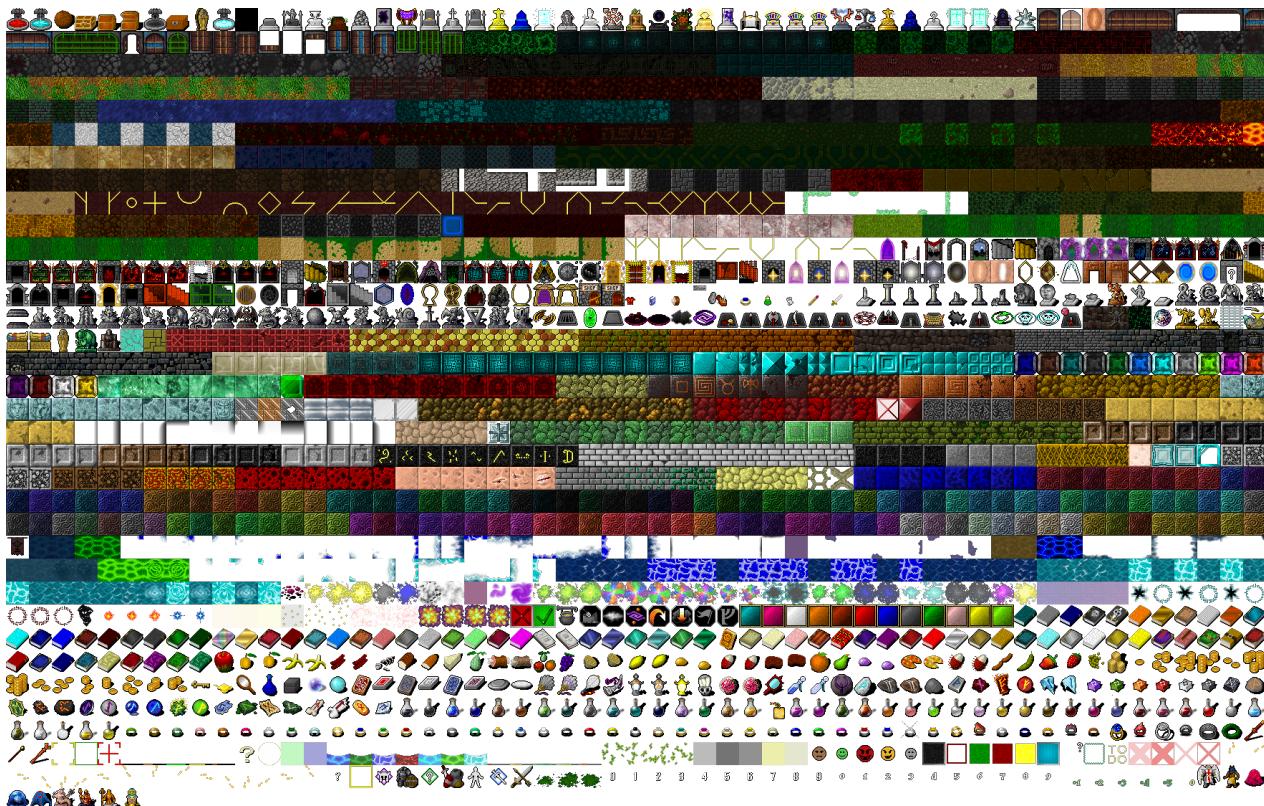
- Δημιουργία εκτελέσιμου αρχείου για πολλές πλατφόρμες (τουλάχιστον GNU/Linux και Windows)**

To development και το testing έγινε κυρίως σε GNU/Linux λειτουργικό σύστημα. Ωστόσο ο compiler και η SFML είναι cross-platform και επιτρέπουν την μεταγλώττιση σε Windows. Στα παραδοτέα της εργασίας υπάρχει εκτελέσιμο για GNU/Linux.



4 Σχεδίαση

Οι χάρτες του παιχνιδιού, όπως πολλά 2D παιχνίδια, θα σχεδιάζονται με Tiles τα οποία αντλούνται από ένα Tileset. Παρακάτω φαίνεται το Tileset που χρησιμοποιεί το παιχνίδι. Το γενικό theme του παιχνιδιού είναι μεσαιωνικό γι' αυτό επιλέχθηκε αυτό το Tileset.



Εικόνα 2: Tileset

Καθώς το παιχνίδι είναι 2D δεν χρησιμοποιεί μοντέλα, αλλά sprites για τους χαρακτήρες. Παρακάτω φαίνονται τα sprites των εχθρών και του παίκτη.



(a) Οι εχθροί στο παιχνίδι



(b) Ο παίκτης

Εικόνα 3



5 Υλοποίηση

5.1 Προ-επισκόπηση

Ο πηγαίος κώδικας του προγράμματος αποτελείται από πολλά ξεχωριστά αρχεία. Όπως έχει αναφερθεί στην εισαγωγή, ο κώδικας βασίζεται στον κεφάλαιο 7 του βιβλίου *SFML Game development by example*. Ο κώδικας του βιβλίου παρέχει μία καλή αρχή για τη δημιουργία ενός παιχνιδιού, έχοντας ορίσει αρκετές βασικές κλάσεις. Ο κώδικας αυτός αξιοποιήθηκε για την δημιουργία ενός πιο ολοκληρωμένου παιχνιδιού. Πιο συγκεκριμένα έγιναν οι παρακάτω προσθήκες:

- Προσθήκη πηχτικών εφέ και μουσικής με μία απλή υλοποίηση
- Υλοποίηση ενός απλού συστήματος *pick up* αντικειμένων για τον παίκτη.
- Προσθήκη HUD με ενδείξεις *healthbar* και αντικείμενα που έχουν συλλεχθεί από τον παίκτη
- Δημιουργία Background Layer στους χάρτες
- Ορισμός ενός πρακτικού συστήματος για την εύκολη δημιουργία χαρτών
- Δημιουργία περισσότερων εχθρών
- Εισαγωγή ενός μεγαλύτερου tileset και αλλαγή του spritesheet για τον παίκτη
- Διόρθωση διαφόρων bugs στον πρωτογενή κώδικα
- Δημιουργία ενός CMakeLists.txt αρχείου (θα εξηγηθεί αναλυτικά σε επόμενη ενότητα)

Πριν όμως περάσουμε σε εξήγηση του κώδικα πρέπει να αναφερθούμε σε ορισμένα βασικά στοιχεία και αρχές που αφορούν το περιβάλλον ανάπτυξης της C++. Ενδεχομένως έμπειροι προγραμματιστές της C++ να γνωρίζουν ήδη ότι πρόκειται να καλυφθεί, αλλά για λόγους πληρότητας θα δοθεί μία αναλυτική περιγραφή στην ενότητα που ακολουθεί.

5.2 Βασικοί ορισμοί

5.2.1 Header files και Implementation files

Ο πηγαίος κώδικας του project αποτελείται από πολλές κλάσεις. Είναι σημαντικό να εξηγηθεί πως κάθε κλάση εμφανίζεται στον κώδικα σε δύο αρχεία:

- Το **header file**, το οποίο έχει κατάλογο **.h** και περιέχει τον ορισμό (definition) της κλάσης και των συναρτήσεών της.
- Το αρχείο που περιέχει την υλοποίηση της κλάσης (implementation) και έχει κατάλογο **.cpp**.

Με αυτόν τον τρόπο, αν η υλοποίηση μίας κλάσης δεν αλλάζει, δεν χρειάζεται να μεταγλωτίστεί ξανά. Συνήθως τα IDEs θα το κάνουν αυτό αυτόματα, δηλαδή θα μεταγλωτίσουν μόνο τις κλάσεις που έχουν αλλάξει στο πρόγραμμα. Αυτό δεν θα ήταν δυνατό αν για μία κλάση αν αποτελούταν μόνο από ένα αρχείο ή αν υπήρχε υλοποίηση στο header file. Παρακάτω δίνεται ένα απλό παράδειγμα:

- Αρχείο **Num.h**



```

1 class Num {
2     private:
3         int num;
4
5     public:
6         Num();
7         Num(int n);
8         int getNum();
9 }

```

- Αρχείο **Num.cpp**

```

1 #include "Num.h"
2
3 Num::Num() : num(0) { } // constructor definition:
4                         // ": num(0)" is the initializer list
5                         // "{ }" is the function body
6
7 Num::Num(int n) : num(n) { } // similarly ": num(n)" is the initializer list
8
9 int Num::getNum() {
10     return num;
11 }

```

- Αρχείο **main.cpp**

```

1 #include <iostream>
2 #include "Num.h"
3
4 int main()
{
    Num n(35);
    std::cout << n.getNum() << std::endl;
    return 0;
}

```

Για να κάνουμε compile αρκεί να δώσουμε στον μεταγλωττιστή g++ τα αρχεία main.cpp και Num.cpp με την παρακάτω εντολή:

```
$ g++ main.cpp Num.cpp
```

Όπως βλέπουμε στα παραπάνω αρχεία, κάθε .cpp αρχείο μπορεί να χρησιμοποιήσει ένα header αρχείο μέσω include statement (γραμμή 1 στο Num.cpp και γραμμή 2 στο main.cpp). Γενικά τα header files περιέχουν ορισμούς συναρτήσεων και μέσω της δίλωσης #include ο μεταγλωττιστής ενημερώνεται πως πρέπει πρώτα να μεταγλωττιστούν οι συναρτήσεις που είναι καταγεγραμμένες στο header file και στη συνέχεια θα μεταγλωττίσει το τωρινό αρχείο.

Υπάρχουν δύο ειδών header files:



- **Standard library header files** (όπως το `iostream` στη γραμμή 1 του αρχείου `main.cpp`)
Τα αρχεία αυτά περιέχονται ήδη στον `g++` μεταγλωττιστή. Παρόμοια βιβλιοθήκες που έχουν εγκατασταθεί, όπως και η `SFML` μπορούν να κληθούν με τον ίδιο τρόπο εφόσον έχουν οριστεί στο `PATH` του υπολογιστή.
- **User-defined header files** (όπως το `Num.h` του παραδείγματος)
Αυτά τα header files έχουν δημιουργηθεί από τον χρήστη. Μπορούμε να τα ξεχωρίσουμε καθώς στο `include` statement το όνομά τους είναι μέσα σε double quotes και όχι μέσα σε "`<`" και "`>`".

Πριν προχωρήσουμε καλό είναι να επισημάνουμε τα `initializer list` που αναφέρονται και στα σχόλια του παραδείγματος, καθώς χρησιμοποιούνται αρκετά στον κώδικα του project. Πιο συγκεκριμένα, το `initializer list` χρησιμοποιείται για να αρχικοποιήσει πεδία μίας κλάσης. Δηλώνεται μετά το όνομα του constructor και των παραμέτρων του. Το `initializer list` ξεκινάει με τον χαρακτήρα ":" και αποτελείται από μία λίστα μεταβλητών οι οποίες θα αρχικοποιηθούν. Οι μεταβλητές χωρίζονται με κόμμα και οι τιμές των μεταβλητών δίνονται μέσα σε παρενθέσεις. Παρακάτω φαίνεται άλλο ένα παράδειγμα χρήσης των `initializer list` από τον constructor της κλάσης EntityBase του πηγαίου κώδικα από το παιχνίδι.

```
EntityBase::EntityBase(EntityManager* l_entityMgr)
    :m_entityManager(l_entityMgr), m_name("BaseEntity"),
     m_type(EntityType::Base), m_id(0), m_referenceTile(nullptr),
     m_state(EntityState::Idle), m_collidingOnX(false), m_collidingOnY(false){}
```

Το `initializer list` βοηθάει ώστε ο κώδικας να είναι πιο περιεκτικός και καθαρός, ειδικά σε περιπτώσεις όπως στο παραπάνω παράδειγμα όπου μόνος σκοπός του constructor είναι οι ανάθεση τιμών σε πεδία της κλάσης.

5.2.2 Κλήση ενός header file

Επιστρέφουμε στο παράδειγμα της κλάσης `Num` και επισημάνουμε πως δεν πρέπει στην C++ ένα header file να γίνεται `include` πολλές φορές. Στα προηγούμενα αρχεία αυτό δεν προκάλεσε πρόβλημα καθώς τα `main.cpp` και `Num.cpp` μεταγλωτίζονται ξεχωριστά. Έστω όμως ότι έχουμε το παρακάτω αρχείο:

- Αρχείο `Foo.h`

```
1 #include "Num.h"
2 class Foo {
3 public:
4     Num n;
5 };
```

Δεν υπάρχει `Foo.cpp` αρχείο καθώς δεν έχουμε συνάρτηση που απαιτεί κάποια υλοποίηση. Συνεχίζουμε κάνοντας τις παρακάτω αλλαγές στο αρχείο `main.cpp`.

- Αρχείο `main.cpp`

```
1 #include <iostream>
2 #include "Num.h"
3 #include "Foo.h"
```



```

4
5 int main()
6 {
7     Num n(35);
8     std::cout << n.getNum() << std::endl;
9
10    Foo f;
11    std::cout << f.n.getNum() << std::endl;
12
13    return 0;
14 }
```

Αν δοκιμάσουμε πάλι να μεταγλωττίσουμε τα αρχεία, ο compiler θα εμφανίσει ένα μήνυμα σφάλματος το οποίο θα αναφέρει πώς γίνεται redefinition της κλάσης Num στο αρχείο main.cpp. Πράγματι αυτό έχει νόημα, επειδή κατά την μεταγλώττιση κάθε δίλωση include αντικαθίσταται από τον αντίστοιχο κώδικα. Έτσι στο αρχείο main.cpp γίνεται πρώτα include το Num.h άρα γίνεται "inject" ο κώδικας της κλάσης Num και στη συνέχεια γίνεται include το Foo.h το οποίο όμως αν δούμε τον κώδικά του καλεί το Num.h. Αυτό θα έχει ως αποτέλεσμα να γίνει "inject" ο κώδικας της κλάσης Num δύο φορές στο αρχείο main.cpp.

Λύση σε αυτό το πρόβλημα δίνει η δίλωση **#ifndef**. Αυτό ονομάζεται **directive** και λειτουργεί ως μήνυμα στο compiler το οποίο τον ενημερώνει να αγνοήσει το αρχείο στο οποίο υπάρχει η δίλωση, αν το αρχείο αυτό έχει ήδη διαβαστεί. Η χρήση του **#ifndef** είναι ως εξής:

```

1 #ifndef NUM_H
2 #define NUM_H
3
4     // your code ...
5
6 #endif
```

Ο compiler όταν διαβάσει το αρχείο, η δίλωση **#ifndef** του υποδεικνύει πως αν έχει ήδη γίνει **defined** το **NUM_H**, θα το αγνοήσει. Αν κάνουμε αυτή τη τροποποίηση στο αρχείο Num.h τότε ο κώδικας θα λειτουργεί κανονικά.

Συνηθίζεται στη δίλωση **#ifndef** να δίνεται σαν όνομα το όνομα του αρχείου (για παράδειγμα όπως είδαμε **NUM_H** για το αρχείο Num.h). Εναλλακτικά μπορεί να χρησιμοποιηθεί το directive **#pragma once** το οποίο επιτυγχάνει το ίδιο με το **#ifndef** χωρίς να χρειάζεται να οριστεί ένα όνομα. Παρακάτω δίνεται ένα παράδειγμα χρήσης του **#pragma once**.

```

1 #pragma once
2
3 // your code ...
4
```

5.2.3 Μεταγλώττιση κατά τμήματα - Separate Compilation

Ο κώδικας που παρουσιάστηκε μέχρι στιγμής ήταν διαχωρισμένος σε header files και implementation (.cpp) files. Χρησιμοποιήθηκε μία φορά (δηλαδή με μία εντολή) ο μεταγλωττιστής g++ για να μεταγλωττίσει ταυτόχρονα όλα τα αρχεία. Προκειμένου να μπορούμε να κάνουμε compile τον κώδικα κατά τμήματα πρέπει να ακολουθήσουμε τα εξής βήματα:



1. Μεταγλωτίζουμε κάθε .cpp αρχείο σε ένα object file, το οποίο θα περιέχει τον κώδικα μπχανίς για το συγκεκριμένο αρχείο.
2. Ενώνουμε (link) όλα τα object files σε ένα εκτελέσιμο.

Για να κάνουμε compile τα .cpp αρχεία σε object files χρησιμοποιούμε το flag -c στο g++:

```
$ g++ -c main.cpp Num.cpp
```

Θα έχουμε ως αποτέλεσμα τα αρχεία **main.o** και **Num.o**, τα οποία πράγματι είναι τα object files. Χρησιμοποιούμε πάλι το g++ για να τα ενώσουμε σε ένα εκτελέσιμο:

```
$ g++ main.o Num.o
```

Έτσι θα παραχθεί ένα εκτελέσιμο με όνομα a.exe (ή a.out ανάλογα το λειτουργικό σύστημα).

Το πλεονέκτημα της μεταγλώττισης κατά τμήματα είναι ότι αν γίνει κάποια αλλαγή σε ένα .cpp αρχείο τότε αρκεί απλά να φτιαχτεί το νέο object file και να γίνει πάλι η ένωση με τα ίδια υπάρχοντα object files. Για παράδειγμα έστω ότι αλλάζουμε κάτι στο αρχείο main.cpp. Αρκεί λοιπόν να το μεταγλωτίσουμε:

```
$ g++ -c main.cpp
```

και στη συνέχεια να κάνουμε πάλι το βήμα της ένωσης:

```
$ g++ main.o Num.o
```

Παρατηρούμε πως δεν χρειάστηκε να μεταγλωτίσουμε πάλι το Num.cpp καθώς δεν άλλαξε και υπήρχε ίδιο το object file. Αυτή η δυνατότητα είναι ιδιαίτερα σημαντική καθώς αν το project αποτελείται από πολλά .cpp αρχεία ο χρόνος μεταγλώττισης είναι αρκετά μεγάλος. Μεταγλωτίζοντας μόνο τα αρχεία που άλλαξαν εξουκονομούμε πολύ χρόνο και γι' αυτό θα χρησιμοποιηθεί Separate Compilation στο project μας.

5.2.4 Εργαλείο Make

Η διαδικασία της μεταγλώττισης κατά τμήματα συνήθως γίνεται αυτόμata από τα IDEs. Σε επίπεδο command line όμως πρέπει να οριστεί από τον χρήστη. Εδώ έρχεται το εργαλείο Make το οποίο αναλαμβάνει να εκτελέσει αυτόμata τις εντολές για την παραγωγή των object files και την τελική ένωση τους. Για να χρησιμοποιηθεί το εργαλείο Make αρκεί ο χρήστης να ορίσει ένα **Makefile**. Παρακάτω δίνεται ένα παράδειγμα για το Makefile που θα μπορούσε να έχει το πρόγραμμά που παρουσιάστηκε.

```

1  CFLAGS = -O
2  CC = g++
3
4  NumTest: main.o Num.o
5      $(CC) $(CFLAGS) -o NumTest main.o Num.o
6
7  main.o: main.cpp
8      $(CC) $(CFLAGS) -c main.cpp
9
10 Num.o: Num.cpp

```



```
11 $(CC) $(CFLAGS) -c Num.cpp
12
13 clean:
14     rm -f core *.o
```

Ουσιαστικά στο αρχείο αυτό δηλώνεται ότι το NumTest εξαρτάται από τα main.o και Num.o (γραμμή 4) και στη επόμενη γραμμή δηλώνεται πώς θα γίνει η μεταγλωττιση.

Στη συνέχεια δηλώνεται στο Makefile πώς θα δημιουργηθούν τα αρχεία main.o (γραμμή 7) και Num.o (γραμμή 10). Μπορούμε να μεταγλωττίσουμε πληκτρολογώντας:

```
$ make
```

ή εναλλακτικά

```
$ make NumTest
```

Σε κάθε περίπτωση γίνεται αυτόματος έλεγχος από το make αν τα main.cpp και Num.cpp έχουν αλλάξει ή όχι και θα τα μεταγλωττίσει εφόσον αυτό είναι αναγκαίο.

Η τελευταία δήλωση στο Makefile είναι το clean το οποίο επιτρέπει την αυτόματη διαγραφή όλων των object files όταν ο χρήστης πληκτρολογήσει make clean.

Περισσότερες πληροφορίες για την σύνταξη και την χρήση των Makefiles δίνεται σε ένα σχετικό άρθρο στην βιβλιογραφία.

5.2.5 Εργαλείο CMake

Στο εργαλείο Make που μόλις παρουσιάστηκε παρατηρήσαμε ότι πρέπει να συνοδεύεται από το αντίστοιχο Makefile το οποίο περιέχει ουσιαστικά όλες τις εξαρτήσεις που έχουν τα αρχεία κώδικα, πώς πρέπει να συνδεθούν και να μεταγλωττιστούν. Στη περίπτωση που έχουμε λίγα αρχεία είναι εύκολο να γράψουμε το Makefile. Όμως το τωρινό project αποτελείται από τουλάχιστον 20 κλάσεις (δηλαδή .cpp αρχεία) οι οποίες συνδέονται μεταξύ τους, κάτι που παίζει αρκετό ρόλο για την μεταγλωττιση κατά την παραγωγή. Δεν θα ήταν αρκετά βολικά αν μπορούσαμε με κάποιο τρόπο να παράγουμε αυτόματα ένα Makefile χωρίς να χρειάζεται να μελετήσουμε όλες τις συνδέσεις που έχουν τα αρχεία κώδικα; Σε αυτό το σημείο έρχεται το εργαλείο CMake το οποίο θα μας επιτρέψει να έχουμε Separate Compilation με πολύ εύκολο τρόπο.

Το CMake είναι γνωστό ως ένα **meta build system**. Στην πραγματικότητα δεν κάνει αυτό build τον πηγαίο κώδικα αλλά δημιουργεί τα απαραίτητα build αρχεία για την πλατφόρμα στην οποία θα τρέξει το πρόγραμμα. Για παράδειγμα, το CMake στα Windows θα παράγει ένα Visual Studio solution αρχείο, ενώ στα Linux θα παράγει ένα Makefile. Αυτό σημαίνει η λέξη meta: Το CMake χρησιμοποιείται ουσιαστικά για να κάνει build ένα build system και μπορεί να το επεξεργαστεί ο χρήστης ανεξάρτητα του λειτουργικού συστήματος στο οποίο τελικά θα γίνει το build. Με άλλα λόγια ορίζουμε στο CMake πώς θα γίνει build ένα project και με το ακριβώς ίδιο configuration θα έχουμε επιτυχής compile σε Linux, Mac OS και Windows.

Πιο συγκεκριμένα το CMake γίνεται configure μέσω του αρχείου **CMakeLists.txt** το οποίο υπάρχει στον φάκελο του πηγαίου κώδικα και θα παρουσιαστεί.

Αρχικά καθορίζεται η minimum version CMake στην κορυφή του αρχείου ακολουθούμενη από ένα όνομα project και έναν αριθμό έκδοσης:



```
cmake_minimum_required(VERSION 3.12 FATAL_ERROR)  
project(ScrollOfTheUndead VERSION 0.1)
```

Στη συνέχεια, ζητάμε να χρησιμοποιηθούν χαρακτηριστικά του προτύπου C++ 11 για το build του project. Η μεταβλητή CMAKE_CXX_EXTENSIONS έχει επίσης οριστεί σε OFF για να διασφαλιστεί ότι το flag -std=c++11 μεταβιβάζεται στον μεταγλωττιστή κατά το build (το flag -std=c++11 λέει στον g++ να χρησιμοποιήσει C++11 όταν γίνεται η μεταγλώττιση).

```
set(CMAKE_CXX_STANDARD 11)  
set(CMAKE_CXX_EXTENSIONS OFF)  
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
```

Στη συνέχεια, ορίζεται το default build type. Γενικά ο όρος **build type** αναφέρεται συνήθως είτε **debug**, είτε σε **release**. Στο δικό μας αρχείο ορίζουμε το **debug build** ως **default**. Το debug build συνήθως έχει ταχύτερο compile και επιτρέπει την χρήση εργαλείων αποσφαλμάτωσης (debugging).

```
if(NOT CMAKE_BUILD_TYPE)  
    set(CMAKE_BUILD_TYPE Debug  
        CACHE STRING  
            "Choose the type of build (Debug or Release)" FORCE)  
endif()
```

Η επόμενη εντολή δημιουργεί ένα header file που ονομάζεται **config.h** στον φάκελο **include**, το οποίο περιλαμβάνει ορισμένες πληροφορίες, όπως την έκδοση του project. Το CMake διαβάζει το **config.h.in** για να δημιουργήσει αυτό το header file.

```
configure_file(include/config.h.in config.h)
```

Σε αυτό το σημείο καλούμε τη συνάρτηση **find_package()** για να εντοπίσουμε τις SFML shared libraries και τα αντίστοιχα header files τους. Το CMake είναι αρκετά έξυπνο ώστε να ανιχνεύει το λειτουργικό σύστημα και να εντοπίσει πού βρίσκεται η βιβλιοθήκη SFML. Το μόνο που χρειάζεται να κάνουμε είναι να περάσουμε την έκδοση του SFML μαζί με τα συγκεκριμένα components που θέλουμε να εντοπιστούν:

```
find_package(SFML 2.5 COMPONENTS system window graphics audio REQUIRED)
```

Συνεχίζουμε με την εντολή **add_executable** η οποία θα κάνει compile τα .cpp αρχεία που θα της δοθούν σε ένα εκτελέσιμο με όνομα **ScrollOfTheUndead**. Απαιτείται να καταγράψουμε όλα τα .cpp αρχεία στον φάκελο **src**.

```
add_executable(ScrollOfTheUndead  
    src/Anim_Base.cpp  
    src/Anim_Directional.cpp  
    src/Character.cpp  
    src/Enemy.cpp  
    src/EntityBase.cpp  
    src/EntityManager.cpp  
    src/EventManager.cpp
```



```
src/Game.cpp
src/Main.cpp
src/Map.cpp
src/Player.cpp
src/SpriteSheet.cpp
src/State_ChooseMap.cpp
src/State_Game.cpp
src/State_GameOver.cpp
src/State_Intro.cpp
src/State_LevelCompleted.cpp
src/State_MainMenu.cpp
src/StateManager.cpp
src/State_Paused.cpp
src/State_YesNoMenu.cpp
src/Window.cpp )
```

Επιπλέον πρέπει να δηλώσουμε στο CMake ποιος είναι ο φάκελος με τα .h αρχεία

```
target_include_directories(ScrollOfTheUndead
    PRIVATE
        "${PROJECT_BINARY_DIR}"
        "${CMAKE_CURRENT_SOURCE_DIR}/include"
)
```

Επίσης πρέπει να συνδέσουμε το εκτελέσιμο με τα components της SFML που χρησιμοποιεί.

```
target_link_libraries(ScrollOfTheUndead sfml-graphics sfml-audio)
```

Για να μεταγλωτίσουμε όλο το project αρκεί να δημιουργήσουμε έναν νέο φάκελο. Συνθίζεται ο φάκελος αυτός να ονομάζεται "**build**". Στη συνέχεια θα μεταβούμε μέσα στον φάκελο αυτό και θα καλέσουμε το CMake

```
$ mkdir build && cd build
$ cmake ..
$ cmake --build .
```

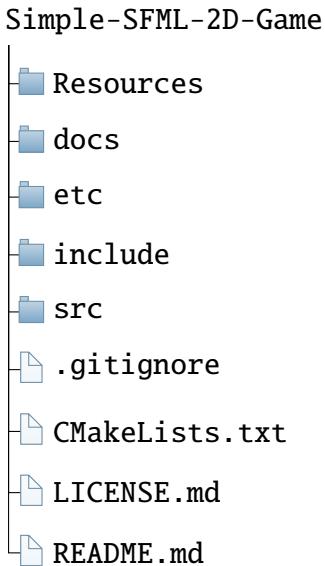
Με την πρώτη εντολή δημιουργούμε έναν φάκελο **build** και μεταβαίνουμε μέσα σε αυτόν. Με την δεύτερη εντολή καλούμε το CMake στον φάκελο που βρίσκεται μία θέση πάνω από το τωρινό path. Το CMake θα διαβάσει το αρχείο CMakeLists.txt και θα δημιουργήσει ειδικά αρχεία μέσα στον φάκελο **build**. Σε αυτό το σημείο είναι έτοιμο το project για να μεταγλωτιστεί και το κάνουμε είτε με την εντολή **make** είτε με την τρίτη εντολή που φαίνεται παραπάνω.

Σημειώνεται ότι τα βήματα μεταγλωτισης ενδεχομένως να διαφέρουν ανάλογα το λειτουργικό σύστημα. Περισσότερες οδηγίες υπάρχουν στην ενότητα **Εκτέλεση**.



6 Πηγαίος κώδικας

Παρακάτω δίνεται η αρχειοθέτηση του πηγαίου κώδικα



Εικόνα 4: Αρχειοθέτηση του πηγαίου κώδικα

- Ο πηγαίος κώδικας βρίσκεται στον φάκελο **Simple-SFML-2D-Game** και αποτελείται από **.cpp** αρχεία και **.h** αρχεία. Τα **.cpp** αρχεία βρίσκονται στον φάκελο **src** και τα **.h** αρχεία στον φάκελο **include** αντίστοιχα.
- Στον φάκελο **Resources** βρίσκονται αρχεία ήχου, εικόνες και αρχεία κειμένου σε κατάλληλες μορφές ώστε να διαβάζονται από το εκτελέσιμο πρόγραμμα κατά την εκτέλεση. Τα αρχεία αυτά ουσιαστικά είναι τα assets του παιχνιδιού.
- Στον φάκελο **docs** υπάρχουν screenshots, latex και pdf αρχεία που αφορούν το παρόν πρόγραμμα.
- Στον φάκελο **etc** υπάρχουν βοηθητικά αρχεία που δεν σχετίζονται άμεσα με τον πηγαίο κώδικα ή τα Resources. Σε αυτόν τον φάκελο υπάρχουν τα asesprite/libresprite αρχεία για τα διάφορα spritesheets του παιχνιδιού καθώς και αρχεία που αφορούν την δημιουργία χαρτών.
- Το αρχείο **CMakeLists.txt** αφορά το build και έχει ήδη παρουσιαστεί σε προηγούμενη ενότητα.
- Τα αρχεία **README.md** και **.gitignore** αφορούν αποκλειστικά το Github. Το **README.md** περιέχει τα περιεχόμενα που εμφανίζονται στην αρχική σελίδα του αντίστοιχου repository και το αρχείο **.gitignore** έχει καταγεγραμμένα τα αρχεία που αγνοούνται από το git source control.
- Το αρχείο **LICENSE.md** περιέχει την άδεια του προγράμματος (GPL-3.0 license).

Στην σελίδα που ακολουθεί δίνεται η πλήρης λίστα των αρχείων στους φακέλους **src** και **include**.



```
include
└── Anim_Base.h
└── Anim_Directional.h
└── BaseState.h
└── Character.h
└── DebugOverlay.h
└── Directions.h
└── Enemy.h
└── EntityBase.h
└── EntityManager.h
└── EventManager.h
└── Game.h
└── Map.h
└── Player.h
└── ResourceManager.h
└── SharedContext.h
└── SpriteSheet.h
└── StateManager.h
└── State_ChooseMap.h
└── State_Game.h
└── State_GameOver.h
└── State_Intro.h
└── State_LevelCompleted.h
└── State_MainMenu.h
└── State_Paused.h
└── State_YesNoMenu.h
└── TextureManager.h
└── Utilities.h
└── Window.h
└── config.h.in
```

(a) Φάκελος include

```
src
└── Anim_Base.cpp
└── Anim_Directional.cpp
└── Character.cpp
└── Enemy.cpp
└── EntityBase.cpp
└── EntityManager.cpp
└── EventManager.cpp
└── Game.cpp
└── Main.cpp
└── Map.cpp
└── Player.cpp
└── SpriteSheet.cpp
└── StateManager.cpp
└── State_ChooseMap.cpp
└── State_Game.cpp
└── State_GameOver.cpp
└── State_Intro.cpp
└── State_LevelCompleted.cpp
└── State_MainMenu.cpp
└── State_Paused.cpp
└── State_YesNoMenu.cpp
└── Window.cpp
```

(b) Φάκελος src



Υπενθυμίζεται ότι κάθε κλάση αποτελείται από το **header file** (το οποίο βρίσκεται στον φάκελο **include**) και το **.cpp file** (που βρίσκεται στον φάκελο **src**). Επίσης και τα δύο αρχεία θα έχουν σαν όνομα, το όνομα της κλάσης και σαφώς θα ακολουθεί η ανάλογη κατάληξη. Στον φάκελο **src** όλα τα αρχεία εκτός από το **Main.cpp** είναι αρχεία κλάσεων. Υπάρχουν στον φάκελο **include** τα αντίστοιχα **header** αρχεία των κλάσεων αυτών καθώς επίσης ορισμένα επιπρόσθετα αρχεία **.h** τα οποία θα παρουσιαστούν στη συνέχεια.

Είναι σαφές ότι ο πηγαίος κώδικας είναι αρκετά μεγάλος και ως εκ τούτου δεν θα εξηγηθεί σε όλη την έκταση σε αυτό το αρχείο. Αντίθετα έχουν επιλεχθεί κατάλληλα μέρη τα οποία αξίζουν μία επεξήγηση και θα παρουσιαστούν σε επόμενες ενότητες. Τα μέρη αυτά αφορούν:

1. την διαχείριση εισόδου από τον χρήστη
2. τις καταστάσεις του προγράμματος
3. την κλάση **Map** (μία από τις αρκετά σημαντικές κλάσεις)
4. τον τρόπο με τον οποίο δημιουργείται ένας χάρτης στο παιχνίδι
5. τον τρόπο με τον οποίο δημιουργείται ένας "εχθρός" στο παιχνίδι

Αρχικά όμως θα γίνει μία σύντομη παρουσίαση του συνόλου των αρχείων του κώδικα στην ενότητα που επακολουθεί.

6.1 Βασικά αρχεία και διαγράμματα κλάσεων

6.2 Διαχείριση εισόδου από τον χρήστη - EventManager

6.2.1 Χρήση του Event της SFML

Ο τρόπος με τον οποίο μπορούμε στην SFML να ελέγχουμε σε πραγματικό χρόνο αν κάποιο πλήκτρο πατήθηκε από τον χρήστη με την μέθοδο **isKeyPressed**.

```
if(sf::Keyboard::isKeyPressed(sf::Keyboard::W)){  
    // Do something if the W key is pressed.  
}
```

Υπάρχει αντίστοιχη συνάρτηση και για τα "κλικ" του ποντικιού.

```
if(sf::Mouse::isButtonPressed(sf::Mouse::Left)){  
    // Do something if the left mouse button is pressed.  
}
```

Σημείωση

Η SFML υποστηρίζει και είσοδο από Joystick. Στο βιβλίο SFML Game Development By Example γίνεται μια σχετική αναφορά αλλά δεν θα συμπεριληφθεί στο παρόν αρχείο ή στο project. Για περισσότερες πληροφορίες σχετικά με την χρήση joystick για είσοδο μπορεί να μελετηθεί η κλάση **sf::Joystick** από το επίσημο documentation της SFML.



Παρακάτω βλέπουμε πώς μπορούμε να χειριστούμε ένα Event στην SFLM.

```
sf::Event event;
while(m_window.pollEvent(event)){
    switch(event.type){

        case sf::Event::Closed:
            m_window.close();
            break;

        case sf::Event::KeyPressed:
            if(event.key.code == sf::Keyboard::W){
                // Do something when W key gets pressed once.
            }
            break;
    }
}
```

Αρχικά, το πεδίο sf::Event με όνομα event συμπληρώνεται από τη μέθοδο pollEvent(). Στη συνέχεια, μπορούμε να ελέγξουμε τον τύπο του event, ο οποίος ορίζεται από τον πίνακα enum sf::Event::Type και να βεβαιωθούμε ότι χρησιμοποιούμε το σωστό τύπο πριν διαβάσουμε κάποια πληροφορία. Αν για παράδειγμα ο τύπος του event είναι sf::Event::Closed και εμείς προσπαθήσουμε να διαβάσουμε το event.key.code, θα έχουμε απρόβλεπτη συμπεριφορά.

Σημείωση

Η χρήση του sf::Event::KeyPressed για κάτι σαν την κίνηση του χαρακτήρα σε πραγματικό χρόνο είναι **κακή** ιδέα. Αυτό το συμβάν αποστέλλεται μόνο μία φορά πριν εφαρμοστεί μια μικρή καθυστέρηση και στη συνέχεια αποστέλλεται ξανά. Σκεφτείτε εδώ έναν επεξεργαστή εγγράφων. Όταν πατάτε ένα πλήκτρο και το κρατάτε πατημένο, στην αρχή εμφανίζεται μόνο ένας χαρακτήρας πριν γράψει περισσότερους. Αυτός είναι ακριβώς ο ίδιος τρόπος με τον οποίο λειτουργεί αυτό το συμβάν. Η χρήση του για οποιαδήποτε ενέργεια που πρέπει να είναι συνεχής όσο το πλήκτρο κρατιέται πατημένο δεν είναι καν κοντά στο βέλτιστο και θα πρέπει να αντικατασταθεί με το sf::Keyboard::isKeyPressed() για να ελέγχεται η πραγματική κατάσταση του πλήκτρου. Το ίδιο ισχύει για την είσοδο του ποντικιού και του ελεγκτή. Η χρήση αυτών των συμβάντων είναι ιδανική για πράγματα που πρέπει να συμβαίνουν μόνο μία φορά ανά πάτημα του πλήκτρου.

Ενώ αυτή η προσέγγιση είναι διαχειρίσιμη σε περιπτώσεις μικρών έργων, λίγο πολύ όπως ήταν και το προηγούμενο παραδειγμα εισόδου, μπορεί να ξεφύγει γρήγορα σε μεγαλύτερη κλίμακα. Ας είμαστε ειλικρινείς, ο χειρισμός όλων των συμβάντων, των πληκτρολογήσεων και των καταστάσεων κάθε συσκευής εισόδου με τον τρόπο που κάναμε στο προηγούμενο έργο είναι εφιάλτης. Φανταστείτε να έχετε μια εφαρμογή όπου θέλετε να ελέγχετε αν πατιούνται πολλά πλήκτρα ταυτόχρονα και να καλέσετε κάποια συνάρτηση όταν πατιούνται. Αυτό προσθέτει άλλο ένα επίπεδο πολυπλοκότητας, κάνοντας πιο δύσχρονο τον ίδιο τον κώδικα παρά χρήσιμο.

Μια αυτοματοποιημένη προσέγγιση θα είναι ευέλικτη, θα μπορεί να φορτώνει οποιονδήποτε συνδυασμό πλήκτρων και συμβάντων από ένα αρχείο και θα διατηρεί τον κώδικα εξίσου τακτοποιημένο και καθαρό όπως ήταν πριν.



6.2.2 Εισαγωγή στην κλάση EventManager

Θέλουμε να αυτοματοποιηθεί το δομή που θα ορίσουμε να υποστηρίζει τις παρακάτω λειτουργίες:

- Δυνατότητα σύζευξης οποιουδήποτε συνδυασμού πλήκτρων, κουμπιών ή events με την επιθυμητή λειτουργικότητα.
- Να γίνονται bind οι εν λόγω λειτουργίες σε συναρτήσεις που καλούνται εάν όλες οι συνθήκες (π.χ. το πάτημα ενός πλήκτρου, το πάτημα του αριστερού κλικ, ή το παραθύρο χάνει την εστίαση) για τη δέσμευση ικανοποιούνται.
- Φόρτωση των bindings από ένα αρχείο κειμένου.

Θα χρησιμοποιήσουμε το αρχείο EventManager.h για να συμπεριλάβουμε όλα τα μικρά κομμάτια που το καθιστούν δυνατό, εκτός από τον ορισμό της κλάσης. Το πρώτο πράγμα που πρέπει να ορίσουμε είναι όλοι οι τύποι συμβάντων με τους οποίους θα ασχοληθούμε σε ένα enum.

```
enum class EventType{
    KeyDown = sf::Event::KeyPressed,
    KeyUp = sf::Event::KeyReleased,
    MButtonDown = sf::Event::MouseButtonPressed,
    MButtonUp = sf::Event::MouseButtonReleased,
    MouseWheel = sf::Event::MouseWheelMoved,
    WindowResized = sf::Event::Resized,
    GainedFocus = sf::Event::GainedFocus,
    LostFocus = sf::Event::LostFocus,
    MouseEntered = sf::Event::MouseEntered,
    MouseLeft = sf::Event::MouseLeft,
    Closed = sf::Event::Closed,
    TextEntered = sf::Event::TextEntered,
    Keyboard = sf::Event::Count + 1, Mouse, Joystick
};
```

Στη συνέχεια, θα καταστήσουμε δυνατή την αποθήκευση ομάδων από events για κάθε bind. Γνωρίζουμε ότι για να κάνουμε bind σε ένα key, χρειαζόμαστε τόσο το event type όσο και τον κωδικό του key που μας ενδιαφέρει. Ορισμένα events με τα οποία θα δουλέψουμε πρέπει να έχουν αποθηκευμένο μόνο το event type, και σε αυτές τις περιπτώσεις μπορούμε απλά να αποθηκεύσουμε μια ακέραια τιμή 0 με το type. Γνωρίζοντας αυτό, ας ορίσουμε μια νέα δομή που θα μας βοηθήσει να αποθηκεύσουμε αυτές τις πληροφορίες:

```
struct EventInfo{
    EventInfo(){m_code = 0; }
    EventInfo(int l_event){ m_code = l_event; }

    union{
        int m_code;
    };
};
```

Για να αφήσουμε χώρο για επεκτάσεις, χρησιμοποιούμε **union** για την αποθήκευση του κωδικού συμβάντος. Στη συνέχεια, μπορούμε να ορίσουμε τον τύπο δεδομένων που θα χρησιμοποιήσουμε για την αποθήκευση των πληροφοριών συμβάντος:



```
using Events = std::vector<std::pair<EventType, EventInfo>>;
```

Δεδομένου ότι θα χρειαστεί να μοιραστούμε τις πληροφορίες συμβάντος με τον κώδικα που χρησιμοποιεί αυτή την κλάση, τώρα είναι η κατάλληλη στιγμή για να δημιουργήσουμε έναν τύπο δεδομένων που θα μας βοηθήσει να το κάνουμε αυτό:

```
struct EventDetails{
    EventDetails(const std::string& l_bindName)
        : m_name(l_bindName)
    {
        Clear();
    }
    std::string m_name;
    sf::Vector2i m_size;
    sf::Uint32 m_textEntered;
    sf::Vector2i m_mouse;
    int m_mouseWheelDelta;
    int mKeyCode; // Single key code.

    void Clear(){
        m_size = sf::Vector2i(0, 0);
        m_textEntered = 0;
        m_mouse = sf::Vector2i(0, 0);
        m_mouseWheelDelta = 0;
        mKeyCode = -1;
    }
};
```

Τώρα οντα σχεδιάσουμε το binding structure, το οποίο θα περιέχει όλες τις πληροφορίες για τα events.

```
struct Binding{
    Binding(const std::string& l_name)
        : m_name(l_name), m_details(l_name), c(0){}

    void BindEvent(EventType l_type, EventInfo l_info = EventInfo()){
        m_events.emplace_back(l_type, l_info);
    }

    Events m_events;
    std::string m_name;

    int c; // Count of events that are "happening".
    EventDetails m_details;
};
```

Ο constructor λαμβάνει το όνομα της ενέργειας στην οποία θέλουμε να δεσμεύσουμε τα events και χρησιμοποιεί το initializer list για να ρυθμίσει τα μέλη της κλάσης. Έχουμε επίσης μια μέθοδο BindEvent(), η οποία απλώς λαμβάνει ένα event type και μια δομή EventInfo προκειμένου να το προσθέσει στο vector των events. Ένα πρόσθετο μέλος που δεν έχουμε αναφέρει προηγουμένως είναι ο ακέραιος αριθμός με το όνομα c. Όπως υποδηλώνει το



σχόλιο, παρακολουθεί πόσα γεγονότα λαμβάνουν πραγματικά χώρα, το οποίο θα είναι χρήσιμο αργότερα προκειμένου να προσδιοριστεί αν όλα τα keys και τα events στο binding είναι ενεργοποιημένα.

Αυτά τα bindings πρέπει να αποθηκεύονται σε μία δομή δεδομένων και την ορίζουμε παρακάτω

```
using Bindings = std::unordered_map<std::string, Binding*>;
```

Η χρήση του std::unordered_map για τα bindings μας εγγυάται ότι θα υπάρχει μόνο ένα binding ανά action, επειδή το κλειδί είναι το όνομα του action.

Τέλος στον πηγαίο κώδικα έχουμε ορίσει function wrappers τα οποία θα καλεί ένα event ώστε να ενεργοποιεί οποιαδήποτε μέθοδο θέλουμε.

6.3 Καταστάσεις του προγράμματος - BaseState και StateManager

Ένα βιντεοπαιχνίδι δεν αποτελείται μόνο από γραφικά. Είναι πολύ συνηθισμένο σήμερα ένα παιχνίδι να έχει μια εισαγωγή πριν την έναρξη του, η οποία μάλιστα θα συνοδεύεται από κάποιο animation. Επίσης είναι αναγκαίο να υπάρχει και ένα μενού από το οποίο ο χρήστης θα ξεκινάει το παιχνίδι ή θα επιλέγει ανάμεσα σε ορισμένες θυμητήρες ή θα κλείνει την εφαρμογή. Είναι σημαντικό επίσης το παιχνίδι να έχει τη δυνατότητα να διακόπτεται (pause) από τον χρήστη. Όλες αυτές οι λειτουργικότητες πρέπει να ληφθούν υπ' όψη από τα πρώτα στάδια ανάπτυξης ενός παιχνιδιού. Ο τρόπος με τον οποίο θα αντιμετωπιστούν αυτές οι απαιτήσεις είναι μέσω της υλοποίησης των **καταστάσεων**.

Πριν συνεχίσουμε πρέπει να εξηγηθεί ακριβώς η έννοια της **κατάστασης** σε ένα παιχνίδι. Πρόκειται για ένα από τα πολλά επίπεδα (layers) που μπορεί να έχει ένα παιχνίδι, όπως το κεντρικό μενού, το animation που εμφανίζεται πριν το κεντρικό μενού και τέλος το ίδιο το παιχνίδι. Κάθε ένα από αυτά τα επίπεδα έχει τον δικό του τρόπο να ενημερώνεται και να κάνει render τα περιεχόμενά του στην οθόνη.

Το βιβλίο SFML Game Development by example προτείνει μία υλοποίηση αρκετά καλή για τα πλαίσια της εργασίας και έχει ενσωματωθεί στον πηγαίο κώδικα. Βασικό στοιχείο της υλοποίησης είναι η **δημιουργία ξεχωριστών κλάσεων** για την κάθε κατάσταση. Όλες οι κλάσεις θα έχουν ορισμένες κοινές μεθόδους για να ενημερώνονται και να κάνουν render, κάτι που επιτυχγάνεται με την **κληρονομικότητα**. Παρακάτω ορίζουμε την κλάση BaseState, από την οποία θα κληρώνομούν όλες οι κλάσεις που αντιστοιχούν σε μία κατάσταση του παιχνιδιού.

```

1  class BaseState{
2      friend class StateManager;
3
4  public:
5      BaseState(StateManager* lStateManager)
6          :m_stateMgr(lStateManager), m_transparent(false),
7          m_transcient(false){}
8
9      virtual ~BaseState(){}
10
11     virtual void OnCreate() = 0;
12     virtual void OnDestroy() = 0;
13
14     virtual void Activate() = 0;
15     virtual void Deactivate() = 0;
```



```

16     virtual void Update(const sf::Time& l_time) = 0;
17     virtual void Draw() = 0;
18
19
20     void SetTransparent(const bool& l_transparent){
21         m_transparent = l_transparent;
22     }
23
24     bool IsTransparent()const{ return m_transparent; }
25
26     void SetTranscendent(const bool& l_transcendence){
27         m_transcendent = l_transcendence;
28     }
29
30     bool IsTranscendent()const{ return m_transcendent; }
31
32     StateManager* GetStateManager(){ return m_stateMgr; }
33
34 protected:
35     StateManager* m_stateMgr;
36     bool m_transparent;
37     bool m_transcendent;
38 };

```

Επειδή είναι αναγκαίο κάθε κλάση που κληρωνούμει από την BaseState να έχει υλοποιήσει τις μεθόδους της, όλες οι μέθοδοι δηλώνονται ως **purely virtual** (έχουν το keyword 'virtual' και στο τέλος '=0').

Οι μέθοδοι που πρέπει να υλοποιήσει οποιαδήποτε κλάση που κληρωνούμει από την BaseState είναι οι παρακάτω:

1. **OnCreate** και **OnDestroy**, οι οποίες καλούνται όταν δημιουργείται η κατάσταση και εισάγεται στη στοίβα, και αργότερα όταν αφαιρεθεί από τη στοίβα
2. **Activate** και **Deactivate**, που καλούνται όταν μια κατάσταση μετακινείται στην κορυφή της στοίβας καθώς και όταν αφαιρείται από την κορυφή της στοίβας.
3. **Update** και **Draw**, οι οποίες χρησιμοποιούνται για την ενημέρωση της κατάστασης και το σχεδιασμό του περιεχομένου της κατάστασης.

Κάθε κατάσταση του προγράμματος θα οριστεί ως μία κλάση η οποία θα κληρωνούμει από την BaseState. Παρόμοια με τις καταστάσεις θα ορίσουμε μια κλάση Manager η οποία θα διαχειρίζεται τις καταστάσεις σε μία στοίβα και θα υλοποιεί την εναλλαγή ανάμεσα στις κλάσεις.

```

1
2 enum class StateType{ Intro = 1, MainMenu, Game, Paused, GameOver, Credits,
3                         LevelCompleted, ChooseMap, YesNoMenu };
4
5 // State container.
6 using StateContainer = std::vector<std::pair<StateType, BaseState*>>;
7 // Type container.
8 using TypeContainer = std::vector<StateType>;
9 // State factory.
10 using StateFactory = std::unordered_map<StateType, std::function<BaseState*(&void)>>;

```



```

11
12 class StateManager{
13     public:
14         StateManager(SharedContext* l_shared);
15         ~StateManager();
16
17         void Update(const sf::Time& l_time);
18         void Draw();
19
20         void ProcessRequests();
21
22         SharedContext* GetContext();
23         bool HasState(const StateType& l_type);
24
25         void SwitchTo(const StateType& l_type);
26         void Remove(const StateType& l_type);
27     private:
28         // Methods.
29         void CreateState(const StateType& l_type);
30         void RemoveState(const StateType& l_type);
31
32         template<class T>
33         void RegisterState(const StateType& l_type){
34             m_stateFactory[l_type] = [this]()->BaseState*
35             {
36                 return new T(this);
37             };
38         }
39
40         // Members.
41         SharedContext* m_shared;
42         StateContainer m_states;
43         TypeContainer m_toRemove;
44         StateFactory m_stateFactory;
45
46         sf::Music m_musicMain;
47
48         sf::SoundBuffer m_bufferUIsound;
49         sf::Sound m_UIsound;
50
51 };

```

Ο StateManager διατηρεί και πεδία που αφορούν ήχο (sf::Music, sf::SoundBuffer και sf::Sound). Αυτό γίνεται διότι πρέπει να παίζει μία μουσική στην κατάσταση του αρχικού μενού και της επιλογής χάρτη. Επίσης κάθε φορά που γίνεται αλλαγή καταστάσεων θα παίζει ένας ήχος.

6.4 Χάρτες

Ο τρόπος με τον οποίο δημιουργείται ένα χάρτης είναι ορίζοντας ένα αρχείο κείμενο το οποίο έχει την παρακάτω μορφή



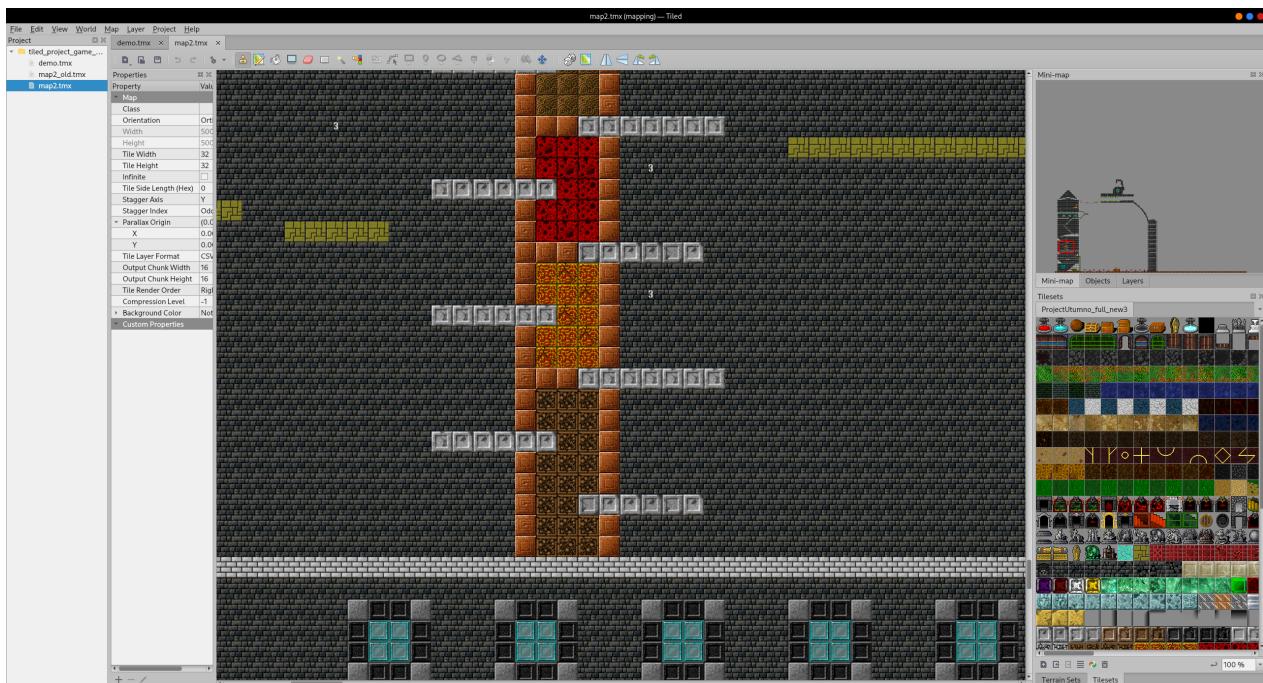
```

BACKGROUND MountainDusk
MUSIC Nightmare.wav
SIZE 500 500
GRAVITY 512
DEFAULT_FRICTION 0.8 0
TILE 872 159 258
TILE 872 160 258
TILE 872 161 258
...
ENEMY Skeleton 3424 9888
TILE 1070 5 310
TILE 1070 6 310
TILE 1070 62 310
TILE 1070 63 310
PLAYER 2624 9920
...

```

Ο χάρτης θα αποτελείται από tiles. Αρχικά ορίζεται η βαρύτητα και η τριβή που θα έχουν τα tiles. Με τι δίλωση **TILE 872 159 258**, το 872 είναι το id του tile και τα 159, 258 οι συντεταγμένες του μέσα στον χάρτη με αντίστοιχο τρόπο βάζουμε έναν εχθρό στον χάρτη (**ENEMY Skeleton 3424 9888**) καθώς και ορίζουμε την θέση που θα ξεκινήσει ο χρήστης (**PLAYER 2624 9920**).

Σαφώς είναι δύσκολο να φτιάξουμε έναν χάρτη απλά γράφοντας με το χέρι το αρχείο αυτό. Εδώ μπορούμε να χρησιμοποιήσουμε ένα εξωτερικό εργαλείο που ονομάζεται Tiled και θα μας βοηθήσει να δημιουργούμε χάρτες με γραφικό τρόπο.



Εικόνα 6: Το πρόγραμμα Tiled (<https://www.mapeditor.org/>)

Το Tiled δέχεται το Tileset που έχουμε και κάνει export τον χάρτη που δημιουργούμε σε .csv αρχείο. Μπορούμε να μετατρέψουμε το .csv στην μορφή που θέλουμε με αυτοματοποιημένο τρόπο καλώντας ένα python script που έχουμε φτιάξει.



Κάτι που δεν αναφέραμε είναι ότι ο χάρτης αποτελείται και από ένα δεύτερο layer που θα βρίσκεται στο παρασκήνιο. Τα tiles που βρίσκονται στο δεύτερο layer δεν θα έχουν συγκρούσεις με τους χαρακτήρες του παιχνιδιού.

Το σημαντικό είναι ότι ένας χάρτης αποτελείται μόνο από το αρχείο κειμένου (.map το ονομάζουμε) που περιέχει όλες τις δηλώσεις για το που θα είναι οι εχθροί τα tiles και ο παίκτης. Το πρόγραμμα διαβάζει δυναμικά αυτό το αρχείο και δημιουργεί τον χάρτη. Αυτό σημαίνει ότι δεν χρειάζεται ο κώδικας να γίνει compile κάθε φορά που αλλάζουμε ή δημιουργούμε έναν χάρτη. Αυτό είναι μια πολύ καλή ιδιότητα που θέλουμε να έχουμε στο Game Development γενικά.



7 Εκτέλεση

7.1 Build

Πριν την παρουσίαση του εκτελέσιμου θα αφιερωθεί μία υπο-ενότητα για το build του παιχνιδιού. Η διαδικασία γενικά διαφέρει ανάλογα το λειτουργικό σύστημα, την έκδοση της SFML και τον C++ μεταγλωττιστή.

Ξεκινάμε κάνοντας clone τον πηγαίο κώδικα από το Github

```
$ git clone https://github.com/mataktelis11/Simple-SFML-2D-Game.git  
$ cd Simple-SFML-2D-Game
```

Μεταβαίνουμε στο develop branch γιατί είναι το πιο ενημερωμένο

```
$ git checkout develop
```

Δημιουργούμε έναν φάκελο build

```
$ mkdir build && cd build
```

Μεταγλωτίζουμε με το CMake.

```
$ cmake ..  
$ cmake --build .
```

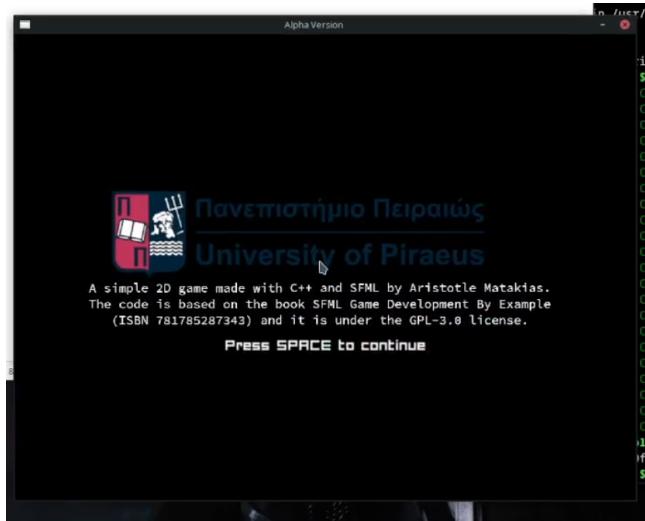
Σημείωση

Στην github σελίδα υπάρχουν περισσότερες οδηγίες και για διαφορετικά OS για την μεταγλώτιση.



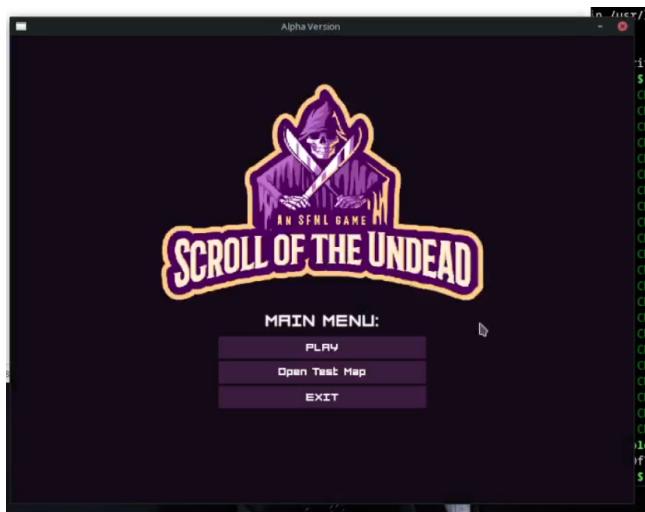
7.2 Gameplay

Ανοίγουμε το εκτελέσιμο και έχουμε την παρακάτω εικόνα.



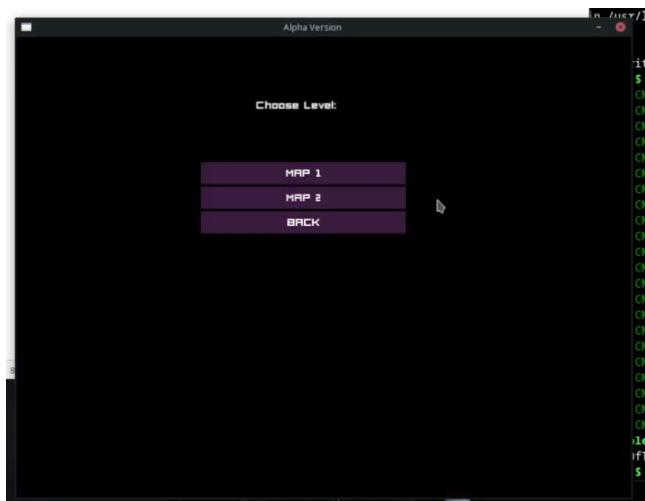
Εικόνα 7

Πατάμε Space για να μεταβούμε στο αρχικό μενού



Εικόνα 8

Επιλέγουμε Play για να μεταβούμε στο μενού επιλογής χάρτη



Εικόνα 9

Controls

Με τα W,A,S,D μετακινείται ο χαρακτήρας, το SPACE μπορεί να πηδήξει και με το Enter επιτίθεται.

Με το Esc μπορούμε να κάνουμε παύση στο παιχνίδι.

Παρακάτω φαίνεται ο χάρτης 1



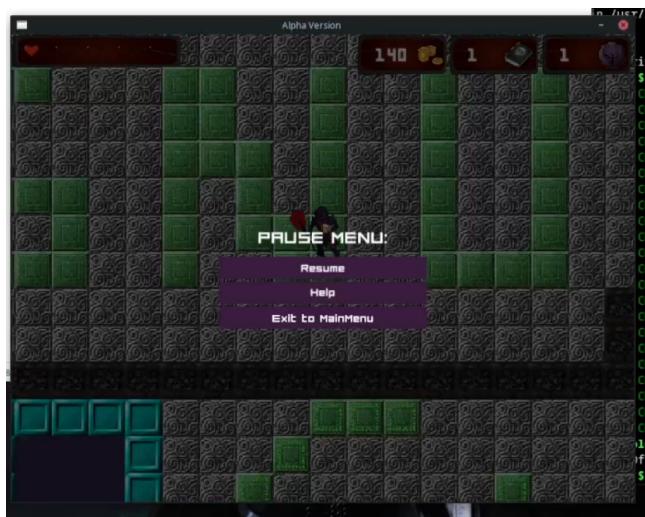
Εικόνα 10

Αν πατήσουμε το πλήκτρο O θα εμφανιστούν τα collision boxes των χαρακτήρων καθώς και τα attack boxes.



Εικόνα 11

Αν πατήσουμε το πλήκτρο Esc θα γίνει παύση και θα εμφανιστεί το Pause menu



Εικόνα 12

Ο παίκτης μπορεί να συλλέγει αντικείμενα αγγίζοντάς τα και το UI ενημερώνεται ανάλογα



Εικόνα 13



8 Συμπεράσματα και παρατηρήσεις

8.1 Κάλυψη κριτηρίων

• Αληθιοφάνεια

Το παιχνίδι είναι 2D και sprite based αλλά έχει υλοποιημένη βαρύτητα, τριβή και επιτάχυνση, κάτι είναι απαραίτητο και αναμενόμενο από τον χρήστη και δίνει στο παιχνίδι μια αληθιοφάνεια.

• Περιεχόμενο

Το περιεχόμενο του παιχνιδιού είναι καθαρά φανταστικό. Δεν υπάρχει κάποιο ορισμένο backstory καθώς η εργασία εστίασε κυρίως στην υλοποίηση και όχι τόσο στην ιστορία του παιχνιδιού.

• Πληρότητα

Χάρης στις καλά ορισμένες δομές το τελικό πρόγραμμα έχει λειτουργικά μενού, εχθρούς και ένα rickup system. Το περιβάλλον των χαρτών είναι ολοκληρωμένο για τα δεδομένα ενός απλού 2D side scroller.

• Σχεδιασμός

Οι χάρτες του παιχνιδιού είναι απλοί σε υλοποίηση και σχεδιασμό αλλά αξιοποιούν ένα αισθητικά ωραίο tileset και μέσω του Tiled Editor μπορούμε εύκολα να φτιάξουμε νέους χάρτες.

• Αισθητική

Σε ένα 2D παιχνίδι η αισθητική εξαρτάται αποκλειστικά από το tileset και τα spritesheets. Όλα τα assets προέρχονται από καλός δημιουργούς και είναι αισθητικά ωραία, δίνοντας το απαραίτητο ενδιαφέρον.

• Πρωτοτυπία

Βασική πρωτοτυπία της εργασίας είναι το περιβάλλον και η έμφαση στο Development. Ο πηγαίος κώδικας είναι εκτενής αλλά διαχειρίσιμος και μπορεί να μελετηθεί σε βάθος, ακόμη και χωρίς μεγάλο υπόβαθρο στην C++.

• Χρηστικότητα

Αξιοποιώντας τις καλές αρχές που έθεσε το βιβλίο στον κώδικα, το UI και ο γενικός χειρισμός του παιχνιδιού είναι λειτουργικός και ανταποκρίνεται άμεσα στον χρήστη κάνοντας την εμπειρία καλύτερη.

• Κίνηση (animation)

Στο συγκεκριμένο παιχνίδι τα animation υπάρχουν υπό την μορφή sprites τα οποία εναλλάσσονται σε καρέ για να δημιουργήσουν το εφέ της κίνησης. Επειδή τα spritesheets που χρησιμοποιήσαμε έχουν πολλά καρέ για κάθε κίνηση πυ μπορεί να κάνει ένας χαρακτήρας, το τελικό αποτέλεσμα είναι μία ομαλή κίνηση παρόλο που είναι ένας δισδιάστατος χαρακτήρας.

• Λειτουργικότητα (functionality)

Το περιβάλλον στο παιχνίδι περιέχει μουσική, ηχητικά εφέ και spritesheet animations. Υπάρχουν επίσης rick up items. Ο χρήστης μετακινείται μέσα στον κόσμο και αλληλεπιδρά με τους εχθρούς έστω σε έναν απλό βαθμό. Οι υποδομές που έχουμε ορίσει σε επίπεδο κώδικα μας εξασφαλίζουν ένα πλήρως λειτουργικό game engine πάνω στο οποίο ένας σχεδιαστής μπορεί να φτιάξει μεγαλύτερους χάρτες και να εισάγει όσους εχθρούς θέλει, χωρίς να ασχοληθεί καθόλου με ζητήματα υλοποίησης.



• Ανάπτυξη

Το περιβάλλον ανάπτυξης είναι πολύ πιο απλό από το Unity, καθώς δεν έχει κάποιο περίπλοκο GUI editor με πολλές επιλογές. Ο κώδικας είναι χωρισμένος σε αρχεία ώστε να μπορεί να διαβαστεί τμηματικά και το πιο σημαντικό: ο κώδικας είναι **ευανάγνωστος**. Μπορεί να μην έχουμε σχόλια για κάθε μία γραμμή (κάτι που συνήθως σημαίνει ότι ο κώδικας δεν είναι σωστός) αλλά αν κάποιος διαβάσει τον κώδικα έχοντας έστω μία μικρή εμπειρία σε C++ και κυρίως σε Object Oriented Programming θα του φανεί εξαιρετικά οικείος.

8.2 Σημεία προς βελτίωση

• Έλλειψη Sound Manager

Οι ήχοι έχουν εισαχθεί στον κώδικα με έναν αρκετά απλό τρόπο. Γενικά δεν υπάρχει κάποιο άμεσο πρόβλημα με αυτή την υλοποίηση αλλά σύμφωνα με τον επίσημο documentation της βιβλιοθήκης SFML δεν μπορούν να υπάρχουν ταυτόχρονα σε ένα πρόγραμμα 256 ήχοι. Το πρόγραμμά μας δεν έχει ορίσει κάποιο όριο στους ήχους. Ενδεχομένως αν σε έναν χάρτη υπάρχουν πάρα πολλοί εχθροί το όριο των ήχων να υπερβεί, καθώς για κάθε εχθρό το πρόγραμμα "φορτώνει" τον ήχο, ακόμη και αν δεν έχει φορτωθεί και έτσι επιβαρύνει το λειτουργικό σύστημα.

Στο βιβλίο **SFML Game Development By Example** υπάρχει μία υλοποίηση ενός Sound Manager, ο οποίος, όπως δηλώνει και το όνομά του, διαχειρίζεται όλους τους ήχους του παιχνιδιού. Μάλιστα ότιδιος ο Sound Manager έχει ορισμένο το όριο των ήχων και διαχειρίζεται πολύ καλά τους ήχους που πρέπει να γίνουν paused όταν το παιχνίδι περνάει από την μία κατάσταση στην άλλη, κάτι που δεν γίνεται στο πρόγραμμά μας.

• Έλλειψη GUI System

Το GUI παρόμοια με τους ήχους, έχει υλοποιηθεί με έναν απλό τρόπο. Τα κουμπιά στα Menu λειτουργούν απλώς ελέγχοντας την θέση του ποντικιού. Στο βιβλίο **SFML Game Development By Example** υπάρχει μία πλήρης υλοποίηση για GUI elements που οποία επεκτείνεται και σε textboxes με scrollbar. Δεν προχωρήσαμε στην υλοποίησή τους ωστόσο γιατί δεν υπήρχε άμεση ανάγκη για πιο εξελιγμένα GUIs και επίσης ο κώδικας είναι λίγο πιο απλός με αυτόν τον τρόπο.

• FPS Limit

Το βασικό gameloop από το οποίο ενημερώνεται όλο το παιχνίδι είναι ένας ατέρμονος βρόχος while που δεν σταματάει, μέχρι ο χρήστης να κλείσει το πρόγραμμα. Αυτό σημαίνει πώς όταν τρέχουμε το πρόγραμμα θα φαίνεται ότι η CPU έχει 100% χρήση. Αυτό γίνεται γιατί με τον ατέρμονο βρόχο ο υπολογιστής προσπαθεί να τρέξει την λογική του παιχνιδιού όσο πιο γρήγορα μπορεί. Αυτό μάλιστα μπορεί να έχει και κακές επιπτώσεις γιατί το παιχνίδι θα τρέχει πιο γρήγορα σε δυνατότερους υπολογιστές. Η SFML προσφέρει δύο λύσεις σε αυτό το πρόβλημα: **setVerticalSyncEnabled** και **setFramerateLimit**. Και οι δύο συναρτήσεις κάνουν το πρόγραμμα να "κοιμάται" ώστε να τρέχει με σταθερά "ticks" η συνάρτηση update του παιχνιδιού. Στο παρόν project έχει χρησιμοποιηθεί η συνάρτηση **setVerticalSyncEnabled** η οποία αξιοποιεί την κάρτα γραφικών. Ωστόσο παρατηρήθηκε ότι επιφέρει ένα bug: ο χαρακτήρας μπορεί να κάνει Jump ενώ βρίσκεται ήδη στον αέρα, κάτι που δεν γίνεται αν το **setVerticalSyncEnabled** δεν ενεργοποιηθεί.

Υπενθυμίζεται πως ο πηγαίος κώδικας του παιχνιδιού είναι στο Github (<https://github.com/mataktelis11/Simple-SFML-2D-Gamer>). Όποιος επιθυμεί να μελετήσει αυτά τα προβλήματα και να βρει λύσεις, μπορεί να κάνει ένα Pull request στο Github repository. Επίσης



αν κατά την εκτέλεση εμφανιστούν νέα προβλήματα, μπορεί οποιοσδήποτε να τα κάνει raise σαν Issues.

8.3 Συμπεράσματα

Η εργασία έγινε με στόχο την δημιουργία μιας απλής αλλά ολοκληρωμένης game engine και ο στόχος αυτός πέτυχε. Αξιοποιώντας τον καλογραμμένο κώδικα του βιβλίου **SFML Game Development By Example** δημιουργήσαμε ένα παιχνίδι του οποίου ο πηγαίος κώδικας μπορεί να μελετηθεί και να επεκταθεί εύκολα, ειδικά αν μελετηθεί και το παρόν έγγραφο.

Δόθηκε ιδιαίτερη σημασία σε όλο το development του παιχνιδιού και κεντρικό ρόλο έπαιξε το source control (github) και η δημοσίευση του κώδικα ως public για να είναι ένα παιχνίδι ανοιχτού κώδικα το οποίο μάλιστα φτιάχτηκε αποκλειστικά με εργαλεία ανοιχτού κώδικα και δωρεάν assets. Έγινε ιδιαίτερη προσπάθεια ώστε ο κώδικας να είναι ευανάγνωστος και επεκτάσιμος, καθώς πάντα θα υπάρχει περιθώριο για βελτίωση και είναι σαφές ότι δεν πρόκειται να είναι απόλυτα πλήρες ένα παιχνίδι όταν φτιάχνεται μόνο από ένα άτομο.

Μία από τις σημαντικές επιτεύξεις της εργασίας είναι η δυναμική φόρτωση των assets (χάρτες και χαρακτήρες) από την μηχανή, χωρίς δηλαδή να χρειάζεται να γίνει recompile ο κώδικας. Αυτό πραγματικά σημαίνει ότι έχουμε ένα game engine που μπορεί να φτιάξει ένα 2D παιχνίδι που έχουμε σχεδιάσει. Απλά δηλώνουμε τα assets και η μηχανή αυτόματα θα δημιουργήσει το παιχνίδι που θέλουμε. Έτσι γίνεται κατανοπτός ο διαχωρισμός ανάμεσα στο **Game Development** και το **Game Design**. Θα μπορούσε κάποιος να επεκτείνει τα assets περισσότερο (να εισάγει περισσότερους εχθρούς και χάρτες) αλλά δεν θα μπορεί να το κάνει εύκολα αν η μηχανή δεν έχει φτιαχτεί για αυτό το σκοπό. Ορισμένα εμπορικά παιχνίδια πάνε ένα βίντα παραπάνω και επιτρέπουν τους σχεδιαστές να περιγράψουν πώς θέλουν να είναι το παιχνίδι σε μία απλή μεταγλώσσα.

Συνίσταται όποιος θέλει να πάρει μία καλή ιδέα για το Game Development με C++ να μελετήσει αυτήν την εργασία και να χρησιμοποιήσει τον κώδικα της. Υπενθυμίζεται ότι όλος ο πηγαίος κώδικας βρίσκεται στο Github και τα pull requests είναι ευπρόσδεκτα.



9 Βιβλιογραφία

9.1 Πηγές

- **C++ Separate Header and Implementation Files**
<http://www.math.uaa.alaska.edu/~afkjm/csce211/handouts/SeparateCompilation.pdf>
- **Guide To Understand C++ Header Files**
<https://www.simplilearn.com/tutorials/cpp-tutorial/guide-to-understand-cpp-header-files>
- **Constructors and member initializer lists**
<https://en.cppreference.com/w/cpp/language/constructor>
- **Makefile Tutorial**
<https://makefiletutorial.com/>
- **Introduction to modern CMake for beginners**
<https://www.internalpointers.com/post/modern-cmake-beginner-introduction>
- **CMake: Building SFML and Game Projects on Linux**
<https://dane-bulat.medium.com/cmake-building-sfml-and-game-projects-on-linux-3947b3ba6e8>

9.2 Assets

- Necromancer sprites.
<https://creativekind.itch.io/necromancer-free>
- Enemy sprites.
<https://luizmelo.itch.io/monsters-creatures-fantasy>
- Tileset .
<https://opengameart.org/content/dungeon-crawl-32x32-tiles>
- Heart sprites.
<https://nicolemariet.itch.io/pixel-heart-animation-32x32-16x16-freebie>
- HUD Sprite.
<https://dandann1.itch.io/hud-buttons>
- Music.
<https://freepd.com>
- Sound effects.
<https://pixabay.com/>

Όλα τα assets βρίσκονται και στην σελίδα Github του project.