ΤΜΗΜΑ ΕΠΙΚΟΙΝΩΝΙΑΣ ΚΑΙ ΨΗΦΙΑΚΩΝ ΜΕΣΩΝ

ΣΧΟΛΗ ΚΟΙΝΩΝΙΚΩΝ ΚΑΙ ΑΝΘΡΩΠΙΣΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
Π.Μ.Σ. «ΑΝΑΠΤΥΞΗ ΨΗΦΙΑΚΩΝ ΠΑΙΧΝΙΔΙΩΝ ΚΑΙ ΠΟΛΥΜΕΣΙΚΩΝ ΕΦΑΡΜΟΓΩΝ»

# ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

## Μηχανή δημιουργίας παιχνιδιών (χωρίς γραφικό περιβάλλον) : Αρχιτεκτονικός σχεδιασμός, Υλοποίηση, Δημιουργία παραδειγμάτων παιχνιδιών.

**Γούλας Κωνσταντίνος**

**Επιβλέπων Καθηγητής :** Μηνάς Δασυγένης, Αναπληρωτής Καθηγητής.

**Μέλη Τριμελούς:**
1) Μπίμπη Σταματία, Αναπληρώτρια Καθηγήτρια
2) Πλόσκας Νικόλαος, Αναπληρωτής Καθηγητής

Ιούνιος, 2026

COMMUNICATION AND DIGITAL MEDIA DEPARTMENT

SCHOOL OF SOCIAL SCIENCES AND HUMANITIES
MSc PROGRAM «GAMING AND MULTIMEDIA APPLICATION DEVELOPMENT »

# MSc Thesis

## A Code First Game Engine: Architecture, Implementation, and Case Study Games.

**Goulas Konstantinos**

**Supervising Professor:** Minas Dasygenis, Associate Professor.

**3-Member Examination Committee:**
1) Bibi Stamatia, Associate Professor
2) Ploskas Nikolaos, Associate Professor

June, 2026

# ΠΕΡΙΛΗΨΗ

Η παρούσα εργασία εξετάζει τον σχεδιασμό και την υλοποίηση μιας μηχανής παιχνιδιών τύπου «code-first» (χωρίς γραφικό περιβάλλον), γραμμένης σε C++, με χρήση ενός ελάχιστου συνόλου βασικών βιβλιοθηκών (OpenGL, GLFW και GLM). Πρωταρχικός στόχος είναι η εμβάθυνση στην κατανόηση της εσωτερικής αρχιτεκτονικής των μηχανών παιχνιδιών μέσω της κατασκευής μιας μηχανής από το μηδέν και της ανάλυσης των επιμέρους υποσυστημάτων και αρχιτεκτονικών προτύπων που τη συνθέτουν.

Η έρευνα ξεκινά με μια βιβλιογραφική ανάλυση των ορισμών των μηχανών παιχνιδιών και των βασικών υποσυστημάτων τους (rendering, resource management, scene representation, physics, input, audio και tooling), αναδεικνύοντας τις κυριότερες αρχιτεκτονικές επιλογές που συναντώνται στη σχετική βιβλιογραφία. Με βάση το θεωρητικό αυτό πλαίσιο, η εργασία παρουσιάζει στη συνέχεια ολόκληρη τη διαδικασία σχεδιασμού και υλοποίησης της μηχανής, αρχικά σε θεωρητικό και έπειτα σε πρακτικό επίπεδο, εντοπίζοντας και καταγράφοντας συστηματικά τις προκλήσεις που προέκυψαν, καθώς και τις λύσεις που επιλέχθηκαν για την αντιμετώπισή τους, με γνώμονα τη σχετική βιβλιογραφία.

Για την αξιολόγηση της μηχανής αναπτύσσονται διάφορα μικρής κλίμακας παιχνίδια, ώστε να αξιοποιηθούν και να δοκιμαστούν όσο το δυνατόν περισσότερα από τα χαρακτηριστικά της. Η αξιολόγηση επικεντρώνεται στην απόδοση, τη συντηρησιμότητα και την ευκολία ανάπτυξης παιχνιδιών. Επιπλέον, αναπτύσσεται ένα μικρό παιχνίδι εξ ολοκλήρου με τη χρήση τεχνητής νοημοσύνης, χωρίς άμεση παρέμβαση στον παραγόμενο κώδικα, αξιοποιώντας τρία διαφορετικά μοντέλα. Με τον τρόπο αυτό αξιολογούνται περαιτέρω τόσο η χρηστικότητα και το επίπεδο αφαίρεσης που παρέχει η μηχανή όσο και η ικανότητα των μοντέλων να προσαρμόζονται σε ένα άγνωστο προγραμματιστικό περιβάλλον.

Η εργασία ολοκληρώνεται με την παρουσίαση των συμπερασμάτων, τα οποία αφορούν τόσο τη σχέση μεταξύ των σχεδιαστικών επιλογών και των αποτελεσμάτων που προέκυψαν όσο και την καταγραφή των βασικών δυσκολιών και εμποδίων που αντιμετωπίστηκαν κατά τη διαδικασία ανάπτυξης. Στόχος είναι η εργασία να αποτελέσει έναν πρακτικό οδηγό για όσους επιθυμούν να κατανοήσουν τόσο τη λειτουργία όσο και την υλοποίηση μιας μηχανής παιχνιδιών, ανεξάρτητα από τη γλώσσα προγραμματισμού ή τις βιβλιοθήκες που χρησιμοποιούνται.

**Λέξεις-κλειδιά:** μηχανή παιχνιδίων, βιντεοπαιχνίδια, αρχιτεκτονική μηχανής παιχνιδιών

# ABSTRACT

This thesis examines the design and implementation of a code-first game engine (without a graphical editor), developed in C++ using a minimal set of core libraries, namely OpenGL, GLFW, and GLM. The primary objective is to achieve a deeper understanding of the internal architecture of game engines through the development of an engine from scratch and the analysis of the subsystems and architectural patterns that comprise it.

The study begins with a literature review of game engine definitions and their core subsystems, including rendering, resource management, scene representation, physics, input handling, audio, and development tools. Particular emphasis is placed on the architectural approaches and design decisions commonly adopted in modern game engines. Based on this theoretical foundation, the thesis subsequently presents the complete design and development process of the engine, initially from a conceptual perspective and later through practical implementation. Throughout this process, the challenges encountered are systematically identified, documented, and analyzed, while the selected solutions are justified with reference to the relevant literature.

To evaluate the capabilities of the engine, several small-scale games are developed, enabling a broad range of its features to be utilized and tested in practice. The evaluation focuses on performance, maintainability, and ease of game development. Furthermore, a small game was developed entirely by AI-generated code, without any manual intervention in the implementation process, using three different language models. This experiment enabled a further assessment of both the game engine itself and the capability of modern AI models to understand and operate within a previously unseen development environment.

The thesis concludes with a discussion of the results, examining the relationship between the design decisions adopted and the outcomes achieved, while also documenting the major challenges and obstacles encountered during development. The aim of this work is to serve as a practical guide for individuals seeking to understand both the operation and implementation of a game engine, regardless of the programming language or libraries employed.

**Keywords:** Game Engine, Video Games, Game Engine Architecture

# Περιεχόμενα

# ΚΕΦΑΛΑΙΟ 1 Εισαγωγή

## 1.1 Κίνητρο

Η κατασκευή βιντεοπαιχνιδιών αποτελεί από μόνη της ένα πολύ ενδιαφέρον κομμάτι μελέτης, με πολλά παρακλάδια σε διάφορους και διαφορετικούς τομείς. Ένας από τους σημαντικότερους άξονες μελέτης είναι ο προγραμματιστικός τομέας. Κίνητρο, λοιπόν, για την παρούσα εργασία αποτελεί η ανάγκη για καλύτερη κατανόηση όλων εκείνων των συστημάτων και υποσυστημάτων που απαιτούνται για να δημιουργηθεί ένα παιχνίδι. Η κατανόηση των συστημάτων αυτών προϋποθέτει τη μελέτη της ίδιας της μηχανής παιχνιδιών. Μηχανές όπως η Unity[1] και η Unreal Engine[2] παρέχουν ολοκληρωμένα εργαλεία για την ανάπτυξη 2D και 3D παιχνιδιών, προσφέροντας γραφικά περιβάλλοντα, ενσωματωμένα συστήματα φυσικής και scripting. Παράλληλα, μηχανές ανοιχτού κώδικα, όπως οι Godot[3] και Ogre3D[4], προσφέρουν μεγαλύτερη ελευθερία παραμετροποίησης και πρόσβαση στον πηγαίο κώδικα. Ωστόσο, ο στόχος αυτών των μηχανών είναι η ευκολία χρήσης για την κατασκευή παιχνιδιών, γεγονός που αυξάνει την πολυπλοκότητά τους. Η εκμάθηση και η κατανόηση όλων των επιμέρους λειτουργικών ενοτήτων της μηχανής είναι αρκετά δυσκολότερη. Σε αντίθεση με αυτές, η παρούσα εργασία επικεντρώνεται στη δημιουργία μιας, "ελαφριάς" μηχανής, δίνοντας βάρος στην εύκολη κατανόηση και διαχείριση των επιμέρους συστημάτων. Για τον λόγο αυτό, δόθηκε προτεραιότητα στην αναγνωσιμότητα και τη δομική σαφήνεια του κώδικα έναντι της μέγιστης απόδοσης, με στόχο τη βαθύτερη κατανόηση των θεμελιωδών αρχών που διέπουν τη λειτουργία μιας μηχανής παιχνιδιών.

## 1.2 Μηχανές παιχνιδιών

Στον χώρο της ανάπτυξης ψηφιακών παιχνιδιών, κεντρικό ρόλο διαδραματίζουν οι μηχανές παιχνιδιών (game engines). Μια μηχανή παιχνιδιών αποτελεί ένα σύνολο λογισμικού που παρέχει τα βασικά εργαλεία και τις υποδομές για την ανάπτυξη και εκτέλεση ενός παιχνιδιού. Μέσω αυτής, οι προγραμματιστές και οι σχεδιαστές μπορούν να δημιουργούν και να διαχειρίζονται τα

[1] https://unity.com/
[2] https://www.unrealengine.com/
[3] https://godotengine.org/
[4] https://www.ogre3d.org/

στοιχεία ενός παιχνιδιού, όπως τα γραφικά, την αλληλεπίδραση με τον χρήστη και τη συνολική δομή της εφαρμογής, χωρίς να χρειάζεται να υλοποιούν κάθε λειτουργία από την αρχή.

Στη σύγχρονη βιομηχανία ανάπτυξης παιχνιδιών χρησιμοποιούνται πολυάριθμες μηχανές παιχνιδιών. Ενδεικτικά παραδείγματα αποτελούν οι Unity, Unreal Engine και Godot, οι οποίες χρησιμοποιούνται για την ανάπτυξη παιχνιδιών σε πολλές διαφορετικές πλατφόρμες. Οι μηχανές αυτές παρέχουν ένα ολοκληρωμένο περιβάλλον ανάπτυξης που διευκολύνει τη διαδικασία δημιουργίας παιχνιδιών και επιτρέπει σε ομάδες ανάπτυξης να επικεντρωθούν περισσότερο στον σχεδιασμό και την εμπειρία του παιχνιδιού.

## 1.3 Μηχανές παιχνιδίων χωρίς γραφικό περιβάλλον

Μια ιδιαίτερη κατηγορία μηχανών παιχνιδιών είναι οι λεγόμενες headless ή code-first μηχανές, οι οποίες δεν διαθέτουν γραφικό περιβάλλον επεξεργασίας (GUI Editor) για τη δημιουργία και διαχείριση περιεχομένου. Σε αυτές τις υλοποιήσεις, η διαμόρφωση των σκηνών, των αντικειμένων και των επιμέρους συστημάτων πραγματοποιείται κυρίως μέσω κώδικα. Παρότι η πλειονότητα των σύγχρονων μηχανών παιχνιδιών συνοδεύεται από ολοκληρωμένα γραφικά περιβάλλοντα ανάπτυξης, υπάρχουν εργαλεία και frameworks, όπως τα MonoGame[5], libGDX[6] και Raylib[7], τα οποία υιοθετούν μια περισσότερο code-first φιλοσοφία. Η προσέγγιση αυτή οδηγεί συνήθως σε ελαφρύτερα και λιγότερο πολύπλοκα συστήματα, καθώς δεν απαιτείται η ανάπτυξη και συντήρηση ενός πλήρους γραφικού περιβάλλοντος. Παράλληλα, προσφέρει μεγαλύτερο έλεγχο στον προγραμματιστή και διευκολύνει τη μελέτη της εσωτερικής αρχιτεκτονικής και των αλληλεπιδράσεων μεταξύ των υποσυστημάτων της μηχανής. Για τους λόγους αυτούς, οι headless μηχανές χρησιμοποιούνται συχνά σε εκπαιδευτικά, ερευνητικά και πειραματικά έργα, όπου η κατανόηση των θεμελιωδών αρχών λειτουργίας μιας μηχανής παιχνιδιών είναι σημαντικότερη από την παροχή εργαλείων ταχείας ανάπτυξης περιεχομένου.

Αξίζει να σημειωθεί ότι αρκετές σύγχρονες μηχανές παιχνιδιών, όπως οι Unity, Unreal Engine και Godot, παρέχουν δυνατότητες λειτουργίας σε headless mode, κυρίως για σκοπούς αυτοματοποίησης και δοκιμών. Ωστόσο, οι μηχανές αυτές εξακολουθούν να βασίζονται σε ολοκληρωμένα γραφικά περιβάλλοντα ανάπτυξης και δεν θεωρούνται εγγενώς code-first ή headless συστήματα.

[5] https://monogame.net/
[6] https://libgdx.com/
[7] https://www.raylib.com/

## 1.4 Ιστορία των βιντεοπαιχνιδιών

Ιστορικά, μετά την πρωτοεμφάνιση των προσωπικών υπολογιστών (PC), ξεκίνησε και η βιομηχανία βίντεο-παιχνιδιών [10]. Ξεκινάμε προς τα τέλη της δεκαετίας του 1950 με τα πρώτα "πειραματικά" παιχνίδια, "Tennis for Two", 1958 από τον William Higinbotham (US Dept. of Energy) και το "Spacewar!" (1962) [16], παιχνίδια που βασίζονταν αρχικά στην ατομική δημιουργικότητα, και χωρίς την ύπαρξη τυποποιημένων εργαλείων. Η δημιουργία ενός βιντεοπαιχνιδιού, απαιτούσε την κατασκευή όλων των συστημάτων και υποσυστημάτων του "από το μηδέν", από τον τρόπο διαχείρισης του ήχου και των γραφικών, μέχρι τη λογική του παιχνιδιού. Η απουσία της πολυπλοκότητας (που υπάρχει σήμερα στην κατασκευή βιντεοπαιχνιδιών), επέτρεπε σε μεμονωμένους προγραμματιστές ή μικρές ομάδες να ολοκληρώνουν ένα έργο χωρίς την ανάγκη χρήσης εξωτερικών εργαλείων.

Με την έλευση της δεκαετίας του 1970 τα βιντεοπαιχνίδια απέκτησαν μια πιο εμπορική εξάπλωση. Τίτλοι όπως το "Pong" (1972) και το "Space Invaders" (1978) έγιναν ιδιαίτερα γνωστοί αποκτώντας μεγάλη απήχηση και εισήλθαν σταδιακά στα σπίτια μέσω των πρώτων κονσόλων και προσωπικών υπολογιστών.

Κατά τη δεκαετία του 1980, η τεχνολογική πρόοδος σε επίπεδο υλικού (επεξεργαστές, γραφικά, μνήμη) οδήγησε σε εκθετική αύξηση της πολυπλοκότητας των παιχνιδιών [3]. Η δημιουργία ενός τίτλου πλέον απαιτούσε πολλαπλές ειδικότητες και συνέργεια ανάμεσα σε προγραμματιστές, καλλιτέχνες και σχεδιαστές. Σύμφωνα με τον Fabio Petrilo [10], η περίοδος αυτή σηματοδότησε μια μετάβαση από την "τεχνική κουλτούρα" στην "πολιτισμική κουλτούρα" των παιχνιδιών. Τα παιχνίδια πλέον για το κοινό δεν αποτελούσαν απλώς τεχνολογικά προϊόντα αλλά απέκτησαν και μια πιο "καλλιτεχνική" φύση, αντιμετωπίζονταν ως μέσα αφήγησης και έκφρασης.

Στη δεκαετία του 1990, τα βιντεοπαιχνίδια γνωρίζουν επιπλέον πρόοδο. Εδραιώνονται για πρώτη φορά τρισδιάστατα παιχνίδια. Σύμφωνα με το άρθρο [6], Ένα από τα πρώτα τρισδιάστατα παιχνίδια ήταν το "Maze War" (1973), ένα πρωτοποριακό παιχνίδι που επέτρεπε στον παίκτη να κινηθεί ελεύθερα σε ένα τρισδιάστατο περιβάλλον, εισάγοντας για πρώτη φορά την έννοια της "προοπτικής πρώτου προσώπου". Κατά τη δεκαετία του 1990, η ραγδαία εξέλιξη του υλικού (hardware) και η εμφάνιση των επιταχυντών γραφικών (3D accelerators) έδωσαν τη δυνατότητα στους δημιουργούς να αυξήσουν θεαματικά τόσο την ποιότητα των γραφικών όσο και την πολυπλοκότητα της φυσικής αλληλεπίδρασης. Τίτλοι όπως το Super Mario 64 (1996) σηματοδότησαν μια νέα εποχή για το gameplay, καθώς έδειξαν για πρώτη φορά τι μπορούσε να προσφέρει ο πλήρης τρισδιάστατος χώρος στον παίκτη, ελευθερία κινήσεων, ανοιχτά επίπεδα, και πιο ρεαλιστική αίσθηση του κόσμου.

## 1.5 Η ανάγκη για μηχανές παιχνιδιών

Η ανάγκη για επαναχρησιμοποίηση λογισμικού είχε αρχίσει να εμφανίζεται ήδη πριν την καθιέρωση των σύγχρονων game engines. Χαρακτηριστικό παράδειγμα αποτελεί το SCUMM (Script Creation Utility for Maniac Mansion), το οποίο αναπτύχθηκε από τη Lucasfilm Games το 1987 για την ανάπτυξη παιχνιδιών περιπέτειας. Το σύστημα αυτό επέτρεπε τον διαχωρισμό της λογικής και του περιεχομένου του παιχνιδιού από τον χαμηλού επιπέδου κώδικα, επιτρέποντας την ανάπτυξη πολλαπλών τίτλων πάνω στην ίδια τεχνολογική βάση. Η προσέγγιση αυτή μείωσε σημαντικά τον χρόνο ανάπτυξης και ανέδειξε τα οφέλη της επαναχρησιμοποίησης λογισμικού, προαναγγέλλοντας πολλές από τις αρχές που υιοθετήθηκαν αργότερα από τις σύγχρονες μηχανές παιχνιδιών.

Με την εξέλιξη των βιντεοπαιχνιδιών και ειδικότερα με τη μετάβαση από τα απλά δισδιάστατα (2D) περιβάλλοντα στα τρισδιάστατα (3D), οι απαιτήσεις σε επίπεδο τεχνολογίας και υπολογιστικής πολυπλοκότητας αυξήθηκαν. Όπως περιγράφει το βιβλίο [6], η ανάπτυξη των 3D παιχνιδιών απαίτησε τη διαχείριση χιλιάδων πολυγώνων, προηγμένων φωτισμών, φυσικής προσομοίωσης και ρεαλιστικής απεικόνισης του χώρου. Έτσι, ο προγραμματισμός ενός παιχνιδιού "από το μηδέν" όπως γινόταν στις πρώτες δεκαετίες της βιομηχανίας έγινε πλέον σχεδόν ανέφικτος για μεμονωμένους δημιουργούς ή μικρές ομάδες.

Αυτό οδήγησε στην ανάγκη δημιουργίας επαναχρησιμοποιήσιμων λογισμικών υποδομών, γνωστών ως μηχανές παιχνιδιών (game engines). Μια μηχανή παιχνιδιών λειτουργεί ως ενδιάμεσο στρώμα (middleware) μεταξύ του προγραμματιστή και του υλικού (hardware), παρέχοντας ένα σύνολο έτοιμων εργαλείων, βιβλιοθηκών και δομών που απλοποιούν τη διαδικασία ανάπτυξης . Αντί κάθε προγραμματιστής να γράφει εξ αρχής τον δικό του κώδικα για το rendering, τον ήχο, τη φυσική ή τη διαχείριση των πόρων, οι μηχανές προσφέρουν αυτά τα υποσυστήματα σε ενιαίο πλαίσιο.

Σύμφωνα με τον Jason Gregory [4], η δημιουργία των πρώτων εμπορικών game engines στα μέσα της δεκαετίας του 1990 (όπως η id Tech 1 του Doom, 1993, και η Unreal Engine, 1998) σηματοδότησε μια νέα εποχή "λογισμικής βιομηχανοποίησης" των παιχνιδιών. Πλέον, τα στούντιο μπορούσαν να διαχωρίσουν την "τεχνολογική βάση" από το "περιεχόμενο", επιτρέποντας επαναχρησιμοποίηση, ευκολότερες ενημερώσεις και γρηγορότερους κύκλους ανάπτυξης. Οι μηχανές παιχνιδιών μετατράπηκαν έτσι σε κεντρικό άξονα παραγωγής και σε βασικό εργαλείο κατανόησης του πώς λειτουργεί ένα παιχνίδι εσωτερικά. Η εξέλιξη αυτή κατέστησε τις μηχανές παιχνιδιών θεμελιώδες στοιχείο της σύγχρονης ανάπτυξης βιντεοπαιχνιδιών, καθιστώντας απαραίτητη τη μελέτη της αρχιτεκτονικής και των επιμέρους υποσυστημάτων τους.

## 1.6 Δομή της εργασίας

Στη συνέχεια της παρούσας εργασίας, παρουσιάζεται αρχικά στο κεφάλαιο δύο το θεωρητικό υπόβαθρο της μηχανής, καθώς και οι ορισμοί και οι περιγραφές των επιμέρους στοιχείων που τη συνθέτουν. Έπειτα, στο κεφάλαιο τρία αναλύονται οι στόχοι της εργασίας και τα εργαλεία που χρησιμοποιήθηκαν για την ανάπτυξη της μηχανής. Ακολουθεί το τέταρτο κεφάλαιο, στο οποίο περιγράφεται η διαδικασία και τα στάδια σχεδιασμού και υλοποίησης της μηχανής. Στο πέμπτο κεφάλαιο παρουσιάζονται ενδεικτικά παιχνίδια που αναπτύχθηκαν με τη χρήση της. Τέλος, παρατίθεται το κεφάλαιο έξι των συμπερασμάτων, όπου συνοψίζονται τα βασικά ευρήματα και μελοντικές επεκτάσεις.

# ΚΕΦΑΛΑΙΟ 2 Θεωρητικό υπόβαθρο και ορισμοί

Σε αυτή την ενότητα θα δοθεί μια θεωρητική ανάλυση γύρω από τις μηχανές παιχνιδιών. Θα παρουσιαστούν όλοι οι απαραίτητοι ορισμοί και αναλύσεις, έτσι ώστε να γίνουν κατανοητά και διακριτά όλα τα βασικά συστήματα και υποσυστήματα μιας μηχανής. Σκοπός είναι η κατανόησή τους σε θεωρητικό επίπεδο, ώστε στη συνέχεια να προχωρήσουμε στην υλοποίηση, έχοντας τις κατάλληλες βάσεις. Όλα τα παραπάνω θα παρουσιαστούν με αναφορά σε σχετικές εργασίες.

## 2.1 Τι είναι μια μηχανή παιχνιδιών;

Ένας απλός ορισμός της μηχανής παιχνιδιών είναι ότι πρόκειται για λογισμικό το οποίο χρησιμοποιείται για τον σχεδιασμό και την ανάπτυξη παιχνιδιών και πολυμεσικών εφαρμογών. Η παραπάνω πρόταση δίνει μια πρώτη εικόνα κατανόησης της έννοιας αλλά δεν μπορεί να αποτελέσει καλό ορισμό για μια βαθύτερη κατανόηση τόσο του τι είναι αλλά και των λειτουργιών που πρέπει να έχει ένα λογισμικό για να χαρακτηριστεί ως μηχανή παιχνιδιών. Στη συνέχεια η έννοια προσεγγίζεται από διαφορετικές οπτικές ώστε να καταλήξουμε σε έναν ενιαίο και πλήρη για τα πλαίσια της εργασίας αυτής ορισμό.

Μια μηχανή παιχνιδιών (game engine) μπορεί να οριστεί ως ένα επαναχρησιμοποιήσιμο λογισμικό πλαίσιο (framework), το οποίο παρέχει τις βασικές λειτουργίες που απαιτούνται για την ανάπτυξη ψηφιακών διαδραστικών παιχνιδιών. Σε αντίθεση με τον κώδικα που γράφεται αποκλειστικά για ένα συγκεκριμένο παιχνίδι, η μηχανή προσφέρει ένα σύνολο υποσυστημάτων που μπορούν να χρησιμοποιηθούν σε διαφορετικούς τίτλους, μειώνοντας τον χρόνο ανάπτυξης και ενισχύοντας την επαναχρησιμοποίηση. Ο ([4]) στο βιβλίο του, τονίζει ότι μια μηχανή παιχνιδιών διαχωρίζεται σε δύο βασικά μέρη: το runtime, που περιλαμβάνει τον βρόχο εκτέλεσης (game loop), την απόδοση γραφικών και τη φυσική (rendering, physics engine), και το "pipeline" εργαλείων, δηλαδή editors, asset compilers και debuggers, που διευκολύνουν την ανάπτυξη αλλά δεν διανέμονται απαραίτητα με το τελικό προϊόν. Με τον ορισμό αυτό δίνει μια διαφορετική οπτική στο τι είναι μηχανή παιχνιδιών.

Στην εργασία ([11]) εξετάζει τις μηχανές παιχνιδιών υπό το πρίσμα των λογισμικών ως (software frameworks) και καταλήγει ότι, σε αντίθεση με τα παραδοσιακά frameworks, οι μηχανές χαρακτηρίζονται από υψηλό βαθμό σύζευξης (coupling) ανάμεσα στα υποσυστήματα και από την ανάγκη για απόδοση πραγματικού χρόνου και ντετερμινιστική συμπεριφορά.

Σύμφωνα με ([2]), ο βασικός στόχος μιας μηχανής παιχνιδιών είναι, να παρέχει με έναν πιο αφαιρετικό (γενικευμένο) τρόπο όλα τα βασικά χαρακτηριστικά που συνήθως συναντάμε σε

ένα βίντεο-παιχνίδι και που απαιτούνται για την παραγωγή αυτού. Για τον σκοπό αυτό δίνει μια σειρά από βασικές λειτουργικότητες, που πιστεύει ότι είναι απαραίτητες για μια μηχανή παιχνιδιών:

- Μια μηχανή για την παραγωγή 2, 3 διαστάσεων γραφικών (rendering engine for 2d 3d graphics)
- Μηχανισμό διαχείρισης δεδομένων εισόδου από πληκτρολόγιο, ποντίκι κλπ. (input handling)
- Την βασική δομή επανάληψης του παιχνιδιού με τις βασικές της ρουτίνες (game loop)
- Μηχανή διαχείρισης της Φυσικής (συγκρούσεις αντικειμένων κλπ.) (Physics engine)
- Διαχείριση του ήχου (Sound)
- Διαχείριση των κινούμενων γραφικών (Animation for 2d and 3d)
- Δυνατότητες παραλληλισμού με πολλαπλά νήματα (Process threading)

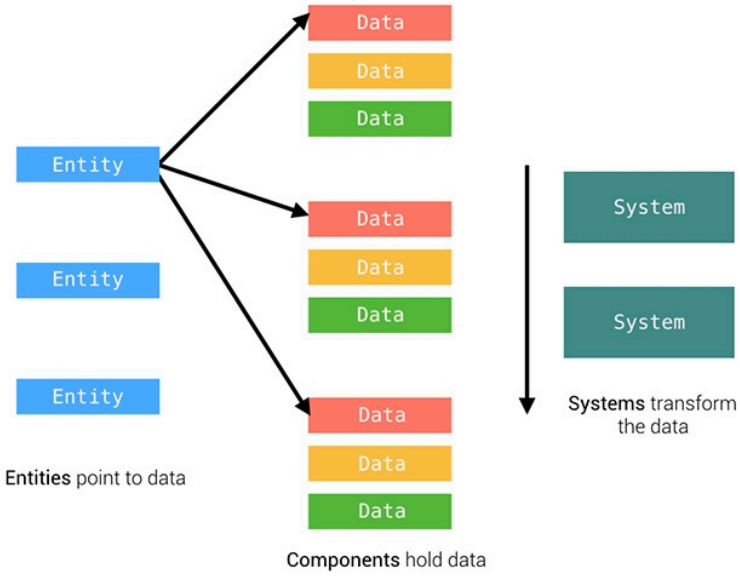
Ενώ ως επιπλέον αναφέρονται και τα παρακάτω:

- Δυνατότητα scripting
- Τεχνητή νοημοσύνη
- Ύπαρξη δικτύου
- Υποστήριξη για πολλαπλές πλατφόρμες
- Τοπική υποστήριξη (Localization support)

Παρόμοια συμπεράσματα αντλούνται και από τη δουλειά ([15]) με τη μελέτη τους για την ανάκτηση αρχιτεκτονικής μιας μηχανής παιχνιδιών δείχνουν ότι οι πιο κοινές ενότητες που συναντώνται σε όλες τις μηχανές είναι το επίπεδο πλατφόρμας, η διαχείριση πόρων και το rendering, επιβεβαιώνοντας ότι υπάρχει μια σχετική «συναίνεση» γύρω από τα υποσυστήματα που θεωρούνται θεμελιώδη.

Στο βιβλίο ([13]) δίνει ένα ακόμα χαρακτηριστικό στις μηχανές παιχνιδιών ορίζοντάς τες ως λογισμικά που αποτελούνται από διάφορα, διαφορετικά εργαλεία (κομμάτια προγράμματος) βοηθητικά για τον δημιουργό παιχνιδιών που όμως κρύβουν τις λεπτομέρειες χαμηλού επιπέδου (low level) των λειτουργιών και των υλοποιήσεων από αυτόν.

Τέλος, ένα σημαντικό στοιχείο για την σωστή κατανόηση του ορισμού θα πρέπει να αφορά και τα αρχιτεκτονικά πρότυπα και επιλογές που υιοθετούνται ([9]). Οι παραδοσιακές μηχανές χρησιμοποιούσαν κυρίως ιεραρχικά scene graphs. Οι πιο σύγχρονες, όμως, προσεγγίσεις υιοθετούν αρχιτεκτονικές όπως η Entity–Component–System (ECS) Σχήμα 2.1, ([12]) καθώς προσφέρει καλύτερη απόδοση, χαμηλότερη σύζευξη και μεγαλύτερη ευελιξία, και θα αναλυθεί σε μεγαλύτερο βάθος σε επόμενο κεφάλαιο.




Σχήμα 2.1: Διάγραμμα Entity Component System

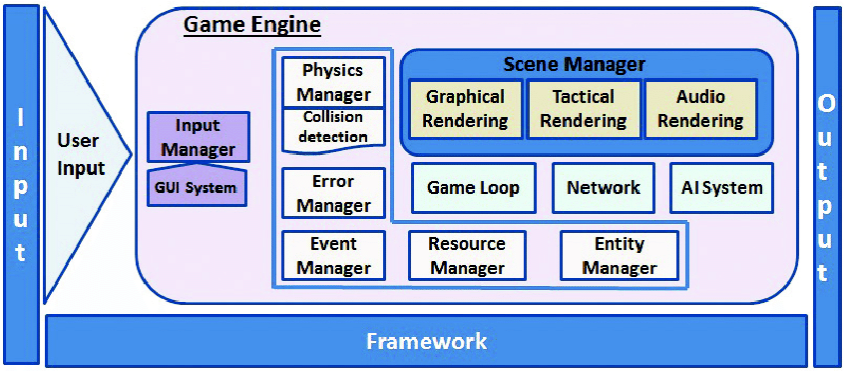
Συνοψίζοντας λοιπόν, μια μηχανή παιχνιδιών είναι ένα επαναχρησιμοποιούμενο λογισμικό (framework) με υψηλό βαθμό σύζευξης (coupling) ανάμεσα στα υποσυστήματα και από την ανάγκη για πραγματικού χρόνου απόδοση και ντετερμινιστική συμπεριφορά, αποτελείται από διάφορα, διαφορετικά εργαλεία (κομμάτια προγράμματος) βοηθητικά για τον δημιουργό παιχνιδιών που όμως κρύβουν τις λεπτομέρειες χαμηλού επίπεδου (low level) των λειτουργιών και των υλοποιήσεων. Τα βασικότερα από αυτά τα κομμάτια, τα οποία φαίνεται να περιλαμβάνονταί στις περισσότερες μηχανές είναι : Μια μηχανή για την παραγωγή 2, 3 διαστάσεων γραφικών (rendering engine for 2d 3d graphics), Μηχανισμό διαχείρισης δεδομένων εισόδου από πληκτρολόγιο, ποντίκι κλπ (input handling), Τη βασική δομή επανάληψης του παιχνιδιού με τις βασικές της ρουτίνες (game loop), μηχανή διαχείρισης της Φυσικής (συγκρούσεις αντικειμένων κλπ) (Physics engine), διαχείριση του ήχου (Sound), διαχείριση των κινούμενων γραφικών (Animation for 2d and 3d), δυνατότητες παραλληλισμού με πολλαπλά νήματα (Process threading).

Τέλος, να αναφερθούμε και πιο ειδικά για της μηχανές παιχνιδιών (code-first) ή (no GUI). Μια no GUI μηχανή παιχνιδιών είναι ένα σύστημα λογισμικού που παρέχει όλες τις βασικές δυνατότητες μιας μηχανής παιχνιδιών (όπως ανέλυσα παραπάνω) — όπως pipeline περιουσιακών στοιχείων, βρόχο παιχνιδιού, σύστημα απόδοσης και διαχείριση υποσυστημάτων — αλλά δεν διαθέτει γραφική διεπαφή (όπως οπτικό πρόγραμμα επεξεργασίας σκηνών, γράφημα κόμβων ή παράθυρο επιθεώρησης).

## 2.2 Κύριες υποδομές και υποσυστήματα μιας μηχανής παιχνιδιών

Παρά το γεγονός ότι οι μηχανές παιχνιδιών μπορεί να διαφέρουν σημαντικά ως προς τα εργαλεία και τις δυνατότητές τους (π.χ. εμπορικές μηχανές όπως Unity, Unreal, Godot ή custom lightweight μηχανές), η βιβλιογραφία δείχνει ότι όλες στηρίζονται σε έναν κοινό πυρήνα υποσυστημάτων. Όπως περιγράφηκε, και στο προηγούμενο κεφάλαιο, κάθε μηχανή παιχνιδιών έχει έναν βασικό πυρήνα υποδομών: **rendering engine, input, event handling, game-loop, physic engine, sound handling, 2d-3d animation** και **threading** και ένα βασικό πυρήνα υποσυστημάτων: το επίπεδο πλατφόρμας, το σύστημα **resource managment** (Διαχείριση Πόρων), και ένα **math library (Μαθηματική βιβλιοθήκη)**. Στα παρακάτω κεφάλαια της θεωρίας θα ασχοληθούμε με την ανάλυση και κατανόηση όλων αυτών των υποδομών και των υποσυστημάτων ένα ένα ξεκινώντας από τα υποσυστήματα.

Στην εργασία του, ([17]) παρουσιάζει τα βασικά, κατά τον ίδιο, συστήματα μιας μηχανής παιχνιδιών, Σχήμα 2.2 αναφέροντας πως δεν υπάρχει στη βιβλιογραφία κάποια πιο συγκεκριμένη ταξινόμηση.




Σχήμα 2.2: Αρχιτεκτονική Μηχανής Παιχνιδιών

### *2.2.1 Επίπεδο πλατφόρμας (Platform Layer)*

Το επίπεδο πλατφόρμας είναι το χαμηλότερο στρώμα μιας μηχανής παιχνιδιών. Αποτελεί τη «γέφυρα» ανάμεσα στη μηχανή και το λειτουργικό σύστημα, παρέχοντας ένα αφαιρετικό API που επιτρέπει στα ανώτερα υποσυστήματα να λειτουργούν χωρίς να χρειάζεται να γνωρίζουν τις λεπτομέρειες κάθε πλατφόρμας.

Σύμφωνα με ([4]), το επίπεδο πλατφόρμας περιλαμβάνει τις εξής βασικές λειτουργίες:

- Δημιουργία παραθύρου και context γραφικών (OpenGL, Vulkan, DirectX).
- Χειρισμό γεγονότων (events) και είσοδο από πληκτρολόγιο, ποντίκι, gamepads.
- Πρόσβαση σε χρονόμετρα υψηλής ακρίβειας (high resolution timers).
- Διαχείριση αρχείων (file I/O).

- Διαχείριση νημάτων (threads) και συγχρονισμού, εφόσον απαιτείται.

Στην πράξη, οι περισσότερες σύγχρονες μηχανές ή custom projects βασίζονται σε εξωτερικές βιβλιοθήκες για την αφαίρεση αυτών των λεπτομερειών. Ενδεικτικά:

- GLFW: δημιουργία παραθύρου, χειρισμός εισόδου και events.
- SDL2: εναλλακτική λύση με περισσότερες δυνατότητες (ήχος, δικτύωση).
- GLAD / GLEW: φόρτωση επεκτάσεων OpenGL.

Για μια «code-first» μηχανή σε C++, η χρήση της GLFW σε συνδυασμό με GLAD θεωρείται η πιο ελαφριά και καθαρή λύση, καθώς επιτρέπει: τη δημιουργία cross-platform παραθύρου και context, τον χειρισμό πληκτρολογίου/ποντικιού/χειριστηρίου μέσω ενιαίου API, τη βασική διαχείριση χρόνου (time queries). Έτσι, το Platform Layer επίπεδο, ουσιαστικά, παρέχει τις βάσεις πάνω στις οποίες θα χτιστούν τα υπόλοιπα υποσυστήματα (π.χ. rendering, input, resource management).

### *2.2.2 Διαχείριση πόρων (Resource Management)*

Η διαχείριση πόρων (resource management) είναι ένα από τα πιο κρίσιμα υποσυστήματα μιας μηχανής παιχνιδιών, καθώς αφορά τον χειρισμό όλων των εξωτερικών δεδομένων που χρειάζεται ένα παιχνίδι: textures, meshes, ήχους, animations, shaders, scripts κ.ά.

Σύμφωνα με ([4]), οι πόροι αποτελούν «το αίμα της μηχανής», αφού χωρίς σωστή φόρτωση, caching και απελευθέρωση μνήμης, κανένα άλλο υποσύστημα δεν μπορεί να λειτουργήσει. Ένα τυπικό υποσύστημα resource management πρέπει να εξασφαλίζει:

- Φόρτωση (Loading) – Δυνατότητα ανάγνωσης διαφόρων τύπων αρχείων από τον δίσκο (π.χ. PNG, OBJ, WAV).
- Αφαίρεση (Abstraction) – Ομοιόμορφο API για διαφορετικά format (π.χ. όλες οι εικόνες να εκτίθενται ως “Texture” αντικείμενα).
- Caching / Reuse – Αν ένα texture ή mesh ζητηθεί ξανά, πρέπει να επιστραφεί από cache αντί να φορτωθεί εκ νέου.
- Αποδέσμευση Μνήμης – Αυτόματη ή ελεγχόμενη αποδέσμευση ώστε να μην υπάρχουν memory leaks.
- Hot-Reload – Σε πιο εξελιγμένα συστήματα, δυνατότητα ανανέωσης πόρων «εν κινήσει» (χρήσιμο σε development tools).

Η Διαχείριση πόρων, είναι παρόν σε όλες τις σύγχρονες μηχανές, ανεξαρτήτως κλίμακας, και συνήθως συνδέεται με το filesystem και το asset pipeline . Επίσης, στο άρθρο του ο C. Politowski [11] δείχνει ότι το resource management είναι ένα από τα πιο «πυκνά» υποσυστήματα από άποψη εξαρτήσεων, αφού όλα σχεδόν τα modules (rendering, physics, audio) το χρησιμοποιούν.

### *2.2.3 Μαθηματική βιβλιοθήκη (Math Library)*

Η μαθηματική βιβλιοθήκη αποτελεί θεμέλιο λίθο κάθε μηχανής παιχνιδιών. Όλα τα βασικά υποσυστήματα – rendering, physics, animation, collision detection – βασίζονται σε μαθηματικές δομές και αλγορίθμους για να λειτουργήσουν σωστά.

Κύριες Δομές & Συναρτήσεις:

- Σύμφωνα με ([5]) , οι ελάχιστες δομές που χρειάζονται είναι:
- Διανύσματα (Vectors) – 2D, 3D, 4D, με βασικές πράξεις (πρόσθεση, αφαίρεση, εσωτερικό/εξωτερικό γινόμενο, κανονικοποίηση).
- Πίνακες (Matrices) – 3×3 και 4×4 για μετασχηματισμούς (translation, rotation, scaling).
- Quaternions – αναπαράσταση περιστροφών χωρίς gimbal lock· γρήγορη σύνθεση περιστροφών.
- Γεωμετρικοί Μετασχηματισμοί – affine transformations, perspective/orthographic projections.
- Collision Volumes – bounding boxes (AABB, OBB), bounding spheres, planes.
- Αριθμητική Γραμμικής Άλγεβρας – απαραίτητη για interpolation (lerp, slerp), normal transformations, και physics integration.

Η βιβλιογραφία προτείνει ([1]) να υλοποιείται αυτή η βιβλιοθήκη με χαμηλό κόστος σε μνήμη και CPU, καθώς χρησιμοποιείται σε tight loops (περιοχές στον κώδικα που κατά την εκτέλεση και λειτουργία του προγράμματος εκτελούνται πολλαπλές φόρες ανά δευτερόλεπτο).

Έτσι ολοκληρώσαμε τα τρία υποσυστήματα υποδομής, Από εδώ και πέρα, περνάμε στον Πυρήνα της Μηχανής.

### *2.2.4 Υποσύστημα απόδοσης γραφικών (Rendering Engine) 2D/3D*

Το υποσύστημα απόδοσης γραφικών (rendering engine) είναι ίσως το πιο χαρακτηριστικό και απαιτητικό τμήμα μιας μηχανής παιχνιδιών, καθώς είναι υπεύθυνο για τη μετατροπή των δεδομένων του παιχνιδιού σε εικόνες που προβάλλονται στην οθόνη σε πραγματικό χρόνο. Το rendering system αποτελείται από μια σειρά από στάδια που ακολουθούν το μοντέλο του (graphic pipeline):

- Μετασχηματισμοί (model, view, projection).
- Σχεδίαση γεωμετρίας (meshes, sprites).
- Εφαρμογή υλικών & shaders (textures, materials, GPU programs).
- Φωτισμός & σκιές.

- Μετα-επεξεργασία (post-processing effects: bloom, HDR, motion blur).

Οι σύγχρονες μηχανές υποστηρίζουν data-oriented σχεδίαση, όπου τα δεδομένα (meshes, textures) οργανώνονται με τρόπο βελτιστοποιημένο για την GPU, ώστε να επιτυγχάνεται μέγιστη απόδοση ([14]).

**Στόχοι του Rendering Engine**

- Αφαιρετικό API: να επιτρέπει στον προγραμματιστή να δουλεύει σε ανώτερο επίπεδο (mesh.draw()) χωρίς να γράφει χαμηλού επιπέδου κλήσεις. Να είναι ανεξάρτητος από την υλοποίηση στην βάση (OpenGl, DirectX, Vulcan, ...).
- Απόδοση (Performance): υποστήριξη batching, frustum culling, level-of-detail (LOD).
- Ευελιξία (Flexibility): modular σύστημα shaders, πολλαπλά rendering passes.
- Υποστήριξη 2D και 3D: sprites και tilemaps για 2D, meshes, skeletal animation και φωτισμό για 3D.

**Βασικές Δομές**

- Στην πράξη, το rendering system βασίζεται σε πυρηνικές κλάσεις/δομές, όπως:
- Mesh / Model : περιέχει vertices, indices, normals, UVs.
- Material / Shader : ορίζει πώς θα αποδοθεί ένα αντικείμενο (χρώμα, υφή, εφέ).
- Camera : καθορίζει view/projection matrices.
- Renderer : συντονίζει την απόδοση της σκηνής.

Ο ([8]) έδειξε ότι το rendering είναι ένα από τα πιο πυκνά και «κεντρικά» modules, αφού εξαρτάται από το math library (μετασχηματισμοί), το resource management (φόρτωση textures/shaders), και το platform layer (δημιουργία παραθύρου, context).

Στις εμπορικές μηχανές:
Η Unity χρησιμοποιεί Scriptable Render Pipeline (SRP) για modular rendering.
Η Unreal Engine προσφέρει deferred + forward rendering, με προηγμένα εφέ φωτισμού.
Η Godot παρέχει rendering abstraction ώστε να υποστηρίζει πολλαπλά backends (OpenGL, Vulkan).

### *2.2.5 Διαχείριση εισόδου (Input Handling)*

Η διαχείριση εισόδου είναι το υποσύστημα της μηχανής που είναι υπεύθυνο για τη συλλογή και ερμηνεία σημάτων από τον παίκτη μέσω πληκτρολογίου, ποντικιού, gamepad, ή άλλων συσκευών. Χωρίς αυτήν, δεν υπάρχει αλληλεπίδραση με τον κόσμο του παιχνιδιού. Η διαχείριση

εισόδου θα πρέπει να είναι, αξιόπιστο (reliable) , να καταγράφει όλα τα γεγονότα (events) χωρίς απώλειες, γενικευμένη (abstracted), να κρύβει δηλαδή τις διαφορές ανάμεσα σε λειτουργικά συστήματα ή hardware και τέλος να είναι ευέλικτη (flexible) , να επιτρέπει αλλαγή (remapping) πλήκτρων και χρήση πολλαπλών συσκευών.

Κύρια στοιχεία τις διαχείρισης εισόδου είναι τα εξής:

- Event System – κάθε ενέργεια (π.χ. πάτημα πλήκτρου) αντιστοιχεί σε event (KeyPressed, MouseMoved).
- State Management – το input system πρέπει να κρατά τρέχουσα κατάσταση (π.χ. "το W είναι πατημένο").
- Mapping / Binding – τα raw inputs (πλήκτρα) συνδέονται με ενέργειες του παιχνιδιού (π.χ. "W → Move Forward").
- Device Independence – το ίδιο API πρέπει να δουλεύει είτε το input έρχεται από πληκτρολόγιο είτε από controller.

Η διαχείριση εισόδου (input handiling system) αποτελεί «υποσύστημα-γέφυρα» που συνδέει τον χρήστη με τον game loop, και γι' αυτό είναι στενά δεμένο με το platform layer (λειτουργικό σύστημα) και με το game loop (ώστε να καταγράφονται οι εντολές στον σωστό χρόνο).
Ο ([8]) δείχνει ότι το input handling έχει μεσαία πυκνότητα εξαρτήσεων, αλλά είναι αναγκαίο για οποιοδήποτε είδος παιχνιδιού.

### *2.2.6 Βασικός βρόχος παιχνιδιού (Game Loop)*

Το game loop είναι η βασική ροή εκτέλεσης του παιχνιδιού, ένας επαναλαμβανόμενος κύκλος που ελέγχει τη ροή δεδομένων και την αλληλεπίδραση όλων των υποσυστημάτων της μηχανής. Χωρίς αυτό, η μηχανή δεν θα μπορούσε να ενημερώνει τον κόσμο, να επεξεργάζεται την είσοδο ή να εμφανίζει εικόνες στην οθόνη. Είναι η κεντρική «μηχανή» που συγχρονίζει:

- Την επεξεργασία της εισόδου (input handling).
- Την ενημέρωση της κατάστασης του παιχνιδιού (game state update, physics, AI).
- Την απόδοση των γραφικών (rendering).

Ο βρόχος παιχνιδιού (game-loop) αποτελεί θεμελιώδες σχεδιαστικό μοτίβο (design pattern) που ο ([9]) το κατηγοριοποιεί στα "Sequencing Patterns" και είναι αυτό που εξασφαλίζει ότι το παιχνίδι συνεχίζει να «τρέχει» ακόμα κι όταν ο χρήστης δεν κάνει καμία ενέργεια.

Σύμφωνα με ([5]), υπάρχουν διάφορες στρατηγικές υλοποίησης του βρόχου:

- Simple Loop – μια ενιαία συνάρτηση while(running) που διαχειρίζεται input, update, render.

- Fixed Time Step – η λογική του παιχνιδιού ενημερώνεται σε σταθερά χρονικά βήματα (π.χ. 60Hz), ανεξάρτητα από το rendering.
- Variable Time Step – η λογική βασίζεται στον πραγματικό χρόνο (delta time) μεταξύ frames.
- Hybrid Loops – συνδυασμοί των παραπάνω για σταθερότητα + απόδοση.

### *2.2.7 Physics engine (Μηχανή Φυσικής)*

Η μηχανή φυσικής είναι υπεύθυνη για την προσομοίωση ρεαλιστικής κίνησης, δυνάμεων και συγκρούσεων στον κόσμο του παιχνιδιού. Πρόκειται για ένα από τα πιο πολύπλοκα υποσυστήματα, καθώς βασίζεται σε μαθηματικά μοντέλα και αλγορίθμους βελτιστοποίησης. Η μηχανή φυσικής (physics engine) μπορεί να χωριστεί σε δύο βασικά μέρη: **Dynamics** – προσομοίωση κίνησης με βάση δυνάμεις (π.χ. βαρύτητα, ώθηση, τριβή) και **Collision Detection** – εντοπισμός συγκρούσεων μεταξύ αντικειμένων, ώστε να εμποδίζεται η διείσδυσή τους.

Οι ([7]) αναφέρουν ότι οι μηχανές φυσικής μπορούν να είναι είτε απλουστευμένες (arcade physics) για παιχνίδια 2D/retro, είτε ρεαλιστικές (rigid body dynamics) για 3D παιχνίδια και προσομοιώσεις. Κάποια υποσυστήματα φυσικής:

- Collision Detection
- Broad Phase: γρήγορη ανίχνευση πιθανών συγκρούσεων (π.χ. με spatial partitioning: quadtrees, octrees, BVH).
- Narrow Phase: ακριβής υπολογισμός σημείων επαφής.
- Collision Response
- Εφαρμογή κατάλληλων δυνάμεων για να αντιδράσουν τα σώματα (π.χ. bounce, slide, stop).
- Rigid Body Dynamics
- Υπολογισμός θέσης/προσανατολισμού με βάση τις δυνάμεις (F=ma, ολοκλήρωση Euler/Verlet).
- Constraints / Joints
- Εφαρμογή περιορισμών (π.χ. άρθρωση τύπου hinge, spring, slider).

Στις εμπορικές μηχανές:
Η Unity χρησιμοποιεί το PhysX (NVIDIA).
Η Unreal Engine χρησιμοποιεί το Chaos Physics Engine.
Η Godot διαθέτει δικό της physics system για 2D και 3D.

### *2.2.8 Audio system (Σύστημα Ήχου)*

Το υποσύστημα ήχου είναι υπεύθυνο για την αναπαραγωγή όλων των ηχητικών στοιχείων ενός παιχνιδιού: μουσικής υπόκρουσης, ηχητικών εφέ, διαλόγων, ambient ήχων. Παρά το γεγονός ότι συχνά υποτιμάται σε σχέση με τα γραφικά, ο ήχος συμβάλλει καθοριστικά στη συνολική εμπειρία και εμβύθιση του παίκτη.

Κύριες Λειτουργίες του Audio System

1. Αναπαραγωγή Ήχου (Playback) – υποστήριξη πολλών format (WAV, MP3, OGG).
2. Μίξη Ήχων (Mixing) – δυνατότητα αναπαραγωγής πολλαπλών καναλιών ταυτόχρονα.
3. Χωρικός Ήχος (3D Spatialization) – οι ήχοι να ακούγονται από διαφορετικές κατευθύνσεις ανάλογα με τη θέση/κίνηση της κάμερας.
4. Εφέ (Effects Processing) – reverb, doppler effect, filtering.
5. Διαχείριση Πόρων Ήχου – φόρτωση/αποδέσμευση audio assets (ενιαία με το resource manager).

Σύμφωνα με ([2]), το audio system εμφανίζεται σε όλες τις εμπορικές μηχανές παιχνιδιών, συνήθως ως συνεργαζόμενο υποσύστημα με το rendering (π.χ. συγχρονισμός animation–ήχου)
.

Στις εμπορικές μηχανές:
Η Unity προσφέρει AudioMixer για advanced mixing + 3D effects.
Η Unreal Engine έχει SoundCue system για procedural ήχους.
Η Godot διαθέτει AudioServer με υποστήριξη bus και real-time effects.

### *2.2.9 Animation system (Σύστημα Κίνησης)*

Το σύστημα κίνησης (animation system) είναι υπεύθυνο για την αναπαράσταση και διαχείριση των κινήσεων μέσα στο παιχνίδι, τόσο σε 2D (π.χ. sprite sheets) όσο και σε 3D (skeletal animation). Η ομαλή και ρεαλιστική κίνηση είναι κρίσιμη για την εμβύθιση του παίκτη.

Τύποι Animation:

1. 2D Animation (Sprites)
    - Sprite Sheets (σειρές από frames).
    - Frame-by-frame animation.
    - Texture atlas για καλύτερη απόδοση.
2. 3D Animation
    - Skeletal Animation: ένα “skeleton” (αρθρώσεις/joints, bones) ελέγχει το mesh.

- Skinning: το mesh «δένει» με τα bones μέσω weight maps.
- Interpolation: blending ανάμεσα σε keyframes (linear, spherical interpolation).
- Animation Blending: συνδυασμός διαφορετικών animations (π.χ. walk + shoot).

3. Procedural Animation
    - Κίνηση που παράγεται από αλγόριθμο (π.χ. ragdolls, inverse kinematics, procedural walking).

Βασικές Λειτουργίες

- Playback – δυνατότητα να παίζει, σταματά, κάνει pause και loop ένα animation.
- State Machine – οργάνωση animations σε καταστάσεις (idle, walk, run, jump).
- Blending – ομαλή μετάβαση ανάμεσα σε animations.
- Event Sync – συγχρονισμός με ήχο/physics (π.χ. ήχος βημάτων σε κάθε keyframe).

Στις εμπορικές μηχανές:
Η Unity χρησιμοποιεί το Mecanim system (state machines, blending, inverse kinematics).
Η Unreal Engine διαθέτει Animation Blueprints, με πλήρη έλεγχο blending και procedural animation. Η Godot παρέχει AnimationPlayer node για 2D/3D με timeline.

### *2.2.10 Multithreading (Παραλληλοποίηση)*

Το multithreading επιτρέπει σε μια μηχανή παιχνιδιών να εκτελεί πολλές λειτουργίες ταυτόχρονα, κατανέμοντας την εργασία σε διαφορετικούς επεξεργαστικούς πυρήνες. Αυτό μπορεί να γίνει απαραίτητο, καθώς τα σύγχρονα παιχνίδια έχουν πολλαπλά, απαιτητικά υποσυστήματα (rendering, physics, AI, ήχο, δικτύωση).

Βασικές Προσεγγίσεις

1. Task-based Parallelism
    - Χωρισμός της εργασίας σε «tasks» (μικρές, ανεξάρτητες εργασίες).
    - Χρήση ενός scheduler που κατανέμει tasks στα διαθέσιμα threads.
    - Υιοθετείται σε Unreal Engine και Frostbite.
2. Data Parallelism
    - Εφαρμογή της ίδιας λειτουργίας σε διαφορετικά δεδομένα ταυτόχρονα (π.χ. particle systems).
3. Subsystem Parallelism

- Ανάθεση διαφορετικών υποσυστημάτων σε διαφορετικά threads (π.χ. physics σε ένα thread, rendering σε άλλο).

Οι ([7]) επισημαίνουν ότι το threading σε game engines απαιτεί ιδιαίτερη προσοχή για να αποφευχθούν race conditions, deadlocks, synchronization bottlenecks.
Στις εμπορικές μηχανές:
Η Unreal Engine διαθέτει task graph system για scheduling.
Η Unity εισήγαγε το C# Job System και ECS (Entity Component System) για scalable parallelism.
Η Godot 4 υποστηρίζει multithreaded physics και rendering pipelines.

## 2.3 Σύνοψη

Στην παρούσα ενότητα αναλύθηκαν τα βασικά υποσυστήματα που συγκροτούν μια μηχανή παιχνιδιών. Από το Rendering Engine και το Game Loop, μέχρι τα υποσυστήματα Φυσικής, Ήχου, Κίνησης και Πολυνημάτωσης, καταδείχθηκε ότι η αρχιτεκτονική μιας μηχανής αποτελεί ένα σύνολο αλληλοεξαρτώμενων μονάδων που συνεργάζονται προκειμένου να επιτύχουν την ομαλή λειτουργία και την εμβύθιση του παίκτη.

Παρόλη την πολυπλοκότητα του αντικειμένου και τις σημαντικές διαφορές που μπορεί να συναντήσουμε ανάμεσα σε διαφορετικές εμπορικές ή ακαδημαϊκές μηχανές παιχνιδιών, τα στοιχεία που παρουσιάστηκαν συνιστούν, κατά τη γνώμη μου και με βάση τη σχετική έρευνα και βιβλιογραφία, μια κοινή τομή. Αποτελούν δηλαδή τον ελάχιστο πυρήνα υποσυστημάτων που συναντάμε σχεδόν σε κάθε μηχανή, ανεξάρτητα από την υλοποίηση, την πλατφόρμα στόχο ή το είδος του παιχνιδιού.

Επιπλέον, η παρουσίασή τους εδώ εστίασε κυρίως στη θεωρητική βάση και στις αρχές που τα διέπουν, ώστε να προσφέρει ένα στέρεο υπόβαθρο κατανόησης. Στην επόμενη ενότητα, η ανάλυση θα μεταφερθεί σε πιο πρακτικά ζητήματα και υλοποιήσεις, παρουσιάζοντας τη δική μου πορεία ανάπτυξης μιας μηχανής παιχνιδιών σε C++ με τη χρήση βιβλιοθηκών χαμηλού επιπέδου (OpenGL, GLFW, GLM), καθώς και τα προβλήματα και οι λύσεις που αναδύθηκαν στη διαδικασία.

Με την ολοκλήρωση της παρουσίασης του θεωρητικού υποβάθρου, η εργασία προχωρά στην ανάλυση των στόχων και της μεθοδολογίας που ακολουθήθηκε. Στο επόμενο κεφάλαιο παρουσιάζονται οι επιδιώξεις της έρευνας, καθώς και τα εργαλεία που χρησιμοποιήθηκαν για την ανάπτυξη της μηχανής.

# ΚΕΦΑΛΑΙΟ 3 Προδιαγραφές και στόχοι

Σκοπός της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός και η υλοποίηση μιας ελαφριάς μηχανής παιχνιδιών (game engine), η οποία θα παρέχει τα βασικά εργαλεία για την ανάπτυξη κυρίως δισδιάστατων παιχνιδιών, καθώς και απλών τρισδιάστατων εφαρμογών. Η μηχανή στοχεύει στη modular αρχιτεκτονική, στην επεκτασιμότητα και στη δυνατότητα χρήσης της τόσο από C++ όσο και από Python. Επιπλέον, τόσο για την αξιολόγηση της μηχανής όσο και για την ανάδειξη των διαφορετικών χαρακτηριστικών και δυνατοτήτων της, δημιουργήθηκαν τρία παιχνίδια, εκ των οποίων το ένα αναπτύχθηκε εξ ολοκλήρου με τη χρήση τεχνητής νοημοσύνης.

Στη συνέχεια παρουσιάζονται οι τεχνολογικές επιλογές που αποτέλεσαν τη βάση της μηχανής. Ακολούθως, αναλύονται οι σχεδιαστικές και αλγοριθμικές επιλογές που υιοθετήθηκαν. Έπειτα, εξετάζεται ο εκπαιδευτικός χαρακτήρας της μηχανής, ενώ τέλος παρουσιάζονται τα εργαλεία τεχνητής νοημοσύνης που χρησιμοποιήθηκαν κατά την ανάπτυξή της.

## 3.1 Επιλογές στη βάση της μηχανής

Η μηχανή αναπτύσσεται χρησιμοποιώντας τη γλώσσα προγραμματισμού C++, σε συνδυασμό με τη βιβλιοθήκη OpenGL για την απόδοση γραφικών και τη GLFW για τη διαχείριση παραθύρων και εισόδου χρήστη. Για τη διαδικασία μεταγλώττισης και διαχείρισης του έργου επιλέγεται το CMake, με στόχο τη φορητότητα και την ευκολία συντήρησης του κώδικα.

Οι επιλογές αυτές πραγματοποιήθηκαν με γνώμονα, αφενός, το γεγονός ότι οι περισσότερες σύγχρονες μηχανές παιχνιδιών είναι υλοποιημένες σε C++ και πλαισιώνονται από εργαλεία που αναπτύσσονται γύρω από τη συγκεκριμένη γλώσσα ή από παρεμφερείς τεχνολογίες. Αφετέρου, η C++ αποτελεί καθιερωμένη επιλογή για την ανάπτυξη εφαρμογών υψηλών απαιτήσεων ως προς την απόδοση και την ταχύτητα εκτέλεσης.

Η επιλογή της OpenGL, έναντι εναλλακτικών τεχνολογιών όπως η Vulkan, πραγματοποιήθηκε κυρίως λόγω της συγκριτικής απλότητας και της ευκολίας χρήσης που προσφέρει, ιδίως στο πλαίσιο μιας εκπαιδευτικής και ερευνητικής υλοποίησης.

Ένας βασικός στόχος της μηχανής είναι η υποστήριξη πολλαπλών πλατφορμών, και συγκεκριμένα των λειτουργικών συστημάτων Linux και Windows, χωρίς αλλαγές στον πηγαίο κώδικα των εφαρμογών που αναπτύσσονται πάνω σε αυτή.

## 3.2 Προδιαγραφές και σχεδιαστικές επιλογές εντός της μηχανής

Η μηχανή επικεντρώνεται κυρίως στην ανάπτυξη δισδιάστατων παιχνιδιών, προσφέροντας ωστόσο και βασική υποστήριξη για απλές τρισδιάστατες σκηνές, ώστε να καλύπτεται ένα ευρύτερο φάσμα εφαρμογών χωρίς να αυξάνεται υπερβολικά η πολυπλοκότητα του συστήματος. Επιπλέον είναι κατασκευασμένη με τρόπο ώστε να είναι εύκολα επεκτάσιμη και σε δυνατότητες δημιουργίας πιο πολύπλοκων τρισδιάστατων παιχνιδιών, υπάρχει δηλαδή όλη εκείνη η απαιτούμενη υποδομή, για τη μελλοντική επέκταση και σε τέτοιες λειτουργικότητες.

Ιδιαίτερη έμφαση δίνεται στη δυνατότητα χρήσης της Python ως scripting γλώσσας. Όλα τα εργαλεία και τα υποσυστήματα που αναπτύσσονται στη μηχανή είναι διαθέσιμα και μέσω αντίστοιχης βιβλιοθήκης Python (ge), επιτρέποντας στον χρήστη να γράφει όλη τη λογική του παιχνιδιού του και τις συμπεριφορές με πιο γρήγορο και ευέλικτο τρόπο.

Οι περισσότερες λειτουργικότητες της μηχανής έχουν κατασκευαστεί πάνω στο ίδιο μοντέλο-σύστημα, που αποτελεί και βασική αρχιτεκτονική επιλογή της μηχανής. Το εν λόγω σύστημα είναι το Entity Component System (εν συντομία ECS), το οποίο περιγράφεται ως εξής, κάθε "αντικείμενο" της σκηνής εκφράζεται ως ένα entity, οποιοδήποτε "χαρακτηριστικό" θέλουμε να προσδώσουμε σε αυτό το αντικείμενο έχει τη μορφή ενός component (π.χ. έχουμε στη σκηνή μας τον entity player,ο όποιος έχει μια σειρά από components όπως το sprite, transformation, ... ). Περισσότερες λεπτομέρειες για το σύστημα αυτό υπάρχουν στη θεωρητική εισαγωγή της εργασίας αυτής.

Η μηχανή παρέχει υποστήριξη για την εισαγωγή δισδιάστατων assets, όπως sprites, καθώς και για τη δημιουργία δισδιάστατων animations. Ο χρήστης έχει τη δυνατότητα να ορίζει τη διάρκεια των animations και τη σειρά εναλλαγής των frames, επιτυγχάνοντας πλήρη έλεγχο στη ροή της κίνησης.

Επιπλέον, υποστηρίζεται η δημιουργία γραφικού περιβάλλοντος χρήστη (GUI), με βασικά στοιχεία όπως κουμπιά και κείμενα. Παρέχεται επίσης η δυνατότητα δημιουργίας custom UI στοιχείων, καθώς και η χρήση προσαρμοσμένων γραμματοσειρών, ώστε ο χρήστης να μπορεί να διαμορφώσει πλήρως την αισθητική του περιβάλλοντος.

Στον τομέα του ήχου, η μηχανή προσφέρει δυνατότητες εισαγωγής και αναπαραγωγής ήχων και μουσικής. Ο χρήστης μπορεί να ελέγχει παραμέτρους όπως η ένταση, η επαναλαμβανόμενη αναπαραγωγή (looping), καθώς και τη χωρική τοποθέτηση της πηγής ήχου και του ακροατή, επιτρέποντας τη δημιουργία βασικού spatial audio.

Τέλος, υποστηρίζεται η χρήση φωτισμού (lighting) εντός της σκηνής, με δυνατότητα παραμετροποίησης της θέσης της φωτεινής πηγής, του τρόπου διάχυσης του φωτός (distribution), καθώς και της περιοχής στην οποία επηρεάζει τη σκηνή, μέσω ορισμού γωνιών ή κυκλικών τομέων. Παρέχεται επίσης έλεγχος του χρώματος και της έντασης του φωτός.

Η μηχανή που αναπτύσσεται στο πλαίσιο της παρούσας εργασίας δεν υποστηρίζει ορισμένα χαρακτηριστικά που συναντώνται σε σύγχρονες, πλήρως ανεπτυγμένες μηχανές παιχνιδιών. Ενδεικτικά, δεν παρέχεται δυνατότητα δικτυακής λειτουργίας για την ανάπτυξη παιχνιδιών με πολ-

λαπλούς παίκτες, ούτε υποστήριξη σύνθετων τρισδιάστατων σκηνών με προηγμένες τεχνικές, όπως τρισδιάστατος φωτισμός και αντίστοιχα εφέ.

Παρά τους περιορισμούς αυτούς, καταβλήθηκε προσπάθεια υλοποίησης των βασικότερων και ουσιωδέστερων χαρακτηριστικών που συνιστούν τον πυρήνα μιας μηχανής παιχνιδιών. Τα χαρακτηριστικά αυτά λειτουργούν ως θεμελιώδης βάση και οδηγός για μελλοντική επέκταση, δεδομένου ότι η προσθήκη επιπλέον λειτουργικοτήτων μπορεί να πραγματοποιηθεί ακολουθώντας παρόμοια σχεδιαστικά πρότυπα και τεχνικές που έχουν ήδη υιοθετηθεί.

## 3.3 Εκπαιδευτικός χαρακτήρας της μηχανής

Ένας επιπλέον και εξίσου σημαντικός στόχος της παρούσας εργασίας είναι ο εκπαιδευτικός χαρακτήρας της μηχανής παιχνιδιών. Η ανάπτυξή της δεν αποσκοπεί μόνο στη δημιουργία ενός λειτουργικού εργαλείου, αλλά και στη σε βάθος κατανόηση των βασικών αρχών που διέπουν τον σχεδιασμό και την υλοποίηση σύγχρονων game engines.

Η αρχιτεκτονική της μηχανής έχει σχεδιαστεί με γνώμονα τη διαφάνεια και την αναγνωσιμότητα του κώδικα, επιτρέποντας την εύκολη μελέτη και επέκτασή του. Κάθε υποσύστημα (γραφικά, ήχος, είσοδος χρήστη, scripting, διαχείριση πόρων) είναι σαφώς διαχωρισμένο, ώστε να αναδεικνύεται ο ρόλος του και η αλληλεπίδρασή του με τα υπόλοιπα μέρη της μηχανής.

Η υποστήριξη της γλώσσας Python ως scripting γλώσσας ενισχύει περαιτέρω τον εκπαιδευτικό χαρακτήρα της μηχανής, καθώς επιτρέπει τον διαχωρισμό μεταξύ της χαμηλού επιπέδου υλοποίησης (C++) και της λογικής του παιχνιδιού. Με αυτόν τον τρόπο, ο χρήστης μπορεί να επικεντρωθεί στη σχεδίαση συμπεριφορών και μηχανισμών παιχνιδιού, χωρίς να απαιτείται βαθιά γνώση διαχείρισης μνήμης ή συστημικού προγραμματισμού.

Επιπλέον, σε σύγκριση με εμπορικές μηχανές παιχνιδιών, οι οποίες καλύπτουν σαφώς ευρύτερο φάσμα δυνατοτήτων και επιλογών, η απλότητα της παρούσας υλοποίησης (ως αποτέλεσμα του περιορισμένου συνόλου λειτουργιών) την καθιστά κατάλληλο εργαλείο για την εκμάθηση της κατασκευής παιχνιδιών.

Ο "no-GUI" χαρακτήρας της ενισχύει περαιτέρω τον εκπαιδευτικό της προσανατολισμό, καθώς επιτρέπει την εστίαση στον προγραμματισμό και στους αλγορίθμους. Σε αντίθεση με εμπορικές μηχανές, οι οποίες παρέχουν τη δυνατότητα ανάπτυξης ολοκληρωμένων παιχνιδιών χωρίς την απαραίτητη συγγραφή κώδικα, η παρούσα μηχανή προϋποθέτει άμεση ενασχόληση με τον προγραμματισμό, ενισχύοντας έτσι τη βαθύτερη κατανόηση των υποκείμενων μηχανισμών.

## 3.4 Εργαλεία και τεχνητή νοημοσύνη

Για την υλοποίηση της εφαρμογής "μηχανής παιχνιδιών" χρησιμοποιήθηκαν ως περιβάλλοντα ανάπτυξης τα VS Code, Vim και Cursor. Ως σύστημα κατανεμημένου ελέγχου εκδόσεων

(distributed version control) αξιοποιήθηκε το Git, σε συνδυασμό με την πλατφόρμα GitHub.

Η επιλογή των VS Code και Vim πραγματοποιήθηκε καθώς και τα δύο αποτελούν ισχυρά αλλά ταυτόχρονα ευέλικτα εργαλεία, ικανά να υποστηρίξουν την ανάπτυξη κώδικα σε C++, Python και Bash (για τα scripts που χρησιμοποιήθηκαν στο λειτουργικό σύστημα Linux), καθώς και τη διαχείριση αρχείων τεκμηρίωσης σε μορφή Markdown (.md). Επιπλέον, παρέχουν δυνατότητα σημαντικής επεκτασιμότητας μέσω της εγκατάστασης κατάλληλων επεκτάσεων (extensions), οι οποίες ενισχύουν τη λειτουργικότητά τους.

Το Cursor επιλέχθηκε λόγω της ιδιαίτερα αποτελεσματικής (κατά τον χρόνο συγγραφής της παρούσας εργασίας) ενσωμάτωσης μεγάλων γλωσσικών μοντέλων (Large Language Models – LLMs), γεγονός που διευκολύνει την αλληλεπίδραση με εργαλεία τεχνητής νοημοσύνης κατά τη διαδικασία ανάπτυξης.

Καθ' όλη τη διάρκεια της διαδικασίας υλοποίησης έγινε εκτεταμένη χρήση εργαλείων τεχνητής νοημοσύνης, κυρίως υπό τη μορφή συνομιλίας (chat) και αυτόματης συμπλήρωσης κώδικα (autocomplete), μέσω της δωρεάν έκδοσης του GitHub Copilot στο περιβάλλον του VS Code.

Επιπλέον, μετά την ολοκλήρωση της ανάπτυξης της εφαρμογής και των αντίστοιχων Python bindings (όπως αναφέρθηκαν σε προηγούμενη ενότητα), εργαλεία τεχνητής νοημοσύνης χρησιμοποιήθηκαν και για την παραγωγή της τεκμηρίωσης (documentation) του αντίστοιχου Python API.

Στο πλαίσιο αυτό τέθηκε ένας επιπλέον στόχος, συγκεκριμένα η ανάπτυξη ενός πλήρους παιχνιδιού με αποκλειστική χρήση εργαλείων τεχνητής νοημοσύνης, ακολουθώντας την προσέγγιση "vibe coding", κατά την οποία ο προγραμματιστής παρέχει υψηλού επιπέδου οδηγίες σε έναν AI agent χωρίς άμεση παρέμβαση στον παραγόμενο κώδικα. Η διαδικασία αυτή επιτρέπει, παράλληλα, την αξιολόγηση της μηχανής ως προς τον βαθμό κατανόησης της δομής της, καθώς και ως προς την ευχρηστία της από την πλευρά του τελικού χρήστη.

## 3.5 Σύνοψη κεφαλαίου

Στο παρόν κεφάλαιο παρουσιάστηκαν οι βασικές προδιαγραφές, οι στόχους και οι σχεδιαστικές επιλογές που καθόρισαν την ανάπτυξη της μηχανής παιχνιδιών στο πλαίσιο της παρούσας διπλωματικής εργασίας. Αναλύθηκαν οι τεχνολογίες και τα εργαλεία που επιλέχθηκαν για την υλοποίηση της μηχανής, καθώς και οι αρχιτεκτονικές αποφάσεις που διαμορφώνουν τη λειτουργία και την επεκτασιμότητά της. Παράλληλα, περιγράφηκαν οι βασικές δυνατότητες που προσφέρει στους τομείς των γραφικών, του ήχου, του scripting και της διαχείρισης πόρων, καθώς και οι περιορισμοί της σε σχέση με πιο σύνθετες εμπορικές μηχανές παιχνιδιών. Τέλος, παρουσιάστηκει ο εκπαιδευτικός χαρακτήρας της υλοποίησης και ο ρόλος των σύγχρονων εργαλείων ανάπτυξης και τεχνητής νοημοσύνης που αξιοποιήθηκαν κατά τη διαδικασία σχεδιασμού, υλοποίησης και τεκμηρίωσης του έργου.

Έχοντας πλέον καθορίσει τις προδιαγραφές και τους στόχους της παρούσας εργασίας, διαθέτουμε το απαραίτητο υπόβαθρο για να προχωρήσουμε στην υλοποίησή της. Στο επόμενο κεφάλαιο παρουσιάζεται αναλυτικά η πρακτική ανάπτυξη της μηχανής, περιγράφοντας βήμα προς βήμα τη διαδικασία σχεδιασμού και υλοποίησής της.

# ΚΕΦΑΛΑΙΟ 4 Υλοποίηση

## 4.1 Γενική αρχιτεκτονική

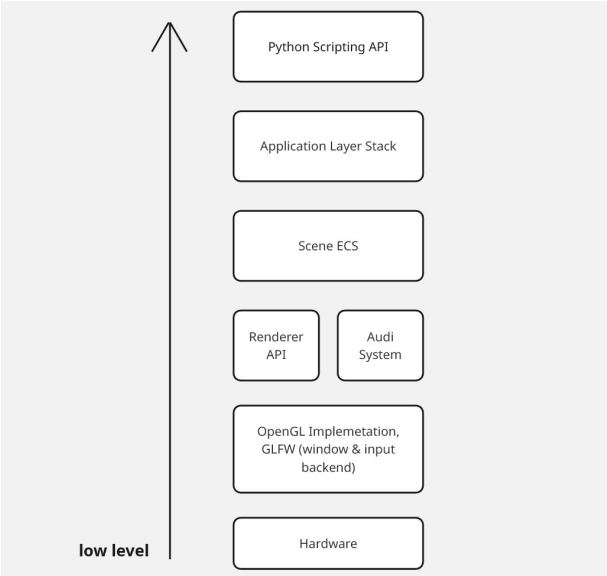
Προτού προχωρήσουμε στην ανάλυση και την πρακτική υλοποίηση όλων των συστημάτων και υποσυστημάτων της μηχανής παιχνιδιών, κρίνεται σκόπιμο να γίνει ένα βήμα πίσω και να εξεταστεί, σε αφαιρετικό επίπεδο, η αρχιτεκτονική που οφείλει να διαθέτει η μηχανή. Η αρχιτεκτονική αποτελεί το θεμέλιο πάνω στο οποίο δομούνται όλα τα επιμέρους τεχνικά στοιχεία. Επιπλέον, η κατανόησή της είναι ουσιώδης, καθώς επιτρέπει την ερμηνεία των σχεδιαστικών επιλογών που θα παρουσιαστούν στα επόμενα στάδια της εργασίας.

Η μηχανή που αναπτύσσεται στο πλαίσιο της παρούσας εργασίας βασίζεται στην πολυεπίπεδη αρχιτεκτονική (layered architecture). Τα κατώτερα επίπεδα, τα οποία συνιστούν τον πυρήνα της μηχανής, είναι υπεύθυνα για την απόδοση και τη διαχείριση πόρων. Αντίθετα, τα ανώτερα επίπεδα περιλαμβάνουν τα επιμέρους συστήματα που διαμορφώνουν το επίπεδο αλληλεπίδρασης με τον χρήστη και εκθέτουν το αντίστοιχο προγραμματιστικό περιβάλλον διεπαφής (API) Σχήμα 4.1.

Στο ανώτερο επίπεδο βρίσκεται η κλάση Application, η οποία αποτελεί το κεντρικό σημείο ελέγχου της μηχανής. Η κλάση αυτή είναι υπεύθυνη για τη δημιουργία και διαχείριση των παραθύρων, την αρχικοποίηση των βασικών υποσυστημάτων και την εκτέλεση του κύριου βρόχου (main loop) της εφαρμογής. Σε κάθε κύκλο εκτέλεσης πραγματοποιείται επεξεργασία γεγονότων, ενημέρωση της κατάστασης του συστήματος και απόδοση της σκηνής.

Για τη διαχείριση της λογικής οργάνωσης της εφαρμογής, η μηχανή παρέχει έναν μηχανισμό στοίβας επιπέδων (LayerStack). Κάθε επίπεδο (Layer) αντιπροσωπεύει ένα διακριτό τμήμα λειτουργικότητας, όπως για παράδειγμα τη λογική του παιχνιδιού ή το σύστημα γραφικού περιβάλλοντος (GUI). Η κλάση Application διατρέχει διαδοχικά τα επίπεδα και καλεί τις αντίστοιχες μεθόδους ενημέρωσης και διαχείρισης γεγονότων, επιτυγχάνοντας σαφή διαχωρισμό ευθυνών και ευέλικτη σύνθεση της τελικής εφαρμογής.

Στο χαμηλότερο επίπεδο της αρχιτεκτονικής εντοπίζονται τα υποσυστήματα που σχετίζονται με την άμεση αλληλεπίδραση με το υλικό και τις εξωτερικές βιβλιοθήκες. Η διαχείριση παραθύρου και εισόδου υλοποιείται μέσω αφηρημένων (abstract) κλάσεων, οι οποίες αποσυνδέουν τον πυρήνα της μηχανής από τη συγκεκριμένη βιβλιοθήκη που χρησιμοποιείται — στην παρούσα υλοποίηση, τη GLFW. Με τον τρόπο αυτό διατηρείται η ανεξαρτησία από εξωτερικές τεχνολογίες και καθίσταται δυνατή η αντικατάσταση ή επέκτασή τους χωρίς σημαντικές τροποποιήσεις στον πυρήνα.




Σχήμα 4.1: Γενική Αρχιτεκτονική

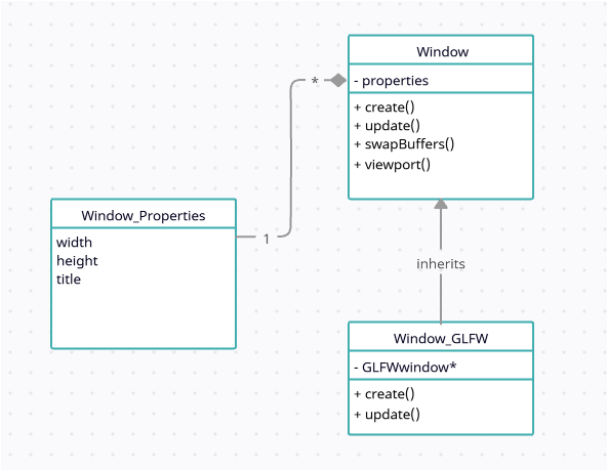
Η απόδοση γραφικών οργανώνεται επίσης μέσω επιπέδου αφαίρεσης (Rendering API abstraction), το οποίο δημιουργεί ένα ενδιάμεσο στρώμα πάνω από τη συγκεκριμένη υλοποίηση σε OpenGL. Η προσέγγιση αυτή επιτρέπει, σε θεωρητικό επίπεδο, την αντικατάσταση του συστήματος απόδοσης με διαφορετική τεχνολογία (π.χ. Vulkan ή Metal), περιορίζοντας τις απαιτούμενες αλλαγές σε σαφώς οριοθετημένο τμήμα του κώδικα.

Σε επίπεδο μοντελοποίησης των αντικειμένων της σκηνής, η μηχανή υιοθετεί αρχιτεκτονική βασισμένη στο πρότυπο Entity–Component–System (ECS). Οι οντότητες (Entities) αναπαρίστανται ως ελαφριές δομές αναγνώρισης, τα δεδομένα αποθηκεύονται σε επιμέρους στοιχεία (Components), ενώ η λογική υλοποιείται από Συστήματα (Systems) που επεξεργάζονται σύνολα οντοτήτων με συγκεκριμένους τύπους components (π.χ. για την απόδοση 2D sprites ή για το γραφικό περιβάλλον). Η προσέγγιση αυτή προσφέρει αυξημένη ευελιξία στη σύνθεση συμπεριφορών, υψηλό βαθμό επαναχρησιμοποίησης και καλύτερη διαχείριση της πολυπλοκότητας σε σύγκριση με την κλασική ιεραρχική κληρονομικότητα.

Τέλος, η μηχανή διαχωρίζει σαφώς τον πυρήνα της από το επίπεδο scripting. Μέσω της ενσωμάτωσης της γλώσσας Python παρέχεται ένα ανώτερο επίπεδο αφαίρεσης για την υλοποίηση της λογικής του παιχνιδιού, χωρίς να απαιτείται τροποποίηση του κώδικα C++ της μηχανής. Η επιλογή αυτή επιτρέπει ταχεία ανάπτυξη και πειραματισμό, διατηρώντας παράλληλα την απόδοση και τον έλεγχο που προσφέρει η γλώσσα C++ στον πυρήνα του συστήματος.

## 4.2 Σύστημα δημιουργίας παραθύρου (Window System – GLFW abstraction)

Η δημιουργία παραθύρου αποτελεί το πρώτο λειτουργικό βήμα για την εκτέλεση μιας εφαρμογής γραφικών σε πραγματικό χρόνο, καθώς μέσω αυτού δημιουργείται το απαραίτητο γραφικό περιβάλλον και το OpenGL context. Χωρίς την ύπαρξη ενεργού context, δεν είναι δυνατή η εκτέλεση εντολών απόδοσης στη GPU.




Σχήμα 4.2: Διάγραμμα Κλάσεων Παραθύρου

Στην παρούσα υλοποίηση, η διαχείριση παραθύρου πραγματοποιείται μέσω της βιβλιοθήκης GLFW, η οποία παρέχει διαλειτουργικότητα μεταξύ διαφορετικών λειτουργικών συστημάτων (Linux και Windows), καθώς και υποστήριξη δημιουργίας OpenGL context και διαχείρισης γεγονότων εισόδου.

Ωστόσο, η μηχανή δεν εξαρτάται άμεσα από την GLFW. Για την αποφυγή ισχυρής σύζευξης (tight coupling), σχεδιάστηκε μια αφαιρετική κλάση Window, η οποία ορίζει τη γενική διεπαφή διαχείρισης παραθύρου. Η συγκεκριμένη κλάση περιλαμβάνει μεθόδους όπως:

- create()
- on_update()
- viewport()
- swap_baffer()

Η υλοποίηση μέσω GLFW πραγματοποιείται σε υποκλάση (Window_GLFW), η οποία περιέχει τον πραγματικό κώδικα αρχικοποίησης και διαχείρισης του παραθύρου. Με αυτόν τον τρόπο, ο πυρήνας της μηχανής παραμένει ανεξάρτητος από τη συγκεκριμένη βιβλιοθήκη, επιτρέποντας μελλοντική αντικατάσταση του backend χωρίς τροποποίηση της υπόλοιπης αρχιτεκτονικής.

Επιπλέον, για την καλύτερη οργάνωση των δεδομένων που σχετίζονται με ένα παράθυρο, όπως οι διαστάσεις και ο τίτλος του, δημιουργήθηκε μια δομή (struct) με την ονομασία WindowProperties, όπως παρουσιάζεται στο διάγραμμα κλάσεων του αντίστοιχου σχήματος. 4.2.

Η κλάση Application είναι υπεύθυνη για τη δημιουργία του παραθύρου κατά την αρχικοποίηση της μηχανής, καθώς και για την εκτέλεση της κύριας λούπας, κατά την οποία γίνεται η επεξεργασία γεγονότων και η ανανέωση της οθόνης.

## 4.3 Σύστημα διαχείρισης εισόδου

Η διαχείριση εισόδου χρήστη αποτελεί βασικό υποσύστημα μιας μηχανής παιχνιδιών, καθώς επιτρέπει την αλληλεπίδραση μεταξύ του χρήστη και της εφαρμογής. Στην παρούσα υλοποίηση, το σύστημα εισόδου σχεδιάστηκε με γνώμονα την ανεξαρτησία από τη βιβλιοθήκη διαχείρισης παραθύρου, ακολουθώντας την ίδια φιλοσοφία αποσύζευξης που εφαρμόστηκε και στο σύστημα δημιουργίας παραθύρου.

Όπως και στην περίπτωση της διαχείρισης παραθύρου (προηγούμενη ενότητα), έτσι και εδώ, αν και η GLFW παρέχει έτοιμους μηχανισμούς καταγραφής γεγονότων πληκτρολογίου και ποντικιού, η μηχανή δεν χρησιμοποιεί άμεσα τους τύπους και τις σταθερές της βιβλιοθήκης. Αντίθετα, ορίζεται αφαιρετική (abstrac) κλάση Input, η οποία λειτουργεί ως διεπαφή υψηλού επιπέδου για τον έλεγχο καταστάσεων εισόδου, όπως:

- έλεγχος αν ένα πλήκτρο είναι πατημένο,
- έλεγχος κατάστασης πλήκτρων ποντικιού,
- ανάγνωση θέσης δρομέα.

### *4.3.1 Enum KeyCode και ανεξαρτησία από backend*

Για την αποφυγή εξάρτησης από τις σταθερές πλήκτρων της GLFW, δημιουργήθηκε τύπος enum class KeyCode, ο οποίος αντιστοιχίζεται ένα-προς-ένα με τους κωδικούς της βιβλιοθήκης. Με τον τρόπο αυτό:

Ο υπόλοιπος κώδικας της μηχανής δεν γνωρίζει την ύπαρξη της GLFW.

Ενδεχόμενη αλλαγή backend δεν απαιτεί τροποποίηση του gameplay κώδικα.

Ενισχύεται η αναγνωσιμότητα και η ασφάλεια τύπων (type safety).

Η επιλογή χρήσης enum class αντί για απλές ακέραιες σταθερές προσφέρει ισχυρότερη τυποποίηση και αποτρέπει σφάλματα κατά τη χρήση των πλήκτρων.

### *4.3.2 Event System*

Κάθε φορά που προκύπτει ένα γεγονός (event) από τη βιβλιοθήκη GLFW, η κλάση event_distpacher το ταξινομεί με βάση το είδος του.

```
class Event {
public:
    virtual ~Event() = default;
    virtual event_type get_type() const = 0;
    virtual const char* get_name() const = 0;
    virtual std::string to_string() const { return get_name(); }
public:
    bool handled = false;
};
```

```
enum class event_type{
    None = 0,
    Window_close,
    Window_resize,
    Window_focus,
    Window_lost_focus,
    Window_move,
    App_tick,
    App_update,
    App_render,
    Key_pressed,
    Key_released,
    Key_typed,
    Mouse_button_pressed,
    Mouse_button_released,
    Mouse_moved,
    Mouse_scrolled
};
```

```
class Event_dispatcher {
    public :
    Event_dispatcher( Event &e ) : m_event(e) {};

    template < typename T, typename F >
    bool dispatch( F const & func ){
        if( m_event.get_type() == T::get_static_type() ) {
            m_event.handled = func( static_cast<T&>(m_event) );
            return true;
        }
        return false;
    }

    private :
    Event &m_event;
};
```

Τα γεγονότα αυτα μετατρέπονται σε εσωτερικά γεγονότα της μηχανής και διαβιβάζονται στην κλάση Application, η οποία τα προωθεί στα ενεργά Layers.

Έχοντας πλέον καθορίσει τόσο τον τρόπο δημιουργίας παραθύρου όσο και τον μηχανισμό διαχείρισης των γεγονότων εισόδου (event system), έχει τεθεί η απαραίτητη βάση για τον σχεδιασμό και την υλοποίηση του rendering pipeline, με σκοπό την απόδοση γραφικών σε πραγ-

ματικό χρόνο.

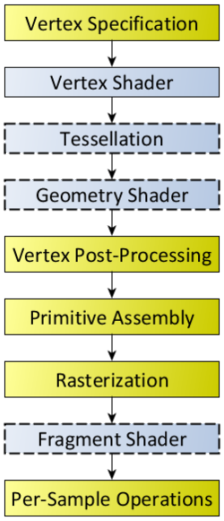
## 4.4 Σύστημα απόδοσης γραφικών Rendering Pipeline

Η απόδοση γραφικών σε πραγματικό χρόνο αποτελεί τον πυρήνα λειτουργίας μιας μηχανής παιχνιδιών. Το Rendering Pipeline περιγράφει τη διαδοχική επεξεργασία των δεδομένων της σκηνής με σκοπό τη μετατροπή τους στο τελικό τους αποτελεσμα (pixel) στην οθόνη. Η διαδικασία αυτή εκτελείται κυρίως στη μονάδα επεξεργασίας γραφικών (GPU) και ακολουθεί ένα καθορισμένο σύνολο σταδίων, τα οποία είναι προγραμματιζόμενα μέσω shaders. Σε γενικό επίπεδο, το pipeline απόδοσης ακολουθεί τα εξής στάδια:

1. Ορισμός γεωμετρίας (Vertices)
2. Μεταφορά δεδομένων στη GPU μέσω buffer objects
3. Vertex Processing (Vertex Shader)
4. Primitive Assembly & Rasterization
5. Fragment Processing (Fragment Shader)
6. Framebuffer Output

Στο στάδιο του vertex shader εφαρμόζονται οι μετασχηματισμοί θέσης (Model–View–Projection), ενώ στο fragment shader υπολογίζεται το τελικό χρώμα κάθε pixel, λαμβάνοντας υπόψη υλικά, υφές και φωτισμό.

Η μηχανή αξιοποιεί το προγραμματιζόμενο pipeline της OpenGL, επιτρέποντας πλήρη έλεγχο των σταδίων μέσω custom shader προγραμμάτων. Σχήμα 4.3.




Σχήμα 4.3: Rendering pipeline

Στην παρούσα μηχανή, το σύστημα απόδοσης σχεδιάστηκε με γνώμονα:

την απομόνωση της OpenGL από τον πυρήνα της μηχανής,
τη δυνατότητα μελλοντικής αντικατάστασης του backend,
την αποδοτική διαχείριση μεγάλου αριθμού δισδιάστατων αντικειμένων.

### *4.4.1 Διαχείριση γεωμετρίας και Buffer Objects*

Για την αποφυγή άμεσης εξάρτησης από την OpenGL, εισήχθη ενδιάμεσο επίπεδο αφαίρεσης (abstruction). Οι κλάσεις που διμιουργίθηκαν γαι τα βασικά αντικείμενα του rendering pipeline είναι:
- **Shader class** . Υπέφθηνη για την μεταγλώτηση των shader προγαραμμάτων και την διμιουργια των uniform δεδομένων που τιχόν θελουμε να στείλουμε στην καρτα γραφικών.
- **Vertex_buffer class** (VBO). Αποθήκευση δεδομένων κορυφών.
- **Index_buffer class** (IBO). Αποθήκευση δεικτών για επαναχρησιμοποίηση κορυφών.
- **Vertex_array class** (VAO). Περιγραφή διάταξης δεδομένων κορυφών.
- **Texture class**. Δεδομένα που αφορούν τα χρώματα και τις υφές.

### *4.4.2 Renderer και Renderer2D*

Η αρχιτεκτονική του rendering διαχωρίζεται σε δύο επίπεδα:

- Η κλάση **Renderer** διαχειρίζεται τη διαδικασία απόδοσης (π.χ. state management, ενεργοποίηση pipeline).
- Η κλάση **Renderer2D** καθορίζει τα είδη αντικειμένων που αποδίδονται (sprites, τετράπλευρα, tiles).

Ο διαχωρισμός αυτός επιτρέπει μελλοντική προσθήκη Renderer3D, χωρίς αναδιάρθρωση του βασικού μηχανισμού απόδοσης.

### *4.4.3 Βελτιστοποίηση μέσω Batch Rendering*

Στην αρχική υλοποίηση, κάθε αντικείμενο αποδιδόταν με ξεχωριστό draw call. Αν και λειτουργικά ορθό, το μοντέλο αυτό προκαλεί σημαντική επιβάρυνση στην επικοινωνία CPU–GPU, ιδιαίτερα σε δισδιάστατα παιχνίδια με μεγάλο αριθμό αντικειμένων. Για τον λόγο αυτό υλοποιήθηκε τεχνική Batch Rendering, όπου πολλαπλά αντικείμενα συγκεντρώνονται σε ενιαίο buffer και αποδίδονται με μειωμένο αριθμό draw calls. Η τεχνική αυτή μειώνει:

- state changes,
- δεσμεύσεις (bind operations),
- uniform uploads,

- κόστος συγχρονισμού CPU–GPU.

Στην υλοποίηση προστέθηκε επιπλέον buffer για διαχείριση πολλαπλών texture indices, επιτρέποντας την απόδοση διαφορετικών υφών εντός του ίδιου batch.

Η προσέγγιση αυτή βελτιώνει σημαντικά την απόδοση σε περιπτώσεις tile maps ή συστημάτων με μεγάλο πλήθος sprites.

## 4.5 Αναπαράσταση αντικειμένων: Από την κλάση GameObject στο Entity Component System (ECS)

Ως επόμενο βήμα, και δεδομένου ότι αποτελεί θεμελιώδες στοιχείο μιας μηχανής παιχνιδιών, υλοποιήθηκε ο μηχανισμός αναπαράστασης αντικειμένων στη σκηνή. Η επιλογή αυτή συνιστά φυσική συνέχεια της ανάπτυξης, καθώς επιτρέπει τον πρακτικό έλεγχο (testing) των προηγουμένως υλοποιημένων συστημάτων και, ιδίως, των επιμέρους μηχανισμών του rendering.

Στην αρχική εκδοχή της μηχανής, τα αντικείμενα της σκηνής μοντελοποιήθηκαν μέσω της κλάσης GameObject, η οποία περιλάμβανε βασικές ιδιότητες μετασχηματισμού, όπως θέση, κλίμακα και περιστροφή — καθώς και αναφορά σε γεωμετρικά δεδομένα (mesh). Η προσέγγιση αυτή βασίζεται στο παραδοσιακό αντικειμενοστραφές μοντέλο, σύμφωνα με το οποίο κάθε οντότητα υλοποιείται ως διακριτό αντικείμενο που ενσωματώνει τόσο τα δεδομένα όσο και τη συμπεριφορά που το χαρακτηρίζει.

### *4.5.1 Αναπαράσταση πλέγματος με δομή Half-Edge*

Για την ευέλικτη αναπαράσταση και επεξεργασία γεωμετρικών πλεγμάτων, υλοποιήθηκε δομή δεδομένων τύπου Half-Edge Mesh. Σε αντίθεση με την απλή αναπαράσταση μέσω λίστας κορυφών και δεικτών (vertex–index buffers), η δομή half-edge επιτρέπει την αποδοτική πλοήγηση στις τοπολογικές σχέσεις μεταξύ κορυφών, ακμών και εδρών.

Η βασική ιδέα της δομής half-edge είναι η αναπαράσταση κάθε ακμής ως δύο κατευθυνόμενες ημι-ακμές (half-edges). Κάθε half-edge περιέχει δείκτες προς:

- την κορυφή προέλευσης (origin vertex),
- την αντίθετη half-edge (twin),
- την επόμενη half-edge της ίδιας έδρας (next),
- την έδρα (face) στην οποία ανήκει.

Παρακάτω παρουσιάζεται ο σκελετός της κλάσης Half-Edge-Mesh, όπως υλοποιήθηκε στο πλαίσιο της παρούσας εργασίας.

```
class Mesh {
public:
    Mesh();
    ~Mesh();

    void quad_mesh( glm::vec3 const & bottom_left, glm::vec3 const & top_right );

    Vertex* add_vertex(glm::vec3 const & position);

    Face* add_face(std::vector<Vertex*> const & vertices);

    std::vector<Vertex*> const & get_vertices();
    std::vector<Face*> const & get_faces();
    std::vector<HalfEdge*> const & get_half_edges();
    void clear();

private:
    std::vector<Vertex*> vertices_;
    std::vector<HalfEdge*> half_edges_;
    std::vector<Face*> faces_;
};
```

Έχοντας ορίσει την παραπάνω κλάση, καθίσταται δυνατή η δημιουργία αντικειμένων που ενσωματώνουν μια τέτοια δομή. Σε αυτό το σημείο, κρίνεται σκόπιμο να διευκρινιστεί ότι, προκειμένου να διευκολυνθεί η μετάβαση από τη δομή Half-Edge Mesh στα δεδομένα που απαιτεί ο μηχανισμός απόδοσης (όπως vertex buffers και index buffers), υλοποιήθηκε μία επιπλέον ενδιάμεση κλάση με την ονομασία Mesh.

```
class Mesh {
public :
    Mesh( he::Mesh const & he_mesh );
    Mesh const & operator=( Mesh const & );
    ~Mesh();
    void draw() const;
    std::vector< glm::vec3 > get_vertices() const;
    std::vector< glm::vec3 > get_colors() const;
    std::vector< glm::vec2 > get_tex_coords() const;
    void set_tex_coords(const std::vector<glm::vec2>& tex_coords);
private:
    std::unique_ptr< Mesh_impl > m_impl;
};
```

Όπου η Mesh_impl κλάση, εχει την μορφή:

```
class Mesh_impl {
public:
    Mesh_impl(std::vector<vertex> const & vertices_data, std::vector<GLuint> & indices);
    Mesh_impl(Mesh_impl const &);
    Mesh_impl& operator=(Mesh_impl const &);
    ~Mesh_impl() = default;

    void draw() const;
    void bind() const;
    void unbind() const;

    std::vector< glm::vec3 > get_vertices();

    std::vector< glm::vec3 > get_colors();

    std::vector< glm::vec2 > get_tex_coords();

    void set_tex_coords(const std::vector<glm::vec2>& tex_coords);

private:
    std::unique_ptr< ge::Vertex_array > m_VAO;
    std::unique_ptr< ge::Vertex_buffer > m_VBO;
    std::unique_ptr< ge::Vertex_buffer > m_VBO2;
    std::unique_ptr< ge::Vertex_buffer > m_VBO3;
    std::unique_ptr< ge::Index_buffer > m_IBO;

    std::vector< GLfloat > m_vertices;
    std::vector< GLfloat > m_vertex_colors;
    std::vector< GLfloat > m_tex_coords;
};
```

### *4.5.2 Αρχική προσέγγιση: Ιεραρχία Κλάσεων*

Στην αρχική δομή, η μηχανή περιλάμβανε:

- Κλάση Entity (βάση)
- Κλάση GameObject (2D/3D αντικείμενα με mesh)
- Κλάση Sprite (2D αντικείμενα με texture και animation)

Κάθε οντότητα υλοποιούσε μεθόδους:

- update(dt)
- draw()

Η σκηνή (Scene) διατηρούσε μια συλλογή από αντικείμενα τύπου Entity, τα οποία ενημερώνονταν και αποδίδονταν διαδοχικά.

Αν και λειτουργικά είναι επαρκής, η προσέγγιση αυτή παρουσιάζει σημαντικούς περιορισμούς:

- Δυσκολία συνδυασμού χαρακτηριστικών (π.χ. sprite με φυσική και ήχο).
- Ανάγκη δημιουργίας νέων κλάσεων για κάθε πιθανό συνδυασμό λειτουργιών.
- Περιορισμένη ευελιξία σύνθεσης συμπεριφοράς.
- Αυξημένη σύζευξη δεδομένων και λογικής και πολυπλοκότητας.

Οι περιορισμοί αυτοί οδήγησαν στην αναζήτηση πιο ευέλικτου αρχιτεκτονικού μοντέλου.

### *4.5.3 Μετάβαση στην αρχιτεκτονική Entity–Component–System (ECS)*

**Entity**

Για την αντιμετώπιση των παραπάνω περιορισμών, υιοθετήθηκε το πρότυπο Entity–Component–System (ECS), το οποίο διαχωρίζει αυστηρά:

- Entity, η ταυτότητα.
- Component, τα δεδομένα.
- System, η λογική επεξεργασίας των δεδομένων.

Η πίο απλή μορφή που μπορεί να πάρει η κλάση entity είναι η παρακάτω.

```
struct Entity {
    uint32_t id;
};
```

**Components**

Τα components αποτελούν καθαρά δομές δεδομένων. Ενδεικτικά παραδείγματα components που υλοποιήθηκαν στο πλαίσιο της παρούσας εργασίας είναι τα εξής:

- TransformComponent
- SpriteRendererComponent
- LightComponent
- VelocityComponent

Ενδικτικά δεινω παρακάτω το TransformComponent.

```
struct Transform_Component
{
    glm::vec3 position;
    glm::vec3 rotation;
    glm::vec3 scale;

    Transform_Component()
        : position(0.0f), rotation(0.0f), scale(1.0f) {}
};
```

**Systems**

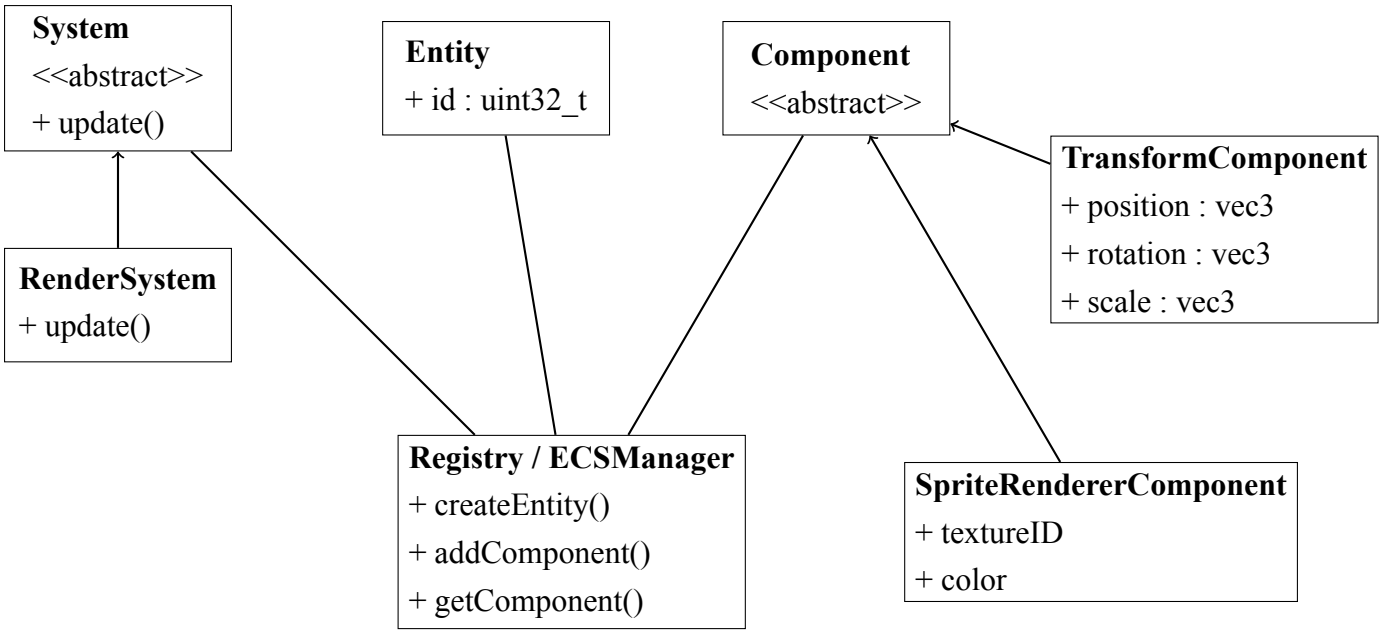
Τα Systems είναι υπεύθυνα για την επεξεργασία οντοτήτων που διαθέτουν συγκεκριμένο σύνολο components. Για παράδειγμα:

- Το RenderSystem επεξεργάζεται οντότητες με Transform και SpriteRenderer.
- Το PhysicsSystem επεξεργάζεται οντότητες με Transform και PhysicsComponent.
- Το GUISystem καλεί μεθόδους για οντότητες με UiComponent.

Η προσέγγιση αυτή επιτρέπει:

- δυναμική σύνθεση συμπεριφοράς,
- καλύτερη κλιμάκωση σε μεγάλο αριθμό οντοτήτων,
- καθαρό διαχωρισμό δεδομένων και λογικής.

Συνοψίζοντας, ο χρήστης της μηχανής έχει πλέον τη δυνατότητα να συνθέσει μια δική του οντότητα (game object), επιλέγοντας τα επιθυμητά χαρακτηριστικά μέσω της προσθήκης των κατάλληλων components και ορίζοντας τη διάρκεια και τον τρόπο αξιοποίησής τους σύμφωνα με τις ανάγκες της εφαρμογής.




Σχήμα 4.4: Class Diagram της αρχιτεκτονικής ECS

## 4.6 Σύστημα διαχείρισης σκηνής (Scene Management)

Η έννοια της σκηνής (Scene) αποτελεί θεμελιώδη δομικό στοιχείο σε κάθε μηχανή παιχνιδιών, καθώς λειτουργεί ως το περιβάλλον εντός του οποίου συνυπάρχουν και αλληλεπιδρούν οι οντότητες του κόσμου. Στο πλαίσιο της παρούσας μηχανής, η σκηνή υλοποιείται ως ανώτερο επίπεδο διαχείρισης, το οποίο ενοποιεί το σύστημα ECS, το σύστημα απόδοσης και τα επιμέρους συστήματα λογικής.

### *4.6.1 Η κλάση Scene*

Η κλάση Scene είναι υπεύθυνη για:

- Δημιουργία και καταστροφή οντοτήτων
- Διαχείριση του ECS Registry
- Κλήση των Systems (RenderSystem, PhysicsSystem, ScriptSystem)
- Διατήρηση αναφοράς στην ενεργή κάμερα
- Οργάνωση του update loop

Η κλάση Scene λειτουργεί ως επίπεδο απομόνωσης μεταξύ της μηχανής και της εφαρμογής του χρήστη. Ο χρήστης (δημιουργός παιχνιδιού) έχει τη δυνατότητα να κατασκευάζει μία σκηνή (δηλαδή ένα αντικείμενο της κλάσης Scene) για κάθε επίπεδο του παιχνιδιού του. Με τον τρόπο αυτό επιτυγχάνεται η απομόνωση των αντικειμένων και των επιμέρους λειτουργικοτήτων κάθε επιπέδου σε μια διακριτή οντότητα, διευκολύνοντας την οργάνωση, τη διαχείριση και την επεκτασιμότητα της εφαρμογής.

```
class Scene
{
public:
    Scene();
    ~Scene();
    void on_start();
    // Entity Management
    Entity create_entity();
    void destroy_entity(Entity entity);
    // Component Management
    template <typename T, typename... Args>
    T &add_component(Entity entity, Args &&...args);
    template <typename T>
    T &get_component(Entity entity);
    template <typename T>
    std::shared_ptr<T> get_component_ptr(Entity entity);
    template <typename T>
    void remove_component(Entity entity);
    template <typename T>
    bool has_component(Entity entity);
    void set_active_camera(Entity entity);
    Orthographic_camera &get_active_camera();
    bool has_active_camera();
    Entity create_default_camera(float aspect_ratio = 1.0f);
    void update_active_camera_aspect_ratio(float aspect_ratio);
    void update(float delta_time);
    void draw();
public:
    std::vector<Entity> m_entities;
    std::unordered_map<std::type_index, std::unordered_map<uint32_t, std::shared_ptr<void>>>
        m_components;
private:
    uint32_t m_next_entity_id = 1;
    std::optional<Entity> m_active_camera_entity;
};
```

### *4.6.2 Κύκλος ζωής σκηνής*

Κάθε σκηνή ακολουθεί συγκεκριμένο κύκλο ζωής:

1. Initialization

2. Runtime Update
3. Rendering Phase
4. Shutdown / Cleanup

Κατά τη φάση ενημέρωσης (update), καλούνται διαδοχικά τα ενεργά συστήματα. Κατά τη φάση απόδοσης, το RenderSystem συλλέγει τα απαραίτητα components και ενεργοποιεί το rendering pipeline.

## 4.7 Σύστημα Components

Στην αρχιτεκτονική Entity–Component–System (ECS), τα Components αποτελούν το βασικό δομικό στοιχείο περιγραφής των ιδιοτήτων μιας οντότητας. Όπως έχει ήδη αναφερθεί, τα components αποτελούν απλές δομές δεδομένων, χωρίς να ενσωματώνουν λειτουργικότητα. Στην παρούσα ενότητα παρουσιάζονται τα components που υλοποιήθηκαν στο πλαίσιο της συγκεκριμένης εργασίας.

Το πλήθος των components που διαθέτουν οι εμπορικές μηχανές παιχνιδιών είναι σαφώς μεγαλύτερο από εκείνο που παρουσιάζεται εδώ. Ωστόσο, στόχος της εργασίας είναι, αφενός, η κάλυψη των βασικών και θεμελιωδών στοιχείων και, αφετέρου, η διαμόρφωση της απαραίτητης αρχιτεκτονικής βάσης. Η λογική που ακολουθείται για την προσθήκη επιπλέον components παραμένει η ίδια και μπορεί να επεκταθεί με συνέπεια, ακολουθώντας τα ήδη καθορισμένα πρότυπα σχεδιασμού.

### *4.7.1 Transform Component*

Το Transform Component αποτελεί το βασικότερο component της μηχανής, καθώς περιγράφει τη γεωμετρική κατάσταση μιας οντότητας στον χώρο. Συγκεκριμένα, καθορίζει τη θέση, τον προσανατολισμό και την κλίμακα ενός αντικειμένου, επιτρέποντας τον υπολογισμό του τελικού μετασχηματισμού που χρησιμοποιείται στο rendering pipeline.

Στο πλαίσιο της αρχιτεκτονικής ECS, το Transform υλοποιείται ως καθαρή δομή δεδομένων, χωρίς επιχειρησιακή λογική, διατηρώντας τον διαχωρισμό μεταξύ δεδομένων και συστημάτων επεξεργασίας. Στην ενότητα *3.5.3. Μετάβαση στην Αρχιτεκτονική Entity–Component–System (ECS)* έχει δωθεί ώς παράδειγμα η κλάση Transform_Component.

### *4.7.2 SpriteRenderer Component*

Το SpriteRenderer Component είναι υπεύθυνο για την οπτική αναπαράσταση δισδιάστατων οντοτήτων στη σκηνή. Αποτελεί το βασικό component γραφικής απόδοσης για 2D εφαρμογές και λειτουργεί σε συνδυασμό με το Transform Component και το RenderSystem.

Στο πλαίσιο της αρχιτεκτονικής ECS, το SpriteRenderer υλοποιείται ως καθαρή δομή δεδομένων, η οποία περιγράφει τα χαρακτηριστικά απόδοσης ενός sprite, χωρίς να περιλαμβάνει λογική rendering.

Το SpriteRenderer Component περιλαμβάνει:

- Αναφορά σε texture
- Χρώμα (color)
- Συντεταγμένες texture (UV coordinates)
- Σημαία ενεργοποίησης (enabled/visible)

### *4.7.3 Light Component 2D*

Το Light Component χρησιμοποιείται για την αναπαράσταση πηγών φωτός εντός της σκηνής και συνεργάζεται άμεσα με το RenderSystem και τα shaders για τον υπολογισμό του φωτισμού των αντικειμένων.

Στο πλαίσιο της αρχιτεκτονικής ECS, το Light Component υλοποιείται ως δομή δεδομένων που περιγράφει τα φυσικά χαρακτηριστικά της πηγής φωτός, ενώ ο υπολογισμός του φωτισμού πραγματοποιείται στο επίπεδο του shader (GPU).

Το Light Component περιλαμβάνει:

- Σχετική θέση (relative position)
- Χρώμα (color)
- Κατέφθυνση (direction)
- Ακτίνα επιροής (radius)
- Γωνίες (angles)
- Σημαία on/off

Ο τελικός φωτισμός υπολογίζεται στον fragment shader.

### *4.7.4 Box_Collider Component*

Το Box collider Component χρησιμοποιείται για την περιγραφή ορθογωνικών περιοχών σύγκρουσης (collision volumes) σε δισδιάστατο χώρο. Αποτελεί βασικό στοιχείο για την ανίχνευση συγκρούσεων μεταξύ οντοτήτων και συνεργάζεται με το αντίστοιχο Physics ή Collision System. Στο πλαίσιο της αρχιτεκτονικής ECS, το Box_Collider υλοποιείται ως δομή δεδομένων που περιγράφει αποκλειστικά τα γεωμετρικά χαρακτηριστικά της περιοχής σύγκρουσης, χωρίς να περιλαμβάνει λογική ανίχνευσης. περιλαμβάνει:

- Διαστάσεις (width, height)
- Σημαία ενεργοποίησης
- Ιδιότητες φυσικής (π.χ. isTrigger)

```
struct BoxColliderComponent
{
    glm::vec2 size {1.0f, 1.0f};
    glm::vec2 offset {0.0f, 0.0f};

    bool isTrigger = false;
    bool enabled = true;
};
```

Η θέση του collider προκύπτει από τον συνδυασμό του TransformComponent της οντότητας και του τοπικού offset.

### *4.7.5 Camera Component*

Στα πλαίσια της εργασίας αυτής υλοποιήθηκε η ορθογραφική καμερα "orthographic".

**Ορθογραφική Προβολή (Orthographic Projection)** Η ορθογραφική προβολή χρησιμοποιείται κυρίως σε 2D εφαρμογές, καθώς δεν εισάγει προοπτική παραμόρφωση. Τα αντικείμενα διατηρούν σταθερό μέγεθος ανεξαρτήτως βάθους.

Ο projection matrix ορίζεται ως:

$$P = \text{ortho}(\text{left}, \text{right}, \text{bottom}, \text{top})$$

Το Camera Component περιλαμβάνει:

- Τύπο προβολής (Orthographic ή Perspective)
- Παραμέτρους προβολής
- Αναλογία διαστάσεων (aspect ratio)
- Near και far clipping planes
- Σημαία ενεργοποίησης (primary camera)

### *4.7.6 Audio Component*

Το Audio Component χρησιμοποιείται για την ενσωμάτωση ηχητικών πηγών εντός της σκηνής. Επιτρέπει την αναπαραγωγή ηχητικών εφέ και μουσικής, καθώς και την προσομοίωση χωρικού ήχου (spatial audio) σε 2D ή απλές 3D σκηνές.

Στο πλαίσιο της αρχιτεκτονικής ECS, το Audio Component περιγράφει αποκλειστικά τις ιδιότητες της ηχητικής πηγής, ενώ η διαχείριση και η αναπαραγωγή πραγματοποιούνται από το Audio System.

```
class Audio_Component
{
public:
    Audio_Component();
    ~Audio_Component();

    void load_wav(const std::string& file_path);
    void attach_buffer();

public:
    bool play_on_start = false;
    bool looping = false;
    float gain = 1.0f;
    float pitch = 1.0f;
    bool initialized = false;
    Audio_Source source;

private:
    Audio_Buffer* m_buffer = nullptr;
};
```

Το σύστημα απόστασης ήχου (audio popeline), χρησιμπολοιήθηκε η βιβλιοθήκη *opleal*.

### *4.7.7 UI Text Component*

Το UI_Text_Component χρησιμοποιείται για την απόδοση δυναμικού κειμένου στη διεπαφή χρήστη (Graphical User Interface). Η υλοποίηση βασίζεται στη βιβλιοθήκη stb_truetype.h, η οποία επιτρέπει την ανάγνωση και rasterization γραμματοσειρών τύπου TrueType (TTF).

Σε αντίθεση με την απλή χρήση bitmap γραμματοσειρών, η προσέγγιση αυτή επιτρέπει:

- Δυναμική φόρτωση γραμματοσειρών
- Δημιουργία atlas χαρακτήρων
- Υποστήριξη διαφορετικών μεγεθών κειμένου
- Ευελιξία στη χρωματική απόδοση

**Δημιουργία Font Atlas**

Η διαδικασία απόδοσης κειμένου περιλαμβάνει τα εξής στάδια:

1. Φόρτωση αρχείου γραμματοσειράς (.ttf)
2. Rasterization χαρακτήρων μέσω stb_truetype
3. Δημιουργία texture atlas που περιέχει glyph bitmaps
4. Υπολογισμό UV συντεταγμένων για κάθε χαρακτήρα
5. Απόδοση μέσω textured quads

Η χρήση atlas μειώνει σημαντικά τον αριθμό texture bindings κατά το rendering.
Η υλοποίηση του component είναι:

```
struct UI_Text_Component
{
public:
    float font_size;

    UI_Text_Component()
        : text("test"),
          font_path("assets/fonts/Tiny5/Tiny5−Regular.ttf"),
          font_size(16.0f) {}

    void build();
    void set_text(const std::string &new_text);
    bool needs_rebuild();
    Text &get_text_object();

private:
    std::string text;
    std::string font_path;
    glm::vec3 color = glm::vec3(1.0f);
    int font_color_mode = 0;

    bool m_needs_rebuild = true;
    Text ∗text_object = nullptr;
};
```

Κάθε φορά που αλλάζει το περιεχόμενο του κειμένου μέσω της μεθόδου:

```
void set_text(const std::string &new_text)
```

ενεργοποιείται η σημαία:

```
m_needs_rebuild = true;
```

Η προσέγγιση αυτή αποφεύγει περιττό υπολογισμό σε κάθε frame και βελτιώνει την απόδοση.

## 4.8 Python API

Στην παρούσα ενότητα περιγράφεται η βιβλιοθήκη της μηχανής για χρήση μέσω της γλώσσας προγραμματισμού Python.

Προτού προχωρήσουμε στο πρακτικό μέρος της υλοποίησης, κρίνεται σκόπιμο να παρουσιαστεί συνοπτικά η αιτιολόγηση της επιλογής της γλώσσας Python. Οι περισσότερες μηχανές παιχνιδιών (και ιδίως οι εμπορικές) εκτός από τον γραφικό τρόπο διεπαφής του χρήστη με τις επιμέρους λειτουργίες και τους μηχανισμούς τους, παρέχουν και μία scripting γλώσσα, μέσω της οποίας ο χρήστης μπορεί να αξιοποιεί προγραμματιστικά τις δυνατότητες της μηχανής.

Η γλώσσα scripting δεν ταυτίζεται κατ' ανάγκη με τη γλώσσα στην οποία έχει υλοποιηθεί ο πυρήνας της μηχανής, καθώς οι στόχοι των δύο επιπέδων είναι διαφορετικοί. Ο πυρήνας της μηχανής επιδιώκει την αποδοτικότερη δυνατή εκτέλεση αλγορίθμων και λειτουργιών, εστιάζοντας στην απόδοση και τον έλεγχο χαμηλού επιπέδου. Για τον λόγο αυτό, η υλοποίησή του πραγματοποιείται συνήθως σε γλώσσες χαμηλότερου επιπέδου, όπως C, C++, Rust ή Java, οι οποίες προσφέρουν μεγαλύτερο έλεγχο ως προς τη διαχείριση μνήμης και τους μηχανισμούς εκτέλεσης.

Αντίθετα, η scripting γλώσσα αποσκοπεί στη διευκόλυνση του χρήστη σε υψηλότερο επίπεδο αφαίρεσης. Στο πλαίσιο αυτό, ζητούμενο δεν είναι ο άμεσος έλεγχος πόρων, αλλά η απλότητα, η ταχύτητα ανάπτυξης και η ευχρηστία. Ενδεικτικές επιλογές scripting γλωσσών στον χώρο των μηχανών παιχνιδιών αποτελούν η C#, η Lua και η Python.

Με γνώμονα τα παραπάνω, και λαμβάνοντας υπόψη ότι η Python αποτελεί μια σύγχρονη και ώριμη γλώσσα προγραμματισμού με σταθερό οικοσύστημα και ευρεία αποδοχή, επιλέχθηκε ως γλώσσα διεπαφής της μηχανής.

Στο πλαίσιο της παρούσας εργασίας, δεδομένου ότι δεν υλοποιείται γραφικό περιβάλλον διεπαφής χρήστη (GUI) για την αλληλεπίδραση με τη μηχανή, ο ρόλος της Python υπερβαίνει αυτόν μιας απλής scripting γλώσσας. Η Python λειτουργεί ως το βασικό μέσο επικοινωνίας του χρήστη με τη μηχανή, γεγονός που οδήγησε στην ανάπτυξη μιας ολοκληρωμένης βιβλιοθήκης Python, η οποία εκθέτει (exposes) το σύνολο των λειτουργιών και των εργαλείων της μηχανής.

### *4.8.1 Τρόπος υλοποίησης*

Για τη δημιουργία της Python βιβλιοθήκης της μηχανής χρησιμοποιήθηκε εξωτερική βιβλιοθήκη ανοιχτού κώδικα, συγκεκριμένα η *pybind11*. Η βιβλιοθήκη αυτή επιτρέπει την έκθεση (binding) κλάσεων, συναρτήσεων και δομών δεδομένων της C++ προς τη γλώσσα Python, δημιουργώντας μια διεπαφή διασύνδεσης μεταξύ των δύο περιβαλλόντων.

Η *pybind11* βασίζεται σε σύγχρονα χαρακτηριστικά της C++ (C++11 και νεότερα) και παρέχει μηχανισμούς για σχεδόν άμεση αντιστοίχιση (one-to-one mapping) κλάσεων και συναρτήσεων της C++ σε αντίστοιχα Python αντικείμενα. Με τον τρόπο αυτό, αντικείμενα της μηχανής (όπως Entities και Components) μπορούν να γίνουν προσβάσιμα από Python scripts με ελάχιστη επιπλέον λογική.

Ωστόσο, η διαδικασία διασύνδεσης C++ και Python δεν είναι τετριμμένη. Η C++ επιτρέπει εκτεταμένη χρήση:

- templates
- pointers και references
- overloading συναρτήσεων
- custom memory management
- σύνθετων δομών δεδομένων

Τα χαρακτηριστικά αυτά δεν αντιστοιχούν πάντα άμεσα στο δυναμικό μοντέλο τύπων της Python. Η *pybind11* απλοποιεί σημαντικά τη διαδικασία αυτή, παρέχοντας:

- Αυτόματη μετατροπή βασικών τύπων (int, float, std::string)
- Υποστήριξη STL containers (std::vector, std::map)
- Διαχείριση ownership και lifetime αντικειμένων
- Μηχανισμούς έκθεσης κλάσεων με μεθόδους και ιδιότητες

Με τον τρόπο αυτό καθίσταται δυνατή η δημιουργία ενός Python API που αντικατοπτρίζει τη δομή της C++ μηχανής, διατηρώντας παράλληλα τον έλεγχο απόδοσης στον πυρήνα της εφαρμογής. Παρακάτω ένα μικρό τεχνικό παράδειγμα binding.

```
PYBIND11_MODULE(engine, m)
{
    py::class_<Entity>(m, "Entity")
        .def("get_transform", &Entity::get_transform);
}
```

Αυτό μεταφράζεται στην Python σε:

```
entity.get_transform()
```

### *4.8.2 Χρήση*

Έτσι λειπόν το Python API επιτρέπει στον χρήστη :

- Να δημιουργεί και να εκτελεί ένα αντικείμενο της κλάσης (application).
- Να δημιουργεί επίπεδα (Layers), όπως για παράδειγμα επίπεδο διεπαφής χρήστη (UI Layer) ή επίπεδο λογικής παιχνιδιού (Game Logic Layer).

- Να δημιουργεί σκηνές (Scenes), για διαφορετικά επίπεδα του παιχνιδιού ή για το κεντρικό μενού.
- Να δημιουργεί και να διαχειρίζεται οντότητες (Entities).
- Να ορίζει και να προσθέτει τα επιθυμητά components σε κάθε οντότητα.
- Να ελέγχει τη ροή εκτέλεσης του παιχνιδιού.
- Να υλοποιεί τη λογική του gameplay (gameplay logic).

Παρακάτω βλέπουμε πως θα μπορούσε να μίαζει ενα παράδειγμα χρήσης του python api της μηχανής.

```
if __name__ == "__main__":
    win_prop = ge.Window_properties(APP_WIDTH, APP_HEIGHT, "test")
    app = ge.Application(win_prop)

    # Create game layer (starts inactive)
    game_layer = GameLayer()

    # Create menu layer with reference to game layer
    menu_layer = MenuLayer(game_layer)

    # Push menu layer first (will be shown first)
    app.push_layer(menu_layer)

    # Push game layer second (will be hidden initially)
    app.push_layer(gamelayer)

    app.run()
```

Στις γραμμές 2 και 3 δημιουργείται το αντικείμενο της εφαρμογής, με τον καθορισμό των απαιτούμενων χαρακτηριστικών του παραθύρου (τίτλος και διαστάσεις).

Στις γραμμές 5 έως 9 δημιουργούνται τα δύο επίπεδα (layers), συγκεκριμένα το επίπεδο μενού (Menu Layer) και το επίπεδο παιχνιδιού (Game Layer). Στη συνέχεια, στις γραμμές 11 έως 15, τα επίπεδα αυτά προστίθενται στην εφαρμογή.

Τέλος, στη γραμμή 17 εκκινείται η εκτέλεση της εφαρμογής.

Ακολούθως παρουσιάζεται ένα ενδεικτικό παράδειγμα υλοποίησης κλάσης επιπέδου (Layer class).

```
class myLayer(ge.Layer) :
    def __init__(self) :
        ge.Layer.__init__(self, "my␣layer")
        # init scane ...
    def on_attach(self):
        # create entities ...
        # create components atached to entities ...
    def on_start(self):
        # on start logic
    def on_update(self):
        # update logic (...)
```

## 4.9 Σύστημα διεπαφής χρήστη

Για να καταστεί η μηχανή φιλική προς τον τελικό χρήστη και να διευκολυνθεί η ανάπτυξη παιχνιδιών και διαδραστικών πολυμεσικών εφαρμογών, είναι απαραίτητη η παροχή ενός απλού και ευέλικτου μηχανισμού δημιουργίας στοιχείων διεπαφής χρήστη (User Interface – UI). Τα στοιχεία αυτά επιτρέπουν την αλληλεπίδραση του χρήστη με το σύστημα μέσω γραφικών αντικειμένων, όπως κουμπιά, πλαίσια και κείμενο.

### *4.9.1 ui components*

Όπως αναφέρθηκε σε προηγούμενη ενότητα του κεφαλαίου, στην αρχιτεκτονική της μηχανής υφίσταται διακριτή κατηγορία components που σχετίζονται αποκλειστικά με το γραφικό περιβάλλον διεπαφής. Τα components αυτά ορίζονται ως UI Components και λειτουργούν ανεξάρτητα από τα components που αφορούν τον κόσμο της σκηνής (world space), καθώς αποδίδονται σε ξεχωριστό επίπεδο (screen space).

Μέχρι στιγμής παρουσιάστηκε το UI_Text_Component, το οποίο επιτρέπει την απόδοση δυναμικού κειμένου. Στην ίδια κατηγορία εντάσσονται επιπλέον τα:

- Button_Component
- Panel_Component
- Interactable_Component
- Hierarchy_Component

Το ButtonComponent αποθηκεύει τα δεδομένα που αφορούν την οπτική αναπαράσταση του κουμπιού, όπως το sprite, τυχόν κείμενο που το συνοδεύει, καθώς και τις διαστάσεις του.

Το Interactable_component χρησιμοποιείται συνήθως σε συνδυασμό με το Button_component, σχηματίζοντας ένα λειτουργικό ζεύγος. Τα δεδομένα που περιλαμβάνει σχετίζονται με τη διαχείριση της αλληλεπίδρασης του χρήστη με το αντίστοιχο στοιχείο διεπαφής, όπως τα:

```
bool is_hoved;
bool is_focused;
bool is_clicked;
std::function<void()> on_click_callback;
```

Το Panel_component εχει δεδομένα οπτική αναπαράσταση του και διαστάσεων. Τέλος, το Hierarchy_component περιλαμβάνει στα δεδομένα του έναν δείκτη σε μία οντότητα που λειτουργεί ως "γονέας" (parent), καθώς και ένα std::vector δεικτών που αντιστοιχούν στις οντότητες–"παιδιά" (children). Με τον τρόπο αυτό υποστηρίζεται η ιεραρχική οργάνωση των οντοτήτων της σκηνής.

### *4.9.2 ui rendering system*

Για τον έλεγχο ύπαρξης οντοτήτων, καθώς και των αντίστοιχων components μέσα στη σκηνή, όπως έχει ήδη αναφερθεί, υπεύθυνος είναι ο μηχανισμός του RenderingSystem.

Όπως υπάρχει ένα τέτοιο σύστημα για τη διαχείριση αντικειμένων γενικού τύπου (όπως sprites, φωτισμός κ.λπ.), έτσι, για λόγους καλύτερου διαχωρισμού ευθυνών και αυξημένης επεκτασιμότητας, υλοποιήθηκε μία ξεχωριστή κλάση UIRenderingSystem, η οποία είναι υπεύθυνη αποκλειστικά για την απόδοση των στοιχείων διεπαφής χρήστη.

Η κλάση αυτή είναι υπεύθυνη για την αναζήτηση και την απόδοση αποκλειστικά των αντικειμένων που σχετίζονται με το σύστημα διεπαφής χρήστη (UI). Όπως και στην περίπτωση του Rendering_system, η μοναδική μέθοδος της κλάσης (render) έχει οριστεί ως στατική (static), ώστε να μην απαιτείται η δημιουργία αντικειμένου της κλάσης για την κλήση της.

Κατά συνέπεια, σε κάθε κλήση της μεθόδου draw() της σκηνής, εκτελείται αρχικά η μέθοδος render του Rendering_system και στη συνέχεια η αντίστοιχη μέθοδος του UI_Rendering_system, διασφαλίζοντας τον διαχωρισμό μεταξύ της απόδοσης της σκηνής και της απόδοσης των στοιχείων διεπαφής.

```
void ge::Scene::draw()
{
    Renderer_System::render(*this);
    Renderer_System_GUI::render(*this);
}
```

### *4.9.3 ui διαχειριστής (manager)*

Για τη διευκόλυνση της δημιουργίας και διαχείρισης του γραφικού περιβάλλοντος διεπαφής χρήστη (UI), υλοποιήθηκε η κλάση UI_manager.

Κατά την κατασκευή της, η κλάση δημιουργεί μία οντότητα-ρίζα (root entity), στην οποία προστίθενται ένα Hierarchy_component και ένα Transform_component. Με τον τρόπο αυτό, κάθε στοιχείο διεπαφής χρήστη που δημιουργείται στη συνέχεια καθίσταται παιδί της συγκεκριμένης οντότητας, διαμορφώνοντας μια συνεκτική ιεραρχική δομή.

Ο διαχειριστής διεπαφής (UI_manager) παρέχει, επιπλέον, έτοιμες ρουτίνες για τη δημιουργία panels, κειμένου και κουμπιών, απαλλάσσοντας τον χρήστη από την ανάγκη χειροκίνητης σύνθεσης των αντίστοιχων οντοτήτων και components.

```
class UI_Manager
{
public:
    UI_Manager(Scene &scene)
        : m_scene(scene)
    {
        m_root_entity = m_scene.create_entity();
        m_scene.add_component<UI_Transform_Component>(m_root_entity);
        m_scene.add_component<UI_Hierarchy_Component>(m_root_entity);
    }
    ~UI_Manager() = default;

    Entity get_root_entity() const { return m_root_entity; }

    Entity create_button(glm::vec2 position, glm::vec2 size, std::string text);

    Entity create_text(glm::vec2 position, glm::vec2 size, std::string text, int color_mode);

private:
    Scene &m_scene;
    Entity m_root_entity;
};
```

## 4.10 Συμπληρωματικές λειτουργικότητες

Ορισμένες επιπλέον υλοποιήσεις που κρίθηκαν απαραίτητες στο πλαίσιο της παρούσας εργασίας παρουσιάζονται στη συνέχεια.

### *4.10.1 input output*

Αρχικά, για την υποστήριξη της δημιουργίας και απεικόνισης τρισδιάστατων πλεγμάτων (3D meshes), παρότι ο βασικός προσανατολισμός της μηχανής εστιάζει κυρίως σε δισδιάστατα (2D) αντικείμενα, έχει ήδη παρουσιαστεί η κλάση HalfEdgeMesh. Η κλάση αυτή αποτελεί δομή

αποθήκευσης γεωμετρικών δεδομένων πλέγματος, προσφέροντας πλεονεκτήματα ως προς την ευελιξία και την αποδοτική διαχείριση της τοπολογίας.

Ωστόσο, προκειμένου η δομή αυτή να μπορεί να αξιοποιηθεί εύκολα από τον χρήστη, απαιτείται ένας μηχανισμός εισαγωγής δεδομένων από εξωτερικά αρχεία αναπαράστασης πλεγμάτων. Για τον σκοπό αυτό υλοποιήθηκε η κλάση IO (Input/Output), στην οποία προστέθηκε η μέθοδος read_obj(). Η μέθοδος αυτή επιτρέπει τη φόρτωση αρχείων μορφής .obj και τη μετατροπή τους σε αντικείμενα της κλάσης Mesh, καθιστώντας δυνατή την ενσωμάτωσή τους στο σύστημα απόδοσης της μηχανής.

### *4.10.2 Μαθηματικά εργαλεία*

Όπως έχει αναφερθεί σε προηγούμενη ενότητα, ένας από τους components που είναι ιδιαίτερα χρήσιμος για την εύκολη ανάπτυξη δισδιάστατων παιχνιδιών είναι το BoxColliderComponent. Ο συγκεκριμένος component αποθηκεύει ως δεδομένα ένα διάνυσμα μετατόπισης (offset) με συνιστώσες x και y, ένα μέγεθος (size), το οποίο καθορίζει την κλίμακα (scale) του συγκρουστή, καθώς και δύο λογικές σημαίες (flags), is_enabled και is_trigger.

Για να μπορεσουμε να δουμε αν δυο αντικειμενα που εχουν αυτον τον component βρισκοντε σε επαφή το ενα με το αλο χριαζόμαστε ενα μηχανισμό που να το υπολογίζει.

Για τον λόγο αυτό δημιουργήθηκε μια δομή (struct) με την ονομασία AABB2D (Axis-Aligned Bounding Box), η οποία περιλαμβάνει ως δεδομένα δύο σημεία στον χώρο: ένα ελάχιστο (min) και ένα μέγιστο (max).

```
struct AABB2D{
    glm::vec2 min;
    glm::vec2 max;
};
```

Πάνω στη δομή AABB2D υλοποιήθηκαν δύο συναρτήσεις εντοπισμού τομής (intersection). Η πρώτη ελέγχει τη σύγκρουση μεταξύ δύο αντικειμένων τύπου AABB2D, ενώ η δεύτερη ελέγχει εάν ένα σημείο στον χώρο εμπίπτει εντός ενός αντικειμένου AABB2D.

```
bool intersect(AABB2D const& aabb1, AABB2D const& aabb2)
{
    return (aabb1.min.x < aabb2.max.x && aabb1.max.x > aabb2.min.x &&
            aabb1.min.y < aabb2.max.y && aabb1.max.y > aabb2.min.y);
}

bool intersect(AABB2D const& aabb, glm::vec2 const& point)
{
    return (point.x >= aabb.min.x && point.x <= aabb.max.x &&
            point.y >= aabb.min.y && point.y <= aabb.max.y);
}
```

Με την ολοκλήρωση της παρουσίασης της αρχιτεκτονικής και των βασικών υποσυστημάτων της μηχανής, έγινε σαφές πώς υλοποιούνται οι κύριοι μηχανισμοί που υποστηρίζουν τη λειτουργία της, όπως το σύστημα απόδοσης γραφικών, η αρχιτεκτονική Entity Component System, καθώς και τα επιμέρους components που επιτρέπουν την αναπαράσταση αντικειμένων, φωτισμού, ήχου και στοιχείων διεπαφής χρήστη. Η ανάλυση αυτή επικεντρώθηκε κυρίως στις τεχνικές επιλογές και στις δομές λογισμικού που συγκροτούν τον πυρήνα της μηχανής.

Στο επόμενο κεφάλαιο η προσέγγιση μετατοπίζεται από την εσωτερική υλοποίηση προς την πρακτική αξιοποίηση της μηχανής. Πιο συγκεκριμένα, παρουσιάζεται ο τρόπος με τον οποίο τα εργαλεία και τα συστήματα που περιγράφηκαν μπορούν να χρησιμοποιηθούν για την ανάπτυξη δισδιάστατων παιχνιδιών. Μέσα από παραδείγματα και περιγραφή της διαδικασίας δημιουργίας μιας απλής εφαρμογής, αναδεικνύεται η ροή εργασίας που ακολουθεί ο χρήστης της μηχανής, καθώς και ο τρόπος με τον οποίο τα επιμέρους components συνεργάζονται για τη δημιουργία ενός ολοκληρωμένου παιχνιδιού.

# ΚΕΦΑΛΑΙΟ 5 Εφαρμογές χρήσης της μηχανής

Η ανάπτυξη μιας μηχανής παιχνιδιών δεν περιορίζεται μόνο στον σχεδιασμό και την υλοποίηση των επιμέρους υποσυστημάτων της, αλλά επεκτείνεται και στον τρόπο με τον οποίο αυτά μπορούν να αξιοποιηθούν στην πράξη για τη δημιουργία εφαρμογών. Στο προηγούμενο κεφάλαιο παρουσιάστηκε αναλυτικά η αρχιτεκτονική της μηχανής, καθώς και τα βασικά συστήματα που την αποτελούν, όπως το σύστημα απόδοσης γραφικών, το μοντέλο Entity–Component–System, τα components που περιγράφουν τα αντικείμενα της σκηνής, καθώς και τα υποσυστήματα εισόδου, ήχου και διεπαφής χρήστη.

Στο παρόν κεφάλαιο παρουσιάζεται η μηχανή παιχνιδιών από την πλευρά του χρήστη. Συγκεκριμένα, περιγράφεται το περιβάλλον διεπαφής του χρήστη με τη μηχανή, καθώς και ο τρόπος με τον οποίο μπορεί να δομηθεί μια εφαρμογή που αξιοποιεί τις δυνατότητές της.

Σκοπός του κεφαλαίου είναι αφενός η ανάδειξη των λειτουργιών που παρουσιάστηκαν στο προηγούμενο κεφάλαιο και αφετέρου η παρουσίαση του τρόπου χρήσης τους, δίνοντας έμφαση στην απλότητα και την ευχρηστία του συστήματος. Επιπλέον, μέσω της υλοποίησης ορισμένων μικρών παιχνιδιών θα επιδιωχθεί να αναδειχθεί και η επάρκεια της μηχανής, καθώς τα παραδείγματα αυτά ενσωματώνουν όλα τα βασικά στοιχεία που απαιτούνται για την ανάπτυξη ενός παιχνιδιού.

Προχωρώντας στις επόμενες ενότητες, θα παρουσιαστεί η κατασκευή τριών παιχνιδιών με χρήση της μηχανής που αναπτύχθηκε στο πλαίσιο της παρούσας εργασίας. Το πρώτο παιχνίδι θα είναι ένα δισδιάστατο παιχνίδι, στο οποίο ο χαρακτήρας κινείται οριζόντια (δεξιά και αριστερά). Το δεύτερο παιχνίδι αποτελεί επίσης δισδιάστατη υλοποίηση τύπου top-down, όπου ο χαρακτήρας διαθέτει ελευθερία κίνησης προς όλες τις κατευθύνσεις. Τέλος, θα παρουσιαστεί η ανάπτυξη ενός παιχνιδιού τύπου space shooter, το οποίο υλοποιείται εξολοκλήρου με τη χρήση τεχνητής νοημοσύνης.

## 5.1 Δημιουργία δισδιάστατου παιχνιδίου

Το πρώτο παιχνίδι που θα υλοποιηθεί με χρήση της μηχανής είναι ένα απλό δισδιάστατο παιχνίδι, το οποίο όμως θα αξιοποιεί όσο το δυνατόν περισσότερα από τα συστήματα που παρέχει η μηχανή. Στους παρακάτω πίνακες 5.1, 5.2, 5.3 καταγράφονται αρχικά κάποια βασικά στατιστικά του παιχνιδιού ως προς τον κώδικα (γραμμές κώδικα και πλήθος αρχείων), ενώ στη συνέχεια παρουσιάζεται ένας αναλυτικός πίνακας με τις κλάσεις και τις συναρτήσεις της μηχανής παιχνιδιών που χρησιμοποιήθηκαν για την κατασκευή του παιχνιδιού.

Ειδικότερα, από τον Πίνακα 5.3 μπορούμε να εξάγουμε το συμπέρασμα της σημασίας ύπαρξης της μηχανής για την εύκολη κατασκευή του παιχνιδιού, καθώς η λίστα των συναρτήσεων και των κλάσεων είναι μεγάλη και, για καθεμία από αυτές, η αντικατάστασή της, σε περίπτωση μη χρήσης της μηχανής, θα συνεπαγόταν μερικές δεκάδες και, σε ορισμένες περιπτώσεις, ακόμη και εκατοντάδες επιπλέον γραμμές κώδικα.

Πίνακας 5.1: Στατιστικά παιχνιδιού

| **Μετρική** | **Μέγεθος** |
|---|---|
| Python Αρχεία | 9 |
| Πλύθος γραμμών κώδικα | 1166 |
| Κλάσεις μέσα στο παιχνίδι | 7 |

Πίνακας 5.2: Γραμμές ανά αρχείο python

| **Αρχείο** | **Γραμμές** |
|---|---|
| player.py | 319 |
| enemy.py | 187 |
| game_logic.py | 167 |
| scane_entities.py | 153 |
| audio_manager.py | 105 |
| end_game.py | 86 |
| main_menu.py | 76 |
| main.py | 57 |
| constants.py | 16 |

Πίνακας 5.3: Κατανομή χρήσης κλάσεων και συναρτήσεων ανά κλάση της βιβλιοθήκης της μηχανής παιχνιδιών (ge)

| **Engine Class / API** | **Methods / Members Used** |
|---|---|
| ge.Application | push_layer(), run() |
| ge.Window_properties | __init__(width, height, title) |
| ge.Layer | __init__(name), on_attach(), on_start(), on_update(), quit() |

Συνέχεια στην επόμενη σελίδα

Πίνακας 5.3 -- Συνέχεια από την προηγούμενη σελίδα

| **Engine Class / API** | **Methods / Members Used** |
|---|---|
| ge.Scene | create_entity(), destroy_entity(), update(dt), draw(), on_start(), add_sprite_component(), add_transform_component(), add_box_colider_2d_component(), add_light_2d_component(), add_audio_component(), get_*() component accessors, get_active_camera() |
| ge.Sprite_Animation_t | __init__(path, cols, rows, frame_time, frames), set_interupted_loop(), set_manual_update() |
| SpriteComponent | set_layer_id(), add_animation(), play_animation(), update(frame) |
| TransformComponent | position(), position_x(), position_y(), rotation(), scale(), scale_x() |
| BoxColider2DComponent | offset(), size(), show_box(), is_trigger |
| Light2DComponent | position(), color(), radius(), intensity_type(), angle_vector(), direction(), layer_id(), on() |
| AudioComponent | set_looping(), set_gain(), load_wav(), play_on_start(), is_playing(), play(), attach_buffer() |
| Camera | set_position(), position_x() |
| ge.UI_Manager | __init__(scene), add_button(), add_text() |
| UITextComponent | set_text() |
| UITransformComponent | position_x(), position_y() |
| UIInteractableComponent | set_on_click_callback() |
| Entity | is_valid() |
| ge.Input | is_key_pressed(key) |
| ge.Key | F, D, A, W, Space |

Συνέχεια στην επόμενη σελίδα

Πίνακας 5.3 -- Συνέχεια από την προηγούμενη σελίδα

| **Engine Class / API** | **Methods / Members Used** |
|---|---|
| ge.Intensity_type_e | Gaussian, SmoothStep |
| ge.compute_AABB2D() | Free function |
| ge.intersect() | Free function |

Στη συνέχεια παρουσιάζεται αρχικά μια συνοπτική περιγραφή του παιχνιδιού. Έπειτα αναλύεται η δομή της σκηνής και τα βασικά αντικείμενα που τη συνθέτουν, καθώς και η χρήση ορισμένων components. Τέλος, παρουσιάζεται η υλοποίηση της λογικής του παιχνιδιού.

### *5.1.1 Περιγραφή του παιχνιδιού*

Το παιχνίδι ξεκινά με ένα στατικό μενού. Στο μενού αυτό παρουσιάζονται δύο επιλογές στον χρήστη: η πρώτη επιτρέπει την έξοδο από το παιχνίδι (Quit), ενώ η δεύτερη την έναρξη του παιχνιδιού (Start).

Όπως έχει ήδη αναφερθεί, το παιχνίδι είναι δισδιάστατο. Υπάρχει ένας βασικός χαρακτήρας (player), ο οποίος μπορεί να κινείται δεξιά και αριστερά στον χάρτη. Επιπλέον, έχει τη δυνατότητα να επιτίθεται σε τυχόν αντιπάλους, καθώς και να πραγματοποιεί άλματα.

Σχήμα 5.1: Βασικός χαρακτήρας (Sprite)

Όλες οι παραπάνω ενέργειες πραγματοποιούνται μέσα σε μία σκηνή (χάρτη), ενώ κατά τη διάρκεια της κίνησης του χαρακτήρα η κάμερα τον ακολουθεί, διατηρώντας τον στο οπτικό πεδίο του χρήστη. Επιπλέον, ο χαρακτήρας διαθέτει μία μπάρα ζωής, η οποία αποτελείται από οκτώ μονάδες ζωής. Κάθε φορά που δέχεται επίθεση από κάποιον αντίπαλο, αφαιρείται μία μονάδα από το σύνολο της ζωής του.

Στη σκηνή, σε διάφορα σημεία του χάρτη, είναι τοποθετημένοι αντίπαλοι οι οποίοι κινούνται περιοδικά δεξιά και αριστερά. Κάθε αντίπαλος διαθέτει επίσης μπάρα ζωής αποτελούμενη από τέσσερις μονάδες. Κάθε φορά που ένας αντίπαλος έρχεται σε επαφή με τον παίκτη, αφαιρείται μία μονάδα ζωής από τον παίκτη. Παράλληλα, ο παίκτης έχει τη δυνατότητα να επιτεθεί στους αντιπάλους, όπου κάθε επιτυχημένο χτύπημα αφαιρεί μία μονάδα ζωής από τον αντίπαλο. Όταν η μπάρα ζωής ενός αντιπάλου μηδενιστεί, ο αντίπαλος αφαιρείται από τη σκηνή. Ο παίκτης έχει επίσης τη δυνατότητα να προσπεράσει έναν αντίπαλο και να επιστρέψει αργότερα για να

Σχήμα 5.2: Αντίπαλος (Sprite)

τον αντιμετωπίσει, καθώς η κίνηση του χαρακτήρα δεν περιορίζεται σε μία μόνο κατεύθυνση, καθώς μπορεί να μετακινείται ελεύθερα προς τα εμπρός και προς τα πίσω, όσες φορές επιθυμεί, εντός των ορίων του χάρτη.


Σχήμα 5.3: Στιγμιότυπο από το δισδιάστατο παιχνίδι, στο οποίο απεικονίζεται ο παίκτης ενώ φαίνεται και το εφέ του φωτισμού.

Στη σκηνή, σε διάφορα σημεία του χάρτη, υπάρχουν επίσης διάσπαρτα και άλλα αντικείμενα. Συγκεκριμένα, εμφανίζονται λάμπες και πυρσοί με καθαρά διακοσμητικό ρόλο. Η μοναδική τους επίδραση στο παιχνίδι σχετίζεται με τον φωτισμό της σκηνής, καθώς επηρεάζουν τα χρώματα των γύρω αντικειμένων εντός ενός συγκεκριμένου εύρους, δημιουργώντας μια κιτρινοκόκκινη ατμόσφαιρα. Επιπλέον, στον χάρτη υπάρχουν μικρά αστέρια τοποθετημένα σε διάφορα σημεία στον αέρα. Σε αντίθεση με τα προηγούμενα αντικείμενα, τα αστέρια έχουν λειτουργικό ρόλο στο παιχνίδι. Ο παίκτης μπορεί να τα συλλέξει και, για κάθε αστέρι που συλλέγεται, αναπληρώνεται μία χαμένη μονάδα ζωής. Όπως και στην περίπτωση των αντιπάλων, ο παίκτης έχει τη δυνατότητα να προσπεράσει ένα αστέρι και να επιστρέψει αργότερα για να το συλλέξει.

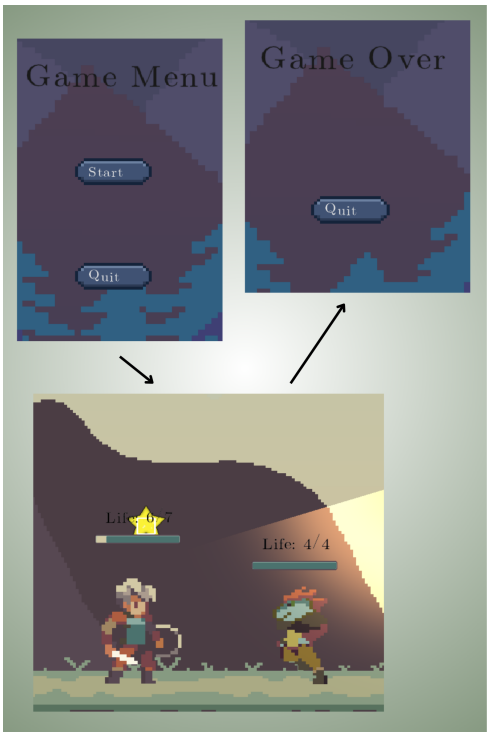
Το παιχνίδι ολοκληρώνεται με επιτυχία όταν ο παίκτης καταφέρει να εξουδετερώσει όλους τους αντιπάλους του χωρίς να εξαντληθούν οι μονάδες ζωής του. Αντίθετα, σε περίπτωση που οι μονάδες ζωής του παίκτη μηδενιστούν πριν καταφέρει να εξουδετερώσει όλους τους αντιπάλους, το παιχνίδι τερματίζεται με αποτυχία.

Μετά την ολοκλήρωση του παιχνιδιού εμφανίζεται ένα τελικό μενού, το οποίο περιλαμβάνει ένα μοναδικό κουμπί εξόδου (Exit).

### *5.1.2 Διάρθρωση της εφαρμογής*

Προχωρούμε λοιπόν στην υλοποίηση του παιχνιδιού που περιγράφηκε παραπάνω. Αρχικά, το παιχνίδι χωρίζεται σε τρία layers. Το πρώτο layer αποτελεί το στατικό μενού εισόδου στο παιχνίδι. Όπως έχει ήδη περιγραφεί, διαθέτει δύο κουμπιά: ένα για έξοδο από το παιχνίδι και ένα για την εκκίνηση του παιχνιδιού. Το δεύτερο layer αντιστοιχεί στο ίδιο το παιχνίδι. Περιλαμβάνει τα διάφορα αντικείμενα του παιχνιδιού, τον χαρακτήρα, και εκεί υλοποιείται η βασική λογική του παιχνιδιού. Τέλος, υπάρχει ένα τρίτο στατικό layer, το οποίο αντιστοιχεί στο τελικό μενού. Σε αυτό εμφανίζεται το αποτέλεσμα του παιχνιδιού (νίκη ή ήττα), καθώς και ένα κουμπί εξόδου.

Κάθε layer δημιουργεί και τη δική του σκηνή, ενώ διατηρεί μια εξάρτηση από το προηγούμενο layer, ώστε να διασφαλίζεται η σωστή ροή από σκηνή σε σκηνή. Τόσο η δημιουργία των layers και των σκηνών όσο και η κατασκευή των στοιχείων του γραφικού περιβάλλοντος χρήστη (UI), όπως κουμπιά, κείμενα και λοιπά διαδραστικά στοιχεία, πραγματοποιούνται μέσω των αντίστοιχων κλάσεων της Python βιβλιοθήκης που παρέχει η μηχανή.




Σχήμα 5.4: Βασικό μενού, κύριο μέρος παιχνιδιού και τελικό μενού του δισδιάστατου παιχνιδιού.

Στα πλαίσια του παιχνιδιού αυτού δημιουργήθηκαν τέσσερις βασικές κλάσεις. Η πρώτη κλάση είναι η Player (Παίκτης). Σε αυτή φορτώνονται όλα τα animations του χαρακτήρα, η θέση και το μέγεθός του, η μπάρα ζωής, καθώς και οι βασικές λειτουργίες του, όπως χτύπημα και άλμα. Η δεύτερη κλάση είναι η Enemy (Αντίπαλος). Όπως και στην κλάση του παίκτη,

εδώ διαχειρίζεται τα αντίστοιχα animations, οι θέσεις, η κίνηση και η τεχνητή νοημοσύνη του αντιπάλου.

Στη συνέχεια υπάρχει η κλάση SceneObject (Αντικείμενα Σκηνής). Σε αυτή φορτώνονται όλα τα αντικείμενα που αναφέρθηκαν στην προηγούμενη ενότητα, όπως τα αστέρια ζωής, οι λάμπες και οι πυρσοί, καθώς και οι θέσεις τους και ο τρόπος ανανέωσης αυτών μέσα στη σκηνή.

Τέλος, υπάρχει η κλάση AudioManager (Διαχειριστής Ήχου), στην οποία φορτώνονται όλοι οι ήχοι που χρησιμοποιούνται κατά τη διάρκεια του παιχνιδιού. Όλες οι παραπάνω κλάσεις αξιοποιούνται εντός του παιχνιδιού για να διευκολύνουν την υλοποίηση, να αυξήσουν τη διαχειρισιμότητα του κώδικα και να επιτρέψουν την εύκολη επέκταση της λειτουργικότητας.

Για την υλοποίηση όλων των παραπάνω απαιτείται η χρήση ορισμένων components, όπως αυτά που περιγράφηκαν αναλυτικά στο προηγούμενο κεφάλαιο και αποτελούν μέρος του συστήματος ECS (Entity–Component–System).

Αρχικά, για τον παίκτη (Player) απαιτείται ένα SpriteComponent, το οποίο περιέχει τα animations των καταστάσεων idle, run, jump και hit. Επιπλέον, απαιτείται ένα TransformComponent, το οποίο καθορίζει τη θέση, την κλίμακα και τον προσανατολισμό του χαρακτήρα μέσα στη σκηνή.

Παράλληλα, χρησιμοποιείται ένα BoxColliderComponent, μέσω του οποίου είναι δυνατή η ανίχνευση συγκρούσεων, όπως για παράδειγμα η επαφή του χαρακτήρα με έναν αντίπαλο ή με κάποιο άλλο αντικείμενο της σκηνής (π.χ. ένα αστέρι ζωής).

Τέλος, για την απεικόνιση της μπάρας ζωής του παίκτη χρησιμοποιούνται επιπλέον ένα SpriteComponent, ένα TransformComponent και ένα TextComponent.

Για τον αντίπαλο (Enemy) χρησιμοποιείται αντίστοιχη δομή components. Συγκεκριμένα, απαιτούνται SpriteComponent και TransformComponent για την αναπαράσταση και τη θέση του στη σκηνή, καθώς και BoxColliderComponent για τον έλεγχο συγκρούσεων. Επιπλέον, καθώς και οι αντίπαλοι διαθέτουν μπάρα ζωής, χρησιμοποιείται και εδώ συνδυασμός TransformComponent, SpriteComponent και TextComponent για την απεικόνισή της.

Επιπλέον, για τα αντικείμενα της σκηνής απαιτούνται SpriteComponent και TransformComponent, τα οποία χρησιμοποιούνται για την απεικόνιση και την τοποθέτησή τους στον χώρο. Στην περίπτωση των αστεριών ζωής απαιτείται επιπρόσθετα και BoxColliderComponent, ώστε να είναι δυνατή η ανίχνευση σύγκρουσης με τον παίκτη και η ενεργοποίηση του μηχανισμού συλλογής τους.

Αντίστοιχα, για τα αντικείμενα φωτισμού, όπως οι λάμπες και οι πυρσοί, εκτός από SpriteComponent και TransformComponent, χρησιμοποιείται και LightComponent, το οποίο είναι υπεύθυνο για την προσομοίωση της φωτεινής πηγής και την οπτική επίδραση που αυτή προκαλεί στο περιβάλλον της σκηνής.

Τέλος, υπάρχουν και τα αντικείμενα διεπαφής με τον χρήστη στα δύο μενού του παιχνιδιού. Σε αυτή την περίπτωση χρησιμοποιούνται ButtonComponent, τα οποία, όπως έχει ήδη αναφερθεί στο προηγούμενο κεφάλαιο, διαχειρίζονται τόσο την οπτική αναπαράσταση και τα animations των κουμπιών όσο και τη θέση και το μέγεθός τους μέσα στη διεπαφή χρήστη.

### *5.1.3 Υλοποίηση λογικής του παιχνιδιού*

Έχοντας υλοποιήσει όλα τα παραπάνω, το επόμενο βήμα είναι η τελική σύνθεσή τους, ώστε να προκύψει ένα ολοκληρωμένο αποτέλεσμα.

Όπως αναφέρθηκε προηγουμένως, η πρώτη και η τελευταία σκηνή του παιχνιδιού είναι στατικές και η μοναδική τους λειτουργικότητα παρέχεται μέσω των κουμπιών που περιέχουν. Για τον λόγο αυτό υλοποιούνται κατάλληλες συναρτήσεις ανάκλησης (callbacks) για κάθε κουμπί. Το κουμπί Exit εκτελεί την προφανή λειτουργία του τερματισμού της εφαρμογής, ενώ το κουμπί Start αρχικοποιεί τη μεταβλητή startgame, η οποία σηματοδοτεί τη μετάβαση στην επόμενη σκηνή.

Η σκηνή του παιχνιδιού είναι εκείνη που συγκεντρώνει το μεγαλύτερο μέρος της λειτουργικότητας. Αρχικά, ορίζονται και αρχικοποιούνται διάφορες σταθερές του παιχνιδιού, όπως το μέγεθος της γραμματοσειράς, η αρχική θέση του παίκτη, το πλήθος και οι αρχικές θέσεις των αντιπάλων, καθώς και το πλήθος και οι θέσεις των στατικών αντικειμένων της σκηνής. Στη συνέχεια αρχικοποιούνται τα αντίστοιχα αντικείμενα των κλάσεων Player, Enemy, SceneObject, καθώς και ο AudioManager.

Κατά τη διάρκεια κάθε ενημέρωσης (update) της σκηνής πραγματοποιείται αρχικά έλεγχος για συνθήκες νίκης ή ήττας, δηλαδή ελέγχεται αν υπάρχουν ακόμη ενεργοί αντίπαλοι ή αν οι μονάδες ζωής του παίκτη έχουν μηδενιστεί.

Στη συνέχεια, για κάθε αντίπαλο που βρίσκεται ακόμη ενεργός στη σκηνή, πραγματοποιείται έλεγχος απόστασης και σύγκρουσης με τον παίκτη (collision detection). Σε περίπτωση επαφής, αφαιρείται μία μονάδα ζωής από τον παίκτη. Αντίστοιχα, πραγματοποιείται έλεγχος και για την περίπτωση κατά την οποία ο παίκτης εκτελεί ενέργεια επίθεσης· εφόσον το πλήκτρο επίθεσης έχει ενεργοποιηθεί και υπάρχει σύγκρουση με έναν αντίπαλο, αφαιρείται μία μονάδα ζωής από τον αντίπαλο.

Τέλος, εκτελείται παρόμοιος έλεγχος και για τα αστέρια ζωής που βρίσκονται στη σκηνή. Σε περίπτωση σύγκρουσης μεταξύ του παίκτη και ενός τέτοιου αντικειμένου, ο παίκτης λαμβάνει μία επιπλέον μονάδα ζωής.

Σε κάθε λειτουργία του παιχνιδιού που συνοδεύεται από ηχητική ανατροφοδότηση, κατά τη διαδικασία ενημέρωσης (update) καλείται και ο αντίστοιχος ήχος μέσω του συστήματος διαχείρισης ήχου. Για παράδειγμα, αναπαράγονται κατάλληλα ηχητικά εφέ κατά την εκτέλεση ενεργειών όπως το χτύπημα και το άλμα του χαρακτήρα, καθώς και σε περιπτώσεις απώλειας ή ανάκτησης μονάδων ζωής. Επιπλέον, αντίστοιχα ηχητικά εφέ ενεργοποιούνται κατά την ολοκλήρωση του παιχνιδιού, τόσο στην περίπτωση νίκης όσο και στην περίπτωση ήττας.

## 5.2 Δημιουργία (Top down) παιχνιδίου

Στην παρούσα υποενότητα παρουσιάζεται ο σχεδιασμός και η υλοποίηση ενός ακόμη παιχνιδιού, αυτή τη φορά ενός δισδιάστατου παιχνιδιού τύπου top-down. Η γενική δομή παραμένει αντίστοιχη με αυτή που παρουσιάστηκε προηγουμένως.

Αρχικά, παρουσιάζεται μια γενική περιγραφή του παιχνιδιού. Στη συνέχεια αναλύεται η δομή της σκηνής και των επιμέρους αντικειμένων της, καθώς και η χρήση των αντίστοιχων components και ορισμένων επιπλέον κλάσεων που κρίθηκαν απαραίτητες. Τέλος, παρουσιάζεται η υλοποίηση της λογικής του παιχνιδιού.

Στους παρακάτω πίνακες 5.4, 5.5, 5.6 καταγράφονται αρχικά κάποια βασικά στατιστικά του παιχνιδιού ως προς τον κώδικα (γραμμές κώδικα και πλήθος αρχείων), ενώ στη συνέχεια παρουσιάζεται ένας αναλυτικός πίνακας με τις κλάσεις και τις συναρτήσεις της μηχανής παιχνιδιών που χρησιμοποιήθηκαν για την κατασκευή του παιχνιδιού. Όπως και στο προηγούμενο παιχνίδι, έτσι και εδώ παρουσιάζουμε αρχικά σε μια σειρά από πίνακες κάποια βασικά στατιστικά για το παιχνίδι.

Όπως και προηγουμένως, ιδιαίτερη σημασία έχει το μέγεθος του Πίνακα 5.6, καθώς αποκαλύπτει τη διαφορά που προκαλεί η ύπαρξη και χρήση της μηχανής.

Πίνακας 5.4: Στατιστικά παιχνιδιού

| **Μετρική** | **Μέγεθος** |
|---|---|
| Python Αρχεία | 10 |
| Πλύθος γραμμών κώδικα | 2121 |
| Κλάσεις μέσα στο παιχνίδι | 7 |

Πίνακας 5.5: Γραμμές ανά αρχείο python

| **Αρχείο** | **Γραμμές** |
|---|---|
| game_layer.py | 936 |
| constants.py | 360 |
| tile_map.py | 297 |
| enemy.py | 150 |
| main.py | 108 |
| collision.py | 82 |
| end_game_layer.py | 84 |
| player.py | 70 |
| __init__.py | 20 |
| run_game.py | 14 |

Πίνακας 5.6: Κατανομή χρήσης κλάσεων και συναρτήσεων ανά κλάση της βιβλιοθήκης της μηχανής παιχνιδιών (ge)

| **Engine Class / API** | **Methods / Members Used** |
|---|---|
| ge.Application | push_layer(), run() |
| ge.Window_properties | __init__(width, height, title) |
| ge.Layer | __init__(name), on_attach(), on_start(), on_update(), quit() |
| ge.Scene | create_entity(), create_default_camera(), update(dt), draw(), on_start(), add_sprite_component(), add_transform_component(), add_audio_component(), get_sprite_component(), get_transform_component(), get_audio_component(), get_ui_text_component(), get_ui_transform_component(), get_ui_interactable_component(), get_active_camera() |
| ge.Sprite_Animation_t | __init__(path, cols, rows, frame_time, frames), set_interupted_loop(), set_manual_update() |
| SpriteComponent (via Scene) | set_layer_id(), add_animation(), play_animation(), update(frame) |
| TransformComponent (via Scene) | position(), position_x(), position_y(), rotation(), scale(), scale_x(), scale_y() |
| Camera (via Scene) | set_position(), position_x(), position_y() |
| ge.UI_Manager | __init__(scene), add_button(), add_text() |
| UITextComponent (via Scene) | set_text() |
| UITransformComponent (via Scene) | position_x(), position_y() |
| UIInteractableComponent (via Scene) | set_on_click_callback() |
| AudioComponent (via Scene) | set_looping(), set_gain(), load_wav(), play_on_start(), is_playing(), play(), attach_buffer() |

Συνέχεια στην επόμενη σελίδα

Πίνακας 5.6 -- Συνέχεια από την προηγούμενη σελίδα

| **Engine Class / API** | **Methods / Members Used** |
|---|---|
| ge.Input | is_key_pressed(key) |
| ge.Key | W, A, S, D, Space, LeftControl, RightControl |

### *5.2.1 Περιγραφή του παιχνιδιού*

Αρχικά, όπως προϊδεάζει και ο τίτλος της ενότητας, το συγκεκριμένο παιχνίδι είναι ένα δισδιάστατο παιχνίδι τύπου top-down. Με άλλα λόγια, όλοι οι χαρακτήρες του παιχνιδιού έχουν τη δυνατότητα να κινούνται προς όλες τις κατευθύνσεις (πάνω, κάτω, δεξιά, αριστερά), ενώ η κάμερα παρατηρεί τη σκηνή από μία άνωθεν οπτική γωνία, με ελαφρά κλίση.

Πιο συγκεκριμένα, το παιχνίδι ξεκινά με ένα αρχικό μενού, στο οποίο εμφανίζεται ο τίτλος του παιχνιδιού και παρέχεται ένα κουμπί εκκίνησης (Start). Αντίστοιχα, στο τέλος του παιχνιδιού εμφανίζεται ένα τελικό μενού, στο οποίο αναγράφεται το αποτέλεσμα (νίκη ή ήττα), καθώς και ένα κουμπί εξόδου (Quit).

Στο κύριο μέρος του παιχνιδιού, ο παίκτης βρίσκεται πάνω σε ένα μικρό νησί και μπορεί να κινηθεί αποκλειστικά εντός των ορίων του. Πάνω στο νησί βρίσκονται διάσπαρτοι αντίπαλοι, οι οποίοι αναπαρίστανται ως μπλε χαρακτήρες. Οι αντίπαλοι κινούνται επίσης μόνο εντός του νησιού, ακολουθώντας τυχαίες κατευθύνσεις. Κατά διαστήματα, οι αντίπαλοι μεταβαίνουν σε μία «ενισχυμένη» κατάσταση, κατά την οποία αποκτούν κόκκινο χρώμα και κινούνται με αυξημένη ταχύτητα.

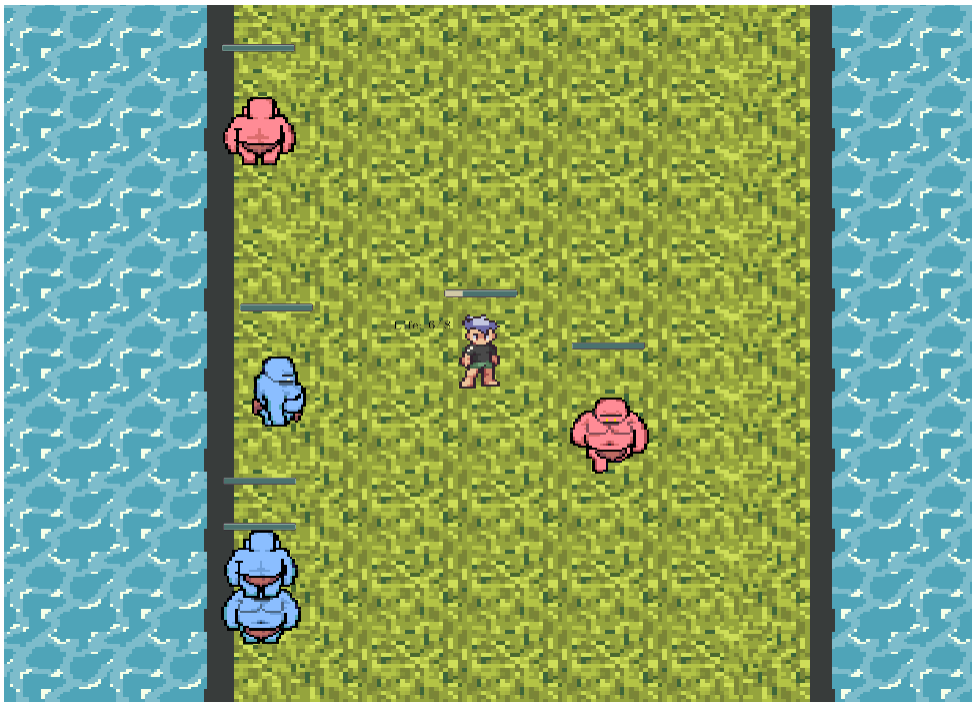
Σκοπός του παίκτη είναι να εξουδετερώσει όλους τους αντιπάλους που βρίσκονται στη σκηνή. Ο παίκτης, προσεγγίζοντας τους αντιπάλους, έχει τη δυνατότητα να τους επιτεθεί χρησιμοποιώντας το σπαθί του (μέσω του πλήκτρου space). Ωστόσο, όταν οι αντίπαλοι βρίσκονται στην κόκκινη μορφή τους, η επαφή μαζί τους προκαλεί ζημιά στον παίκτη, οδηγώντας σε απώλεια μονάδων ζωής. Κατά συνέπεια, ο στόχος του παιχνιδιού είναι η εξουδετέρωση όλων των αντιπάλων χωρίς την πλήρη εξάντληση των μονάδων ζωής του παίκτη.

### *5.2.2 Διάρθρωση της εφαρμογής και υλοποίηση λογικής του παιχνιδιού*

Συνεχίζουμε με την υλοποίηση του παιχνιδιού. Όπως και στο προηγούμενο παράδειγμα, η εφαρμογή οργανώνεται σε τρία layers. Κάθε layer διαθέτει τη δική του σκηνή, ενώ διατηρεί και εξαρτήσεις από το προηγούμενο, ώστε να εξασφαλίζεται η ομαλή ροή της εφαρμογής.

Το πρώτο layer αντιστοιχεί στο στατικό μενού εισόδου. Το δεύτερο αποτελεί τον βασικό κορμό του παιχνιδιού, στον οποίο υλοποιείται και η κύρια λογική του. Τέλος, το τρίτο layer αντιστοιχεί στο στατικό μενού εξόδου, όπου παρουσιάζεται και το αποτέλεσμα του παιχνιδιού (νίκη ή ήττα).

Τα βασικά entities της σκηνής είναι ο παίκτης (Player), το αντίστοιχο entity για τη μπάρα



Σχήμα 5.5: Στιγμιότυπο από το “top-down” παιχνίδι.

ζωής του, τα entities των αντιπάλων (Enemies) μαζί με τα αντίστοιχα entities που αναπαριστούν τη μπάρα ζωής τους, καθώς και το entity της κάμερας. Όλα τα παραπάνω entities συνδυάζονται με τα απαραίτητα components (όπως Transform, Sprite κ.λπ.), ώστε να επιτυγχάνεται η πλήρης λειτουργικότητά τους.

Σχήμα 5.6: Βασικός χαρακτήρας.

Για την καλύτερη διαχείριση του παίκτη και των αντιπάλων υλοποιήθηκαν αντίστοιχες κλάσεις Player και Enemy. Η κλάση Player διαχειρίζεται τις βασικές λειτουργίες του χαρακτήρα, όπως την κίνηση και την επίθεση, καθώς και τη μπάρα ζωής και τα όρια εντός των οποίων επιτρέπεται να κινείται. Αντίστοιχα, η κλάση Enemy διαχειρίζεται τη ζωή του αντιπάλου, τα επιτρεπτά όρια κίνησης, καθώς και την τυχαία περιπλάνησή του στον χώρο. Επιπλέον, υλοποιεί τη μετάβαση μεταξύ των διαφορετικών καταστάσεων του αντιπάλου (κανονική – μπλε και ενισχυμένη – κόκκινη).

Για τη δημιουργία και τη διαχείριση του νησιού, πάνω στο οποίο κινούνται όλοι οι χαρακτήρες, υλοποιήθηκε η κλάση TileMap. Η κλάση αυτή αναπτύχθηκε στο επίπεδο της Python (Python API), αν και θα μπορούσε εξίσου να ενσωματωθεί στον πυρήνα της μηχανής (C++), καθώς αποτελεί συχνά χρησιμοποιούμενη λειτουργικότητα σε δισδιάστατα παιχνίδια. Η TileMap δέχεται ως είσοδο έναν πίνακα, τα στοιχεία του οποίου λειτουργούν ως σημαίες (flags) που

Σχήμα 5.7: Αντίπαλος κόκκινος.

αντιστοιχούν σε συγκεκριμένα tiles. Με τον τρόπο αυτό, αξιοποιώντας την τοπολογία (θέση) κάθε tile, καθίσταται δυνατή η δημιουργία της δισδιάστατης σκηνής.

Σχήμα 5.8: Αντίπαλος μπλέ.

Με τη χρήση της παραπάνω κλάσης καθίσταται δυνατή η δημιουργία του νησιού. Επιπλέον, στα entities του νησιού που αντιστοιχούν σε περιοχές νερού μπορεί να προστεθεί ένα ColliderComponent.

Με τον τρόπο αυτό είναι δυνατός ο εντοπισμός περιπτώσεων κατά τις οποίες ένας χαρακτήρας επιχειρεί να εξέλθει εκτός των ορίων του νησιού, επιτρέποντας έτσι τον περιορισμό της κίνησής του εντός της επιτρεπτής περιοχής.

## 5.3 Δημιουργία παιχνιδιού με χρήση τεχνητής νοημοσύνης

Στην παρούσα ενότητα επιχειρείθηκε μια διαφορετική προσέγγιση. Συγκεκριμένα, η ανάπτυξη ενός ακόμη παιχνιδιού, αυτή τη φορά εξολοκλήρου με τη χρήση τεχνητής νοημοσύνης, αποφεύγοντας οποιαδήποτε άμεση παρέμβαση στον πηγαίο κώδικα. Η υλοποίηση πραγματοποιήθηκε αποκλειστικά μέσω της διατύπωσης κατάλληλων προτροπών (prompts). Η διαδικασία ανάπτυξης του παιχνιδιού πραγματοποιήθηκε με τη χρήση του περιβάλλοντος ανάπτυξης που παρέχει το Cursor και του γλωσσικού μοντέλου Claude 3.5 Sonnet. Στη συνέχεια, το ίδιο πείραμα επαναλήφθηκε δύο ακόμη φορές, χρησιμοποιώντας το ίδιο αρχικό prompt, αλλά διαφορετικά μοντέλα τεχνητής νοημοσύνης, συγκεκριμένα τα Claude Haiku 4.5 και Big Pickle της OpenCode.

### *5.3.1 Περιγραφή του παιχνιδιού*

Αρχικά πραγματοποιήθηκε η επιλογή του παιχνιδιού, καθώς και ο καθορισμός της βασικής (αρχικής) λογικής του. Στη συγκεκριμένη περίπτωση επιλέχθηκε η ανάπτυξη ενός παιχνιδιού τύπου space shooter, στο οποίο ο παίκτης χειρίζεται ένα διαστημόπλοιο, το οποίο κινείται οριζόντια (δεξιά και αριστερά).

Στη σκηνή εμφανίζονται εχθρικά διαστημόπλοια, τα οποία κινούνται προς τον παίκτη, και ο στόχος του είναι να τα αποφεύγει. Παράλληλα, ο παίκτης έχει τη δυνατότητα να επιτίθεται, εκτοξεύοντας βολίδες προς τα αντίπαλα διαστημόπλοια.

Με την καταστροφή ενός εχθρικού διαστημοπλοίου, ο παίκτης κερδίζει πόντους. Σκοπός του παιχνιδιού είναι η συγκέντρωση όσο το δυνατόν περισσότερων πόντων, αποφεύγοντας ταυτόχρονα την ήττα.

### *5.3.2 Προετοιμασία του παιχνιδιού*

Έχοντας λοιπόν καθορίσει το παιχνίδι, προχωρούμε στη διαμόρφωση της αρχικής διατύπωσης ενός κατάλληλου κειμένου προτροπής (prompt). Μαζί με την περιγραφή του παιχνιδιού, παρέχονται και ορισμένα αρχικά assets, όπως η εικόνα του φόντου (διάστημα) καθώς και το διαστημόπλοιο του παίκτη.

Επιπλέον, δίνονται οδηγίες σχετικά με τη δομή και την οργάνωση των αρχείων που πρόκειται να δημιουργηθούν, καθώς και αναφορές στην τεκμηρίωση (documentation) της μηχανής, ώστε να είναι δυνατή η σωστή αξιοποίησή της.

Για τα επιπρόσθετα assets που απαιτούνται, δημιουργείται ένας ξεχωριστός φάκελος και ζητείται από το σύστημα τεχνητής νοημοσύνης να προτείνει τα χαρακτηριστικά τους (όπως διαστάσεις, μορφή sprite sheets κ.λπ.), προκειμένου να δημιουργηθούν στη συνέχεια από τον χρήστη.

Παρακάτω παρατίθεται το αρχικό prompt που χρησιμοποιήθηκε:

Prompt 5.1: Αρχικό prompt για την ανάπτυξη του space shooter

I want to create a 2D space shooter game using the ’engines Python API.
Please create a development plan and break it down into smaller, clear tasks.
Project structure:
− The game should be placed in @assets/scripts inside a new folder named "space_shooter".
Available assets:
− @assets/2D/space/MainShipIdle.png (336x48): ship idle animation (1 row, 6 frames).
− @assets/2D/space/MainShipRL.png (336x48): ship movement animation (left/right) (1 row, 6 frames ).
− @assets/2D/space/bg.png (120x120): background image (single frame).
Game concept:
− The player controls a spaceship positioned at the bottom of the screen.
− The ship can only move horizontally (left and right).
− The background should scroll downward continuously and loop when it reaches the end.
− Enemy ships spawn from the top and move downward.
− The player can shoot bullets to destroy enemies.
Your tasks:
1. Suggest the minimum additional sprites needed for the game.
For each sprite, specify:
− File name
− Resolution (pixels)
− Sprite sheet layout (rows x columns)
− Short description of its purpose
2. Generate a step−by−step implementation plan in Markdown (.md format) that I can follow to build the game using the engine.
3. Keep the design simple (do not suggest too many assets or overly complex systems).
If anything is unclear, ask me before proceeding.

### *5.3.3 Αποτελέσματα μετά την χρήση του Claude 3.5 Sonnet*

Στο πρώτο πείραμα, με τη χρήση του μοντέλου Claude 3.5 Sonnet, το αποτέλεσμα του αρχικού prompt ήταν η δημιουργία μιας καλά οργανωμένης δομής αρχείων και κλάσεων. Η παραγόμενη εφαρμογή (εκτελέσιμο αρχείο) περιλάμβανε τα ακόλουθα χαρακτηριστικά:

1. Ένα λειτουργικό παράθυρο εφαρμογής (application window).
2. Ορθή διαχείριση της αλληλεπίδρασης μεταξύ πληκτρολογίου και εφαρμογής (keyboard interaction handling).
3. Ορθή αξιοποίηση των παρεχόμενων συστημάτων της μηχανής παιχνιδιών.

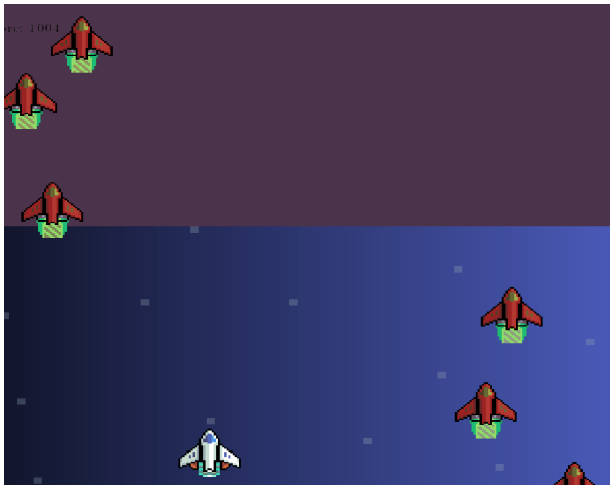
Πιο συγκεκριμένα, η εφαρμογή περιλάμβανε ένα αρχικό μενού, μέσω του οποίου ο χρήστης μπορούσε να μεταβεί στο κύριο παιχνίδι επιλέγοντας το κουμπί “Start”. Στο κύριο περιβάλλον

του παιχνιδιού υποστηριζόταν κίνηση του παίκτη προς τα δεξιά και τα αριστερά, καθώς και η δυνατότητα πυροβολισμού μέσω του πλήκτρου “Space”.

Τα βασικά προβλήματα του παραγόμενου παιχνιδιού σχετίζονταν αποκλειστικά με το οπτικό του μέρος. Πιο συγκεκριμένα, παρατηρήθηκαν τα εξής:

1. Λανθασμένη τοποθέτηση των διαφόρων αντικειμένων του παιχνιδιού (sprites).
2. Λανθασμένες διαστάσεις σε διάφορα αντικείμενα.
3. Λανθασμένος προσανατολισμός (στροφή) ορισμένων αντικειμένων.
4. Λανθασμένη τοποθέτηση του φόντου (background).

Τα παραπάνω προβλήματα δεν επηρέαζαν τη λειτουργικότητα του παιχνιδιού, αλλά επηρέαζαν σημαντικά την οπτική του παρουσίαση και την εμπειρία χρήσης Σχημα 5.9.

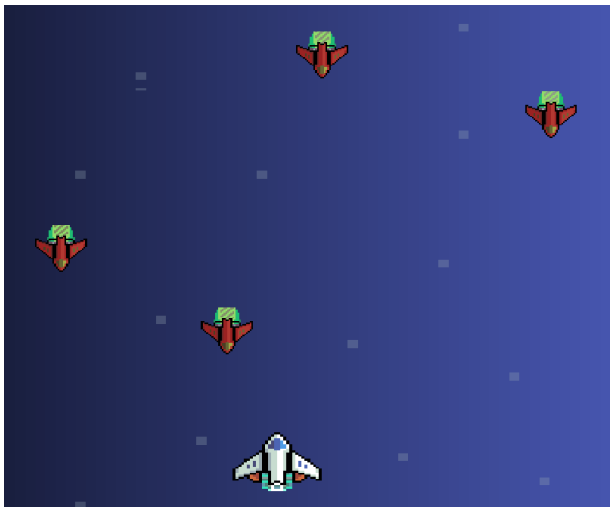


Σχήμα 5.9: Αρχικό αποτέλεσμα, με λανθασμένη τοποθέτηση των διαφόρων assets, τόσο ως προς τη θέση όσο και ως προς τη στροφή τους.

Στο Σχήμα 5.9 παρουσιάζεται ένα στιγμιότυπο από το αρχικό αυτό αποτέλεσμα. Ο χαρακτήρας κινείται σωστά οριζόντια (δεξιά και αριστερά), ενώ και οι αντίπαλοι, καθώς και οι βολίδες, εμφανίζουν την αναμενόμενη κίνηση. Παρ’ όλα αυτά, όπως διακρίνεται και από την εικόνα, το φόντο δεν αποδίδεται σωστά και η κίνησή του είναι λανθασμένη. Επιπλέον, οι αντίπαλοι έχουν εσφαλμένο προσανατολισμό, ενώ και οι βολίδες εμφανίζονται με λανθασμένη περιστροφή.

Στη συνέχεια, και μετά από μια σειρά στοχευμένων prompts για κάθε επιμέρους πρόβλημα, διορθώθηκαν όλα τα ζητήματα του αρχικού αποτελέσματος. Όλα αυτά επιτεύχθηκαν χωρίς να χρειαστεί οποιαδήποτε επέμβαση στον κώδικα. Παρακάτω παρουσιάζεται ένα στιγμιότυπο οθόνης του τελικού αποτελέσματος.

Έχοντας μέχρι στιγμής ένα σωστά υλοποιημένο παιχνίδι, προχώρησα σε μερικές επιπλέον προσθήκες και βελτιώσεις. Θέλοντας να προσδώσω ένα βαθμό δυσκολίας στον παίκτη του παιχνιδιού, αποφάσισα να προσθέσω την εξής λειτουργία: ο παίκτης θα διαθέτει πέντε σφαίρες. Κάθε φορά που χρησιμοποιεί μία από αυτές, αφαιρείται από το «καλάθι» των σφαιρών του. Θα



Σχήμα 5.10: Διορθωμένο αποτέλεσμα της σκηνής και των assets του παιχνιδιού Space Shooter.

υπάρχει χρόνος ανανέωσης (reload) για κάθε σφαίρα, ο οποίος θα είναι μεγαλύτερος από τον χρόνο μεταξύ δύο διαδοχικών χτυπημάτων. Επιπλέον, αυτός ο χρόνος θα αυξάνεται όσο ο παίκτης παραμένει στο παιχνίδι. "Επιπλέον, του ζήτησα να εμφανίζει πάνω αριστερά τις σφαίρες που απομένουν στον παίκτη.

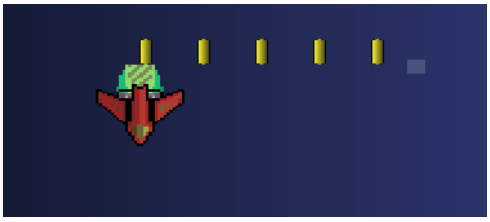
Παρακάτω δείνω το ακριβές prompt που χρησιμοποίησα:

Prompt 5.2: προσθήκες και βελτιώσεις

I want to make the game a bit more difficult. The player should have a limited number of bullets, for example 5. Bullets should be reloaded one by one, and the reload time for each bullet should be longer than the minimum time between hits. The player can only shoot if they have bullets available.

Additionally, I want the remaining bullets to appear at the top−left corner of the game window as small sprites, displayed in a horizontal line (one next to the other).

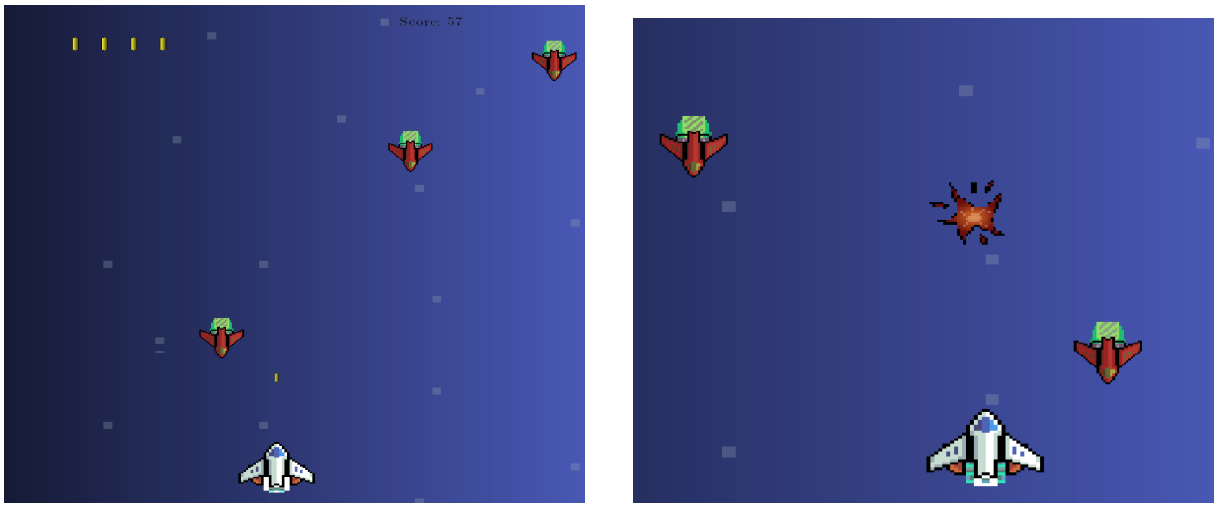
I also want the reload time to increase over time, up to an upper limit.



Σχήμα 5.11: Στιγμιότυπο του παιχνιδιού που απεικονίζει τις διαθέσιμες σφαίρες του παίκτη.

Σε σχέση με πριν, τώρα το αποτέλεσμα είναι σωστό, χωρίς να χρειάζεται να κάνουμε κάποιο επιπλέον prompt για διορθώσεις.

Μερικά σημαντικά στατιστικά για το εν λόγω μοντέλο (Claude 3.5 Sonnet) και το παιχνίδι που δημιουργήσαμε είναι τα παρακάτω:




Σχήμα 5.12: Τελικό αποτελέσμα. Στα αριστερά, ένα στιγμιότυπο του παιχνιδιού που απεικονίζει ολόκληρο το παράθυρο, ενώ στα δεξιά ένα στιγμιότυπο σε ζουμ που απεικονίζει την καταστροφή ενός αντιπάλου.

- Ο συνολικός αριθμός tokens που χρησιμοποιήθηκαν ήταν 683K.
- Το αποτέλεσμα ήταν 7 αρχεία.
- Παρήχθησαν 721 γραμμές Python κώδικα.
- Το κόστος ανά εκατομμύριο input tokens είναι 3 dollars.
- Το κόστος ανά εκατομμύριο output tokens είναι 15 dollars.
- Το συνολικό κόστος ήταν περίπου 2 dollars.

### *5.3.4 Αποτελέσματα μετά την χρήση του Claude Haiku 4.5*

Το πείραμα επαναλήφθηκε για ακόμη μία φορά, διατηρώντας το ίδιο αρχικό "prompt" και τις ίδιες αρχικές συνθήκες, με τη χρήση του μοντέλου Claude Haiku 4.5. Το περιβάλλον στο οποίο εκτελέστηκε το πείραμα ήταν "καθαρό", με την έννοια ότι δεν υπήρχε κανένα ίχνος του προηγούμενου πειράματος.

Συγκριτικά με το προηγούμενο μοντέλο, τα αποτελέσματα ήταν λιγότερο ενθαρρυντικά. Μετά το αρχικό "prompt", το αποτέλεσμα ήταν η δημιουργία ενός μοναδικού αρχείου και η εφαρμογή δεν μπορούσε να εκτελεστεί (δεν δημιουργούνταν ποτέ το παράθυρο της εφαρμογής).

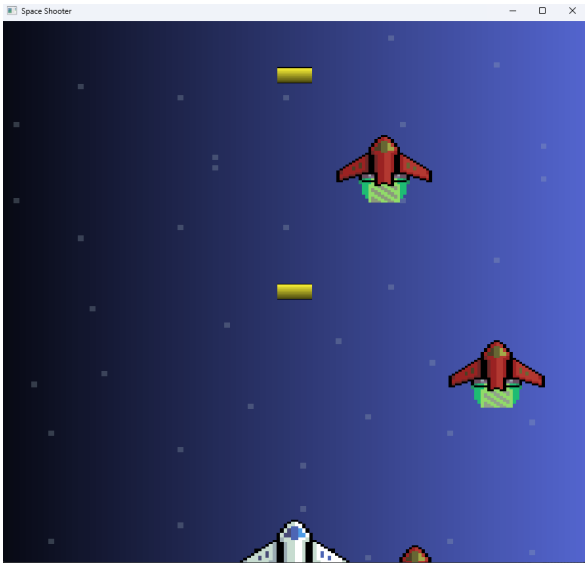
Στη συνέχεια, μετά από μια σειρά νέων "prompts", τα οποία ζητούσαν πιο εξειδικευμένα :

- Τη δημιουργία πολλαπλών αρχείων και κλάσεων.
- Τη διόρθωση των βασικών προβλημάτων που προέκυπταν (μέσω αντιγραφής και επικόλλησης του terminal output).

η εφαρμογή έφτασε τελικά σε μια σταθερή κατάσταση.

Όπως και με το Claude 3.5 Sonnet, έτσι και σε αυτή την περίπτωση υπήρξαν κάποια βασικά προβλήματα σε σχέση με το οπτικό κομμάτι Σχημα 5.13:

- Λανθασμένα μεγέθη αντικειμένων.
- Λανθασμένες στροφές (προσανατολισμός).
- Λανθασμένες τοποθετήσεις των αντικειμένων.




Σχήμα 5.13: Αρχικό αποτέλεσμα με τη χρήση του μοντέλου Claude Haiku 4.5.

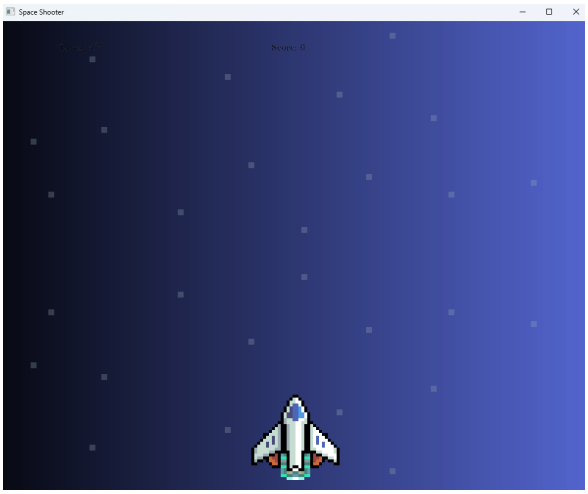
Και εδώ, μετά από μια σειρά νέων prompts, καταφέραμε χωρίς να χρειαστεί να επέμβουμε στον κώδικα να φτάσουμε σε ένα σωστό αποτέλεσμα. Παρακάτω δίνονται κάποια στατιστικά σε σχέση με το Claude Haiku 4.5 και το παιχνίδι που δημιουργήθηκε:

- Συνολικός αριθμός tokens: 601K
- 13 αρχεία
- 1513 γραμμές Python κώδικα
- Κόστος input tokens ανά εκατομμύριο: 1 dollars
- Κόστος output tokens ανά εκατομμύριο: 5 dollars
- Συνολικό κόστος: περίπου 0.9 dollars

### *5.3.5 Αποτελέσματα μετα την χρήση του Big Pickle*

Η τρίτη αξιολόγηση πραγματοποιήθηκε με τη χρήση του δωρεάν διαθέσιμου agent Big Pickle μέσω του περιβάλλοντος OpenCode. Και σε αυτή την περίπτωση χρησιμοποιήθηκε το ίδιο αρχικό prompt και τα ίδια γραφικά assets, ώστε να διασφαλιστεί η συνέπεια μεταξύ όλων των πειραμάτων.

Στην πρώτη προσπάθεια, το μοντέλο κατάφερε να δημιουργήσει επιτυχώς ένα λειτουργικό παράθυρο εφαρμογής. Σε αντίθεση με τα προηγούμενα μοντέλα, η αρχική τοποθέτηση των γραφικών στοιχείων του παίκτη και του background ήταν σχετικά ακριβής ως προς τη θέση και την κλίμακα. Ωστόσο, η παραγόμενη εφαρμογή δεν διέθετε αρκετά από τα απαραίτητα συστήματα του παιχνιδίου Σχήμα 5.14.




Σχήμα 5.14: Αρχικό αποτέλεσμα με τη χρήση του μοντέλου Big Pickle..

Πιο συγκεκριμένα, το αρχικό αποτέλεσμα περιλάμβανε:

- Στατικό χαρακτήρα (δεν ήταν δυνατή η κίνηση δεξιά και αριστερά).
- Στατικό background.
- Έλλειψη αντιπάλων μέσα στη σκηνή.
- Απουσία συστήματος βλημάτων (projectile/bullet system).

Τα βασικά προβλήματα σε αυτή την περίπτωση σχετίζονταν κυρίως με τον βασικό βρόχο ενημέρωσης της εφαρμογής και τη λογική του παιχνιδιού, και όχι τόσο με την τοποθέτηση των γραφικών στοιχείων. Συγκεκριμένα, η παραγόμενη υλοποίηση παρουσίαζε λανθασμένη συμπεριφορά στο update loop, γεγονός που εμπόδιζε τη δυναμική αλληλεπίδραση μέσα στον κόσμο του παιχνιδιού. Μετά από επαναληπτικές διορθώσεις και prompts εντοπισμού σφαλμάτων, η εφαρμογή εξελίχθηκε σταδιακά σε ένα πιο ολοκληρωμένο παιχνίδι.

Παρόλα αυτά, όπως και στα προηγούμενα πειράματα, τα προβλήματα με τη στροφή (rotation) των assets παρέμειναν καθ’ όλη τη διάρκεια της ανάπτυξης.Παρακάτω δίνονται τα βασικά στατιστικά και για αυτό το πείραμα:

- Συνολικός αριθμός tokens: 202K
- 10 αρχεία με 978 γραμμές κώδικα Python
- Το μοντέλο Big Pickle AI coding είναι επί του παρόντος δωρεάν, με κόστος 0.00 dollars ανά εκατομμύριο input και output tokens, και προσφέρεται ως stealth free-to-use μοντέλο μέσω της πλατφόρμας OpenCode Zen.

## 5.4 Σύνοψη κεφαλαίου

Στο παρόν κεφάλαιο παρουσιάστηκαν ενδεικτικές εφαρμογές που αναπτύχθηκαν με τη χρήση της μηχανής παιχνιδιών που υλοποιήθηκε στο πλαίσιο της εργασίας. Μέσα από την ανάπτυξη διαφορετικών τύπων παιχνιδιών, καθώς και μέσω της χρήσης εργαλείων τεχνητής νοημοσύνης για τη δημιουργία ενός ολοκληρωμένου παραδείγματος, αναδείχθηκαν στην πράξη οι δυνατότητες της μηχανής, η ευελιξία της αρχιτεκτονικής της και η αποτελεσματικότητα των υποσυστημάτων που παρουσιάστηκαν στα προηγούμενα κεφάλαια. Τα παραδείγματα αυτά λειτούργησαν τόσο ως μέσο αξιολόγησης της λειτουργικότητας της μηχανής όσο και ως απόδειξη ότι η υλοποίηση μπορεί να υποστηρίξει την ανάπτυξη πραγματικών εφαρμογών με διαφορετικές απαιτήσεις και χαρακτηριστικά. Με την ολοκλήρωση της παρουσίασης των εφαρμογών χρήσης, η εργασία προχωρά στο τελικό κεφάλαιο, όπου συνοψίζονται τα βασικά συμπεράσματα, αξιολογούνται οι σχεδιαστικές επιλογές που υιοθετήθηκαν και συζητούνται πιθανές κατευθύνσεις για μελλοντική επέκταση και βελτίωση της μηχανής.

# ΚΕΦΑΛΑΙΟ 6 Συμπεράσματα

## 6.1 Συμπεράσματα με βάση την θεωρητική έρευνα

Πρωταρχικός στόχος της παρούσας εργασίας είναι η κατανόηση του τρόπου οργάνωσης και λειτουργίας μιας μηχανής παιχνιδιών από τα θεμελιώδη δομικά της στοιχεία. Από τη θεωρητική έρευνα που πραγματοποιήθηκε και παρουσιάστηκε στο Κεφάλαιο 2 προέκυψαν ορισμένα βασικά συμπεράσματα.

Αρχικά, οι μηχανές παιχνιδιών αποτελούν ιδιαίτερα πολύπλοκα λογισμικά συστήματα. Ειδικότερα, οι σύγχρονες εμπορικές μηχανές περιλαμβάνουν μεγάλο αριθμό συστημάτων και υποσυστημάτων, η πλήρης υλοποίηση των οποίων, στο πλαίσιο μιας διπλωματικής εργασίας, είναι εξαιρετικά δύσκολη έως και ανέφικτη.

Παράλληλα, διαπιστώθηκε ότι τόσο οι εμπορικές όσο και οι μικρότερης κλίμακας μηχανές παιχνιδιών παρουσιάζουν σημαντικές ομοιότητες ως προς τη βασική τους αρχιτεκτονική και τα κύρια συστήματα που τις απαρτίζουν. Η παρατήρηση αυτή αποτέλεσε τη βάση πάνω στην οποία σχεδιάστηκε και αναπτύχθηκε η μηχανή που παρουσιάζεται στην παρούσα εργασία.

Επιπλέον, κατά τη μελέτη της αρχιτεκτονικής των μηχανών παιχνιδιών καθίσταται εμφανές ότι τα επιμέρους συστήματα και υποσυστήματα αλληλεπιδρούν σε μεγάλο βαθμό μεταξύ τους και απαιτούν γνώσεις από διαφορετικά επιστημονικά και τεχνολογικά πεδία. Ενδεικτικά, η ανάπτυξη μιας μηχανής παιχνιδιών προϋποθέτει ισχυρό μαθηματικό και γεωμετρικό υπόβαθρο, κατανόηση της αρχιτεκτονικής των υπολογιστικών συστημάτων και της επικοινωνίας με την κάρτα γραφικών, γνώση αλγορίθμων, καθώς και βασικές αρχές σχεδιασμού. Παράλληλα, απαιτούνται γνώσεις σχετικές με την απεικόνιση χρωμάτων και φωτισμού, την επεξεργασία ήχου και άλλους συναφείς τομείς.

## 6.2 Συμπεράσματα απο την πρακτική υλοποίηση της μηχανής

Προχωρώντας στην πρακτική υλοποίηση της εργασίας, επιλέχθηκε η ανάπτυξη μιας ελαφριάς μηχανής παιχνιδιών για τη δημιουργία δισδιάστατων παιχνιδιών, χρησιμοποιώντας τη γλώσσα προγραμματισμού C++ σε συνδυασμό με τις βιβλιοθήκες OpenGL και GLFW.

Κατά τη διαδικασία υλοποίησης, μία από τις πρώτες και σημαντικότερες σχεδιαστικές αποφάσεις αφορούσε τον τρόπο αναπαράστασης και διαχείρισης των αντικειμένων της σκηνής. Η αρχική προσέγγιση βασιζόταν σε μια γενική κλάση GameObject, η οποία περιείχε βασικά δεδομένα, όπως η θέση και το sprite κάθε αντικειμένου. Κάθε αντικείμενο της σκηνής υλοποιούνταν

ως υποκλάση της συγκεκριμένης κλάσης.

Ωστόσο, κατά την εξέλιξη της ανάπτυξης διαπιστώθηκε ότι η συγκεκριμένη αρχιτεκτονική παρουσίαζε περιορισμένη επεκτασιμότητα και θα μπορούσε να οδηγήσει σε προβλήματα συντήρησης και διαχείρισης καθώς το έργο μεγάλωνε. Για τον λόγο αυτό, επιλέχθηκε η υιοθέτηση της αρχιτεκτονικής Entity Component System (ECS), όπως αυτή παρουσιάστηκε στο Κεφάλαιο 4.

Από τη σύγκριση των δύο προσεγγίσεων προκύπτει ότι η αρχική λύση ήταν απλούστερη ως προς την υλοποίηση και θα μπορούσε να αποτελέσει κατάλληλη επιλογή για την ανάπτυξη ενός συγκεκριμένου παιχνιδιού με σαφώς καθορισμένες απαιτήσεις. Αντίθετα, στην περίπτωση μιας γενικής μηχανής παιχνιδιών, η οποία προορίζεται να υποστηρίξει διαφορετικά είδη έργων και μελλοντικές επεκτάσεις, η χρήση του ECS αποτελεί σαφώς καταλληλότερη επιλογή.

Ένα ακόμη σημαντικό ζήτημα που προέκυψε κατά την ανάπτυξη της μηχανής αφορούσε τη διαδικασία αποστολής των δεδομένων στην κάρτα γραφικών. Στην αρχική υλοποίηση, κάθε αντικείμενο της σκηνής αποστελλόταν ξεχωριστά προς επεξεργασία από τη GPU. Η προσέγγιση αυτή ήταν επαρκής για μικρής κλίμακας παιχνίδια με περιορισμένο αριθμό αντικειμένων. Ωστόσο, καθώς ο αριθμός των αντικειμένων αυξανόταν, παρατηρούνταν αισθητή υποβάθμιση της απόδοσης, η οποία σε ορισμένες περιπτώσεις οδηγούσε ακόμη και σε προσωρινές παύσεις ή καθυστερήσεις κατά την εκτέλεση.

Για την αντιμετώπιση του προβλήματος υιοθετήθηκε μια δεύτερη προσέγγιση, κατά την οποία τα δεδομένα οργανώνονται και αποστέλλονται στην κάρτα γραφικών σε παρτίδες (batches). Η τεχνική αυτή μειώνει σημαντικά τον αριθμό των απαιτούμενων draw calls και βελτιώνει την απόδοση του συστήματος, επιτρέποντας την αποτελεσματική διαχείριση μεγαλύτερου αριθμού αντικειμένων και την υποστήριξη πιο απαιτητικών παιχνιδιών.

Με βάση τα παραπάνω παραδείγματα, μπορεί να εξαχθεί ένα γενικό συμπέρασμα σχετικά με τη φύση και τον ρόλο των μηχανών παιχνιδιών. Η πολυπλοκότητα της ανάπτυξης μιας μηχανής παιχνιδιών αντανακλά σε μεγάλο βαθμό την πολυπλοκότητα της ίδιας της ανάπτυξης ενός παιχνιδιού, καθώς πολλές από τις σχεδιαστικές και τεχνικές αποφάσεις που λαμβάνονται κατά την κατασκευή μιας μηχανής θα έπρεπε, ελλείψει αυτής, να ληφθούν από τον ίδιο τον προγραμματιστή του παιχνιδιού.

Η σημασία του διαχωρισμού μεταξύ της μηχανής παιχνιδιών και του τελικού παιχνιδιού είναι ιδιαίτερα σημαντική, καθώς συμβάλλει στη βελτίωση της διαδικασίας ανάπτυξης με δύο βασικούς τρόπους. Αρχικά, μειώνει σημαντικά την πολυπλοκότητα που καλείται να διαχειριστεί ο προγραμματιστής του παιχνιδιού, απαλλάσσοντάς τον από ένα μεγάλο πλήθος χαμηλού επιπέδου τεχνικών και αρχιτεκτονικών αποφάσεων. Παράλληλα, επιτρέπει την απομόνωση και την ανεξαρτητοποίηση των επιμέρους συστημάτων και υποσυστημάτων της μηχανής.

Ως αποτέλεσμα, ο σχεδιαστής ή προγραμματιστής της μηχανής μπορεί να πραγματοποιεί βελτιώσεις, τροποποιήσεις ή επεκτάσεις στα εσωτερικά συστήματα της μηχανής, επηρεάζοντας θετικά το τελικό αποτέλεσμα χωρίς να απαιτούνται ουσιαστικές αλλαγές στον τρόπο ανάπτυξης ή στη λογική του ίδιου του παιχνιδιού. Η ιδιότητα αυτή αποτελεί ένα από τα σημαντικότερα

πλεονεκτήματα των σύγχρονων μηχανών παιχνιδιών και έναν από τους βασικούς λόγους για τους οποίους έχουν καθιερωθεί ως αναπόσπαστο εργαλείο της ανάπτυξης βιντεοπαιχνιδιών.

## 6.3 Συμπεράσματα απο την δημιουργία παιχνιδίων

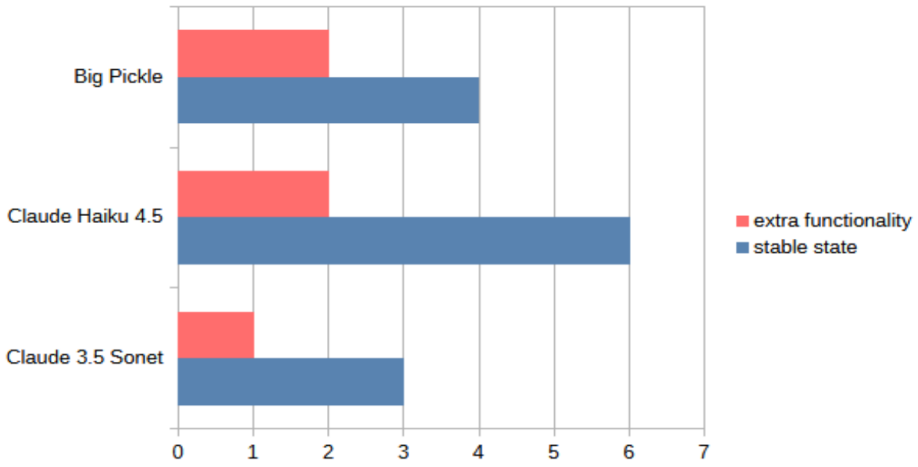
Η ανάπτυξη παιχνιδιών με τη χρήση της μηχανής αποτέλεσε ένα από τα βασικά στάδια αξιολόγησης της παρούσας εργασίας. Ο πρωταρχικός στόχος ήταν αφενός να διερευνηθεί κατά πόσο είναι εφικτή η δημιουργία λειτουργικών παιχνιδιών με τη χρήση της μηχανής και αφετέρου να επαληθευτεί η ορθή λειτουργία των επιμέρους συστημάτων και υποσυστημάτων της. Για τον λόγο αυτό επιλέχθηκαν παιχνίδια διαφορετικού τύπου και πολυπλοκότητας, ώστε να αξιολογηθεί όσο το δυνατόν μεγαλύτερο εύρος των δυνατοτήτων που προσφέρει η μηχανή.

Τα αποτελέσματα των πειραμάτων έδειξαν ότι η μηχανή ήταν σε θέση να υποστηρίξει επιτυχώς την ανάπτυξη διαφορετικών ειδών παιχνιδιών, επιβεβαιώνοντας τη σωστή λειτουργία των βασικών συστημάτων της. Παράλληλα, από τα στατιστικά στοιχεία που συλλέχθηκαν προέκυψε ένα ιδιαίτερα σημαντικό συμπέρασμα σχετικά με τη μείωση της πολυπλοκότητας της διαδικασίας ανάπτυξης. Όπως παρουσιάστηκε στο Κεφάλαιο 5, κάθε κλάση ή συνάρτηση που παρέχεται από τη μηχανή αντιστοιχεί σε δεκάδες ή ακόμη και εκατοντάδες γραμμές κώδικα χαμηλότερου επιπέδου, οι οποίες διαφορετικά θα έπρεπε να υλοποιηθούν εκ νέου από τον προγραμματιστή. Ως αποτέλεσμα, η χρήση της μηχανής μειώνει σημαντικά τον απαιτούμενο χρόνο ανάπτυξης, περιορίζει την πολυπλοκότητα του κώδικα και επιτρέπει στον προγραμματιστή να επικεντρωθεί περισσότερο στη λογική και στον σχεδιασμό του παιχνιδιού.

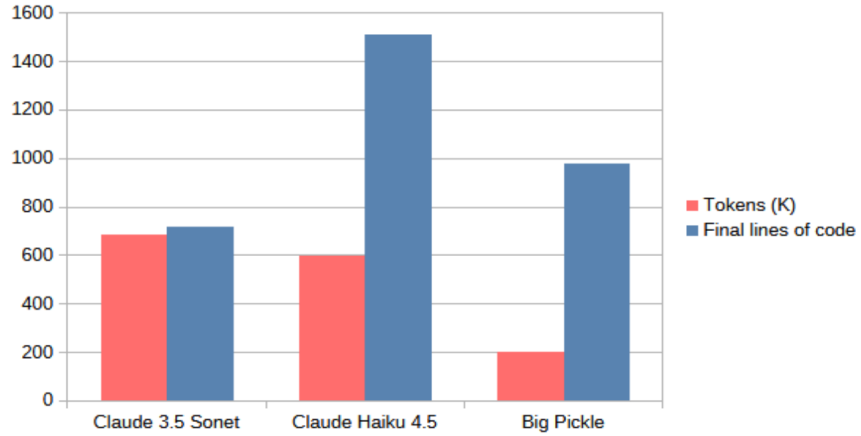
Η παραπάνω παρατήρηση αποκτά ακόμη μεγαλύτερη σημασία αν ληφθεί υπόψη ότι η μηχανή που αναπτύχθηκε στο πλαίσιο της παρούσας εργασίας αποτελεί μια σχετικά ελαφριά υλοποίηση, η οποία περιλαμβάνει μόνο τα βασικά συστήματα που κρίθηκαν απαραίτητα. Η ενσωμάτωση επιπρόσθετων λειτουργιών και εργαλείων στο μέλλον θα μπορούσε να αυξήσει περαιτέρω το επίπεδο αφαίρεσης που παρέχει και να απλοποιήσει ακόμη περισσότερο τη διαδικασία ανάπτυξης παιχνιδιών.

Η αποτελεσματικότητα της μηχανής επιβεβαιώθηκε επιπλέον μέσω των πειραμάτων που πραγματοποιήθηκαν με τη χρήση σύγχρονων συστημάτων τεχνητής νοημοσύνης. Τα πειράματα έδειξαν ότι οι σύγχρονοι AI agents είναι σε θέση να προσαρμόζονται σε άγνωστα προγραμματιστικά περιβάλλοντα και να παράγουν λειτουργικές υλοποιήσεις παιχνιδιών βασιζόμενοι αποκλειστικά σε οδηγίες υψηλού επιπέδου. Παρά το γεγονός ότι κανένα από τα μοντέλα που αξιολογήθηκαν δεν είχε προηγούμενη γνώση της συγκεκριμένης μηχανής ή του API της, όλα κατάφεραν να κατανοήσουν τις παρεχόμενες αφαιρέσεις και να παράγουν εκτελέσιμο κώδικα.

Τα αποτελέσματα έδειξαν επίσης ότι ακόμη και μικρότερα ή δωρεάν διαθέσιμα μοντέλα μπορούν να προσαρμοστούν σε άγνωστα APIs και να δημιουργήσουν μερικώς ή πλήρως λειτουργικές εφαρμογές. Ωστόσο, τα μεγαλύτερα εμπορικά μοντέλα παρουσίασαν υψηλότερη αποτελεσματικότητα, απαιτώντας μικρότερο αριθμό προτροπών (prompts) και παράγοντας συνήθως πιο συμπαγείς και οργανωμένες λύσεις. Ενδεικτικά, το Claude Sonnet 3.5 κατέλαβε την πρώτη




Σχήμα 6.1: Σύγκριση των μοντέλων Claude 3.5 Sonnet, Claude Haiku 4.5 και Big Pickle ως προς τον συνολικό αριθμό των χρησιμοποιηθέντων tokens και το πλήθος των παραγόμενων γραμμών κώδικα.




Σχήμα 6.2: Σύγκριση των μοντέλων Claude 3.5 Sonnet, Claude Haiku 4.5 και Big Pickle ως προς τον αριθμό των προτροπών (prompts) που απαιτήθηκαν για την επίτευξη λειτουργικού αποτελέσματος και την προσθήκη επιπρόσθετης λειτουργικότητας.

θέση, ακολουθούμενο από το Claude Haiku 4.5 και το Big Pickle όπως ειναι φανερό και απο το Σχήμα 6.1. Όσον αφορά το κόστος χρήσης, όπως αυτό εκφράζεται μέσω του αριθμού των καταναλωθέντων tokens, καθώς και τον όγκο του παραγόμενου κώδικα, το Claude Sonnet 3.5 παρουσίασε την πιο συμπαγή υλοποίηση, παράγοντας τις λιγότερες γραμμές κώδικα. Ωστόσο, το πλεονέκτημα αυτό συνοδεύτηκε από το υψηλότερο συνολικό κόστος μεταξύ των αξιολογούμενων μοντέλων Σχήμα 6.2.

Ένα ιδιαίτερα ενδιαφέρον εύρημα ήταν ότι η λογική ορθότητα του παραγόμενου κώδικα ήταν σημαντικά υψηλότερη από την οπτική ορθότητα των παιχνιδιών. Στις περισσότερες περιπτώσεις, τα μοντέλα κατάφεραν να αξιοποιήσουν σωστά την αρχιτεκτονική Entity Component System και να υλοποιήσουν λειτουργικούς μηχανισμούς παιχνιδιού. Αντίθετα, παρατηρήθηκαν συχνότερα προβλήματα που σχετίζονταν με την οπτική αναπαράσταση, όπως λανθασμένες θέσεις αντικειμένων, ασυνέπειες στην κλίμακα, παραμορφώσεις αναλογιών και σφάλματα στη

διάταξη στοιχείων της διεπαφής.

Συνολικά, τα αποτελέσματα καταδεικνύουν ότι η μηχανή παρέχει ένα επαρκώς σαφές και κατανοητό επίπεδο αφαίρεσης, το οποίο μπορεί να αξιοποιηθεί όχι μόνο από ανθρώπινους προγραμματιστές αλλά και από σύγχρονα συστήματα τεχνητής νοημοσύνης χωρίς προηγούμενη εκπαίδευση στο συγκεκριμένο περιβάλλον. Παράλληλα, αναδεικνύονται τόσο οι δυνατότητες όσο και οι σημερινοί περιορισμοί της υποβοηθούμενης από τεχνητή νοημοσύνη ανάπτυξης παιχνιδιών, οι οποίοι αναμένεται να μειωθούν σημαντικά καθώς τα μοντέλα συνεχίζουν να εξελίσσονται.

## 6.4 Μελλοντικές επεκτάσεις

Η παρούσα εργασία είχε ως βασικό στόχο τη μελέτη, τον σχεδιασμό και την υλοποίηση μιας μηχανής παιχνιδιών από το μηδέν, καθώς και την αξιολόγηση των δυνατοτήτων της μέσω της ανάπτυξης παιχνιδιών και της χρήσης σύγχρονων συστημάτων τεχνητής νοημοσύνης. Παρόλο που οι στόχοι που τέθηκαν επιτεύχθηκαν σε σημαντικό βαθμό, υπάρχουν αρκετές δυνατότητες περαιτέρω επέκτασης και βελτίωσης της μηχανής.

Ως μελλοντικές κατευθύνσεις προτείνονται οι ακόλουθες:

1. **Υποστήριξη προηγμένων τρισδιάστατων γραφικών.** Η παρούσα υλοποίηση επικεντρώθηκε στην ανάπτυξη δισδιάστατων παιχνιδιών. Η επέκταση της μηχανής με πλήρη υποστήριξη τρισδιάστατης απεικόνισης, προηγμένων τεχνικών φωτισμού, σκιών και σύγχρονων rendering pipelines θα επέτρεπε τη δημιουργία πιο σύνθετων εφαρμογών και θα διεύρυνε σημαντικά το πεδίο χρήσης της.

2. **Ενσωμάτωση επιπλέον συστημάτων φυσικής και εφέ.** Η προσθήκη προηγμένων μηχανισμών φυσικής, συστημάτων σωματιδίων (particle systems), animation systems και άλλων εξειδικευμένων υποσυστημάτων θα μπορούσε να βελτιώσει σημαντικά τις δυνατότητες της μηχανής.

3. **Υποστήριξη δικτυακών λειτουργιών.** Η ανάπτυξη μηχανισμών δικτυακής επικοινωνίας και συγχρονισμού θα επέτρεπε τη δημιουργία multiplayer παιχνιδιών, επεκτείνοντας περαιτέρω τις δυνατότητες της πλατφόρμας.

4. **Ανάπτυξη γραφικού περιβάλλοντος χρήστη.** Αν και η φιλοσοφία της παρούσας εργασίας βασίστηκε σε μια code-first προσέγγιση, η προσθήκη ενός βασικού γραφικού περιβάλλοντος (GUI) ή ενός απλού editor θα μπορούσε να βελτιώσει σημαντικά τη χρηστικότητα της μηχανής και να την καταστήσει πιο προσιτή σε νέους χρήστες.

5. **Περαιτέρω διερεύνηση της χρήσης τεχνητής νοημοσύνης.** Τα πειράματα που πραγματοποιήθηκαν έδειξαν ότι τα σύγχρονα γλωσσικά μοντέλα μπορούν να προσαρμοστούν ακόμη και σε άγνωστα προγραμματιστικά περιβάλλοντα. Μελλοντική έρευνα θα μπορούσε να εξετάσει μεγαλύτερο αριθμό μοντέλων, νεότερες εκδόσεις τους, καθώς και πιο

σύνθετες αρχιτεκτονικές βασισμένες σε tool-calling ή multi-agent συστήματα, με στόχο την περαιτέρω αυτοματοποίηση της διαδικασίας ανάπτυξης παιχνιδιών.

Συνολικά, η κατασκευή μιας μηχανής παιχνιδιών από το μηδέν αποδείχθηκε μια ιδιαίτερα απαιτητική αλλά ταυτόχρονα εξαιρετικά εκπαιδευτική διαδικασία. Η ανάπτυξη της μηχανής επέτρεψε την εις βάθος κατανόηση των αρχιτεκτονικών επιλογών, των υποσυστημάτων και των τεχνικών προκλήσεων που συναντώνται στη σύγχρονη ανάπτυξη βιντεοπαιχνιδιών. Παράλληλα, η εργασία ανέδειξε τον καθοριστικό ρόλο που διαδραματίζουν οι μηχανές παιχνιδιών στη μείωση της πολυπλοκότητας της ανάπτυξης λογισμικού και στη διευκόλυνση της δημιουργίας διαδραστικών εφαρμογών.

Η μηχανή που αναπτύχθηκε στο πλαίσιο της παρούσας εργασίας αποτελεί μια ισχυρή βάση για μελλοντικές επεκτάσεις, τόσο σε ερευνητικό όσο και σε επαγγελματικό επίπεδο, ενώ τα συμπεράσματα που προέκυψαν μπορούν να αξιοποιηθούν για την περαιτέρω μελέτη της ανάπτυξης μηχανών παιχνιδιών και της αλληλεπίδρασής τους με σύγχρονες τεχνολογίες τεχνητής νοημοσύνης.

# References


[1] Tomas Akenine-Möller et al. *Real-Time Rendering*. 4th ed. CRC Press, 2024. ISBN: 9781138627000.

[2] Anderson Andrade. ``Game Engines: A Survey." In: *EAI Endorsed Transactions on Serious Games* (2015). DOI: https://doi.org/10.4108/eai.5-11-2015.150615.

[3] Tristan Donovan. *Replay: The History of Video Games*. Yellow Ant, 2010.

[4] Jason Gregory. *Game Engine Architecture*. CRC Press, 2019.

[5] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. 3rd ed. Course Technology, 2011. ISBN: 9781435458864.

[6] MainLeaf Studios. *The First 3D Game: A History of 3D Game Development*. URL: https://mainleaf.com/first-3d-game-a-history-of-3d-game-development/ (visited on 01/01/2024).

[7] Mike McShaffry and David Graham. *Game Coding Complete*. 4th ed. Cengage Learning, 2012. ISBN: 9781133776574.

[8] Ian Millington and John Funge. *Artificial Intelligence for Games*. Morgan Kaufmann, 2009. ISBN: 9780123747310.

[9] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. ``Cowboys, Ankle Sprains, and Keepers of Quality: How Is Video Game Development Different from Software Development?" In: *Proceedings of the 36th International Conference on Software Engineering* (2014). DOI: https://doi.org/10.1145/2568225.2568226.

[10] Fabio Petrillo et al. ``What went wrong? A survey of problems in game development." In: *ACM Computers in Entertainment* 7.1 (2009), pp. 1–22. DOI: https://doi.org/10.1145/1486508.1486521.

[11] Cristiano Politowski. ``Are Game Engines Software Frameworks? A Three-Perspective Study." In: *Journal of Systems and Software* (2022). DOI: https://doi.org/10.1016/j.jss.2020.110846.

[12] Vincenzo Romeo. ``Analysis of Entity Encoding Techniques, Design and Implementation of a Multithreaded Compile-Time Entity-Component-System C++14 Library." 2022. DOI: https://doi.org/10.13140/RG.2.1.1307.4165.

[13] Allen Sherrod. *Ultimate 3D Game Engine Design and Architecture*. Charles River Media. ISBN: 9781584504733.

[14] Gabriel C. Ullmann. ``An Exploratory Approach for Game Engine Architecture Recovery." In: *arXiv preprint* (2023). DOI: https://doi.org/10.48550/arXiv.2303.02429.

[15] Gabriel C. Ullmann and Yann-Gaël Guéhéneuc. ``SyDRA: An Approach to Understand Game Engine Architecture." In: *arXiv preprint* (2024). DOI: https://doi.org/10.48550/arXiv.2406.05487.

[16] Zach Whalen and Laurie N. Taylor, eds. *Playing the Past: History and Nostalgia in Video Games*. Vanderbilt University Press, 2008.

[17] Anis Zarrad. ``Game Engine Solutions." In: *Simulation and Gaming*. IntechOpen, 2018. DOI: https://doi.org/10.5772/intechopen.71429.

# Δήλωση Πνευματικών Δικαιωμάτων

Δηλώνω ρητά ότι, σύμφωνα με το άρθρο 8 του Ν. 1599/1986 και τα άρθρα 2, 4, 6 παρ. 3 του Ν. 1256/1982, η παρούσα Μεταπτυχιακή Διπλωματική Εργασία με τίτλο:
«Μηχανή δημιουργίας παιχνιδιών (χωρίς γραφικό περιβάλλον) : Αρχιτεκτονικός σχεδιασμός, Υλοποίηση, Δημιουργία παραδειγμάτων παιχνιδιών.»

καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας και αναφέρονται ρητώς μέσα στο κείμενο που τη συνοδεύουν, η οποία έχει εκπονηθεί στο Πρόγραμμα Μεταπτυχιακών Σπουδών «Ανάπτυξη Ψηφιακών Παιχνιδιών και Πολυμεσικών Εφαρμογών» του Τμήματος Επικοινωνίας & Ψηφιακών Μέσων του Πανεπιστημίου Δυτικής Μακεδονίας, υπό την επίβλεψη του Δρ. Μηνά Δασυγένη, αποτελεί αποκλειστικά προϊόν προσωπικής εργασίας και δεν προσβάλλει κάθε μορφής πνευματικά δικαιώματα τρίτων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής. Οι πηγές που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον.

Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και μόνο.



Υπογραφή Φοιτητή
Κωνσταντίνος Γούλας.