BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

A Dynamic Application for Generating Rogulelike Games

Supervisors

Lect. Dr. Ioan Lazăr

Drd. Zsigmond Imre

Author

Csóka Ervin

2020

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENȚĂ

O Aplicație Dinamică pentru Generarea Jocurilor Roguelike

Conducători ştiinţifici

Lect. Dr. Ioan Lazăr

Drd. Zsigmond Imre

Absolvent

Csóka Ervin

2020

Abstract

The game development industry has progressed rapidly over the years. The scale of modern games keeps increasing, requiring specialized tools and interdisciplinary teams to create performant games that are rich in content. The purpose of this work is to detail the building blocks required to create an interactive system that provides the necessary tools to develop a Roguelike game and describe the techniques we used to implement such a system. We employ the Entity-Component-System pattern to implement a game object system. Behavior is specified in scripts external to the application making use of the AngelScript environment. To generate the game environment we use the Feasible-Infeasible Two-Populations genetic algorithm. Experiments using the algorithm have promising results, pointing us to the conclusion that the algorithm can be used to generate game space both during gameplay and during development.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Csóka Ervin

Contents

Chapter 1

Introduction

Modern games are complex pieces of software and, as the industry advances, their complexity is likely to keep growing. [2]. Game development requires people of different disciplines to co-operate demanding systems to program interactions, creation of game assets such as characters or environment, animations and many more [31]. As such, there exist a necessity for a high level of abstraction as well as for specialized tools intended to be more inclusive of the non-technical staff involved in a project. These are usually encompassed in interactive systems commonly called game engines.

A game engine facilitates the game development process by providing a collection of systems with reusable components that offer a high level of abstraction while not directly specifying the game logic or environment. A team of content creators should be able to develop a game without having to understand the implementation details of the engine. This is achieved by software engineers creating tools to meet the needs of game designers [38]. While these tools should not take away flexibility from the designers, they must be specific enough to meet the requirements for the game under development. As such, certain optimisations are required for each genre. This dependency between the resulting product and the underlying engine reduces the types of games that can be built using it. Typical tools provided by an engine include: asset management, input handling, rendering & animation, event handling, game object management, GUI creation, sound effects and physics.

The Rogulike game genre is a subcatgory of the Role-Playing-Game (RPG) genre. It can be traced back to the 1980's game Rogue [7, 13, 14, 27] which is a turn-based RPG with ASCII graphics that can generate a vast number of unique levels featuring enemies and collectibles. Discussions regarding what is and is not a Roguelike or what exactly are the core characteristics of such games do not have clear conclusions and an in-depth analysis of such is outside the scope of this work. A list of Roguelike features has been assembled in 2008 at the International Roguelike Development Conference in Berlin (*http://www.roguebasin.com/index.php?title=Rgrd_meeting*), but the term is commonly used to describe games that have above average difficulty and provide procedural environment generation [14].

We undertake the task of building a game engine providing the necessary functionality to create a Roguelike game. The engine should be able to scale to a large number of game objects and provide tools for the non-technical staff to specify their properties and behavior. Additionally it should implement a content generation mechanism and allow designers to manipulate its outputs.

In Chapter(2) we present previous work published on the subject. We compare the traditional inheritance-based object oriented paradigm with a data-oriented approach focused on composition for implementing a game object system. In the next section we argue for the value brought by an integrated scripting environment, allowing scripts that specify behavior to be moved outside of the engine. This should reduce the complexity of the engine API and provide a protected environment for game designers to work in. Moving forward we present generation techniques used in other game development projects and detail the approach we have taken in this respect. We end this chapter by presenting a number of collision detection techniques used in 2D games and several space partitioning techniques to speed up computations.

In Chapter(3) we set forth the architecture of our application and provide insight on the implementation details. We detail the utilitarian tools we have used, followed by the inner-workings of the event system. Next, we illustrated the render mechanism, focusing on an architecture that decouples the underlying graphics API. We continue by presenting the way game objects are handled by our application, both with respects to their attributes as well as to their behavior which is specified in external scripts. We end this chapter by describing the way our content generation tool is used to generate game environment.

In Chapter(4) we present the way applications can be created using our engine and display the results we have obtained by experimenting with the content generation tool, reaching our conclusions in Chapter(5).

Chapter 2

State of the Art

2.1 Game Object Systems

Creating an interactive experience is not a trivial task as people of different disciplines are required to co-operate. Because of the complexities involved in producing it, it is generally agreed upon that specialised tools encompassed in a game engine are a requirement for such a project [2].

The goal of a game engine is to provide a team with the necessary tools to build an interactive experience. The basic building block of such is the game object [10, 32]. A game object is a representation of an actor in the world. Game objects can be classified as either static or dynamic. A static game object is one that cannot interact with the other objects in the game, for example a scene background, whereas a dynamic one, such as the player character, can.

The method chosen to represent game objects is a central factor to consider when developing an engine. An overview of 2 approaches is detailed in the next subsection. Another issue to be addressed is the storage of game data as this can significantly impact performance.

2.1.1 Inheritance-based and Component-based Game Object Systems

A hierarchic approach to developing the game objects can be viable in a small scale project, given that the hierarchy may be shallow and adding new types is easy. The advantage of this approach is the fact that it is simple to understand. Modelling objects is intuitive, following the well-known OOP principles and a game can be developed relatively quickly from scratch [32].

The problems of this approach become apparent as the scope of the project begins to grow. As new types are introduced, the hierarchy becomes progressively harder to understand [10, 30, 32]. Changes to a class may require additional changes further down the hierarchy, as such it must be carefully maintained. Adding or removing functionality require the developer to understand the hierarchy structure, applying changes accordingly to all the affected subclasses. Another issue arises when new functionality must be added to types in different branches. If multiple inheritance is supported then multiple hierarchies must be accounted for any time a change is made. In addition this approach introduces coupling between the engine subsystems as, in order to achieve certain functionalities, a game object may require an aggregation of references to other systems in order to achieve its behavior. This structure enforces that game designers are dependent on software developers to introduce changes and maintain the hierarchy.

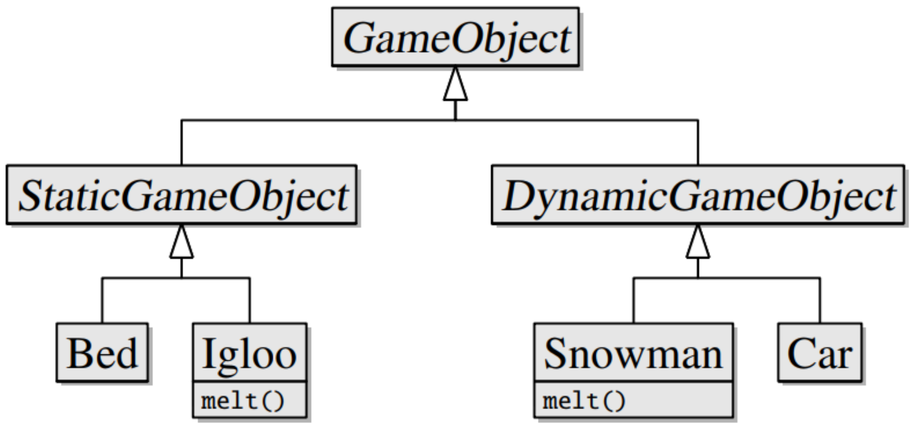


Figure 2.1: Adding the same behavior to types on different branches. Source: [32]

A game object system should be built in such a way that types and behaviors can be easily added or removed, while changes to a game object should not affect the types already defined. The design goals of such a system are high scalability and reusability, as games require a considerable number of game objects. There is also a need for various tools for the non-technical staff, such as visualization or content authoring tools, the maintenance of which can become problematic if a deep hierarchy is to be supported.

A convenient solution comes in the form of the Entity-Component-System (ECS) pattern. Following it enables developers to assemble various components into a robust architecture by favoring composition over inheritance [20, 24, 33].

A component can be viewed as an independent element of a system. The component only defines data and is not aware of other components or of the systems operating on it. Game objects are essentially defined by their aggregation of components. This allows them to be completely defined in data, which enables creating tools for non-technical staff to manipulate game objects independent of the engine developers.

Following this approach, game objects are no longer part of a deep hierarchy. Instead, a game object is modelled as an entity which is identified by an ID and the components that are mapped to it specify its capabilities. The systems contain logic and process the components with which they share the same aspect. For example a *Render-System* would only act on the entities that have a *Renderable* component and a *Transform*, disregarding any other entity. Such an architecture allows dynamical addition and removal of components to an owner as well as a way to notify said owner of certain events. The decoupling of components makes them reusable packages across multiple entities, adding a layer of indirection between components and system through the entity.

In a system that relies on composition in favor of inheritance, decoupling of high-level systems from the low-level objects is achieved, resulting in higher flexibility and easier maintainability. The system can scale to a large number of object types and components can be reused across different games. Behaviors can be changed at runtime by adding or removing components and system implementations can be easily swapped.

The biggest disadvantage of this approach is that it is not trivial to develop.

Artemis Engine [4, 32, 33] can be taken as a good example. The game objects are just an ID and an entity manager maps that ID with various components. The same approach is implemented in our engine. These components only hold state, while behavior is defined by a system that acts on a subset of an entity's components. The systems communicate via function calls and a system manager provides access to them.

Polyphony [33], a Javascript GUI framework, applies the ECS pattern to GUI programming. An entity is a JS native object encapsulated in a Proxy object, a component is a prototype object and a system is a function encapsulated in a Proxy object. Entities are used to represent both GUI elements as well as input devices. The state is updated in response to incoming events, each event having an associated action that is to be performed.

2.1.2 The Entity Component System

Central to an interactive system is the management of world state supported by the interaction of the underlying subsystems which must satisfy constraints of realtime performance and responsiveness [20].

The ECS pattern alone does not guarantee performance since it provides no well-defined rules for implementation. For example the way systems access the components of an entity is usually an iteration over the container(s) of components, but the type of container or rules of iteration are to be decided on by the implementor [20, 33].

This pattern has no fixed definition for the size of the components either. The requirements for a component are for it to provide an interface, be replaceable, able to communicate with other components and deployable at run-time [3, 6, 8, 21]. It is up to the component implementors to decide their size and versatility. Components should be designed in such a way that one component provides the necessary data to satisfy one aspect of a system.

In [20] a lock-free hash-map is used to store all existing components while systems and entities access them through their unique keys. Entities are represented as an ID and hold a key list for their associated components. Each system runs continuously in a thread. Each component has a consumer and producer version in the hash map in order to avoid waits while a component is being updated.

Nystrom [30] addresses the issue of data locality. The way data is stored and retrieved from memory can greatly impact cache performance. By maintaining packed arrays for each component type and implementing systems that process only the components with which they share the same aspect cache performance can be increased.

Our engine uses the packed array approach for storing components. Every component type is represented by a unique identifier and a component is identified by its type and owner entity. An entity is only and ID and can only map to one component of a given type. A system needs to implement the *System* interface and register itself with the coordinator class. The singleton coordinator keeps track of all the entities, systems and component containers and acts as an interface to the underlying containers. The engine provides the user with a default package of components and systems. To add a new component type it must be defined by deriving from *Component* and registered with the coordinator. The system that will process it must implement the *System* interface. If the user wishes the component to be accessible via script, then the new component type must also be registered with the *ScriptManager*.

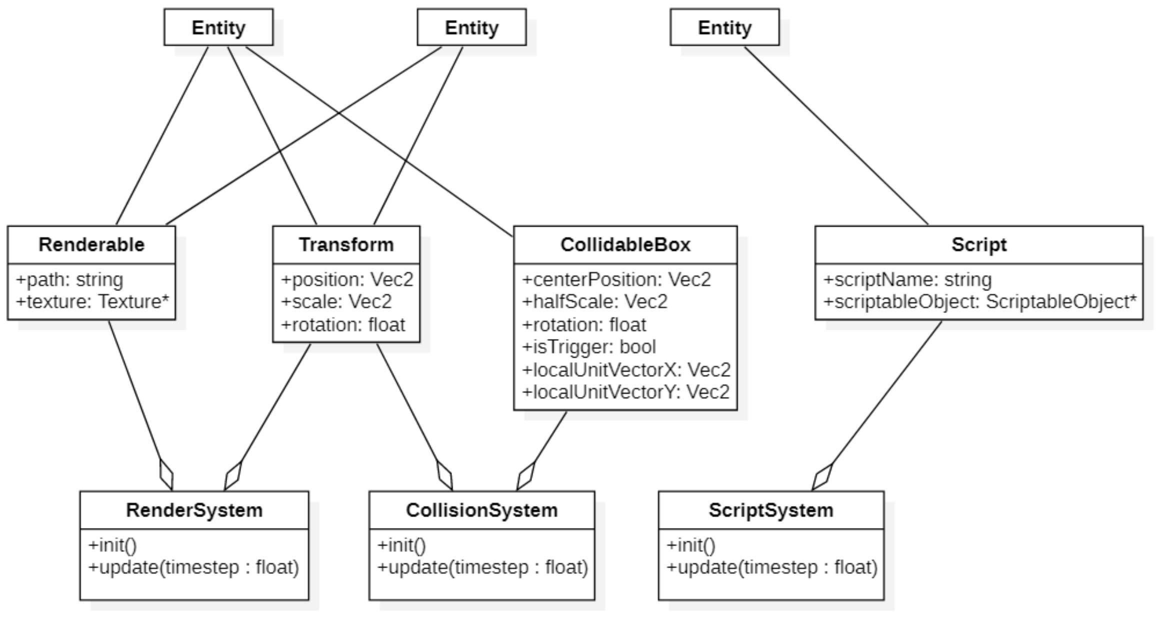


Figure 2.2: Matching components with entities and systems.

2.2 The AngelScript Environment

Complex software systems usually require multiple programming languages in order to meet the needs of today's industry [28]. Most game engines implement at least an additional programming language with the goal of providing users with a safe environment to specify game properties [38, 39].

In this section we detail the uses of integrated scripting environments and provide a set of examples of tools used for game development purposes.

2.2.1 The Need for a Scripting Environment

A game engine provides a platform that specifies how to create an application, but it does not provide the actual functionality of the application. The core components of the engine are separated from the game specific code, thus adding another layer of abstraction and reducing engine complexity. A data-driven architecture is achieved by placing game code outside of the engine, effectively transforming it into an asset that will be loaded at run-time. Thus game programmers can focus on gameplay aspects, while engine developers can focus on the core technology that supports the game [10, 15, 38]. This limits the scope of game types that an engine can produce, but it also implies that it is highly reusable within a particular genre [1].

To allow designers to specify behavior without the need to learn a complex API, game engines often incorporate scripting systems. This can be seen as early as 1987 in SCUMM [15] an engine designed for adventure game development created by LucasArts.

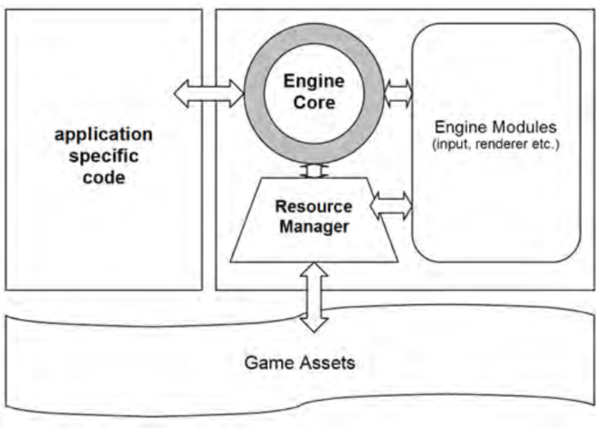


Figure 2.3: The application specific code is stored as external assets. Source: [1]

Scripting environments provide the game programmers with a protected environment to express game properties for large sets of data. Scripting languages offer the necessary level of control to specify entity behavior, without the complexities of understanding the workings of the underlying engine.

A classification of scripting systems for computer games is presented in [1]. Scripts are divided into 3 categories. ST1 scripts run once at program startup to set internal parameters and are usually just a list of values. ST2 scripts run on event triggers according to the state-effect pattern. These are further divided into Event Handler Scripts which define behavior to be executed when predefined events occur and Event Oriented Scripts which require the definition of both the event and the event handler. Finally ST3 scripts can be used as looping scripts which execute each cycle and act as a control loop or as regular scripts that execute once from start to finish concurrently with the engine.

An important design decision is whether to use an existing scripting language or develop a new one. Creating a custom language has the benefit of allowing complete control of the engineering team over the features of the language. The drawbacks to be considered are the time and costs needed to develop and test such a language [39]. On the other hand using an existing language can save development time and does not incur the cost of creating a new one [9]. The issue becomes finding a good fit for the problem at hand.

2.2.2 Integrated Scripting Tools

An additional benefit brought by scripting tools is the fact that they can be made available with the shipping game and allow the public to modify the existing game. This usually results in an increased shelf life of the product [1]. Sometimes these tools can be used to create entirely different experiences, one of the popular examples being the birth of the Multiplayer Online Battle Arena genre in 2003, which was originally a Warcraft 3 mod created by the community.

Aurora Toolkit shipped with the game 'Neverwinter Nights' and provides tools for creating game environment, virtual actors and specifying their behavior [1].

Popular commercial game engines such as Unreal Engine or Unity provide multiple scripting mechanisms for various aspects of a game. Unity supports C# and Javascript for specifying game object behavior as well as a visual scripting tools for animations. Unreal supports scripting through C++ or its visual Blueprint system.

Dark Forces [15] uses an opcode language called INF to move levels, spawn enemies and track player progress.

Python has been used to create games such as EveOnline and Blade of Darkness. Libraries like SWIG simplify the binding process of Python to C++, but there still exists additional overhead, as it requires maintenance of glue code checks [9].

We chose Angelscript (AS) [10, 12] as the scripting language for our engine. AS is a strongly typed, interpreted language that supports automatic memory management. Currently the engine implements ST3 scripts. The memory is managed via reference counting in conjunction with a grabage collector. Scripts are ran inside a VM and allow code adjustments while the engine is running. The reason for choosing AS is that it shares the calling convention of C++. Thus engine-side functionality can be registered directly to the AS environment, without the need to define intermediate functions. AS operates by retrieving pointers to engine-side classes, functions or variables that were registered with it via its API. Parameters to script functions are passed using helper classes via *asIScriptContext* class, which acts as the interface for script execution. The script contexts are aggregated in the *ScriptManager* class. It is responsible for creating, storing and executing script controllers, signaling errors in the scripts and registering types and functions with AS. In order for an entity to implement a script it is required to hold a *ScriptComponent*, which specifies a script file to define game object behavior. The *ScriptSystem* executes the scripts for all entities that hold such a component.

2.3 Constructing the Game Environment

Level design typically requires the involvement of a team of dedicated designers. Constraints must be met regarding playabaility and, in addition, playing through a level must be considered a fun experience by the player. A level is built by arranging static components, in our case 2D tiles, and dynamic components such as enemies or other non-player-characters [36]. Tiles determine the way in which a level can be traversed. Their arrangement together with the placement of enemy characters determine the level playability, difficulty and player enjoyment, often requiring precision and speed. Thus constructing a level is both an optimization problem with respects to the amount of fun and difficulty it provides as well as a constraint satisfaction problem regarding its playabaility.

As the demand for content rises, Procedural Content Generation (PCG) has become an option developers are increasingly taking into account in order to lessen the burden of designers or replace them altogether [29]. For the most part it has been used for the generation of game environment [7, 25], though there are examples of PCG being used to generate game systems, game scenarios, visuals or items [13]. We employ PCG techniques to generate the game environment, taking inspiration form the Roguelike genre in doing so.

We incorporate into our engine a tool for procedural two-dimensional tile-based level generation. In this section we provide a brief overview of content generation used in various game development projects followed by the techniques we have taken into account to implement such a mechanism.

2.3.1 Procedural Content Generation

The algorithmic generation of game assets provides benefits in terms of development cost and development time as well as cost of change, allowing designers to specify the properties a level should have, as opposed to how levels should be assembled, by tweaking sets of parameters. What must be taken into account when employing PCG techniques is the difficulty of achieving the proper level of control over the generation of assets, the complexity involved in the development and testing of tools, the lack of standardized tools for game content generation as well as the fact that such methods tend to be domain-specific [29, 34, 36].

The use of PCG methods is noted in realizing systems such as particle systems, an animation system and rendering systems in [34]. The animation system changes the animation of in-game characters according to a notoriety value associated to the player. Difficulties are encountered in the development of a procedural sky rendering pipeline and the assurance of correct algorithm outputs.

A tool for supporting the designer with suggestions in creating 2D maps through Novelty Search is proposed in [27]. It provides a map editor for the user to design the map. The tool evaluates playability constraints and gameplay properties, providing live alternative suggestions to the map design via Novelty Search.

In [36] the Feasible-Infeasible Two-Populations genetic algorithm is employed with the goal of generating levels that meet constraints expressed in terms of difficulty and perceived fun, being general enough to be applied in a variety of games. It provides examples of generating entire levels by taking as input the specification of level elements and functions that define the desired constraints said elements must satisfy.

A geometry assembly algorithm applied in Infinite Mario is presented in [29]. Its main goal is to generate interesting geometry without knowledge of game-specific mechanics, focusing on variety over playability.

2.3.2 Generation Techniques

This part details the methods we have considered to implement an environment generation mechanism. We have searched for techniques that allow specifying playability constraints and have the potential to produce relatively diverse results without being bound to a single game. A desirable trait is that users should be able to influence the outputs, without requiring the aid of software engineers.

Occupancy-Regulated Extension

The main goal of Occupancy-Regulated Extension (ORE) is to generate diverse levels by assembling chunks of tiles provided in a library, where each chunk has associated anchors that denote potential locations the player might find himself in [29].

Chunks are selected from the library and placed in the level such that the anchors are preserved and content is iteratively generated. Chunk placement is achieved by selecting an anchor of the existing geometry to expand, searching through the library for one chunk that matches the existing content and finally placing the chunk in the world. Chunk selection uses weights computed per chunk. They are obtained according to the annotation values in the library and to how many times a chunk was used. The chance that a chunk is selected is proportional to its weight.

The fact that ORE uses an authored library of elements can be considered both a strength and a drawback. On the one hand it allows for complex custom components to be used and can also function as a partner for a designer, since the algorithm is not hindered by existing elements. As such, changes to level style require merely updating the library. Experiments [29] show interesting outputs with complex level patterns that introduce safe zones and layering [17]. On the other hand in order to use the algorithm a library of chunks must be created and maintained. In addition, ORE only strives to satisfy the constraints regarding anchor points and, depending on the mechanics available to the player, might generate environment that cannot be traversed. The cost of creating and managing said library deterred us from pursuing this approach.

Genetic Algorithms

A Genetic Algorithm (GA) appears as a good fit for implementing a content generation tool [18]. GAs mirror natural selection as they select the fittest individuals of each generation for reproduction. They are used to optimize an objective (fitness) function with multiple variables that represents the distance of an individual to some objective state. In the case of constrained optimization the usual approach is to penalize solutions proportionally to the graveness of violation.

The general form of a maximization constrained optimization problem is illustrated in Figure(2.4), where *Z(x)* is the fitness function value for individual *x*, *E*,*F*,*G* model constraints as inequalities that must be satisfied by a solution and S_i is the set of decision variables.

The penalty function in Figure(2.5) represents the problem in a way that allows for it to be directly encoded in a GA, where *W(x)* is the fitness of individual *x*, *P(x)* is the combined penalty of all constraint violations of *x* and *Z(x)*, as defined above is the fitness of the individual. This can lead to quick rejection of solutions, which may be unwanted

The genotype is represented linearly as an array of Design Elements (DE). A DE represents one element of a level, regardless of its size. This means that DEs can represent equally one tile of a level as well as an entire room. A level is encoded by specifying the elements it should contain.

$$\max_{x_i} z = Z(\boldsymbol{x}), \text{subject to } E(\boldsymbol{x}) \geq \boldsymbol{a}, F(\boldsymbol{x}) \leq \boldsymbol{b}, G(\boldsymbol{x}) = \boldsymbol{c}, x_i \in \mathcal{S}_i$$

Figure 2.4: A maximization constrained optimization problem. Source: [18]

$$\max_{x_i} z = W(\boldsymbol{x}) = Z(\boldsymbol{x}) - P(\boldsymbol{x})$$

Figure 2.5: Penalty function encoding. Source: [18]

Crossover between the individuals of a population is used to obtain the population of a future generation of individuals. There are many ways to implement such an operation, of which we enumerate only a few. Single-point crossover divides each set of genes of the individuals into two disjunct sets and creates children that contain a division from each parent. K-Point crossover follows the same principle, but instead of creating two sets it results in K disjunct sets of genes which are passed to the offspring. Finally, uniform crossover states that every gene is equally probable to be passed to the offspring.

Mutation operations are also used to modify the genes of an individual. The mutation of an individual typically has a smaller probability compared to crossover and results in some change to one of the genes of an individual.

Genetic algorithms are especially sensitive to how the fitness function model and may produce undesirable results in the context of a vague definition of it [18]. It is argued in [22, 23] that positive results can be obtained by abolishing such a function and instead searching only for novelty, that is, genetic diversity within a population.

Novelty Search & Minimal Criteria Novelty Search

Novelty search (NS) is an evolutionary approach used when fitness is difficult to quantify. Its main goal is that of exploring the search space as opposed to optimizing a fitness function [22]. Its objective is to increase the diversity of the solution set by favoring individuals situated at a greater distance from the population. Solutions are generated such that a function of the average distance between *k* neighbours in a set of individuals is maximized, where *k* is chosen experimentally. The average distance *p(x)* for individual *x* is computed as in Figure(2.6), where μ_i represents the *i*-th closest neighbour of *x* and the *dist* function quantifies how different 2 individuals are.

$$\rho(x) = \frac{1}{k}\sum_{i=0}^{k} \mathrm{dist}(x, \mu_i),$$

Figure 2.6: The function of novelty. Source: [22]

Minimal Criteria Novelty Search (MCNS) [23] provides a more radical take on classic NS by stating that that if an individual does not satisfy the minimal criteria for reproduction then it should not be allowed to further reproduce, thus setting the fitness of individuals that do not qualify as solutions to 0. This can improve fitness values obtained through Unconstrained NS, but limits the effectiveness of the technique in high-constraint problems.

Both NS and MCNS seem reasonable options for implementing a generation system. Results [22, 23, 26, 27] show that Unconstrained NS performs well in relatively small spaces. Still it suffers from a small set set of feasible individuals in larger spaces as the likelihood of generating infeasible offspring from feasible parents is high due to the focus on exploring the search space. MCNS tends to perform random search on large maps as it cannot find feasible individuals. Nonetheless it yields a high number of feasible individuals due to mutations, though at the cost of offering low diversity.

Feasible-Infeasible Two-Population

FI-2Pop genetic algorithm [18, 25, 26, 36] allows specifying a set of hard constraints and high-level goals that must be satisfied for the generation of a level that is considered playable. It maintains a population of 'fit' individuals which satisfy all constraints and a population of 'unfit' individuals which do not. The algorithm requires 2 fitness functions: one that drives change of the valid individuals towards a global optimum and one for targeting the satisfaction of the specified set of constraints for the invalid individuals. The individuals that were rejected due to a violation of constraints resulting from crossover or mutation are continuously evolved such that the number of violations is minimized. An individual that satisfies all the constraints is moved to the set of feasible individuals and continues evolution. Diversity is maintained by frequent migration between populations and by the surviving genetic material in infeasible individuals

A fitness function is applied to them based on the criteria specified by the level designer. A common practice is to evolve the infeasible individuals towards feasibility while the feasible ones pursue novelty.

We have chosen to implement this method for the generator of our engine. As opposed to ORE, FI-2Pop does not require the management of a chunk library and only calls for specifying the tile types a map should have. FI-2Pop also avoids the random searches NS and MCNS tend to perform due to the genetic material kept in the infeasible population. In addition, once a feasible individual is found the whole population starts moving towards it, providing better outcomes especially in small search spaces. This results in slightly lower diversity but more fit solutions and better satisfies our needs for interesting playable solutions.

In our case a DE in the genotype represents one of 11 types of rooms, out of which 3 represent the Safe Zone pattern [17]. Rooms contain different kinds of paths that connect them to other rooms. Multiple connected rooms form a component. Thus the constraint that must be met by an individual to be considered feasible follows: the player should be able to reach any room from the room it is currently in. In other words the level must be formed of a single component. The individual is penalized in proportion to the number of components that exceed one. Once an individual is deemed feasible it evolves towards distinguishing itself from the other individuals. The individual is rewarded for having a room rotated differently as well as for using a different room type compared to the rest of the population.

2.4 Collision Detection

Collision detection is a major factor in most games as it is one of the main ways interactivity between game objects is achieved. The goal of a collision system is to be aware about whether or not the objects in the world are in contact and to perform some actions if they are. This typically involves setting the objects back to the positions they were in the frame before they collided and/or send a message to other systems that should perform some action in response to the collision event.

Collision detection is a broad field and what we present here is not an in-depth analysis on the topic as a whole. Nevertheless we present in the following a set of methods we have considered to detect collision between objects in a two-dimensional setting as well as techniques to reduce the number of computations required to do so.

2.4.1 Space Partitioning

Space partitioning is a method used to divide the game space into smaller subspaces and perform collision computations only on the objects contained in each subspace [19]. A variety of methods exist for handling this problem, which usually involve the construction and maintenance of some data structure.

In [30] a grid-based approach is taken to separate the game world. For each object, collision checking is performed only with the objects present in its cell and in the adjacent cells. The advantage of this approach is that it is simple to implement and does not require maintaining a tree structure. Thus it can perform quite well in small-scale games. The issue is apparent in larger games. Usually one does not need to track the whole game space for collisions, only the parts that affect the player experience. A similar approach is presented in [35]. Each object in the array is required to map to a tile or multiple adjacent tiles on a grid with fixed dimensions. The time complexity of tile arrays for insertion and removal is *O(1)*. Speed is achieved at the cost of space as the representation requires *O(w*h)* space, where *w* and *h* are the dimensions of the grid. Search operations achieve *O(w′*h′)*, with *w′* and *h′* being the dimensions of the subspace. This can prove to be a benefit as most tile-based games don't span complex game entities, thus *w′* and *h′* can take appropriately small values, but it still suffers from performance issues in larger games, due to it tracking the whole game space.

An evaluation of the performance of quadtress and k-d trees is also presented in [35] of which we provide a brief overview.

A Quadtree node represents an area in the game space. Each node in the tree has four children such as to divide the parent in four equal areas and each area can contain a given maximum number of elements before requiring to be divided itself. Search, remove and insert operations can be performed in *O(logn)* time with *O(n)* at worst. An issue is that objects of large sizes that do not fit any subspace are completely rejected by the algorithm, but this can be worked-around if care is taken into implementing it.

With k-d trees subsets of elements are built by splitting the space with alternating vertical and horizontal axes at each level. Every leaf node represents an entity, while non-leaf nodes are equivalent to a sub-space. K-d trees require *O(nlogn)* time to be constructed and have a search time cost of *O(logn + k)*, with *k* being the number of entities in a subspace, and removal time of *O(logn)*. Experiment results indicate a poorer performance as opposed to quadtrees.

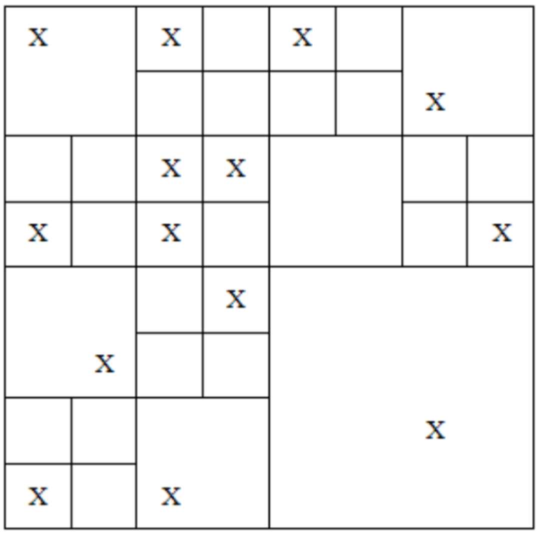


Figure 2.7: Quadtree Partitioning. Source: [35]

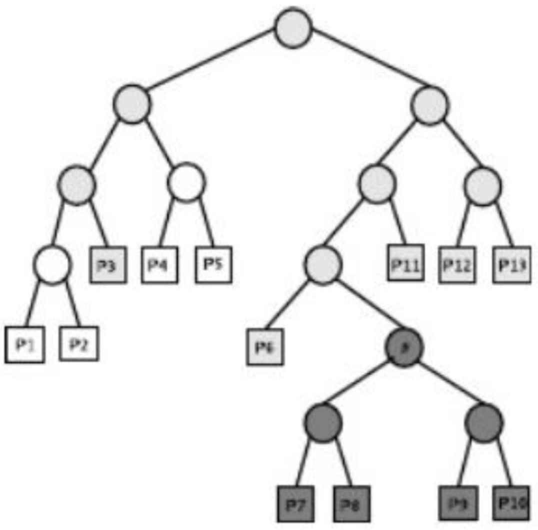


Figure 2.8: K-D Tree Partitioning. Source: [35]

2.4.2 Determining Object Collision

Once the data structure for dividing the game space is chosen, the nature of how object-to-object collision will be performed must be decided upon. We focus on the bounding boxes and bounding spheres approaches, particularly in the two-dimensional space.

Axis-Aligned Bounding boxes are analyzed in [37]. The method encloses an object into rectangles that are parallel to the coordinate system axes, that is, they are defined by their minimum and maximum coordinate on each axis. This allows for fast computation of collision, but comes with the drawback that the bounding box must remain axis-aligned even when the object is rotated. This can result in boxes that do not match properly to the object they are bound to.

In [5] the bounding spheres approach is used as a preliminary test to check whether or not two objects interpenetrate. It states that if the result of summing up each object's radius is smaller than the distance between their centers then the objects cannot collide. Only if this condition is not fulfilled does the collision system produce further calculations

The Separating Axis Theorem, used to determine intersection of convex polygons, is presented in [16]. A separating axis of two polygons is a line on which the projections of the polygons do not intersect. This is equivalent to the existence of a separating line between the polygons, such that the two exist on separate sides of the line. This, in turn, implies that if a separating line exists then it is parallel to at least

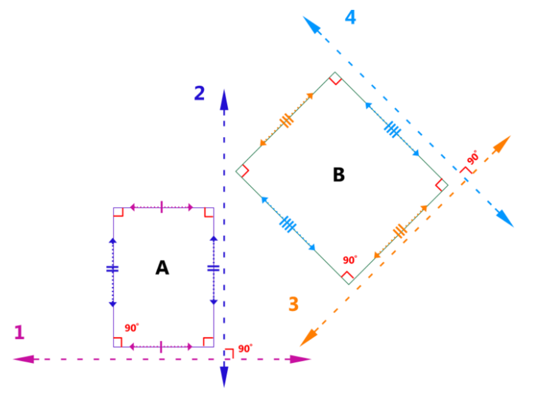


Figure 2.9: Separating Axis theorem for OBBs in 2D. Source: [16]

$$\begin{aligned}
&| T \bullet Ax | > W_A + | (W_B * Bx) \bullet Ax | + |(H_B * By) \bullet Ax | \\
&| T \bullet Ay | > H_A + | (W_B * Bx) \bullet Ay | + |(H_B * By) \bullet Ay | \\
&| T \bullet Bx | > | (W_A * Ax) \bullet Bx | + | (H_A * Ay) \bullet Bx | + W_B \\
&| T \bullet By | > | (W_A * Ax) \bullet By | + | (H_A * Ay) \bullet By | + H_B
\end{aligned}$$

Figure 2.10: Conditions for the existence of separating axes. Source: [16]

one edge of one of the rectangles. The theorem states that if at least one separating axis exists between two polygons then they do not intersect. This requires checking the existence of separating lines for each dimension of each polygon, that is at most four tests in our two-dimensional setting. Figure(2.9) illustrates this process for two rectangles. The conditions that must be met are given in Figure(2.10), where *Pa, Pb* are the centers of the rectangles, *Ax,Ay,Bx,By* are the unit vectors corresponding to the local X and Y axes of each rectangle and *Wa,Wb,Ha,Hb* are the halved width and height. The halved dimensions slightly increase the computation speed and can be used since each half of a rectangle split by the middle will always be symmetrical to the other half. As stated, if at least one of the conditions is true then the two rectangles are not colliding

This is the method we have chosen to implement, since we wanted our engine to properly support rotating objects. In the context of the ECS we provide a *Collidable* component that stores the box center, halved scales and rotation as well as caches the unit vectors of the box for its local X and Y axes. The *CollisionSystem* checks the conditions in Figure(2.10) for each pair of objects which have the distance between their centers larger than the sum of each box's maximum dimension side. This essentially means that bounding circle testing is done prior to OBB testing, the latter only being performed if the objects are close to each other.

Chapter 3

Software Application

In this chapter we present the general architecture of our dynamic application and provide an in-depth view of its systems. High performance and responsiveness constraints must be met due to the interactive nature of games.

The objective we set ourselves was to provide users with a platform providing the necessary systems to create a Roguelike game without requiring advanced technical knowledge on their part. We also wanted to have an architecture that enables software engineers to extend the application with their own systems. To this end we have used the ECS pattern.

In addition we provide a scripting system such that game specific code can be separated from the engine and treated as an asset. The scripting environment should tolerate faults in the user-defined scripts by providing a safe environment which protects the underlying engine.

Finally we implemented a PCG tool using the FI2-Pop algorithm. We provide a GUI to give access to the algorithm parameters, but it is to be noted that some specific changes require the aid of software engineers.

Other tools include a logging mechanism, input handling and a 2D renderer.

We have chosen C++ as the programming language to implement the engine, using the C++17 dialect. It offers modern programming language features while still allowing for lower-level control of memory, creating a balance between high performance and ease of use. Furthermore C++ still remains an industry standard in game engine development, not only because of its runtime speed due to it being a compiled language, but also because of the abundance of libraries available for it. We have used a number of C++ libraries, the details of which we will present in the following sections.

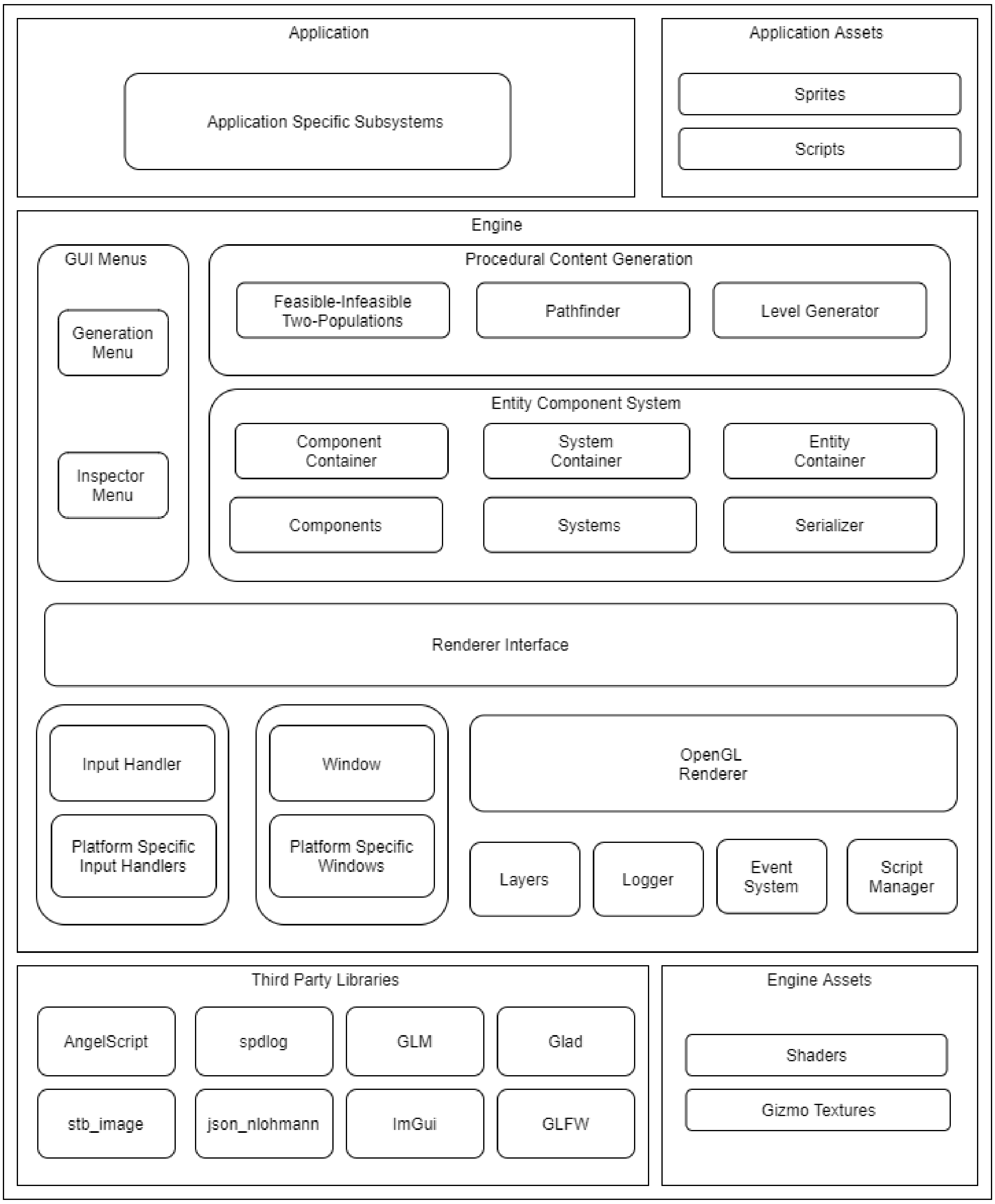
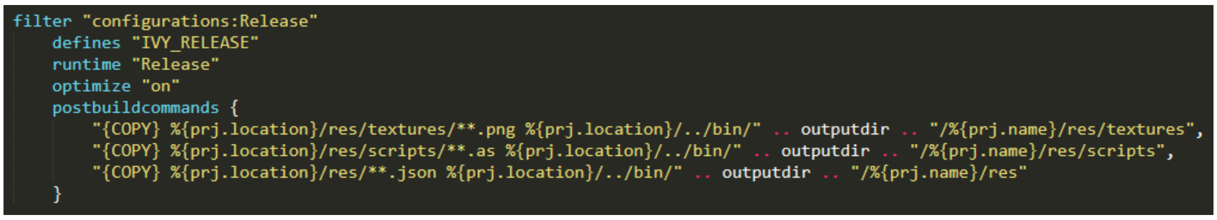


Figure 3.1: High-level architectural view of the engine.

3.1 Utilities

We have used the Premake5 utility (*https://github.com/premake/premake-core*) to generate project files and manage dependencies. It allows specifying various workspace settings such as build configurations and project dependencies through a Lua scripting environment and offers support for C, C++ and C# based projects. The workspace encompasses multiple projects and specifies for each the include files and directories they are dependent on, links to external libraries, symbol definitions and build configurations (for instance Debug and Release). Filters are used to specify build settings for specific configurations. The specification of a filter for the Release configuration of the application project that uses the engine is depicted in Figure(3.2). It defines the "IVY_RELEASE" preprocessor definition for the Release configuration which turns on compiler optimizations and, after building, copies the resource files that the project is dependent on to the output directory. Besides the Application project, configurations are provided for the Engine, AngelScript, GLAD, GLFW and ImGUI projects, which will be detailed in subsequent sections.

We provide a batch file for Visual Studio users on Windows that invokes the Premake5 utility to generate the project files.



```
filter "configurations:Release"
    defines "IVY_RELEASE"
    runtime "Release"
    optimize "on"
    postbuildcommands {
        "{COPY} %{prj.location}/res/textures/**.png %{prj.location}/../bin/" .. outputdir .. "/%{prj.name}/res/textures",
        "{COPY} %{prj.location}/res/scripts/**.as %{prj.location}/../bin/" .. outputdir .. "/%{prj.name}/res/scripts",
        "{COPY} %{prj.location}/res/**.json %{prj.location}/../bin/" .. outputdir .. "/%{prj.name}/res"
    }
```

Figure 3.2: Premake5 excerpt from the application project script. Specifies options for the Release configuration.

A logging system built on top of Spdlog library (*https://github.com/gabime/spdlog*) is used for capturing the important system operations. Spdlog is a lightweight single-header logging library for C++. Loggers are obtained via a Factory function and offer convenient format customization with a wide range of readily available patterns. Engine operations are logged by an engine-side logger and the logs are available to the user through the console. A client-side logger is made available in order to log application operations.

We provide Layers to logically separate the entities in a game. The Layers are kept in a stack and updated in bottom-up fashion. This structure is illustrated in Figure(3.3). The *onAttach* and *onDetach* methods are implemented to defer the initialization and destruction actions of the elements in the layer. For instance on attaching a GUI layer for displaying player properties, health for example, one could define in *onAttach* font properties such as the style or size for the elements of the layer in question. In the *update* method the game programmer specifies game logic specific to each layer according to the Update Loop pattern. Generally speaking the game logic can be transformed into an asset by moving it to scripts, but in cases where more fine-grained control is desired, the ability to specify logic in C++ may come in handy. Layers also provide an *imGuiRender* method, used to define GUI elements. It's purpose is not to build a game user interface, but rather to allow software engineers to implement visualization tools to support the game designers.

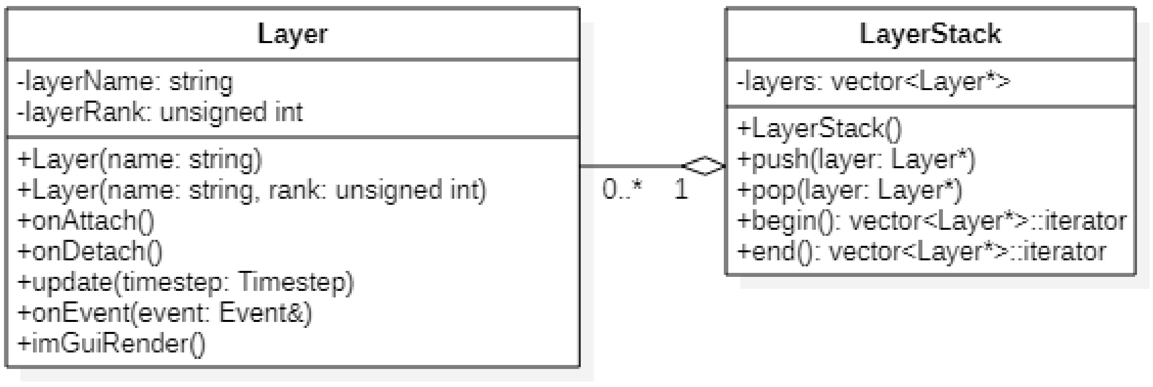


Figure 3.3: LayerStack: An aggregation of Layers that logically separate game entities

In addition, layers determine the way events are processed by the event system, every layer implementing an *onEvent* method that acts on a given Event.

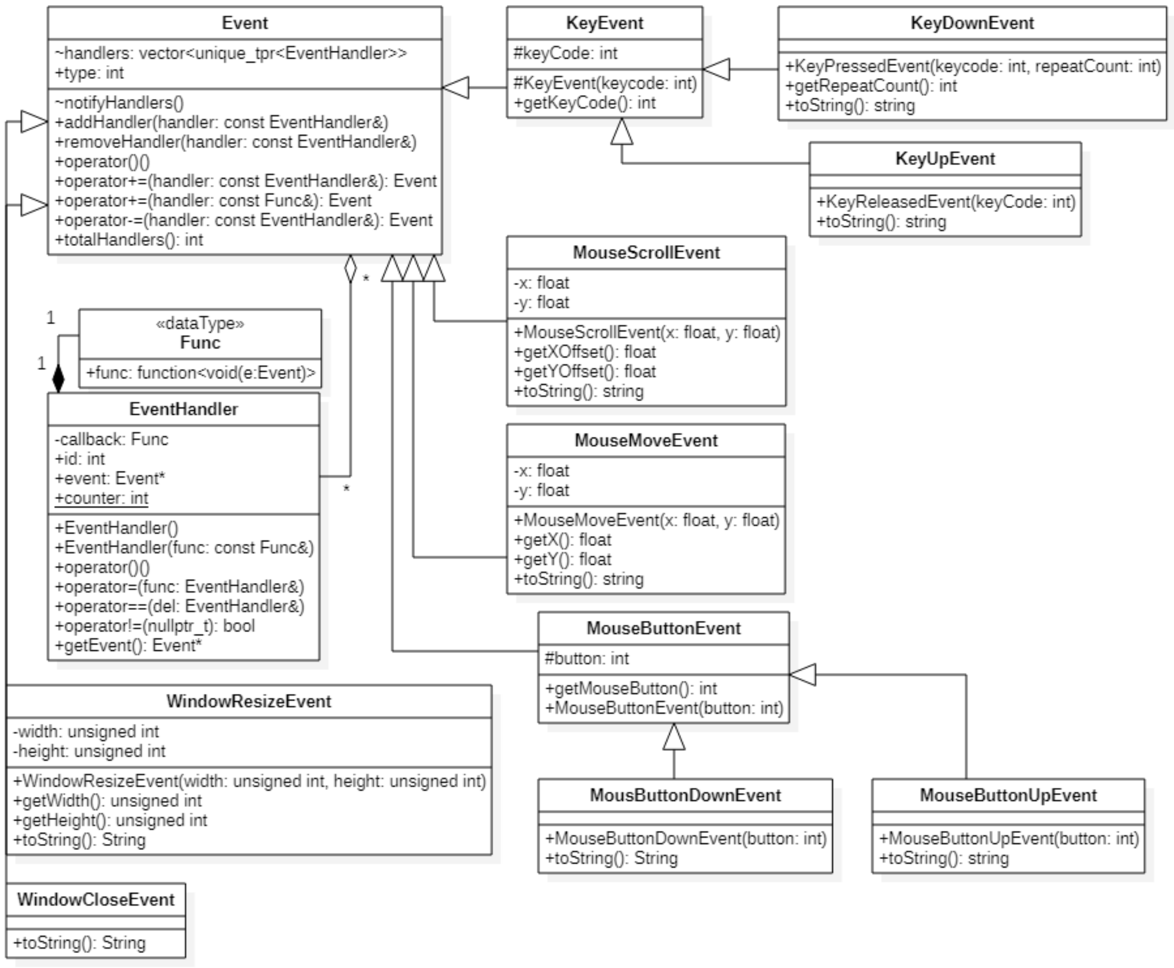
3.2 The Event System

The Event System functions according to the Observer pattern. The *Event*s act as subjects and each *Event* instance holds references to the *EventHandler*s that subscribed to it and notifies them whenever it occurs. The *EventHandler*s encapsulate a callback function that gets executed in response to a raised event.

*KeyEvent*s and *MouseButtonEvent*s encapsulate the code of the key which generated the event. The *MouseMovedEvent*s and *MouseScrollEvent*s store the coordinates of the cursor and scroll offset respectively. Finally *WindowEvent*s are used to notify the application about changes regarding the window it is running in. There exists a single window, so the events will always be raised relative to it.

Events are processed by the layers in a top-down fashion. The point of this is that the user usually interacts with the layer that is at the utmost top of the layer stack, thus it should be the first to handle the event that was raised before the event has a chance to reach subsequent layers.

As of now the engine does not provide a separate phase of event resolution, thus events are handled the moment they occur. Creating an event processing stage as well as adapting the system to function in a multi-threaded environment are considerations for future development.



Event
~handlers: vector<unique_tpr<EventHandler>>
+type: int
~notifyHandlers()
+addHandler(handler: const EventHandler&)
+removeHandler(handler: const EventHandler&)
+operator()()
+operator+=(handler: const EventHandler&): Event
+operator+=(handler: const Func&): Event
+operator-=(handler: const EventHandler&): Event
+totalHandlers(): int
KeyEvent
#keyCode: int
#KeyEvent(keycode: int)
+getKeyCode(): int
KeyDownEvent
+KeyPressedEvent(keycode: int, repeatCount: int)
+getRepeatCount(): int
+toString(): string
KeyUpEvent
+KeyReleasedEvent(keyCode: int)
+toString(): string
MouseScrollEvent
-x: float
-y: float
+MouseScrollEvent(x: float, y: float)
+getXOffset(): float
+getYOffset(): float
+toString(): string
MouseMoveEvent
-x: float
-y: float
+MouseMoveEvent(x: float, y: float)
+getX(): float
+getY(): float
+toString(): string
MouseButtonEvent
#button: int
+getMouseButton(): int
+MouseButtonEvent(button: int)
MousButtonDownEvent
+MouseButtonDownEvent(button: int)
+toString(): String
MouseButtonUpEvent
+MouseButtonUpEvent(button: int)
+toString(): string
«dataType»
Func
+func: function<void(e:Event)>
1
1
*
*
EventHandler
-callback: Func
+id: int
+event: Event*
+counter: int
+EventHandler()
+EventHandler(func: const Func&)
+operator()()
+operator=(func: EventHandler&)
+operator==(del: EventHandler&)
+operator!=(nullptr_t): bool
+getEvent(): Event*
WindowResizeEvent
-width: unsigned int
-height: unsigned int
+WindowResizeEvent(width: unsigned int, height: unsigned int)
+getWidth(): unsigned int
+getHeight(): unsigned int
+toString(): String
WindowCloseEvent
+toString(): String

Figure 3.4: Class diagram depicting the event system structure

3.3 Platform Support

We have mentioned events being raised in response to user actions such as button presses and window manipulation, but these actions require handling according to the platform the engine is running on.

Currently Windows is the only supported platform and both input handling and window manipulation are built on top of the GLFW3 library (*https://github.com/glfw/glfw*). GLFW3 is a multi-platform open source library for OpenGL and provides an API for creating windows and contexts as well as input handling.

In the case of windows, GLFW provides support for both "windowed" windows as well as full screen windows, allowing modifications to resolution, size, refresh rate, pixel dimensions and many more properties. The windows implement double buffering and GLFW provides the methods *glfwSwapBuffers* and *glfwSwapInterval* to specify buffer swapping and the rate at which this occurs relative to the monitor refresh rate. We provide an abstract *Window* class with a Factory method method that takes as input the window properties, namely its name and dimensions, and returns a window handle. This decoupling warrants that extending the engine to support window creation for multiple platforms should not pose issues.

Input handling is achieved by polling the window for the received input events. Key Events return the key code of the key in question and an associated action which can be one of *GLFW_PRESS*, *GLFW_RELEASE* and *GLFW_REPEAT* which signal a key press, release and a maintained press respectively. Mouse input events set the cursor position relative to the screen coordinates. The mouse key events function similarly to the keyboard key events. Here too, the abstract *InputHandler* is decoupled from the actual handler which is implemented per platform.

3.4 Renderer

We chose to use OpenGL as the primary graphics API for the engine due to it being cross-platform and having a relatively comfortable API. It's designed such that it is agnostic to the underlying graphics hardware, each GPU provider implementing it separately according to its interface. It was developed by Silicon Graphics, with the first version being released in 1992. Currently it is maintained by the Khronos Group (*https://www.khronos.org*). The goal of this paper is not to provide an extensive look on OpenGL so we point the reader to [40] for an in-depth look on the subject and to the official Wiki page at *https://www.khronos.org/opengl/wiki/*.

We continue this section with an overview of the tools used to build the renderer of the engine and a description of its general architecture.

We have used GLAD (*https://github.com/Dav1dde/glad*) as the OpenGL function loader. The OpenGL Application Binary Interface only offers support for version 1.1 of OpenGL on Windows, version 1.2 on Linux, while on MacOS it depends on the OS version. These versions offer very limited flexibility, having a fixed-function pipeline and much functionality has been deprecated with the release of OpenGL 3.0. In addition OpenGL is implemented by the graphics driver provider and cannot be linked as a library. Thus in order to support Modern OpenGL functionality, one is required to retrieve the appropriate function pointers from the driver, which is automated through GLAD. GLFW3 is added as another layer on top of GLAD.

Rendering is achieved through a layer of abstraction which decouples the engine rendering API from the used graphics API. We use the Factory Pattern and Command Pattern to achieve this decoupling. In order to create a renderer that uses a given graphics API the abstract *RenderAPI* class is used to return a pointer to a concrete API. The *RenderCommand* is a static class. It holds a pointer to the used *RenderAPI* and issues it commands through its methods to execute functions. As mentioned, currently the engine only supports OpenGL, but, given this decoupling, extending it to support other APIs should be trivial.

Abstractions are provided for the Render Context, Vertex Arrays, Vertex Buffers, Index Buffers, Textures and Shaders, all of which require an implementation depending on the used graphics API.

A Vertex Array Object (VAO) encapsulates a Vertex Buffer Object (VBO) and an Index Buffer Object (IBO). The VBO stores the vertex positions and texture coordinates of the object to be rendered. The IBO is responsible for specifying the order in which vertices must be connected in order to form triangles that can be rendered.

A shader is a user-defined program that executes on the GPU and provides code for the programmable stages of the rendering pipeline. Currently we provide a single shader program defined in OpenGL Shading Language (GLSL). It specifes a vertex shader and a fragment shader.

The vertex shader takes as input the position of the render target and the texture coordinates. It receives two 4x4 matrix uniforms. One of them is the view-projection matrix which is set on the beginning of each frame. The other represents the model matrix of the object to be rendered. The vertex shader calculates the model-view-projection matrix of the object and passes the texture coordinates as well as the object's position to the fragment shader.

Before reaching the fragment shader the triangles pass through a rasterazation phase where the parts of the triangle that do not fit the screen are discarded and the remaining parts are broken up into fragments having the size of a pixel. The fragment shader takes as input the rasterized output of the vertex shader. It runs for every pixel and sets it to the appropriate color according to the texture coordinates.

The Shader class provides the necessary functionality to read and compile a shader as well as to upload uniforms to the shader program

For the above mentioned matrix calculation we use the OpenGL Mathematics or GLM (*https://github.com/g-truc/glm*) library. GLM is a mathematics library based on GLSL, primarily intended for graphics. It provides vector types, matrix types, geometric functions and other useful mathematic functions. Matrices are used to implement the model, view and projection matrix transformations applied to the game objects, while vectors are used to internally represent the dimensions and position of objects.

In order to load textures we rely on the single file library *stb_image* of the Stb collection. Stb is a collection of single file libraries for C/C++ which is available at *https://github.com/nothings/stb*. *Stb_image* supports multiple image formats, most notably PNG, JPEG and BMP. The image data is encapsulated by a *Texture* object and stored in a buffer of unsigned 8-bit values corresponding to the pixel data. This data is then passed to OpenGL to generate a texture image in RGBA format.

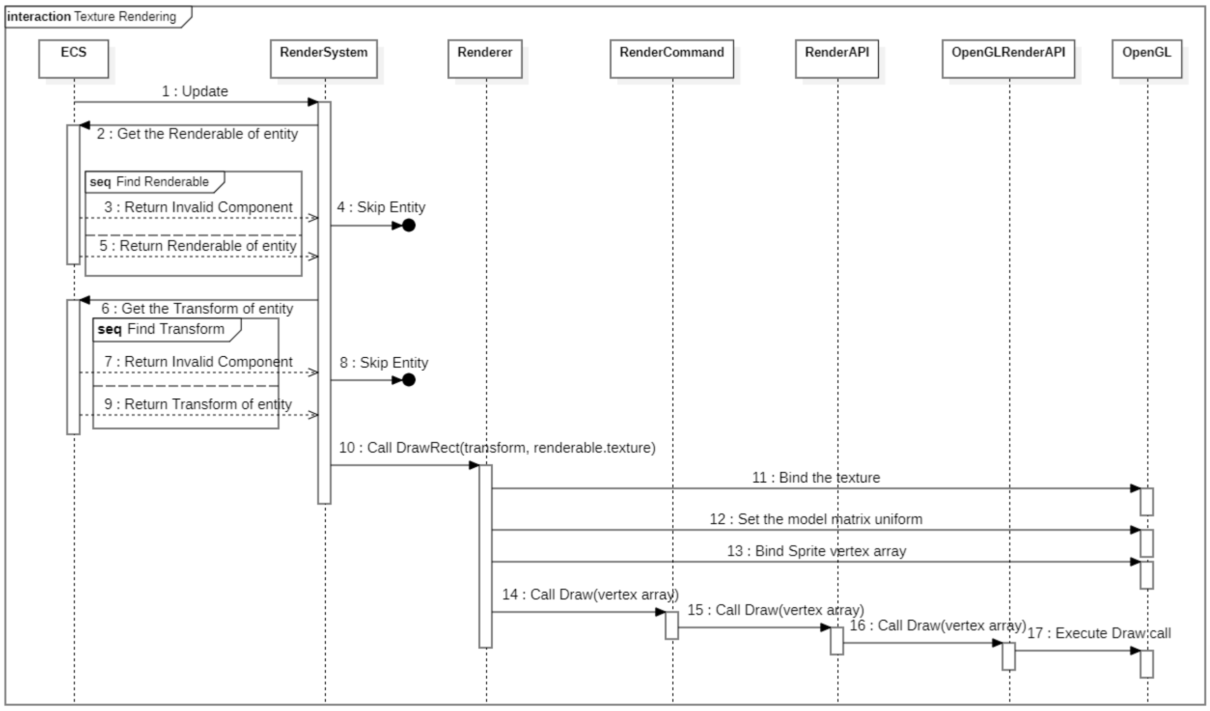

interaction Texture Rendering
ECS
RenderSystem
Renderer
RenderCommand
RenderAPI
OpenGLRenderAPI
OpenGL
1 : Update
2 : Get the Renderable of entity
seq Find Renderable
3 : Return Invalid Component
4 : Skip Entity
5 : Return Renderable of entity
6 : Get the Transform of entity
seq Find Transform
7 : Return Invalid Component
8 : Skip Entity
9 : Return Transform of entity
10 : Call DrawRect(transform, renderable.texture)
11 : Bind the texture
12 : Set the model matrix uniform
13 : Bind Sprite vertex array
14 : Call Draw(vertex array)
15 : Call Draw(vertex array)
16 : Call Draw(vertex array)
17 : Execute Draw call

Figure 3.5: Sequence diagram of the rendering process for an entity.

The typical process of rendering a texture is illustrated in Figure(3.5). The *RenderSystem* first tries to acquire the *Transform* and *Renderable* components of an entity. We will detail these in the next section, but to put it briefly, the *Transform* encapsulates the transformation of an entity and the *Renderable* its texture. If it is not successful in acquiring either of them then it moves on to the next entity. Otherwise It makes a call to the static *Renderer* class, passing the data obtained from the components. The *Renderer* binds the texture of the *Renderable* to an arbitrary texture slot, sets the uniforms required by the shader and binds the vertex array to OpenGL. It then calls the *Draw* command which determines the used render API, in our case *OpenGLRenderAPI*, to issue the draw call.

3.5 Game Object System

We use the ECS pattern to implement the game object system. The class diagram of the ECS is depicted in Figure(3.6). An entity is a primitve type, namely *uint16_t*, which is equivalent to its unique identifier. Entities are contained by an *EntityContainer* which is resposible of their lifetime. It keeps track of the active entities, provides accessing and iteration mechanisms and holds a set of the entities that were destroyed. There can exist a finite number of entities at any time. The set keeps track of the destroyed entities in order to be able to place them back in the world. Only if this set is empty will the container generate new entity identifiers.

We chose to implement the underlying container, both for entities as for components, as an array allocated at program startup. This is because resizing an array can be a costly operation. The drawback of this approach is, of course, the fact that memory which might not be used gets allocated and stays fixed. It is a trade-off between memory and execution speed, but in the case of interactive systems, we consider speed to be the more valuable of the two.

A component is represented as a a struct which defines data and provides construction mechanisms. The base *Component* class contains the component's identifier and its owner entity, 0 being used as an invalid component id. Components are required to derive from this base class and register themselves with the ECS. A component identifier is created when a new component type is registered with the application.

We currently provide the following components: *Transform*, *Renderable*, *Collidable*, *Tag* and *ScriptComponent*. The *Transform* defines the position of the object in the world, its scale and rotation relative to the local coordinates. A *Renderable* holds a *Texture* and the path to it relative to the resource directory of the project. The *Tag* acts as the name of the entity and is used to display it in the GUI. It is also used to obtain a reference to an entity in script, given the name of its *Tag*. The *ScriptComponent* encapsulates the

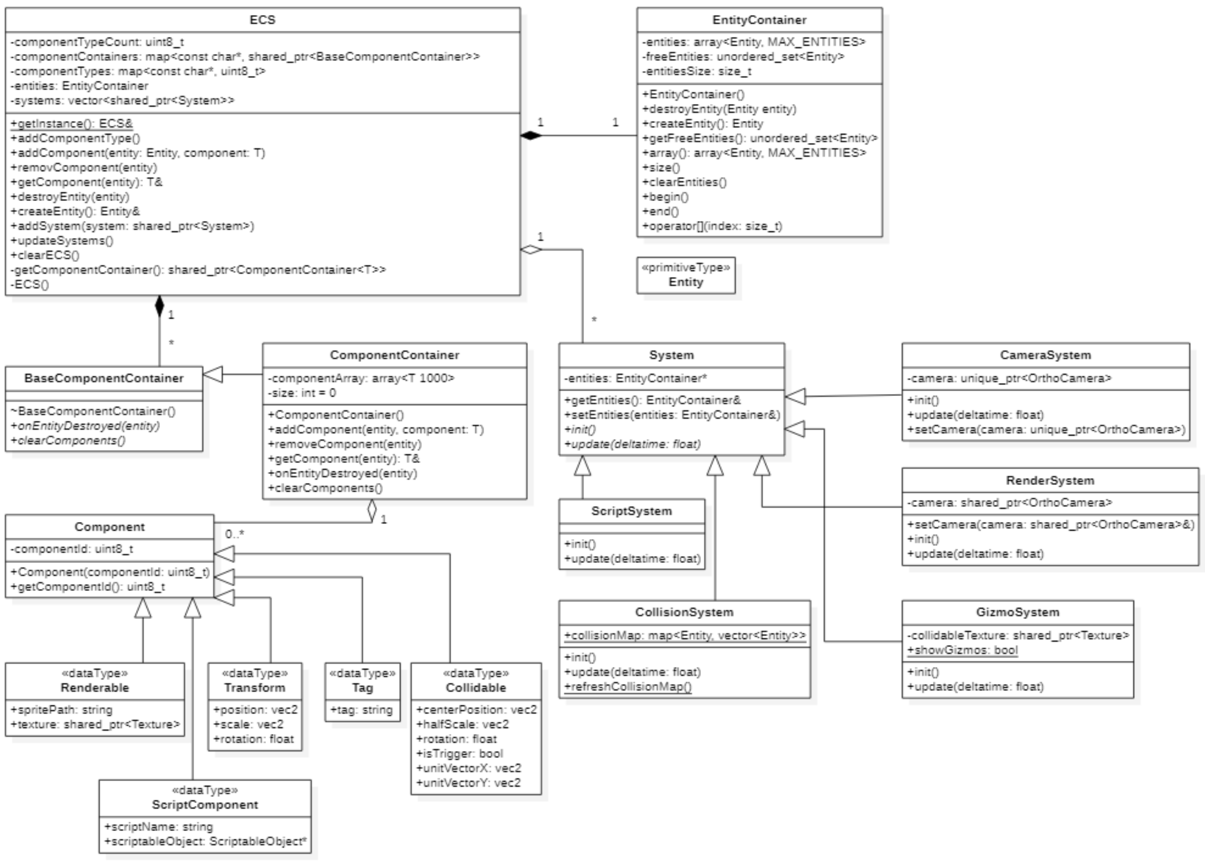

ECS
-componentTypeCount: uint8_t
-componentContainers: map<const char*, shared_ptr<BaseComponentContainer>>
-componentTypes: map<const char*, uint8_t>
-entities: EntityContainer
-systems: vector<shared_ptr<System>>
+getInstance(): ECS&
+addComponentType()
+addComponent(entity: Entity, component: T)
+removComponent(entity)
+getComponent(entity): T&
+destroyEntity(entity)
+createEntity(): Entity&
+addSystem(system: shared_ptr<System>)
+updateSystems()
+clearECS()
-getComponentContainer(): shared_ptr<ComponentContainer<T>>
-ECS()
EntityContainer
-entities: array<Entity, MAX_ENTITIES>
-freeEntities: unordered_set<Entity>
-entitiesSize: size_t
+EntityContainer()
+destroyEntity(Entity entity)
+createEntity(): Entity
+getFreeEntities(): unordered_set<Entity>
+array(): array<Entity, MAX_ENTITIES>
+size()
+clearEntities()
+begin()
+end()
+operator[](index: size_t)
«primitiveType»
Entity
1
1
1
1
*
*
BaseComponentContainer
~BaseComponentContainer()
+onEntityDestroyed(entity)
+clearComponents()
ComponentContainer
-componentArray: array<T 1000>
-size: int = 0
+ComponentContainer()
+addComponent(entity, component: T)
+removeComponent(entity)
+getComponent(entity): T&
+onEntityDestroyed(entity)
+clearComponents()
1
0..*
System
-entities: EntityContainer*
+getEntities(): EntityContainer&
+setEntities(entities: EntityContainer&)
+init()
+update(deltatime: float)
CameraSystem
-camera: unique_ptr<OrthoCamera>
+init()
+update(deltatime: float)
+setCamera(camera: unique_ptr<OrthoCamera>)
RenderSystem
-camera: shared_ptr<OrthoCamera>
+setCamera(camera: shared_ptr<OrthoCamera>&)
+init()
+update(deltatime: float)
ScriptSystem
+init()
+update(deltatime: float)
CollisionSystem
+collisionMap: map<Entity, vector<Entity>>
+init()
+update(deltatime: float)
+refreshCollisionMap()
GizmoSystem
-collidableTexture: shared_ptr<Texture>
+showGizmos: bool
+init()
+update(deltatime: float)
Component
-componentId: uint8_t
+Component(componentId: uint8_t)
+getComponentId(): uint8_t
«dataType»
Renderable
+spritePath: string
+texture: shared_ptr<Texture>
«dataType»
Transform
+position: vec2
+scale: vec2
+rotation: float
«dataType»
Tag
+tag: string
«dataType»
Collidable
+centerPosition: vec2
+halfScale: vec2
+rotation: float
+isTrigger: bool
+unitVectorX: vec2
+unitVectorY: vec2
«dataType»
ScriptComponent
+scriptName: string
+scriptableObject: ScriptableObject*

Figure 3.6: Class diagram of the ECS package

name of the AS script and the *ScriptableObject* which points to the actual script object used to control the entity. We will go into more detail regarding the scripting system in Section(3.6). Finally the *Collidable* holds the representation of an OBB, caching the necessary fields required for collision detection which we detailed in Section(2.4) and recall them here. They are the position of the center of the box, the half-scale of each dimension, it's rotation and the unit vectors on each axis. In addition to this the *isTrigger* flag denotes that objects should not be set back to their previous positions when a collision occurs. For instance the designer might want to implement a trap that lays on the ground and activates when it collides with the player, but the the position of the trap should not change in response to the collision.

Each component type is aggregated in a templated *ComponentContainer* which has an underlying array. The operations of the containers are performed in a way such that the array is maintained packed at all times, that is to say that the owned components in the array are contiguous. This is done in order to maximize the cache hit ratio.

Systems operate on the components that match their aspect as illustrated in Figure(2.2). Each system is self-sufficient thus it can be easily swapped for a different implementation. Systems must implement the *System* interface. This specifies the functions that must be implement, namely an initialization *init()* function to prepare the system for execution and an *update(float)* function to apply transformations to components. The interface maintains a pointer to the *EntityContainer* in order to have access to the entities.

The *CameraSystem* is in charge of moving the camera position, which in turn will affect its view matrix. The camera can be bound to one of the entities, for instance the player, in order to follow its movements in the game world. The *ScriptSystem* is in charge of calling the *update* function implemented by every script. The *RenderSystem* was detailed in the previous section

Collision detection is performed by the *CollisionSystem*. The system holds a map where each entry maps an Entity to the collection of entities it is colliding with at any time. To check whether two entities are colliding it suffices to retrieve the entry of one of the entities in case and query for the other one in the related collection. This operation can theoretically have a worst-case outcome of *O(n)*, where *n* is the number of entities, though the likelihood of a large amount of entities to be colliding in the same frame is unlikely. To detect whether or not two entities are colliding the system first applies the bounding spheres test, using as a radius the object's

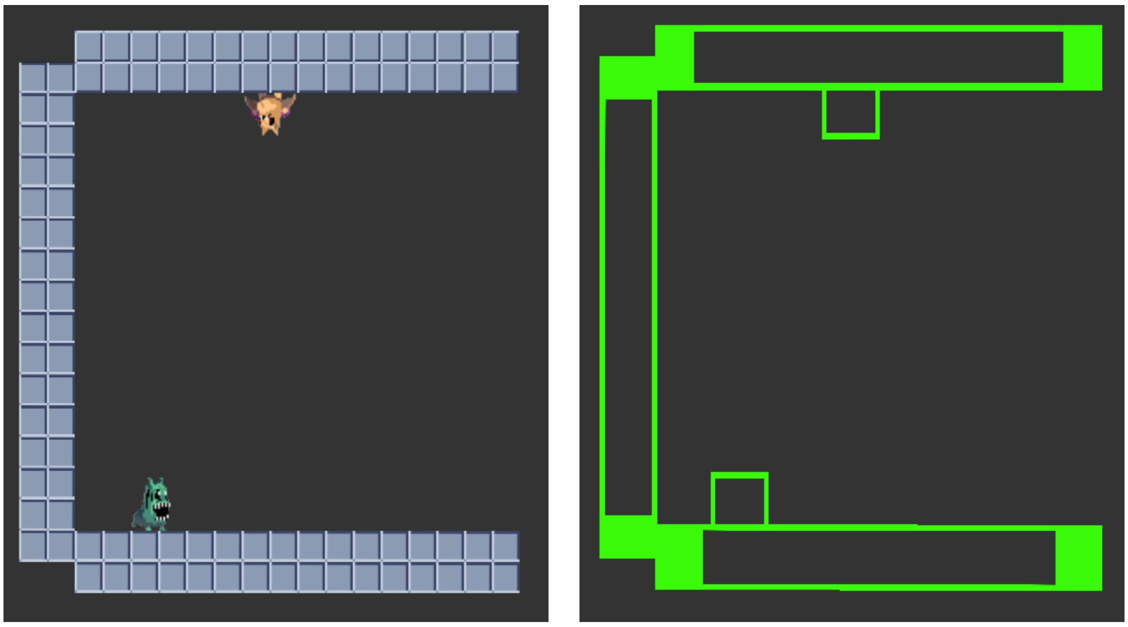
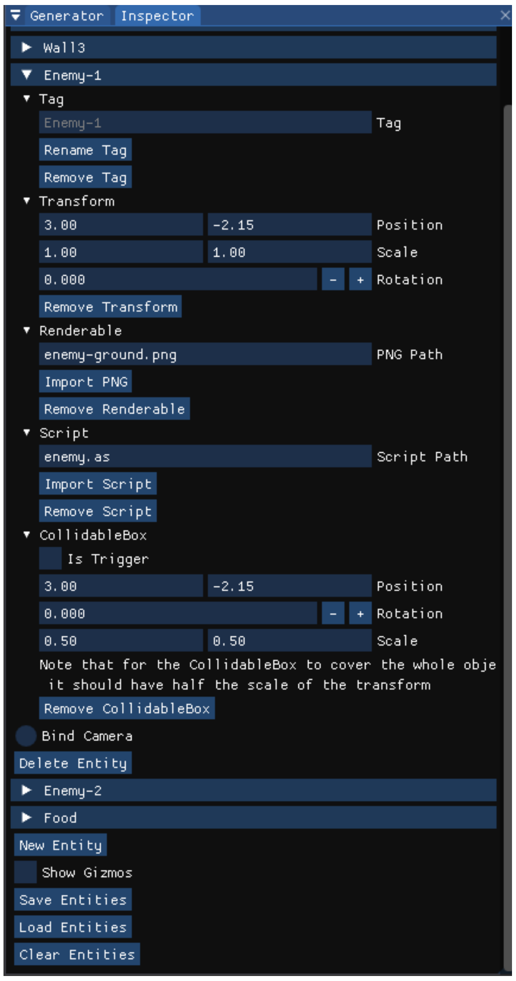

(a) Standard game world view (b) Collision world view

Figure 3.7: The Collision world

largest dimension. If the test succeeds then no further calculations are required since we know that the objects are not colliding. Otherwise it performs the Separating Axis tests to decide if the objects are colliding and finally updates the map accordingly. A gizmo system is implemented separately in order to visualize the collision world.

We have also implemented a space partitioning structure following the Tile Array principle. We have tested this in a relatively small scale game (the details of which we present in Chapter(4)), but it did not seem to positively affect runtime speed so we decided to remove it. It might prove to be useful for games of larger scale so we hold on to it as an idea for future improvement.



Enemy-1 Tag
3.00 -2.15 Position
1.00 1.00 Scale
0.000 - + Rotation
enemy-ground.png PNG Path
enemy.as Script Path
Is Trigger
3.00 -2.15 Position
0.000 - + Rotation
0.50 0.50 Scale
Note that for the CollidableBox to cover the whole obje
it should have half the scale of the transform

Figure 3.8: ECS GUI

The *ECS* class aggregates all the *ComponentContainers*, *Systems* and *Entities* and acts as the interface to the underlying component data. A GUI is exposed to the user to visualize existing components. It allows the user to add, remove, or modify the components of every entity. An exception to this is the *OrthoCamera*, the existence of which is necessary since it is the single camera used to display the game space. That being said the user can manipulate its properties through the GUI.

Changes to components are persisted to a JSON file on the user machine. This is achieved through json_nlohmann (*https://github.com/nlohmann/json*) library which provides serialization functions to C++ data-types and STL containers. Functions are defined for each Component subclass to convert them to and from JSON.

Dear ImGUI *(https://github.com/ocornut/imgui)* was used to implement the visualization tools provided by the engine. It is a GUI library specifically designed for game engines and follows the Immediate Mode GUI paradigm, as opposed to the traditional Retained Mode paradigm. A comparison of the two paradigms is detailed in [11].

In Retained Mode the GUI elements are responsible for the management of their state. It functions based on event callbacks and all the elements are centralized. The main advantage of this approach is that components only need to be re-drawn when their state changes. On the other hand a large amount of callbacks has the potential to slow the system down and.

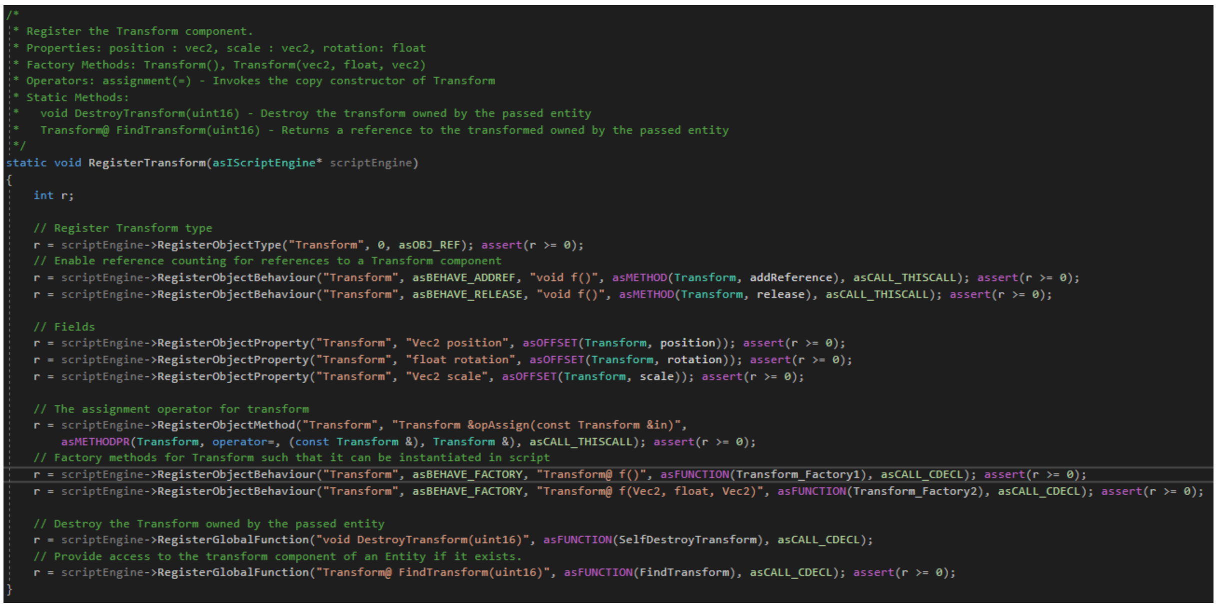
In Immediate Mode the GUI widget's internal state keeps track of whether or not the widget is being interacted with. If it is then it executes the functionality it is assigned, otherwise the code responsible for its behaviour is simply skipped. Setting up Immediate Mode GUI components is achieved merely by invoking a draw function which makes setting them up an easy process. Such a GUI is drawn every frame, which makes this approach a good fit for a real-time interactive system, but not so much for more static systems.

3.6 Scripting

Scripting is achieved using the AngelScript library, which is an open-source cross-platform library available at *https://www.angelcode.com/angelscript*. It provides an environment to run scripts and an interpreted scripting language based on C++ syntax.

We mentioned previously that a *ScriptableObject* is contained in a *ScriptComponent*. This *ScriptableObject* encapsulates the name of the script file and a pointer to the actual script object provided by AS through the *asIScriptObject* interface. In addition it caches the identifier of the entity that owns it and holds a reference counter to determine whether or not it is referenced by any other object. The entity is cached in order to have access to its components in script. The reference count is kept in order to determine whether or not it is safe to destroy a *ScriptableObject*, since it might be referenced in other scripts. The garbage collector provided by the scripting environment is run incrementally at every update by the engine to resolve references that become invalid. We also provide the *garbageCollect* method through the *ScriptManager* to control its execution if it is determined that an application requires it to run more often.

The *asIScriptEngine* object is used to register the application interface available to the scripts and also provides the necessary functionality to create script execution contexts. The context of a script is built when a new script object is constructed and it stores that script's call stack. All contexts are aggregated into a pool. To execute a function defined in script the context of the script is retrieved from the pool and passed the handle of the function to be called. At the end the context is returned to the pool



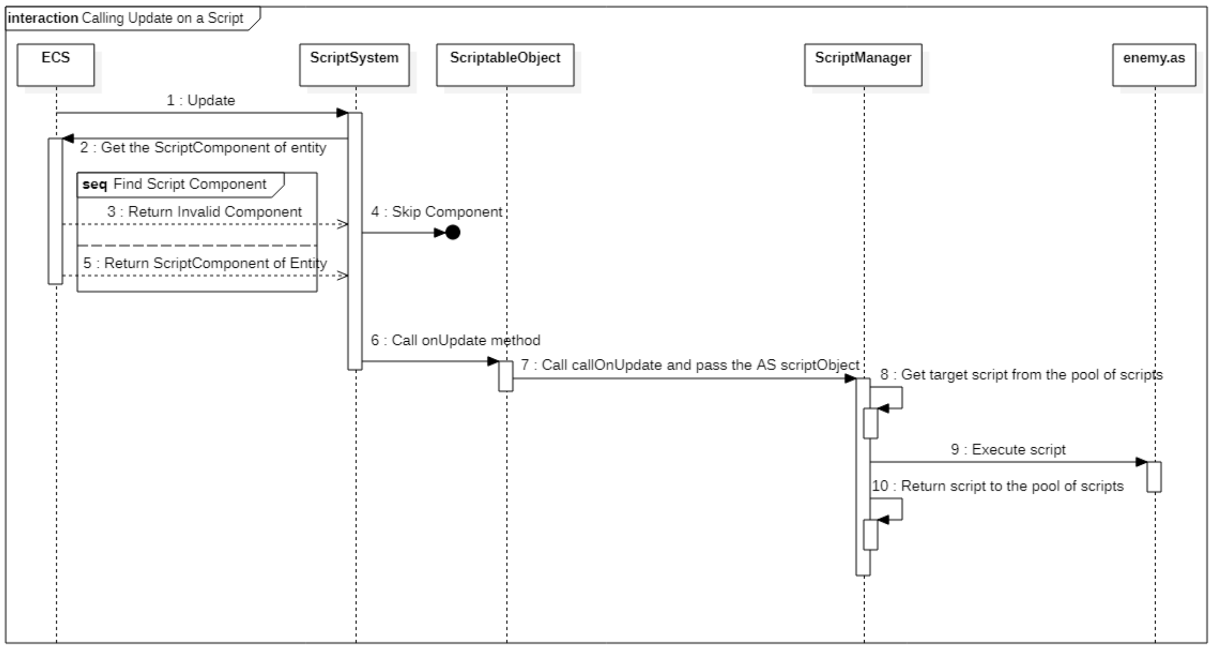
```
/*
* Register the Transform component.
* Properties: position : vec2, scale : vec2, rotation: float
* Factory Methods: Transform(), Transform(vec2, float, vec2)
* Operators: assignment(=) - Invokes the copy constructor of Transform
* Static Methods:
*   void DestroyTransform(uint16) - Destroy the transform owned by the passed entity
*   Transform@ FindTransform(uint16) - Returns a reference to the transformed owned by the passed entity
*/
static void RegisterTransform(asIScriptEngine* scriptEngine)
{
    int r;

    // Register Transform type
    r = scriptEngine->RegisterObjectType("Transform", 0, asOBJ_REF); assert(r >= 0);
    // Enable reference counting for references to a Transform component
    r = scriptEngine->RegisterObjectBehaviour("Transform", asBEHAVE_ADDREF, "void f()", asMETHOD(Transform, addReference), asCALL_THISCALL); assert(r >= 0);
    r = scriptEngine->RegisterObjectBehaviour("Transform", asBEHAVE_RELEASE, "void f()", asMETHOD(Transform, release), asCALL_THISCALL); assert(r >= 0);

    // Fields
    r = scriptEngine->RegisterObjectProperty("Transform", "Vec2 position", asOFFSET(Transform, position)); assert(r >= 0);
    r = scriptEngine->RegisterObjectProperty("Transform", "float rotation", asOFFSET(Transform, rotation)); assert(r >= 0);
    r = scriptEngine->RegisterObjectProperty("Transform", "Vec2 scale", asOFFSET(Transform, scale)); assert(r >= 0);

    // The assignment operator for transform
    r = scriptEngine->RegisterObjectMethod("Transform", "Transform &opAssign(const Transform &in)",
        asMETHODPR(Transform, operator=, (const Transform &), Transform &), asCALL_THISCALL); assert(r >= 0);
    // Factory methods for Transform such that it can be instantiated in script
    r = scriptEngine->RegisterObjectBehaviour("Transform", asBEHAVE_FACTORY, "Transform@ f()", asFUNCTION(Transform_Factory1), asCALL_CDECL); assert(r >= 0);
    r = scriptEngine->RegisterObjectBehaviour("Transform", asBEHAVE_FACTORY, "Transform@ f(Vec2, float, Vec2)", asFUNCTION(Transform_Factory2), asCALL_CDECL); assert(r >= 0);

    // Destroy the Transform owned by the passed entity
    r = scriptEngine->RegisterGlobalFunction("void DestroyTransform(uint16)", asFUNCTION(SelfDestroyTransform), asCALL_CDECL);
    // Provide access to the transform component of an Entity if it exists.
    r = scriptEngine->RegisterGlobalFunction("Transform@ FindTransform(uint16)", asFUNCTION(FindTransform), asCALL_CDECL); assert(r >= 0);
}
```

Figure 3.9: Source code: Registering the Transform component with AngelScript.

AS allows registration of global functions, global properties, object types and class interfaces. Figure(3.9) depicts the registration process for the *Transform* component. The type is registered via *RegisterObjectType*, specifying that it is a reference type. Reference types are managed by the application, while value types registered with *asOBJ_VALUE* are managed by the scripting environment. *RegisterObjectProperty* registers members of the class. These require a specification of the displacement at which the property resides, relative to the class. AS provides the *asOFFSET* macro to specify this. *RegisterObjectBehavior* is used to register memory management and operator functions. It is used here to define the factory functions of the *Transform* component, as denoted by the *asBEHAVE_FACTORY* macro. The *asFUNCTION* macro is used to specify the name of the function to be called. The functions *addReference* and *release* respectively are used to increase and decrease the reference counter of the object and must be implemented for every reference type. *RegisterObjectMethod* is used to register the methods available to a type. In the case of *Transform* it only defines the assignment operator. Here *asMETHODPR* is used to pass a pointer to the overloaded assignment operator of *Transform* and requires the specification of the function header in order to be able to pinpoint the exact overload. Finally, *RegisterGlobalFunction*, as its name suggests, is used to register static functions in the global namespace of AS. *FindTransform* is used to retrieve the *Transform* component of an entity, while *DestroyTransform* is used to remove it.

Naturally, in order for the script engine to support new types, specific to one application, they must be registered with it in a similar way.

The *ScriptSystem* performs the calls to the update function of each component, invoking the *ScriptManager* to perform the actual call to the script function. This process is illustrated in Figure(3.10). The *ScriptableObject* invokes the *ScriptManager* which obtains the context of the script from the pool of contexts, executes the script and finally returns the context to the pool. As mentioned, a script object holds a reference count of the other objects that depend on it. Whenever the count reaches 0 the script object is destroyed. Scripts are stored as external assets and run in a protected environment. As such they can be easily swapped through the GUI or modified at runtime.



interaction Calling Update on a Script
ECS
ScriptSystem
ScriptableObject
ScriptManager
enemy.as
1 : Update
2 : Get the ScriptComponent of entity
seq Find Script Component
3 : Return Invalid Component
4 : Skip Component
5 : Return ScriptComponent of Entity
6 : Call onUpdate method
7 : Call callOnUpdate and pass the AS scriptObject
8 : Get target script from the pool of scripts
9 : Execute script
10 : Return script to the pool of scripts

Figure 3.10: Sequence diagram depicting the message flow of calling the update function of a script.

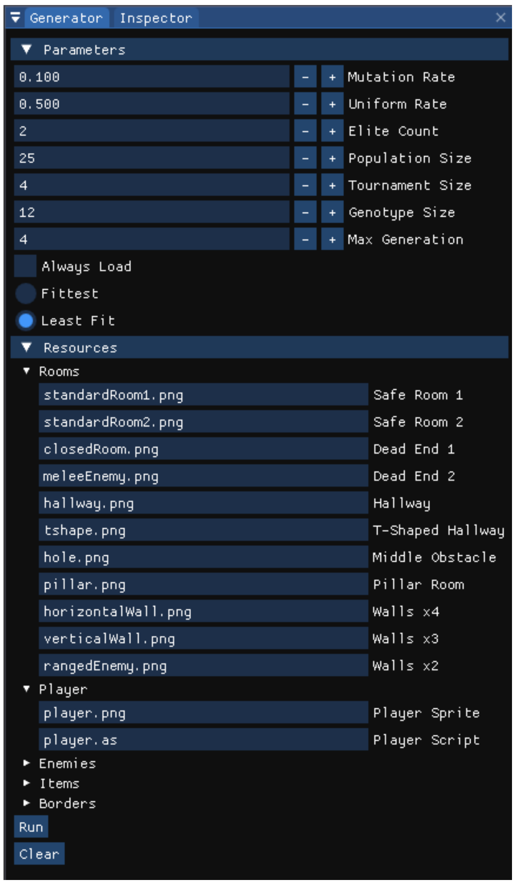
3.7 Procedural Content Generation

As mentioned in Section(2.3), we implement a generation tool using FI-2Pop. Design Elements, which represent a gene, should be as lightweight as possible since they will be copied often in the course of generation through crossover operations. That being said a DE is represented as an entity owning a *Transform* component that specifies its position and an *enum* that represents 1 of the 11 *ElementTypes*.

Individuals aggregate DEs according to the size of the genotype. In turn, individuals are aggregated into populations which provide access to the most fit (or most diverse in the case of feasible individuals) and least fit individuals, as well as accessors to arbitrary individuals and functionality to add or remove them from the populations.

The algorithm runner encapsulates two populations, the feasible and infeasible respectively, keeps track of the current generation, the most fit and least fit individuals of it and provides the necessary methods for evolution. We implement tournament selection, uniform crossover, single-point crossover and mutations that can affect the *ElementType* or rotation of a room. These parameters, as well as the other components of the rooms and the entities they contain can be customized by the user through the GUI presented in Figure(3.11).

As mentioned, DEs are required to hold a *Transform*. This is because it specifies the position and rotation of the object which are necessary to determine the paths that the room creates. We remind the reader that the constraint which must be met by the individuals is the fact that the player should be able to reach any other room from the room it is currently positioned in.



```
Generator  Inspector
▼ Parameters
0.100        - + Mutation Rate
0.500        - + Uniform Rate
2            - + Elite Count
25           - + Population Size
4            - + Tournament Size
12           - + Genotype Size
4            - + Max Generation
☐ Always Load
○ Fittest
● Least Fit
▼ Resources
  ▾ Rooms
    standardRoom1.png     Safe Room 1
    standardRoom2.png     Safe Room 2
    closedRoom.png        Dead End 1
    meleeEnemy.png        Dead End 2
    hallway.png           Hallway
    tshape.png            T-Shaped Hallway
    hole.png              Middle Obstacle
    pillar.png            Pillar Room
    horizontalWall.png    Walls x4
    verticalWall.png      Walls x3
    rangedEnemy.png       Walls x2
  ▾ Player
    player.png            Player Sprite
    player.as             Player Script
  ▸ Enemies
  ▸ Items
  ▸ Borders
Run
Clear
```

Figure 3.11: UI to the PCG parameters. The 'Parameters' section allows tweaking generation algorithm parameters. The 'Resources' section is used to specify the assets and scripts that will be added to the generated entities.

We modelled this by relating a directed graph to each Individual such that every room in the genotype represents a node and every room type is annotated with the edges that are created when the room is placed in the world. As such, in order to satisfy the above constraints, the rooms must be generated in such a way that the graph they create has only 1 strongly connected component. The fitness function is modelled as in Figure(3.13), where *NV* is the number of vertices of the graph, *CC* is the number of strongly connected components and *p(x)* is the fitness of individual *x*. Essentially an infeasible individual is penalized for every component that exceeds one. Feasible individuals perform novelty search, aiming to raise diversity inside the population.

In calculating diversity a level is rewarded for by 1 measure if it has a different rotation compared to the other individuals and by 2 measures for using a different DE. Both rotation and element type can have a significant impact on how the level is traversed but we consider that seeing a different type of element produces a more pronounced sense of novelty in the player, as opposed to encountering the same element at a different angle.

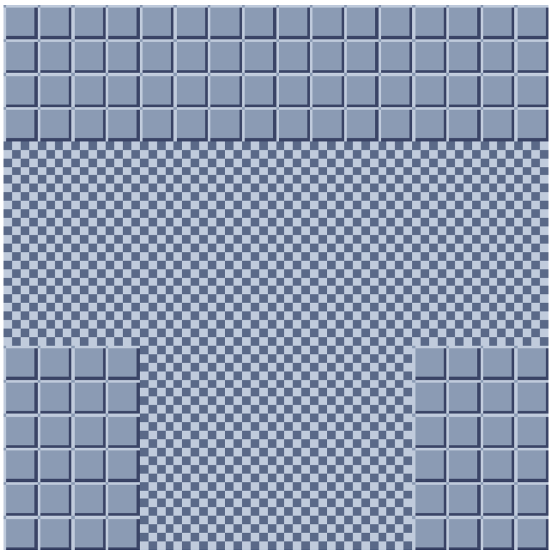


Figure 3.12: Design Element. Creates edges to the node below and the nodes to its left and right

$$p(x) = \frac{NV - CC + 1}{NV}$$

Figure 3.13: Objective function that evaluates fitness based on the strongly connected components of a graph

The *LevelGenerator* acts as an interface to the generation algorithm and is responsible for creating the components represented by the DEs in an individual. *Renderables* are added to all the spawned entities, *Transforms* and *Collidables* are created to enable collision with the walls of every room and AS scripts are attached to the enemy entities. These properties can be modified through the GUI, but special care must be taken when changing the *Renderable* of a room. *Collidables* of a room are added as separate entities, since one entity can only map to one *Collidable*. If the layout of a room is changed then a software engineer is required to adapt the properties of *Collidables* to properly match the walls. Designers should only change the room representation and collaborate with a software engineer if a change of layout is required.

Finally, the user has the options to always load the most or least fit individual of a generation. This can come in handy if the user desires to visualize the two extremes of the population. If this option is not enabled then a random individual will be spawned. The state of the populations is maintained at the end of generation, but the user has the option to reset it to the initial state if he so wishes.

Chapter 4

Experiment

We present here the process of creating an application on top of our engine and detail the results we obtained by experimenting with the content generation algorithm.

We implemented a simple game where the player is required to collect all the collectibles in a level in order to advance to the next level. The levels are generated via FI2Pop and enemies are spawned in the rooms that are not considered Safe Zones. In Roguelike fashion, the player loses all progress on colliding with an enemy.

The application project is required to link with the engine and push at least one layer onto the layer stack. We have created a *GameLayer* derived from *SortingLayer* who's responsibility is to interface with the *LevelGenerator* and generate a new level once the current one is beaten. The algorithm runs for 4 generations between each level. The layer declares a a static integer to keep track of the player's progress in each

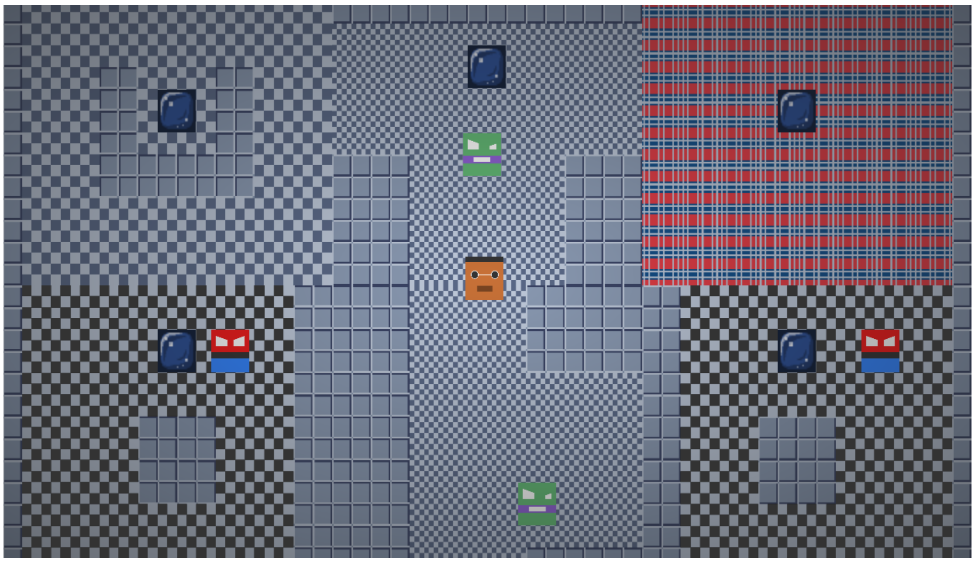


Figure 4.1: The resulting game. Building for Release strips the UI provided by ImGui

level and registers it with AS as a global property. *Renderables* are provided for all the rooms and enemies in the scene, as well as for the player. Scripts are implemented to define player, enemy and collectible behavior and require a *Transform* component in order to have access to their position in-script. Enemy scripts define their movement pattern and send a message to the player script on collision. The player script modifies its *Transform* according to player input and receives messages on colliding with enemies. Finally the collectible scripts merely react to collision with the player. They destroy themselves on contact and increment the score counter.

As an alternative, the generation algorithm could also be used to generate a set of levels in the development phases of the game. This could enable designers to edit the resulting levels and export them to create a customized collection of levels.

We have obtained promising results for a tournament size equal to 4 with a total population of 25, crossover rate of 0.3 and mutation rate of 0.1, employing elitism to keep the two most promising individuals of a generation unchanged. The results for our experiments are displayed in Table(4.1), with feasible individuals being discovered in the first generation in every setup.

The early discovery of feasible individuals could be a consequence of the way the rooms in this setup were designed, as nearly half the rooms generate paths to all their neighbours, making it relatively easy to achieve feasibility. We also noted that changing the population size did not affect this fact.

Changing the tournament size did not seem to have an effect on the results either. Eventually we decided to keep it at 4, assuming it would apply less selection pressure and thus give less fit individuals better chances to reproduce. Another reason was efficiency, since even though the larger tournament did not affect results it did negatively affect the time needed to run the algorithm. We also noted that doubling the mutation

Tour. Size	8	4	4	4
Crossover Rate	0.50	0.50	0.40	0.50
Mutation Rate	0.10	0.10	0.10	0.05
Avg. Diversity	0.69	0.69	0.68	0.61
Avg. Fitness	0.70	0.70	0.69	0.67
Final Gen.	6	6	24	17

Table 4.1: Experimental results for level generation using FI-2Pop

rate from 0.1 to 0.2 did not significantly affect the results. Changing the room types so as to create tighter constraints might provide more interesting results in this respect as well.

On the other hand halving the mutation rate to 0.05 proved to significantly impact the time needed to assemble a complete feasible population while also lowering overall fitness. This suggests that individuals get stuck more often on local optima due to the lower mutation rate.

Finally reducing the crossover rate to 0.4 seemed to have the most impact on the ability of the algorithm to find solutions, requiring to run for 24 generations to fill the population and not providing any significant improvement to fitness or diversity.

Chapter 5

Conclusions

The objective we set for ourselves was to build an interactive system performant enough to serve for game development purposes, providing a game object system, scripting facilities and content generation tools.

We compared the inheritance-based and component-based approaches for creating a component system, concluding that the former is not a good fit for such a system due to it requiring constant maintenance of the hierarchy. We then detailed the ECS pattern, following a data-oriented design, and used it in our engine to ensure it can scale to a large number of game objects. We focused on composing independent components and implementing swappable systems to act on the compoentns. We stored these components in packed arrays to maximize the cache hit ratio.

In the context of ECS we implemented a collision system and presented several techniques for collision detection and space partitioning. We noted the use of grids, quadtrees and k-d trees as data structures used to divide the game space such as to decrease the number of collision tests performed per frame. For detecting collision we note the use of AABBs, Bounding Spheres and the Separating Axis Theorem for OBBs. We implement the Bounding Spheres as a preliminary test to the Separating Axis tests.

With regards to scripting, we presented a set of tools used in other game development projects and the AS environment as integrated in our engine. A data-driven approach is taken where the scripts, treated as external assets, control the execution.

To construct the game environment we employed PCG using the FI-2Pop genetic algorithm. A level is represented in the genotype as a collection of rooms, requiring to form a strongly connected graph in order to be considered feasible. The results we obtained by experimenting with the algorithm indicate early discovery of feasible individuals with relatively high fitness and diversity values for only a couple of generations. We conclude by noting that the algorithm can successfully be used to generate levels both in the development phase as well as at game runtime.

We consider that we have achieved our objective, providing a scalable game object system that functions well in conjunction with the content generation algorithm and scripting environment.

Currently the application does not implement any kind of parallel processing. Changing the underlying component container to properly support a multi-threaded environment should prove to have a great impact on the performance of the engine. Furthermore, changing the way the relationship between entities and components is represented could allow for entities to map to multiple components of the same type. A use of this could be mapping multiple *Collidables* to one entity and program responses to collision depending on which *Collidable* is in contact.

The application also lacks a sound system and an animation system. The animation system could be implemented via a finite state machine, while for the sound system there exists a number of libraries for playing sounds and interfacing with audio devices which could be integrated into the engine.

Other improvements to be taken into account include support for multiple platforms and input devices, support for multiple rendering APIs such as DirectX on Windows or Metal on MacOS and extending the gizmo system, as currently it only offers gizmos to represent the collision world. An example would be allowing the user to manipulate game object *Transforms* through the GUI, by providing interactable gizmos.

List of Figures

List of Tables

Bibliography

[1] Eike Falk Anderson. "A classification of scripting systems for entertainment and serious computer games". In: *2011 Third International Conference on Games and Virtual Worlds for Serious Applications*. IEEE. 2011, pp. 47–54.

[2] Eike Falk Anderson et al. "The case for research in game engine architecture". In: *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*. 2008, pp. 228–231.

[3] Mikio Aoyama et al. "New age of software development: How component-based software engineering changes the way of software development". In: *1998 International Workshop on CBSE*. 1998, pp. 1–5.

[4] Arni Arent and Tiago Costa. "Artemis entity system framework". In: *URL http://www.gamadu.com/artemis* (2012).

[5] Jonathan Blow. "Practical collision detection". In: *Proceedings of the Computer Game Developer's Conference*. 1997.

[6] Xia Cai et al. "Component-based software engineering: technologies, development frameworks, and quality assurance schemes". In: *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*. IEEE. 2000, pp. 372–379.

[7] Kate Compton and Michael Mateas. "Procedural Level Design for Platform Games." In: *AIIDE*. 2006, pp. 109–111.

[8] Ivica Crnkovic. "Component-based software engineering for embedded systems". In: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 712–713.

[9] Bruce Dawson. "Game scripting in Python". In: *Proc. GDC*. 2002.

[10] Mike Dickheiser. *Game Programming Gems 6 (Book & CD-ROM)(Game Development Series)*. Charles River Media, Inc., 2006. Chap. 4.

[11] Alex Feldmeier. "GUI Programming". In: *URL: http://people.uwplatt.edu/ yangq/ CSSE411/csse411-materials/f13/feldmeiera-gui%20*$_{p}rogramming.doc$ (2013).

[12] Stuart Golodetz. "Simplifying the C++/AngelScript Binding Process". In: *Overload* 95 (2010).

[13] Mark Hendrikx et al. "Procedural content generation for games: A survey". In: *ACM Transactions on Multimedia Computing, Communications, and Applications* 9.1 (2013), pp. 1–22.

[14] Xavier Ho, Martin Tomitsch, and Tomasz Bednarz. "Finding design influence within roguelike games". In: *International Academic Conference on Meaningful Play*. 2016, pp. 1–27.

[15] Robert Huebner. "Adding languages to game engines". In: *Game Developer* 4.9 (1997).

[16] Johnny Huynh. "Separating axis theorem for oriented bounding boxes". In: *URL: http://www.jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented %20Bounding%20Boxes.pdf* (2009).

[17] Ahmed Khalifa, Fernando de Mesentier Silva, and Julian Togelius. "Level design patterns in 2D games". In: *2019 IEEE Conference on Games (CoG)*. IEEE. 2019, pp. 1–8.

[18] Steven Orla Kimbrough, Ming Lu, and David Harlan Wood. "Exploring the evolutionary details of a feasible-infeasible two-population GA". In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2004, pp. 292–301.

[19] Tomasz Koziara and Nenad Bicanic. "Bounding box collision detection". In: *13th acme conference: university of sheffield*. Citeseer. 2005.

[20] Patrick Lange, Rene Weller, and Gabriel Zachmann. "Wait-free hash maps in the entity-component-system pattern for realtime interactive systems". In: *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE. 2016, pp. 1–8.

[21] Jong Kook Lee et al. "Component identification method with coupling and cohesion". In: *Proceedings Eighth Asia-Pacific Software Engineering Conference*. IEEE. 2001, pp. 79–86.

[22] Joel Lehman and Kenneth O Stanley. “Efficiently evolving programs through the search for novelty”. In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. 2010, pp. 837–844.

[23] Joel Lehman and Kenneth O Stanley. “Revising the evolutionary computation abstraction: minimal criteria novelty search”. In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. 2010, pp. 103–110.

[24] Tom Leonard. “Postmortem: Thief: The Dark Project”. In: *Game Developer Magazine* 3.28 (1999).

[25] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. “Neuroevolutionary constrained optimization for content creation”. In: *2011 IEEE Conference on Computational Intelligence and Games (CIG’11)*. IEEE. 2011, pp. 71–78.

[26] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. “Enhancements to constrained novelty search: Two-population novelty search for generating game content”. In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 2013, pp. 343–350.

[27] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. “Sentient sketchbook: computer-assisted game level authoring”. In: *8th International Conference on the Foundations of Digital Games, Chania* (2013).

[28] Panagiotis K Linos et al. “A tool for understanding multi-language program dependencies”. In: *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE. 2003, pp. 64–72.

[29] Peter Mawhorter and Michael Mateas. “Procedural level generation using occupancy-regulated extension”. In: *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE. 2010, pp. 351–358.

[30] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[31] Ian Parberry, Max B Kazemzadeh, and Timothy Roden. “The art and science of game programming”. In: *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. 2006, pp. 510–514.

[32] Nicolas Porter. “Component-based game object system”. In: *Carleton University, URL: https://raw.githubusercontent.com/surjikal/cbgos-experiment/master/doc/nicolasporter-cbgos-paper.pdf* (2012).

[33] Thibault Raffaillac and Stéphane Huot. “Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model”. In: *Proceedings of the ACM on Human-Computer Interaction* 3.EICS (2019), pp. 1–22.

[34] Chris Remo. “MIGS: Far Cry 2’s Guay on the importance of procedural content”. In: *Gamasutra* (2008).

[35] Matthew Shelley. “Examining Quadtrees, kd Trees, and Tile Arrays”. In: *Carleton University, URL: https://pdfs.semanticscholar.org/422f/63b62aaa6c8209b0dcbe8a53e360 ad90514d.pdf* (2012).

[36] Nathan Sorenson and Philippe Pasquier. “Towards a generic framework for automated video game level creation”. In: *European conference on the applications of evolutionary computation*. Springer. 2010, pp. 131–140.

[37] Subhash Suri, Philip M Hubbard, and John F Hughes. “Analyzing bounding boxes for object intersection”. In: *ACM Transactions on Graphics (TOG)* 18.3 (1999), pp. 257–277.

[38] Walker White et al. “Better scripts, better games”. In: *Communications of the ACM* 52.3 (2009), pp. 42–47.

[39] B Wilcox. “Reflections on Building Three Scripting Languages”. In: *Gamasutra* (2007).

[40] Richard Wright, Benjamin Lipchak, and Nicholas Haemel. *OpenGL SuperBible: comprehensive tutorial and reference*. Pearson Education, 2007.