Dreams Eater
Group 6

Barbiero Alessandro
aleba@itu.dk

Giacometti Giovanni
giog@itu.dk

Lucas Schütt Nielsen
lucn@itu.dk

January 2, 2023

1 Introduction

Dreams Eater is a Roguelike - RPG shooter 2d game. The protagonist is a wraith, whose goal is to explore a dungeon of rooms and fight all the enemies that have come to haunt his dreams. The game is based on randomly generated levels of increasing complexity. The player gets access to the following level when he's able to kill the boss of the level. Rooms may contain powerups, special badges that can be collected by the player to enhance his abilities.
The game leverages on Box2d library to handle physics. Multiple types of keyboard inputs are supported and can be chosen through the starting menu, as well as the starting difficulty.

2 Folder organization

The code of the game is available at: *https://github.itu.dk/aleba/Dreams-Eater*. Folders are organized as follows: Data folder contains the JSON file describing enemies features, Sprites folder includes all files required to create sprite atlases, the Header folder contains all the headers files, the Src folder includes all the source files and rapidjson folder contains rapidjson library files. Sprites folder is in turn divided into many folders, each of them dedicated to a specific entity. The Character folder is among them and contains a folder for each character. Each character has then two folders, one containing sprites facing right and the other the ones facing left. Each of these folder has a subfolder for each animation the character is able to implement. Images name is their progressive number in the animation, starting from 0.
Src and Header folders are organized in the same way: Builders folder contains builder-related files, all components are inside the Components folder and the classes inheriting from IEnemyBehaviour class are inside the EnemyBehaviours folder. Standalone classes are just placed in the folders.

3 Software Architecture

Figure 1 contains the first part of the class diagram. Many attributes and methods are omitted for space reasons. The starting point is the DreamsEater

singleton. It implements all the methods needed to handle the different phases of the game, takes care of the physics setup, handles SDL events and runs the render and the update loops. It also holds a pointer to the current scene, which is the class managing the visible game objects. The available scenes are the game, the starting menu and the pause menu. The pointer is assigned based on the game state, which is contained in a dedicated attribute and changes based on the dynamic of the game. DreamsEater class is also composed by a Level, that is the class representing the current level of the game.
GameObject class is used to define all the objects of the game. A game object is composed of multiple components that define its behaviour. Components don't exist without the related object, while game objects can persist through scenes.

Figure 2 contains the second part of the class diagram. It includes all the builders, which are the classes used to create the relevant entities of our game along with the enumeration and the structs they leverage on. These utilities are designed to be easily expandable and customizable.

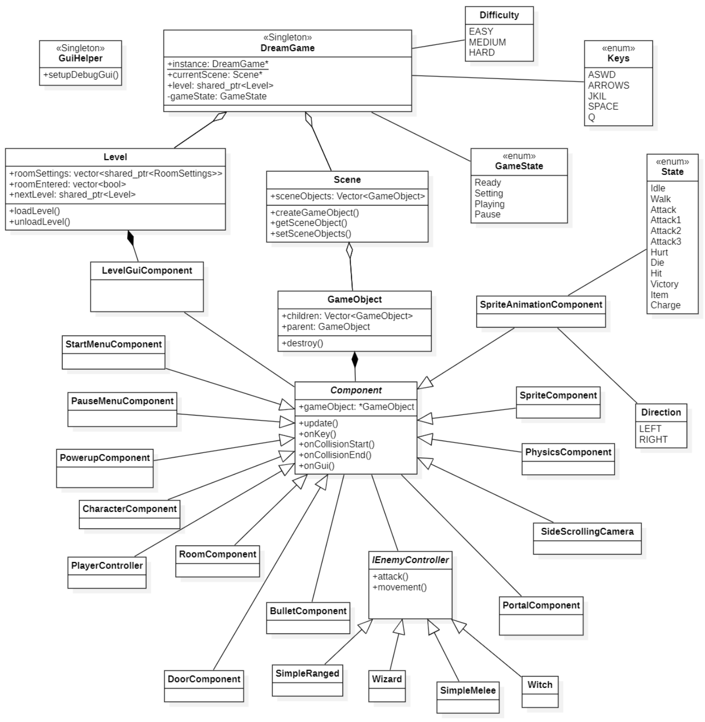

Figure 1: Components Class Diagram .

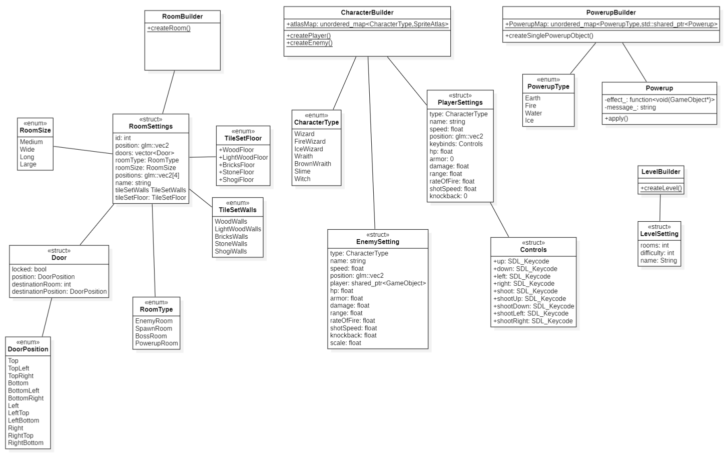

Figure 2: Class Diagram Builders.

4 Implementation

4.1 Level Generation

Each level consists of a number of rooms connected by doors. Only one room is loaded at a time. Levels are randomly generated. We considered two ways to implement it. The way we decided to do it is inspired by breadth-first search in a graph: Create a starting node, generate neighbour nodes, and put them in a FIFO queue, while the queue is not empty pop a node from the queue and generate its data and its neighbours. When a node is put in the queue only some initial data is generated (id, position), the rest is generated when it is popped from the queue.

We allow a room to have 0 doors to new rooms (so we can have dead ends), so in case the queue is empty but we need more rooms, we add a new door to an existing room and continue the algorithm.

The difficult part of this method was to ensure the layout physically makes sense, with no rooms ”overlapping” in the map. We did this by keeping track of each room’s position in a coordinate system. We standardized rooms in certain sizes, with the smallest size being 1 unit in the coordinate system. This allows us to check neighbour coordinates when deciding the size of a room and placing doors to new/existing rooms. We use a Map from a pair of integers to a shared pointer to the corresponding room.

The other way we considered was to first generate all the rooms we need and then connect them afterwards. This would simplify the room generation part (no need to check if there is space for a Large room for example). The complexity is then in deciding how to place doors between the rooms at the end to create the layout. Creating a chain of rooms would be a simple solution, but is not suited for the kind of gameplay we want. We decided not to do it this way since the first way naturally gives us the kind of layout we want.

We decided to generate the contents of each room only when it is entered. When unloading a room we decided to keep objects in memory that were randomly generated (enemies, powerups), and delete static objects that can be derived from the room's size and other data (walls, floor, doors). The room keeps a separate list of shared pointers to objects that should be saved, while the rest of the objects are set as children of the room's root GameObject. The list of pointers is copied before destroying the room. All of the objects are removed from the current scene, which means they won't be updated in the main game loop. When loading the room again the objects are put back into the scene. The main problem with this method is that objects with a physicsComponent also need to be removed from box2D's physics loop. We handle this by saving the object's physic's body, fixture, and shape data, and using this to recreate them later. The advantage of this way of saving the room is that we don't need to destroy and create the GameObject and all of its components every time it's loaded. The disadvantage is that we use more memory by saving the whole object in memory.

Another approach is to save the data needed to recreate the objects (position, scale, sprite, hp, current state, etc.). The advantage to this approach is using less memory by only saving the minimum necessary information, instead of the entire object. The disadvantage is that creating and destroying objects can be expensive operations, possibly increasing load times. We decided that keeping objects in memory was not too big a tradeoff for our purposes.

4.2 Animations

Animations are created thanks to the interaction between two components: *SpriteComponent* and *SpriteAnimationComponent*. The Sprite component holds the current sprite to display in the frame and every visible object on the scene has this component. The Animation component offers the possibility to change the image inside the sprite component following certain vectors of Sprites. The animation component can be used in 2 main modalities:

- Swap between pre-saved animations indexed by the relative State (Used for all the characters)
- Reproduction of single animation, possibly loop animation (Used for the animation of the special attacks)

In every game loop cycle, a vector of sprites and a moving index based on the time passed are used to detect which Sprite will be displayed next. To understand which vector to choose the Sprites from, the component establishes three levels of priority (Figure 3):

- **Default**: If there is no animation asked to be reproduced, search for the current state inside the character component and display the relative animation (Idle, Walk...)
- **Complete Animation**: If an animation has to be reproduced entirely in order to make sense and it should happen only in particular conditions

it can be displayed by calling one of the several methods to display a complete animation. Useful parameters to pass in this case are the total duration of the animation, used to compute the time to give at each frame knowing the length of the animation, and a possible callback function to be called at the end of the animation (e.g. At the end of the attack animation spawn the bullet).

- **Urgent**: On top of that an animation can be urgent. It means that it can override a previous complete animation that is being shown at the moment. This is fundamental for example for the Die animation that cannot wait for the attack animation to be finished.

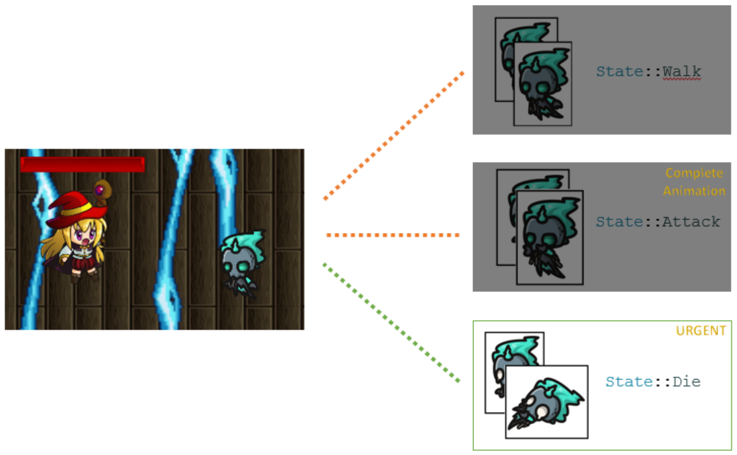

Figure 3: Priority of animations.

If used in the first modality, the animation component works by storing two different hash maps to link State and animation vector. Each of them is used to display a different direction (LEFT, RIGHT). It is for this reason important to keep the animation component always updated about the direction the player is facing. This separation for the animations has been used to keep the most general interface possible and to solve visualization issues that happened with animations made of images of different widths.

The vectors for the animations are created eagerly at the instantiation of the character object and passed to the animation component inside the CharacterBuilder functions. The *CharacterBuilder* class stores a static version of the SpriteAtlases used to create the animation vectors, these atlases are retrieved eagerly at the start of the game, after the first scene change.

4.3 Enemy Behaviours

To control the characters our game makes use of two components: *PlayerController* and *EnemyController*.

In order to implement a game that could be easily expanded with various types of enemies we created a standard component interface *IEnemyController* from which all the enemy behaviours should inherit. In this way, it is easy to create a new final boss or a new type of enemy just by implementing his own versions of *attack()*, *move()* and *setBulletSprites()*.

4.4 Powerup System

The expandability of the game was always in our minds. To create the most ductile powerup system possible, the *Powerup* class stores a callback that can be applied over the game object of the player that activates it. In this way, it is possible to change statistics as well as aesthetic features. Following the same paradigm as the rest of the game, a builder has been used to create the collectable powerups, and each effect is associated with a specific enumeration value (*PowerupType*) inside a static hashmap.

4.5 GUI

The graphical user interface is built upon Dear ImGui library. It follows the Immediate Mode paradigm, according to which no game-related state is held in GUI components. The *Component* class contains the *onGui* method, which is implemented by all its children that aim to display something. The actual rendering occurs during the render pass by iterating on all components and calling *onGui* on each of them.
Gui has a twofold purpose: on one hand, we used it to debug mechanics and algorithms by tweaking values during the game and visualizing how changes would affect the game behaviour, on the other side it was employed to offer a gracious and appealing game experience.
The debug window is set up by a method implemented by the GuiHelper singleton. All components can append content by beginning a window with the same name. In-game GUI features include a start and a pause menu and various aiding windows, such as a minimap that shows the room where the player is located along with the rooms already visited.

5 Performance Measurements

A bottleneck of the system is characterized by the loading of resources, that causes a slow down when the game is started. We tried to solve that by implementing lazy loading. This would speed up the start of the game but also increase the time to load and prepare each room, resulting in a coarse and unpleasing game experience. We eventually decided to stick with eager loading and display a loading page after the play button is clicked. That is done by pushing a custom SDL event to start the game while also finishing the render of the interface.

6 Discussion

The game satisfies our expectations and provides a nice and smooth gameplay. Many are the possible improvements: add various obstacles in rooms (e.g., chairs, tables, rocks, etc.), possibly read predefined sets of obstacles(e.g., table with 2 chairs, couch and tv, etc.) and enemy formations (e.g., 2 wizards and 1 slime, 3 different wizards, etc.) from json files. These can be used for more balanced/interesting ”randomness” in room generation, as well as easier iterative changes during development (no need to recompile the code). Another nice improvement would be to implement a scoreboard to keep track of scores in different runs.

7 Responsibilities

	Aleba	Giog	Lucn
Animations	X		
Enemy Behaviours	X		
Powerup System	X		
Level Generation			X
GUI		X	

Table 1: Work division.