Coursework Report

Marco Moroni
40213873@napier.ac.uk
Edinburgh Napier University - Games engineering (SET09121)

Keywords – Games engineering, SFML, Box2D, C++

1 Introduction

The game that was built, *Azucena*, is a single player RPG that focuses on combat with some aspects of exploration. It takes inspiration mainly from two games: *The Legend Of Zelda* and *Hyper Light Drifter*.
The player controls Azucena, a llama mother who has to rescue her three babies. Azucena can spit at enemies or evade them by dashing.
There are three levels and the player can choose in which order it want to explore them. In each level there is a baby llama waiting for Azucena; they are behind a door that has to be opened with a key.
The game can downloaded from https://marcomoroni.github.io/azucena/.

2 Changes from original game design document

Although some things differ from what stated in the original game design document, the core idea of the game remained the same throughout the development.

2.1 Story

Azucena was meant to be rescued by a human along with her babies. In the final version she is the one rescuinge them.

2.2 Bosses

The game includes a total of three enemy types, all with its own AI. It does not, however, include bosses. This also means that the keys will only let Azucena reach her babies instead of opening a boss area.

2.3 Player actions

The main (and only) attack of Azucena is spitting: she does not have a melee attack as decided in the original document. This was purely a design choice.

2.4 Collectibles

Azucena can pick up keys and healing herbs (potions). The player has to hold the pick up key for a few second. If the key is picked it will follow the player (and then open a door when near one).
Coins have not been implemented due to the limited time. It would have required a system to buy items and/or upgrades. Instead, upgrades will be given when a baby llama is rescued.

2.5 Menu

There is no pause menu: it would have required many changes on the engine. Instead, the player can hold "Esc" for a few seconds to return to the main menu. If the player "Continue"s the game it will reload the scene but the game previous state is restored.

2.6 Tutorial

The original plan was to have one of the three levels as a tutorial level. The final game does not have such level, but instead the game explains mechanics and controls in different ways:

- Goal of the game: At the beginning of the game a cutscene plays. Here there is Azucena sleeping, while suddenly her babies run away, each to the direction of one of three three levels. The llama wakes up and the player can start playing, knowing where to go and what it has to find find.
- End game: Another cutscene will play when Azucena rescue all three of her babies. They will be happly jumping around.
- Controls: At the bottom of the screen there can be messages to help the player play the game:
  - After the first cutscene a message explains the three basic controls (movement, spit and dash).
  - The first time the player is close to a collectible a message tells what key to hold to pick it up.
  - The first time the player picks up a healing herb a message tells what key to press to use it.
- Rewards: When the player rescue a baby llama a message shows what reward the player received.

2.7 Editor

Using an image to generate the tiles by pixel colours could have been very beneficial, but due to the limited time the engine load maps from a text file.

3 Software design

3.1 Prefabs

The concept of prefabs from the *Unity* engine has been used here. In the load function of every scene there is a list of functions which create entities of various types, for instance `create_player()`, `create_key()`, `create_game_ui()`, etc. These functions are stored in a `prefabs.cpp` file and this avoids code repetition.
The prefabs are also used to instantiate entities in real-time, such as a bullets. The number of entities that can be created in this way is very limited and simple so this method was used instead of object pooling.

3.2 Map generation

The map has sprites as tiles. When the level is loading, all these sprites are merged into a single render texture, from which a single bigger sprite is created. In such a way the map is only a single image and it makes the game more performant.

3.3 Game UI

When playing, AzucenaâĂŹs health and healing herbs are always shown on the top left corner of the screen. Messages also stick to a position on the screen.
This is done by using an entity that stays always at the center of the current view.

3.4 Camera

The camera smoothly follows the player because at every frame a new `View` is set. However it stops moving before reaching the player. By doing so if the player moves by just a small amount, the camera doesn't necessarily follow it.

3.5 Changing scenes

It has been important to remember to put the code a scene only at the end of a scenes's `Update()` function. The reason is because if a scene is changed its update keep running until it reaches the end, and if there are reference to entities they won't be found because the are unloaded.
To archive this there are multiple flags that if set to `true` they will let the engine change the scene at thend of the `Update()`.

3.6 Remappable controls

To make controls remappable a lookup table as been made by using a dictionary of `<string, Keyboard::Key>` where the `string` is the name of the action. This static dictionary is stored in a `Controls` class. The class is used in the following ways.
To set or change a keyboard key use

```
1  Controls::SetKeyboardKey("Up", Keyboard::W)
```

To get a keyboard key use

```
1  Controls::GetKeyboardKey("Up")
```

For example

```
1  if(Keyboard::IsKeyPressed(Controls::GetKeyboardKey("Up"))↩
       )
2  {
3      // Go up
4  }
```

3.7 Save and load

Game informations that can be changed and should be read thourghout the game are centrally stored in static variables inside a `Data` class. This class is also responsible for saving (`Data::Save()`) and loading (`Data::Load()`) the game data to a text file.

3.8 State machines

The engine for this coursework was based on a game engine built during the semester. This engine has a state machine component, but early in the development it was clear that states had to include a method that would run only when the state is entered. For instance, if a state runs only for a fixed amount of time, a timer has to be reset every time the state is entered. Therefore states now include an `enterState()` function.

3.8.1 Enemy *A*

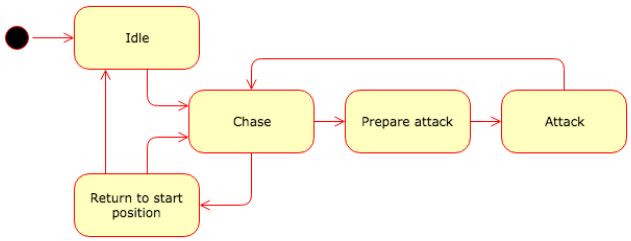

Figure 1: **Enemy *A* AI**

Enemy *A* will chase the player if it comes close enough. If the player goes away, enemy *A* will return to its starting position. This enemy attacks by dashing to the player. The *Prepare attack* runs for a fixed amount of time and it simulates a running start. The *Attack* state also stops after the timer reaches 0 and in here the enmy dashes towards the player. The direction of the dash is calculated when entering the state, so it will follow a straight line. This behaviour makes it possible for the player to evade it.

3.8.2 Enemy *B*

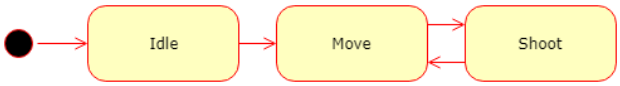

Figure 2: **Enemy *B* AI**

When the player comes close to this enemy, it will start looping through two behaviuors:

- Move three times in three orthogonal random directions;
- Shoot four bullets (up, down, left and right).

Since the maps are not very large it was not necessary for the enemy to return to the idle state if the player is far away (the same is valid for enemy *C*).

3.8.3 Enemy *C*

Figure 3: **Enemy *C* AI**

Like the other enemies, when the player comes close, this enemy will start looping though two behaviours:

- Move following a circular path with the player position as the center of such circle.
- Shoot three bullets, one towards the player and the other two in the same direction but displaced by a small angle.

3.8.4 Other

States machines have been implemented for many other non-AI behaviours, for example in the key.
A key when picked up follows the player and when the player is close enough to a door the key will move toward the door. When it reaches it, the door is unlocked. To archive this behaviour the key state machine uses these three states: *Not taken*, *Taken* and *Used*.
It could have been useful to have a state machine for the player as well. However its behaviours were programmed early in the development and it would have required a lot of work to amend them.

3.9 Physics

The game uses the *Box2D* physics engine, but not in the porper way: it sets velecities each frame instead of using impulses. This may not be the correct way to use it, but it gives compete control overe every movement. This choice has been made also because the game has a top-down view therfore its entities do not need to b affected by gravity.

4 Game implementation

5 Implementation evaluation

Overall, the original game idea has been almost fully implemented.
It resembles a lot *Hyper Light Drifter*: interaction with objects, shooting, movement, etc. However, this is very standard formula so the changes between these type of games are mostly aesthetic.
The game can be fully completed and if it had more

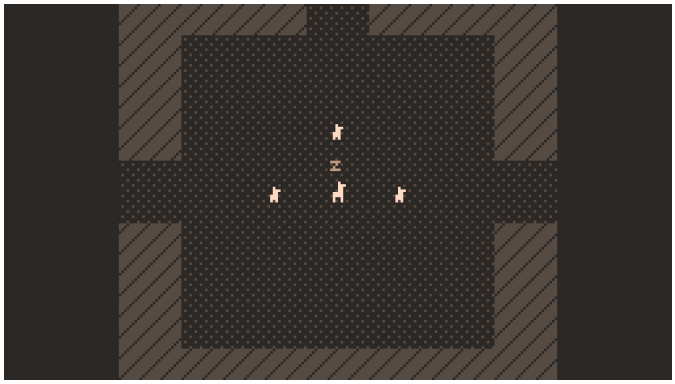

Figure 4: **Game still**

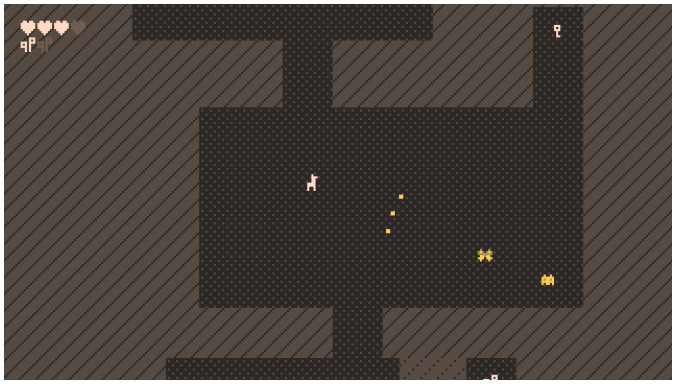

Figure 5: **Game still**

ememy variety and bigger maps, it could potentially be considered a *complete* game.

6 Performance analysis and optimisation

7 Playtest

Most players were satisfied by the quality of the game, especially by the dash controls. A common comment was that it "looks like a complete game". Sometimes there are problems with the physics system, but they can be temporarily fixed by restarting the application. One playtesterâĂŹs game was unable to load the map texture.

8 Project managment

GitHub has been the main tool for project management. Issues have been used to track bugs, mandatory features and features to add if there was additional time (and they have been tagged accordingly).
The repo has implemented continuous integration with *AppVeyor* on the `master` branch. Features have been added in new branches and then merged into the `master`

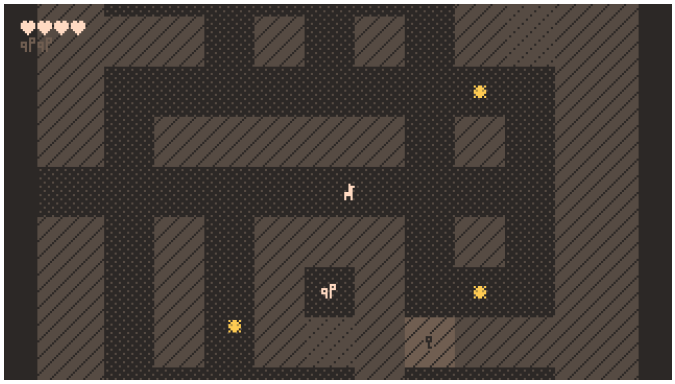

Figure 6: **Game still**

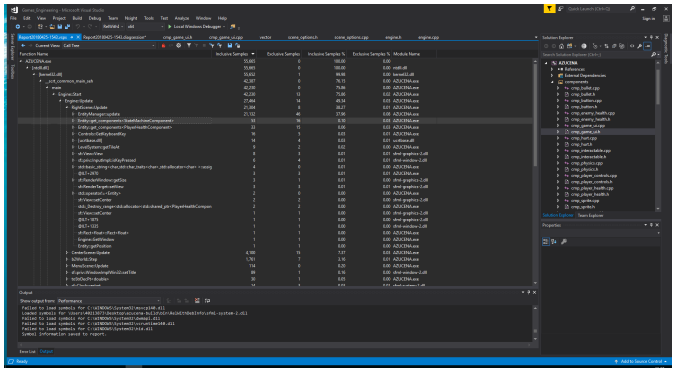

Figure 7: ***Visual Studio* performance analysis**

branch. It can be seen that some feature have been attempted multiple times.

9 Resources

- Sound effects: https://opengameart.org/content/512-sound-effects-8-bit-style
- Music: http://ericskiff.com/music/
- Font: https://datagoblin.itch.io/monogram
- *OpenAL Soft*: http://openal-soft.org/
- Color palette: http://simurai.com/projects/2016/01/01/duotone-themes

References

[1] C. Ericson, *Real-time collision detection*.