Technical Report: Galaga

Software Development 2021
Department of Computer Science
University of Copenhagen

Kristoffer Colding-Poulsen <bwk298>
Wadah Ahmed <rds343>
Oscar Kofod <qhm995>

Friday, April 2, 15:00

1 Introduction

We have developed a galaga-like game with the DIKUArcade game engine. The development of the game was part of assignments 4, 6 and 7 of the Software Development (SU) course at DIKU. The game itself is hosted on github[1].

The project started with us getting an engine –the DIKUArcade engine– and implementing a simple game using that engine in assignment 4. This was a game without menus and without moving of enemies. The next assignment saw the game transferring from a very simple and boring game to a more challenging game with more intricate features, thus making it more satisfying to play.

Assignment 7 saw the game going from a very simple yet somewhat enjoyable game to a more functional and more playable and enjoyable game and game interface. This report will look at the changes made from the beginning of this process as well as the considerations and decisions made as this project went on.

2 Background

Galaga is an old Japanese arcade game developed in 1981 by the Japanese video game developer Namco. It is a fixed shooter arcade game where the player, plying as a starship in outer space, is trying to destroy the enemy aliens. The objective of the game is said to be to defend all of mankind from enemy aliens by firing the enemies down with ones starship. As the game progresses the game gets harder by the aliens increasing speed towards the lone savior of mankind.

[1] https://github.com/kcp98/DIKUGames

3 Analysis

In this project we will be implementing a galaga-like game with the help of the `DIKUArcade` game engine. In this project we will not be diving into the details of how the `DIKUArcade` is implemented. We aim to create a fully functioning galaga game that can be played with the OpenGL library. The game should be written in C#, run on any operating system and include the following design goals;

- The player should be able to control a spaceship using the arrow keys.
- Enemies that the player can shoot to earn points.
- A losing condition for the game to end.
- An interface that serves to control the states of the game.

Our design of the game is to be object oriented with each class being responsible for a different part of the game. Methods and classes should also be able to be individually tested using the `NUnit` framework. Designing this way ensures us that the different parts of the game can be worked at in parallel, as this is a group project after all.

4 Design & Implementation

We implemented the game as a single C# project. As a consquence of our design choice it is expected that we end up with many different classes that all serve contribute to meet the design goals discussed in section 3. We will only be discussing the main parts of the game. The structure the game is as follows.

4.1 Game

This is the main class of the game, which is responsible for rendering, running and updating the game as a whole. The class creates an `OpenTK` window with relevant data and an `EventBus` which is class implemented in `DIKUArcade` responsible for registering and processing events that occur during the game. The class has a `Run` method that updates and renders the states of the game.

4.2 Player

The player class contains the method and properties that enable the player to move. The class implements the `IGameEventProcessor` interface which is also implemented in `DIKUArcade`. The interface requires a method that is used to register and subscribe entities to receive events from the `EventBus`. The player consists of a shape and an image of a pixelated spaceship.

The implementation deviates from the description in the exercise set, because the player did not move quite as we expected.

4.3 Squadron

This namespace contains the squadrons which are the enemy in the game. They are in fact the only enemy but have different variations. When a squadron is killed it initiates *enrage* mode and transforms into a more powerful squadron. The normal and enraged squadrons are displayed as sprite placed in the `Assets` directory . The squadrons implement an interface with a method that creates enemies.

4.4 MovmentStrategy

Contains the implementation for the movement of the squadrons. There are three possible movements a squadron can do. It can either `NoMove`, move `Down` or move `ZigZagDown`. These three methods implement the `IMovementStrategy` interface. As the game progresses the movement strategy alternates and the movement of the squadrons become unpredictable and therefore harder to beat.

4.5 GalagaStates

The game has different states that should be displayed in different stages of the game. We created a namespace that include the different states of the game. These states do construct the gameplay loop as shown in figure 1. The classes are implements as singleton pattern. This way we can make sure every state has only one instance. Using singletons we can also change states without effecting the fields and properties.

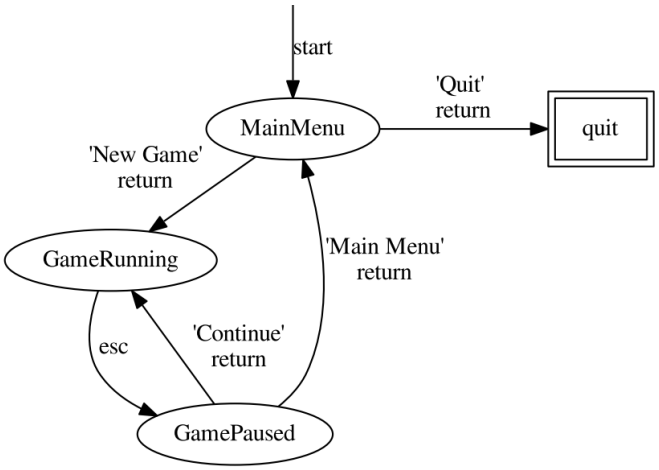

Figure 1: The different states of the game

4.6 MainMenu

When running the game the player is met with the `MainMenu` state. The main menu is kept very simple. There is a background and two button the player can pick from. The play can start the game buy pressing *New Game* and exit the game by pressing *Quit*. A mechanism to distinguish the active button from the inactive one was also implemented.

4.7 GamePaused

This state is similar to the main menu. It has a background and two buttons, continue and quit. However, the difference is in the registered events. `GamePaused` should not start a new game like the main menu but continue the present game.

4.8 GameRunning

`GameRunning` is the core of the game. The class a constructor that initializes the objects needed for the game to run, such as the player, the enemies and the background. While the `Run` method in `Game` is the main gameloop, `GameRunning` keeps track of the variables and entities while the game is running. The class also includes two function which are used to detect key inputs from the player. The inputs are then registered by the `EventBus` as an event that the player can subscribe to. The class also includes a method that checks if the game is ended, if that is the case, then the player is shown a game over screen with the score. We did not implement it in a standalone state as we thought that it was trivial.

4.9 StateMachine

The `StateMachine` is used as a mean for the `Game` class to communicate with the different states. It is responsible for keeping track and switching states appropriately. Like the `Game` class `StateMachine` implements the `IGameEventProcessor` and therefore also implements `ProcessEvent`. `StateMachine` processes two types of events. InputEvents and PlayerEvents. The InputEvent can be any key pressed or released. Once a key is pressed or released the input is transffered by the `HandleKeyEvent` method to the appropriate state. If the event is a `GameStateEvent` the `SwitchState` method is called. The method determines which state to change to based on the input given.

Other than these main classes there are minor classes that contribute to the functionality of the classes described above. When the program is executed the `Program` class' main method is run. The methods is only a couple of lines and only creates a new `Game` instance and calls the `Run` method. These classes should be able to help us satisdy most of our design goals.

5 Evaluation

As it is most aspects of the game can be easily tested by manually running and testing the game. We wrote tests for all non-trivial parts of our implementation to better support further developments and convince ourselves of the quality of our implementation.

The Score class was not tested, as its only public methods were, those that increased the score and displayed. Therefore we could not test whether the score was rendered correctly without human observation. The limited functionality of the class, allowed us to easily test its implementation by manually running the game.

6 Conclusion

We have made a simple galaga-like game. We have run the game on a mac and windows machine, and have found that it runs reliably on windows. But due to the graphics library, which the DIKUArcade engine uses, we have not been able to run the tests on a mac, and have found a few inconsistencies in how the game is rendered. That is beyond the scope of this project, instead further development, may focus on different levels in the game and adding more difficulty to the enemies, such as shooting at the player. The structure of the game makes these extensions manageable.