Technical Report: Breakout

Software Development 2021
Department of Computer Science
University of Copenhagen

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

Wednesday, June 9, 15:00

Contents

1 Introduction

This report describes the ongoing implementation of a Breakout game using the DIKUArcade engine.[1] The application has been developed in C# using the .NET 5.0 SDK as part of the course *Software Development* at UCPH. The game is hosted on GitHub.[2]

2 Background

Breakout is an arcade game developed by Atari in 1976. The goal of the game is to destroy rows of blocks by using a ball and a paddle. The ball bounces on the paddle, hits the blocks and bounces back. The ball is lost when it leaves the bottom of the screen and the player has three balls to complete two levels. At certain intervals the ball increases in speed and the paddle narrows.

[1] github.com/diku-dk/DIKUArcade
[2] github.com/kcp98/DIKUGames

level4.txt

```
Map:
---------
xxxxxxxxx
---------
aaaaaaaaa
---------
vvvvvvvvv
---------
---------
Map/

Meta:
Name: LEVEL 4
Time: 30
PowerUp: v
Unbreakable: x
Hardened: a
Meta/

Legend:
a) red-block.png
v) darkgreen-block.png
x) grey-block.png
Legend/
```

Figure 1: Level .txt file

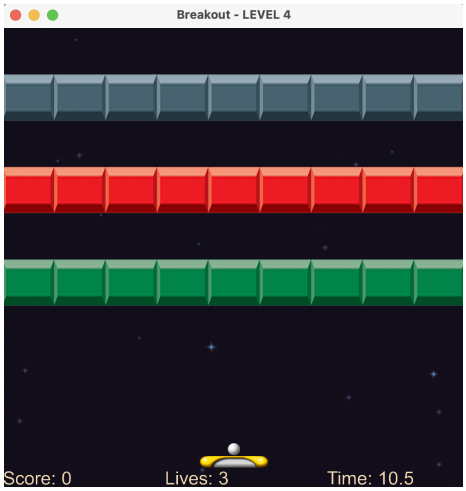


Figure 2: Rendered level

Our game consists of far more levels, and figures 1 and 2 show how one might extend the game with personalised levels. As can be seen, the blocks are displayed in the top four fifths of the window, whereas the original game places them in the top third only. In the meta field one can also specify blocks with special behaviors, such as releasing power ups. Furthermore the difficulty does not increase as the game progresses, instead the time can be reduced for later levels.

3 Analysis

As mentioned, this project will be using the `DIKUArcade` game engine to implement the breakout game, furthermore we have been given assets to utilise in doing so. The game can be categorised into several components, whose requirements have been provided in intervals of two weeks. The combined specifications, we have extracted are listed for each component below.

`Player`

The first and most essential part of any game is a user controlled player. With the given requirements, the following specifications were made. The player should:

- Start in the horizontal center and move between the edges on the x-axis with the right and left arrows of the keyboard.
- Be a `DIKUArcade` `entity` and must be of a rectangular shape.
- Reside in the bottom part of the screen.

`Blocks`

Just as essential for a breakout game is the block which the player must destroy. The blocks uses the level files to get their placement (mentioned below) and should have the following features, specified from the requirements:

- Must be a `DIKUArcade` `entity`.
- Must have a health and value property, and the health property must decline when the block is hit and cause deletion when the blocks health reaches zero.
- Must use the level files to get the placement of the blocks, and must be rendered corresponding to the level files.
- Should be easy to implement new block types.

Level Loading

Part of the assets given, were a folder of `.txt` files representing levels, which allows for effortless level creation. As such we need functionality to read these files and create a playable level. With the given requirements, the following specifications were made. The functionality of level loading should include:

- Reading the lines of a `.txt` file.
- Processing the different fields of a file, and handle variations in the meta field.
- Relaying the fields' information in appropriate data structures.
- Creating the blocks of a level using these data structures.
- Handling empty or invalid `.txt` files gracefully.

`StateMachine`

For a smoother experience the game further needs a menu, pause menu and a score displayed when the game is over. To handle switches between these and the running game, we were required to add a state machine, along with states for all these different player interactions. This also helps clarify the responsibilities of the different states. We believe the following specifications satisfies this goal. The functionality shall provide:

- One active state at a time, as a property of the state machine.
- An ability to switch between states.
- The following states, which must implement the `IGameState` interface.
- A main menu state for starting a new game or closing the window.
- A running game state containing the game logic and level progression.
- A game paused state for pausing the active game.
- A game over state displaying the final score, when winning, or when running out of lives or time.

`Ball`

The ball is an important aspect of the game, which enables the player to break blocks. The specifications for the ball listed below are mostly derived from the given requirements. Furthermore we have given the ball a few responsibilities which might have been handled in the game running state, but have been delegated to this both for clarity and to not clutter the game running class.

- Be able to check for collisions.
- Change direction upon collisions, and not get stuck in trajectories.
- Remain the same magnitude of its velocity.
- Stay within the borders of the window, but leave the screen in the bottom.
- Follow the player until they release it in an upwards direction.

Points, Lives and Time

The points is important as an informative tool for the player to know how well he or she is doing. The requirements both called for displaying points, and time. We felt that showing the points and time together with lives made for a cohesive showing of adjoined components. Therefore the specifications were made for these three components combined. We have chosen to call this combination status further down in the report and in the code.

- Points, lives and time must at all times be non negative.
- Should be displayed on the screen for the player to see.
- Points should increment when a block is destroyed, based on the value of the block.
- Lives should decrement when the last remaining ball leaves the screen.
- Player should gain a ball as long as lives remain if not the game should be over.
- Time should decrement only when game is running.
- Levels without time, should not display it.
- Game over if time reaches zero.

`PowerUp`

The following specifications are required to handle power ups in the game.

- Special blocks should release a random power up upon destruction, with a visual representation of the power up.
- Power ups should fall straight down and only interact with the player.
- When colliding with the player it should activate its power.
- The event bus shall handle activated power up events, so that subscribed classes have the necessary logic for the different power ups.

4 Design

The engine provides a class `DIKUGame`, which opens a window and includes a method to enter the game loop and run the game. It includes two functions which must be overridden; `Update`, wherein we must call the active states `UpdateState` method and process the events in the event bus, and `Render` which should call the states `RenderState` method.

`Player`

The player class represents the platform that is used to keep the ball from falling. To make the interaction between the blocks, the ball and the player easy, they all were designed to be sub-classes of `Entity`. The `Player` class also implements the `IGameEventProcessor` interface to handle registered player events like moving. This implementation follows the observer design pattern, in that a subject registers the player events that are sent to the `Player`, which in this case is an observer.

`Blocks`

The blocks, like the `Player` class should inherit from entity. However, we will be implementing different kind of blocks with different special attributes. Thus, we can create a single default `Block` class and special sub-classes which inherits it. This design adheres to the Liskov substitution principle. By having the special blocks be sub-classes of the default block we can use the blocks interchangeably. It also adheres to the open-closed principle since adding new blocks will not need modification to the base block.

Level Loading

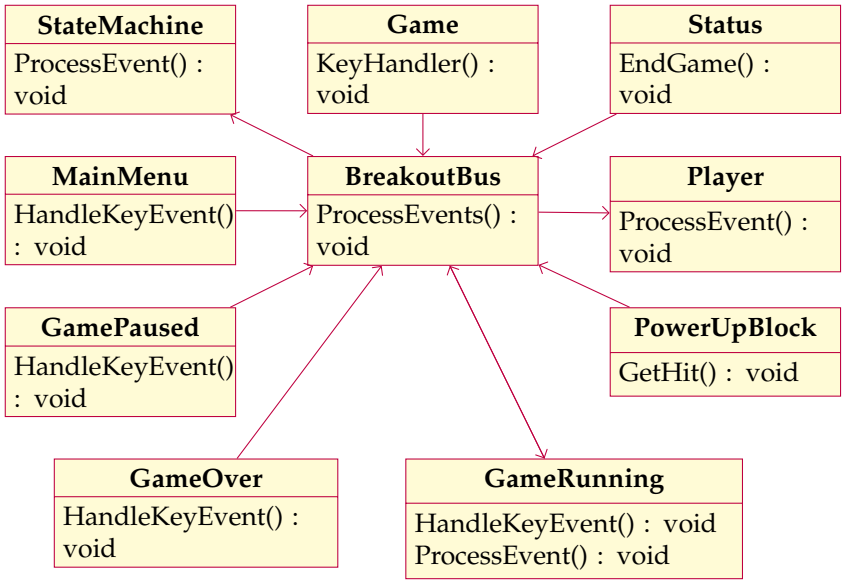
In implementing the level loading we followed the single responsibility principle. First off we split the functionality into two classes. `LevelLoader` being responsible for reading the contents of a level file and storing its contents in the appropriate data structures. `LevelConstructor` then uses these to place blocks in an `EntityContainer` among other things, such that an instance of this class represents a playable level. Furthermore the `LevelLoader` class splits the processing of the three fields of a level file into three distinct methods.

`StateMachine`

The state machine implements the `IGameEventProcessor` interface to process events for switching between states. Each state implements the `IGameState` interface, to include the logic of updating and rendering the states. To allow for easy pauses of a running game, we decided to make the `GameRunning` a singleton class, and for clarity we decided on the same pattern for all states.

Event Bus

The UML class diagram below conveys the complexity involved in communicating events between relevant classes of the program. This became especially clear when designing the states and state machine. Furthermore most of the classes shown exist at different levels of the application. As such we decided to create a static event bus with the `BreakoutBus` class, following the singleton pattern.

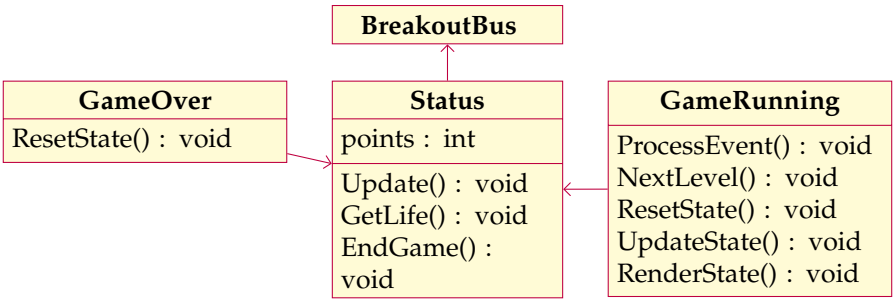


`Ball`

To implement the changing of direction the single responsibility principle was followed. We thought it best to create several methods for the different ways a collision might change the direction of the ball. This means that the direction change after collisions can be changed in any number of ways with minimal changes only to the appropriate methods.

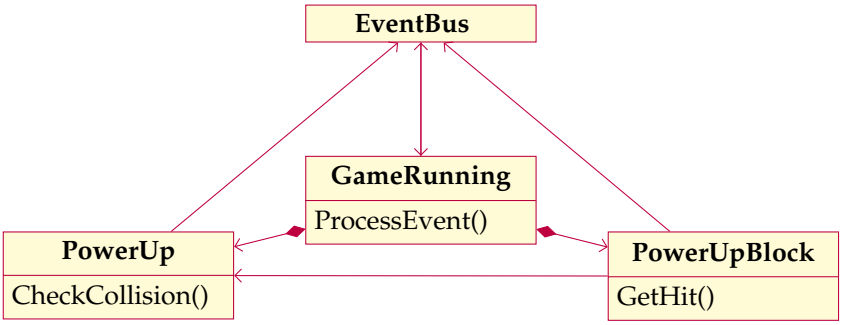
`Status`

As we wanted to display the score, life and time at the same place, they were implemented in a single class, Status. The implementation follows the singleton pattern of having it as a static class. As the functionality of score, life and time were limited, the functionality was chosen to be implemented mostly from existent classes. The function and relations can be seen in the UML class diagram below.



Power Ups

We decided to create five relatively simple power ups, still we realised that the effects of these had to be created at a number of different places in the program. The first thing we had to do though, was create a new special block type, which would spawn the power ups upon destruction. These powers ups then had to be sent to running game, to be updated and rendered, and upon collision with the player, be responsible for activating powers. The relations can be seen in the UML class diagram below.



5 Implementation

Our Breakout is implemented as a single C# Console Project, with an adjacent NUnit Library Project for unit tests. The entry point of Breakout is located in Program.cs, but does little else than fetch an instance of Game, and call its Run method. The following is the implementation details of the design discussed above.

`Player`

As mentioned, this inherits the `Entity` class from the engine. With private move methods it is also able to move the position left and right when processing events. The position is then updated by calling the public `Move` method before rendering. Though the coordinate system of the window goes from 0 to 1, the player, with its current size, must not move beyond 0.7 in the x axis, otherwise it would appear to move out of the window. Furthermore when a ball, which can interact with the player, is added, it will

be easy to gather the players position either from the base classes `Shape` property or adding a `GetPosition` method doing it for us.

`Blocks`

The block class inherits the `Entity` class from DIKUArcade. It has a health(`Int`) and a value(`Int`) property. Every block has a `filename` value which represents the image of the block. It also implements the `GetHit` method. The method is responsible for the appropriate behaviour when a collision between a ball and a block appear. If the health reaches zero the block is deleted.

Some blocks has special attributes that serve to make the game more challenging. They inherit from the `Block` class. Through the assignments, three subtypes of blocks has been created as well as the simple blocks. These blocks are `Unbreakable Blocks`, `Hardened blocks` and `PowerUp Blocks`. They were made by manipulating the block's properties through inheritance. However, our design implementation allows us to easily extend and add special block types without the need to modify the code.

The `BlockCreator` class is a static class that is called in `LevelConstructor`. The class has a static `CreateBlock` method that takes a position, extent, filename and a string that represents the type if the block has a special type. The method uses these data in a switch statement and return the correct type of block. The choice of using a static class was that the only need for the class was to use its one method, and there was no need for it to inherit or be inherited and to make sure of that not happening the class was made static.

`LevelLoading`

A level is represented by the blocks in the `EntityContainer` property of this class. When making a level with the `LevelLoader` class, the `LevelConstructor` class is used to read the data of a level file. `LevelLoader` then fills the container with blocks using the agreed upon data structures. The render method, simply calls the `RenderEntities` method of the container, whose `Iterate` method can later be used to handle interactions between a ball and block.

The `LevelLoader` class handles the transformation of `.txt` files into the aforementioned data structures. It contains only private methods which are called in the constructor. So that when a `LevelLoader` class is instantiated, it first validates that the files format is as expected, and otherwise throws an appropriate error. It then calls the respective methods to fill the data structures of the map, meta, and legend fields. These methods being separated has the benefit that future changes in the format of one field, is easily updated in its respective method.

`StateMachine`

The `StateMachine` class implements the `IGameEventProcessor` interface to handle events for switching between states, which are stored in its only property `ActiveState`.

The four states were then added, implementing the `IGameState` interface. The `GameRunning` state includes all the relevant game logic. For one it includes all the entities, and handles the interactions of these in the `UpdateState` method. Furthermore it includes a `NextLevel` level, which resets the entities and use the `LevelConstructor` to prepare the next level. Lastly it implements the `IGameEventProcessor` to process power up events, as most of the power ups are methods concerning the entities of the game.

`Ball`

The ball is an `Entity`, and it has a move method that takes a player as a parameter so that the ball itself can keep score of whether the ball has been released. It has a public method using the `CollisionDetection` to check collisions and set its direction after a collision. The ball also has a private border which represents the border of the window, to keep it from leaving the screen in unwanted places.

`Status`

The status bar holds and displays the points and lives for a game as well as the time for a given level. It was made as a singleton class to simplify the placement of the `Text` elements in the window, in a pleasing manner. It also has the benefit that the relevant information is easily available.

`PowerUps`

PowerUps inherits from `Entity` as the physical part of a power up being released and then being able to "pick up", needs the same functionality of an Entity, the same way as a `Ball` for example. It has a static list of file names, representing the different power ups chosen to implement, and a random integer given to the constructor serves as the chooser for which of these power ups with a corresponding file name will be used. PowerUps has a `move` method that moves the entity directly downwards from its release point and deletes itself if reaching the bottom of the grid without making contact with the player. The `CheckCollision` method checks for a collision between powerUp and player and if occured sends out a `GameEvent` to our `BreakoutBus` with the given instance of powerUp's index before deletion.

6 Evaluation

Our development have been only partly test driven. Before implementing the game we have written tests that vaguely describe the intended behaviour of the different classes. To ensure that our implementation satisfies the requirements of the classes we derived specifications that we test for using NUnit test fixture. Our main strategy in testing was to use assertion to verify if the functions return the expected output. Thus we had mainly used White Box testing.

Every feature has been partly tested visually, since many of them have private properties that are only used to render in the window. Where possible, we wrote black box tests for the constraints set by our specifications for each feature. We did not follow a formal test-driven development. Sometimes we implemented some functions and then tested if the functionality is working as expected. This was easy to do because most of our implementation follow the single responsibility principle and we could create tests that were not complicated.

Due to the fact that our implemented project is a game, rendering and graphical problems made mechanical testing challenging. By using a plugin for Visual Studio Code we found that our statement coverage was 50%. Although we had tested 75% of the specifications that we made. The rest could not be tested because of their properties. All of the tests passed and we believe that the parts not covered were better tested visually.

7 Conclusion

We have made a Breakout-like game. Both the implementation and tests have been made and they have both been ran on mac computers and a windows computer. After the issues found with testing in the last project the tests have been made without regard to the graphics library to prevent problems with running tests on mac computers during the development process. In the final delivered code, the tests have been set up using the graphics library and tested finally on a windows computer.

The given requirements of the project has been met and makes a good building block for further development of the game. It is now a fully playable game and it corresponds with the given requirements through the assignment texts given to us over the course of these last two months.