Breakout

Software Development 2022

Department of Computer Science
University of Copenhagen

Samual Cadell, Artur Barcij og Ludvig Schroll

Thursday, April 28, 16:00

Github link: https://github.com/ilikecats17yo/DIKUGames

1 Abstract

This rapport deals with the basics of setting up the game "Breakout", and the design that has laid the foundation for our work.

2 Introduction

For this paper we aim to document the first steps of setting up a game of Breakout and giving an insight into the design processes and decisions taken in the early stages of development

3 Background

The game "Breakout" was realeased by "ATARI" back in 1976, it was orignally designed by Steve Wozniak. The game in it's simplest form consists of a player in the lower half of the screen that has to eliminate a range of different blocks in the upper half of the screen.

We have been given the assignment to emulate this, and in doing so we have to implement different classes, that are supposed to simulate entities, for the player, ball and the block. Furthermore we have been asked to create this game on top of the "DIKUArcade" game-engine that has allot of the functionalities important to set up a computer game, like user-interaction and support for the graphical interface.

4 Analysis

To sum up the requirements: We need to make a full functioning game that resembles the old "Atari" game "Breakout". To do this we ofcouse need a player, blocks and level-loading. Furthermore we have been asked to implement a variety of different power-ups and blocks that can bring more variety into the game.

The requirements that we are expected to fullfill will be met by using unit testing and integration testing. Furthermore we strive to achieve c1-test coverage, in order to verify that all pars of the game works properly.

The "DIKUArcade" game-engine has all the functionalities relevant to loading the graphical interface of the game. Furthermore is also makes it easy for us to setup interaction between the user and the game, which is also integral to the gameplay.

4.1 Formal specification:

4.1.1 Level loading:

- The level-loader should be able to handle differences in metadata.
- The data should be stored in a appropriate data-structures
- Invalid ASCII-files shouldn't crash the program

4.1.2 Blocks:

- All blocks must be a subclass of the `Entity` class within "DIKUArcade".
- All blocks must have a health- and value-property.
- The position must match it's corresponding position in the ascii-map
- When the block is hit, it's health, **BlockHealth**, must decrement. It's health before it is hit **PrevBlockHealth** minus the damage it has taken when hit must be equal to it's current health: PrevBlockHealth-damage = **BlockHealth**, when hit.
- When the block's health is zero, it shouldn't be rendered. If **BlockHealth** <= 0, then it shouldn't be rendered.

4.1.3 Player:

- A players position **P** must be in the horizontal center of the screen, therefore **P.x** = 0.5f.
- Upon pressing "a" or "d" the player should move horizontally. When "a" is pressed the player is supposed to move left and it's previous position **PrevP.x** must therefore be larger than it's current position **P**: **Prevp.x** > **P.x**, when "a" is pressed. The opposite should therefore hold when "d" is pressed: **PrevP.x** < **P.x**, when "d" is pressed.
- The player shouldn't be able to leave the screen, therefore **P.x** >= 0.0f and **P.x** <= 1.0f, at all times.
- The players position must not stutter when attempting to leave the screen, this is visually tested.
- The players must be a subclass of `Entity` from "DIKUArcade".
- The player must default to a rectangular shape.
- The player must be in the bottom half of the screen: P.y < 0.25f.

4.1.4 Ball:

- The ball's position, **BallPosition**, should only be able to leave the bottom of the screen, therefore **BallPosition.X** <= 1.0f, **BallPosition.X** >= 0.0f and **BallPosition.Y** <= 1.0f.
- When colliding with a block, the blocks is "hit".
- All balls, **AllBalls**, must move with the same speed, **BallSpeed**: $\forall x \in$ **AllBalls**, **BallSpeed** must stay the same

4.1.5 The statemachine

- There should only be one state active
- All states, **AllStates**, should inherit the `IGameState` interface at some poin therefore: $\forall x \in$ **AllStates** should inherit `IGameState` interface
- There must be a state for, when the game starts, when it is running and when it's paused
- The active-state within the statemachine should automatically be changed when different states are instantiated (when the game is started, when it is paused etc.)

4.2 Goals for Design:

- Should be readable and easy to understand from an outside perspective
- Should abide the solid principles and the observer- and facade pattern
- Should be easy to test and adaptable to new changes and fixes

4.3 Goals for implementation:

The goals for the implementation can be summed up in the following:

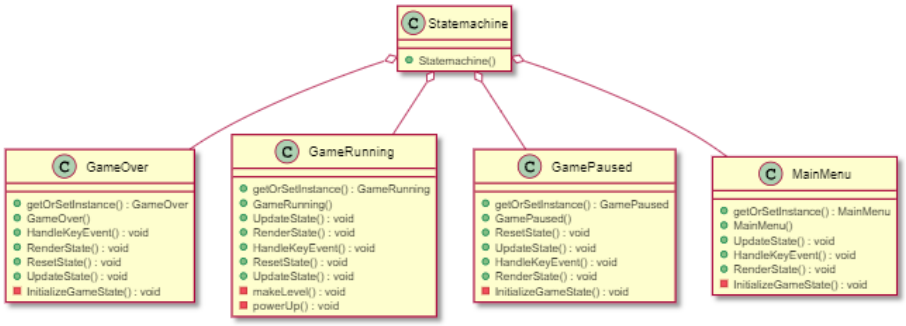
- Keep the implementation readable and easy to extend
- Seperate code into their respective files, and making the state-transition smooth and easy to understand
- Hide away lower-level code in the `facade` class
- Utilizing the observer pattern in order to improve encapsulation
- Should be easy to test

5 Design

This section will explain the design of our game aswell as the process that went into it. Beneath here are the most important parts of the game:

5.1 Statemachine and states:

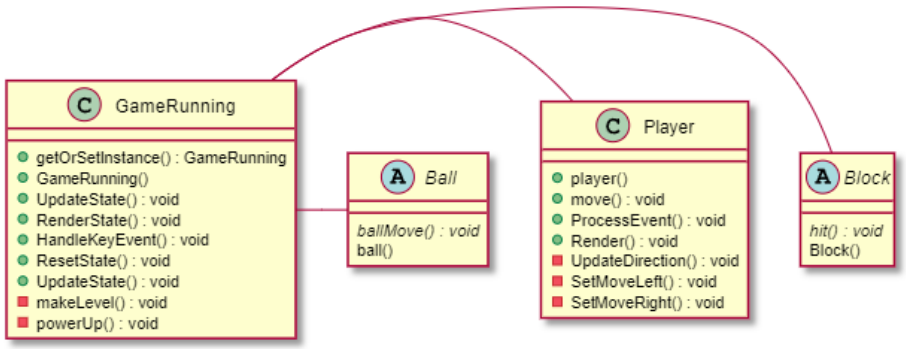
The statemachine is responsible for switching between the different states, which is the `MainMenu`, `GameOver`, `GameRunning`, and `GamePaused`. The `MainMenu`, `GameOver` and `GamePaused`, as the names suggests, only has the information and interaction required for pressing the different buttons. The `GameRunning` state has besides the basic user interaction, the functionality responsible for loading different levels, and spawning the different powerups when they are activated. The `GameRunning` state also switches to the different states when the game is either won or lost.



5.2 `GameRunning` and the different game entities:

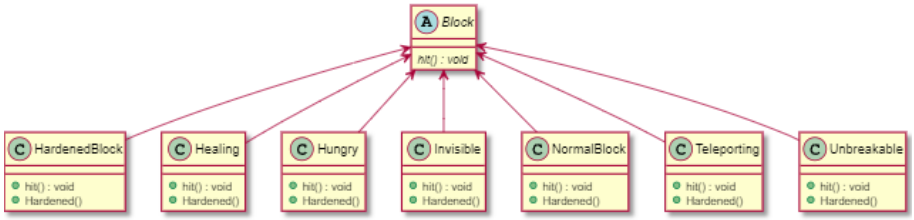
The `GameRunning` as mentioned before is responsible for loading much of what is important to loading each level, that includes the different entities of the game, blocks, the player and a ball, and more. The blocks and ball all inherit from their respective abstract class that have some basic functionality that all the subclasses in turn must inherit.

These abstract classes only have methods that are of direct relevance for it's subclasses. When designing these abstract classes we were very careful not to create functionality that could be broken by the subclasses, and we therefore chose this very simplistic approach of letting the subclasses be dependent on only one method that just makes them compatible with the other parts of the game. Furthermore this complies with the: "Liskov substitution principle".



5.3 Collision detection and different block types:

The collision between the ball and the blocks and the player is handled with in the specific block. All the block inherit from the abstract class `Block`, which only require it's subclass's to implement the `hit` method, since all the different block get hit in different ways.



5.4 `Facade` class:

We have chosen to use the `facade` pattern, since it becomes very easy to read and maintain the codebase by using this design pattern. When working with the "DIKUArcade" game engine it can quickly become very difficult and confusing to manage the program, the `facade` class hides alot of the lower level code from the classes related to the "Breakout" game.

This also fullfils the some of the "SOLID" principles, specifically the "Single Responsibility" principle and "Dependency cohesion", because it hides away alot of the lower level game implementation from the "Breakout" classes, which should only be responsible for the higher level game-logic. It does this by creating an intermediary layer between the classes concerned with the higher level game-logic and the `facade` class that only deals with the lower level implementation and making both of them dependent on abstractions, like the

abstract classes mentioned before used to handle different blocks and balls.

The methods within the `facade` class are all adaptable. One example could be the `placeBlocks` function which also takes a list of different block types so we can spawn multiple different types of blocks instead of having to create a method for each different block-type. The `checkForCollision` and `checkForNewPosition` are both generic and can check for collisions with both blocks and the player. These aforementioned functionalities support the second principle: "Open-closed principle" of the "SOLID" principles, since the methods are adaptable to many different types and can't be modified.

5.5 Powerups:

The different powerups are all handled within the `PowerUpBall`, since the powerup is first spawned when the player hits the powerup-ball. And we have therefore chosen this class to signal to all the other entites of the game when different powerups are spawned.

The `PowerUpBall` class uses the observer pattern to send out different signals to blocks and the `GameRunning` state. This makes a lot of sense since the game needs to move on while powerups are spawned at different times. By using this design pattern we can also utilize the features of sending timed events that are proceessed after a certain amount of time has passed, which makes a lot of sense when dealing with timed powerups.

5.6 Responsibility table

Responsibility Description	Type	Concept Name
Coordinate between different states of the game	D	C. 5
Extract and interpret data from ascii-files	D	C. 1
Container with all the information relevant to each block	K	C. 2
A player that can be interacted with, and that can interact with the ball	D	C. 4
A container that holds all the relevant information relevant to the player (rewards, points etc.)	K	C. 4
A ball that the player can interact with and is able to interact with other parts of the game	D	C. 3
A bridge between the higher level game-logic and the actual game-engine	K	C. 6
A `GameRunning` state that load blocks, and their associated attributes, a movable player and a ball	D	C. 9
A `GameMenu` state from where the player can quit the game or start a game	D	C. 7
A `GamePaused` state that can be instantiated during the game and navigate to other states of the game	D	C. 8

5.7 Association table

Concept pair	Association description	Association name
$8 \leftrightarrow 7$	The statemachine should be able to initialize the main-menu	Change state
$7 \leftrightarrow 9$	From the main-menu the player should be able to start a game	Change state
$9 \leftrightarrow 1$	The `GameRunning` state should be able to retrieve information about the formation and metadata relevant to each block	Retrieve information
$9 \leftrightarrow 6$	The `GameRunning` should spawn blocks, a player and a ball	Generate game
$4 \leftrightarrow 3$	The player should be able to interact with the ball	Interaction
$3 \leftrightarrow 2$	The ball should be able to interact with the blocks	Interaction
$9 \leftrightarrow 8$	While the actual game is running the user should be able to pause it	Change state

5.8 Accordance with the goals for design:

The design is easy to understand since all the responsibilites are seperated and aren't clustered in the same class or function. It does abide the solid principles, by utilizing the `facade` pattern and the observer pattern.

The use of abstract classes for different entities does make the code adaptable to new changes in the future and fixes that need to happen during testing.

6 Implementation

6.1 `facade` class:

Almost all of the classes use the `facade` class. All of the game-states use it to load different entities like, text, blocks, a player etc. This function within the `facade` spawns these entities using the information given to them in the parameters. Furthermore we have also overloaded the function, so they don't have to take a whole lists of different positions, shapes etc, but only a tuple and strings and so on, this make the code easier to read and use from the calling class. The following is the `placeBall` method that can spawn different balls, by giving a list of different types of balls:

```
1 public List<ball> placeBall(List<(float,float)> positions, List<(float,float)> shapes,
    List<string> imagePath, List<Type> specialities, gameLevel caller)
```

Here is an example from the `placeBall` method. This method also needs to take instance of `GameRunning` state, in order to get the different entity containers. There also exists `placePlayer`, placeStationaryEntities and `placeBlocks` methods for spawning other types of entities, as the names suggests.

We also have a function in the `facade` class responsible for checking for collisions between different entities, this is used alot by the different balls. This is also in line with the "Dependency cohesion" principle since it hides lower level code from the `DIKUArcade` game-engine that is concerned with checking collisions between entities, from the higher level modules in the "Breakout" folder:

```
1 public (T,bool,float,float,CollisionDirection) checkForCollision<T>(DynamicShape actor,
    List<T> collidingElements) where T : Entity
```

This method can take a `DynamicShape` and a list of `Entity`. By not specifying which `Entity`, users of this method can use it to check for collisions between players and blocks, and not just one of them, since they are both entities.

6.2 Abstract entities:

We have used two different abstract classes in this project to simulate balls and blocks. We haven't used an abstract class for the player, since there is only one type of player.

Besides having an abstract method that the subclasses must implement, the abstract class `Ball` also has a ProcessEvents method used to retrieve messages. This is necessary for spawning and deactivating powerups, more specifically the powerups directly related to the balls. The non-abstract class `Player` also has this method since we need to handle powerups that are only relevant to the player, which wouldn't make sense to place elsewhere.

To handle timed powerups, powerups that are activated and deactivated later, each message recieved by the `ProcessEvents` method has a unique id, that has to be matched by the message arriving later. Doing this we can extend the time of powerups, without having to cancel them. Here is a code-snippet from the switch-statement within the `ProcessEvents` method in the abstract class `Ball`:

```
1   case("Hard Ball-second"):
2       if(gameEvent.Id == hardballTracker){
3           hardball = false;
4       }
5       break;
```

Nothing is done if the last id given doesn't match.

Each of the different powerups are spawned randomly, using the `Random` class. More specifically the actual powerup is chosen randomly from a dictionary consisting of all the different powerups and their asscociated picture.

The actual ball that represents the powerup is spawned using the `PlacePowerUp` method within the `GameRunning` state. This method adds information to the dictionary `currentPowerUpBalls` that has information about which powerup is linked to which specific ball, which is then accessed by the `PowerUpBall` class when the player collides with one of the powerup-balls. This is effective because the time taken to look up which powerup matches which specific instance of a `PowerUpBall` is constant in time. Here is short code-snippet from the `PowerUpBall` class showing how the powerups are loaded within the `powerUp` method:

```
1   if(caller.getCurrentPowerUpBalls[this] == "Extra Life"){
2       Console.WriteLine("extra life");
3       caller.getHealth = caller.getHealth+1;
4   }
```

6.3 Statetransitioning:

All the different states passes on an instance of the current statemachine that is in use. The current state that is being processed is the one that is equal to the `ActiveState` field within the `StateMachine` class.

By passing on an instance of the `StateMachine` we can easily change states by simply accessing this field and changing it. This could also be achieved by using the observer pattern to send messages back to the statemachine, which increase the encapsulation since each state can't access and change the field. Here is an example of the `UpdateState` method in the `GameRunning` state, where the state is changed to the `GameOver` state because there aren't any lives left:

```
1   if(health <= 0){
2       instance = null;
3       stateHandler.ActiveState = GameOver.getOrSetInstance(stateHandler,pointsAwarded);
4   }
```

6.4 Accordance with goals for implementation:

Alot of the code did get centered around the `GameRunning` state which doesn't abide the first goal. This could have been achieved by utilizing the observer pattern more.
Much of the code did get allocated to the `facade` class, which made some of the code easier to read, and test since the we isolated much of the functionality in the `facade` class making it easier to test.

6.5 Refactoring:

Much of the refactoing has been centered around the `facade` class, to make the methods capable of spawning different entities. A lot of refactoring has also went into seperating code into their respective methods and making new methods in order to make the code more readable.

7 Guide

The game can be launched by going to the "Breakout" folder executing the following command from the "DIKUGames" folder:

```
cd Breakout
```

From here the game can be run using the following command:

```
dotnet run
```

To run the tests from the DIKUGames folder:

```
cd BreakoutTests
```

From here the game can be run using the following command:

```
dotnet test
```

8 Evaluation

The following diagram shows the test coverage for some of the most important modules of the game, modules like `Game.game` and `Program.StartingPoint` aren't included since they just responsible for launching the game-window and sending the key-events. The overall line-coverage is 88- and the branch-coverage is 87.8 percentage:

Module	Line-coverage	Branch-coverage
ASCIILoader.asciiLoader	97.3	90.7
BallTypes.normallBall.NormalBall	80.3	72.2
BallTypes.powerUpBall.PowerUpBall	92.7	87.5
BlockTypes.hardened.Hardened	100	100
BlockTypes.healing.Healing	87.5	66.6
BlockTypes.hungry.Hungry	94.1	75
BlockTypes.invinsible.Invinsible	83.3	100
BlockTypes.normalBlock.NormalBlock	81.2	100
BlockTypes.teleporting.Teleporting	95.7	95
BlockTypes.unbreakable.Unbreakable	88	100
Entities.Ball.ball	92.1	94.4
Entities.Block.block	91.6	0
Facade.facade	98.6	90
Player.player	81.1	83.3
States.GameOver.GameOver	93.9	100
States.GamePaused.GamePaused	82.1	88.4
States.GameRunning.GameRunning	89.2	90.2
States.MainMenu.MainMenu	83.6	91.6

We did achieve a fairly high code coverage but their were some of the functionalities of the different blocks and balls that were difficult to test since the functionality isn't that easy to isolate during unit testing. To reproduce the tests we created a game and sent different key-events to simulate user-interaction with the game, which was necessary in order to test many of the different modules of the game.

For some of the testing we also had to resort to making public properties within the classes which takes a strain on the encapsulation, which was something that we didn't account for in the design.

8.1 Unsolved issues:

We had problems incorporating the `Split` powerup with the observer pattern without breaking the game. We couldn't solve this problem since we didn't

quite know how to test it and recreate the problem in order to solve them. We didn't have time for it, but this could possibly be solved by making a new type of ball.

Furthermore we also had problems pausing the time on powerups when pausing the game. We tried using a lot of different techniques, but we could find one that worked and was in accordance with our design patterns. If given more time we could probably have fixed it by using the `StaticTimer` module from `DIKUArcade`.

9 Conclusion

In this report we have outlined the progress with our game of breakout and given the reader some insight into the decision-making process of this program.