Assignment #2

Week 4

Group 2 - Bubble Bobble

Remi Flinterman
4362950, remiflint@live.nl

Arthur Guijt
4377338, a.guijt@student.tudelft.nl

Laurens Kroesen
4350286, l.kroesen@student.tudelft.nl

Thomas Overklift
4080890, t.a.r.overkliftvaupelklein@student.tudelft.nl

Lars Stegman
4365801, l.s.stegman@student.tudelft.nl

TI2206 Software Engineering Methods
of the Computer Science curriculum
at the Delft University of Technology.

Supervisor: Dr. A. Bacchelli
Teaching Assistant: Aaron Ang

Table of Contents

Exercise 1 – 20-Time

We decided that we wanted to implement AI movement as a major improvement in our game this week. The enemies will try to chase the player taking semi-random routes.

1 - Requirements

1.1 Must Haves

- Enemies shall be able to move.

1.2 Should Haves

- Enraged enemies shall have unique images.
- Enemies shall be able to move in the direction of the player
- Enemies shall be able to update their direction if the player moves.
- Enemies shall be able to get faster when they're the last enemy left.

1.3 Could Haves

- Enemies shall have multiple kinds that have different speeds.
- Enemies shall have multiple kinds that have different movement patterns.

1.4 Won't Haves

- Enemies shall be self-learning.

2 – RDD, UML & explanation

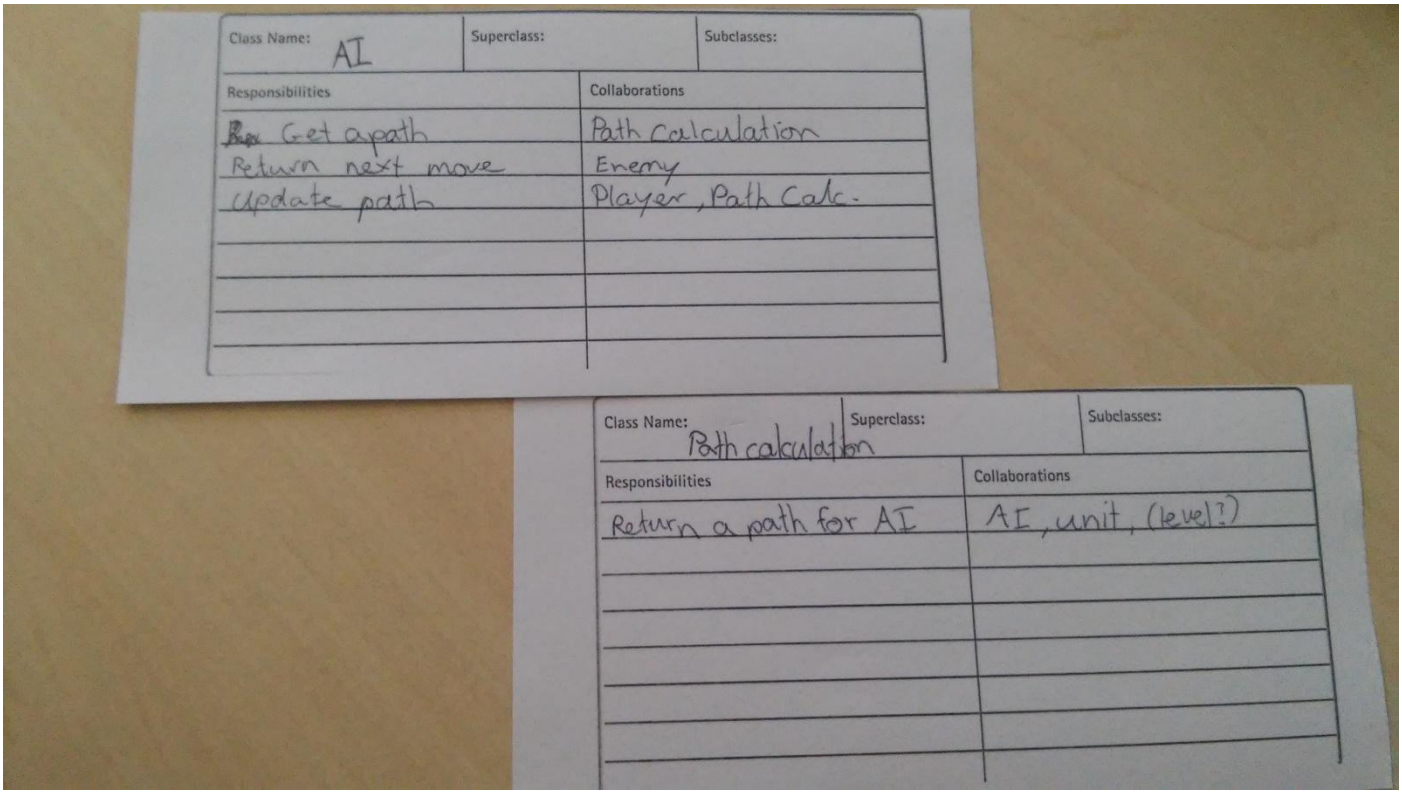

Figure 1 – First draft.

In *figure 1* you can see the CRC cards that we originally created. There is an AI class that governs for each enemy where it goes, it uses the class path calculation in the process.

During the implementation the class PathCalculation was expanded a bit. We split the movement in the x and the y direction into different methods to make it a little easier, we decided to return the different between the coordinates of the player and the enemy so the AI class could decide in what direction to move the enemy unit.

Additionally we later added a method in the AI class that randomizes movement in a percentage (currently 20%) of the cases. We also added extra logic that deals with falling of platforms. We thought it was necessary to implement this because the AI was performing poorly in the case that the player was below an enemy.

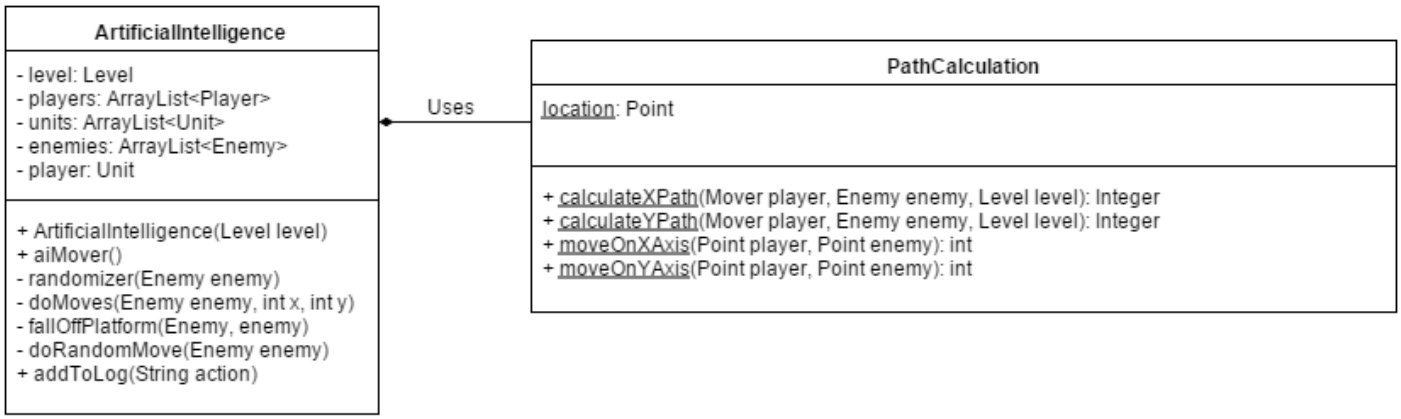

Figure 2 – Current situation.

In *figure 2* you can see the UML class diagram of the current situation within the AI package. We chose for the composition relation between AI and PathCalculation because PathCalculation has no use apart from feeding the AI paths to take.

Exercise 2 – Your wish is my command

Part A – Enemy Collisions

Our TA decided that we would implement the actions that occur when a player collides with an enemy under different circumstances.

1 – Requirements

1.1 Must Haves

- An enemy shall be able to be captured by an empty bubble.
- The player shall be able to die from touching an enemy.
- The player shall be able to pop a bubble that has collided with a wall or enemy.

1.2 Should Haves

- A bubble shall be able to float upwards after a wall collision.
- A bubble shall have an unique sprite when containing an enemy.

1.3 Could Haves

- The player shall be able to jump on a bubble
- A bubble shall be able to pop nearby bubbles when it pops.

1.4 Won't Haves

- None.

2 – RDD, UML & explanation

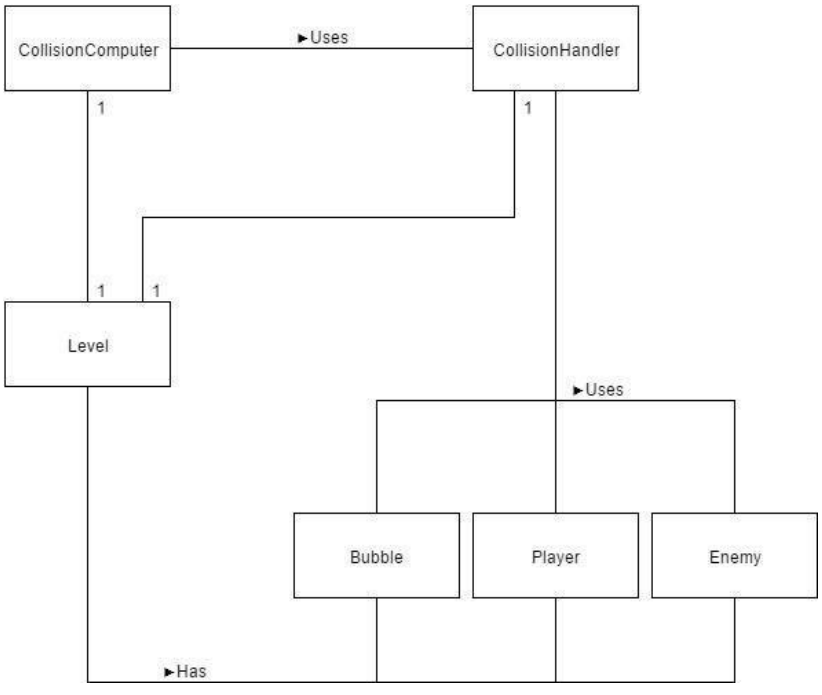

Figure 3: The new collision structure

Originally CollisionComputer and CollisionHandler were the same thing. However, this made testing both components harder or impossible, and gave this class too many responsibilities. Because of this, we split the class into two new ones as can be seen in *figure 3.*

The CollisionComputer computes the next position and finds collisions is now separate. The CollisionHandler takes care of the logic of a collision between two units. Apart from that, implementing these features was just a case of adding some code to the right methods. Thanks to a framework, this was easier than expected.

Part B – Unit Warping

Our TA decided that we would implement the warping of characters from the bottom of the level to the top, when a player falls through a gap, and the other way around when the player jumps through the top of the level.

1 – Requirements

1.1 Must Haves

- The player shall appear at the top of the level, when it goes through the bottom of the level.
- The player shall collide as if there is a wall when it tries to go through the left or the right of the level.
- An enemy shall appear at the top of the level, when it goes through the bottom of the level.
- An enemy shall collide as if there is a wall when it tries to go through the left or the right of the level.

1.2 Should Haves

- None.

1.3 Could Haves

- The player shall appear at the bottom of the level, when it jumps while on a platform at the top of the level, and the top and the bottom of the level are open.
- An enemy shall appear at the bottom of the level, when it jumps while on a platform at the top of the level, and the top and the bottom of the level are open.
- A bubble shall appear at the bottom of the level, when it moves to top of the level, and the top and the bottom of the level are open.

1.4 Won't Haves

- None.

2 – RDD, UML & explanation

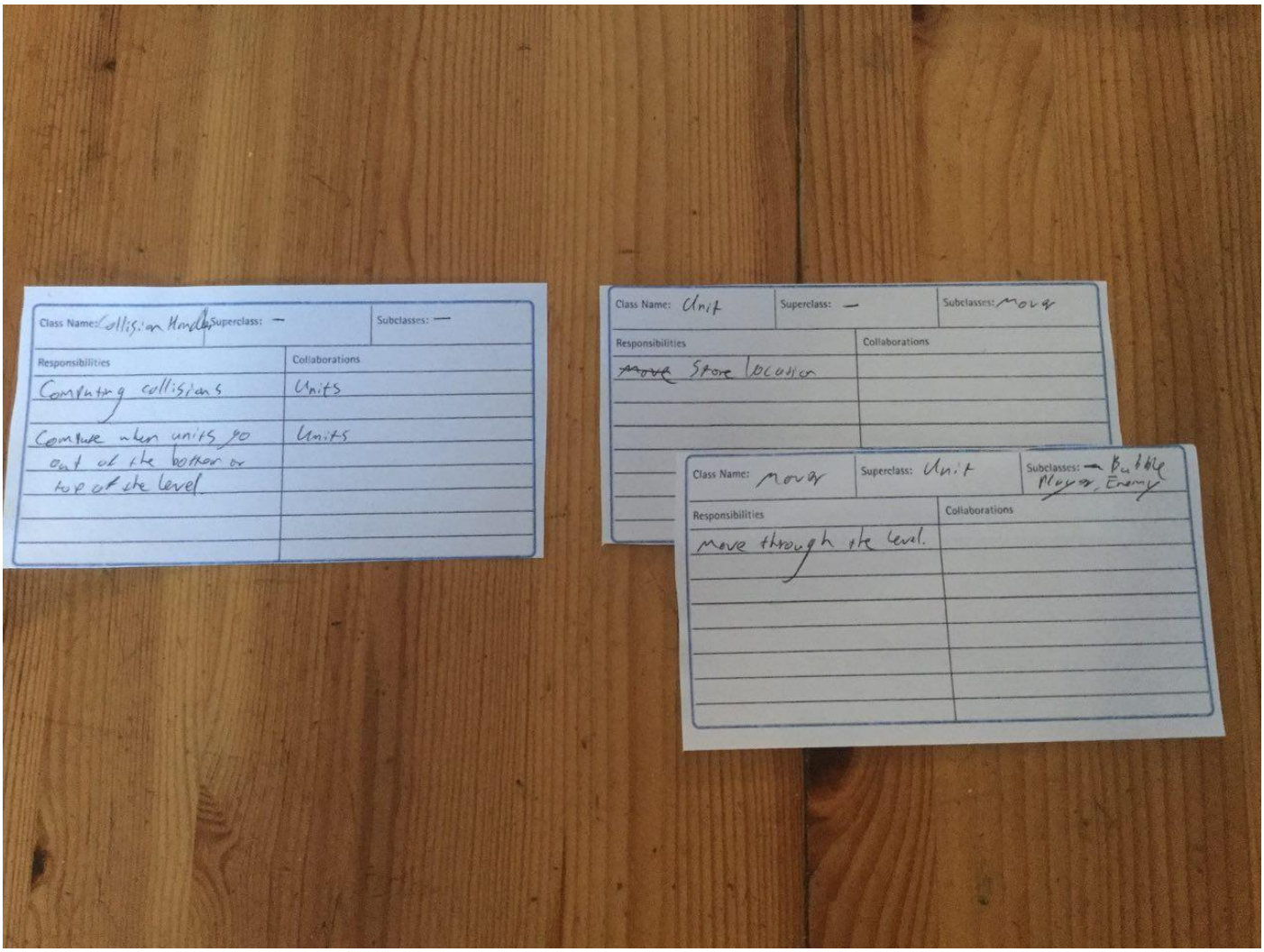

Figure 4: Warping CRC

To warp the Units through the bottom and top of the level, no additional classes are needed. The CollisionHandler already contains a lot of logic for collisions, thus adding the logic for collisions with the borders of the level to the handler seemed appropriate.

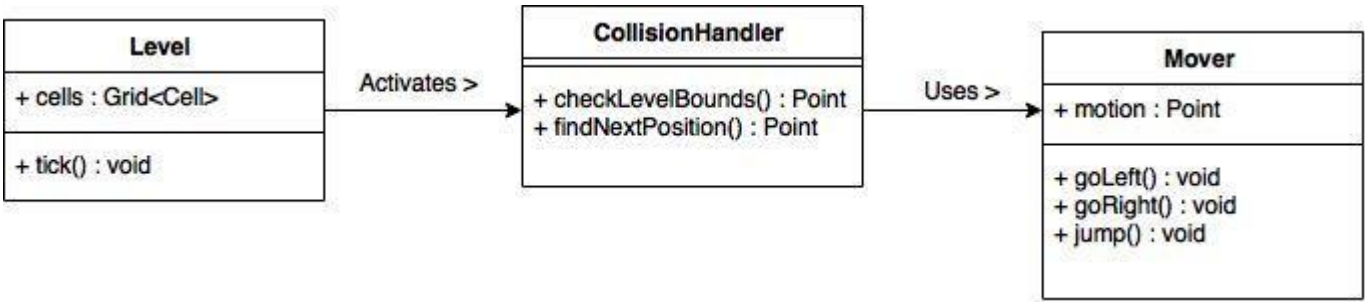

Figure 5: The classes involved with warping