CS4303 P1 - Artillery

150006108

September - October 2018

1 Overview

The aim of this practical was to create a game that utilises a basic physics engine to model projectile motion. The game should have two controllable tanks, with each tank being able to move across the terrain, aim, and fire a projectile. The projectile should move through the air in the desired direction, whilst also having it's trajectory being altered by gravity, air resistance, and potentially wind. The option to play against an AI opponent should also be given.

2 How To Play

I developed the game using the Processing Java Libraries, which gave me the main advantage of using the Intellij IDE rather than the Processing one. It also allowed me to build an executable Jar file containing the game. To play, simply use the command **java -jar P1_Artillery.jar**.

2.1 Controls

- ”a” - Move left
- ”d” - Move right
- ”space” - Switch phase
- ”mouse” - Click and drag to aim, let go to shoot
- ”p” - Toggle 1 or 2 player mode.
- ”r” - Start a new game when one player has won

3 Design and Implementation

3.1 The Map

The map is made of destructible blocks. These are simply rectangles that are drawn whenever the *draw* method is called. Each block has it's coordinates stored within a *MapBlock* object. The *GameMap* class stores an *ArrayList* of these. This is essentially the map. When the draw method is called, every MapBlock stored within the list is drawn.

3.1.1 Random Map Generation

The map is randomly generated. At the start of each round, (when a tank is hit, a new round begins), a new map is generated. Each column of blocks can be 1 - 8 blocks high (inclusive). Originally, the maximum was 5 blocks high, but I found it was too easy to win on your first shot by shooting straight over the terrain. By making the map higher, it's a lot harder to visualise the projectiles trajectory over the higher terrain, therefore allowing players to learn and adapt over multiple shots, allowing them to experience more of the game.
Each map consists of eight columns of terrain, with each column containing a minimum of one block. Each block has dimensions of 100 x 50 (same as the tanks). The play area itself is 1200 x 800. Each tank is positioned at opposite sides of the play area, and given one block length of space between the tank and the terrain. The map is generated, and the final product can be seen below.

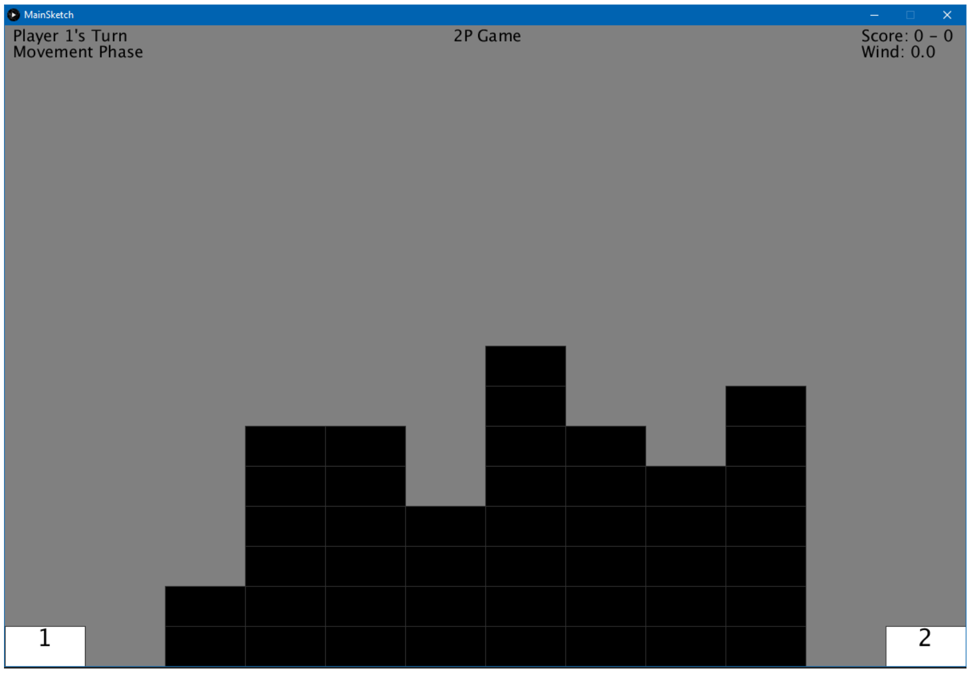

Figure 1: Start of a new game, showing starting positions for each tank and the randomly generated map.

3.1.2 Relative Forces

Due to the objects being quite large relative to the screen, the force values that are going to be used to alter the shells trajectory will have to be very small. When testing these values, I found a gravity value of 0.05, a dampening factor of 0.01 for air resistance, and an absolute value between 0 and 0.01 for wind seemed to prove feasible for the chosen environment, allowing the shell to follow what looks to be a realistic trajectory through the air.

Regarding the wind, I decided to introduce a fairly random element to it, to ensure the players weren't able to learn wind patterns. At first, a random number generator chooses a number between 0 and 4 inclusively. If that number is 0, there shall be no wind that turn. If that number is 1, the current wind shall not change. However, if the random number is either 2, 3 or 4, a new value of wind power shall be generated.

If it is decided that wind shall be present, a random float shall be generated between 0 and 1, which is then divided by 100 in order for it to scale to the large environment. Another random number is generated to simulate a coin flip. If 1, the wind value is multiplied by -1 to switch the direction of the wind. Finally, the new wind vector is instantiated. Wind is set to 0 at the start of each turn, to give the player who lost the last round a slight advantage.

3.1.3 Extension - Falling Terrain

As an extension, I decided to implement falling terrain. This means that when a supporting block is hit, all blocks above it shall fall. This includes the tanks. If a supporting block is removed whilst a tank is positioned above, it shall fall with the rest of the terrain.

3.2 Tanks

Tanks are white rectangles with the same dimensions as the map blocks. The tanks are labelled "1" and "2" with respect to the player who controls it.

3.2.1 Turn Phases

There are two phases to a players turn. A movement phase and a shooting phase. Both are pretty self explanatory. To switch between phases, the player can press *space*. However, once a player makes their shot, their turn is over, and more changes can be made. The current phase the player is in is displayed in the top left corner.

3.2.2 Movement Phase

Whilst in the movement phase, the player can reposition their tank. Movement occurs using an incremental system, with each increment being the width of the tank/a block in the map.

A tank can only move one place forward or backward each turn, and cannot leave the play area. I implemented this limit because without it the player who went first would be able to traverse the entire map until they were right next to their opponent and simply kill them in a single turn. This is neither fair nor fun.

Movement can be controlled using the "a" and "d" keys, which move the tank left and right respectively, and provided the tank isn't trying to move off the edge of the map or on top of the opponents tank.

The tanks traverse the terrain by simply moving to the top of whatever map column is next to them in the chosen direction of movement. It was because of this that I implemented the falling terrain, as before it was possible to have a tank be positioned on top of a floating block, and that was just a bit too surreal for my liking...

When moving across the terrain, the system inspects the game map and counts how many blocks are in the column to be traversed to. Using this number, the necessary elevation of the tank can be calculated, and the tanks coordinates can be altered in order to position it at the very top of the column.

Collision detection is used if the player attempts to move outside the map or onto a column occupied by another player. The players current column number is inspected, and if the desired direction would result in an erroneous case, the movement simply does not occur.

3.2.3 Shooting Phase

Only in the shooting phase can the player align and make their shot. I wanted the shooting mechanism to be as simple as possible, and I am very happy with the results.

Essentially, the player *draws* the shells intended trajectory. By simply clicking and dragging, they are able to adjust the angle and power of the shot, with a line acting as a visual aid.

The start of the line is anchored to the centre of the tank, as this is where the shell is fired from. However, the end of the line is anchored to the mouse. So long as the player holds down the mouse button, they are able to quickly and easily see and adjust their trajectory.

Upon release of the mouse button, the shell is fired, following the chosen path. Once the shell collides with something (either map block, edge or player), the firing player's turn is over, and the next player is placed into their movement phase.

I feel this method of shooting is an improvement to just typing in the desired elevation and power of the shot. Using this method, not only do you get a useful visual aid to actually *see* where your shot will go, you can fire with a much higher degree of accuracy, with the aiming using the mouse position rather than fixed integer values. It allows the player to actually get a *feel* for the shot, as if they are throwing the shell themselves, rather than just entering some numbers and clicking fire.

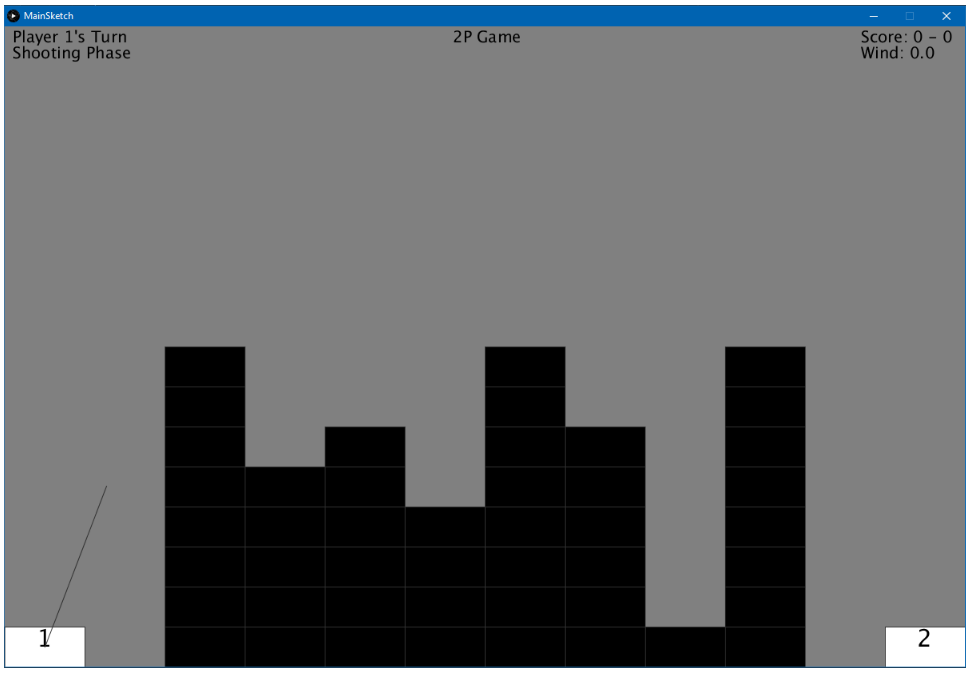

Figure 2: Image of a player making a shot. The start of the line is anchored to the centre of the tank, the end is where the users mouse is currently positioned. The line represents the path the shell would follow without being subject to other forces.

3.3 The Shell

The shell is a small ellipse, with a radius value of 16. When launched, it appears at the centre of the firing tank, and moves outwards along the specified path.

3.3.1 Movement

The location of the shell is stored as the vector *loc*. The vector drawn by the player when aiming is stored as the vector *dir*. The accumulated forces as stored as the vector *acc*.

There are three forces we are concerned with. Gravity, wind and friction - in the form of air resistance. Gravity is stored as a vector with a constant y value and an x value of 0. Wind is the opposite, with a y value of 0 and a constant x value throughout the turn, although the x value will potentially change next turn.

Air resistance is dependant on what direction the shell is moving. The air resistance must always be pushing in the opposite direction at a constant rate. This meant the value of the vector was to be dependant on the direction the shell was moving at that moment. For example, if it was falling to the left, air resistance should be pushing it up and to the right.

Gravity and wind are slightly different, as they *accelerate* the shell, rather than just dampen it's motion. This means that rather than simply adding the gravity and wind vectors to the direction vector, we must instead add them to all the previous gravity and wind vectors that have been used, and then add *that* vector to the the direction vector. This is what the vector *acc* is for. It is used to accumulate the acceleration of gravity and wind.

Once all the forces are accumulated, *acc* is added to *dir*, as well as the correct air resistance vector. This resultant vector is then normalised, and multiplied by 10 before being added to the location vector, moving the shell 10 units along the new trajectory. The shell is then drawn in it's new position. This method gives the shell a smooth, realistic, parabolic path through the environment.

3.3.2 Collision Detection

Each time the shell's location is updated, it's position is compared with all over objects in the map. If the new coordinates are found to cross any specified borders, it counts as a collision, and the firing players turn is ended.

Each map block is checked, and if the shells coordinates are found to be inside a block, that block is destroyed. This means the block is removed from the list of map blocks. Before this happens though, the falling terrain is implemented, wherein any blocks found to be above the block to be removed have their y value incremented by the height of a block, thereby moving them all down by one space.

If no blocks have been hit, the map borders are then checked. If the shells x coordinate is bigger or smaller than the screen edges, a collision has occurred. The same happens if the shells y value is larger than screen. This means the shell has hot the ground. However, if the shell's y value is less than 0, it is allowed, as this means the shell is high in the sky, and will eventually fall back down into view.

If no edges have been hit either, friendly fire is checked. This was slightly more complicated, as the shell originally appears within the firing tank. If we were simply checking if the shell has entered the tanks area, a collision would be triggered the moment a tank fires. To fix this, it was required to store when the shell has left the tanks area after being fired. If this has happened, and then the shell hits the firing tank, the tank has hit itself.

The final check is if the opponents tank has been hit. This is simply checking if the coordinates of the shell have intruded inside the opponent tanks area. If either tanks are hit, the winning players score is incremented, and a new map is generated.

When a player reaches a score of 3 wins, a win screen is displayed, telling the players who won the game, and prompting them to press "r" to reset the game and start again.

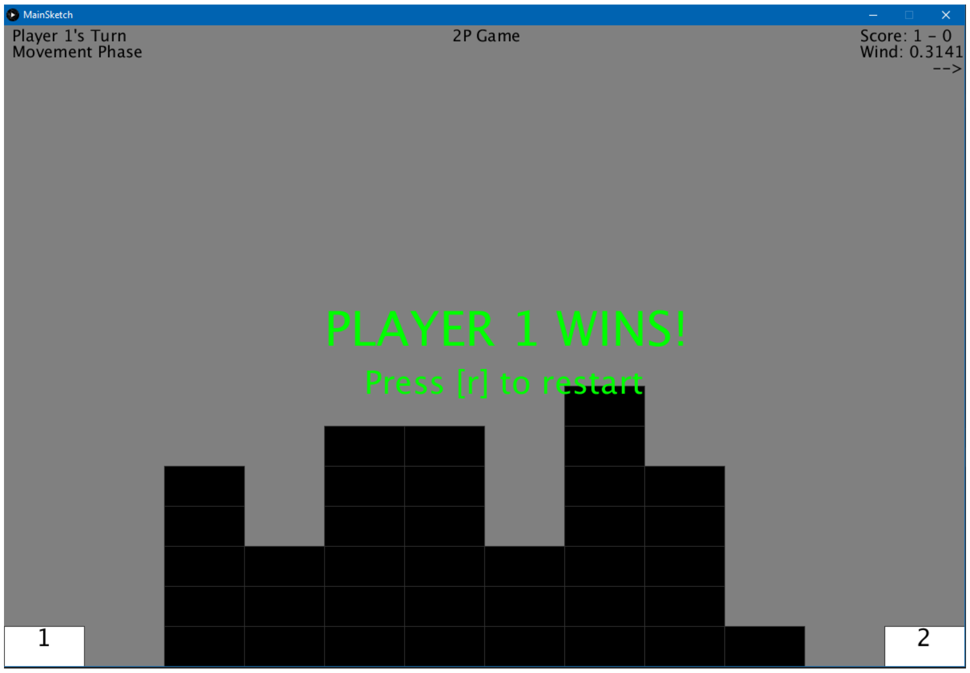

Figure 3: Message displayed after player 1 has won. N.B. Set winning score to 1 for testing purposes.

3.4 AI

Single player mode can be accessed by pressing the ”p” key at any point. This will create a new game, with tank 2 being controlled by the AI. The player can tell whether or not they are in 1 or 2 player mode by looking at the top centre of the screen. If a player wishes to switch back to 2 player, the ”p” key will toggle them back.

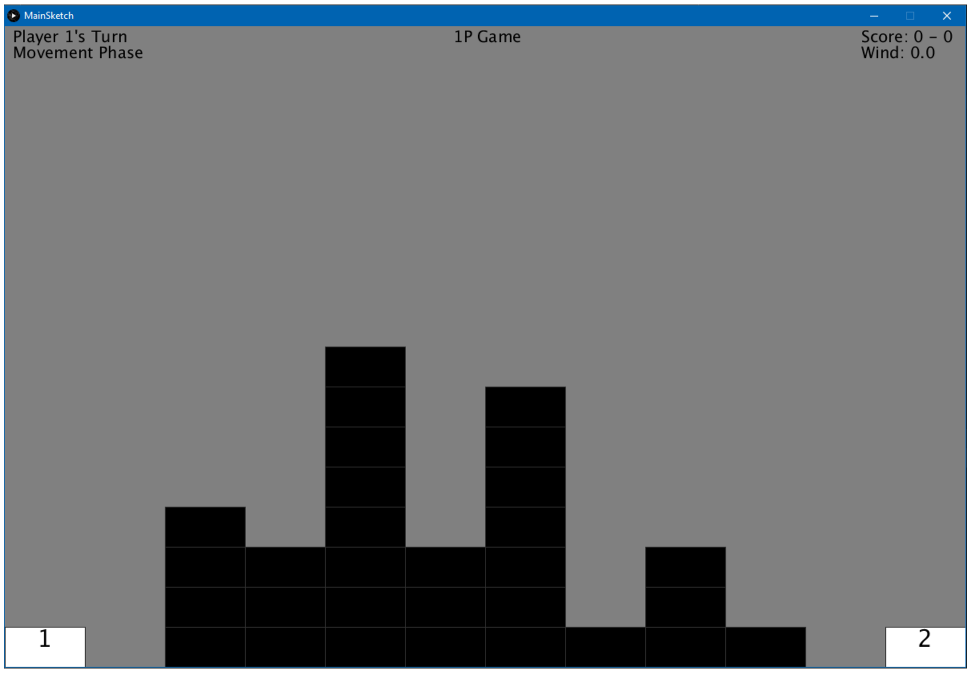

Figure 4: A game in single player mode. Notice the top has changed from saying ”2P Game” to ”1P Game”.

3.4.1 AI Shooting

The way the AI makes a shot is pretty straightforward. The AI starts shooting at an angle of 255 degrees - this is simply what I found to be a sufficient starting angle for most map generations - and a power of 100. Using these two starting values, we can have the AI make a shot.

To create the directional vector for the AI, I used a method called *PVector.fromAngle*, which takes an angle in radians and creates a vector with a fixed magnitude. I convert the AI's current angle to radians, then pass it to this method to retrieve the directional vector. Unfortunately, the fixed magnitude of this vector is way too small to make usable shot.

To fix this, I simply normalise the vector, and then multiply it by the AI's power attribute. This new vector can then be used by the physics engine as the starting directional vector, and the shot can be processed like any other players.

Upon a collision detection, the AI compares the final location of the shell with the current position of the opponents tank. If the shot didn't reach far enough, the angle is lowered by 2 and the power is increased by 20%. If the shot has gone too far, the angle is raised by 2 and the power is decreased by 20%.

Altering the power by a percentage of the previous value allows for the AI to target a much wider range of locations than if it simply increased or decreased the power by a fixed amount each turn based on the previous shots outcome.

3.4.2 AI Movement

Not only did I want the AI to learn and adjust it's shooting trajectory based on the outcome of the previous shot, I also wanted the AI to react to the opponents shots.

In response to this, if the opponent makes a shot that lands within a certain radius of the AI's tank, the tank shall adjust it's position.

To be more specific, the "warning radius" is set at 200. If a shell falls within the radius, the tank shall attempt to distance itself from the impact site. If the shell lands to the left of the tank, it shall move right - unless it on the edge of the map, in which case it will move left. If the shell lands within the radius to the right of the tank, it shall move left.

4 Evaluation and Conclusion

To conclude, I feel I have met every aspect of the specification, and even extending it. The game contains a map that is randomly generated with destructible blocks that fall is supporting blocks beneath them are destroyed. There are 2 tanks that can move across the terrain without driving through blocks or outside the map, as well as being able to aim and fire a projectile with a desired power and elevation that is affected by gravity, wind and air resistance, causing it to follow a parabolic trajectory through the air. If a tank is hit by the shell, a point is awarded to the winning player, a new map is generated, and another round can begin. Once a player reaches a score of 3, the game ends, and the option to start a new game is given. There is also the option to play against an AI opponent, which can shoot a shell and adjust it's parameters based on the previous shot's outcome, as well as being able to react to it's opponents shot, moving away from an impact site if it falls too close to the AI's tank.