# Artillery

## Overview

The assignment asks to write an artillery game. The goal is to practice implementing real world physics in video games. I have written my game in Java, using the Processing library. I have completed all basic requirements as described in the practical specification. Additionally, I have implemented three extensions. One is a ‘pre-game’ menu that offers a couple of gameplay options, another is affecting the terrain with gravity, and the last one is friction with the ground when tanks are moving.

I have included a Makefile for easy compilation and running. Simply execute ‘make’ to run the game. This command will also compile the game if that hasn’t been done. Use: ‘make compile’ to compile only; ‘make clean’ to remove all created files; ‘make run’ is the same as ‘make’.

## Design

The following subsections go over the design of each aspect of the game. They are ordered in the same ways as in the practical specification. The last subsection describes how the defaults can be changed without re-compiling the code.

### Play Area

The play area, by default, spawns on the whole screen. It is made entirely of blocks (by default 5 pixels wide). Tanks, by default, are 50 pixels wide, and thus stay on top of multiple columns of blocks. Several images are provided in the submission. They are used for the background, blocks, tank, shell, and explosion. Each of these also has a default shape which is used when no image is provided. Images for the tanks, shell, and blocks are scaled down to the default size. The tanks can optionally have a barrel which moves to reflect the current elevation. The colour and dimensions of each tank and barrel is customizable.

The landscape is generated at random every time. It uses a midpoint displacement algorithm with customizable ‘roughness’ and ‘displacement’ values (default to 30 percent displacement, and 70 percent roughness). These define how tall column blocks can get and how much difference there can be between two adjacent ones. Further details are provided in the ‘Implementation’ section.

All accelerations are measured in meters per second squared. Whenever an acceleration is applied to an object, the value gets multiplied to the ratio between pixels and meters. That is, by default, 10. Gravity is 9.8m/s$^2$ and wind speed varies, by default, between 0 and 40 m/s. I have referred to the Beaufort scale when implementing wind, and I adapted this formula to calculate the force the wind exerts on objects. The wind has random velocity. By default, there is 1% chance that the wind velocity will change. When that happens, both direction and speed can become anything (within range).

## Controls

To control a tank, a player uses both the keyboard and the mouse (by default). Both players use the same controls (by default). The default controls are: W to increase elevation; S to decrease elevation; A – move left; D – move right; Left mouse button – increase strength; Right mouse button – decrease strength; Space – shoot. The controls are also available from the ‘controls’ screen.

The elevation angle is displayed in degrees starting from 0 (pointing right) goes through 90 (pointing up) and ends at 180 degrees (pointing left). When player 1 increases elevation, the angle is increased, causing the aim to go upwards from the starting elevation. When player 2 increases elevation, the angle is decreased, causing the aim to go upwards from the starting elevation. Player 1 aims from left to right, and player 2 – from right to left. It appears impossible to have the reverse situation but the code can handle that if needed. Since player 1 aims to the right, the button to increase elevation moves the aim up from 0 to 90 degrees. Since player 2 aims to the left, the button to increase elevation moves the aim up from 180 to 90 degrees. In other words, while the elevation numbers change differently, the effect is the same (aim upwards).

When pressing a button to move the tank, a force is applied to tank, causing it to move in that direction. Simply releasing the button will leave the tank to inertia. I have implemented friction with the ground and air drag which will cause the tank to slow to stop. Alternatively, the player can press a button to move in the opposite direction and apply additional opposing force. The weight of the tanks and the strength of force applied by buttons are by default 60 000kg and 120 000N. This type of movement results in smooth motion and more difficulty compared to regular movement. It will have an effect on players, as they will be required to think more about positioning themselves before shooting. It also opens up the possibility to move after shooting, as inertia will keep the tank in motion, even when its not that player’s turn. This can inspire defensive strategies to shoot and then take cover.

I have implemented collision detection for the tanks and the shell. If a tank hits the edge of the screen, or a high column of blocks, or the other tank, it will ‘bounce’ off – it’s velocity will be reversed and the speed will be multiplied by a constant impulse factor (0.9 by default).

## Shell

The shell is a, by default, a green ball with size that just fits in a 10 by 10 pixels square. If an image is provided, it will be scaled down to 10 pixels. The shell has a mass of 25kg (by default). This gets used when calculating the acceleration due to wind force. When the Shell collides with a tank or with a block, it explodes, destroying nearby objects. The explosion expands over the course of 2 seconds at a rate of 25 pixels per second and is red (all are defaults).

## User Interface

The UI is made up of four ‘screens’. Each displays different information to the players and provides different functionality. All elements mentioned in the following four subsections take up a percentage from the screen (values can be adjusted). All text has customizable sizes and locations on screen relative to screen size.

### Menu

The menu screen provides three clickable options for each tank, as well as 2 buttons that take the player to other screens. The ‘Play’ button starts the game and switches to the ‘Game’ screen. The ‘Controls’ button switches to the ‘Controls’ screen. The clickable options are: AI – whether this player should be operated by the AI; Barrel – whether the barrel for the tank should be displayed; Image – whether the provided image should be used (or the default shape).

### Controls

This screen displays the default buttons used to control the tank. This will not reflect any changes from the defaults. The screen also contains a ‘Back to Menu’ button which returns the players to the Menu screen.

### Game

This is the screen that is displayed while playing. Whose turn it is displayed at the top left of the screen (by default). The wind speed and direction are displayed directly below (by default). The score, elevation, and strength for each player is displayed at the top left and right below the wind parameters (by default). At the bottom right (by default) is a button that returns to the Menu screen. Any progress is lost when that button is clicked.

### Victory

This screen displays the winner and the score for each player. It is displayed when a player scores the required number of hits on the opponent. An ‘OK’ button sends the player back to the Menu screen where a new game can be started.

## AI

The AI operates by taking the location of its opponent, the landing location of its own previous shot, and obstacles directly in front. If there is no previous shot, it will simply make a shot at the starting elevation and strength. The AI is aggressive. It tries to make a clear straight shot at the opponent. It will move forwards unless there is an obstacle (in which case it will move away from it); increase strength if the opponent is further from the previous shot (or decrease strength if the opponent is closer); increase elevation if the previous shot landed while the shell was going upwards (has hit an obstacle); it will decrease elevation otherwise in an attempt to make a straight forward shot. The AI has a time limit of 2.5 seconds (by default) to make a move. After that it shoots. The time limit prevents it from getting stuck in narrow spaces between blocks. The AI is not protective at all. It might go into a place where the opponent’s shot last landed (making it an easy target). On a couple of occasions it has moved into its own explosion. The adjustments, however, will often allow it to hit a still target in about 5 turns.

## Configuration

Three files stored in assets/config determine most of the constants across the game. Every parameter mentioned above that has a default value is stored in one of those files. One can easily open up and edit the props in a text editor. There are also multiple properties that are not mentioned above. Those include but are not limited to AI aim adjustments, colours, checkbox sizes, air density.

The controls have the values of Processing keycodes. In theory, if a mouse button and a keyboard button have the same code, they will have the same effect. I have not found such a pair.

The width and height of the window can be set to integers or to the strings ‘displayWidth’ and ‘displayHeight’.

I recommend seeing how the game works with various block widths (default 5), and pixels to meters rations (default 10).

## Implementation

I have split my code in four packages – main, objects, screen, ai. The package ‘util’ contains java class files that I am re-using from previous practicals. They make programming easier in my opinion.

### Main

This package contains the main files, necessary for running the game. The Run class simply starts invokes the Processing interface to start the program. The ScreenManager is the class that behaves like a main Processing file. It first calls the settings method similar to ‘setup’ in Processing. It defines the window size and initializes a static field to reference the object that is running the application (called ‘instance’). This object is used by many classes to access Processing functionality and draw to the screen. The ScreenManager also contains a wrapper for the default ‘loadImage’ so that images can be loaded by referring to their properties (rather than file paths).

The Area class describes the play area and is entirely comprised of static classes (there can only be one area at a time). It contains methods to create the terrain, pull objects to the ground (gravity), detect shell collision with the ground, and destroy blocks caught in explosions. It makes use of LinkedHashSet when storing Blocks. This avoids repeating elements while preserving insertion order. The order is useful because blocks high above the ground are inserted first and thus they get iterated first when determining a collision.
When creating the terrain, I only borrowed the idea for the generation via midpoint displacement. I did not use the code there because it had unnecessary complexity. My implementation is easy to read and maintain. It uses a recursive function that gets called on each fraction of the line. Here are some details on the recursive call. Start with a left end, a right end, and a max displacement. Calculate the mean height between the two heights. Multiply the displacement by a random value between -1 and 1 and add it to the mean. Assign the addition result as the height of the point in the middle between left and right. Make two recursive calls – one for each side of the middle point. The left side has ends left and middle, the right side – middle and right. Reduce the displacement by the roughness. The bottom of the recursion is when the passed left and right end are adjacent. The recursion starts with the two ends of the screen.

### Objects

This package contains classes the instances of which need to have physics applied to them.

PhysicalObject is an abstract class similar to ‘Particle’ from the lectures. It has width, height, position, velocity, mass, and two boolean status (moving and falling). All of these are encapsulated with setters and getters. Instance methods allow: accelerations to be applied to the object;

intersections with other PhysicalObjects to be detected; and to move the object according to its velocity. It is expected that methods are called within one frame. The velocity is in pixels per second. When moving, the position is changed by a portion of that, according to the current frame rate. Similarly, when applying an acceleration, the frame rate determines the net change. This way, if the game is running at 30 frames per second, a method that is called 30 times will apply one 30th of the acceleration every time, resulting in a total of 1 application over 1 second (as desired).
This class also contains several static methods that take care of creating gravity and wind.

The Block class describes a single block that gets displayed on the screen. It inherits all the properties of PhysicalObject. It keeps a static reference to the image and default shape for the blocks, so that they only need to be created once (not every time a block is drawn). The methods specify how to draw a block on the screen and how to compare them.

The Player class describes player tanks and the actions they can take. Players are created via a static 'createPlayer' method that determines what constructor to call depending on whether a tank is operated by AI or by a human on the keyboard.
Each player instance keeps track of which buttons are pressed and reacts to them. A hash set contains all the pressed button and a hash map contains mapping between key codes and methods to run. A 'react' method is called once per frame, causing the tank to react to the buttons. Most methods are short, simply changing elevation and strength as necessary or drawing the tank image/shape on the screen. The two methods of interest are the override of 'halt' and 'move' (originally inherited from PhysicalObject). Move now checks for collisions with obstacles and calls the super methods to apply external forces such as wind and friction. Halt now first invokes the super halt method and then applies and saved velocity and starts moving again. Here is why. Bumping and moving back is implemented by reversing the direction, calling the overridden halt. Subsequent calls to move will move the tank away from the obstacle. Some of these calls are done immediately after bump to avoid a tank getting stuck inside an obstacle.

The last class in this package is the Shell class. It contains only static methods and a reference to a single instance of this class. The instance is needed to make use of the methods inherited from PhysicalObject. Again, a lot of methods take care of drawing the shell on the screen only when necessary. There are variables to determine what should be on the screen – the shell, the explosion, or neither. The remaing methods change these variables when a shot has been fired, when the shell intersects a block or a tank, and when the explosion expires. The 'moveS' method takes care of moving the shell and applying forces (gravity, wind) to it.

## Screen

This package contains multiple classes, each describing a displayable screen. There is a common abstract class providing common functionality. The package also contains an enum used by ScreenManager to determine which is the current active screen. There is a subpackage caled 'components' which contains three classes and an interface.

### Components

The Label class describes a piece of text that has a fixed position on the screen. It contains a string text, coordinates x and y, and dimensions width and heigh. It only has a constructor and a draw method. The class has a static constant highlight colour which is used by the other two classes.

The Clickable interface specifies three methods that anything clickable on the screen should have. These are render, click, and within. They are used by the AbstractScreen class. Render is used to display the component. Click is called when the component is clicked. Within takes a pair of coordinates and returns whether or not they are within the component (usually called with the coordinates of the mouse).

The Button class extends Label class and implements the Clickable interface. It keeps a LinkedHashSet of actions that need to be triggered on click. It provides a method to add such actions. Thanks to the LinkedHashSet, they are executed in insertion order.

The Checkbox implements the Clickable interface and contains a field of type Label. When rendered, it draws the label and a box next to it, that shows whether the box is checked.

### Screens

The AbstractScreen contains a set of Clickables and a set Labels that should be displayed on the screen. The draw methods set the background and draw all the contents of the two sets. The Menu, Controls, and Victory classes simply provide constructors that fill the two sets with Buttons, Labels, and Checkboxes. The ScreenManager instance then invokes the draw methods inherited from the AbstractScreen class.

The Game screen overrides manages the players, providing methods to score points, change turn, determine whether input should be accepted, and moving them. This class overrides the inherited draw methods, so that the players and area are also drawn on the screen. When instantiated, it creates new player instances, and generates a new terrain.

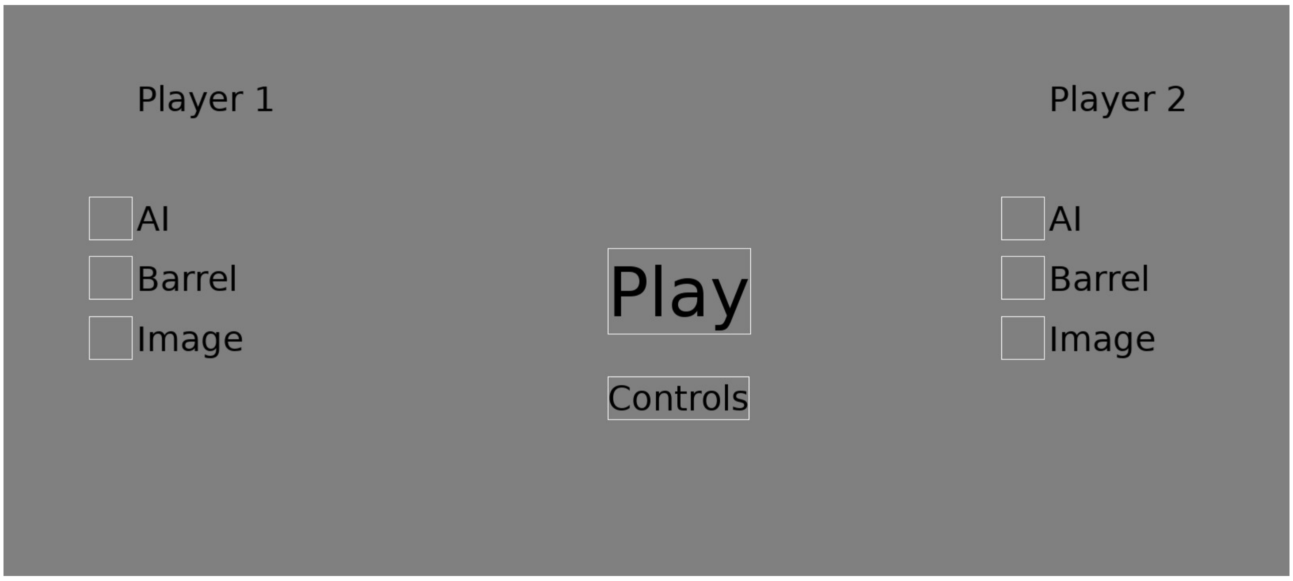
## AI

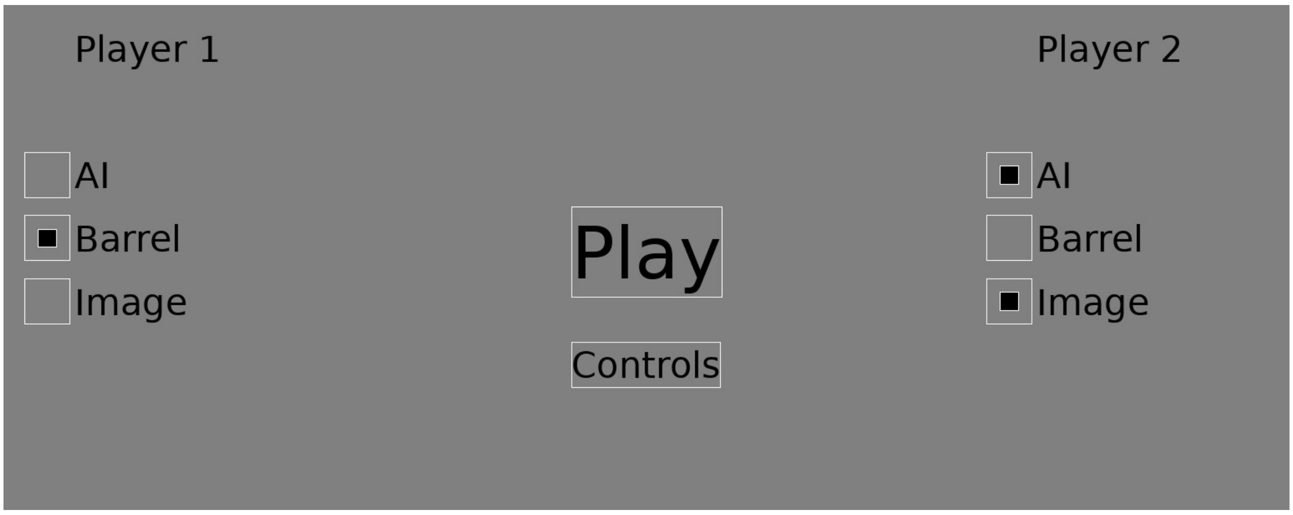
This package contains two classes needed to implement the AI for this game. An instance of PreviousShot stores the location and angle at which the last shot by the AI landed. It has methods to determine whether the shot hit an obstacle, whether it was overshot, or undershot.

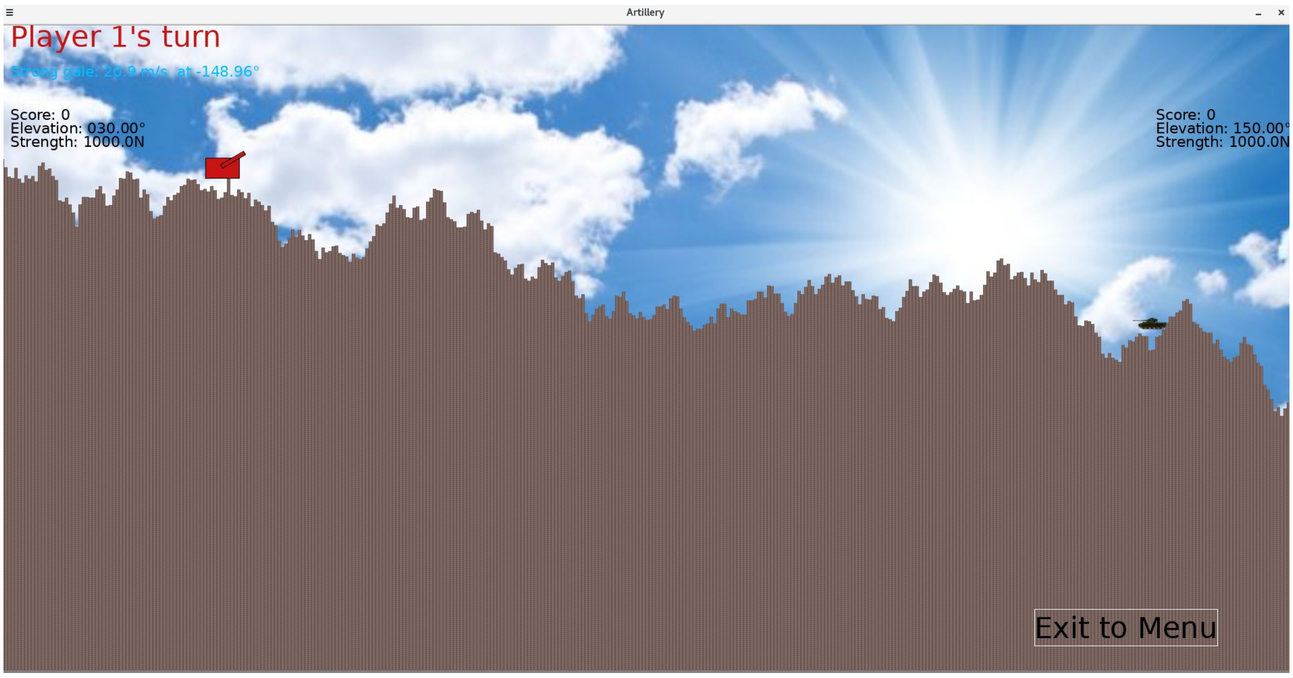
The AI class extends the Player class. Effectively, the game does not need to make a difference between AI and Player. This only happens when a button is pressed, to prevent the player from disturbing the AI. The ‘react’ method is overridden to first ‘think’, then call the super react, then shoot if there is nothing to react to. The think method sets target values for the elevation and strength that need to be reached before a shot is fired. It also causes the AI to move for up to 2.5 seconds (by default). All actions are done by simulating button presses. When the target values are reached, the buttons are ‘released’ and then a shot can be fired.

## Appendix

Here are a few screenshots of my working game.





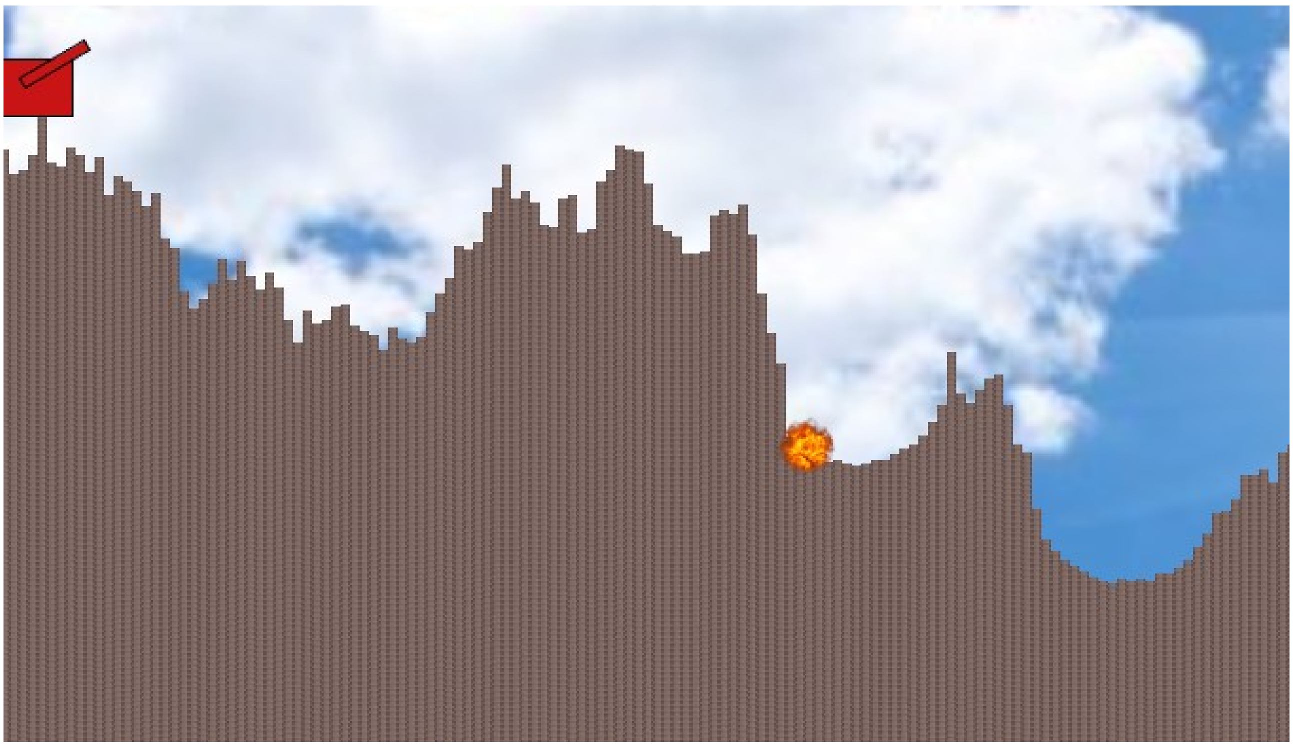

Artillery

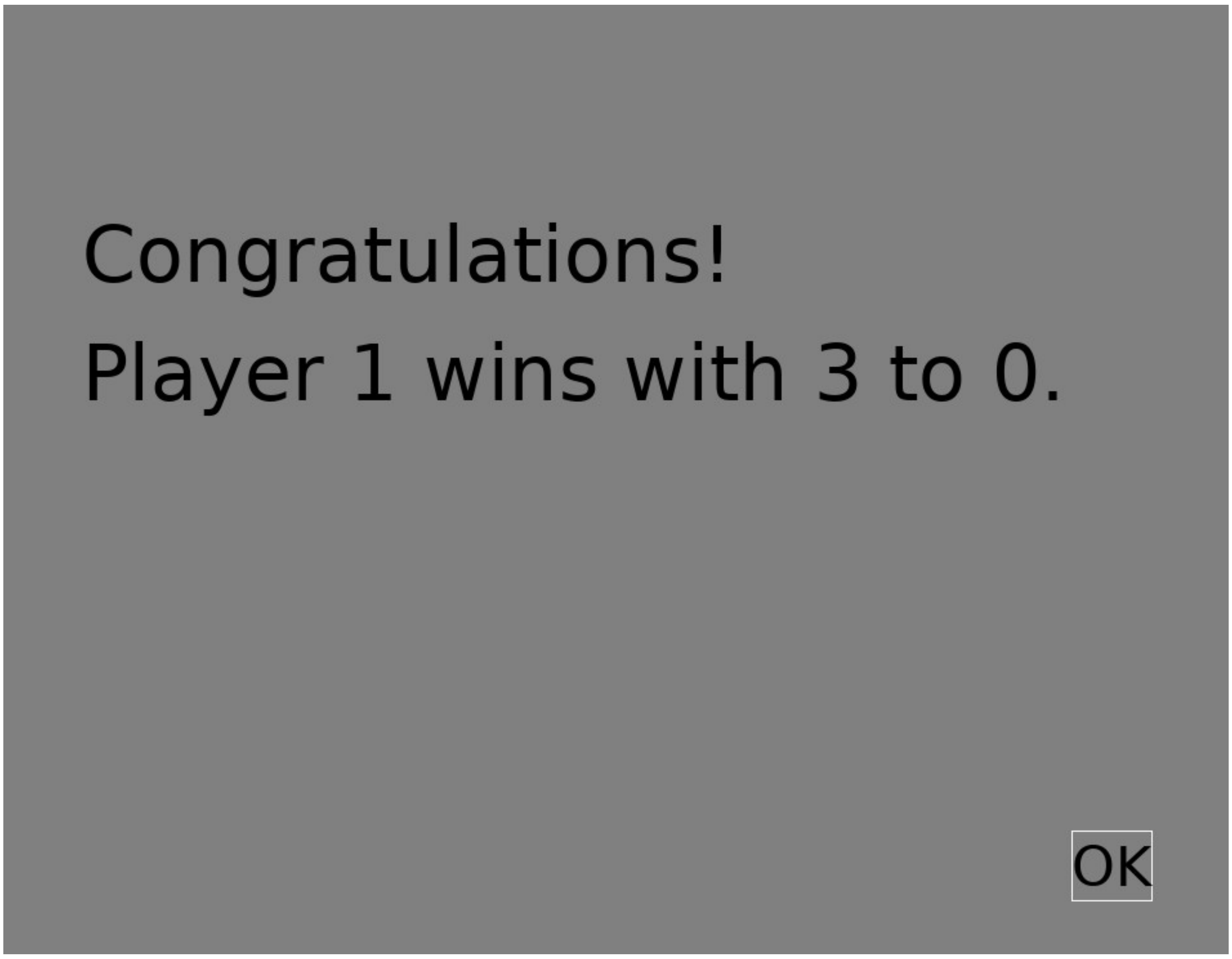
Player 1's turn

Strong gale: 20.9 m/s at -148.96°

Score: 0
Elevation: 030.00°
Strength: 1000.0N

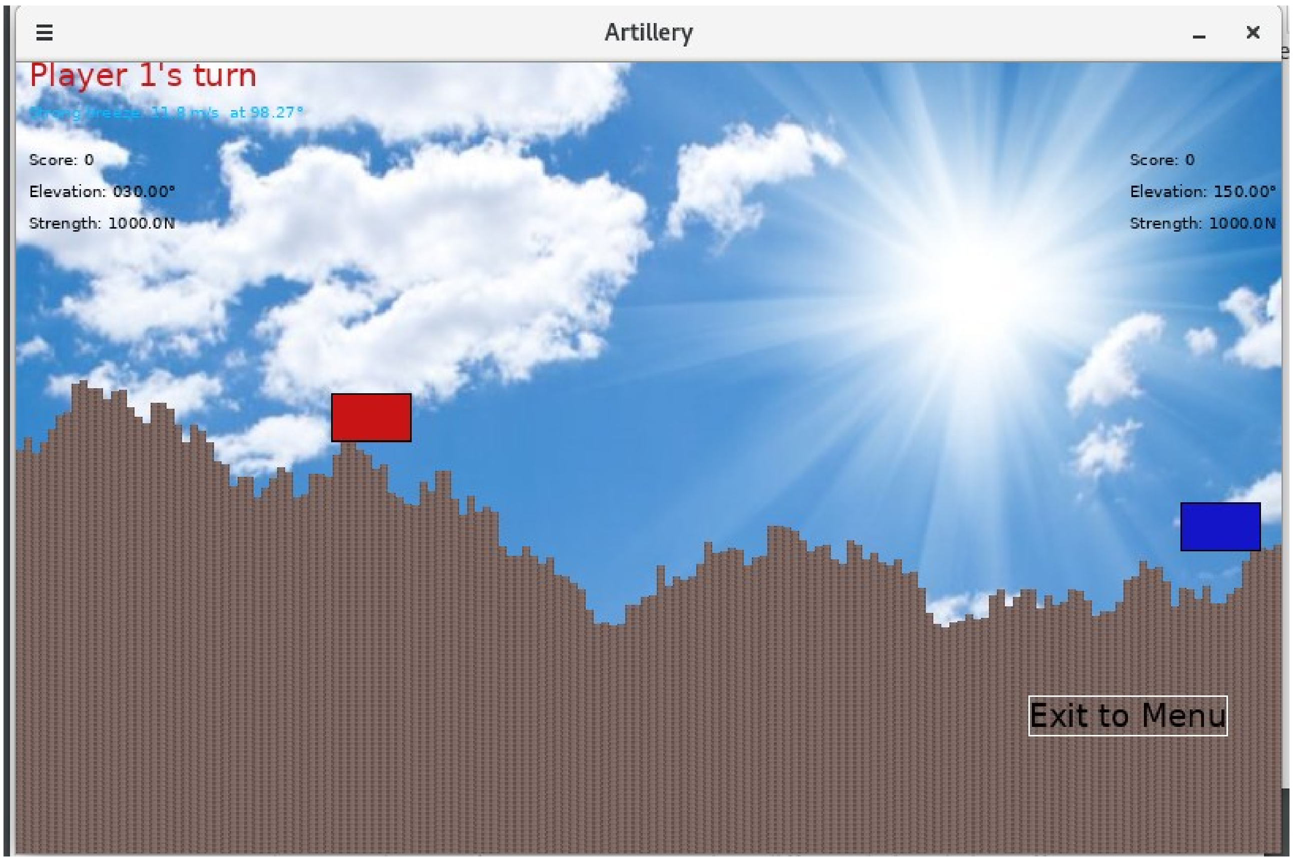
Score: 0
Elevation: 150.00°
Strength: 1000.0N





Congratulations!

Player 1 wins with 3 to 0.



Player 1's turn

Strong breeze 11.8 m/s at 98.27°

Score: 0

Elevation: 030.00°

Strength: 1000.0N

Score: 0

Elevation: 150.00°

Strength: 1000.0N

## References

All resources were accessed on the 3rd of October 2018 to confirm they work prior to submission.

[1] Beaufort scale for wind
https://en.wikipedia.org/wiki/Beaufort_scale

[2] Wind speed needed to move a car
https://www.quora.com/What-wind-speed-is-needed-for-wind-to-move-a-stationary-car

[3] Tank dimensions
https://science.howstuffworks.com/m1-tank2.htm
https://en.wikipedia.org/wiki/M1_Abrams

[4] Midpoint displacement algorithm
http://www.somethinghitme.com/2013/11/11/simple-2d-terrain-with-midpoint-displacement/

[5] Background image
http://interaztv.com/society/264029

[6] Tank image
http://www.tanks-encyclopedia.com/cold-war/ireland/a34-comet-irish-service/

[7] Shell image
https://fallout.wikia.com/wiki/Cannonball

[8] Explosion image
http://www.stickpng.com/img/nature/fire/explosion-circle