Overview:

In this practical, we were assigned to create a Tank game as a variant of the Artillery Game – a turn based game in which two players are trying to eliminate each other by firing shots. There were to be two tanks on either side of the screen which could move left and right, and fire a shot. In between them, there would be a landscape made out of rectangular blocks, and the shot fired would be affected by gravity, drag and wind. If a player is hit by a shot, they are eliminated and the other player's score goes up. In addition, there would have to be collision detection to prevent tanks from going through each other or through blocks. Finally, an AI option was to be made available to the user so that they could decide to play in single player.

Design:

To begin with, I used a Button class to be able to easily create buttons on the screen. A button is initialised by being given its position on the screen, its size and the text that should be in it. The button's colour changes when the mouse hovers over it to give a nicer UI, and there is a method to check whether the mouse is over the button to handle when the button is clicked in the main game class. This class is used for the user to select single or two player, to select the AI's difficulty (extension – see more further down), and finally to go back to the main menu when the game is over.

In terms of the play area, I decided to have randomly generated blocks between the two players, which change at each turn (when someone has been shot). I decided this because I believe it makes the game more fun if the landscape is always changing, and different landscapes can favour different players. If the landscape was always the same, each player could identify common ways to win or be stronger, and the game could get tedious. With the randomly generated landscape, players can find themselves in tricky and bizarre situations that will make them think differently about their approach for each turn. Therefore, the row each block is placed in is decided randomly, and then that block is simply placed on top of the previous block in that column, so that it is more realistic and there weren't any blocks seemingly flying/levitating. I used a 2D array to store the blocks, with a predefined number of columns, rows and number of blocks. I also created a Block class to represent each block, to be able to access each block's x and y coordinates, as well as their row and column in the 2D array.

To represent the tanks, I used a Tank class, which is initialised with the coordinates of the tank, its size, and gun angle (each player's gun faces a different direction to begin with). The tank's body is represented by a rectangle, and the gun is represented by a line. I decided to use simple shapes because after all, it is a simple and classic game, and the simple line used to represent the gun is easy to rotate and does the necessary job. The tank's body is a rectangle and enables for easier collision detection. Finally, the Tank class handles the tank's movement, i.e. moving left and right. When I started implementing this Tank class, I based it on the PlayerShip class from the first Tutorial.

Furthermore, to represent the missile, I used the Missile class. The missile is represented by a small rectangle, and is initialised with the coordinates of the missile, its size and the damping factor there is on it to represent drag. This class handles the movement of the missile by using vectors. The initial position of the missile is a vector, and then every time the missile moves, we update this vector with the velocity of the missile (which is based on the strength the player chose to shoot with and the elevation of the tank's gun), the gravity which is always a set vector, the wind speed which is

also passed through as an argument as it changes for each turn, and finally the damping factor to represent the drag. After a series of testing with trial and error, I found that it was best to divide the strength by 200, the velocity's x and y coordinates by 5, and the wind speed by 1000. This helped create a nicer and more realistic flow to the game, while keeping the values for strength and wind speed realistic to the user. When I started implementing this Missile class, I based it on the PlayerMissile class from the first Tutorial.

The Artillery file controls the game as a whole, using the classes mentioned above. I have a number of static variables to define the proportions and sizes of the tanks, missile and blocks. I also use a number of boolean variables to control the flow of the game (what is happening and what should be displayed), and other variables that control the game settings, e.g. wind speed, power, player scores, etc.. I have a number of 'setup' methods, the main one setting up the screen for the first menu, where the user selects single or two player. If single player is selected, the 'setupDifficulty' method is called to setup the screen where the use selects the difficulty of the AI. After this, or if two player was selected, the 'setupGame' method is called to setup the game on the screen. This method handles drawing the landscape with the blocks, the tanks, and the text to show whose turn it is, the power and the wind speed. The draw, mousePressed, keyPressed and keyReleased functions handle what is displayed on the screen as well as the player's input and decisions, by using the boolean variables to check the game state. I then have multiple functions that check for collisions – between tanks, blocks, and the missile – and also functions that help with the arrangement of blocks.

In addition, the AI is also handled in the Artillery file. When it is the AI's turn to play, the function 'AIfire' is called, and it starts by calculating where the AI's tank is, where the enemy is, and other variables about the tank's gun and missile. Then, starting with a low power, it simulates a missile path fired and, using helper functions, checks whether the missile would hit the enemy tank, or end up off the screen or hitting a block. The simulation is done exactly how it is done in the missile class, with vectors representing the missile's position, velocity, wind, gravity and drag. If the power is insufficient to hit the enemy tank, we increase the power and re-simulate the missile shot. This keeps happening until the enemy tank is hit. If the enemy tank is never hit, the tank's gun angle is changed (we move the gun up, increasing the elevation), and we start over with a low power, increasing it for every shot. Most of the time, a successful shot is found, in which case we store the gun's power and elevation. Otherwise, the elevation and power are given default values. After these values have been chosen, to make the game more interesting (otherwise the AI would have a successful shot every time), I decided to add an element of randomness when picking the strength of the shot, by using the random method to pick a value around the one that we found to be the perfect strength. Therefore, randomly, the AI will sometimes just miss or successfully hit the player.

In terms of game play, I decided to allow the user to change the elevation of the gun on the tank using the up and down arrow keys, and to move the tank left and right using the left and right arrow keys, as these are quite simple and comprehensible. To fire, I decided to make things a little bit more interesting that I thought made quite a difference to the game play. I decided that when the user hits the space bar, the power of the shot starts rising quickly from 0 to 500, and then falling quickly from 500 to 0, and continuing in this loop. When the user hits the space bar again, the power of the shot is selected at the value it was at in the loop. This adds a little bit of skill and randomness to getting the power you want to shoot at, rather than just knowing what power you want and selecting it. Finally, I decided to make the game end when one of the players reaches a score of at least 10, and has a score of at least 2 higher than the other player. This makes for a game of good

length, and removes any bias to the player who starts the first turn. Also, to make things more fair, if a player loses a turn, they are the ones who begin the next one – it isn't always player 1 that starts. Overall, I found that these rules make the game fair and last a good amount of time, depending on the players' skills. For the rest, the requirements of the practical were met, meaning that blocks disappear when they are hit and tanks cannot go through blocks or the other tank.

Extensions:

The first extension I decided to do was to add some landscape blocks on the whole screen, not just between the two tanks. This way, there can be blocks behind and under the tanks, and the two tanks can start at elevated positions and traverse through the map, as long as there aren't any blocks stopping them. This adds a different element to the game, as the players can find better positions to put themselves in, and makes it more exciting as their starting positions change every time (they can be very high up on the screen or low down). To implement this, I increased the number of blocks generated to fit the whole screen. Then, I created a function to check which is the highest block under a tank, which is called getBlockUnderTank. Once we know which is the highest block in the tank's row of blocks, we know where to place the tank on the screen for its initial position.

Once this was implemented, I decided to create my 2nd extension, which was for tanks to be able to fall, as affected by gravity, when there is no more blocks underneath them. This was necessary because once tanks were able to start in elevated positions, and since they could move left and right, there were times when they could move to spaces that didn't have blocks under them, which didn't make the gameplay very realistic. I did this by using the tankOnBlock function, and checked that there was always a block under the tank. If there wasn't, the tank's moveDown function was called, which makes the tank move down on the screen. As soon as there is a block under the tank, the tank stops moving down. This in turn creates for an even more engaging and fun experience, as the player can have much more choice in selecting where to position their tank, and perhaps hide behind some blocks which makes for good cover from the enemy's missile. However, the player has to choose where they go wisely because once they fall, they can't jump or fly back up onto blocks. Although I considered making this a possibility, I thought it would be quite unrealistic and would make it too easy for the two tanks to get very close to each other (meaning for a very easy winning shot).

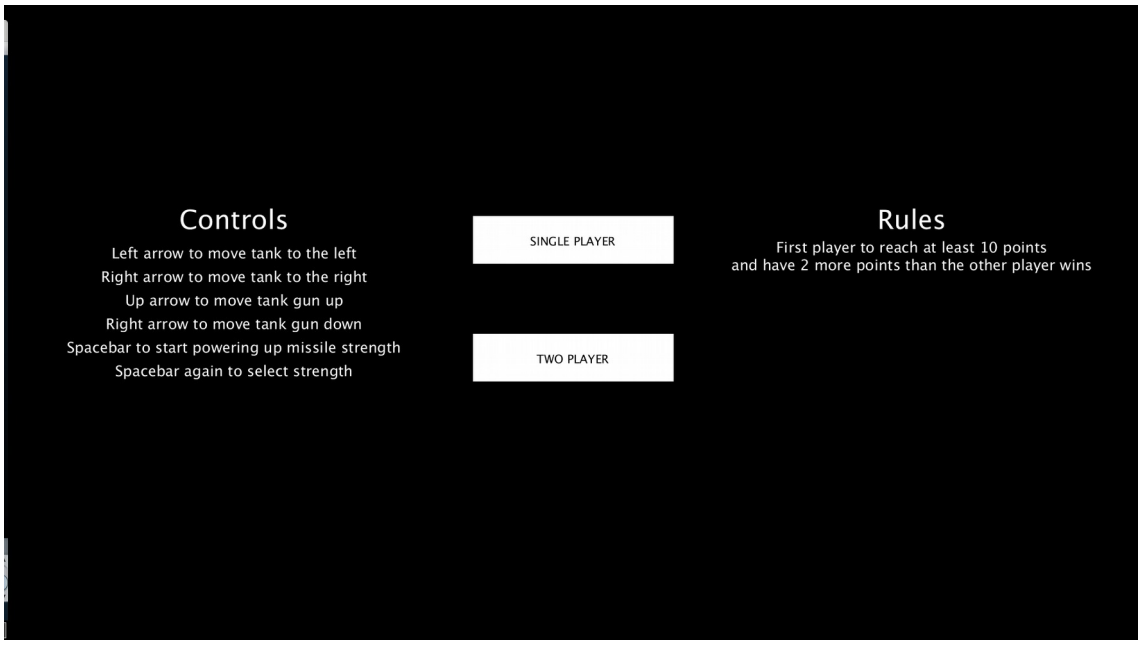
Moreover, I decided to implement the extension of making some blocks fall when the blocks below them were shot, using a similar implementation as above. I created the rearrangeBlocks function, which is called when a block is hit. When this happens, I check to see if there are blocks above the hit block, in which case I call the block's moveDown method to slowly make it move down the screen, and stop at its new Y coordinate (which is the Y coordinate of the block below). I also adjust the 2D array of blocks to contain the new blocks' position. This way, again, the game is more realistic in the sense that you don't get blocks levitating once the blocks beneath them are destroyed, which makes the game more strategic and enjoyable. It also provides more cover for tanks, because before, you could shoot the blocks in front of the tank to remove its cover, and the blocks above would remain levitating in the air. Now, since those blocks fall down and replace the destroyed blocks, it's harder for a tank to destroy the other tanks cover.

In addition, I decided to make my own extension, which was to introduce a selection of difficulty for the AI of easy, medium and hard. This way, the player can practice and get better by playing on easy, and slowly work their way up to the hard difficulty to have more of a challenge and therefore a more fun experience, as it is much more rewarding beating the AI when it is playing very well. I thought this was necessary because it was hard to find a balance between the two – either the game was too hard for a new player and would therefore discourage them from carrying on playing, or the game was too easy for someone who played for a little while and got used to it, and therefore stopped being fun. In this way, the game is much more grasping for new or experienced players, who don't lose interest. I did this by creating buttons to allow the user to select their difficulty at the beginning of the game. Then, in the AIfire method, I simply changed the error in randomness in selecting the value for the perfect strength needed for the winning shot. In easy mode, the AI chooses a value for strength randomly between 50 below and 50 above the perfect strength, meaning there is large space for it picking a wrong value. In medium, it picks between 20 below and 20 above the perfect strength, and in hard it picks between 5 below and 5 above the perfect strength. This way, in hard mode, the AI usually hits the player in 1 or 2 turns, meaning that it is very hard to beat – the player's shot has to be successful within the first two turns, otherwise they will lose the point. And in easy, the AI takes much longer to hit the player, so the player has many more chances to score a point.

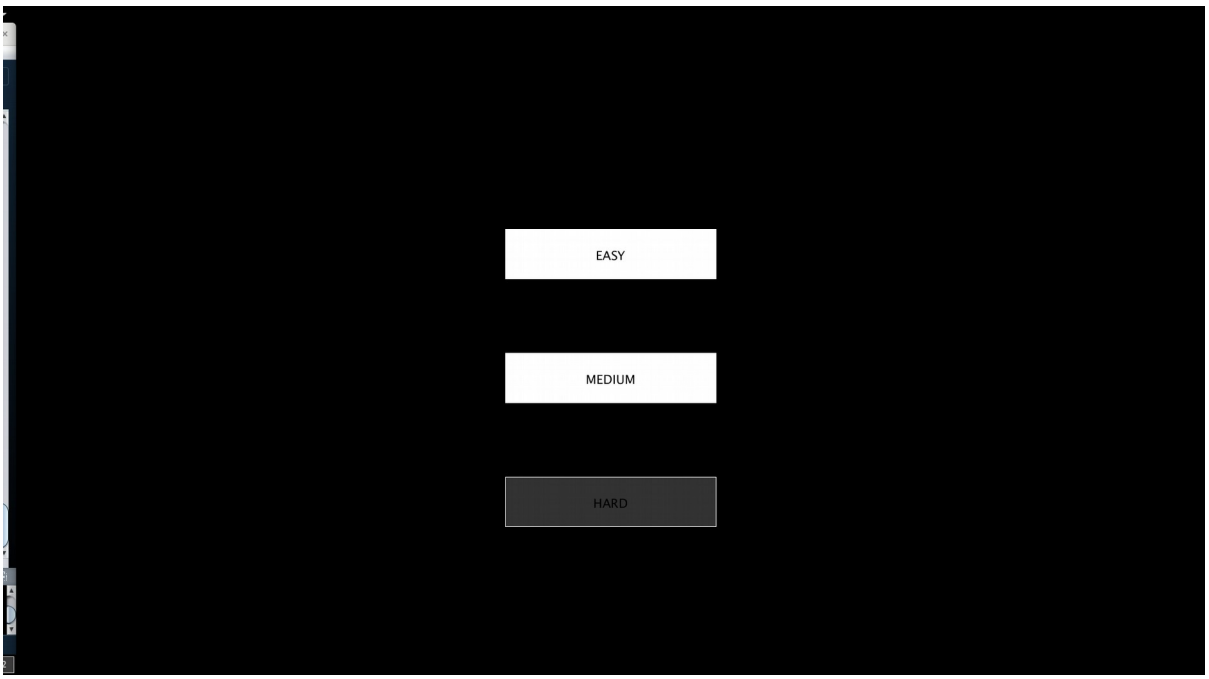
Finally, I did the other extension mentioned in the specification, which was to add sound effects. I did this using the minim library, which I import at the top of the Artillery file. Using minim, I load the mp3 files in the data directory and play them when appropriate. To not overdo the sound effects, I decided to have one sound effect when a shot is fired, one when a tank is eliminated, and one when a player wins. I downloaded the shot fired and tank explosion mp3 files for free from https://www.freesoundeffects.com/free-sounds/explosion-10070/ and the winning trumpet sound from http://www.orangefreesounds.com/fanfare-sound/.

Testing:

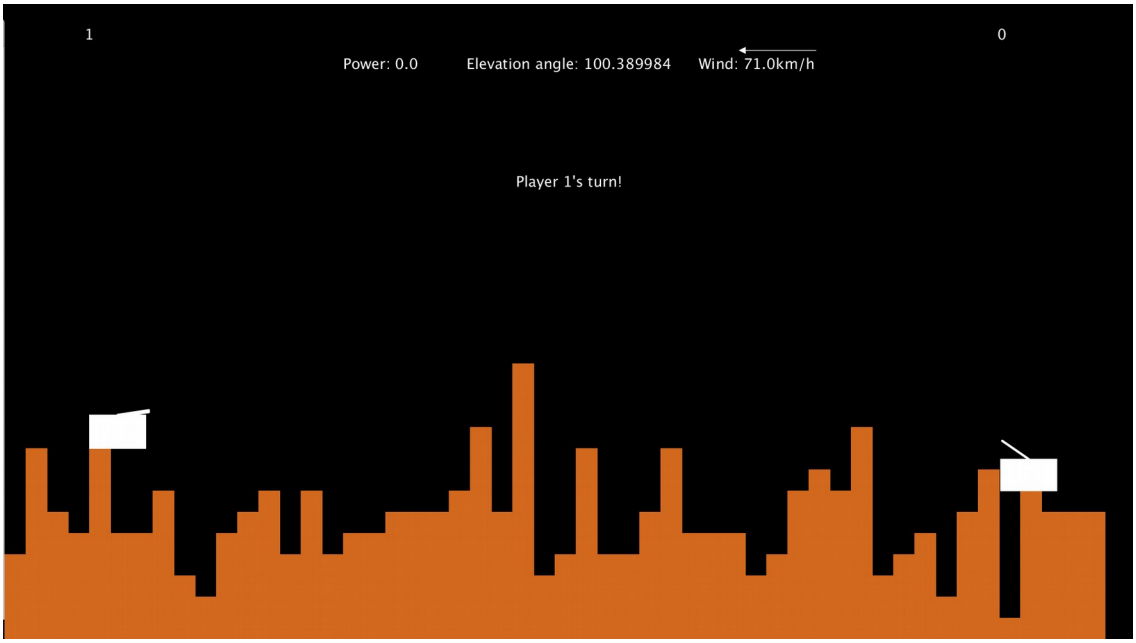
Initial screen, picking single player or two player, rules and controls:



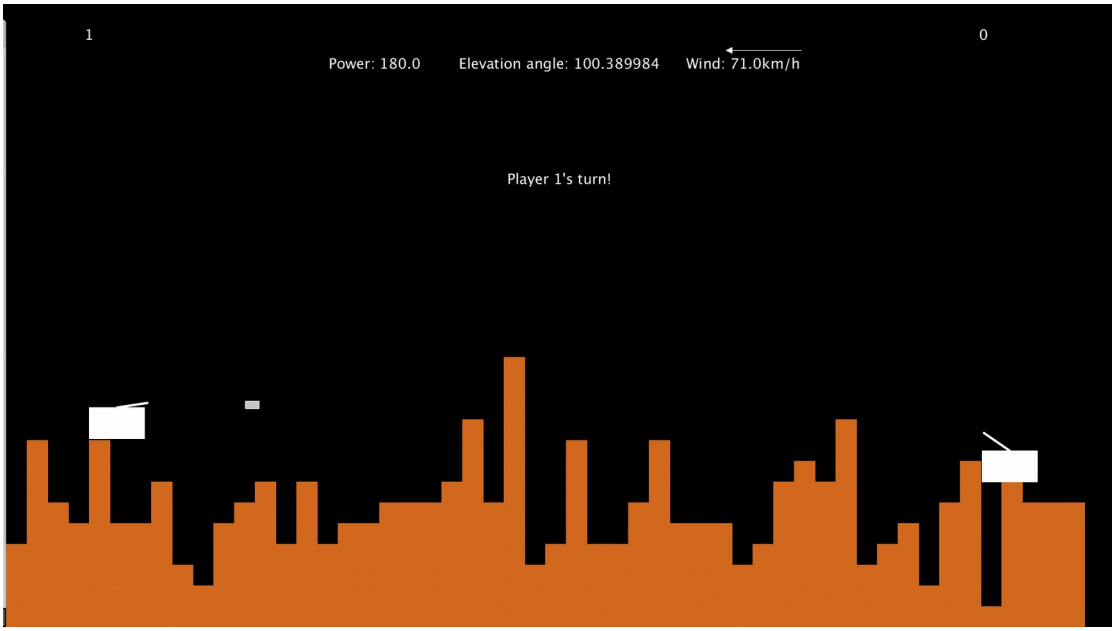
Picking difficulty for AI:

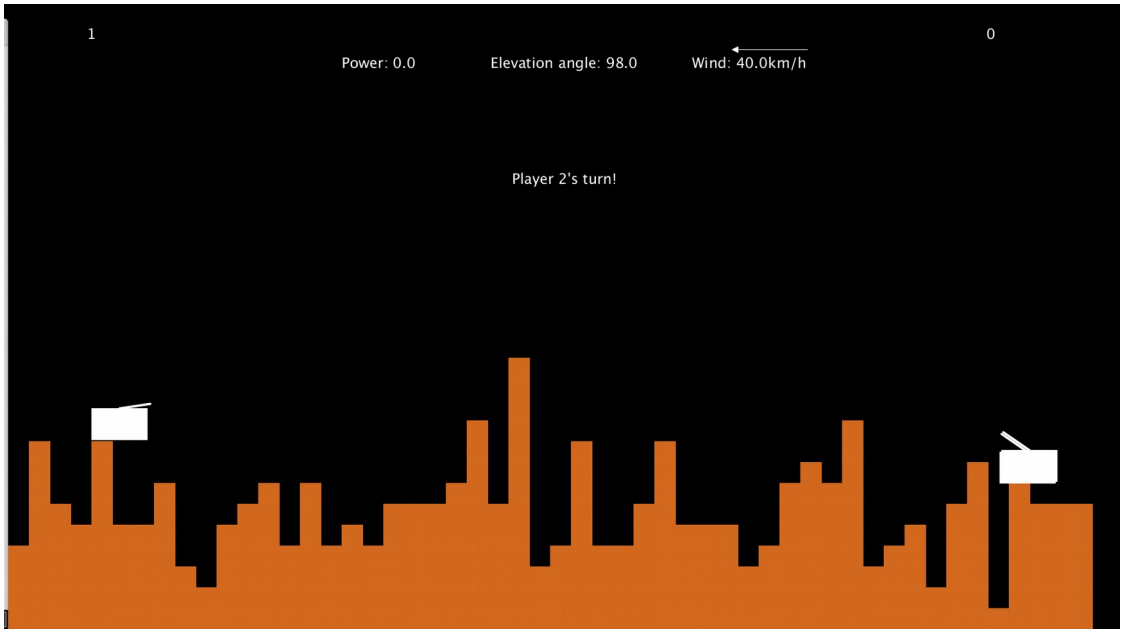


Initial game, with wind speed, power, elevation, and player's turn indicated, and randomly generated blocks below the tank as well as between the tanks, making the tanks start in elevated positions:

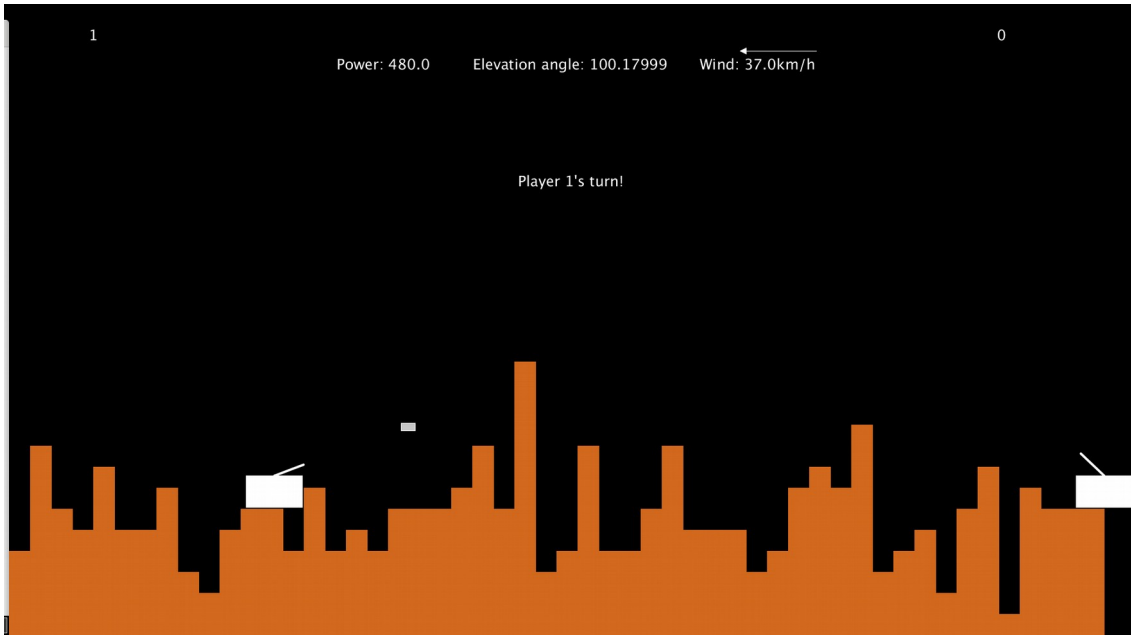


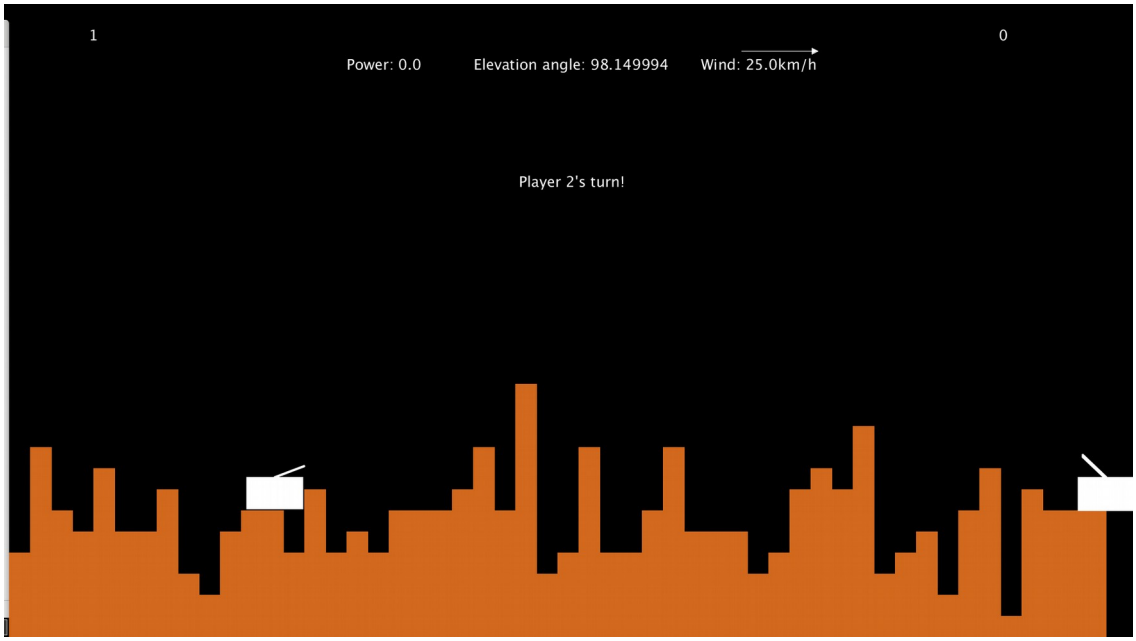
Blocks disappearing when hit:



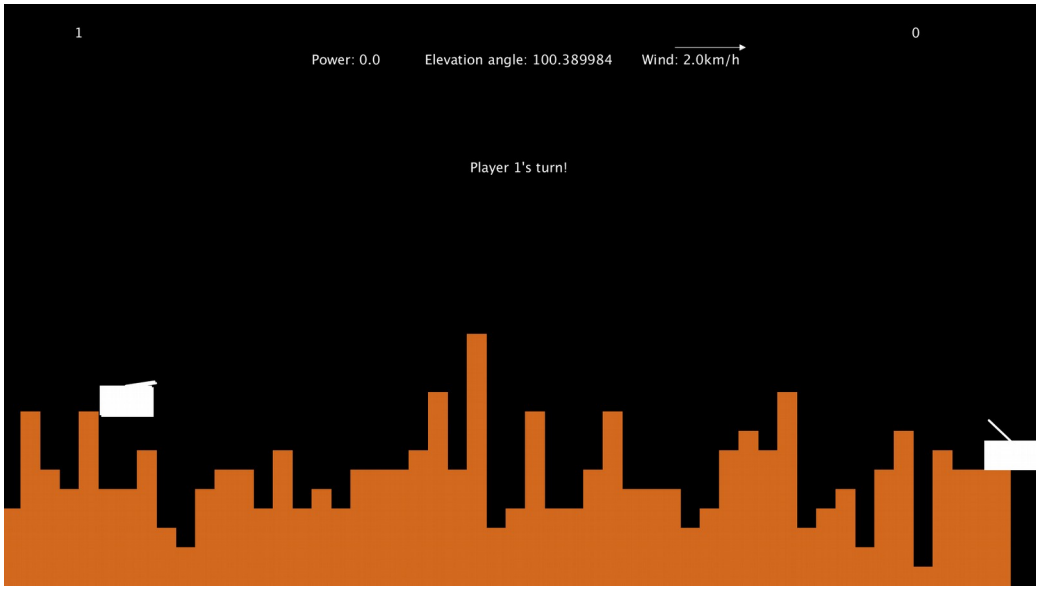


Blocks falling down when a lower block is hit:





Tank (on the left) falling when no block below it:



Player winning when reaching score of at least 10 and 2 more than the other player:

Known issues:

Some gameplay freezing when playing against AI, while it is making a decision (sometimes the missile freezes on a block before disappearing, or in the air).

Sometimes AI can't find perfect shot due to the wind speed so it resorts to a default strength and elevation, and can get stuck there for a while.

Sometimes, when running this application on Linux, the sound effects sometimes don't play and an error message is printed to the console about the files not being able to be loaded (even though most of the time they work). After doing some research, I found that this is a common problem people found when using minim on Linux.

Evaluation & Conclusion:

Overall, I believe that I have met the basic requirements of the specification, and completed some of the mentioned extensions, as well as some of my own. I believe that the decisions I made in terms of scaling, for example the sizes of everything, how much the missile is affected by the different factors and forces, have made the game quite fun and enjoyable and fast paced. The few elements of randomness in terms of the positioning of blocks, tanks, wind speed and even the power (you don't always manage to select the power you want to because of how quickly the power increments) make the game harder and more strategic, and make it so that the players can't exploit landscapes or positions that make it easier for them. Although it is a turn based game, I tried to make it as even as possible, so that the first player doesn't have an advantage, and I believe that the game length is good and that the player who wins is definitely the fair winner, rather than being just being lucky or having any sort of advantage over the other player. Finally, I am happy with the AI because it accommodates for players of different skill levels, and therefore makes the game more enjoyable for everyone. I am now much more comfortable programming in Processing and working with physics, and am keen on working on the next practicals to improve these skills. If I had had more time, I would have loved to make the look of the game better by improving the graphics and the background, as well as adding some animations and sound effects. I would also want to fix some of the known issues, such as the slight freezing in the gameplay when the AI makes a decision, and the problem with minim not being able to load the mp3 files sometimes on Linux.