CS4303 Physics Practical: Particle Artillery 160026526 26 February 2021

• Overview:
The aim of the practical was to get familiar with Processing language, implement the concepts based off the Physics portion of the module and have though on game design, implementation and tuning. More particularly, an implementation of the classic Artillery Game.

• Design:
Basic artillery design
A PlayerShip class is made, along with constructors, getters and setters, to represent the artillery which initially takes x and y values for its coordinates on the play area, width and height for its dimensions, a move increment to control its movement speed. It contained a function which allowed the tank to be drawn as a white rectangle using the position and dimensions. The tank was left as a rectangle for simplicity purposes when calculating future collision. Two more functions were then added to control the movement of the tank with respect to the play area boundary with a specified increment. In the main method, the proportions are initialised as constants and calculated in the setup and two player artilleries are created at opposite ends of the play area with respect to the proportions to the display although the game was made to default in fullscreen with a resolution of 1920x1080. The artilleries are then rendered in main method using draw.

Shell design
The PlayerMissile class is made to represent the shell of the artillery to be fired. Similar to the artillery, constructors, getters and setters were implemented. Initially the shell had x, y values for coordinates, width and height for dimensions of shell, a move increment which will be used for a constant increase or decrease of the magnitude, and defaulted values of 0 and PVector(0, 0) for elevation, strength (magnitude) and velocity as the game has not started yet when these are initialised. A draw function is made to render the shell as a grey rectangle which will help in collision detection. A move function will be used to update the position of the shell when it is fired which stops if it hits the edge of the play area. Two functions for magnitude control are added, limiting the lowest magnitude to be 0. Two more functions are added for elevation control in degrees in which allows for the shell to be fired 180 degrees (in front and behind it). A single shell is initialised in the setup of the main method in which its position will be swapped between both artillery in between turns.

User controls
User inputs are processed by reading the keyboard for inputs. Booleans are used to for keyboard inputs so that the keyboard does not have to polled in each frame. Keys are then assigned for the movement of artillery, change of elevation and change of magnitude. The keys magnitude are assigned to are “,” and “.” which and are not assigned “<” and “>” as it would require to input shift although in the rules to be made, it will be mentioned magnitude is controlled by “<” and “>” since they share the same keys. In the main method for draw, firing is then implemented and checks if the shell is being fired which then renders the shell until it collides with a display border and the shell is integrated here . A firing function is made which fist checks if a shell is already being fired and if not, it resets the position of the shell to the current players top corner facing the other player and sets firing to true which will move the shell.

Velocity of shell

Taking an initial normal vector (1, 0) (which is just a direction) to be the normal vector in the east direction, we want to use and angle to shift this normal vector to another normal vector at the elevated angle. This is done in the centerRotation function. Taking x = rcosB and y = rsinB where r is in radians, we can obtain a normal vector with x' and y' by using the formulas,

$x' = r\cos(A+B) = r(\cos A\cos B - \sin A\sin B) = r\cos B\cos A - r\sin B\sin A = x\cos A - y\sin A$
$y' = r\sin(A+B) = r(\sin A\cos B + \cos A\sin B) = r\cos B\sin A + r\sin B\cos A = x\sin A + y\cos A$

and as to get the new normal PVector(x', y'). thus, the normal vector (1, 0) and the angle in degrees are used to calculate this. The angle is first converted into radians and multiplied by -1 to ensure an anti-clockwise rotation from the start normal vector of (1, 0). The sin and cos values are calculated and then the new PVector is returned using the formula above.

Back in the firing function, now that we have the normal vector, force is set to this normal vector and is then multiplied by the missile strength (which is divided by 100 due to the shell initially being too fast) which outputs the missile velocity. This shell velocity is then set to this velocity for its initial velocity. The spacebar key press is assigned to fire which will then fire the shell at the velocity. In keyPressed, it is also checked to make sure that movement is only allowed when the artillery is not currently firing to ensure players don't try to dodge. In the case where this is not checked, players were still allowed to freely move although the shells were flying and would be less of an artillery game and more of a dodge the bullet game.

Forces acting on shell

Gravity and friction are now implemented onto the shell. Simplified friction is implemented via a damping factor . This is added to the PlayerMissile class at a float of 0.995 and gravity was set as a PVector(0, 0.49f). The value of 0.49 was chosen as it allowed for a natural looking descent when firing the shells. Back in the move function, the movement and integration will happen here. The velocity set by firing is then added to the position of the shell which updates the shell at each frame to remove the factor of time. Gravity is then added to the velocity and finally the velocity is multiplied by the damping factor which simulates friction in the air. To consider wind, and additional PVector called wind is created in the shell class. In the setup, wind is initialised and a random value from -0.3 to 0.3 is taken for its x value which will be used to simulate wind. In the testing, using more than 0.3 for either value causes too much wind making the game unenjoyable. The missile wind is then set to the wind's PVector and in the shell class, added to velocity right after gravity.

Swapping players after firing

Two more PlayerShips are created in the main method called currentPlayer and nextPlayer. These will initially be set as Player 1 and Player 2 respectively. To swap, right after the shell is done firing, the function swapPlay is called first checks if the missile has stop firing and then swaps the players. To make the game more user friendly, it is necessary to retain the current strength and elevation for both players when it swaps between them. Two extra variables are created in the PlayerShip class for this. At the end of the player swapping, the current players strength and elevation are set to its previous values to allow it to "remember". When firing, the current player will save the elevation and strength it currently is on.

Shell collision with artillery and point counting

To check shell collision with enemy artillery, we check each edge of the shell is past the opposite artillery side. This is checked in an if statement right after the missile moves and if it is true, the current player is scored. Since the player is scored, the PlayerShip class now required a point variable. Right after scoring, we check the score to see if the current player has reached a score limit. Setting the score limit to 10, if the limit is not reached, swapPlay is called to swap the players.

Terrain generation and collision for shells

A new terrain class is made in a similar fashion which takes x, y, width and height to find the position and dimension of each terrain block. The terrain class contains a draw method which renders the block in a dirt brown colour. In the setup, some values for terrain generation are initialised along with an ArrayList to store all the terrain blocks. The initial left most column of terrain blocks will be called numBlocks which is first set to a random value from 4 to 14. This is to ensure that when the game is restarted, the initial column will always have a different height. The terrain blocks are generated in a for loop and each column after the first one will have -1 to 1 more blocks to it in an attempt to randomly generate a slope. The choice of 4 to 14 was chosen before to ensure there is no chance where the land is generated too “flat”. A check also keeps from generating negative blocks. All that is left to do is render the terrain. To render, a for loop loops around the ArrayList and draws for every terrain in the list. Each time the program is ran or restarted, the terrain is randomly generated. For collision detection with shells, when the shells are fired and the shell is moving, iterate around the terrain ArrayList and similar to collision detection on artillery, if the terrain block is hit by the shell, the terrain is removed from the terrain ArrayList and i is iterated once down. Swap players when a terrain is destroyed.

Player movement on terrain

It is envisioned that the artillery should always be above the terrain and has the ability to climb up block and fall from higher blocks to lower blocks. An onTerrain function in the artillery is used to ensure the artillery are above the blocks and can move down to lower blocks. To falls to lower block, gravity is added to the artillery class and is set to PVector(0, 0.1f) as a factor of 0.1 allowed for a gentler descent to a lower block and created less artifacting on the blocks. A Boolean called fall is used to trigger the gravity effect on each block which is defaulted on true. If the artillery is falling, integrate the position by adding gravity which emulates gravity pulling the artillery downwards. Two checks are then used to examine terrain within the same x axis range of the artillery and shifts the artillery above the terrain if it is not already above it and fall is set to false, so the artillery stays in place. A check is also used to maintain the y position of the artillery to ensure it does not fall off the play area. Back in the main method, in draw, onTerrain is called for both artillery on every single block in terrain and two extra checks follow suit to turn off gravity if the block is at the bottom of the screen and turn it back on when its not. Each artillery was also limited to the how far it can travel to ensure they don’t drive right next to each other. If this was allowed, players would just drive next to the opponent and fire. Currently artilleries, when moving into an x value which has a block higher than it, they are placed onto the topmost block. This could be improved but having it allows for less options for artillery to hide under the terrain to draw out games.

Changing of the wind

The wind changes every 20 turns and a turn is counted when the players swap. After a shell has fired and players have swapped. There is a check to see if the wind turn counter has reached 20 turns and if so it generates a new PVector of wind with the same values as mentioned before. It then sets the missile wind to the new wind and reset the wind turn counter to 0.

User Interface during game

A majority of the mid game interface being displayed are done by updating text in the play area. Both players scores are always set in the top left corner. The current player is displayed in the top middle of the screen. This is followed by the wind's x vector and a message indicating if the wind strength and direction. After this, the next text shows how many turns till the wind change as it will be useful to know when the wind changes. The elevation and magnitude of the current player is then displayed and finally the velocity of the missile as it moves through air.

Clouds

A good visual indicator of where the wind is blowing would be the movement of clouds. Three clouds are created using the cloud class and set at different position. The clouds are rectangles similar to the artillery but in a grey colour. The clouds velocity are all set to the value of the wind whenever a new value for wind is generated. Clouds have two function, one to draw and one to move. When the clouds move, velocity is added twice to make the clouds appear faster which allows for better estimate of wind. There are then two if statements in the move function to ensure the cloud re-enters the screen after it leave from one side to another. The background is painted sky blue to emulate the sky.

Restarting when the game ends

To begin, we look at the check score function in the main method amend the check if the current players gets 10 points. We add a lock Boolean which is false when the game is playing and true when the current player reaches the number of points required to win. A if not lock loop is put around the player movement and firing and as such the game will only play when it is unlocked. An if lock loop is then used to display the current player as the winner. A restart button is created that highlights when hovered over with the mouse using a recent added update function that checks if the mouse if over the restart rectangle. If the mouse is over the restart rectangle, it changes a new Boolean called restartOver to true and if not its false. A mouse pressed function is then added and if the mouse is hovering over the restart rectangle, restartOver is true and a reset function is called. To implement this reset function, a majority of initialisation from the startup is moved to the function called reset and reset is called at the end of startup to start the game initially.

Start instruction screen

A start screen is implemented using a Boolean startScreen. It is first initialised in setup to be true and in draw, every line of code is encapsulated in an if not start screen. During the start screen, nothing from the game will be rendered. An if start scree is true loop is written at the end of draw which showcases text of the name of the game, the goal of the game, what the keybindings are, a note for player 2 to keep in mind and how to start multiplayer. The enter key is then added to keyPressed to make startScreen is false when pressed. The enter key has to be pressed to start. When the game is started, click on the screen with the mouse to ensure that the screen is active. There are occasions when pressing enter doesn't start the game and the program has to be restarted.

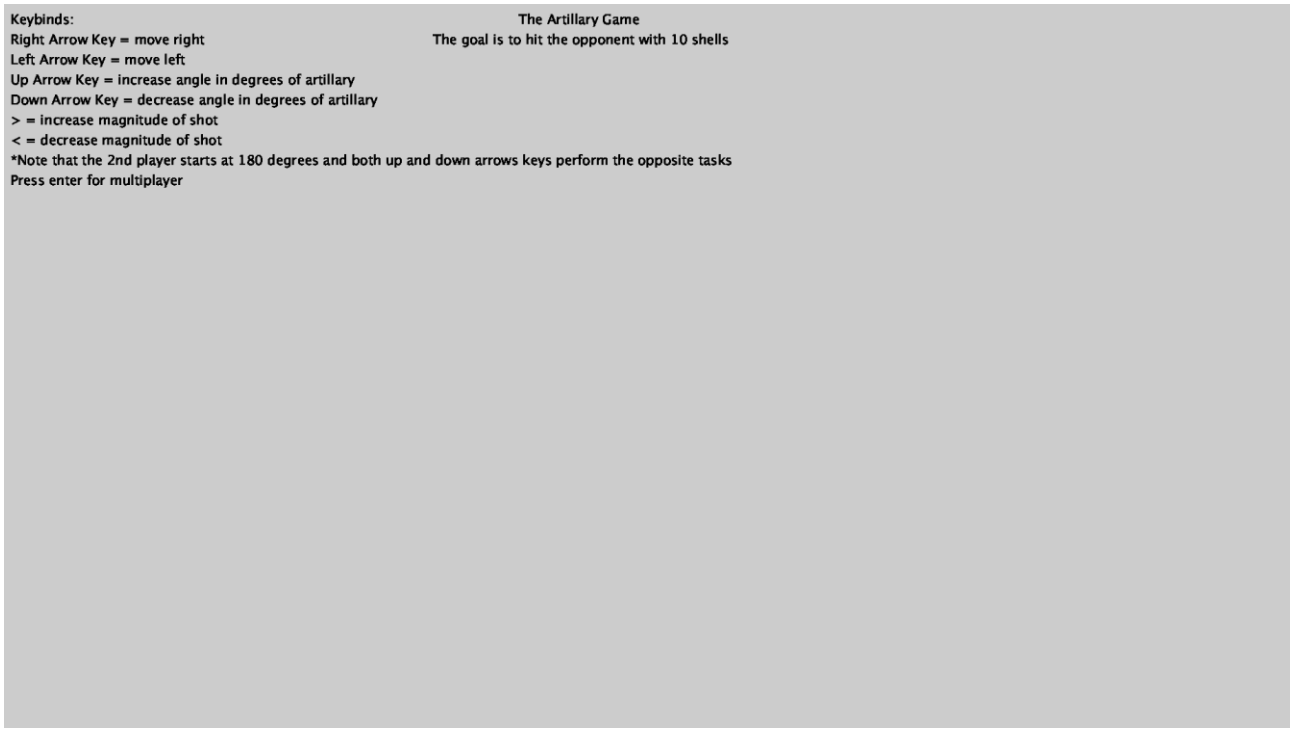
AI

No AI was implemented but if it was, the AI simulate a fired shot and apply all forces to locate where the shell will land. This is then adjusted until the simulation hits the player's tank. With the location of the tank found and the approximation of strength and elevation to set, the AI can easily hit the player's tank. To make the game more fair, can calculate to where the AI will miss a certain

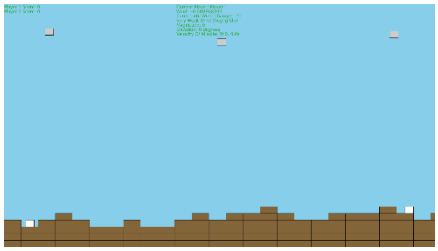
percentage of its shots, for example 60% of its shots will miss. AI movement control can check if there are terrain blocks blocking its trajectory it can navigate either forwards or backwards, whichever giving it a cleaner shot.

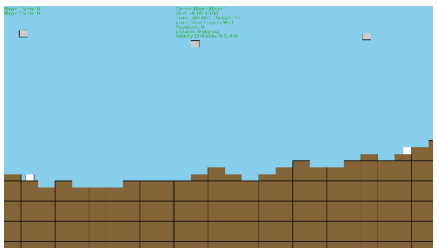
Screenshots of the game

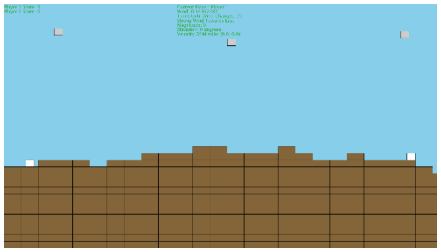
• Start Screen

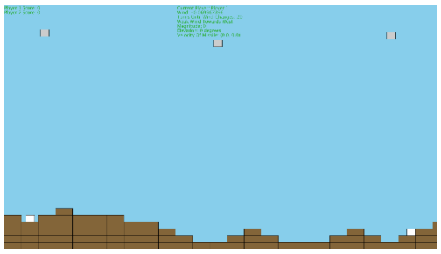


• Gameplay (Terrain generation)

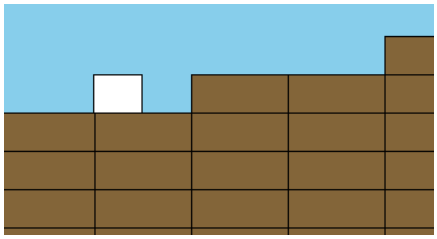


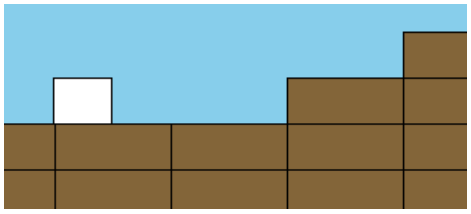




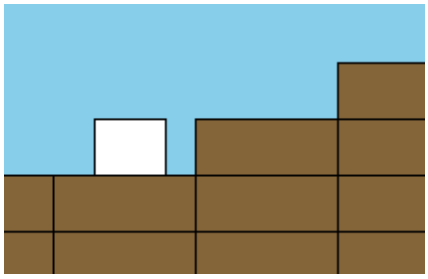


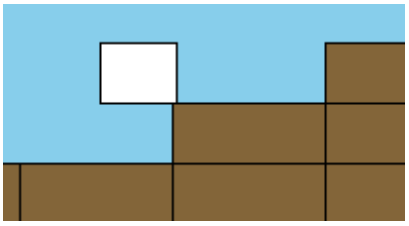
• Shooting destroys block

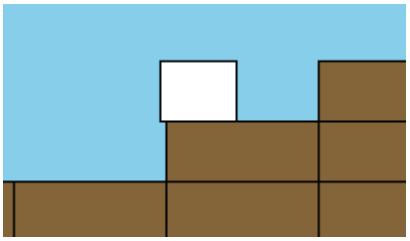




• Artillery climbs terrain







• Player 1 scored 10 points and won, restart button to play again

• Evaluation:
In this practical, the multiplayer artillery game was successfully implemented. The artilleries were successfully created along with the play area and the shell. The landscape blocks are also randomly generated. The force of gravity, wind and friction are successfully implemented and wind changes every 20 turns. Clouds are also generated to indicate the direction and velocity of wind. The controls for keyboard controls have been set up and left click is only required to restart the game. Enter is used to enter the 2-player game. Collision detection works for the artillery and the shells. Most of the information that users may require are displayed during gameplay. The program is implemented properly but will require an additional single player with AI.

• Conclusion:
In conclusion, the program was able to play a 2-player artillery game with the implementation proper game design and physics that affect the artillery and its shells. AI was not implemented and with more time, would probably have been implemented along with additional menus to support it before the game starts. In the future, additional features can be added to simulate sound and visual effects but at the moment is sufficient for a physics engine and is adequate as a introduction to Processing.