CS4303 Particle Command Report

140011146

November 8, 2017

1 Introduction

In this practical we were tasked to implement the video game *Particle Command*, which is a variant on missile command where the particles are blasted away from the explosion rather than being destroyed. In my game, I have implemented all the features in the practical specifications as well as some additional features.

To build and run the game, run `ant` in the submission directory.

2 Game features

2.1 Basic

Meteors

The particles falling are implemented in the `Meteor` class. They spawn from a random location at the top of the screen and have a random initial velocity. This initial random velocity goes in both the x and y direction. The random y velocity makes some particles come down faster than others to make the game a bit more interesting and the x velocity makes it so the particles don't all just fall straight down. They do not have a random negative y velocity because all particles not in the screen are destroyed so they should not be spawned then move up initially.

The meteors are also affected by a force of gravity and drag, implemented as `ForceGenerators`. `Gravity` is implemented just as a downwards force because it is much simpler compared to modelling the ground as Earth and doing gravitational attraction. The force of gravitational attraction is implemented, but not used for the "gravity" pushing the meteors down.

`Drag` is fully implemented following the formula given in the lecture slides with `k1` and `k2` constants of drag.

Cities

The cities placed on the ground are static and do not change. This was done for simplicity and a fair balance so there are not occasional games where the cities are very far away, making the game more difficult. Although the cities are drawn as rectangles, their detection uses a circle so it is not completely accurate, but the difference is very small and hard to notice for a player.

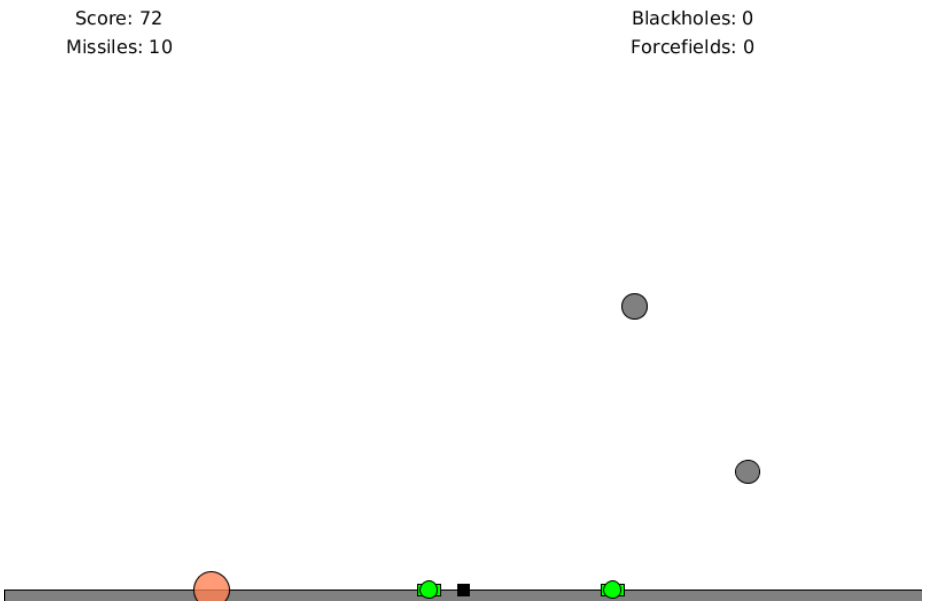


Figure 1: Meteors falling from the sky. The one on the left has hit the ground and caused an explosion. Remaining cities can be seen in green. The circle indicates their detection radius for being destroyed.

Explosions

All particles can create explosions when they are destroyed as it is an abstract method all `Particle` subclasses must implement. In my game, both the meteors and player missiles create explosions. The explosion radius is based off of the particle's initial radius, increasing with each time step until they've reach the end of their lifespan. Any particles caught in the blast radius are blown away with an `Explosive` force.

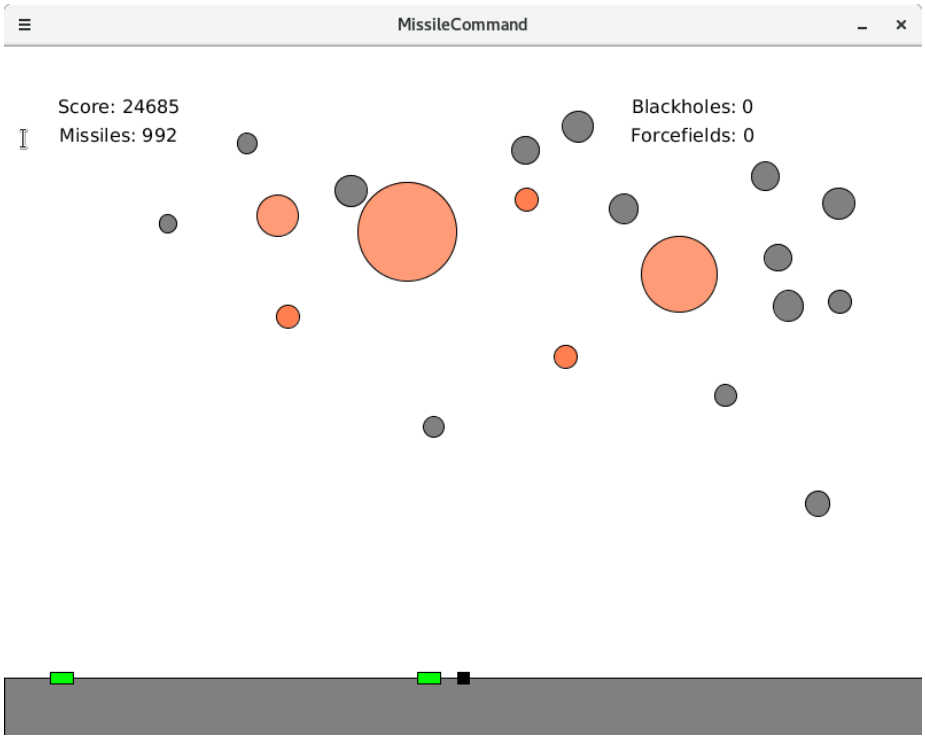


Figure 2: Normal explosions from player missiles.

Initially both player missile explosions and meteor explosions can destroy cities. This was because they were the same piece of code and did not differentiate between them. However, I thought it was good to keep it like this because it prevents the player from just clicking missiles on top of their cities and deflecting all meteors. Since I have also implemented a shop, I have allowed players to purchase a permanent upgrade so missile explosions do not destroy the cities any more. The colour of the missiles and explosions also change to indicate this.

Player missiles

The player missiles fire from the center of the ground. Their velocity is normalised and not very fast so it takes a bit of planning to properly defend the cities that are further away. The missiles explode either when they reach their destination - the position of the mouse cursor when the missile was fired - or when they collide with a meteor during their trajectory.

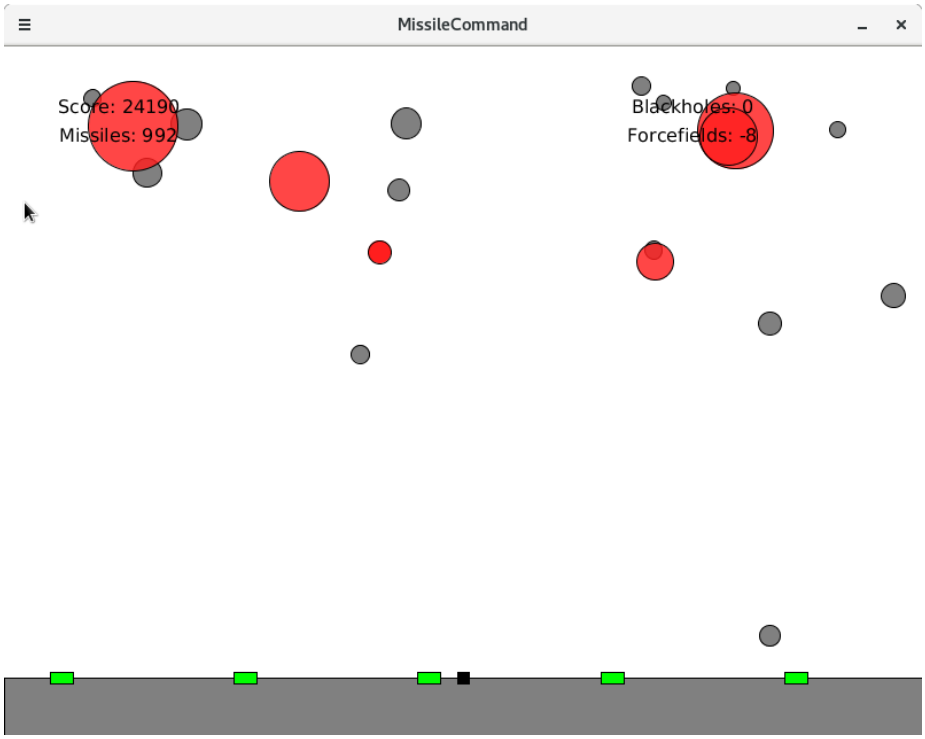
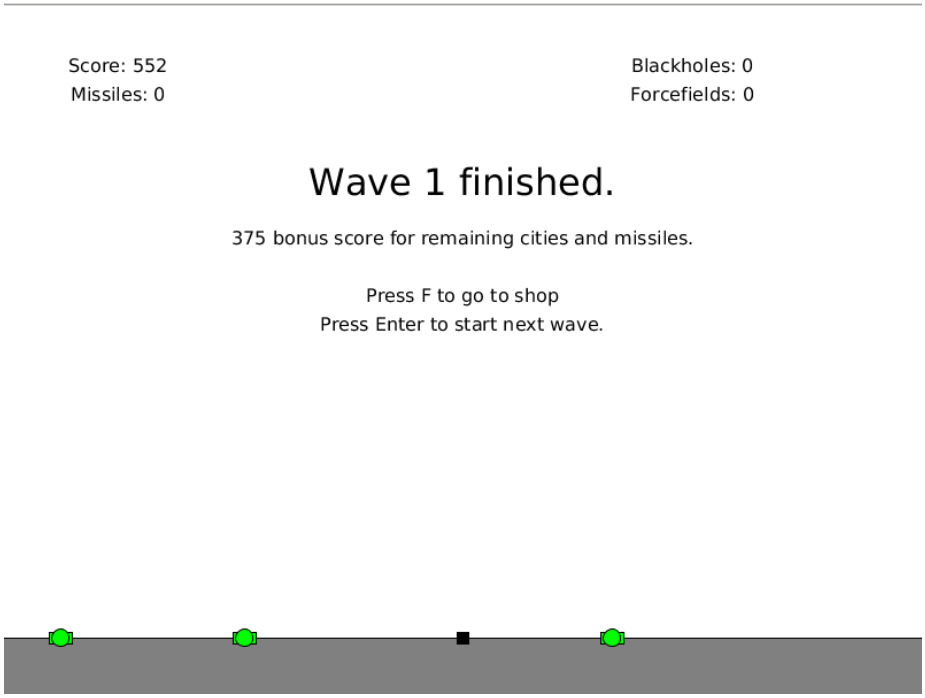


Figure 3: Upgraded explosions from player missiles. These do not destroy cities. The colour is different to indicate the change.

To calculate where they have to go and when to explode, the missiles store their destination position vector which lets them calculate where to go and check if they are close to the destination before exploding.

Waves

The game is organised into waves. At the beginning of each wave, the number of missiles the player has is increased by a random scaling amount. Each wave has increasing difficulty as more meteors are spawned. Every few waves, the player is rewarded with a black hole or forcefield to help them survive. However, every few waves a bomber also comes which makes the game more difficult.



2.2 Extensions

Split into child meteors

To make the game more difficult, the extension for splitting meteors into child meteors is also implemented. Meteors that are larger than a certain radius have a chance to split into 2 or more child meteors. As the player goes into higher levels, the number of child meteors that spawn is increased. Meteors only begin to split at after a certain level so the beginning of the game isn't too hard.

Bombers

In addition to splitting the meteors, bombers that drop "bombs" are also implemented in the game. These bombers fly across the screen and drop "bombs" that are much heavier than normal meteors. They are also given an initial downwards velocity to make them fly faster towards the player's cities. I have not allowed players to be able to destroy bombers because I found that a player will always go to destroy the bomber immediately so it is not there for long enough to add a challenge.

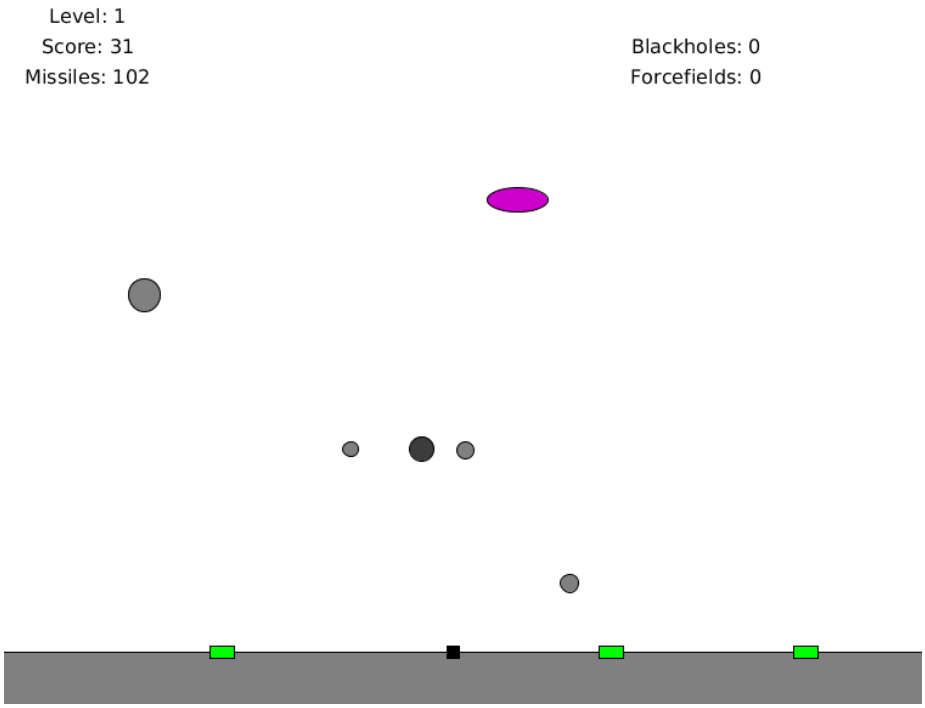
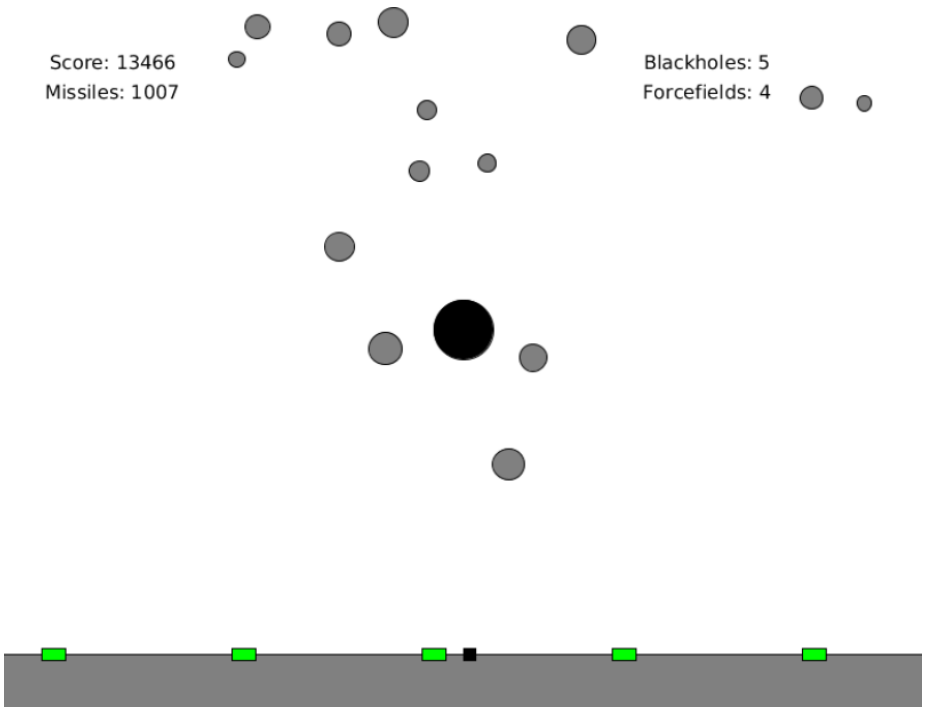


Figure 4: Bomber flying across the screen. The darker grey meteor is one of the "bombs".

In the code, the bombs that are dropped are implemented as meteors but drawn with a darker colour. This is because have the exact same properties as meteors, only they have a set mass, radius and larger initial downwards velocity.

Black holes

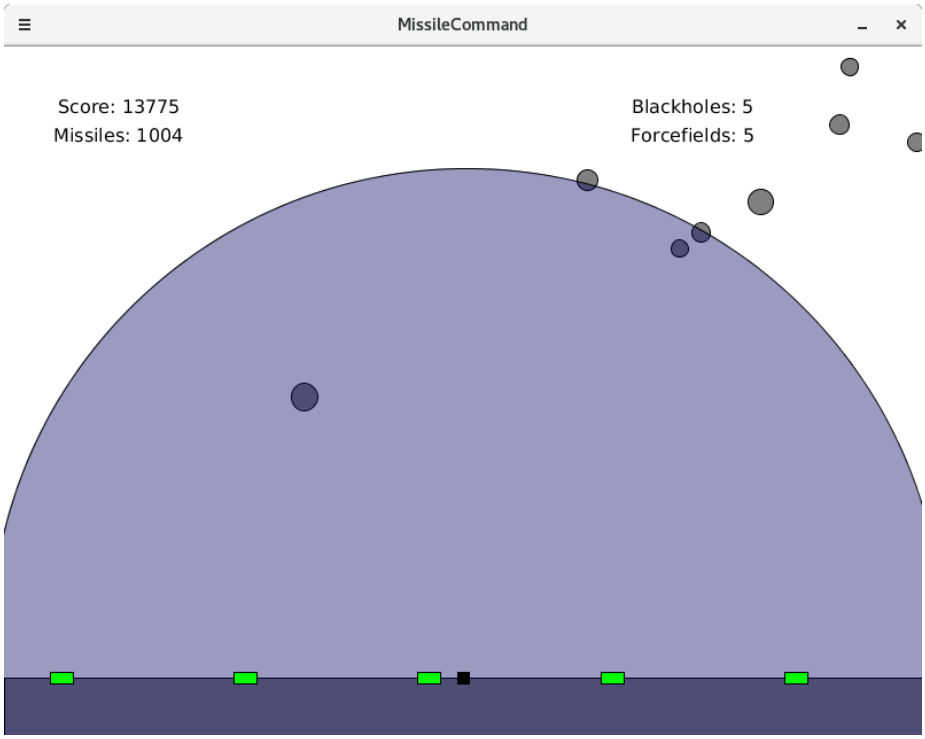
To add the force of gravitational attraction to the game, I added a special missile that player's can shoot which would create a black hole. The black holes apply gravitational attraction to all meteors on screen and because they are intentionally much heavier than the meteors, most will be sucked into the black hole and destroyed. Meteors that are further away may be affected less according to the equation.



The black holes only last for a short while before disappearing. This means that if a meteor was being accelerated towards it but did not get sucked in, it will no longer have a force applied to it, but it will retain its fast velocity, only being slowed by drag. This makes the use of black holes more strategic and a double edged sword if used improperly as the disappearance of one at the wrong time could cause a meteor to crash down really fast onto a city. Because of this danger, the black hole missiles will not create a black hole on impact, only when they reach their destination. Use right click instead of left click to shoot out a missile that creates a black hole.

Forcefield

Unlike explosions, forcefield use a different kind of force to repel particles which makes them a more global and powerful tool to deflect meteors compared to missiles. Instead of applying an explosive force to knock the meteors away, forcefields use the opposite force of gravitational attraction. The meteors are repulsed away from the forcefield by multiplying the force of gravitational attraction by -1.



This also means that the display of the forcefield moving outwards does not properly reflect how it works as the radius does not affect how much the particles are pushed away by. However, it does show what the forcefield does in an easy way to understand. The button for using forcefields is the middle mouse button.

Shop

To allow the player the ability to use more of these special particles, I have added a little shop screen for players to spend score to purchase extra black holes, forcefields and missiles. Because these cost score to buy, the player must choose between keeping their score high, or using it to ensure they don't lose in the next round.

3 Design

3.1 Scoring system

There are two components to the scoring system: during the wave and at the end of the wave.

During the wave, if a meteor hits the ground and explodes, it adds a bit of score. If the meteor is instead knocked away outside of the screen, more score is added. By having meteors give score regardless, it allows players who are not very skilled to still get score that they can use in the shop.

At the end of every wave, the number of remaining cities and missiles are counted up and added to the player's score. This is the same as in the original *Missile Command* game and rewards skilled players as they were able to keep more cities alive and also use their missiles strategically enough to still have some remaining at the end of the round. Remaining cities give a lot more score as it takes more skill to keep them alive and often I found to only have one city remaining for a long period of the game. This provides a trade off when deciding to spend score to rebuild cities as they can provide higher "income" of score at the end of every round, but the player must be able to keep them alive. A good strategy may be to only have one city because it is easy to defend, but this will

take a long time to get a higher score. Also, any mistakes may lead to game over as there is only the one city remaining.

3.2 Increasing difficulty

There are multiple things that I've done to increase the difficulty of the game.

- Increased number of meteors every wave. - the number of meteors is increased every level randomly so some levels may have more meteors than the next level. But in general the number of meteors slowly increases every level.
- Bombers start to appear after a few waves and adds another bomber every few waves.
- Meteors have increased mass which causes them to have a higher chance of splitting into smaller meteors.
- Number of child meteors from a meteor splitting also increases with the level

The configuration for different object speeds, spawn rates and scoring are kept as constants in `GameConfig` as a central place for most of these numbers. Some constants are kept in their own respective classes if they make more sense to stay there, for example parameters for which level to add bombers and increase the mass of a meteor are kept in the `Level` class as that is the only class that uses that constant.

I have added an option for a player to "win" the game if they have all their cities alive and pay a large sum of score. The idea behind this is not to have an endless game, but rather have a final goal. For some players, it may be quite an achievement just to reach that goal, while others will try to maximise their score before deciding to end the game.

3.3 City arrangement

The city arrangement is static and in evenly spread locations. This is so there aren't any balance issues with cities being too concentrated in certain locations. The number of cities can be changed and the game is able to place any number of them evenly.

Because the score given for remaining cities is high, I have chosen to have only five cities in the game by default. Having less cities also makes the game a little easier as there are less cities to defend and less to rebuild for the endgame.

Figure 5: Five cities

Figure 6: Eight cities

4 Implementation

4.1 Particles

Almost all the game objects inherit from the abstract `Particle` class. This class contains all the fields that game objects need like position, velocity and the force accumulator. The `Particle` super class also implements the `IDrawable` interface, which means all particle sub classes must include a display function.

This design allows each class to define its own draw function and change it accordingly based on its properties. For example an explosion has increasing radius as it is drawn, or a missile should stop displaying after reaching its destination.

4.2 Physics

Following the lectures and examples, I implemented the forces in physics as a `ForceGenerator`, with the game having a `ForceRegistry` to register all the particles and the forces acting on them. The registry and global forces are kept in the `PhysicsEngine`. Each new particle that is registered will register the two global gravity and drag forces. Any other forces are registered as they are needed, for example an `Explosive` force on a meteor is only registered when the game has checked that the meteor is within the blast radius.

With the `ForceGenerator`, I can implement many different forces which each has their own formula for updating the force. For example `Gravity` is simply a downwards force while `Attractive` uses the proper gravitational attraction formula. The class `PhysicsStep` is an abstraction over what each type of particle should do on each step of the game. For example, meteors and missiles must first `integrate` to work out and move to their next position. Then meteors are affected by other forces like black holes and explosions so those forces must be

added to the register. This allows each class to define what happens in its own step and for the physics engine to take in any generic step and apply it.

4.3 Collision

I use the collision code provided in the lectures to deal with meteor/meteor collisions. To prevent the bug where the meteors would stick together due to them overlapping, I check for the overlap and force them apart before resolving the collision as normal. This collision makes the game a bit more realistic and slightly more difficult because the meteors could collide with each other and make them move into different trajectories.

4.4 State management

To split the game up into different states such as start of game, end of wave, shop etc, I've implemented each state as its own separate class and made the game change states like a state machine. I got this idea from the state chapter of the book Game Programming Patterns (http://gameprogrammingpatterns.com/state.html).

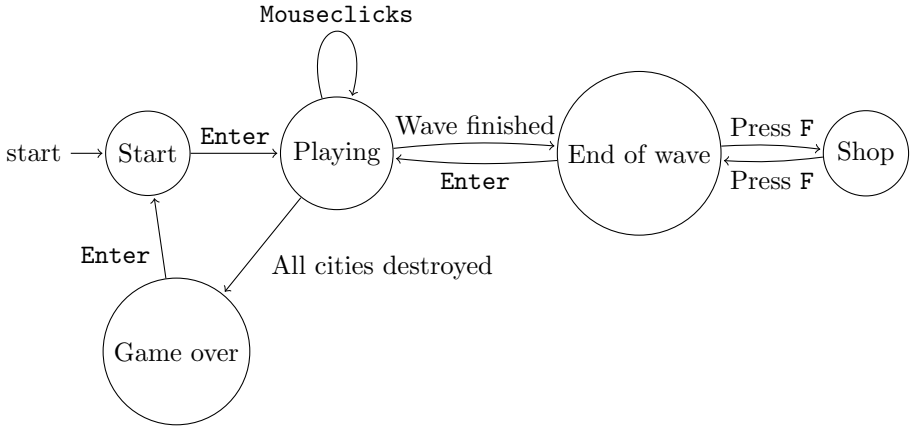


Figure 7: State machine of the game.

The idea is that on each `update()` or `handleInput()` step, the current state will return what the next state should be. For example if the game is currently in the `EndOfWaveState` and it gets the enter key - the key to start the next wave - it returns a `PlayingState` with the next wave of enemies. This allows each state to only keep track of themselves and handle everything that happens in the state within their own class. The controller then does not care about what each state is or what it does, but simply calls the update and draw functions of the current state.

To complement the state machine, there are three classes that encapsulate many game properties. The `GameContext` class is used to encapsulate all information about the game, such as the lists of objects, the physics engine, the current level etc. `GameInfo` is for all information that is displayed to the player, for example the number of missiles left or the score. `GameInput` represents all the input from the player and has fields such as the mouse position and the key that was pressed.

5 Conclusion

In conclusion, I've been able to implement many features into my game. Some features make the game more difficult, but others try to help the player be more successful. The different features allow me to show off a few different forces and how they all interact with each other. Having each force as a `ForceGenerator` and using the `ForceRegistry` to keep track of all the forces and particles make it easily extensible to add new types of forces. The state machine allows each state to define its own functionality and again makes it easy to extend and add new states for different phases or parts of the game. I have also added a final goal for players to try and reach rather than make the game endless so the objective it to survive to a certain point and not just survive forever.