190003657

Video Games
CS4303

Practical 1
12/02/2024

Missile Command

University of St Andrews

Contents

Introduction

The aim of this practical was to learn the Processing language and implement some concepts from the physics lectures. This was done through the implementation of the classic video game 'Missile Command'.

1 Design and Implementation

1.1 Initial U.M.L. Diagram

Before coding, an initial UML diagram was done to plot out what classes were necessary. This is shown in figure 1.

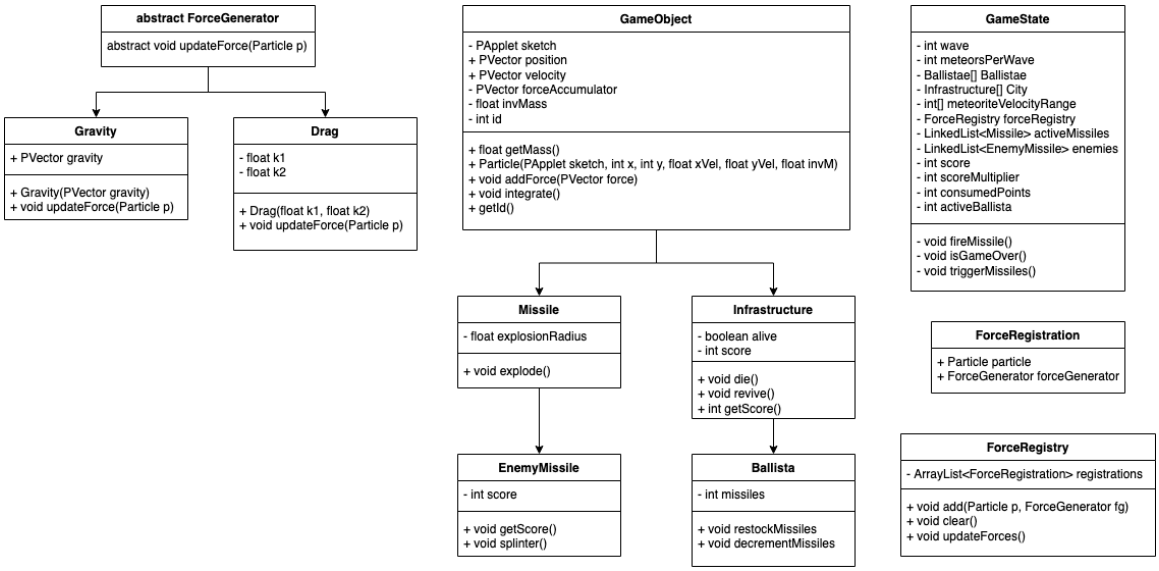


Figure 1: Initial UML Diagram

A general game object class was created to hold the attributes that were thought of as common between infrastructure and missile objects. Infrastructure referring to the cities and missiles anything that is flying through the air like bombers, or the missiles you launch.

The GameState class was planned to maintain all the variables that vary each frame, acting as a centralised location so that you don't have to hop around multiple files to figure out what is being manipulated.

The Force Generator and Registry were taken from the lectures due to them being a cleaner way of handling forces on objects. This is because the number of objects on the screen at once varies heavily, so allowing a force registry to handle all forces once each frame helps to keep things simple and tidy.

The final UML diagram for this project after implementation can be seen in figure 2.

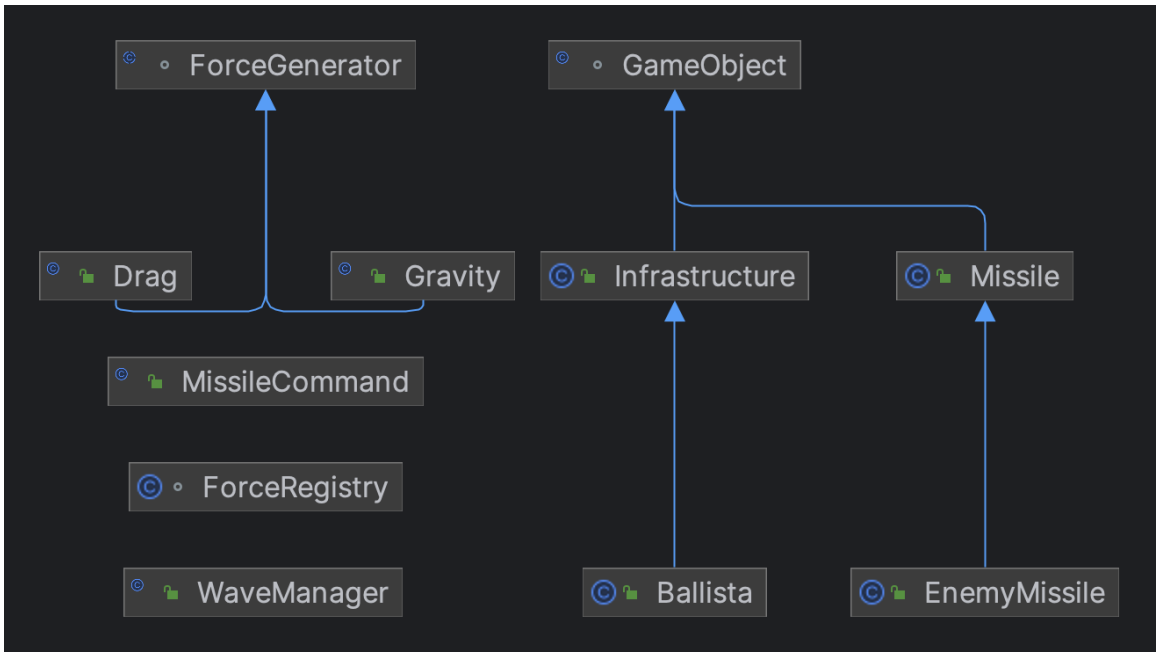


Figure 2: Final UML Diagram

GameState was renamed to MissileCommand, and a WaveManager class was created to handle game parameters like scoring and spawning enemies each wave.

1.2 Attributes

1.2.1 Tuning Attributes

At the start of the MissileCommand class, attributes are initialised for use in the game. Many of these attributes are final attributes used to tune various game parameters like the duration of an explosion or the probability of a meteor splitting.

These were all gathered here so that you would not need to hop around multiple files to edit variable parameters, instead including them all in one place. It also makes it a lot easier to tune them to your preference.

1.3 List Attributes

Many attributes were grouped together into lists as they are related to each other. For example, missiles are grouped in order through a linked hash map so that the detonation order of the missiles would be the same as the sequence in which they were launched. Linked hash maps were used in cases where insertion order matters, and fast accesses were needed. Linked hash maps provides constant time read and writes whilst maintaining insertion order, so they were perfect for the task.

Before an example can be given where this is important, an explanation of how explosions are handled is required. Linked hash maps are used to keep a list of active missiles that a user has launched, in addition to enemy missiles, which are generally any enemies in the air. Active missiles

can be triggered, from which they enter another linked hash map called triggered missiles, where the missiles are primed. In this list, after some set amount of time passes, the first element is removed and then inserted into a list of exploding missiles. This can be seen in figure 3.

Figure 3: Missile Explosion Pipeline

The problem arises when an explosion in the list triggers an explosion in a missile later on in the list, e.g. somewhere in the middle or near the end. You now have two explosions happening concurrently. Linked hash maps solve this issue by allowing you to remove from anywhere in the list in constant time. So, what we can do is remove any missile that is triggered from either triggered missiles or active missiles, and add them directly into exploding missiles, thereby allowing missiles to explode in any order. This can be seen in figure 4.

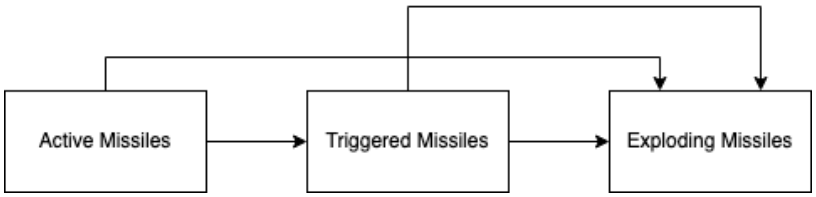


Figure 4: Concurrent Missile Explosion Pipeline

On a side note, whenever explosions are triggered, active missiles are added to the end of triggered missiles to maintain order. Enemy missiles are also added to exploding missiles without a triggering step.

1.3.1 Lag Attributes

A lag attribute was used to create delays in time before some event can occur, as otherwise, the events would happen each frame. In total, there are three lag attributes for trigger lag, explosion lag, and meteorite lag. All lag attributes are incremented each frame, with each have a respective threshold variable upon which an action occurs.

Trigger lag is used for the timing between explosions so that they happen in sequence. It is initially set to some arbitrary high value above the threshold so that when the trigger is pulled for an explosion, the first explosion occurs immediately. Subsequent explosions then have to wait for the trigger lag to again exceed the threshold.

Similarly, explosion lag is used for the opposite purpose for giving a delay before an explosion is removed from the list in which it is held. This is implemented through an additional linked list called exploded after objects in exploding missiles reach the maximum explosion radius.

Lastly, meteorite lag is used so that the spawning of meteors is not occurring one frame after the other. Instead, meteors spawns are spaced in time through this lag variable.

1.3.2 Singletons

In addition to the tuning attributes, singletons, when classes are restricted to one instance, were used. Examples of this include gravity and drag. This is because gravity and drag were set to be the same for all objects, and as such, they can all refer to the same instances of these force generators - thereby saving memory and time from instantiating these force generators.

1.4 Force Registry and Generators

The physics in this game were implemented via point masses for ease of development.

1.4.1 Force Generators

Gravity and drag were implemented through force generators. What this means is that you can pass objects to these force generators and it will impart a force to them when asked to do so.

Gravity does this through taking the gravity vector and scaling it off of a given objects mass. Meanwhile, drag does so by calculating the drag coefficient and then normalising an objects velocity before scaling it using the drag coefficient.

1.4.2 Force Registry

The force registry was implemented as a linked hash map, with the key being a game object, and then the value being a linked list of force generators.

A linked hash map was chosen so that the force registry was iterable and accesses would be constant. What this means in practice is that insertions and removals of objects from the linked hash map can both be done in constant time, saving us time in each draw loop.

A linked list of force generators was used for the value so that each force generator you add to an object can simply be inserted here. Then, if you remove an object the linked list is ejected together with the removed object - leaving less programmer error as the programmer does not need to individually remove the force generators.

Finally, to update forces, you simply loop over the key set of the linked hash map, giving us our game objects, then the force generators for the given game object, and update the force for it.

1.5 Collisions and Explosions

Collisions are implemented through the use of a distance check. As objects are simulated as point masses, we can check another object's position, find its distance, and see if it is within our radius. If the distance is indeed within our radius, then the object is colliding.

In addition to this, a special case for if an object is touching the ground is done by checking if the object bypasses a certain y level.

In either case, when objects collide, an explosion occurs. Explosions can be triggered by either a collision or the trigger button being engaged.

All missile objects use the explode method of the missile class. This keeps things simple as the code can be propagated to sub classes, leaving you to tinker with one method. This explode method takes into account all objects on the screen though. As what occurs when two objects collide and there is an explosion varies between objects.

For explosions, two important attributes, the maximum explosion radius, and the explosion state are used. These two are then used to calculate the current explosion radius by dividing the maximum explosion radius by the explosion state. The current explosion radius is used in the radius check during an explosion.

This is done as when an explosion occurs, it triggers each frame, so with these two variables we can then create an expanding explosion until the maximum explosion state is reached at state 1. In addition, increasing the number of explosion states will also increase the duration of the initial expansion to the maximum explosion state. Another variable is used for the duration of the maximum explosion state.

Examples of different triggers when an explosion occurs with another object include adding to the score if a missile hits an enemy, killing a city on any missile hit (including friendlies!), ballistae emptying their ammunition to 0, and missiles in range of other missiles and its sub class triggering a subsequent explosion.

The subsequent explosion of a missile after another missile explodes is handled by adding the missile to explode into a linked list. This linked list is then built up until all objects are accounted for. At this point, the explosion state is decremented except for when we already reach the max state of 1, and then we return the linked list. On return, these missiles are handled by adding them to a list of exploding missiles, removing them from the force registry and the lists where they were being kept track of in, and then later blown up.

1.6 Trajectories of Enemies and Missiles

1.6.1 Missiles

Some vector manipulation is required when launching missiles and meteorites. For missiles, first we must know the active ballista. This is done through the use of an integer variable that changes depending on a key press corresponding to another ballista. The mouse position is then taken into account along with the position of the active ballista, from which the vector between the two is found by subtracting the mouses position from the ballista's.

With the direction calculated, the vector is then normalised to a unit vector before being multiplied by some initial missile velocity which acts as an impulse. It was decided that missiles would all fire with the same initial velocity for simplicity in game play.

The missile is then created and put into active missiles and the force registry with gravity and drag.

1.6.2 Meteorites

An instance of the wave manager class handles the spawning of meteorites. The initial velocity of the meteor is set through a function which takes the current wave number, then randomly selects a value $\pm$ 10%. The meteor than gets assigned some random x coordinate at the top of the screen and then randomly selects a ballista or city. The position of the meteor is then subtracted from the position of the city to get the direction of the missile. This is then normalised and scaled by the initial velocity randomly selected from before. It is then spawned and added to a list of enemies and the force registry.

1.6.3 Bomber

The bomber is handled by the wave manager and only spawns after a random number between 0 and 1 is found to be lesser than it's spawn threshold. From here, a random position between some set minimum height and maximum height that it may spawn in is selected for its y coordinate. It's x coordinate is set to 0. It is then given a set initial velocity and is not added to the force registry, just the list of enemies. This way, the bomber can glide across the screen as not other forces act on it.

While the bomber moves from the left side of the screen to the right, another probability check is done for spawning a meteorite at its current location. If passed, a meteorite is spawned in the same way as in the meteorites section, except the position is set to the current location of the bomber and is not random.

Bombers are removed once they get off the screen.

1.6.4 Smart Bomb

A smart bomb is initialised in the same way as the meteorite, except with a probability check with which a random value between 0 and 1 needs to be under. The difference comes from when it is in play and falling. Each frame, it tries to detect explosions in some larger radius around it by taking in the exploding linked hash map. If it detects any explosions in this radius, it creates a vector by subtracting the explosions position from itself, giving a vector towards the explosion. This is then inverted so that the smart bomb points away from the explosion. From here, the vector is normalised and then multiplied by a set displacement force for smart bombs so that the smart bomb moves away from the explosion.

This force used for a smart bomb, although occurring over multiple frames until it exits an explosions radius, is not implemented as a force generator. This is because this 'thrust' applied by a smart bomb to itself is inherent to smart bombs and is not an attribute that can be assigned to other objects. If explosive force was something that other objects also experienced and were repelled by like here, the same calculation could then be abstracted to a force generator.

1.6.5 Splitting Meteorites

The chance for meteorites splitting also need to pass a probability check with a random value between 0 and 1. Once passed, the meteor uses the wave manager to spawn a new meteor at its current rotation with a set split radius. This means that the new meteor will be randomly targeting a city or ballista, same as in the meteorite trajectory section. The split probability for the new meteorite is then set to 0, as is that of the current meteorite so that splitting would not occur again.

2 Gameplay

2.1 Controls

Controls are handled on key/mouse released, as otherwise the key seems to spam. This would lead to a lot of unnecessary inputs. In comparison, detecting when a key/mouse is released only gives one event.

Keys used in the game were implemented using a switch so that there would be no conditional overhead like is found with if statements. In addition, the keys assigned are also attributes that may be changed as necessary.

2.2 Wave Manager

The wave manager manages enemies and other attributes of the game like scores. When a new wave is started, the main attributes to be looked at are the meteors per wave, meteors spawned, and the enemies alive.

Meteors per wave limits the meteor count to a set number, from which meteors may be spawned. Meteors are spawned after some lag, and will continue to spawn until it equals the number of meteors per wave. The wave then ends once the enemies alive reaches 0 or lower, and the meteors per wave is equal to or lower than meteors spawned. The equal to or lower is done simply as a sanity check, as it may be that the variables are not handled exactly, but go beyond 0. In this case, the game should still function the same way, hence lesser than or equal to 0.

Scores are added when explosions occur through the add score method of the wave manager class. This takes the score of the blown up object and then uses the get score multiplier method to get the multiplier for the current wave. This is done through a switch statement as it allows quick access without a conditional. However, if more score multipliers were to be included, a formula may be preferable to calculate what the score multiplier should be.

2.2.1 Cities and Ballistae

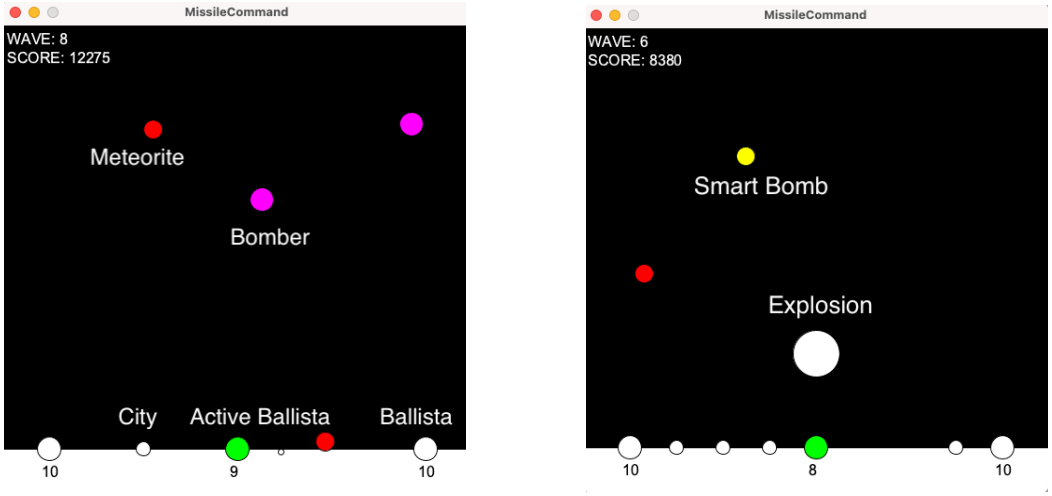
Cities are from the infrastructure class and have attributes like a score which is added at the end of a wave. Cities determine when a game is over through how many are still alive, which is done by a boolean for each city. This is then checked by the wave manager each frame.

If a score of 10,000 is reached, a city is revived. This adds to a consumed points attribute of the wave manager class that keeps track of if a new city should be given or not at the end of a wave.

Ballistae also have a score, but their score is calculated by multiplying the ballista score by the number of missiles it has remaining. This is decremented when a missile is fired until 0, where it cannot be fired anymore. It is also restocked to a set number at the end of a round if the game is not over.

2.3 Graphics

Graphics are simple as the focus of this implementation was on the design and implementation of the physics. The missiles, cities, and ballistae are formed through white sphere of differing radii. Meteorites are red, bombers are purple, smart bombs are yellow, and explosions are yellow too. Examples can be seen in figure 5.



(a) (b)

Figure 5: Labelled Objects in Gameplay

Graphics are drawn through the use of a draw function that each GameObject has.

3 Conclusion

In conclusion, I believe I have written an efficient and correct implementation of the physics and gameplay for Missile Command. Starting off with a UML diagram of how I wanted classes to be structured and careful choice of data structures helped make this practical's implementation tidy and efficient. Next steps that could be taken from here would be to integrate sprites through images for objects on the screen, along with sound effects, a scoreboard, and other similar gameplay features. The game may be a bit difficult for some, but I found the gameplay to be quite fun and fast paced! Overall, I really enjoyed this practical.