CS4303 P1

Physics Game

180015975
Tutor: Ian Miguel

18th February, 2022

Contents

1 Introduction

Ballista Command is set in the aftermath of the original Atari game Missile Command. All the fuel has run out for missiles, but the enemy is instead able to redirect asteroids that can fall down onto the cities that you are defending below. Without missiles, the players use three ballista's to defend against the cascading asteroids from above. Periodically satellites will appear and traverse the screen, dropping asteroids as they go along. Smart bombs, a style of bomb that can dodge explosions unless a shot is perfectly placed appear as well. The purpose of this practical is to implement the physics involved in this game, such as the drag and gravity experienced by the objects.

2 Specification Ambiguities

From the specification, there were a certain number of open-ended questions that needed to be decided upon before designing the game. The specification says "a key or button should be pressed to make all the bombs in the screen detonate. Bomb explosions should happen in the same order as they were fired". This seems slightly contradictory, so the first of the two options were decided to implement, meaning if a key is pressed, all bombs on the screen detonate *at once*. This was partially decided after some game testing where the queued explosions of bombs made for more difficult gameplay.

The additional points added at the end of a round for unused bombs are assumed to be only added on for each ballista that is not destroyed. Also, the game is over once all cities are destroyed in a round. Even if the number of points to rebuild a city are available, cities can only be rebuilt if there is one city left standing at the end of a round.

The specification left whether asteroids, satellites, or missiles interact with one another. The choice was made for these elements not to interact with one another because the player will want their missile to go where they are shooting, and in a similar fashion for some levels of predictability the asteroids should not collide. Hopefully as well the enemy satellites are smart enough to dodge their own falling asteroids, thus asteroids and satellites do not collide.

3 Design Implementation

3.1 Classes

The hierarchy of the classes was designed to reduce the duplication of code, whether this is variables or methods, to give a cleaner implementation. The classes were split into GameObject, Moveable, or PhysicsObject, with the additional interface of a Collidable object. Collidable was an interface because of the realization not all collidable objects require physics, and not all physics objects may be collidable. Due to the way Java handles inheritance, two classes could not be merged/inherited from both like a trait (such as being collidable), but a class can implement an interface and inherit from a parent class, giving the illusion and functionality of another class/trait. As a result, Collidable was handled this way to allow for the choice of which "traits" to inherit from.

The Moveable abstract class is given as well as the PhysicsObjects class because not all Moveable objects have physics applied to them, such as the Satellite class, but all physics objects are moveable, giving flexibility and even further abstraction to group together objects and move them as necessary.

The additional purpose of the hierarchy was to have abstract methods and variables to allow for simple grouping of objects with common methods and attributes. For example, all classes inheriting from GameObjects must have a draw method to be rendered. This means all objects can be placed into a single list and iterated through to draw rather than having separate lists of objects for each type to call a draw method on. Variables could be duplicated across classes, but instead this can be abstracted into classes to be extended from such as the PhysicsObject class. The PhysicsObject class contains the velocity, inverse mass, and force accumulator as variables that each object affected by physics need to use. This has the benefit of grouping together physics objects and calling these common attributes in a list together than individually.

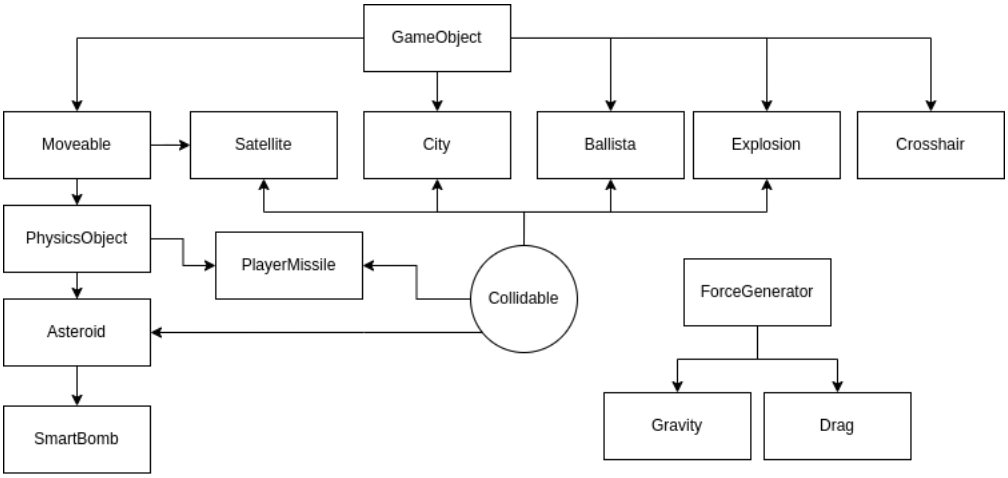

Figure 1: Class hierarchy

3.2 Physics

Physics for the game implements both the full "correct" way of performing drag and gravity. Following the lectures, physics was implemented using abstract classes and inheritance, along with a registry to apply the forces. An abstract class forceUser defines the abstract method all forces must implement. The reasoning is forces that inherit from this class can be grouped together in lists and call the same method, with each separate force adding onto the passed in particle. This simplifies matters by not needing to hold different lists for each type of force when each is called in the same way.

A forceRegistry from lectures holds the particle (the object which forces will be acting upon), and the list of forces acting upon that particle. This allows for simple iteration over the particles and applying the associated forces upon them. Adapting the class from lectures, instead of using a separate class for linking a particle with its list of forces, a HashMap is used instead, hashing the particle to the value of a list of forces to act upon the particle. Using a HashMap gives O(1) lookup when adding in new forces as necessary, as well as O(1) removal once a particle is no longer being displayed (ex. out of bounds). The reasoning for hashing the particle rather than the force is because the particle will need to be removed than a force associated with some particles in the program, gravity or drag will not disappear throughout the running of the program (at least not in this implementation). A particle on the other hand can go out of bounds for example and thus no longer requires forces to be acting upon it.

It should be noted all particles within the registry only have gravity and drag as forces associated with them, and as a result the forceRegistry class is redundant given forceRegistry is useful when there are separate forces associated with separate particles. However, the use of a forceRegistry abstracts out the application of forces onto the particles into its own separate class rather than in the main class, and critically this structure of classes allows for a more complex game and forces to be added if a developer wishes to expand upon this implementation in the future.

All objects with physics are inherited from the PhysicsObject class. This class contains all relevant attributes and methods required for objects to apply physics to such as the current velocity of the object, its inverse mass for calculations, and the forceAccumulator. The forceAccumulator is the vector in which forces applied to a physics object are accumulated to then be applied to its position and velocity when the object begins moving. In the context of this game, the forceAccumulator each computational round accumulates the effects of gravity and drag upon the object it represents to then use further in calculations. Note the convenience of having a force accumulator means not only is gravity and drag applied to a physics object, but additional external forces can be accumulated and applied conveniently. For example, when an asteroid is dropped from a satellite, the asteroid carries with it the momentum of the satellite it is dropped from, carrying it slightly forward in the direction of the satellite before continuing its parabola to the ground.

Gravity is implemented in a straightforward way as Millington and lectures described [1]. An object is acted upon a force using the famous equation F = ma. The resulting force as a result is the mass of the object gravity is applying to, multiplied by the chosen value for gravity (0.1f). This force is then added to the force accumulator of each respective physics particle gravity affects within the forceRegistry. Gravity acts in one direction only, and this is down, thus the acceleration from calculations will always be positive, and will only affect the y-component of the velocity. It should be noted without air resistance mass does not need to be multiplied as acceleration is applied equally to all objects regardless of mass.

Drag, like gravity, is implemented from lectures and the Millington book in a simplified, but still realistic way. To calculate Drag, two constants are used, k1 and k2. These are used in the equation of the force of drag as follows: $dragForce = k1 * speed + k2 * speed^2$ [1]. Constant k1 applies to the object more so at lower speeds, whereas k2 applies itself at higher speeds, slowing the object rapidly down the faster it goes due to the square. The speed of the particle is a simple calculation of the magnitude of the velocity ($sqrt(x^2+y^2)$). As drag is a counteractive force to the direction of travel, the negative of the drag force is multiplied by the normalized vector of the velocity, accumulating onto the relevant particles.

Once all the forces have been accumulated/summed, the calculations may begin. Each object where physics applies requires a mass. Mass values are assigned arbitrarily, where the mass hierarchy goes asteroid ¿ smart bomb ¿ bomb for some realism. Given the accumulated forces have been found (F), the resulting acceleration of the forces needs to be determined. This can be found using the formula $F = ma$ once more. Re-arranging the equation for acceleration, the resulting acceleration is $1/m * F = a$. As per Millington [1], the inverse mass is stored in the PhysicsObject class to speed up calculations to avoid repeating division of mass. The final acceleration is added onto the velocity, and physics is done.

3.3 Game Loop

Following the lectures, the game loop implements handling lag based upon the time taken by the program to compute, render, and handle input.

```
public void draw() {
    double current = System.currentTimeMillis();
    double elapsed = current - previous;
    previous = current;
    lag += elapsed;
    handleInput();
    while(lag >= MS_PER_UPDATE) {
        compute();
        lag -= MS_PER_UPDATE;
    }
    playSounds(); // play game sounds if there are any to play
    render();
}
```

Figure 2: Game loop implemented

Given input is handled asynchronously by Processing, boolean flags are used to determine what input is pressed. These flags are read within a method handlingInput() to enact effect rather than in the async methods to linearize and follow the lecture game loop. All computation is moved into a method compute(), such as computing the physics of each object, checking collisions, changing rotations of the ballistas etc. Computation is wrapped within a while loop decrementing based on the chosen MS_PER_LOOP. The while loop is based upon a static set MS_PER_LOOP, and the choice of 20ms for updates is arbitrary, it is fast enough not to interfere with gameplay, but slow enough to run smoothly all the physics involved as found in testing.

Once this while loop terminates and all the updates on positions are complete, then the call to render GameObjects happens. This while loop for lag is used because it smooths out position updates, so each position update happens in unison together for each frame. If computation and render were mixed, position updates and velocity updates would not be coordinated, resulting in unexpected behaviour to the user with regards to what they see and collisions.

To avoid confusion with the user, objects are moved first, bounds are checked upon those objects, then collision detection is run upon the recently moved objects. What this achieves is no two objects are perceived to be overlapping by the user once render is called because the collision is handled, meaning two objects should never be rendered within one another. The check of bounds of the objects before collision detection means the user should never perceive an object that is out of bounds interacting with an object that is within bounds for whatever reason.

3.4 Collisions and QuadTrees

Collisions are detected through the implementation of quad trees. Quad trees are a simple technique used in video games to split the objects on the screen into quads (groups) of four squares based upon their positions, then splitting these squares further into quads and grouping objects accordingly depending on the number of objects within. This is done such that collisions are only checked within a respective quad, objects outside of a quad

are not colliding with the objects within a quad. This saves the expensive computation of collision detection by only performing calculations with objects within a quad.

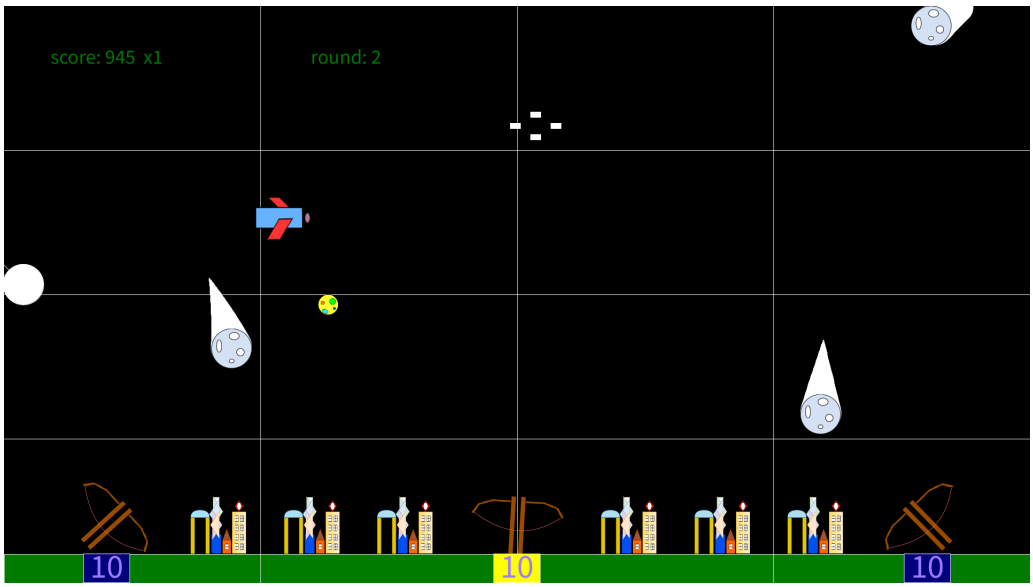

Figure 3: QuadTree during an instance of the game

A QuadTree node is formed of a list of objects within that quad, and any leaf quad nodes within that quad. If an object does not fit into a quad, the object is put into the closest overlapping quads and removed from the parent node. In other words, the overlapping objects are pushed from the parent quad nodes into the children/leaf nodes. For example, if an object only fits into the top two quads, but not in the top left or top right, the object is put into both the top left and top right QuadTree nodes. It is worth noting the object could have been left in the parent node rather than being inserted into the children. However, when checking for collisions the parent node calls the children nodes to check the overlapping quads anyways, so this expediates the process and makes the logic easier, removing the case of checking a parent node against children nodes, and only dealing with a leaf node checking collisions internally. To avoid checking collisions twice, as this can occur by pushing down the parent node objects into the children, boolean flags are passed to Collidable objects to say whether they have been destroyed already so they do not spawn two explosions at the same point.

When checking for collisions, the tree of QuadNodes is iterated through until a leaf node is found (ie. no other quads/node children within the respective quad). The computation for collision detection between these nodes is performed, and if a collision is detected, the method handleCollision is called upon the objects colliding. As some objects are moving, such as asteroids or satellites, the quad tree must be updated upon each movement. The simple approach was taken when dealing with this problem by clearing out the quad tree for each update of positions, then re-inserting all objects back into the quad tree. The disadvantage of doing this is there is plenty of removals of static objects within the quad tree such as the cities or ballistas, a waste of computation. However, the simplicity of removing and re-adding all objects may still save time than the potentially complex and expensive re-arrangement of the tree nodes. Further, this game is very simple with a limited number of objects, and as a result the game can handle the extra computation of adding and removing the small number of these objects.

To determine which objects are collidable, an interface Collidable was created to dis-

tinguish the GameObjects. An interface was used because it allowed for flexibility in deciding what traits an object may have because in Java a class can extend and implement two separate classes. These traits, for example, include an object only having physics applied to it, or only being a collidable object, or both a physics and collidable object. As an interface, the class Collidable requires the implementing classes to implement the boolean method handleCollision(GameObject element). This method determines the outcome of the current object colliding with the object passed in. If there is no outcome or the collision will be handled in a separate class, false is returned from the method. If the collision is handled from this object, such as adding in an explosion and removing the object from display, then true is returned. The use of handleCollision means if one object does not handle the collision, handleCollision is passed onto the other colliding object. For example, if A collides with B, and A returns false on A.handleCollision(B), then B.handleCollision(A) is called instead.

With the use of instanceof and class inheritance/extensions, the handleCollision(GameObject element) element instance is used to determine the outcome of the handleCollision and appropriate response. This clean abstraction makes for simple casting and class checking to handle collisions in a concise way. The other benefit of the interface collidable is when inserting the list of GameObject elements back into the QuadTree, the instance of a GameObject can be checked if it is Collidable and inserting it into the QuadTree.

3.5 Asteroids

Asteroids are an extension of a physics object and collidable, and work simply. Unless an asteroid is dropped from a satellite/bomber, all asteroids drop from the top of the screen at a random point, and they pick a random target to drop towards, meaning asteroids can fly across the screen or simply drop down, making it more fun for the player as asteroids have dynamic and random drops. To simulate trails, the past 30 positions of the asteroid are held and drawn as smaller and smaller circles, simulating the effect of the burning trail of an asteroid as it falls. This is done through a Queue of PVector positions because the front of the queue will be the furthest circle away, and as you iterate over the queue the circles get larger and larger because these are the PVector positions that were most recently added onto the list. This also means the furthest circle away is polled to then be updated on its position.

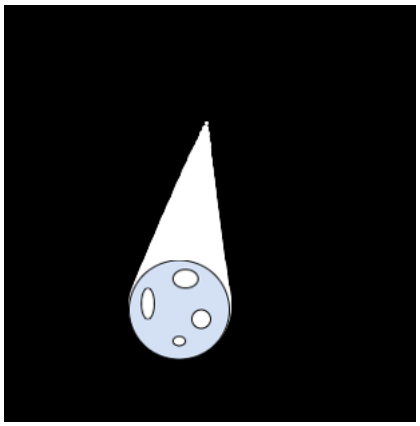

Figure 4: Trails of an asteroid

Asteroids can split as well once the level reaches past 2. To implement this, new

asteroids are spawned at the location of the parent asteroid, ie. the asteroid that is splitting, and a force is applied to the asteroid to move it to either side of the parent asteroid that is splitting.

3.6 Ballista

A ballista is an instance of a GameObject and Collidable object. The ballista class contains two Stacks of missiles, one that is yet to be used and one for object reuse. Once a missile is fired, the missile is popped from the missiles stack and pushed onto the toUse stack. This means the missiles are not created for each round, saving object creation. A stack was used because missiles will be removed each round, and stack removal is an O(1) operation. Ballistas can rotate to point at the mouse cursor, and missiles can only be fired once at a time. What this means is the spacebar key (the firing key), must be fully released before you can press it again to fire another missile.

The initial force of the missile to be fired at is determined by the distance of the mouse cursor away from the ballista. This means the further away the mouse cursor is, the faster the missile initially flies. This is because due to the forces acting upon the missile such as gravity, the missile must fly with more force to reach the mouse cursor. Note the calculation of the exact amount of force is not done given this computation is expensive, so instead a heuristic based off the distance between the point of the mouse cursor and the ballista position is used, giving an idea of the magnitude of the force required to add onto the missile. This heuristic works well enough and gives acceptable aiming for the game to be enjoyable.

To show which ballista has been selected, yellow is highlighted around the base of the ballista, aiding the player visually.

3.7 Smart Bomb

A Smart Bomb is a bomb used by the enemy that can detect and avoid explosions to an extent. It extends from the Asteroid class because it shares many of the properties of an asteroid with a few extra features. It was used in the original 1980 Atari Missile Command, and upon watching videos of the behaviour of the smart bombs, a version of a Smart Bomb following the style of the Missile Command was created for this game. In the original missile command, smart bombs move out of the way of explosions but keep on the same initial projected vector after dodging the explosion. Given our game is set after all fuel has run out, these smart bombs work in an analogous way of dodging the explosion, but do not move back to the original vector because this would require an external force such as fuel. Instead of using fuel to dodge explosions, the smart bomb "rides" the explosion blast to dodge out of the way of the harmful explosion that would destroy the smart bomb. In translation this means the explosion acts upon the smart bomb as an external force, pushing the smart bomb away from the explosion. This is done through simple vector subtraction between the explosion position and the smart bomb position, then giving it some trivial nudge force value to add onto its velocity vector.

To detect the explosions to dodge, the smart bomb is given two radii. The first radius is the dodge radius of the smart bomb, and the second is the actual radius of the smart bomb as displayed. The dodge radius is the radius used in collision detection, meaning if an explosion is detected to "collide" with a smart bomb, the smart bomb handles the collision by applying the explosion force and moving away from the explosion. Of course,

the smart bomb initially checks if the explosion has indeed collided with it using its real radius in the collision check and removing itself and spawning an explosion if it has.

Note smart bombs do not have trails because this makes it harder to predict where the smart bomb is going, like the original Atari game. Also, smart bombs only start appearing around wave 4, giving the player time to settle in and get used to the controls.

3.8 Round Screen

The round screen is where all the round numbers, points, and computations for how long to display the round screen happen. This is a simple class that abstracts out the work from the main class. The only notable decision decision is to keep two separate point scores, one which deducts the city rebuilding score and the actual score. This is so the game knows to rebuild a city for every 10000 points.

3.9 Game Values

The game values, such as object widths and heights, spawn times, number of object intervals etc is all defined at the top of the main controller class. The game values that depend on the screen dimensions such as heights and widths of objects are all given as proportions, meaning the game will be displayed the same on every sized screen. By keeping the game values as final numbers at the top of the controller class, it makes it simple to find and fine tune numbers as necessary that are used throughout the program to change the mechanics of the game simply, abstracting this out and not using magic numbers.

3.10 Game Control

Game control is done all through time. Time is independent of frame rate and thus can be used on any machine at any speed. One critical use of time is in spawning in new waves.

3.10.1 Waves

The chosen implementation of enemy spawns is through waves. This gives coherent structure to the rate of dropping of enemies rather than spawning them in intermittently, creating greater challenge for the player. Enemies such as asteroids, smart bombs and satellites are spawned together as waves from the top of the screen. Each level has a set number of each type of enemy to spawn for that level (ex. 10 asteroids, 1 satellite/bomber for level 2). For each wave there is a set maximum number of enemies. This can be a combination of any of a normal asteroid, a smart bomb, or a bomber/satellite. For example, if the maximum for that level was 3, this could be 2 asteroids and 1 satellite bomber. Each enemy has a range between a maximum and minimum number per wave per level, such as a normal asteroid being able to spawn between 1-3 asteroids per wave. The progression of how the spaces is filled for the objects to fall in a wave are: normal asteroids $\rightarrow$ satellites/bombers $\rightarrow$ smart bombs. This proceeds in the level of difficulty of objects.

What this means for example consider if a wave has a maximum of 4 enemies. A random number is chosen between 1-3 for asteroids, lets say rand(1,3)=1, then satellites/bombers choose from their random number range of say 1-2, but this random number is determined by rand(min(1,4-1), min(2,4-1)). Remember, there can only be 4 enemies

at once per wave and there will be 1 asteroid this wave, so the random number is chosen between the minimum of the range of satellite/bomber and the remaining number of enemy places to fill. This process is repeated for smart bombs as well. This ensures the maximum number of enemies per wave is not exceeded and the enemies chosen to fall mix nicely together. This process is repeated until the total number of enemies per category has been spawned and exhausted.

The time between each wave is given between a minimum and a maximum time, or if the screen is completely empty of enemies so the player is not waiting around. This means potentially two waves could spawn close to one another, increasing the difficulty for the player if they have yet to deal with the first wave.

3.11 Difficulty

To increase difficulty, the number of enemy spawns per level increases at a pre-defined rate, including the number of max enemies that can be spawned per wave. Further to this, the speed at which the asteroids/smart bombs start move faster as the level progresses. There is a terminal velocity given the game implements drag and gravity, however the speeds of the asteroids will exceed the terminal velocity initially, covering the ground faster to the ballistas, then reach terminal velocity and fall as usual. As a result, terminal velocity is not taken into account for the initial speed of the asteroids, not because it was too difficult to cap the speed, but to ensure the difficulty of the game kept going up. For fun the terminal velocity can be calculated for an object by equating the force of gravity to the force of drag upon the object:

$\mathrm{gm} = \mathrm{k}_1 * v + k_2 * v^2$
(1)

Where g is acceleration of gravity, m is the mass of the object, k_1 and k_2 are the drag coefficients, and v is the magnitude of the velocity of the object (ie. its speed). Rearranging the equation we get:

$0 = \mathrm{k}_1 * v + k_2 * v^2 - gm$
(2)

This is now in the form of a quadratic equation and thus can be solved for v:

$$v = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Substituting in the values, this resolves as:

$$v = \frac{-k_1 \pm \sqrt{k_1^2 - 4 * k_1 * gm}}{2 * k_2}$$

Another layer of difficulty is the time between waves decreases between the levels, meaning waves start coming faster and faster. There is a cap put onto the minimum time for a wave spawn otherwise enemies would be constantly falling from the top of the screen, making it too easy to chain explosions together and get lots of points.

3.12 Satellites and Bombers

Satellites and bombers are a class of enemy who move horizontally from one side of the screen to the other, dropping asteroids periodically as they go along like the original Atari game. Asteroids are dropped based between a random minimum and maximum time to drop an asteroid, adding in a better game feel than purely predictable satellites/bombers. The only difference between a satellite and a bomber besides the images used is a satellite is circular, and a bomber is rectangular for collision detection. As an asteroid drops from a satellite/bomber, it must retain the initial horizontal momentum from the satellite/bomber it was dropped from. The amount of force applied is of course found through the equation $F = ma$, where m is the mass of the asteroid, and a is the acceleration provided by the satellite/bomber. This force is simply added onto the asteroid as it drops from the satellite/bomber. Note satellites and bombers begin appearing after wave 3.

3.13 Sound

Sound for this game was implemented using the Minim sound library. Royalty free sound tracks were sourced from online, playing at crucial events during the game to highlight spawns or triggers. The full set of sounds include a destruction sound for a city or ballista being destroyed, a satellite/bomber spawning in, a smart bomb spawning in, explosions, shots being fired from ballistas, and low ammo sounds once a ballista is running low on bombs. These sounds were chosen to draw attention to and notify the player of notable events as they happen.

Sounds are handled by playing them from a list of AudioPlayer before the render is called. When an event like one mentioned above happens during computation or input, the sound is added onto the list to be played during that frame before render. This follows the linearlisation of processing done by the gameloop, handling all the sound at once rather than having it scattered well before render. This is because sounds can be played before an image is seen by the user, notifying them of what is about to happen, rather than letting them know what has already happened. If a satellite is rendered before its sound, the player may be slightly surprised given the audio warning is slightly too late. Of course, we are talking about milliseconds between render and the audio playing, but this could be significant to some.

3.14 Images and Visuals

Images/sprites were added to some game objects to add an extra layer to the game. Unfortunately, due to poor drawing skills and conceptions of hit boxes, the images give a false sense of where the collision detection boundary lies for an image. For example, the cities do not fill the rectangle they are meant to be in, meaning if an asteroid clips the top corner of the city where there is no image but the dimensions say there is meant to, the city is still destroyed. For future reference sprites will either fill the borders or adjust hit boxes accordingly to satisfy the dimensions of the images. These images and visuals are documented in the game guide for reference.

3.15 Object Reuse

Object creation for every spawn is unnecessary, objects can be reused and reset once they are destroyed or go out of bounds eventually. This does require more bookkeeping, such

as having a Queue of toUse objects of each data type (explosions, missiles etc) to iterate and pull from and add to as the game progresses, but the resources saved are worth it. Should all objects in a Queue be used up, then a new object is created and added onto the Queue. The use of a Queue means the objects are rotated through; no two consecutive asteroid spawns should be the same object. This gives greater freedom to the developer should they need to use hashing upon objects for encounter checks for example, where if a projectile is flying and can only interact with a type of object once, if this object is reused and spawned again then unexpected behaviour may happen as it is the same object, but to the perception of the player it is a new object.

3.16 Data Types and Lists Used

The datatype holding all of the gameobjects toDraw is a Set. This is because it allows for O(1) removal when objects need to be removed from the canvas to draw. The disadvantage of using a Set is it has no ordering, meaning if layers were critical to this game, then certain objects may not be rendered, as necessary. Sets were also used for the handling of the missiles, asteroids, and satellites that are currently on display. The reasoning behind keeping these lists instead of handling everything through the toDraw is because it simplifies needing to iterate through the Set each time to find these objects to perform special operations upon. Note originally before abstraction all objects were separated out into individual lists, where each list needed to be iterated over to draw, move and so forth. This saves a few number of iterations by keeping all the objects separate, but the clean abstraction of having a single Set list is by far better. Given there are only a few objects in this game, the extra computation of extra iterations is worth the overhead of keeping things tidy.

3.17 Screens

The game has four simple screens, the welcome screen, a new wave screen, the game screen, and the game over screen. This gives a small improvement of states and flow to the game, this is really only visual artefact.

4 Game Running

It should be noted, many of the game features are difficult to demonstrate through screenshots given the game moves and has interactions. Sound is especially impossible to demonstrate, so please take the time to play the game to get the full experience. See appendix of all the game images.

5 Evaluation and Conclusion

The game runs smoothly, implements all the required functionality and more, and is a challenge and is enjoyable to play. The code is all very well commented on and is relatively well abstracted out, although the files could have been refactored a bit more. Overall, the game was a success, and I am happy with it. Clearly there was far more that could be implemented into this game, such as pausing, displaying extra points, saving scores at the end, restarting the game from the end screen, better graphics, and physics and so on.

But, for the scale of the practical this went beyond the original scope of physics and was fun to create.

Overall, this practical took quite a bit of time to complete due to all the refactors and problems run into such as collision detection, but it was satisfying once it was complete. An immense amount was learned about the physics in games, as well as collision detection and the way this is handled within games. Overall, I am reasonably happy with the outcome, however I truly wished I did more, there was far more to do.

References

[1] Ian Millington. *Game Physics Engine Development*. Second. London: Morgan Kaufmann, 2010.

6 Appendix

Figure 5: Welcome screen

Figure 6: Multiplier of the game working

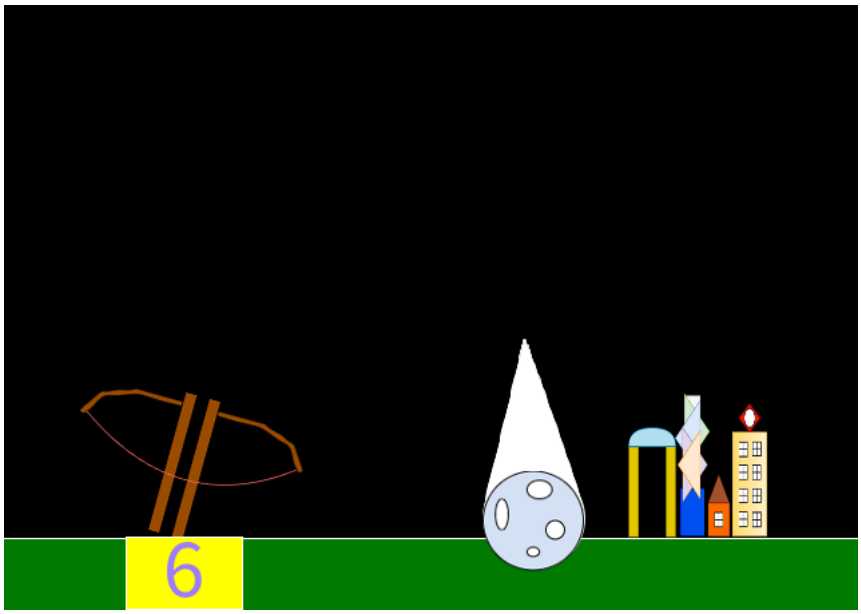

Figure 7: City being destroyed

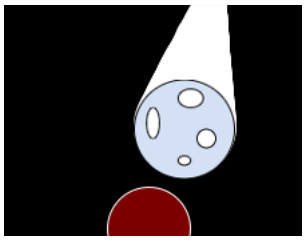

Figure 8: Imminent collision between an explosion and an asteroid

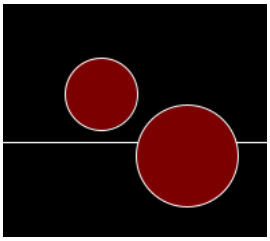

Figure 9: Explosion collision happening over the quad tree, ie. works over two quads

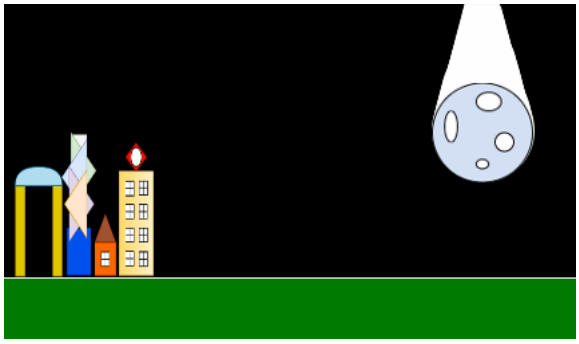

Figure 10: Ballista being destroyed by a asteroid

Figure 11: Low ammo ballista highlighted in red

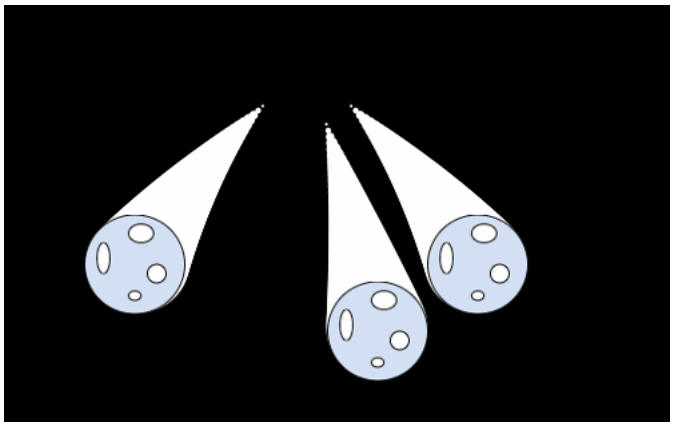

Figure 12: Asteroids splitting

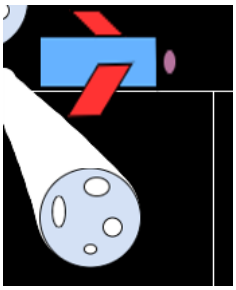

Figure 13: Satellite bomber dropping an asteroid

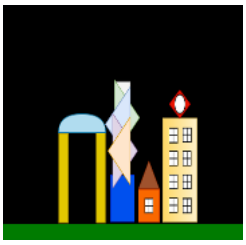

Figure 14: City restored

Figure 15: Score of city restored

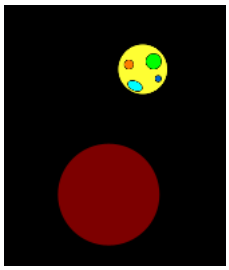

Figure 16: Smart bomb dodging an explosion

Figure 17: Multiplier of the game working

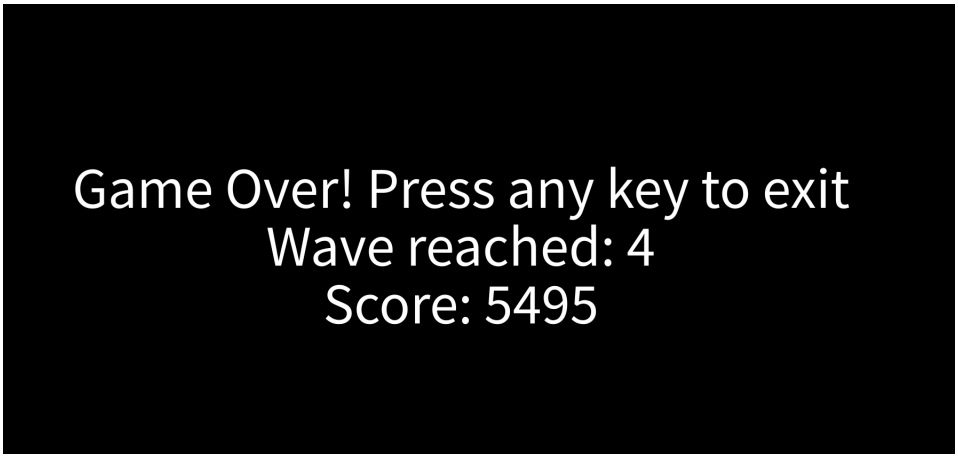

Figure 18: Game over once all cities destroyed