

University
of
St Andrews

**CS4303 Video Games:
Physics Practical: Ballista Command**

Word Count: 6491

Introduction

We have been asked to create a game in the same vein as the Atari classic Missile Command. The game differs in a few ways, however. For example, instead of missiles being launched, the defence utilises bombs. These bombs are influenced by both gravity and drag, unlike the missiles in the original game. The standard enemy threats – meteors - are also influenced by the same forces.

Another key difference in gameplay is that since bombs are being utilised, they must be detonated manually, rather than on-impact like the missiles in the original game.

Some additional features that I have implemented are:

- An active meteor count during wave
- The ability to fire with mouse click, which uses an intelligent firing system
- Bomber/satellite enemies
- Splitting bombs
- Audio for the game via the minim library

Design & Implementation

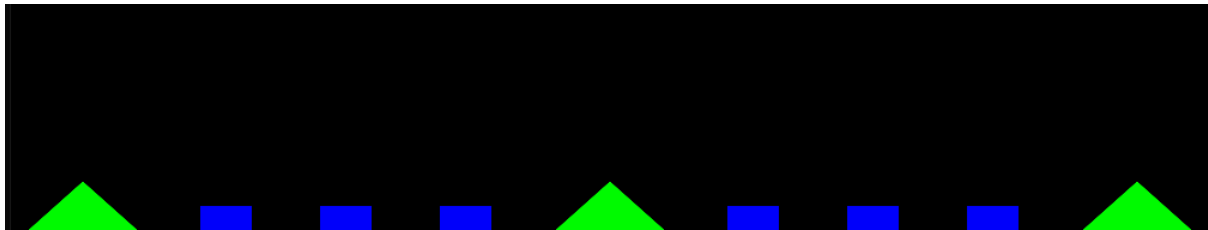
The first aspect of the game that I decided to implement was the first subtitle within the specification – that of the play area. The play area consists of both cities and ballistae. Therefore, I created two classes for the two types of 'buildings'. These, upon original creation, consisted of integer variables for their respective shape requirements and their accompanying draw methods. I chose to make my ballistae triangles, so that they could be easily identifiable from cities and tower over them in a sort of double valley as seen in the original game.

After testing with different dimensions, I settled on a screen size of 1000x700. I felt this was the biggest size achievable (at a good ratio) where the game did not look stretched.

After the screen size has been set up and the cursor has been disabled, I call another function within the setup function. This function, aptly named 'reset', initialises many of the game's variables and is where the game begins. Here, I initialise both the array of cities and ballistae, with the arrays being of size 6 and 3, respectively, much like in the original game. I also created two Boolean arrays of the same size as listed before that will be used to track if the cities/ballistae are alive. These are initially set to all true via the use of for loops. These state arrays are used because of the continuous initialisation of both objects in the main draw function. If a simple state Boolean was used, then it would be constantly set to the initial value. Therefore, the state arrays were used to keep state of the objects separate from the objects themselves.

I created two 'draw' methods in 'BandCHandling' for the filling of the ballistae and cities arrays where at the end, their draw methods would be called. The drawing of the ballistae was the first of the two implemented. Here, two separate functions are required, one to fill the ballista array with hardcoded values and another to draw the ballistae, also handling their visible bomb count. Two separate methods were required as the drawBallistae() function must be called at each frame to keep the bomb count up to date, whilst the createBallistae() function must be called at the end of every successful wave to reset the ballistae back to their original state. I will touch more on the active bomb counts and the checking of both the city/ballistae states

further on in this section. Hardcoded values were used for the positioning of the ballistae as this is possible and simplest to do as each set of coordinates is unrelated to the others, with the left-most ballista's x values being based on the left side of the screen (0), the right-most being based on the right side of the screen (width) and the middle being based on the centre of the screen (width/2). The three y-values for every triangle were the same, with the base coordinates starting at the bottom of the screen (height) and the top coordinate being 40 pixels up from the bottom of the screen. The result was three triangles, all with a height of 40 and a width of 90. The drawCities method was a little more complex as the cities must be placed relative to one another (two groups of three) and thus, a little more work was required. Three of the four rectangle parameters would be the same for every city, as the top-left y-coordinate would all be the same (20 pixels off the bottom) as well as the width and height of the cities, which are fractions of the screen's width and height, respectively. The first parameter, the top-left x-coordinate is where the work is required as this would be different for each city. For this, I figured out a good distance between the cities using a fraction of the overall width, before multiplying by the city's position in the cities array to create a greater distance along the x coordinate as the cities go along. To split the cities into two groups of three, I used an if statement that checks if the city being looked at is after the third position in the array. If this is the case then it adds a greater integer to the overall x value so that the cities can be found to the right of the centre ballista, as opposed to the first groups' which has an integer only applied to it to get it to the right of the left-most ballista. The setup of cities and ballistae can be seen below:




Something to note is that I am aware I use values proportional to the screen size for the cities, but hardcoded values for the ballistae. This would cause a problem if I was to allow for varying screen size. However, since I have decided to go against this extension and keep a single screen size, this is fine. I talk about why I went against this extension a little in my Evaluation section.

The next feature I chose to implement was a non-mandatory one, but one I felt would be easy to implement at the beginning and could be built on for further extensions – the main menu. I implemented this by creating a pair of final integer values to represent the menu state and the game state. A 'screenView' integer would then be assigned the values of the fixed integers to represent what screen was being viewed at the current point in time. Upon execution, the game is on the menu screen. At the bottom of the draw function, a simple if statement is used to check if the screenView value is that of the menu. If that is the case, then a few lines of text are shown to the user displaying the name of the game as well as instructions on how to start the game. As shown on the menu, once the user presses '0', they are taken to the game screen and the game begins. A small thing to note here is the createFont call at the start of the draw function. I made use of this as I found a cool looking Atari-style font online that I wanted to implement, and thus I downloaded it into the game files (can be found in the submission folder) and imported it using said call.



There is also a check in draw to see if the screenView value is that of the game screen's. Within this if statement all parts necessary to be drawn for the game can be found, including the draw functions for the ballistae and city as well as some text that is to be displayed at the top of the screen that will show the user's score along with the round number and number of meteors left.

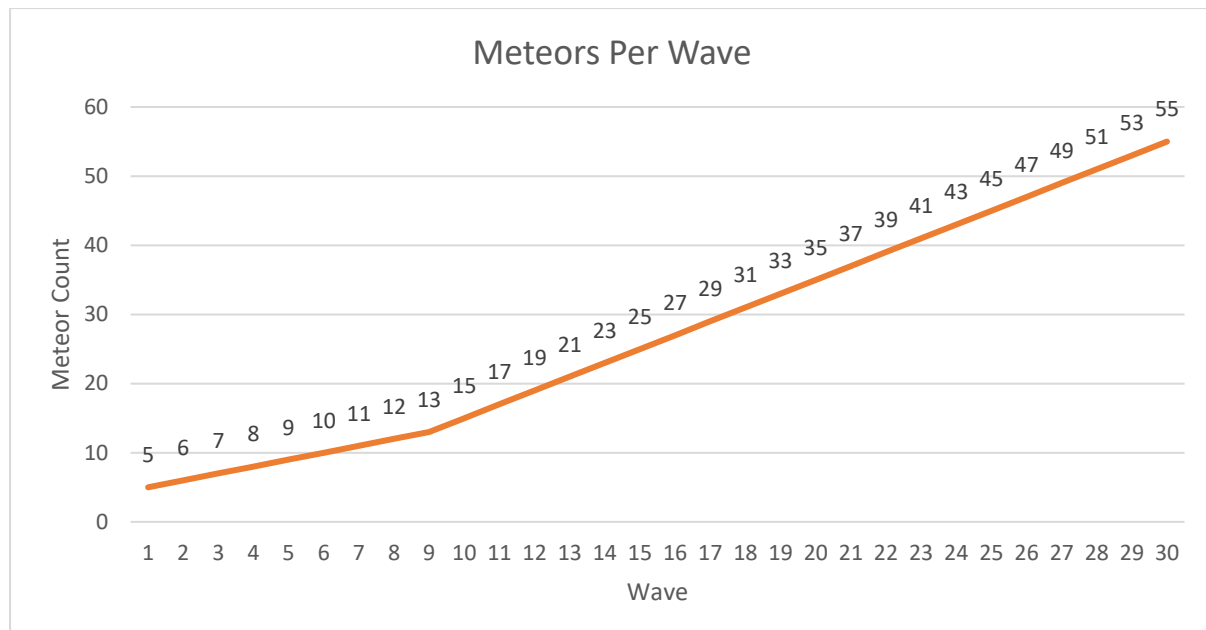


Another feature that is displayed in the draw function is that of the crosshair. This is simply composed of two lines in an X shape, much like the classic game: 

The next aspect implemented was one of the two major aspects of the game – the meteors. When creating the basis for the meteor class, I relied heavily on the files used in the Physics demonstration, specifically taking the drag and inverse mass features from those files. To begin with, the Meteor class consisted of three vectors for velocity, position, and acceleration as well as a float for the drag – set to 0.995 as seen in the lectures – and an integer for the invMass. Again, much like the lecture examples, meteors have an integrate function where the gravity vector (defined as a constant in the reset function) will be passed into it. This function begins with the check for infinite mass, which is not necessary to have in my implementation, but I thought it was good to keep either way. The velocity vector is then added to the position vector before the gravity is added to the acceleration vector before being multiplied by the mass. It is here that I introduced terminal velocity. When testing the flights of the meteors, I found that they were getting too fast, too quickly. This made early rounds much harder and late rounds near impossible to survive unless randomness was on the player's side. Therefore, I came up with the idea of a terminal velocity that meteors can reach. This will increase with each wave, ensuring a gradual difficulty increase, whilst also preventing meteors from being too fast, too soon. I will speak more on the specific of why terminal velocity was implemented in a later section of this design. The terminal velocity check works by seeing if the meteor's y velocity is less than that of the terminal velocity, and if so, adds the acceleration to the velocity. This ensures that once the terminal velocity has been reached, its velocity remains unchanged. The final point of note within the integrate function is that I have used the code from the lecture examples that results in the meteors bouncing off the sides of the screen and staying in play. I decided to include this as it prevents any meteors from going out of bounds, but it also adds a new sense of challenge as players need to predict how the meteors will bounce.

When determining the value of the gravitational constant, I simply spent time changing a ratio of the height of the screen until I found one that made the meteors fall at a rate that I found fair. The meteors were implemented into the game by calling the addMeteors function. This is called in the reset function as well as in the one that checks if a wave has been completed. In this method the wave number is incremented as when this function is called, a new wave is beginning as meteors will be spawning. Then, the length of the meteor array is stored as an

integer. If the meteors array has not been initialised before, then 4 is used. To get the number of meteors in a wave, 1 is added to the previous count – ensuring the meteor count is always increasing at least by one – before $\log(\text{previous count})$ is added. This produces the following results:



I feel these are adequate values, as only after wave 17 do the number of meteors outweigh the number of bombs, resulting in users having to choose wisely when to use their bombs. I feel this is a late game feature, and since the final score multiplier starts at wave 11, I feel this can be defined as 'late game'. I also believe the starting waves, whilst not having too many meteors, also have just enough so that the initial waves aren't seen as too boring as these are also the meteors with the lowest terminal velocity.

Once the meteor count for the wave has been calculated, it is used to initialise three arrays; one to store the instances of the meteors, one to store the Boolean state of the meteors and another to store the state of the meteor's explosion counter. The final of the three arrays will not be touched on until the design of bombs is touched on. A for loop is then used to acquire the values necessary to initialise the instances of the meteors in the meteors array. With each iteration in the loop, the meteorState for that meteor is set to true and the meteor's explosion counter is set to 1. The starting X and Y coordinates of the meteor are also acquired. The starting X coordinate is acquired by getting a random position between the left (0) and right (width) side of the screen. The starting Y coordinate is slightly more complex as it is a random number between the top of the screen (0) and an area above the screen based on the number of meteors there are ($\text{meteors.length} * 100$). This means that all meteors will spawn at the same time, but at varying heights above the screen, adequately splitting up the meteors. This is the reason why the terminal velocity was implemented, as some meteors were spawning so far up that by the time they were showing on the screen, they had been impacted by gravity to such a degree that they were moving at a ridiculously fast rate. Speaking of terminal velocity, this is also where it is set for the meteor, with a fix minimum of $0.9 + \text{a tenth of the wave number}$. This allows for a linear increase in the terminal velocity of the meteors, ensuring a gradual increase in difficulty as the game goes on.

The final line of the addMeteors function simply initialises the meteors, with the previously acquired starting coordinates, terminal velocity, and a mass of 1. The only new values are those of the starting X and Y velocities. The X velocity is a random float between 0 and 3 whilst

the Y velocity is a random float between 0 and 1 plus a tenth of the wave number. I feel like the limits between the potential X velocities are a good range as this allows for some meteors to fly straight down whilst others can fly across the entire width of the screen, creating a variety of different meteor paths that the player must understand and deal with. In truth, the initial Y velocity formula is just there so that the requirement in the specification can be met:

'They should have a random initial velocity, but you should use the wave number to influence the initial velocity of each meteorite so as to increase difficulty in later waves'

As much as I do feel this has an influence on the game when you compare wave 1 to wave 10, for example, I feel the main sense of increasing difficulty in terms of meteor velocity in this game comes from my implementation of terminal velocity instead. This again, is due to the meteors spawning at a potentially large distance above the screen, where they have time to build velocity, resulting in the player not often seeing meteors at their initial velocity. I feel as though this implementation follows the specification but improves upon it in both a controllable and realistic way (terminal velocity is a quintessential part of physics when dealing with gravity) and that I am not penalised for holding my own system over that of the one outlined in the specification.

Another function must be spoken about before I touch on the drawing of the meteors and that is the `removeMeteorCheck` function. This function simply checks to see if the meteors have gone below the visible screen, and if so, sets the meteor's state to false in the `meteorState` array.

The meteors are drawn in the main draw function, where a for loop iterates through every meteor in the `meteors` array, first checking if any meteors are out of bounds using a call to `removeMeteorCheck`, before storing the current meteor's position at a position vector. A check is carried out to see if the meteor is still active (`meteorState[i]`), in which case the meteor's `integrate` method is called using the pre-defined gravity constant, before circle is drawn at the position vector. Also in the for loop are the two draw function calls for cities and ballistae, where checks for meteor collisions are checked.

As mentioned, collisions are checked in the two respective draw functions for cities and ballistae where they utilise the `checkState` methods of their respective classes, which I will touch on first. The city class' `checkState` method simply consists of an if statement to check if the passed in meteor co-ordinates are within the city's dimensions (between x coordinate and width && y coordinate and height). Since the circle's coordinate is the middle of the circle, each check is increased by 9 at either end so that a city is deleted when nearer the edge of the circle touches the city and not necessarily the middle of it. This improves hit detection.

Due to the ballistae being in the shape of a triangle, hit detection is a little more complex as there could be a point between two corners that is not between all corners and thus is not in the triangle. Thankfully, I found a page online that referred to a formula used to find if a point is within a triangle. This formula is known as Heron's formula. The page I found had a java representation of this. Since this is a mathematical formula, I felt that it was acceptable for me to use some of this code as it is the representation of a known algorithm as opposed to something I would have to come up with myself. I have both commented the link above the code as well as linking it in the References section of this report. Back to the draw functions, the cities draw function checks the `checkState` and `cityState` of the city, where if both are true, then the city is drawn, else the `cityState` is set to false. A similar check occurs in the ballistae draw function, only if the check fails, then the ballista's state is set to false whilst the bomb counter is set to 0 (deactivated) as no matter what the ballistae are always drawn.

The live meteor count in the HUD is produced by iterating through the meteorState array, tallying up every true encountered. This value is then set back to 0 after it is displayed in the draw so that it can be refreshed with each frame drawn rather than continuously increasing.

The next aspect of the game that was implemented was that of waves, scoring and the loss condition. This is done by calling the waveStatusCheck function in the main draw so that the status of the wave is continually checked for. The status of the round is represented by an enum of three possible values: won, lost or ongoing. This value is produced by calling the waveStatusCheckCondition function within waveStatusCheck. In this function, an initial for loop is used to see if a city is still alive, if that is the case, then a check is carried out to see if there are any active meteors. If both are true, then that means the wave is still ongoing. If there is still a city alive, but no active meteors, then that means the wave has been won. Finally, if there are no alive cities, then that means the wave (and the game) has been lost.

With the state returned, a check is carried out to see if that state is won or lost. If the wave has been won, a multiplier for the score is acquired based on the wave number using the scoreMultiplier function. This function is a simple series of if statements checking what score bracket the wave is in, returning said multiplier.

A for loop runs through the cityState array, counting the number of trues to gather the total number of cities alive at the end of the round. The same is done for the ballistaeState array, where if true, the ballista's final bomb count is added to a total. This is done as at the end of every wave, the player's score is increased by the number of cities left alive and bombs left unused. A check is then carried out to see if a user's score has surpassed 10,000. If this is the case, then a check is carried out to find a dead city. This city's state is then set to true again, so that it can be displayed again. The score is then deducted by the 10,000 so that the player must gain another 10,000 to get another city revived. However, to still display the score accurately, a counter is used to total the number of cities revived. This counter is then multiplied by 10,000 in the main draw function and added to the current score to display the overall score earned by the player. A Boolean is also set to true to notify a city has been revived and the current time is acquired. This is needed as I display the message that a city has been rebuilt for a set time. This is achieved in the main draw function by checking if the Boolean is true. If this is the case, then for 2 seconds ($\text{current time} < \text{time taken} + 2000\text{ms}$) the message is displayed before the Boolean is set back to false. The final few steps in starting a new round are to set the enemy counter back to 0, before calling the addMeteors function for the next round's meteors to spawn. Finally, every ballista is set back to active with a full bomb count, whilst the bomb array is emptied with the explosion counter array being reset so all explosions' sizes are back at 1.

If the wave has been lost, a simple game over message is shown to the player along with the wave number and their score. They then have an option to play again or return to the main menu using key presses.

Once I had meteors spawning in waves that were able to deactivate ballistae and destroy cities, it was time to implement the bombs to blow them up. The bomb class is very similar to that of the meteor class, with the same three vectors, drag constant and inverse mass. The integrate method also works much like the one found within the Meteor class, except without the terminal velocity aspect. I also decided that bombs would not bounce off the sides of the screen, unlike meteors, as if a bomb were to hit of the side of the screen, then that is due to a poor shot from the player and thus, they should be penalised. The only extra aspect of the bomb class that should be noted is the Boolean 'active'. This Boolean simply tracks if the bomb has been set off (detonation button has been pressed after the bomb has been fired) and is required for the ordered, multi-frame explosion.

An array of bombs is initialised in the reset function along with an int array of explosions, which is initialised, with every value being set to 1. I have decided to implement explosions as an integer value, this will be the radius of the explosion of the bomb. This will be used when drawing the circles for the explosions and will be incremented at each frame, creating the visuals of a growing explosion.

To fire the bombs, I first had to implement a method of firing. The specification mentions three separate keys for the three ballistae, but I thought this was a little unnecessary at points, so I also implemented a mouse click method. I touch on the utilisation of both methods in the Evaluation section. The three-key system was the easiest of the two methods to implement as this simply consisted of a check to see which key was chosen, seeing if that ballista had bombs left and calling the fireBomb function. It is important to note that both methods of firing contain if statements to ensure that the current wave has not been lost and that there are still bombs that can be fired (`bombsInPlay < 30`). This prevents users from exploding meteors after they have lost the game to increase their score and from firing a bomb that does not exist within the bomb array, avoiding exception errors.

The mouse method was a little trickier as I implemented the feature that when the ballista closest to the crosshair runs out of bombs, the next closest ballista will fire instead. This adds more functionality to the mouse click method of firing. The mouse click function starts the same as the key pressed one, with the same if statements and coordinate initialisation, however it differs when choosing a ballista as that is based on the X coordinate of the crosshair. Then a series of if statements are carried out to see if the chosen ballista has no bombs left. If this is the case, then the next closest ballista is chosen and is checked for bomb count, where the third ballista will be chosen if needs be. The left-most and right-most ballistae are dealt with a simple if/else statement each, but the middle ballista was a little more complex as depending on the crosshair's X coordinate, what ballistae would be second and third would be different. Again, once the ballista is set, the fireBomb function is called. The fireBomb function simply calls the createBomb function for the ballista chosen before adding said bomb to the overall bomb array and incrementing the counter for the bombs used each round. I believe the only real point to note for this function is the values used for the starting velocity of the bomb. As the specification states that the initial velocity is based on the placement of the crosshair, I take away the position of the top corner of the ballista from the x and y coordinate, before dividing by a constant. The constant is greater for the x coordinate as I believe horizontal velocity should be reduced more to create a more realistic 'launch'. The result of this means that the greater the x and y value aka the further away the crosshair was from the chosen ballista, the greater the initial velocity for the bombs. I chose the dividing constant to be the values that they are as they were the greatest denominators that still allowed the bombs to reach where the crosshair was placed at the time of launching the bomb, something I feel is important for an accurate aiming system. I made the denominators as big as I could as I didn't want the bombs flying off-screen due to too great an initial velocity.

The bombMovement function is called continually in the main draw function as this is responsible for continually drawing the bombs and calling the integrate method to update the bombs' vectors.

Another function that is called continually in the main draw function is that of the bombSetOffCheck. This function has two real purposes: to activate (set to explode) any bombs that are currently in flight and to call the explosion functions if the bombs are still yet to reach their full explosion size. The first part is implemented by checking if the user has pressed the spacebar – this game's detonate key. Once a spacebar press has been detected, for every bomb that has been initialised and currently not exploding, its active Boolean is set to true,

indicating that it is set to explode. The second part of this function runs through every bomb, and for the activated bombs, will call the explosion function to visualise its explosion, until the explosion size limit (60) is reached. To create the staggered effect of the bombs exploding, an if statement checks to see if the bomb previous' explosion size has reached 20 yet. This means that bombs will explode after 10 frames between one another (explosion is incremented by 2 at a time) or instantaneously if the bomb before it has been set off in a previous explosion. Of course, there is no bomb before bombs[0] and thus an if statement is required to handle that one separately to prevent an ArrayOutOfBounds exception. The bomb status is also set to false in these if statements so that the bomb explosion does not move in the same direction as the bomb was going, and instead remains where the bomb blows up. I originally had it so the explosion continued to move, and as much as this made the game easier as the explosions cover a larger portion of the screen as they continue to travel, it looked a little 'off' and thus it was replaced by the stationary explosions.

The explode function is made up of an overarching if statement that checks to see if a bomb is still to explode fully (explosions[i] < 60). If the condition is met, then a representation of the explosion is drawn with a radius equal to that of the current value of explosions[i]. A for loop is contained within the if statement to see if the bomb's explosion is colliding with any meteor. This is done using the explosionTouching function which returns a Boolean. This function consists of two if statements to see if the coordinates of the meteor are within the dimensions of the explosion (bombs[i].position.x + explosion[i]). The additional '20's referenced in the if statements are there to help improve collision detection from play testing. If a collision is detected, then the colliding meteor's state is set to false, its Boolean to track whether it is to explode is set to true and the score is increased by the standard 25 points per meteor multiplied by the result of scoreMultiplier. To help reduce the time the explosion is on screen for, whilst also maintaining a larger explosion size, the explosion is incremented by 2 every loop. Once the max explosion size has been reached, the bomb's active status is set to false to signal the explosion is done with.

Much akin to the bombSetOffCheck function, the explodeMeteorCheck function is continually called within main draw function. This function works exactly like the explode function, drawing the explosion if it is still to be displayed and checking for collisions with meteors. The only difference between this and the explode function is that it deals with meteor explosions instead of bomb explosions and thus handles different arrays. The explodeMeteorCheck also uses a collision check function like explosionTouching, only specific for handling two instances of the meteor array as opposed to a bomb and a meteor.

The first of the additional features implemented was that of the bombers and satellites. Since these enemies are functionally the same, I implemented them using the same class. This class consisted of the two standard vectors (no acceleration needed as the additional enemy moves at a constant speed), two Booleans to track if the additional enemy was alive and if it was currently exploding, an integer to track what type it was (bomber or satellite) and an integer to track its explosion size. The final aspect of the additional enemy is an array of integers which defines where the additional enemies will fire their meteors if they can. The integrate method for the additional enemy is incredibly simple as it moves at a constant speed along the x coordinate, keeping a constant y coordinate.

In the main draw function, a check is carried out to see if the wave one has been completed as it is on wave 2 onwards that the additional enemies can appear. If that check indeed passes and there are no enemies (start of the round), then the addAdditionalEnemy function is called.

The addAdditionalEnemy calculates flips two coins as there are to X or Y decisions that need to be made; what side the additional enemy spawns on and what the additional enemy is

(bomber or satellite). Once, these coin flips have occurred, the additional enemy's Y coordinate needs to be decided. To continually improve difficulty, the additional enemies spawn closer and closer to the cities with every passing round, until a set limit is reached. Based on what side the additional enemy is spawning, it is initialised with those X and Y coordinates. Finally, the fireAt array is filled with random X coordinates that can be anywhere along the width of the screen.

The additional enemy's movement is carried out by continually calling a function in the main draw that calls its integrate and draw methods.

Whilst I originally wanted to implement the additional enemy's meteors as their own individual entities, I found issues with theorised implementation that are discussed in the Evaluation section. Instead, I decided that an additional enemy would fire meteors when there were meteors in the main array that were able to be replaced (destroyed/out of bounds). This allowed for the additional enemy to carry out its purpose, without the total number of meteors in a round increasing to a possibly difficult level too soon. The firing function works by continually checking if the additional enemy is at and of the fire at x coordinates. If this is the case, then a meteor is searched for that has been used (!meteorState), it is initialised again at the coordinates of the additional enemy, its explosion counter is set back to 1 and its state is set back to true. This essentially spawns a new meteor at the additional enemy, producing the 'firing' effect.

The additional enemy follows the same collision detections as meteors. There are also lines added to the bomb explosion and meteor explosion functions to manage what happens if they collide with the additional enemy.

The splitting meteors additional feature was implemented in much the same style as the additional enemies, where meteors would only split and create new meteors if there were meteors available to be re-initialised. The meteor class now has two new integers, a number between 1 and 10 (where 1 represents a meteor can split i.e. meteors have a 10% chance to split) and a Y coordinate at which the meteors can split. A continual check is carried out after wave 1 in the main draw function where the split function is called. In this function, each meteor is checked to see if their split value is 1 and that they are at the meteor's split Y coordinate. Then, a check is carried out to see if there are meteors that can be initialised again. The maximum number meteors that can be re-initialised is 2, so at most a meteor can split into three meteors. After two meteors have been initialised, a break is used to prevent any further from being initialised.

Testing

Something that came around through continual testing of my game was the accurate collision detection between explosions and enemies as well as meteors and cities/ballistae. Whilst implementing other features, I would play test my game. During these tests, if I noticed explosions occurring when they shouldn't or vice versa, I would readjust the values between which explosions would be set off or cities would be killed, etc.

Along with the report and the player guide, a video can be found in the same folder. This video is a game demo. Below I will list timestamps of the demo that show off different features of my implementation:

00:18 – A chain of explosions

00:45 – A satellite spawning

00:53 – Satellite firing a meteor

1:37 – Satellite being destroyed

1:47 – A city being destroyed

2:06 – A bomber spawning and being immediately destroyed

3:05 – A bomber firing a meteor

4:08 – A meteor hits a ballista, deactivating it for the round

4:58 – A meteor splitting

5:10 – A city is rebuilt

6:51 – Game over

Evaluation

As previously mentioned, one potential extension that I originally attempted, but eventually gave up on was that of a varying screen size. This would have worked by activating an option in the main menu that switches the screen from the fixed small screen to full screen. However, when researching this extension, I found a lot of information going against this idea, such as how `size()` and `fullscreen()` can only be called on setup and not during execution. I did find one forum, that spoke about having two unique screens for the game, where toggling the option in the main menu made one visible and the other invisible. This seemed rather complex and a strange workaround for such a small non-mandatory feature. Thus, I gave up on my attempts to implement it.

When reading the specification, I felt that a three-key ballistae system would have been complicated when compared to a simple mouse click to fire from the nearest ballistae with bombs remaining. Therefore, I implemented both methods. However, I later found that both implementations have their own advantages due to the velocities that bombs are fired at based on their distance to the crosshair. This meant that if I wanted a bomb to travel as fast as possible to a target, place the crosshair across the screen to get the highest initial velocity. This would be impossible to do if only the mouse click method was implemented and that does not allowed choice of ballista. However, the mouse click method did simplify easier shots, such as ones where the meteors were directly overhead. I think this is interesting as if it was not for its mandatory nature, I would not have implemented the key press method of firing. Instead, my game now offers more functionality and since both methods have their own benefits, a higher skill ceiling for those who understand how to utilise both methods.

Due to my game initialising meteors at the start of the round, I would be unable to add any meteors to this array mid round from bombers/satellites as the array was of a fixed size. Therefore, my original plan was to handle potential bomber/satellites in their own meteor array. However, this resulted in a lot of code replication which doubled the size of many files. In the end because of this, I opted for the implementation that can be found within the submission.

The only real difficulty I had with implementing the audio for the game was that my original files were not being loaded in properly. After testing to ensure that it was not the file name/location was the problem, I realised that it could have been the format. Since the examples of audio for Processing that I read were all using mp3 files, I converted my files to

that format. After this was done, I faced no more issues with locating and loading the audio files.

Conclusion

I am very pleased with my submission for this practical. I have implemented the additional features asked for and then some. I also think this practical has been the perfect introduction for myself for Processing as I had no prior experience before this module. I am hoping to kick on in this module from this practical and apply my learnings from this on any future practicals and work utilising Processing and in-game physics particularly.

If I had more time, I would have loved to have implemented a high score screen. Unfortunately, time just did not allow for me to be able to do that with another deadline on the same day as this.

References

Help with rect() - https://processing.org/reference/rect_.html

Help with triangle() - https://processing.org/reference/triangle_.html

Atari font - <https://www.fontspace.com/press-start-2p-font-f11591>

Triangle hit detection help - <http://www.jeffreythompson.org/collision-detection/tri-point.php>

Bomb exploding sound - <https://freesound.org/people/derplayer/sounds/587196/>

Meteor exploding sound - <https://freesound.org/people/derplayer/sounds/587185/>

Additional enemy exploding sound - <https://freesound.org/people/derplayer/sounds/587184/>

Bomb firing sound - <https://freesound.org/people/InspectorJ/sounds/448226/>

Game startup sound - <https://freesound.org/people/EVRetro/sounds/583627/>

Game over sound - <https://freesound.org/people/jivatma07/sounds/173859/>

Coin flip for randomiser - <https://stackoverflow.com/questions/21246696/generating-a-number-between-1-and-2-java-math-random>