**IN2026: Games Technology – Game Demo Coursework**

**Part I**

Plan

I have chosen option B, to create asteroids that split into smaller asteroids when hit. My plan was to create a second type of Asteroid class called Asteroid\_Small. Asteroid\_Small would have the same properties as Asteroid but rather than be created when the level begins, they would be created once an Asteroid is destroyed. I also planned to not have an explosion when the first asteroid is created, but to have the explosion occur when the smaller asteroid is destroyed. This would require the creation of classes for the small asteroids, then changing the onObjectDestroyed() method so that when an asteroid is destroyed, these are instead created. We then have to set the points to increment for every small asteroid destroyed and for the level to move on when all of these are created.

Process

The first step is to create two files named ‘Asteroid\_Small.cpp’ and ‘Asteroid\_Small.h’. Given that both Asteroid\_Small and Asteroid have the same functions and the same appearance, we can use the same code found in the Asteroid files, creating GameObject functions for CollisionTest and OnCollision. We must also remember to include ‘Asteroid\_Small.h’ in the Asteroids file.

Next, we have to create the small asteroids. In the Class Asteroid we can create a method called CreateAsteroids\_Small using the similar code to the CreateAsteroids method. However, we must adapt the code to generate an asteroid of half the size. We also must add a line of code to get the location of the destroyed asteroid to ensure the smaller asteroids appear in the correct spot. We can ensure this by including a reference to a pointer of the asteroid and then getting the position of the asteroid and assigning it to our smaller ones

Text

Description automatically generated

So now we have two methods, each of which creates each size asteroid. Now we need to have to stop an explosion happening when the larger asteroid is destroyed and instead have two smaller asteroids spawn in the same place. We also must ensure that when the smaller asteroids are destroyed, the explosion then takes place. In order to do this, we can go to the method ‘OnObjectRemoved’. Now we can check if an asteroid is destroyed and if it is, we can call the create asteroid method for two asteroids including the reference to the current object (the asteroid being destroyed) to give us the correct location. If a small asteroid is destroyed, we can use the same code originally found if an Asteroid is destroyed, meaning that on its destruction an explosion takes place.

Text

Description automatically generated

Running the code should mean that when large asteroids are shot by the ship, no explosion takes place and instead two smaller asteroids appear in their place. When these are shot, they should be destroyed, and an explosion should occur.

However, now the small asteroids will collide with larger asteroids and destroy one another. So, we have to adapt our collision system to prevent this. In Asteroid\_Small.cpp we can add a line of code to prevent a collision occurring if the small asteroid collides with a large one. (Shown in bold)



We now must do the same in Asteroid, adding a line to check if the collision occurs with a small asteroid and if it does, cancel the collision. This is because collisions occur for each of the objects colliding. The text to add is shown in bold here.



Now each type of Asteroid should not collide with one another. As the bullets are set to destroy anything, we do not need to adapt the bullet class. But right now, if the ship collides with a small asteroid, the ship will not be destroyed, only the small asteroid will be. So, we must add this line of code here shown in bold here, which will destroy the ship on a collision.



Finally, we have to adapt the points and levelling system. Currently, when an Asteroid is destroyed, the points counter is updated with +10 points and when all 10 Asteroids are destroyed, the level is updated. But since all asteroids must be destroyed and then all small asteroids are destroyed, we must alter this system. First, we can adapt the variable mAsteroidCount, which keeps track of all asteroids in the game. Since each level incrementally increases the number of Asteroids, we can add this line in bold, to double the number of Asteroids that must be destroyed. Then we can set to decrease the mAsteroidCount when small asteroids are destroyed but for nothing to happen when Asteroids are destroyed.



We can then add this statement here to the code for when small asteroids are destroyed. This means that a new level will be started when all small and large asteroids will be destroyed.

Text

Description automatically generated with medium confidence

Now we have to adapt the score. In the scorekeeper class, add a statement to check whether an object is a small asteroid, if it is then we can increase the score by 5.

Graphical user interface, text, application

Description automatically generated with medium confidence

Now our task of creating small asteroids when normal ones are destroyed, is completed.

Background pattern

Description automatically generated

Evaluation

When creating the small asteroids, I had to decide how to change the levelling. The game progresses through levels when the Asteroids are destroyed. Initially, I kept this however, in order to fully recognise the functionality of the powerup, I changed the code so that the level ended when all the small asteroids were destroyed instead. I did this by changing the mAsteroidCount required for the level update to 2 times the number of asteroids that spawn. This means the level only changes when all asteroids and small asteroids are destroyed.

**Part II**

Plan

I have chosen option B, to implement a bonus/powerup system. This will take the form of an Hourglass powerup, which will spawn at the beginning of each level. When it collides with the player’s spaceship, all asteroids and small asteroids will freeze for five seconds, during which a timer will appear counting down the seconds. This will allow the player to more easily shoot the asteroids. When the timer runs out, the asteroids and small asteroids will return to their original velocities and the timer will disappear.

Process

First, we have to create two hourglass classes, hourglass.cpp and hourglass.h. As the hourglass is an extension of GameObject, we can populate hourglass.h and hourglass.cpp in a similar way to asteroid and Small\_Asteroid, using less code however as we do not need to set any angle or velocity. Including GameObject functions for CollsionTest and OnCollision We can then include Hourglass.h in the asteroids.cpp file. We also must set our collisions so that the hourglass only collides with a spaceship. This means excluding bullets and asteroids from activating collisions. We can also set that the hourglass will not cause a collision with the other objects in their respective methods.

Text

Description automatically generated

A picture containing sandglass, object

Description automatically generated

Now we have to create the Hourglass in the game world. To do this we can write a method called CreateHourglass() in Asteroids.cpp. We can create a shared pointer to Hourglass (this will be useful later when deleting the hourglass) and then set a bounding sphere around it to handle collisions. We will define all of this in Asteroids.h. We also need to set our visuals for the Hourglass. In the process of making this I took a royalty free image for use as our powerup. As shown here.

We can then use the code shown here to add in our hourglass. We bring in the animation file (stored in the assets folder) as well as creating this CreateHourglass() method.

Graphical user interface, text, application

Description automatically generated

We then simply call this method when creating a level. Now when we run our game, we will have an hourglass in a random position. (We can set the number of asteroids to zero for now whilst we implement the powerup).

Graphical user interface, application

Description automatically generated

Our powerup lasts for 5 seconds and has two parts. The freezing of the asteroids and the timer which counts down for us. Given the latter will help us test the former, we should create our timer now. In the method ‘OnObjectRemoved’, we can add a statement for an Hourglass being destroyed. When it is, we can create two timers, on called START\_HOURGLASS\_POWERUP, which begins immediately, and another called END\_HOURGLASS\_POWER, which begins after 5 seconds (5000 milliseconds in the code). These must be defined in Asteroids.h each given an individual ID.

Text

Description automatically generated

Then, in the ‘OnTimer()’ method we can create two statements for both the start and end of the powerup determined by the timer. Now in the method CreateGUI() we can create our timer. We can create the label ‘mTimerLabel’ (remembering to define this in Asteroids.h), setting its initial value to “Time Remaining: 5”, aligning the label to the top-right of the screen to avoid the other labels, and setting the label’s visibility to false.

A screenshot of a computer

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generatedBack in the ‘OnTimer()’ method, we can set the visibility of the label to true at the start of the powerup and false when the powerup end function is called. Meaning that when the ship collides with the powerup, it disappears and the timer label appears, 5 seconds later this is removed.

Text

Description automatically generated with low confidence

Now we must make our timer count down. To do this we must keep track of what number must be displayed. We are going to create a variable of type int called ‘time\_left’ in asteroids.h, along with a getter, setter (that sets the value to 5) and a decrease function that decreases the value by one on each call.

Text, letter

Description automatically generated

Text

Description automatically generated

We can then create a new set of timers in START\_HOURGLASS\_POWERUP called TIME\_DECREASE, calling one every second between zero and 4 seconds. We can also create TIME\_DECREASE as a function within OnTimer(). Each time this is called we can get the value of ‘time\_left’ and set this as the text in the label. After which we call ‘decrease\_time\_left’ to prepare for the next call. Now we have to call ‘set\_time\_left’ at the beginning of the powerup to set the value to 5.

Text

Description automatically generatedText

Description automatically generated

Now when we collide with the powerup, a timer will appear that will count down from five to one and then disappear. Not only is this half of the powerup functionality completed, but we can use the functions just written for knowing when to freeze and unfreeze our asteroids.

A picture containing graphical user interface

Description automatically generatedGraphical user interface, application

Description automatically generated

Now we can set the freezing of our asteroids. In order to do this the code must be able to find all asteroids and small asteroids in the game. To do this we can create a vector of shared pointers for both the asteroids and small asteroids and search through them with a for loop whenever we need to adapt them. We can create two vectors ‘storingAsteroid’ and ‘storingAsteroidSmall’ which take shared pointers of game objects.

A picture containing text, indoor, screenshot

Description automatically generated

We can then create two methods ‘AddToVector’ and ‘AddToVectorSmall’ which we can call whenever a new asteroid or small asteroid is created. This will store the pointers to each asteroid in the game within the vector.

Graphical user interface, text, application

Description automatically generated

Now we have to store the velocities of these asteroids. This way once the powerup is finished we can loop through the vectors of the pointers and re-add the velocities we removed earlier so that the asteroids simply ‘unfreeze’ after the powerup is over. Since the velocities of the asteroids are unchanging, they can be stored as fixed values. We can do this by creating two more vectors ‘storingVelocity’ and ‘storingVelocitySmall’. Like with storing the pointers to the asteroids, we now will store the velocities of the pointers when creating a new asteroid.





Now we can go to our START\_HOURGLASS\_POWERUP timer method. We can add this code here to loop through our vector of shared pointers setting the velocity of them to zero.

A screenshot of a computer

Description automatically generated with medium confidence

Now when the powerup starts, the asteroids will freeze in position, now we must return the velocity after the powerup ends so that the asteroids ‘unfreeze’. To do this we will loop through both of our vectors of shared pointers, setting the velocity to what we stored earlier.

Graphical user interface, text, application

Description automatically generated

Now the asteroids freeze in position we must freeze their rotation. The reason the asteroids continue to rotate is their animation, in order to stop this and completely freeze them for the powerup’s duration, we must find a way to freeze the animation. To do this we can adapt the Sprite.h and Sprite.cpp classes. We can add a Boolean variable called ‘pauseCheck’ to Sprite.h which will record whether or not the powerup is active. We can then wrap the animation code in Sprite.cpp in an if statement so that the code will only run if the variable is false.

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

In order to have this work for all asteroids, we can use the same technique we have used before. When creating the asteroids and small asteroids we can store shared pointers to their locations in a vector that we define in Asteroids.h.

A picture containing text, indoor, screenshot

Description automatically generated

We can then loop through the vector of sprites when starting and ending the powerup to steal and restore their velocity. All we need to do is set pauseCheck to true for each sprite pointer when starting the powerup and to false when ending it.

Graphical user interface, text, application

Description automatically generated

Now our Hourglass powerup successfully freezes the asteroids for 5 seconds, during which a timer counts down, until the 5 seconds have passed. At which point the Asteroids return to their original velocities and animation.

The only thing left to fix is the number off Hourglasses that are present in the game. As it stands, the game creates an hourglass in each level, meaning that if we do not use it, more than one can be present at once. This will be unfair for gameplay. So, we must only spawn an hourglass if one is not already present in the game. To do this we will create a Boolean variable (with getters and setters) in Asteroids.h which will keep track of if an hourglass is in the game. We will simply set it as true when we create an hourglass, and false when one is destroyed.

Graphical user interface, text

Description automatically generatedText

Description automatically generated

Graphical user interface, text, application

Description automatically generatedNow when we create a new level we must simply check if the hourglass is present and only spawn another if one does not already exist.

Text

Description automatically generated

There is one part left to fix however, if the asteroids are all destroyed whilst the hourglass is active, a new level starts. At the moment this produces two problems. Firstly, the timer is still visible despite the ability no longer being active, and once the powerup ends, all the asteroids freeze. This is because these new asteroids had their velocity saved as zero, and when the powerup ends this velocity of zero is given to the moving asteroid, freezing them.

To fix the issue of the timer still being on screen, we can go to START\_NEW\_LEVEL and set the visibility of mTimerLabel to false. This assures that it will be invisible when a new level starts.

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generatedIn order to stop our freezing asteroids problem, we can create a Boolean variable called level change (as well as a setter for it) in Asteroids.h. This can initially be set to false. We can then wrap the returning of velocities in END\_HOURGLASS\_POWERUP in an if statement so they can only run if level change is false. All we need to do now is set levelChange to true when we start another level and the problem is fixed.

Graphical user interface, text, application

Description automatically generated

A picture containing text

Description automatically generatedNow our powerup is complete.

Evaluation

One issue I had trouble with, when developing the powerup, was returning the velocities of the asteroids and small asteroids. In order to return the values of the velocity we have to store them as well as pointers to the asteroids and small asteroids, this way we can loop through the pointers in order and return the velocities. This is what I did. However, initially I stored the velocity when I ran the START\_HOURGLASS\_POWERUP timer and returned them in END\_HOURGLASS\_POWERUP. However, this caused a bug where the program crashed with a null pointer exception sometimes. After testing I found that the reason for the crash, was that asteroids had been shot by the player, causing more small asteroids to appear. These asteroids did not have a stored velocity for them. To fix this I changed the program so that I was storing the velocity when creating the asteroid. As their velocity is unchanging, this means that I can store the correct number of velocities and stop this bug from occurring.

The biggest problem I had was in the freezing of the asteroids. Whilst it was easy to freeze their velocity (GameObject had built in setters for this), I found it very difficult to freeze their rotation. Of course, if they stopped moving around but still appeared to change and rotate, this would not be a very good powerup. First, I tried freezing the rotation or adapting the angle (since there were built in setters for these), however this is not what caused the rotation. Then I tried changing the animation to try and stop the animation or pause it for some time. After testing this in lots of different classes (both in asteroids and game engine), I eventually found the animation was being updated in the sprite class. I tried to pause this for five seconds but again I could not find a way to get this to work. After realising this was running repeatedly, I added a Boolean variable so that the sprite can only run if this variable pauseCheck is true. This meant I could freeze the asteroids completely.

**Part III**

Plan

I have chosen option B, to implement an Alien spaceship. This will take the form of a Boss level, which will spawn in the place of a normal level (being where asteroids are created instead of aliens) once every three levels. This boss level will have a large alien spaceship at the top of the screen which requires multiple hits to be destroyed, as well as a counter showing the health of the alien as it is shot by the player. There will be a label showing that the boss level is beginning as well as a smaller more player-sized alien ship. This will spawn from the larger mothership and will follow the player, shooting it, whilst avoiding the player’s bullets. Whilst destroying this ship will give no points, 100 points will be given for killing the larger ship which will then end the level. Both ships will explode on their destruction.

Process

Firstly, we must create our alien spaceship. As we have been given the animation for it, we can create a reference to it at the beginning of our asteroids.cpp class. This will allow the system to locate the image required.

We must also create our alien classes, named ‘Alien.h’ and ‘Alien.cpp’ with the base code incorporating the same functions as our spaceship including Shoot, Thrust, CollisionTest and OnCollision. This is because it will move around, and shoot based on what is happening in the game. We can then write a class called ‘CreateAlien()’ which will attached a bounding sphere to our ship (needed for collision tests) as well as using the animation we connected earlier. We must also ensure the code is defined in Asteroids.h.

Graphical user interface, text

Description automatically generated

A picture containing calendar

Description automatically generated

We can now create our Alien ship. Calling the method ‘CreateAlien()’ when starting the game will spawn our alien ship, with its animation, in the game.



Now we can create our level system. As the aliens will only be present in the ‘boss level’ we will have to overwrite the current system. At the moment once all the asteroids in the level are destroyed, the ‘START\_NEW\_LEVEL’ timer is called. This will update the level counter and spawn new asteroids for our ship to destroy. Instead, we are going to create another timer (defining it in Asteroids.h) called ‘ACTIVATE\_BOSS\_LEVEL’. In order to ensure that the boss level is called every three levels, we can include the following code in the ‘OnObjectRemoved()’ methods where the ‘START\_NEW\_LEVEL’ is called. We can check if the level number is a multiple of three and if so, create a boss level instead.

Text

Description automatically generated

We can now go to our timers and inside our ‘ACTIVATE\_BOSS\_LEVEL’ timer and increment our level counter. We can also create a second timer called ‘START\_BOSS\_LEVEL’. This creates two states. The activate state, called when the boss level begins where we will set up the boss level, and the start state where the game is played. This will solve lots of problems as we will see later on. We can include a line of code in ACTIVATE\_BOSS\_LEVEL which will call START\_BOSS\_LEVEL 2 seconds later.

Graphical user interface, text, application

Description automatically generated

The next job is to create a label that tells the player a boss level has begun. We can do this in the GUI section of our Asteroids.cpp file. We can create a new shared pointer to a GUI label, which we will define in our Asteroids.h file. We can set the alignment to the middle and centre of the screen as well as setting the text to read “Boss Level!”. We must also ensure the label’s visibility is initially set to false so it we can choose when to display it.

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

However, we only want the label to introduce the level, not stay there indefinitely. To ensure this we can utilise our set-up and play states created earlier. We can set the visibility of the label to true in ACTIVATE\_BOSS\_LEVEL and to false in START\_BOSS\_LEVEL. This means the label will only appear for two seconds, and before any of the action begins, so not to obscure what is going on.

Graphical user interface, text, application

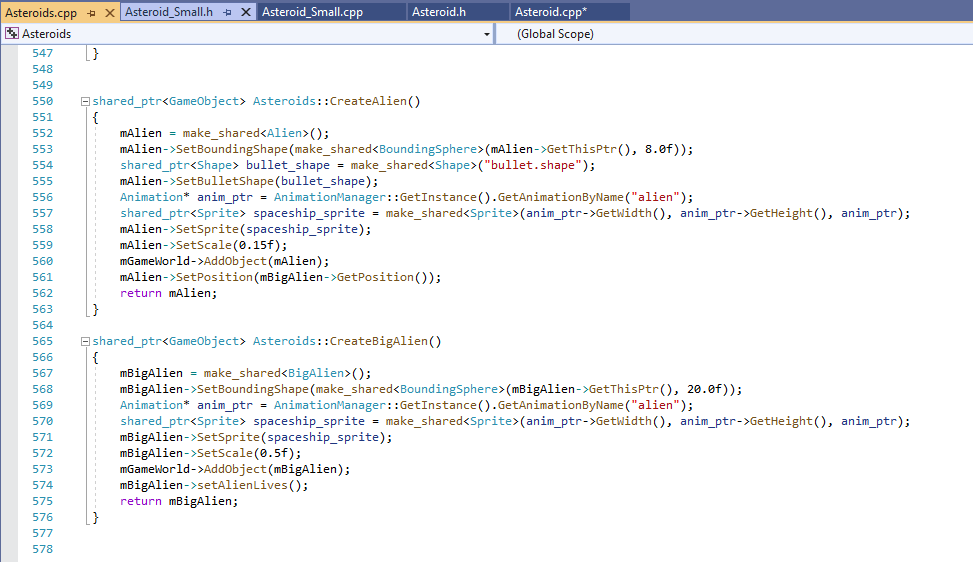
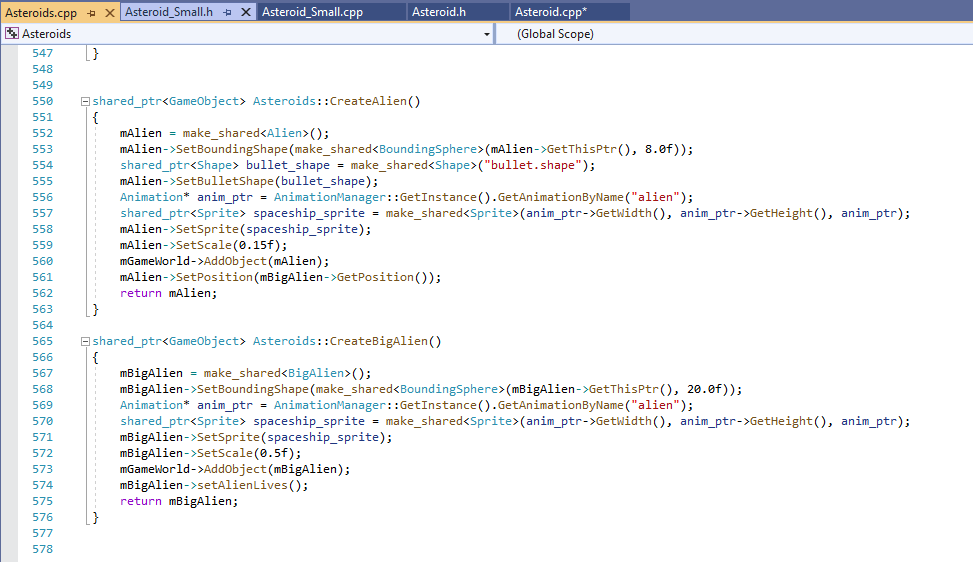
Description automatically generated

We now must create our boss, the big alien. We will do this by creating the classes ‘BigAlien.h’ and ‘BigAlien.cpp’. The code for this will be along the same structure as the asteroids files created earlier including the GameObject methods CollisionTest and OnCollision. One important change however, is to the positioning of the mothership. As we want the ship a the top of the screen, we have to code in the specific coordinates in which it will spawn, i.e. (0, 60, 0).

Graphical user interface

Description automatically generated with low confidence

We can then create the method, ‘CreateBigAlien’, creating a shared pointer to the alien as well as using the same animation as for our smaller alien. We will also create a bounding shape around the big alien. After testing I set the size to 20.0f which covers the middle section of the ship. I chose this to make the ship a smaller and harder to hit target despite its size.



A picture containing text, light, night sky

Description automatically generatedWe can now create our big alien, we do this by calling the ‘CreateBigAlien()’ function in START\_BOSS\_LEVEL. This will create our mothership when the boss levels start.

Graphical user interface, text, application

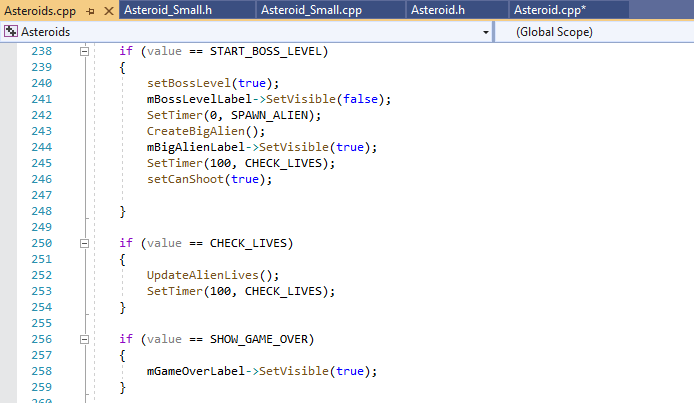
Description automatically generated

Now, since we want our mothership to survive more than one hit, it is going to have to have multiple lives. In order to show this to the player, we can create a new GUI label called mBigAlienLabel (remembering to define it in Asteroids.h). For now, let’s create a new GUI label at the bottom-right of the screen that initially states, “Alien Health: 100” and who’s visibility is set to false.

Graphical user interface, text

Description automatically generated

We can then set this visibility to true in START\_BOSS\_LEVEL and to false in START\_NEW\_LEVEL, meaning the health label is only visible in our boss level.



A picture containing text, light

Description automatically generatedNow when we open our boss level, the label should appear.

We have to set up the health of the Big Alien in order to change the health, but first we have to set up the collisions for each of our entities. For our boss level this will be our big alien, alien and bullet. For example, the spaceship should register a collision with the big alien, but the alien shouldn’t so that it can ‘spawn’ from within the mothership. We can add the collision exceptions shown below.

For Bullet:

Graphical user interface, text, application

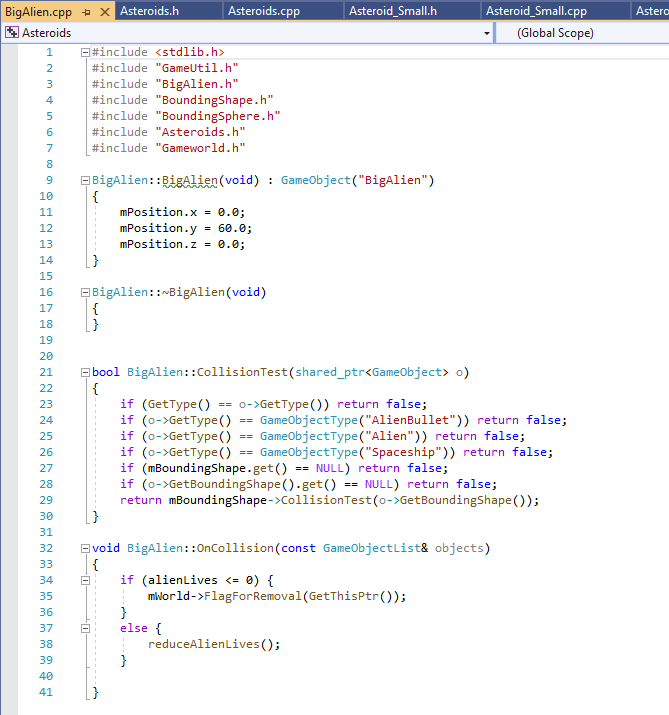
Description automatically generated

For Alien:

Text

Description automatically generated

For Big Alien:



Now that our collision systems are set up, we can implement the systems for our alien health. In BigAlien.h we can create a variable of type int called alienLives which will store the current life of the big alien. We can then create three functions to get, decrease by 5 and initially set the value to 100. We can use these methods to count the lives down from 100 to 0 over the course of 20 hits.

Graphical user interface, application

Description automatically generated

We can use the function setAlienLives() when the big alien ship is created, setting its life counter to 100. This means calling the function in the method CreateBigAlien().

Graphical user interface, text, application

Description automatically generated

Graphical user interface

Description automatically generated with low confidenceNow we can go to the OnCollision method of our Big Alien class and add the following code. This will check if the big alien has any more lives when it is hit. If so, the lives are decreased. If the ship has no more lives, the ship is destroyed.

Now that our alien ship has lives, we can need to be able to update the GUI label, so the user is aware of how they are doing. To do this we can create a method called ‘UpdateAlienLives()’ in Asteroids.cpp. This takes the shared pointer to our Big Alien and then calls the getAlienLives() function we created earlier. It then updates the GUI label with the current value found.

Graphical user interface, text, application

Description automatically generated

We can then create a timer in our OnTimer method called CHECK\_LIVES. Both of these functions must be stated in Asteroids.h. In this timer we can run our UpdateAlienLives() method as well as create a new timer of CHECK\_LIVES. This recursive timer will call once every 0.1 seconds to continuously update the lives GUI. We can initially create this in START\_BOSS\_LEVEL.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

Next, we have to design what happens when the Big Alien is destroyed. To do this we can go to our OnObjectRemoved method and add some new code for if a big alien is destroyed. This will signify the end of the boss level.

Text

Description automatically generatedThere are three things we must code here. First, we have to create an explosion in the exact position of the big alien when it was destroyed.

Then we can create a timer for START\_NEW\_LEVEL which will create a normal asteroid level.

Finally, we will set the label for the alien lives to invisible so that it disappears during a non-boss level.

To complete the coding of our Big Alien, we need to give the player some points. In the IScoreKeeper class, there is a function for when objects are removed. We can simply some code to give the player 200 points for every big alien destroyed. This will automatically update on the player’s display.

Graphical user interface

Description automatically generated with low confidence

So now we have a boss level that appears once every three levels which opens with a label saying that a boss level has begun. During this level, a Big Alien spawns, with a label showing its current health, that is destroyed after 20 hits. After which, an explosion occurs, and a normal level begins again. We also have a method to create a player sized alien in the Gameworld. Now we can give the alien some abilities.

The alien needs to spawn from within the mothership, move towards the player, shoot the player, dodge the player’s bullets and then (on its destruction) another must spawn from the mothership.

Text

Description automatically generatedFirst, let’s start with the spawning of the alien ship. In the OnTimer method we can create a new timer called SPAWN\_ALIEN where we can call the function ‘CreateAlien()’ that we wrote earlier. We can then create a call to this timer in the START\_BOSS\_LEVEL timer. This will create an alien ship at the start of each boss level. We also want to call this timer when an alien is destroyed so the player always has an enemy, this keeps the level interesting. However, since we only want to spawn aliens in a boss level, we need to add an if statement to check the current level. Since we know that the boss level only occurs once every three levels, we can use the remainder function to find out what level we are on and whether or not an alien should spawn. This will prevent any aliens appearing in normal levels.

Text

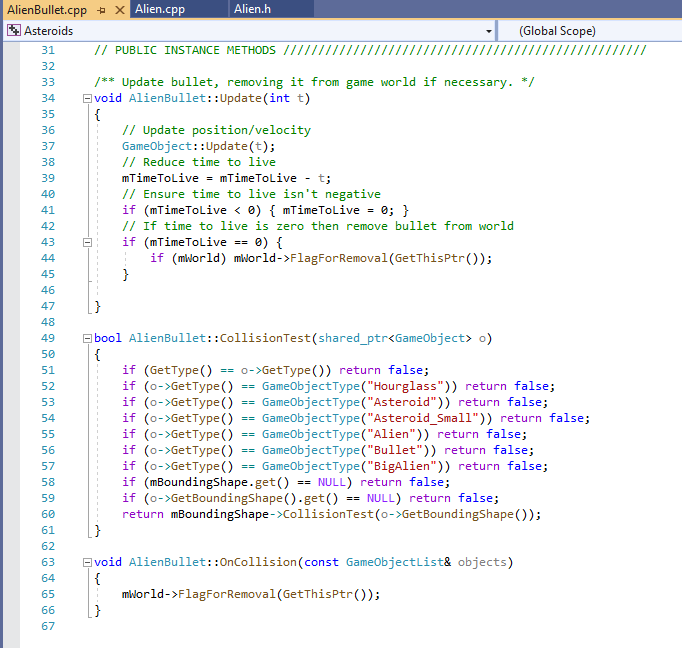
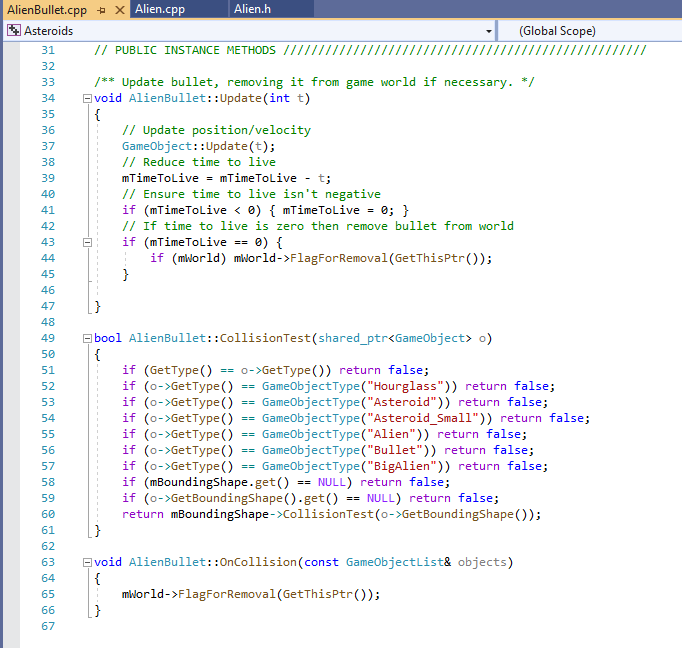
Description automatically generatedNow let’s, code the moving of our alien. Since there is a shared pointer to our alien stored in Asteroids.h, we can call the alien’s methods from the main game class. This means that we can create a method called Thrust in alien.cpp (defining it in alien.h). In order to figure out how to move our alien, we are going to need to know the location of the alien and the player (since this is the alien’s target). We can get the player’s coordinates from our function call so here we can just create a local variable called ‘player\_position’. In order to work out the direction needed; we can subtract the current alien position ‘mPosition’ from the player position creating a new variable called direction of type GLVector3f. However, since the vectors can be of wild sizes, this can cause the ship to move very quickly. As such we have to normalise the vector, meaning we can remove the magnitude from our calculation. We can use some Pythagoras theorem to work out the hypotenuse of the vector. Given that a vector is magnitude multiplied by direction, we can divide our x and y portions of the direction variable by the hypotenuse, so we are only left with the direction. We can then add this ‘normalised’ vector to the current velocity to have the velocity increase in the direction of the player. After testing I found that multiplying the value by the constant 0.7 gave the best alien movement experience. I.e., so, the ship wasn’t too fast or too slow.

We need to call this method repeatedly as the player is always moving. To do this we will create a timer called ‘ALIEN\_MOVING’ in our OnTimer function, which we will declare in Asteroids.h and will include a call to this method, referencing the pointer to our player ship. This also must be a recursive timer. As such we will call the timer again 100 milliseconds later. This will keep the direction of the alien ship changing so it constantly follows the player. We can then call this timer 3 seconds after the SPAWN\_ALIEN timer is called. This gives the player some time to move before the alien ship chases it.

Text

Description automatically generated

Now our alien moves towards the player, it’s time to implement another crucial part, the shooting. If the alien were to shoot with the same bullet class that the player uses, this would cause lots of bugs. Such as the alien being destroyed or the big alien taking damage. To avoid this, we will create a new class called AlienBullet. This will inherit the same methods as a normal bullet. The only difference will be the collision exceptions. The Alien bullet must not cause collisions with the Alien, Big Alien or Bullet. This is because the player and alien ship can easily have bullets collide which will end in their destruction which will not make for a fun game. We also must go to the Alien and BigAlien classes adding in this exception for them too.



Back in our Alien.cpp file we can adapt the function shoot. We can take the player position and take away the alien position to calculate the direction of the player. We can then multiply this by the pre-set bullet speed (after testing I found the best value for this to be 0.5). We can then create a new AlienBullet with the velocity we just created. This will create a bullet that shoots at the player.

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generatedIn order to have the alien repeatedly shoot the player, we will need another timer. This will be called ‘ALIEN\_SHOOTING’ and like others in the program will be recursive. We can set the timer to reoccur every three seconds. I tested this multiple times and found three to be the best number. In this method we can call our shoot method whilst referencing the player to create the AlienBullet.

All we need to do now is call the initial ALIEN\_SHOOTING in SPAWN\_ALIEN.

A picture containing text

Description automatically generatedA picture containing text

Description automatically generatedText

Description automatically generated

Now we have an alien which spawns from the big alien, that moves towards the player and regularly shoots at the player.

Graphical user interface, text, application, email

Description automatically generatedThe only major part left is to implement the bullet dodging. We can edit the alien ship’s velocity when the player shoots to have it quickly ‘dodge’ out of the way. This is good as it will avoid the first bullet or two, but the player can then go on to hit it. This therefore adds to gameplay without making the game too difficult. To do this we can create a method called dodge in Alien.cpp. This method will take the player position and alien position and use a series of if statements to work out the alien’s position in relation to the player, defining the position in terms of quadrants. With the first being west to north of the player, the second being north to east etc. Then, based on the quadrant, the velocity is set to move the alien diagonally out of the way. With testing I found the best values for this to be using the values 15 and 5 both positively and negatively. This was good to have the ship dodge out of the way fast enough to actually dodge the first bullet, but meaning it is not going so fast that the player cannot hit the ship.

All that is left now is to call the method. We can do this in the main asteroids class where the player shoots. In the OnKeyPressed method we can call our function. However, in order to do this without the code breaking, we can only call the method if an alien spaceship exists. So, we must first create a Boolean variable called bossLevel in Asteroids.h. We can also create a setBossLevel function. Now we can set this to true in START\_BOSS\_LEVEL and to false in START\_NEW\_LEVEL. Then all we need to do is check if the boss level is true and if it is, when the player shoots, the alien ship will dodge.

Graphical user interface, text

Description automatically generatedGraphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generatedNow the main functionality works, all we need to do is clean up some small details. Firstly, a big bug with the current code is that if the player shoots before the alien ship spawns, there is a null pointer exception. This is because the code is telling the alien to dodge when there is no alien. In order to fix this, we can create a Boolean variable called canShoot along with a setter to change it.

In our START\_BOSS\_LEVEL timer, we can initially set the canShoot variable to false. This suspends shooting whilst the set up phase occurs. Then in ACTIVATE\_BOSS\_LEVEL when the SPAWN\_ALIEN function has been called and an alien exists, we can set the canShoot variable back to true.

Graphical user interface, text, application

Description automatically generated

Finally, we can wrap the shooting function in an if statement so we can only shoot if canShoot is true. This fixes our bug.

Graphical user interface, text

Description automatically generated

Another details to fix is that whilst an hourglass doesn’t spawn on a boss level (they only spawn using START\_NEW\_LEVEL), if one isn’t used in the previous level, it will be carried into the boss level where it has no function. So, we need to remove it if this is the case. We already have functions which tell us if an hourglass is present, so we can use this to tell if at the start of a boss level an hourglass exists. In ACTIVATE\_BOSS\_LEVEL we can check with an if statement if there is an hourglass present in the level. If there is an hourglass, we can use the shared pointer for the hourglass to remove it from the level.

Graphical user interface, text, application

Description automatically generated

Evaluation

One problem I had here was with moving the alien spaceship. My initial attempt had me setting the velocity of the alien to the difference in the position between the alien and player position. This did not work as the alien shot at different speeds and directions. I needed to have the speed kept more in control. I then divided the direction by set values but again this didn’t work, and you just had the same problem just at lower speeds. I then wrote out the vectors as triangles and used Pythagoras to find a variable value with which to divide by. This solved most of the problem but after lots of testing I found that multiplying by the constant 0.7 gave me the best result for the alien movement. This gave the most controlled, realistic experience. I also repeatedly tested how often to call the method of movement to give the most realistic (following mechanic).

The biggest problem I experienced in this section was definitely in the dodging of the player’s bullets. My initial thought was to place a second bounding shape over the alien ship, so that if it collided with a bullet the alien would jump out of the way, but I wasn’t able to add a second bounding shape. I then considered that rather than using a second bounding shape, I could use a whole second object. Creating an invisible object that only collided with bullets, that was over the alien ship and followed its movements exactly. My plan was that if the bullet hit the cover object, we could move the cover object and the alien itself out of the way. I went so far as to create this object (with its own classes) setting its position and coding all the movement to match the alien ship. The problem here came in calling the methods for when the object is hit. When it is, we need to call methods in the asteroids class which I couldn’t figure out how to access without breaking the game. In the end I realised that the movement had to be run from within Asteroids.cpp. Unlike many of the other alien controls, which are run by timers, the dodging of bullets cannot be run this way. Otherwise, they will not be dodging the bullets, just erratically moving. This led to my attaching the dodging function to the shooting mechanic. This means that when the player shoots, I can move the ship in some way.

Now that I had a way to call the method, I had to figure out how exactly to move them. Given that the bullets travel in a constant trajectory, we cannot just move the alien back. This would mean the alien was still hit and no dodging actually takes place. When planning out how to move the ship, I thought about moving it clockwise around the screen when dodging. This led to my idea of splitting the screen into quadrants and moving the alien in a different way depending on its positioning. However, I first had the quadrants in terms of the screen, this stopped the entire moving mechanic as ships got stuck on certain paths. I then changed the code to have the quadrants defined as around the player instead, this was much better. I then just had to test to find what vector values to use to have the ship dodge in the best way. My initial thought was to move the ship clockwise (using values such as 15, 0), however this just ends up spiralling the alien into the player. So, I changed the code to use values such as 15, 5. This instead shifts the alien to the side and then slightly further back to prevent this. Now I had an alien which would dodge the first alien ship, but then would be hit by subsequent bullets.

Another issue I had was in updated the GUI label with the current health of the alien. I initially tried to use scorelisteners to update the GUI automatically, but I had lots of problems with access. I created a new class and implemented all the functions to run it, but I had a lot of problems with the setting the code to change given that the information was actually stored in the alien class. Realising that the best way to do this was through Asteroids.cpp, I decided to use a timer to recursively call the update function repeatedly. This fixed the problem and allowed the timer to update as soon as hits occur.

I also had a big problem with the bug discussed earlier, whereby if the player shoots before the alien ship is created, the game crashes with a null pointer exception. I fixed this by locking the shooting so that the player can only shoot in the first place if the alien is definitely in the game.