



# IN2026: Games Technology – Game Demo Coursework

Shaun Sevume

## Contents

|                |    |
|----------------|----|
| Part I .....   | 1  |
| Part II .....  | 5  |
| Part III ..... | 15 |

## Part I

For part one, I chose to implement option b – asteroids that split into smaller asteroids when hit.

Asteroids.cpp contained the method CreateAsteroids() which took in an unsigned integer that would determine how many asteroids to create. Whenever an object was removed from the GameWorld, the OnObjectRemoved() listener would check to see if it was an asteroid. If it was, then an explosion would be generated wherever the object last was prior to its removal.

As the extra asteroids created when one was hit had to be smaller, calling CreateAsteroids() again would not be sufficient, as the size was always 0.2f, so it would simply create extra asteroids of the same size.

Instead, I created a separate method called CreateSmallerAsteroids() (Figure 1), which always created two asteroids that were half the size of the original. Since the asteroids created would have to spawn at the location of the asteroid that had just been destroyed, I changed the parameters of CreateSmallerAsteroids() from an unsigned int (no longer required as the loop ran a fixed amount of times) to a shared\_ptr<GameObject>. This was because the OnObjectRemoved() listener also took in a shared\_ptr<GameObject> as one and used its GetType() member function to check if it was an asteroid or not. If it turned out to be one, then the same object could be passed into my CreateSmallerAsteroids() method (Figure 2), where I could use the GetPosition() member function of the same object to determine where the asteroid was destroyed. The value obtained could then be used as an argument for the SetPosition() member functions of the smaller asteroids to be created. Each smaller asteroid would also add 1 to the asteroid counter to keep it balanced.

```
void Asteroids::CreateSmallerAsteroids(shared_ptr<GameObject> object)
{
    for (uint i = 0; i < 2; i++) //Only runs twice, since only two new asteroids are being created.
    {
        Animation* anim_ptr = AnimationManager::GetInstance().GetAnimationByName("asteroid1");
        shared_ptr<Sprite> asteroid_sprite
            = make_shared<Sprite>(anim_ptr->GetWidth(), anim_ptr->GetHeight(), anim_ptr);
        asteroid_sprite->SetLoopAnimation(true);
        shared_ptr<GameObject> asteroid = make_shared<Asteroid>();
        asteroid->SetBoundingShape(make_shared<BoundingSphere>(asteroid->GetThisPtr(), 10.0f));
        asteroid->SetSprite(asteroid_sprite);
        asteroid->SetScale(0.1f); //Smaller asteroids = smaller scale
        asteroid->SetPosition(object->GetPosition()); //Spawn these two smaller asteroids where the original was destroyed.
        mGameWorld->AddObject(asteroid);
        mAsteroidCount++; //Add one to the counter, since the original asteroid being destroyed took away one and we added two more just now.
    }
}
```

Figure 1

```
void Asteroids::OnObjectRemoved(GameWorld* world, shared_ptr<GameObject> object)
{
    if (object->GetType() == GameObjectType("Asteroid"))
    {
        shared_ptr<GameObject> explosion = CreateExplosion();
        explosion->SetPosition(object->GetPosition());
        explosion->SetRotation(object->GetRotation());
        mGameWorld->AddObject(explosion);
        CreateSmallerAsteroids(object);
        mAsteroidCount--;
    }
}
```

Figure 2

The issue was that the game now spawned two smaller asteroids for every asteroid destroyed (including new ones), effectively meaning that for every asteroid destroyed, two more were created which made the game impossible to beat as the number of asteroids would only grow. (Figure 3)

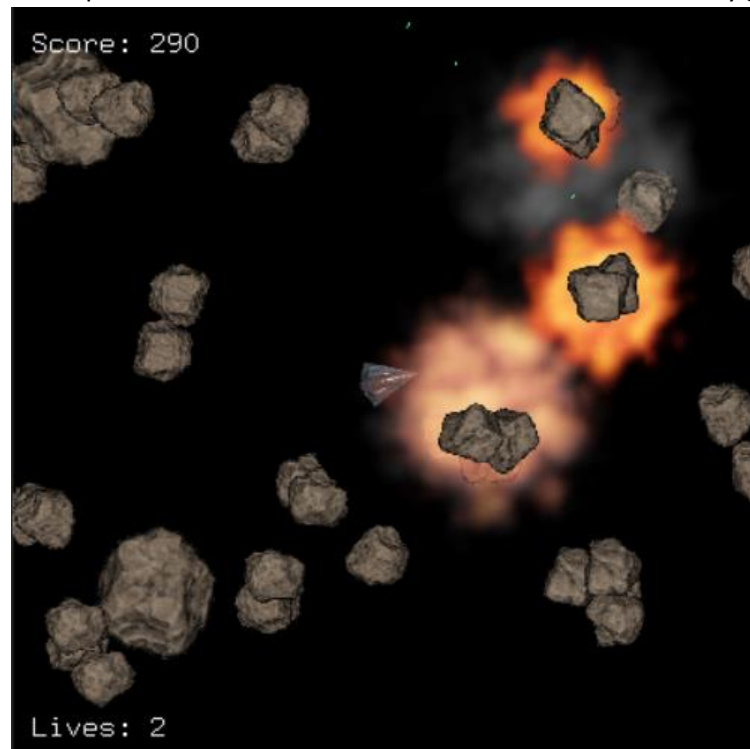


Figure 3

To solve this, the game would need to know when and when not to create a smaller asteroid. I decided to make asteroids only able to be split once. By adding a boolean called `split` to each asteroid created, it would be initialised to false by default, but when a smaller asteroid was created, it would be set to true. This way, the `OnObjectRemoved()` listener could use the value of this attribute to determine whether to create smaller asteroids or not. In the “Asteroid” header file, I defined a boolean called `split`, as well as getter and setter methods for them (Figure 5, Figure 4).

```
class Asteroid : public GameObject
{
    bool split;
```

Figure 5

```
bool isSplit() { return split; }
void setSplit(bool s) { split = s; }
```

Figure 4

Back in `OnObjectRemoved()` however, these member functions were inaccessible. This was because it belonged to asteroids only, but the listener was dealing with all `GameObject`s. This meant that I had to move the `split` member attribute and member functions to the `GameObject` header file, so that it could be used here. This also opened up the possibility of exploring the `split` functionality of other objects, such as bullets if required later down the line. Now, adding a block of code to verify if the asteroid had already split or not was made possible (Figure 6).

```
if (!object->isSplit()) { //If the asteroid hasn't split already...
    CreateSmallerAsteroids(object); //Call this method to create two smaller ones.
    //Smaller asteroids will have their 'split' attribute set to true, and cannot split any further.
    //The asteroid itself is also passed in as its position is needed to determine where to spawn the two new asteroids.
}
```

Figure 6

And now all smaller asteroids created would have this attribute set to true (Figure 7).

```
asteroid->SetPosition(object->GetPosition()); //Spawn these two smaller asteroids where the original was destroyed.  
asteroid->setSplit(true); //Makes sure the smaller asteroids cannot split again.  
mGameWorld->AddObject(asteroid);
```

Figure 7

Which resulted in asteroids only being able to split once, before disappearing for good. (**Error!**

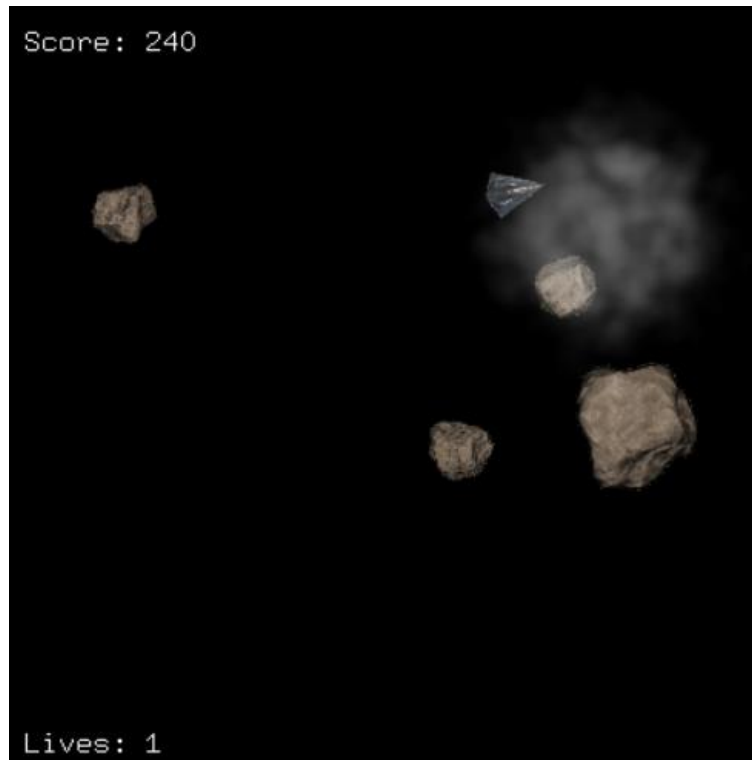


Figure 8

Reference source not found.).

## Part II

For part two, I chose to implement option a – the high score table.

The first thing to do was defining five new labels in 'Asteroids.h' (Figure 9) and create them in 'Asteroids.cpp' (Figure 10) with their visibility set to false. They would be displayed above each other on the game over screen to make a list of the top five scores. This was fairly simple to do, following the structure of the other labels being created. (Figure 10 only contains the creation of one label for brevity, as the code is the same for the other four and their inclusion would make the screenshot very large.) The location of the GameOver label was also adjusted to facilitate for the inclusion of the highscore labels. It would now be displayed above the first high score.

```
shared_ptr<UILabel> scoreOne;  
shared_ptr<UILabel> scoreTwo;  
shared_ptr<UILabel> scoreThree;  
shared_ptr<UILabel> scoreFour;  
shared_ptr<UILabel> scoreFive;
```

Figure 9

```
mGameDisplay->GetContainer()->AddComponent(game_over_component, GLVector2f(0.5f, 0.8f));  
  
// Create a new UILabel and wrap it up in a shared_ptr  
scoreOne = shared_ptr<UILabel>(new UILabel("1st: "));  
// Set the horizontal alignment of the label to GUI_HALIGN_CENTER  
scoreOne->SetHorizontalAlignment(GUIComponent::GUI_HALIGN_CENTER);  
// Set the vertical alignment of the label to GUI_VALIGN_MIDDLE  
scoreOne->SetVerticalAlignment(GUIComponent::GUI_VALIGN_MIDDLE);  
// Set the visibility of the label to false (hidden)  
scoreOne->SetVisible(false);  
  
// Add the UILabel to the GUIContainer  
shared_ptr<GUIComponent> score_one_component  
    = static_pointer_cast<GUIComponent>(scoreOne);  
mGameDisplay->GetContainer()->AddComponent(score_one_component, GLVector2f(0.5f, 0.7f));
```

Figure 10

These labels would be displayed at the same time the GameOver label, so they were set to become visible upon the game over screen alongside it (Figure 11).

```
if (value == SHOW_GAME_OVER)  
{  
    mGameOverLabel->SetVisible(true);  
    scoreOne->SetVisible(true);  
    scoreTwo->SetVisible(true);  
    scoreThree->SetVisible(true);  
    scoreFour->SetVisible(true);  
    scoreFive->SetVisible(true);  
}
```

Figure 11

This allowed the labels to be displayed in their correct locations upon reaching game over, but they were empty for now (Figure 12).



Figure 12

The next step would be to give them values to hold and display. These values would have to come from a txt file which the program would read, and then write back to if the scores needed updating. In order to eventually test the reading functionality, a txt file was created to hold 5 pre-existing high scores, to see if they could be read from. High scores were to be stored in the format of a 3-character name and a number afterwards denoting the score (Figure 13).

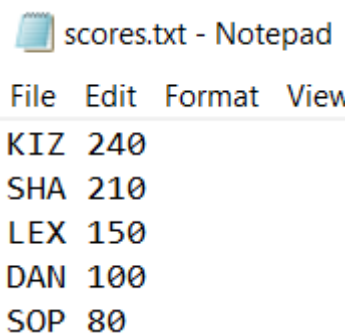


Figure 13

The idea was to have the program read in the file line by line, splitting each one into its components of a name and a score. Awaiting them would be a vector of strings and a vector of ints, which they would be pushed onto, respectively for further operations. Additionally, as the file would be read in as a string, the score would first have to be converted to int by using the `stoi()` method. All of this code was written in a new method, `CalculateHighScore()`, which would be called upon game over (Figure 14). To assist with the splitting of the line, an additional method called `split()` was created (Figure 15), which would take a string as input and push each word (separated by spaces) onto a vector which it would return. This way, the name would be at the first index of the vector returned, and the score at the second.

```
void Asteroids::CalculateHighScore()
{
    //Goes through the list of scores to see if the current score is a high score
    ifstream f; //The file to be opened
    string line; //The line to be read
    int i = 0; //amount of times to loop
    vector<string> names;
    vector<int> scores;
    f.open("scores.txt"); //Opens the file

    if (f.is_open()) {
        while (getline(f, line) && i < 5) { //Loop runs 5 times to get the high scores, reading the file line by line
            auto v = split(line); //Splits the line read into the name and the score
            names.push_back(v[0]); //Pushes the name onto the vector of names
            scores.push_back(stoi(v[1])); //Converts each line to an int and pushes it onto the vector of scores
            i++;
        }
    }
    f.close(); //Closes the file
}
```

Figure 14

```
vector<string> Asteroids::split(const string& s) {
    vector<string> v;
    auto space = [](char c) {return c == ' ';}; //A lambda function to define whether a character is a space or not.
    auto start = find_if_not(s.cbegin(), s.cend(), space); //Find the first non space.
    while (start != s.cend()) { //Check to see the end hasnt been reached
        auto end = find_if(start, s.cend(), space); //Find the next space.
        v.push_back(string(start, end)); //Put the characters between the spaces into a string and push it onto the vector.
        start = find_if_not(end, s.cend(), space); //Find the next space.
    }
    return v;
}
```

Figure 15

These two methods would also need to be defined in 'Asteroids.h' (Figure 17), as well as some additional `#include` statements in 'Asteroids.cpp' (Figure 16) to ensure these methods functioned properly.

```
vector<string> split(const string& s);
void CalculateHighScore();
```

Figure 17

```
#include<fstream>
#include<vector>
#include<algorithm>
```

Figure 16



Next, a for loop would iterate through the vector of scores and check if the player's score was greater than or equal to any of the elements inside (Figure 19). If it was, the score would be pushed onto the vector, which would then be sorted in descending order. The last value would then be popped, as it would be the lowest score and would not be displayed as one of the top 5 high scores. The index at which the player's score was found to be greater than or equal to one of the high scores was also stored in the variable 'pos', as the player's name would have to be inserted at this same index in the vector of strings containing the names. The ScoreKeeper object 'mScoreKeeper' was keeping track of the score and handling when to change the score upon an asteroid's destruction and letting other listeners know, but it did not have any member function to retrieve the actual score it was storing as 'mScore' inside itself. As access to this value was required for score valuation, I created a getScore() member function for the ScoreKeeper class that would return the already defined 'mScore' variable (Figure 18).

```
for (auto &i:scores) { //Cycle through the list of scores and see if the current score is greater than any of the others.
    if ((mScoreKeeper.getScore()) >= i) { //If the player's score is bigger than or equal to any score found in the list...
        int pos = i; //Gives the index which will be replaced in the list of names
        scores.push_back(mScoreKeeper.getScore()); //Add it to the list of scores
        sort(scores.begin(), scores.end(), greater<int>()); //Sort the list in ascending order
        scores.pop_back(); //Removes the lowest score as it is no longer a high score
        namePrompt->SetVisible(true); //Make the name prompt show up
        break;
    }
}
```

Figure 19

```
int getScore() { return mScore; }

private:
    int mScore;
```

Figure 18

If the player turned out to have a high score, the label 'namePrompt' would be made visible, which would prompt the user to input their name and then display the high scores with theirs included. If the user didn't have a high score, the scores could simply be displayed as they were. Since the program could branch off in two different directions here, I created an if statement for clarity after the for loop, which would either do nothing whilst the vectors of names and scores were being rearranged, or display the scores if no further action needed to be performed on them (Figure 20). As the code for placing the names and scores into the five highscore labels was quite lengthy, I placed them in a separate method called ShowHighScores() (Figure 21). The score labels would also become visible in this method, and not immediately upon the game over screen being shown. (Similar to the case of Figure 10, the formatting of only one highscore label has been shown for the sake of brevity.)

```
if (namePrompt->GetVisible()) {
    //This means the program is prompting the user to enter their name, which means they got a high score, so nothing should be done here.
}
else {
    ShowHighScores(); //If not, the user's score was not a high score, and the list can be shown without any changes being made.
}
```

Figure 20

```

void Asteroids::ShowHighScores() {
    // Format each high score label using a string-based stream
    std::ostringstream msg_stream_one;
    msg_stream_one << "1st: " << names.at(0) << " " << scores.at(0);
    std::string first = msg_stream_one.str();
    scoreOne->SetText(first);

    scoreOne->SetVisible(true);
}

```

Figure 21

This allowed the program to successfully read in the high scores if the player did not set one themselves, validating the 'else' part of the if-else statement in Figure 20. The high scores shown in Figure 22 can be compared to the contents of the text file in Figure 13.

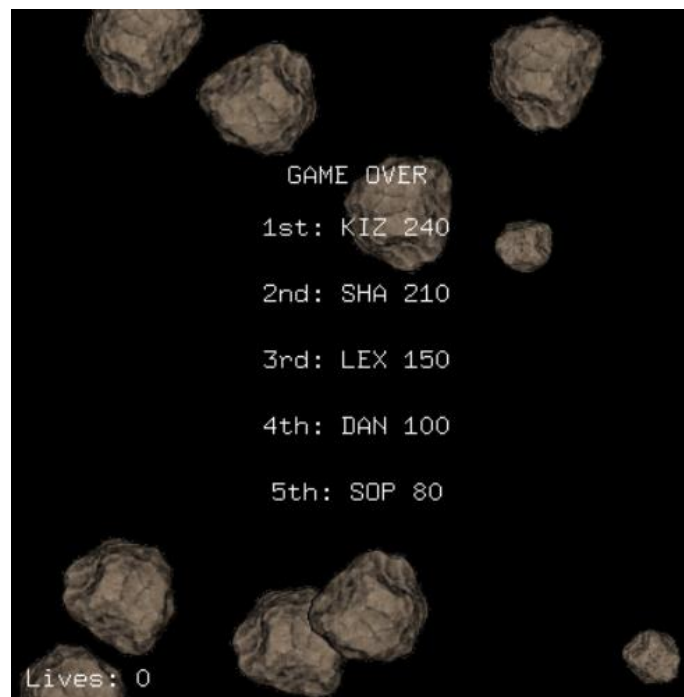


Figure 22

After creating the ShowHighScores() method, I had to make the vector of scores and names global, as well as the 'pos' variable (Figure 23). This was because even though I could pass them in from CalculateHighScores(), I would not be able to do the same for when the user had to input their own name to be added to the list of names. Traditionally, a while loop would be used to get the user's input which could be then added to the list of names at the index of whatever the value of 'pos' was, but this relied on the program collecting all three keypresses at once during the loop. In this program, input/output with cin/cout did not work, and the only other way I could think of obtaining user input was by utilising the OnKeyPressed() method that already existed, which caught the value of each key pressed to see if it was one of those bound to the player's controls. The issue with this method was that it was constantly running in the background, so I could not use it as a condition for a while loop in the same way I could for cin. Instead, a more creative approach was taken to solve the problem.

The user's input would only be required if they were being prompted to enter their name, which in turn meant that they had set a high score. The code already knew how to determine whether they'd set a highscore or not, and the visibility of the 'namePrompt' label could serve as a good indicator of when they were being prompted to enter their name. After this, any keys pressed could simply be added to a string 'playerName' which would hold their name. If the prompt label's visibility was being used to determine when to register user's input for their name, all that would be required would be a way to determine when to hide the label and stop adding any more keys pressed to the user's name. I came up with a variable called 'charCount' of type int, which would track how many keypresses had resulted in a character being added to the string containing the player's name. After each successful addition, the counter would increment, and once it reached 3, it would hide the namePrompt label, thus preventing the OnKeyPressed() listener from adding any more characters to the string.

Both variables 'playerName' and charCount' would have to be global in 'Asteroids.h' (Figure 23) as well, since there was nowhere to locally define them because the code which used them was in OnKeyPressed() which ran as a constant loop. Declaring them there would simply cause the method to redeclare them each time a key was pressed, losing both the player's input, and the number of characters that had been logged to the string containing their name. 'charCount' was also initialised to have a default value of 0 upon the game starting in the CreateGUI() method of 'Asteroids.cpp' (Figure 24).

```
vector<string> names;  
vector<int> scores;  
int pos;  
int charCount;  
string playerName;
```

Figure 23

```
charCount = 0;  
pos = 0;
```

Figure 24

After this, the final piece of the puzzle was to find a way of making sure the character added to the string playerName was actually a valid alphanumeric character. The OnKeyPressed() listened for all keys pressed, since it needed the arrow keys and space for player input, but these would not be representable in a string format. In other words, the character could only be added to the player's name if it were in the range of a-z, A-Z and 0-9. Fortunately, chars have the ability to be evaluated as a char or as its ASCII value. While it would be extremely long and tedious to evaluate the input character against every possible valid character, it would be much easier to compare its numerical ASCII value within a specific range. To handle this check, the CharCheck() method was created (Figure 26), accepting an unsigned char as a parameter and with a return type of bool. If the ASCII value of the character passed in fell within the respective ranges of the three valid character groups, the function would return true – otherwise, it returned false.

```

bool Asteroids::CharCheck(uchar input_char)
{
    // CHECKING FOR ALPHABET
    if ((input_char >= 65 && input_char <= 90) || (input_char >= 97 && input_char <= 122)) {
        return true;
    }
    else if (input_char >= 48 && input_char <= 57) { // CHECKING FOR DIGITS
        return true;
    }
    else { // OTHERWISE SPECIAL CHARACTER
        return false;
    }
}

```

Figure 26

With this method in place, the code for logging the players input while the namePrompt was visible could be implemented (Figure 25).

```

void Asteroids::OnKeyPressed(uchar key, int x, int y)
{
    switch (key)
    {
        case ' ':
            mSpaceship->Shoot();
            break;
        default:
            break;
    }

    if (namePrompt->GetVisible()) {
        if (CharCheck(key)) { //Check if characters are valid by evaluating the ASCII value
            playerName += key; //If it is, add it to the string that will hold the player's name
            charCount++; //Increment the count
            namePrompt->SetText("Enter Name: " + playerName); //Update the name prompt with the user's input
        }
        if (charCount == 3) {
            namePrompt->SetVisible(false); //Once 3 characters have been verified and added to the name string, hide the name prompt
        }
    }
}

```

Figure 25

The final step would be to insert the string at the element 'pos' in the vector 'names' and pop the last value from it as it would no longer to be a high score. As vectors have a built in insert() method, the task seemed fairly easy to do. After this was done, ShowHighScores() could be called, which would display the updated vectors of 'names' and 'scores'. The insert() method only took iterators as parameters, but combining an iterator pointing to the beginning of a vector with an integer and passing them as arguments in the next() method would return an iterator pointing at the index specified by the integer value. Since 'pos' was tracking where the new high score was to be inserted, it became a parameter for next() alongside a newly created iterator which would assume the type of whatever it was assigned to via use of the 'auto' keyword (Figure 27). The resulting iterator would then be able to be used as an argument for the insert() method, alongside the vector of names.

```

if (charCount == 3) {
    namePrompt->SetVisible(false); //Once 3 characters have been verified and added to the name string, hide the
    auto it = next(names.begin(), pos); //Create an iterator that points to the same position the player's high
    names.insert(it, playerName); //Insert the player's name at the same position the player's high score was
    names.pop_back(); //Remove the last name in the list. Their score is no longer top 5
    ShowHighScores(); //Now the high score list can be shown, where the updated lists will be reflected.
}

```

Figure 27

However, upon testing, an error popped up with the expression “cannot seek vector iterator after end”(Figure 28). This meant that the iterator used to figure out where to insert the player’s name if they had a highscore was pointing out of the range of the ‘names’ vector.

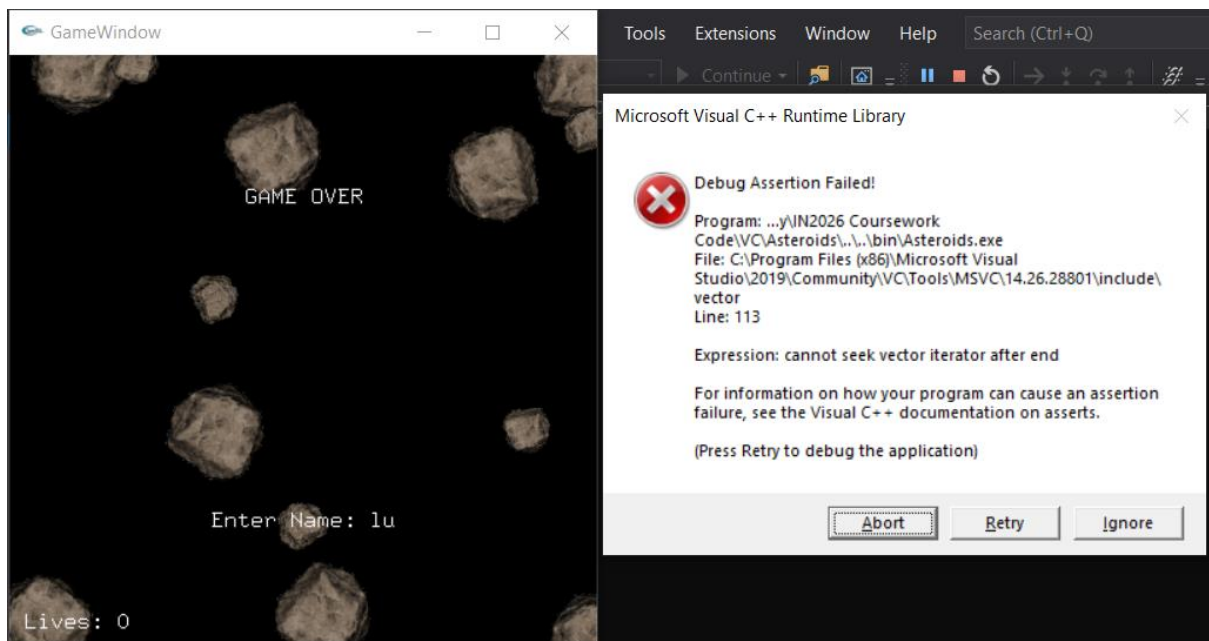


Figure 28

The only thing influencing that iterator was the variable ‘pos’, which had an integer value of what I believed to be the index at which the player’s score was found to be higher than an already existing highscore. As printing its value with cout was not possible, I opted to set the value of the ‘lives’ label to whatever the value of ‘pos’ was, in order to see what value it was holding (Figure 29). I also temporarily commented out the line that would make the name prompt visible, to stop the code that listened for a name from the player to insert into the vector ‘names’ from running, that would inevitably crash the program.

```
for (auto &i:scores) { //Cycle through the list of scores and see if the current score is greater than any of the others.
    if ((mScoreKeeper.getScore()) >= i) { //If the player's score is bigger than or equal to any score found in the list...
        pos = i; //Gives the index which will be replaced in the list of names

        std::ostringstream msg_stream_one;
        msg_stream_one << pos;
        std::string first = msg_stream_one.str();
        mLivesLabel->SetText(first);

        scores.push_back(mScoreKeeper.getScore()); //Add it to the list of scores
        sort(scores.begin(), scores.end(), greater<int>()); //Sort the list in ascending order
        scores.pop_back(); //Removes the lowest score as it is no longer a high score
        //namePrompt->SetVisible(true); //Make the name prompt show up
        break;
    }
}
```

Figure 29

Upon reaching the game over screen this time, the lives indicator changed to show the value of pos. Instead of a number between 0 and 4, it contained a value of 210 – coincidentally the same value of the 3<sup>rd</sup> place highscore (Figure 30). The score attained before dying was 230, which had been successfully passed into the vector of scores and was being displayed. (Since the functionality to enter a name was disabled, the scores were simply reassigned to the existing names. This behaviour was expected, and would return to normal once the functionality to enter a name was restored.)

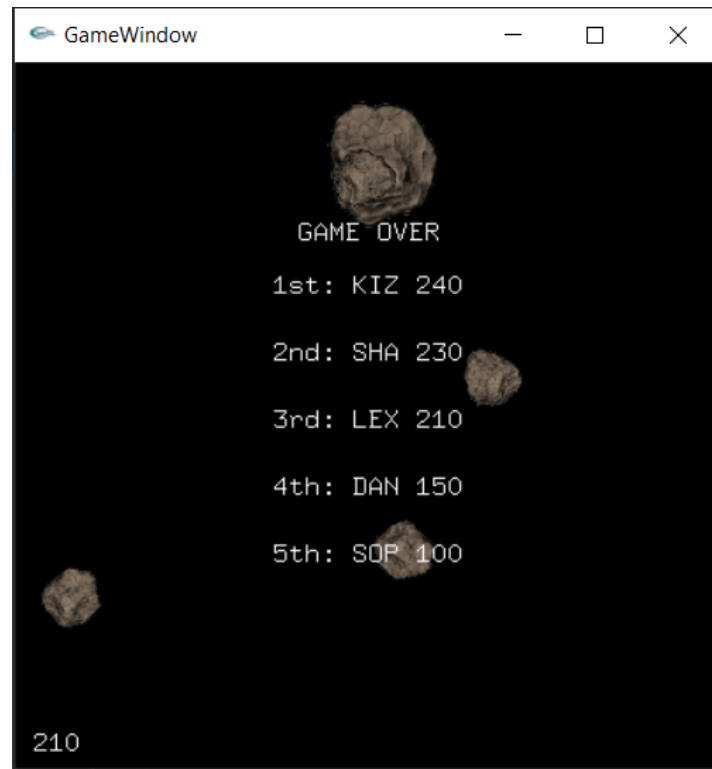


Figure 30

For this instance, the value of 'pos' was supposed to be 1, which would represent second place as 230 would be the second highest score in the vector of scores. From this, it was clear to see that 'pos' was taking the value of the score to be replaced, rather than the index of its value, which is what caused the error in Figure 28.

To fix this, the loop in CalculateHighScore() was refined to iterate through the vector of scores with an iterator. The distance() method would then be used to calculate the integer value of the index at which the player's score was found to be higher than any of the highscores – if any existed (Figure 31). From here, that value could be assigned to 'pos', which would allow it to stay within the range of the vector of names during insertion.

```
for (auto it = begin(scores); it != end(scores); ++it) { //Cycle through the list of scores and see
    if ((mScoreKeeper.getScore()) >= *it) { //If the player's score is bigger than or equal to any
        pos = distance(scores.begin(), it); //Gives the index which will be replaced in the list of
        scores.push_back(mScoreKeeper.getScore()); //Add it to the list of scores
        sort(scores.begin(), scores.end(), greater<int>()); //Sort the list in ascending order
        scores.pop_back(); //Removes the lowest score as it is no longer a high score
        namePrompt->SetVisible(true); //Make the name prompt show up which will cause the listener
        break;
    }
}
```

Figure 31

The solution proved to be successful, allowing SHA to reclaim 1<sup>st</sup> place on the list from KIZ! (Figure 32)



Figure 32

To make the changes stick, the values held in the names and scores vector were re-written to the text file after the scores were displayed in ShowHighScores() using a for loop and the ofstream class (Figure 34, Figure 33).

```
//Save the scores
ofstream f2;
f2.open("scores.txt");

for (int i = 0; i < 5; ++i) {
    f2 << names.at(i) << " " << scores.at(i) << '\n';
}
f2.close();
```

Figure 33

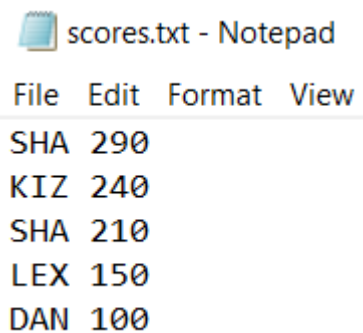


Figure 34

## Part III

For part three, I chose to implement option b – an alien spaceship.

Since both the alien and the player were spaceships, implementation was very similar – their cpp and header files sharing the same methods and attributes. In 'Asteroids.cpp', I made a separate method called CreateEnemy(), where the relevant values were set for a newly created object 'mEnemy' of type 'Enemy' rather than 'mSpaceship' of type 'Spaceship' in the CreateSpaceship() method (Figure 35).

```
shared_ptr<GameObject> Asteroids::CreateEnemy()
{
    // Create a raw pointer to a spaceship that can be converted to
    // shared_ptrs of different types because GameWorld implements IRefCount
    mEnemy = make_shared<Enemy>();
    mEnemy->SetBoundingShape(make_shared<BoundingSphere>(mEnemy->GetThisPtr(), 4.0f));
    shared_ptr<Shape> bullet_shape = make_shared<Shape>("bullet.shape");
    mEnemy->SetBulletShape(bullet_shape);
    Animation* anim_ptr = AnimationManager::GetInstance().GetAnimationByName("spaceship");
    shared_ptr<Sprite> enemy_sprite =
        make_shared<Sprite>(anim_ptr->GetWidth(), anim_ptr->GetHeight(), anim_ptr);
    mEnemy->SetSprite(enemy_sprite);
    mEnemy->SetScale(0.1f);
    // Return the spaceship so it can be added to the world
    return mEnemy;
}
```

Figure 35

With this, a second spaceship spawned in the centre of the screen, but it was idle for the moment (Figure 36).

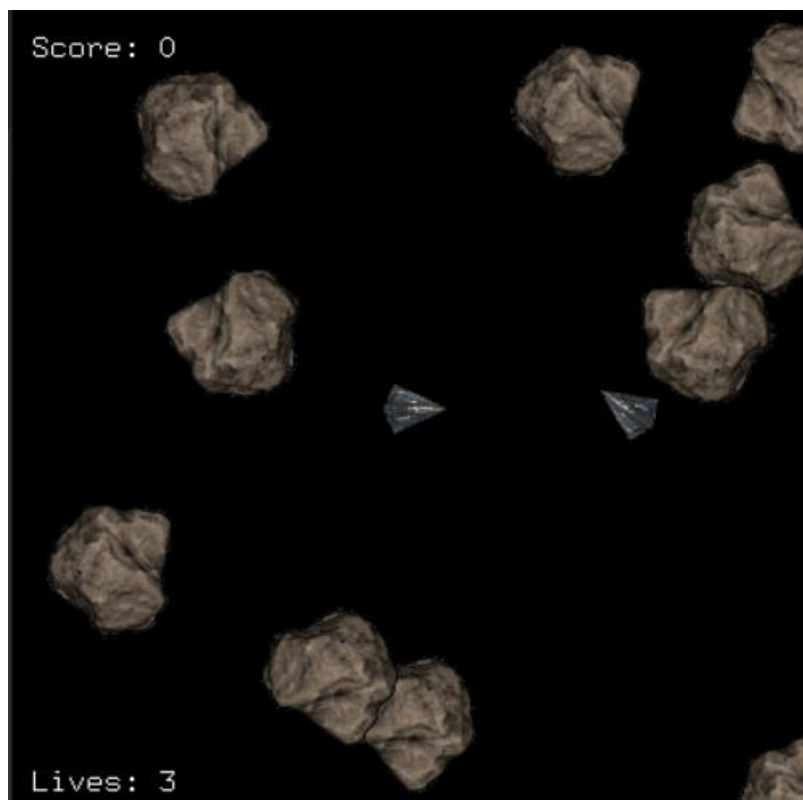


Figure 36



As it contained all the same methods as a spaceship, I created a quick test for them by assigning a call to its Shoot() in the OnKeyPressed() method if the key 'e' was pressed (Figure 37). This allowed the enemy to shoot, but there was no distinction between player and enemy bullets, so the enemy was actually helping the player by destroying asteroids for them which added to the score. Any asteroid that collided with the enemy was also destroyed and added a score to the player's total (Figure 38).

```
void Asteroids::OnKeyPressed(uchar key, int x, int y)
{
    switch (key)
    {
        case ' ':
            mSpaceship->Shoot();
            break;
        case 'e':
            mEnemy->Shoot();
            break;
        default:
            break;
    }
}
```

Figure 37

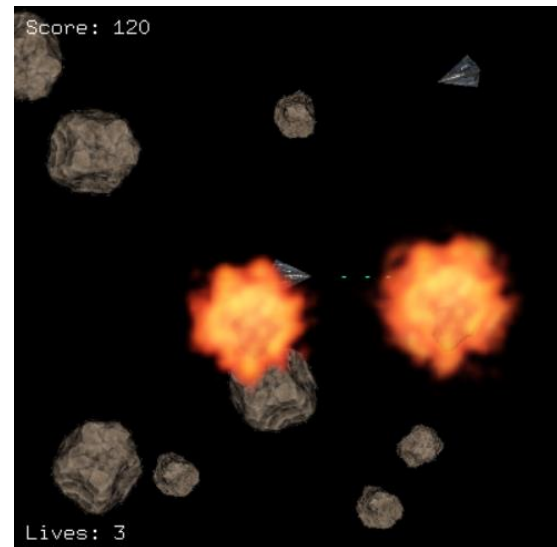


Figure 38

As a solution, a separate class for enemy bullets were created called 'EnemyBullet'. These bullets would harm the player instead of destroying asteroids, so the CollisionTest() methods had to be modified for both 'Spaceship.cpp' () and 'EnemyBullet.cpp' (Figure 39) as the objects they were safe to touch were redefined.

```
bool Spaceship::CollisionTest(shared_ptr<GameObject> o)
{
    //if (o->GetType() != GameObjectType("EnemyBullet")) return true;
    if (o->GetType() != GameObjectType("Asteroid") && o->GetType() != GameObjectType("EnemyBullet")) return false;
    if (mBoundingShape.get() == NULL) return false;
    if (o->GetBoundingShape().get() == NULL) return false;
    return mBoundingShape->CollisionTest(o->GetBoundingShape());
}
```

Figure 40 – The spaceship can now die to asteroids OR enemy bullets.

```
bool EnemyBullet::CollisionTest(shared_ptr<GameObject> o)
{
    if (o->GetType() != GameObjectType("Spaceship")) return false;
    if (mBoundingShape.get() == NULL) return false;
    if (o->GetBoundingShape().get() == NULL) return false;
    return mBoundingShape->CollisionTest(o->GetBoundingShape());
}
```

Figure 39 – The enemy bullet will disappear if it touches a spaceship.

Asteroids also had to be made immune to enemy bullets, and the enemy itself – to stop it from unintentionally giving the player points whilst trying to kill them (Figure 41).

```
bool Asteroid::CollisionTest(shared_ptr<GameObject> o)
{
    if (GetType() == o->GetType() || o->GetType() == GameObjectType("EnemyBullet") || o->GetType() == GameObjectType("Enemy")) return false;
    if (mBoundingShape.get() == NULL) return false;
    if (o->GetBoundingShape().get() == NULL) return false;
    return mBoundingShape->CollisionTest(o->GetBoundingShape());
}
```

Figure 41 – Asteroids are now immune to enemy bullets and the enemy itself. They will simply pass through.

The next step in the implementation of an alien spaceship would be to define a way for it to track and follow the player, instead of remaining static in the middle of the screen the whole time.

First, it would need to have a target, something to actually track and follow. Inside 'Enemy.h', I defined a shared pointer to a GameObject called 'mTarget', and a setter method for it so that it would become whatever object was passed in (Figure 43). Back in the CreateEnemy() method, I added an extra line where I passed in mSpaceship to be used as the target for mEnemy (Figure 42).

```
void SetTarget(shared_ptr<GameObject> target) { mTarget = target; }

//bool CollisionTest(shared_ptr<GameObject> o);
void OnCollision(const GameObjectList& objects);

private:
    float mThrust;
    shared_ptr<GameObject> mTarget;
```

```
mEnemy->SetTarget(mSpaceship);
```

Figure 42

Figure 43

Then, I modified the Rotate() function to calculate the angle the enemy needed rotate to in order to be facing the player. The formula for this was  $\text{atan2}(y_1 - y_2, x_1 - x_2)$  where  $(x_1, y_1)$  is the target's position and  $(x_2, y_2)$  is the object's position – in this case the enemy. As the value calculated by this formula was in radians, it would have to be divided by  $(180/\pi)$  to get the angle in degrees.

Afterwards, the rotation of the enemy could simply be set to this value, which would make it 'look at' the player's ship wherever they were in the world (Figure 44).

```
/** Set the rotation. */
void Enemy::Rotate()
{
    float beta = atan2(mTarget->GetPosition().y - mPosition.y, mTarget->GetPosition().x - mPosition.x);
    beta = beta * (180 / M_PI);

    mAngle = beta;
}
```

Figure 44

Now that the enemy would always be rotating to look at the player, the second step was to define when it could move towards them. It made the most sense to have the enemy move to follow the player until it was within a certain “range”, that would be comfortable enough to be shooting from. To determine this “range”, the distance between the player and the enemy would have to be calculated, which could be done by getting the length of the vector between the enemy’s position and its target’s position. An if statement would then have a condition to check whether the distance was greater than a specified value – I chose 40. If the condition was true, then the enemy would move forwards with a constant velocity, with its forward direction already defined as towards the player thanks to the code in the Rotate() function (Figure 45)

```
/** Fire the rockets. */
void Enemy::Thrust(float t)
{
    GLVector3f dist = (mTarget->GetPosition() - mPosition);

    if (dist.length() > 40) {
        // move.
        mVelocity.x = t * cos(DEG2RAD * mAngle);
        mVelocity.y = t * sin(DEG2RAD * mAngle);
    }
}
```

Figure 45

To test this code, I bound calls to these functions on the ‘r’ key, similar to how I had tested the Shoot() function with a keypress of ‘e’. For now, the goal was to ensure that the functionality worked before making it automated. The issue that arose from this code however, was that once the enemy reached the specified range, it would simply drift past the player to the other side, until it was at that range, at which point it would start having velocity again and drift back across to the other side of the player. Whilst within the specified range, it appeared to be just drifting in air in the last direction it was headed in (Figure 46).

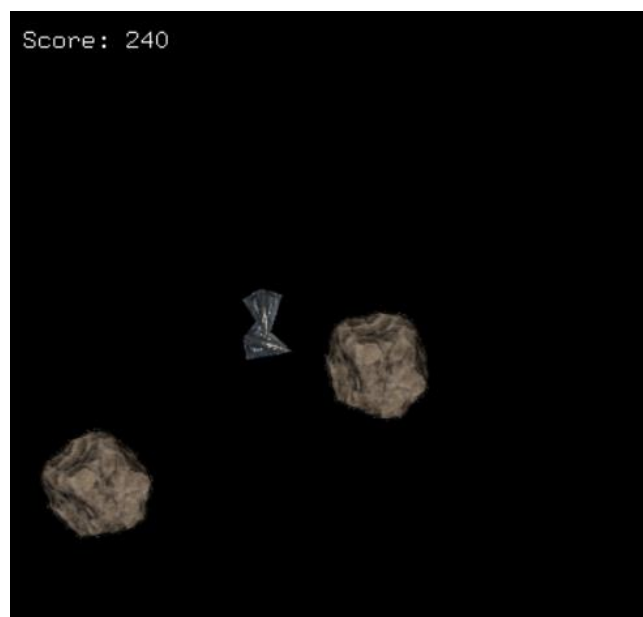


Figure 46 – The enemy spaceship ‘drifts’ past the player to the specified range on the other side of them.

An else statement to stop the enemy when within the range by setting its velocity in both components to 0 was enough of a fix to solve this issue (Figure 49, Figure 49). Additionally, this set up the perfect conditions for the enemy to open fire on the player (Figure 47).

```

/** Fire the rockets. */
void Enemy::Thrust(float t)
{
    GLVector3f dist = (mTarget->GetPosition() - mPosition);

    if (dist.length() > 40) {
        // move.
        mVelocity.x = t * cos(DEG2RAD * mAngle);
        mVelocity.y = t * sin(DEG2RAD * mAngle);
    }
    else {
        mVelocity.x = 0;
        mVelocity.y = 0;
    }
}

```

Figure 48

```

else {
    mVelocity.x = 0;
    mVelocity.y = 0;
    Shoot();
}

```

Figure 47

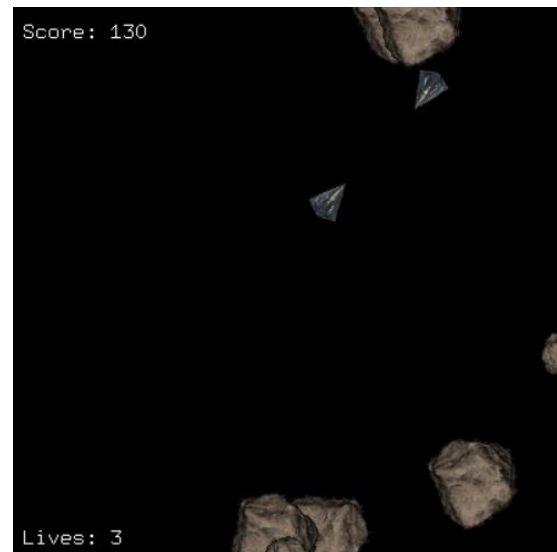


Figure 49 – The enemy spaceship (lower) now stops once it reaches the specified distance of 40 from the player.

Currently, moving towards the player and rotating to face them were defined in two separate methods. These were combined into a single method called `TrackPlayer()`, which took a float as an argument that would become the speed for the alien spaceship to move at (Figure 51). This method was then placed in the `Update()` method, where it would be called alongside whenever the object itself was updated. This marked the transition of its functionality from manual (through holding down a key) to automated (called a certain amount of times per second) (Figure 52). Finally, the enemy was set to spawn at a random location, rather than always spawning at the centre of the screen like the player.

```

/** Update this enemy. */
void Enemy::Update(int t)
{
    // Call parent update function
    GameObject::Update(t);
    TrackPlayer(10);
}

```

Figure 51

```

/** Update this enemy. */
void Enemy::Update(int t)
{
    // Call parent update function
    GameObject::Update(t);
    TrackPlayer(10);
}

```

Figure 52

```

/** Default constructor. */
Enemy::Enemy()
: GameObject("Enemy"), mThrust(0)
{
    mPosition.x = rand() / 2;
    mPosition.y = rand() / 2;
    mPosition.z = 0.0;
}

```

Figure 50

Now the enemy spaceship followed the player around the world, but it was hunting them down very relentlessly. Once it got into range, it immediately fired a stream of bullets so rapidly that it became difficult to distinguish each individual bullet anymore. This rapid firing was due to the frequency at which `Update()` in the `Enemy` class was being called, so this was the next thing that would need to be addressed.

The enemy's player hunting was so relentless in fact, that it went to the spawn point of the player upon killing them, waiting within the specified range and shooting before the player had even respawned. Effectively, it had been programmed with very aggressive 'spawnkilling' tendencies and was difficult to evade (Figure 53).

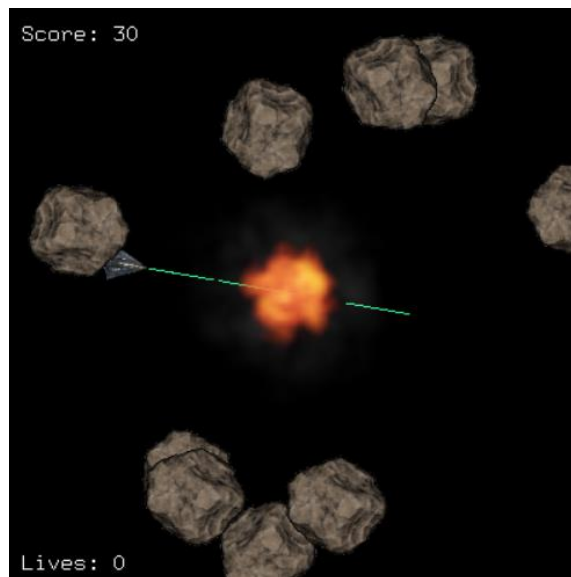


Figure 53 – The enemy spaceship 'spawnkills' by rapidly firing bullets at the spawn location of the player before they've even spawned.

An interesting solution I had to this was to give the enemy a 'teleporting' feature (). If they teleported to a random location upon killing the player, they would have time to move again before being 'spawnkilled' (Figure 55). Additionally, the random behaviour would keep the player wary of the enemy and constantly on the lookout for where it could appear next (Figure 54).

```
void Enemy::Teleport() {  
    mPosition.x = rand() / 2;  
    mPosition.y = rand() / 2;  
    mPosition.z = 0.0;  
}
```

Figure 55

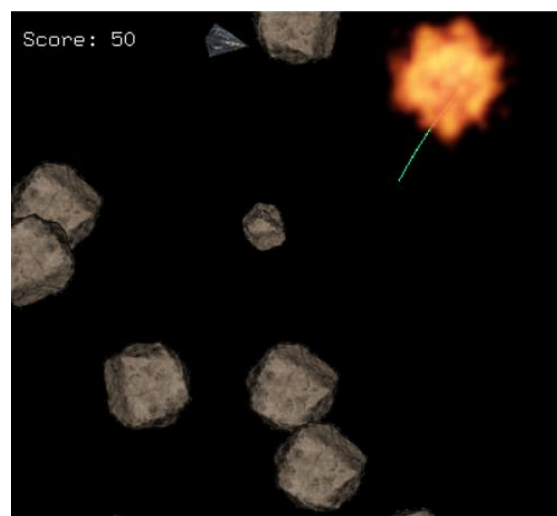


Figure 54 – The player died at the explosion and the enemy was where the stream of bullets were. After killing the player, it teleported to a random location as shown.

To combat the issue of the enemy firing too rapidly, a more ‘unpredictable’ shooting frequency was created. This consisted of a function to generate a random integer between 0 and 99 every time Update() was called. If the value of this integer was greater than 95, then the enemy could shoot. This code was placed in the TrackPlayer() function of ‘Enemy.cpp’ (Figure 56), and would give some variance and inconsistency to the shooting patterns of the enemy, whilst relieving the player of some pressure as they were not under constant fire all the time (Figure 57).

```
int r = rand() % 100;

if (dist.length() > 40) {
    // move.
    mVelocity.x = 20 * cos(DEG2RAD * mAngle);
    mVelocity.y = 20 * sin(DEG2RAD * mAngle);
} else {
    mVelocity.x = 0;
    mVelocity.y = 0;
    if (r > 95) {
        Shoot();
    }
}
```

Figure 56

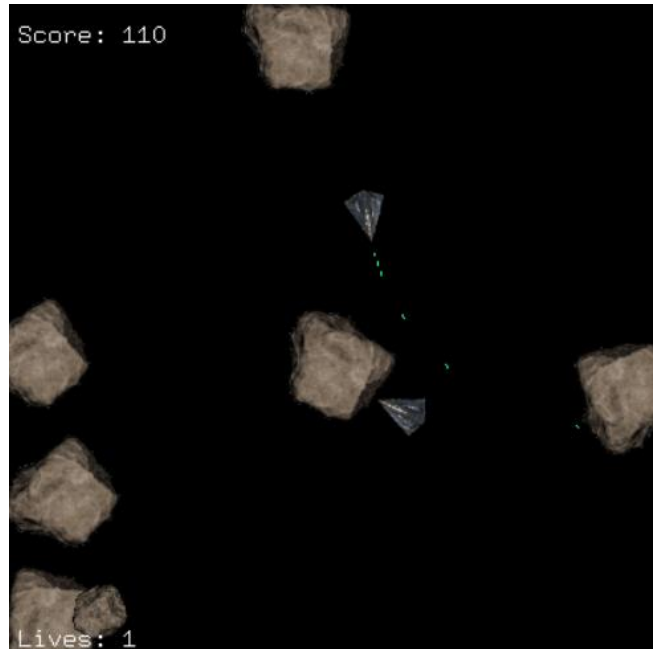
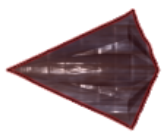


Figure 57 – The spray pattern for the enemy is a little more erratic, rather than a constant ‘stream’ of bullets.

With the enemy having the basic functionality to track and shoot at the player in a more ‘reasonable’ manner, steps were taken to ensure that the two were distinguishable from each other. I modified the enemy’s image attached to its animation to make it more distinguishable from the player’s spaceship (Figure 58), specifying it as “enemy\_fs.png” (Figure 59). The CreateEnemy() function was also slightly modified to allow the alien spaceship to use its newly created design (Figure 60).



```
Animation *enemy_anim = AnimationManager::GetInstance().CreateAnimationFromFile(
    "enemy", 128, 128, 128, 128, "enemy_fs.png");
```

Figure 59

Figure 58

```
Animation* anim_ptr = AnimationManager::GetInstance().GetAnimationByName("enemy");
shared_ptr<Sprite> enemy_sprite =
    make_shared<Sprite>(anim_ptr->GetWidth(), anim_ptr->GetHeight(), anim_ptr);
mEnemy->SetSprite(enemy_sprite);
```

Figure 60

This allowed for a clearer distinction between the enemy and player spaceships (Figure 61).

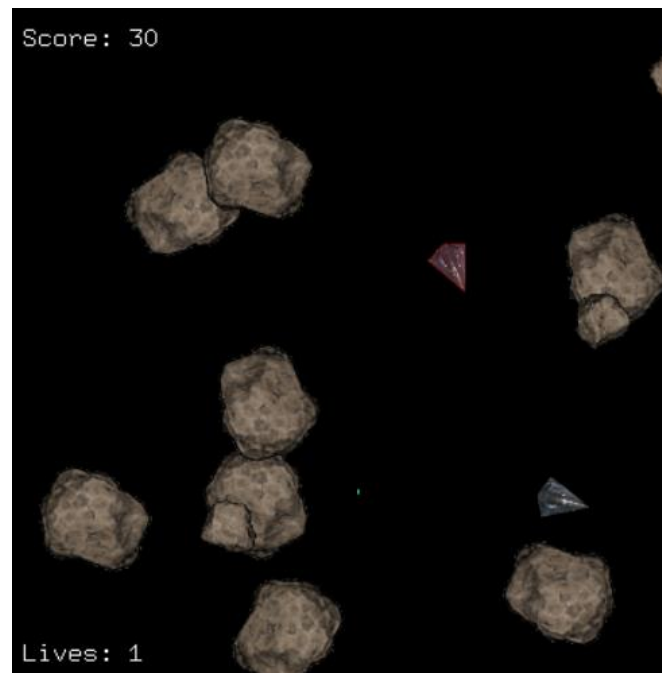


Figure 61

After successfully creating a new image to attach to the animation of the enemy spaceship, I took the opportunity to do the same with the “smaller asteroids”, to make them look different. This was done in the same manner, by specifying the “asteriod3\_fs.png” filmstrip and creating a new animation of it. After this, CreateSmallerAsteriods() was slightly modified to contain an animation pointer that pointed to “asteroid3” instead of “asteroid1”, and its scale was reverted back to 0.2, since the asteroid was already smaller in the filmstrip itself. The variance in asteroids gave the game a more polished look. (Figure 62).

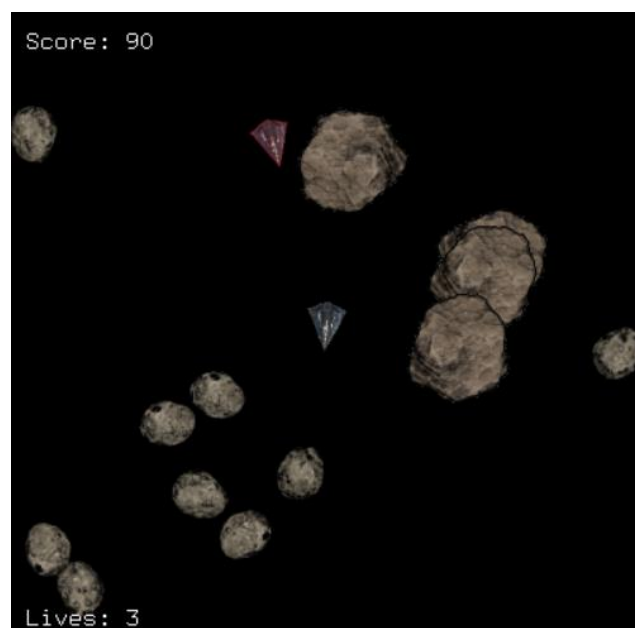


Figure 62 – Smaller asteroids appear different to their larger counterparts.

As of now, the enemy ship was currently invincible, as I had disabled its CollisionTest() method whilst defining its other functionalities. Since it would be too easy to make it die in one hit, a 'hp' variable was introduced in 'Enemy.h', of type int. By setting up CollisionTest() to return false as long as the enemy was not touching a (player's) bullet (Figure 65), logic could be implemented in OnCollision() for when this collision event did happen. What was specified was that its hp would first be decremented, and then checked to see if it was at 0. If it was, then it would be flagged for removal. If not, nothing would happen (Figure 64). Additionally, the enemy would teleport upon being hit to prevent the player from killing it too quickly by simply spraying bullets at it. The CollisionTest() method for player's bullets also had to be modified so that they disappeared when they touched an enemy (Figure 63).

```
bool Enemy::CollisionTest(shared_ptr<GameObject> o)
{
    if (o->GetType() != GameObjectType("Bullet")) return false;
```

Figure 65

```
void Enemy::OnCollision(const GameObjectList& objects)
{
    --mHp;
    Teleport();

    if (mHp == 0) {
        mWorld->FlagForRemoval(GetThisPtr());
    }
}
```

Figure 64

```
bool Bullet::CollisionTest(shared_ptr<GameObject> o)
{
    if (o->GetType() != GameObjectType("Asteroid") && o->GetType() != GameObjectType("Enemy")) return false;
```

Figure 63



Although the enemy's shooting frequency was a little more randomised, all the bullets were heading to the same place. To give the spray pattern more variance, each bullet was given an offset angle which could be anywhere between 15 degrees greater than or 15 degrees less than the angle the enemy ship was facing – which would be towards the player (Figure 66). This made it so that there was a chance of the ship 'missing' or 'tracking ahead' of the player. Making it not be accurate 100% of the time made it more fair on the player, who would now have a better chance of avoiding enemy bullets (Figure 66).

```
void Enemy::Shoot(void)
{
    mAngle += rand()%15 - 7.5;
```

Figure 67

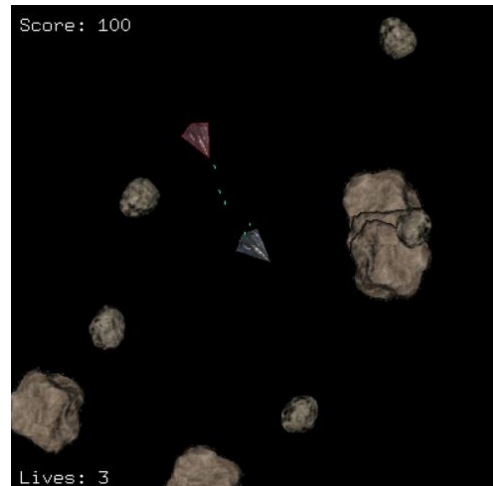


Figure 66

From here on, little optimisations were made around the code to help reduce the likelihood of the enemy 'spawnkilling' the player. The time to live for enemy bullets was decreased to 1750 (1.75s), and the respawn timer for the player was increased to 2000 (2s) so that any bullets fired by the enemy would be gone by the time the player respawned.

Since the enemy could track the player even in death, a boolean called 'mAggro' was defined in 'Enemy.h', with a setter method. The intention of this variable was to determine when to 'aggro' the player, i.e when to track and shoot at them. Calls to TrackPlayer() and Shoot() were only made possible when this mAggro was set to be true (Figure 69, Figure 70), and the state of this variable was controlled in 'Asteroids.cpp'. In the OnPlayerKilled() method, it would be set to false after the enemy teleported, offering a 'grace period' in which the enemy ship would remain passive as it waited for the player to respawn (Figure 68). Then, in the OnTimer() function, it would be set to true again once a new spaceship for the player had been added to the world again (Figure 71).

```
if (mAggro) {
    TrackPlayer(); //Hunt the player down!
}
```

Figure 70

```
if (value == CREATE_NEW_PLAYER)
{
    mSpaceship->Reset();
    mGameWorld->AddObject(mSpaceship);
    mEnemy->SetAggro(true);
}
```

Figure 71

```
}else {
    mVelocity.x = 0;
    mVelocity.y = 0;
    if (r > 95 && mAggro) {
        Shoot();
    }
}
```

Figure 69

```
mEnemy->Teleport();
mEnemy->SetAggro(false);
SetTimer(2000, CREATE_NEW_PLAYER);
```

Figure 68

The teleport method itself was also modified so that instead of the x and y positions becoming `rand() / 2`, they became `rand() + 60`, which meant that the enemy could no longer teleport within 60 units of the centre (Figure 72). This, combined with the 'mAggro' boolean to pacify the enemy at times helped to reduce the likelihood of spawnkilling happening to the point where it was impossible to play the game.

```
void Enemy::Teleport() {  
    mPosition.x = rand() + 60;  
    mPosition.y = rand() + 60;  
    mPosition.z = 0.0;  
}
```

Figure 72

To make things fairer on the player, 'Bullets' and 'EnemyBullets' were made able to take each other out, so they had a way of 'defending themselves' against the enemy. They still had to watch out for asteroids not to crash into, so it seemed like a reasonable adjustment to make.

Although the enemy ship could be killed, it did not have the functionality to respawn. Adding it was very similar to how the player's respawn functionality worked, except since there was no `OnEnemyKilled` listener, the call to respawn the enemy would have to come from the `OnObjectRemoval()` method, if the object removed was of `GameObjectType` "Enemy" (Figure 73). On this call, the game would wait 5 seconds before spawning a new enemy with a hp of 5 to chase down the player again (Figure 74). The respawn timer was set longer to allow some breathing room for the player after seemingly destroying the enemy spaceship for good.

```
if (object->GetType() == GameObjectType("Enemy"))  
{  
    SetTimer(5000, CREATE_NEW_ENEMY);  
}
```

Figure 73

```
if (value == CREATE_NEW_ENEMY)  
{  
    mEnemy->Teleport();  
    mEnemy->SetHp(5);  
    mGameWorld->AddObject(mEnemy);  
}
```


Figure 74

As it could sometimes be quite difficult to deal with the enemy, the `OnObjectRemoved()` function in the `ScoreKeeper` was modified to award the player with 50 points if they killed the enemy (Figure 75).

```
void OnObjectRemoved(GameWorld* world, shared_ptr<GameObject> object)  
{  
    if (object->GetType() == GameObjectType("Asteroid")) {  
        mScore += 10;  
        FireScoreChanged();  
    }  
    else if (object->GetType() == GameObjectType("Enemy")) {  
        mScore += 50;  
        FireScoreChanged();  
    }  
}
```

Figure 75

Lastly, the bullet colour for enemy bullets was modified to be red rather than green, by simply copying the 'bullet.shape' file and renaming it 'enemy\_bullet.shape' and switching the colour values for R and G (Figure 76, Figure 77).

 enemy\_bullet.shape - Notepad

File Edit Format View Help

strip

1.0 0.2 0.6

2.0 0.0

1.0 0.0

Figure 77

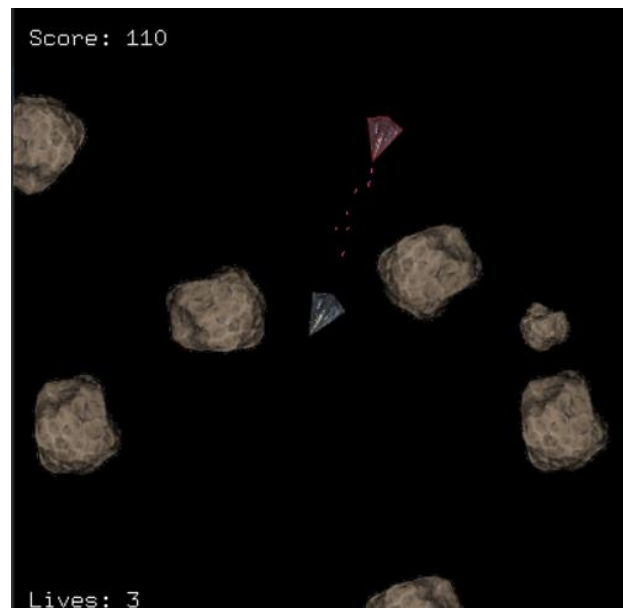


Figure 76 – The enemy now fires red bullets.