Games Technology

Part 1

Objectives:

I've been tasked to pick between implementing a start screen or splitting asteroids into smaller ones when hit. I've decided to implement a **Start Screen** for this part of the work.

The objectives for this task were:

- When the game starts, the player is shown a start game message.

- To wait for a key press event before the game is playable.

Proposed Changes:

To ensure I can meet these objectives, I am proposing the following changes:

- Extending the `void Asteroids::CreateGUI()` method by introducing labels.

- Adding a new method called StartGUI() and calling it at the start of Asteroids::Start() so that it appears first when the game is run.

Explanation of implementation:

To implement the **Start Screen** we are required to extend our CreateGUI() method to include additional information for the start screen. To do this, the CreateGUI() method was edited to include the following:

Adding new GUI components & labels

1) Creating new GUI labels that show a title screen such as "Asteroids Game". This is done by introducing a new GUILabel and adding it as a shared component.

The following code achieved this:

```
// Create a new GUILabel and wrap it up in a shared_ptr
mTitleLabel = shared_ptr<GUILabel>(new GUILabel("-->> ASTEROIDS GAME <<--"));
```

Label Visibility

2) Furthermore, to ensure that that the user did not continue to see this line after the game starts, correcting the visibility so that when a key press event happened. I achieved this by storing a variable called gameStatus, which remained false unless the **TAB** (`\'t'`) key was pressed by the player.

The code below demonstrates this implementation:

```
else if (key == '\t' && !gameStatus)
{
		mTitleLabel->SetVisible(false);

		gameStatus = true;
	}
```

Once the key press happens, we set the game state to true using this code:

```
gameStatus = true; // Mark the game as active
```

Furthermore, I had to ensure that the component didn't show on Game End, as there were other GUI labels alongside mTitleLabel. To ensure this didn't happen, I had to adjust the visibility code for "SHOW_GAME_OVER" state in onTimer.

I did this by simply adding:

```
if (value == SHOW_GAME_OVER)
{
mTitleLabel->SetVisible(false);
}
```

Rules Page

To add some extra touches, I added a rules page separate to the screen. I did this by implementing another press key "R" which represents rules. To do this I worked with Booleans. I checked if R key was toggled and if it was, I would hide start menu components. Example of the code below shows how the visibility works:

```
    else if (key == 'r')
    {
        mRulesLabelVisible = !mRulesLabelVisible;
        mRulesLabel->SetVisible(mRulesLabelVisible);

        // Toggle visibility of the title and intro labels
        if (!mRulesLabelVisible)
        {
            mTitleLabel->SetVisible(true);
            mIntroLabel->SetVisible(true);
        }
        else
        {
            mTitleLabel->SetVisible(false);
            mIntroLabel->SetVisible(false);
        }
    }
```

Declaring and adding pointer

To ensure that the Start Screen labels show during the running of the game, I needed to adjust the Astreroids.h file.

This was done by adding the line `shared_ptr<GUILabel> mTitleLabel;` under the private fields, found on line 56. If this is non-existent, our code would compile without issues but nothing would be displayed.

Avoiding bugs

During compilation of the game, I noticed that an error was thrown every time the "S" key was pressed in the main menu. To fix this, I adjusted line 91 to `if (key == 's' && mSpaceship)` rather than `if (key == 's')`

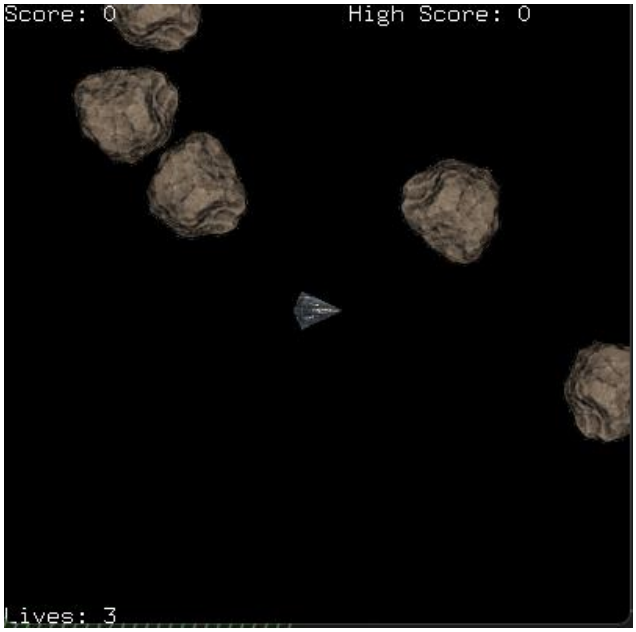
Pseudocode Example of implementation

```
if input is tab:
if gameStatus is equal to false:
    set visibility of score label to true
    set visibility of lives label to true
    set visibility of title label to false
    set visibility of start label to false
    add a spaceship object to the game world
    create 5 asteroids set
    gameStatus to equal true
```

The pseudocode example is a low level attempt to create a structured output of the implementation. Writing code like this helped me in understanding the logical concepts of the objective more thoroughly.

Game screenshots:

When the game starts, players will see this start menu, asking them to press TAB to start the game. It also tells the player the instructions they need to play.



Correct labels been displayed after the **Start Screen** event passes. The start game scene no longer shows, and the player can continue with playing the game.

Rules window when R key is pressed.

Part 2

Objectives:

The next task I chose to implement a high score table so that players could see new high scores.

The objectives for this task were:

- To display the highest score in the game.

- To appropriately grab and present highest value from local file.

Proposed Changes:

To ensure I can meet these objectives; I am proposing the following changes:

- Duplicating the mScoreLabel and introducing a new label called mHScoreLabel.

- Rather than storing the score in a local variable (for that session), instead I would grab the data from a local text file.

Explanation of implementation:

Showing High Score Counter

To show the high scores, I duplicated the GUILabel found in `void Asteroids::CreateGUI()` to extend the functionality of the high score counter.

This is done by adding (and other code such as centring and location):

```
// Create a new GUILabel and wrap it up in a shared_ptr
mHScoreLabel = shared_ptr<GUILabel>(new GUILabel("High Score: 0"));
```

This label will now be shown alongside the Score label counter.

Key Press Event

I had to also check to ensure the label was visible when gameStatus became true. I achieved this by editing the onKeyPressed Method and adding:

```
else if (key == '\t' && !gameStatus)
  {
   mHScoreLabel->SetVisible(true);
}
```

Once the player pressed TAB to start the game, this code is executed, making the high score label visible again.

Storing high score data in a text file

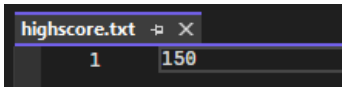
To store the high score, I had to initialise a variable that temporarily stores the sessions current score. This is simple as declaring it:

```
int highScore = 0;
```

To implement the high score, I had to use an input file stream object. This was done by using the code:

```
std::ifstream inputFile("highscore.txt");
```

The stored data was simply a numerical value:



Once we have the file, we check if the file saved to disk has a greater integer value then the value stored in our variable, highScore. We do this using simple math functions and comparing both numbers, as our aim is to store the highest possible value.

If the number in our current session is greater, we write this number to the file. This was achieved by doing:

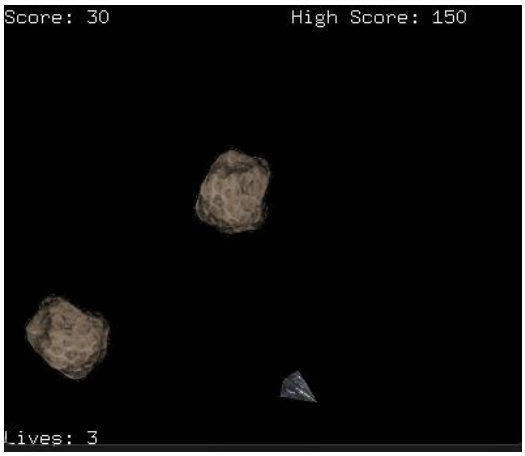
```
inputFile >> highScore;
```

Finally, I had to ensure the 'High Score' been displayed in the game correctly, with the correct number from the file. The easiest way to do this was simply updating our GUI label, mHScoreLabel and setting the text to:

```
mHScoreLabel->SetText(highScoreText);
```

To avoid conflicts with our Score change method, it was easier for me to simply update the text label with the current high score, as we did not require for the counter to be dynamic. As the information was only needed at the start of the game, I believe that the approach taken to implement this objective was the done in the most efficient way.

Game screenshots:



Game displaying high score counter alongside the current session game score.

Part 3

Objectives:

The final task was to create a demo AI that would play the game as a player would.

The objectives for this task were:

- Create an AI demo spaceship that is loaded before a player starts an actual game.

- Shoot bullets at asteroids and other objects on the map.

Proposed Changes:

To ensure I can meet these objectives, I am proposing the following changes:

- Duplicating the Spaceship logic and to create a similar type of Spaceship object.
-
Assigning demo control functions such as movement so the Demo spaceship can navigate the space world.

Explanation of implementation:

Duplicating the Spaceship Class

The task essentially requires us to create an 'AI Spaceship' that mimics human behaviour, giving it's own set of pre-defined control parameters.

Spaceship.cc -> Duplicated to AISpaceship.cpp

To ensure that there were no conflictions during runtime, we must rename all the definitions. This was done simply by adding "AI" to the start of the Spaceship term.

A similar task was carried out in AISpaceship.h, a copy file of Spaceship.h, where all references were updated to be "AISpaceship" and NOT "Spaceship".

An example of this is shown below:

```
class AISpaceship : public GameObject
{
public:
	AISpaceship();
	AISpaceship(GLVector3f p, GLVector3f v, GLVector3f a, GLfloat h, GLfloat
r);
	AISpaceship(const AISpaceship& s);
```

Creating An AI Spaceship

Just like our normal CreateSpaceship method, I also created an AICreateSpaceship method. To ensure I was using the correct pointers, I had to implement new shared_ptrs, mAISpaceship, AIBullet_shape etc.

An example of this is:

```
shared_ptr<GameObject> Asteroids::CreateAISpaceship()
{mAISpaceship = make_shared<AISpaceship>()
```

Setting pointers in Asteroids.h

To ensure that our game can reach our pointers, we must define these. I have done this by adding the lines:

```
shared_ptr<AISpaceship> mAISpaceship;
```

In addition, I have created an additional game object by adding this line:

```
shared_ptr<GameObject> CreateAISpaceship();
```

Randomising key press

To simulate AI behaviour, I created a method in my AISpaceship class that controls the demo Spaceship:

In the example below, I have shown use of KeyPress event.

```
void AISpaceship::OnKeyPressed(uchar key, int x, int y)
```

As an example, I defined "W" key movement:

```
	// Simulate random keypress for AI movement
	if (key == 'w')
	{
		// Move forward
		mVelocity += mForwardDirection * mSpeed;
	}
```

Due to time constraints, I was unable to fully complete the necessary lines of code to make this part of the codebase work. Therefore, I have commented out the code to allow the game to still compile.