Games Programming 3

Coursework

Text

Description automatically generated

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

*Signature*.

Aaron Jeffers

S1802126

Contents

[Introduction 2](#_Toc93499158)

[Model Behaviour 2](#_Toc93499159)

[Camera Behaviour 4](#_Toc93499160)

[First Person Camera 4](#_Toc93499161)

[Third Person Camera 5](#_Toc93499162)

[Chase Camera 5](#_Toc93499163)

[Asteroid Camera 6](#_Toc93499164)

# Introduction

This project set out to develop an advanced camera system as part of an extension material for a game. The game itself is a three-dimensional ‘asteroids’ type game in which the player controls a ship moving in a space and can shoot homing missiles in order to destroy the asteroids. Along with this the player has a choice of five different camera types listed as follows: First-Person, Third-Person, Chase, Missile and Asteroid. The functionality and application of these different cameras will be explained in the proceeding sections.

# Model Behaviour

As will be discussed further on in this document, the model behaviour is central to the camera behaviour. This is a result of the fact that the camera functionality makes use of various model transform components in order to perform it’s calculations.

Text

Description automatically generated

Figure 1. GameObject.cpp methods

Figure 1 above shows the methods of the GameObject.cpp class, a class that both the missiles and ship models inherit from.

Graphical user interface, text

Description automatically generated A screenshot of a computer

Description automatically generated with low confidence

Figure 2. GameObject.h variables Figure 3. GameObject.h methods

Figure 2 shows the variables held by each model inheriting from GameObject and figure 3 shows the return methods for these variables accessible from another class. The vectors in figure 2 represent the forward, up and right directional vectors held by each model. The moveForward() method utilises these in order to move a model along it’s forward vector as opposed to using world coordinates. The yaw() method rotates the model about the world z-axis, as such the directional vectors are changed so the calculateForward() method get’s called. This function is responsible for calculating the new directional vectors after yaw() is called, it works by calculating the rotation value about the z-axis, then multiplying this by the forward and right vectors before normalising the result. The up vector is the cross product of the former two vectors. This method is effective for the ship model as it’s position is controlled by user keyboard inputs which call the yaw() and moveForward() functions. The homing missiles however act autonomously once fired as they seek out their target and thus a further method setting the forward vector towards the target is required. The setForward() method takes in a position vector and sets the forward vector of the missiles towards this target.

Text

Description automatically generated

Figure 4 MainGame:fireMissiles() method

The homing missile behaviour is seen in figure 4 above. It functions by calculating the target vector, that is the target asteroid’s position minus the missile’s position. The target velocity is then next calculated by the normalisation of this target vector time’s the missile speed (missile velocity \* delta time). Next the rotation is calculated as 360 degrees converted to radians times the normalised target vector. Finally the transform of the missile is updated with these values. Multiple instances of this class can be called at once by each active missile. Each missile is set active individually (on the ship’s current position as seen in figure 5) when the player presses the space bar and is only set inactive upon a collision between the missile and an asteroid. Each missile seeks out it’s corresponding asteroid such the missile[n] will seek out asteroid[n] where n is an integer representation of each array element. Once the missile reaches it’s target and collision has been detected both the missile and asteroid are set inactive and a ‘bang’ sound effect is played to indicate a successful strike.

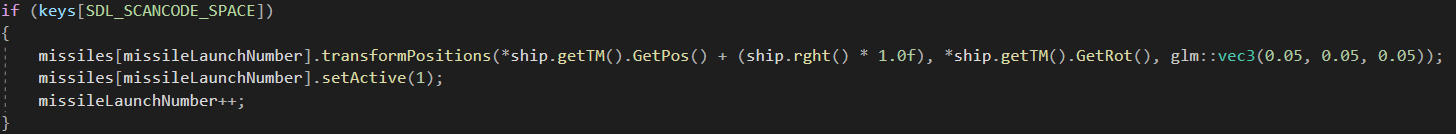


Figure 5. Missile launch player input

# Camera Behaviour

As stated in the introduction the camera has 5 separate behaviours. Upon loading the game the default camera is instantiated set at zero for the world’s x-axis and y-axis but with an offset along the z-axis. Upon player input another camera type may be selected. The functionality of these is explained below.

Text

Description automatically generated

Figure 6. setCameraTarget() initial variables

Figure 6 shows the variables used by the setCameraTarget() method. Each time the method is called the ship’s directional and position vectors are gathered by calling the methods in the GameObject .h file as seen in figure 3. Furthermore, this method contains a switch statement to handle the different camera states, denoted by an integer variable that is set dependent upon player input in the processInput() method.

Within the camera class itself (camera.h) there are 3 methods that get called from setCameraTarget() which must be understood to comprehend the following methods.

1. setForward(vec3 fwd): Sets the forward vector of the camera equal to fwd.
2. setLook(vec3 modelPos): Sets the forward vector of the camera equal to modelPos minus the camera’s position.
3. setPos(): Set’s the camera’s position.

## First Person Camera

Text

Description automatically generated

Figure 7. First Person Camera Code Snippet

The first-person camera has quite simple functionality in that it sets the camera’s forward direction and position equal to that of the ship’s with an offset applied along the forward vector.

## Third Person Camera

Graphical user interface, text, application

Description automatically generated

Figure 8. Third Person Camera Code Snippet

The third-person camera works in much the same way as the first person however the positional offset is applied in reverse along the forward vector and positively along the ship’s up vector. This gives the effect of the camera following the ships from behind with a slight vertical offset in order to see past the model.

## Chase Camera

Text

Description automatically generated

Figure 9. Chase Camera Code Snippet

The chase camera is slightly more complicated than the former two. It follows the ship as it moves around the world space but also allows the user to pan the camera around the ship’s model. The positional offset vector is calculated as x = sin(theta) \* cameraOffset, y = 0 and z = sin(theta) \* cameraOffset, where theta is the angle of rotation about the relative y-axis or up vector of the ship and cameraOffset acts as the radius of angular rotation about the ship. Theta is set in the processInput() method by the player so they can pan around in real time. The setLook() is called with the ship’s position plus a slight offset along the forward axis as the local position of the model is at the rear. Finally, the camera’s position is set to that of the ship’s plus the relative offset.

## Asteroid Camera

Text

Description automatically generated

Figure 10. Asteroid Camera Code Snippet

The asteroid camera’s function is to set the camera to look at an individual asteroid. It does so by calling the setLook() method with the asteroid’s position. Which asteroid from the array of asteroids is selected is denoted by the integer asteroidCamTarget, which can be cycled through by the player pressing z/x, each raising or lowering the integer by one. If this integer exceeds the length of the array it is cycled around to the other end, for example negative one will become the length of the array minus one to be the final array element. The position of the camera is not altered with this method however, the player may choose to move the camera along the new set forward axis using the scroll-wheel.

## Missile Camera

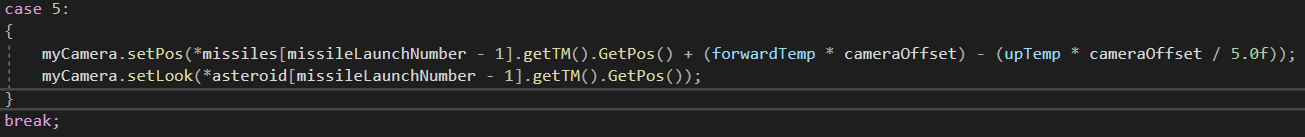


Figure 11. Missile Camera Code Snippet

The missile camera tracks the most recent missile that has been fired until it reaches the target position. This is achieved by setting the camera’s position equal to that of the missile plus an arbitrary offset. The camera is also set to look at the target, in this case an asteroid, to give the illusion the missile is being tracked into the destination. If the player presses space multiple times in this mode, the camera will update to track the most recent missile.