# Cover Page

Logo, company name

Description automatically generated

Phillip Ross

S1905606

Bsc Software development (computer games)

Word count: 1637

GitHub link: https://github.com/PhilUchiha/GP3

*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*.

*Phillip Ross*

Contents

[Cover Page 1](#_Toc124710728)

[1.0 Code description: 2](#_Toc124710729)

[1.1 Basic camera movements: 2](#_Toc124710730)

[1.2 Moving the ship/model: 3](#_Toc124710731)

[1.3 Look at. 5](#_Toc124710732)

[1.4 Chase camera 6](#_Toc124710733)

[1.5 First person camera 7](#_Toc124710734)

[1.6 Implementing missiles 8](#_Toc124710735)

[1.7 Implementing multiple types of cameras together 9](#_Toc124710736)

[1.8 Code optimisation/abstraction 10](#_Toc124710737)

[2.0 References 10](#_Toc124710738)

# 1.0 Code description:

## 1.1 Basic camera movements:

The area in which I am most interested in from what we have worked on is cameras. There are many different types of cameras that could be used in a game where the player controls a ship around asteroids. Some of the most obvious examples are chase/follow cameras which can follow the ship as it is controlled by the player. As well as more simple options such as a static camera if the game’s view did not need to move and instead all the objects in the scene would move. I will be researching how to do many of these camera types and then attempting to implement them into our project. In order to iterate on our camera movement, we need a starting point which will be the following code given in the games programming three labs:

case SDLK\_LEFT:

myCamera.MoveLeft(10.0f\*deltaTime);

break;

case SDLK\_RIGHT:

myCamera.MoveRight(10.0f\*deltaTime);

break;

case SDLK\_UP:

myCamera.MoveUp(1.0f);

break;

case SDLK\_DOWN:

myCamera.MoveDown(1.0f);

break;

This code allows for the camera to be moved through previously created functions and the float variable given to the function acts as a multiplier for how far the camera will move for every time the user presses the set input. Multiplying this value by delta time will integrate the value over that second.

A screenshot of a computer

Description automatically generated with medium confidence(Figure 1 – Functions for basic camera movement)

## 1.2 Moving the ship/model:

The first piece of extension material I am attempting is to move a chosen model in the scene after this we will aim to create a chase camera that will follow the model as it moves/rotates.

The code I have created for this within the same switch statement I have used for the camera movement is:

case SDLK\_w:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x, ship.getTM().GetPos()->y + 1.0f \* deltaTime, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_a:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x + 1.0f \* deltaTime, ship.getTM().GetPos()->y, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_s:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x, ship.getTM().GetPos()->y - 1.0f \* deltaTime, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_d:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x - 1.0f \* deltaTime, ship.getTM().GetPos()->y, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

This code is designed so that the programmer can easily use this code to change the position of the model (which will allow it to move) as well as the rotation of the object if they wish to do so and even the scale of the object if they wanted to increase or decrease its scale. In order to create this different functionality, the only section that would need to be changed is the +1.0/-1.0 as this controls what is changed as well as by how much and can be multiplied by delta time just like the camera movement in order to total up and integrate the values over the set time interval.

**Here is the code for altering the ship’s rotation:**

case SDLK\_t:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x, ship.getTM().GetPos()->y, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y + 1.0f \* deltaTime, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_f:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x, ship.getTM().GetPos()->y, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x + 1.0f \* deltaTime, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_g:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x, ship.getTM().GetPos()->y, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y - 1.0f \* deltaTime, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_h:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x, ship.getTM().GetPos()->y, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x - 1.0f \* deltaTime, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

**Here is the code for altering the scale of the ship model:**

case SDLK\_i:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x, ship.getTM().GetPos()->y + 1.0f \* deltaTime, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_j:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x + 1.0f \* deltaTime, ship.getTM().GetPos()->y, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_k:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x, ship.getTM().GetPos()->y - 1.0f \* deltaTime, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

case SDLK\_l:

ship.transformPositions(glm::vec3(ship.getTM().GetPos()->x - 1.0f \* deltaTime, ship.getTM().GetPos()->y, ship.getTM().GetPos()->z), glm::vec3(ship.getTM().GetRot()->x, ship.getTM().GetRot()->y, ship.getTM().GetRot()->z), glm::vec3(ship.getTM().GetScale()->x, ship.getTM().GetScale()->y, ship.getTM().GetScale()->z));

break;

Due to us being able to access the ship’s position, rotation and scale it can be useful to control these with inputs while in the scene as when creating an asteroid game, the developer may wish for certain abilities or power ups to be available that would change the size of the ship or its rotation. These tools allow for us to see how the model would look for testing purposes. This code is very simple to understand and allows for a large amount of freedom when deciding what value to alter these variables by as well as what attributes we want to change.

## 1.3 Look at

The first new type of camera to create will be the “look at” camera. This camera involves keeping a chosen target (in our case the ship) in the centre of our view. This is different from the chase camera that will also be made as the camera does not move with the model but instead just tilts the view to always be looking at the chosen model.

**The code for the look at camera is as follows:**

myCamera.setLook(glm::vec3(ship.getTM().GetPos()->x,ship.getTM().GetPos()->y, ship.getTM().GetPos()->z));

Text

Description automatically generatedThis piece of code is added to the switch statement that features the w a s d movement. This code uses the setLook function which keeps the camera on the model.

A picture containing dark, night, light, night sky

Description automatically generated(Figure 2 setLook function code)

(Figure 3 Look at camera 1)

A picture containing dark, night sky

Description automatically generated(Figure 4 Look at camera 2)

As shown in the above images no matter how the ship moves the camera will always be aimed towards it.

## 1.4 Chase camera

To create a chase camera, we need our camera to mimic the movements of its target which will be the ship. In order to do this, we use the following code.

**Code for chase camera:**

myCamera.MoveUp(5.0f \* deltaTime);

myCamera.MoveRight(5.0f \* deltaTime);

myCamera.MoveDown(5.0f \* deltaTime);

myCamera.MoveLeft(5.0f \* deltaTime);

Each of these lines of code are added individually to their respective movement keys (w a s or d). This code uses 5.0f \* deltatime as that is the same number that is used by the ship for movement. This allows the camera to follow the ship without any error in it lining up. Whenever this number is altered either in this camera code or in the ship movement code it will need to be changed in both so that they always mirror one another.

Background pattern

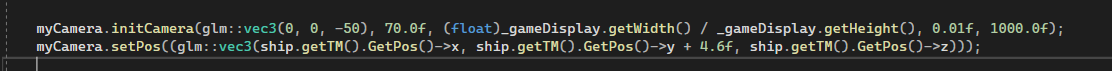
Description automatically generatedA group of jellyfish in the water

Description automatically generated with medium confidence(Figure 5 Chase camera 1)

(Figure 6 chase camera 2)

## 1.5 First person camera

A third camera we can implement is a first-person camera as this mostly requires us altering the position of the camera as well as the direction it faces. This is because we need the camera to be on the tip of the ship for it to truly seem first person to the player.

The code for this is as follows:

(Figure 7 first-person camera)

In this code we simply initialise the camera and then set the position of the camera to be that of the ship with 4.6 added to the y coordinate so that it lines up with the tip of the ship. This code when used with the chase camera movement code set to WASD will allow the player to move the ship while maintaining the camera position on the tip of the ship model.

Background pattern

Description automatically generated(Figure 8 First-person camera view)

## 1.6 Implementing missiles

Text

Description automatically generated with low confidenceA key part of this asteroid game is that the ship must fire missiles upwards, and they must have collision detection as well. For this we create a fireMissiles function and add it to an input key (left click on the mouse in this case)

(Figure 9 code for missiles input)

**Text

Description automatically generatedThe fireMissiles function is as follows:**

(Figure 10 fireMissiles function)

This code takes from the drawMissiles function and uses the array of missiles. It spawns the missiles on the ships position when the click happens. This prevents any spawning issues when the ship moves. Each click it adds one to the array, this is important because the array works as an ammo system as well since there is only a limited number of missiles in the array and when the array reaches 20 left click will no longer fire missiles.

**Text

Description automatically generatedThe drawMissiles function is as follows:**

(Figure 11 drawMissiles function)

This function is almost the same as the drawAsteroids function that is used for that array of meshes. The main difference is that a rimShader is applied to the missile meshes.

## 1.7 Implementing multiple types of cameras together

There are a few different methods we can use to implement multiple types of cameras at the same time. One method would be to have different camera objects that we can switch between with inputs. This method would involve changing from using the myCamera object only and instead having multiple camera objects and a variable that switches between them. For example, there could be “myCamera1” that is a chase camera and in a separate object of “myCamera2” we could have a still camera. The developer can choose how many of these to include as well as set them to start at different positions in the world space if they choose or at different angles. The method I will use is even simpler however and does not require us to create new camera objects as it will use the same camera but through different inputs can act in different ways. In this case W A S D will be the chase camera

Graphical user interface

Description automatically generated with medium confidence(Figure 12 WASD chase camera)

Text

Description automatically generated with medium confidenceNow using a different input of T F G H, we can implement our lookAt camera.

(Figure 13 TFGH lookAt camera)

The disadvantage to this method is that if there were many more cameras it would become a problem as we have a limited number of potential inputs but as we are only using a couple of cameras this will not be a problem. This method also allows us to see how these cameras can combine as when the look at camera is used it changes the viewing angle of the camera to the ship. This means that when the chase camera is then used it follows from a different angle which makes the game feel less flat and more like the user can move around the 3D space freely.

## 1.8 Audio system changes

Text

Description automatically generatedOnce we have our basic audio system set up, we need to make some modifications to allow it to fit our game. There are no changes needed for the background music but for the “bang” sound that plays when missiles are fired, we need a few changes.

(Figure 14 playing the missile sound)

Text

Description automatically generatedThe way that we play the missile sound is by playing it on the left click input. For the sound to play again afterwards, we must stop the audio. This is done after a delay which is a new function, I’ve created so that the sound can play fully before being stopped to allow for it to play again with the next shot.

(Figure 15 delay function)

Text

Description automatically generatedI also had to create the stop audio function which is similar in design to the play audio function, with a few differences.

(Figure 16 stopAudio function)

This function checks the state of the audio device and if the device is in the playing state it will stop the sound coming from it until it is played again with left click.

## 1.9 Code optimisation/abstraction

Another area that is also important to a project such as a game is optimisation. When possible, it is always useful to try and do as much cleaning up of code and memory management. In this project that means we will remove unnecessary code and comment out any code that is useful but not needed for certain specific tests. In C++ when optimising it is also important to consider passing by reference and using pointers as much as possible when using certain parameters in functions. This can be seen in our initModels function in our mainGame used with the asteroid parameter.

(Figure 17 passing by reference)

There are quite a few functions in main game no longer being used and some that have unneeded code as there are many shaders no longer being used including the geometry shader and the fog shader so these can be removed to optimise as well as to keep the code more readable.

Abstraction is also a valuable process to use in our project. This involves us hiding any unnecessary information, we do this through the use of header files and keeping any variables or functions not needed elsewhere in code private.

# 2.0 References

<https://learnopengl.com/Getting-started/Camera>