**CMP405 Coursework Report**

**Callum Myers**

**2102549**

1. **Summary**

Of the features which were outlined in the assessment brief, I chose to implement the following features:

**Usability:**

* Object selection via mouse picking.
* Multiple object selection via mouse picking.
* Use of click and drag to move, rotate and scale objects.
* User-defined camera speed parameters.
* Camera focusing on the selected object.
* Arcball camera orbiting.

**World Editing**

* Copying and pasting of the selected object.
* Object creation window.
* Object manipulation.
* Usage of multiple cameras (the object focusing uses its own separate camera to preserve the position of the free-moving camera, and both use different functions).

1. **Controls:**

**Camera Controls:**

* Use A and D to move the camera left or right.
* Use W and S to move the camera forwards or backwards.
* Use Q to move the camera up, and E to move it down.
* Hold the right mouse button and move the mouse to rotate the camera.
* Press F to focus on the object which is currently selected. Whilst in this mode, hold the right mouse button and move the mouse to use the arcball camera.
* Press R to exit the focus mode and return to the previous camera position and allow free camera movement again.

**Mouse Picking:**

* Press the left mouse button whilst hovering over any object in the scene to select it.
* Hold shift whilst selecting objects to select them all at once. This will allow you to manipulate them all simultaneously.

**Object Functions:**

* Hold control and press V to paste the object currently being selected, at a slight offset to the current object.
* When using any of the object manipulation modes, hold control and move the mouse to manipulate in the X and Y axis, or press W/S whilst holding control to manipulate it in the Z axis.
* Press delete to remove the currently selected object from the scene graph.

**UI Menus:**

* To edit camera movement speed and rotation speeds, go to File->Camera Controls, and use the sliders to adjust.
* Use the modes menu to select between normal (free moving camera), translate (move objects), scale (change object sizes) and rotation (rotate objects).
* To create a new object, go to Edit->Create Object, and fill out fields in the dialogue window which appears.
* Using the red paste button has the same effect as pressing ctrl+V and will paste the currently selected object.

1. **Features**
   1. **Mouse Picking**

Mouse picking is common in game engines and simply allows the user to select the object they want to edit from directly within the scene, making it easy to select the correct one. I decided to adapt this to also allow users to select more than one object at once, which is a feature found in both Unreal Engine and Unity, and lets the user do all the usual editing features, such as transforming, deleting or copying, but with multiple objects at once. This improves user experience because it saves a lot of time when they need multiple objects grouped together. For example, if the user were adding trees to a field, they may want to paste multiple at the same time to fill the field quicker.

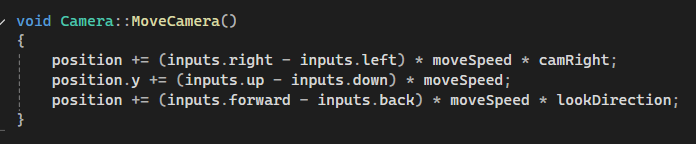
* + 1. **Single Mouse Picking**
    2. **Multi-Object Picking**
  1. **Cameras**

The plan for the camera was to have two separate camera types. One would be a normal camera, where it can be moved around or rotated by the user, then a second camera which would allow the user to take a closer look at specific objects in the scene. The idea for this came from a combination of the scene cameras which are available in the Unity game engine. In the standard camera, Unity allows the user to move freely or select an object in the hierarchy and press F to go directly to it. It also has a prefab inspector, where the user can select a prefab from the content window and inspect its model. I decided to combine both features and put them both into the scene itself, where focusing onto an object gives you the same view you would get in the prefab inspector, but in the scene context.

Doing it this way improved the WOFCC program because it makes it easier to decide where to place objects and you can look around it to see how it looks beside the objects near it.

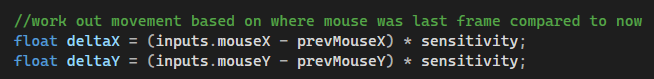
* + 1. **Camera Movement**

Moving the camera simply involves updating the camera’s position vector as shown here:

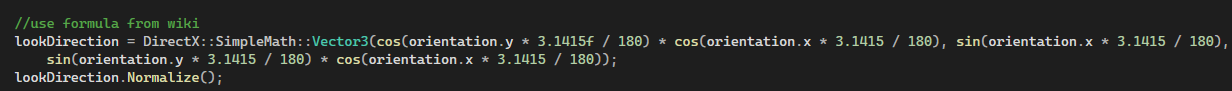


* The first line adds to the camera's position vector by taking the combined input of the right button – the left button (this would return 1 if holding D, and -1 if holding A, hence saving multiple if checks for each individual key), multiplying this by the camera’s movement speed and multiplying by the camera’s right vector to ensure the user always moves sideways based on where the camera is currently facing.
* The second line uses the same logic to affect the Y position. Note this could also have been done by adding to the entire vector with this same line of code multiplied by the camera’s current Up vector, but in this case, it directly affects the Y component individually for simplicity.
* The third line is the same again but will move towards or away from the camera’s current lookDirection, allowing the user to move forwards or backwards relative to the current view.
  + 1. **Camera Rotation**

The camera rotation is adapted from the formulas given in the lab, but uses the formula given on the wiki instead, and calculates rotation using mouse inputs instead of button presses. To do this, the rotation function is run whenever the user is holding the control key, and calculates how much the mouse has moved in the X/Y axis since the last frame, by subtracting the mouse’s position in the previous frame from its’ position in the current one, then multiplying by the sensitivity scalar value, as seen below:



These values are then added to the camera’s current orientation values, with the orientation.x being clamped between -89 and 89 because going beyond these values causes the controls to invert when looking up and down. Finally, the lookDirection vector is calculated using the formula from the wiki, before being passed back to game.cpp.



* + 1. **Camera Focusing**

When focusing on an object, the program simply activates the focus camera instead of the standard camera, which prevents the user from moving around using standard controls. Then, the FocusOnObject function in camera.cpp is run. This moves the camera position to match the object’s position, then applies a slight offset to keep the object in view. It then changes the camera’s lookAt point to be the position of the object and subtracts the camera’s position from this value to create the lookDirection vector, keeping it focused on the object.

* + 1. **Arcball Camera**

To implement the arcball camera used when focused on an object, I got the logic from a blog called A Slice of Rendering. The ArcballCamera function uses the same code as above to calculate mouse movement and then uses the built in SimpleMath function CreateFromAxisAngle to rotate around the X and Y axis accordingly. Then, using SimpleMath::Transform, the camera is rotated relative to the object’s position as opposed to its’ own, then by adding the object’s position back to the resulting matrix the camera is moved back into world space. Finally, the lookDirection is calculated by subtracting the camera’s updated position from the object position so that its’ view always keeps the object central.

* + 1. **Camera Switching**

To implement the camera switching, I first had to create a custom class to create multiple cameras. To do this, I simply moved any camera-related functions and variables from the game.cpp file to the camera.cpp file. I then made an Update function in this camera class which runs at the same time as the game.cpp Update function, and this is where I handle camera rotation, movement, etc.

Next, I created three instances of this camera class in the game.cpp file. One handles the free-moving camera, one handles the camera which focuses on an object, and one is an empty instance which has its’ variables such as position, view, lookDirection and the right vector constantly set to whichever of the other two cameras is currently active. The reason I set the system up this way is because there are multiple functions such as Draw and SetView within game.cpp which take in variables from the camera, so by having one controller instance it allowed me to set all of these functions to use this instance’s variables, meaning I did not have to use if then else checks on all of these functions individually, which could significantly decrease performance of the project.

Switching between the cameras simply works by updating the “cameraActive” bool within each camera instance appropriately, and a single if statement at the beginning of game.cpp’s Update function will set the camera controller’s variables to the camera which currently holds true in this variable.

* 1. **Object Editing**

Copying and pasting objects is a feature which is available in most game engines and is extremely useful when placing commonly used objects. By adding it to the WOFFC program, level designers would be able to simply create one instance of an object, and then quickly use it as many times as they like throughout the scene, without having to manually add more objects with the same variables, which can quickly become cumbersome and time consuming.

* + 1. **Copying**

When the user selects an object, the ID of the object is added to a vector which stores any IDs currently selected. This ID links to the objects position in the SceneGraph vector.

* + 1. **Pasting**

When the user presses control+V, or the paste button in the toolbar, the PasteObject function in toolMain.cpp is run. This function first takes the ID of the selected object and creates an exact copy of it into a new SceneObject, matching all the same variables to ensure they all have a valid value. Next, it will assign this new object its’ own ID, which is the SceneGraph.size() + 1 to ensure it is unique. The X and Z position of the new object is then slightly edited to give an offset to the position, before adding this new object to the scene graph. This process is then repeated for all objects currently selected. Finally, the display list is updated in game.cpp to add this new object onto the scene.

* + 1. **Object Deletion**

Another feature which is extremely useful for engines is having the ability to delete objects entirely. Implementing this was relatively simple since the scene graph is stored as a vector and therefore has a built-in erase function. By simply running this function on the object at the selected ID, it is removed from the scene graph, then the display list is rebuilt to reflect this on the user’s screen.

* 1. **Object Manipulation**

Object manipulation is vital for any game engine to allow designers to move objects around the scene. It was intended to be based on the Unreal Engine object manipulation modes, where it would display the x, y and z axis around the object, however I could not work out how to display this, so I had to simplify it. This is an extremely useful addition to the WOFCC program because it allows the user to see in real-time how the object will look when manipulated, instead of constantly having to run SQL queries on the database file and then reload the program to see how it looks in the scene.

* + 1. **Using Mouse Movement**

These functions all use the same mouse logic as the camera, by subtracting its’ new position from its’ position in the last frame, so I will refer to the movement as deltaX and deltaY, the same as in the code.

* + 1. **Translating Objects**

When the mode is switched to Translate in the menu, the state in the enum of InputCommands is switched to Translate, which will then run the Translate function in toolMain.cpp during every frame where the user holds down control. This function:

* Takes in the ID in the scene graph of the object(s) being selected.
* Changes the object’s posX and posY component based on the mouse deltas, moving the object in the X or Y axis.
* Changes the object’s posZ component based on whether the user is holding W or S, moving the object forward or back in the Z axis.
* Sets the object’s components in the displayList to match its’ components in scene graph using UpdateDisplayList in game.cpp. This just updates the object on the user’s screen, since the game will display objects based on their displayList values, not what is stored in the scene graph. This function works the same as the pre-existing BuildDisplayList, however it is rewritten to target only the currently selected object, since rebuilding all objects multiple frames in a row was causing serious performance issues.
  + 1. **Rotating Objects**

When the mode is switched to Rotation in the menu, the state in the enum of InputCommands is switched to Rotate, which will then run the Rotate function in toolMain.cpp during every frame where the user holds down control. This function:

* Takes in the ID in the scene graph of the object(s) being selected.
* Changes the object’s rotX and rotY component based on the mouse deltas, rotating around X or Y axis accordingly.
* Changes the object’s rotZ component based on whether the user is holding W or S, rotating around the Z axis.
* Runs the UpdateDisplayList function in game.cpp again.
  + 1. **Scaling Objects**

When the mode is switched to Scale in the menu, the state in the enum of InputCommands is switched to scale, which will then run the Scale function in toolMain.cpp during every frame where the user holds down control. This function:

* Takes in the ID in the scene graph of the object(s) being selected.
* Changes the object’s rotX and rotY component based on the mouse deltas, scaling in the X or Y axis accordingly. This allows the user to make the object wider or taller as required.
* Changes the object’s rotZ component based on whether the user is holding W or S, rotating around the Z axis. This will increase or decrease the object’s breadth.
* Runs the UpdateDisplayList function in game.cpp again.
  1. **Object Creation**

The object creation window is designed to be a simplified way to add new objects into the scene. Without this window, adding a new object would require manually adding it into the database, by going into the .db file and running an SQL script. With the window however, the user can select the model, texture and transform of the object they wish to add into the scene. This is also a safer way to add new entries to the database, since created objects get most of their field values from an object which is already added to the database, which will help to prevent accidental invalid values being entered into the database, which could potentially cause the engine to crash.

* + 1. **Using the Window**
    2. **Creating the Object**
    3. **Adding to Scene**

1. **Conclusion**
2. **References**

<https://learn.microsoft.com/en-us/cpp/mfc/reference/csliderctrl-class?view=msvc-170> – Using the sliders in Forms (MS Documentation), used in camera controls window to edit the speeds.

<https://stackoverflow.com/questions/21865034/how-to-set-a-default-value-for-edit-control-box-in-a-dialog-that-is-added-to-m> - Setting the default value in edit boxes (StackOverflow), used in object creation window to enter default position, rotation and scale.

<https://learn.microsoft.com/en-us/cpp/mfc/reference/ccombobox-class?view=msvc-170> – Using the combo boxes in Forms (MS Documentation), used in object creation window to create the dropdown menus for models and textures.

<https://github.com/microsoft/DirectXTK/wiki/Rendering-a-model> - Adding new models (WFCC Wiki), used to add the fence model to the project.

<https://github.com/microsoft/DirectXTK/wiki/Matrix> - Arcball camera functions

<https://asliceofrendering.com/camera/2019/11/30/ArcballCamera/> - Using the arcball camera (Programmer Blog), used for the object focusing camera.