World of Flim-Flam Craft Editor

CMP405: Tools Programming

Dylan James Black

Contents

Chapter 1 Summary

An extensive review of current world editing software was carried out before beginning development. As a result, various features were decided upon for implementation. Chapter 3 includes a comprehensive discussion of each feature, whereas Table 1-1 briefly describes them:

Enhancement	Feature
Usability	Mouse use
Usability	Camera improvements
Usability	Object highlighting
Usability	Actions
World Editing	Object inspector
World Editing	Light inspector
World Editing	Spawn inspector
World Editing	Terrain inspector
World Editing	Paint inspector
World Editing	Sculpt inspector
Efficiency	Manager classes

Table 1-1 Summary of features

Chapter 2 Controls

2.1. Camera

- Respecting the control scheme of a typical computer application, the WoFFC Editor utilises WASD keys for camera movement.
- Keys Q and E translate the camera along its up vector.
- Hold the middle mouse button to enable mouse track movement.

2.2. Global Key Mappings

- CTRL + Z to undo action.
- CTRL + SHIFT + Z to redo action.

2.3. Dialogue Specific

It is worth noting, some functionality is only accessible when the corresponding dialogue is active. The following table details how both mouse picking and specific key mappings are used for each inspector, as well as displaying their designated toolbar button:

Object or Light Inspector *Will only allow selection of either objects or lights, respectively.*	● 💡
Mouse Picking	
Transform option disabled, right click to select objects.	
Transform option enabled, right click and drag to manipulate objects.	
SHIFT + right click to select multiple objects.	
CTRL + right click to deselect objects or right click away from object(s).	
Key Mappings	
DEL to delete selected objects.	
CTRL + X to cut selected objects.	
CTRL + C to copy selected objects.	
CTRL + V to paste selected objects.	
Spawn Inspector	
Mouse Picking	
Right click to spawn selected object type.	

Terrain Inspector
Mouse Picking
When sculpting isn’t selected, right click to select terrain.
When sculpt mode is selected, hold right click to perform the chosen sculpt on selected terrain.
Paint Inspector
Mouse Picking
Right click to select terrain. *Selected paint is applied to terrain when picking.*
Sculpt Inspector
Mouse Picking
When sculpt mode is selected, right click to select terrain. *Selected sculpt is applied to terrain when picking.*

Table 2-1 Dialogue control definitions

Chapter 3 Features

The implemented tool system for this project operates almost entirely using static manager classes. Each of these classes is responsible for a specific set of functionalities. As the tool system manipulates large amounts of data, it was decided early in development to use this efficient structuring method. It is also worth noting that due to the order of the database being unreliable, all scene graph functionality is controlled using their ID column. The following sections detail each feature in turn, describing how they are made possible through the corresponding manager class – where applicable.

3.1. Mouse Use

3.1.1. Design

Allowing the user to interact with the editor via mouse input is a vital feature for any editor software. Without it, the editing capabilities of the program would be heavily restricted to operating by UI and/or key inputs only. The design of mouse functionality within the tool system has been influenced by Blender (Roosendaal, 2020). This model editing software utilises the right mouse button for transforming objects based on its position after being dragged. The software also uses the right mouse button to select objects, while the left button deselects them. It could be argued that alternating between mouse buttons like this might confuse the user, especially if they have only started using the software (Falconi, 2010, pp. 21-22). Therefore, the WoFFC mouse editor has discarded the use of the left mouse button to reduce complexity. Instead, the user can either select an empty space or hold CTRL to deselect objects.

3.1.2. Execution

All mouse functionality is implemented in the Mouse Manager class, detailed in the following points:

- **Picking Objects and Lights**
  Depending on the current editor mode, calling this function will return the ID of object(s) or light(s). For efficient use of mouse

picking, this function automatically sets up a ray trace from the current mouse position along the forward vector of the camera. All available objects are then looped through and their meshes checked for an intersection with the ray trace. If the check returns true, the current ID of the object is stored. The loop continues to check if a closer object has been intersected before returning the final intersected object ID. The returned ID is then simply added to the stored object IDs within the primary tool class for editing elsewhere.

- **Picking Terrain**
  A similar function is called which returns the intersected terrain. This function loops through the entire terrain size, checking if either triangle in the current quad intersects with the ray trace.

- **Picking Spawn**
  A default distance from the mouse position is initialised as ten. The terrain function is then called to check for an intersection. If successful, the distance is updated to meet the returned terrain and the Y value of the final point is increased by one. This results in a spawn point either ten units along the camera forward vector or slightly above a picked piece of terrain.

- **Dragging**
  Upon each right mouse button press, the current position of the mouse is stored. While the button is held, the position storage is updated, and the previous position is declared as such. This allows the tool system to determine whether the mouse has been dragged or not by comparing these two positions. The Mouse Manager class contains this function and is used throughout all object/light manipulations.

3.1.3. Benefit to Editor

The mouse functionality utilises commonly desired features for world editing tools. The performance of the mouse allows designers to easily make changes to objects. Thus, removing the sole need for hard input values such as coordinates (Krogsæter, 2009, pp. 38).

3.2. Camera

3.2.1. Design

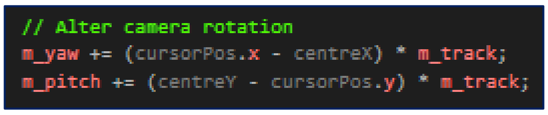
Like mouse functionality, an appropriate implementation of a dynamic camera is another key feature of a successful world editor. While most computer programs use similar controls for camera movement as used in this project, the camera design has been tailored specifically to mimic that of Unreal Engine 4 (Epic Games, 2014). The game engine is an excellent representation of a dynamic camera, as it allows users to effortlessly navigate their game worlds. Unlike the engine, the camera class in this project permits mouse tracking movement by holding the mouse wheel. This has been implemented to comply with the design of the mouse input, explained in the previous section.

3.2.2. Execution

Rather than controlling the camera through a manager class, the camera is declared as an object and used throughout the primary tool class only. All functionality of the camera can be encapsulated as the following:

- **Input**
  As the camera is one of two chief features for world navigation, it makes use of a continuous input function. This function handles all incoming user input relating to camera control. Each input key is checked before applying any updates to the position. Once a check returns true, the dedicated function is accessed. If the camera is not focussing on an object, the movement functions simply update the camera position by the coherent vector (forward, right or up) by the current speed value. Otherwise, only the appropriate parts of the

position vector are updated to ensure a smooth translation. If the check for the mouse right button returns true, the yaw and pitch of the camera is updated. This is done by calculating the distance between the current mouse position and the centre of the screen, before multiplying it by the tracking value. Note this calculation is only applied if the camera is not focussing on an object. Figure 3-1 shows this in a code snippet.

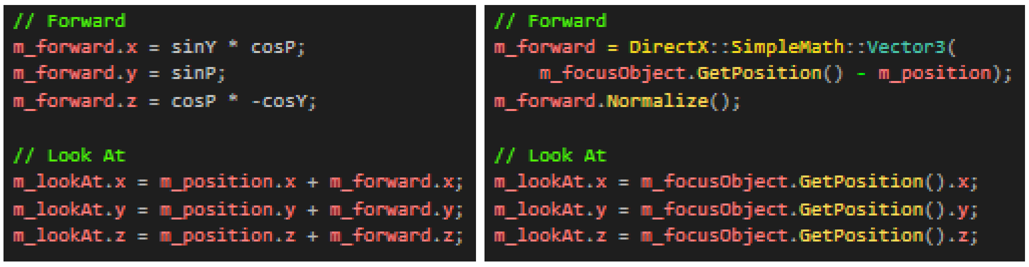


```
// Alter camera rotation
m_yaw += (cursorPos.x - centreX) * m_track;
m_pitch += (centreY - cursorPos.y) * m_track;
```

Figure 3-1 Camera mouse tracking

- **Update**

  Directly after handling user input, the update function of the camera is called from the primary tool class. This function instantly converts the yaw, pitch and roll values of the camera into radians for further calculations. If the camera is not focussing on an object, the forward and look at vectors are calculated as normal, shown in Figure 3-2. Otherwise, the forward vector is set to the normalised distance from the camera to the object to ensure a suitable distance is maintained. The look at is also defined as the objects position, to 'focus' the camera always directly on the object, this is the arc-ball implementation of the project. The remainder of this function calculates the up and right vector to keep the camera appropriately aligned.



```
// Forward
m_forward.x = sinY * cosP;
m_forward.y = sinP;
m_forward.z = cosP * -cosY;

// Look At
m_lookAt.x = m_position.x + m_forward.x;
m_lookAt.y = m_position.y + m_forward.y;
m_lookAt.z = m_position.z + m_forward.z;
```

```
// Forward
m_forward = DirectX::SimpleMath::Vector3(
    m_focusObject.GetPosition() - m_position);
m_forward.Normalize();

// Look At
m_lookAt.x = m_focusObject.GetPosition().x;
m_lookAt.y = m_focusObject.GetPosition().y;
m_lookAt.z = m_focusObject.GetPosition().z;
```

Figure 3-2 Camera focus vs not focussed

3.2.3. Benefit to Editor

The implemented camera system provides users an easy route to exploring the game world, at little computational cost. While allowing the user to traverse freely or around a specific object, the usability of the editor is improved.

3.3. Object Highlighting

3.3.1. Design

To indicate the user has selected an object, the highlighting feature has been implemented. Taking inspiration from both Blender (Roosendaal, 2020) and Unreal Engine 4 (Epic Games, 2014), the highlighting of objects appropriately displays the local transform and bounding boxes of selected objects. The local axis widget was implemented as an addition to the bounding box highlight as it is expected users may require local transform information of objects.

3.3.2. Execution

Original intentions for object highlighting were directed towards applying a semi-transparent tint. However, due to complications with 3D model/texture importing, this feature has been recorded for future work. Therefore, the current approach for object highlighting was implemented to suitably notify the user of their current selection. As the highlighting is specific to rendering, its functionality is localised within the game class:

- **Highlight**
  Selected objects are looped through and appropriate lines are drawn to represent both their bounding boxes and local axes (Glampert, 2017).

- **Axes**
  For each selected object, vectors are setup each of a size ~three units along their given axis and positioned at the origin of the object. These vectors are then used in the primitive batch draw line

function. The X, Y, Z vectors are signified by R, G, B colours when drawn, respectively.

3.3.3. Benefit to Editor

This feature is common in other world editors (Krogsæter, 2009, pp. 154) and is rather simple when compared to others. However, it can be viewed as necessary when informing the user of selected objects.

3.4. Editor Modes and Constraints

Terrain editor modes are applied through the designated dialogues as they require less input to operate. Whereas the object editor modes are handled via the tool system class, as they require more input information to determine which functionality to apply.

3.4.1. Design

3.4.2. Execution

3.4.3. Benefit to Editor

3.5. Actions

3.5.1. Design

In order to increase the usability aspect of the editor, multiple actions have been implemented and are available through toolbar/menu options. To ensure users are already familiarised with the provided actions, their key mappings replicate that of common computer applications.

3.5.2. Execution

The actions are split across two manager classes, as they operate on either objects or the scene itself. Descriptions of each action and their relevant manager functionality follows:

- **Undo – Redo**

Whenever the scene graph or display chunk is updated throughout the entire application, the current state of both is stored in the Scene Manager. To store the scene graph, the class updates adds it to a local vector, containing all previous states. For the display chunk, terrain geometry proved difficult to implement storage in the same manner as the scene graph. Therefore, the display chunk itself is stored and saves all current geometry positions to external CSV files. This approach has also been implemented to allow the potential of loading in previous states of terrain geometry. Every time the states are stored, the history index is increased. This index is used whenever the undo or redo functions are called, reducing or increasing the index and returning the appropriate stored state.

Previous implementations of this feature saw the undo/redo functionality of the scene graph and display chunk kept separate. However, this forced either the application or the user to individually undo/redo each state. This caused undesirable results as the history indexes of both states were not aligned. Thus, the combined approach was implemented.

- **Save**

  As an extension on the original provided feature, the Scene Manager save function begins by updating the game scene graph to match its display list. This is to ensure any alterations that may have been made to the display list rather than the scene graph will be saved. Furthermore, the game display chunk is saved, and the entire scene graph is rebuilt from the database. This is processed by calling the query and save functions of the SQL Manager class, as described in the pertinent section. A message box informs the user of a successful save or not and is accessible via the file menu. When saving the display chunk, all current paint values are saved to CSV files outside the database to be loaded and reapplied when the application is restarted.

- **Quick Save**
  This is a direct copy of the previous feature, although without the message box to prevent halting the application. The quick save feature is only accessed when the autosave timer triggers it, as described in the next listed feature.

- **Autosave**
  The Scene Manager contains a timer, enabled by the user when selecting the Autosave option via the file menu. The primary tool class is constantly updating the Scene Manager to check if its autosave is activated. Once active, the timer is initialised to count thirty seconds. Incrementing per frame, the timer displays a countdown on-screen when there's only ten seconds left. Reaching zero triggers the quick save function mentioned above and resets the timer. This process continues until the user disables autosave.

- **Save As – Load**
  With intentions of allowing the user to save and load multiple worlds, this feature is accessed through the file menu. Once selected, the user is prompted to input a name for the current save or to select a previously made one for loading. Currently, these features appropriately save and load multiple chunks from the database (via the SQL Manager). After some reconsideration, it was decided more functionality would be needed to appropriately load the new chunk data. This includes an updated heightmap file/path, and additional folders for storing the textures and positions of geometry, if not already defined by the [new] heightmap. To load an entire world with objects, another database table would be required to store object data. To respect the constraints of the project, these features have been recorded as future work.

- **Delete**

As this feature is focussed on deleting objects, its functionality is contained within the Object Manager class. The delete function is called from various classes throughout the application, including the primary tool class and object/light dialogues. When called, the function uses the selected object IDs to call the SQL Manager remove function. Afterwards, both the IDs container and current scene graph are cleared before rebuilding from the database.

- **Cut, Copy & Paste**

  These features are commonly used together and thus, have been grouped together for explanation. They all operate via the Object Manager class, make use of the same storage container of objects and are called throughout the application.

  The cut function retrieves the scene graph to loop through as well as the selected object IDs. When the selected object is found via their matching ID value, a temporary object is created. This object copies all details from the current scene graph object and is then added to storage. Once the scene graph size is reached, the selected objects are removed from the database via the SQL Manager.

  The copy function utilises the same process as the cut function, although doesn’t delete the selected objects after storing. Another addition to this function is when creating the temporary object, all available IDs are retrieved and the first is assigned to the ID value of the object. This is to ensure the ID column of the objects table does not skip any numbers. In addition, the X and Z positions of the copied are offset by five units to avoid pasting the object directly on top of the original.

  The paste function is much simpler than the previous two. After fetching the scene graph and stored objects, each one is added to the database via the SQL Manager. The scene graph is then updated to include the new object and rebuilt.

3.5.3. Benefit to Editor

Many software applications such as Blender (Roosendaal, 2020) and Unreal Engine 4 (Epic Games, 2014) use these features. Though they may seem trivial when using in day-to-day life, the defined actions are a central component of efficient editing capabilities.

3.6. Inspectors

3.6.1. Design

To achieve a user-friendly environment throughout the world editor, each main feature category operates a unique dialogue. Throughout development, these dialogues have undergone constant iterations to comply with common UX design examples. Inspiration has been sourced from applications such as Blender (Roosendaal, 2020), Unreal Engine 4 (Epic Games, 2014) and Adobe Photoshop (Adobe Inc., 2020). Various aspects of UX design can be found throughout these applications, ranging from toolbar buttons to entire windows.

A core principle for UX design is clean, clear representations of data and efficient functionalities (Gothelf, 2013, pp. 8-9). The world editor UX has been designed to focus on this rule by displaying all relevant data to the user in an easy-to-read format.

3.6.2. Execution

The dialogues have been implemented to operate via pointers to the entire tool system. The primary MFC class updates each dialogue relevant to their activation, optimising their performance by allowing localised calls to tool functions. An additional method was implemented to utilise the IDCANCEL operation, allowing users to exit the dialogue by pressing the close button instead of OK. This method avoids errors when attempting to reopen an exited dialogue as dialogues are hidden and their values reset instead of destroyed. On initialisation, all dialogues store a local pointer to the tool system for use throughout their operations. Whilst updating, each dialogue computes their own checks to handle unique tasks. The following list describes each inspector in detail and their manager classes:

- **Object and Light**

  These dialogues are separate instances, operating on different object types. Although as much of their functionality is similar, they have been grouped together for explanation.

  Upon creation, the dialogues setup their local pointers to their designated objects by retrieving the scene graph from the primary tool class. The dialogue entries are then prepared to include all relevant data, where applicable (i.e. object IDs and types).

  The first check computed by these dialogues is to determine if the user has selected an object by mouse picking or by selecting an entry from their ID lists. When either is true, the other class' selected IDs container is updated to match the current selection. If the user has selected an object through mouse picking, the dialogue entries are updated to display the traits of the selected object. It is worth noting if there is more than one object selected, the dialogue entries are not updated. The update function also determines if the focus checkbox has been marked, sending an index value through the tool system to the camera object for arc-ball motion. Finally, the function ensures the tool system editor mode and constraint are up to date, matching the dialogue selection.

  Both dialogues display a count for the current amount of object/lights in the scene and numerous buttons, each applying the appropriate editor mode and constraint to the tool system. In addition to these, delete and duplicate buttons are also present. When selected, the dialogues delete or copy/paste selected objects via the Object Manager, respectively.

  When an object type is changed by selecting an entry from the appropriate combo box, the dialogue updates their selected object pointer and calls the Object Manager replace function. In this operation, the number of active lights is first retrieved to check if it has reached the maximum allowed (ten lights). This maximum has been defined to comply with the integrated HLSL shader capabilities. Continuing, the selected object is removed from the database via the SQL Manager. The relevant object data is then updated depending on the replace function, a new object is created and re-added to the database before rebuilding the scene graph.

A local focus dialogue resides within both classes to handle the user trying to focus on more than one object. If the checkbox is marked while there are multiple selections, the focus dialogue is created to prompt the user to select one object. Once selected, the camera is instantly updated to focus on the object. Before closing the dialogue, the user can switch between objects to view each in turn.

- **Spawn**

  Presenting the user with a choice of fourteen unique objects, this dialogue makes use of mouse picking to determine where objects should be placed. During its update, the tool system spawn type is kept up to date, relevant to whichever button the user has applied. As development reached the stage of importing models to the editor, some difficulty was encountered when applying textures. It was decided the editor would only feature generic colours for its objects, acting as white box models for the user. This is considered an appropriate method as it is expected users would be working closely with an art team, providing them with WoFFC assets. A list of all objects can be found in Table 3-1 below:

Residential		
House #1	House #2	Cave
Nature		
Grass	Palm Tree	Pine Tree
Props		
Bridge	Fence	Boat
Shapes		
Cube	Cylinder	Cone

Miscellaneous	
Light	Water

Table 3-1 Object variations

- **Terrain**

  This dialogue has been implemented to act as a terrain information panel with additional features. During the update function, the state of the right mouse button is checked before determining if a sculpting mode is selected. If false, terrain is picked via the Mouse Manager class. Upon a successful intersection, the dialogue details are updated to display the appropriate values – including a coordinate system to allow accurate placement and manipulation. Otherwise, if the user has applied a sculpt mode the selected geometry position is updated according to the mode and constraint, via the Terrain Manager. The sculpt modes in this dialogue only permit the user to sculpt the pre-selected geometry. The dialogue also features texture changing, allowing the user to update the paint applied to the selected geometry. Included are row and column dropdown boxes, permitting the user to either select or input an existing value. Once geometry is selected, the user can alter the texture by selecting one from the dropdown menu. This calls the *overwrite* paint function in the display chunk class, explained in the following bullet point. It is worth noting, all terrain operations outside the display chunk class utilise a global struct. This struct contains terrain information such as row, column, ID, position and an intersection controller. Moreover, a paint enumeration value is stored to determine which paint is applied to a piece of geometry.

- **Paint**

  A total of six paints have been implemented within the editor, including an invisible type for the user to define unrenderable terrain. For the remaining five paints, standard terrain textures have been applied as seen in Table 3-2. The user has the option to apply blending between textures by checking the relevant dialogue button, informing the display chunk of the selection. Whilst updating, this dialogue determines if the right mouse button is being pressed. If so, the selected texture is sent to the Terrain Manager class for painting.

Within the Terrain Manager paint function, the selected terrain is determined via the Mouse Manager class. For each successful intersection, the display chunk paint function is called with the selected terrain and paint type. The display chunk provides its own paint function to keep its geometry array localised.

The display chunk stores containers for each paint type and all possible blends. The containers hold geometry rows and columns used to define which texture to apply when rendering. During the paint function, all containers are checked for duplicates and removed. This is to ensure only one pair of rows and columns resides in one container at a time. Following is a method which switches between the currently applied paint and the selected paint. Depending on whether the user has declared blending, the selected terrain row and column is added to the appropriate container. If the current paint is opaque, the user can blend another texture into it. However, if the current paint is already blended, the selected paint simply overwrites to apply an opaque paint. Original intentions for this technique saw the blending of more than one texture. Yet, due to the five available paints and high number of combinations, this was recorded as future work to keep within the scope of the project. For the same reason, another intended method of painting which involved blending surrounding paints was discarded. The functionality of this was successfully implemented for the first three paints, before the addition of the remaining two. Therefore, the code has been left in the editor for reference inside the display chunk Check Surroundings function.

Type	Texture

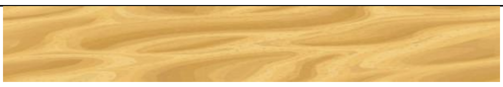
Grass	
Dirt	
Sand	
Stone	
Snow	

Table 3-2 Texture types

- **Sculpt**

  The available sculpt modes include increasing, flattening and decreasing geometry. The user can easily switch between modes by pressing the dialogue buttons, as well as adding a constraint to the function. It is recommended to only use the Y constraint when sculpting, although X and Z constraints have been included for completion. While sculpting, terrain geometry is altered based on a scale factor. This can be changed by inputting the desired value in the dialogue input box. Like the paint dialogue, the state of the right mouse button is checked when updating. If the button is pressed, the Terrain Manager sculpt function is accessed with the defined sculpt mode and constraint.

  The Terrain Manager sculpt function first establishes whether a piece of terrain has been specified or not. This is to allow the terrain dialogue sculpt method of sculpting individual geometry. If not specified, the function fetches picked terrain from the Mouse Manager. The terrain information is then sent to the display chunk sculpt function. In addition, if the selected sculpt mode is flatten, the selected terrain position is stored one the first mouse click. This is used to determine which point the rest of the selected geometry positions should be increased/decreased to. Again, the display chunk provides its own sculpt function to keep its array localised.

If the selected sculpt mode is either increase or decrease, the selected geometry positions of the entire quad are updated based on the defined scale factor, while respecting the selected constraint. When the flatten sculpt mode is applied, each selected geometry is checked for either a higher or lower position than the previously stored. Complying with the selected constraint and until the stored position is matched, the geometry positions are increased or decreased, respectively. Notably, geometry is restricted to prevent a Y position of below zero.

An additional sculpt technique was considered, although was recorded for future work. This method utilises the mouse drag feature. Much like object manipulation, the user can select a piece of terrain and drag the mouse in the direction they wish to sculpt. This feature is implemented within the editor, although unavailable to the user. The operation is successful but needs polishing before being added to the final editor. The aim of this method was to allow the user, for example, to increase terrain on the Y axis and then select a point down the extruded terrain to drag outwards. This was deemed impossible with the current geometry setup as the terrain would need additional quads created once extruded. To explain the intentions further, Figure 3-3 shows the planning of this feature:

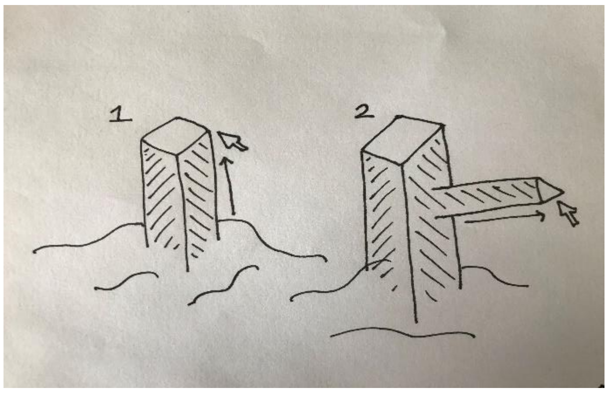


Figure 3-3 Sculpt drag concept

3.6.3. Benefit to Editor

3.7. Other Managers

3.7.1. Design

In terms of design and benefits, this section regards all implemented manager classes throughout the application.

3.7.2. Execution

whilst expanding on two executions that haven't been highlighted previously.

- SQL Manager
- Shader Manager
  - *Add feature to allow switching between texture & toon shader?!*

3.7.3. Benefit to Editor

Enhancement	Feature	Operation
Usability	Mouse picking	Select single & multiple objects, deselection
Usability	Mouse drag	Manipulates object transforms
Usability	Camera	Arc-ball motion, focus on selected objects
Usability	Object highlighting	Local axis, bounding box
Usability	Actions	Undo, Redo, Delete, Save, Save As, Load, Cut, Copy, Paste
World Editing	Object inspector	Allows object selecting, focussing, type changing, value editing, deleting, duplicating, transforming and constraining
World Editing	Light inspector	Allows light selecting, enabling, focussing, type changing, value

		editing, deleting, translating and constraining
World Editing	Spawn inspector	Allows type changing, object type selection and placement
World Editing	Terrain inspector	Allows terrain selection by inputting row/column or mouse picking, texture changing, sculpt selecting, constraint selecting, scale factor and displays coordinates

For each feature / system you have added to the tool:

- *Full explanation of what it is, does and designed to do. With reference to why this would be useful in the context of content create for the WOFFC game. Refer to any user experience goals or design intended to be UX friendly or reference systems in other tools (unreal, photoshop etc) that you used as a reference point*
- *Technical discussion how you implemented the feature. HOW you did it. Algorithms, coding structure and technical discussion into what you created and the techniques that you used to achieve them.*

Chapter 4 Conclusion

Conclusion: provide reflection and critical analysis of your work. What went right, what went wrong and why? What would you do differently next time?

Chapter 5 References

Adobe Inc. (2020) *Adobe Photoshop*. Adobe.

Falconi, R. F. (2010) ‘Usability and game design: Improving the MITAR Game Editor’. Massachusetts Institute of Technology.

Glampert (2017) *DebugDraw*. Available at: https://github.com/glampert/debug-draw.

Gothelf, J. (2013) *Lean UX: Applying lean principles to improve user experience*. O’Reilly Media, Inc.

Krogsæter, T. G. (2009) ‘World of Wisdom-World Editor: User-interface for creating game worlds for World of Wisdom’. Institutt for datateknikk og informasjonsvitenskap.

Roosendaal, T. (2020) *Blender*. Blender Foundation. Available at: https://www.blender.org/ (Accessed: 8 May 2020).

Sweeney, T. (2014) *Unreal Engine*. Epic Games. Available at: https://www.unrealengine.com/en-US/get-now (Accessed: 8 May 2020).