|  |
| --- |
| World of Flim-Flam Craft Editor  CMP405: Tools Programming  Dylan James Black |

**Contents**

[Chapter 1 Summary 2](#_Toc39874924)

[Chapter 2 Controls 3](#_Toc39874925)

[2.1. Camera 3](#_Toc39874926)

[2.2. Global Key Mappings 3](#_Toc39874927)

[2.3. Dialogue Specific 3](#_Toc39874928)

[Chapter 3 Features 5](#_Toc39874929)

[3.1. Mouse Use 5](#_Toc39874930)

[3.1.1 Design 5](#_Toc39874931)

[3.1.2. Execution 5](#_Toc39874932)

[3.1.3. Benefit to Editor 7](#_Toc39874933)

[3.2. Camera 7](#_Toc39874934)

[3.2.1. Design 7](#_Toc39874935)

[3.2.2. Execution 7](#_Toc39874936)

[3.2.3. Benefit to Editor 8](#_Toc39874937)

[3.3. Object Highlighting 9](#_Toc39874938)

[3.3.1. Design 9](#_Toc39874939)

[3.3.2. Execution 9](#_Toc39874940)

[3.3.3. Benefit to Editor 10](#_Toc39874941)

[3.4. Actions 10](#_Toc39874942)

[3.4.1. Design 10](#_Toc39874943)

[3.4.2. Execution 10](#_Toc39874944)

[3.4.3. Benefit to Editor 13](#_Toc39874945)

[3.5. Inspectors 13](#_Toc39874946)

[3.5.1. Object 13](#_Toc39874947)

[3.5.2. Light 13](#_Toc39874948)

[3.5.3. Spawn 13](#_Toc39874949)

[3.5.4. Terrain 13](#_Toc39874950)

[3.5.5. Paint 14](#_Toc39874951)

[3.5.6. Sculpt 14](#_Toc39874952)

[3.6. Other Notable Features 14](#_Toc39874953)

[3.6.1. SQL Manager 14](#_Toc39874954)

[3.6.2. Tool Main enums!?!?! 14](#_Toc39874955)

[Chapter 4 Conclusion 16](#_Toc39874956)

[Chapter 5 References 17](#_Toc39874957)

# 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

# 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

# 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 transformations.

### 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.



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.

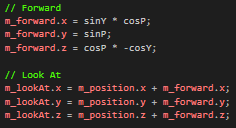
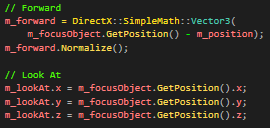
 

Figure 3‑2 Camera focus vs not focussed

### 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. Actions

### 3.4.1. 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.

### 3.4.2. 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 below. A message box informs the user of a successful save or not and is accessible via the file menu.

* **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.

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.

### 3.4.3. 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.

## 3.5. Inspectors

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

### 3.5.1. Object

### 3.5.2. Light

### 3.5.3. Spawn

### 3.5.4. Terrain

Note, all terrain functionality outside of the display chunk class operates using a struct containing…

### 3.5.5. Paint

### 3.5.6. Sculpt

## 3.6. Other Notable Features

### 3.6.1. SQL Manager

### 3.6.2. Tool Main enums!?!?!

|  |  |  |
| --- | --- | --- |
| **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.*

# 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?*

# References

*References for any code or techniques incorporated in your application. Cases of plagiarism will be taken very seriously.*