CMP405: Tools Programming Report

Overview

Several improvements were made to the tool provided for this coursework which are as follows:

- A camera class was created to contain all the associated functionality including rotation using the mouse cursor and the ability to focus on the currently selected object.
- Mouse picking was added to allow the user to select objects in the scene using the mouse, rather than use the dialogue box provided by default. The dialogue box used for object selection was also modified to feature the object ID as well as its assigned name.
- An inspector dialogue box was created to allow the user to view and modify information regarding the currently selected object's position, rotation, and scale as well as the name of the object. This dialogue, and the functionality associated with it, is toggleable through a menu entry or a button in the toolbar.
- Objects can now also be created, duplicated, and removed from the scene using keyboard shortcuts or options in the menu.
- Mouse picking was also adapted to allow for terrain manipulation using the mouse using a ray-triangle intersection algorithm to detect if a ray cast from the mouse pointer intersects any faces on the terrain.
- The ability to raise, lower, and flatten the terrain was added, in addition to an option to smooth the terrain. The customisation of this functionality was enabled using a dialogue box which can be toggled from either a button on the toolbar or an entry in the menu system.
- Undo and redo capabilities were implemented to allow the user to revert changes they make in the scene or redo actions they reverted. This functionality includes changes to the objects in the scene in addition to changes made to the terrain and can be accessed using keyboard shortcuts or buttons in the toolbar.

User Guide

Keyboard Shortcuts

Ctrl + Z: Undo

Ctrl + Y: Redo

Ctrl + A: Create new object

Ctrl + D: Duplicate selected object

Ctrl + X: Remove selected object

Ctrl + F: Focus the camera on the selected object

Ctrl + S: Save the scene

Detailed User Guide

The following features can be accessed from the menu tab titled “Edit”:

- **Object Selection:** A dialogue box containing a selectable list of all objects in the scene can be accessed to allow the user to select an object from anywhere, even if it is not currently visible to the camera. This functionality was modified to show the object name as well as the ID.
- **Create a new object**: A new object can be created, which will be spawned at (0, 0, 0) and will automatically be selected (viewable from the inspector). This object can then be modified using the inspector to fit the desired needs. This is also achievable using the keyboard shortcut **“Ctrl + A”**.
- **Duplicate an existing object**: The currently selected object can be duplicated using the menu or with the keyboard shortcut **“Ctrl + D”**. The new object will be spawned 5 units above the original on the y-axis and will be automatically selected (viewable from the inspector).
- **Delete an object**: The currently selected object can be removed from the scene using the menu option or using the keyboard shortcut **“Ctrl + X”**.

The following features can be accessed from the “Window” menu tab:

- **Inspector Window:** The inspector window provides information regarding the currently selected object. It also allows the user to **rename/move/rotate/scale** the object by typing the desired number into the corresponding text box and clicking the “Update Game Object” button to apply the changes. *This window is also toggleable from a toolbar button.*
- **Terrain Edit Window:** The terrain edit window allows the user to customise the current terrain edit brush using the dropdowns and the text boxes provided in the window. This allows for **raising, lowering, flattening,** and **smoothing** the terrain with a square brush with a customisable width. *This window is also toggleable from a toolbar button.*

Additional features include:

- **Camera:** A moveable camera using **WASD**, and rotatable by pressing the right mouse button, and moving the cursor to the edges of the screen. The **“E”** and **“Q”** keys can also be used to move the camera up and down. Additionally, **“Ctrl + F”** can be used to focus the camera on the selected object.

- **Mouse Picking:** Objects in the scene can be selected by clicking on them when not in terrain edit mode (i.e. the terrain editor window is toggled on). When terrain edit mode is active, clicking on a point on the terrain will manipulate it according to the current brush settings.
- **Undo/Redo:** Changes made to the scene including both object and terrain manipulation can be undone/redone using either the toolbar buttons or the keyboard shortcuts **"Ctrl + Z"** and **"Ctrl + Y"** respectively. *Up to a maximum of 10 changes can be reverted.*
- **Saving:** A minor modification to the save functionality provided in the default framework was made to allow both terrain and objects to be saved with the same button, rather than two individual methods. This can be achieved using the option in the "File" menu tab, the toolbar button, or using the keyboard shortcut **"Ctrl + S"**.

Feature List

Camera Class

A camera class was implemented to contain all the functionality regarding the camera such as movement, rotation, and the ability to focus on the currently selected object. This class was designed to allow for the creation of a self-contained camera object, which could prove useful in the event of additional camera functionality being desired (e.g. an arcball camera) or for the creation of a game with multiple cameras, however as it stands, it functions primarily as a tool for navigating the scene within the editor. The camera movement is as standard, making use of the Input Commands header file to receive information from the MFCMain class regarding keypresses and the mouse position. This information informs the camera when to move or rotate and updates the view matrix accordingly. There was no need for further modifications to this process with the current functionality, beyond adding in a move up and down flag. Camera rotation is adapted from the code found on the editor's wiki and allows the user to use the mouse for this. The mouse position on the screen is tracked in MFCMain and sent to the camera class via Input Commands, and when the position of the mouse is within a certain distance of the edges of the window, rotation occurs. While a fully first-person camera would be more appropriate, due to problems regarding diagonal rotations resulting in a wavy effect in the camera's rotation, the solution devised provides adequate functionality with the intent of creating a scene within the editor.

The camera focus function was created for the purposes of locating the currently selected object in the scene. The bulk of this feature is contained within a single function which receives a world space coordinate and moves the camera adjacent to this, with a higher elevation and a view of the object itself. The distance from the object that the camera is moved to, including the elevation value, is determined by the scale of the object to ensure the camera is not clipped into larger objects when focusing on them. A minor improvement to this would involve taking into account the size of the object's model, to prevent clipping in larger objects that have not been scaled up or down within the editor, as well as an arcball feature to rotate around the object after moving. However, this feature allows the user to move to the currently selected object in the scene, which becomes particularly useful when moving the object large distances away, or when trying to locate an unseen object using the select dialogue box.

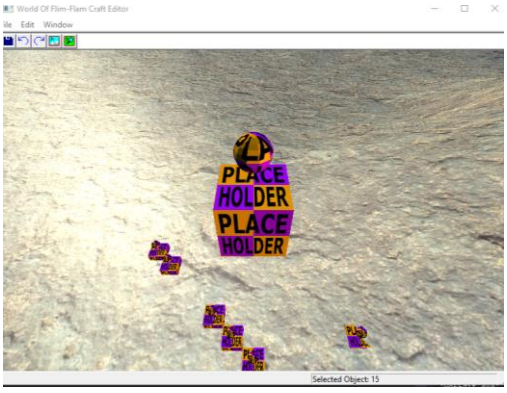

Figure 1: Shows the camera focusing on a selected object

Mouse Picking

Basic mouse picking was implemented as discussed as part of the lab, which allows the user to click on an object in the scene to select it. The primary benefit to this is usability of the tool, as the dialogue box method is no longer required for object selection, despite its usefulness in some situations. Overall, this method of object selection significantly streamlined the process of object manipulation as only the inspector window is required. This process occurs by casting a ray from the mouse's screen space position into the scene and detects all objects it collides with. This was improved to detect the distances between each of the objects the ray hits to determine the closest one to the origin of the ray, this ensures the first object is selected, rather than an object behind the one being clicked on as without this improvement the selection between the objects intersected by the ray would be somewhat random.

In addition to the standard mouse picking for object selection, a modification was made for terrain manipulation. To carry this out, an if statement is used to check if the user is in terrain edit mode before performing the picking process. If the terrain edit mode is active, then a slightly different raycast is carried out which uses an identity matrix as the world matrix, rather than one formed from the displayed objects transforms. After translating the mouse screen space position to world space and calculating the direction of the raycast, the information is passed into a function in the display chunk class where a ray-triangle intersection test is carried out to determine where on the terrain the user is clicking on.

The ray-triangle intersection algorithm itself was found online and used when looping through each vertex in the terrain in the DisplayChunk class to determine if the ray intersects the triangle. If an intersection occurs, the function returns true and the triangle's vertices are added to a standard library vector. After all vertices have been checked, the one closest to the camera is selected, and based on the terrain edit mode and brush type, the appropriate manipulation is carried out on this vertex.

Object Manipulation & The Inspector Window

The ability to add and remove objects from a scene is one of the most basic requirements for a game-making tool, as without this functionality, there is no way for a game scene to be made using the tool. To achieve this, two functions were made for each of the processes. The instantiate function is used to create an empty scene object with default parameters. A new ID is calculated for the object by finding the largest ID currently in the scene and incrementing it, the object is then added to the scene graph in ToolMain and subsequently added to the display object list in Game for rendering. Additionally, the currently selected object ID is set to that of the new object to allow the user to manipulate the newly created object as desired. The remove function is much simpler and just removes the currently selected object from the scene graph and updates the display list accordingly. This functionality is extremely foundational to a game-making tool, which is why it was implemented in the first place, however as an additional feature the ability to duplicate a selected object was also added. The corresponding function for this operated similarly to the instantiate function in many ways, the only difference being the data being added to each of the new object's parameters was that of the existing object being selected. Of course, to prevent issues regarding object selection etc. the duplicate object requires a unique ID, and so the same process that takes place when instantiating an object occurs to give the duplicate a new ID before adding it to the scene graph. In order for this functionality to be easily accessible to the user, two methods were added to perform these actions; an option in the "Edit" menu as well as a keyboard shortcut. In many applications, for example Blender, Unity, even Microsoft Word, keyboard shortcuts are a

fundamental part of making a program easy to use. It is for this reason that the keyboard shortcuts were implemented alongside the menu options, for those that prefer/require an alternative approach.

The creation and deletion of objects alone is not enough to make a game in a given tool, to achieve this much more functionality is required. To remedy this, the inspector window was developed which allows an object's transform to be manipulated. Additionally, it allows the name of the object to be changed which proves to be more user-friendly than just referring to each object by its ID number, particularly in the selection dialogue box and when there are larger numbers of objects in the scene. The largest inspiration for this feature came from the Unity game engine, as the inspector window in that application is an extremely useful tool which makes the creation of game scenes much easier than if it were not present. The inspector window in this application consists of a dialogue box with an edit control box (an input text box) for each of the modifiable parameters (e.g. the (x, y, z) value for the position, rotation, and scale, as well as the name of the object), in addition to a button for updating the game object with the values entered. Each of the (x, y, z) components are represented by their own edit control box to make it easier for the user to manipulate specific axis etc. as in most applications (such as Unity, Blender, etc.). Unlike Unity however, a button for updating the game object with the changes was added. This was primarily for efficiency's sake, to prevent the inspector and scene updating with each change, however, this would be very easy to implement if the button was not desired, as there is a On_EN_CHANGE message that can track when the contents of an edit control is changed, and could be used to call the necessary update functions.

The inspector window itself is represented in code by its own class derived from the standard dialogue box class as can be seen in the selection dialogue box class provided with the framework. This allows the dialogue box to track its own messages and handle the necessary functionality such as extracting numerical values from the text boxes, and updating them where appropriate with the current values of the selected object, in addition to updating the scene graph in toolMain with the new values for the object. Initially, the desire was for the inspector window to be docked at the side of the main window as can be seen in Unity, however this produced far too many problems during implementation. The idea was to use a CDialogBar to represent the inspector, which is naturally dockable at the side of the main window, however there appeared to be no way to extract the data from the edit controls this way meaning the functionality would be lost at the expense of aesthetics, and so the feature was cut due to time constraints.

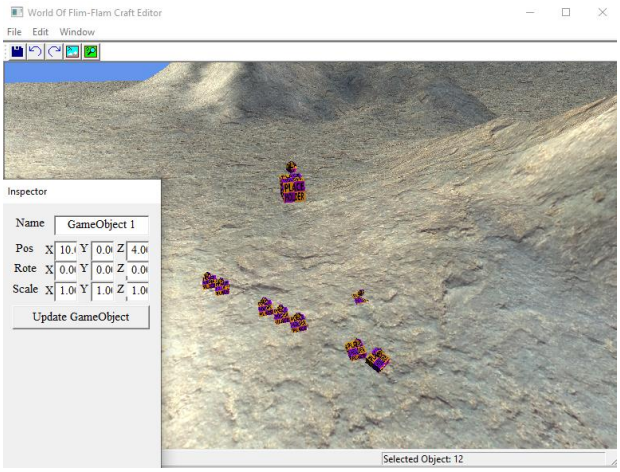

Figure 2: Shows an object in the scene being renamed and repositioned using the inspector

Terrain Manipulation & The Terrain Editor

Another feature that is extremely desirable for a game-making tool, particularly when an exterior scene is being created, is the ability to manipulate a terrain. Many game engines and game-development tools such as Unity make use of terrain editing/sculpting, so an implementation of this would prove very useful for the creation of outdoor levels etc. For this tool, the terrain editing process is comprised of two components: the ray-triangle intersection algorithm for determining where on the terrain is being clicked by the mouse pointer, and the terrain editor dialogue box.

The goal of this feature is to allow the sculpting of the terrain in an easy to use manner, that can be saved like the default terrain provided. The structure used for the controlling of this functionality was primarily inspired by Unity's terrain editor where a component in the inspector window allows for brush customisation and a choice of edit modes. To achieve this, 4 different techniques for terrain manipulation were implemented: raise, lower, flatten, and smooth. These techniques are present in Unity's terrain tools as with many others, and so their implementation here was to create a solid foundation from which more advanced terrain editing functionality could be explored.

The majority of the functionality for this feature is contained within the DisplayChunk class which was provided as standard with the initial framework. This class initially contained all the relevant functions for saving and loading a heightmap, as well as translating it into a piece of renderable geometry. This provides a significant head start in regards to manipulating the terrain at will, as additional functionality can be somewhat extrapolated from the existing codebase.

As with Unity's terrain tools, a "brush" was created for editing the terrain, this took the form of a square shape around the selected vertex, the width of which is editable from the terrain editor dialogue box. Other brush types were investigated such as a circle brush to produce a more realistic result, however, the implementation of a brush of this shape would increase the complexity significantly, and since the square brush could be modified to produce an equally adequate result in conjunction with the smoothing process, it was deemed unnecessary to spend further resources in this area. Instead, a dropdown was created in the editor dialogue to allow for additional brushes to be added with ease if desired. To improve the realism of the square brush when raising or lowering the terrain, weights were applied to each index being affected by the brush. This allowed vertices closer to the centre of the square to be moved more than those around the perimeter. This produces a nice "mound" effect when used for smaller height increments, and when combined with smoothing at larger height differences, can produce a more than adequate result. The editor dialogue box also allows the user to manually input a height value, which represents how much the terrain will be raised/lowered or where it will be flattened to. This, along with the ability to choose the brush width, allows for a great deal of sculpting options using the 4 core techniques provided.

The 3 manipulation techniques, excluding the smoothing algorithm, follow the same general process in their respective functions. First the bounds of the square brush are calculated using the brush width and the point selected, which are clamped to ensure they lie within an appropriate range for accessing the terrain geometry array. Then the terrain geometry array is looped through from the bottom left of the brush, to the upper right. For raising and lowering, the weights for the current index is calculated as can be seen in Figure 2 which results in the indices further from the brush's centre having lower weights than those closer. This process is not necessary for the flatten function as all vertices in the brush's influence are flattened to the same height.

```
int numberOfCuts = (currentBrush.width - 1) / 2;
float weight = 0;

if (i == SelectedVertX && j == SelectedVertY)
{
    weight = 1.0f;
}
else
{
    int step = Vector2::Distance(Vector2(SelectedVertX, SelectedVertY), Vector2(i, j));

    weight = 1.0f - ((0.9 / numberOfCuts) * step);
}
```

Figure 3: The calculation used to determine the weight of the current index of the square brush.

After the weights are calculated, the value is multiplied with the height value to determine how much to move the vertex, which is then added to its current height. A value of either 1 or -1 is also multiplied depending on whether the user is raising or lowering the terrain, or in the case of flattening, the height is simply set to the height value.

The smoothing algorithm takes the same approach regarding the calculation of the brush bounds. After which it loops through the brush indices and sets their height to the average of those around it. The results of which are a much smoother terrain with significantly fewer steep slopes.

Each of these processes manipulate the terrain geometry itself rather than the heightmap for easier implementation, so a function for updating the heightmap array values based on the geometry array was created by reverse-engineering the existing update terrain function.

The terrain editor is contained within its own class derived from the CDialogEx class. This allows the data to be extracted from the components of the dialogue box and sent down the chain to toolMain, to game, and finally to the display chunk class where the assigned brush and edit parameters can be utilised when clicking on the terrain in terrain edit mode. Due to the simplistic nature of the changes made within the terrain editor dialogue box (i.e. changing values behind the scenes), the necessary update functions are called whenever the text/selection is changed, unlike in the inspector window where the operations are slightly more computationally expensive due to the need to update elements of the scene in real-time. Additionally, the ability to toggle the terrain editor via toolbar button or menu option was used to allow both a graphical and textual method of toggling the editor which increases the usability of the tool for more people with differing preferences.

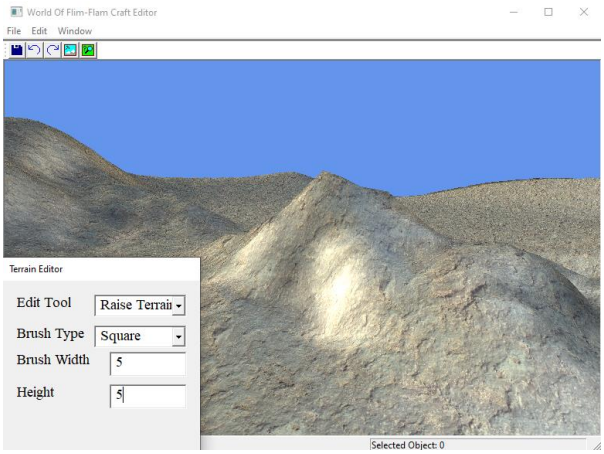

Figure 4: Example terrain segment sculpted using the editor

Undo/Redo

The ability to undo and redo actions are an extremely common feature in pretty much every application, it can be seen in many art programs such as Blender, game engines such as Unreal and Unity, as well as most Microsoft apps. For this reason, an undo/redo feature was essential for the creation of this tool. Within the scope of this project, there are ultimately two kinds of actions that can be carried out: terrain manipulation, or object manipulation. For it to be possible to undo/redo these actions, two data structures would have to be stored. The scene graph in toolMain is used to tell the renderer what objects are in the scene for displaying, and the heightmap in DisplayChunk is used to determine the heights of every vertex in the chunk. For the purposes of storing both of these data types, a struct was created, and a standard library vector was used for storing the backups. Every time a terrain or object manipulation action is performed, a function is called after its completion to store the current scene graph and heightmap in the backups vector. Up to a total of 10 backups can be stored to save on memory, and if an action is performed after undoing something, all later backups are removed as is the case in almost all other programs. An index integer is used to determine which "backup" is the current state of the scene, so whenever undo is called, the index decrements where possible, while whenever redo is called, it increments where possible. When the contents of the current index in the vector don't match up with the current state of the scene (i.e. undo or redo has been called) the scene graph and terrain heightmap are updated with the contents of the vector. This ability works as intended and can be accessed using either keyboard shortcuts or toolbar buttons. Both of which increase the usability of this feature.

Miscellaneous Improvements

The save button in the menu and toolbar was changed to allow both to save the terrain and the objects in the scene to improve the usability of the tool as most users wouldn't want to have to save two separate things in order for their scene to be saved.

The icon for the save button on the toolbar was replaced to more accurately represent the universal standard used in most applications. The icons for the undo and redo toolbar buttons also follow this rule, while the terrain editor and inspector toggles attempt to represent their functions graphically through their icons to improve readability.

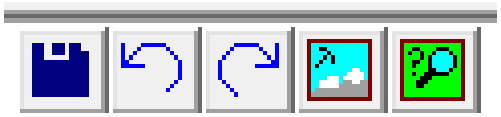

Figure 5: The toolbar present at the top of the main window

An extremely minor improvement to the selection window to improve its usability, which simply involved including both the ID and the object name in the list box. This required a modification to the way it selects the object as it was no longer a simple case of converting the string to an integer. However, very little needed to be changed and the overall functionality remains the same.

Conclusion

Overall, this project went rather well, as the initial intent for the application was, for the most part, successfully achieved. Due to unforeseen circumstances and the time constraints present in this project, the development of the application ended up being somewhat rushed, so if this task were to be undertaken again, it is expected that with more time available and better time management, a significant improvement to the resulting tool could have been made. Despite this fact however, almost everything that was attempted was successfully implemented.

Additional Improvements

In terms of object manipulation and the inspector window, several improvements could have been made to more closely imitate popular game engines such as Unity or Godot. For example, a hierarchy window could have been developed to allow a more visual representation of all the objects in the scene similar to the selection window. This could have been further developed to allow objects to be children of others, where the parent object's transforms affect the associated children. This proves to be a useful tool in game engines, particularly for moving/connected objects which is a commonly used feature in games. Additionally the ability to import and assign models to the objects in the scene via the inspector window would have been a significant visual improvement to the base functionality as it would allow for a greater variety of objects in the scene to make it feel more like a game. If gizmos were to be added when selecting an object in the scene the mouse picking algorithm could be adapted further than it was for this project as it would allow the user to manipulate an object's transform by clicking and dragging, which would be a much more streamlined process than the current method, as the usability proves to be somewhat clunky despite the functionality being present. However, these improvements would have required much more development time, and so, while the current tool lacks some of the smoothness and user-friendliness present in larger, more well-established game engines, the base functionality is there which could be easily built upon to produce significantly better results.

The terrain editor could also have been improved to allow for more brush types similar to Unity's terrain tools. However, for this it is likely that an entirely new method would have to be developed to replace the existing process, as a mask would most likely have to be created and translated onto the terrain for more complex brush shapes. Additionally, the ability to generate a random terrain could provide a good starting point for building an exterior level using the sculpting tools already implemented.

The above improvements were initially considered for inclusion in this project, however, were dropped to reduce the scope and focus more resources on the implementation of the current features. This proved to be a wise decision as the features that are currently implemented are reasonably well polished and work fully as intended.

Areas of Difficulty

There were difficulties attempting to implement the mouse controls for the terrain editor as the raycast seemed to be off by a large margin however, after researching more into the ray-triangle intersection algorithm and altering the existing mouse to world position process, it ended up working successfully as intended. This could still be improved further by making use of the calculated position within the intersected triangle to determine the closest vertex rather than adding all three to the vertex list to provide a more accurate result. A more accurate result could also be achieved by choosing a vertex based off the distance to the ray's destination rather than origin as when the camera is closer to the terrain surface, a more noticeable margin of error is present. Despite this, the

accuracy seems to be sufficient and very rarely is there any noticeable error so overall the terrain sculpting was successful. Ideally the inspector and terrain editor would have also been docked in the main window, however, this proved to be more difficult than initially anticipated as accessing the contents of the edit control boxes in the dialogue boxes was not as simple as it is when accessing them from a standard dialogue box. When researching this, a solution involving a “CPane” was found, however the results seemed questionable and with more time available, this would have been attempted. However, the current state of the tool, with these features as separate dialogue boxes doesn’t appear to be detrimental to the usability of the tool so this would be a minor improvement.

Summary

In conclusion, the resulting tool from this project is somewhat successful and meets the desired outcomes. The ability to manipulate objects within the scene as well as create and remove them provides a solid foundation of any game making tool. The terrain sculpting capabilities allows the user to thoroughly customise the appearance of the chunk, and with some further modifications could provide a high-quality external cell for a game.

References

Scratchapixel.com. 2020. Ray Tracing: Rendering A Triangle (Möller-Trumbore Algorithm). [online] Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection [Accessed 10 May 2020].

Akeinine-Moller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M. and Hillaire, S., 2020. Real-Time Rendering. 4th ed. [ebook] Available at: http://dl.booktolearn.com/ebooks2/computer/graphics/9781138627000_Real_Time_Rendering_4th_Edition_5726.pdf [Accessed 10 May 2020].