CMP405 Report

Feliks

Contents

Introduction

The application was extended in multiple ways. The most prominent user facing features are the property window, camera navigation changes, and renderer updates. Other features include safer saving, restructuring, and cleaning up of code, and general usability enhancements. This report will cover how these features were implemented and the design process behind them.

How to Use

To open the objects properties window, select ‘Window’ and click ‘Object Properties’. An object must be selected to view the properties. These properties can then be modified. To save changes, press the save button located in the main window or in the menu bar.

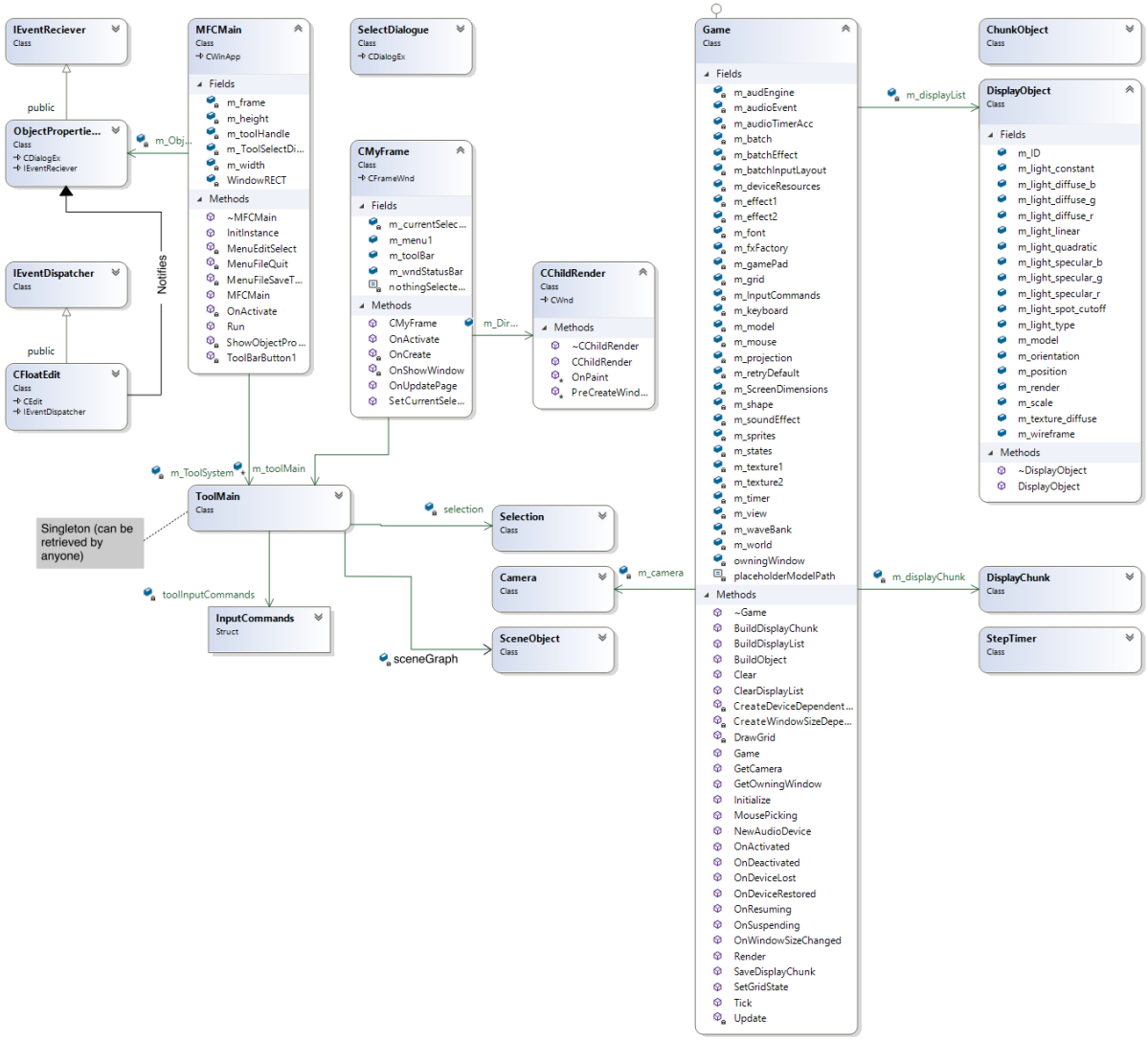
Mouse Look

Right click on the preview viewport, then move mouse around.

Keyboard

- W, A, S, D, R, F – Movement forward/left/back/right/up/down.
- Shift – Speed up movement.
- Q, E, Z, X – Rotation left/right/up/down

Final Class Diagram



Property Window

The property window was inspired by Unity's Property editor. This was deemed to be an appropriate choice as the Unity inspector window has key, identifying information such as the name and the transform first, and then works down to specialised elements with ways to collapse unwanted ones.

The remaining options were added to a property grid and sorted into groups with similar properties, and descriptions provided to make navigation easier. The dialog window was made to be responsive to size changes.

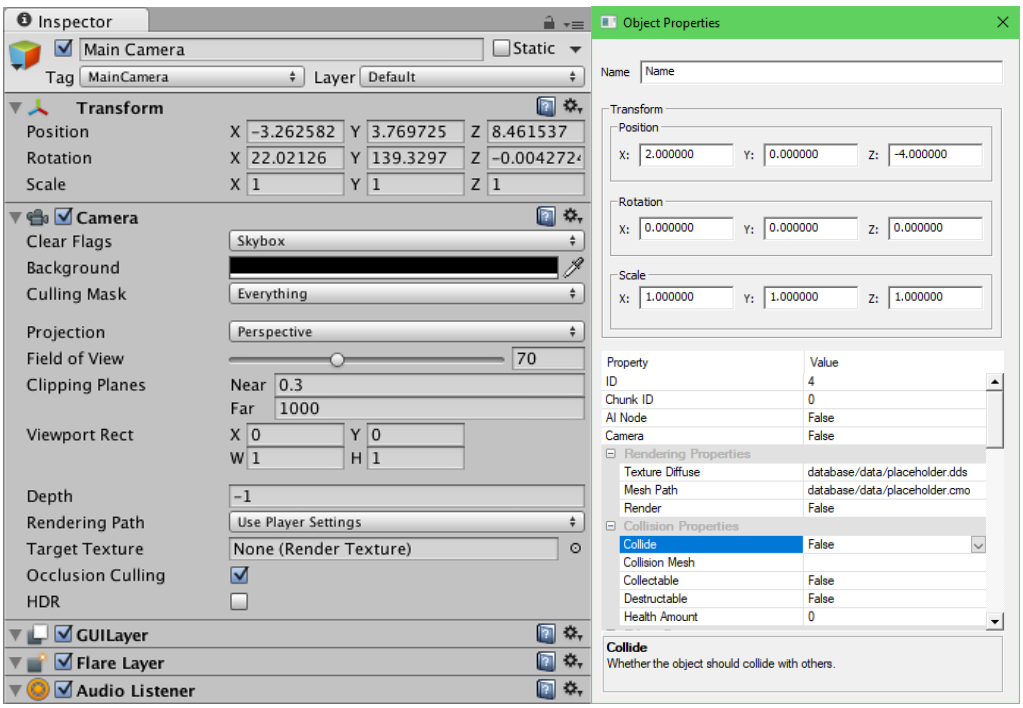


Figure 1 & 2 – Unity Inspector (Left), Properties Window (Right).

Name and Transform Fields

A new class, CFloatEdit, was created (derived from CEdit) to ensure that the inserted values can be changed into floats. The check is performed using the std::stof function – if it fails the number is determined to be invalid. This does sometimes allow for characters to be inserted however the conversion has not failed and allows for expressions such as 1.001E12 to be correctly evaluated.

CFloatEdit implements the IEventDispatcher, which uses the interface design pattern to add events. The dispatches calls the the OnRecievedEvent() on ObjectPropertiesDialog (which inherits from IEventReciever). The interface programming pattern was decided upon as to make the objects as detached from each other but also to allow for reuse (*Interfaces in C++ (Abstract Classes)*, no date). This is in order to call the DoDataExchange() function on the receiver; and due to the inability create custom events for MFC's event system.

The key changes over a CEdit include – applying the value both on pressing enter and loosing focus (CEdit only applied on the latter) and notifying subscribed classes (in this case ObjectPropertiesDialog) of the change.

Along with the name edit field, the transform edit fields use the DoDataExchange() to sync values between the currently selected SceneObject and themselves. This also automatically converts the field data to the correct value – on top of the std::stof check mentioned previously.

Property Grid

The CMFCPropertyGridCtrl was used to store the scene object's properties, as it was an inbuilt method to display large number of properties. Its properties (CMFCPropertyGridProperty) have inbuilt input validation through the use of COleVariant; as well as a file selection and colour picking dialog.

With most properties, the COleVariant had to be changed into the correct type before being set as a property. For example, a Variant uses the VARIANT_BOOL type which represents true as -1 and false as 0. This needed to be cast back to a standard template library bool (which represents true as 1 and false as 0). Another key cast was between the string value – first cast to CString then to std::wstring or std::string as needed. The namespace COleVariantHelpers was added with functions that were primarily used to help with changing between types.

The CMFCPropertyGridFileProperty had to be handled slightly differently as the file paths are relative to the working directory (where the executable is typically located). To convert the path from absolute to relative, an inbuilt MFC method was used. Additionally, checks for whether the file actually exists (by using the std::filesystem::exists() method) were added, in addition to a popup which will tell the user it doesn't exist. As a consequence of the CMFCPropertyGridFileProperty using a COleVariant which requires an reference value, r-value, to be initialised, the path to the model/texture/collision mesh/audio file are stored in a CString, which is then later cast to a string to be put in the database. This is in order to be able to pass a r-value into the constructor; and not a temporary l-value which is not guaranteed to stick around.

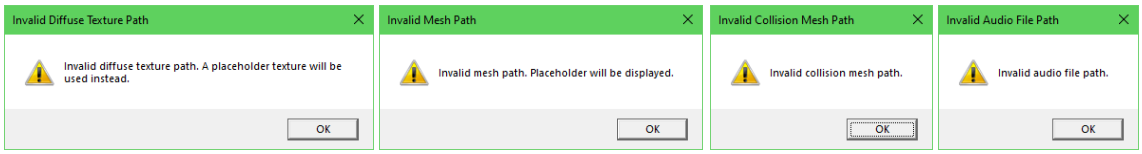


Figure 3 – Invalid path warning message boxes.

An assumption was made that the paths can be empty if the scene object is used for purposes other than display; and that the path doesn't have to be valid always as the object may be in the works for the time being. For these reasons, an invalid path can be set.

The CMFCPropertyGridColorProperty was used for setting the light diffuse and specular as it provides a quick preview along side pre-sets. The retrieval and setting of the values uses the GETRValue() and RGB(r,g,b) macros, respectively.

Notably, the ID and Chunk ID fields are not editable but visible. This is as the parent_ID requires the ID of an object, which cannot be inferred otherwise.

Object Properties Dialog Class

The class wraps the logic for the object properties dialog. On top of the aforementioned IEventReciever and DoDataExchange() implementations, it is responsible for dealing with updating display elements related to the currently selected scene object, and updating the scene object with new information.

The class is informed of changes to the selection by the Selection class. After being notified, the fields and property grid are updated. If no selection is present, the properties are greyed out. When an object is selected, it is also marked as changed, as to inform the renderer to rebuild the texture and model.

Additionally, the CMFCPropertyGridProperties are indexed by using a std::unordered_map looking up to their database field name, in order to make finding and accessing the properties linear (O(1)), according to Varun (2017). This would help with making changes to large scenes, as the previous std::vector storage method required iterating sequentially through all elements and finding them in an std::vector, causing worst complexity to be O(n).

Camera Navigation

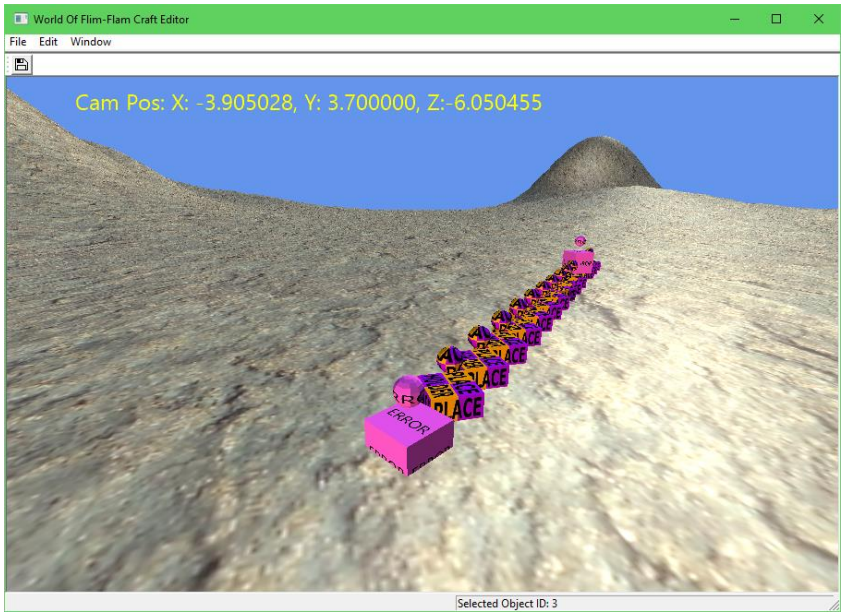


Figure 4 – Camera preview.

The camera navigation was improved by adding mouse look and up/down movement control, as well as keyboard bindings for the new motions. This was added as navigating a scene using mouse look is standard in most editors and games (Unity Technologies, no date). In addition, the camera functionality was moved away from the Renderer class to its own class called Camera.

The mouse look was achieved by calculating the change in mouse coordinates, then mapping them onto x/y rotation. This addition required changes to the handling of rotation, to where it was changed to Euler angles, then later converted to radians when the view matrix was being calculated. Additionally, the input is ignored if the window with the renderer is not in focus. The mouse is also moved to the middle of the screen, as to avoid the mouse going out of frame and losing focus.

Lastly, for keyboard inputs, the InputCommands struct was expanded to reflect the additions to movement; and additional bindings were added to be able to move the camera only using the keyboard.

Selection Changes

Mouse selection was added as it is a standard selection technique in many 3d editors such as Unity (Unity Technologies, no date). It was added using the method described in the practical (ray collision with model angle aligned bounding boxes), however the functionality was moved to its own class. Additionally, objects that are marked as not to be rendered in editor are excluded from the mouse picking. This is as it would be confusing for the user to be able to select invisible objects. Support for selecting multiple objects or getting all objects in sight exists though is not used.

The move of the selection functionality to its own class helped maintainability and portability. It maintains compatibility with previous classes – potentially helping if this was a larger codebase. The current selection was moved to be sorted by ID.

The selection window was kept, as it allows the selection of objects that may be hidden from rendering.

Renderer Changes

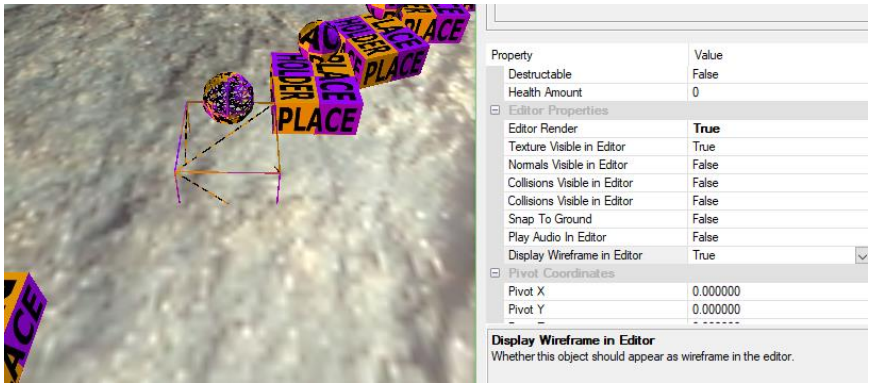


Figure 5 – Example of hidden in editor and wireframe in editor objects.

Support was added for not rendering objects, and rendering them in wireframe, as per the editor_render and editor_wireframe properties in the database. This was added to accurately reflect the value in the properties window; and to allow the user to hide objects that may be occluding others, but also to see the wireframe of a model.

In addition, when objects are modified, only ones which have been modified are rebuilt. This is to cut down on time required to insert a new scene (especially scenes containing many scene objects, which is probable in larger scenes in context of the game). A modification was added to the rebuild process, if the model path cannot be found the placeholder model is now loaded. This is to avoid exceptions being thrown if an asset is missing.

The display graph has also been changed to be indexed by the ID of the object it is rendering. This was done as it is easier and quicker to find the DisplayObject that must be rebuilt. The performance should be O(1) for finding an object (Varun, 2017).

ToolMain Changes

Like with the Renderer, the scene graph was changed to be indexed by the object ID in order to make lookup quicker (about O(1), according to Varun (2017)) and easier.

The currently selected object is now managed by the Selection class. The current selection has also been changed to reference in terms of the scene objects ID in the database – not its index in the scene graph. This is to avoid mistakes and make cross-operation safer; as well as making the selection refer to the database ID which helps in coherence of the items displayed in the property window (i.e. the index of the select object matches the actual object ID, not the order it was loaded in).

The ToolMain class also was made to use the singleton design pattern. This was to avoid extra setup complexity, for example when new classes require a reference to ToolMain that would otherwise require it to be passed in. In addition, it is assumed that there will only ever be one instance of this class. This pattern generally simplifies code; however, it affects performance as the instance of ToolMain is unlikely to be close in memory to the window instance. The implementation has a static method SetInstance() where the static instance variable is updated when the constructor is called with a reference to the instance; and the static GetInstance() function is used to retrieve the instance.

Save Function

The save function has been modified to use an UPDATE query as it is safer than the previous approach of dropping all the previous records, then adding them back in individually as new objects. It also is faster as there is one less major operation (Dyptorden, 2015).

Extra functionality was added to inform the user whether the save of a specific object has failed through a message box. Otherwise attempts are made to save as many as possible objects. This is to minimise the amount of work lost in the event of a failed query. If there are some errors saving this is indicated both when an attempt to save the object is made, but also after iterating through all the objects.

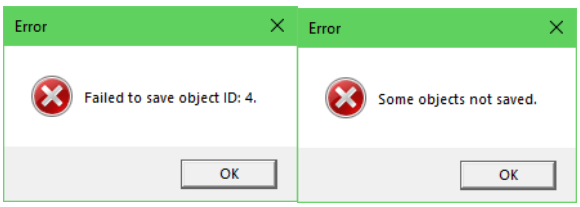


Figure 6 – Dialogs after a failed save of object with ID 4, on fail (left), on completing save action (right).

Other Changes

- The save icon is a floppy disk, not a smiley face, to follow conventional icon schemes.
- SceneObject has getters and setters for paths, as a reference (not an l-value) is required for the COleVariant value, which dictates what is displayed with a CMFCPropertyGridFileProperty.

Reflection

Renderer

I would consider the changes to the renderer to be worthwhile and useful, as they help reflect the properties on the database better, and aid with reusability.

The addition of rebuilding of only select objects on change will impact user experience positively too, as the whole scene does not have to be rebuilt. Moreover, this improves the existing code by moving building an object to its own class, making it easier for future development.

Further rendering enhancements would have to take place to reflect all available editor options, however. The following properties have not been implemented:

- Texture visibility in renderer – would have to investigate how to render an object without a texture using the current renderer.

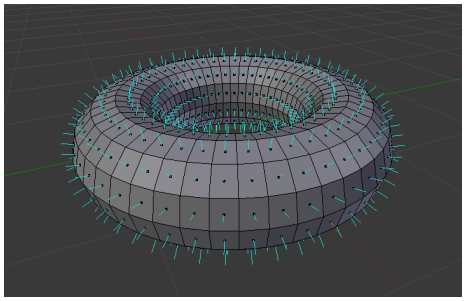


Figure 7 – Suggested normal rendering approach, retrieved from *Normals — Blender Manual* (no date)

- Normal's visibility in renderer – a line would have to be rendered away from the vertex/face, similar to the normals view option in Blender (Normals — Blender Manual, no date). Additionally, an option to render face/point normal would have to be added to the menu bar.
- Collisions visible in editor – rendering a semi-translucent, wireframed collision mesh over the current mesh.
- Pivot location view – rendering a chequered sphere on top of the current model (would need to be drawn after the model, disregarding the depth buffer).

3D View/Camera

The 3D view was improved substantially by adding mouse look. The implementation is largely reliable, though the implementation of the mouse look uses Euler angles, which could suffer from gimbal lock. This could be mitigated by adding reset camera button or by moving the rotation to quaternions (Neale, 2019).

The implementation could be polished by hiding the mouse while using mouse look and returning it to where the user originally clicked once they finish the mouse look.

Focusing on objects would also be a useful feature, especially if they are hidden in the editor. This was planned, but due to time constraints not implemented. The functionality would simply 'move the camera' (technically the world) to the object making sure its bounds fit into the viewport.

ToolMain

The ToolMain class underwent many changes. Generally, the move to a singleton pattern helps to decouple it from other aspects of the code, making future implementation easier, though potentially

affecting memory locality, which may decrease performance affecting the user experience negatively.

The ToolMain class could use further splitting up of functionality to keep it to a single responsibility. The following functions of the class could be made into their own classes:

- Input and which window is focused.
- Saving and loading from the database.

Object Properties Window

The primary focus of this expansion of the tool editor was implementing an object properties window. The final implementation achieves its goals of being easy to use, displaying all available properties, and validating input.

The implementation and setup largely revolve in the ObjectPropertiesDialog class. While the code is idiomatic, creating an independent class to represent the object properties grid would be beneficial, as it would reduce the number of responsibilities that the class has – aiding maintainability. Furthermore, this would allow for the grid to be used in more places than just the object properties window, increasing reusability.

The window could benefit from a 3D preview of the object that is being edited. This would make changes more apparent. However, this would require reworking the renderer class and making it reusable, on top of reworking which render window gets the focus.

During the implementation of the CMFCPropertyGrid, a large problem was adapting the scene object's properties to work with the SetData() and GetData() functions, which used a COleVariant. This required adapting the paths to be represented as a CString, as a COleVariant requires a reference to the value – which could not be given as originally the path was stored as a string and required conversion via a function which would return a temporary value.

The storage of paths and string should also be changed to use a std::wstring in virtually all cases, as it is largely compatible with a CString and requires less conversions to take place or to lose Unicode characters such as Emojis; as well as not breaking paths with Unicode characters on Windows (Robertson, 2018). This, however, is not possible as the database uses UTF-8 encoding, changes to which should be considered in the future.

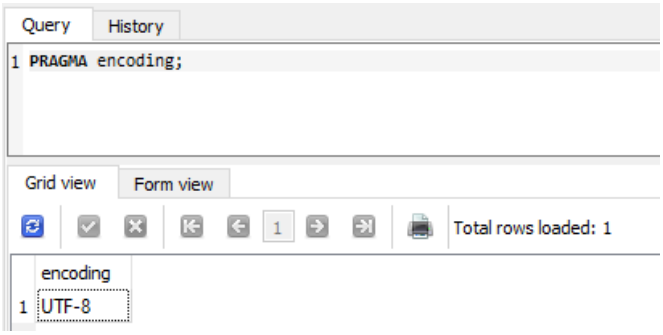


Figure 8 – Database encoding.

Additional issues were faced when trying to restrict the values the user can type into the transform fields. This resulted in the creation of the CFloatEdit class, which was derived from CEdit.

UI/UX Design

The consideration behind the UI design of the object properties window is outlined earlier. Overall, it achieves its goals well – using standard user interface elements and make the most identifiable elements stick out. The addition of scalability for the window will also aid in usability in case the user has a smaller screen.

An area that can be improved is the tab controls. When tab is pressed it should move onto the next edit field. Another field for improvement is the ability to edit in a similar fashion to Unity, where when the field is clicked and held and mouse moved to the left, the value is changed to be a slider. This would make quick increments a lot faster to add.

Adding and Removing Scene Objects

The tool provides some basic functionality for editing scene objects, however, lacks adding and deleting objects. This was not implemented due to time constraints. This addition would allow the user to edit a scene fully, not just edit existing scenes.

The general procedure for adding new objects would have been:

1. Display a dialog which asks user for position of the new object, which would be accessed by a new 'Object' drop down on the menu bar.
    a. A duplicate button inside of the object properties window could also be used to 'add' new objects.
2. The position would then be used to determine which chunk it must be placed into by checking the against each of the chunks bounds/location.
3. Query database for a new valid ID (as the ID is not autogenerated in the database configuration).
4. Initialize object with default parameters (no texture etc.) and place into scene graph.
5. Mark for rebuild so the object would be built into the display list.
6. Set as current selection.
7. Add to items to save as new (requiring new INSERT query) when the save function is called (also potentially move database loading/saving operations to its own class).
    a. Changing scene graph should not modify the database until the user clicks 'save'.

Deleting objects would be implemented in a similar fashion:

1. Add a delete button to object properties window, as well as in the 'Object' drop down of the menu bar.
2. Unset from current selection
3. Mark as to be deleted in the 'Objects' table.
    a. Ensure that the database not modified until the user hits 'save'.
4. Delete from scene graph.
5. Delete from display list.

Selection

The change of the current selection representing the ID of the object in the database (as opposed to the index in the scene graph) would have benefited its type changed into a struct (or typedef) to make refactoring easier.

Another way to update certain selections would be to add support for events to be dispatched to subscribers (perhaps through IEventDispatcher to IEventRecievers), which would make classes that

rely on changing their contents when the selection is updated (such as ObjectPropertiesDialog) easier to implement in the future.

Multiple selection would have been a very useful addition but was not added due to time constraints. The base functionality for multiple object selection has been implemented – the Game::MousePicking() function returns all items selected in a row. This function would require changes to return all items that are in a selection box (made by clicking and dragging on the render view). The object properties window would have to be modified to reflect the multiple properties, and handle changing multiple properties at once.

Conclusion

Overall, I am satisfied with the implementation of the object properties dialog, mouse look, and usability changes. These additions will make the tool easier and more intuitive to use, as well as expanding editing functionality using standard user interface components. Flexibility was kept in mind when designing the user interface and input checks (such as with paths). The code was made more maintainable and reusable which would help in the future development of this tool.

References

Dyptorden (2015) *In SQL, is UPDATE always faster than DELETE+INSERT?*, *Stackoverflow.com*. Available at: https://stackoverflow.com/a/29141391 (Accessed: 22 April 2021).

Interfaces in C++ (Abstract Classes) (no date) *Tutorialspoint.com*. Available at: https://www.tutorialspoint.com/cplusplus/cpp_interfaces.htm (Accessed: 22 April 2021).

Neale, P. (2019) *Gimbal lock*, *Paulneale.com*. Available at: https://paulneale.com/gimbal-lock/ (Accessed: 23 April 2021).

Normals — Blender Manual (no date) *Blender.org*. Available at: https://docs.blender.org/manual/en/2.79/modeling/meshes/editing/normals.html (Accessed: 23 April 2021).

Robertson, C. (2018) *char, wchar_t, char16_t, char32_t*, *Microsoft.com*. Available at: https://docs.microsoft.com/en-us/cpp/cpp/char-wchar-t-char16-t-char32-t?view=msvc-160 (Accessed: 23 April 2021).

Unity Technologies (no date) *Scene view navigation*, *Unity3d.com*. Available at: https://docs.unity3d.com/Manual/SceneViewNavigation.html (Accessed: 22 April 2021).

Varun (2017) *map vs unordered_map*, *Thispointer.com*. Available at: https://thispointer.com/map-vs-unordered_map-when-to-choose-one-over-another/ (Accessed: 22 April 2021).