Robert Grier
3/15/2024

Demo Video:
https://youtu.be/mWYN3-QDYbc

Research Project: World Editor

Introduction

The “World Editor” is a mode in a game engine which allows the game environment to be modified with a detailed user interface. For this research project, I implemented a world editor on top of my pre-existing 3D game framework. The key features are the GUI integration, selection architecture, transform handles, action history, scene management, the clipboard, object hierarchy management, and engine state transitions.

ImGui

The GUI for the Azul Editor is built with the ImGui framework. This library was chosen because it is an open-source project specifically designed for real-time tools. IMGUI is encapsulated in the **EditorGui** class and its subcomponents. While active, the game engine allows ImGui to draw to the main window and create a “dock-space” where any sub-window can be docked and moved around based on the user’s preferences. The game world is rendered to an intermediate DirectX texture object, which is simply drawn as an image every frame in one of the docked windows. This behavior is encapsulated in the **Viewport** and **WorldWindow** classes. The other windows were developed incrementally to enable editor features as they were added.

Object Selection

In order to start editing the world, objects need to be selected so that they can be worked on in isolation or as a small group. The **Selection** class contains the collection of selected objects. The selection really stores a collection of **EditorObjectReference** instances. These serve as a non-owning list node class for game objects as well as a layer of indirection for defining state and behavior that is relative to a game object, but only relevant to the editor systems. It is natural for users to select objects by clicking on them with the mouse. The is done by casting a ray from the mouse position into the 3D world and testing for an intersection with the objects’ bounding spheres. The closest intersecting object will be selected. More objects can be added to the selection with the CTRL key modifier. The selection can be cleared by clicking away from the objects.

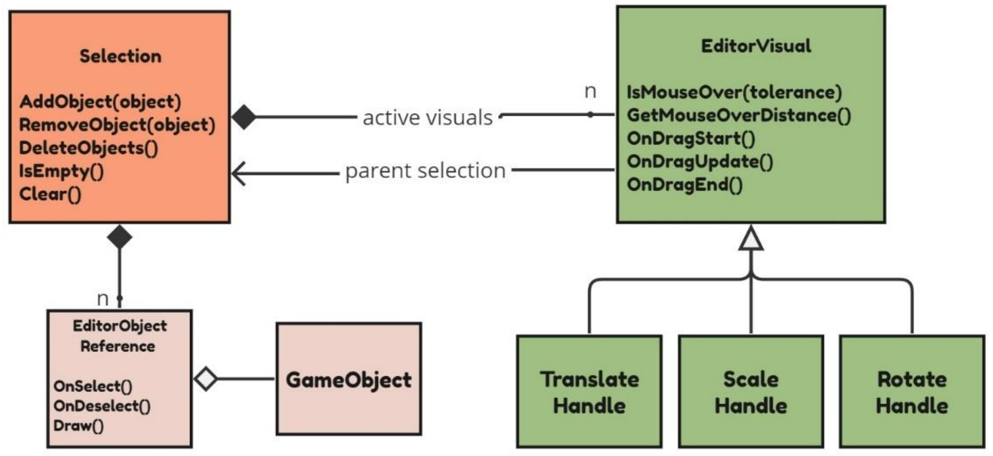
```
Vec3 Camera::GetRay(float x, float y) {
  const Vec4 pointCameraLocalSpace = Vec4(x, y, 0.0f, 1.0f) * projMatrix.getInv();
  const Vec4 rayCameraLocalSpace(pointCameraLocalSpace.x(), pointCameraLocalSpace.y(), -1.f, 0.f);
  const Vec4 rayWorldSpace = (rayCameraLocalSpace * viewMatrix.getInv());
  return Vec3(rayWorldSpace).getNorm();
}
```

The mouse ray calculation places the 2D mouse position into screen space coordinates, where depth is from 0 to 1 ($z = 0$ and $w = 1$ for a point on the near plane). The point is then transformed by the inverse of the projection. The resulting point is then transformed into a pure vector from the origin to a point on the near plane of the NDC, where depth is from -1 to 1 (z = -1 and w = 0 for a vector to the near plane). Finally, it is transformed by the inverse of the view matrix and the ray direction in world space is returned.

Visual Handles

There are three affine transform types that can be applied to the selected objects: translation, rotation, and scale. It feels natural when there is a handle in the game world that can be clicked and dragged to apply the desired transformation. To support these handles, I defined a new type of object called an **EditorVisual**. These are 3D models that appear on top of the selected object. The three types are the **TranslateHandle**, **ScaleHandle**, and **RotateHandle**. The translation and scale handles appear as axis bars that apply the relevant transform along that axis. The rotation handle appears as a ring around an axis and applies a rotation around that axis.

The **EditorVisual** contract allows each transform handle type to have unique behaviors under a common interface using the strategy pattern. The high-level code responding to input only needs to call these methods for each active handle. With these abstract methods, the editor can determine which handle is hovered or dragged and can call the corresponding events. This also makes it easy to add new types of handles that only need to fulfill this contract. When the transform mode is changed by the user, the corresponding visuals are activated. Bespoke edit modes with unique combinations of visuals can easily be configured.

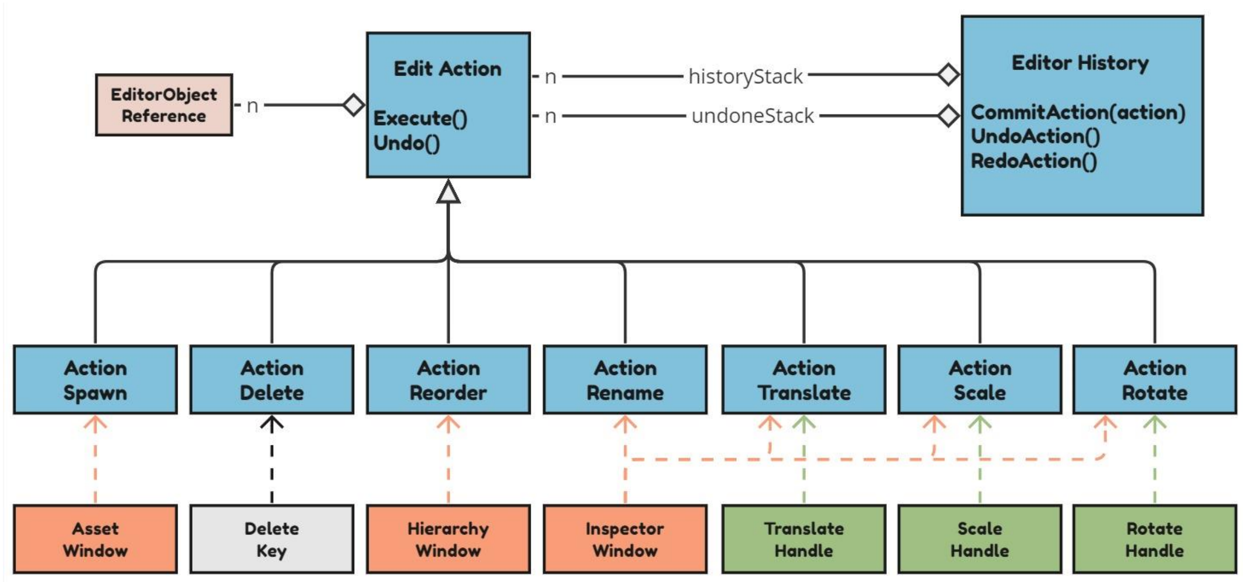


The translate and scale handles have similar implementations. They both simulate a cylinder bounding volume around their handle model because the models are basically arrows pointing down an axis. Once the handle is clicked, dragging is calculated based on how far along the relevant axis the mouse is dragged. Rotation is significantly different. The rotation ring uses a disk-like collision volume. The ray vs disk algorithm calculates the point on the disk's plane where the ray will hit, then determines if that point is within the width of the disk. For dragging the ring, a tangent line is calculated based on where the ring is grabbed. The amount of drag is then based on how far up or down that tangent line the mouse is moved. This creates a natural steering wheel effect where "pushing" the ring in the right direction rotates the object like it's attached to a wheel.

The object needs to respond to the dragged transform in real-time. For example, as the user translates an object along the x-axis, the object should move and follow the cursor such that the resulting translation can be previewed. This is done by updating the handle every frame as it is being dragged, which in turn updates the underlying game object with the transform-in-progress. However, once the drag is released, only a single transformation should be committed to the action history. The is done by having a callback when the drag ends, which calculates the total delta transformation and adds it to the history. The history is described in the next section.

Actions and History

A professional editor needs to support a history so that users can undo and redo their actions. This drastically improves the usability of the tool and prevents frustration. For maximum functionality, as many features of the editor as possible should utilize the history. The **EditorHistory** class implements this behavior with the command pattern. Each action type is represented by an **EditAction** class. The editor history stores two stacks of heterogeneous edit actions: the history stack and the undone stack.



When the user does something in the engine, such as using a transform handle or simply renaming an object, it executes and commits a specialized **EditAction** object for that action to the **EditorHistory**, and the action gets pushed onto the history stack. When the user presses the undo key, the latest action on the history stack gets popped from the stack and a virtual **Undo()** function is called to undo the effect of the action. Then, the action is placed onto the undone stack. When the user presses the redo key, the action is popped off the undone stack, the virtual **Execute()** is called to redo the effect, then it is placed back onto the history stack. When a new, manual action is performed, the undone stack is cleared, and any undone actions can no longer be redone. Jakubec et al describe this pattern and some more complex patterns in the **Undo/Redo Operations in Complex Environments** paper.

Scene Management

These tools allow a relatively complex scene to be created. However, this is only useful if the resulting state of the scene can be saved and reloaded for future editing and usage in a game. The **SceneManager** class manages the saving and loading of scene files. The state is marshaled into a JSON object with a top-level array where each JSON object array element represents a game object. Each JSON object contains the information needed to recreate the object including the graphics, transform, and tree structure data. The JSON is simply written and read from a file. The user can save and load scenes using the GUI, and the last scene is automatically loaded when the engine starts. The advantage of a human-readable file format is that changes to the scene can be visualized in a code difference tool or in source control. Additionally, changes to the scene from multiple developers could potentially be merged.

Clipboard

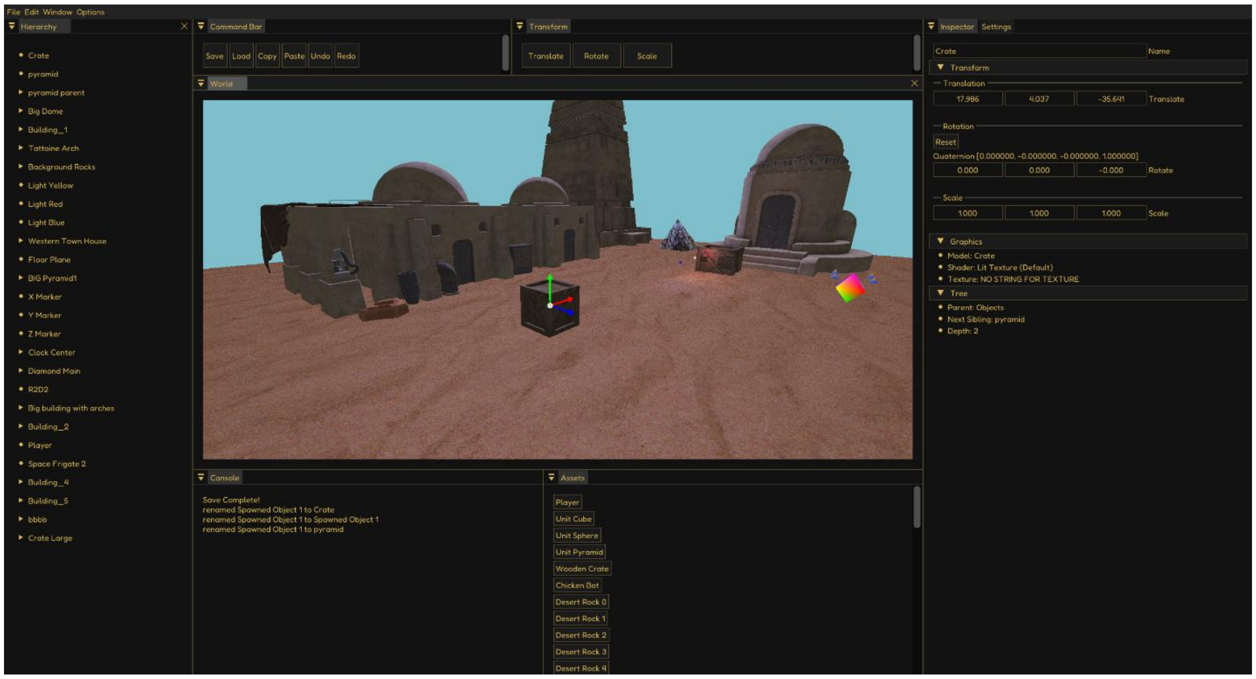
Another set of UI tools that are often taken for granted are copy, cut, and paste, which are implemented in the **Clipboard** class. This has a special meaning in the game world where the content is a set of game objects and their state. When a game object is copied and pasted, it should spawn in the exact same state as the previous object when it was copied. However, it would make sense to change the position of the object for convenience, because you usually want your copied object to be somewhere new. Cut is simply the same as copy but with the deletion of the original object. Whenever one of these commands removes or adds an object, it does so through an action command so that it can be undone and redone. In order to preserve the state, the copied object is marshalled to JSON using the same code and format as the save system. The pure JSON representation lives in memory on the **Clipboard**. When you paste, the content of the JSON is read and produces the new objects, just like the loading code of the scene system. The advantage of the marshaled format is that it is independent of the original object and will capture a snapshot of its state at the time of copying. Additionally, the JSON representation could potentially be written to the clipboard of the operating system and reused between executions of the program.

Object Hierarchy

My existing engine for the world editor contained a tree structure for organizing the game objects. An important part of a modern editor is the ability to visualize and interact with this tree. The **HierarchyWindow** class contains GUI code to display the structure as a list of dropdowns and selectable elements. Each object can be selected with a click or navigated to with a double click. Additionally, the objects can be reordered and re-parented. This is accomplished by placing small UI spacers between each element. The elements can then be dragged onto the UI spacers and the system will reorder the objects accordingly. Elements can be dragged onto other elements to make a parent-child relationship. Reordering the tree is done through an edit action so that any changes will be stored on the history stack.

Object Inspection

Another window that most editors provide is the **InspectorWindow**. The purpose of this window is to give detailed information about the selected object. My inspector is functional and allows the user to apply 3D transformations through a slider or text input as a more precise alternative to the 3D model handles. These transformations go through the history-system and benefit from undo and redo commands. The inspector also allows objects to be renamed. To create a transactional renaming action, a name cache is stored and only applied to the object once the input is committed.



- Crate
- pyramid
- pyramid parent
- Big Dome
- Building_1
- Tattoine Arch
- Background Rocks
- Light Yellow
- Light Red
- Light Blue
- Western Town House
- Floor Plane
- BIG Pyramid1
- X Marker
- Y Marker
- Z Marker
- Clock Center
- Diamond Main
- R2D2
- Big building with arches
- Building_2
- Player
- Space Frigate 2
- Building_4
- Building_5
- bbbb
- Crate Large

```
Crate                                  Name
▼ Transform
— Translation
17.986      4.037       -35.641       Translate
— Rotation
Reset
Quaternion [0.000000, -0.000000, -0.000000, 1.000000]
0.000       0.000       -0.000        Rotate
— Scale
1.000       1.000       1.000         Scale
▼ Graphics
• Model: Crate
• Shader: Lit Texture (Default)
• Texture: NO STRING FOR TEXTURE
▼ Tree
• Parent: Objects
• Next Sibling: pyramid
• Depth: 2
```

```
Save Complete!
renamed Spawned Object 1 to Crate
renamed Spawned Object 1 to Spawned Object 1
renamed Spawned Object 1 to pyramid
```

Game Engine States

To iterate on a game project, users of the editor need to be able to quickly jump in and play their game. Users also need to be able to quickly exit the game and return to the editor to make changes. While in the editor, gameplay code should not be executing and changing the state of the scene while the systems and levels are being designed. Likewise, when the game is running, editor systems should not be interfering with or slowing down the game's execution.

This dichotomy is achieved in my editor with a state machine that can either be in the **EditorState** or the **PlayState**. The editor state draws the GUI with the 3D world as a sub-window. It also updates all of the editor systems. When switching to the play state, a couple of changes happen. The camera changes from the editor's god camera to the desired player camera. A special virtual **Start()** method is called on all existing objects to indicate that the game has started. For drawing, the editor GUI is not present, and the 3D world is drawn directly to the swap chain. For updating, the game objects start receiving update calls and potentially start moving around and acting out their behavior. The editor systems are no longer updated, except for a single check for exiting play mode. When the user exits play mode, the camera is switched again, and the update and draw behaviors return to the editor behavior. The original state of the scene written on-file is reloaded. This system could be enriched in the future by added pause and observe states, where the user can jump back into the editor with the game state paused or still running.

Sources

ImGui
https://github.com/ocornut/imgui

Undo/Redo Operations in Complex Environments
https://www.sciencedirect.com/science/article/pii/S1877050914006619?via%3Dihub

OpenGL implementation of Mouse Position to Camera Ray (ported for selection algorithm)
https://stackoverflow.com/questions/71731722/correct-way-to-generate-3d-world-ray-from-2d-mouse-coordinates

Closest Point Between Lines Implementation (ported for drag along axis algorithm)
https://math.stackexchange.com/questions/1993953/closest-points-between-two-lines

Ray-Plane and Ray-Disk Intersection (ported for rotation ring handle)
https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-plane-and-ray-disk-intersection.html

Unity Mouse Position to Camera Ray Interface (as method of the Camera)
https://docs.unity3d.com/Manual/CameraRays.html

Godot Scene File Format
https://docs.godotengine.org/en/stable/contributing/development/file_formats/tscn.html

Unity Scene File Format
https://docs.unity3d.com/Manual/TextSceneFormat.html

Unity Documentation
https://docs.unity3d.com/Manual/index.html

Godot Documentation
https://docs.godotengine.org/en/stable/index.html

Unreal 4 Documentation
https://docs.unrealengine.com/4.26/en-US/