Summary of features

I have added a fly through camera, An input system using the observer pattern, A visual transform gizmo that is used to interact directly with the object to easily add transforms by non technical users in all translate, scale and rotate. The object selection, gizmo interactions and transforms are used with a mixture of standard and modified mouse picking logic, keyboard and mouse inputs. I've also added am MFC dialogue box the user can use to observe the position values and edit the position of objects using buttons.

Controls

I have implemented some of the controls visually with my transform gizmo implementation.

Translate is the default with no buttons held. Drag the line translators to drag in that specific line. Drag the plane translators (the small squares between the line translators) to translate in that specific plane.

Scale is activated while holding SHIFT. You can use the gizmos lines to scale the object in that local dimension. While holding SHIFT you can use the scroll wheel to scale the object evenly in all directions.

Rotation is activated while holding CRTL. Use the gizmos lines to rotate the object in that orientation.

Input Controls

Camera

Look Camera	RMB
Move Camera	WASD
Camera Up	Q
Camera Down	E

With object selected

Enable Scale	SHIFT
Scale all directions	Scroll Wheel
Enable Rotate	CTRL

Gizmo system

I crated a gismo system that the user can control objects visually by choosing and dragging the objects. The user can shift click to choose multiple objects and the gizmo will manipulate al objects simultaneously. The gizmo itself stays located on the master object (the first object chosen). When the player hovers over a gizmo direction (arm) that direction changes from Its respective colour to white to indicate what section of the gizmo is being intersected and what would be chosen. All three; translate, rotate and scale are controlled by the same gizmo.

Position control

The user can select any object and move the object in linear directions by selecting the longer arms or move the object in a planar direction by closing the smaller squares. When a plane is chosen, the object is moved with a plane line intersection, intersecting with a plane made with the location of the object in that dimension and position and the picking ray crated by the camera and the picking logic. If a linear direction is chosen the plane is still created but only one of the two dimensions are set depending on what linear mover has been selected.

Rotation control

The user can hold CTRL to change to rotation mode and choose one of the three arms and it will rotate in that direction. The horizontal arms are controlled by an up and down movement and the vertical arm is controlled by a side to side movement.

Scale control

The user can use SHIFT to control the scaling in a similar way to rotation by selecting one of the arms and dragging to change the scale. The horizontal arms are controlled by an up and down movement and the vertical arm is controlled by a side to side movement.

The user can use the scroll wheel while holding shift to scale the object on all directions at once.

Design

The gizmo is designed to increase user experience and be able to be used by almost anyone by letting the user control the objects in a visual way. Selecting and dragging instead of typing out coordinates. Using the gizmo system speeds up development time by letting users move, scale and rotate the objects quickly and efficiently. The ability to

multiselect lets users select many objects and move many objects at once, for example the user could select a campfire, tents, etc and move them all at once without needing to move them one by one and realign everything at the new location. I was inspired by the gizmos used in programs like unreal engine and blender.

Technical Discussion

I created the visual gizmo procedurally using DXTK primitives, creating the arms and squares with primitive lines and the cubes at the end of the arms with quads. I created a second overloaded picker function that takes in an array of bounding boxes so the boxes remain separate, generative and invisible. The bounding boxes on the gizmo surround the arms and planar squares. This function checks every frame to enable the ability to have the gizmo sections turn white when hovered. This will stay efficient as it is only 6 checks each frame.

A third picker was created to check the plane line intersection of the selected plane, this takes the plane in and returns a vector3 and the requesting function adjusts the position of the object so the object remains at the same offset that it started with. This acts the same with multiple objects with their own offsets to the intersection coordinate.

I moved the picker system to a class to be all static functions. This allows the picker to be used anywhere so long as the programmer has the appropriate input conditions. This also reduces code complexity and repetition.

Input observer system

I created an observer system for the inputs as the input system will be connected to many different parts of the program. I used Refactor Guru and Refactor Gurus C++ implementation to learn about the observer pattern and the C++ implementation as an example to follow for my own implementation to control the inputs.

My inputs tie to an interface class called *InputObserver.h* comprised of virtual functions, some functions for the input system to cycle though and subscribe the observer to the corresponding inputs, and the rest of the virtual functions are for the input system to call when the corresponding input arrives. When an input is activated, mouse movement, key press etc. The input class goes through the according vector calling each objects virtual function and passing in the specific input press.

The subscribers are held in vectors, for keyboard, an array of 256 of vectors of subscribers to separate each key in to a vector. Another for mouse clicks (LMB, MMB, RMB), one for wheel scroll and one for mouse movement. Subscribers can subscribe to as many as it asks, most subscribers have more than one subscription. Of the subscriber doesn't want to subscribe to a specific vector, it will pass back an empty

vector. The inputs are all set at initialisation but because I use vectors and functions inputs can be easily subscribed during run time by adding them to the input class subscriber vectors.

Command Observer System

I crated a second observer system to control the keyboard combo commands like ctrl c, ctrl v for copy and paste. The command manager is a subscriber of the inputs to take in the desired inputs for each command, an unordered map of command (an Enum that consists of copy, paste, undo, redo) to a vector of ints. The vector of ints means the programmer can add ay number of keys needing pressed to activate the command but will usually be 2 keys. This can be found in *CommandManager.h* and the Enum can be found in *CommandObserver.h*. Any object needing command input can use the command observer interface class and subscribe to any command by passing in a vector of commands. This it is future proof as a future programmer can edit the Enum by adding in more commands at the end of the Enum and adding in more key combo at the end of the unordered map and this will not affect any subscription system already set up. This was created for an undo redo system and a copy paste system, but I unfortunately ran out of time to complete the systems so the command observer system remains as progression towards an implementation that uses commands.

I attempted to create an undo redo system while only recording deltas, another simpler solution would be to record the entire display list which would work but for a real scenario of a world with thousands of objects, it would be very inefficient. The failed undo redo system remains on the github in a branch as broken-undo if interested.

Camera Movement

I added camera look movement with the mouse as this is standard practice and a fly through camera as it gives the user control over the camera to see anything from any angle. Camera WASDQE controls are accessible by the user any time to give them more control and these buttons aren't used while in object manipulation mode so the user can have some camera control during object manipulation. I used a cursor anchor so while looking with the camera

I added a new camera class that includes the logic and deals with the desired camera related input through the InputObserver interface class.

Object Inspector

I crated a MFC window so the users can observe the location and change the positions of the objects if desired. This allows the users compare positions of different objects for alignment and localisation.

Conclusion

I really enjoyed this tools module and developing the gizmo, learning more design patterns was very useful and satisfying, using interface classes and developing data structures that are flexible and easily expanded. I spent a long time on the undo redo system, I didn't research any methods and made it up as I went along. If I was to continue development, I would research methods and make some diagrams to understand the structure.

Video demonstration

https://www.youtube.com/watch?v=OWcHAxYrDIY

References

Observer (no date). Available at: https://refactoring.guru/design-patterns/observer (Accessed: 26 March 2025).

Observer in C++ / Design Patterns (no date). Available at: https://refactoring.guru/design-patterns/observer/cpp/example (Accessed: 26 March 2025).

PrimitiveBatch (no date) *GitHub*. Available at: https://github.com/microsoft/DirectXTK/wiki/PrimitiveBatch (Accessed: 26 March 2025).