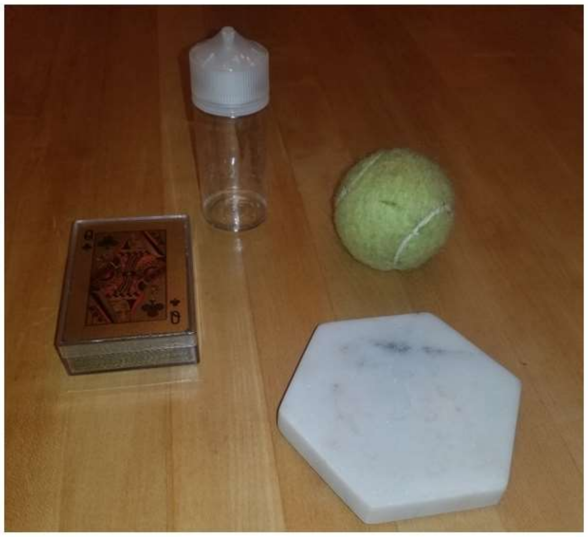
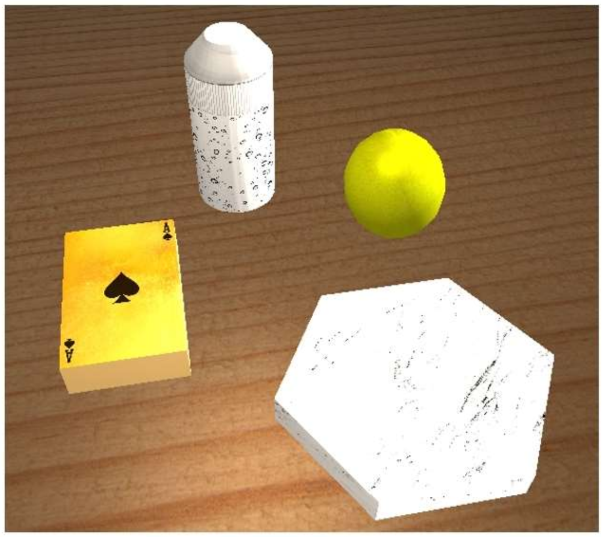
The objects I chose involved a juice bottle, a deck of playing cards, a coaster, and a tennis ball. The objects gave me the opportunity to learn and build a 3D scene comprised of inherently simple objects. However, each object was recreated using different shapes, which would allow me to experience how to utilize different meshes. These objects consisted of a sphere, cube, cylinder, cone, plane, and a hexagonal polygon. As most objects can be recreated in a 3D environment with these primitive shapes, the selected objects serve as building blocks for more complex object creation in future projects. Overall, each object ensured the project was realistic and achievable while gaining exposure to primitive shapes with varying difficulties.

Original



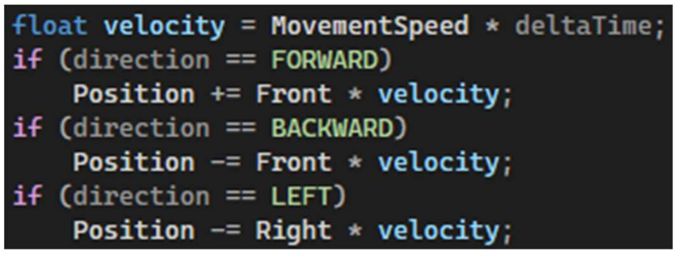
3D Recreation



During the project, there were a few different approaches I considered in developing the 3D scene. Initially, I had wanted to utilize a flashlight as one of my light sources to match the effects of the camera flash when taking the pictures for the project proposal. However, after implementation, it felt unrealistic to the feel of the scene as the ambient lighting had to be reduced to notice the effects of the flashlight. This was not acceptable as the picture was taken during the day, when ambient lighting would have greater impacts. Alternatively, I created two light sources, one overhead and one level with and to the right of the scene. The overhead light has a yellow-white color to match the overhead light fixture, and the right light is pure white to match the sunlight illuminating the scene from the nearby window. This approach provided a more realistic depiction of the scene with slightly noticeable object surfaces that were either illuminated or hidden in shadow.

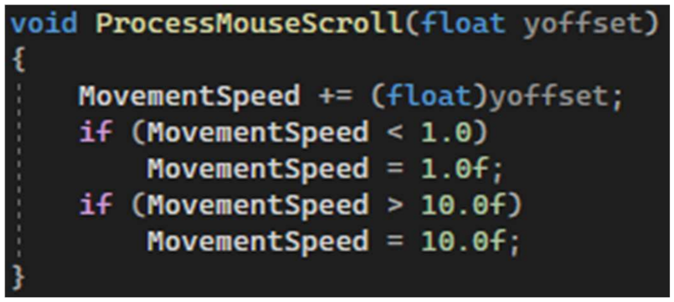
A second approach I had considered was how to go about object creation. The two design choices I had to compare were whether to fully design and implement objects one at a time or slowly implement them simultaneously. After deliberating, I decided it would be beneficial to create objects one at a time, as I had only worked on the same object throughout the milestones. This ensured I was comfortable while adding to the project incrementally, where I could frequently run to check for errors. Therefore, whenever an error or design flaw occurred, it was easy to identify the source of the issue and correct it as necessary. Furthermore, it allowed me to later consolidate duplicated code and verify no unintentional side effects were introduced into the code base.

The user can navigate the 3D scene using two separate input devices. I ensured to include highly customizable and user-friendly navigation controls. This is because it allows users to better immerse themselves in the 3D scene and provides the ability to verify proper object creation, and small adjustments can be performed precisely. The navigation controls for the 3D scene are like those used in video games, where W is forward, S is backward, A is left, D is right, Q is up, and E is down. This sequence adds a velocity value to the current camera position based on the user's selected input to change the camera’s location. As the frames are continually rendered by the program, this produces an effect of smooth camera traversal for the user. A few of these user inputs for modifying the camera location are shown in the figure below.



```
float velocity = MovementSpeed * deltaTime;
if (direction == FORWARD)
    Position += Front * velocity;
if (direction == BACKWARD)
    Position -= Front * velocity;
if (direction == LEFT)
    Position -= Right * velocity;
```

Furthermore, the mouse can be utilized to change the direction of the camera with precision by modifying the vector for the front of the camera. Camera speed is a preference for the user, so it was important to implement adjustability. The user can scroll up on the mouse wheel to increase the camera speed and scroll down to decrease the camera speed. It is important to put boundaries on this function to prevent the camera from going so fast or slow that it becomes unfriendly for the user or detracts from their experience. The sensitivity of the camera is shown in the figure below.



```
void ProcessMouseScroll(float yoffset)
{
    MovementSpeed += (float)yoffset;
    if (MovementSpeed < 1.0)
        MovementSpeed = 1.0f;
    if (MovementSpeed > 10.0f)
        MovementSpeed = 10.0f;
}
```

While a perspective view of the 3D scene is familiar, a user may want to have an orthographic view of the scene. The O key swaps the view to orthographic, and the P key changes the view back to perspective.

From the beginning, I wanted the code base to be modular and organized. In doing so, it was important to consolidate code wherever possible so developers could quickly identify locations to add, update, or remove code. One of the more crucial modularization tasks was separating the meshes and camera functions into separate .cpp or .h files. As the camera would likely remain unchanged, it did not need to take up space in source.cpp. Additionally, the meshes involved a significant amount of code to identify each vertex, normal, and texture coordinate. When isolated into a separate .cpp and .h file, each mesh is easily retrievable while not impeding on the organization of other sections. These separate .cpp and .h files could be of great benefit to other programs, as they contain targeted and focused functions that can be easily translated. They provide functionality that, if included, would supply the inner workings for a camera system and meshes that can be modified or added to easily.

I elected to not consolidate the code for instantiating each mesh in the URender function. This is because they did not contain a significant amount of code and were distinctly separated into commented sections for easy readability and maintainability. Furthermore, if objects were to be duplicated, it would allow developers to scale, rotate, and translate the objects quickly and within a relevant section. Nonetheless, the mesh and texture rendering could have been encapsulated into separate custom functions. This approach would significantly cut down on the size of the URender function, and the new functions could accept values in function calls to perform the scaling, rotation, and translation operations. In doing so, multiple objects could be created or duplicated by passing predefined information to other functions. However, it would increase the runtime cost as multiple function calls are performed as each frame is rendered and separate relevant code, which may increase the time it takes for developers to implement updates.