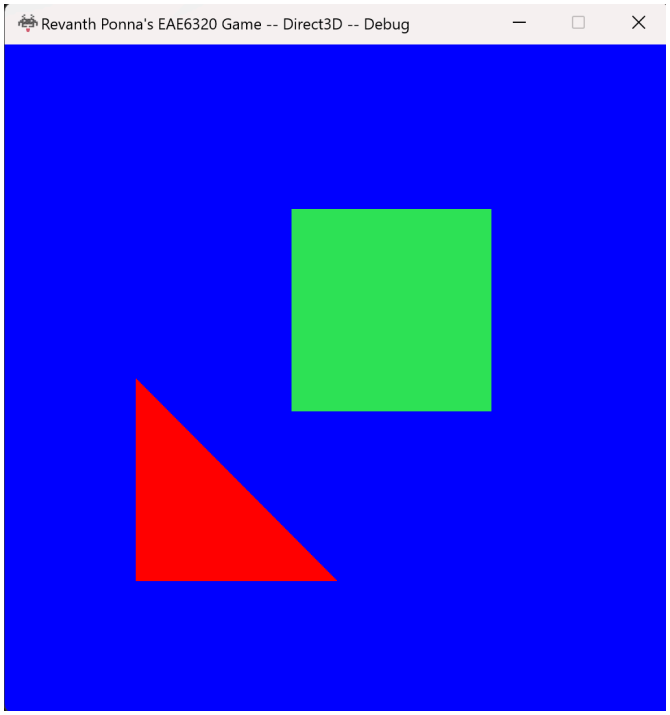
Write Up

Assignment 05 - GAMES6320

[Download Game]

Controls: WASD - Move Mesh, Arrow Keys - Move Camera, Space Bar - Change Mesh

Assignment 05 was a great opportunity for me to learn about how cameras work in game engines, and how movement and rigid bodies take input from the user and update velocities and positions. Additionally, I learned a lot about optimizing shader files and making them platform independent.

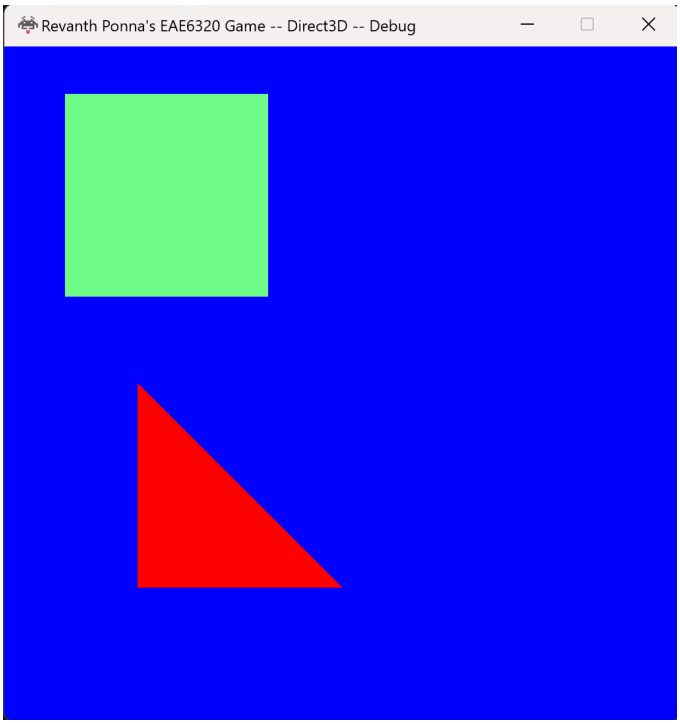


Default State Of Game

Movement

To move the square mesh around the screen, I set the user input to be the WASD keys. First, I initialized a member variable called playerVelocity in myGame, so that I could easily adjust this value for testing. Then, based on the input, velocity is added to the game object in that specific direction to ensure

smooth movement across the screen. Updating the velocity is done based on input, whereas the position is updated based on time.



Game After Moving The Square Mesh

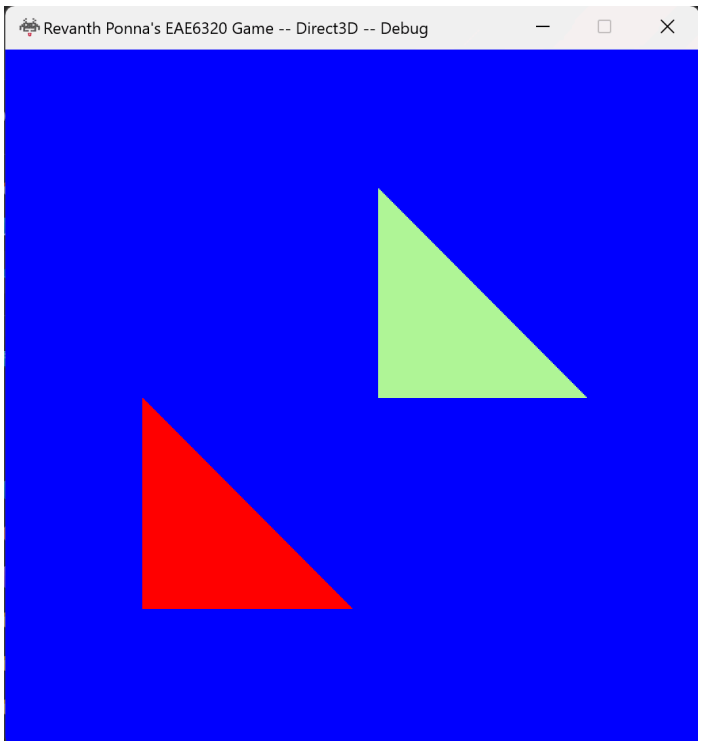
In order to make the movement smooth and not jerky, the method of extrapolation is used. Basically, this means that the program predicts where things will be based on how much time has passed. This is important because the Graphics engine runs at machine time (total time), and Physics runs at simulation time. So, most times, the Graphics will finish what it's doing, and keeps waiting for updates from other parts of the engine. Hence, to solve this, we multiply by deltaTime to smoothen out the movement.

Game Object Representation

To initialize a game object in my program, the data needed is a mesh, an effect, and a rigid body state. The class also stores the position and orientation of the game object, to keep track of where things are in the scene. I have also included member functions to set and add velocity to the rigid body, to ensure movement functionality for each object.

```
namespace Assets {
    class GameObject {
    public:
        GameObject(Graphics::Mesh* pMesh, Graphics::Effect* pEffect, Physics::sRigidBodyState);
        ~GameObject();

        void SetMesh(Graphics::Mesh* pMesh) noexcept;
        void SetEffect(Graphics::Effect* pEffect) noexcept;

        void Update(float i_secondCountToIntegrate) noexcept;
        void Rotate(const Math::cQuaternion& rotation) noexcept;

        void SetVelocity(Math::sVector pVelocity) noexcept;
        void AddVelocity(Math::sVector pVelocity) noexcept;

        Math::cMatrix_transformation GetTransformLocalToWorld(float i_secondCountToExtrapolate) const noexcept;

        Graphics::Mesh* GetMesh() const noexcept;
        Graphics::Effect* GetEffect() const noexcept;

        Math::sVector GetPosition() const noexcept;
        void SetPosition(const Math::sVector& i_position);
        Math::cQuaternion GetOrientation() const;
        void SetOrientation(const Math::cQuaternion& i_orientation);

    private:
        Graphics::Mesh* m_mesh;
        Graphics::Effect* m_effect;
        Math::sVector m_position;
        Math::cQuaternion m_orientation;
        Physics::sRigidBodyState m_pRigidBody;
    };
}
```

Additionally, the SetMesh function allows a game programmer to easily change the geometry of a mesh. Right now, this happens when the Space Bar is pressed down.



Mesh Changes When Space Bar Is Pressed Down

Renderable Objects

To make it as easy as possible for a game programmer to submit game objects to be rendered, I created an array of structs called renderableObjects. Each element of this array gets a mesh and an effect from the corresponding game object to be rendered. It then passes all this data to the Graphics engine to render the next frame.

```
void eae6320::cMyGame::SubmitDataToBeRendered(const float i_elapsedSecondCount_systemTime, const float i_elapsedSecondCount_since
{
    Graphics::ObjectRenderData* renderableObjects = new Graphics::ObjectRenderData[activeGameObjects.size()];

    unsigned int i = 0;

    for (Assets::GameObject* currObj : activeGameObjects)
    {
        renderableObjects[i].mesh = currObj->GetMesh();
        renderableObjects[i].effect = currObj->GetEffect();
        renderableObjects[i].drawData.g_transform_localToWorld = currObj->GetTransformLocalToWorld(i_elapsedSecondCount_sinceLast

        ++i;
    }

    Graphics::SetRenderableObjects(renderableObjects, (uint16_t)activeGameObjects.size());
```

The total size of the sDataRequiredToRenderAFrame struct is now 1752 bytes, with the addition of the draw call.

Platform Independent Shader Files

Before this class, I had never worked with shader files before. But throughout this semester so far, I have learned a lot about how shaders are structured, what they exactly do, and how they integrate with a graphics system.

In this assignment, I first refactored the shaders.inc file to create a platform-independent interface as was discussed in class. After that, it was pretty straightforward to restructure the rest of the shader files, so that they have a single constant buffer declaration and can have a single implementation that works for either platform.

[Download Game]