Easel: Performance Analysis and Overhaul

October 14, 2020

Ruben Young – Rochester Institute of Technology (RIT)

Introduction

Easel is the name given to a homebrew attempt at learning Direct3D 11 by making a test bed for implementing specific rendering techniques. As the primary objective at the time was educational, emphasis was put into readability and modularity in case mistakes were made in setup and systems had to be quickly changed. Unfortunately, this led to a highly object-oriented architecture with data update patterns that do not make advanced use of the hardware and the API. This has not been very conducive to good runtime performance, which should be first and foremost in the context of a game engine.

This report contains analytics on the engine's performance as is and documents efforts to improve it based on a combination of specific readings from industry experts, performance data from profiling tools, and qualitative analysis of the current architecture.

Goals

Expanded feature set: Currently, *Easel* only has the capacity to handle basic rendering with a very limited set of features. Core functionality such as alpha blending is entirely unhandled and would require the construction of entirely new systems. The rigidity and highly object-oriented nature of the current architecture makes expandability tough. Once alleviated, expanding the capabilities should be a priority.

Parallelism: For simplicity, *Easel* was made entirely single-threaded. This does not come close to making the best use of modern hardware, so it would be best to implement some sort of multi-threading or at least architect the engine such that it can be done in the future (i.e. through the implementation of a job-queuing system).

Adherence to recommended patterns: Present-day Direct3D 11 has matured enough for there to be well-documented ways of organizing its different components in order to maximize performance. One example of this is the update pattern for constant buffers, which changed significantly in DirectX 11.1. This engine overhaul hopes to bring *Easel* more in line with these types of paradigms which tend to have a better track record.

Baseline

As it stands, *Easel* is built for the Windows 10 platform only, using the Visual Studio 2019 toolset (v142) and the C++17 standard. Expanding it to work on other platforms is outside the scope of this overhaul but may be an area for future development should this project succeed.

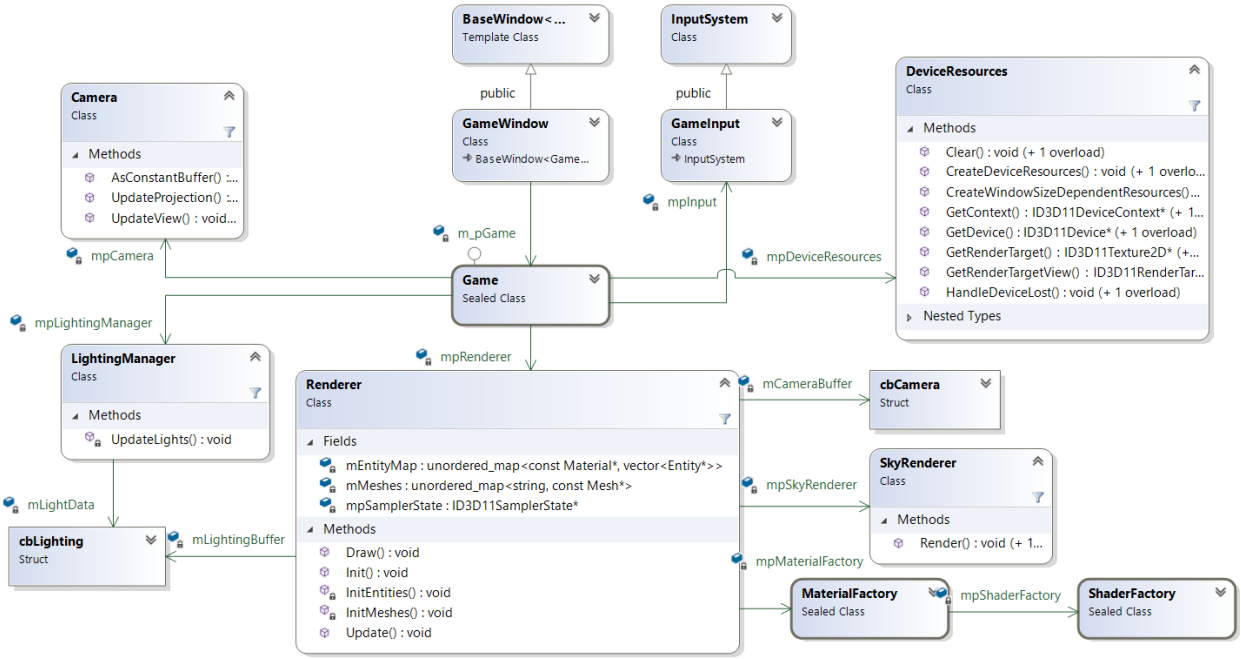

Fig 1 Class Diagram of Easel

As can be seen through the class diagram, *Easel* relies on a monolithic set of central classes (*Game, Renderer, DeviceResources*) in order to perform key application tasks such as rendering and managing the state of DirectX and its draw sequence.

Furthermore, the input system could use improvements. As seen in Figure 1, Input is kept separate from other systems as a private member of Game. This makes the task of updating tedious as the necessary data would need to be passed in explicitly from within Game, such as within *Game::Update*.

Additionally, explicit classes were made to control specific parts of the update-draw cycle, such as *LightingManager* to move and aim the lights and *SkyRenderer* for drawing the skybox and holding its associated pipeline states. This is not inherently an issue and can be a positive, but the issue comes with the highly object-oriented design for these systems and the lack of adherence from classes such as *Renderer*, which handles the creation and storage of meshes and entities, but delegates the actual drawing to each of the objects' *Draw* method.

```
void Renderer::Draw(ID3D11DeviceContext* context)
{
    using Game::Entity;
    for (std::pair<const Material*, std::vector<Entity*>> const& element : mEntityMap)
    {
        // Set material data, then bind
        // "Binding" a material means binding its internal VS/PS
        const Material* material = element.first;

        // Set data for lighting, camera buffers
        material->GetPixelShader()->SetBufferData(context,
               (UINT)ReservedRegisters::RR_PS_LIGHTS, sizeof(cbLighting),
               &mLightingBuffer);

        material->GetVertexShader()->SetBufferData(context,
               (UINT)ReservedRegisters::RR_VS_CAMERA, sizeof(cbCamera),
               &mCameraBuffer);

        // Bind() sets the contents of material params constant buffer automatically
        before PSSetConstantBuffers
        material->Bind(context);

        for (Entity* entity : element.second)
        {
            // Set Per Entity world matrix in the associated material's vertex shader.
            cbPerEntity perEntityCB = {entity->GetTransform()->GetWorldMatrix()};

            material->GetVertexShader()->SetBufferData(context,
                (UINT)ReservedRegisters::RR_VS_WORLDMATRIX, sizeof(cbPerEntity),
                &perEntityCB);

            // draw entity
            entity->Draw(context);
        }
    }

    mpSkyRenderer->GetMaterial()->GetVertexShader()->SetBufferData(context,
        (UINT)ReservedRegisters::RR_VS_CAMERA, sizeof(cbCamera), &mCameraBuffer);

    mpSkyRenderer->Render(context); // Render binds the material internally
}
```

Fig 2 Main drawing logic [Graphics/Renderer.cpp]

Looking at the main drawing code, several issues become apparent:

- High levels of indirection needed to store data into constant buffers
- Overly simple, rigid classification of drawables: everything is an "*Entity*".
- Inconsistent update pattern for subresources and bindables:
    - The lights and camera constant buffer' data must be explicitly updated, but the material parameters buffer is handled inflexibly within *Material::Bind*
    - Sky's material is bound within *Render* as opposed to explicitly with entities.
- Camera buffer requires a second data update every frame. This one is particularly bad and is due to the underlying logic for updating and binding constant buffers within the shader objects. This will be discussed later.

Baseline Performance

To observe the performance of *Easel*, a performance test has been set up with the following parameters:

- 75,000 Entities (Sphere, Cube, Torus, or Cylinder) using either of two materials spawned at a random position between (-40, -40, -40) and (40, 40, 40).
- Each entity is constantly rotated in the same way as others in its same material group
- A directional light is rotated in the XZ-plane uniformly

These parameters were not chosen for any specific reason, but only to simulate an arbitrary amount of work for the rendering system such that it hovers around 60 FPS.

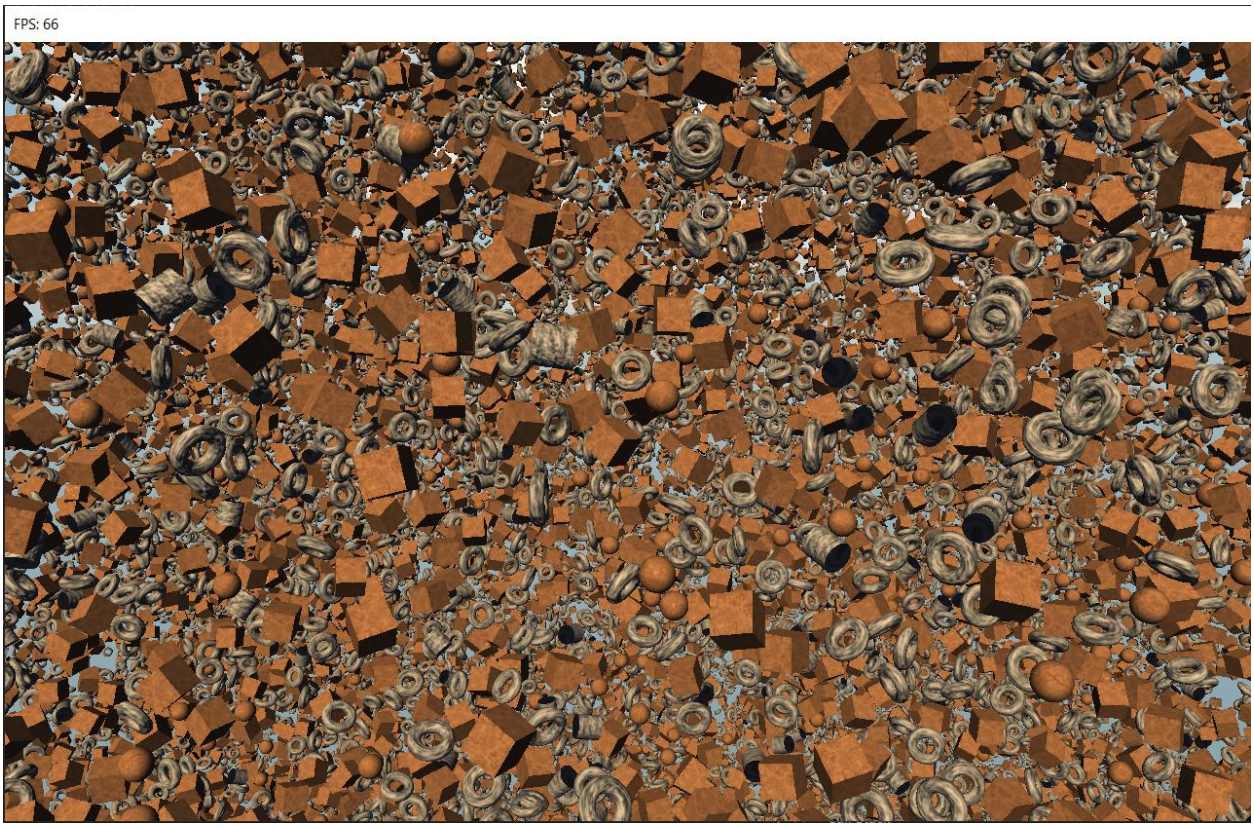

Fig 3 Test running at 66 FPS

However, the CPU Usage report shows a very concerning amount of work done by a fairly simple function, *Transform::GetWorldMatrix,* which is using over twice the total CPU time of *Mesh::Draw* and nearly **24 times** the Self CPU time, which clearly represents a performance bottleneck.

The reason for this is likely due to two things: The world matrix is recomputed from the individual position, rotation, and scale values whenever it is marked as dirty, which

implicitly dictates a branch is needed to check the member Boolean within *Transform* that denotes this (See Fig 5).

Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
Easel.exe (PID: 6472)	16142 (100.00%)	0 (0.00%)	Easel.exe
[External Code]	16142 (100.00%)	9574 (59.31%)	Multiple modules
__scrt_common_main_seh	11428 (70.80%)	0 (0.00%)	Easel.exe
System::GameWindow::RunGame	11428 (70.80%)	0 (0.00%)	Easel.exe
wWinMain	11428 (70.80%)	0 (0.00%)	Easel.exe
Game::Game::Render	10256 (63.54%)	1 (0.01%)	Easel.exe
Graphics::Renderer::Draw	10194 (63.15%)	192 (1.19%)	Easel.exe
Game::Transform::GetWorldMatrix	5183 (32.11%)	3899 (24.15%)	Easel.exe
Game::Mesh::Draw	2223 (13.77%)	212 (1.31%)	Easel.exe
DirectX::XMQuaternionRotationRollPitchYawFrom...	1189 (7.37%)	1189 (7.37%)	Easel.exe
Game::Game::Update	1081 (6.70%)	892 (5.53%)	Easel.exe
Input::GameInput::Frame	187 (1.16%)	0 (0.00%)	Easel.exe

Fig 4 Baseline CPU Usage

```
DirectX::XMFLOAT4X4 Transform::GetWorldMatrix()
{
using namespace DirectX;

if (mMatrixDirty) // Recalculate World Matrix?
{

    // Calculate individual pieces
    XMMATRIX translation =
        XMMatrixTranslationFromVector(XMLoadFloat3(&mPosition));

    XMMATRIX rotation =
        XMMatrixRotationRollPitchYawFromVector(XMLoadFloat3(&mPitchYawRoll));

    XMMATRIX scaling =
        XMMatrixScalingFromVector(XMLoadFloat3(&mScale));

    // Calculate world matrix
    XMMATRIX worldmat = scaling * rotation * translation;
    XMStoreFloat4x4(&mWorld, worldmat);

    // Switch flag
    mMatrixDirty = false;
}

return mWorld;
}
```

Fig 5 GetWorldMatrix Subroutine

Fixing the Transform Component - Plan

The main goals with this first correction were as follows:

- Remove branch in lieu of a less flexible, but more performant update pattern
- Reduce or eliminate the need for loading and storing DirectXMath constructs
- Store rotation as a quaternion rather than an Euler-angle based orthogonal matrix
- Maintain the ability to modify the position, rotation, and scale without incurring an automatic recompute of the world matrix.

Renderer::Update

```
std::vector<Game::Entity>::iterator it = mat1List->begin();
for (;it != mat1List->end(); ++it)
{
    it->mTransform.Rotate(rot1*dt);
}

it = mat2List->begin();
for (;it != mat2List->end(); ++it)
{
    it->mTransform.Rotate(rot2*dt);
}
```

Renderer::Render

```
std::vector<Entity>::iterator it = element.second.begin();
for(;it != element.second.end(); ++it)
{
    // Set Per Entity world matrix in the associated material's vertex shader.
    DirectX::XMFLOAT4X4 world = it->mTransform.Recompute();
    material->GetVertexShader()->SetBufferData(context,
              (UINT)ReservedRegisters::RR_VS_WORLDMATRIX, sizeof(world), &world);
    it->Draw(context);
}
```

Fig 6 Snippets of new update-draw code.

To this end, I've altered the *Transform* component in the following ways:

- Position, Scale, and Rotation members are now *XMVECTOR*s (a.k.a *__m128*s) rather than *XMFLOAT3*s. This was done to avoid the load/store calls required to do work with them and isn't a problem since they do not get sent to the GPU and thus do not need any special data formatting/alignment, unlike *mWorld*.
- Rotation is now stored as a quaternion rather than a roll-pitch-yaw vector and the camera rotation code has been updated accordingly.
- World Matrix is now public and deciding whether it's stale is handled externally by the public *Transform::Recompute* function. This was done to allow for explicit updating of the world matrix whenever convenient for cache and after all the translations, scales, and rotations are final.

```
DirectX::XMFLOAT4X4 Transform::Recompute()
{
    // Recompute the world matrix
    XMMATRIX tempWorld = XMMatrixAffineTransformation(mScale, XMVectorZero(),
       mQuatRotation, mPosition);
    XMStoreFloat4x4(&mWorld, tempWorld);

    // Return it.
    return mWorld;
}
```

Fig 7 The Recompute Function

Fixing the Transform - Results

These changes resulted in a roughly ~20 FPS increase from the previous run, with the application hovering at 84-88 FPS up from 66. However, fixing the reliance on *GetWorldMatrix* has confirmed a deeper flaw within the render cycle. Of the **~46%** of CPU time spent within *Renderer::Render*, **~25%** was just within a single expression, the updating of the constant buffer data with the already updated world matrix.

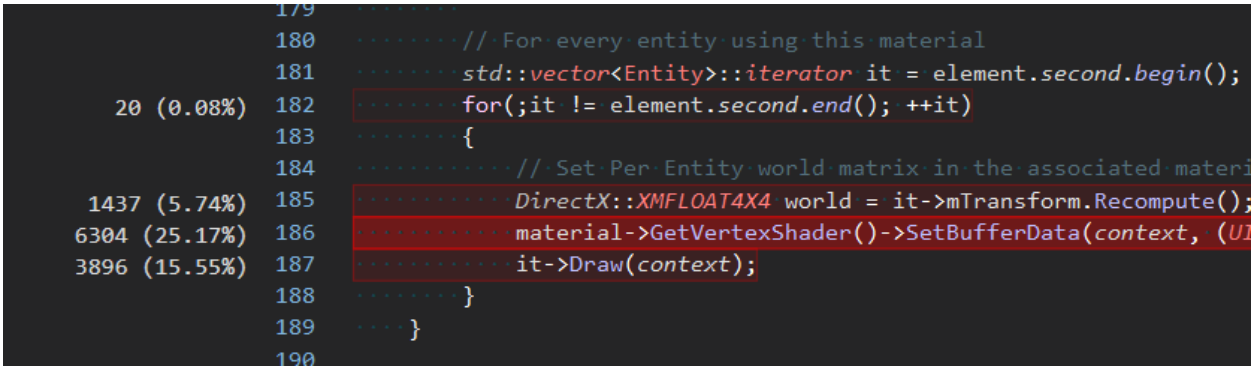

Fig 8 CPU Usage Per Expression within Renderer::Render (see Fig 6)

Next Steps:

The next and most essential overhaul to *Easel* must be the updating of the constant buffer architecture. This is the root cause of the issue seen in Fig 8, and much of the other issues described in Page 3. As such, the next major overhaul will be a reorganization of shader, constant buffer, and material representation and update pattern.