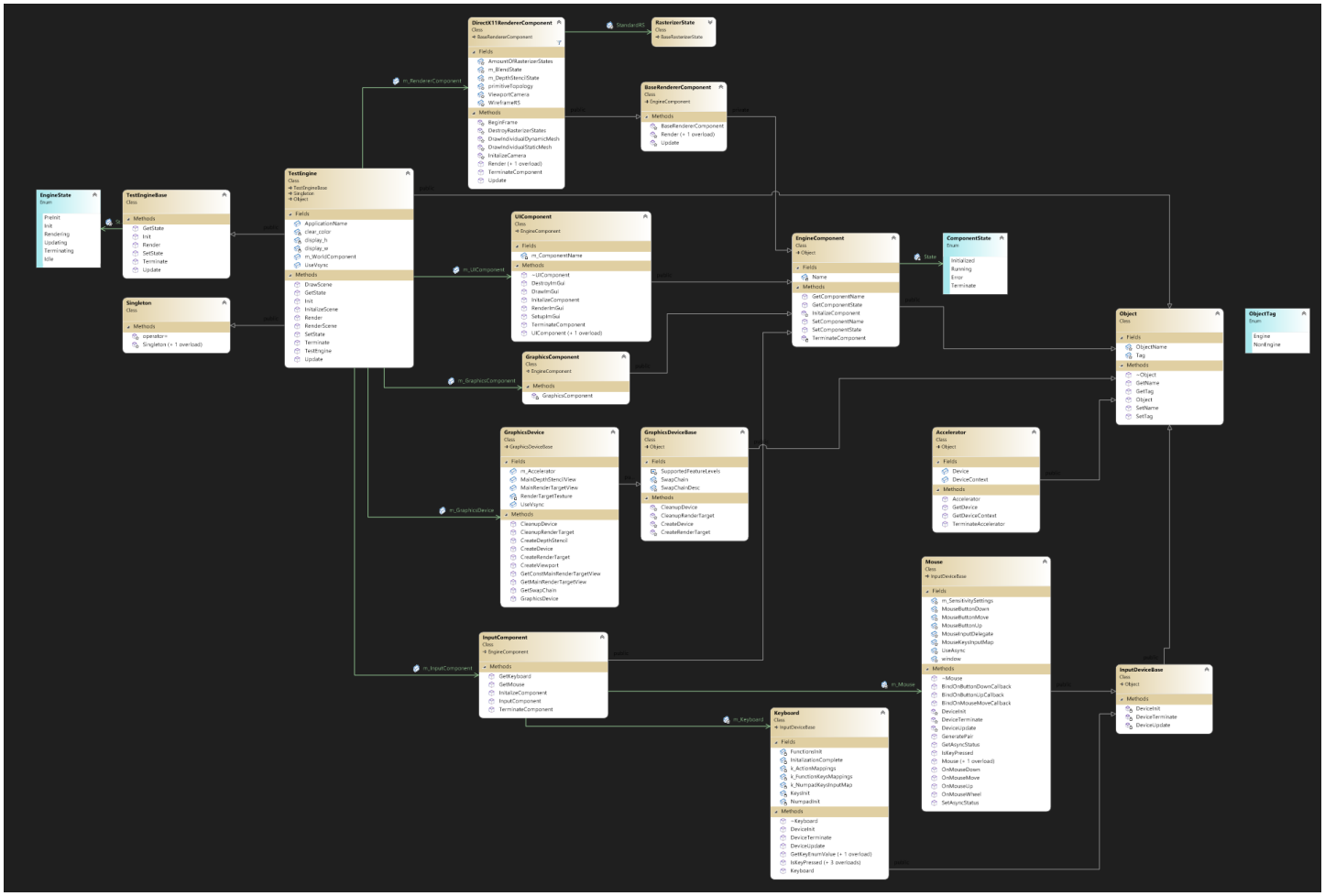
Morgan Ruffell
Bolt6.ai test – Documentation
Sunday, 12 June 2022

Test Engine – Documentation

This is my project that I developed over the past week. I'm not completely sure how I did, but this was extremely fun and allowed me to get myself stuck into a new SDK with Autodesk FBX.

First I planned out the engine architecture, I chose to have one major class that represented the engine, this would have a virtual parent class. Following an architecture similar to what I developed with Creative Engine. Where the base application could have a parent class with pure virtual methods that manage the initialization.




```cpp
namespace EngineObjTag
{
    enum ObjectTag
    {
        Engine,
        NonEngine,
    };
}


class Object
{
public:
    Object() { }

    ~Object()
    {
    }

    std::string GetName()
    {
        return ObjectName;
    }

    void SetName(std::string Name)
    {
        ObjectName = Name;
    }

public:

    EngineObjTag::ObjectTag GetTag()
    {
        return Tag;
    }

    void SetTag(EngineObjTag::ObjectTag TagType)
    {
        Tag = TagType;
    }

protected:

    std::string                 ObjectName;
    EngineObjTag::ObjectTag     Tag;
};
```

These classes would be added through raw pointers that are on the base engine class, all of the Engine components derive from the Engine Component class, that inherits from Object. All classes that are inside of Creative with some notable exceptions inherit from Object so that it can take advantage of the tagging behaviour in the base class.

Objects also have a name, making it easier to identify them. Although I only really utilize the tagging system inside of the constructors.

I designed this to be as extensible as possible, allowing for it to grow into something greater. I've also included this documentation to showcase what is happening in several of the major classes.

On an important note, I really struggled to get this to work in OpenGL. So I did this in DirectX11 as it was originally going to be in DirectX12 and I'd rather showcase something that I know I can get working, at least to a point in a short period of time.

Key Terms

As I did write quite a sizable amount of code and build quite a large project architecture, I'd like to just list some key terms that are used throughout this project and this documentation. As to the uninitiated it seems like a sizable chunk of work. I drew a lot of inspiration from commercial Engines like Unreal in the development of the Engine backend, except for the fact that unreal initializes parts of the engine in modules rather than in components inside it's *build.cs* file.

This solution is nowhere near as large as unreal, so it utilizes a significantly smaller architecture. With a greater level of elements specialized to the demonstration of animation.

Static Mesh:

Simply the term for a loaded mesh that does not have any deformers, or any animations loaded.

Skeleton:

A data object that contains a dynamic array of bones, as well as an individual parent bone that is loaded from a DirectXMath Position (Usually a XMMatrix or similar data structure that includes Euclidean position data)

Dynamic Mesh :

Mesh that contains animation data and relies upon an array of several animation data structs to contain information about the animation bound to its skeleton.

Vertex Shader

Not to be confused for the HLSL file 'that shares the same name, inside this engine this is the object that controls the vertex shader it contains a ID3D11VertexShader object. It shares a similar inheritance hierarchy to its pixel shader cousin. I am only using these two due to time, given more time, I'd integrate geometry shaders and work on a per-primitive basis.

Singleton

A design pattern that locks down copying and moving of the object, a lot of elements derive from the singleton base class.

Animation Structures

There are three animation structures inside of my solution,

- **Animation Clips** which contain data about the animation, again I'm borrowing small amounts of names from Unreal to make it easier for those familiar with Unreal.
- **Animation Composition** This contains two dynamic arrays, one for the names and one for the Clips. This ideally will be a hashmap in future.
- **Animation Sequence** This contains all of the information about an animation and lives inside a dynamic mesh where it is paired with a skeleton.

TESamplerState

This contains a ID3D11SamplerState and is initialized within a Texture, allowing for specialized rendering of types

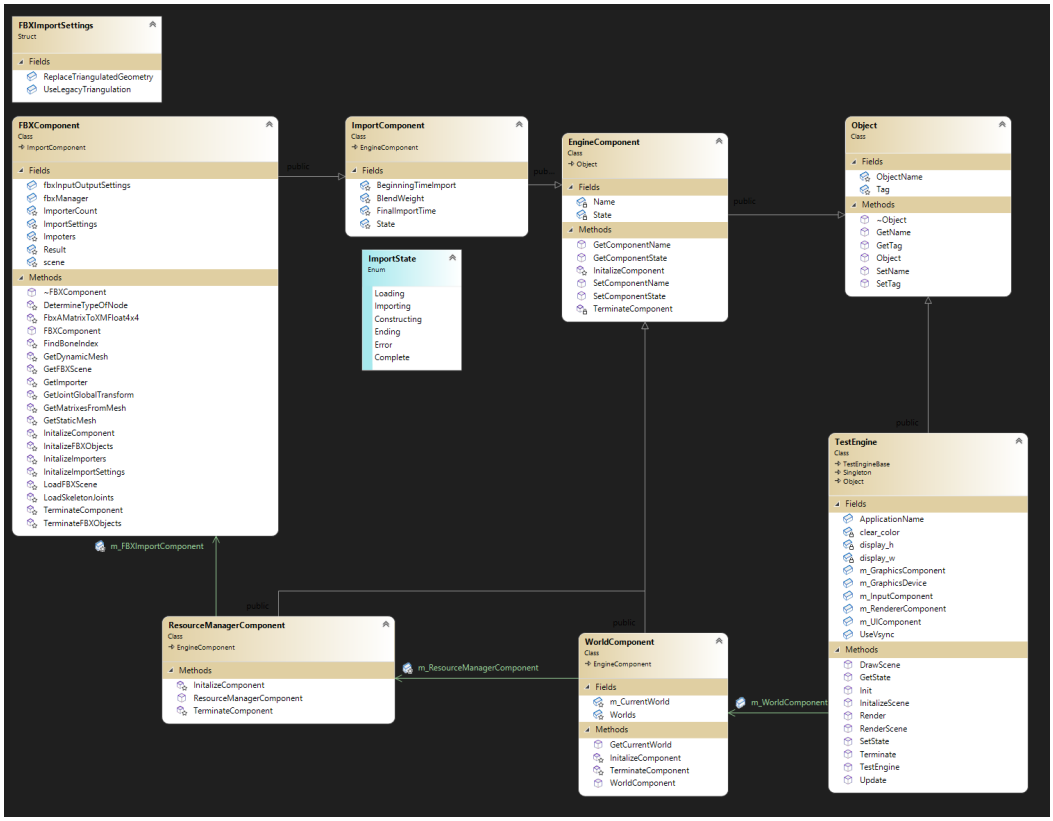
Textures & Materials

Textures live inside a material, there are normally two, one for Base Colours and for Normal Maps. These are accessed through getters and setters protecting the underlying pointer through encapsulation.

Morgan Ruffell
Bolt6.ai test – Documentation
Sunday, 12 June 2022

FBX Component

Although it's ambitious for a week, I really wanted to dig into the .FBX SDK as it was something I wanted to do with my undergraduate studies.



In order to try and make this work smoothly, as well as create everything in this component based workflow, I decided to attach these to a resource manager component that manages all of the resources inside a world. A World component manages the world that is drawn on the screen.

This uses the FbxSDK from 2016, there are implementations available in public software but I wanted to dig into this and create something bespoke. I used 2016 because it will be more stable than the latest versions.

```
class FBXComponent : public ImportComponent
{
public:
    FBXComponent(){ ... }

    ~FBXComponent(){ ... }

public:

    fbxsdk::FbxManager* fbxManager;
    fbxsdk::FbxIOSettings* fbxInputOutputSettings;

protected:

    void InitalizeFBXObjects();
    void TerminateFBXObjects();

    void GetFBXScene(fbxsdk::FbxString name, fbxsdk::FbxString Filepath);

protected:
    void InitalizeImportSettings();
    void InitalizeImporters(const char* Filename);

    fbxsdk::FbxNodeAttribute::EType DetermineTypeOfNode(fbxsdk::FbxNode* Node);

protected:

    bool LoadFBXScene(_In_ const char* Filename, FbxScene* Scene, World* world);
    void LoadSkeletonJoints(_In_ ::FbxNode*, _Inout_ Skeleton* s_kl);
    StaticMesh* GetStaticMesh(_Inout_::FbxNode* Node, _In_  Accelerator* _accel);

protected:

    //These two function similar to Unreal, Dynamic Meshes have a skeleton.
    DynamicMesh* GetDynamicMesh(_Inout_ ::FbxNode* Node, _In_ Accelerator* _accel);


    int FindBoneIndex(const std::string& name, std::vector<Bone2*>& BoneCollection);

protected:

    void GetMatrixesFromMesh(_Inout_ ::FbxNode* Node, _In_ Accelerator* _accel, _In_ std::vector<Socket>&);

    XMFLOAT4X4 GetJointGlobalTransform(int, _In_ std::vector<Socket>& Collection);
    XMFLOAT4X4 FbxAMatrixToXMFloat4x4(FbxAMatrix Matrix);

    FbxImporter* GetImporter();

    FBXImportSettings ImportSettings;

protected:

    FbxScene* scene;
    std::map<fbxsdk::FbxStatus,FbxImporter*> Importers;

protected:

    int ImporterCount = 3;

    bool Result;

    virtual void InitalizeComponent();
    virtual bool TerminateComponent();
};
```

Most recent is linked here for your reference:

https://www.autodesk.com/developer-network/platform-technologies/fbx-sdk-2020-2-1

```cpp
void FBXComponent::InitalizeFBXObjects()
{
    fbxManager = FbxManager::Create();

    if (!fbxManager)
    {
        std::cout << "FBX Manager Initalization Failed" << std::endl;
        return;
    }

    fbxInputOutputSettings = FbxIOSettings::Create(fbxManager, IOSROOT);
    fbxManager->SetIOSettings(fbxInputOutputSettings);

    //Until I know what these are, just lock them away..
#ifdef USE_EXTERNAL_FBX_PLUGINS

    FbxString LoadingPath = FbxGetApplicationDirectory();
    assert(LoadingPath.IsEmpty() != true);
    fbxManager->LoadPluginsDirectory(LoadingPath.Buffer());

#endif // USE_EXTERNAL_FBX_PLUGINS

    if (scene != nullptr)
    {
        std::cout << "You cannot initalize these components, there is already a scene loaded" << std::endl;
        std::cout << "Only one scene is supported right now" << std::endl;
        return;
    }
    else
    {
        assert(fbxInputOutputSettings != nullptr);

        scene = FbxScene::Create(fbxManager, "FbxSceneAlpha");
        fbxManager->SetIOSettings(fbxInputOutputSettings);

        if (!scene)
        {
            std::cout << "Unable To Initalize FBX Scene. Terminating..." << std::endl;
        }

    }
}
```

This engine, first initializes the FBX object this method is called from the class constructor, which contains the initialize component method which is a pure virtual method inside of all *EngineComponent* objects/classes.

```cpp
void FBXComponent::InitalizeImportSettings()
{
    ImportSettings.ReplaceTriangulatedGeometry = true;
    ImportSettings.ReplaceTriangulatedGeometry = false;
}

void FBXComponent::InitalizeImporters(const char* Filename)
{
    assert(fbxManager != nullptr);

    FbxImporter* PrimaryImporter = FbxImporter::Create(fbxManager, Filename);

#ifdef USE_REDUDANT_LOADING

    FbxImporter* SecondaryImporter = FbxImporter::Create(fbxManager, Filename);
    FbxImporter* TertiaryImporter = FbxImporter::Create(fbxManager, Filename);

#endif // USE_REDUDANT_LOADING

    PrimaryImporter->Initialize(Filename, -1, fbxManager->GetIOSettings());

#ifdef USE_REDUDANT_LOADING

    SecondaryImporter->Initialize(Filename, -1, fbxManager->GetIOSettings());
    TertiaryImporter->Initialize(Filename, -1, fbxManager->GetIOSettings());

    fbxsdk::FbxStatus SStatus = SecondaryImporter->GetStatus();
    fbxsdk::FbxStatus TStatus = TertiaryImporter->GetStatus();

#endif

    fbxsdk::FbxStatus PStatus = PrimaryImporter->GetStatus();


    if (PStatus == fbxsdk::FbxStatus::EStatusCode::eSuccess)
    {
        std::cout << "Fbx Primary Import Status is Successful" << std::endl;
    }

#ifdef USE_REDUDANT_LOADING

    if (SStatus == fbxsdk::FbxStatus::EStatusCode::eSuccess)
    {
        std::cout << "Fbx Primary Import Status is Successful" << std::endl;
    }

    if (TStatus == fbxsdk::FbxStatus::EStatusCode::eSuccess)
    {
        std::cout << "Fbx Primary Import Status is Successful" << std::endl;
    }

#endif

    Importers.insert({ PStatus, PrimaryImporter });

#ifdef USE_REDUDANT_LOADING

    Importers.insert({ SStatus, SecondaryImporter });
    Importers.insert({ TStatus, TertiaryImporter });

#endif

}
```

It will initialize the importer objects these are used to import a .FBX object these are initialized into a hashmap with their status as the key, meaning you can always get an importer that works in case of redundancy. FBX manages a lot of it's files itself through the *FbxManager* and the *FbxIOSettings.*

When loading a FBX file it's comprised of a series of *Nodes*, composed into a *Scene*, similar to that of a graph in computer science. These nodes are all linked together and are assigned a type by the FBX file 'creator' inside of Autodesk Maya, Max or whichever other software you are using to generate FBX files.

These nodes are composed into a *FbxScene* Object, I chose to get the exact version of this as well as set up all of the input settings allowing for links (These link individual nodes together)

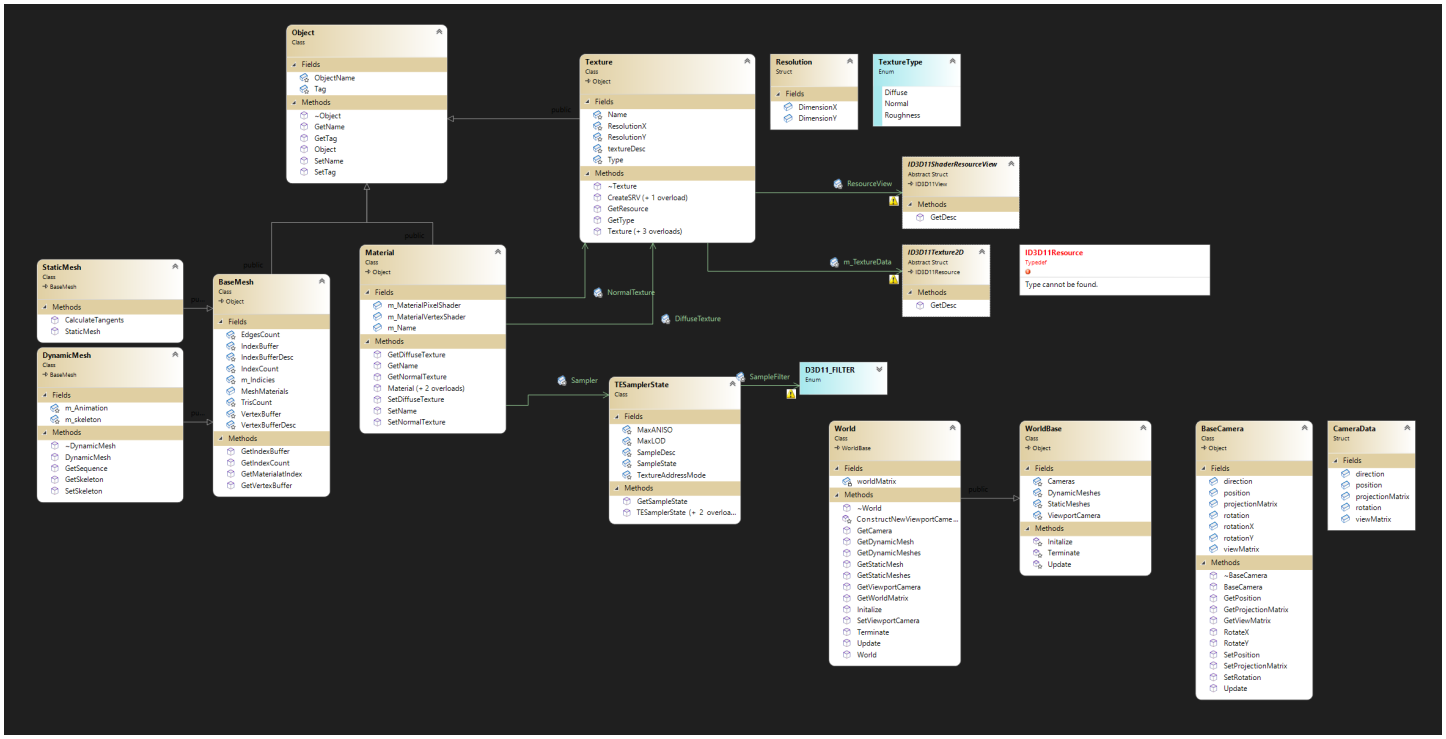
```
bool FBXComponent::LoadFBXScene(const char* Filename, FbxScene* Scene, World* world)
{
    int lFileMajor, lFileMinor, lFileRevision;
    int lSDKMajor, lSDKMinor, lSDKRevision;

    bool Result;

    FbxManager::GetFileFormatVersion(lSDKMajor, lSDKMinor, lSDKRevision);

    InitalizeImporters(Filename);

    std::cout << "FBX file format for this is FBX SDK" << lSDKMajor << "." << lSDKMinor << std::endl;

    auto Importer = GetImporter();

    if (Importer->IsFBX())
    {
        fbxInputOutputSettings->SetBoolProp(IMP_FBX_MATERIAL, true);
        fbxInputOutputSettings->SetBoolProp(IMP_FBX_TEXTURE, true);
        fbxInputOutputSettings->SetBoolProp(IMP_FBX_LINK, true);
        fbxInputOutputSettings->SetBoolProp(IMP_FBX_SHAPE, true);
        fbxInputOutputSettings->SetBoolProp(IMP_FBX_GOBO, true);
        fbxInputOutputSettings->SetBoolProp(IMP_FBX_ANIMATION, true);
        fbxInputOutputSettings->SetBoolProp(IMP_FBX_GLOBAL_SETTINGS, true);
    }

    Result = Importer->Import(Scene);
    Importer->ContentUnload();

    return Result;
}
```

Because of the way that I've architected the systems inside of this project, where the *Skeleton Object* is separate from the *Dynamic Mesh* which is a mesh that contains an animation

```
FbxNodeAttribute::EType FBXComponent::DetermineTypeOfNode(fbxsdk::FbxNode* Node)
{
    return Node->GetNodeAttribute()->GetAttributeType();
}

//Recursive method that gets all of the bones and loads them into a skeleton on a dynamic mesh
void FBXComponent::LoadSkeletonJoints(fbxsdk::FbxNode* Node, Skeleton* s_kl)
{
    if (DetermineTypeOfNode(Node) == FbxNodeAttribute::EType::eSkeleton)
    {
        if (s_kl == nullptr)
        {
            fbxsdk::FbxAMatrix BonePos;
            fbxsdk::FbxAMatrix GlobalMatrix;

            fbxsdk::FbxString Name = Node->GetName();
            fbxsdk::FbxVector4 Translation = Node->GetGeometricTranslation(FbxNode::eSourcePivot);
            fbxsdk::FbxVector4 Rotation = Node->GetGeometricRotation(FbxNode::eSourcePivot);
            fbxsdk::FbxVector4 Scale = Node->GetGeometricScaling(FbxNode::eSourcePivot);

            GlobalMatrix = Node->EvaluateGlobalTransform();

            BonePos.SetT(Translation);
            BonePos.SetR(Rotation);
            BonePos.SetS(Scale);

            fbxsdk::FbxAMatrix FinalMatrix = GlobalMatrix * BonePos;

            const char* stdName = Name;

            Bone2* Bone = new Bone2(stdName, FbxAMatrixToXMFloat4x4(FinalMatrix), Parent);
            s_kl = new Skeleton(Bone);
            s_kl->mBones.push_back(Bone);
        }
    }

    int childCount = Node->GetChildCount();

    for (int i = 0; i < childCount; i++)
    {
        LoadSkeletonJoints(Node->GetChild(i), s_kl);
    }
}
```

Meshes, Worlds & Resource Managers

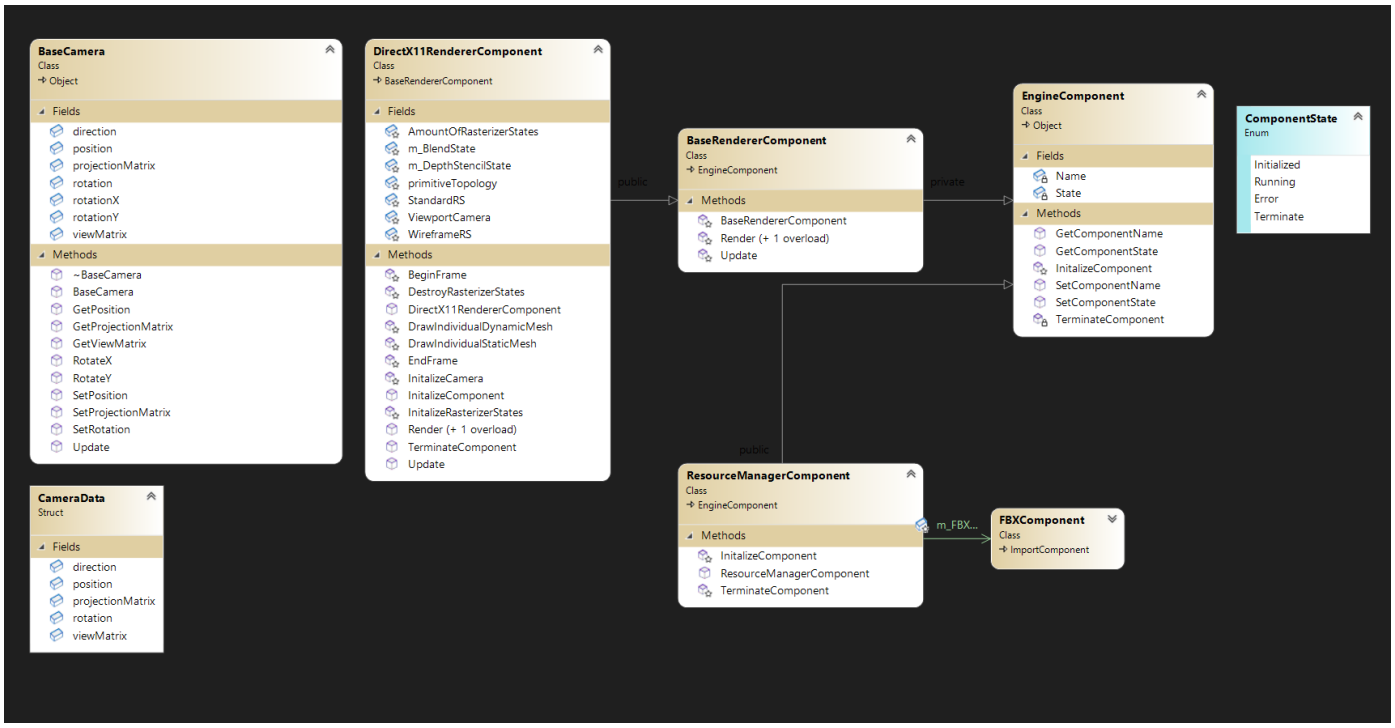
In order to render something, I felt it prudent to build up a solution that is easily extensible and demonstrates that software needs to be able to react to the changes in the business environment quickly.



A world is like a huge container of information, it contains information about the location in memory of everything that will be rendered by the *BaseCamera* instance in renderer component. I've included these objects inside of the class diagram for your reference.

Some information about the skeleton and animation has been omitted.

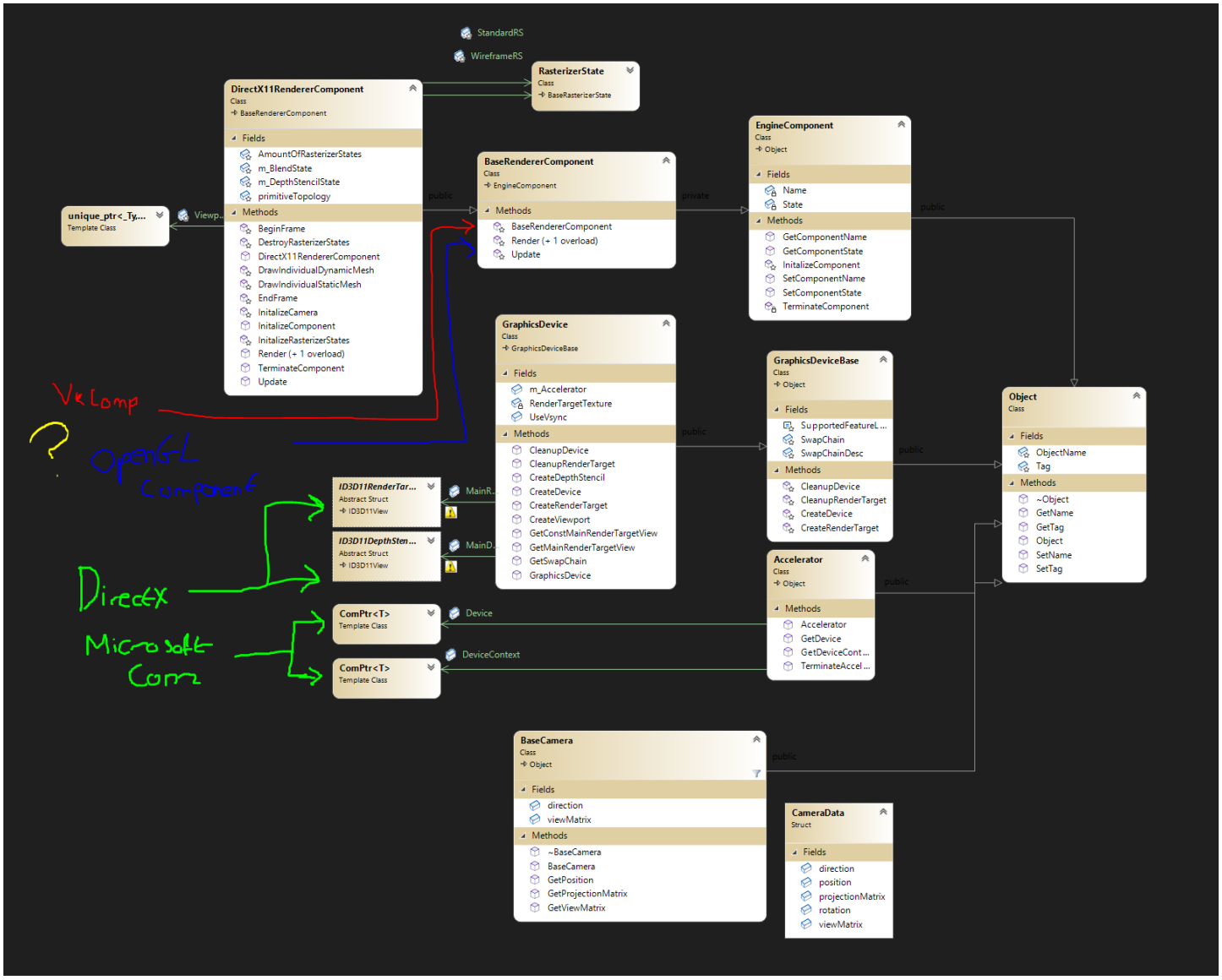
The renderer component will use its Base Camera to render the objects along with their position data to the screen.



Information pertaining to the everything inside of the world is managed by the resource manager component.

Rendering

Rendering relies of three components, A derivative of the *Base Renderer Component,* which in this case is *DirectX11 RendererComponent* although due to the way this is structured this could very easily be adapted to work with another slightly higher Graphics Library such as *OpenGL*.



These are all of the objects involved in the rendering process, I've shown you some on the previous page. But I'm introducing you to the Graphics Device, which acts as a wrapper which contains the *IDXGISwapChain* (The DirectX equivalent of the *SwapBuffers()* method) This class also contains the Accelerator, a custom class that contains the *ID3D11Device* and the *ID3D11DeviceContext* These are similar in nature to the *VkPhysicalDevice* and *VkPhysicalDeviceProperties* if you have any previous experience with Vulkan.

Textures

```cpp
//Similar to descriptor structs, this functions in a similar way for our textures.
struct TextureContext
{
    LPCWSTR             TextureFilename;
    Resolution          Resolution;
    TextureType         Type;
    bool                AllowOverwriting;
};

class Texture : public Object
{
public:
    Texture(float _DimensionX, float _DimensionY, TextureType _Type, std::string& _Name, Accelerator* _Accel, LPCWSTR File)
    {
        assert(_Accel != nullptr);

        SetTag(EngineObjTag::Asset);

        ResolutionX = _DimensionX;
        ResolutionY = _DimensionY;
        Type = _Type;
        ResourceView = CreateSRV(_Accel, _DimensionX, _DimensionY);
        Name = _Name;

        ID3D11Resource* Resource;

        CreateWICTextureFromFile(_Accel->GetDevice(), _Accel->GetDeviceContext(), File, &Resource, &ResourceView, 4096);

        m_TextureData = reinterpret_cast<ID3D11Texture2D*>(Resource);


        std::cout << "Texture initalization succeeded!" << std::endl;
    }

    Texture(Resolution Res, TextureType _Type, std::string& _Name, Accelerator* _Accel, LPCWSTR File)
    {
        assert(_Accel != nullptr);
        assert(Res.DimensionX != 0 && Res.DimensionY != 0);

        SetTag(EngineObjTag::Asset);

        ResolutionX = Res.DimensionX;
        ResolutionY = Res.DimensionY;
        Type = _Type;
        ResourceView = CreateSRV(_Accel, Res);
        Name = _Name;

        ID3D11Resource* Resource;

        CreateWICTextureFromFile(_Accel->GetDevice(), _Accel->GetDeviceContext(), File, &Resource, &ResourceView, 4096);

        m_TextureData = reinterpret_cast<ID3D11Texture2D*>(Resource);

        std::cout << "Texture initalization succeeded!" << std::endl;

    }

    Texture(TextureContext context, std::string& _Name, Accelerator* _Accel)
    {
        assert(_Accel != nullptr);

        SetTag(EngineObjTag::Asset);

        ResolutionX = context.Resolution.DimensionX;
        ResolutionY = context.Resolution.DimensionY;
        Type = context.Type;
        ResourceView = CreateSRV(_Accel, context.Resolution);
        Name = _Name;

        ID3D11Resource* Resource;
        CreateWICTextureFromFile(_Accel->GetDevice(), _Accel->GetDeviceContext(), context.TextureFilename, &Resource, &ResourceView, 4096);
        m_TextureData = reinterpret_cast<ID3D11Texture2D*>(Resource);

        std::cout << "Texture initalization succeeded!" << std::endl;
    }
```

These are how textures are initialized and stored in memory.

Textures in DirectX11 are a combination of ID3D11Texture2D (In DirectX12 this is an ID3D12Resource, I use the DirectX11 version as ID3D11Texture2D is derived from it). And a ID3D11ShaderResourceView, this acts as a handle to the texture.

```cpp
ID3D11ShaderResourceView* CreateSRV(Accelerator* Device, float _DimensionX, float _DimensionY)
{
    //textureDesc

    ZeroMemory(&textureDesc, sizeof(textureDesc));

    textureDesc.Width = _DimensionX;
    textureDesc.Height = _DimensionY;
    textureDesc.MipLevels = 1;
    textureDesc.ArraySize = 1;
    textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    textureDesc.SampleDesc.Count = 1;
    textureDesc.SampleDesc.Quality = 0;
    textureDesc.Usage = D3D11_USAGE_DEFAULT;
    textureDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
    textureDesc.CPUAccessFlags = 0;
    textureDesc.MiscFlags = 0;

    HRESULT TextureRes = Device->GetDevice()->CreateTexture2D(&textureDesc, nullptr, &m_TextureData);


    if (SUCCEEDED(TextureRes))
    {
        D3D11_SHADER_RESOURCE_VIEW_DESC shaderResourceViewDesc;
        shaderResourceViewDesc.Format = textureDesc.Format;
        shaderResourceViewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
        shaderResourceViewDesc.Texture2D.MostDetailedMip = 0;
        shaderResourceViewDesc.Texture2D.MipLevels = 1;

        HRESULT SRV = Device->GetDevice()->CreateShaderResourceView(m_TextureData, &shaderResourceViewDesc, &ResourceView);

        if (SUCCEEDED(SRV))
        {
            std::cout << "Texture initalization succeeded!" << std::endl;
            return ResourceView;
        }
    }
    else
    {
        throw new std::exception("Texture Failed to initalize");
    }


}

ID3D11ShaderResourceView* CreateSRV(Accelerator* Device, Resolution res)
{
    //textureDesc

    ZeroMemory(&textureDesc, sizeof(textureDesc));

    textureDesc.Width = res.DimensionX;
    textureDesc.Height = res.DimensionY;
    textureDesc.MipLevels = 1;
    textureDesc.ArraySize = 1;
    textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    textureDesc.SampleDesc.Count = 1;
    textureDesc.Usage = D3D11_USAGE_DEFAULT;
    textureDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
    textureDesc.CPUAccessFlags = 0;
    textureDesc.MiscFlags = 0;

    //Fix this as we are using the resource directly and not texture 2D
    HRESULT TextureRes = Device->GetDevice()->CreateTexture2D(&textureDesc, 0, &m_TextureData);

    if (SUCCEEDED(TextureRes))
    {
        D3D11_SHADER_RESOURCE_VIEW_DESC shaderResourceViewDesc;
        shaderResourceViewDesc.Format = textureDesc.Format;
        shaderResourceViewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
        shaderResourceViewDesc.Texture2D.MostDetailedMip = 0;
        shaderResourceViewDesc.Texture2D.MipLevels = 1;

        HRESULT SRV = Device->GetDevice()->CreateShaderResourceView(m_TextureData, &shaderResourceViewDesc, &ResourceView);

        if (SUCCEEDED(SRV))
        {
            std::cout << "Texture initalization succeeded!" << std::endl;
            return ResourceView;
        }
    }

    return nullptr;

}
```

HLSL Shaders

There are two HLSL shaders in my solution, keeping it as simple as possible. There is no need for mesh tessellation or per geometry or anything bleeding edge like *barycentrics* which are inside of Creative.

```
//Morgan Ruffell - 2022
//Base Pixel Shader
//HLSL - SM5.1 Pixel Shader

Texture2D diffuseTexture        : register(t0);
Texture2D normalTexture         : register(t1);

SamplerState PixelSampler       : register(s0);

bool UseNormals = true;

struct VertexToPixel
{
    float4 position     : SV_POSITION;
    float3 normal       : NORMAL;
    float2 uv           : TEXCOORD;
    float3 worldPos     : POSITION;
    float3 tangent      : TANGENT;
};

struct NormalParams
{
    float UnpackExponent = 2.1f;
    float UnpackSubtractor = 1.0f;
};

float3 calculateNormalFromMap(float2 uv, float3 normal, float3 tangent)
{
    NormalParams params;

    float3 normalFromTexture = normalTexture.Sample(PixelSampler, uv).xyz;

    float3 unpackedNormal = normalFromTexture * params.UnpackExponent - 1.0f;

    float3 ComputedNormal = normal;
    float3 Tangent = normalize(tangent - ComputedNormal * dot(tangent, ComputedNormal));
    float3 Binormal = cross(ComputedNormal, Tangent);


    float3x3 TBN = float3x3(Tangent, Binormal, ComputedNormal);

    return normalize(mul(unpackedNormal, TBN));
}

float4 main(VertexToPixel input) : SV_TARGET
{
    float4 Result;

    float4 BaseColour = diffuseTexture.Sample(PixelSampler, input.uv);

    if(UseNormals == true)
    {
        float3 ComputedNormal = calculateNormalFromMap(input.uv, input.normal, input.tangent);

        input.normal = normalize(input.normal);
        ComputedNormal = normalize(ComputedNormal);

        Result = BaseColour;

        Result.x + ComputedNormal.x;
        Result.y + ComputedNormal.y;
        Result.z + ComputedNormal.z;

        return Result;
    }
    else
    {
        input.normal = normalize(input.normal);
        Result = BaseColour;
        return Result;
    }

    //If displaying normals fails just draw base colour
    input.normal = normalize(input.normal);
    Result = BaseColour;
    return Result;
}
```

This is the pixel shader that I used based on knowledge of the rendering pipeline, it contains options for normal displays.

Textures are stored in Texture2D objects at specific GPU registers. The text in yellow as well as SV_POSITION are shader semantics, and link with the C++ Code in ID3D11.

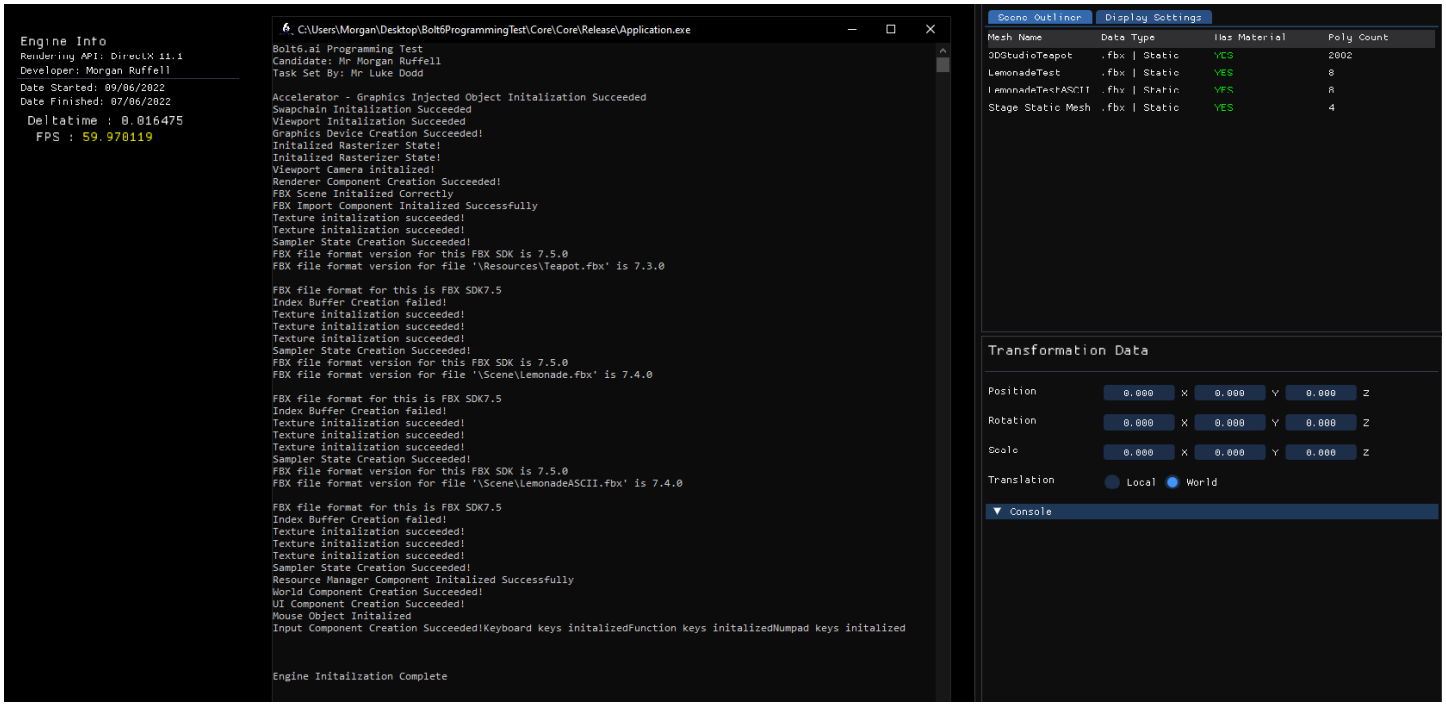
The implementation is similar to that of GLSL, we just do not have to dictate the versioning. These are SM 5.1. I've linked the documentation for your reference https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl

Underneath is the code for the Vertex shader, I've left code comments detailing what specific lines are doing. I'll make modifications to this to get it working but I hope it details the general idea behind translations, and rotations on a per vertex basis in response to changes in C++ code inside the engine

```
//Morgan Ruffell - 2022
//Base Vertex Shader - Animation
//HLSL - SM5.1 Vertex Shader

int ShaderMaximumOfActiveBones = 30;

struct VertexShaderInput
{
    float4 position     : POSITION;
    float3 normal       : NORMAL;

    float4 boneid       : BONEID;
    float4 weight       : WEIGHT;

    float2 uv           : TEXCOORD;
    float2 uv1          : TEXCOORD1;

    float3 tangent      : TANGENT;
};

//Following the render pipeline, at it's most primitive
//This goes to the pixel shader, you can have tessalation options
//but we're going for simplicity.
struct VertexToPixel
{
    float4 position     : SV_POSITION;
    float3 normal       : NORMAL;

    float2 uv           : TEXCOORD;
    float2 uv1          : TEXCOORD1;

    float3 worldPos     : POSITION;
    float3 tangent      : TANGENT;
};

//This must match the inside of the C++ struct or it will not compile
//as we are sending that data to the GPU in here.
struct Bone
{
    matrix BoneTransform;
    matrix InvBoneTransform;
    int NumberOfChildren;
    int MaximumNumberOfChildBones = ShaderMaximumOfActiveBones;
};

cbuffer WorldData : register(b0)
{
    matrix world;
    matrix view;
    matrix projection;
};

//This is that same data mapped to a GPU registry -- These are 16 bytes aligned
cbuffer bones : register(b1)
{
    Bone bones[40];
    float blendWeight; //The blending of transformation per bone
}
```

```
VertexToPixel main(VertexShaderInput input)
{
    VertexToPixel output;

    matrix worldViewProj = mul(mul(world, view), projection);
    matrix bonetransform = 0;

    if(!bones[0].NumberOfChildren > ShaderMaximumOfActiveBones)
    {
        //These scales the vertex bound to a bone through a blendshape by the position in the bonetransform matrix.
        if (input.boneid.x != -1)
        {
            bonetransform = mul(mul(bones[input.boneid.x].BoneTransform, input.weight.x), bones[input.boneid.x].InvBoneTransform) * blendWeight;
        }

        if (input.boneid.y != -1)
        {
            bonetransform += mul(mul(bones[input.boneid.y].BoneTransform, input.weight.y), bones[input.boneid.y].InvBoneTransform)* blendWeight;
        }

        if (input.boneid.z != -1)
        {
            bonetransform += mul(mul(bones[input.boneid.z].BoneTransform, input.weight.z), bones[input.boneid.z].InvBoneTransform)* blendWeight;
        }

        output.position = mul(mul(bonetransform, input.position), worldViewProj); //We have to multiply the vertexes position in 3D space by the world view projection matrix
        output.normal = mul((float3)mul(bonetransform, float4(input.normal,1.0)), (float3x3)world);
        output.worldPos = mul(input.position, world).xyz;

        output.uv = input.uv;
        output.uv1 = input.uv1;

        output.tangent = normalize(mul((float3)mul(bonetransform, float4(input.tangent,2.0)), (float3x3)world));

        return output;
    }

    //If there are too many bones for the shader to handle, which I've limited because of the GPU requirements, then just apply minimal operations per vertex
    output.position  = input.position; //We have to multiply the vertexes position in 3D space by the world view projection matrix
    output.normal    = input.normal;
    output.worldPos  = mul(input.position, world).xyz;

    output.uv        = input.uv;
    output.uv1       = input.uv1;

    output.tangent   = input.tangent;

    return output;
}
```

Output Log




```cpp
StaticMesh* FBXComponent::CreateStaticMesh(fbxsdk::FbxNode* Node, Accelerator* _accel)
{
    assert(fbxManager != nullptr);
    assert(_accel != nullptr);

    fbxsdk::FbxString Name = Node->GetName();
    fbxsdk::FbxGeometryConverter converter(fbxManager);

    converter.Triangulate(scene, ImportSettings.ReplaceTriangulatedGeometry, ImportSettings.UseLegacyTriangulation);

    if (!DetermineTypeOfNode(Node)->GetAttributeType() == FbxNodeAttribute::EType::eMesh)
    {
        return nullptr;
    }

    std::vector<Vertex1> Vertexes;
    std::vector<UINT> Indicies;

    fbxsdk::FbxMesh* Mesh = (fbxsdk::FbxMesh*)Node->GetNodeAttribute();
    FbxVector4* ControlPoints = Mesh->GetControlPoints();
    UINT VertexCount = Mesh->GetControlPointsCount();

    for (unsigned int i = 0; i < VertexCount; i++)
    {
        Vertex1 v;

        v.Position.x = ControlPoints[i].mData[0];
        v.Position.y = ControlPoints[i].mData[1];
        v.Position.z = ControlPoints[i].mData[2];

        v.Normal = XMFLOAT3(0, 0, 0);
        Vertexes.push_back(v);
    }

    int PolygonCount = Mesh->GetPolygonCount();
    int PolygonSize = Mesh->GetPolygonGroup(0);
    UINT indexCount = PolygonCount * PolygonSize;

    for (int i = 0; i < Mesh->GetPolygonCount(); i++)
    {
        for (int j = 0; j < Mesh->GetPolygonSize(i); j++)
        {
            int Individual_Indicie;

            Individual_Indicie = Mesh->GetPolygonVertex(i, j);
            Indicies.push_back(Individual_Indicie);

            FbxVector4 norm(0, 0, 0, 0);

            Mesh->GetPolygonVertexNormal(i, j, norm);

            Vertexes[Individual_Indicie].Normal.x += (float)norm.mData[0];      // Vertex Normals
            Vertexes[Individual_Indicie].Normal.y += (float)norm.mData[1];
            Vertexes[Individual_Indicie].Normal.z += (float)norm.mData[2];

            FbxVector2 PolygonUVCoordinates(0, 0);

            const char* uvSet = "FILENAME DELETE WHEN LOADING TEXTURES TO MATERIALS";
            bool UVFlag = true;

            Mesh->GetPolygonVertexUV(i, j, uvSet, PolygonUVCoordinates, UVFlag);
        }
    }

    std::string MeshName = Node->GetName();

    StaticMesh* SM = new StaticMesh(&Vertexes[0], VertexCount, &Indicies[0], indexCount, _accel, MeshName);

    //SM->CalculateTangents(&Vertexes[0], VertexCount, &Indicies[0], indexCount);

    return SM;
}
```

This is the code that manages the static mesh import process on top, Dynamic Mesh import process on the right.

The output log gives you information about what is going on internally.

This builds upon what I talked about in the previous chapter, and imports Static Meshes into the world. Ready for them to be rendered.

```cpp
int NumberOfDeformers = Mesh->GetDeformerCount();

if (NumberOfDeformers == 0) { return new DynamicMesh(DynamicMeshSkeleton, &Vertexes[0], VertexCount, &Indicies[0], indexCount, _accel); }

for (int deformerIndex = 0; deformerIndex < NumberOfDeformers; deformerIndex++)
{
    fbxsdk::FbxSkin* skin = reinterpret_cast<FbxSkin*>(Mesh->GetDeformer(0, fbxsdk::FbxDeformer::eSkin));
    int NumberOfSkinClusters = skin->GetClusterCount();

    //Enumerate all of the skin clusters inside of the mesh, skin clusters act as a subset of a geometry control points -- Similar to a geometry shader
    for (int indexOfSkinCluster = 0; indexOfSkinCluster < NumberOfSkinClusters; ++NumberOfSkinClusters)
    {
        //Get the current cluster and it's link mode.
        fbxsdk::FbxCluster* CurrentCluster = skin->GetCluster(indexOfSkinCluster);
        fbxsdk::FbxCluster::ELinkMode JointLinkMode = CurrentCluster->GetLinkMode();

        std::string CurrentJointName = CurrentCluster->GetLink()->GetName();

        int CurrentBoneIndex = FindBoneIndex(CurrentJointName, DynamicMeshSkeleton->mBones);

        //Three matrixes, one for the transform of the cluster (the cluster of control points)
        fbxsdk::FbxAMatrix            transformMatrix;
        fbxsdk::FbxAMatrix            transformLinkMatrix;
        fbxsdk::FbxAMatrix            globalBindposeInverseMatrix;

        CurrentCluster->GetTransformMatrix(transformMatrix);
        CurrentCluster->GetTransformLinkMatrix(transformLinkMatrix);
        globalBindposeInverseMatrix = transformLinkMatrix.Inverse();

        //A control point is a synoynm for a vertex
        int ControlPointIndCount = CurrentCluster->GetControlPointIndicesCount();

        globalBindposeInverseMatrix = transformLinkMatrix.Inverse();
        transformLinkMatrix = transformLinkMatrix.Transpose();

        //Get the inverse of each bone transform.
        DynamicMeshSkeleton->mBones[CurrentBoneIndex]->InvBoneTransform = XMFLOAT4X4((float)globalBindposeInverseMatrix.GetRow(0)[0],
            (float)globalBindposeInverseMatrix.GetRow(0)[1], (float)globalBindposeInverseMatrix.GetRow(0)[2],
            (float)globalBindposeInverseMatrix.GetRow(0)[3], (float)globalBindposeInverseMatrix.GetRow(1)[0],
            (float)globalBindposeInverseMatrix.GetRow(1)[1], (float)globalBindposeInverseMatrix.GetRow(1)[2],
            (float)globalBindposeInverseMatrix.GetRow(1)[3], (float)globalBindposeInverseMatrix.GetRow(2)[0],
            (float)globalBindposeInverseMatrix.GetRow(2)[1], (float)globalBindposeInverseMatrix.GetRow(2)[2],
            (float)globalBindposeInverseMatrix.GetRow(2)[3],
            (float)globalBindposeInverseMatrix.GetRow(3)[0], (float)globalBindposeInverseMatrix.GetRow(3)[1],
            (float)globalBindposeInverseMatrix.GetRow(3)[2], (float)globalBindposeInverseMatrix.GetRow(3)[3]);

        //Set the FBX Node on the Bone itself
        DynamicMeshSkeleton->mBones[CurrentBoneIndex]->SetFbxNode(CurrentCluster->GetLink());

        //Set the transform of the Bone as an XMFloat4x4 Matrix
        DynamicMeshSkeleton->mBones[CurrentBoneIndex]->BoneTransform = XMFLOAT4X4((float)transformLinkMatrix.GetRow(0)[0],
            (float)transformLinkMatrix.GetRow(0)[1], (float)transformLinkMatrix.GetRow(0)[2],
            (float)transformLinkMatrix.GetRow(0)[3], (float)transformLinkMatrix.GetRow(1)[0],
            (float)transformLinkMatrix.GetRow(1)[1], (float)transformLinkMatrix.GetRow(1)[2],
            (float)transformLinkMatrix.GetRow(1)[3], (float)transformLinkMatrix.GetRow(2)[0],
            (float)transformLinkMatrix.GetRow(2)[1], (float)transformLinkMatrix.GetRow(2)[2],
            (float)transformLinkMatrix.GetRow(2)[3], (float)transformLinkMatrix.GetRow(3)[0],
            (float)transformLinkMatrix.GetRow(3)[1], (float)transformLinkMatrix.GetRow(3)[2],
            (float)transformLinkMatrix.GetRow(3)[3]); //

        DynamicMeshSkeleton->mBones[CurrentBoneIndex]->SetFbxNode(CurrentCluster->GetLink());
        DynamicMeshSkeleton->mBones[CurrentBoneIndex]->SetFbxTransform(&transformLinkMatrix);
        DynamicMeshSkeleton->mBones[CurrentBoneIndex]->SetBoneIndex(CurrentBoneIndex);

        //For all of the control points inside each cluster, set thier weighting by getting the indicies.
        for (int i = 0; i < CurrentCluster->GetControlPointIndicesCount(); ++i)
        {
            //indexes and Ids of each vertex
            int index = CurrentCluster->GetControlPointIndices()[i];
            int vertexid = Indicies[CurrentCluster->GetControlPointIndices()[i]];

            if (Vertexes[index].Boneids.x == -1 && Vertexes[index].Weights.x == -1)
            {
                Vertexes[index].Boneids.x = (float)CurrentBoneIndex;
                Vertexes[index].Weights.x = (float)CurrentCluster->GetControlPointWeights()[i];
            }
            else if (Vertexes[index].Boneids.y == -1 && Vertexes[index].Weights.y == -1)
            {
                Vertexes[index].Boneids.y = (float)CurrentBoneIndex;
                Vertexes[index].Weights.y = (float)CurrentCluster->GetControlPointWeights()[i];
            }
            else if (Vertexes[index].Boneids.z == -1 && Vertexes[index].Weights.z == -1)
            {
                Vertexes[index].Boneids.z = (float)CurrentBoneIndex;
                Vertexes[index].Weights.z = (float)CurrentCluster->GetControlPointWeights()[i];
            }
```