3d Game Programming with DirectX 11

ianaesthetic

December 7, 2017

1 向量（Vector）

1.1 正交化（Orthogonalization）

对于一组向量 $\{\mathbf{v}_0, \mathbf{v}_1 \cdots \mathbf{v}_n\}$，正交化的过程为：

$$\mathbf{w}_0 = \mathbf{v}_0$$

$$\mathbf{w}_i = \mathbf{v}_i - \sum_{j=0}^{i} \text{proj}_{\mathbf{w}_j}(\mathbf{v}_i)$$

$$\mathbf{w}_i = \frac{\mathbf{w}_i}{\| \mathbf{w}_i \|}$$

1.2 数学库 DirectXMath

数学库已经从原来的 XNAMath 转变为集成到 Windows SDk 的 DirectXMath 。相较于使用 <xnamath.h> ，现在是使用 <DirectXMath.h> 并且在所有的函数都将在命名空间 DirectX 下。

使用 SIMD 的向量 XMVECTOR 用于计算，而相对应的储存由专门的其他类型来储存，以 3D 为例 XMFLOAT3 。整形向量使用数组来定义。之间的转换定义为：

```
XMVECTOR XMLoadFloat3(const XMFLOAT3 *source);
void XMStoreFloat3(XMFLOAT3* destination, FXMVECTOR v);
```

单独获取一个分量或者修改一个分量的定义为（分量替换为 X, Y, Z, W）：

```
FLOAT XMVectorSetX(FXMVECTOR v, FLOAT x);
FLOAT XMVectorGetX(FXMVECTOR v);
```

在定义函数的时候，参数需要使用 FXMVECTOR 或者 CXMVECTOR 以更好的利用 SIMD 。函数的前三个 XMVECTOR 参数为 FXMVECTOR，后面的一律使用 CXMVECTOR 。引用参数依然为 XMVECTOR 。常量 XMVECTOR 定义为

```
XMVECTORF32 v = {1.0f, 1.0f, 1.0f, 1.0f};
XMVECTORU32 u = {1, 1, 1, 1};
```

一些很常用的 3D 函数列表（注意有一些返回标量的函数为了保持 SIMD 选择依然返回一个 XMVECTOR ，并且将所有的分量都设置为结果标量）：

```
XMVECTOR XMVectorSet(
    FLOAT x, FLOAT y, FLOAT z, FLOAT w
);
XMVECTOR XMVector3Length(FXMVECTOR v);
XMVECTOR XMVector3LengthSq(FXMVECTOR v);
XMVECTOR XMVector3Dot(FXMVECTOR v, FXMVECTOR u);
XMVECTOR XMVector3Cross(FXMVECTOR v, FXMVECTOR u);
XMVECTOR XMVector3Normalize(FXMVECTOR v);
BOOL XMVectorEqual(FXMVECTOR v, FXMVECTOR u);
```

2 矩阵 (Matrix)

类似于 DirectXMath 中的向量的命名方式，同时实质上在运算时是由 4 个 XMVECTOR 构成。参数一律为 CXMMATRIX 类型。一些函数的定义：

```
XMMATRIX  XMMatrixSet(
    FLOAT m00, FLOAT m01, FLOAT m02, FLOAT m03,
    FLOAT m10, FLOAT m11, FLOAT m12, FLOAT m13,
    FLOAT m20, FLOAT m21, FLOAT m22, FLOAT m23,
    FLOAT m30, FLOAT m31, FLOAt m32, FLOAT m33
);
XMMATRIX XMMatrixIdentity();
BOOL XMMatrixIsIdentity(CXMMATRIX a);
XMMATRIX XMMatrixMultiply(CXMMATRIX a, CXMMATRIX b);
XMMATRIX XMMatrixTranspose(CXMMATRIX a);
XMMATRIX XMMatrixDeterminant(CXMMATRIX a);
XMMatrixInverse(XMMATRIX *p, CXMMATRIX a);
```

3 矩阵变化 (Transformation)

在 DirectXMath 这个库中，所有的向量都是行向量。这也就导致所有的变化矩阵都和列向量时的矩阵呈转置关系，下面的笔记仍然按照列向量来标记。针对于点的坐标轴旋转要加上点的坐标偏移量，对于点 $\mathbf{p}=\mathbf{v}+\mathbf{o}$:

$$\begin{pmatrix} \mathbf{u}_x & \mathbf{u}_y & \mathbf{u}_z & \mathbf{o}_x \\ \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z & \mathbf{o}_y \\ \mathbf{w}_x & \mathbf{w}_y & \mathbf{w}_z & \mathbf{o}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

一些在 DirectXMath 中定义的旋转矩阵：

```
XMMATRIX XMMatrixScaling(FLOAT x, FLOAT y, FLOAT z);
XMMATRIX XMMatrixScalingFromVector(FXMVECTOR v);
XMMATRIX XMMatrixRotationX(FLOAT angleX);
XMMATRIX XMMatrixTranslation(FLOAT x, FLOAT y, FLOAT z);
XMMATRIX XMMatrixTranslationFromVector(FXMVECTOR v);
XMMATRIX XMVector3Transform(FXMVECTOR v, CXMMATRIX m);
```

4 初始化 DirectX3D

组件对象模型 (Component Object Model, COM) 表示抽象出来，具有接口的模型。在编程过程中，我们不会对于这个模型进行内部访问，而是向对类一样通过接口进行内部的访问以及删除操作。

数据的储存类型有:

```
DXGI_FORMAT_R32G32B32_FLOAT;
DXGI_FORMAT_R16G16B16_UNORM;
DXGI_FORMAT_R32G32_UINT;
DXGI_FORMAT_R8G8B8_UNORM;
DXGI_FORMAT_R8G8B8_SNORM;
DXGI_FORMAT_R8G8B8_UINT;
DXGI_FORMAT_R8G8B8_SINT;
```

对于每一个像素，都会有一个深度缓存 (depth-buffering) 用来处理隐藏平面。所有的深度缓存会记录当前的深度值 (在区间 [0, 1] 之间)，并记录对应的数据。这实际上被储存为一种材质，有以下的几种类型:

```
DXGI_FORMAT_D32_FLOAT_S8X24_UINT;
DXGI_FORMAT_D32_FLOAT;
DXGI_FORMAT_D24_UNORM_S8_UINT;
DXGI_FORMAT_D16_UNORM ;
```

对于每一个材质，一定要转化为资源视图 (resource view) 才能被硬件在管道中使用。对于材质来说，会有不同的标签 (binding flag) 来表示可以被用作何种材质，如 shader resource, render target:

```
D3D11_BIND_RENDER_TARGET;
D3D11_BIND_SHADER_RESOURCE;
```

材质的反锯齿，在 MSAA 模式下有一个 DXGI_SAMPLE_DESC 的结构需要设定。其中有两个成员，Count 表示每一个像素会被分为几个子像素，而 Quality 是一个定义最后反锯齿质量的值，有材质本身的存储格式和 Count 决定，对于 Quality 上限的查询为 [0, *pNumQualityLevels - 1]:

```
HRESULT ID3D11Device::CheckMultisampleQualityLevels(
    DXGI_FORMAT format,
    UINT sampleCount,
    UINT *pNUmQualityLevels
);
```

4.1 计时器 (GameTimer)

Windows 里面的时间是由 Counts 计算的。每一秒内的 count 数目可以由以下得到:

```
__int64 CountsPerSecond;
QueryPerformanceFrequency((LARGE_INTEGER*)CountsPerSecond);
```

得到当前有 Counts 作为单位的时间:

```
__int64 CurrTime;
QueryPerformanceCounter((LARGE_INTEGER*)CurrTime);
```

```
Class GameTimer {
public:
    GameTimer();

    float TotalTime(); //Two situations: paused or not paused
    float DeltaTime();

    void Tick();    // Update every frame
    void Pause();   // Update when paused
    void Start();   // Update when restarted
    void Reset();   // Update when set current time to base time

private:
    double _SecondPerCounts;
    double _DeltaTime;

    __int64 _BaseTime;  //Time recognized to be origin
    __int64 _CurrTime;  //Time of current frame
    __int64 _PrevTime;  //Time of previous frame
    __int64 _StopTime;  //Time of recent pause
    __int64 _PauseTime; //Time total length of pause time

    bool _Stop;       // Whether program is paused
};
```

4.2 DirectX 基本框架

所有的流程的基本框架是:

(a) 通过 D3D11CreateDevice() 创建 Device(ID3D11Device) 和 Context(ID3D11DeviceContext) 接口

```
DriverType = D3D_DRIVER_TYPE_HARDWARE
SDKVersion = D3D11_SDK_VERSION
```

(b) 通过 CheckMultisampleQualityLevels() 来检查支持的 MSAA QualityLevel。注意 assert(mMsaa4xQualityLevel > 0)

(c) 通过 DXGI_SWAP_CHAIN_DESC 结构来描述 mSwapChain

```
BufferDesc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;
BufferDesc.ScanlineOrdering =
    DXGI_MODE_SCANLINE_ORDERING_UNSPECIFIED;
BufferUsage = DXGI_USAGE_RENDER_TARGET_VIEW;
SwapEffect = DXGI_SWAP_EFFECT_DISCARD;
```

(d) 使用 IDXGIFACTORY 的成员函数 CreateSwapChain() 来创建 mSwapChain; IDXGIFACTORY 要先用 mDevice->QueryInterface() 得到 IDXGIDevice, 然后再通过成员函数 GetAdapter() 和 GetParent() 得到 IDXGIFACTORY

(e) BackBuffer 本质上也是一种 Texture, 创建 ID3D11Texture2D* backBuffer, mSwapChain->GetBuffer() 得到 backBuffer 然后再用 mDevice-> CreateRenderTargetView() 创建 ID3D11RenderTargetView mRender-TargetView

(f) depthStencilBuffer 也是一种 texture, 与 backBuffer 类似。与其不同的是对于 depthStencilBuffer, 需要设定 D3D11_TEXTURE2D_DESC。

之后类似于之前方法，ID3D11TEXTURE2D* mDepthStencilBuffer 由 mDevice->CreateTexture2D() 得到，然后通过 mDevice->CreateDepthStencilView() 创建 ID3D11DepthStencilView() 创建 mDepthStencilView；再将得到的 mRenderTargetView 和 mDepthStencilView 绑到 Output Merger: mDeviceContext-> OMSetRenderTarget()

```
BindFlags = D3D11_BIND_DEPTH_STENCIL;
Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
USAGE = D3D11_USAGE_DEFAULT;
```

(g) 设定 D3D_VIEWPORT mViewport，然后再作为一种状态加入 Context: mDeviceContext-> RSSetViewport()

4.3 D3DApp.h 整体框架

与图形部分直接相关的框架函数有：

```
virtual bool Init();
virtual void OnResize();
virtual void UpdateScene(float dt);
virtual void DrawScene();
int Run();
```

其中 Run() 函数尤其值得注意：

```
int D3App::Run() {
    MGS msg;
    mTimer.Reset();

    While(msg.Message != WM_QUIT) {
        if(PeekMessage(&smg, 0, 0, 0, PM_REMOVE)) {
            TranslateMessage(&msg);
            DisPatchMessage(&msg);
        }
        else {
            if(!mAppPaused) {
                Timer.Tick();
                UpdateScene();
                DrawScene();
            }
            else {
                Sleep(100);
            }
        }
    }
    return (int)msg.wParam;
}
```

5 Some BackGround Knowledge

nothing new to say

6 Direct3D 初步

6.1 Vertex 读入

顶点读入数据元素格式的设定：所有的顶点的信息是由结构体存储的，所以需要指定每一个元素的读入规则来对应到之后的 shader / effect 文件的读入。读入的格式由一个 D3D11_INPUT_ELEMENT_DESC 数组来制定。这之后再用 ID3D11Device::CreateInputLayout() 来创建。其中为了让 shader 中的读入和结构体中的名字对应，要用 effect 中的 technique 的有关读入的 pass 的 passDesc 来作为参数。

```
mTech–> GetPassByIndex(0)–> GetDesc(&passDesc);
passDesc.pIAInputSignature;
passDesc.IAInputSignatureSize;
```

顶点数据缓冲区的设定：所有的顶点数据将会先储存在缓冲区 (buffer) 中。所以缓冲区要先用 D3D11_BUFFER_DESC 来描述，然后再用 D3D11_SUBRESOURCE_DATA 来指定缓冲区的数据来源,然后再用 ID3D11DEVICE::CreateBuffer() 来创造。根据之前对于 Vertex 机构体读入的 input slot 的描述通过 ID3D11DeviceContext::IASetVertexBuffers() 来绑定缓冲区和 Input slot

Indices 设定： Indices 用于描述 triangle mesh ，由 UINT 数组直接描述。相类似的，它也是由 D3D11_BUFFER_DESC 来描述的，同时由 D3D11_SUBRESOURCE_DATA 来制定缓冲数据，再由 ID3D11DeviceContext::IASetIndexBuffers() 来制定。

6.2 Shader & Effect

Shader 中的常量数据制定在 cbuffer 之中。

Render States 可以由 D3D11 中已经设置好的变量来更改。在这里引入 Rasterizer State; 它首先是由 D3D11_RASTERIZER_DESC 描述，然后由 ID3D11Device::CreateRasterizerState() 来得到 ID3D11RasterizerState*; 通过 ID3D11DeviceContext::RSSetState() 来改变状态自动机。同时也可以写在 effect 文件当中类似于：

```
RasterizerState WireframeRs {
    FillMode = Wireframe;
    CullMode = Back;
    FontCounterClockwise = false;
}

pass p0 {
    ...
    SetRasterizerState(WireframeRs);
}
```

在开源到的 effect.lib 中,C++ 程序使用 D3DX11CompileEffectFromFile() 来编译 effect 文件。在这个函数中，原本的 pFunction, pProfile, pPump, pHResult 都被弃用了。然后可以使用 DX11CreateEffectFromMemory() 函数来创造 effect 接口 (e.g mFX) 通过这个接口，我们可以访问 shader 中定义的内部结构：

```
ID3DX11EffectMatrixVariable* fxWVPVar;
fxWVPVar = mfx—> GetVariableByName("gWVP")—> AsMatrix();
fxWVPVar—> SetMatrix((float)* &M);
```

ID3DX11EffectVectorVariable 指 4D，而 ID3DX11EffectVariable 指 3D 数组。通过 ID3DX11Effect::GetTechniqueByName 可以获得 technique；使用 effect 文件来画的方式为：

```
D3DX11_TECHI
```