常量缓冲区_备忘录

笔记本： DirectX 12
创建时间： 2022/8/4 13:21 **更新时间：** 2022/8/15 19:16
作者： handsome小赞
URL： https://copyfuture.com/blogs-details/a293884bcc0ca4553467cd053b616dab

> 常量缓冲区（ **cbuffer** ）也是一种GPU资源，和其他缓冲区资源一样，可以被着色器程序引用。

> **1.创建常量缓冲区**
>
> 1. 不同于顶点缓冲区和索引缓冲区，常量缓冲区通常由CPU每帧更新一次，所以需要创建在上传堆。
> 2. 常量缓冲区的大小必须是硬件最小分配空间（256B）的整数倍
> 3. 每个着色器阶段最多允许15个常量缓冲区，并且每个缓冲区最多可以容纳4096个标量。HLSL的cbuffer需要指定register(b#), #的范围为0到14

4. 创建（支持常量缓冲区数组）

```
// 将纹理资源绑定到纹理寄存器槽 0
Texture2D  gDiffuseMap : register(t0);

// 把下列采样器资源依次绑定到采样器寄存器槽 0~5
SamplerState gsamPointWrap        : register(s0);
SamplerState gsamPointClamp       : register(s1);
SamplerState gsamLinearWrap       : register(s2);
SamplerState gsamLinearClamp      : register(s3);
SamplerState gsamAnisotropicWrap  : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

// 将常量缓冲区资源绑定到常量缓冲区（cbuffer）寄存器槽 0
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
  float4x4 gTexTransform;
};

// 绘制过程中所用的杂项常量数据
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gProj;
  [...] // 为篇幅而省略的其他字段
};

// 绘制每种材质所需的各种不同的常量数据
cbuffer cbMaterial : register(b2)
{
  float4   gDiffuseAlbedo;
  float3   gFresnelR0;
  float    gRoughness;
  float4x4 gMatTransform;
};
```

4. 支持自动隐式填充HLSL

尽管我们已经按照上述方式在程序中分配出了 256 整数倍字节大小的数据空间，但是却无须为 HLSL 结构体中显式填充相应的常量数据，这是因为它会暗中自行完成这项工作：

```
// 隐式填充为 256B
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
};

// 显式填充为 256B
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
  float4x4 Pad0;
  float4x4 Pad1;
  float4x4 Pad1;
};
```

2.更新常量缓冲区

由于常量缓冲区是用 D3D12_HEAP_TYPE_UPLOAD 这种堆类型来创建的，所以我们就能通过 CPU 为常量缓冲区资源更新数据。为此，我们首先要获得指向欲更新资源数据的指针，可用 Map 方法来做到这一点：

```
ComPtr<ID3D12Resource> mUploadBuffer;
BYTE* mMappedData = nullptr;
mUploadBuffer->Map(0, nullptr, reinterpret_cast<void**>(&mMappedData));
```

第一个参数是子资源（subresource）的索引[1]，指定了欲映射的子资源。对于缓冲区来说，它自身就是唯一的子资源，所以我们将此参数设置为 0。第二个参数是一个可选项，是个指向 D3D12_RANGE 结构体的指针，此结构体描述了内存的映射范围，若将该参数指定为空指针，则对整个资源进行映射。第三个参数则借助双重指针，返回待映射资源数据的目标内存块。我们利用 memcpy 函数将数据从系统内存（system memory，也就是 CPU 端控制的内存）复制到常量缓冲区：

```
memcpy(mMappedData, &data, dataSizeInBytes);
```

当常量缓冲区更新完成后，我们应在释放映射内存之前对其进行 Unmap（取消映射）操作[2]：

```
if(mUploadBuffer != nullptr)
  mUploadBuffer->Unmap(0, nullptr);

mMappedData = nullptr;
```

Unmap 的第一个参数是子资源索引，指定了将被取消映射的子资源。若取消映射的是缓冲区，则将其置为 0。第二个参数是个可选项，是一个指向 D3D12_RANGE 结构体的指针，用于描述取消映射的内存范围，若将它指定为空指针，则取消整个资源的映射。

3.上传缓冲区辅助函数

定义于 UploadBuffer.h

实现了：

1. 上传缓冲区资源的构造与析构函数
2. 处理资源的映射和取消映射
3. 提供了CopyData方法来更新缓冲区内的特定元素

4.常量缓冲区描述符

1. 常量缓冲区描述符都需要存放在以 D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV 类型所创建的描述符堆里
2. 描述符的描述参数 SizeInBytes 也必须是**256B**的整数倍
3. 在创建供着色器程序访问资源的描述符时，需要把Descriptor Heap的 Flags 指定为DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE
4. 通过填写 D3D12_CONSTANT_BUFFER_VIEW_DESC 实例，再调用 ID3D12Device_CreateConstantBufferView 方法，便可创建常量缓冲区

5.根签名和描述符表

1. 不同类型的资源会被绑定到特定的寄存器槽上

```
// 将纹理资源绑定到纹理寄存器槽 0
Texture2D  gDiffuseMap : register(t0);

// 把下列采样器资源依次绑定到采样器寄存器槽 0~5
SamplerState gsamPointWrap        : register(s0);
SamplerState gsamPointClamp       : register(s1);
SamplerState gsamLinearWrap       : register(s2);
SamplerState gsamLinearClamp      : register(s3);
SamplerState gsamAnisotropicWrap  : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);
```

2. **根签名**
   - 根签名必须与使用它的着色器相兼容（即在绘制开始之前，根签名一定要为着色器提供其执行期间需要绑定到渲染流水线的所有资源）
     在创建流水线状态对象时会对此进行验证
   - t 着色器资源描述符
     s 采样器
     u 无序访问描述符
     b 常量缓冲区描述符
   - 根签名由 ID3D12RootSiganture 接口来表示，并以一组描述绘制调用过程中着色器所需资源的**根参数**定义而成
   - **根参数**：根常量、根描述符、根描述符表
     - 根参数数据类型：CD3DX12_ROOT_PARAMETER
       1. 根常量
       2. 根描述符
       3. 根描述符表 CD3DX12_DESCRIPTOR_RANGE
          需要初始化

```
// 创建一个只存有一个 CBV 的描述符表
CD3DX12_DESCRIPTOR_RANGE cbvTable;
cbvTable.Init(
  D3D12_DESCRIPTOR_RANGE_TYPE_CBV,
  1, // 表中的描述符数量
  0);// 将这段描述符区域绑定至此基准着色器寄存器（base shader register）
```

     - 根参数初始化

```
// 根参数可以是描述符表、根描述符或根常量
CD3DX12_ROOT_PARAMETER slotRootParameter[1];


slotRootParameter[0].InitAsDescriptorTable(
  1,           // 描述符区域的数量
  &cbvTable); // 指向描述符区域数组的指针
```

   - 根签名由一组根参数构成

```
// 根签名由一组根参数构成
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(1, slotRootParameter, 0, nullptr,
  D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

   - 必须先将根签名的描述布局进行序列化处理，待其转换为以 ID3DBlob 接口表示的序列化数据后，才可以传入 CreateRootSignature 方法

- 创建一个仅含一个槽位的根签名

```
// 创建仅含一个槽位（该槽位指向一个仅由单个常量缓冲区组成的描述符区域）的根签名
ComPtr<ID3DBlob> serializedRootSig = nullptr;
ComPtr<ID3DBlob> errorBlob = nullptr;
HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc,
  D3D_ROOT_SIGNATURE_VERSION_1,
  serializedRootSig.GetAddressOf(),
  errorBlob.GetAddressOf());
ThrowIfFailed(md3dDevice->CreateRootSignature(
  0,
  serializedRootSig->GetBufferPointer(),
  serializedRootSig->GetBufferSize(),
  IID_PPV_ARGS(&mRootSignature)));
```

3. **绑定资源流程**
    1. 创建根参数（以描述符表作为根参数进行示范）
    2. 创建一个只有一个CBV的描述符表，并初始化
    3. 堆根参数初始化（以描述符表进行初始化）
    4. 以根参数创建根签名的描述 desc
    5. 创建根签名描述 desc 的序列化
    6. 以序列化创建根签名
    7. 将根签名和CBV堆设置到命令列表上
    8. 获取CBV堆的一个CBV
    9. 以获取的CBV为参数设置描述符表到命令列表上