

学习笔记_实例化与视锥体剔除

笔记本: DirectX 12

创建时间: 2022/9/11 11:08

更新时间: 2022/9/14 8:55

作者: handsome小赞

- 绘制实例数据

其实前面就一直在绘制实例数据, 但 DrawIndexedInstanced 方法的第二个参数被设为了 1, 所以一直是只绘制了一个几何体实例。

```
cmdList->DrawIndexedInstanced(ri->IndexCount, 1,
    ri->StartIndexLocation, ri->BaseVertexLocation, 0);
```

- 实例数据

传统方法是在创建输入布局 (实例数据都是自输入装配阶段获取的) 时, 可以通过枚举型 D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA 来替代 D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA 来指定输入的数据为逐实例数据流, 而非逐顶点数据流。随后再将第二个顶点缓冲区与含有实例数据的输入流相绑定。

新方法

为所有实例都创建一个存有其实例数据的结构化缓冲区。如, 若要将某个对象实例化 100次, 则应当创建 100个实例数据元素的结构化缓冲区。接着把此结构化缓冲区资源绑定到渲染流水线上, 并根据要绘制的实例在顶点着色器中索引相应的数据。

同时, Direct3D提供了系统标识符 SV_InstanceID, 以0起始

- 创建实例缓冲区

- 实例数据结构体

```
struct InstanceData
{
    DirectX::XMFLAOT4X4 World = MathHelper::Identity4x4();
    DirectX::XMFLAOT4X4 TexTransform = MathHelper::Identity4x4();
    UINT MaterialIndex;
    UINT InstancePad0;
    UINT InstancePad1;
    UINT InstancePad2;
};
```

- 修改渲染项

因为渲染项含有实例化次数的相关信息, 所以位于系统内存中的实例数据也

应算作渲染项结构体的组成部分

```
struct RenderItem
{
    ...
    std::vector<InstanceData> Instances;
    ...
};
```

- **以 InstanceData 元素类型创建结构化缓冲区**

目的是为了使得 GPU可以访问到这些实例数据

- 动态缓冲区 (Upload Buffer)
- 仅将可见 (用户视野中可见的物体) 实例的实例数据复制到此结构化缓冲区内 (这与视锥体剔除有关)

使用结构化缓冲区, 其目的是为了只绑定一个描述符。目前可以用数组的只有 纹理和结构化缓冲区, 而结构化缓冲区是以SRV和UAV为描述符, 纹理也是以SRV为描述符。但纹理是先创建描述符堆 (创建时指定描述符数量), 然后以创建的描述符堆创建每个纹理的SRV, 将描述符堆的头指针绑定**根描述符表** `mCommandList-`

`>SetGraphicsRootDescriptorTable(3, mSrvDescriptorHeap-`

`>GetGPUDescriptorHandleForHeapStart());`, 根签名会自行推断知道

描述符表里到底含有多少个描述符, 其对应于 HLSL 内的

Texture2DArray 或 Texture2D 数组 (区别是 Texture2D数组支持不

不同类型的纹理)。而结构化缓冲区是先得有数据数组, 创建UAV缓冲

区 和 利用数据数组创建 SRV 缓冲区 (因为一般以SRV为输入, UAV

为输入, **注意**, 因为CPU要读取UAV, 所以应当创建上传缓冲区, 而

SRV请根据具体需求创建缓冲区), 将创建的SRV缓冲区绑定**根描述**

符 `mCommandList->SetGraphicsRootShaderResourceView(0,`

`instanceBuffer->GetGPUVirtualAddress());`

- 包围体与视锥体

- **DirectXMath 碰撞检测库**

DirectXCollision.h 工具库, 它是 DirectXMath 库的一部分。此库提供了一份常见几何图元相交测试的快速实现

- **包围盒**

- 两种表示方法
 1. 最小点 v_{min}
最大点 v_{max}
 2. 中心 c
扩展(extents)向量 e

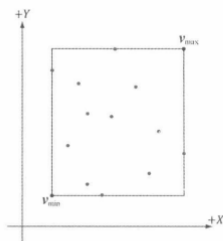


图 16.2 用最小值点和最大值点表示包围目标点集的 AABB

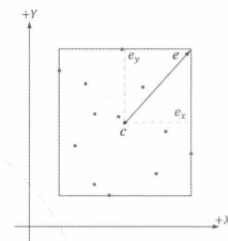


图 16.3 用包围盒中心与扩展向量表示包围目标点集的 AABB

- 两者转换

$$c = 0.5(v_{min} + v_{max})$$

$$e = 0.5(v_{max} - v_{min})$$
- DirectXMath 碰撞检测库采用的是 包围盒中心与扩展向量组合的表达方式
数据结构体 `struct BoundingBox`
- 轴对齐包围盒及其旋转操作
 AABB: 任意朝向的包围盒
 OBB: 定向包围盒
 OOBB: 位于局部空间的OBB

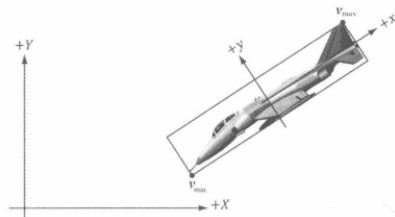


图 16.4 包围盒与 xy 标架的坐标轴相对齐，却没有对齐于标架的 XY 坐标轴

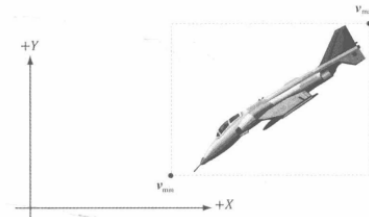


图 16.5 轴对齐于 XY 标架的包围盒

在实际工作中，我们总是先将网格变换到其局部空间，再以局部空间内的轴对齐包围盒进行碰撞检测

数据结构体 `struct BoundingBoxOriented`

保存了 OBB 相对于世界空间的朝向，`XMFLAOT4`

`Orientation` (表示包围盒旋转 [box -> world] 的单位四元数)

- DirectX碰撞检测库提高了用于创建 AABB 和 OBB 的静态成员函数

```

void BoundingBox::CreateFromPoints(
    _Out_ BoundingBox& Out,
    _In_ size_t Count,
    _In_reads_bytes_(sizeof(XMFLOAT3)+Stride*(Count-1)) const XMFLOAT3* pPoints,
    _In_ size_t Stride );

void BoundingOrientedBox::CreateFromPoints(
    _Out_ BoundingOrientedBox& Out,
    _In_ size_t Count,
    _In_reads_bytes_(sizeof(XMFLOAT3)+Stride*(Count-1)) const XMFLOAT3* pPoints,
    _In_ size_t Stride );

```

如果我们定义了顶点结构体如下:

```

struct Basic32
{
    XMFLOAT3 Pos;
    XMFLOAT3 Normal;
    XMFLOAT2 TexC;
};

```

并且, 构成网格所用的顶点数组为:

```

std::vector<Vertex::Basic32> vertices;

```

那么, 我们就能按下面那样调用函数来生成包围盒:

```

BoundingBox box;
BoundingBox::CreateFromPoints(
    box,
    vertices.size(),
    &vertices[0].Pos,
    sizeof(Vertex::Basic32));

```

函数中的 Stride (步长) 参数表示的是需要越过多少字节才能到达下一个顶点元素处。

• 注意

为了计算出目标网格的包围体, 我们要在系统内存中准备一份可供使用的顶点列表副本, 并存在如 `std::vector` 这样的类型中。这样做的原因是, CPU 无法从以渲染为目的而创建的顶点缓冲区中读取数据。针对这种情况, 应用程序中常见的做法就是, 为这种数据维护一份存于系统内存中的副本, 像拾取 (picking, 第 17 章的主题) 与碰撞检测 (collision detection) 两种技术就是这样实现的。

• 包围球

- **数据结构体** `struct BoundingSphere`
- 包围球的中心 Center 是以 AABB 求取的中心
- 包围球的半径 Radius 是以球心 c 至网格体上任意顶点 p 之间的最大距离
- DirectX 碰撞检测库提供了下列静态函数用于创建包围球

```

void BoundingSphere::CreateFromPoints(
    _Out_ BoundingSphere& Out,
    _In_ size_t Count,
    _In_reads_bytes_(sizeof(XMFLOAT3)+Stride*(Count-1)) const XMFLOAT3* pPoints,
    _In_ size_t Stride );

```

• 视锥体

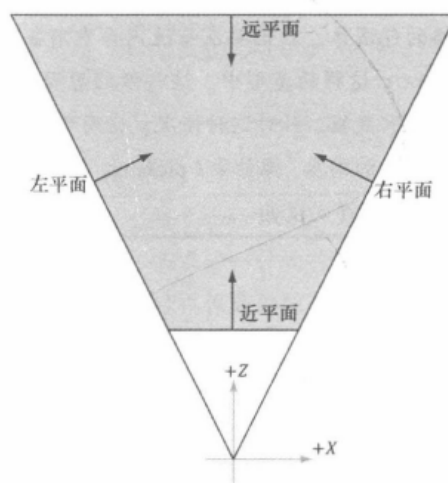


图 16.6 视锥体众平面正半空间的交集定义了视锥体的空间范围

- **构建视锥体的众平面**

简便方法：在观察空间中，采取以原点为中心，沿z轴正方向俯瞰视锥体的标准形式。此时，可根据在Z轴上至原点的距离来方便地确定近平面和远平面，而左平面和右平面、顶平面与底平面这两组平面则既对称又经过原点。因此，我们在表达观察空间中的视锥体时，就不必存储所有的平面方程，只需要简单地记录 顶、底、左、右 4个平面的平面斜率，以及近平面与远平面在Z轴上至原点的距离即可。

数据结构体 `struct BoundingFrustum`

```
struct BoundingFrustum
{
    static const size_t CORNER_COUNT = 8;

    XMFLOAT3 Origin;           // 视锥体（及其投影）的原点
    XMFLOAT4 Orientation;      // 表示旋转操作的四元数

    float RightSlope;          // X轴上的正斜率（X/Z），即右平面的斜率
    float LeftSlope;           // X轴上的负斜率，即左平面的斜率
    float TopSlope;             // Y轴上的正斜率（Y/Z），即顶平面的斜率
    float BottomSlope;         // Y轴上的负斜率，即底平面的斜率
    float Near, Far;           // 近平面与远平面在Z轴上至原点的距离
    ...
}
```

- 视锥体与球体的相交检测

- **注意** 视锥体是有 6 个内向平面围成的空间范围，球体与视锥体平面需要6次相交检测，所以它与球体相交，判断球体是否位于视锥体平面的负半空间。
- 其中，检测算法如下：借助球心到平面的距离和半径进行比较判断

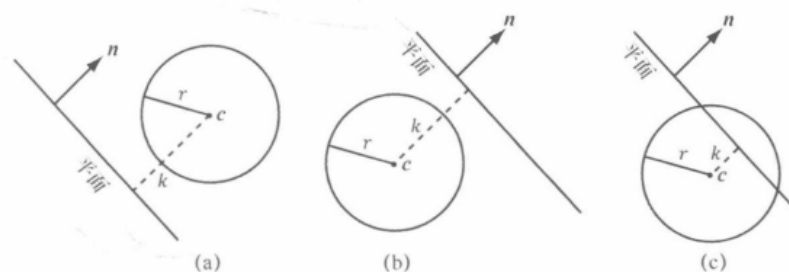


图 16.7 球体与平面相交检测可能会遇到的几种情况

(a) $k > r$ 且球体与平面的正半空间相交 (b) $k < -r$ 且球体完全位于平面后侧的负半空间之内

(c) $|k| \leq r$ 且球体与平面相交

BoudingFrustum 类提供了检测函数 `BoudingFrustum::Contains`

- 视锥体与轴对齐包围盒的相交检测

- 大致与上一节相同
- 其中，检测算法如下：先求出穿过包围盒体中心且方向与平面法线n 最为接近的包围盒体对角向量 $v = \overrightarrow{PQ}$ 。如图，如果 p 位于平面的前侧，则 Q 也一定位于平面的前侧。若 P 位于平面后侧而 Q 位于前侧，则相交。对于该对角向量，选取 Vmin与Vmax进行差运算可得到。差运算的顺序由

法线 n 的正负决定。

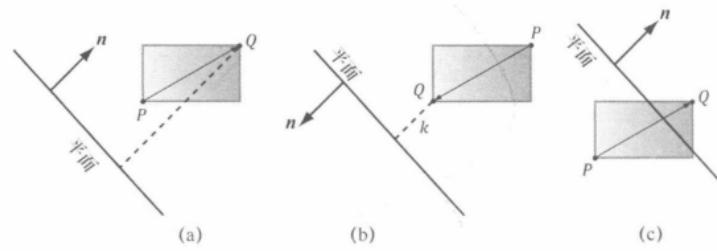


图 16.8 AABB 与平面相交测试可能会出现的情况。

\overline{PQ} 是与平面向量方向最为相近的包围盒体对角线

- 视锥体剔除

- 在CPU阶段，测试包围体与视锥体是否相交，并仅将相交的物体交由 Direct3D 绘制。