# 实例化/视锥体剔除(Frustum Culling)

# 实例化

实例化技术常用于需要对一个对象反复进行绘制的场景，物体的位置，朝向，缩放大小，材质，纹理等可能都不相同。

这样我们只需存储一份顶点与索引数据，对不同实例使用不同的世界矩阵与材质即可。并且只需调用一个API(**ID3D11DeviceContext::DrawIndexedInstanced**)即可一次绘制多个实例。

# 实例数据

不同的实例需要一些不同的数据，我们可以用一个结构化缓冲区来储存这些数据。Direct3D提供了系统值**SV\_InstanceID**来标识不同的实例，我们可以用其作为索引来获得相应的数据。

|  |
| --- |
| struct InstanceData  {  float4x4 World;  uint MaterialIndex;  uint InstPad0;  uint InstPad1;  uint InstPad2;  };      struct MaterialData  {  Material Mat;  uint DiffuseMapIndex;  uint MatPad0;  uint MatPad1;  uint MatPad2;  };      Texture2DArray gDiffuseMapArray : register(t0);      StructuredBuffer<InstanceData> gInstanceData : register(t1);  StructuredBuffer<MaterialData> gMaterialData : register(t2);      ...      struct VertexOut  {  float4 PosH : SV\_POSITION;  float3 PosW : POSITION;  float3 NormalW : NORMAL;  float2 TexUV: TEXUV;      // nointerpolation is used so the index is not interpolated  // across the triangle.  nointerpolation uint MatIndex : MATINDEX;  };      VertexOut VS(VertexIn vin, uint instanceID : SV\_InstanceID)  {  VertexOut vout;    // Fetch the instance data.  InstanceData instData = gInstanceData[instanceID];  float4x4 world = instData.World;  uint matIndex = instData.MaterialIndex;      vout.MatIndex = matIndex;      // Transform to world space space.  float4 posW = mul(float4(vin.PosL, 1.0f), world);  vout.PosW = posW.xyz;  vout.NormalW = mul(vin.NormalL, (float3x3)world);    // Transform to homogeneous clip space.  vout.PosH = mul(posW, gViewProj);      vout.TexUV = vin.TexUV;      **return** vout;  }      float4 PS(VertexOut pin) : SV\_Target  {  MaterialData matData = gMaterialData[pin.MatIndex];      float4 texColor = gDiffuseMapArray.Sample(gSampler, float3(pin.TexUV, matData.DiffuseMapIndex));  Material mat = matData.Mat;    ...  } |

我们使用gInstanceData来存储实例数据，包括其世界转换矩阵和材质ID，然后用gMaterialData来存储所有可能的材质，最后用gDiffuseMapArray来存储所有可能的贴图。

可以看到，该代码已经没有了cbPerObject，每个物体的数据将由实例缓冲区提供。

实例缓冲区存有绘制每个实例所需的数据，在CPU端，我们使用以下代码来创建实例缓冲区存。

|  |
| --- |
| struct InstanceData  {  DirectX::XMFLOAT4X4 World;  UINT MaterialIndex;  UINT InstPad0;  UINT InstPad1;  UINT InstPad2;  };      ...      **void** InstancingGame::BuildInstancedBuffer()  {  **float** width = 200.0f;  **float** height = 200.0f;  **float** depth = 200.0f;      **int** instanceN = 5;      m\_instanceCount = instanceN \* instanceN\*instanceN;    m\_instancedDataArray.resize(m\_instanceCount);  m\_cullingDataArray.reserve(m\_instanceCount);      **float** x = -0.5f\*width;  **float** y = -0.5f\*height;  **float** z = -0.5f\*depth;  **float** dx = width / (instanceN - 1);  **float** dy = height / (instanceN - 1);  **float** dz = depth / (instanceN - 1);  **for** (**int** k = 0; k < instanceN; ++k)  {  **for** (**int** i = 0; i < instanceN; ++i)  {  **for** (**int** j = 0; j < instanceN; ++j)  {  // Position instanced along a 3D grid.      // We need transpose here.  m\_instancedDataArray[k\*instanceN\*instanceN + i \* instanceN + j].World = XMFLOAT4X4(  1.0f, 0.0f, 0.0f, x + j \* dx,  0.0f, 1.0f, 0.0f, y + i \* dy,  0.0f, 0.0f, 1.0f, z + k \* dz,  0.0f, 0.0f, 0.0f, 1.0f);        m\_instancedDataArray[k\*instanceN\*instanceN + i \* instanceN + j].MaterialIndex = rand() % m\_instanceCrate->m\_matCount;  }  }  }      UINT byteWidth = m\_instanceCount \* sizeof(InstanceData);  CD3D11\_BUFFER\_DESC instanceDataDesc(byteWidth, D3D11\_BIND\_SHADER\_RESOURCE);  instanceDataDesc.MiscFlags = D3D11\_RESOURCE\_MISC\_BUFFER\_STRUCTURED;  instanceDataDesc.StructureByteStride = sizeof(InstanceData);      D3D11\_SUBRESOURCE\_DATA initData;  initData.pSysMem = m\_instancedDataArray.data();  initData.SysMemPitch = 0;  initData.SysMemSlicePitch = 0;      HRESULT hr = m\_d3dDevice->CreateBuffer(&instanceDataDesc, &initData, m\_instanceDataBuffer.GetAddressOf());  DX::ThrowIfFailed(hr);      CD3D11\_SHADER\_RESOURCE\_VIEW\_DESC srvDesc(D3D11\_SRV\_DIMENSION\_BUFFER);  srvDesc.Buffer.NumElements = m\_instanceCount;      hr = m\_d3dDevice->CreateShaderResourceView(m\_instanceDataBuffer.Get(), &srvDesc, m\_instanceDataSRV.GetAddressOf());  DX::ThrowIfFailed(hr);  } |

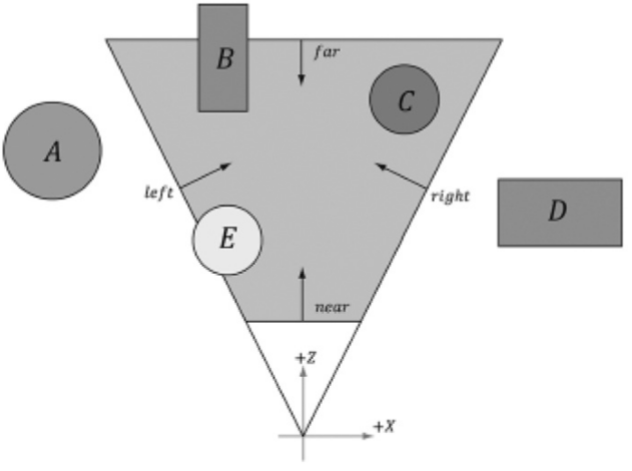
我们可以在每帧中对其进行更新，只绘制可见实例。

# 视锥体剔除(Frustum Culling)

硬件会在裁剪阶段自动丢弃位于视锥体以外的三角形。但当我们有数百万的三角形时，仍需先通过绘制调用将其提交至渲染流水线，经过顶点着色器，可能还要经过曲面细分阶段和几何着色器，之后才进入裁剪阶段。可见此流程效率相当低下。

视锥体剔除则是在应用程序这边，在高于以三角形为基本单位的层级中，按组剔除三角形。

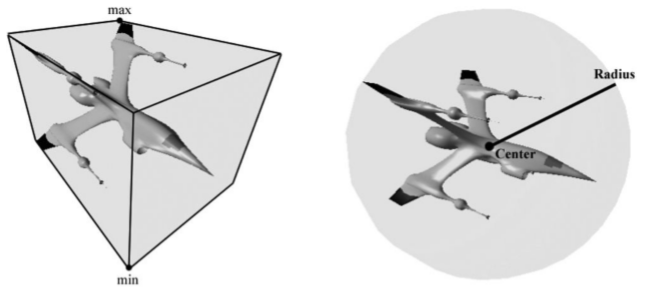
我们可以按物体构建对应的包围体，如果视锥体与这些包围体不相交，则无需将其提交给GPU。



假设有一台水平和垂直视场角（FOV）均为90度且远平面无穷远的摄像机，6个同样的观察范围可以严丝合缝地覆盖整个场，该摄像机的视锥体仅占用世界空间的1/6。假设世界中的物体均匀分布，则启用视锥体剔除后，5/6的物体会被丢弃。

# 包围体(Bounding Volume)

包围体即近似于目标物体体积的基本几何对象。尽管其只是与物体的形状相似，但是用数学表示起来非常简单，使其在工作中易于使用。

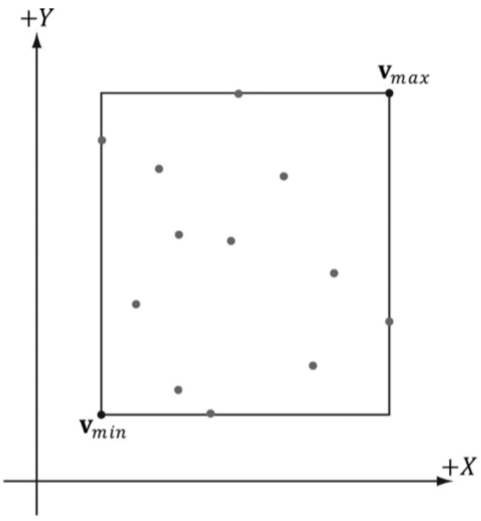


我们使用DirectXCollision.h工具库，它是DirectX数学库的一部分。此库提供了一些常见几何图元相交测试的快速实现。

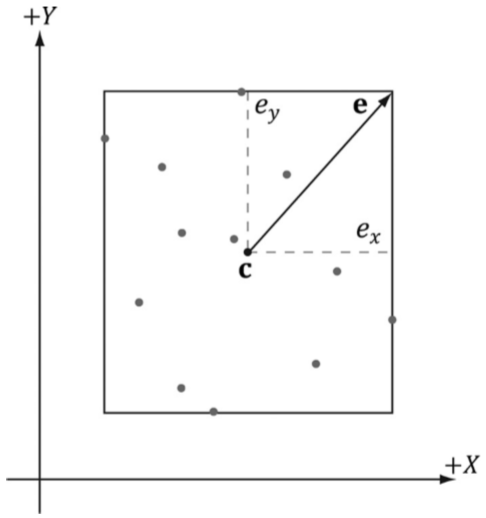
# 包围盒

轴对齐包围盒(axis-aligned bounding box, AABB)是一种将目标网格紧密包围，且各面皆平行于坐标主轴的长方体。

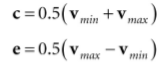
我们可以通过最小点和最大点来描述AABB。通过查找目标网格体中所有顶点在xyz轴上坐标的最小值和最大值即可求得两个点的坐标。



也可以使用另一种方式来表示，以盒的中心c，和扩展向量e来表示，后者是有中心沿坐标轴到各盒面的距离，也即中心到最大点的向量。



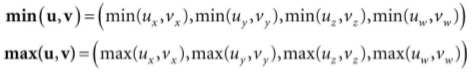
DirextX碰撞库采用第二种方式来表示。两种方式也可以相互转换:



我们用以下代码来计算AABB

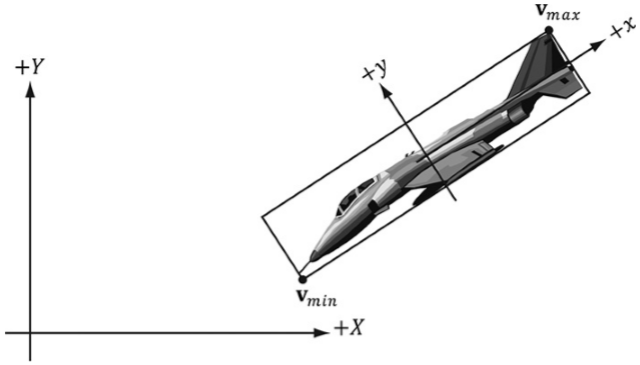
|  |
| --- |
| XMFLOAT3 vMinF3(FLT\_MAX, FLT\_MAX, FLT\_MAX);  XMFLOAT3 vMaxF3(-FLT\_MAX, -FLT\_MAX, -FLT\_MAX);  XMVECTOR vMin = XMLoadFloat3(&vMinF3);  XMVECTOR vMax = XMLoadFloat3(&vMaxF3);      **int** vertexCount = ARRAYSIZE(vertices);  **for** (**int** i = 0; i < vertexCount; ++i)  {  XMVECTOR pos = XMLoadFloat3(&vertices[i].position);  vMin = XMVectorMin(pos, vMin);  vMax = XMVectorMax(pos, vMax);  }      XMStoreFloat3(&(m\_bounds->Center), 0.5f\*(vMin + vMax));  XMStoreFloat3(&(m\_bounds->Extents), 0.5f\*(vMax - vMin)); |

XMVectorMin和XMVectorMax返回的向量分别为:

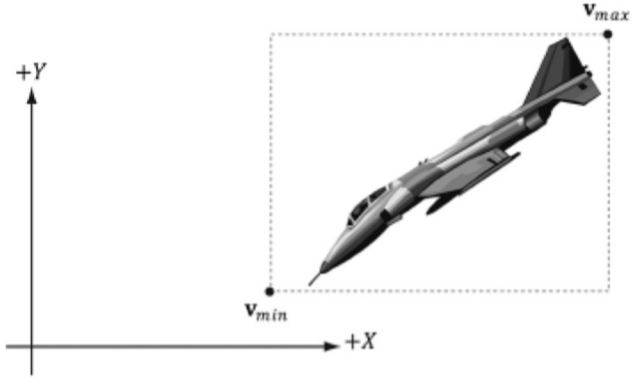


# 轴对齐包围盒的旋转操作

在某坐标系中的轴对齐包围盒，可能没有与其他不同坐标系的中坐标轴对齐。特别是局部空间中计算的AABB，需要将其变换到世界坐标系。实际工作中，我们通常是变换到局部坐标系中，再以局部空间内的轴对齐包围盒执行碰撞检测。



另一种方式是在世界坐标系中计算AABB，但可能得到一个与实际物体偏差较大的长方体。



我们也可以转而使用定向包围盒(oriented bounding box, OBB， 或者叫有向包围盒)。

我们在AABB的基础上保存其相对于世界空间的朝向即可。我们用四元数来表示旋转。

|  |
| --- |
| struct BoundingOrientedBox  {  **static** **const** size\_t CORNER\_COUNT = 8;      XMFLOAT3 Center; // Center of the box.  XMFLOAT3 Extents; // Distance from the center to each side.  XMFLOAT4 Orientation; // Unit quaternion representing rotation (box -> world).  ...  } |

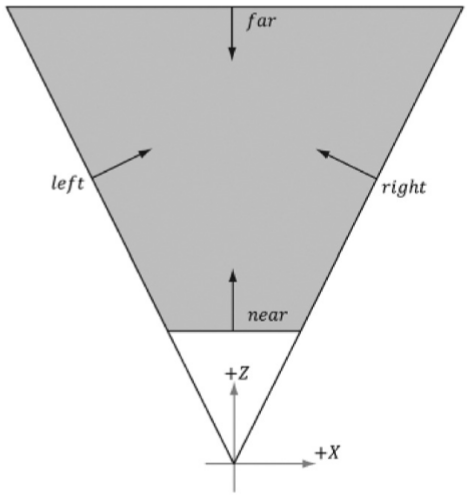
# 包围球

包围球是一种紧密围绕目标的球体。我们通过球心与半径来描述它。

一种方法是先计算其AABB，然后以中心作为球心，再以球心到网格体上任意点间的最大距离作为半径。

# 视锥体

我们知道视锥体由6个平面构成。



构建视锥体的一种简便方法是：在观察空间中以原定为中心，沿z轴正方向观察视锥体。此时可根据z轴上至原点的距离来确定近平面与远平面。而其余四个平面均与远近平面相交，且经过原点，我们就只需记录下这四个平面的斜率即可。

|  |
| --- |
| struct BoundingFrustum  {  **static** **const** size\_t CORNER\_COUNT = 8;      XMFLOAT3 Origin; // Origin of the frustum (and projection).  XMFLOAT4 Orientation; // Quaternion representing rotation.      **float** RightSlope; // Positive X (X/Z)  **float** LeftSlope; // Negative X  **float** TopSlope; // Positive Y (Y/Z)  **float** BottomSlope; // Negative Y  **float** Near, Far; // Z of the near plane and far plane.  ...  } |

我们也可以指定Origin和Orientation来在世界坐标中对其进行定位和旋转。

DirectX碰撞库也提供了通过投影矩阵计算观察空间中视锥体的方法**BoundingFrustum::CreateFromMatrix**

视锥体和包围体均有**Contains**方法来进行相交检测

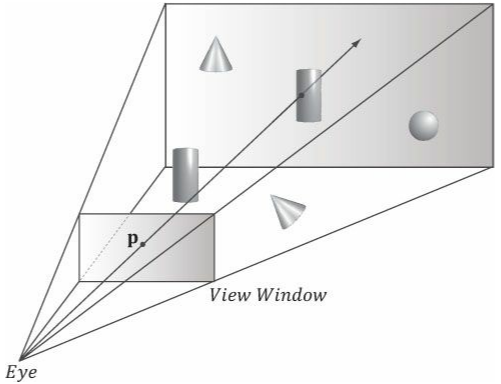
|  |
| --- |
| m\_cullingDataArray.clear();      BoundingFrustum frustum;  XMMATRIX proj = XMLoadFloat4x4(&m\_proj);  BoundingFrustum::CreateFromMatrix(frustum, proj);      XMMATRIX view = XMLoadFloat4x4(&m\_view);  XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(view), view);      **for** (**int** i = 0; i < m\_instancedDataArray.size(); ++i)  {  BoundingFrustum localFrustum;      XMMATRIX world = XMMatrixTranspose(XMLoadFloat4x4(&m\_instancedDataArray[i].World));  XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(world), world);      frustum.Transform(localFrustum, XMMatrixMultiply(invView, invWorld));      **if** (localFrustum.Contains(\*(m\_instanceCrate->m\_bounds)) != ContainmentType::DISJOINT)  {  m\_cullingDataArray.push\_back(m\_instancedDataArray[i]);  }  }      m\_d3dContext->UpdateSubresource(m\_instanceDataBuffer.Get(), 0, nullptr, m\_cullingDataArray.data(), 0, 0);      m\_d3dContext->DrawIndexedInstanced(m\_instanceCrate->m\_indexCount, m\_cullingDataArray.size(), 0, 0, 0); |

# 拾取

所谓拾取既通过用户鼠标的2D屏幕坐标，推断出位于该投影点上的3D物体。我们通常都是从3D空间变换到屏幕空间，要解决这一问题，我们则需要从屏幕空间变换回3D空间。

考虑下图所示的视域体。这里，p是屏幕坐标在投影窗口上的位置。现在，如果我们从观察点引出一条穿过点p的拾取射线，那么该射线将会与所有投影到点  上的物体相交，在本例中与射线相交的是圆柱体。

所以，我们的实现思路是：只要我们计算出一条拾取射线，就可以遍历场景中的每个物体，测试物体是否与该射线相交。与射线相交的物体就是被用户选中的物体。射线可能会与场景中的多个物体相交。我们可以将与摄像机距离最近的相交物体作为最终的拾取物体。



# 拾取射线

射线可以用原点和方向来表示。

|  |
| --- |
| XMFLOAT4X4 proj = m\_proj;      **float** vx = (+2.0f\*x / m\_outputWidth - 1.0f) / proj(0, 0);  **float** vy = (-2.0f\*y / m\_outputHeight + 1.0f) / proj(1, 1);      XMVECTOR origin = XMVectorSet(0.f, 0.f, 0.f, 1.f);  XMVECTOR dir = XMVectorSet(vx, vy, 1.f, 0.f); |

这里计算的只是观察空间拾取射线。与视锥体剔除一样，我们需要将其变换到局部空间后再进行相交测试。

|  |
| --- |
| XMMATRIX view = XMLoadFloat4x4(&m\_view);  XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(view), view);      XMMATRIX world = XMMatrixTranspose(XMLoadFloat4x4(&m\_instancedDataArray[i].World));  XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(world), world);      XMMATRIX viewToLocal = XMMatrixMultiply(invView, invWorld);      XMVECTOR localOrigin = XMVector3TransformCoord(origin, viewToLocal);  XMVECTOR localDir = XMVector3TransformNormal(dir, viewToLocal);  localDir = XMVector3Normalize(localDir); |

我们使用**XMVector3TransformCoord**和**XMVector3TransformNormal**来变换3D向量，区别是XMVector3TransformCoord将第四个分量w视为1，而XMVector3TransformNormal视为0。故我们用XMVector3TransformCoord来变换点，用XMVector3TransformNormal来变换向量。

# 射线网格相交测试

如果我们对每个三角形进行相交测试，则其计算时间与三角形数量成正比，三角形数量越多，计算时间也就越 长。即使网格与拾取射线之间的距离很远，我们仍然要通过遍历每个三角形来决定射线是否与网格相交；这是有悖于效率的做法。我们可以先用外接体来代替原始网格进行射线相交测试。如果射线未穿过外接体，那么它也不会穿过三角形网格，这样我们就可以节省很多计算时间。反之，如果射线穿过了外接体，那么我们再做进一步的相交测试。如果射线与场景中的大多数外接体都不相交，那么我们就可以少做很多次射线/三角形相交测试。多个三角形通过测试时，我们取距离最近的一个作为最终结果。

|  |
| --- |
| **float** tmin = FLT\_MAX;  bPickSuccess = **false**;  **for** (**int** i = 0; i < m\_instancedDataArray.size(); ++i)  {  XMMATRIX world = XMMatrixTranspose(XMLoadFloat4x4(&m\_instancedDataArray[i].World));  XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(world), world);      XMMATRIX viewToLocal = XMMatrixMultiply(invView, invWorld);      XMVECTOR localOrigin = XMVector3TransformCoord(origin, viewToLocal);  XMVECTOR localDir = XMVector3TransformNormal(dir, viewToLocal);  localDir = XMVector3Normalize(localDir);      **float** tb = 0.f;  **if** (m\_instanceCrate->m\_bounds->Intersects(localOrigin, localDir, tb))  {  **for** (**int** j = 0; j < m\_instanceCrate->m\_indices.size(); j += 3)  {  UINT i0 = m\_instanceCrate->m\_indices[j];  UINT i1 = m\_instanceCrate->m\_indices[j+1];  UINT i2 = m\_instanceCrate->m\_indices[j+2];      XMVECTOR v0 = XMLoadFloat3(&m\_instanceCrate->m\_vertices[i0].position);  XMVECTOR v1 = XMLoadFloat3(&m\_instanceCrate->m\_vertices[i1].position);  XMVECTOR v2 = XMLoadFloat3(&m\_instanceCrate->m\_vertices[i2].position);      **float** t = FLT\_MAX;  **if** (TriangleTests::Intersects(localOrigin, localDir, v0, v1, v2, t))  {  **if** (t < tmin)  {  bPickSuccess = **true**;  tmin = t;  m\_pickedInstaceIndex = i;  m\_pickedTriangleIndex = j;  }  }  }  }  } |