Week 11

Instancing And Frustum Culling

Hooman Salamat

Objectives

1. To learn how to implement hardware instancing.

2. To become familiar with bounding volumes, why they are useful, how to create them, and how to use them.

3. To discover how to implement frustum culling.

HARDWARE INSTANCING

Instancing refers to drawing the same object more than once in a scene, but with different positions, orientations, scales, materials, and textures.

Examples:

1. A few different tree models are drawn multiple times to build a forest.

2. A few different asteroid models are drawn multiple times to build an asteroid field.

3. A few different character models are drawn multiple times to build a crowd of people.

So far, we store a single copy of the geometry (i.e., vertex and index lists) relative to the object's local space. Then we draw the object several times, but each time with a different world matrix and a different material if additional variety is desired.

It still requires per-object API overhead: we must set its unique material and its world matrix.

The Direct3D 12 instancing API allows you to instance an object multiple times with a single draw call with dynamic indexing.

Drawing Instanced Data

In our previous examples, the instance count has always been 1.

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

The next step is to figure out how to specify additional per-instance data so that we can vary the instances by rendering them with different transforms, materials, and textures.

When creating an input layout, you can specify that data streams in per-instance rather than at a per-vertex frequency by using D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA

You would then bind a secondary vertex buffer to the input stream that contained the instancing data.

The modern approach is to create a structured buffer that contains the per-instance data for all of our instances.

How do we know which instance is being drawn in the vertex shader? SV_InstanceID , you can use it in your vertex shader.

For example, vertices of the first instance will have id 0, vertices of the second instance will have id 1, and so on.

```
struct InstanceData
{
float4x4 World;
float4x4 TexTransform;
uint     MaterialIndex;
uint     InstPad0;
uint     InstPad1;
uint     InstPad2;
};

struct MaterialData
{
float4   DiffuseAlbedo;
    float3   FresnelR0;
    float    Roughness;
float4x4 MatTransform;
uint     DiffuseMapIndex;
uint     MatPad0;
uint     MatPad1;
uint     MatPad2;
};
```

The HLSL file

Note that we no longer have a per-object constant buffer. The per-object data comes from the instance buffer. Observe also how we use dynamic indexing to associate a different material for each instance, and a different texture.

```
Texture2D gDiffuseMap[7] : register(t0);

StructuredBuffer<InstanceData>
gInstanceData : register(t0, space1);

StructuredBuffer<MaterialData>
gMaterialData : register(t1, space1);
```

```
VertexOut VS(VertexIn vin, uint instanceID : SV_InstanceID)
{
VertexOut vout = (VertexOut)0.0f;

// Fetch the instance data.
InstanceData instData = gInstanceData[instanceID];
float4x4 world = instData.World;
float4x4 texTransform = instData.TexTransform;
uint matIndex = instData.MaterialIndex;

vout.MatIndex = matIndex;

// Fetch the material data.
MaterialData matData = gMaterialData[matIndex];

    // Transform to world space.
…..
}

float4 PS(VertexOut pin) : SV_Target
{
// Fetch the material data.
MaterialData matData = gMaterialData[pin.MatIndex];
float4 diffuseAlbedo = matData.DiffuseAlbedo;
    float3 fresnelR0 = matData.FresnelR0;
    float  roughness = matData.Roughness;
uint diffuseTexIndex = matData.DiffuseMapIndex;

// Dynamically look up the texture in the array.
    diffuseAlbedo *= gDiffuseMap[diffuseTexIndex].Sample(gsamLinearWrap, pin.TexC);
```

BuildRootSignature

The following corresponding root signature description is shown corresponds to the shader programs:

```
void InstancingAndCullingApp::BuildRootSignature()
{
CD3DX12_DESCRIPTOR_RANGE texTable;
texTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 7, 0, 0);

    // Root parameter can be a table, root descriptor or root constants.
    CD3DX12_ROOT_PARAMETER slotRootParameter[4];

// Perfomance TIP: Order from most frequent to least frequent.
    slotRootParameter[0].InitAsShaderResourceView(0, 1);
    slotRootParameter[1].InitAsShaderResourceView(1, 1);
    slotRootParameter[2].InitAsConstantBufferView(0);
slotRootParameter[3].InitAsDescriptorTable(1, &texTable,
D3D12_SHADER_VISIBILITY_PIXEL);

auto staticSamplers = GetStaticSamplers();

    // A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(4, slotRootParameter,
(UINT)staticSamplers.size(), staticSamplers.data(),
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

Draw

We bind all the scene materials and textures once per-frame, and the only per draw call resource we need to set is the structured buffer with the instanced data:

```
void InstancingAndCullingApp::Draw(const GameTimer& gt)

{

    auto cmdListAlloc = mCurrFrameResource->CmdListAlloc;


……
// Bind all the materials used in this scene.  For structured buffers, we can bypass the heap and  set as
a root descriptor.
auto matBuffer = mCurrFrameResource->MaterialBuffer->Resource();
mCommandList->SetGraphicsRootShaderResourceView(1, matBuffer->GetGPUVirtualAddress());

auto passCB = mCurrFrameResource->PassCB->Resource();
mCommandList->SetGraphicsRootConstantBufferView(2, passCB->GetGPUVirtualAddress());

// Bind all the textures used in this scene.
mCommandList->SetGraphicsRootDescriptorTable(3, mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());

    DrawRenderItems(mCommandList.Get(), mOpaqueRitems);
```

DrawRenderItems

```
void InstancingAndCullingApp::DrawRenderItems(ID3D12GraphicsCommandList* cmdList, const std::vector<RenderItem*>& ritems)
{
    // For each render item...
    for(size_t i = 0; i < ritems.size(); ++i)
    {
        auto ri = ritems[i];

        cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
        cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
        cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

// Set the instance buffer to use for this render-item.  For structured buffers, we can bypass
// the heap and instead set it as a root descriptor.
auto instanceBuffer = mCurrFrameResource->InstanceBuffer->Resource();
mCommandList->SetGraphicsRootShaderResourceView(0, instanceBuffer->GetGPUVirtualAddress());

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

Creating the Instanced Buffer

The instance buffer stores the data that varies per-instance. It looks a lot like the data we previously put in our per-object constant buffer. On the CPU side, our instance data structure looks like this:

-----FrameResource.h---------

```
struct InstanceData
{
DirectX::XMFLOAT4X4 World =
MathHelper::Identity4x4();
DirectX::XMFLOAT4X4 TexTransform =
MathHelper::Identity4x4();
UINT MaterialIndex;
UINT InstancePad0;
UINT InstancePad1;
UINT InstancePad2;
};
```

The per-instance data in system memory is stored as part of the render-item structure, as the render-item maintains how many times it should be instanced:

---------------InstancingAndCullingApp.cpp---------------------------------

```
// Lightweight structure stores parameters to draw a shape.  This will
// vary from app-to-app.
struct RenderItem
{
RenderItem() = default;
RenderItem(const RenderItem& rhs) = delete;
XMFLOAT4X4 World = MathHelper::Identity4x4();
XMFLOAT4X4 TexTransform = MathHelper::Identity4x4();
…..
std::vector<InstanceData> Instances;

// DrawIndexedInstanced parameters.
UINT IndexCount = 0;
UINT InstanceCount = 0;
UINT StartIndexLocation = 0;
int BaseVertexLocation = 0;
};
```

Creating a dynamic buffer for InstanceData

For the GPU to consume the instance data, we need to create a structured buffer with element type InstanceData. Moreover, this buffer will be dynamic (i.e., an upload buffer) so that we can update it every frame.

we copy the instanced data of only the *visible* instances into the structure buffer (this is related to frustum culling), and the set of visible instances will change as the camera moves/looks around.

Creating a dynamic buffer is simple with our UploadBuffer helper class:

```
// Stores the resources needed for the CPU to build the command lists for a frame.
struct FrameResource
{
public:

FrameResource(ID3D12Device* device, UINT passCount, UINT maxInstanceCount, UINT
materialCount);
FrameResource(const FrameResource& rhs) = delete;
FrameResource& operator=(const FrameResource& rhs) = delete;
~FrameResource();

Microsoft::WRL::ComPtr<ID3D12CommandAllocator> CmdListAlloc;
std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
std::unique_ptr<UploadBuffer<MaterialData>> MaterialBuffer = nullptr;

std::unique_ptr<UploadBuffer<InstanceData>> InstanceBuffer = nullptr;

UINT64 Fence = 0;
};
```

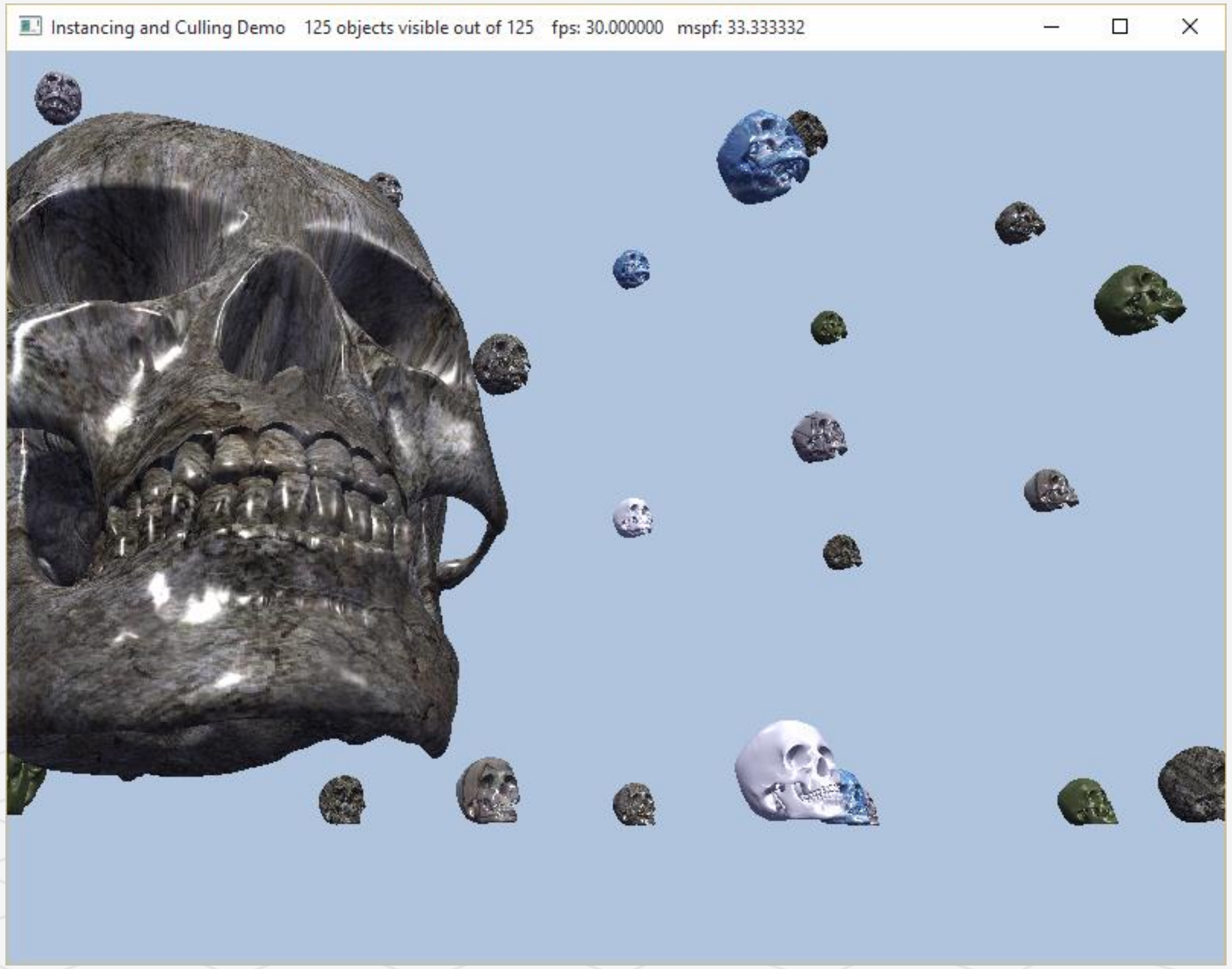
InstanceBuffer

Note that InstanceBuffer is not a constant buffer, so we specify false for the last parameter.

```
FrameResource::FrameResource(ID3D12Device* device, UINT passCount, UINT
maxInstanceCount, UINT materialCount)

{

ThrowIfFailed(device->CreateCommandAllocator(D3D12_COMMAND_LIST_TYPE_DIRECT,

IID_PPV_ARGS(CmdListAlloc.GetAddressOf())));

PassCB = std::make_unique<UploadBuffer<PassConstants>>(device, passCount, true);

MaterialBuffer = std::make_unique<UploadBuffer<MaterialData>>(device, materialCount,
false);

InstanceBuffer = std::make_unique<UploadBuffer<InstanceData>>(device,
maxInstanceCount, false);

}
```

Instancing Demo

we submit all 125 instances to the rendering pipeline for processing.


Instancing and Culling Demo 125 objects visible out of 125 fps: 30.000000 mspf: 33.333332

BOUNDING VOLUMES AND FRUSTUMS

Bounding volumes are primitive geometric objects that approximate the volume of an object.

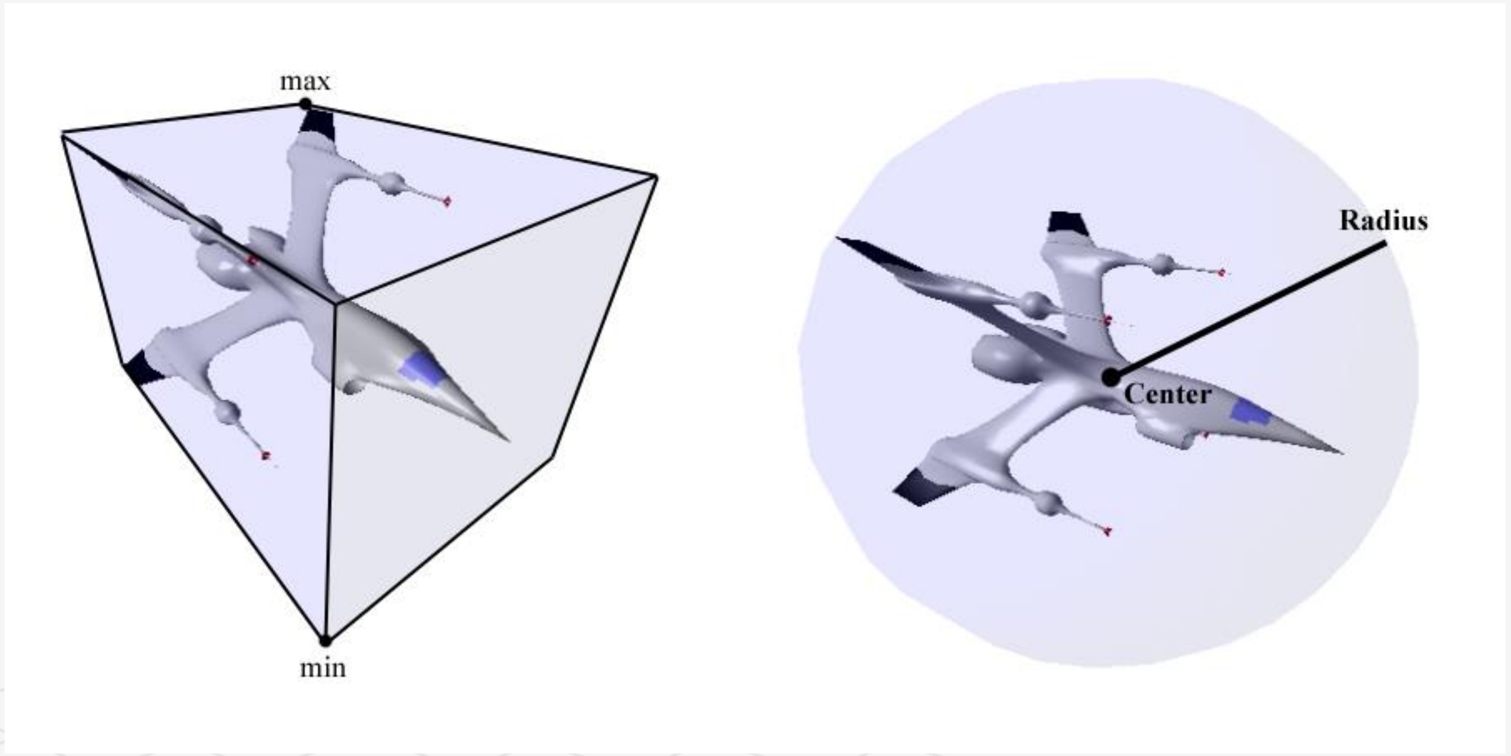
In many applications the bounding box is aligned with the axes of the coordinate system, and it is then known as an **axis-aligned bounding box** (**AABB**).

(AABB) of a mesh is a box that tightly surrounds the mesh and such that its faces are parallel to the major axes.

To distinguish the general case from an AABB, an arbitrary bounding box is sometimes called an **oriented bounding box** (**OBB**), or an **OOBB** when an existing object's local coordinate system is used.

The figures show a mesh rendered with its AABB and bounding sphere.

We use the *DirectXCollision.h* utility library, which is part of DirectX Math. This library provides fast implementations to common geometric primitive intersection tests such as ray/triangle intersection, ray/box intersection, box/box intersection, box/plane intersection, box/frustum, sphere/frustum, and much more.

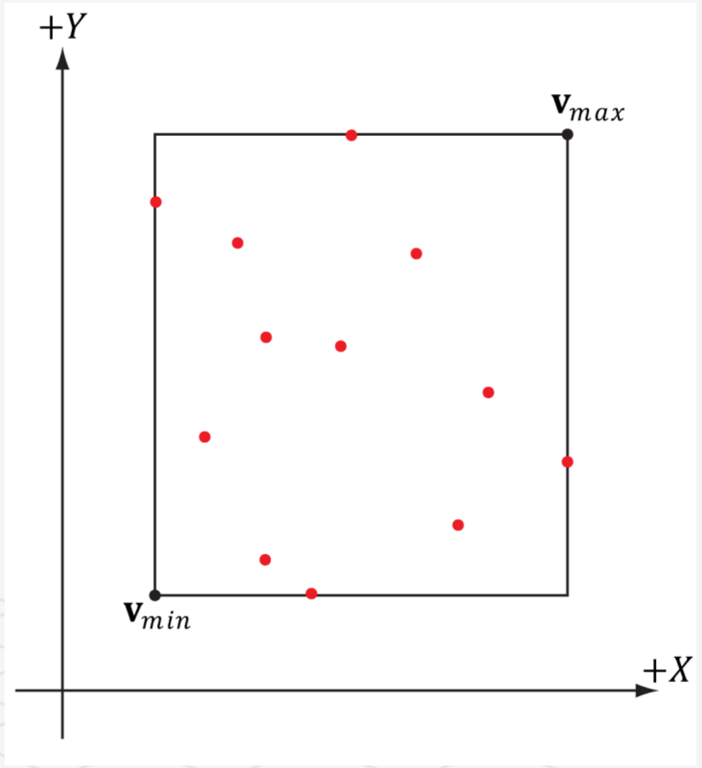


The axis-aligned bounding box (AABB)

The *axis-aligned bounding box* (AABB).

An AABB can be described by a minimum point $\mathbf{v}_{min}$ and a maximum point $\mathbf{v}_{max}$.

The minimum point $\mathbf{v}_{min}$ is found by searching through all the vertices of the mesh and finding the minimum x-, y-, and z-coordinates, and the maximum point $\mathbf{v}_{max}$ is found by searching through all the vertices of the mesh and finding the maximum x-, y-, and z-coordinates.

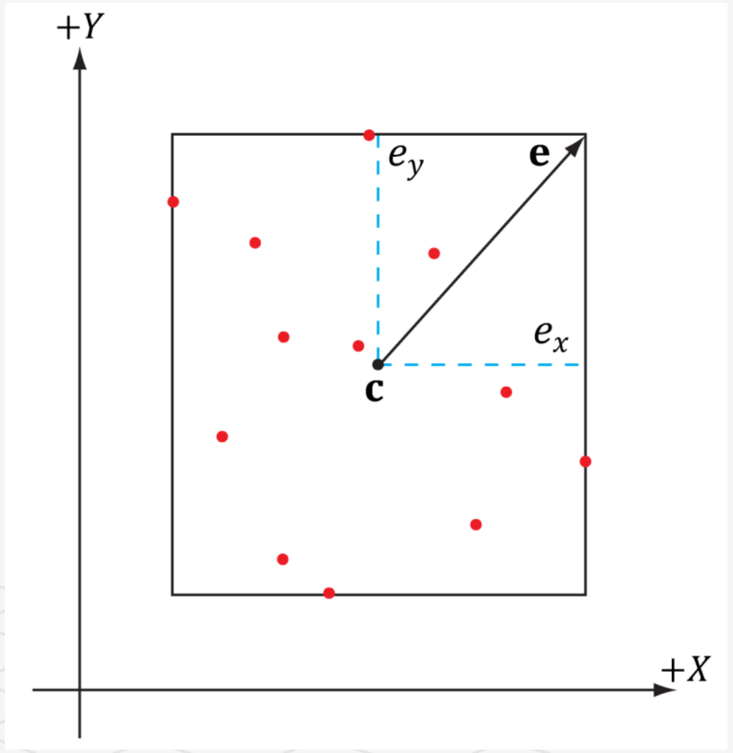


Center point and extents vector

Alternatively, an AABB can be represented with the box center point **c** and *extents* vector **e**, which stores the distance from the center point to the sides of the box along the coordinate axes.

The DirectX collision library uses the center/extents representation:

```
struct BoundingBox

{

static const size_t CORNER_COUNT = 8;

XMFLOAT3 Center;     // Center of the box.

XMFLOAT3 Extents; // Distance from the center to each side.
```



XMVectorMin and XMVectorMax functions

Makes a per-component comparison between two vectors, and returns a vector containing the smallest components.

XMVECTOR XM_CALLCONV XMVectorMin(FXMVECTOR V1, FXMVECTOR V2);

The following pseudocode demonstrates the operation of the function:

```
XMVECTOR Result;

Result.x = (V1.x < V2.x) ? V1.x : V2.x;

Result.y = (V1.y < V2.y) ? V1.y : V2.y;

Result.z = (V1.z < V2.z) ? V1.z : V2.z;

 Result.w = (V1.w < V2.w) ? V1.w : V2.w;

return Result;
```

Makes a per-component comparison between two vectors, and returns a vector containing the largest components.

XMVECTOR XM_CALLCONV XMVectorMax(FXMVECTOR V1, FXMVECTOR V2);

The following pseudocode demonstrates the operation of the function:

```
XMVECTOR Result;

Result.x = (V1.x > V2.x) ? V1.x : V2.x;

Result.y = (V1.y > V2.y) ? V1.y : V2.y;

Result.z = (V1.z > V2.z) ? V1.z : V2.z;

 Result.w = (V1.w > V2.w) ? V1.w : V2.w;

return Result;
```

The bounding box of the skull mesh

It is easy to convert from one representation to the other. For example, given a bounding box defined by $\boldsymbol{v}_{min}\ and\ \boldsymbol{v}_{max}$, the center/extents representation is given by:

$$\boldsymbol{c} = \mathbf{0.5}(\boldsymbol{v}_{max} + \boldsymbol{v}_{min})$$

$$\boldsymbol{e} = \mathbf{0.5}(\boldsymbol{v}_{max} - \boldsymbol{v}_{min})$$

The following code shows how we compute the bounding box of the skull mesh.

The XMVectorMin and XMVectorMax functions return the vectors:

$$\mathbf{min}(\mathbf{u},\mathbf{v}) = \left(\min(u_x, v_x), \min(u_y, v_y), \min(u_z, v_z), \min(u_w, v_w)\right)$$

$$\mathbf{max}(\mathbf{u},\mathbf{v}) = \left(\max(u_x, v_x), \max(u_y, v_y), \max(u_z, v_z), \max(u_w, v_w)\right)$$

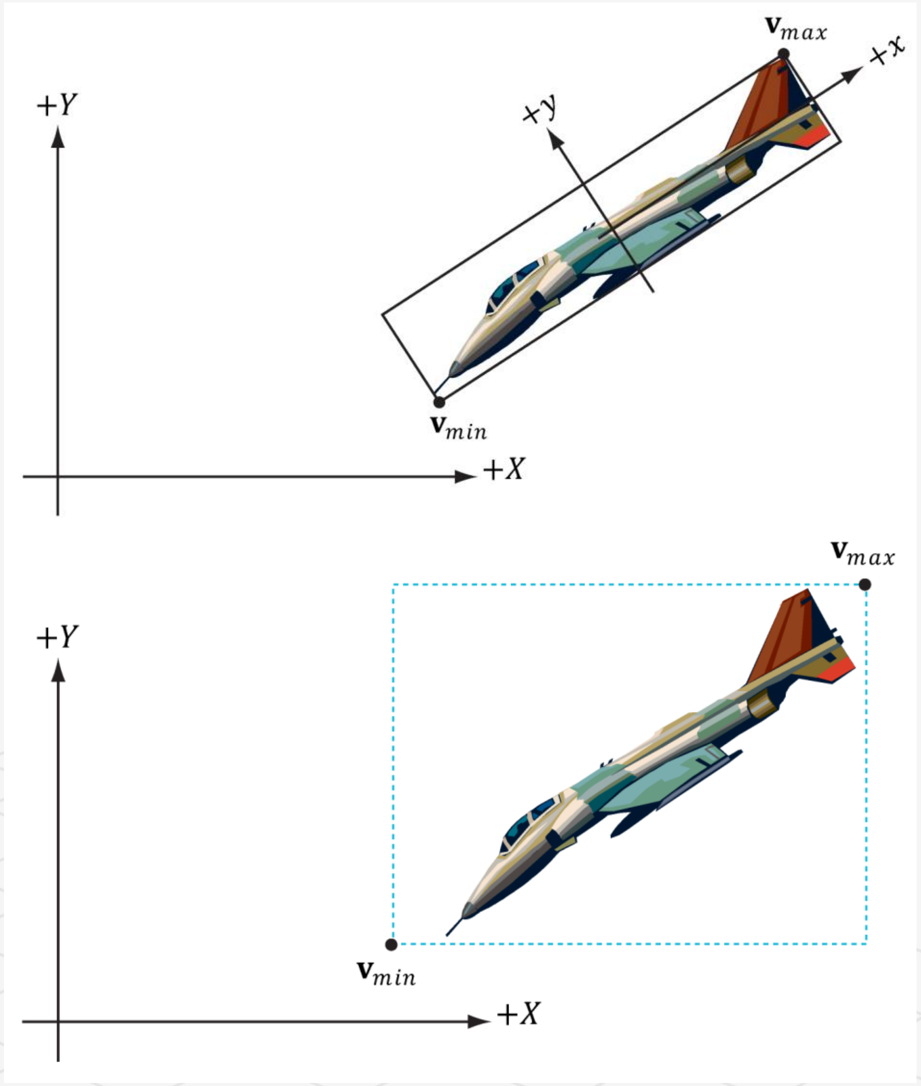
```cpp
void InstancingAndCullingApp::BuildSkullGeometry()
{
std::ifstream fin("Models/skull.txt");

if(!fin)
{
MessageBox(0, L"Models/skull.txt not found.", 0, 0);
return;
}

UINT vcount = 0;
UINT tcount = 0;
std::string ignore;

fin >> ignore >> vcount;
fin >> ignore >> tcount;
fin >> ignore >> ignore >> ignore >> ignore;

XMFLOAT3 vMinf3(+MathHelper::Infinity, +MathHelper::Infinity, +MathHelper::Infinity);
XMFLOAT3 vMaxf3(-MathHelper::Infinity, -MathHelper::Infinity, -MathHelper::Infinity);

XMVECTOR vMin = XMLoadFloat3(&vMinf3);
XMVECTOR vMax = XMLoadFloat3(&vMaxf3);
…..….
vMin = XMVectorMin(vMin, P);
vMax = XMVectorMax(vMax, P);
}
BoundingBox bounds;
XMStoreFloat3(&bounds.Center, 0.5f*(vMin + vMax));
XMStoreFloat3(&bounds.Extents, 0.5f*(vMax - vMin));
```

Rotations and Axis-Aligned Bounding Boxes

A box axis-aligned in one coordinate system may not be axis aligned with a different coordinate system.

If we compute the AABB of a mesh in local space, it gets transformed to an *oriented bounding box* (OBB) in world space. However, we can always transform into the local space of the mesh and do the intersection there where the box is axis-aligned.

When we can recompute the AABB in the world space, this can result in a “fatter” box which is a poorer approximation to the actual volume.



OBB

Yet another alternative is to abandon axis-aligned bounding boxes, and just work with oriented bounding boxes, where we maintain the orientation of the box relative to the world space.

The DirectX collision library provides the following structure for representing an oriented bounding box.

```
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).
```

CreateFromPoints

An AABB and OBB can also be constructed from a set of points using the DirectX collision library with the following *static* member functions:

```
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);
```

Example

If your vertex structure looks like this:

```
struct Basic32

{

XMFLOAT3 Pos;

XMFLOAT3 Normal;

XMFLOAT2 TexC;

};
```

```
And you have an array of vertices forming your mesh :

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

Then you call this function like so :

DirectX::BoundingBox box;

DirectX::BoundingBox::CreateFromPoints(

box,

vertices.size(),

&vertices[0].Pos,

sizeof(Vertex::Basic32));
```

The stride indicates how many bytes to skip to get to the next position element.

Spheres

The bounding sphere of a mesh is a sphere that tightly surrounds the mesh.

A bounding sphere can be described with a center point and radius. One way to compute the bounding sphere of a mesh is to first compute its AABB.

We then take the center of the AABB as the center of the bounding sphere:

$$\boldsymbol{c} = 0.5(\boldsymbol{v}_{max} + \boldsymbol{v}_{min})$$

The radius is then taken to be the maximum distance between any vertex **p** in the mesh from the center **c**:

$$r = \max\{\|\mathbf{c}-\mathbf{p}\| : \mathbf{p} \in mesh\}$$

Suppose we compute the bounding sphere of a mesh in local space.

After the world transform, the bounding sphere may not tightly surround the mesh due to scaling.

Therefore the radius needs to be rescaled accordingly.

Another possible strategy is to avoid scaling all together by having all your meshes modeled to the same scale of the game world.

This way, models will not need to be rescaled once loaded into the application.

```
struct BoundingSphere

{

    XMFLOAT3 Center;            // Center of the sphere.

    float Radius;               // Radius of the sphere.
```

And it provides the following *static* member function for creating one from a set of points:

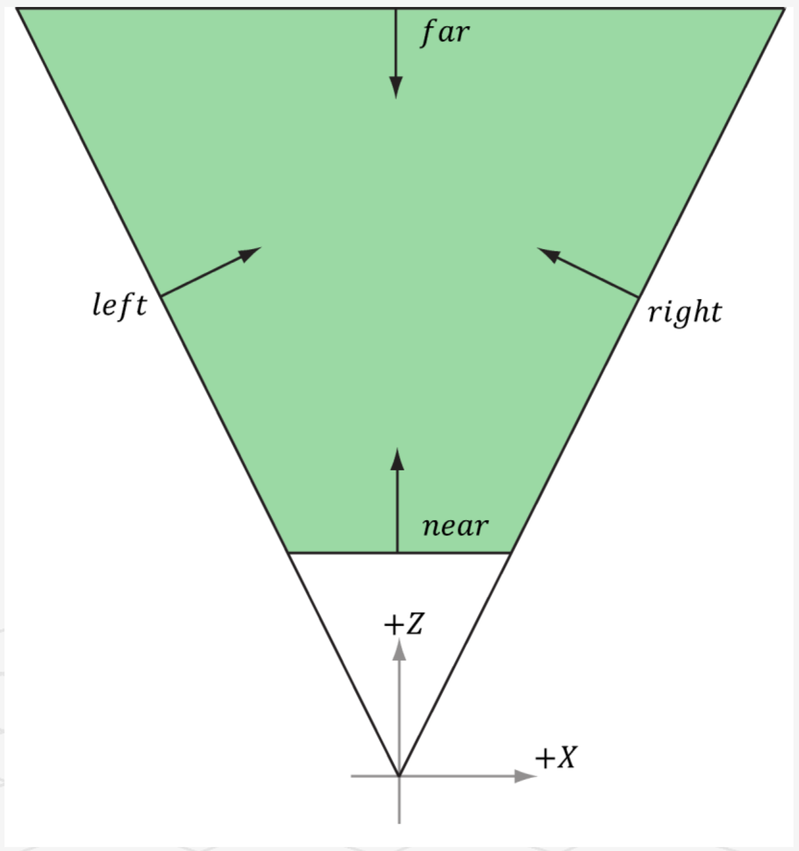
```
 static void CreateFromPoints( _Out_ BoundingSphere& Out, _In_ size_t Count,
_In_reads_bytes_(sizeof(XMFLOAT3)+Stride*(Count-1)) const XMFLOAT3* pPoints, _In_ size_t
Stride );
```

Frustums

One way to specify a frustum mathematically is as the intersection of six planes:

the left/right planes, the top/bottom planes, and the near/far planes.

We assume the six frustum planes are "inward" facing.



Constructing the Frustum Planes

The DirectX collision library provides the following structure for representing a frustum:

```
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.
```

In the local space of the frustum (e.g., view space for the camera), the Origin would be zero, and the Orientation would represent an identity transform (no rotation).

We can position and orientate the frustum somewhere in the world by specifying an Origin position and Orientation quaternion.

Corners of the projection frustum in homogenous space:

```
static XMVECTORF32 HomogenousPoints[6] =
{
{ 1.0f, 0.0f, 1.0f, 1.0f }, // right (at far plane)
{ -1.0f, 0.0f, 1.0f, 1.0f }, // left
{ 0.0f, 1.0f, 1.0f, 1.0f }, // top
{ 0.0f, -1.0f, 1.0f, 1.0f }, // bottom
{ 0.0f, 0.0f, 0.0f, 1.0f }, // near
{ 0.0f, 0.0f, 1.0f, 1.0f } // far
};
```

CreateFromMatrix

The following DirectX collision code computes the frustum in view space from a projection matrix:

```
//-----------------------------------------------------------------------------
// Build a frustum from a persepective projection matrix.  The matrix may only
// contain a projection; any rotation, translation or scale will cause the
// constructed frustum to be incorrect.
//-----------------------------------------------------------------------------
_Use_decl_annotations_
inline void XM_CALLCONV BoundingFrustum::CreateFromMatrix( BoundingFrustum& Out, FXMMATRIX Projection )
{
    // Corners of the projection frustum in homogenous space.
    static XMVECTORF32 HomogenousPoints[6] =
    {
        { { {  1.0f,  0.0f, 1.0f, 1.0f } } },   // right (at far plane)
        { { { -1.0f,  0.0f, 1.0f, 1.0f } } },   // left
        { { {  0.0f,  1.0f, 1.0f, 1.0f } } },   // top
        { { {  0.0f, -1.0f, 1.0f, 1.0f } } },   // bottom

        { { { 0.0f, 0.0f, 0.0f, 1.0f } } },     // near
        { { { 0.0f, 0.0f, 1.0f, 1.0f } } }      // far
    };
```

Frustum/Sphere Intersection

For frustum culling, one test we will want to perform is a frustum/sphere intersection test.

If there exists a frustum plane *L* such that the sphere is in the negative half-space of *L*, then we can conclude that the sphere is completely outside the frustum.

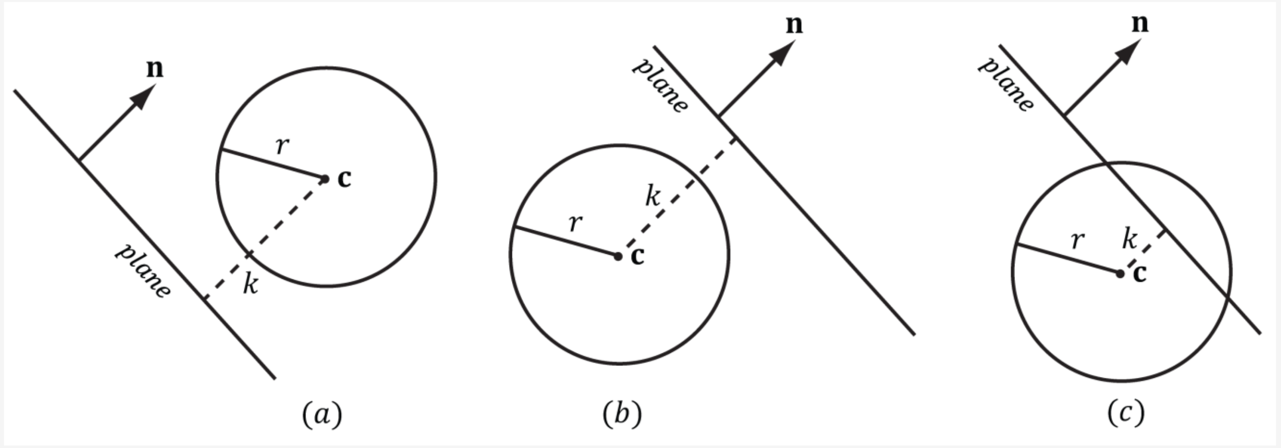
If such a plane does not exist, then we conclude that the sphere intersects the frustum.

Let the sphere have center point c and radius *r*. Then the signed distance from the center of the sphere to the plane is $k = n \cdot c + d$

If $|k| \leq r$, then the sphere intersects the plane.

If $k < -r$, then the sphere is behind the plane.

If $k > r$, then the sphere is in front of the plane and the sphere intersects the positive half-space of the plane.



Frustum/AABB Intersection

The frustum/AABB intersection test follows the same strategy as the frustum/sphere test.

Because we model a frustum as six inward facing planes, a frustum/AABB test can be stated as follows:

If there exists a frustum plane *L* such that the box is in the negative halfspace of *L*, then we can conclude that the box is completely outside the frustum.

If such a plane does not exist, then we conclude that the box intersects the frustum.

So a frustum/AABB intersection test reduces to six AABB/plane tests.

The figure shows AABB/plane intersection test. The diagonal $\overrightarrow{PQ}$ is always the diagonal most directed with the plane normal.

The algorithm for an AABB/plane test is as follows.

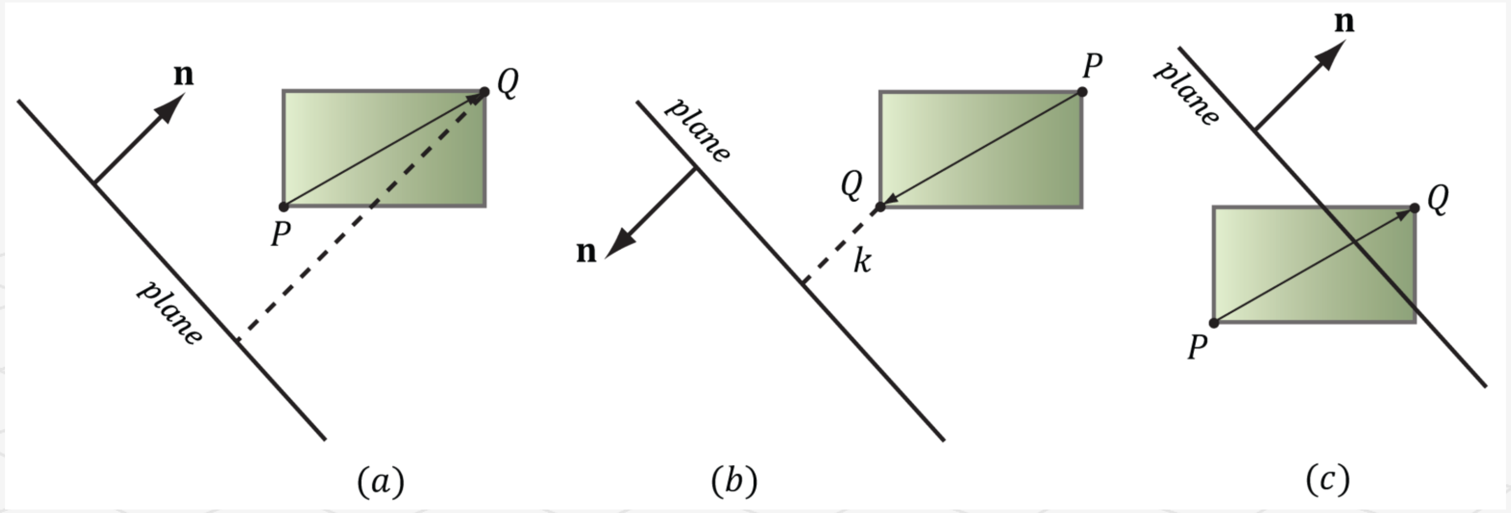
1. Find the box diagonal vector $v = \overrightarrow{PQ}$

2. Pass through the center of the box, that is most aligned with the plane normal **n**.

3. (a) if *P* is in front of the plane, then *Q* must be also in front of the plane;

4. (b) if *Q* is behind the plane, then *P* must be also be behind the plane;

5. (c) if *P* is behind the plane and *Q* is in front of the plane, then the box intersects the plane.



Bounding Frustum

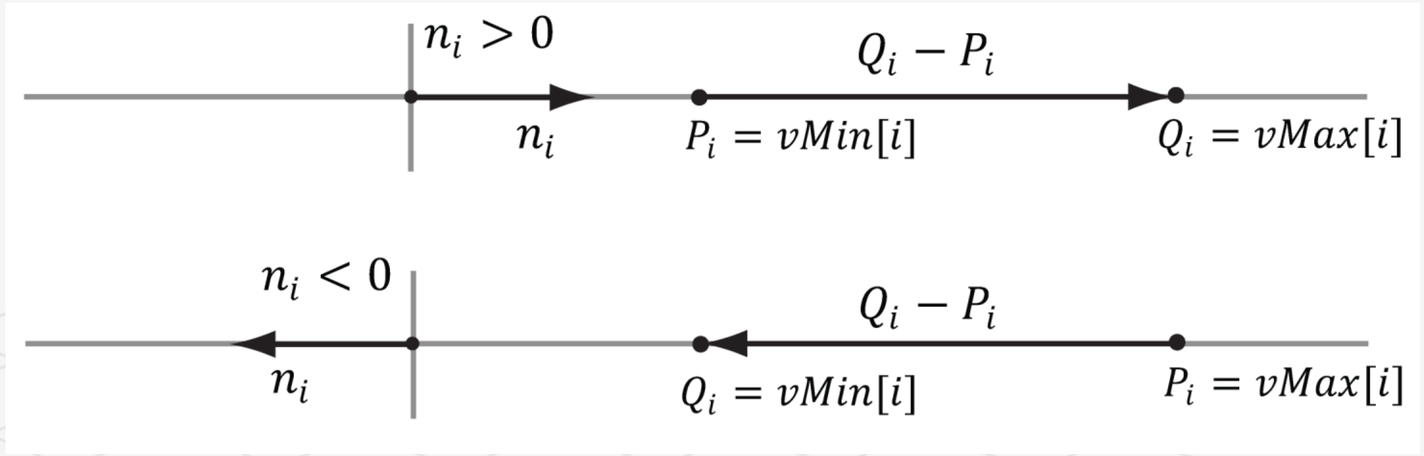
Finding *PQ* most aligned with the plane normal vector **n** can be done with the following code:

```
// For each coordinate axis x, y, z…
for (int j = 0; j < 3; ++j)
{
// Make PQ point in the same direction as
// the plane normal on this axis.
if (planeNormal[j] >= 0.0f)
{
P[j] = box.minPt[j];
Q[j] = box.maxPt[j];
}
else
{
P[j] = box.maxPt[j];
Q[j] = box.minPt[j];
}
}
```

This code just looks at one dimension at a time, and chooses P_i and Q_i such that $Q_i - P_i$ has the same sign as the plane normal coordinate n_i.

(Top) The normal component along the ith axis is positive, so we choose $P_i = vMin[i]$ and $Q_i = vMax[i]$ so that $Q_i - P_i$ has the same sign as the plane normal coordinate n_i.

(Bottom) The normal component along the *i*th axis is negative, so we choose $P_i = vMax[i]$ and $Q_i = vMin[i]$ so that $Q_i - P_i$ has the same sign as the plane normal coordinate n_i.



Frustum/Sphere Intersection

The BoundingFrustum class provides the following member function to test if a sphere intersects a frustum. Note that the sphere and frustum must be in the same coordinate system for the test to make sense.

```
enum ContainmentType

{

    DISJOINT = 0,

    INTERSECTS = 1,

    CONTAINS = 2

};
```

ContainmentType indicates whether an object contains another object.

DISJOINT :The object does not contain the specified object

INTERSECTS : The objects intersect

CONTAINS : The object contains the specified object

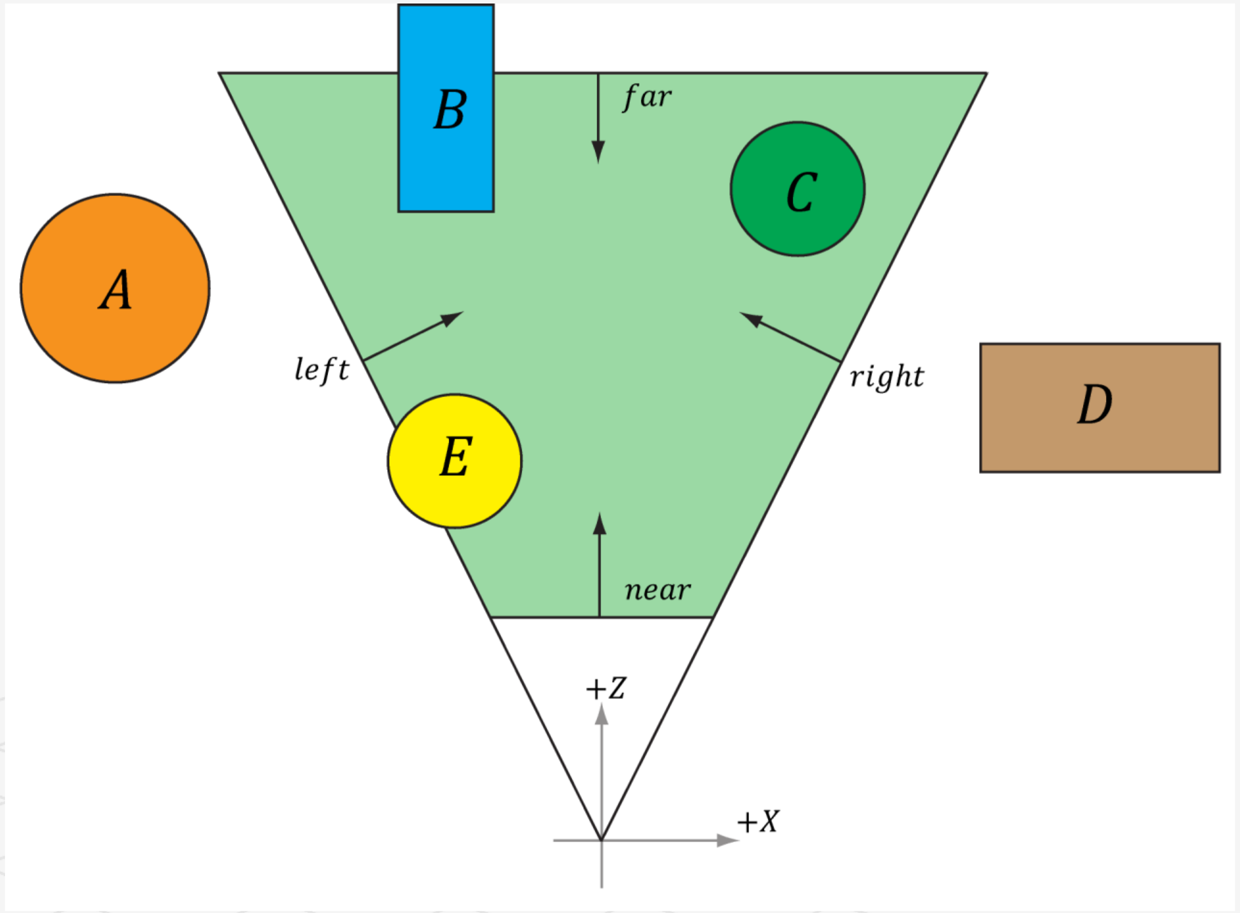
```
inline ContainmentType BoundingFrustum::Contains( const BoundingFrustum& fr )
const
```

FRUSTUM CULLING

The idea of frustum culling is for the application code to cull groups of triangles at a higher level than on a per-triangle basis.

We build a bounding volume, such as a sphere or box, around each object in the scene. If the bounding volume does not intersect the frustum, then we do not need to submit the object (which could contain thousands of triangles) to Direct3D for drawing.

The objects bounded by volumes *A* and *D* are completely outside the frustum, and so do not need to be drawn. The object corresponding to volume *C* is completely inside the frustum, and needs to be drawn. The objects bounded by volumes *B* and *E* are partially outside the frustum and partially inside the frustum; we must draw these objects and let the hardware clip and triangles outside the frustum.



“Instancing and Culling” demo

In our demo, we render a 5 × 5 × 5 grid of skull meshes) using instancing.

We compute the AABB of the skull mesh in local space.

In the UpdateInstanceData method, we perform frustum culling on all of our instances.

If the instance intersects the frustum, then we add it to the next available slot in our structured buffer containing the instance data and increment the visibleInstanceCount counter.

This way, the front of the structured buffer contains the data for all the visible instances.

```
void InstancingAndCullingApp::UpdateInstanceData(const GameTimer& gt)
{
XMMATRIX view = mCamera.GetView();
XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(view), view);

auto currInstanceBuffer = mCurrFrameResource->InstanceBuffer.get();
for(auto& e : mAllRitems)
{
const auto& instanceData = e->Instances;

int visibleInstanceCount = 0;

for(UINT i = 0; i < (UINT)instanceData.size(); ++i)
{
XMMATRIX world = XMLoadFloat4x4(&instanceData[i].World);
XMMATRIX texTransform = XMLoadFloat4x4(&instanceData[i].TexTransform);

XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(world), world);

// View space to the object's local space.
XMMATRIX viewToLocal = XMMatrixMultiply(invView, invWorld);

// Transform the camera frustum from view space to the object's local space.
BoundingFrustum localSpaceFrustum;
mCamFrustum.Transform(localSpaceFrustum, viewToLocal);

// Perform the box/frustum intersection test in local space.
if((localSpaceFrustum.Contains(e->Bounds) != DirectX::DISJOINT) ||
(mFrustumCullingEnabled==false))
{
```

void InstancingAndCullingApp::UpdateInstanceData

```
typedef enum ContainmentType { DISJOINT,
INTERSECTS, CONTAINS } ;
```

Of course, the structured buffer is sized to match the number of instances in case all the instances are visible.

Because the AABB of the skull mesh is in local space, we must transform the view frustum into the local space of each instance in order to perform the intersection test.

```
void InstancingAndCullingApp::UpdateInstanceData(const GameTimer& gt)
{
…..
// Perform the box/frustum intersection test in local space.
if((localSpaceFrustum.Contains(e->Bounds) != DirectX::DISJOINT) ||
(mFrustumCullingEnabled==false))
{
InstanceData data;
XMStoreFloat4x4(&data.World, XMMatrixTranspose(world));
XMStoreFloat4x4(&data.TexTransform, XMMatrixTranspose(texTransform));
data.MaterialIndex = instanceData[i].MaterialIndex;

// Write the instance data to structured buffer for the visible objects.
currInstanceBuffer->CopyData(visibleInstanceCount++, data);
}
}

e->InstanceCount = visibleInstanceCount;

std::wostringstream outs;
outs.precision(6);
outs << L"Instancing and Culling Demo" <<
L"    " << e->InstanceCount <<
L" objects visible out of " << e->Instances.size();
mMainWndCaption = outs.str();
}
```

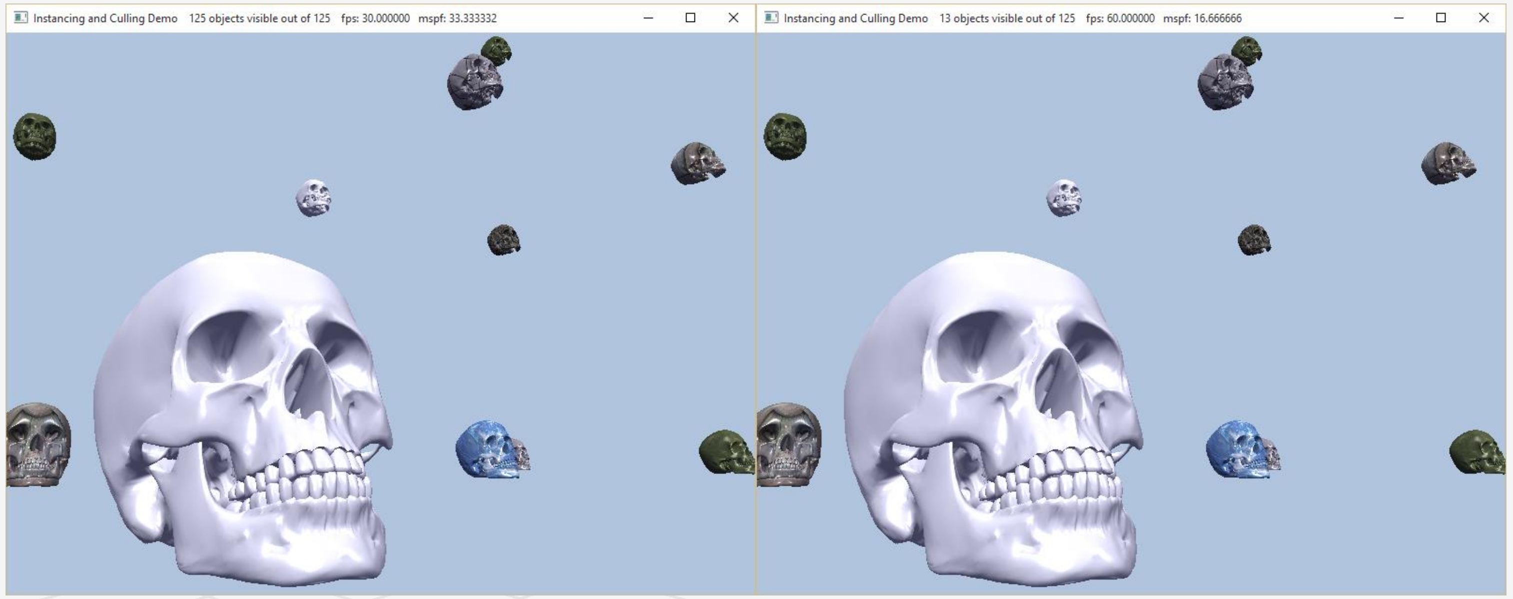
Instancing and Culling Demo

With frustum culling, we only submit eleven instances to the rendering pipeline for processing. Without frustum culling, we submit all 125 instances to the rendering pipeline for processing.

Each skull has about 60K triangles, so that is a lot of vertices to process and a lot of triangles to clip per skull.

(Left) Frustum culling is turned off, and we are rendering all 125 instances, and it takes about 33.33 ms to render a frame

(Right) Frustum culling is turned on, and we see that 13 out of 125 instances are visible, and our frame rate doubles.


Instancing and Culling Demo   125 objects visible out of 125   fps: 30.000000   mspf: 33.333332

Instancing and Culling Demo   13 objects visible out of 125   fps: 60.000000   mspf: 16.666666