Week 12

Picking

Hooman Salamat

Objectives

1. To learn how to implement the picking algorithm and to understand how it works. We break picking down into the following four steps:

1. Given the clicked screen point **s**, find its corresponding point on the projection window and call it **p**.

2. Compute the picking ray in view space. That is the ray originating at the origin, in view space, which shoots through **p**.

3. Transform the picking ray and the models to be tested with the ray into the same space.

4. Determine the object the picking ray intersects. The nearest (from the camera) intersected object corresponds to the picked screen object.

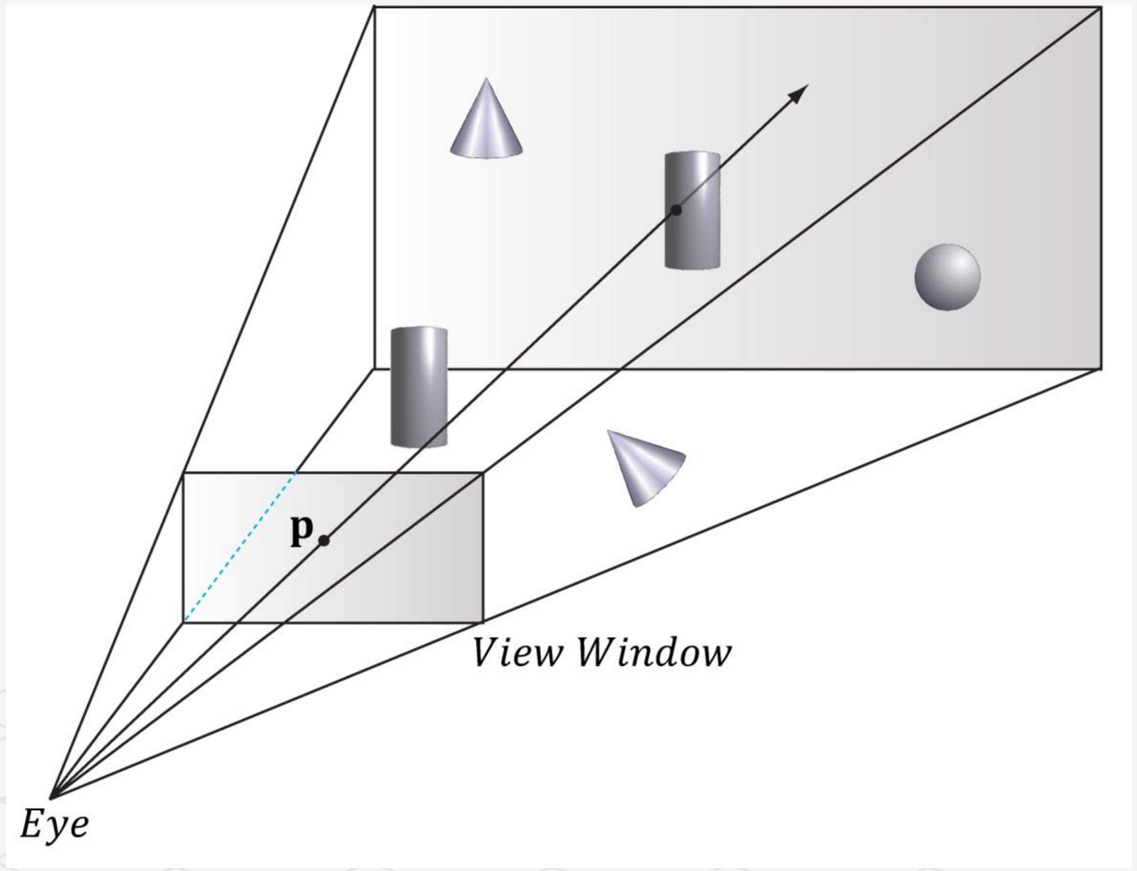
Picking by Clicking

1. Given the clicked screen point **s**, find its corresponding point on the projection window and call it **p**.

2. Compute the picking ray in view space. That is the ray originating at the origin, in view space, which shoots through **p**.

3. Transform the picking ray and the models to be tested with the ray into the same space.

4. Determine the object the picking ray intersects. The nearest (from the camera) intersected object corresponds to the picked screen object.



Viewport and Viewing Frustum

Conceptually, a viewport is a two-dimensional (2D) rectangle into which a 3D scene is projected.

In Direct3D, the rectangle exists as coordinates within a Direct3D surface that the system uses as a rendering target.

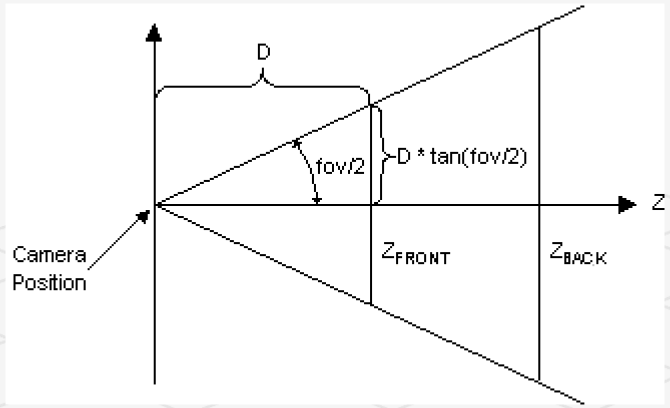
The projection transformation converts vertices into the coordinate system used for the viewport.

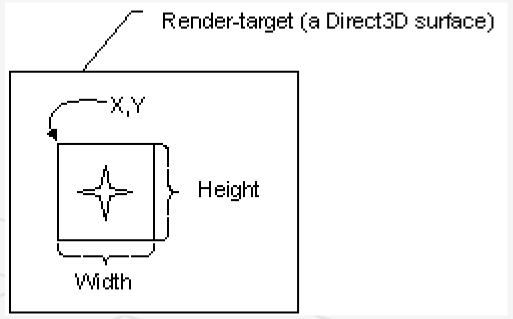
A viewport is also used to specify the range of depth values on a render-target surface into which a scene will be rendered (usually 0.0 to 1.0).

A viewing frustum is 3D volume in a scene positioned relative to the viewport's camera.

The shape of the volume affects how models are projected from camera space onto the screen.

Direct3D assumes that the viewport clipping volume ranges from -1.0 to 1.0 in X, and from -1.0 to 1.0 in Y.





Defining a frustum

We can define a frustum in view space, with center of projection at the origin and looking down the positive *z*-axis, by the following four quantities:

a near plane *n*, far plane *f*, vertical field of view angle α, and aspect ratio *r*.

Note that in view space, the near plane and far plane are parallel to the *xy*-plane; therefore we simply specify their distance from the origin along the *z*-axis.

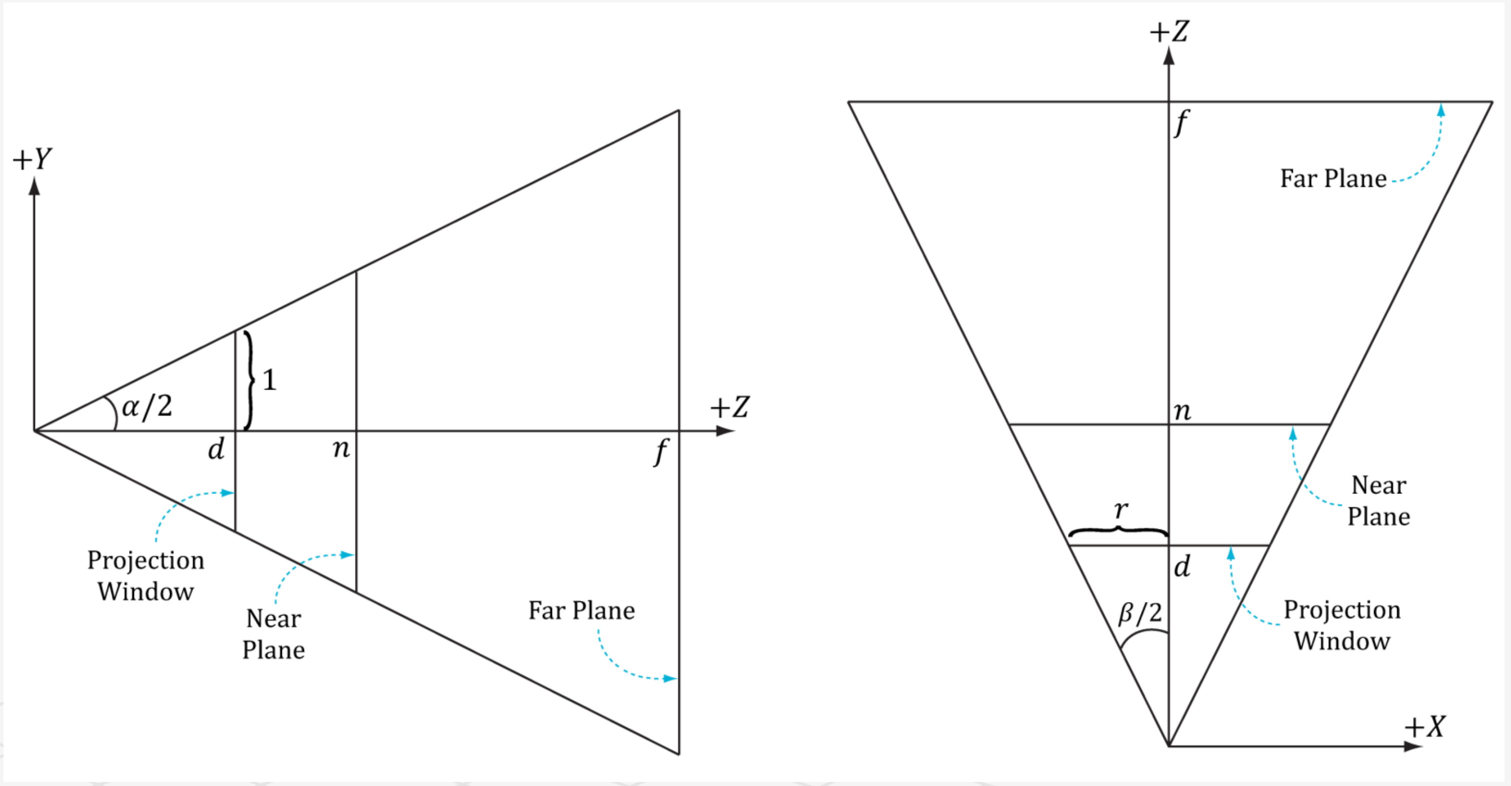
We label the horizontal field of view angle β, and it is determined by the vertical field of view angle α and aspect ratio *r*.

The aspect ratio is defined by *r* = *w*/*h* where *w* is the width of the projection window and *h* is the height of the projection window (units in view space).

The projection window lies at a distance

$$d = \cot\left(\frac{\alpha}{2}\right)$$

from the origin, where α is the vertical field of view angle.



D3D12_VIEWPORT

The first task is to transform the clicked screen point to normalized device coordinates (NDC)

The viewport matrix transforms vertices from normalized device coordinates to screen space.

The variables of the viewport matrix refer to those of the D3D12_VIEWPORT structure:

```
typedef struct D3D12_VIEWPORT
    {
    FLOAT TopLeftX;
    FLOAT TopLeftY;
    FLOAT Width;
    FLOAT Height;
    FLOAT MinDepth;
    FLOAT MaxDepth;
    } D3D12_VIEWPORT;

 D3D12_VIEWPORT mScreenViewport;
 mCommandList->RSSetViewports(1, &mScreenViewport);
```

SCREEN TO PROJECTION WINDOW TRANSFORM

Direct3D uses the viewport location and dimensions to scale the vertices to fit a rendered scene into the appropriate location on the target surface.

Internally, Direct3D inserts these values into the following matrix that is applied to each vertex.

In a game, the viewport is the entire backbuffer

```
// Update the viewport transform to cover the
client area.
mScreenViewport.TopLeftX = 0;
mScreenViewport.TopLeftY = 0;
mScreenViewport.Width    =
static_cast<float>(mClientWidth);
mScreenViewport.Height   =
static_cast<float>(mClientHeight);
mScreenViewport.MinDepth = 0.0f;
mScreenViewport.MaxDepth = 1.0f;
```

$$\mathbf{M} = \begin{bmatrix} \frac{Width}{2} & 0 & 0 & 0 \\ 0 & -\frac{Height}{2} & 0 & 0 \\ 0 & 0 & MaxDepth - MinDepth & 0 \\ TopLeftX + \frac{Width}{2} & TopLeftY + \frac{Height}{2} & MinDepth & 1 \end{bmatrix}$$

$$\mathbf{M} = \begin{bmatrix} w/2 & 0 & 0 & 0 \\ 0 & -h/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ w/2 & h/2 & 0 & 1 \end{bmatrix}$$

NDC to View Space

Assume we have a point **p** in normalized device coordinates (i.e., $-1 \le x_{ndc} \le 1$, $-1 \le y_{ndc} \le 1$, and $0 \le z_{ndc} \le 1$), and we want to transform it into screen space.

$\boldsymbol{p}_{ndc} = (x_{ndc}, y_{ndc}, z_{ndc}, 1)$

Transforming $\mathbf{p}_{ndc}$ to screen space yields:

$$[x_{ndc}, y_{ndc}, z_{ndc}, 1]\begin{bmatrix} w/2 & 0 & 0 & 0 \\ 0 & -h/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ w/2 & h/2 & 0 & 1 \end{bmatrix} = \left[\frac{x_{ndc}w + w}{2}, \frac{-y_{ndc}h + h}{2}, z_{ndc}, 1\right]$$

The coordinate z_{ndc} is just used by the depth buffer and we are not concerned with any depth coordinates for picking. The 2D screen point $\boldsymbol{p}_s = (x_s, y_s)$ corresponding to $\mathbf{p}_{ndc}$ is just the transformed *x*- and *y*-coordinates.

We now have the clicked point in NDC space. But to shoot the picking ray, we really want the screen point in view space. We mapped the projected point from view space to NDC space by dividing the x-coordinate by the aspect ratio r. To get back to view space, we just need to multiply the *x*-coordinate in NDC space by the aspect ratio.

$$x_s = \frac{x_{ndc}w + w}{2}$$

$$y_s = \frac{-y_{ndc}h + h}{2}$$

$$x_{ndc} = \frac{2x_s}{w} - 1$$

$$y_{ndc} = -\frac{2y_s}{h} + 1$$

$$x_v = r\left(\frac{2s_x}{w} - 1\right)$$

$$y_v = -\frac{2s_y}{h} + 1$$

Draw the Ray

The projection window is essentially the 2D image of the scene in view space.

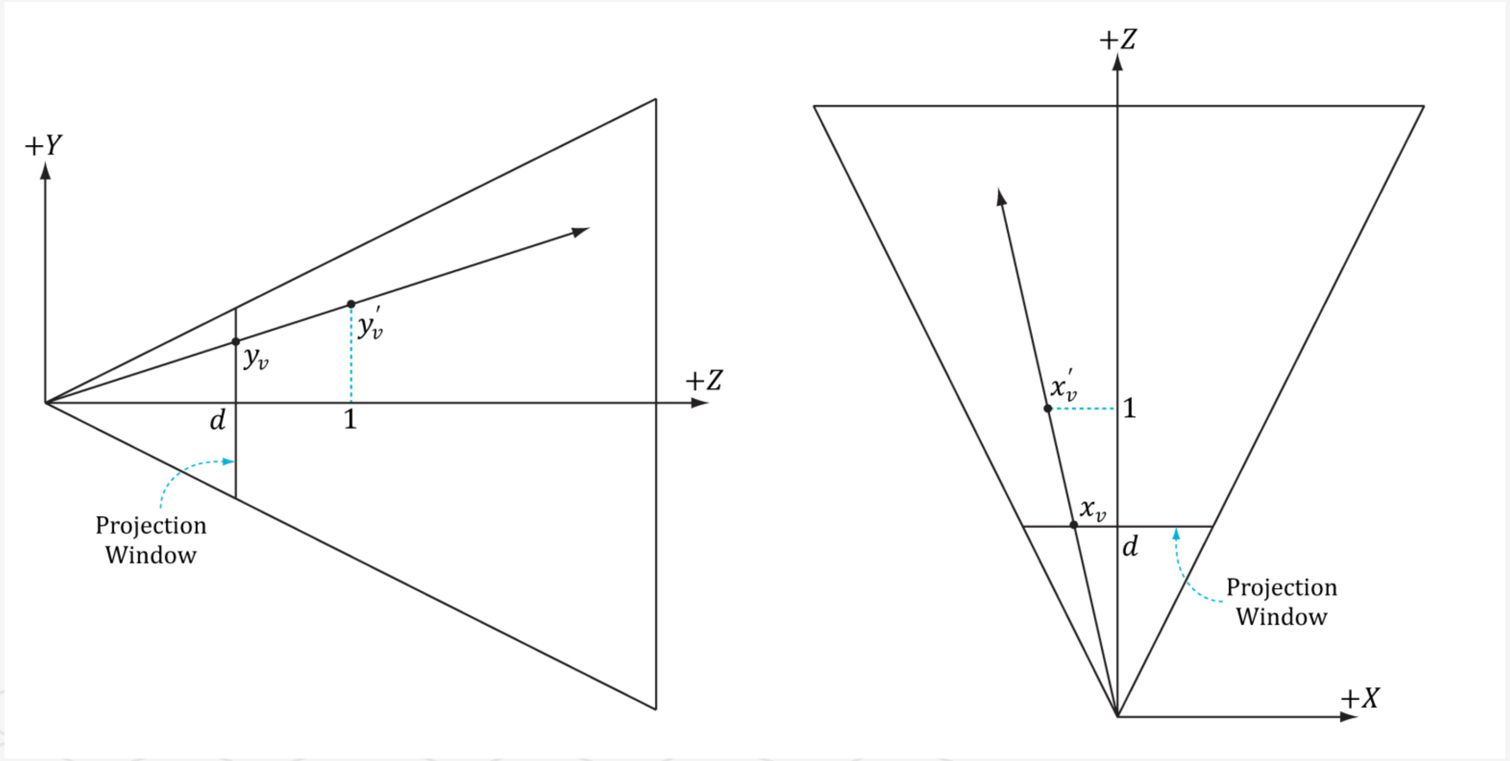
The image here will eventually be mapped to the back buffer; therefore, we like the ratio of the projection window dimensions to be the same as the ratio of the back buffer dimensions.

Note that the actual dimensions of the projection window are not important, just the aspect ratio needs to be maintained.

So we could shoot the picking ray through the point (x_v, y_v, d) on the projection window. However, this requires that we compute ***d***.

$$\frac{x_v}{d} = \frac{x'}{1}$$

$$\frac{y_v}{d} = \frac{y'}{1}$$



Projecting Vertices

Given a point (*x*, *y*, *z*), we wish to find its projection (*x*′, *y*′, *d*), on the projection plane *z* = *d*.

By considering the *x*- and *y*-coordinates separately and usingsimilar triangles, we find:

$$\frac{x'}{d}=\frac{x}{z}\Rightarrow x'=\frac{xd}{z}=\frac{x\cot(\alpha/2)}{z}=\frac{x}{z\tan(\alpha/2)}$$

and

$$\frac{y'}{d}=\frac{y}{z}\Rightarrow y'=\frac{yd}{z}=\frac{y\cot(\alpha/2)}{z}=\frac{y}{z\tan(\alpha/2)}$$

Observe that a point (*x*, *y*, *z*) is inside the frustum if and only if

$$\begin{aligned}-r&\le x'\le r\\ -1&\le y'\le 1\\ n&\le z\le f\end{aligned}$$

Draw the picking ray

A simpler way is to compute the picking ray by looking at the last figure. Recalling $\mathbf{P}_{00}$ and $\mathbf{P}_{11}$ in the projection matrix.

$$x'_v = \frac{x_v}{d} = \frac{x_v}{\cot\left(\frac{\alpha}{2}\right)} = x_v \cdot \tan\left(\frac{\alpha}{2}\right) = \left(\frac{2s_x}{w} - 1\right) r \tan\left(\frac{\alpha}{2}\right) \quad \mathbf{P}_{00} = \frac{1}{r\tan\left(\frac{\alpha}{2}\right)}$$

$$y'_v = \frac{y_v}{d} = \frac{y_v}{\cot\left(\frac{\alpha}{2}\right)} = y_v \cdot \tan\left(\frac{\alpha}{2}\right) = \left(-\frac{2s_y}{h} + 1\right) \tan\left(\frac{\alpha}{2}\right) \quad \mathbf{P}_{11} = \frac{1}{\tan\left(\frac{\alpha}{2}\right)}$$

$$\mathbf{P} = \begin{bmatrix} \frac{1}{r\tan(\alpha/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\alpha/2)} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix}$$

we can shoot our picking ray through the point $\boldsymbol{(x'_v, y'_v, 1)}$ instead. Note that this yields the same picking ray as the one shot through the point $\boldsymbol{(x_v, y_v, d)}$.

$$x'_v = \left(\frac{2s_x}{w} - 1\right) \Big/ \mathbf{P}_{00}$$

$$y'_v = \left(-\frac{2s_y}{h} + 1\right) \Big/ \mathbf{P}_{11}$$

PickingApp::Pick

The code that computes the picking ray in view space is given below. Note that the ray originates from the origin in view space since the eye sits at the origin in view space.

```
void PickingApp::Pick(int sx, int sy)

{

XMFLOAT4X4 P = mCamera.GetProj4x4f();

// Compute picking ray in view space.

float vx = (+2.0f*sx / mClientWidth - 1.0f) / P(0, 0);

float vy = (-2.0f*sy / mClientHeight + 1.0f) / P(1, 1);

// Ray definition in view space.

XMVECTOR rayOrigin = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);

XMVECTOR rayDir = XMVectorSet(vx, vy, 1.0f, 0.0f);
```

WORLD/LOCAL SPACE PICKING RAY

So far we have the picking ray in view space, but this is only useful if our objects are in view space as well.

Because the view matrix transforms geometry from world space to view space, the inverse of the view matrix transforms geometry from view space to world space.

If $\mathbf{r}_v(t) = \mathbf{q} + t\mathbf{u}$ is the view space picking ray (**q** is the ray origin and **u** is the ray direction**)** and **V** is the view matrix, then the world space picking ray is given by:

$$\mathbf{r}_w(t) = qV^{-1} + t\mathbf{u}V^{-1} = \mathbf{q}_w + t\mathbf{u}_w$$

Note that the ray origin **q** is transformed as a point (i.e., $q_w = 1$) and the ray direction **u** is transformed as a vector (i.e., $u_w = 0$).

A world space picking ray can be useful in some situations where you have some objects defined in world space.

However, most of the time, the geometry of an object is defined relative to the object’s own local space.

Therefore, to perform the ray/object intersection test, we must transform the ray into the local space of the object.

If **W** is the world matrix of an object, the matrix $\mathbf{W}^{-1}$ transforms geometry from world space to the local space of the object.

$$\mathbf{r}_\mathbf{L}(t) = q_w W^{-1} + t\mathbf{u}_w W^{-1}$$

Generally, each object in the scene has its own local space. Therefore, the ray must be transformed to the local space of each scene object to do the intersection test.

PickingApp::Pick

The following code shows how the picking ray is transformed from view space to the local space of an object:

```
// Assume nothing is picked to start, so the picked render-item is invisible.
mPickedRitem->Visible = false;

// Check if we picked an opaque render item.  A real app might keep a separate "picking list"
// of objects that can be selected.
for(auto ri : mRitemLayer[(int)RenderLayer::Opaque])
{
auto geo = ri->Geo;

// Skip invisible render-items.
if(ri->Visible == false)
continue;

XMMATRIX W = XMLoadFloat4x4(&ri->World);
XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(W), W);

// Tranform ray to vi space of Mesh.
XMMATRIX toLocal = XMMatrixMultiply(invView, invWorld);

rayOrigin = XMVector3TransformCoord(rayOrigin, toLocal);
rayDir = XMVector3TransformNormal(rayDir, toLocal);

// Make the ray direction unit length for the intersection tests.
rayDir = XMVector3Normalize(rayDir);
```

The XMVector3TransformCoord and XMVector3TransformNormal functions take 3D vectors as parameters, but note that with the XMVector3TransformCoord function there is an understood $w = 1$ for the fourth component. On the other hand, with the XMVector3TransformNormal function there is an understood $w = 0$ for the fourth component. Thus we can use XMVector3TransformCoord to transform points and we can use XMVector3TransformNormal to transform vectors.

RAY/MESH INTERSECTION

Once we have the picking ray and a mesh in the same space, we can perform the intersection test to see if the picking ray intersects the mesh.

The following code iterates through each triangle in the mesh and does a ray/triangle intersection test.

If the ray intersects one of the triangles, then it must have hit the mesh the triangle belongs to.

Otherwise, the ray misses the mesh.

Typically, we want the nearest triangle intersection, as it is possible for a ray to intersect several mesh triangles if the triangles overlap with respect to the ray.

For picking, we use the system memory copy of the mesh geometry stored in the MeshGeometry class.

This is because we cannot access a vertex/index buffer for reading that is going to be drawn by the GPU.

```
float tmin = 0.0f;
if(ri->Bounds.Intersects(rayOrigin, rayDir, tmin))
{

auto vertices = (Vertex*)geo->VertexBufferCPU->GetBufferPointer();
auto indices = (std::uint32_t*)geo->IndexBufferCPU->GetBufferPointer();
UINT triCount = ri->IndexCount / 3;

// Find the nearest ray/triangle intersection.
tmin = MathHelper::Infinity;
for(UINT i = 0; i < triCount; ++i)
{
// Indices for this triangle.
UINT i0 = indices[i * 3 + 0];
UINT i1 = indices[i * 3 + 1];
UINT i2 = indices[i * 3 + 2];

// Vertices for this triangle.
XMVECTOR v0 = XMLoadFloat3(&vertices[i0].Pos);
XMVECTOR v1 = XMLoadFloat3(&vertices[i1].Pos);
XMVECTOR v2 = XMLoadFloat3(&vertices[i2].Pos);

// We have to iterate over all the triangles in order to find the nearest intersection.
float t = 0.0f;
if(TriangleTests::Intersects(rayOrigin, rayDir, v0, v1, v2, t))
{
if(t < tmin)
{
// This is the new nearest picked triangle.
tmin = t;
UINT pickedTriangle = i;

mPickedRitem->Visible = true;
mPickedRitem->IndexCount = 3;
mPickedRitem->BaseVertexLocation = 0;

// Picked render item needs same world matrix as object picked.
mPickedRitem->World = ri->World;
mPickedRitem->NumFramesDirty = gNumFrameResources;

// Offset to the picked triangle in the mesh index buffer.
mPickedRitem->StartIndexLocation = 3 * pickedTriangle;}}}}}
```

Ray/AABB Intersection

We use the DirectX collision library function BoundingBox::Intersects to see if the ray intersects the bounding box of the mesh.

A popular strategy is to approximate the mesh with a simple bounding volume, like a sphere or box. Then, instead of intersecting the ray with the mesh, we first intersect the ray with the bounding volume.

If the ray misses the bounding volume, then the ray necessarily misses the triangle mesh and so there is no need to do further calculations. If the ray intersects the bounding volume, then we do the more precise ray/mesh test.

The BoundingBox::Intersects function returns true if the ray intersects the box and false otherwise; it is prototyped as follows:

```
//------------------------DirectXCollision.h-------------------------------------------------

//-- C:\Program Files (x86)\Windows Kits\10\Include\10.0.18362.0\um---

// Compute the intersection of a ray (Origin, Direction) with an axis aligned

// box using the slabs method.

bool XM_CALLCONV BoundingBox::Intersects(

FXMVECTOR Origin, // ray origin

FXMVECTOR Direction, // ray direction (must be unit length)

float& Dist   // ray intersection parameter

) const
```

Given the ray $\mathbf{r}(t) = \mathbf{q} + t\mathbf{u}$, the last parameter outputs the ray parameter t_0 that yields the actual intersection point $\mathbf{p}$:

$\mathbf{p} = \mathbf{r}(t_0) = \mathbf{q} + t_0\mathbf{u}$

DirectXMath Triangle Test Functions

Lists the triangle test functions provided by the DirectXMath **TriangleTests** namespace.

```
bool Intersects(

 XMVECTOR Origin,    // The origin of the ray.

XMVECTOR Direction,    // The direction of the ray.

XMVECTOR V0,   // A vector defining the triangle.

XMVECTOR V1,     // A vector defining the triangle.

XMVECTOR V2, // A vector defining the triangle.

[ref] float    &Dist ); // The distance along the ray where the intersection occurs.
```

Return value: A boolean value indicating whether the triangle intersects with the ray.

Term	Description
ContainedBy	Tests whether a triangle is contained within six planes (typically a frustum).
Intersects (triangle-plane)	Tests whether a triangle and a plane intersect.
Intersects (triangle-ray)	Test whether a triangle intersects with a ray.
Intersects (triangle-triangle)	Test whether two triangles intersect.

Ray/Triangle Intersection

For performing a ray/triangle intersection test, we use the DirectX collision library (DirectXCollision.inl) function TriangleTests::Intersects:

Let $\mathbf{r}(t) = \mathbf{q} + t\mathbf{u}$ be a ray and $\mathbf{T}(u, v) = \mathbf{v}0 + u(\mathbf{v}1 - \mathbf{v}0) + v(\mathbf{v}2 - \mathbf{v}0)$ for $u \geq 0$, $v \geq 0$, $u + v \leq 1$ be a triangle. We wish to simultaneously solve for t, u, v such that $\mathbf{r}(t) = \mathbf{T}(u, v)$ (i.e., the point the ray and triangle intersect).

The point $\mathbf{p}$ in the plane of the triangle has coordinates (u, v) relative to the skewed coordinate system with origin $\mathbf{v}0$ and axes $\mathbf{v}1 - \mathbf{v}0$ and $\mathbf{v}2 - \mathbf{v}0$.

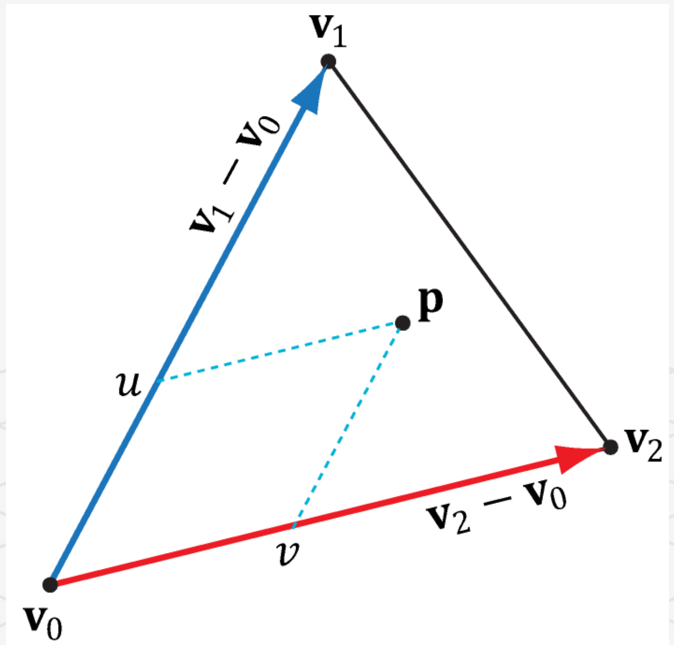
```
namespace TriangleTests {

// Compute the intersection of a ray (Origin, Direction) with a triangle

// (V0, V1, V2).  Return true if there is an intersection and also set *pDist

// to the distance along the ray to the intersection.

bool XM_CALLCONV Intersects( FXMVECTOR Origin, FXMVECTOR Direction, FXMVECTOR V0,
GXMVECTOR V1, HXMVECTOR V2, float& Dist )
```



DEMO APPLICATION

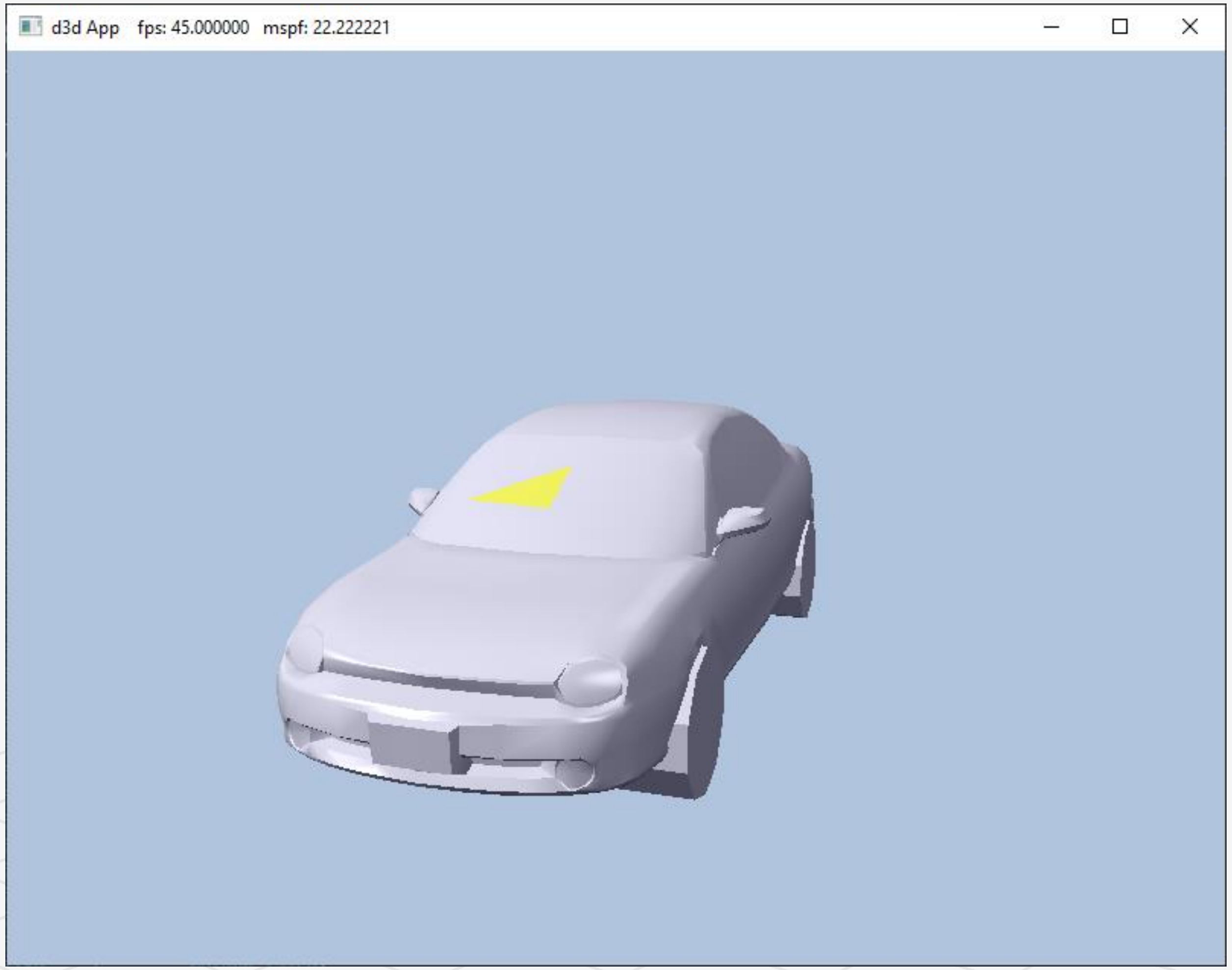
The demo renders a car mesh and allows the user to pick a triangle by pressing the right mouse button, and the selected triangle is rendered using a "highlight" material.

To render the triangle with a highlight, we need a render-item for it.

Because we do not yet know which triangle will be picked, and so we do not know the starting index location and world matrix.

Therefore, we have added a Visible property to the render-item structure. An invisible render-item will not be drawn.

```
struct RenderItem
{
RenderItem() = default;
RenderItem(const RenderItem& rhs) = delete;
bool Visible = true;
BoundingBox Bounds;
```



Picking Demo

```
RenderItem* mPickedRitem = nullptr;
```

The code snippet (PickingApp::Pick method) shows how we fill out the remaining render-item properties based on the selected triangle:

```
// Assume nothing is picked to
start, so the picked render-item is
invisible.

mPickedRitem->Visible = false;
```

```
// We have to iterate over all the triangles in order to find the nearest intersection.
float t = 0.0f;
if(TriangleTests::Intersects(rayOrigin, rayDir, v0, v1, v2, t))
{
if(t < tmin)
{
// This is the new nearest picked triangle.
tmin = t;
UINT pickedTriangle = i;

mPickedRitem->Visible = true;
mPickedRitem->IndexCount = 3;
mPickedRitem->BaseVertexLocation = 0;

// Picked render item needs same world matrix as object picked.
mPickedRitem->World = ri->World;
mPickedRitem->NumFramesDirty = gNumFrameResources;

// Offset to the picked triangle in the mesh index buffer.
mPickedRitem->StartIndexLocation = 3 * pickedTriangle;
}
}
```

PickingApp::Draw

This render-item is drawn after we draw our opaque render-items. It uses a special highlight PSO, which uses transparency blending and sets the depth test comparison function to D3D12_COMPARISON_FUNC_LESS_EQUAL.

This is needed because the picked triangle will be drawn twice, the second time with the highlighted material.

The second time the triangle is drawn, the depth test would fail if the comparison function was just D3D12_COMPARISON_FUNC_LESS.

```
void PickingApp::Draw(const GameTimer& gt)
{
    auto cmdListAlloc = mCurrFrameResource->CmdListAlloc;

ThrowIfFailed(cmdListAlloc->Reset());

ThrowIfFailed(mCommandList->Reset(cmdListAlloc.Get(), mPSOs["opaque"].Get()));

mCommandList->RSSetViewports(1, &mScreenViewport);
mCommandList->RSSetScissorRects(1, &mScissorRect);

mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET));

mCommandList->ClearRenderTargetView(CurrentBackBufferView(), Colors::LightSteelBlue, 0,
nullptr);
mCommandList->ClearDepthStencilView(DepthStencilView(), D3D12_CLEAR_FLAG_DEPTH |
D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr);

mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(), true, &DepthStencilView());

ID3D12DescriptorHeap* descriptorHeaps[] = { mSrvDescriptorHeap.Get() };
mCommandList->SetDescriptorHeaps(_countof(descriptorHeaps), descriptorHeaps);

mCommandList->SetGraphicsRootSignature(mRootSignature.Get());

auto passCB = mCurrFrameResource->PassCB->Resource();
mCommandList->SetGraphicsRootConstantBufferView(1, passCB->GetGPUVirtualAddress());

auto matBuffer = mCurrFrameResource->MaterialBuffer->Resource();
mCommandList->SetGraphicsRootShaderResourceView(2, matBuffer->GetGPUVirtualAddress());

mCommandList->SetGraphicsRootDescriptorTable(3, mSrvDescriptorHeap-
>GetGPUDescriptorHandleForHeapStart());

DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Opaque]);

mCommandList->SetPipelineState(mPSOs["highlight"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Highlight]);
```

PSOs

```
// PSO for opaque objects.

    ZeroMemory(&opaquePsoDesc, sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));

opaquePsoDesc.InputLayout = { mInputLayout.data(), (UINT)mInputLayout.size() };

opaquePsoDesc.pRootSignature = mRootSignature.Get();

opaquePsoDesc.VS =

{

reinterpret_cast<BYTE*>(mShaders["standardVS"]->GetBufferPointer()),

mShaders["standardVS"]->GetBufferSize()

};

opaquePsoDesc.PS =

{

reinterpret_cast<BYTE*>(mShaders["opaquePS"]->GetBufferPointer()),

mShaders["opaquePS"]->GetBufferSize()

};

opaquePsoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);

opaquePsoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);

opaquePsoDesc.DepthStencilState = CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);

opaquePsoDesc.SampleMask = UINT_MAX;

opaquePsoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;

opaquePsoDesc.NumRenderTargets = 1;

opaquePsoDesc.RTVFormats[0] = mBackBufferFormat;

opaquePsoDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;

opaquePsoDesc.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;

opaquePsoDesc.DSVFormat = mDepthStencilFormat;

    ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&opaquePsoDesc,
IID_PPV_ARGS(&mPSOs["opaque"])));
```

```
//
// PSO for highlight objects
//

D3D12_GRAPHICS_PIPELINE_STATE_DESC highlightPsoDesc = opaquePsoDesc;

// Change the depth test from < to <= so that if we draw the same triangle twice, it will
// still pass the depth test.  This is needed because we redraw the picked triangle with a
// different material to highlight it.  If we do not use <=, the triangle will fail the
// depth test the 2nd time we try and draw it.
highlightPsoDesc.DepthStencilState.DepthFunc = D3D12_COMPARISON_FUNC_LESS_EQUAL;

// Standard transparency blending.
D3D12_RENDER_TARGET_BLEND_DESC transparencyBlendDesc;
transparencyBlendDesc.BlendEnable = true;
transparencyBlendDesc.LogicOpEnable = false;
transparencyBlendDesc.SrcBlend = D3D12_BLEND_SRC_ALPHA;
transparencyBlendDesc.DestBlend = D3D12_BLEND_INV_SRC_ALPHA;
transparencyBlendDesc.BlendOp = D3D12_BLEND_OP_ADD;
transparencyBlendDesc.SrcBlendAlpha = D3D12_BLEND_ONE;
transparencyBlendDesc.DestBlendAlpha = D3D12_BLEND_ZERO;
transparencyBlendDesc.BlendOpAlpha = D3D12_BLEND_OP_ADD;
transparencyBlendDesc.LogicOp = D3D12_LOGIC_OP_NOOP;
transparencyBlendDesc.RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL;

highlightPsoDesc.BlendState.RenderTarget[0] = transparencyBlendDesc;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&highlightPsoDesc,
IID_PPV_ARGS(&mPSOs["highlight"])));
```