24. Picking

This lesson builds directly off the last lesson, normal mapping.

Here we will learn how to turn the 2d screen position (in pixels) that the mouse cursor is at, into a 3d ray in world space. We can then check to see if that ray intersects with any of the objects on the screen. If the ray intersects with any of the bottles in this lesson, we will display the distance from the camera to that bottle, increase the score, and remove that bottle from being displayed any longer and from being checked if the ray picks it again.

Introduction

This lesson build directly off the last lesson, Normal Mapping. We will learn how to pick objects in Direct3D 11.

Picking in D3D11 takes an extra step than picking in D3D10. This is because the mesh interface is not available in D3D11 anymore, which before had a method to test if a ray intersected with an object. We will now do this ourselves.
Picking a 3D Object

To pick a 3D object, we will need to follow a series of steps:

1. Get a 2D vector (x, y) of the Position of the mouse cursor in screen space (the client window) (in pixels).
2. Transform the 2D vector representing the mouses position in screen space to a 3D vector (x, y, z) representing a ray in view space.
3. Transform the view space ray into a world space ray.
4. Transform the model, or objects vertex positions from model (or local) space to world space.
5. Loop through each of the triangles in the model to find out which (if any) triangle the ray intersects with. (This is the step we were able to skip in D3D10 by using a method provided by the mesh interface.)

1. Get the Cursor Position in Screen Space

This is easy. To get the cursor position in screen space (the client window), we call two functions, **GetCursorPos()**, and **ScreenToClient()**. These will store the x and y positions of the cursor in a variable of type POINT. This variable will be the position of our cursor measured in units of pixels, where the top left of the client window is (0, 0), and the bottom right is the (clientWidth, clientHeight). We will then pass the x and y coordinates of the cursor to a function which will transform it into a 3D vector representing a ray in view space, then transform that 3D ray in view space to a 3D ray in world space.

2. Transform 2D Screen Space Ray to 3D View Space Ray

This is where we start running into a little math. First, we Transform the Picked point's x- and y- axis (in screen space) to View space coordinates. View space ranges from -1 to 1 on the x and y axis, and 0 to 1 on the z axis. To do this is not very difficult, we multiply the picked x and y positions by two, divide that by the width and height of the client window, then subtract 1. This is probably more clear by example. Say our window is 800 by 600. we click at the point (100, 100). We will start by finding the x axis in view space. First, we multiply 100 by 2. This gives us 200. We then divide 200 by the width of the window (800), giving us 0.25. We can then subtract 1 from this answer, which gives us -0.75. Now this point's x axis has been converted to view space, where the value is between -1 and 1. You can imagine the same for the Y axis. We now know the equation to get the x and y axis in screen space converted to view space:

```
ViewSpaceX = (2 * PickedPointX) / ClientWidth
ViewSpaceY = (2 * PickedPointY) / ClientHeight
```

That was easy right? Well now for the more difficult math problem to kick in. We need to solve for ViewSpaceZ. We can do this in one of two ways. Either set Z to equal 1, and modify the ViewSpaceX and ViewSpaceY equations, or actually solve for Z. Z is actually the distance from the cameras origin, to the projection window. We can find Z, or the distance to the projection window with the equation Z = cotangent(angle/2). Angle is the vertical field of view in radians, which we set before in our code when we set camProjection using the function **XMMatrixPerspectiveFovLH()**. If you look for this line in the lessons code, you can see that we set the vertical field of view angle to 0.4f * 3.14. To find the cotangent we can use 1/tangent, so to find Z:

```
ViewSpaceZ = 1 / tan((0.4 * 3.14) / 2)
```

Now instead of actually solving for Z, we can easily modify our two equations to find view spaces x and y values by using the projection matrix represented by camProjection in this lessons code. For ViewSpaceX, all we need to do is divide by the variable stored in camProjection(0, 0). This variable is "1 / (ratio * tangent(angle / 2))". For ViewSpaceY, we divide by camProjection(1,1), which is "1 / tangent(angle / 2)". So Our final code will look like this:

```
ViewSpaceX = ((2 * PickedPointX) / ClientWidth)  / camProjection(0,0)
ViewSpaceY = ((2 * PickedPointY) / ClientHeight) / camProjection(1,1)
ViewSpaceZ = 1
```

First Person Shooter Picking

If we want to use the center of the screen (like in a first person shooter) to be the point we are picking with, that is very easy. Since View Space ranges from -1 to 1 on the x and y axis, and 0 to 1 on the z axis, we can use (0, 0, 1) as our View Space point. The reason this works is because we are

going straight into view space from the origin of the camera, which is at point (0, 0, 0) in view space. In fact, we could actually set the z value to anything greater than 0 for this to work, since we essencially creating a ray using the two points (origin and direction). So our code for a first person shooter would look like this:

```
ViewSpaceX = 0
ViewSpaceY = 0
ViewSpaceZ = 1
```

3. View Space to World Space

Remember to go from local to world to view to screen space, we just multiply them in order? Well now we need to go backwards from View space to World space. How do we do this? easy, we find the **inverse of our view space matrix**, which is actually our world space matrix! We can find the inverse of the matrix by using the function **XMMatrixInverse()**, where the first parameter we will not use, and the second parameter is our cameras view space matrix. The funny thing about this function, is we cannot set the first parameter to null, even though we don't need it. We need to supply it a vector, which it will then send the "inverse determinant" vector to whatever we supply for the first parameter. Anyway, after we find our world matrix by inversing our view matrix, we transform our rays position and direction in view space using the world space matrix, which results in our world space ray!

4. Transform Vertices to World Space

We're not done transforming things yet! But this is our last transform. We need to take the vertices positions (in the models space) that make up the model, and transform them to world space, using the models world space matrix. The problem with this step that you might find, is that we need to actually get our vertex positions from the model. In D3D10, this was not a problem, all we had to do was use the models world space matrix and a function provided by the mesh interface which would detect if a ray intersected with the model. However, in D3D11, the mesh interface is no long an option. Because of this, we need to get the vertex positions ourselves, and check for intersection ourselves. Getting the Vertex positions is not actually difficult, but we will not be taking them from our vertex buffer, since the CPU can only read from a staging buffer, and a buffer that the CPU can read from cannot be directly binded to the pipeline for rendering. So what we need to do now, is when we create our vertices and indices to put into our vertex and index buffer for the model, we ALSO need to create two arrays or vectors to hold the indices and just the positions of our vertices. We will then use these positions in the vertex array to get the positions of the triangles in our model, and the indices to find which vertices make up each triangle. After we do all that, we can then transform the vertices into world space using the models world space matrix.

One thing I should also mention, is up until this point, everything we have been doing, only needs to be done once per click, because all objects we check if the pick ray intersects with will use that same picking ray. But from this part on (step 4), we need to start doing these steps for every model or model's subsets we want to check.

5. Check for an Intersection Between the Pick Ray and Object (Model)

Here's where the real math starts. This step can be broken down into smaller steps:

First, fill in the plane equation the triangle lies on ("Ax + By + Cz + D = 0").
Find where (on the ray) the ray intersects with the triangles plane.
Find the point on the triangles plane in which the ray intersects.
Find if the point the ray intersects with the triangles plane is inside the triangle.

1. Fill in the plane equation the triangle lies on ("Ax + By + Cz + D = 0")

To find the plane the triangle lies on (which is described by the triangles normal), we first get two

edges of the triangle, U and V. We will then use these two edges to get the normal of the triangle, which describes the plane, and the A, B, and C components of our equation. We can use the following equation to find the face normal:

```
faceNormal = (V2 - V1) x (V3 - V1)
```

Where "x" is a cross multiplication of vectors. We can do this in code like this:

```
//Gets the two edges
U = TriangleVertex2 - TriangleVertex1;
V = TriangleVertex3 - TriangleVertex1;

//Gets the triangles normal
faceNormal = XMVector3Cross(U, V);

//Normalize the normal
faceNormal = XMVector3Normalize(faceNormal);
```

Now we have the A, B, and C components of our plane equation, which is just the x, y, and z parts of the planes normal. Now we need to find the x, y, and z components of our equation. These are just another point on the plane, and since all three vertices make up the plane, any of them will work, so in our code, we will just use the first vertex that makes up the triangle. Now our equations looks like this:

```
(faceNormal.x * TriangleVertex1.x) + (faceNormal.y * TriangleVertex1.y) + (faceNormal.z *
```

The last thing we need to do in this step is to find "D". This only takes a little algebra, so it isn't very difficult. Remember learning in algebra whatever you do to one side of the equals sign, you have to do to the other? Well taking that into account, we will first subtract D from both sides, making the equation look like this:

```
Ax + By + Cz = -D
```

Now we have found negative D (-D), but we need just plain "D". How do we do this? Easy, just multiply both sides by -1!

```
-1 * (Ax + By + Cz) = D

//Turns into:

-Ax - By - Cz = D

//In Code:

D = -(faceNormal.x * TriangleVertex1.x) - (faceNormal.y * TriangleVertex1.y) - (faceNormal
```

We have now found our plane equation and can move on to find where on the ray intersects with the plane.

2. Find where (on the ray) the ray intersects with the triangles plane.

Now we find a value we will name "t". "t" is the distance from the rays origin (position of the camera) that the ray intersects with the triangles plane. If "t" is positive, we know the intersection happened in front of the camera, and if "t" is negative, the intersection happened behind the camera, which we will then skip the rest (unless you want to pick objects behind the camera). We have to watch out for a divide by zero here, in the off chance that the picked ray runs perfectly parallel to the plane. To find "t" we can use this parametric equation:

```
t = -(A*x2 + B*y2 + C*z2 + D) / (A*(x1-x2) + B*(y1-y2) + C*(z1-z2))
```

Where x1, y1, and z1 are the origin of the ray, and x2, y2, and z2 are the direction of the ray. To make this easier to read in code, we have first created two parts to our equation, "ep1" and "ep2". These parts are found by plugging the position and orgin of our ray into the plane equation, like this:

```
ep1 = (pickRayInWorldSpacePos.x * (Ax)) + (pickRayInWorldSpacePos.y * (By)) + (pickRayInWorldS
ep2 = (pickRayInWorldSpaceDir.x * (Ax)) + (pickRayInWorldSpaceDir.y * (By)) + (pickRayInWorldS
```

After we have these two parts, we can solve for "t" much easier:

```
t = -(ep1 + D)/(ep2)
```

Remember the divide by zero I mentioned? this is where it could happen. So to prevent a vortex which swallows up the entire world (a divide by zero), we check to make sure "ep2" does not equal zero before we run this equation.

3. Find the Point on the Triangles Plane in Which the Picked Ray Intersects.

This is pretty easy actually. We can find the x, y, and z values of this point with the following equations:

```
planeIntersectX = pickRayInWorldSpacePos.x + pickRayInWorldSpaceDir.x * t;
planeIntersectY = pickRayInWorldSpacePos.y + pickRayInWorldSpaceDir.y * t;
planeIntersectZ = pickRayInWorldSpacePos.z + pickRayInWorldSpaceDir.z * t;
```

4. Find if the point the ray intersects with the triangles plane is inside the triangle.

This is also not terribly difficult, and there are many ways to do this. One possibility is to find the area of the triangle, create three smaller triangles using the intersection point and the three vertices that make up the first triangle (the triangle you are checking for an intersection with). Add up the areas of the three smaller triangles, and if they equal the area of the first triangle, the point is inside.

Another way to do this is use barycentric coordinates, which is a tad bit more confusing, but the most efficient way of doing this.

The way we will do it, is check which side of each of the three edges that make up the triangle that the intersection point is located. If the intersection point is on the "correct" side of each of the edges, then it is inside the triangle, otherwise if it is on the wrong side of any of the triangles edges, it is not inside the triangle and we can exit early.

I want to direct you to here if you are interested in learning how to use barycentric coordinates to do this job, or for a more complete explanation on the way I will show you how to do it.

To test a side of the triangle to find if the point is on the "correct" side, we will need another point that IS on the correct side of the line. We will use the third, or unused corner of the triangle (i say unused, but i mean unused in the edge we are checking against). We create two cross products, the first using the point and the edges two vertices, and the second using the third vertex, and the edges two vertices. We will get the dot product of these two cross products, and if the answer is greater than or equal to 0, the point is on the "correct" side. Otherwise, the point is outside the triangle and we can exit. Here is the code:

```
XMVECTOR cp1 = XMVector3Cross((triV3 - triV2), (point - triV2));
XMVECTOR cp2 = XMVector3Cross((triV3 - triV2), (triV1 - triV2));
if(XMVectorGetX(XMVector3Dot(cp1, cp2)) >= 0)
{
    cp1 = XMVector3Cross((triV3 - triV1), (point - triV1));
    cp2 = XMVector3Cross((triV3 - triV1), (triV2 - triV1));
    if(XMVectorGetX(XMVector3Dot(cp1, cp2)) >= 0)
    {
        cp1 = XMVector3Cross((triV2 - triV1), (point - triV1));
        cp2 = XMVector3Cross((triV2 - triV1), (triV3 - triV1));
        if(XMVectorGetX(XMVector3Dot(cp1, cp2)) >= 0)
        {
            return true;
        }
        else
            return false;
    }
    else
        return false;
}
return false;
```

I want to mention one last thing. Don't make fun of my bottle

Model's Vertex Position and Index Arrays

Now that our program needs to read the models vertex positions and vertex indices, we need to create a vector to hold them, since we are not able to read the D3D buffers they are held in directly with the CPU. So here is the two new ones for our ground model.

```
std::vector<XMFLOAT3> groundVertPosArray;
std::vector<DWORD> groundVertIndexArray;
```

Bottle Model Members

These are all the variables and stuff that our bottle model's need. Notice how we have a an array of the bottles world matrix and an array of integers named "bottleHit", also the "numBottles" variable. These are because we will be displaying more than a single bottle in our screen, specifically here, 20.

```
ID3D11Buffer* bottleVertBuff;
ID3D11Buffer* bottleIndexBuff;
std::vector<XMFLOAT3> bottleVertPosArray;
std::vector<DWORD> bottleVertIndexArray;
int bottleSubsets = 0;
std::vector<int> bottleSubsetIndexStart;
std::vector<int> bottleSubsetTexture;
XMMATRIX bottleWorld[20];
int* bottleHit = new int[20];
int numBottles = 20;
```

New Global Variables

We have a couple new global variables. The first being "isShoot". This variable will be true while the first mouse button is being pressed, and false while it is not. We do this so we can't just hold the mouse button down and glide the cursor over the objects to be picked. We only want to be able to pick one time per click. The next two are integers which will hold the width and height of our client window in pixels. As you will see later, these are updated as the screen is resized. Luckily, the "WM_SIZE" windows message is automatically called when we start our program, so these variables will be filled as our program starts. Otherwise we would have to fill them manually at first, which might not be 100% accurate if we are in windowed mode (since the border and the top of the applications window take up a little space). Next is a global variable that will keep track of our score (the number of bottles picked), and the one after that is the distance from the camera to the last picked object (only updated when the object was picked).

```
bool isShoot = false;

int ClientWidth = 0;
int ClientHeight = 0;

int score = 0;
float pickedDist = 0.0f;
```

New Function Prototypes

We have three new functions. The first will calculate the world space pick ray from the 2d coordinates of our mouse cursor. The next calculates whether the object was picked or not, calling the last function. The last function is called from the pick function, and it says if a point is inside a triangle (the point has to be on the triangles plane to know for sure).

```
void pickRayVector(float mouseX, float mouseY, XMVECTOR& pickRayInWorldSpacePos, XMVECTOR& pi
float pick(XMVECTOR pickRayInWorldSpacePos,
    XMVECTOR pickRayInWorldSpaceDir,
    std::vector<XMFLOAT3>& vertPosArray,
    std::vector<DWORD>& indexPosArray,
    XMMATRIX& worldSpace);
bool PointInTriangle(XMVECTOR& triV1, XMVECTOR& triV2, XMVECTOR& triV3, XMVECTOR& point );
```

New LoadObjModel() Function Prototype

We have added two new arguments to this function. They are so we can get the models vertex positions and indices and do operations like check for a ray intersection with the model. We need to do this because we cannot read from the D3D buffers that hold the vertex and index information we bind to the pipeline.

```
bool LoadObjModel(std::wstring filename,            //.obj filename
    ID3D11Buffer** vertBuff,                    //mesh vertex buffer
    ID3D11Buffer** indexBuff,                    //mesh index buffer
    std::vector<int>& subsetIndexStart,            //start index of each subset
    std::vector<int>& subsetMaterialArray,        //index value of material for each subset
    std::vector<SurfaceMaterial>& material,        //vector of material structures
    int& subsetCount,                            //Number of subsets in mesh
    bool isRHCoordSys,                            //true if model was created in right hand co
    bool computeNormals,                        //true to compute the normals, false to use th
    ///////////////**************new**************////////////////////
    std::vector<XMFLOAT3>& vertPosArray,        //Used for CPU to do calculations on the Geome
    std::vector<DWORD>& vertIndexArray);        //Also used for CPU caculations on geometry
    ///////////////**************new**************////////////////////
```

Initializing Direct Input

We have to modify the function that initializes direct input so we can see the mouse when running our application, because until now (and since we implemented direct input), we have had the cooperation level for the mouse set to exclusive, and we could not see the cursor, but now we will set it to non-exclusive, so we can see the cursor.

```
bool InitDirectInput(HINSTANCE hInstance)
{
    hr = DirectInput8Create(hInstance,
        DIRECTINPUT_VERSION,
        IID_IDirectInput8,
        (void**)&DirectInput,
        NULL);

    hr = DirectInput->CreateDevice(GUID_SysKeyboard,
        &DIKeyboard,
        NULL);

    hr = DirectInput->CreateDevice(GUID_SysMouse,
        &DIMouse,
        NULL);

    hr = DIKeyboard->SetDataFormat(&c_dfDIKeyboard);
    hr = DIKeyboard->SetCooperativeLevel(hwnd, DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);

    hr = DIMouse->SetDataFormat(&c_dfDIMouse);
    ///////////////**************new**************////////////////////
    hr = DIMouse->SetCooperativeLevel(hwnd, DISCL_NONEXCLUSIVE | DISCL_NOWINKEY | DISCL_FOR
    ///////////////**************new**************////////////////////

    return true;
}
```

Checking for Input

Now we will check for the left mouse button to be pressed. The left mouse button is represented by mouseCurrState.rgbButtons[0]. Remember that global variable, isShoot? Well now we get to use it. Remember when checking for input, if the button is down, the code will keep running, same with if its up. That means, the code to run when a button is pressed will not be only ran once per click, but it will be ran as many times possible while the button is currently down. We need to check to make sure the code has not already been ran since the button was last up. If you look, you can see, when the mouse button is pressed, it checks isShoot, if its false, it runs the code then sets isShoot to true. That way, the next time around, if the mouse button is still being held down, we don't check for picking again, since we only want to check for picking once per click. When the button is released, isShoot is set to false again.

We create a variable of type POINT which will store our mouses position. We then get our mouses position, and translate that position to the client windows position (in case we are in windowed mode, the client window might not align exactly with the monitors screen).

We create a couple variables. The first, tempDist will store the picked distance to the current object being checked. If the current object being check for picking is not actually picked, this value will be FLT_MAX, which is the largest float value possible. The next variable is the distance to the closest object that was picked. Next we have hitIndex, which will store the index value of the closest picked bottle. We need these last two variables so we only pick the closest bottle, and not bottles behind.

We then create two vectors to hold the world space picking ray position and direction, and call our function which will transform our mouse cursor's position on our monitor to two 3d vectors describing the world space picking rays position and direction.

The next part, the loop, is where we will check every pickable object for an intersection with our picking ray. The first thing we do in this loop is call the function to check if the current object is picked or not, and store its picked distance in tempDist. Next, if tempDist is smaller than closestDist, we know that the current picked object is closer to the camera than the last one (if there even was a last one),

and set the new closestDist and hitIndex variables.

After we check each model, we make sure that an object was even picked by making sure closestDist is smaller than FLT_MAX. If an object was picked, this is where we would do whatever it is we want to do when an object is picked, and in the case of this lesson, we will make sure the picked bottle is no longer displayed or checked for picking by setting its bottleHit value to 1, then increase our score, and set pickedDist to the distance to this bottle, which we will then later display on the screen.

```
void DetectInput(double time)
{
    DIMOUSESTATE mouseCurrState;

    BYTE keyboardState[256];

    DIKeyboard->Acquire();
    DIMouse->Acquire();

    DIMouse->GetDeviceState(sizeof(DIMOUSESTATE), &mouseCurrState);

    DIKeyboard->GetDeviceState(sizeof(keyboardState),(LPVOID)&keyboardState);

    if(keyboardState[DIK_ESCAPE] & 0x80)
        PostMessage(hwnd, WM_DESTROY, 0, 0);

    float speed = 10.0f * time;

    if(keyboardState[DIK_A] & 0x80)
    {
        moveLeftRight -= speed;
    }
    if(keyboardState[DIK_D] & 0x80)
    {
        moveLeftRight += speed;
    }
    if(keyboardState[DIK_W] & 0x80)
    {
        moveBackForward += speed;
    }
    if(keyboardState[DIK_S] & 0x80)
    {
        moveBackForward -= speed;
```

Clean Up

Don't forget to release our bottles vertex and index buffers.

```
void CleanUp()
{
    SwapChain->SetFullscreenState(false, NULL);
    PostMessage(hwnd, WM_DESTROY, 0, 0);

    //Release the COM Objects we created
    SwapChain->Release();
    d3d11Device->Release();
    d3d11DevCon->Release();
    renderTargetView->Release();
    VS->Release();
    PS->Release();
    VS_Buffer->Release();
    PS_Buffer->Release();
    vertLayout->Release();
    depthStencilView->Release();
    depthStencilBuffer->Release();
    cbPerObjectBuffer->Release();
    Transparency->Release();
    CCWcullMode->Release();
    CWcullMode->Release();

    d3d101Device->Release();
    keyedMutex11->Release();
    keyedMutex10->Release();
    D2DRenderTarget->Release();
    Brush->Release();
    BackBuffer11->Release();
    sharedTex11->Release();
    DWriteFactory->Release();
    TextFormat->Release();
    d2dTexture->Release();

    cbPerFrameBuffer->Release();
```

The pickRayVector() Function

This is the function that will transform our cursor position to a 3d ray in world space. This was explained at the top, so I shouldn't really have to say much about it.

```
void pickRayVector(float mouseX, float mouseY, XMVECTOR& pickRayInWorldSpacePos, XMVECTOR& pi
{
    XMVECTOR pickRayInViewSpaceDir = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
    XMVECTOR pickRayInViewSpacePos = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);

    float PRVecX, PRVecY, PRVecZ;

    //Transform 2D pick position on screen space to 3D ray in View space
    PRVecX =  (( 2.0f * mouseX) / ClientWidth ) - 1 ) / camProjection(0,0);
    PRVecY = -(( 2.0f * mouseY) / ClientHeight) - 1 ) / camProjection(1,1);
    PRVecZ =  1.0f;    //View space's Z direction ranges from 0 to 1, so we set 1 since the ra

    pickRayInViewSpaceDir = XMVectorSet(PRVecX, PRVecY, PRVecZ, 0.0f);

    //Uncomment this line if you want to use the center of the screen (client area)
    //to be the point that creates the picking ray (eg. first person shooter)
    //pickRayInViewSpaceDir = XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f);

    // Transform 3D Ray from View space to 3D ray in World space
    XMMATRIX pickRayToWorldSpaceMatrix;
    XMVECTOR matInvDeter;    //We don't use this, but the xna matrix inverse function requires

    pickRayToWorldSpaceMatrix = XMMatrixInverse(&matInvDeter, camView);    //Inverse of View

    pickRayInWorldSpacePos = XMVector3TransformCoord(pickRayInViewSpacePos, pickRayToWorldS
    pickRayInWorldSpaceDir = XMVector3TransformNormal(pickRayInViewSpaceDir, pickRayToWorld
}
```

The Pick() Function

This is the funciton that will check our object to see if our pick ray intersects with it. This funciton returns a float, which is the distance to the picked object. If the object was not picked, this function returns FLT_MAX.

```
float pick(XMVECTOR pickRayInWorldSpacePos,
    XMVECTOR pickRayInWorldSpaceDir,
    std::vector<XMFLOAT3>& vertPosArray,
    std::vector<DWORD>& indexPosArray,
    XMMATRIX& worldSpace)
{
    //Loop through each triangle in the object
    for(int i = 0; i < indexPosArray.size()/3; i++)
    {
        //Triangle's vertices V1, V2, V3
        XMVECTOR tri1V1 = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
        XMVECTOR tri1V2 = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
        XMVECTOR tri1V3 = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);

        //Temporary 3d floats for each vertex
        XMFLOAT3 tV1, tV2, tV3;

        //Get triangle
        tV1 = vertPosArray[indexPosArray[(i*3)+0]];
        tV2 = vertPosArray[indexPosArray[(i*3)+1]];
        tV3 = vertPosArray[indexPosArray[(i*3)+2]];

        tri1V1 = XMVectorSet(tV1.x, tV1.y, tV1.z, 0.0f);
        tri1V2 = XMVectorSet(tV2.x, tV2.y, tV2.z, 0.0f);
        tri1V3 = XMVectorSet(tV3.x, tV3.y, tV3.z, 0.0f);

        //Transform the vertices to world space
        tri1V1 = XMVector3TransformCoord(tri1V1, worldSpace);
        tri1V2 = XMVector3TransformCoord(tri1V2, worldSpace);
        tri1V3 = XMVector3TransformCoord(tri1V3, worldSpace);

        //Find the normal using U, V coordinates (two edges)
        XMVECTOR U = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
```

The PointInTriangle() Function

This function was also explained above. It will check if a point on a triangles plane is inside the triangle. If the point is inside the triangle, it returns true, otherwise it returns false.

```
bool PointInTriangle(XMVECTOR& triV1, XMVECTOR& triV2, XMVECTOR& triV3, XMVECTOR& point )
{
    //To find out if the point is inside the triangle, we will check to see if the point
    //is on the correct side of each of the triangles edges.

    XMVECTOR cp1 = XMVector3Cross((triV3 - triV2), (point - triV2));
    XMVECTOR cp2 = XMVector3Cross((triV3 - triV2), (triV1 - triV2));
    if(XMVectorGetX(XMVector3Dot(cp1, cp2)) >= 0)
    {
        cp1 = XMVector3Cross((triV3 - triV1), (point - triV1));
        cp2 = XMVector3Cross((triV3 - triV1), (triV2 - triV1));
        if(XMVectorGetX(XMVector3Dot(cp1, cp2)) >= 0)
        {
            cp1 = XMVector3Cross((triV2 - triV1), (point - triV1));
            cp2 = XMVector3Cross((triV2 - triV1), (triV3 - triV1));
            if(XMVectorGetX(XMVector3Dot(cp1, cp2)) >= 0)
            {
                return true;
            }
            else
                return false;
        }
        else
            return false;
    }
    return false;
}
```

The Updated LoadObjModel() Function

Here is the function with the two new parameters which are vectors for the models vertex positions and index array.

```
bool LoadObjModel(std::wstring filename,
    ID3D11Buffer** vertBuff,
    ID3D11Buffer** indexBuff,
    std::vector<int>& subsetIndexStart,
    std::vector<int>& subsetMaterialArray,
    std::vector<SurfaceMaterial>& material,
    int& subsetCount,
    bool isRHCoordSys,
    bool computeNormals,
    ///////////////**************new**************////////////////////
    std::vector<XMFLOAT3>& vertPosArray,
    std::vector<DWORD>& vertIndexArray)
    ///////////////**************new**************////////////////////
{
```

Storing the Models Vertex Position List and Index List

The only modification in this function is very easy. All we do is store the position of each vertex in our vertPosArray vector, then store the entire index list in vertIndexArray.

```
    //Create our vertices using the information we got
    //from the file and store them in a vector
    for(int j = 0 ; j < totalVerts; ++j)
    {
        tempVert.pos = vertPos[vertPosIndex[j]];
        tempVert.normal = vertNorm[vertNormIndex[j]];
        tempVert.texCoord = vertTexCoord[vertTCIndex[j]];

        vertices.push_back(tempVert);

        //////////////**************new**************////////////////////
        //Copy just the vertex positions to the vector
        vertPosArray.push_back(tempVert.pos);
        //////////////**************new**************////////////////////
    }

    //////////////**************new**************////////////////////
    //Copy the index list to the array
    vertIndexArray = indices;
    //////////////**************new**************////////////////////
```

Loading the Models

We have added two arguments when loading our models. They are at the end, and are the vertex position and index list vectors.

```
if(!LoadObjModel(L"ground.obj", &meshVertBuff, &meshIndexBuff, meshSubsetIndexStart, meshSubs
    return false;
if(!LoadObjModel(L"bottle.obj", &bottleVertBuff, &bottleIndexBuff, bottleSubsetIndexStart, bo
    return false;
```

Setting the Bottles World Space Matrices

At the bottom of our initScene function, we have created a loop to go through each of the bottle world space matrices and set them in order, so that the bottles are all lined up in the world.

```
    float bottleXPos = -30.0f;
    float bottleZPos = 30.0f;
    float bxadd = 0.0f;
    float bzadd = 0.0f;

    for(int i = 0; i < numBottles; i++)
    {
        bottleHit[i] = 0;

        //set the loaded bottles world space
        bottleWorld[i] = XMMatrixIdentity();

        bxadd++;

        if(bxadd == 10)
        {
            bzadd -= 1.0f;
            bxadd = 0;
        }

        Rotation = XMMatrixRotationY(3.14f);
        Scale = XMMatrixScaling( 1.0f, 1.0f, 1.0f );
        Translation = XMMatrixTranslation( bottleXPos + bxadd*10.0f, 4.0f, bottleZPos + bz
        bottleWorld[i] = Rotation * Scale * Translation;
    }
```

The RenderText() Function

We want to display the score and distance to the last picked object on the screen, so we modify the RenderText() function to do this.

```
std::wostringstream printString;
printString << text << inInt << L"Wn"
    << L"Score: " << score << L"Wn"
    << L"Picked Dist: " << pickedDist;
```

The DrawScene() Function

We now need to display our bottles. First we loop through each bottle. If the bottle has not been hit yet, we will display it. Also, you might notice we do not display the transparent parts of the bottle in this lesson. That is because we know there are no transparent parts, so we will not waste space.

```
    //draw bottle's nontransparent subsets
    for(int j = 0; j < numBottles; j++)
    {
        if(bottleHit[j] == 0)
        {
            for(int i = 0; i < bottleSubsets; ++i)
            {
                //Set the grounds index buffer
                d3d11DevCon->IASetIndexBuffer( bottleIndexBuff, DXGI_FORMAT_R32_UINT, 0);
                //Set the grounds vertex buffer
                d3d11DevCon->IASetVertexBuffers( 0, 1, &bottleVertBuff, &stride, &offset );

                //Set the WVP matrix and send it to the constant buffer in effect file
                WVP = bottleWorld[j] * camView * camProjection;
                cbPerObj.WVP = XMMatrixTranspose(WVP);
                cbPerObj.World = XMMatrixTranspose(bottleWorld[j]);
                cbPerObj.difColor = material[bottleSubsetTexture[i]].difColor;
                cbPerObj.hasTexture = material[bottleSubsetTexture[i]].hasTexture;
                cbPerObj.hasNormMap = material[bottleSubsetTexture[i]].hasNormMap;
                d3d11DevCon->UpdateSubresource( cbPerObjectBuffer, 0, NULL, &cbPerObj, 0,
                d3d11DevCon->VSSetConstantBuffers( 0, 1, &cbPerObjectBuffer );
                d3d11DevCon->PSSetConstantBuffers( 1, 1, &cbPerObjectBuffer );
                if(material[bottleSubsetTexture[i]].hasTexture)
                    d3d11DevCon->PSSetShaderResources( 0, 1, &meshSRV[material[bottleSubs
                if(material[bottleSubsetTexture[i]].hasNormMap)
                    d3d11DevCon->PSSetShaderResources( 1, 1, &meshSRV[material[bottleSubs
                d3d11DevCon->PSSetSamplers( 0, 1, &CubesTexSamplerState );

                d3d11DevCon->RSSetState(RSCullNone);
                int indexStart = bottleSubsetIndexStart[i];
                int indexDrawAmount =  bottleSubsetIndexStart[i+1] - bottleSubsetIndexStart[
                if(!material[bottleSubsetTexture[i]].transparent)
                    d3d11DevCon->DrawIndexed( indexDrawAmount, indexStart, 0 );
```

The WndProc() Function

Now we go to our windows procedure function, which checks for windows messages. We need to check for a window resize message. This message will be sent when the program first starts, and sent every time we change the size of our window. We need to do this and store the windows new height and width in the global variables ClientWidth and ClientHeight.

```
LRESULT CALLBACK WndProc(HWND hwnd,
    UINT msg,
    WPARAM wParam,
    LPARAM lParam)
{
    switch( msg )
    {
    case WM_KEYDOWN:
        if( wParam == VK_ESCAPE ){
            DestroyWindow(hwnd);
        }
        return 0;

    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    //////////////**********new**********//////////////////
    case WM_SIZE:
        ClientWidth  = LOWORD(lParam);
        ClientHeight = HIWORD(lParam);
        return 0;
    }
    //////////////**********new**********//////////////////
    return DefWindowProc(hwnd,
        msg,
        wParam,
        lParam);
}
```

Now we know how to pick 3D models on the screen with Direct3D 11! Hope you got some use out of this! Don't forget to leave a comment if you find any mistakes or just found it usefull or anything!

Exercise:

1. Make a First Person Shooter style picking, where objects in the center of the screen are picked rather than under the cursor.

2. Make a First Person Shooter!