25. Bounding Volumes

Think if your scene has a couple thousand models in it, and each model has a couple thousand triangles. Now if you were to pick them using just the picking method from the lesson above, it might take several seconds just to finish that picking operation. Several seconds on a single frame is just not acceptable, and this is where bounding volumes come into play.

We will be learning how to create and use a Bounding Box and a Bounding Sphere. The bounding box is usually more accurate than the bounding sphere, but it also takes a little more time to do the picking operation than the sphere takes.

We will use our High Resolution timer from an ealier lesson to time exactly how long the operation for each picking method takes.

Introduction

Here we will learn how to create bounding volumes. Bounding volumes are used to speed up operations like picking and collision testing.

In this lesson we will render 1000 bottles (the previous lesson was only 20). Try running the last lesson using 1000 bottles. You will see that every time you try to pick one of the bottles, the game freezes for a small amount of time. Well thats no good! We can have games freezing every time we try to pick an object in the scene! This is where bounding volumes come into play. Instead of testing against every triangle in every object, we will only test agains the few triangles in a bounding box, or test whether or not our picking ray is within the range of the bounding sphere of the model. This speeds things up SOOO much!

We will be making a bounding box and a bounding sphere. Later, for real complex models, you might consider making a bounding model for that complex one, which contains many many less triangles to test against. We do not render bounding volumes, they are used only for testing usually.

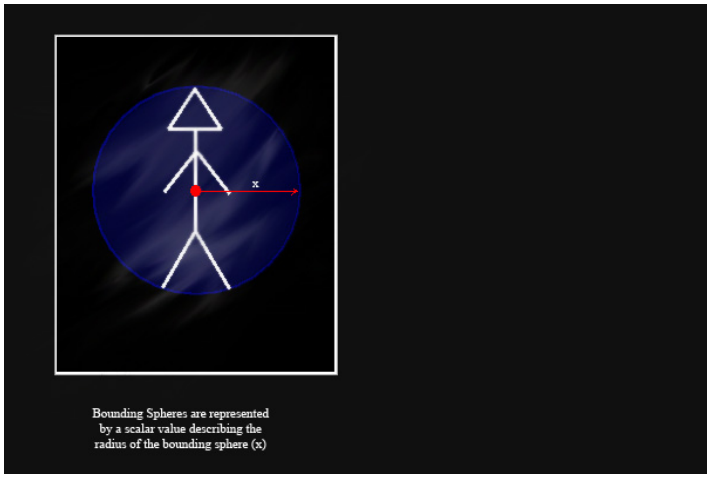
Bounding Volumes

As I explained above, bounding volumes are used for operations such as picking and collision detection. They speed up the time it takes to do these processes either by providing less geometry

(bounding box and bounding meshes) to test against, or by providing a much more efficient way of testing (bounding sphere). With the speed bounding volumes provide, they also provide less accuracy, the sphere being the least accurate (usually) than the bounding box or bounding mesh.

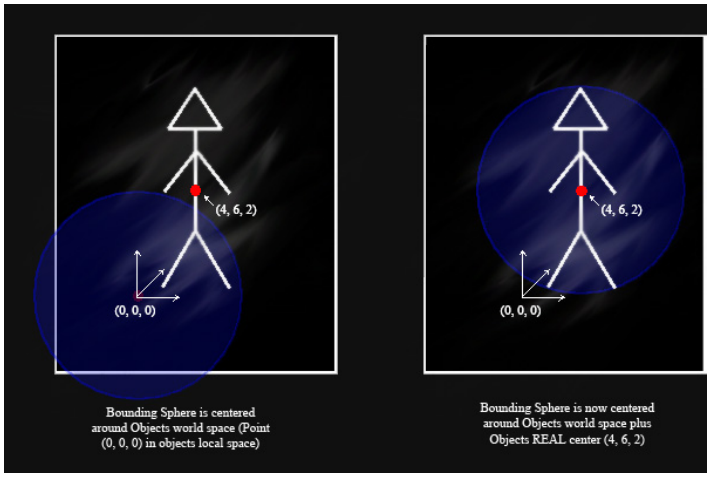
The less accuracy thing is nothing to worry about though! First of all, less accuracy can actually be something you want in your game, maybe for certain objects you want to make sure are a little easier to pick by "estimating" their position when picking. But the real reason the less accuracy of the bounding volumes is nothing to worry about, is because you are still able to do the accurate testing of the model itself. How this works, is instead of testing every single model for picking, we only test their bounding volume. If the bounding volume was picked, then we can test the actual model. We still get the high performance of the bounding sphere, and the accuracy of testing against the model itself!

Bounding Sphere



The bounding sphere is a scalar value that defines the radius of the bounding sphere. This is the fastest way to detect picking or collision. When testing for picking, we only have to see if the picking ray comes within the bounding spheres radius of the center of the object we are testing for picking. For collision detection (next lesson), we only need to test if the distance between the center of two objects are within the summed value of both of their bounding spheres radius'.

To test for picking with the bounding sphere, first we will find the closest point on the ray to the center of the object. You will notice we create a vector called objectCenterOffset when creating the bounding sphere. This is because a model might not (and usually won't be) centered around the point (0, 0, 0) in its local space, and we will be using the objects world space as the center point of the model, which is probably not the REAL center of the model. objectCenterOffset is a vector we will add to the objects world space, so we can use the objects REAL center in world space when checking for picking. Otherwise the bounding sphere will not cover the model correctly.



We can find the closest point on the ray to the center of the object, using this equation: (Where N is

the closest point on the ray to the center of the object, prO is the position or origin of the pick ray, prD is the direction of the pick ray, and oP is the objects position)

```
N = prO + Dot((oP - prO), prD) / Dot(prD, prD) * prD;
```

After we have the nearest point on the ray to the objects position, all we have to do is find the distance between them, which we can do simply with the xna vector function XMVector3Length(), which returns a vector with each of the x, y, z, and w components holding the same value, which is the distance between the two vectors we used as arguments to the function. We can extract one of the components from the returned vector to get the scalar value which is the distance between the two points using the xna vector function XMVectorGetX (We could also change "X" in this function to either "Y", "Z", or even "W", since they all hold the same value after the length function).

Finally, we check to see if the value returned by XMVector3Length() is less than our objects bounding sphere value. If it is, then the pick ray has intersected with the bounding sphere, and we can move on to a more accurate test, like using the actual model for testing, which we will do in this lesson, although I did put in a commented out line which we could use instead of testing the model itself if you wanted to only test against the bounding sphere.

Bounding Box

Bounding Boxes are usually described by two vertices, or vectors, a min and a max, which we can find by iterating through the models vertices, and store the smallest x, y, and z values in the min vector, and the largest x, y, and z values in the max vector. We will use these two vectors in this lesson to create an actual box mesh so we can check it for an intersection with the picking ray.

There are two kinds of Bounding Boxes, Axis-Aligned Bounding Boxes (AABB), and Oriented Bounding Boxes (OBB). AABB's faces are aligned with the world space axes, while OBB's faces are aligned with the objects space axes. In this lesson however, we will not be using the bounding box in this sense, but instead as an entire mesh. We will learn how to use AABB's and OBB's in the next lesson for collision detection.

Bounding boxes are usually a little more accurate, especially for "long" models, like a space shit or pencil. They are slightly more time consuming than the bounding sphere though, as we now need to test each of the 12 triangles which make up the bounding box if the pick ray intersects with them. Testing if a ray intersects with a triangle takes more computing power and time than just testing if a ray is within a certain distance to a point. However, you can imagine how much faster testing 12 triangles for an intersection is than testing thousands of triangles in a model.

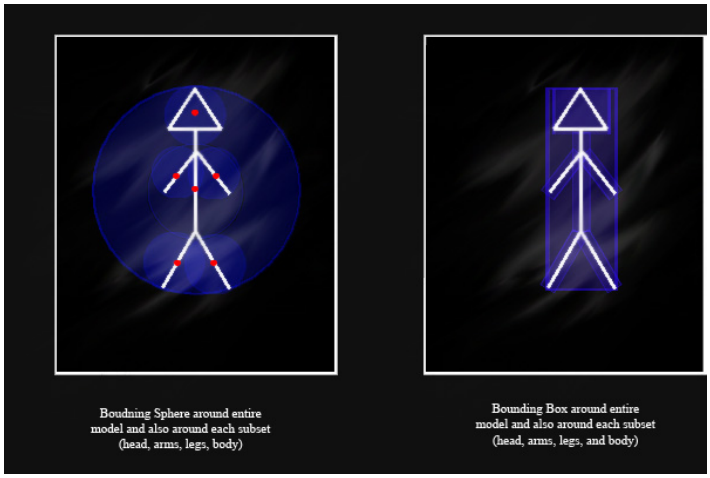
Checking for picking with a bounding box is the same as checking picking for a model, so we can use the same pick method for that.

Bounding Model (Bounding Mesh)

This is something you would create in the modeling program you made your original model in, so we will not be using this in this lesson. If you have a REEEEALY complex model, with a TON of triangles, and just using a bounding box or bounding sphere is too inaccurate, and it would just take too long to test the actual model for picking or collision, you could use something called a bounding model. What this is, is a model made in the a modeling program, that does not get rendered to the scene. It is only used for testing purposes. It is the same shape (or better approximation) of the model you want to do testing for, but contains MANY MANY less triangles to test against.

Bounding Models are tested the same way as you would test the original model, but contain less triangles in order to speed up the process.

Subsets



One last thing I'll mention before we get started, is bounding volumes for subsets. Suppose you have a first person shooter, and you want headshots to be fatal, while shots to the limbs doing less damage. To do this, you might want to create separate bounding volumes for each of the subsets on this model. Maybe the entire model has a bounding box, and when the bounding box has passed the picking test, you want to test each of the limbs, body, and head for picking. You could test each of their subsets, and finally, if one of their subsets has passed, do the actual picking against the subsets itself.

New Globals for Models

We have some new globals for models that we load in. These globals have to do with our bounding volumes. The first is a float value describing the radius of the models bounding sphere. The second two store the bounding box geometry, and the last one is a vector describing the difference between the (0, 0, 0) in the models local space (which is where the bounding box is centered around in world space by default), and the models real center. Using this vector, we can center the models bounding sphere around the models real center, and not just the point (0, 0, 0) in the models local space. (This was explained above)

I forgot to add in the number of bottles we are going to be using. Of course, you can choose any number you want, but 1000 bottles should be enough for you to see quite a difference between each of the picking methods (bounding box, bounding sphere, and model)

```
XMMATRIX bottleWorld[1000];
int* bottleHit = new int[1000];
int numBottles = 1000;

float bottleBoundingSphere = 0.0f;
std::vector<XMFLOAT3> bottleBoundingBoxVertPosArray;
std::vector<DWORD> bottleBoundingBoxVertIndexArray;
XMVECTOR bottleCenterOffset;
```

Three More Global Declarations

The first one is used so we can change between picking methods (eg. bounding sphere, bounding box, model) when we press "p" on the keyboard. The second is the time it takes from when we start our picking operation, until we end. This is so we can see how much faster it is to use bounding volumes than just pick the model directly, which can be the difference in SECONDS! Being stuck on a single frame while the picking operation is being completed is not going to happen in OUR games!

The last one is just so we can keep track of when the key "p" is being pressed, so we only change between picking methods ONCE per keydown, otherwise, it will flip through picking methods each frame the key "p" is held down.

```
int pickWhat = 0;

double pickOpSpeed = 0.0f;

bool isPDown = false;
```

The CreateBoundingVolumes() Function Prototype

Here is the prototype of the function we will call to create the bounding volumes for our models. The first parameter is the array or vector storing our models vertex positions. The second is the returned array storing our bounding box vertex positions, third is the bounding box's index array, which we will need for our picking operation. The fourth is a float value holding the radius of our bounding sphere, and the fifth is the vector describing our models REAL center in model space.

```
void CreateBoundingVolumes(std::vector<XMFLOAT3> &vertPosArray,     // The array containing
    std::vector<XMFLOAT3>& boundingBoxVerts,                        // Array we want to st
    std::vector<DWORD>& boundingBoxIndex,                          // This is our bounding b
    float &boundingSphere,                                          // The float containing
    XMVECTOR &objectCenterOffset);                                 // A vector containing t
```

Picking Bounding Volumes

Let's now go down to the function where we detect input. Remember in our last lesson we are using the left mouse button to detect picking. We will go through each model that we want to detect picking for, and then do our picking operation

Notice the line double pickOpStartTime = GetTime();. This line will store the time right before we start our picking operation. If you look past our picking operation, you will see the other line to this, pickOpSpeed = GetTime() - pickOpStartTime;. This line will get the difference from the current time now, and the time before we started our picking operation, and store it in our global variable pickOpSpeed. This value is the time in seconds it takes to complete our picking operation, which down further you can see we will be displaying on the screen.

```
    if(mouseCurrState.rgbButtons[0])
    {
        if(isShoot == false)
        {
            POINT mousePos;

            GetCursorPos(&mousePos);
            ScreenToClient(hwnd, &mousePos);

            int mousex = mousePos.x;
            int mousey = mousePos.y;

            float tempDist;
            float closestDist = FLT_MAX;
            int hitIndex;

            XMVECTOR prwsPos, prwsDir;
            pickRayVector(mousex, mousey, prwsPos, prwsDir);

            double pickOpStartTime = GetTime();         // Get the time before we start our

            for(int i = 0; i < numBottles; i++)
            {
                if(bottleHit[i] == 0) // No need to check bottles already hit
                {
                    tempDist = FLT_MAX;

                    if(pickWhat == 0)
                    {
                        float pRToPointDist = 0.0f; // Closest distance from the pick ray t

                        XMVECTOR bottlePos = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
                        XMVECTOR pOnLineNearBottle = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
```

Choosing the Best Bounding Volume for the Job

This is where we have an option of how we are going to be detecting picking. We can choose a variety of different methods, but we will choose three different ways here.

If pickWhat is 0, we will check for picking with the bounding sphere. If the models bounding sphere was picked, we can move on to the more accurate picking method of directly checking the model. As you will find out, this is by far the quickest method.

The second method usually offers a bit more accuracy than just the bounding sphere alone (without checking the model directly), at least for "long" models. Checking for picking with this takes a tad longer as you will see when you run this lessons code.

Finally, if pickWhat is 2, We check the model for picking DIRECTLY without using bounding volumes. You will notice this takes MUCH MUCH longer, up to seconds to complete the operation. As you can see, this is exactly why we use bounding volumes ;)

```
                    if(pickWhat == 0)
                    {
                        float pRToPointDist = 0.0f; // Closest distance from the pick ray t

                        XMVECTOR bottlePos = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
                        XMVECTOR pOnLineNearBottle = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);

                        // For the Bounding Sphere to work correctly, we need to make sure w
                        // the distance from the objects "actual" center and the pick ray. W
                        // the distance from (0, 0, 0) in the objects model space to the obj
                        // center in bottleCenterOffset. So now we just need to add that dif
                        // the bottles world space position, this way the bounding sphere wi
                        // on the object real center.
                        bottlePos = XMVector3TransformCoord(bottlePos, bottleWorld[i]) +

                        // This equation gets the point on the pick ray which is closest to
                        pOnLineNearBottle = prwsPos + XMVector3Dot((bottlePos - prwsPos), p

                        // Now we get the distance between bottlePos and pOnLineNearBottle
                        // This line is slightly less accurate, but it offers a performance
                        // estimating the distance using XMVector3LengthEst()
                        //pRToPointDist = XMVectorGetX(XMVector3LengthEst(pOnLineNearBottle
                        pRToPointDist = XMVectorGetX(XMVector3Length(pOnLineNearBottle -

                        // If the distance between the closest point on the pick ray (pOnLin
                        // is less than the bottles bounding sphere (represented by a float
                        // then we know the pick ray has intersected with the bottles boundi
                        // to testing if the pick ray has actually intersected with the bott
                        if(pRToPointDist < bottleBoundingSphere)
                        {
                            // This line is the distance to the pick ray intersection with t
                            //tempDist = XMVectorGetX(XMVector3Length(pOnLineNearBottle - pr
```

The CreateBoundingVolumes() Function

This is the function we call when we want to create bounding volumes for a model. The first thing we do is find two points, which contain the maximum and minimum x, y, and z values. We loop through each vertex in the array passed in (vertPosArray), and first check the minimum x, then y, then z agains the passed in vertex position. If the passed in vertex position (either x, y, OR z) is SMALLER than the one (x, y OR z) stored in minVertex, we update minVertex's x, y, or z component to the x, y, or z component thats smaller in the passed in vertex position. Then we do the same for the maxVertex, except of course looking for the largest values instead of the smallest. The result of this is two points, when used as opposite corners of a box, can create a box that TIGHTLY covers the entire model.

Next you will see we will also use these two points for our bounding sphere. The first thing we do is find the point in the center of these two points, and store that point in objectCenterOffset, which we can later use to make sure our bounding sphere is centered around the objects real center in world space, and not just the point (0, 0, 0) in the object model space. After we have the objects real center, we can create our bounding sphere. This is easy, as all we have to do is find the distance between the center of the model to either the min or max vertex (since both of these vertices are equally far away from the center). We can find the difference between two vectors by calling the xna function XMVector3Length().

Next we store the 8 vertices that will make up our bounding box in the boundingBoxVerts vector using the x, y, and z values of our min and max vertices. And finally, we create and store our bounding box's vertex indices in boundingBoxIndex.

```
void CreateBoundingVolumes(std::vector<XMFLOAT3> &vertPosArray,
    std::vector<XMFLOAT3>& boundingBoxVerts,
    std::vector<DWORD>& boundingBoxIndex,
    float &boundingSphere,
    XMVECTOR &objectCenterOffset)
{
    D3DXVECTOR3 minVertex = D3DXVECTOR3(FLT_MAX, FLT_MAX, FLT_MAX);
    D3DXVECTOR3 maxVertex = D3DXVECTOR3(-FLT_MAX, -FLT_MAX, -FLT_MAX);

    for(UINT i = 0; i < vertPosArray.size(); i++)
    {
        // The minVertex and maxVertex will most likely not be actual vertices in the model,
        // that use the smallest and largest x, y, and z values from the model to be sure AL
        // covered by the bounding volume

        //Get the smallest vertex
        minVertex.x = min(minVertex.x, vertPosArray[i].x);    // Find smallest x value in mo
        minVertex.y = min(minVertex.y, vertPosArray[i].y);    // Find smallest y value in mo
        minVertex.z = min(minVertex.z, vertPosArray[i].z);    // Find smallest z value in mo

        //Get the largest vertex
        maxVertex.x = max(maxVertex.x, vertPosArray[i].x);    // Find largest x value in moo
        maxVertex.y = max(maxVertex.y, vertPosArray[i].y);    // Find largest y value in moo
        maxVertex.z = max(maxVertex.z, vertPosArray[i].z);    // Find largest z value in moo
    }

    // Compute distance between maxVertex and minVertex
    float distX = (maxVertex.x - minVertex.x) / 2.0f;
    float distY = (maxVertex.y - minVertex.y) / 2.0f;
    float distZ = (maxVertex.z - minVertex.z) / 2.0f;

    // Now store the distance between (0, 0, 0) in model space to the models real center
    objectCenterOffset = XMVectorSet(maxVertex.x - distX, maxVertex.y - distY, maxVertex.z
```

Creating Our Bottle's Bounding Volumes

Now we need to call the CreateBoundingVolumes() function to create our bottles bounding volumes. We first pass in our bottles vertex positions, and then the rest of the parameters are what the bounding volumes information will be stored in.

```
CreateBoundingVolumes(bottleVertPosArray, bottleBoundingBoxVertPosArray, bottleBoundingBoxVe
```

New Text Stuff

Now we go down to our RenderText() function, where we will display some extra information relating to this lesson. We want to display the picking method we are currently using, so we first check pickWhat and set the string pickWhatStr to the method we are currently using. Then we also display the time it takes to finish the picking operation in seconds, pickOpSpeed.

```
    // Display which picking method we are doing
    std::wstring pickWhatStr;
    if(pickWhat == 0)
        pickWhatStr = L"Bounding Sphere";
    if(pickWhat == 1)
        pickWhatStr = L"Bounding Box";
    if(pickWhat == 2)
        pickWhatStr = L"Model";

    //Create our string
    std::wostringstream printString;
    printString << text << inInt << L"\n"
        << L"Score: " << score << L"\n"
        << L"Picked Dist: " << pickedDist << L"\n"
        << L"Pick Operation Speed: " << pickOpSpeed << L"\n"
        << L"Picking Method (P): " << pickWhatStr;
```

If you will be doing any sort of collision testing or picking in your game, which is most likely the case, you now know how to do it more efficiently!

Exercise:

1. Modify the CreateBoundingVolumes() function to create bounding volumes for each of a models subsets (eg. a mans legs, arms, head).