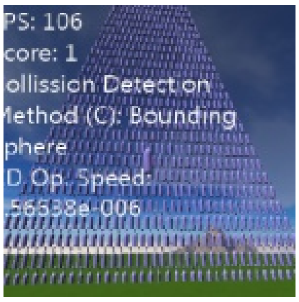
26. Bounding Volume Collision Detection



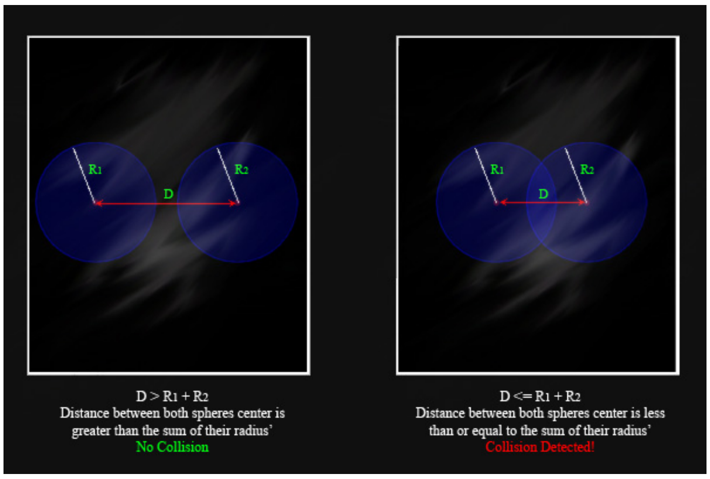
We will learn how to detect a collision between two objects in this lesson. We will be learning how to use their bounding volumes to do the collision detection instead of the object themselves, as it is much more simpler, and much faster to compute than triangle to triangle collision detection.

In this lesson, we will build a pyramid of bottles or something, and "throw" a bottle when the mouse button is clicked. If the "thrown" bottle collides with one or more of the bottles in the pyramid, then they both just dissapear and the score is increased. It is not very realistic physics, but I have decided to skip all that extra stuff so we can just focus plainly on the actual collision detection methods.

Introduction

This lesson is actually very simple really. We will be detecting collisions between two objects using their bounding volumes. We can switch between bounding sphere and bounding box collision tests. The bounding box collision test is usually MUCH more accurate as you will see in this lesson, at least for "long" objects. Of course if you were testing collision tests between objects more "round" without any protrusions, a bounding sphere might be more accurate. I would use bounding boxes for the most part, but I want to show you how to do both collision tests here.

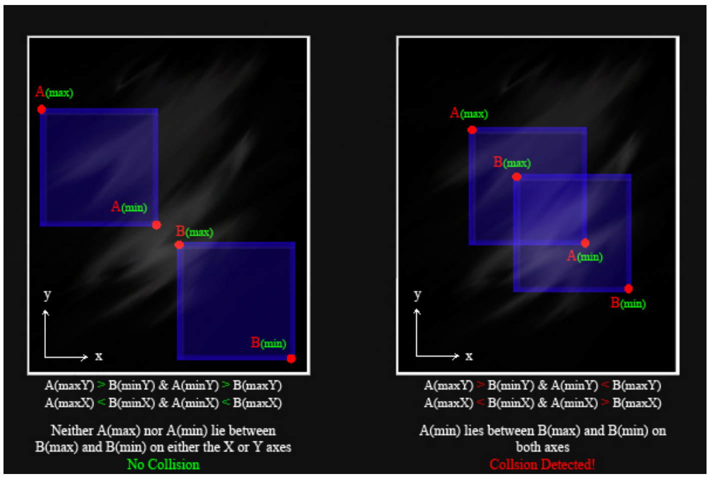
Bounding Sphere Collision Test


R1
R2
D
D > R1 + R2
Distance between both spheres center is greater than the sum of their radius'
No Collision
D <= R1 + R2
Distance between both spheres center is less than or equal to the sum of their radius'
Collision Detected!

Bounding Sphere Collision Detection

We can check for a collision between two bounding spheres by finding the distance between their centers, and comparing it with the sum of their radius'. If the distance between their centers is LESS than the sum of their radius', they are colliding, otherwise, there is no collision.

Axis-Aligned Bounding Box Collision Test (AABB)


A(max)
A(min)
B(max)
B(min)
y
x
A(maxY) > B(minY) & A(minY) > B(maxY)
A(maxX) < B(minX) & A(minX) < B(maxX)
Neither A(max) nor A(min) lie between B(max) and B(min) on either the X or Y axes
No Collision
A(max)
B(max)
A(min)
B(min)
y
x
A(maxY) > B(minY) & A(minY) < B(maxY)
A(maxX) < B(minX) & A(minX) > B(maxX)
A(min) lies between B(max) and B(min) on both axes
Collsion Detected!

Notice how the faces of the two boxes are aligned with the world space axes. This is an AABB. AABB collision detection is much more simple than OBB (oriented bounding box) because OBB's can be rotated, which means that the detection method we will be using for bounding boxes no longer work. One problem with AABB's is that they need to be "re-calculated" every time the object is transformed, unlike the OBB (since the OBB will be transformed with the object). Another thing about AABB's is they are usually a lot less accurate than OBB's. In this lesson we will be learning how to check for a collision between two AABB's, since this test is very easy to impliment.

Since AABB's have to be aligned to the 3 axes in world space, we need to "re-calculate" the min and max vectors of the AABB EVERY time the object is transformed. This means that the AABB is actually described in world space, so we do not have to do any transformations from object to world space when checking for collisions.

Axis-Aligned Bounding box collision tests might be slightly more complicated than bounding sphere collision tests, but once you understand it, you will see it's not much more complicated than the bounding sphere collision test.

An AABB (Axis-Aligned Bounding Box) can be described by two vectors. One vector holds the smallest, or minimum x, y, and z values, while the other holds the largest, or maximum x, y, and z values. We need to check if the two bounding boxes intersect on all three axes (x, y, z). You can see an example of how this works in 2D from the picture above. For 3D, we just need to add the third dimension, z.

First we check if they intersect on the X axis. If the first objects (obj1 from now on) MaxX is LESS than the second objects (obj2 from now on) MinX, we know that obj1 is completely to the left of obj2, which also tells us there was no collision, otherwise, we move on the check if obj1's MinX is GREATER than obj2's MaxX. If this is true, we know obj1 is completely to the right of obj2, so no collision. Then we do the same for the Y and Z axes. If there was in intersection on ALL three axes, we know there was a collision!

New Globals for Models

We need to keep track of EACH model's AABB's min and max vertices, which describe the min and max x, y, and z values for the model, which we will use later for collision detection.

We also have another model (which actually uses the same bottle model as the others), which is the bottle we will be throwing at the other bottles. The first two are it's AABB's min and max vectors, the second is it's position in world space, third is the direction it's traveling in, and the fourth is a bool so that the bottle is only drawn after we throw it and before it hits another bottle, so that it dissapears when it hits another bottle.

```
XMVECTOR bottleBoundingBoxMinVertex[820];
XMVECTOR bottleBoundingBoxMaxVertex[820];

XMVECTOR thrownBottleBoundingBoxMinVertex;
XMVECTOR thrownBottleBoundingBoxMaxVertex;
XMMATRIX thrownBottleWorld;
XMVECTOR thrownBottleDir;
bool bottleFlying;
```

Three More Global Declarations

The first is because we will be switching between bounding sphere and bounding box collision tests. The second keeps track of the time it takes to complete the collision detection (in this lesson it will be pretty much zero whichever method we use, but in the next lesson, triangle to triangle collision detection, we will see a big difference in time). The third is just to keep track of when the "C" key is being pressed.

```
int cdMethod = 0;

double cdOpSpeed = 0.0f;

bool isCDown = false;
```

The Bounding Volume Collision Detection Function Prototypes

We have two new functions. One for bounding sphere collision tests, and the other for bounding box collision tests.

```
bool BoundingSphereCollision(float firstObjBoundingSphere,
    XMVECTOR firstObjCenterOffset,
    XMMATRIX& firstObjWorldSpace,
    float secondObjBoundingSphere,
    XMVECTOR secondObjCenterOffset,
    XMMATRIX& secondObjWorldSpace);

bool BoundingBoxCollision(XMVECTOR& firstObjBoundingBoxMinVertex,
    XMVECTOR& firstObjBoundingBoxMaxVertex,
    XMMATRIX& firstObjWorldSpace,
    XMVECTOR& secondObjBoundingBoxMinVertex,
    XMVECTOR& secondObjBoundingBoxMaxVertex,
    XMMATRIX& secondObjWorldSpace);
```

Calculating the AABB

We have another function that we will call whenever an object is transformed. The pyramid of bottles will only be transformed once during set-up, so we only need to call this function once for each of them when setting up. The "thrown" bottle however will be updated each frame, so we need to call this function each time it's updated.

```
void CalculateAABB(std::vector<XMFLOAT3> boundingBoxVerts,
    XMMATRIX& worldSpace,
    XMVECTOR& boundingBoxMin,
    XMVECTOR& boundingBoxMax);
```

Throwing the Bottle!

I know... This is really a lame example, not very in depth either, but it should get the idea across. Anyway, instead of picking objects in this lesson, we will be throwing a bottle (which by the way doesn't use physics so gravity is not taken into account, its like throwing a bottle in space, except that it doesn't even spin, hehe). We need to check if the thrown bottle collides with any of the other bottles in the scene, which we will do later, in our update scene function. But for now, we check if the mouse button was pressed, if it was, we just chuck the bottle straight in the direction our camera is facing and set bottleFlying to true so we know we should be drawing the bottle now.

```
    if(mouseCurrState.rgbButtons[0])
    {
        if(isShoot == false)
        {
            bottleFlying = true;

            thrownBottleWorld = XMMatrixIdentity();
            Translation = XMMatrixTranslation( XMVectorGetX(camPosition), XMVectorGet

            thrownBottleWorld = Translation;
            thrownBottleDir = camTarget - camPosition;
            /*    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 p

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

The BoundingSphereCollision() Function

Here is our function to detect a collision between two bounding spheres in world space. It's pretty straight forward, and was explained above, so I don't think there's a whole lot to be said.

```
bool BoundingSphereCollision(float firstObjBoundingSphere,
    XMVECTOR firstObjCenterOffset,
    XMMATRIX& firstObjWorldSpace,
    float secondObjBoundingSphere,
    XMVECTOR secondObjCenterOffset,
    XMMATRIX& secondObjWorldSpace)
{
    //Declare local variables
    XMVECTOR world_1 = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
    XMVECTOR world_2 = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
    float objectsDistance = 0.0f;

    //Transform the objects world space to objects REAL center in world space
    world_1 = XMVector3TransformCoord(firstObjCenterOffset, firstObjWorldSpace);
    world_2 = XMVector3TransformCoord(secondObjCenterOffset, secondObjWorldSpace);

    //Get the distance between the two objects
    objectsDistance = XMVectorGetX(XMVector3Length(world_1 - world_2));

    //If the distance between the two objects is less than the sum of their bounding spheres..
    if(objectsDistance <= (firstObjBoundingSphere + secondObjBoundingSphere))
        //Return true
        return true;

    //If the bounding spheres are not colliding, return false
    return false;
}
```

The BoundingBoxCollision() Function

This is the function that will be called to check for bounding box collisions. Again, this was explained above, so it should be fairly easy to follow ;)

```
bool BoundingBoxCollision(XMVECTOR& firstObjBoundingBoxMinVertex,
    XMVECTOR& firstObjBoundingBoxMaxVertex,
    XMVECTOR& secondObjBoundingBoxMinVertex,
    XMVECTOR& secondObjBoundingBoxMaxVertex)
{
    //Is obj1's max X greater than obj2's min X? If not, obj1 is to the LEFT of obj2
    if (XMVectorGetX(firstObjBoundingBoxMaxVertex) > XMVectorGetX(secondObjBoundingBoxMinVe

        //Is obj1's min X less than obj2's max X? If not, obj1 is to the RIGHT of obj2
        if (XMVectorGetX(firstObjBoundingBoxMinVertex) < XMVectorGetX(secondObjBoundingBoxM

            //Is obj1's max Y greater than obj2's min Y? If not, obj1 is UNDER obj2
            if (XMVectorGetY(firstObjBoundingBoxMaxVertex) > XMVectorGetY(secondObjBounding

                //Is obj1's min Y less than obj2's max Y? If not, obj1 is ABOVE obj2
                if (XMVectorGetY(firstObjBoundingBoxMinVertex) < XMVectorGetY(secondObjBoun

                    //Is obj1's max Z greater than obj2's min Z? If not, obj1 is IN FRONT OF o
                    if (XMVectorGetZ(firstObjBoundingBoxMaxVertex) > XMVectorGetZ(secondObj

                        //Is obj1's min Z less than obj2's max Z? If not, obj1 is BEHIND obj2
                        if (XMVectorGetZ(firstObjBoundingBoxMinVertex) < XMVectorGetZ(secon

                            //If we've made it this far, then the two bounding boxes are colli
                            return true;

    //If the two bounding boxes are not colliding, then return false
    return false;
}
```

The CalculateAABB() Function

This is where we will be calculating our object's AABB. We will be using the bounding box created during set-up to calculate the objects AABB. It's really simple, in fact, We pretty much did this same thing when we created our bounding box. The only difference is we are not creating the two vectors in World Space instead of Object Space, as you can see when we transform each bounding box's vertices to the objects world space.

```
void CalculateAABB(std::vector<XMFLOAT3> boundingBoxVerts,
    XMMATRIX& worldSpace,
    XMVECTOR& boundingBoxMin,
    XMVECTOR& boundingBoxMax)
{
    XMFLOAT3 minVertex = XMFLOAT3(FLT_MAX, FLT_MAX, FLT_MAX);
    XMFLOAT3 maxVertex = XMFLOAT3(-FLT_MAX, -FLT_MAX, -FLT_MAX);

    //Loop through the 8 vertices describing the bounding box
    for(UINT i = 0; i < 8; i++)
    {
        //Transform the bounding boxes vertices to the objects world space
        XMVECTOR Vert = XMVectorSet(boundingBoxVerts[i].x, boundingBoxVerts[i].y, boundingBo
        Vert = XMVector3TransformCoord(Vert, worldSpace);

        //Get the smallest vertex
        minVertex.x = min(minVertex.x, XMVectorGetX(Vert));    // Find smallest x value in m
        minVertex.y = min(minVertex.y, XMVectorGetY(Vert));    // Find smallest y value in m
        minVertex.z = min(minVertex.z, XMVectorGetZ(Vert));    // Find smallest z value in m

        //Get the largest vertex
        maxVertex.x = max(maxVertex.x, XMVectorGetX(Vert));    // Find largest x value in mo
        maxVertex.y = max(maxVertex.y, XMVectorGetY(Vert));    // Find largest y value in mo
        maxVertex.z = max(maxVertex.z, XMVectorGetZ(Vert));    // Find largest z value in mo
    }

    //Store Bounding Box's min and max vertices
    boundingBoxMin = XMVectorSet(minVertex.x, minVertex.y, minVertex.z, 0.0f);
    boundingBoxMax = XMVectorSet(maxVertex.x, maxVertex.y, maxVertex.z, 0.0f);
}
```

Updating the Scene

Since we are "throwing" a bottle, we need to make sure that it's position is updated every scene. This is also where we will check for collisions between the thrown bottle and the bottles in the pyramid thing. We first check if the bottle is "flying", if it is, we move on to update it's position in the world by adding it's direction vector (multiplying it by time to make sure it goes the same speed no matter the frame rate, and then by 10 to speed it up a little after it was multiplied by time) to it's current position in world space.

After we have updated it position, we need to check for collision. We iterate through each bottle in the pyramid, and skip the bottles that have already been hit. Now we have the option to do either a bounding sphere collision check or a bounding box collision check. We can change this by pressing "C" on the keyboard. You can also see we will be timing the speed of this collision check. In this lesson, you won't see much of a difference between these two collision tests, but I wanted to add it in anyway, since the next lesson is the very accurate triangle to triangle collision detection method, which is a much much slower process.

```
    //Update our thrown bottles position
    if(bottleFlying)
    {
        XMVECTOR tempBottlePos = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
        tempBottlePos = XMVector3TransformCoord(tempBottlePos, thrownBottleWorld) + (thrc
        Rotation = XMMatrixRotationY(3.14f);

        thrownBottleWorld = XMMatrixIdentity();
        Translation = XMMatrixTranslation( XMVectorGetX(tempBottlePos), XMVectorGetY

        thrownBottleWorld = Translation;

        for(int i = 0; i < numBottles; i++)
        {
            if(bottleHit[i] == 0) // No need to check bottles already hit
            {
                double cdOpStartTime = GetTime();
                if(cdMethod == 0)
                {
                    if(BoundingSphereCollision(bottleBoundingSphere, bottleCenterOffset,
                    {
                        bottleHit[i] = 1;
                        score++;
                        bottleFlying = false;
                    }
                }

                if(cdMethod == 1)
                {
                    if(BoundingBoxCollision(thrownBottleBoundingBoxMinVertex, thrownBottl
                    {
                        bottleHit[i] = 1;
                        score++;
```

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 collision detection method we will be using, which is either the bounding sphere or the bounding box. We will also be displaying the time it takes to finish the collision detection operation. Like I said a couple other times, you might not see a difference in speed yet, but next lesson you sure will ;)

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

    //Create our string
    std::wostringstream printString;
    printString << text << inInt << L"₩n"
        << L"Score: " << score << L"₩n"
        << L"Collission Detection Method (C): " << cdMethodString << L"₩n"
        << L"CD Op. Speed: " << cdOpSpeed;
```

Drawing the Thrown Bottle

Now we get to draw the thrown bottle. We first check if the bottle is flying (has been thrown but no collision was detected yet), then we draw it.

```
    if(bottleFlying)
    {
        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 = thrownBottleWorld * camView * camProjection;
            cbPerObj.WVP = XMMatrixTranspose(WVP);
            cbPerObj.World = XMMatrixTranspose(thrownBottleWorld);
            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, 0 );
            d3d11DevCon->VSSetConstantBuffers( 0, 1, &cbPerObjectBuffer );
            d3d11DevCon->PSSetConstantBuffers( 1, 1, &cbPerObjectBuffer );
            if(material[bottleSubsetTexture[i]].hasTexture)
                d3d11DevCon->PSSetShaderResources( 0, 1, &meshSRV[material[bottleSubsetTextu
            if(material[bottleSubsetTexture[i]].hasNormMap)
                d3d11DevCon->PSSetShaderResources( 1, 1, &meshSRV[material[bottleSubsetTextu
            d3d11DevCon->PSSetSamplers( 0, 1, &CubesTexSamplerState );

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

Bounding volumes are usually pretty important when it comes to collision detection in games, as directly testing each model themselves for a collision would not usually keep the game running smoothly, which we will see in the next lesson. Easy lesson, hope you got something out of it!

Exercise:

1. Find another way to detect collisions between bounding boxes (many ways)

2. Experiment with the number of bottles in the scene. Try to add a bunch more and see if you can find a noticeable difference in the speed of both collision tests.

3. Try to add some physics to the thrown bottle ;) like some spinning and take gravity into account when tossing it so it doesn't travel only in a straight line!