**Simtech 3D**

**1b/c**

Silvan Herrema

Frans-Jan Poort

16-02-2013

**The BasicModel class**

ball = new Ball(Game.Content.Load<Model>(@"Models/Ball LowDef"), new Vector3(0,20,0),1);

To draw a model in XNA the most important thing is of course the model itself. Which is loaded in XNA during the creation of a ‘BasicModel’ object in the LoadContent function. I decided to do that to keep the code well structured. You load objects in in the LoadContent function. The second thing a model needs is a Vector3 position. That vector3 is used to create a translation matrix.

public BasicModel(Model m, Vector3 position, float scale, Color colour)

{

model = m;

this.position = position;

scaleMatrix = Matrix.CreateScale(scale);

translationMatrix = Matrix.CreateTranslation(position);

this.colour = colour;

}

Further down the rabbit hole in the Draw() function. It draws the bones, gives it some lighting so you can see it’s a 3 dimensional object. And the AmbientLightColor gives the object the colour you passed along in the constructor.

public void Draw(Camera3D camera)

{

Matrix[] transforms = new Matrix[model.Bones.Count];

model.CopyAbsoluteBoneTransformsTo(transforms);

foreach (ModelMesh mesh in model.Meshes)

{

foreach (BasicEffect be in mesh.Effects)

{

be.EnableDefaultLighting();

be.AmbientLightColor = colour.ToVector3();

be.Projection = camera.projection;

be.View = camera.view;

be.World = GetWorld() \* mesh.ParentBone.Transform;

}

mesh.Draw();

}

}

I’ve used a matrix to create a ‘world’ for the model. The draw function is somewhat complicated, and hard to explain. But here’s how the ‘world’ matrix gets calculated. First I have the original matrix multiplied by the rotation matrix, because if you multiply it by the translation first, the model will still take the Origin to rotate around. But because the model is most likely a distance away from the Origin, you don’t want it to rotate around that. You want the model to rotate around its own centre. I used the scale matrix at the end, because I want to scale my object at the end, and I don’t want my translation to be scaled as well.

public virtual Matrix GetWorld()

{

return LocalMatrix() \* translationMatrix;

}

public virtual Matrix LocalMatrix()

{

Matrix w = Matrix.Identity;

return w \* rotationMatrix \* scaleMatrix;

}

I’ve separated the local and world transformation, because in case I want to use the local and world matrix separately.

**Action**

This is what the update function looks like in the ball. The plane doesn’t need an update. When space gets pressed gravity gets “added” to acceleration. It’s an ugly way to do it, but otherwise I’d have to reset acceleration to zero every frame and re-add gravity to it.

if(fallTime)

acceleration = gravity;

if (Keyboard.GetState().IsKeyDown(Keys.Space))

fallTime = true;

velocity += acceleration;

position += velocity;

**Gravity**

I already implemented gravity in assignment 1a. Does the ball gets to stand still / reach equilibrium? No. Because the lack of drag/lack of velocity loss it continues infinetly.

**Collision check**

Here is how the collision detection is handled. If the ball collides with the plane, the balls velocity gets reflected.

public bool CollidesWithPlane(Plane plane)

{

if (CollidesWithPlaneCheck(plane) == 0)

{

velocity = Vector3.Reflect(velocity, plane.normal);

return true;

}

return false;

}

public int CollidesWithPlaneCheck(Plane plane)

{

Vector3 planeNormal = plane.normal;

float d = Vector3.Dot(planeNormal, this.position)

– this.radius;

//is on front

if (d >= this.radius)

{

return +1;

}

//is on back of plane

if (d <= -this.radius)

{

return -1;

}

//intersects plane

return 0;

}

The CollidesWithPlaneCheck return an 0 if there’s a collision. If there’s a 1 or -1 there isn’t a collision. This method was copied from the book.

**The normal**

The plane’s normal we get by doing some ugly yet nice coding. Over the past week we tried getting the normal by using points from the plane gotten through the XNA pipeline, however, that didn’t really work well, because while we were looking for the corners, the XNA pipeline got the vertices from triangles created in the pipeline. Because the vertices were so well hidden in the XNA framework there wasn’t a direct way to access them, only indirect. And that method we used, returned vertices with a very minor deviation (2,22 E-16). In normal math it’s negligible, however, I suspect due to rounding errors, while transforming the vertice with a matrix it became a problem with calculating the normal and calculating the dot product for the collision detection.

public Plane(Model model, Vector3 position, float scale)

: base(model, position, scale, Color.Honeydew)

{

planeCorners2[0] = new Vector3(- 0.5f, 0, - 0.5f);

planeCorners2[1] = new Vector3(0.5f, 0, - 0.5f);

planeCorners2[2] = new Vector3(0.5f, 0, 0.5f);

planeCorners2[3] = new Vector3(- 0.5f, 0, 0.5f);

updateNormal();

}

So I declared the four corners like this. In updateNormal() I apply the world matrix to the corners so they will be in the same place as where the plane model is being drawn. This way there are no rounding errors and we get all the corners.

public void updateNormal()

{

for (int i = 0; i < 4; i++)

planeCorners2[i] = Vector3.Transform(planeCorners2[i], GetWorld());

this.normal = GetPlaneNormal();

}

private Vector3 GetPlaneNormal()

{

Vector3 cross = Vector3.Cross(planeCorners2[1] - planeCorners2[0],

planeCorners2[1] - planeCorners2[2]);

cross.Normalize();

return cross;

}

This is how I calculate how the ball moves upwards.