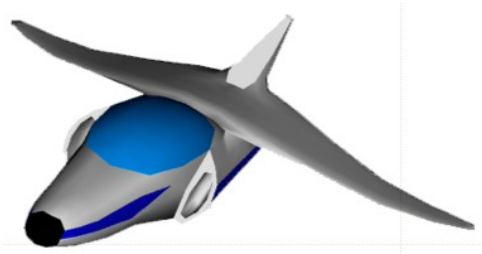


XNA Tutorials

Utah State University
Association for Computing Machinery
XNA Special Interest Group
RB Whitaker
17 October 2007

Lighting

Tutorial 12

Overview

We are still missing a very important feature in our scenes: lighting. With the things we did in the last tutorial, we will be able to make the changes we need fairly quickly. We will begin our lighting tutorial with a discussion of lighting, including the basic types of lighting, and the basic light source types. After that we will take a look at the changes we need to make to our code to actually perform lighting.

Light Types

In the world of computer graphics, there are four basic types of light: Diffuse, Specular, Ambient, and Emission. We will talk about each of these individually.

Diffuse Lighting

Diffuse lighting is the kind of lighting that most people think of first. Diffuse lighting is light that is reflected off of an object in equally every direction. This process of spreading out is called diffusion, so we call light like this diffuse lighting.

The intensity of diffuse lighting depends on the angle the surface makes with direction the light is coming from. You can see in the image of a ball, below, how the angle the surface is making with the light determines how bright the surface is.

The parts of the surface that are most perpendicular to the incoming light reflect the most, and so they are the brightest. As the surface angles away from the incoming light, it reflects less, and so it grows darker as you move around to the back side of the

ball. Eventually, if the surface is parallel with the incoming light rays, there is no reflected light, and so the surface is dark.

Diffuse lighting is the most important kind of lighting. Without it, our objects won't look good, but if we have it, we could almost do without any other kind of lighting.

Specular Light

Specular light is also easy to spot. These are the "shiny" spots on objects. The image below shows our ball with a specular highlight on it.

Specular lighting is affected both by where the light is with respect to the surface, as well as where the viewer is. You could imagine that if you were looking at a shiny surface and moved around, the shiny spots would also move around on the object, even though the surface is staying where it was, and so is the light.

Ambient Light

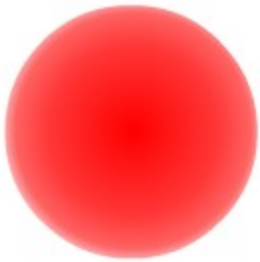
In almost every scenario, there is a little bit of light floating around that does not appear to be coming from a particular light source, but rather from the environment in general. This kind of "background" light is called ambient light. Our ball below has had some ambient light added to it.

This kind of light prevents things from being completely black. You can see that the back side of our ball now is lighter than it was before. The amount of ambient light in the above image is probably slightly exaggerated a little. There is probably not usually even that much ambient light. However, virtually every situation has some ambient light Even in deep space there is a small amount of ambient light coming from the background of space. Including a small amount of ambient light will go a long way to adding realism to our scene.

Emission Light

Emission light is light that is produced by a surface. It is important to know that usually in graphics, if a surface is "producing" emission light, it will *not* actually light up

other objects around it. That has to be done in other ways. The image below shows our ball with a high level of emission light.

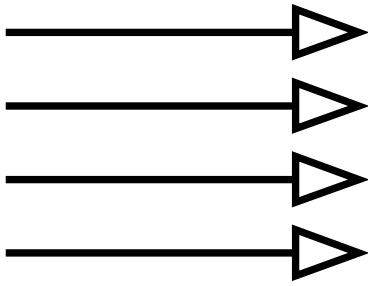


Light Sources

There are three major categories of light sources that are used: directional lights, point lights, and spot lights. We will talk about each of these individually as well.

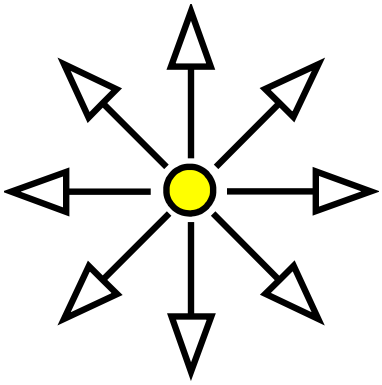
Directional Light

The best example of directional light is the light that comes from the sun. In a directional light, all of the rays are parallel. They are good for representing light sources that are quite a ways away, and are one of the most commonly used lighting types that there are.



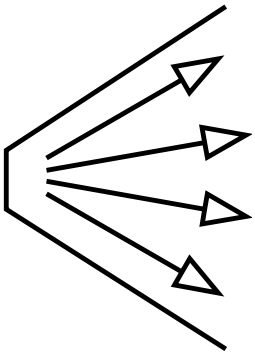
Point Lights

Light bulbs are a good example of point lights. In point light sources, all of the rays are coming from a particular point in space. Point lights are also used quite frequently, since you could use them to represent light bulbs, matches, and numerous other types of lights.



Spot Lights

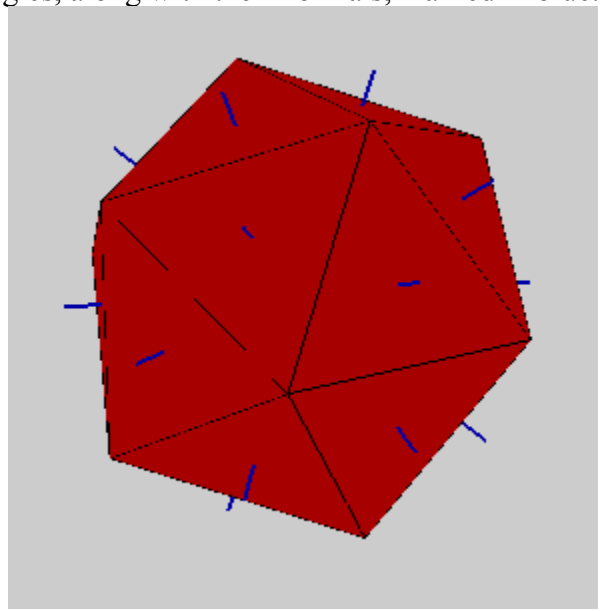
A spot light is a light that is basically a point light in nature, but is not projected in all directions. Spot lights affect objects in a conical section, and so naturally, we need to specify a location, direction, and angle for a spot light before we can use it.



Normals

In our program, we are going to be using diffuse lighting, as well as a little ambient light, and we are going to be using a directional light source. In order to use diffuse light (or even specular light) we need to know what angle our surface is at, compared to the direction of the light rays.

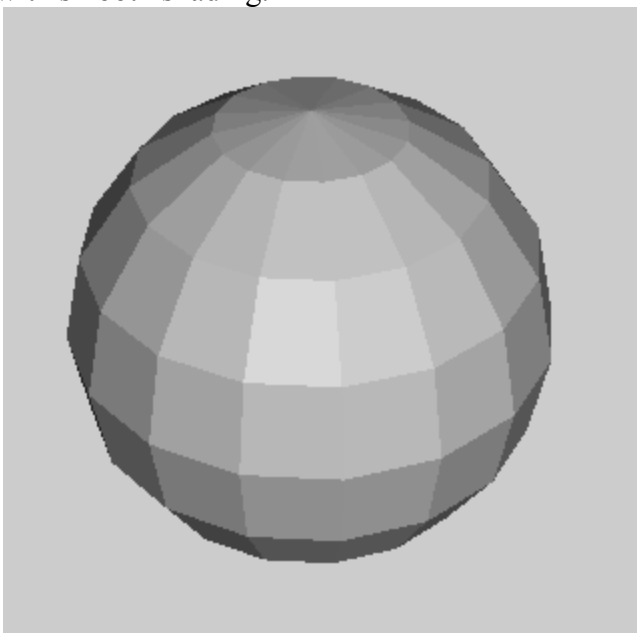
Our effect will do most of this work for us. The only thing we have to specify is what direction our surface is oriented. In graphics, the usual way of doing this is by using normals. A normal is a vector that points exactly perpendicular to a surface. Below are a few triangles, along with their normals, marked in blue.

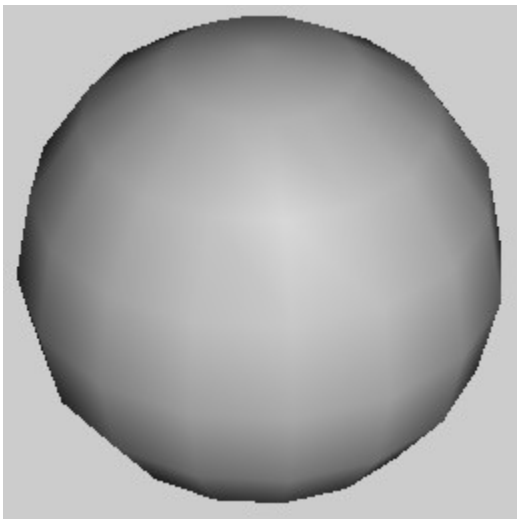


This image was made in a program called Wings3D, which we might get a chance to talk about in a little more detail once we start talking about making models.

In many graphics programs, though, we don't want to specify one normal for each triangle. We would rather specify one for each vertex. This lets us perform a nicer lighting model called smooth shading. We won't go into too much detail about it, other

than to show an example of the differences between smooth shading, and its counterpart, flat shading. Below are two images. The first one is done with flat shading, and the second one is done with smooth shading.





You can see from these images, that most of the time, smooth shading looks a lot nicer, which is why we want to use it. Smooth shading interprets the normals between the vertices, in order to get a smoother appearance.

Adding Normals to our Objects

We now move on to the coding part of this tutorial. Let's first add normals to our icosahedron-asteroid. Go to the `SetupVertexBuffer()` method. The first thing we need to do is to change our data structure from `VertexPositionTexture` to `VertexPositionNormalTexture`. Change the line that says:

```
VertexPositionTexture[] vertexArray = new VertexPositionTexture[12];
```

to say:

```
VertexPositionNormalTexture[] vertexArray = new
                                    VertexPositionNormalTexture[12];
```

Now we will be using vertices that have positions, normals, and texture coordinates. We need to tell our vertex buffer that we will need more space to store all of these, so go to the end of the method and find the line that says:

```
vertexBuffer = new VertexBuffer(device, sizeof(float) * 5 *
                  vertexArray.Length, ResourceUsage.WriteOnly,
                        ResourceManagementMode.Automatic);
```

We want to change the 5 to an 8 so that the line will look like this:

```
vertexBuffer = new VertexBuffer(device, sizeof(float) * 8 *
                  vertexArray.Length, ResourceUsage.WriteOnly,
                        ResourceManagementMode.Automatic);
```

Now we need to define normals for all of our vertices. Rather than having you compute all of the normals, you can just use the code below. Put this code in your SetupVertexBuffer() method somewhere after you declare your array (at the beginning of the method) and before you put the data in the VertexBuffer (towards the end of the method):

```
vertexArray[0].Normal = new Vector3(-0.526f, 0.000f, 0.851f);
vertexArray[1].Normal = new Vector3(0.526f, 0.000f, 0.851f);
vertexArray[2].Normal = new Vector3(-0.526f, 0.000f, -0.851f);
vertexArray[3].Normal = new Vector3(0.526f, 0.000f, -0.851f);
vertexArray[4].Normal = new Vector3(0.000f, 0.851f, 0.526f);
vertexArray[5].Normal = new Vector3(0.000f, 0.851f, -0.526f);
vertexArray[6].Normal = new Vector3(0.000f, -0.851f, 0.526f);
vertexArray[7].Normal = new Vector3(0.000f, -0.851f, -0.526f);
vertexArray[8].Normal = new Vector3(0.851f, 0.526f, 0.000f);
vertexArray[9].Normal = new Vector3(-0.851f, 0.526f, 0.000f);
vertexArray[10].Normal = new Vector3(0.851f, -0.526f, 0.000f);
vertexArray[11].Normal = new Vector3(-0.851f, -0.526f, 0.000f);
```

These numbers aren't arbitrary. They have been calculated very deliberately. There are ways to mathematically calculate what a normal should be, and there are also programs that will calculate normals of objects for you. For now though, this will be good enough.

The final thing we have to do is tell the setup our drawing to recognize that we are using normals. Go down to your `DrawObject()` method so that we can make a few changes. Inside the foreach loop, there are three references that say `VertexPositionTexture`. We want to change these to say `VertexPositionNormalTexture` instead. That way our program will be able to utilize the normals while it is rendering.

Turning on the Lights

We are now ready to turn on our lights. Stay in the `DrawObject()` method, but go just above the foreach loop. Put the following line to turn on lighting:

```
effect.Parameters["xEnableLighting"].SetValue(true);
```

We want to use a directional light, so we need to specify what direction the light is going. We specify this as a vector. This vector has to be normalized before we can give it to the effect. Add the following code to do this:

```
Vector3 lightDirection = new Vector3(1, 1, -1);
lightDirection.Normalize();
            effect.Parameters["xLightDirection"].SetValue(lightDirectio
n);
```

The values for the lightDirection vector can be changed if you want, but these numbers will look fairly good.

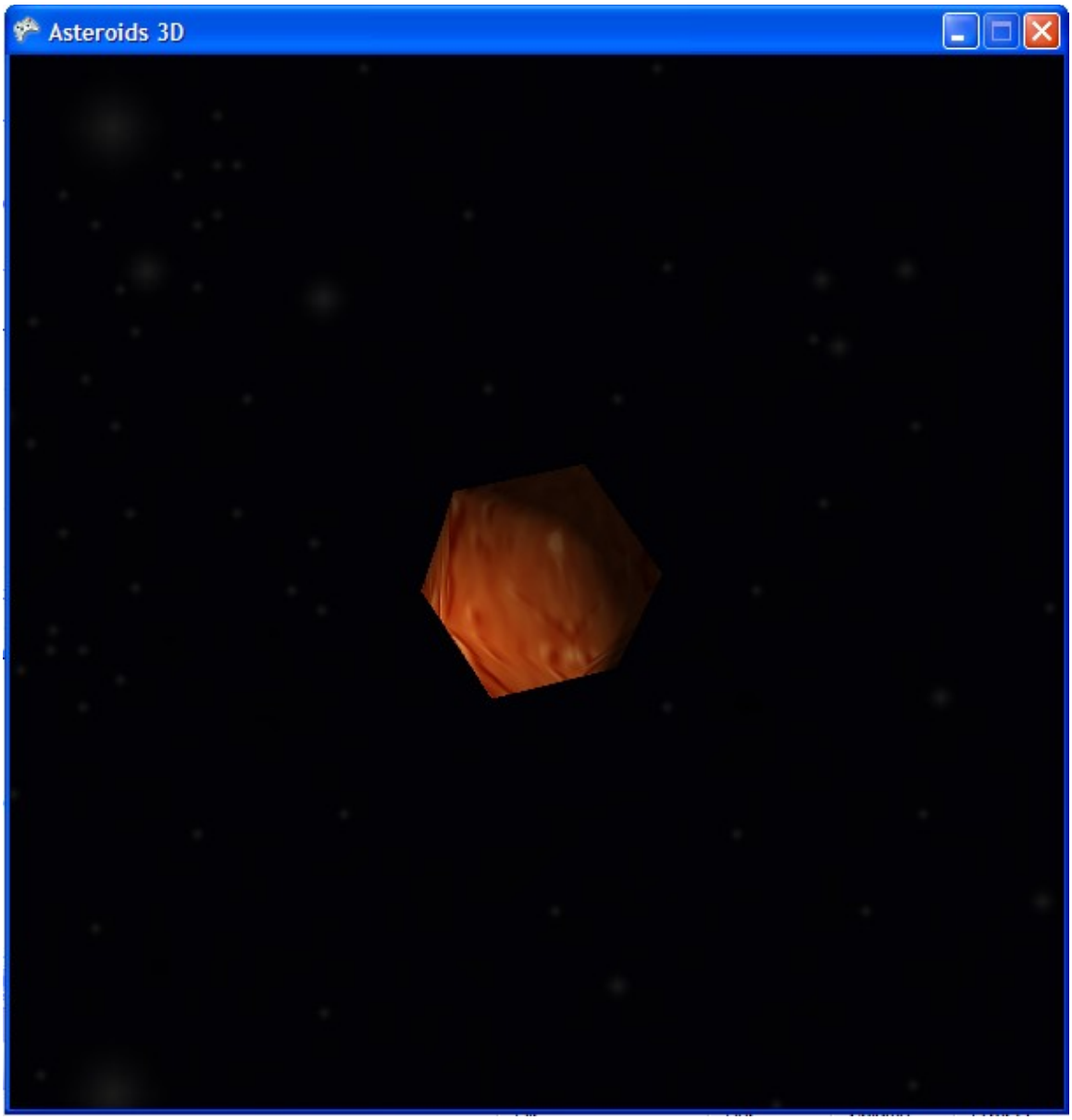
Next we want to turn on a little bit of ambient lighting. Just below the code we just added put the following line:

```
effect.Parameters["xAmbient"].SetValue(.15f);
```

The .15 is a fairly nice amount to put for ambient light. If the number is 0, then there is no ambient light at all, but if it is 1, then it looks like the ambient light is as bright as the directional light, and we don't have a dark side of our object. Feel free to play around with this value and see what it does.

Turning off the Lights

We can now run our program to see what it looks like with lighting. Below is what my program looks like at this point.



It looks like we have a small problem. Our background is having trouble. If you look closely, you can see that it is still there, but for some reason, it has gotten very dark. There's an explanation for that. By "turning on the lights," we didn't flip a light switch. Instead, we told the graphics card "there are lights in our scene, so now you are going to have to calculate them into everything you do." The graphics card has determined that there is not very much light shining on the background, so it should be drawn kind of dark.

Well this could be a problem: we want our asteroids to have a dark side to them, but we want a background that is the right color, which requires no lights. Here is where we can do something that is impossible in the real world. We will have the lights on as we draw our asteroid, but we will turn them off for when we draw the background, and the combination will still be able to be drawn together on the screen.

Let's add one more line to turn the lights back off after we are done rendering our object. Down after the foreach loop, but before we call `effect.end()` let's add the line below:

```
effect.Parameters["xEnableLighting"].SetValue(false);
```

Now our lighting will be off when we get around to drawing the background. Below is what our program should look like now: