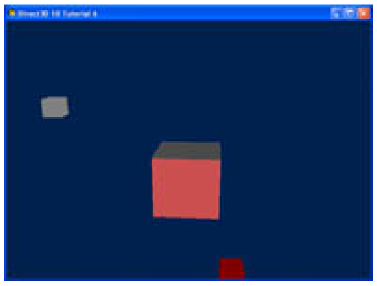
Tutorial 06: Lighting

⊟ Collapse All



Summary

In the previous tutorials, the world looks boring because all the objects are lit in the same way. This tutorial will introduce the concept of simple lighting and how it can be applied. The technique used will be lambertian lighting.

The outcome of this tutorial will modify the previous example to include a light source. This light source will be attached to the cube in orbit. The effects of the light can be seen on the center cube.

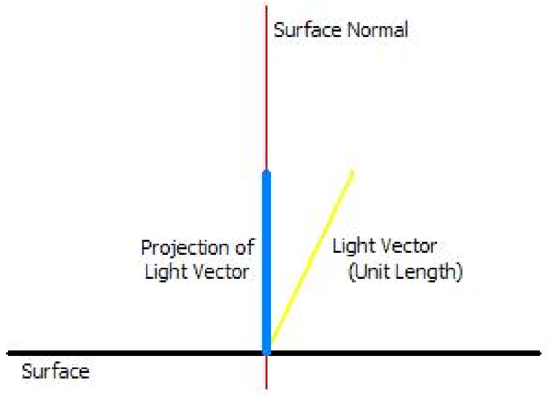
Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial06

Lighting

In this tutorial, the most basic type of lighting will be introduced: lambertian lighting. Lambertian lighting has uniform intensity irrespective of the distance away from the light. When the light hits the surface, the amount of light reflected is calculated by the angle of incidence the light has on the surface. When a light is shined directly on a surface, it is shown to reflect all the light back, with maximum intensity. However, as the angle of the light is increased, the intensity of the light will fade away.

To calculate the intensity that a light has on a surface, the angle between the light direction and the normal of the surface has to be calculated. The normal for a surface is defined as a vector that is perpendicular to the surface. The calculation of the angle can be done with a simple dot product, which will return the projection of the light direction vector onto the normal. The wider the angle, the smaller the projection will be. Thus, this gives us the correct function to modulate the diffused light with.



The light source used in this tutorial is an approximation of directional lighting. The vector which describes the light source determines the direction of the light. Since it's an approximation, no matter where an object is, the direction in which the light shines towards it is the same. An example of this light source is the sun; the sun is always seen to be shining in the same direction for all objects in a scene. In addition, the intensity of the light on individual objects is not taken into consideration.

Other types of light include point lights, which radiate uniform light from their centers, and spot lights, which are directional but not uniform across all objects.

Initializing the Lights

In this tutorial, there will be two light sources. One will be statically placed above and behind the cube, and another one will be orbiting the center cube. Note that the orbiting cube in the previous tutorial has been replaced with this light source.

Since lighting is computed by the shaders, the variables would have to be declared and then bound to the variables within the technique. In this sample, we just require the direction of the light source, as well as its color value. The first light is grey and not moving, while the second one is an orbiting red light.

```
// Setup our lighting parameters
D3DXVECTOR4 vLightDirs[2] =
{
    D3DXVECTOR4( -0.577f, 0.577f, -0.577f, 1.0f ),
    D3DXVECTOR4( 0.0f, 0.0f, -1.0f, 1.0f ),
};
D3DXVECTOR4 vLightColors[2] =
{
    D3DXVECTOR4( 0.5f, 0.5f, 0.5f, 0.0f ),
    D3DXVECTOR4( 0.5f, 0.0f, 0.0f, 0.0f )
};
```

The orbiting light is rotated just like the cube in the last tutorial. The rotation matrix applied will change the direction of the light, to show the effect that it is always shining towards the center. Note that function D3DXVec3Transform is utilized to multiply a matrix with a vector. In the previous tutorial we were multiplying just the transformation matrices into the world matrix, then passed into the shader for transformation; but in this case, we're actually doing the world transform of the light in the CPU, for simplicity's sake.

```
//rotate the second light around the origin
D3DXMATRIX mRotate;
D3DXVECTOR4 vOutDir;
D3DXMatrixRotationY( &mRotate, -2.0f*t );
D3DXVec3Transform( &vLightDirs[1], (D3DXVECTOR3*)&vLightDirs[1], &mRotate );
```

The lights' direction and color are both passed into the shader just like the matrices. The associated variable is called to set, and the parameter is passed in.

```
//
// Update lighting variables
//
g_pLightDirVariable->SetFloatVectorArray( (float*)vLightDirs, 0,    2 );
g_pLightColorVariable->SetFloatVectorArray( (float*)vLightColors, 0,    2 );
```

Rendering the Lights in the Pixel Shader

Once we have all the data setup and the shader properly fed with data, we can compute the lambertian lighting term on each pixel from the light sources. We'll be using the dot product rule discussed above.

Once we've taken the dot product of the light versus the normal, it can then be multiplied with the color of the light to calculate the effect of that light. That value is passed through the saturate function, which converts the range to [0, 1]. Finally, the results from the two separate lights are summed together to create the final pixel color.

Consider that the material of the surface itself is not factored into this light calculation; the final color of the surface is a result of the light's colors.

```
//
// Pixel Shader
//
float4 PS( PS_INPUT input) : SV_Target
{
    float4 finalColor = 0;

    //do NdotL lighting for 2 lights
    for(int i=0; i<2; i++)
    {
        finalColor += saturate( dot( (float3)vLightDir[i],input.Norm) * vLightColor[i] );
    }
    return finalColor;
}
```

Once through the pixel shader, the pixels will have been modulated by the lights and you can see the effect of each light on the cube surface. Note that the light in this case looks flat because pixels on the same surface will have the same normal. Diffuse is a very simple and easy lighting model to compute; more complex lighting models can be used to achieve richer and more realistic materials.