

WORDWARE GAME AND GRAPHICS LIBRARY

Includes chapters
on the
Direct3D High-Level
Shading Language,
the Direct3D
effects framework,
and vertex and
pixel shaders.

Introduction to **3D GAME** ***Programming*** **with** **DirectX[®] 9.0**

Frank D. Luna

Technical Review by Rod Lopez



Introduction to 3D Game Programming with DirectX® 9.0

Frank D. Luna

Technical review by Rod Lopez

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Luna, Frank D.

Introduction to 3D game programming with DirectX 9.0 / by Frank D. Luna.

p. cm.

ISBN-10: 1-55622-913-5 (pbk.)

1. Computer games--Programming. 2. DirectX. I. Title.

QA76.76.C672L83 2003

794.8'15268--dc21

2003005834

CIP

© 2003, Wordware Publishing, Inc.

All Rights Reserved

2320 Los Rios Boulevard
Plano, Texas 75074

No part of this book may be reproduced in any form or by any means
without permission in writing from Wordware Publishing, Inc.

Printed in the United States of America

ISBN-13: 978-1-55622-913-8

ISBN-10: 1-55622-913-5

10 9 8 7 6 5

0305

DirectX is a registered trademark of Microsoft Corporation in the United States and/or other countries.

All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

```
const D3DXCOLOR    YELLOW( D3DCOLOR_XRGB(255, 255,  0) );
const D3DXCOLOR    CYAN(  D3DCOLOR_XRGB( 0, 255, 255) );
const D3DXCOLOR    MAGENTA( D3DCOLOR_XRGB(255,  0, 255) );
}
```

4.2 Vertex Colors

The color of a primitive is determined by the color of the vertices that make it up. Therefore, we must add a color member to our vertex data structure. Note that a `D3DCOLORVALUE` type cannot be used here because Direct3D expects a 32-bit value to describe the color of a vertex. (Actually, by using a vertex shader we could use 4D color vectors for the vertex color, and thereby gain 128-bit color, but that is getting ahead of ourselves for now. Vertex shaders are covered in Chapter 17.)

```
struct ColorVertex
{
    float _x, _y, _z;
    D3DCOLOR _color;
    static const DWORD FVF;
}
const DWORD ColorVertex::FVF = D3DFVF_XYZ | D3DFVF_DIFFUSE;
```

4.3 Shading

Shading occurs during rasterization and specifies how the vertex colors are used to compute the pixel colors that make up the primitive. There are two shading modes that are presently used: flat shading and Gouraud shading.

With flat shading, the pixels of a primitive are uniformly colored by the color specified in the *first* vertex of the primitive. So the triangle formed by the following three vertices would be red, since the first vertex color is red. The colors of the second and third vertices are ignored with flat shading.

```
ColorVertex t[3];
t[0]._color = D3DCOLOR_XRGB(255, 0, 0);
t[1]._color = D3DCOLOR_XRGB(0, 255, 0);
t[2]._color = D3DCOLOR_XRGB(0, 0, 255);
```

Flat shading tends to make objects appear blocky because there is no smooth transition from one color to the next. A much better form of shading is called Gouraud shading (also called smooth shading). With Gouraud shading, the colors at each vertex are interpolated linearly across the face of the primitive. Figure 4.2 shows a red flat shaded triangle and a triangle colored using Gouraud shading.

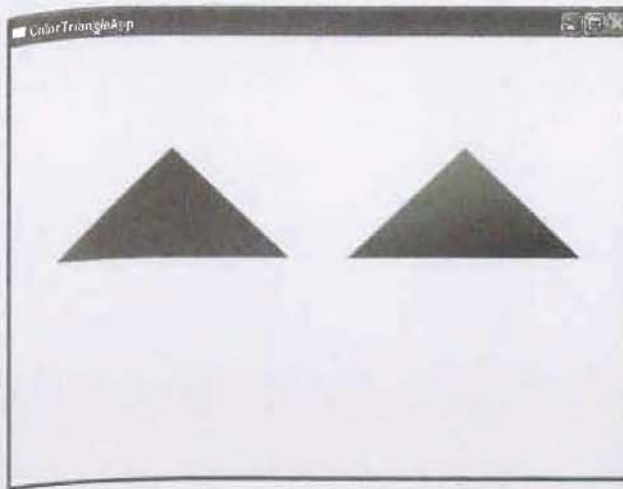


Figure 4.2: On the left is a triangle colored red with flat shading. On the right is a triangle with vertex colors red, green, and blue; notice that with Gouraud shading, the vertex colors are interpolated across the triangle.

Like many things in Direct3D, the shading mode is controlled through the Direct3D state machine.

```
// set flat shading
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);

// set Gouraud shading
Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
```

Part II



4.4 Sample Application: Colored Triangle

The sample program for this chapter demonstrates a triangle colored using flat shading and a triangle colored using Gouraud shading. It renders the image shown in Figure 4.2. First we add the following global variables:

```
D3DXMATRIX          World;
IDirect3DVertexBuffer9* Triangle = 0;
```

We include a `D3DXMATRIX` that is used to store the world transformation of the triangles that we are going to draw. The `Triangle` variable is the vertex buffer that stores the vertex data of a triangle. Notice that we only have to store the geometry of one triangle because we can draw it multiple times at different positions in the world using the world matrix.

The `Setup` method creates the vertex buffer and fills it with the data of a triangle with colored vertices. The first vertex in the triangle is full-intensity red (255), the second is full-intensity green (255), and the third is full-intensity blue (255). Finally, we disable lighting for this sample. Notice that this sample uses the new `ColorVertex` structure, as explained in section 4.2.

```

bool Setup()
{
    // create vertex buffer
    Device->CreateVertexBuffer(
        3 * sizeof(ColorVertex),
        D3DUSAGE_WRITEONLY,
        ColorVertex::FVF,
        D3DPOOL_MANAGED,
        &Triangle,
        0);

    // fill the buffers with the triangle data
    ColorVertex* v;
    Triangle->Lock(0, 0, (void**)&v, 0);

    v[0] = ColorVertex(-1.0f, 0.0f, 2.0f, D3DCOLOR_XRGB(255, 0, 0));
    v[1] = ColorVertex( 0.0f, 1.0f, 2.0f, D3DCOLOR_XRGB(0, 255, 0));
    v[2] = ColorVertex( 1.0f, 0.0f, 2.0f, D3DCOLOR_XRGB(0, 0, 255));

    Triangle->Unlock();

    // set projection matrix
    D3DXMATRIX proj;
    D3DXMatrixPerspectiveFovLH(
        &proj,
        D3DX_PI * 0.5f, // 90 - degree
        (float)Width / (float)Height,
        1.0f,
        1000.0f);
    Device->SetTransform(D3DTS_PROJECTION, &proj);

    // set the render states
    Device->SetRenderState(D3DRS_LIGHTING, false);

    return true;
}

```

Then, the `Display` function draws `Triangle` twice in two different positions and with different shade modes. The position of each triangle is controlled with the world matrix—`World`.

```

bool Display(float timeDelta)
{
    if( Device )
    {
        Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
            0xffffffff, 1.0f, 0);
        Device->BeginScene();

        Device->SetFVF(ColorVertex::FVF);
        Device->SetStreamSource(0, Triangle, 0, sizeof(ColorVertex));
    }
}

```

```

// draw the triangle to the left with flat shading
D3DXMatrixTranslation(&World, -1.25f, 0.0f, 0.0f);
Device->SetTransform(D3DTS_WORLD, &World);

Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

// draw the triangle to the right with gouraud shading
D3DXMatrixTranslation(&World, 1.25f, 0.0f, 0.0f);
Device->SetTransform(D3DTS_WORLD, &World);

Device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
Device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

Device->EndScene();
Device->Present(0, 0, 0, 0);
}
return true;
}

```

4.5 Summary

- Colors are described by specifying an intensity of red, green, and blue. The additive mixing of these three colors at different intensities allows us to describe millions of colors. In Direct3D, we can use the `D3DCOLOR`, the `D3DCOLORVALUE`, or the `D3DXCOLOR` type to describe a color in code.
- We sometimes treat a color as a 4D vector (r, g, b, a) . Color vectors are added, subtracted, and scaled just like regular vectors. On the other hand, dot and cross products do *not* make sense for color vectors, but component-wise multiplication does make sense for colors. The symbol $\otimes$ denotes component-wise multiplication, and it is defined as: $(c_1, c_2, c_3, c_4) \otimes (k_1, k_2, k_3, k_4) = (c_1k_1, c_2k_2, c_3k_3, c_4k_4)$.
- We specify the color of each vertex, and then Direct3D uses the current shade mode to determine how these vertex colors are used to compute the pixel colors of the triangle during rasterization.
- With flat shading, the pixels of a primitive are uniformly colored by the color specified in the *first* vertex of the primitive. With Gouraud shading, the colors at each vertex are interpolated linearly across the face of the primitive.



Once a light is registered, we can turn it on and off using what this next example illustrates:

```
Device->LightEnable(
    0, // the element in the light list to enable/disable
    true); // true = enable, false = disable
```

5.5 Sample Application: Lighting

The sample for this chapter creates the scene shown in Figure 5.7. It demonstrates how to specify vertex normals, how to create a material, and how to create and activate a directional light. Note that in this sample program we do not make use of the `d3dUtility.h/cpp` material and light functionality code because we want to show how it is done manually first. However, the rest of the samples in this book do use the material and light utility code.

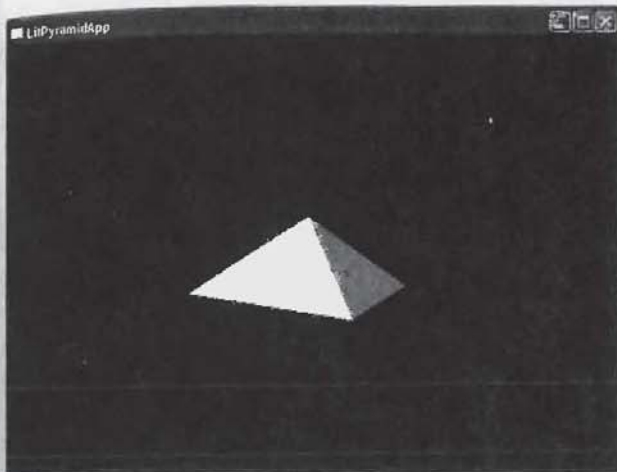


Figure 5.7: Screen shot from the LitPyramid sample

The steps for adding light to a scene are:

1. Enable lighting.
2. Create a material for each object and set the material before rendering the corresponding object.
3. Create one or more light sources, set the light sources, and enable them.
4. Enable any additional lighting states, such as specular highlights.

First we instantiate a global vertex buffer that stores the pyramid's vertices:

```
IDirect3DVertexBuffer9* Pyramid = 0;
```

The `Setup` function contains all the code relevant to this chapter, so we omit the other functions to save space. It implements the previously discussed steps to add lighting to a scene. The `Setup` method starts by



enabling lighting, which isn't necessary because it's enabled by default (but it doesn't hurt either).

```
bool Setup()
{
    Device->SetRenderState(D3DRS_LIGHTING, true);
}
```

Next, we create the vertex buffer, lock it, and specify the vertices that form triangles of the pyramid. The vertex normals were precomputed using the algorithm covered in section 5.3. Notice that while the triangles share vertices, they do not share normals; thus it is not very advantageous to use an index list for this object. For example, all the triangles share the peak point (0, 1, 0); however, for each triangle, the peak vertex normal points in a different direction.

```
Device->CreateVertexBuffer(
    12 * sizeof(Vertex),
    D3DUSAGE_WRITEONLY,
    Vertex::FVF,
    D3DPOOL_MANAGED,
    &Pyramid,
    0);

// fill the vertex buffer with pyramid data
Vertex* v;
Pyramid->Lock(0, 0, (void**)&v, 0);

// front face
v[0] = Vertex(-1.0f, 0.0f, -1.0f, 0.0f, 0.707f, -0.707f);
v[1] = Vertex( 0.0f, 1.0f,  0.0f, 0.0f, 0.707f, -0.707f);
v[2] = Vertex( 1.0f, 0.0f, -1.0f, 0.0f, 0.707f, -0.707f);

// left face
v[3] = Vertex(-1.0f, 0.0f,  1.0f, -0.707f, 0.707f, 0.0f);
v[4] = Vertex( 0.0f, 1.0f,  0.0f, -0.707f, 0.707f, 0.0f);
v[5] = Vertex(-1.0f, 0.0f, -1.0f, -0.707f, 0.707f, 0.0f);

// right face
v[6] = Vertex( 1.0f, 0.0f, -1.0f, 0.707f, 0.707f, 0.0f);
v[7] = Vertex( 0.0f, 1.0f,  0.0f, 0.707f, 0.707f, 0.0f);
v[8] = Vertex( 1.0f, 0.0f,  1.0f, 0.707f, 0.707f, 0.0f);

// back face
v[9]  = Vertex( 1.0f, 0.0f,  1.0f, 0.0f, 0.707f, 0.707f);
v[10] = Vertex( 0.0f, 1.0f,  0.0f, 0.0f, 0.707f, 0.707f);
v[11] = Vertex(-1.0f, 0.0f,  1.0f, 0.0f, 0.707f, 0.707f);

Pyramid->Unlock();
```

After we have generated the vertex data of our object, we describe how the object interacts with light by describing its materials. In this sample, the pyramid reflects white lights, emits no light, and produces some highlights.

```

D3DMATERIAL9 mtrl;
    mtrl.Ambient = d3d::WHITE;
    mtrl.Diffuse = d3d::WHITE;
    mtrl.Specular = d3d::WHITE;
    mtrl.Emissive = d3d::BLACK;
    mtrl.Power = 5.0f;

    Device->SetMaterial(&mtrl);

```

Second to last, we create and enable a directional light. The directional light rays run parallel to the x-axis in the positive direction. The light emits strong white diffuse light (`dir.Diffuse = WHITE`), weak white specular light (`dir.Specular = WHITE * 0.3f`), and a medium amount of white ambient light (`dir.Ambient = WHITE * 0.6f`).

```

D3DLIGHT9 dir;
    ::ZeroMemory(&dir, sizeof(dir));
    dir.Type = D3DLIGHT_DIRECTIONAL;
    dir.Diffuse = d3d::WHITE;
    dir.Specular = d3d::WHITE * 0.3f;
    dir.Ambient = d3d::WHITE * 0.6f;
    dir.Direction = D3DXVECTOR3(1.0f, 0.0f, 0.0f);

    Device->SetLight(0, &dir);
    Device->LightEnable(0, true);

```

Finally, we set the state to renormalize normals and enable specular highlights.

```

Device->SetRenderState(D3DRS_NORMALIZENORMALS, true);
Device->SetRenderState(D3DRS_SPECULARENABLE, true);

// ... code to set up the view matrix and projection matrix
// omitted

return true;
}

```

5.6 Additional Samples

Three additional samples are included for this chapter in the companion files. They use the `D3DXCreate*` functions to create the 3D objects that compose the scene. The `D3DXCreate*` functions create vertex data with the format `D3DFVF_XYZ | D3DFVF_NORMAL`. In addition, these functions compute the vertex normals of each mesh for us. The additional samples demonstrate how to use directional lights, point lights, and spotlights. Figure 5.8 shows a screen shot from the directional light sample.



Example:

```
Device->SetTexture(0, _stonewall);
```



Note: In Direct3D, you can set up to eight textures that can be combined to create a more detailed image. This is called multitexturing. We do not use multitexturing in this book until Part IV; therefore we always set the texture's stage to 0, for now.

To disable a texture at a particular texturing stage, set `pTexture` to 0. For instance, if we don't want to render an object with a texture, we would write:

```
Device->SetTexture(0, 0);
renderObjectWithoutTexture();
```

If our scene has triangles that use different textures, we would have to do something similar to the following code:

```
Device->SetTexture(0, _tex0);
drawTrisUsingTex0();

Device->SetTexture(0, _tex1);
drawTrisUsingTex1();
```

6.3 Filters

As mentioned previously, textures are mapped to triangles in screen space. Usually, the texture triangle is not the same size as the screen triangle. When the texture triangle is smaller than the screen triangle, the texture triangle is *magnified* to fit. When the texture triangle is larger than the screen triangle, the texture triangle is *minified* to fit. In both cases, distortion will occur. *Filtering* is a technique Direct3D uses to help smooth out these distortions.

Direct3D provides three different types of filters; each one provides a different level of quality. The better the quality, the slower it is, so you must make the trade-off between quality and speed. Texture filters are set with the `IDirect3DDevice9::SetSamplerState` method.

- **Nearest point sampling**—This is the default filtering method and produces the worst-looking results, but it is also the fastest to compute. The following code sets nearest point sampling as the minification and magnification filter:

```
Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_POINT);
Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_POINT);
```

- **Linear filtering**—This type of filtering produces fairly good results and can be done very fast on today's hardware. It is