Chapter 8

Lighting and Materials

"Where the light is brightest the shadows are deepest."
J. W. Goethe: *Goetz von Berlichingen*, II, 1771

8.1 Overview

So far we've described vertex processing in terms of what happens to the shape of the model as described by its vertex position and normal. With lighting and materials we start our examination of how the appearance of a vertex is modified during vertex processing.

Lighting is the process of determining colors for the vertices of a model for use during rasterization. Shading is performed by the rasterizer and defines how the vertex colors will be used to color the interior of a primitive. Lighting uses a definition of the light in the scene, the material properties of the object represented by the vertices, and a model of how materials reflect light to compute the colors for each vertex.

It is important to understand that the light is computed only at the vertices of an object; the coarser the representation of an object in terms of its vertices, then the coarser the computed lighting will be for that object. A common mistake is to create a very large cube and position a light very close to the center of one of its faces. With lighting sampled only at the vertices, the vertices are far away from the light and receive little or no color due to the lights, even though the lights are "close" to the face of the cube. The solution is to create more vertices for the face of the cube to increase the sampling density of the lighting in the scene, or use an alternative method of lighting the scene, such as lightmaps using texturing or per-pixel lighting using pixel shaders.

Lighting is a complex process and involves many render states and other parameters. Light emanates from a source and is reflected by surfaces until it reaches the viewer. Lighting in Direct3D is computed locally. That is, the light for each vertex is computed independently of all the other vertices in the scene. This means that Direct3D does not directly compute shadows, because each vertex is lit as if it existed independently of any other surfaces that might occlude light traveling from the source to the vertex. Similarly, Direct3D does not directly compute the light reflected from one surface by a second surface. Rendering methods such as raytracing and radiosity are called global illumination techniques because they account for the light in a scene that is reflected indirectly from a surface to the viewer, shadows, internal reflections, and other global lighting effects as well as light that is directly reflected to the viewer.

Historically, lighting in computer graphics has been an approximation to the physics of light as the calculations necessary for physically-based lighting were too expensive for real-time interactive applications. While this situation has changed somewhat in recent years, most applications still use the same approximations, or variations thereof, for computing the light reflected to a viewer from objects within a scene.

In computer graphics, we usually deal with three distinct types of reflected light: ambient, diffuse and specular. Ambient light is reflected in all directions within a scene, comes from no particular direction and illuminates all objects within the scene equally. Diffuse reflection is dependent upon the surface normal and the direction of the light, but is independent of the viewing direction. Specular reflection is dependent upon the surface normal, the light direction and the viewing direction.

Before delving into the mathematics of Direct3D lighting, we will show how you can provide vertices with lighting of your own choosing to the rasterizer, so-called "lit vertices". Next, we will examine how to define the sources of light within a scene. Next, we will examine the material properties defined by Direct3D and how they interact with the light in the scene. Finally, we will look at how the colors for each vertex are computed from the light definitions and material properties. The chapter concludes with a sample program that lets you interactively explore all the device state associated with lighting.

8.2 Transformed Vertices

In section 6.2, we mentioned that transformed vertices – those with position components specified as `D3DFVF_XYZRHW` – skip all vertex processing and are passed directly to the rasterizer. Since lighting is part of vertex processing, how are transformed vertices colored? The answer is that vertices can provide their own diffuse and specular color components. These are the colors that would be computed by the lighting portion of vertex processing. Each of these colors will be passed to the rasterizer for pixel processing.

By providing your own diffuse and specular colors for each vertex, the application is said to be performing its own lighting calculations and providing "lit" vertices. If your application has special lighting calculations, you can perform them yourself and include the reflected diffuse and specular colors directly in this manner. Lit vertices are also useful when your only intention is to draw primitives of a specific color or gradient without the necessary computations of lighting. A gradient effect can be obtained by providing different colors for each vertex within a primitive as the rasterizer will interpolate the provided colors across the primitive. When these colors are omitted from the vertex, Direct3D uses opaque white[1] for the diffuse color and transparent black[2] for the specular color as demonstrated by the minimal Direct3D application in section 2.2.

8.3 Lighting Calculations

An overview of the control flow for computing the diffuse and specular lighting used by the rasterizer is shown in figure 8.1. Lighting calculations are enabled or disabled with RS Lighting. When this render state is `FALSE`, no lighting computations are performed and the diffuse and specular lighting come from their respective vertex components. If the vertex has no diffuse color component, opaque white is substituted, while transparent black is substituted for a missing specular color component.

When RS Lighting is `TRUE`, Direct3D computes the amount of reflected light at each vertex. The total reflectance of a vertex is computed from the current material properties, the set of enabled lights, and the position, surface normal, diffuse and specular color components of the vertex. While all vertices have a position component, lighting requires that vertices contain a surface normal component. The diffuse and specular color components are optional and supplied from the device's current material if necessary.

If your model does not contain surface normals, but you want to use lighting, you will need to compute the surface normals. The best way is to compute surface normals directly from whatever process created the vertex position data, such as an implicit surface model. Usually you can obtain the surface normal directly from the same process that created the vertex positions. `D3DXComputeNormals` will compute the vertex normals for a mesh object, see chapter 19.

To compute vertex surface normals from an independent triangle, you can take the cross product of two vectors constructed from the sides of the triangle. This is the "face normal" for the triangle, the vector perpendicular to the plane containing the triangle pointing towards the "outside" of the surface. Using the face normal for each vertex in a triangle gives a faceted appearrance to objects. For a triangle mesh approximating a smooth surface, you can take the average of the face normals of all the triangles containing the vertex, see figure 8.2.

If a faceted appearance is desired for a triangle mesh, then common edges between triangles in the mesh will usually have differing vertex normals. This implies that these vertices cannot be shared between primitives because vertices can only be shared in their entirety, you cannot share positions while providing different surface normals. A cube is the pathological example that provides no vertex sharing at all. Normally this is not much of a problem as most models are describing smooth surfaces and adjacent faces will share the same vertex normal at common vertices.

---

[1] `D3DCOLOR_ARGB(255, 255, 255, 255)`
[2] `D3DCOLOR_ARGB(0, 0, 0, 0)`

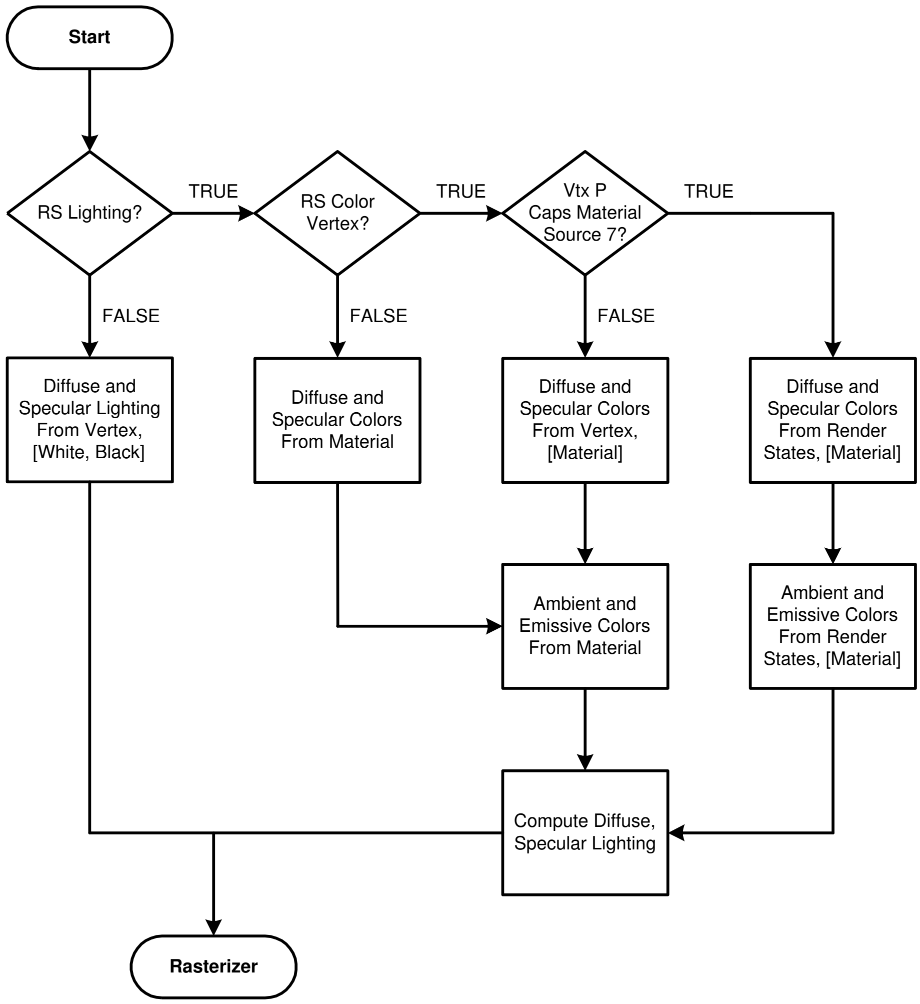

Figure 8.1: Material properties are selected from the default values, the material, or the vertex based on the value of the lighting-related render states. Material source selection requires device support for `D3DVTXPCAPS_MATERIALSOURCE7`. The notation '[Source]' shows the source for the default values of the vertex diffuse and specular color components when absent.

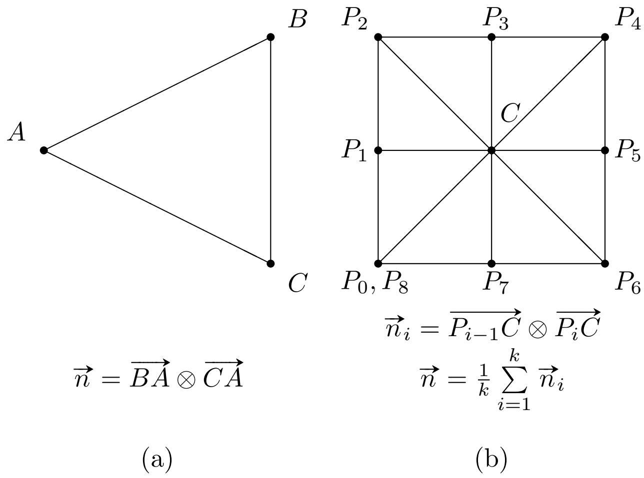

Figure 8.2: Computing vertex surface normals for triangles. (a) the face normal $\vec{n}$ of an isolated triangle is computed. (b) the surface normal $\vec{n}$ for vertex C in a triangle mesh of k triangles approximating a smooth surface. The normal is computed as the average of the face normals for the triangles containing C.

For proper lighting, vertex normals should be unit vectors. However, we saw in chapter 6 that scale transformations will modify the length of the surface normal component. You can deal with this situation in one of several ways: you could not use any scale transformations in your scene, you could compensate in the vertex data by storing the normal vector scaled the inverse of the scaling present in the matrices, or by renormalizing the normals before lighting is performed.

In fixed-function lighting, RS Normalize Normals controls the renormalization of normals and is the easiest way to deal with the scaling issue. When this render state is `TRUE`, normal vectors are scaled to be unit vectors before being used in lighting.

$$\vec{n} \leftarrow \frac{\vec{n}}{\|\vec{n}\|}$$

However, renormalizing the normal vectors before using them in lighting does have a cost, so you might prefer to model everything in your scene to a consistent scale if you want to avoid the cost of renormalization.

Direct3D computes the light using only the local information contained at a single vertex and does not compute shadows or other global illumination effects. The light is modeled as a sum of different types of reflected light. Each of these lighting terms depends on both the reflectance properties of the vertex material and the color of the light source.

$$\mathit{Light} = \mathit{Ambient} + \mathit{Diffuse} + \mathit{Specular} + \mathit{Emissive}$$

Render State	Default
RS Ambient Material Source	D3DMCS_MATERIAL
RS Diffuse Material Source	D3DMCS_COLOR1
RS Specular Material Source	D3DMCS_COLOR2
RS Emissive Material Source	D3DMCS_MATERIAL

D3DMATERIAL9 Member	Default
Ambient	$\langle 0, 0, 0, 0 \rangle$
Diffuse	$\langle 1, 1, 1, 1 \rangle$
Specular	$\langle 0, 0, 0, 0 \rangle$
Emissive	$\langle 0, 0, 0, 0 \rangle$
Power	1

Table 8.1: Default values for the material parameters. The default values are shown for the material source render states and the current material.

Ambient Ambient light models the indirect light present in a scene and can be thought of as the "background light". Ambient light shines in all directions and adds an overall level of brightness to objects within a scene.

Diffuse Rough surfaces scatter incident light resulting in a diffuse reflection. Direct3D computes diffuse reflection with the Lambertian cosine law, giving rough surfaces an appearance like chalk or matte finishes. The intensity of the reflection is dependent upon the relative orientation of the surface normal and the direction of the light, but is independent of the viewing direction.

Specular Smooth surfaces reflect incident light in a mirror-like fashion resulting in specular reflection. Direct3D computes specular reflection with a Blinn-Phong lighting model, giving smooth surfaces an appearance like polished metal, glass or plastic. The intensity of the reflection is dependont upon the relative orientation of the surface normal, the direction of the light and the viewing direction.

Emissive This type of "reflection" is not really a reflection at all – it is light emitted by the surface itself and thus part of its total reflectance. Although it is an "emission" of light, it does not count as a light source to Direct3D. For instance, an area light source might be modelled explicitly in the scene with geometry with purely an emissive refelction component.

8.4 Surface Material Properties

The reflectance properties of a surface can be set through a combination of device state and per-vertex data. These properties define how the surface responds to ambient, diffuse and specular light sources in the scene. The `GetMaterial` and `SetMaterial` methods are used to manipulate the current material properties, which are defined by the `D3DMATERIAL9` structure.

```
HRESULT GetMaterial(D3DMATERIAL9 *value);
HRESULT SetMaterial(const D3DMATERIAL9 *value);

typedef struct _D3DMATERIAL9
{
    D3DCOLORVALUE Diffuse;
    D3DCOLORVALUE Ambient;
    D3DCOLORVALUE Specular;
    D3DCOLORVALUE Emissive;
    float         Power;
} D3DMATERIAL9;
```

The `Diffuse`, `Ambient`, `Specular`, and `Emissive` members of the structure define the reflectance properties of the material. Each member contains an `D3DCOLORVALUE` structure representing an RGBA color as 4 floating-point numbers. The numbers represent the fraction of the light source reflected by the material. For instance, a `Diffuse` value of $\{0.5, 0.5, 0.5, 0.5\}$ represents a material that reflects 50% of the diffuse light in the scene. Unless an object is a source of light, its reflectance coefficients for all light in the scene should sum to 1, conforming to the law of conservation of energy[3]. The `Power` member is a single float that characterizes the spread of the specular highlight on a surface. Smooth surfaces, such as a billiard ball, will have a shiny specular highlight. Rough surface will spread out and scatter a specular highlight. Rough surfaces correspond to lower values of the `Power` member and smooth surfaces correspond to higher values.

RS Color Vertex determines the source of the material properties. When RS Color Vertex is `FALSE`, the material properties are taken from the current material. When RS Color Vertex is `TRUE`, the material properties for the vertex are chosen based on the value of RS Ambient Material Source, RS Diffuse Material Source, RS Specular Material Source, and RS Emissive Material Source. Each of these material source render states is of type `D3DMATERIALCOLORSOURCE`. `D3DMCS_MATERIAL` selects the material property from the material set on the device. `D3DMCS_COLOR1` and `D3DMCS_COLOR2` select the material property from the vertex diffuse and specular color component, respectively.

```
typedef enum _D3DMATERIALCOLORSOURCE
{
    D3DMCS_MATERIAL = 0,
    D3DMCS_COLOR1   = 1,
    D3DMCS_COLOR2   = 2
} D3DMATERIALCOLORSOURCE;
```

[3]If the coefficients summed to more than 1, then the material would reflect more light than is incident upon its surface.

If the D3DVTXPCAPS_MATERIALSOURCE7 bit of D3DCAPS9::VertexProcessingCaps is set, the device supports the material source render states. If this bit is not set, only the default values in table 8.1 are supported for the material source render states.

```
#define D3DVTXPCAPS_MATERIALSOURCE7 0x00000002L
```

8.5 Light Sources

The ambient light in Direct3D's lighting model accounts for all the indirect lighting effects that are not represented explicitly in the model. Without ambient light, any parts of the scene outside the influence of the enabled lights would be completely black. Setting the ambient light for a scene allows these objects to be seen. RS Ambient defines the intensity of the ambient light in the scene as a D3DCOLOR value.

For other light sources, Direct3D maintains an array of D3DLIGHT9 structures, each defining a light source. The application can change any element of the structure array, and can enable or disable any element within the array while rendering a scene. The array maintained by the device will always be large enough to hold the largest index manipulated by the application. Therefore, it makes sense for an application to use light array indices starting from 0, increasing as more lights are required.

The elements of the array are manipulated with the GetLight and SetLight methods. Default values are returned if an element of the array is fetched that has not yet been set by the application. While rendering, the application enables lights that influence the geometry being rendered. Each light can be enabled or disabled with LightEnable. GetLightEnable returns the enabled status of any light.

```
HRESULT GetLight(DWORD index, D3DLIGHT9 *value);
HRESULT SetLight(DWORD index, const D3DLIGHT9 *value);
HRESULT GetLightEnable(DWORD index, BOOL *value);
HRESULT LightEnable(DWORD index, BOOL value);

typedef struct _D3DLIGHT9 {
    D3DLIGHTTYPE  Type;
    D3DCOLORVALUE Diffuse;
    D3DCOLORVALUE Specular;
    D3DCOLORVALUE Ambient;
    D3DVECTOR     Position;
    D3DVECTOR     Direction;
    float         Range;
    float         Falloff;
    float         Attenuation0;
    float         Attenuation1;
    float         Attenuation2;
```

Member	Directional	Point	Spot
Type	Yes	Yes	Yes
Ambient	Yes	Yes	Yes
Diffuse	Yes	Yes	Yes
Specular	Yes	Yes	Yes
Position		Yes	Yes
Direction	Yes		Yes
Range		Yes	Yes
Falloff			Yes
Attenuation0		Yes	Yes
Attenuation1		Yes	Yes
Attenuation2		Yes	Yes
Theta			Yes
Phi			Yes

Table 8.2: The `D3DLIGHT9` members used by each light type.

```
    float         Theta;
    float         Phi;
} D3DLIGHT9;

typedef enum _D3DLIGHTTYPE {
    D3DLIGHT_DIRECTIONAL = 3,
    D3DLIGHT_POINT       = 1,
    D3DLIGHT_SPOT        = 2
} D3DLIGHTTYPE;
```

Direct3D supports three different types of light sources: directional, point, and spot. The `Type` member specifies the type of the light source described by the structure. Not all of the remaining members in the structure apply to all light types. The `Type`, `Ambient`, `Diffuse`, and `Specular` members are valid for all light types. The members used by different light types are summarized in table 8.2. The `Position`, `Direction`, and `Range` members are given in world coordinates.

An application can define as many lights as desired, with the only practical limit being system memory. However, hardware vertex processing supports a finite number of lights, with 8 being a typical number. `D3DCAPS9::MaxActiveLights` gives the number of lights that may be active while rendering. The most common strategy for dealing with a limited number of lights is to enable the lights that contribute most to the lighting of the primitives. If the `D3DVTXPCAPS_DIRECTIONALLIGHTS` bit of `VertexProcessingCaps` is set, the device supports directional lights. If the `D3DVTXPCAPS_POSITIONALLIGHTS` bit of `VertexProcessingCaps` is set, the device supports point and spot lights.

```
#define D3DVTXPCAPS_DIRECTIONALLIGHTS 0x00000008L
#define D3DVTXPCAPS_POSITIONALLIGHTS  0x00000010L
```

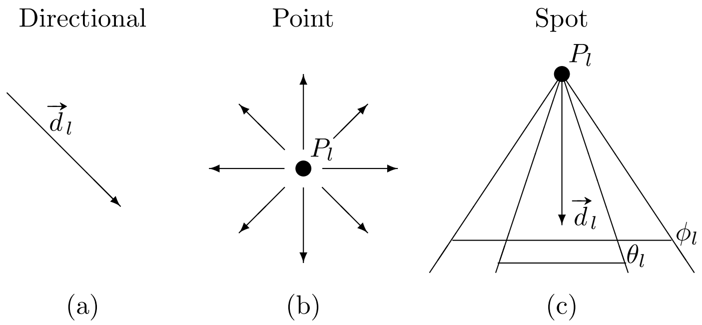

Figure 8.3: Diagram of the geometry of light source types, showing the relevant lighting parameters. The members corresponding to the symbols shown are given in table 8.4.

8.5.1 Directional Lights

Like all lights, a directional light $\mathcal{L}_d$ specifies its ambient color A_l, diffuse color D_l, and specular color S_l. A directional light specifies the direction $\vec{d}_l$ in which the light is shining in the `Direction` member, as in figure 8.3(a).

$$\mathcal{L}_d = \{A_l, D_l, S_l, \vec{d}_l\}$$

For example, the Sun is so distant from the Earth, that its rays appear parallel to each other when they reach the Earth. This makes the Sun essentially a directional light, with the direction vector pointing from the Sun to the Earth.

8.5.2 Point Lights

A point light source $\mathcal{L}_p$ emits light in all directions, originating in a point P_l, as in figure 8.3(b).

$$\mathcal{L}_p = \{A_l, D_l, S_l, P_l, r_l, a_{0l}, a_{1l}, a_{2l}\}$$

When a point light source is very far away from the view volume, it can be approximated by a directional light with a direction vector from the light to the center of the volume. When a point light source is close to the view volume, different portions of the view volume will receive light at a different angle from the light source.

Point lights operate within a limited range, given by the `Range` member. The intensity is attenuated by distance based on the `Attenuation0`, `Attenuation1`, and `Attenuation2` members.

8.5.3 Spot Lights

A spot light source $\mathcal{L}_s$ emits a cone of light from a location in a particular direction, as in figure 8.3(c).

Effect	Attenuation	a_{0l}	a_{1l}	a_{2l}
No Attenuation	1	1	0	0
Linear Attenuation	$1/d$	0	1	0
Quadratic Attenuation	$1/d^2$	0	0	1

Table 8.3: Light attenuation coefficient examples.

$$\mathcal{L}_s = \{A_l, D_l, S_l, P_l, \vec{d}_l, r_l, f_l, a_{0l}, a_{1l}, a_{2l}, \theta_l, \phi_l\}$$

The cone shape and intensity profile are given by the angles θ_l and ϕ_l, and the fallof parameter f_l.

8.5.4 Light Attenuation

Point and spot light sources are attenuated with distance. The attenuation is computed as a modulating factor a_l in the interval $[0, 1]$. Directional lights are not attenuated, giving $a_l = 1$. Point lights are attenuated only by their range from the vertex, giving $a_l = a_r$. Spot lights are attenuated by distance and by the falloff within their light cone, giving $a_l = a_r a_s$.

The range attenuation a_r is computed from the distance d between the vertex and the light source. When d is larger the range parameter of the light r_l, the attenuation is zero. When the distance is inside the light's range, the light's range attenuation is computed from the three attenuation coefficients a_{0l}, a_{1l}, and a_{2l}.

$$\begin{aligned} d &= \|P_v - P_l\| \\ a_r &= \begin{cases} 0, & d > r_l \\ 1/(a_{0l} + a_{1l}d + a_{2l}d^2), & d \leq r_l \end{cases} \end{aligned}$$

The coefficients are drawn from the interval $[0, \infty)$, but at least one of them must be non-zero, or an error will occur when calling `SetLight`. To obtain a point or spot light that does not attenuate inside the range r_l, set $a_{0l} = 1$, $a_{1l} = 0$, and $a_{2l} = 0$ as shown in table 8.3.

The light cone attenuation for spotlights is determined from the inner cone angles θ_l, the outer cone angle ϕ_l and the falloff parameter f_l. When $f_l = 1$, the effect is to linearly ramp the intensity based on the angle of the vertex within the light cone, with the $a_l = 1$ when the angle is θ_l and $a_l = 0$ when the angle is ϕ_l. When $f_l > 1$, the intensity drops more rapidly between the inner and outer cone angles. When $f_l < 1$, the intensity drops off more slowly.

$$\vec{s} = 0 - P'_v$$

D3DLIGHT9 Member	Symbol	D3DMATERIAL9 Member	Symbol
`Ambient`	A_l	`Ambient`	A_m
`Diffuse`	D_l	`Diffuse`	D_m
`Emissive`	E_m	`Specular`	S_m
`Specular`	S_l	`Power`	p_m
`Position`	P_l		
`Direction`	$\vec{d}_l$	Render State	Symbol
`Range`	r_l	RS Ambient	A_{rs}
`Falloff`	f_l		
`Attenuation0`	a_{0l}		
`Attenuation1`	a_{1l}		
`Attenuation2`	a_{2l}		
`Theta`	θ_l		
`Phi`	ϕ_l		

Table 8.4: Symbols used in the lighting equations for render states and members of the `D3DLIGHT9` and `D3DMATERIAL9` structures.

$$\begin{aligned}
\rho_v &= |\vec{d}_l| \cdot |\vec{s}| \\
\rho_\theta &= \cos(\theta_l/2) \\
\rho_\phi &= \cos(\phi_l/2) \\
a_s &= \begin{cases} 0, & \rho_v \le \rho_\phi \\ \left[\frac{\rho_v - \rho_\phi}{\rho_\theta - \rho_\phi}\right]^{f_l}, & \rho_\phi < \rho_v \le \rho_\theta \\ 1, & \rho_\theta < \rho_v \end{cases}
\end{aligned}$$

As lighting, and therefore the spot light attenuation, is calculated only at the vertices, the vertices must be quite dense within the cone of a spot light for the falloff effect to be visible. For this reason, the falloff parameter f_l is often set to 1, which allows the spot light attenuation to be calculated more quickly, avoiding a costly exponeniation.

8.6 The Illumination Model

The illumination model is a mathematical abstraction of how light interacts with matter. While it is based on the physics of matter, it also simplifies the situation to reduce the computational effort required. These simplifications can cause errors and shortcomings in realism of the renderings produced with the model. We've already mentioned that Direct3D simplifies lighting to a local illumination model. This leads to the shortcoming that Direct3D cannot directly render shadows of objects within the scene. If your application demands more than what the Direct3D fixed-function illumination model can provide, you can use more advanced features such as texturing and shader programs to implement a more appropriate lighting model for your scenes.

Direct3D uses its illumination model to compute the diffuse and specular lighting that will be passed to the rasterizer. The diffuse lighting will contain the *Ambient*, *Diffuse*, and *Emissive* terms from the total illumination reflected at a vertex. The specular lighting will contain the *Specular* term if RS Specular Enable is `TRUE`. If RS Specular Enable is `FALSE`, then no specular colors will be computed during lighting and the rasterizer will not use the specular color.

8.6.1 Transforming to Camera Space

Lighting is conceptually performed in camera space[4]. We discussed the transformation of vertices and surface normals in chapter 6. Light positions and direction vectors are specified in world coordinates, so they must also be transformed into camera space by the view transformation matrix $\mathbf{V}$ before the lighting can be calculated. For directional lights, the light direction $\vec{d}_l$ is taken directly from its light definition. For point and spot lights, the direction of the light must be computed for each vertex as the difference between the position of the vertex and the position of the light.

$$\begin{aligned} P'_l &\leftarrow P_l\mathbf{V} \\ \vec{d}_l &\leftarrow \begin{cases} -\frac{\vec{d}_l\mathbf{V}}{\|\vec{d}_l\mathbf{V}\|}, & \text{directional lights} \\ \frac{P'_l - P'_v}{\|P'_l - P'_v\|}, & \text{point and spot lights} \end{cases} \end{aligned}$$

8.6.2 Ambient Reflection

The total ambient reflection for a vertex, *Ambient*, is the total ambient light present in the scene modulated by the material's ambient color. The total ambient light in the scene is the sum of RS Ambient and the ambient light from each enabled light, attenuated by distance as appropriate for the light. Ambient light is not dependent on position or direction, so its intensity is constant regardless of viewing direction.

$$\mathit{Ambient} = A_v(A_{rs} + \sum_l a_l A_l)$$

8.6.3 Diffuse Reflection

The total diffuse reflection for a vertex is the total diffuse light incident on the vertex, modulated by the material's diffuse color and a Lambertian intensity law, as shown in figure 8.4. A Lambertian intensity law, $I = \cos\theta$, describes the intensity of the reflected light as proportional to the cosine of the angle θ

[4]Lighting may be performed in world space as well, its simply a matter of changing the frame of reference.

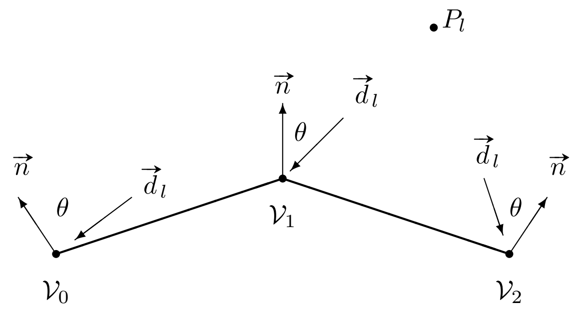

Figure 8.4: Lambert's law of diffuse reflection. The intensity of the light reflected is proportional to the cosine of the angle θ between the surface normal $\vec{n}$ and the direction of the light $\vec{d}_l$. The light at P_l reflects more light at vertices $\mathcal{V}_1$ and $\mathcal{V}_2$ than at vertex $\mathcal{V}_0$.

between the surface normal $\vec{n}$ and the light direction $\vec{d}_l$. The dot product of two unit vectors gives the cosine of the angle between them, giving $I = \vec{n} \cdot (-\vec{d}_l)$. The sign change is needed to keep $\cos\theta$ positive as $\vec{n}$ points away from the surface and $\vec{d}_l$ points away from the light. With this intensity profile, light is maximally reflected from the vertex when the light shines in the same direction as the surface normal.

The diffuse illumination from an enabled light, D_l, is attenuated based on distance and light type by a_l and then modulated by the material's diffuse reflectance coefficients D_v. The total diffuse light at the vertex is the sum of each enabled light's diffuse reflectance at the vertex.

$$\mathit{Diffuse} = D_v \sum_l a_l D_l(\vec{n} \cdot -\vec{d}_l)$$

8.6.4 Specular Reflection

Specular reflection accounts for specular highlights on smooth surfaces. It is modeled as cone of light reflecting from an object at an angle equal to the incident angle of the light on the surface. When the angle of the view incident to the surface is in the cone of specular reflection, a specular highlight results on the object's surface at that point. See figure 8.5.

Direct3D uses the Blinn-Phong specular reflection model that models the intensity of the specular highlight as proportional to the cosine of θ, the angle between the surface normal $\vec{n}$ and the half-way vector $\vec{h}$, raised to the power of p_m, the material specular power parameter. The half-way vector is the vector that is half-way between the surface normal and the light direction.

When RS Local Viewer is `TRUE`, the halfway vector is computed using the camera position[5], vertex position and light direction. When RS Local Viewer is

[5] From the translation components of the view matrix.

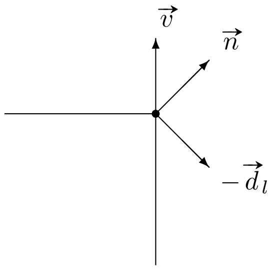

Figure 8.5: The geometry of specular reflection. Vector from origin to the vertex $\vec{v}$, vector from the vertex to the light $-\vec{d}_l$, surface normal vector $\vec{n}$, half-way vector $\vec{h}$.

FALSE, the halfway vector $\vec{h}$ is computed as the average of the vertex vector $\vec{v}$ and the vector from the vertex to the light $-\vec{d}_l$.

$$\begin{aligned}
\vec{v} &= \frac{0 - P_v}{\|0 - P_v\|} \\
\vec{s}_l &= \frac{P_l - P_v}{\|P_l - P_v\|} \\
\vec{h} &= \begin{cases} (\vec{v} + \vec{s}_l)/2, & \text{local viewer enabled.} \\ (\vec{v} + \vec{d}_l)/2, & \text{local viewer disabled.} \end{cases} \\
\mathit{Specular} &= S_v \sum_l a_l S_l (\vec{n} \cdot \vec{h})^{p_m}
\end{aligned}$$

8.6.5 Emission

Light emitted from a vertex is added directly to the reflected light. Emitted light does not depend on the position of any light sources, or the vertex surface normal.

$$\mathit{Emission} = E_v$$

8.6.6 Summary

Now that we've seen how all the terms of the illumination model are computed, let's look at the lighting equation again with a slight rearrangement of the terms:

$$\mathit{Lighting} = \mathit{Ambient} + \mathit{Diffuse} + \mathit{Specular} + \mathit{Emissive}$$

$$= A_v A_{rs} + E_v + \sum_l \left(A_v A_l + D_v a_l D_l (\vec{n} \cdot -\vec{d}_l) + S_v a_l S_l (\vec{n} \cdot \vec{h})^{p_m} \right)$$

Now you can see that the easiest way to use vertex processing and have a simply colored primitive with no enabled lights is to set the emissive material property E_v to the desired color and set the ambient lighting A_{rs} to zero. You can also see that all lighting computed for a vertex is a sum of different reflectance contributions. With fixed-function lighting this formula is the limit of what you can accomplish. To compute the lighting based on a different formula, you must use either "lit" vertices, or a vertex shader program implementing the desired formula.

8.7 `rt_Lighting` Sample Application

This sample application demonstrates lighting and material parameters by drawing a teapot over a background. The sample lets you interactively change all the parameters that affect lighting: render states, light type and associated parameters, and material parameters.

The entire source code is included in the samples. Listed here is `rt_Lighting.cpp`, containing the "interesting" code of the sample. The sample uses small helper classes that encapsulate reusable Direct3D coding idioms. Their meaning should be straightforward and all such helper classes are placed in the `rt` namespace to highlight their use. The sample source code contains their definitions.

Listing 8.1: `rt_Lighting.cpp`: Demonstration of lighting and material parameters.

```
//-----------------------------------------------------------------------------
// File: rt_Lighting.cpp
//
// Desc: DirectX window application created by the DirectX AppWizard
//-----------------------------------------------------------------------------
#include <sstream>
#include <vector>

#define STRICT
#include <windows.h>
#include <commctrl.h>

#include <atlbase.h>

#include <d3dx9.h>

#include "DXUtil.h"
```

```
#include "D3DEnumeration.h"
#include "D3DSettings.h"
#include "D3DApp.h"
#include "D3DFont.h"
#include "D3DUtil.h"

#include "rt/app.h"
#include "rt/dump.h"
#include "rt/hr.h"
#include "rt/hsv.h"
#include "rt/mat.h"
#include "rt/mesh.h"
#include "rt/misc.h"
#include "rt/states.h"
#include "rt/tstring.h"
#include "rt/rt_ColorSel.h"

#include "resource.h"
#include "rt_Lighting.h"

////////////////////////////////////////////////////////////
// vtx_d3dx
//
// Vertex structure given back by ::D3DXCreateXXX
//
struct vtx_d3dx
{
    D3DXVECTOR3 position;
    D3DXVECTOR3 normal;
};

////////////////////////////////////////////////////////////
// vtx_diffuse
//
// A d3dx vertex with a diffuse color.
//
struct vtx_diffuse : vtx_d3dx
{
    D3DCOLOR diffuse;
};

////////////////////////////////////////////////////////////
// vtx_specular
//
// A d3dx vertex with a specular color.
//
```

```
struct vtx_specular : vtx_d3dx
{
    D3DCOLOR specular;
};

/////////////////////////////////////////////////////////////
// vtx_diffuse_specular
//
// A d3dx vertex with a diffuse and specular color.
//
struct vtx_diffuse_specular : vtx_d3dx
{
    D3DCOLOR diffuse;
    D3DCOLOR specular;
};

/////////////////////////////////////////////////////////////
// c_bounding_box
//
// Compute the bounding box for a mesh.
//
class c_bounding_box
{
public:
    c_bounding_box(ID3DXMesh *mesh)
        : m_minima(0, 0, 0),
        m_maxima(0, 0, 0)
    {
        rt::mesh_vertex_lock<D3DXVECTOR3> lock(mesh);
        THR(::D3DXComputeBoundingBox(lock.data(),
            mesh->GetNumVertices(),
            ::D3DXGetFVFVertexSize(mesh->GetFVF()),
            &m_minima, &m_maxima));
    }
    ~c_bounding_box()
    {}

    const D3DXVECTOR3 &minima() const { return m_minima; }
    const D3DXVECTOR3 &maxima() const { return m_maxima; }

private:
    D3DXVECTOR3 m_minima;
    D3DXVECTOR3 m_maxima;
};

//------------------------------------------------------------------------------------
```

```
// Global access to the app (needed for the global WndProc())
//-----------------------------------------------------------------------------
CMyD3DApplication* g_pApp  = NULL;
HINSTANCE          g_hInst = NULL;




//-----------------------------------------------------------------------------
// Name: WinMain()
// Desc: Entry point to the program. Initializes everything, and goes into a
//       message-processing loop. Idle time is used to render the scene.
//-----------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, INT)
{
    try
    {
        CMyD3DApplication d3dApp;

        g_pApp  = &d3dApp;
        g_hInst = hInst;

        ::InitCommonControls();
        if (FAILED(d3dApp.Create(hInst)))
            return 0;

        return d3dApp.Run();
    }
    catch (rt::hr_message &bang)
    {
        rt::display_error(bang);
    }
    return -1;
}

////////////////////////////////////////////////////////////
// diffuse_color
//
// Compute a diffuse color for the mesh by creating a hue
// ramp in the x direction.
//
D3DCOLOR
diffuse_color(const D3DXVECTOR3 &position)
{
    return rt::hsv((position.x + 1.0f)*360.0f, 1.0f, 1.0f);
}
```

```

////////////////////////////////////////////////////////////
// specular_color
//
// Compute a specular color by creating a grayscale gradient
// over the top half of the mesh.
//
D3DCOLOR
specular_color(const D3DXVECTOR3 &position,
               const D3DXVECTOR3 &minima,
               const D3DXVECTOR3 &maxima)
{
    const float fraction =
        (position.y - minima.y)/(maxima.y - minima.y);
    const BYTE intensity = fraction > 0.5f ?
        BYTE(255*min((fraction - 0.5f)*2.0f, 1.0f)) : 0;

    return D3DCOLOR_XRGB(intensity, intensity, intensity);
}

////////////////////////////////////////////////////////////
// join
//
// Join a string literal onto a buffer if the value is true
//
void
join(rt::tostringstream &buff, bool val, LPCTSTR str)
{
    if (val)
    {
        if (buff.str().length())
        {
            buff << _T(", ");
        }
        buff << str;
    }
}

//-----------------------------------------------------------------------------
// Name: CMyD3DApplication()
// Desc: Application constructor.   Paired with ~CMyD3DApplication()
//       Member variables should be initialized to a known state here.
//       The application window has not yet been created and no Direct3D device
//       has been created, so any initialization that depends on a window or
//       Direct3D should be deferred to a later stage.
//-----------------------------------------------------------------------------
```

```
CMyD3DApplication::CMyD3DApplication() :
    CD3DApplication(),
    m_mesh_type(MT_TEAPOT),
    m_mesh_detail(0),
    m_mesh_nocolors(),
    m_mesh(),
    m_ambient(D3DCOLOR_XRGB(50, 50, 50)),
    m_background(D3DCOLOR_XRGB(100, 100, 100)),
    m_ambient_source(D3DMCS_MATERIAL),
    m_diffuse_source(D3DMCS_COLOR1),
    m_specular_source(D3DMCS_COLOR2),
    m_emissive_source(D3DMCS_MATERIAL),
    m_clone_fvf(0),
    m_shade_mode(D3DSHADE_GOURAUD),
    m_fill_mode(D3DFILL_SOLID),
    m_projection(PT_PERSPECTIVE),
    m_animate_view(false),
    m_light_enabled(true),
    m_show_text(true),
    m_lighting(true),
    m_local_viewer(true),
    m_normalize_normals(true),
    m_specular_enable(true),
    m_dithering(true),
    m_color_vertex(true),
    m_bLoadingApp(true),
    m_font(_T("Arial"), 12, D3DFONT_BOLD),
    m_input(),
    m_rot_x(0.0f),
    m_rot_y(0.0f)
{
    m_dwCreationWidth           = 500;
    m_dwCreationHeight          = 375;
    m_strWindowTitle            = TEXT("rt_Lighting");
    m_d3dEnumeration.AppUsesDepthBuffer   = TRUE;
        m_bStartFullscreen                      = false;
        m_bShowCursorWhenFullscreen     = false;

    const D3DMATERIAL9 mat =
    {
        0.5f, 0.5f, 0.5f, 0.5f, // diffuse
        0.1f, 0.1f, 0.1f, 0.1f, // ambient
        0.4f, 0.4f, 0.4f, 0.4f, // specular
        0, 0, 0, 0,             // emissive
        100                     // specular power
    };
```

```
    m_material = mat;
    const D3DLIGHT9 light =
    {
        D3DLIGHT_DIRECTIONAL,   // type
        0.5f, 0.5f, 0.5f, 0.5f, // diffuse
        0.4f, 0.4f, 0.4f, 0.4f, // specular
        0.1f, 0.1f, 0.1f, 0.1f, // ambient
        0, 0, -5,               // position
        0, 0, 1,                // direction
        10,                     // cutoff range
        1,                      // falloff
        1, 0, 0,                // attenuation
        0, D3DX_PI/2            // theta, phi
    };
    m_light = light;

    // Read settings from registry
    ReadSettings();
}




//-----------------------------------------------------------------------------
// Name: ~CMyD3DApplication()
// Desc: Application destructor.  Paired with CMyD3DApplication()
//-----------------------------------------------------------------------------
CMyD3DApplication::~CMyD3DApplication()
{
}




//-----------------------------------------------------------------------------
// Name: OneTimeSceneInit()
// Desc: Paired with FinalCleanup().
//       The window has been created and the IDirect3D9 interface has been
//       created, but the device has not been created yet.  Here you can
//       perform application-related initialization and cleanup that does
//       not depend on a device.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::OneTimeSceneInit()
{
    // TODO: perform one time initialization

```

```
    // Drawing loading status message until app finishes loading
    ::SendMessage(m_hWnd, WM_PAINT, 0, 0);

    m_bLoadingApp = false;

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: ReadSettings()
// Desc: Read the app settings from the registry
//-----------------------------------------------------------------------------
VOID CMyD3DApplication::ReadSettings()
{
    HKEY hkey;
    if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CURRENT_USER, DXAPP_KEY,
        0, NULL, REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL, &hkey, NULL))
    {
        // TODO: change as needed

        // Read the stored window width/height.  This is just an example,
        // of how to use ::DXUtil_Read*() functions.
        ::DXUtil_ReadIntRegKey(hkey, TEXT("Width"), &m_dwCreationWidth, m_dwCreationWidth)
        ::DXUtil_ReadIntRegKey(hkey, TEXT("Height"), &m_dwCreationHeight, m_dwCreationHeig

        ::RegCloseKey(hkey);
    }
}




//-----------------------------------------------------------------------------
// Name: WriteSettings()
// Desc: Write the app settings to the registry
//-----------------------------------------------------------------------------
VOID CMyD3DApplication::WriteSettings()
{
    HKEY hkey;

    if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CURRENT_USER, DXAPP_KEY,
        0, NULL, REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL, &hkey, NULL))
    {
```

```
        // TODO: change as needed

        // Write the window width/height.  This is just an example,
        // of how to use ::DXUtil_Write*() functions.
        ::DXUtil_WriteIntRegKey(hkey, TEXT("Width"), m_rcWindowClient.right);
        ::DXUtil_WriteIntRegKey(hkey, TEXT("Height"), m_rcWindowClient.bottom);

        ::RegCloseKey(hkey);
    }
}





//-----------------------------------------------------------------------------
// Name: InitDeviceObjects()
// Desc: Paired with DeleteDeviceObjects()
//       The device has been created.  Resources that are not lost on
//       Reset() can be created here -- resources in D3DPOOL_MANAGED,
//       D3DPOOL_SCRATCH, or D3DPOOL_SYSTEMMEM.  Image surfaces created via
//       CreateImageSurface are never lost and can be created here.  Vertex
//       shaders and pixel shaders can also be created here as they are not
//       lost on Reset().
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::InitDeviceObjects()
{
    create_mesh();

    // Init the font
    THR(m_font.InitDeviceObjects(m_pd3dDevice));

    return S_OK;
}


//-----------------------------------------------------------------------------
// Name: RestoreDeviceObjects()
// Desc: Paired with InvalidateDeviceObjects()
//       The device exists, but may have just been Reset().  Resources in
//       D3DPOOL_DEFAULT and any other device state that persists during
//       rendering should be set here.  Render states, matrices, textures,
//       etc., that don't change during rendering can be set once here to
//       avoid redundant state setting during Render() or FrameMove().
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::RestoreDeviceObjects()
```

```
{
    // Set up our view matrix. A view matrix can be defined given an eye point,
    // a point to lookat, and a direction for which way is up. Here, we set the
    // eye five units back along the z-axis and up three units, look at the
    // origin, and define "up" to be in the y-direction.
    D3DXMATRIX matView;
    D3DXVECTOR3 vFromPt   = D3DXVECTOR3( 0.0f, 0.0f, -5.0f );
    D3DXVECTOR3 vLookatPt = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    D3DXVECTOR3 vUpVec    = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    D3DXMatrixLookAtLH( &matView, &vFromPt, &vLookatPt, &vUpVec );
    m_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );

    // Set the projection matrix
    set_projection();

    // Restore the font
    THR(m_font.RestoreDeviceObjects());

    HMENU menu = ::GetMenu(m_hWnd);
    // can't select arbitrary color sources?
    bool enable =
        (m_d3dCaps.VertexProcessingCaps & D3DVTXPCAPS_MATERIALSOURCE7) != 0;
    {
        const UINT ids[] =
        {
            IDM_MATERIAL_AMBIENT_COLOR1, IDM_MATERIAL_AMBIENT_COLOR2,
            IDM_MATERIAL_DIFFUSE_COLOR2, IDM_MATERIAL_DIFFUSE_MATERIAL,
            IDM_MATERIAL_SPECULAR_COLOR1, IDM_MATERIAL_SPECULAR_MATERIAL,
            IDM_MATERIAL_EMISSIVE_COLOR1, IDM_MATERIAL_EMISSIVE_COLOR2
        };
        rt::enable_ids(menu, ids, NUM_OF(ids), enable);
    }
    if (!enable)
    {
        if ((D3DMCS_COLOR1 == m_ambient_source) ||
            (D3DMCS_COLOR2 == m_ambient_source))
        {
            rt::check_menu(menu, IDM_MATERIAL_AMBIENT_COLOR1,   false);
            rt::check_menu(menu, IDM_MATERIAL_AMBIENT_COLOR2,   false);
            rt::check_menu(menu, IDM_MATERIAL_AMBIENT_MATERIAL, true);
            m_ambient_source = D3DMCS_MATERIAL;
        }
        if ((D3DMCS_COLOR2 == m_diffuse_source) ||
            (D3DMCS_MATERIAL == m_diffuse_source))
        {
            rt::check_menu(menu, IDM_MATERIAL_DIFFUSE_COLOR1,   true);
```

```
            rt::check_menu(menu, IDM_MATERIAL_DIFFUSE_COLOR2,   false);
            rt::check_menu(menu, IDM_MATERIAL_DIFFUSE_MATERIAL, false);
            m_diffuse_source = D3DMCS_COLOR1;
        }
        if ((D3DMCS_COLOR1 == m_specular_source) ||
            (D3DMCS_MATERIAL == m_specular_source))
        {
            rt::check_menu(menu, IDM_MATERIAL_SPECULAR_COLOR1,   false);
            rt::check_menu(menu, IDM_MATERIAL_SPECULAR_COLOR2,   true);
            rt::check_menu(menu, IDM_MATERIAL_SPECULAR_MATERIAL, false);
            m_specular_source = D3DMCS_COLOR2;
        }
        if ((D3DMCS_COLOR1 == m_emissive_source) ||
            (D3DMCS_COLOR2 == m_emissive_source))
        {
            rt::check_menu(menu, IDM_MATERIAL_EMISSIVE_COLOR1,   false);
            rt::check_menu(menu, IDM_MATERIAL_EMISSIVE_COLOR2,   false);
            rt::check_menu(menu, IDM_MATERIAL_EMISSIVE_MATERIAL, true);
            m_emissive_source = D3DMCS_MATERIAL;
        }
    }
    update_material_menu();

    // can't perform gouraud shading?
    enable = (m_d3dCaps.ShadeCaps & D3DPSHADECAPS_COLORGOURAUDRGB) != 0;
    rt::enable_menu(menu, IDM_SHADING_GOURAUD, enable);
    if (!enable && (D3DSHADE_GOURAUD == m_shade_mode))
    {
        rt::check_menu(menu, IDM_SHADING_FLAT, true);
        rt::check_menu(menu, IDM_SHADING_GOURAUD, false);
        m_shade_mode = D3DSHADE_FLAT;
    }

    // can't perform local viewer calculations?
    enable = (m_d3dCaps.VertexProcessingCaps & D3DVTXPCAPS_LOCALVIEWER) != 0;
    rt::enable_menu(menu, IDM_OPTION_LOCAL_VIEWER, enable);
    m_local_viewer = enable && m_local_viewer;
    rt::check_menu(menu, IDM_OPTION_LOCAL_VIEWER, m_local_viewer);

    // can't handle all light types?
    switch (m_d3dCaps.VertexProcessingCaps &
            (D3DVTXPCAPS_DIRECTIONALLIGHTS | D3DVTXPCAPS_POSITIONALLIGHTS))
    {
    case D3DVTXPCAPS_DIRECTIONALLIGHTS | D3DVTXPCAPS_POSITIONALLIGHTS:
        // can do all light types
        break;
```

```
    case D3DVTXPCAPS_POSITIONALLIGHTS:
        // can't do directional lights
        rt::enable_menu(menu, IDM_LIGHT_TYPE_DIRECTIONAL, false);
        if (D3DLIGHT_DIRECTIONAL == m_light.Type)
        {
            m_light.Type = D3DLIGHT_SPOT;
            update_light_menu(menu);
        }
        break;

    case D3DVTXPCAPS_DIRECTIONALLIGHTS:
        // can't do positional (point or spot) lights
        rt::enable_menu(menu, IDM_LIGHT_TYPE_POINT, false);
        rt::enable_menu(menu, IDM_LIGHT_TYPE_SPOT, false);
        if ((D3DLIGHT_POINT == m_light.Type) ||
            (D3DLIGHT_SPOT == m_light.Type))
        {
            m_light.Type = D3DLIGHT_DIRECTIONAL;
            update_light_menu(menu);
        }
        break;

    case 0:
        // can't do lights!
        rt::enable_menu(menu, IDM_LIGHT_TYPE_DIRECTIONAL, false);
        rt::enable_menu(menu, IDM_LIGHT_TYPE_POINT, false);
        rt::enable_menu(menu, IDM_LIGHT_TYPE_SPOT, false);
        rt::enable_menu(menu, IDM_LIGHT_ENABLE, false);
        rt::check_menu(menu, IDM_LIGHT_TYPE_DIRECTIONAL, false);
        rt::check_menu(menu, IDM_LIGHT_TYPE_POINT, false);
        rt::check_menu(menu, IDM_LIGHT_TYPE_SPOT, false);
        rt::check_menu(menu, IDM_LIGHT_ENABLE, false);
        m_light_enabled = false;
        break;

    default:
        ATLASSERT(false);
    }

    return S_OK;
}
```

```
//-----------------------------------------------------------------------------
// Name: FrameMove()
// Desc: Called once per frame, the call is the entry point for animating
//       the scene.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::FrameMove()
{
    // TODO: update world

    // Update user input state
    UpdateInput();

    // Update the world state according to user input
    if (m_animate_view)
    {
        m_rot_y += m_fElapsedTime;
        if (m_rot_y > 2*D3DX_PI)
        {
            m_rot_y = fmodf(m_rot_y, 2*D3DX_PI);
        }
    }
    else
    {
        if (m_input.m_left && !m_input.m_right)
        {
            m_rot_y += m_fElapsedTime;
        }
        else if (m_input.m_right && !m_input.m_left)
        {
            m_rot_y -= m_fElapsedTime;
        }
    }
    if (m_input.m_up && !m_input.m_down)
    {
        m_rot_x += m_fElapsedTime;
    }
    else if (m_input.m_down && !m_input.m_up)
    {
        m_rot_x -= m_fElapsedTime;
    }

    THR(m_pd3dDevice->SetTransform(D3DTS_WORLD,
        rt::anon(rt::mat_rot_x(m_rot_x)*rt::mat_rot_y(m_rot_y))));

    return S_OK;
}
```

```
//-----------------------------------------------------------------------------
// Name: UpdateInput()
// Desc: Update the user input.  Called once per frame
//-----------------------------------------------------------------------------
void CMyD3DApplication::UpdateInput()
{
    m_input.m_up    = (m_bActive && (GetAsyncKeyState(VK_UP)    & 0x8000) == 0x8000);
    m_input.m_down  = (m_bActive && (GetAsyncKeyState(VK_DOWN)  & 0x8000) == 0x8000);
    m_input.m_left  = (m_bActive && (GetAsyncKeyState(VK_LEFT)  & 0x8000) == 0x8000);
    m_input.m_right = (m_bActive && (GetAsyncKeyState(VK_RIGHT) & 0x8000) == 0x8000);
}




//-----------------------------------------------------------------------------
// Name: Render()
// Desc: Called once per frame, the call is the entry point for 3d
//       rendering. This function sets up render states, clears the
//       viewport, and renders the scene.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::Render()
{
    // clear the frame buffer and start the scene
    THR(m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_TARGET |
        D3DCLEAR_ZBUFFER, m_background, 1.0f, 0L));
    THR(m_pd3dDevice->BeginScene());

    // set the light and enable it, if necessary
    THR(m_pd3dDevice->SetLight(0, &m_light));
    THR(m_pd3dDevice->LightEnable(0, m_light_enabled));

    // set the material parameters
    THR(m_pd3dDevice->SetMaterial(&m_material));

    // set lighting-related and other render states
    rt::s_rs states[] =
    {
        D3DRS_AMBIENT,                  m_ambient,
        D3DRS_COLORVERTEX,              m_color_vertex,
        D3DRS_LIGHTING,                 m_lighting,
        D3DRS_LOCALVIEWER,              m_local_viewer,
```

```
        D3DRS_NORMALIZENORMALS,         m_normalize_normals,
        D3DRS_AMBIENTMATERIALSOURCE,    m_ambient_source,
        D3DRS_DIFFUSEMATERIALSOURCE,    m_diffuse_source,
        D3DRS_SPECULARMATERIALSOURCE,   m_specular_source,
        D3DRS_EMISSIVEMATERIALSOURCE,   m_emissive_source,
        D3DRS_SPECULARENABLE,           m_specular_enable,
        D3DRS_SHADEMODE,                m_shade_mode,
        D3DRS_FILLMODE,                 m_fill_mode,
        D3DRS_ALPHATESTENABLE,          false,
        D3DRS_STENCILENABLE,            false,
        D3DRS_ZENABLE,                  D3DZB_TRUE,
        D3DRS_ZFUNC,                    D3DCMP_LESS,
        D3DRS_ZWRITEENABLE,             true,
        D3DRS_ALPHABLENDENABLE,         false,
        D3DRS_DITHERENABLE,             m_dithering
    };
    rt::set_states(m_pd3dDevice, states, NUM_OF(states));

    // orient and scale the mesh
    THR(m_pd3dDevice->SetTransform(D3DTS_WORLD,
        rt::anon(rt::mat_rot_x(m_rot_x)*rt::mat_rot_y(m_rot_y)*
                 rt::mat_scale(1.4f))));
    THR(m_mesh->DrawSubset(0));

    // show statistics if necessary
    if (m_show_text)
    {
        RenderText();
    }

    THR(m_pd3dDevice->EndScene());

    return S_OK;
}

//-----------------------------------------------------------------------------
// Name: RenderText()
// Desc: Renders stats and help text to the scene.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::RenderText()
{
    D3DCOLOR yellow        = D3DCOLOR_ARGB(255,255,255,0);
    TCHAR szMsg[MAX_PATH] = TEXT("");

    // Output display stats
    FLOAT y = 40.0f;
```

```

    y -= 20.0f;
    THR(m_font.DrawText(2, y, yellow, m_strDeviceStats));

    y -= 20.0f;
    THR(m_font.DrawText(2, y, yellow, m_strFrameStats));

    y = 40.0f;
    {
        rt::tostringstream buff;
        join(buff, m_lighting, _T("lighting"));
        join(buff, m_color_vertex, _T("color vertex"));
        join(buff, m_local_viewer, _T("local viewer"));
        join(buff, m_normalize_normals, _T("normalize normals"));
        join(buff, m_specular_enable, _T("specular"));
        join(buff, m_dithering, _T("dithering"));
        THR(m_font.DrawText(2, y, yellow, const_cast<LPTSTR>(buff.str().c_str())));
    }
    y += 20.0f;
#if 0
    {
        rt::tostringstream buff;
        buff << _T("Ambient: ") << m_ambient_source;
        if (D3DMCS_MATERIAL == m_ambient_source)
        {
            buff << _T(", material: ") << m_material.Ambient;
        }
        buff << _T(", light: ") << m_light.Ambient;
        THR(m_font.DrawText(2, y, yellow, const_cast<LPTSTR>(buff.str().c_str())));
    }
    y += 20.0f;
    {
        rt::tostringstream buff;
        buff << _T("Diffuse: ") << m_diffuse_source << _T(", material: ")
            << m_material.Diffuse << _T(", light: ")
            << m_light.Diffuse;
        THR(m_font.DrawText(2, y, yellow, const_cast<LPTSTR>(buff.str().c_str())));
    }
    y += 20.0f;
    {
        rt::tostringstream buff;
        buff << _T("Specular: ") << m_specular_source << _T(", material: ")
            << m_material.Specular << _T(", light: ")
            << m_light.Specular;
        THR(m_font.DrawText(2, y, yellow, const_cast<LPTSTR>(buff.str().c_str())));
    }
```

```
    y += 20.0f;
    {
        rt::tostringstream buff;
        buff << _T("Emissive: ") << m_emissive_source << _T(", material: ")
            << m_material.Emissive;
        THR(m_font.DrawText(2, y, yellow, const_cast<LPTSTR>(buff.str().c_str())
    }
#endif

    // Output statistics & help
    y = m_d3dsdBackBuffer.Height - 22.f;
    _stprintf(szMsg, TEXT("Arrow keys: Up=%d Down=%d Left=%d Right=%d"),
              m_input.m_up, m_input.m_down, m_input.m_left, m_input.m_right);
    THR(m_font.DrawText(2, y, yellow, szMsg));

    _stprintf(szMsg, TEXT("World State: %0.3f, %0.3f"),  m_rot_x, m_rot_y);
    y -= 20.0f;
    THR(m_font.DrawText(2, y, yellow, szMsg));
    y -= 20.0f;
    THR(m_font.DrawText(2, y, yellow, _T("Use arrow keys to update input")));
    y -= 20.0f;
    THR(m_font.DrawText(2, y, yellow, _T("Press 'F2' to configure display")));
    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: MsgProc()
// Desc: Overrrides the main WndProc, so the sample can do custom message
//       handling (e.g. processing mouse, keyboard, or menu commands).
//-----------------------------------------------------------------------------
LRESULT CMyD3DApplication::MsgProc(HWND hWnd, UINT msg, WPARAM wParam,
                                    LPARAM lParam)
{
    LRESULT result = 0;
    bool handled = false;

    switch( msg )
    {
    case WM_PAINT:
        if (m_bLoadingApp)
        {
            // Draw on the window tell the user that the app is loading
            HDC hDC = ::GetDC(hWnd);
```

```
                RECT rct;
                ::GetClientRect(hWnd, &rct);
                ::DrawText(hDC, _T("Loading... Please wait"), -1, &rct,
                    DT_CENTER | DT_VCENTER | DT_SINGLELINE);
                ::ReleaseDC(hWnd, hDC);
            }
            break;

        case WM_COMMAND:
            result = on_command(hWnd, wParam, lParam, handled);
            break;
        }

        return handled ? result :
            CD3DApplication::MsgProc(hWnd, msg, wParam, lParam);
}




//-----------------------------------------------------------------------------
// Name: InvalidateDeviceObjects()
// Desc: Invalidates device objects.  Paired with RestoreDeviceObjects()
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    // TODO: Cleanup any objects created in RestoreDeviceObjects()
    THR(m_font.InvalidateDeviceObjects());

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: DeleteDeviceObjects()
// Desc: Paired with InitDeviceObjects()
//       Called when the app is exiting, or the device is being changed,
//       this function deletes any device dependent objects.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    m_mesh_nocolors = 0;
    m_mesh = 0;

```

```
    THR(m_font.DeleteDeviceObjects());

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: FinalCleanup()
// Desc: Paired with OneTimeSceneInit()
//       Called before the app exits, this function gives the app the chance
//       to cleanup after itself.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::FinalCleanup()
{
    // TODO: Perform any final cleanup needed

    // Write the settings to the registry
    WriteSettings();

    return S_OK;
}


inline UINT
detail(UINT value)
{
    return 5 + value*10;
}


void
CMyD3DApplication::create_mesh()
{
    rt::dx_buffer<DWORD> adj;

    switch (m_mesh_type)
    {
    case MT_BOX:
        {
            rt::dx_buffer<DWORD> tmp_adj;
            CComPtr<ID3DXMesh> tmp;
            THR(::D3DXCreateBox(m_pd3dDevice, 1.f, 1.f, 1.f, &tmp, &tmp_adj));

            // tesselate it using the N-patch algorithm to get smoother
```

```
                // lighting than with the plain mesh.
                if (m_mesh_detail > 0)
                {
                    THR(::D3DXTessellateNPatches(tmp, tmp_adj,
                        1.f + m_mesh_detail, true, &m_mesh_nocolors, &adj));
                }
                else
                {
                    m_mesh_nocolors = tmp;
                    adj = tmp_adj;
                }
            }
            break;

        case MT_SPHERE:
            THR(::D3DXCreateSphere(m_pd3dDevice, 1.f,
                detail(m_mesh_detail), detail(m_mesh_detail),
                &m_mesh_nocolors, &adj));
            break;

        case MT_CYLINDER:
            THR(::D3DXCreateCylinder(m_pd3dDevice, 0.7f, 1.f, 1.f,
                detail(m_mesh_detail), detail(m_mesh_detail),
                &m_mesh_nocolors, &adj));
            break;

        case MT_TORUS:
            THR(::D3DXCreateTorus(m_pd3dDevice, 0.7f, 1.f,
                detail(m_mesh_detail), detail(m_mesh_detail),
                &m_mesh_nocolors, &adj));
            break;

        case MT_TEAPOT:
            {
                rt::dx_buffer<DWORD> tmp_adj;
                CComPtr<ID3DXMesh> tmp;
                THR(::D3DXCreateTeapot(m_pd3dDevice, &tmp, &tmp_adj));

                if (m_mesh_detail > 0)
                {
                    // tesselate it using the N-patch algorithm to get smoother
                    // lighting than with the plain mesh.
                    THR(::D3DXTessellateNPatches(tmp, tmp_adj,
                        float(min(5, m_mesh_detail)), true, &m_mesh_nocolors, &adj));
                }
                else
```

```
            {
                m_mesh_nocolors = tmp;
                adj = tmp_adj;
            }
        }
        break;

    default:
        ATLASSERT(false);
    }

    // now optimize it for rendering
    std::vector<DWORD> adj_out(m_mesh_nocolors->GetNumFaces()*3);
    std::vector<DWORD> face_remap(m_mesh_nocolors->GetNumFaces());
    CComPtr<ID3DXBuffer> vertex_remap;
    THR(m_mesh_nocolors->OptimizeInplace(D3DXMESHOPT_ATTRSORT, adj,
        &adj_out[0], &face_remap[0], &vertex_remap));

    // clone any per-vertex colors needed
    clone_mesh(m_clone_fvf);
}

////////////////////////////////////////////////////////////////////////
// CMyD3DApplication::set_projection
//
// Set the projection matrix to either perspective or orthographic.
//
void
CMyD3DApplication::set_projection()
{
    const float aspect =
        float(m_d3dsdBackBuffer.Width)/m_d3dsdBackBuffer.Height;
    D3DXMATRIX proj;
    if (PT_PERSPECTIVE == m_projection)
    {
        ::D3DXMatrixPerspectiveFovLH(&proj, D3DX_PI/4, aspect, 2.0f, 20.0f);
    }
    else
    {
        ::D3DXMatrixOrthoLH(&proj, 4*aspect, 4, 2.0f, 20.0f);
    }
    THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &proj));
}

////////////////////////////////////////////////////////////////////////
// CMyD3DApplication::update_light_menu
```

```
//
// When changing the light type, we need to enable or disable the
// relevant lighting parameter menu items.
//
void
CMyD3DApplication::update_light_menu(HMENU menu)
{
    switch (m_light.Type)
    {
    case D3DLIGHT_POINT:
        rt::check_menu(menu, IDM_LIGHT_TYPE_POINT,       true);
        rt::check_menu(menu, IDM_LIGHT_TYPE_SPOT,        false);
        rt::check_menu(menu, IDM_LIGHT_TYPE_DIRECTIONAL, false);
        rt::enable_menu(menu, IDM_LIGHT_POSITION,    true);
        rt::enable_menu(menu, IDM_LIGHT_DIRECTION,   false);
        rt::enable_menu(menu, IDM_LIGHT_RANGE,       true);
        rt::enable_menu(menu, IDM_LIGHT_ATTENUATION, true);
        rt::enable_menu(menu, IDM_LIGHT_CONE,        false);
        break;

    case D3DLIGHT_SPOT:
        rt::check_menu(menu, IDM_LIGHT_TYPE_POINT,       false);
        rt::check_menu(menu, IDM_LIGHT_TYPE_SPOT,        true);
        rt::check_menu(menu, IDM_LIGHT_TYPE_DIRECTIONAL, false);
        rt::enable_menu(menu, IDM_LIGHT_POSITION,    true);
        rt::enable_menu(menu, IDM_LIGHT_DIRECTION,   true);
        rt::enable_menu(menu, IDM_LIGHT_RANGE,       true);
        rt::enable_menu(menu, IDM_LIGHT_ATTENUATION, true);
        rt::enable_menu(menu, IDM_LIGHT_CONE,        true);
        break;

    case D3DLIGHT_DIRECTIONAL:
        rt::check_menu(menu, IDM_LIGHT_TYPE_POINT,       false);
        rt::check_menu(menu, IDM_LIGHT_TYPE_SPOT,        false);
        rt::check_menu(menu, IDM_LIGHT_TYPE_DIRECTIONAL, true);
        rt::enable_menu(menu, IDM_LIGHT_POSITION,    false);
        rt::enable_menu(menu, IDM_LIGHT_DIRECTION,   true);
        rt::enable_menu(menu, IDM_LIGHT_RANGE,       false);
        rt::enable_menu(menu, IDM_LIGHT_ATTENUATION, false);
        rt::enable_menu(menu, IDM_LIGHT_CONE,        false);
        break;

    default:
        ATLASSERT(false);
    }
}
```

```

////////////////////////////////////////////////////////////////////////
// CMyD3DApplication::update_material_menu
//
// Enable/disable items on the materials menu based on current state.
//
void
CMyD3DApplication::update_material_menu()
{
    HMENU menu = ::GetMenu(m_hWnd);
    rt::enable_menu(menu, IDM_MATERIAL_AMBIENT,
        (D3DMCS_MATERIAL == m_ambient_source));
    rt::enable_menu(menu, IDM_MATERIAL_DIFFUSE,
        (D3DMCS_MATERIAL == m_diffuse_source));
    rt::enable_menu(menu, IDM_MATERIAL_SPECULAR,
        (D3DMCS_MATERIAL == m_specular_source));
    rt::enable_menu(menu, IDM_MATERIAL_EMISSIVE,
        (D3DMCS_MATERIAL == m_emissive_source));
}

/////////////////////////////////////////////////////////////
// CMyD3DApplication::clone_mesh
//
// Clone the mesh without per-vertex colors into a mesh with
// the requested per-vertex colors.  Then compute the requested
// colors into the cloned data.
//
void
CMyD3DApplication::clone_mesh(DWORD fvf)
{
    // fvf must only contain specular, diffuse, or both
    ATLASSERT(!(fvf & ~(D3DFVF_DIFFUSE | D3DFVF_SPECULAR)));

    m_mesh = 0;
    m_clone_fvf = fvf;
    if (!fvf)
    {
        m_mesh = m_mesh_nocolors;
        return;
    }

    THR(m_mesh_nocolors->CloneMeshFVF(D3DXMESH_MANAGED,
        m_mesh_nocolors->GetFVF() | fvf, m_pd3dDevice,
        &m_mesh));

    if (D3DFVF_DIFFUSE == fvf)
```

```
        {
            rt::mesh_vertex_lock<vtx_diffuse> lock(m_mesh);
            vtx_diffuse *verts = lock.data();

            for (UINT i = 0; i < m_mesh->GetNumVertices(); i++)
            {
                verts[i].diffuse =
                    diffuse_color(verts[i].position);
            }
        }
        else if (D3DFVF_SPECULAR == fvf)
        {
            rt::mesh_vertex_lock<vtx_specular> lock(m_mesh);
            vtx_specular *verts = lock.data();
            c_bounding_box minmax(m_mesh);

            for (UINT i = 0; i < m_mesh->GetNumVertices(); i++)
            {
                verts[i].specular =
                    specular_color(verts[i].position,
                        minmax.minima(), minmax.maxima());
            }
        }
        else // ((D3DFVF_DIFFUSE | D3DFVF_SPECULAR) == fvf)
        {
            rt::mesh_vertex_lock<vtx_diffuse_specular>
                lock(m_mesh);
            vtx_diffuse_specular *verts = lock.data();
            c_bounding_box minmax(m_mesh);

            for (UINT i = 0; i < m_mesh->GetNumVertices(); i++)
            {
                verts[i].diffuse =
                    diffuse_color(verts[i].position);
                verts[i].specular =
                    specular_color(verts[i].position,
                        minmax.minima(), minmax.maxima());
            }
        }
    }

    ////////////////////////////////////////////////////////////////////
    // CMyD3DApplication::on_command
    //
    // WM_COMMAND message handler
    //
```

```
LRESULT
CMyD3DApplication::on_command(HWND window, WPARAM wp, LPARAM, bool &handled)
{
    handled = false;
    LRESULT result = 0;
    const HMENU menu = ::GetMenu(window);
    const UINT control = LOWORD(wp);

    switch (control)
    {
    case IDM_MESH_BOX:
    case IDM_MESH_SPHERE:
    case IDM_MESH_CYLINDER:
    case IDM_MESH_TORUS:
    case IDM_MESH_TEAPOT:
        rt::check_menu(menu, IDM_MESH_BOX + m_mesh_type, false);
        m_mesh_type = e_mesh_type(control - IDM_MESH_BOX);
        rt::check_menu(menu, IDM_MESH_BOX + m_mesh_type, true);
        recreate_mesh();
        handled = true;
        break;

    case IDM_MESH_DETAIL_1X:
    case IDM_MESH_DETAIL_2X:
    case IDM_MESH_DETAIL_3X:
    case IDM_MESH_DETAIL_4X:
    case IDM_MESH_DETAIL_5X:
    case IDM_MESH_DETAIL_6X:
    case IDM_MESH_DETAIL_7X:
    case IDM_MESH_DETAIL_8X:
    case IDM_MESH_DETAIL_9X:
        rt::check_menu(menu, IDM_MESH_DETAIL_1X + m_mesh_detail, false);
        m_mesh_detail = control - IDM_MESH_DETAIL_1X;
        rt::check_menu(menu, IDM_MESH_DETAIL_1X + m_mesh_detail, true);
        recreate_mesh();
        handled = true;
        break;

    case IDM_VERTEX_COLOR_NONE:
        clone_mesh(0);
        rt::check_menu(menu, IDM_VERTEX_COLOR_NONE, true);
        rt::check_menu(menu, IDM_VERTEX_COLOR_DIFFUSE, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_SPECULAR, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_DIFFUSE_SPECULAR, false);
        handled = true;
        break;
```

```

    case IDM_VERTEX_COLOR_DIFFUSE:
        clone_mesh(D3DFVF_DIFFUSE);
        rt::check_menu(menu, IDM_VERTEX_COLOR_NONE, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_DIFFUSE, true);
        rt::check_menu(menu, IDM_VERTEX_COLOR_SPECULAR, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_DIFFUSE_SPECULAR, false);
        handled = true;
        break;

    case IDM_VERTEX_COLOR_SPECULAR:
        clone_mesh(D3DFVF_SPECULAR);
        rt::check_menu(menu, IDM_VERTEX_COLOR_NONE, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_DIFFUSE, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_SPECULAR, true);
        rt::check_menu(menu, IDM_VERTEX_COLOR_DIFFUSE_SPECULAR, false);
        handled = true;
        break;

    case IDM_VERTEX_COLOR_DIFFUSE_SPECULAR:
        clone_mesh(D3DFVF_DIFFUSE | D3DFVF_SPECULAR);
        rt::check_menu(menu, IDM_VERTEX_COLOR_NONE, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_DIFFUSE, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_SPECULAR, false);
        rt::check_menu(menu, IDM_VERTEX_COLOR_DIFFUSE_SPECULAR, true);
        handled = true;
        break;

#define MODIFY_MATERIAL_COLOR(id_, member_, transparent_)                   \
    case id_:                                                               \
        {                                                                   \
            rt::pauser block(*this);                                        \
            m_material.member_ = D3DXCOLOR(rt::choose_color_transparent(window, \
                D3DXCOLOR(m_material.member_), transparent_));              \
        }                                                                   \
        handled = true;                                                     \
        break
    MODIFY_MATERIAL_COLOR(IDM_MATERIAL_DIFFUSE, Diffuse, true);
    MODIFY_MATERIAL_COLOR(IDM_MATERIAL_SPECULAR, Specular, false);
    MODIFY_MATERIAL_COLOR(IDM_MATERIAL_AMBIENT, Ambient, false);
    MODIFY_MATERIAL_COLOR(IDM_MATERIAL_EMISSIVE, Emissive, true);
#undef MODIFY_MATERIAL_COLOR

#define MODIFY_MATERIAL_SOURCE(state_, id_)     \
    case IDM_MATERIAL_##id_##_MATERIAL:         \
    case IDM_MATERIAL_##id_##_COLOR1:           \
```

```
    case IDM_MATERIAL_##id_##_COLOR2:           \
        rt::check_menu(menu, IDM_MATERIAL_##id_##_MATERIAL + UINT(state_), false
        state_ = D3DMATERIALCOLORSOURCE(control - IDM_MATERIAL_##id_##_MATERIAL)
        rt::check_menu(menu, IDM_MATERIAL_##id_##_MATERIAL + UINT(state_), true)
        update_material_menu();                 \
        handled = true;                         \
        break
    MODIFY_MATERIAL_SOURCE(m_ambient_source, AMBIENT);
    MODIFY_MATERIAL_SOURCE(m_diffuse_source, DIFFUSE);
    MODIFY_MATERIAL_SOURCE(m_specular_source, SPECULAR);
    MODIFY_MATERIAL_SOURCE(m_emissive_source, EMISSIVE);
#undef MODIFY_MATERIAL_SOURCE

    case IDM_MATERIAL_SPECULAR_POWER:
        update_material_parameters(IDD_MATERIAL_SPECULAR_POWER, material_specula
        handled = true;
        break;

#define MODIFY_LIGHT_REFLECTANCE(id_, member_, transparent_)                \
    case id_:                                                               \
        {                                                                   \
            rt::pauser block(*this);                                        \
            m_light.member_ = D3DXCOLOR(rt::choose_color_transparent(window,
                D3DXCOLOR(m_light.member_), transparent_));                 \
        }                                                                   \
        handled = true;                                                     \
        break
    MODIFY_LIGHT_REFLECTANCE(IDM_LIGHT_AMBIENT, Ambient, false);
    MODIFY_LIGHT_REFLECTANCE(IDM_LIGHT_DIFFUSE, Diffuse, true);
    MODIFY_LIGHT_REFLECTANCE(IDM_LIGHT_SPECULAR, Specular, false);
#undef MODIFY_LIGHT_REFLECTANCE

    case IDM_LIGHT_TYPE_POINT:
    case IDM_LIGHT_TYPE_SPOT:
    case IDM_LIGHT_TYPE_DIRECTIONAL:
        m_light.Type = D3DLIGHTTYPE(control - IDM_LIGHT_TYPE_POINT + 1);
        update_light_menu(menu);
        handled = true;
        break;

    case IDM_LIGHT_POSITION:
        update_light_parameters(IDD_LIGHT_POSITION, light_position_proc);
        handled = true;
        break;

    case IDM_LIGHT_DIRECTION:
```

```
            update_light_parameters(IDD_LIGHT_DIRECTION, light_direction_proc);
            handled = true;
            break;

        case IDM_LIGHT_RANGE:
            update_light_parameters(IDD_LIGHT_RANGE, light_range_proc);
            handled = true;
            break;

        case IDM_LIGHT_ATTENUATION:
            update_light_parameters(IDD_LIGHT_ATTENUATION, light_attenuation_proc);
            handled = true;
            break;

        case IDM_LIGHT_CONE:
            update_light_parameters(IDD_LIGHT_CONE, light_cone_proc);
            handled = true;
            break;

        case IDM_SHADING_FLAT:
            m_shade_mode = D3DSHADE_FLAT;
            rt::check_menu(menu, IDM_SHADING_FLAT, true);
            rt::check_menu(menu, IDM_SHADING_GOURAUD, false);
            handled = true;
            break;

        case IDM_SHADING_GOURAUD:
            m_shade_mode = D3DSHADE_GOURAUD;
            rt::check_menu(menu, IDM_SHADING_FLAT, false);
            rt::check_menu(menu, IDM_SHADING_GOURAUD, true);
            handled = true;
            break;

        case IDM_FILL_POINT:
            m_fill_mode = D3DFILL_POINT;
            rt::check_menu(menu, IDM_FILL_POINT, true);
            rt::check_menu(menu, IDM_FILL_WIREFRAME, false);
            rt::check_menu(menu, IDM_FILL_SOLID, false);
            handled = true;
            break;

        case IDM_FILL_WIREFRAME:
            m_fill_mode = D3DFILL_WIREFRAME;
            rt::check_menu(menu, IDM_FILL_POINT, false);
            rt::check_menu(menu, IDM_FILL_WIREFRAME, true);
            rt::check_menu(menu, IDM_FILL_SOLID, false);
```

```
            handled = true;
            break;

        case IDM_FILL_SOLID:
            m_fill_mode = D3DFILL_SOLID;
            rt::check_menu(menu, IDM_FILL_POINT, false);
            rt::check_menu(menu, IDM_FILL_WIREFRAME, false);
            rt::check_menu(menu, IDM_FILL_SOLID, true);
            handled = true;
            break;

        case IDM_PROJECTION_ORTHOGRAPHIC:
            rt::check_menu(menu, IDM_PROJECTION_PERSPECTIVE, false);
            rt::check_menu(menu, IDM_PROJECTION_ORTHOGRAPHIC, true);
            m_projection = PT_ORTHOGRAPHIC;
            set_projection();
            handled = true;
            break;

        case IDM_PROJECTION_PERSPECTIVE:
            rt::check_menu(menu, IDM_PROJECTION_PERSPECTIVE, true);
            rt::check_menu(menu, IDM_PROJECTION_ORTHOGRAPHIC, false);
            m_projection = PT_PERSPECTIVE;
            set_projection();
            handled = true;
            break;

    #define TOGGLE(id_, state_)                 \
        case id_:                               \
            rt::toggle_menu(menu, id_, state_); \
            handled = true;                     \
            break
        TOGGLE(IDM_LIGHT_ENABLE, m_light_enabled);
        TOGGLE(IDM_OPTION_ANIMATE_VIEW, m_animate_view);
        TOGGLE(IDM_OPTION_SHOW_TEXT, m_show_text);
        TOGGLE(IDM_OPTION_NORMALIZE_NORMALS, m_normalize_normals);
        TOGGLE(IDM_OPTION_LOCAL_VIEWER, m_local_viewer);
        TOGGLE(IDM_OPTION_LIGHTING, m_lighting);
        TOGGLE(IDM_OPTION_SPECULAR_ENABLE, m_specular_enable);
        TOGGLE(IDM_OPTION_DITHER, m_dithering);
        TOGGLE(IDM_OPTION_COLOR_VERTEX, m_color_vertex);
    #undef TOGGLE

        case IDM_OPTION_AMBIENT_COLOR:
            {
                rt::pauser block(*this);
```

```
            m_ambient = rt::choose_color(window, m_ambient);
        }
        handled = true;
        break;

    case IDM_OPTION_RESET_VIEW:
        m_rot_x = 0.0f;
        m_rot_y = 0.0f;
        handled = true;
        break;

    case IDM_OPTION_BACKGROUND_COLOR:
        {
            rt::pauser block(*this);
            m_background = rt::choose_color_transparent(window, m_background, true);
        }
        handled = true;
        break;

    default:
        // all our control IDs are > 40006 and we should handle them all
        if (control > 40006)
        {
            ATLASSERT(false);
        }
    }

    return result;
}

void
CMyD3DApplication::recreate_mesh()
{
    m_mesh_nocolors = 0;
    m_mesh = 0;
    create_mesh();
}
```