Preamble to Chapter “WPF-3D”

You've now learned a little bit about how a 3D scene is projected to 2D so that we can render an image, and you know the basic facts about light, reflectance, sensors, and displays -- "facts" that will be substantially clarified in later chapters. The other required ingredient for an understanding of graphics is mathematics. We've found that students often understand mathematics better when they encounter it experimentally; the difference between "rotate this object about the origin, and then move it to the right" and "move this object to the right and then rotate about the origin" becomes obvious when you actually implement the two operations and compare the results. But making such experiments using 3D graphics requires either that you build your own graphics system, for which the preliminary mathematics is critical, or that you use something pre-made. Just as WPD2D provided us with a testbed for 2D experimentation, WPF3D gives us a tool in which to conduct 3D experiments. To use WPF3D effectively, you must be familiar with *its* model of light and reflectance and how objects are represented. This model is *not* based on physics directly, but nonetheless, because of the enormous adaptability of the human visual system, lets us make pictures that our brains perceive as showing a 3D scene. It *also* has the advantage of being widely used in other graphics libraries, despite being somewhat ill-defined; it's a model that researchers in graphics *have* to know, even if it's being rapidly superseded, because of its extensive use in early graphics research and commercial practice.

The following chapter teaches you this model, and the WPF3D system, by example. In it we construct a small 3D scene and then render a view of it, gradually increasing the complexity of the model as the chapter progresses. This entails describing the geometry of the scene, the lights in the scene, and the material from which the "objects" in the scene are made, notably how this material reflects light. With such a description, together with a description of how we wish to view the scene, WPF lets us create a 2D picture of the 3D scene. Once you understand how to do this, you can create your own scenes and lighting, and make your own pictures; you can also start to see how the non-physicality of the model leads to difficulties in making pictures. The desire to produce more realistic pictures motivates the extensive discussions of light, materials, reflectance, and how light moves around in a scene in the remainder of the book.
14. Introduction to WPF 3D

14.1 Overview of WPF

14.1.1 Goals

There are dozens of commonly used 3D graphics platforms, covering a wide variety of design goals: some are focused on image quality/realism regardless of performance cost (e.g., systems used to compute the frames for high-quality 3D animated films), others target real-time interactivity with more-or-less realistic simulation of physical properties (e.g., systems used for creating 3D virtual-reality environments or video games), and others make compromises on image quality in order to achieve reasonably-fast support across a wide variety of hardware platforms.

WPF’s primary goal is bringing 3D into the domain of interactive user interfaces, and as such has these requirements:

- to support the huge variety of hardware platforms supported by Windows
- to support dynamics for low-complexity scenes with a performance that is as close to “real-time” as possible on any hardware that meets basic Windows requirements
- to provide an approximation of illumination and reflection that’s sufficient to allow programmers to create recognizable 3D scenes, with the approximations being simple enough that such scenes can be rendered in real time

14.1.2 Approximating the Physics of the Interaction of Light with Objects

[Note to the reader: some of the motivational material in this section 14.1.2 may have to be rewritten to mesh with explanations already given in earlier chapters – in its current state this chapter is meant only to depend on the WPF-2D chapter.]

As described in the introduction, light (energy in the visible range of the electromagnetic spectrum) is contributed by both external (e.g., sunlight through a window) and internal (e.g., desk lamp) sources. An object’s surface hit by this energy absorbs some of it, and re-radiates some of it back into the scene; the amount being reflected is dependent upon characteristics of the surface’s material, the angle at which light is striking the surface, etc. Some materials absorb nearly all of the incident energy while others will reflect a considerable amount. For partially transparent surfaces, some light may be refracted as well. The energy reflected by surfaces adds to the light emitted directly by light sources.

Complexity #1: Determining the intensities and directions of all light that strikes a point on an object’s surface requires a global analysis of both the original light entering the scene directly from light sources as well as the “re-
radiated” light reflected by every surface in the scene. Algorithms for accurately determining this “light transport” are compute-intensive; they are discussed in great detail in Chapter _____.

Each type of material – wood, plastic, fabric, metal – has unique properties that determine how it absorbs, reflects, and refracts light. Some materials, such as human skin, exhibit sub-surface scattering of light which require even more complex reflection models.

**Complexity #2:** Determining how an object’s surface appears – and the nature/quantity/direction of the re-radiated light that affects other objects – requires formulations that depend upon the material

The physics of a material’s absorption/reflection/refraction can differ for different parts of the spectrum:

**Complexity #3:** A material’s appearance and reflectivity may depend on the wavelength of the incident light

For many surfaces, especially very “shiny” ones, knowing the nature of the light hitting a surface point is necessary but not sufficient for determining how it looks to the eye:

**Complexity #4:** An object’s appearance may be dependent upon the location of the eye (the viewpoint).

Let’s consider the light sources themselves. Sunlight can be modeled simply: the sun can be considered to produce unidirectional rays (by considering the sun to be essentially an infinite distance away from the scene). By contrast, artificial light sources create complexity:

**Complexity #5:** Lights in the scene or close to the scene have geometry themselves, often complex (e.g., the new spiral-like compact fluorescent lights), and the light they emit is distributed (often non-uniformly) across their geometry.

The complexities involved in accurately simulating the physics of light and its interaction with objects in the scene are so great that, even if the algorithms taking all of this into account were sufficiently developed, the amount of processing would be beyond the capabilities of even today’s greatest supercomputers.

Thus, computer graphics involves approximation; moreover, due to the large variation in goals (game applications are being “watched as they execute” and need to paint many frames per second, whereas movie-production applications have the luxury of running “offline” with hours or even days devoted to computing a single frame), a wide variety of approximation techniques have been developed.

The most “classic” techniques were developed decades ago, when the power of computing and graphics hardware power was a fraction of what is available today. These algorithms were developed – first purely in software and then in increasingly powerful GPUs – to meet two key goals:

- Minimizing processing and storage requirements
- Maximizing parallelism, especially in GPUs

Meeting these requirements made simulating the actual physics an impossible luxury, not only because of the processing demands of the calculations, but also because the global analysis needed to handle the recursive nature of inter-object reflection stymied attempts to maximize parallelization. Thus, the most successful approximation techniques developed in the past three decades – and dominant in GPU design until the late 1990s – are for the most part simplistic models that produce visually acceptable results with little or no basis in the real physics of light-object interaction.
Introduction to WPF 3D

A particular sequence of these algorithms, commonly called the **fixed-function 3D graphics pipeline**, involves rendering of triangular meshes, approximating both polyhedral objects and curved surfaces, using simple surface shading rules that ignore the problem of inter-object reflection. This collection of algorithms was first developed in the 70s in software and then implemented in increasingly more powerful GPU hardware starting in the 80s.

The fixed-function pipeline is accessed via software APIs of the type found in classic commodity 3D packages like OpenGL and Direct3D. WPF is one of the newer APIs providing access to the fixed function (FF) pipeline, and as we present WPF’s basic feature set throughout the rest of this chapter, we will provide a brief introduction to the classic FF approximation techniques, how well they “fool the eye”, what their limitations are, etc.

Note: While still important today particularly in real-time applications, fixed-function pipelines are rapidly being replaced\(^1\) by pipelines composed of programmable **shader** modules that allow applications to break free from the fixed algorithms of the classic pipeline – see Chapter ____ for more information.

### 14.1.3 High-level overview of WPF-3D

To meet its performance goals mentioned earlier, WPF-3D makes compromises in image quality and would be not be an appropriate platform for applications like 3D animated-film rendering. However, its straightforward approach to scene specification, and XAML’s excellent support for rapid prototyping without bulky edit/build/test cycles, makes it an excellent way to introduce the types of activities involved in 3D graphics. In chapters ____ , we will step back from this focus on WPF to present a contrast/comparison overview of the gamut of today’s popular 3D APIs, including those which allow (or require) deployment of customized shader modules.\(^2\)

To the novice user of WPF, the most striking evidence of the non-physical approximation in the system is the lack of units of measurement. Light is described by specifying an "intensity" between 0.0 and 1.0, but the precise meaning of intensity is not defined; the most we can really assume is that higher intensities correspond to greater perceived brightness. The mix of different wavelengths in the light isn’t specified by describing the light’s spectrum; instead, it’s described indirectly by a mix of three numbers, indicating how much light is in the "red," "green," and "blue" portions\(^3\) of the spectrum. Furthermore, these red-green-blue values are each specified by a one-byte unsigned integer, i.e., a number from 0 to 255, written in hexadecimal. (You may have seen such color specifications in image-editing programs, or in HTML/CSS webpage sources, already). A similar system is used for indicating the "color" of a material, as you’ll see. Finally, when we describe objects in the world, we’ll use triples of numbers, **without units**, to indicate position.

**EXERCISE:** Imagine building a small physical model of a room out of cardboard, and illuminating it with a 20W lamp in the ceiling. Now imagine building a second model of the same room, but about one third the size of the first, and illuminating it with a 20W lamp in the ceiling. Does the second model appear brighter? Dimmer? If the first model was

---

\(^1\) Clear evidence of this trend can be found in the latest versions of OpenGL and Direct3D, which are deprecating or eliminating their original FF-based APIs.

\(^2\) At press time, WPF-3D does not provide shader support, but future versions may indeed provide such support.

\(^3\) The reason for the quotation marks on the color names is explained in Chapter ______.
about 1 meter long, and the second is about 1 foot long, you certainly expect them to have different appearances.

WPF’s lack of units makes it impossible to determine which model you’re representing mathematically.

WPF’s 3D support is closely integrated with the 2D feature set described in chapter 2, and is accessed in the same way: XAML can be used to initialize content and do some kinds of basic dynamics, and procedural code can be used to respond to user interaction and perform more sophisticated kinds of dynamics.

The key part of the 2D/3D integration in WPF is the 2D Viewport control, which acts as a rectangular canvas on which 3D scenes are drawn by the rendering pipeline. To include a 3D drawing in a WPF user interface, simply place an instance of the Viewport class in your 2D scene, and control its shape and location in the same way you would control the geometry of a button or textbox or any other 2D shape.

A Viewport object is much like that of its 2D equivalent (the Canvas) in that it is nothing more than a blank slate until it is given content – a scene – to display. To specify a scene, you must:

- Create and position a set of geometric objects (“models”), and set the appearance attributes of each
- Place and configure one or more lights
- Place and configure a camera

Once you’ve placed a viewport in a visible 2D layout, and specified the 3D scene it is to display, the viewport immediately displays a rendering of the scene within its rectangular domain. There is no need for an explicit “take a photo now” action; the camera is always “on” as if it were a videocamera, not a still camera, and WPF automatically keeps the viewport’s 2D rendering in sync with the scene. Thus, dynamics are performed by using procedural code (or XAML animation elements) to modify the scene’s characteristics by, for example:

- Adding or removing objects,
- Transforming (e.g., scaling, rotating, or moving) existing objects,
- Changing the visual attributes of objects (e.g., modifying the opacity or changing the color), or
- Changing the positions or attributes of the camera or the lights.

### 14.2 Introducing Mesh and Lighting Specification

In this section, we will use XAML to build a 4-sided, solid-color pyramid, shown at right in a scene embellished with a sky background and a textured desert floor. In this section, we will focus on the construction and lighting of just the pyramid; we will address modeling the sky and desert in later sections of this chapter.

#### a. Planning the scene

The first step in any 3D project is done “offline”: a human or an automated process must plan the geometry of the scene – i.e. the geometries and initial positions/orientations of all objects, including the lights and camera.

WPF’s 3D coordinate system is right-handed, and we will assume our desert floor is coplanar with the XZ “ground plane” and contains the origin, as shown in the figure at right.
We will place our pyramid so its base is on the XZ “ground plane,” with its center at the origin (0,0,0) and its four corners located at (+/-50, 0, +/-50). Its height will be 75 units, so the apex is located at (0,75,0).

We've chosen coordinates that are integers between 0 and 100 because these are easy to read and write; we could have divided every number in this model by 1000, say, and have created an essentially identical shape.

We want the camera to be initially placed so it lies well outside the pyramid, but close enough to ensure the pyramid dominates the rendered image. So we will position the camera at (57, 41, 247) and “aim” it towards the pyramid’s center point.

b. Preparing a viewport for content

In order to be visible, the Viewport must live inside a WPF 2D structure like a window or dialog box. The XAML shown below creates a viewport and houses it in a 2D window of size 640x480:

```xml
<Page xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
      xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'
      Title='Pyramid'>
  <Page.Resources>
     <!-- Materials and meshes will be specified here. -->
  </Page.Resources>
  <Viewport3D x:Name='VP3D' Width='640' Height='480'>
     <!-- The entire 3D scene, including camera, lights, model, will be specified here. -->
  </Viewport3D>
</Page>
```

Note: Whereas the XAML presented in the 2D chapter was “simplified,” here we present XAML fragments exactly as they would appear in a working application; ellipses are used only to omit fragments already presented or described. Some of XAML’s “syntactic vinegar” will be obvious and may inspire questions. However, this chapter is not intended to be a XAML reference; our focus is on semantics, not syntax. For questions about tag names, attributes, and structure, consult references for any .NET versions 3+.

If you’re a “hands-on” learner and want to follow along as we construct this scene, we suggest you copy XAML from the online lab (module #1) XAML pane, rather than trying to copy XAML code from this chapter.

All objects including the perspective camera and lights are specified inside the Viewport tag. The basic template of a viewport and its content looks like this:

```xml
<Viewport3D ...
   Camera>
   <!-- camera characteristics are filled in below... -->
</Viewport3D.Camera>
<Viewport3D.Camera>
   <!-- The next element wraps around the scene’s content -->
   <ModelVisual3D>
      <ModelVisual3D.Content>
         <Model3DGroup>
            <!-- The lights and objects will be specified here. -->
         </Model3DGroup>
      </ModelVisual3D.Content>
   </ModelVisual3D>
</Viewport3D>
```

First, let’s fill in the details of the camera’s position and configuration:

---

4 Determining the numeric values that make a scene “look right” is often the result of trial and error; thus, scene design is greatly facilitated by interactive 3D development environments that offer instant feedback while a designer twiddles with placement/orientation of lights, camera, and objects.
The WPF camera is a geometric object, placed in the scene at a 3D point ("Position=") and oriented via two vectors:

- "LookDirection=" is set to point to the subject that should lie at the center of the rendered image.
- "UpDirection=" rotates the camera about the “look vector” to control what will constitute the “up” direction to the viewer. The value (0,1,0) simulates a stationary tripod on the desert sand set up to photograph the pyramids, but dynamic changes to this vector are essential for “point of view” games like driving/flying simulators, where changing x and z values would accurately simulate “banking” and “pitch”.

Additionally, two “clipping planes” can be specified to prevent anomalies that occur when objects are too close to the camera (“NearPlaneDistance=”) and to reduce computation expense by ignoring objects that are very distant (“FarPlaneDistance=”).

Next we light the scene, initially with white ambient light, a non-realistic, simplified form of lighting that applies the same amount of illumination on all surfaces regardless of their location or orientation. (We’ll supplement this with more realistic lighting, simulating sunlight, in subsection (d) below.)

Ambient light can be considered a very crude method for mimicking inter-object reflection. Additionally, it insures that every surface is illuminated to some degree (however minuscule) thus preventing unrealistic pure-black shading on surfaces facing away from the scene’s light sources. (Such surfaces would, in the “real world”, be subjected to at least some level of inter-object reflection, and thus would not appear pure black.)

We add a pure-white light specification as the first child of the Model3DGroup tag, specifying its color as an “RGB triplet” (a hex-formatted number, ranging from 0x00 to 0xFF, for each of the so-called color primaries Red, Green, and Blue – see Chapter ____ for more detail on color specification).

At this point, we recommend that you enter Lab #1 of the online e-book for this chapter, located at:  
http://www.sklardevelopment.com/graftext/ChapWPF3D/WPF3Ddemo.htm

We will refer to lab #1 throughout this section 14.2, in subsequent gray boxes interspersed like this one in the text.

c. Placing our first triangle

It is no coincidence that we have selected a pyramid as our first example object, since the triangular mesh is the only 3D primitive type supported by WPF. The first step in creating a 3D object is defining a resource object of type MeshGeometry3D by providing a list of 3D Positions (vertices) and a list of Triangles specified via one triplet (three integer indices into the Positions array) per triangle. The order of presentation of the Positions and the Triangles is unimportant for now, but will be discussed later.

In this case, we are specifying a mesh containing just one triangle, the first face of our pyramid. Here is the XAML mesh representation, along with a tabular view:

```xml
<MeshGeometry3D x:Key='RSRCmeshPyramid'
    Positions='0,75,0 -50,0,50 50,0,50'
    TriangleIndices='0 1 2' />
```

This XAML fragment appears in the resource section of the XAML; in this way, it is similar to the WPF-2D template resource.

A resource has no effect until it is used or instantiated. So the next step is to add the 3D object to the viewport’s scene by creating a XAML element of type GeometryModel3D, whose properties include at a minimum:

- The geometry specification, which will be a reference to the geometry resource we created above.

- The material specification, which is usually also a reference to a resource. The material describes the light-reflection properties the surface has; WPF’s materials model provides approximations of a variety of material types, as we shall see in section 14.5.

Let’s keep things simple for now, and create a simple solid-yellow material in the XAML’s resource section, giving it a keyname for later referencing:

```
<DiffuseMaterial x:Key='RSRCmaterialFront' Brush='#FFFF88'/>
```

We now are ready to create the element that will add this simple 1-triangle mesh to our scene. We place this XAML as a child of the Model3DGroup element:

```
<GeometryModel3D
    Geometry='{StaticResource RSRCmeshPyramid}'
    Material='{StaticResource RSRCmaterialFront}'/>
```

Our image of the “pyramid” model now appears as shown here...

In the online lab, select the “Single face” option in the model dropdown list. If you wish, click on the “XAML” tab to examine the source code generating the scene. Activate the turntable to rotate this triangle around the Y axis.

If we were to rotate this triangular face 180 degrees around the Y axis, so we can have a look at its “back side”, we obtain this puzzling image.

The triangle disappears because WPF’s simplified reflectance model (which ignores the effects of inter-object reflection) allows an optimization: objects you cannot see cannot affect the rendered image. If an object is "closed", like a cube or a sphere, any triangular faces that face *away* from the viewer will be obscured by other facets that face *towards* the viewer; such faces can be removed from further processing, thus on average reducing the rendering work by about half. If our pyramid were closed, we would never be able to detect the disappearance of this first face. WPF’s model of the world assumes that all faces are part of some closed model, and that this **backface culling** is therefore an allowable optimization.
To make the faces of a non-closed object like our proto-pyramid visible, we must additionally set the BackMaterial property to refer to another material that we will add to the resource section (a solid-red material, whose XAML specification is not shown here):

```xml
<GeometryModel3D
    Geometry='{StaticResource RSRCmeshPyramid}'
    Material='{StaticResource RSRCmaterialFront}'
    BackMaterial='{StaticResource RSRCmaterialBack'} />
```

As a result, the back face now is visible when the front faces away from the camera, as shown here...

In the online lab, check the box labeled “Use back material” and keep the pyramid spinning.

How does WPF determine which side of a triangle is the front side? The system infers this information based on the order in which the vertex indices are presented when the triangle is defined in the Triangles list. You must ensure that the vertices are presented in a counter-clockwise order from the point of view of someone facing the front side. The order of presentation is taken from the vertex triplets in the Triangles array, not from the vertex list in the Positions array. You never have to worry about the order in which 3D coordinates are presented in the Positions array.\(^5\)

For example, consider how we presented the vertices of the single triangle of our current model. In the Triangles array, the three indices into the Positions array were presented in this order: 0 to 1 to 2. Thus, the vertices are being presented in the order (0,75,0) to (-50,0,50) to (50,0,50), which is a counter-clockwise presentation of the vertices of the front face of our triangle, as shown here...

\[\text{d. Producing more realistic lighting}\]

You may have noticed that the lighting of the scene is rather static. Our triangular face’s color is not changing in any way even as it is rotating on the turntable, as is evident in these two snapshots.

The use of ambient lighting is responsible for this static appearance. Let’s replace this unrealistic lighting with a directional light approximating sunlight, i.e., a light source effectively infinitely far away, therefore emitting parallel light rays in a single direction\(^6\). With this type of lighting, the amount of light reflected by the triangle (and thus the “brightness” of its appearance) is a function of how directly the triangle “faces” the light source.

\[\text{\small\(^5\) Why counter-clockwise? As we will see in chapter \_
\_
\_\_, we can compute a surface normal for a planar surface by taking the cross-product of two consecutive edges lying in the plane; the counterclockwise direction makes the surface normal point outward in WPF’s right-handed coordinate system.}\]

\[\text{\small\(^6\) WPF supports two other kinds of light sources (spotlights and point lights) described in section 14.6.}\]
In our desert scene, we will position the simulated sun (emitting solid white light, full intensity) behind the viewer’s left shoulder by specifying the direction of its emission via the vector \((1, -1, -1)\):

\[
\text{<DirectionalLight Color='\#FFFFFF' Direction='1, -1, -1' />}
\]

Thus, the rays slant down towards the ground at a 45° angle. In fact, the rays are at a 45° angle relative to all three axes, and when projected onto the X-Z ground plane, they appear to travel from the (+X,+Z) quadrant to the (-X,-Z) quadrant.

Let’s examine intuitively how the surface’s reflected brightness varies based on how “directly” it faces the light source:

- **Facing light source** = Maximum light reflection
- **Perpendicular to light source** = No reflection
- **Intermediate** = Some fraction of the light is reflected.

The math behind this intuitive intensity “drop-off” is the **Lambert’s diffuse-reflection cosine law**, whose inputs are these characteristics of the surface and the light source:

\[ I_{dir} \quad \text{A measure of the directional light’s intensity at the point of contact with the surface} \]

\[ \theta \quad \text{The angle between the surface normal (n) and the lighting direction vector (l)} \]

Given these two inputs, the reflected intensity is calculated thusly:

\[ I_{dir} \cos \theta \]

The figure below demonstrates this equation’s operation for various values of \(\theta\) (i.e., different orientations of \(n\) relative to \(l\)): 
This equation computes “diffuse reflection” due to its assumption that the surface reflects light in all directions equally. In the above figure, the lengths of the dashed red rays depict the intensity of the reflection energy emitted at that ray’s particular angle. With diffuse reflection, that length is a constant for any given value of θ. The locus of the endpoints of the reflection rays for all possible angles is thus a perfect hemisphere in the case of diffuse reflection; in our 2D figure, of course, this envelope appears as semicircle.

Since diffuse reflected intensity is dependent only upon θ, it is maximal when θ is 0, and reaches zero as θ approaches 90°. The camera’s point of view is ignored by the calculation; that is, the intensity of the reflected light is independent of the angle from which the surface is viewed. The result is satisfactory for simulating diffuse materials such as rugs and “matte” paint, but is highly inaccurate for glossy materials such as metals and plastic. We’ll show the classic techniques for simulating shiny materials later, in section 14.5.

You may have noticed another assumption made here – the equation assumes that the cosine is the only attenuation factor – i.e., that the surface reflects ALL incoming light when θ = 90°. In reality, some amount of light is absorbed by the material; moreover, for most materials, this “efficiency of reflectivity” factor varies across the color spectrum. To account for material absorption characteristics, we’ll expand this simple equation into the full Blinn-Phong lighting model equation in section 14.5.

In the lab, activate the directional lighting by selecting “directional, over left shoulder” and enabling turntable rotation. Observe the dynamic nature of the lighting of the yellow front face; it may help to occasionally pause/resume the turntable’s motion. Observe the display of the value of θ and cos θ, and note how the yellow face approaches zero illumination as θ approaches and passes 90°.

An important thing to note about this scene’s new, more-realistic lighting is that the face is uniformly lit across its entire surface. That is, at any given moment as the model is being rotated, the very same intensity is being used in the rendering of the entire face. This uniform intensity is exactly what we would expect in this situation of a planar polygon, dressed in a solid color, and exposed only to ambient and/or directional lighting. Indeed, the surface normal vector does not vary at any point on the planar face, so the rendering engine is delivering exactly what we want for this single-triangle model.
14.3 Representing Faceted and Curved Surfaces with Triangle Meshes

The next step in our pyramid model’s construction is to add the other three faces; our goal is to obtain an image like that shown at right. This is a case in which the object being modeled is itself a triangular mesh; thus, the specification exactly corresponds to the target object.

However, an exact match between target object and its representation in the graphics system is actually quite rare. Survey the room around you and you’ll find that most objects have some curved surfaces, and indeed it’s rare to find an object that is purely faceted. In the vast majority of cases, a triangular mesh is used by the graphics system to approximate the target object – the mesh and the object do not correspond exactly.

For example, note that we can approximate a circular cone by increasing the number of facets in our pyramid. At a facet count of 16, we get a fairly good approximation of a cone (as seen at right), but it doesn’t really “fool the eye” to accept it as a curved surface.

Increasing the number of facets (e.g., to 64 sides, as shown at right) does help improve the effect, but the approximation is still apparent. The number of facets needed to make the approximation effectively disappear is large, and the storage/processing costs of using fine-grained meshes for all the curved surfaces in a complex scene would be prohibitive.

Moreover, that approach is not scalable – if the viewpoint approaches (“zooms in on”) the cone, the faceting will again become quite apparent.

You might want to visit the online lab’s module #2 (“Modeling Curved Surfaces”) to see the effect of more/less facets in the approximation mesh. You can zoom in/out by dragging the mouse (within the viewport) while holding down the right mouse button. Note how increasing the number of facets can only fool the eye at a distance – zooming in exposes the fraud easily.

The problem of producing acceptable images of curved surfaces was particularly urgent in the early days of computer graphics, when computer memory was measured in kilobytes and processors were three orders of magnitude slower than today’s. Finding a “trick” that would help fool the eye and “smooth out” the facets of a curve-approximation triangle mesh – at minimal processor- and memory cost – was an important research goal. In 1973, PhD student Henri Gouraud developed a linear interpolation shading rule\(^7\) that produced an acceptable result without any modification of the underlying mesh model. To appreciate the difference in “realism” between the faceted look and the look produced by Gouraud shading, compare the two renderings of the Utah teapot below.

\(^7\) We use the term shading rule to refer to an algorithm for determining the colors of interior pixels based on surrounding key points. This term should not be confused with the term “lighting model” that is used to describe algorithms, such as the Lambert equation, that compute the reflected light at key points.
Let’s first review how the colors of the surface’s interior pixels are determined in the faceted rendering (a horizontal slice of our cone approximation) shown at right. For simplicity’s sake, we’ll use a 2D diagram, in which a curve (shown in light blue) is approximated via 2D faces (the black line segments). Each face’s surface normal (green arrows) is calculated and considered to be uniform across the face. Lambert’s comparison of each normal vector with the light direction $\ell$ computes the corresponding color (the small colored circles), and that result is used to paint all the pixels uniformly for the entire face (depicted in the horizontal bars above each face).

**Gouraud shading** can produce a much better rendering (as shown at right) of the very same curve-approximation mesh, by allowing the color of interior pixels to vary across the face. Interpolation is an obvious way to create a “gradient” color spectrum as shown in the slice through our cone below; however, interpolation from A to B is only of interest if A and B differ! Thus, Gouraud created the artifice of a *vertex normal* to ensure there would be two distinct values to interpolate between. Of course, a vertex is a point of discontinuity, so how does one compute its normal?

In Gouraud’s approach, depicted at right in its 2D operation, each vertex normal (red arrows) is computed by averaging the normals (green arrows) of the two line segments it connects. Note that if the mesh is sufficiently fine-grained, it lies very close to the curved surface being modeled, and a computed vertex normal can be a very good approximation of the normal that would be calculated for
the actual curved surface at points corresponding to the vertex. This accuracy gives a good foundation for the next step: the lighting equation calculates each vertex’s innate color (depicted via the small colored circles) – since the vertex normals vary across the surface, each calculated color will differ from its neighbor to some degree. Finally, linear interpolation between the computed vertex colors determines the interior colors.

In 3D, the vertex normal is determined by the normals of the triangles that meet at that vertex. The figure at right demonstrates this, with a red vertex normal being computed via averaging the normals of its six adjacent triangles.

NOTE TO READER: THIS IS A PLACEHOLDER FOR A PROPER DIAGRAM WITH ACCURATE normals.

You might want to return to the curved-surface module of the online lab, and select “Shared vertices” vertex specification. Note the success of the interpolation even with a minimal number of facets. You will notice that, if the granularity is extremely low (e.g. 4 or 8) and/or the model is rotating, the “silhouette” of the cone – its bottom edge where it meets the ground – unfortunately continues to exhibit the mesh’s structure, reducing the success of the “trick”.

All fixed-function graphics pipelines provide both flat and Gouraud shading, and the application is responsible for mixing/matching these two approaches as appropriate for the target object. The choice is not an either/or proposition! Typical complex objects (e.g., the teapot where the spout joins the body, auto bodies, and airplane wings joining the fuselage) are a hybrid of both curved surfaces and seams/joins where a sharp “crease” is desired and we don’t want interpolation to hide that real discontinuity in the underlying surface. How does the application specify which type of shading is desired for a particular portion of the mesh?

In theory, the mesh should be independent of the rendering technique, and the choice of shading pipeline should be independent of the geometry specification.

However, in WPF’s design, the type of shading produced by the pipeline is determined by how the mesh is specified – in particular, by how each “common vertex” (a single point that provides a vertex for two or more adjoining triangles) is presented in the Positions/Triangles list.
To obtain Gouraud shading across a particular common vertex, place the common vertex in the Positions list once, and refer to that single position for each use of that vertex. For example, take our single triangle model (developed in the previous section) and let’s add a second face. The first face had its base running from V1 (-50,0,50) to V2 (50,0,50). The new face, which shares the same apex (V0), will have its base running from V2 (50,0,50) to V3 (50,0,-50).

Here is the mesh specification that lists each of the two common vertices (V0 and V2) just once in the Positions array. Note that vertices 0 and 2 are “shared” by both of the triangles.

For each vertex that is shared by more than one triangle, WPF will auto-calculate the vertex normal, use that normal to compute the per-vertex light, and perform interpolated shading between each of the three vertex colors using a simple weighted averaging calculation discussed in Section XXX.

Please return to online lab module #1. Choose the “Two faces, shared vertices” model, select directional lighting, and enable the “Show normals” option. You may wish to activate the turntable to observe the gradient shading under the influence of dynamic per-vertex colors.

To obtain flat shading across a particular common vertex, list that point once for each triangle for which it provides a vertex. Ensure the vertex’s index appears only once in the entire Triangles table. The example table shown at right shows the mesh specification that provides for faceted shading of the same two-face pyramid.

Now, choose the “Two faces, common vertices not shared” model and observe the effect of faceted shading.

The completion of the pyramid is an exercise for the reader. The other two faces should be added to the specification without any vertex sharing. Thus, the apex (0,75,0) should appear four times in the Positions list, and that list should have exactly 12 entries.

**XAMLpad Exercise 14-A:** Copy the XAML program (for the two-face pyramid model) into XamlPad. Make sure it works and then add the other two faces to the mesh specification. Remember: our goal is a sharp-edged rendering, so avoid vertex sharing. Our solution is available in the online lab by choosing the “full pyramid” model.

**XAMLpad Exercise 14-B:** Manually create the MeshGeometry3D element that specifies a unit cube (6-sided). Use XamlPad to test your work.
14.4 Surface texture in WPF

Any computer graphics professional confronted with the question “What was the single most effective reality-approximation trick in the history of rendering?” will scarcely skip a beat before exclaiming, “texture mapping!” When faced with the need to display a “rough” or color-varying material like gravel, brick, marble, wood or wood—or to create a background such as a grassy plain or vast desert or thick forest—trying to represent with polygons every detail of the fine-grained structure of the material is out of the question for commodity hardware and/or real-time applications. Consider the complexity of the mesh that would be required if we wanted our pyramid’s geometry to emulate the dimples and crannies of rough-hewn stone; our simple four-triangle mesh would balloon into a mesh of millions of triangles, and the processing requirements would explode as well.

Through the “trick” of texture mapping (the wrapping of a 3D surface with a 2D decal), complex materials (e.g. velvet, linen, asphalt) and complex scenes (e.g. farmland as seen from an airplane) can be simulated with no increase in mesh complexity (albeit at the cost of significant loss in the fidelity of the model).

This technique was applied in our pyramid scene for the stretch of desert sand and the realistic sky backdrop. Let’s examine our scene’s desert floor, which is modeled as a square (two coplanar right triangles) clothed in a material based on a texture pixmap as seen here...

“Texturing” a 3D surface corresponds to the physical process of covering an object’s surface with a stretchable sheet of decorated “contact paper.” Theoretically, to control the end result you must specify, for each point on the 3D surface, exactly which place on the contact paper should touch that point. Of course, it is impossible to explicitly specify this mapping for every point on the 3D surface, so in practice you specify this mapping for each vertex, and interpolation is used for the interior points of each triangular face.

To specify the “texture point” that should be mapped to a particular mesh vertex requires a coordinate system for referring to positions within the texture pixmap. By convention, instead of referring to exact integer pixel coordinates, points on the pixmap are described in an abstract floating-point “texture-coordinate system” whose two axes are conventionally labeled u and v, with coordinate values limited to the range 0 to 1. The origin is located at the upper-left corner of the pixmap.

\[ (0,0) \]
\[ u \text{ axis} \]
\[ (1,0) \]
\[ v \text{ axis} \]
\[ (0,1) \]
\[ (1,1) \]

8 Stored in any standard image format (e.g. JPEG, GIF), and typically generated by either a painting application or extraction from an actual digital photograph.
In XAML, the first step is registering the pixmap as a diffuse material in the resource dictionary. We have registered solid-color diffuse materials previously, but here we create what WPF calls an “image brush”:

```xaml
<DiffuseMaterial x:Key='RSRCtextureSand'>
    <DiffuseMaterial.Brush>
        <ImageBrush ImageSource='sand.gif' />  
    </DiffuseMaterial.Brush>
</DiffuseMaterial>
```

The next step is registering into the resource database the simple two-triangle mesh representing the ground, using the same technique as before but adding a new attribute to specify the corresponding texture coordinate for each vertex in the Positions array:

```xml
<MeshGeometry3D x:Key="RSRCdesertFloor" Positions=" 9999, 0, 9999
9999, 0, 9999
9999, 0, -9999
9999, 0, -9999"
TextureCoordinates="    0,0     1,0     1,1       0,1 ">
    TriangleIndices="0 1 2  2 3 0 "
</MeshGeometry3D>
```

Since this is a mapping from a square (the two coplanar triangles in the 3D model) to a square (the texture pixmap), we declare texture coordinates that are simply the corners of the unit-square TCS, as shown below:

![Texture Coordinate System](image)

With the material and geometry registered as resources, we are ready to instantiate the desert floor:

```xml
<GeometryModel3D
    Geometry='{StaticResource RSRCdesertFloor}'
    Material='{StaticResource RSRCtextureSand}'>
</GeometryModel3D>
```

The result is as shown here, in a snapshot taken from a helicopter high in the sky over the desert.

The result is not acceptable; there is some subtle variation in the color of the desert floor, but the color patches are huge (in comparison with the pyramid).
The problem is that our tiny 64x64-pixel sand decal has been stretched to cover the entire desert floor. The result looks nothing like sand even though our decal provides fairly good realism when viewed unscaled.

This effect is identical to that which we would get if we textured a model of a peacock’s tail by blowing up just one instance of a “peacock eye.” The correct approach is to create copies of the eye and “tile” (replicate) them across the tail.

Our failure to simulate desert sand here is a case of a reasonable texture image being applied to the model incorrectly. Implementing texturing requires choosing between the two mapping strategies: tiling and stretching.

### 14.4.1 TEXTURING VIA TILING

If the texture is being used to simulate a material with a consistent look and no obvious points of discontinuity (e.g. sand, asphalt, brick), the texture image is replicated as needed to cover the target surface. In this case, the texture is typically a small sample image (either synthetic or photographic) of the material, which has been designed especially to ensure adjacent tiles fit together seamlessly.

As an example, consider this texture image showing six rows of red brick.

Applying it to each face of a rectangular prism without tiling produces a realistic-looking image, but the number of brick rows is inconsistent with the concept of a tall brick fortress.

Tiling allows the number of apparent brick rows to be multiplied, producing the intended effect of a tall fortress instead of a low wall.

Consult the texture-mapping section of the online lab for details on how to enable and configure tile-based texturing in WPF.
**14.4.2 TEXTURING VIA STRETCHING**

If a texture is being used as a substitute for a highly complex model (e.g., a city as seen from above, or a cloudy sky), the texture image is often quite large (to provide sufficiently high resolution) and may be either photographic or original artwork (e.g., if being used to represent a landscape in a fantasy world). Most importantly, this kind of texture image is a “scene” that would look unnatural if tiled. The correct application of this kind of texture image is to stretch it to cover the mesh (which must have an appropriate size and shape to avoid undesired “warping” of the scene).

For example, in our desert scene, the sky that provides the scene’s backdrop...

... is modeled as a cylinder whose interior surface is stretch-textured with this actual sky photo.

Consult the texturing section of the online lab for more information on, and examples of, this kind of texture mapping.

---

**14.5 The WPF Lighting Engine**

In section 14.1.2, we very briefly introduced phenomena that must be considered in true physics-based illumination strategies; that discussion was partly meant to introduce the complexities of modeling real physics, and also to whet your appetite for forthcoming discussions on various rendering strategies, including realistic versions (called ray tracing) of the Durer renderer introduced in Chapter _____. But mostly it was intended to justify the computer graphics industry’s pervasive use of approximation strategies that are sometimes ad-hoc, not at all physics based, but which nonetheless produce acceptable results in real time with appropriate graphics hardware.

Section 14.2 presented the Gouraud shading technique — linear interpolation among illumination values calculated at the mesh’s vertices — used to approximate the amount of directional light that is reflected by the interior points of a mesh triangle. That strategy is just one component of the complete WPF “lighting engine” which we now describe in a bit more detail. This engine is based on codes of practice that have been dominant for the last few decades, especially in the fixed-function pipelines present in GPUs that were dominant until relatively recently, when programmable shader technology displaced the fixed function pipeline; this simple model is discussed in more detail in Chapter ____ under the topic of polygonal rendering/rasterization.
14.5.1. Color specification

The word "color" is used to describe multiple things: the spectral distribution of wavelengths in light, the amount of light of various wavelengths that a surface will reflect, and the perceptual sensation we experience upon seeing certain objects ("that banana is green!"). The confounding of these things – and the high status given to the third – is a serious matter, discussed in detail in Chapter ___. The standard approach in graphics APIs and drawing/painting applications – that is, representing color by means of “RGB triplets” (Red, Green, Blue) of numbers – is perhaps the grossest of the approximations made in computer graphics. You should regard it extremely skeptically, and correspondingly should not demand too much logical consistency from it.

WPF’s color specification – which includes information about hue (roughly which color in the spectrum), saturation (roughly how pure the color is, with pink a desaturated, more pastel-like version of pure red), and brightness – is performed for two purposes:

- for a light source, the application must specify the emitted color
- for the material draping a mesh surface, the application must specify its innate color, which can be either:
  - a solid color, or
  - a texture image that will be tiled or stretched to cover a surface

For both, WPF uses the aforementioned RGB triplet – a value ranging from 0 to 255 (0x00 to 0xFF) for each of three primaries – and the resulting color is obtained via addition of the components.

A software or 3D-model designer is not expected to generate RGB values from thin air; art/design applications and interactive programming environments provide “color picker” tools that facilitate determination of the RGB triplet that describes a target color. Inevitably some trial and error will be involved to get the color right. Furthermore, every piece of hardware will render the same digital RGB triplet slightly differently, as discussed in Chapter ___, so that the RGB color specification will always be an approximation to what the designer might have had in mind.

14.5.2: Light geometry

The two types of lights we have seen thus far (ambient and directional) are useful but decidedly unrealistic: they are not considered to be emanating from a specific point in the scene, and their brightness is uniform throughout the entire scene. They are thus mere approximations, to wit:

- Ambient lighting is an artificial construct that can be considered the crudest possible approximation to the highly variable inter-object reflection of light, but its primary purpose is to ensure a minimal amount of light hits all surfaces in the scene, and thus to eliminate completely-dark spots where models lose all surface characteristics and thus become uninterpretable by the viewer (the “black blob” problem).

- As described previously, directional lighting provides a fairly good approximation of sunlight.
WPF provides two additional light-source types that are more realistic in behavior; a light of these types is actually positioned in the scene itself and its brightness is **attenuated** – that is, its effect on a particular surface is dependent upon the surface’s distance from the light’s position, so that if two triangles are oriented in exactly the same way towards one of these light sources, the one further away will receive less illumination.

- **A point light** is positioned at a point in the 3D world and emanates rays equally in all directions. Specification parameters include the 3D position of the light’s origin, and the attenuation rate to be used.\(^9\) A light of this type can mimic the light thrown out by a naked bulb hanging from a ceiling without any shade or baffles. This type is useful but should still be considered an approximation, since real physical light sources have volume and surface area, and thus do not emit light from a single point. It is an interesting problem in itself to model the geometry of a light source and to describe the distribution of light it emits in each direction (see Chapter ____)

- **A spotlight** is similar but it simulates a theatre spotlight in that it spreads light uniformly but in a cone-shaped volume. Specification parameters include the 3D position, the core direction of the light’s domain of effect, the attenuation rate, and the characteristics of the cone volume that is centered on the core direction and that limits the light’s effect on the scene.

**Note:** The online lab includes a section that lets you work with a point-light source so you can become familiar with geometric light sources.

### 14.5.3: Reflectivity

In our modeling of desert scenes throughout this chapter, we have applied material atop our meshes by specifying either a solid color or a texture image. However, there is more to a material than its color; if you’ve ever shopped for wall paint, you know that you must choose both the color (“sunshine yellow”) and the finish (“flat,” “shiny,” “metallic”). This “finish type” is a specification of how the material reflects light.

The physics of how light is reflected from a surface is extremely complex, so here again, conventional fixed-function platforms ignore or at best crudely approximate the actual underlying physics and instead use the classic approximation strategy called the **Blinn-Phong lighting model** that yields a variety of reflectivity simulation.\(^10\)

In much the same way that color is specified by adding together three components, in the Phong lighting model a material’s reflectivity characteristics are specified by configuring three “primary reflectivity types” (ambient, diffuse, and specular) which are then added together to produce the final appearance, as shown in this figure:

---

\(^9\) The application can choose from a variety of attenuation types: constant, linear, and quadratic. For more information, see Chapter ____.

\(^10\) This lighting model was invented at the dawn of raster graphics and rendering research in the 1970s initially by Ph.D. student Phong and then slightly modified by Blinn, and has been remarkably long-lived. Graphics researchers, as well as material scientists, now have far more complex, physically-based models of the surface characteristics based on a microscopic examination of real surfaces such as wood, metal and plastics, as well as animal fur and human hair (see Chapter ____). These more complex models are used, for example, in high-end special effects for animations.
The independent nature of the diffuse and specular material types may at first be counterintuitive. What does it mean to say a solid black bowling ball has a “blue specular color”? As the figure above shows, these independent specifications allows one to achieve “iridescent” effects and is a very inexpensive way to simulate the highly complex reflectivity of real materials whose light reflection properties are determined by details of the material’s geometry at the atomic level. (Consider, for example, the complexity of the human dermis, with its many layers of different types of tissue and vessels, each with its own absorption, reflection, refraction and scattering effects.) Phong’s crude approximation simplifies this enormous complexity by simply stating that many materials reflect light partially in a diffuse way and partially in a specular way, and thus allows a mesh to be draped with independent but “coincident” materials whose distinct reflections are composited to form the result.

The Phong lighting model is fully described in Chapter ____; here, we provide a high-level overview of the Phong calculation equation, whose inputs include:

<table>
<thead>
<tr>
<th>Input (O)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>O</strong></td>
<td>Innate color of the object’s material at a specific point on the surface. For a solid-color material, this is a constant across the entire mesh, typically represented by an RGB triplet. For a material painted by an image brush, the texture mapping algorithm (introduced earlier) computes this value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input (k)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>k</strong></td>
<td>The material’s efficiency at reflecting light, which can vary across the spectrum of wavelengths and is thus often represented via an RGB triplet of variables: $k_r, k_g, k_b$. For example, one would use $k_r = 0$ for a material that absorbs all light in the red part of the spectrum.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input (i)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>i</strong></td>
<td>The intensity (including color information) of a light source as measured at the surface point. For directional light, this is a constant based on the light’s specification. For point/spot lights, this varies based on the light’s attenuation over distance. Here again, the typical representation is an RGB triplet.</td>
</tr>
</tbody>
</table>
Because these same input types are present across the three reflection types, subscripts \( a, d, \text{dir}, \) and \( s \) are used to identify the reflectivity/lighting types involved. For example, \( O_d \) is used to represent the innate color of the “diffuse” component of the surface’s material, and \( i_{\text{dir}} \) represents the intensity of a particular directional light on a particular surface point. Note that when we talk about the intensity of a light source measured at a point on the surface we do that without any concern for whether the light source can reach that individual point because it might be blocked by another element of the scene.

Here is an overview of the Blinn-Phong calculation that is detailed in the following sections:

\[
I = i_a k_a O_d + \sum_{\text{lights}} i_{\text{dir}} \left( k_d O_d (\cos \theta) + k_s O_s (\cos \delta)^n \right)
\]

As we present the various components of the Phong model, we will ask you to use the lighting/materials section of the online laboratory.

(a) Ambient Reflectivity

Ambient light’s effect on a surface is constant regardless of the surface’s orientation, so the ambient-reflectivity equation is extremely simple and devoid of geometric information. In the Phong model, there is no “ambient innate color”, so the material’s diffuse color \( O_d \) is used. The ambient component is a simple computation:

\[
i_a k_a O_d
\]

Please launch the lab and enter section #5 (Materials and Reflectivity). The revolving sphere is a solid-color surface “patched” with occasional texture-mapped images; the lab allows you to control \( O_d \) for the solid-color background but not for the image patches.

Note the decorative night-sky background emits its own light, as explained later in subsection (d), and is thus not affected by any of the lab’s controls.

Make sure directional lighting is set to “None” so you can experiment with just the ambient equation for now. First, try different values for the ambient light color/intensity \( i_a \) and the sphere’s innate solid color \( O_d \), noting that the former affects all parts of the sphere including the image patches.

Next, play with the sphere’s ambient-reflectivity efficiency \( k_a \). What happens if the ambient light is forced to be pure red, but the efficiency factor \( k_a \) is set up to not reflect any red wavelengths? Does this kind of situation occur in nature, or is it purely an artificial effect one can achieve via “extreme” use of Phong’s model?
(b) Diffuse Reflectivity

Adding directional lights allows diffuse reflectivity to take effect, computed via an additional term that is evaluated once for each directional light:

\[
\sum_{\text{lights}} i_{\text{dir}} k_d O_d (\cos \theta)
\]

Note the use of \(\cos(\theta)\), Lambert’s attenuation factor described in section 14.2(d), which ensures that a light source has less impact the more its direction varies from a surface’s normal.

The factors that vary for each directional light source are \(i_{\text{dir}}\) (the light’s intensity) and \(L\) (the light’s direction).

Please return to the lab, and turn on one white directional light (at either half or full intensity). Play with the amount of ambient and directional lighting, and note the accumulative effect that comes from the Phong model’s addition of the two separately computed intensities.

You can also choose to turn on three directional lights, one for each primary color, for an even larger set of variations for your experimentation.

Note that if the ambient light is sufficiently powerful, the effect of the directional light can be erased altogether, a result of the “clamping” that the lighting model performs when the computed intensity exceeds a hardwired maximum threshold. In fact, one of the uses for the \(k_d\) efficiency factors is to reduce a surface’s “sensitivity” to light to reduce the amount of clamping occurring on a particular surface without having to perform a more “global” modification like reducing a light’s intensity.

(c) Specular Reflectivity

Of course, directional lighting also enables specular highlights (as you see on mirrors illuminated by a flashlight or on a metal object in bright light). Because certain materials may reflect specularly quite differently from the way they reflect diffusely (and as with diffuse reflection, wavelength-dependent) we approximate this phenomenon by having separate specular reflection object colors as specified by \(O_{Sub s}\). Here again the component’s contribution is the sum of a computed intensity for each directional light in the scene:

\[
\sum_{\text{lights}} i_{\text{dir}} k_s O_{Sub s}(\cos \delta)^n
\]

This computation also includes a cosine-based attenuation factor, but here the angle \(\delta\) is intended to ensure that the specular highlight is most powerful when the view angle (surface-to-eye vector \(e\)) is very close to the angle of reflection (vector \(r\)) of the incoming light (vector \(\ell\)), as shown in this table:
How should we compute δ? There are two popular algorithms that meet the above requirements. The most obvious technique, and indeed the original one proposed by Phong, is to simply define it to be the angle between vector e and vector r.

However, a variant known as the **Blinn-Phong** model has become extremely popular for two reasons: it is computationally more efficient and empirically shown to produce more accurate highlights. As described in section 1.18.2 and depicted in the figure at right, this variant computes δ (highlighted in red) by comparing the surface normal n and a “halfway” vector h (the average of the vectors e and -e).

The exponent n — called the **specular power** — allows the application to control the speed of the attenuation, i.e. how “sharp” the highlight is. Values for highly shiny surfaces are typically around 100, providing a very sharp peak; a mathematically perfect mirror would have an infinite n allowing the beam to be seen from exactly one direction, while a polished apple might have an n of around 10 and thus a larger area of measurable specular contribution.

Please return to the lab, and perform these steps:

- Turn on one white directional light at full intensity, and make sure the ambient light is weak or off.
- Give the sphere a “bowling ball” appearance by giving it a very dark, or solid black, diffuse color.
- Turn on specular highlighting by assigning the sphere a specular color other than the default black one. For example, try to simulate a shiny black ball that reflects in an iridescent blue.
- Play with various values for the specular power, to observe how the size of the specular highlight changes. Note the anomaly that occurs when n reaches zero.

You might also wish to observe specular reflectivity in the existence of colored directional lighting. The interactions between the lights' colors, the surface's innate diffuse color, the surface's specular color, and the two efficiency filters (k_d for diffuse and k_s for specular) provide for a wide variety of interesting and sometimes hard-to-predict effects.
(d) WPF’s “bonus” reflectivity type: Emissive

WPF adds a fourth, highly artificial type of reflectivity – not present in the classic Phong model – that allows a surface to “reflect” light that is not actually present externally, achieving a “glow” effect. This effect is most useful when combined with the use of a texture image, to model a nighttime cityscape background or a star-filled sky (as in the online lab). This new type of reflectivity is independent of geometry and is not subject to any efficiency attenuation factor, so it is nothing more than the specification of a material’s innate color (either a solid color or a texture image) and is thus represented in WPF’s illumination computation via the very simple term:

$$O_e$$

Please return to the lab, turn off all lighting (directional and ambient), and use the emissive color control knobs (in the “Advanced” section) to force the sphere to glow. You may then wish to restore ambient and/or directional lighting to see the result of the accumulation of WPF’s complete set of material types.

14.5.4: Variable Opacity

WPF provides support for translucent/transparent surfaces via the addition of an extra opacity parameter (0=fully transparent, 1=fully opaque) that you can optionally add to the specification of the innate diffuse color ($O_d$). See Chapter _____ for information on opacity control in rendering pipelines.

Please return to the lab and experiment with the opacity knob (in the “Advanced” section) that affects the sphere’s solid-color background. Try to give the sphere the appearance of a glass ball bearing opaque image patches. Note that the diffuse opacity setting does not affect specular highlighting; you can use that fact to enhance the “glass” effect of the sphere in the presence of directional lighting.

--FIN--