Chapter 3

Materials

3.1 Introduction

The shape of objects is only one part of what determines their appearance. Very important are also their reflection properties, like the diffuse reflection colour, the colour and intensity of specular reflection and their respective dependence on the angles for illumination and viewing. Objects can also be transparent and have various other properties that determine how the incident light interacts with the surface and is reflected or transmitted toward the viewer. Taken together, the surface properties of a 3-D graphics object is often called a *material*. Unlike real world materials, computer graphics materials are often pure 2-D properties defined on the surface. This is sufficient for opaque objects, which are the most common kind of objects in most virtual scenes.

3.2 Reflection models

Early applications of computer graphics used very crude models for how light is reflected from objects. In some circumstances, like in real time rendering where time is short and processing power is limited, these models are still used today. In this chapter, we will present two of these simple but useful models. A more proper treatment of surface reflection will be saved for the chapter on illumination and rendering.

3.2.1 The Phong model

One model that is very popular in textbooks is the *Phong reflection model*, where the reflection is split into three terms: one *ambient* term, one *diffuse* term and one *specular* term:

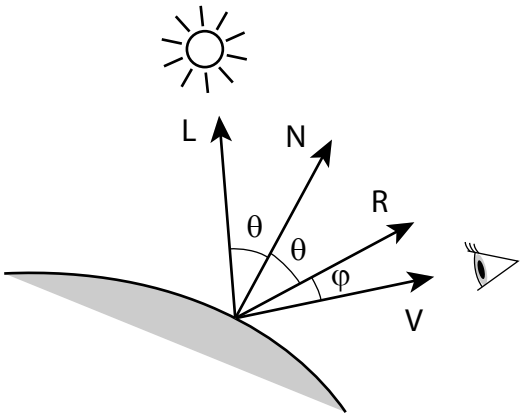

Figure 3.1: The vectors in the Phong reflection model

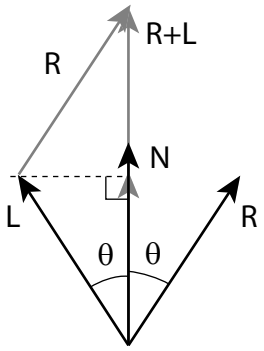

Figure 3.2: The reflection vector R

$$I = I_a k_a + I_d k_d (\mathbf{N} \cdot \mathbf{L}) + I_s k_s (\mathbf{R} \cdot \mathbf{V})^n \tag{3.1}$$

The vector entities in Equation 3.1 are explained in Figure 3.1. **L** is a vector pointing from the surface point towards the light source, **V** is a vector from the point towards the viewer, and **N** is the surface normal. All vectors are normalized.

The vector **R** is the "reflection vector", the direction a perfect specular reflection would have. It can be computed from L and N by observing that L and R are at opposite sides of N at the same angle θ, such that $\mathbf{L} \cdot \mathbf{N} = \mathbf{R} \cdot \mathbf{N} = \cos\theta$, and their sum is parallel to N:

$$\begin{aligned} \mathbf{L} + \mathbf{R} &= 2(\mathbf{L} \cdot \mathbf{N})\mathbf{N} \\ \mathbf{R} &= 2(\mathbf{L} \cdot \mathbf{N})\mathbf{N} - \mathbf{L} \end{aligned} \tag{3.2}$$

The parameters I_a, I_d and I_s describe the illumination intensities for the three kinds of lighting in the model: ambient, diffuse and specular light. The

corresponding k_a, k_d and k_s are the surface reflectances for the respective effects.

The parameters k_d, the diffuse reflectance, is what we refer to as the "colour" of a surface in everyday jargon. The specular reflection, k_s, is also a colour, but except for metallic surfaces, the specular reflection usually has a neutral color (white or grey).

Figure 3.3 shows the effect of these parameters. For the 4×4 spheres to the left, k_d is varied from almost zero to almost one from left to right, and k_s is varied from zero to one from top to bottom. The ambient reflection k_a is kept at zero, so the lower right part of the spheres, where the light does not reach, is completely black. An ambient term would have lightened that up a bit, but it would have washed out the contrast for all other points on the surface as well. In general, ambient illumination should be used very sparingly in modern applications of computer graphics, and only for real time rendering. It looks flat, uninteresting and bad.

The rightmost four spheres in Figure 3.3 show the effect of the parameter n. From top to bottom, n is changed from 2 to around 50. In some sense of the word, the parameter can be thought of as "shininess", how polished the surface is, but the analogy is not very good. A polished surface in the real world does present a smaller highlight, but at the same time it should be brighter, and that is controlled separately by the parameter k_s. Furthermore, a really shiny surface should show a mirror reflection of the light source, whereas the Phong model assumes that all lights are ideal points.

Realism

The Phong model is a *local illumination model*, where a surface has information about light sources, but no knowledge of other surfaces in the scene. This means that reflections from other objects cannot be accurately simulated. Very shiny surfaces will be severely lacking in realism because they reflect only the light sources and not their environment. Just as importantly, indirect diffuse light, which is a prominent contribution to the overall illumination in a general real life scene, has to be ignored or approximated in some crude fashion.

On the plus side, the diffuse reflection term in the Phong model is a pretty good model of what actually happens when light illuminates a diffuse surface and is reflected from it. The cosine falloff with angle that comes from the scalar product $\mathbf{N} \cdot \mathbf{L}$ is geometrically correct, and many real world surfaces have a diffuse reflection that behaves more or less like the Phong model.

The Phong specular term, however, is a crude approximation. The highlight appears in the right place, but that's about it. The ambient term, finally, is so far from reality that its use is strongly discouraged except when

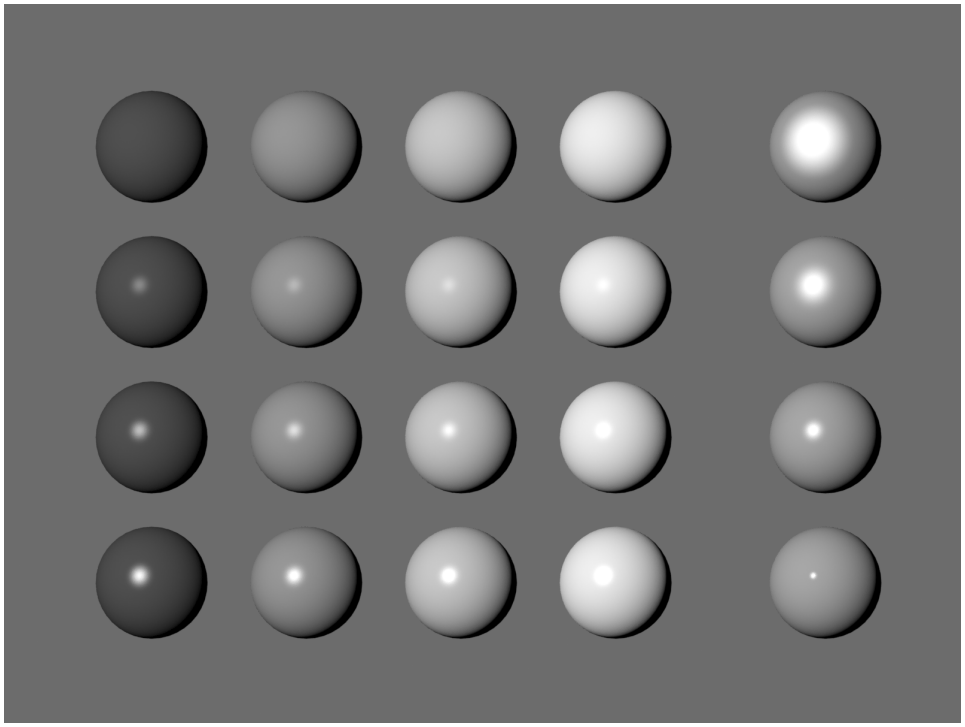

Figure 3.3: The parameters of the Phong model. Left: varying k_d and k_s. Right: varying n.

there are no other options. The very notion of an "ambient light" that shines with equal intensity from all directions and reaches everywhere without being influenced at all by any objects is, quite frankly, preposterous.

Excluding negative values

Equation 3.1 is a version you see in most presentations, and it is useful to get an overview of the model, but if you want to actually implement the model in software, you need to take several additional details into consideration.

First, the scalar product $\mathbf{N} \cdot \mathbf{L}$ will be negative if $\theta > \pi/2$, but there is no such thing as negative light, and the diffuse refection term must never be negative. The real world interpretation of $\theta > \pi/2$ is that the light comes from behind the surface, which means that there is no light directly incident on the surface. In these cases, the term should be set to zero, and a correct form for the diffuse term would be:

$$I_d k_d \max(\mathbf{N} \cdot \mathbf{L}, 0) \tag{3.3}$$

Second, similarly as for the diffuse term, the scalar product $\mathbf{R} \cdot \mathbf{V}$ should be replaced with 0 if it is negative, and the entire specular term should be excluded if $\mathbf{N} \cdot \mathbf{L}$ is negative. (Note that these two conditions are not the same. Both are required for all cases to be treated correctly.)

$$\begin{array}{ll} I_s k_s \max(\mathbf{R} \cdot \mathbf{V}, 0)^n, & \text{when } \mathbf{L} \cdot \mathbf{N} > 0 \\ 0, & \text{otherwise} \end{array} \tag{3.4}$$

Colour and range

For simplicity, the I and k parameters of the reflection model are often presented as if they were scalars, but both the illumination and the reflection are in fact colours. In computer graphics, it is customary to represent colours with RGB vectors, and a multiplication of an RGB light intensity with an RGB reflectance factor should be performed as a component-wise multiplication:

$$I_d k_d = \begin{bmatrix} I_{d,R} \\ I_{d,G} \\ I_{d,B} \end{bmatrix} \begin{bmatrix} k_{d,R} \\ k_{d,G} \\ k_{d,B} \end{bmatrix} = \begin{bmatrix} I_{d,R} k_{d,R} \\ I_{d,G} k_{d,G} \\ I_{d,B} k_{d,B} \end{bmatrix} \tag{3.5}$$

The k parameters represent a surface reflection and should be kept in the range $0 \leq k \leq 1$. Real world illumination is positive ($I \geq 0$) but not bounded upwards – you can always add more light to a scene. Traditionally, the intensity of light sources in computer graphics has been kept in the approximate range $0 \leq I \leq 1$, in order not to generate pixel values that are too large for screen presentation. Monitors are still predominantly 8-bit and have a limited maximum intensity. In recent years, however, rendering

has shifted from using 8-bit pixel formats (integers in the range 0 to 255) to floating point formats with better precision and without strong bounds. Even real time rendering now uses a selection of dim and strong lights spanning a very wide range of intensities, and an *exposure control* is performed after rendering to map the pixel values to an 8-bit representation for display.

Multiple lights

The equation, even with all the details added above, still concerns only one light source. Fortunately, it's not difficult to extend it to multiple light sources. Light in the real world behaves in an additive manner: a light source can only add to the light in the scene, and its influence will be independent on any other light sources. The Phong model, in the simplified form from Equation 3.1, can be extended to a sum over any number of light sources. Each light has its own position and intensity, so the parameters I_d, I_s, $\mathbf{L}$ and by association $\mathbf{R}$ will be different for each term in the sum. The ambient light, if used at all, is usually a single, separate light source that is kept outside of the sum:

$$I = I_a k_a + \sum_{i=0}^{n} (I_{d,i} k_d (\mathbf{N} \cdot \mathbf{L}_i) + I_{s,i} k_s (\mathbf{R}_i \cdot \mathbf{V})^n) \tag{3.6}$$

Self-emission

Some objects emit light on their own, not just reflect it. Although not formally a part of the original Phong model, it is useful and common to add a *self-emission* term I_e to the model – a contribution to the intensity which is independent on any light sources. The same visual effect could be achieved by abusing the ambient term, but self-emission is a property of the *object* and should be kept independent of the ambient light source. Note that a self-emission from a surface does not make it into a light source. In a local illumination model, the self-emission contributes only to the appearance of the object with the emission. It does not have any impact on nearby objects or the scene in general.

$$I = I_e + I_a k_a + \sum_{i=0}^{n} (I_{d,i} k_d (\mathbf{N} \cdot \mathbf{L}_i) + I_{s,i} k_s (\mathbf{R}_i \cdot \mathbf{V})^n) \tag{3.7}$$

Light types and decay

The light sources described by the original formulation of the Phong model are *directional lights* which are assumed to illuminate every object from the same direction regardless of its position. Light sources that are far away behave more or less in this manner, and this type of light is useful despite

its simplicity. However, we also need to be able to model lights that are placed near the scene, or even in the scene. Instead of describing the light as a vector, $\mathbf{L}$, we can set its position $\mathbf{p}_L$ and compute the light direction as the vector from the current surface point $\mathbf{p}$ to the light: $L = \mathbf{p}_L - \mathbf{p}$. This is another common type of light, a *point light*.

The length of the computed vector to a point light is the distance to the light source, which makes it possible to compute a *decay with distance*. Real world light sources have an intensity that is proportional to the inverse of the square of the distance, $I \propto 1/|\mathbf{L}|^2$. This creates a strong dependence on the absolute distance to the light, and in computer graphics the decay is often eliminated or reduced to a less pronounced decay with distance, like an inverse linear decay $I \propto 1/|\mathbf{L}|$ or some polynomial with both linear, quadratic and possibly constant terms, $I \propto 1/(1 + a|\mathbf{L}| + b|\mathbf{L}|^2)$.

Another important type of light source is a *spotlight*, a light that illuminates objects only within a cone with its apex at the light source. Such light sources need to be described by both a position and a direction, and some additional parameters to describe the radial decay of light within the illumination cone. A more in-depth treatment of light types will be presented in the next chapter.

This, finally, concludes our presentation of the Phong reflection model. Most textbooks fail to mention many of the details above.

3.2.2 The Blinn-Phong model

Another, very similar reflection model is the *Blinn-Phong reflection model*. It's similar to the Phong model, and just as crude, but the specular refection is computed differently, such that for reflections at grazing angles, the shape of the specular highlight is more similar to that of real world surfaces. Because of this, the Blinn-Phong model is used a lot more frequently than the original Phong model.

Blinn's modification to Phong's model is that instead of computing the reflection vector $\mathbf{R}$, a *halfway vector* $\mathbf{H}$ is computed which is the normalized average of $\mathbf{L}$ and $\mathbf{V}$, half-way between the two:

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \tag{3.8}$$

And then $\mathbf{R} \cdot \mathbf{V}$ is replaced with $\mathbf{N} \cdot \mathbf{H}$ in the specular term:

$$\begin{array}{ll} I_s k_s \max(\mathbf{N} \cdot \mathbf{H}, 0)^n, & \text{when } \mathbf{N} \cdot \mathbf{L} > 0 \\ 0, & \text{otherwise} \end{array} \tag{3.9}$$

The angle between $\mathbf{N}$ and $\mathbf{H}$ is zero in the same situation as when the angle between $\mathbf{R}$ and $\mathbf{V}$ is zero, but the angle is different, and the exponent n needs to be changed to give the same appearance of shininess as the Phong

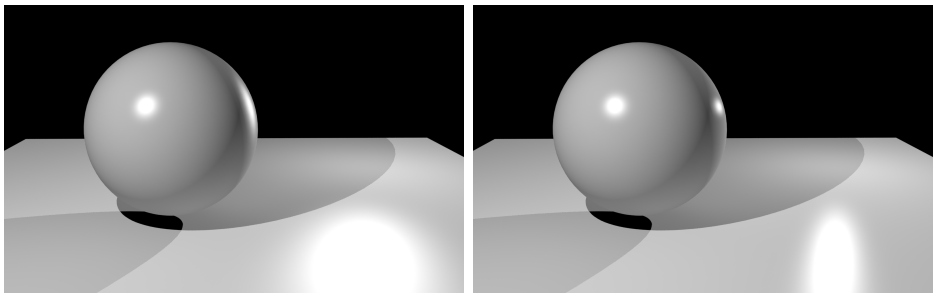

Figure 3.4: The Phong model (left) versus the Blinn-Phong model (right).

model. Setting n four times as high as for the Phong model preserves the size of highlights facing straight towards the viewer.

A visual comparison between the Phong and the Blinn-Phong model is in Figure 3.4. In the Blinn-Phong model, the shapes of highlights at grazing angles are a closer match to reality in most situations.

3.3 Texture mapping

The world around us is full of surfaces with a pattern of some sort. Natural objects and manufactured objects alike tend to have not a single colour, but a colour that varies across the surface. Quite obviously, this has to be modelled by computer graphics if the synthetic scenes are to bear any resemblance to reality. The process of putting patterns to surfaces in 3-D graphics is called *texture mapping*.

3.3.1 Texture coordinates

In general terms, a texture is a function $F(s,t)$ defined over the surface, where s and t are *texture coordinates* mapping the pattern onto the surface. Some surfaces, like parametric surfaces, have an obvious mapping that comes at no extra cost, while other surfaces need to have a mapping assigned to them. Texture coordinates are assigned to each vertex, and interpolated across the polygons during rendering to have every point on the surface correspond to a position in the texture. The texture image is then used to determine the surface colour (or some other property) at a certain point.

Parametric mapping

A natural mapping for the sphere in Equation 2.13 would simply be the surface parameters u and v: $(s,t) = (u,v)$, as shown in Figure 3.5. This *latitude-longitude mapping* has some flaws, because the texture becomes progressively more contracted in the s direction when t approaches its extremes at 0 and 1. At the two poles there is even a pinching of the (s,t) plane

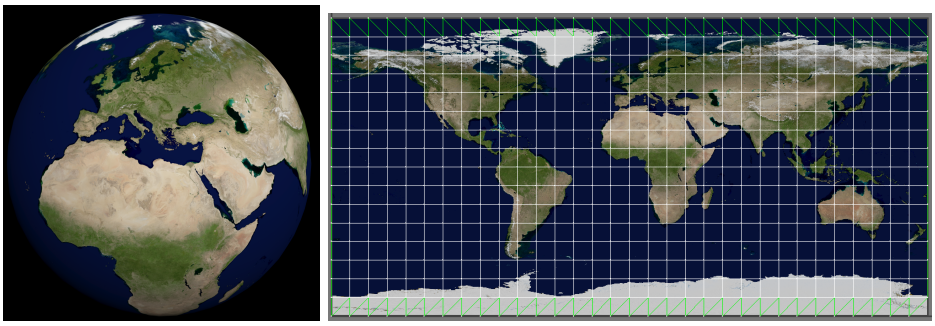

Figure 3.5: A texture mapped sphere and its texture coordinate map.

such that the lines $t = 0$ and $t = 1$ map to a single point. Note that the polygons around the poles in this sphere are triangles, not quadrilaterals, and the mapping doesn't even cover every pixel in the image. Nevertheless, the latitude-longitude mapping is useful as long as the texture images are created to account for the varying area scale.

Explicit mapping

In many cases, no obvious simple mapping can be found, and the (s,t) mapping has to be explicitly designed and specified as per-vertex texture coordinates. Depending on the situation, the mapping can be performed more or less automatically or manually, or by using a combination of automated tools and manual tweaks. Manual mapping requires considerable effort, but such mappings can be of much higher quality. An example of a manual mapping for a model with a low polygon count is shown in Figure 3.6. This particular mapping was carefully designed to make the best possible use of the available pixels, and extra care was taken to make it easy to draw and edit the texture image using ordinary image editing tools. More pixels were spent on the face, because that area contains more important visual features. Mirror symmetry was also used to save texture space: the left and right arms and legs map to the same area in the texture image.

Creating a good texture coordinate map is not an easy task, but it is often required. 3-D models which lack texture coordinates are not terribly useful. A lot of effort in 3-D modeling is spent on texture mapping, both by artists and by engineers, and for good reason.

3.3.2 Mapped properties

Any surface property may be varied by a texture, not only its colour. Clever and well considered use of texture maps is a very useful tool in computer graphics.

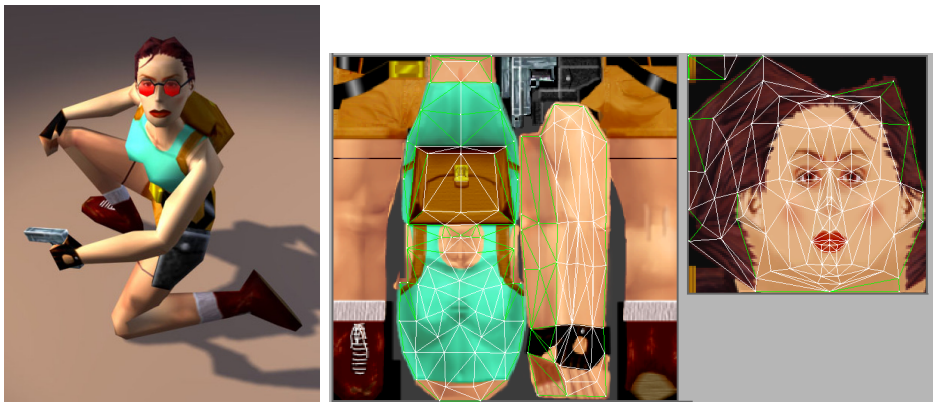

Figure 3.6: A texture mapped character and its texture coordinate maps.

Diffuse and specular colour

Taking the Phong reflection model as an example, the surface colour is k_d, but there is also a specular colour k_s which could be varied. In reality, varying levels of specular reflection could be due to wet spots on the surface, or areas where the surface is worn or dirty. If a texture map is used to tell where there is land and water on a planet, like in Figure 3.5, the water would need to be shiny and the dry land would need to be matte. A suitable specular intensity map would be black where there is land, and white where there is water.

Opacity

A surface property that hasn't been described above is *opacity*, which is the opposite of *transparency*. An opaque surface (opacity = 1) blocks light and hides anything behind it, but a surface with opacity less than 1 lets some light from the objects behind it show through. A surface with zero opacity is completely transparent and doesn't show up at all in a rendering. A completely invisible object is not very useful, but things get considerably more interesting if the surface opacity is modulated by a texture map, such that only some parts of the surface are hidden. An opacity map on a simple rectangle can be used to model a thin, flat object with an arbitrary outline and with holes in it, like a leaf, a sheet of torn paper or a piece of cloth. This comes at a very low cost compared to what it would have taken to model the same level of detail using polygons.

Surface normal

A very prominent property of a surface is its normal direction. We are already playing tricks with the normal by interpolating it across polygons

to make faceted mesh objects appear smooth, and there is nothing to stop us from manipulating the normal even further and use a texture map to create fake bumps and crinkles on the surface. This is called *bump mapping*. It was invented by Jim Blinn in 1978 and has remained an important tool in computer graphics ever since. A bump map is a very efficient way of creating the appearance of geometric detail without actually making the model more complicated. An extension of the method is *displacement mapping*, where a texture map is used to actually move the surface and make it truly rough, but that requires a lot more effort during rendering.

In both bump mapping and displacement mapping, the texture map describes the variation in surface height, and the new normal is calculated by computing the in-plane differentials of the map. A modern variation on bump mapping is *normal mapping*. Normal mapping instead stores the normal direction for each point on the surface directly as a 3-element vector, which saves some effort during rendering. A bump map is a single channel image which shows a sort of topographical height map of the surface roughness, and that concept is easy to understand and quite straightforward to create for an artist. For real time rendering, though, normal maps are often preferred because they involve less computations at render time. Normal maps can be computed from bump maps in an off-line preprocessing step.

Reflections

An application of texture mapping which is not strictly related to a surface property is *reflection mapping*, sometimes called *environment mapping*. A local reflection model has no knowledge of other objects around the point that is being rendered, but reflective surfaces like polished metal need to show a mirror image of their environment. A surprisingly successful way to fake this is to use a texture map of the environment. The way it is done is to reflect the view vector in the surface in the same manner as the light vector is reflected in the Phong model, and use the reflected view direction as a lookup index into a 360-degree panorama of the environment. Common mappings that cover all directions are latitude-longitude spherical maps and *cube maps*, where six square images map to the six faces of a cube. The reflection map can be based on a photo or computed as a rendering with the camera at the center of the reflection mapped object. If the scene is moving, the reflection map can even be recomputed for each frame to match a real reflection quite closely. However, it is surprisingly easy to get away with using a static reflection map for an animated scene. The map of the environment doesn't even have to be very similar to the actual scene. It is usually enough if the reflection has more or less the same colours and contrast and doesn't show, say, a reflection of trees against a clear blue sky when you are indoors.

3.3.3 Image-based textures

The texture function $F(s,t)$ must of course be specified somehow. The most common representation is *image-based textures*, using an ordinary sampled digital image with pixels. RGB pixels that are meant to represent colour can be used also for normal maps. Most image file formats can't store negative pixel values, but instead of directly storing the three components (N_x, N_y, N_z) of a normal, you can store $(N_x + 1, N_y + 1, N_z + 1)/2$. Many image file formats have an RGBA variant with a fourth "alpha" component, which is very suitable for storing an opacity map or a specular map, but such maps can also be stored as separate grey-scale images. For real time rendering with modern shader-capable hardware, it is common practice to save texture bandwidth by storing several different mapped properties in different colour channels in an RGB or RGBA image.

Care should be taken when using compressed image file formats to store textures that are not RGB images. JPEG and other destructive compression algorithms can give unpredictable and undesirable results. Non-destructive compression like PNG is to be preferred.

Image-based textures are digital images, and share both their advantages and disadvantages. The advantages are that they are easy to create and can be edited with commonly available software, and they are completely general in that they can describe any imaginable pattern. Disadvantages are that they have a limited resolution, and they require memory for their storage. Memory is cheap and abundant these days, even though it's not free and infinite. The limited resolution, however, is a serious issue, in particular for real time applications where you don't have complete control over where the camera is going. Close-up views of textured surfaces in games tend to look blurry or pixelated, which breaks immersion and looks bad.

Naively, you might think you only need to reconstruct the image value at one point for a texture lookup. The point (s,t) is typically not at the exact center of a pixel, or *texel* as they are often called, so some interpolation between neighboring pixels is required to reconstruct the value at (s,t). However, the problem is quite a bit more complicated than that. Because objects in a 3-D scene can be viewed from any angle and any distance, one pixel in the rendered image might stretch across several texels in the texture image, and the square boundary of a screen pixel can become an elongated and distorted shape when projected to the texel space. This makes proper texture sampling a very difficult problem, in fact a combination of problems, and we won't go into all the details here.

One mechanism in texture sampling that is definitely worth mentioning, though, is *mipmapping*. (The term MIP was originally short for a Latin phrase, *multum in parvo*, meaning "several in a small place", but nobody seems to care about that. The term "mipmap" is now a name rather than an acronym.) Mipmapping is a good solution to the very common problem of

texture minification: when a textured surface is viewed from afar, the texels are considerably smaller than a screen pixel, and a correct determination of the pixel value would require taking an average over some area in the texture. If this is not done properly, the texture will look bad and "sparkle" in animations because of a strong *undersampling*: some texels will be skipped, and which ones are skipped will depend on the view in an unpredictable manner. A mipmap contains pre-filtered versions of a texture image, where you store not only the original image, but also a sequence of successively smaller versions of it. The simplest kind of mipmap contains the original image and images of half that size, one quarter of it, and so on. The smallest image is a single pixel. The entire stack of images, sometimes referred to as an *image pyramid*, requires only 4/3 more storage than the highest resolution image alone, and yields a significant increase in quality. When a mipmap is sampled during rendering, you pick an image in the stack with a texel resolution that most closely matches the pixel resolution of the image. This gets rid of a lot of of the objectionable artifacts from undersampling.

3.3.4 Procedural textures

The texture function $F(s,t)$ doesn't have to be sampled and stored as a digital, pixel-based image. In some cases, it is reasonably convenient to instead specify it as a mathematical function that is computed by a short program snippet every time you need it. This method is called *procedural texturing*, and it has advantages as well as disadvantages.

In procedural texturing, a value for $F(s,t)$ is computed for every pixel for every frame, and thrown away after use. This might seem wasteful, and sometimes it is, but there are quite a few circumstances where you need to conserve memory bandwidth, so storing the texture image in memory is not always the best idea. In today's computers, the memory bus is slow compared to the internal clock frequency of the CPU or GPU, and the massive parallelism in modern GPUs makes it a difficult task to handle memory accesses from many processing elements at once. Procedural patterns, on the other hand, are described by short programs and require very little data, so spending some work on computing the pattern every time you need it can sometimes be the smarter option.

One clear advantage of procedural textures is that their resolution is not limited: they are *resolution independent*. They can be computed at any required resolution, even in extreme close-ups and distant views.

Another advantage of procedural patterns is that they can be changed at a moment's notice by changing a parameter to a function. Changing an image based texture requires opening the image in an image editing program, editing it and saving it, a process that could take a lot of time and effort even for small changes.

A less obvious feature of procedural textures is that they don't desper-

ately need a 2-D mapping. A mathematical function may be defined over a 3-D domain. By specifying the texture function as $F(s,t,p)$ instead if $F(s,t)$, 3-D object coordinates can sometimes be used as texture coordinates. For objects of irregular shape, this can be a strong advantage, and it corresponds well to a real world situation where an object is made from a material with a patterned internal structure, like wood or stone. Storing 3-D functions as sampled data is not impossible, but it requires a lot of data, and the resolution will be limited. With some care, you can even design procedural patterns that are 3-D functions that vary over time, making them 4-D functions. Storing 4-D functions as sampled data is prohibitively expensive for most purposes, but for procedural patterns, it's just a matter of how you define your texture function.

Creating procedural textures can be difficult, though. Instead of editing a digital image, a very common process known to many people, you need to create a small program that computes the pattern, and programmers with a proper understanding of texture patterns are hard to find, certainly a lot harder than digital image editing artists. Furthermore, not every pattern is suitable for a procedural description. Finding a function that creates a pattern is never easy, but it might be particularly hard or even impossible for some patterns. A digital image is very general and can describe any pattern, while only some patterns are suitable for procedural texturing.

Finally, the computational workload of a procedural texture can sometimes be prohibitive. A memory access has a fixed, low complexity, while a procedural pattern can take a lot of work to compute – possibly more work the closer you get to the surface. This makes it difficult to motivate procedural texturing for real time applications. It's not impossible, though. Modern GPU development has made it perfectly possible to consider using some of the GPU power for procedural texturing. You just need to be aware that it takes extra effort, effort that could be spent on other kinds of processing for the scene.

3.3.5 Composite textures

Most modern 3-D graphics packages have an entire separate section dedicated to the creation and editing of materials for surfaces. Such editors include options for assigning different reflection models to surfaces and to assign texture maps to various properties of the surface, but they also contain a layer-based editing framework to create texture maps that are built from several smaller parts, much like layers in an image editing program. It is not uncommon for a material in 3D graphics to contain dozens of individual texture maps that are combined in different ways to create the final surface pattern. This is true also for real time rendering. So-called *multitexturing* has long been the norm for real time rendering, and for a typical 3-D graphics scene, several textures were used to render one output pixel.

In a composite texture, it is of course perfectly possible to use both procedural maps and image based maps in any combination. Using both to their respective advantages and picking the right tool for the job is an important consideration when creating materials.