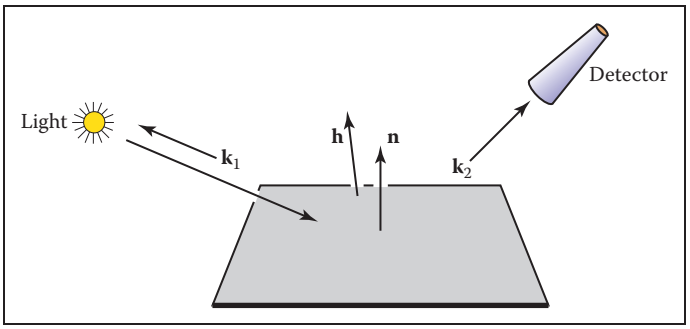

Figure 24.6. Geometry of reflection. Note that $\mathbf{k}_1$, $\mathbf{k}_2$, and $\mathbf{h}$ share a plane, which usually does not include $\mathbf{n}$.

A BRDF with these properties is a Fresnel-weighted, Phong-style cosine lobe model that is anisotropic.

We again decompose the BRDF into a specular component and a diffuse component (Figure 24.6). Accordingly, we write our BRDF as the classical sum of two parts:

$$\rho(\mathbf{k}_1, \mathbf{k}_2) = \rho_s(\mathbf{k}_1, \mathbf{k}_2) + \rho_d(\mathbf{k}_1, \mathbf{k}_2), \tag{24.4}$$

where the first term accounts for the specular reflection (this will be presented in the next section). While it is possible to use the Lambertian BRDF for the diffuse term $\rho_d(\mathbf{k}_1, \mathbf{k}_2)$ in our model, we will discuss a better solution in Section 24.5.2 and how to implement the model in Section 24.5.3. Readers who just want to implement the model should skip to that section.

24.5.1 Anisotropic Specular BRDF

To model the specular behavior, we use a Phong-style specular lobe but make this lobe anisotropic and incorporate Fresnel behavior while attempting to preserve the simplicity of the initial mode. This BRDF is

$$\rho(\mathbf{k}_1, \mathbf{k}_2) = \frac{\sqrt{(n_u+1)(n_v+1)}}{8\pi} \frac{(\mathbf{n}\cdot\mathbf{h})^{n_u \cos^2\phi + n_v \sin^2\phi}}{(\mathbf{h}\cdot\mathbf{k}_i)\max(\cos\theta_i, \cos\theta_o))} F(\mathbf{k}_i \cdot \mathbf{h}) . \tag{24.5}$$

Again we use Schlick's approximation to the Fresnel equation:

$$F(\mathbf{k}_i \cdot \mathbf{h}) = R_s + (1 - R_s)(1 - (\mathbf{k}_i \cdot \mathbf{h}))^5, \tag{24.6}$$

where R_s is the material's reflectance for the normal incidence. Because $\mathbf{k}_i \cdot \mathbf{h} = \mathbf{k}_o \cdot \mathbf{h}$, this form is reciprocal. We have an empirical model whose terms are

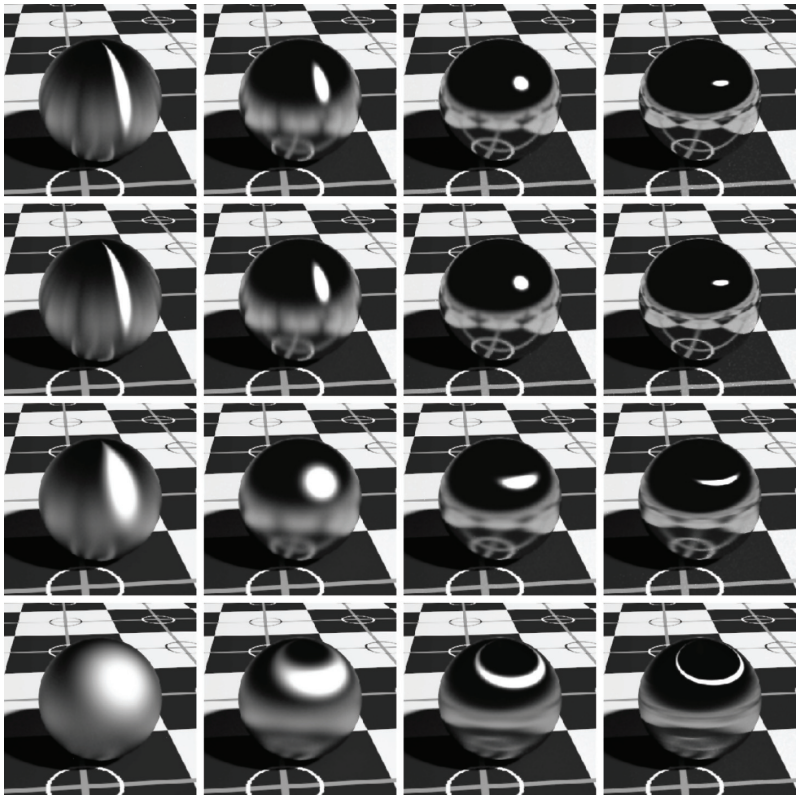

Figure 24.7. Metallic spheres for exponents 10, 100, 1000, and 10,000 increasing both left to right and top to bottom.

chosen to enforce energy conservation and reciprocity. A full rationalization for the terms is given in the paper by Ashikhmin, listed in the chapter notes.

The specular BRDF of Equation (24.5) is useful for representing metallic surfaces where the diffuse component of reflection is very small. Figure 24.7 shows a set of metal spheres on a texture-mapped Lambertian plane. As the values of parameters n_u and n_v change, the appearance of the spheres shift from rough metal to almost perfect mirror, and from highly anisotropic to the more familiar Phong-like behavior.

24.5.2 Diffuse Term for the Anisotropic Phong Model

It is possible to use a Lambertian BRDF together with the anisotropic specular term; this is done for most models, but it does not necessarily conserve energy. A

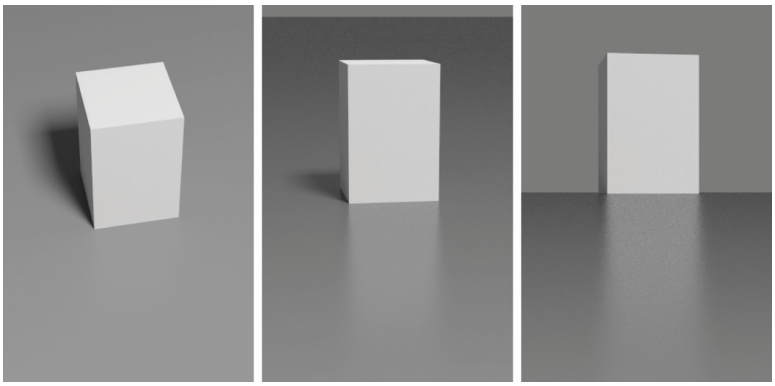

Figure 24.8. Three views for $n_u = n_v = 400$ and a diffuse substrate. Note the change in intensity of the specular reflection.

better approach is a simple angle-dependent form of the diffuse component which accounts for the fact that the amount of energy available for diffuse scattering varies due to the dependence of the specular term's total reflectance on the incident angle. In particular, diffuse color of a surface disappears near the grazing angle, because the total specular reflectance is close to one. This well-known effect cannot be reproduced with a Lambertian diffuse term and is therefore missed by most reflection models.

Following a similar approach to the coupled model, we can find a form of the diffuse term that is compatible with the anisotropic Phong lobe:

$$\rho_d(\mathbf{k}_1, \mathbf{k}_2) = \frac{28R_d}{23\pi}(1 - R_s)\left(1 - \left(1 - \frac{\cos\theta_i}{2}\right)^5\right)\left(1 - \left(1 - \frac{\cos\theta_o}{2}\right)^5\right). \tag{24.7}$$

Here R_d is the diffuse reflectance for normal incidence, and R_s is the Phong lobe coefficient. An example using this model is shown in Figure 24.8.

24.5.3 Implementing the Model

Recall that the BRDF is a combination of diffuse and specular components:

$$\rho(\mathbf{k}_1, \mathbf{k}_2) = \rho_s(\mathbf{k}_1, \mathbf{k}_2) + \rho_d(\mathbf{k}_1, \mathbf{k}_2). \tag{24.8}$$

The diffuse component is given in Equation (24.7); the specular component is given in Equation (24.5). It is not necessary to call trigonometric functions to

compute the exponent, so the specular BRDF can be written:

$$\rho(\mathbf{k}_1, \mathbf{k}_2) = \frac{\sqrt{(n_u+1)(n_v+1)}}{8\pi}(\mathbf{n}\cdot\mathbf{h})^{\frac{(n_u(\mathbf{h}\cdot\mathbf{u})^2+n_v(\mathbf{h}\cdot\mathbf{v})^2)/(1-(\mathbf{hn})^2)}{(\mathbf{h}\cdot\mathbf{k}_i)\max(\cos\theta_i,\cos\theta_o)}} F(\mathbf{k}_i\cdot\mathbf{h}). \tag{24.9}$$

In a Monte Carlo setting, we are interested in the following problem: given $\mathbf{k}_1$, generate samples of $\mathbf{k}_2$ with a distribution whose shape is similar to the cosine-weighted BRDF. Note that greatly undersampling a large value of the integrand is a serious error, while greatly oversampling a small value is acceptable in practice. The reader can verify that the densities suggested below have this property.

A suitable way to construct a pdf for sampling is to consider the distribution of half vectors that would give rise to our BRDF. Such a function is

$$p_h(\mathbf{h}) = \frac{\sqrt{(n_u+1)(n_v+1)}}{2\pi}(\mathbf{nh})^{n_u\cos^2\phi+n_v\sin^2\phi}, \tag{24.10}$$

where the constants are chosen to ensure it is a valid pdf.

We can just use the probability density function $p_h(\mathbf{h})$ of Equation (24.10) to generate a random $\mathbf{h}$. However, to evaluate the rendering equation, we need both a reflected vector $\mathbf{k}_o$ and a probability density function $p(\mathbf{k}_o)$. It is important to note that if you generate $\mathbf{h}$ according to $p_h(\mathbf{h})$ and then transform to the resulting $\mathbf{k}_o$:

$$\mathbf{k}_o = -\mathbf{k}_i + 2(\mathbf{k}_i\cdot\mathbf{h})\mathbf{h}, \tag{24.11}$$

the density of the resulting $\mathbf{k}_o$ is **not** $p_h(\mathbf{k}_o)$. This is because of the difference in measures in $\mathbf{h}$ and $\mathbf{k}_o$. So the actual density $p(\mathbf{k}_o)$ is

$$p(\mathbf{k}_o) = \frac{p_h(\mathbf{h})}{4(\mathbf{k}_i\mathbf{h})}. \tag{24.12}$$

Note that in an implementation where the BRDF is known to be this model, the estimate of the rendering equation is quite simple as many terms cancel out.

It is possible to generate an $\mathbf{h}$ vector whose corresponding vector $\mathbf{k}_o$ will point inside the surface, i.e., $\cos\theta_o < 0$. The weight of such a sample should be set to zero. This situation corresponds to the specular lobe going below the horizon and is the main source of energy loss in the model. Clearly, this problem becomes progressively less severe as n_u, n_v become larger.

The only thing left now is to describe how to generate $\mathbf{h}$ vectors with the pdf of Equation (24.10). We will start by generating $\mathbf{h}$ with its spherical angles in the range $(\theta,\phi) \in [0, \frac{\pi}{2}] \times [0, \frac{\pi}{2}]$. Note that this is only the first quadrant of the hemisphere. Given two random numbers (ξ_1, ξ_2) uniformly distributed in $[0, 1]$, we can choose

$$\phi = \arctan\left(\sqrt{\frac{n_u+1}{n_v+1}}\tan\left(\frac{\pi\xi_1}{2}\right)\right), \tag{24.13}$$

and then use this value of ϕ to obtain θ according to

$$\cos\theta = (1-\xi_2)^{1/(n_u\cos^2\phi + n_v\sin^2\phi + 1)}. \tag{24.14}$$

To sample the entire hemisphere, we use the standard manipulation where ξ_1 is mapped to one of four possible functions depending on whether it is in $[0, 0.25)$, $[0.25, 0.5)$, $[0.5, 0.75)$, or $[0.75, 1.0)$. For example, for $\xi_1 \in [0.25, 0.5)$, find $\phi(1 - 4(0.5 - \xi_1))$ via Equation (24.13), and then "flip" it about the $\phi = \pi/2$ axis. This ensures full coverage and stratification.

For the diffuse term, use a simpler approach and generate samples according to a cosine distribution. This is sufficiently close to the complete diffuse BRDF to substantially reduce variance of the Monte Carlo estimation.

Frequently Asked Questions

• My images look too smooth, even with a complex BRDF. What am I doing wrong?

BRDFs only capture subpixel detail that is too small to be resolved by the eye. Most real surfaces also have some small variations, such as the wrinkles in skin, that can be seen. If you want true realism, some sort of texture or displacement map is needed.

• How do I integrate the BRDF with texture mapping?

Texture mapping can be used to control any parameter on a surface. So any kinds of colors or control parameters used by a BRDF should be programmable.

• I have very pretty code except for my material class. What am I doing wrong?

You are probably doing nothing wrong. Material classes tend to be the ugly thing in everybody's programs. If you find a nice way to deal with it, please let me know! My own code uses a shader architecture (Hanrahan & Lawson, 1990) which makes the material include much of the rendering algorithm.

Notes

There are many BRDF models described in the literature, and only a few of them have been described here. Others include (Cook & Torrance, 1982; He

et al., 1992; G. J. Ward, 1992; Oren & Nayar, 1994; Schlick, 1994a; Lafortune, Foo, Torrance, & Greenberg, 1997; Stam, 1999; Ashikhmin, Premože, & Shirley, 2000; Ershov, Kolchin, & Myszkowski, 2001; Matusik, Pfister, Brand, & McMillan, 2003; Lawrence, Rusinkiewicz, & Ramamoorthi, 2004; Stark, Arvo, & Smits, 2005). The desired characteristics of BRDF models is discussed in *Making Shaders More Physically Plausible* (R. R. Lewis, 1994).

Exercises

1. Suppose that instead of the Lambertian BRDF we used a BRDF of the form $C \cos^a \theta_i$. What must C be to conserve energy?

2. The BRDF in Exercise 1 is not reciprocal. Can you modify it to be reciprocal?

3. Something like a highway sign is a *retroreflector*. This means that the BRDF is large when $\mathbf{k}_i$ and $\mathbf{k}_o$ are near each other. Make a model inspired by the Phong model that captures retroreflection behavior while being reciprocal and conserving energy.

Naty Hoffman

25

Computer Graphics in Games

Of all the applications of computer graphics, computer and video games attract perhaps the most attention. The graphics methods selected for a given game have a profound effect, not only on the game engine code, but also on the art asset creation, and even sometimes on the *gameplay*, or core game mechanics.

Although game graphics rely on the material in all of the preceding chapters, two chapters are particularly germane. Games need to make highly efficient use of graphics hardware, so an understanding of the material in Chapter 17 is important.

In this chapter, I will detail the specific considerations that apply to graphics in game development, from the platforms on which games run to the game production process.

25.1 Platforms

Here, I use the term *platform* to refer to a specific combination of hardware, operating system, and API (application programming interface) for which a game is designed. Games run on a large variety of platforms, ranging from virtual machines used for browser-based games to dedicated game consoles using specialized hardware and APIs.

In the past, it was common for games to be designed for a single platform. The increasing cost of game development has made this rare; *multiplatform* game development is now the norm. The incremental increase in development cost to support multiple platforms is more than repaid by a potential doubling or tripling of the customer base.

Some platforms are quite loosely defined. For example, when developing a game for the Windows PC platform, the developer must account for a very large variety of possible hardware configurations. Games are even expected to run (and run well) on PC configurations that did not exist when the game was developed! This is only possible due to the abstractions afforded by the APIs defining the Windows platform.

One way in which developers account for wide variance in graphics performance is by *scaling*—adjusting graphics quality in response to system capabilities. This can ensure reasonable performance on low-end systems, while still achieving competitive visuals on high-performance systems. This adjustment is sometimes done automatically by profiling the system performance, but more often this control is left in the hands of the user, who can best judge his personal preferences for quality versus speed. Display resolution is easiest to adjust, followed by antialiasing quality. It is also fairly common to offer several quality levels for visual effects such as shadows and motion blur, including the option of turning the effect off entirely.

Differences in graphics performance can be so large that some machines may not run the game at a playable frame rate, even with the lowest quality settings; for this reason PC game developers publish minimum and recommended machine specifications for each game.

As platforms, game consoles are strictly defined. When developing a game for, e.g., Nintendo's Wii console, the developer knows exactly what hardware the game will run on. If the platform's hardware implementation is changed (often done to reduce manufacturing costs), the console manufacturer must ensure that the new implementation behaves *exactly* like the previous one, including timing and performance. This is not to say that the console developer's task is easy; console APIs tend to be much less abstract and closer to the underlying hardware. This gives console development its own set of difficulties. In some sense, multiplatform development (which commonly includes at least two different console platforms and often Windows as well) is the hardest of all, since the multiplatform game developer has neither the assurance of a fixed platform or the convenience of a single high-level API.

Browser-based *virtual machines* such as Adobe Flash are an interesting class of game platforms. Although such virtual machines run on a wide class of hard-

ware from personal computers to mobile phones, the high degree of abstraction provided by the virtual machine results in a stable and unified development platform. The relative ease of development for these platforms and the huge pool of potential customers makes them increasingly attractive to game developers. However, these platforms are defined by the lowest common denominator of the supported hardware, and virtual machines have lower performance than native code on any given platform. For these reasons, such platforms are best suited to games with modest graphics requirements.

Platforms can also be characterized by their openness to development, which is a business or legal distinction rather than a technical one. For example, Windows is open in the sense that development tools are widely available, and there are no gatekeepers controlling access to the marketplace of Windows games. Apple's iPhone is a somewhat more restricted platform in that all applications need to pass a certification process and certain classes of applications are banned outright. Consoles are the most restrictive game platforms, where access to the development tools is tightly controlled. This is opening up somewhat with the introduction of online console game marketplaces, which tend to be more open. A particularly interesting example is Microsoft's Xbox LIVE Community Games service, where the development tools are freely available and the "gatekeeping" is performed primarily by peer review. Games distributed through this service must use a virtual machine platform provided by Microsoft for security reasons.

The game platform determines many elements of the game experience. For example, PC gamers use keyboard and mouse, while console gamers use specialized game controllers. Many console games support multiple players on the same console, either sharing a screen or providing a window for each player. Due to the difficulty of sharing keyboard and mouse, this type of play is not found on PC. A handheld game system will have a different control scheme than a touch-screen phone, etc.

Although game platforms vary widely, some common trends can be discerned. Most platforms have multiple processing cores, divided between general-purpose (CPU) and graphics-specific (GPU). Performance gains over time are due mostly to increases in core count; gains in individual core performance are modest. As GPU cores grow in generality, the lines between GPU and CPU cores are increasingly blurred. Storage capacity tends to increase at a slower rate than processing power, and communication bandwidth (between cores as well as between each core and storage) grows at a slower pace still.

25.2 Limited Resources

One of the primary challenges of game graphics is the need to manage multiple pools of limited resources. Each platform imposes its own constraints on hardware resources such as processing time, storage, and memory bandwidth. At a higher level, development resources also need to be managed; there is a fixed-size team of programmers, artists, and game designers with limited time to complete the game, hopefully without working *too* much overtime! This needs to be taken into account when deciding which graphics techniques to adopt.

25.2.1 Processing Time

Early game developers only had to worry about budgeting a single processor. Current game platforms contain multiple CPU and GPU cores. These processors need to be carefully synchronized to avoid deadlocks or excessive stalls.

Since the time consumed by a single rendering command is highly variable, graphics processors are decoupled from the rest of the system via a *command buffer*. This buffer acts as a queue; commands are deposited on one end and the GPU reads rendering commands from the other. Increasing the size of this buffer decreases the chances of GPU starvation. It is fairly common for games to buffer an entire frame's worth of rendering commands before sending them to the GPU; this guarantees that GPU starvation does not occur. However, this approach requires reserving enough storage space for two full frame's worth of commands (the GPU works on one, while the CPU deposits commands in the other). It also increases the latency between the user's input and the display, which can be problematic for fast-paced games.

Processing budgets are determined by the *frame rate*, which is the frequency at which the frame buffer is refreshed with new renderings of the scene. On fixed platforms (such as consoles), the frame rate experienced by the user is essentially the same one seen by the game developer, so fairly strict frame–rate limits can be imposed. Most games target a frame rate of 30 frames per second (fps); in games where response latency is especially important, the target is often 60 fps. On highly variable platforms (such as PCs), the frame-rate budgets are (by necessity) defined more loosely.

The required frame rate gives the graphics programmer a fixed budget per frame to work with. In the case of a 30 fps target, the CPU cores have 33 milliseconds to gather inputs, process the game logic, perform any physical simulations, traverse the scene description, and send the rendering commands to the graphics

hardware. In parallel, other tasks such as audio and network processing must be handled, with their own required response times. While this is happening, the GPU is typically executing the graphics commands submitted during the previous frame.

In most cases, CPU cores are a *homogeneous* resource; all cores are the same, and any of them are equally well suited to a given workload (there are some exceptions, such as the Cell processor used in Sony's PLAYSTATION 3 console).

In contrast, GPUs contain a *heterogeneous* mix of resources, each specialized to a certain set of tasks. Some of these resources consist of fixed-function hardware (for triangle rasterization, alpha blending, and texture sampling), and some are programmable cores. On older GPUs, programmable cores were further differentiated into vertex and pixel processing cores; newer GPU designs have *unified shader cores* which can execute any of the programmable shader types.

Such heterogeneous resources are budgeted separately. Typically, at any point, only one resource type will be the bottleneck, and the others will have excess capacity. On the one hand, this is good, since this capacity can be leveraged to improve visual quality without decreasing performance. On the other hand, it makes it harder to improve performance, since decreasing usage of any of the non-bottleneck resources will have no effect. Even decreasing usage of the bottleneck resource may only improve performance slightly, depending on the degree of utilization of the "next bottleneck."

25.2.2 Storage

Game platforms, like any modern computing system, possess multi-stage *storage hierarchies*, with smaller, faster memory types at the top and larger, slower storage at the bottom. This arrangement is borne of engineering necessity, although it does complicate life for the developer. Most platforms include optical disc storage, which is extremely slow and is used mostly for delivery. On platforms such as Windows, a lengthy installation process is performed once to move all data from the optical disc onto the hard drive, which is significantly faster. The optical disc is never used again (except as an anti-piracy measure). On console platforms, this is less common, although it does sometimes happen when a hard drive is guaranteed to be present, as on Sony's PLAYSTATION 3 console. More often, the hard drive (if present) is only used as a cache for the optical disc.

The next step up the memory hierarchy is RAM, which on many platforms is divided into general system RAM and VRAM (video RAM) which benefits from a high-speed interface to the graphics hardware. A game level may be too large to

fit in RAM, in which case the game developer needs to manage moving the data in and out of RAM as needed. On platforms such as Windows, virtual memory is often used for this. On console platforms, custom data streaming and caching systems are typically employed.

Finally, both the CPU and GPU boast various kinds of on-chip memory and caches. These are extremely small and fast and are usually managed by the graphics API.

Graphics resources take up a lot of memory, so they are a primary focus of storage budgets in game development. Textures are usually the greatest memory consumers, followed by geometry (vertex data), and finally other types of graphics data such as animations. Not all memory can be used for graphics—audio also takes up a fair bit, and game logic may use sizeable data structures. As in the case of processing time, budgeting tends to be somewhat looser on Windows, where the exact amount of memory present on the user's system is unknown and virtual memory covers a multitude of sins. In contrast, memory budgeting on console platforms is quite strict—often the lead programmer keeps track of memory on a spreadsheet and a programmer requiring more memory for their system needs to beg, borrow, or steal it from someone else.

The various levels of the memory hierarchy differ not only in size, but also in access speed. This has two separate dimensions: *latency* and *bandwidth*.

Latency is the time that elapses between a storage access request and its final fulfillment. This varies from a few clock cycles (for on-chip cache) to millions of clock cycles (for data residing on optical disc). Latency is usually an issue for read access (although write latency can also be an issue if the result needs to be read back from memory soon after). In some cases, the read request is *blocking*, which means that the processor core that submitted the read can do nothing else until the request is fulfilled. In other cases, the read is *non-blocking*; the processing core can submit the read request, do other types of processing, and then use the results of the read after it has arrived. Texture accesses by the GPU are an example of non-blocking reads; an important aspect of GPU design is to find ways to "hide" texture read latency by performing unrelated computations while the texture read is being fulfilled.

For this latency hiding to work, there must be a sufficient amount of computation relative to texture accesses. This is an important consideration for the shader writer; the optimal mix of computation vs. texture access keeps changing (in favor of more computation) as memory fails to keep up with increases in processing power.

Bandwidth refers to the maximum rate of transfer to and from storage. It is typically measured in gigabytes per second.

25.2.3 Development Resources

Besides hardware resources, such as processing power and storage space, the game graphics programmer also has to contend with a different kind of limited resource—the time of his teammates! When selecting graphics techniques, the engineering resources needed to implement each technique must be taken into account, as well as any tools necessary to compute the input data (in many cases, tools can take significantly more time than implementing the technique itself). Perhaps most importantly, the impact on artist productivity must be taken into account. Most graphics techniques use assets created by game artists, who comprise by far the largest part of most modern game teams. The graphics programmer must foster the artist's productivity and creativity, which will ultimately determine the visual quality of the game.

25.3 Optimization Techniques

Making wise use of these limited resources is the primary challenge of the game graphics programmer. To this end, various optimization techniques are commonly employed.

In many games, pixel shader processing is a primary bottleneck. Most GPUs contain hierarchical depth-culling hardware which can avoid executing pixel shaders on occluded surfaces. To make good use of this hardware, opaque objects can be rendered back-to-front. Alternatively, optimal depth-culling usage can be achieved by performing a *depth prepass*, i.e., rendering all the opaque objects into the depth buffer (without any color output or pixel shaders) before rendering the scene normally. This does incur some overhead (due to the need to render every object twice), but in many cases the performance gain is worth it.

The fastest way to render an object is to not render it at all; thus any method of discerning early on that an object is occluded can be useful. This saves not only pixel processing but also vertex processing and even CPU time that would be spent submitting the object to the graphics API. View frustum culling (see Section 8.4.1) is universally employed, but in many games it is not sufficient. High-level occlusion culling algorithms are often used, utilizing data structures such as PVS (potentially visible sets) or BSP (binary spatial partitioning) trees to quickly narrow down the pool of potentially visible objects.

Even if an object is visible, it may be at such a distance that most of its detail can be removed without apparent effect. LOD (level-of-detail) algorithms render different representations of an object based on distance (or other factors, such

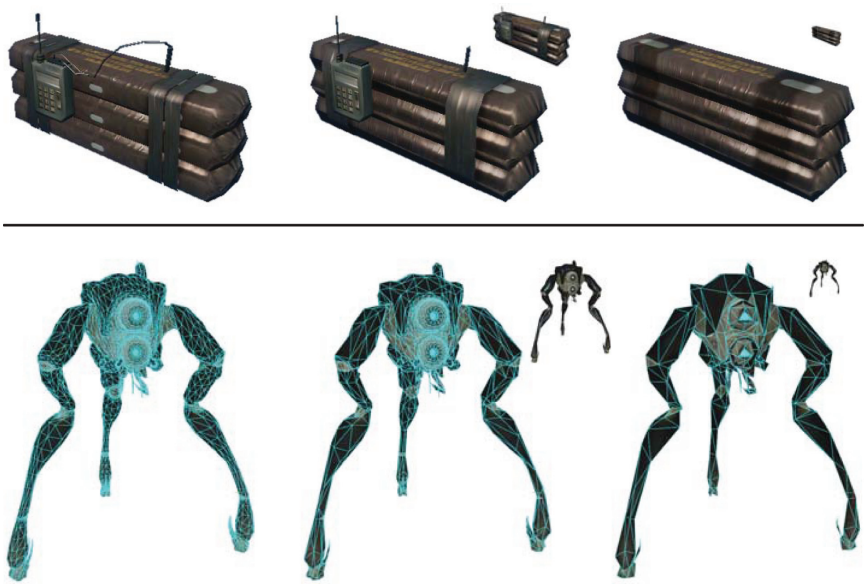

Figure 25.1. Two examples of game objects at a varying level of detail. The small inset images show the relative sizes at which the simplified models might be used. *Upper row of images courtesy Crytek; lower row courtesy Valve Corp.*

as screen coverage or importance). This can save significant processing, vertex processing in particular. Examples can be seen in Figure 25.1.

In many cases, processing can be performed before the game even starts. The results of such *preprocessing* can be stored and used each frame, thus speeding up the game. This is most commonly employed for lighting, where global illumination algorithms are utilized to compute lighting throughout the scene and store it in lightmaps and other data structures for later use.

25.4 Game Types

Since game requirements vary widely, the selection of graphics techniques is driven by the exact type of game being developed.

The allocation of processing time depends strongly on the frame rate. Currently, most console games tend to target 30 frames per second, since this enables much higher graphics quality. However, certain game types with fast gameplay require very low latency, and such games typically render at 60 frames per second.

This includes music games such as *Guitar Hero* and first-person shooters such as *Call of Duty*.

The frame rate determines the available time to render the scene. The composition of the scene itself also varies widely from game to game. Most games have a division between *background geometry* (scenery, mostly static) and *foreground geometry* (characters and dynamic objects). These are handled differently by the rendering engine. For example, background geometry will often have lightmaps containing precomputed lighting, which is not feasible for foreground objects. Precomputed lighting is typically applied to foreground objects via some type of volumetric representation which can take account of the changing position of each object over time.

Some games have relatively enclosed environments, where the camera remains largely in place. The purest examples are fighting games such as the *Street Fighter* series, but this is also true to some extent for games such as *Devil May Cry* and *God of War*. These games have cameras that are not under direct player control, and the game play tends to move from one enclosed environment to another, spending a significant amount of playing time in each. This allows the game developer to lavish large amounts of resources (processing, storage, and artist time) on each room or enclosed environment, resulting in very high levels of graphics fidelity.

Other games have extremely large worlds, where the player can move about freely. This is most true for "sandbox games" such as the *Grand Theft Auto* series and online role-playing games such as *World of Warcraft*. Such games pose great challenges to the graphics developer, since resource allocation is very difficult when during each frame the player can see a large extent of the world. Further complicating things, the player can freely go to some formerly distant part of the world and observe it from up close. Such games typically have changing time of day, which makes precomputation of lighting difficult at best, if not impossible.

Most games, such as first-person shooters, are somewhere between the two extremes. The player can see a fair amount of scenery each frame, but movement through the game world is somewhat constrained. Many games also have a fixed time of day for each game level, for ease of lighting precomputation.

The number of foreground objects rendered also varies widely between game types. Real-time strategy games such as the *Command and Conquer* series often have many dozens, if not hundreds, of units visible on screen. Other types of games have more limited quantities of visible characters, with fighting games at the opposite extreme, where only two characters are visible, each rendered with extremely high detail. A distinction must be drawn between the number of characters visible at any time (which affects budgeting of processing time) and

Figure 25.2. *Crysis* exemplifies the realistic and detailed graphics expected of first-person shooters. *Image courtesy Crytek.*

the number of *unique* characters which can potentially be visible at short notice (which affects storage budgets).

The type or *genre* of game also determines audience expectations of the graphics. For example, first-person shooters have historically had very high levels of graphics fidelity, and this expectation drives the graphics design when developing new games in that genre; see Figure 25.2. On the other hand, puzzle games have typically had relatively simplistic graphics, so most game developers will not invest large amounts of programming or art resources into developing photorealistic graphics for such games.

Although most games aim for a photorealistic look, a few do attempt more stylized rendering. One interesting example of this is *Okami*, which can be seen in Figure 25.3.

The management of development resources also differs by game type. Most games have a closed development cycle of one to two years, which ends after the game ships. Recently it has become common to have downloadable content (DLC), which can be purchased after the game ships, so some development resources need to be reserved for that. Persistent-world online games have a never-ending development process where new content is continually being generated, at least as long as the game is economically viable (which may be a period of decades).

Figure 25.3. An example of highly stylized, non-photorealistic rendering from the game *Okami*. *Image courtesy Capcom Entertainment, Inc.*

The creative exploitation of the specific requirements and restrictions of a particular game is the hallmark of a skilled game graphics programmer. A good example is the game *LittleBigPlanet*, which has a "two-and-a-half-dimensional" game world comprising a small number of two-dimensional layers, as well as a noninteractive background. The graphics quality of this game is excellent, driven by the use of unusual rendering techniques specialized to this type of environment; see Figure 25.4.

25.5 The Game Production Process

The game production process starts with the basic game design or concept. In some cases (such as sequels), the basic gameplay and visual design is clear, and only incremental changes are made. In the case of a new game type, extensive prototyping is needed to determine gameplay and design. Most cases sit somewhere in the middle, where there are some new gameplay elements and the visual design is somewhat open. After this step there may be a *greenlight* stage where

Figure 25.4. The *LittleBigPlanet* developers took care to choose techniques that fit the game's constraints, combining them in unusual ways to achieve stunning results. *LittleBigPlanet © 2007 Sony Computer Entertainment Europe. Developed by Media Molecule. LittleBigPlanet is a trademark of Sony Computer Entertainment Europe.*

some early demo or concept is shown to the game publisher to get approval (and funding!) for the game.

The next step is typically *pre-production*. While other teams are working on finishing up the last game, a small core team works on making any needed changes to the game engine and production tool chain, as well as working out the rough details of any new gameplay elements. This core team is working under a strict deadline. After the existing game ships and the rest of the team comes back from a well-deserved vacation, the entire tool chain and engine must be ready for them. If the core team misses this deadline, several dozen developers may be left idle—an extremely expensive proposition!

Full production is the next step, with the entire team creating art assets, designing levels, tweaking gameplay, and implementing further changes to the game engine. In a perfect world, everything done during this process would be used in the final game, but in reality there is an iterative nature to game development which will result in some work being thrown out and redone. The goal is to minimize this with careful planning and prototyping.

When the game is functionally complete, the final stage begins. The term *alpha release* usually refers to the version which marks the start of extensive internal testing, *beta release* to the one which marks the start of extensive external testing, and *gold release* to the final release submitted to the console manufacturer,

but different companies have slightly varying definitions of these terms. In any case, testing, or *quality assurance* (QA) is an important part of this phase, and it involves testers at the game development studio, at the publisher, at the console manufacturer, and possibly external QA contractors as well. These various rounds of testing result in bug reports which are submitted back to the game developers and worked on until the next release.

After the game ships, most of the developers go on vacation for a while, but a small team may have to stay to work on patches or downloadable content. In the meantime, a small core team has been working on pre-production for the next game.

Art asset creation is an aspect of game production that is particularly relevant to graphics development, so I will go into it in some detail.

25.5.1 Asset Creation

While the exact process of art asset creation varies from game to game, the outline I give here is fairly representative. In the past, a single artist would create an entire asset from start to finish, but this process is now much more specialized, involving people with different skill sets working on each asset at various times. Some of these stages have clear dependencies (for example, a character cannot be animated until it is rigged and cannot be rigged before it is modeled). Most game developers have well-defined approval processes, where the art director or a lead artist signs off on each stage before the asset is sent on to the next. Ideally an asset proceeds through each stage exactly once, but in practice changes may be made that require resubmission.

Initial Modeling

Typically the art asset creation process starts by modeling the object geometry. This step is performed in a general-purpose modeling package such as Maya, MAX or Softimage. The modeled geometry will be passed directly to the game engine, so it is important to minimize vertex count while preserving good silhouettes. Character meshes must also be constructed so as to be amenable to animation.

In this stage, a two-dimensional surface parameterization for textures is usually created. It is important that this parameterization be highly continuous, since discontinuities require vertex duplication and may cause filtering artifacts. An example of a mesh with its associated texture parameterization is shown in Figure 25.5.

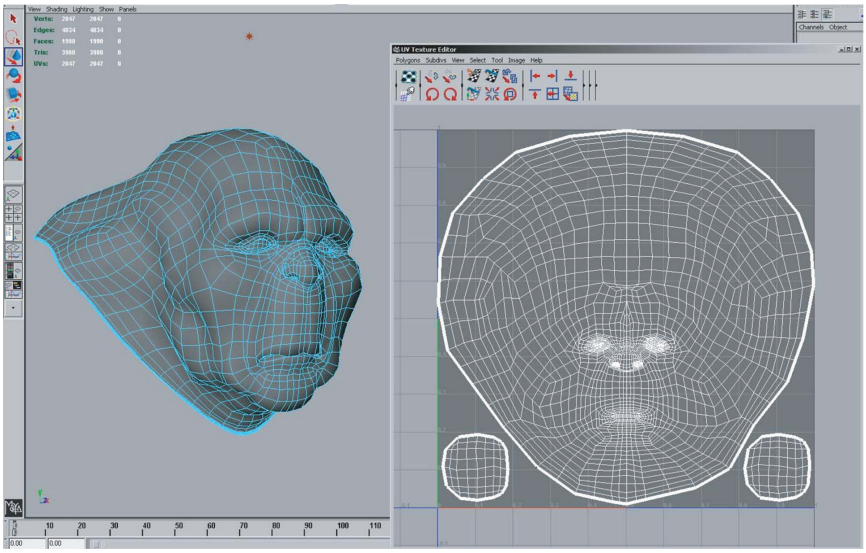

Figure 25.5. A mesh being modeled in Maya, with associated texture parameterization. *Image courtesy Keith Bruns.*

Texturing

In the past, texturing was a straightforward process of painting a color texture, typically in Photoshop. Now, specialized detail modeling packages such as ZBrush or Mudbox are commonly used to sculpt fine surface detail. Figures 25.6 and 25.7 show an example of this process.

If this additional detail were to be represented with actual geometry, millions of triangles would be needed. Instead, the detail is commonly "baked" into a normal map which is applied onto the original, coarse mesh, as shown in Figures 25.8 and 25.9.

Besides normal maps, multiple textures containing surface properties such as diffuse color, specular color, and smoothness (specular power) are also created. These are either painted directly on the surface in the detail modeling application, or in a two-dimensional application such as Photoshop. All of these texture maps use the surface parameterization defined in the initial modeling phase. When the texture is painted in a two-dimensional painting application, the artist must frequently switch between the painting application and some other application which can show a three-dimensional rendering of the object with the texture applied. This iterative process is illustrated in Figures 25.10, 25.11, 25.12, and 25.13.

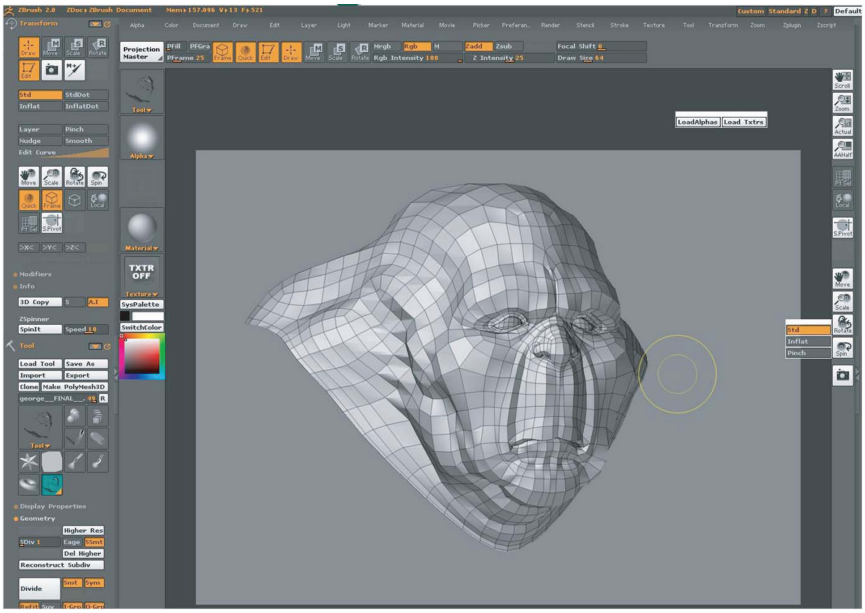

Figure 25.6. The mesh from Figure 25.5 has been brought into ZBrush for detail modeling. *Image courtesy Keith Bruns.*

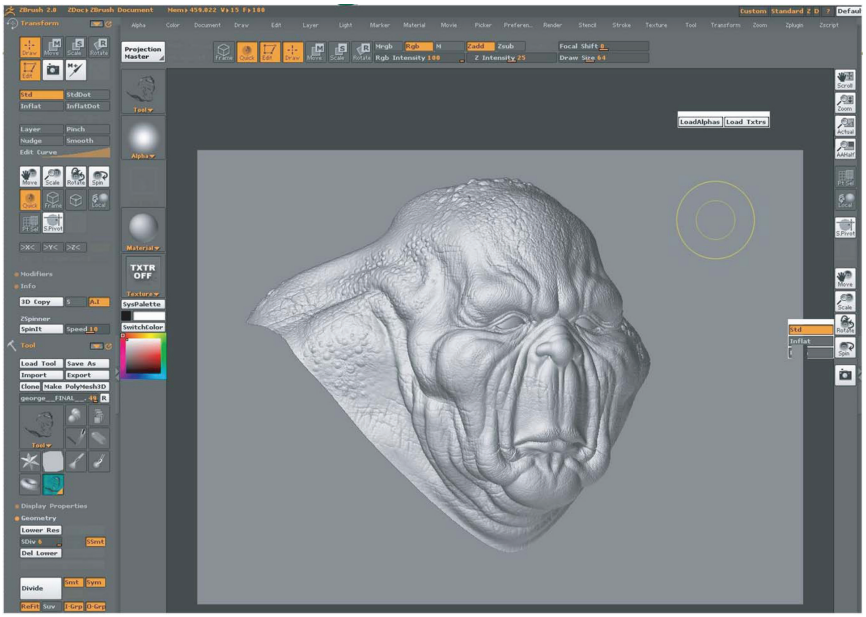

Figure 25.7. The mesh from Figure 25.6, with fine detail added to it in ZBrush. *Image courtesy Keith Bruns.*

Figure 25.8. A visualization (in ZBrush) of the mesh from Figure 25.6, rendered with a normal map derived from the detailed mesh in Figure 25.7. The bottom of the figure shows the interface for ZBrush's "Zmapper" tool, which was used to derive the normal map. *Image courtesy Keith Bruns.*

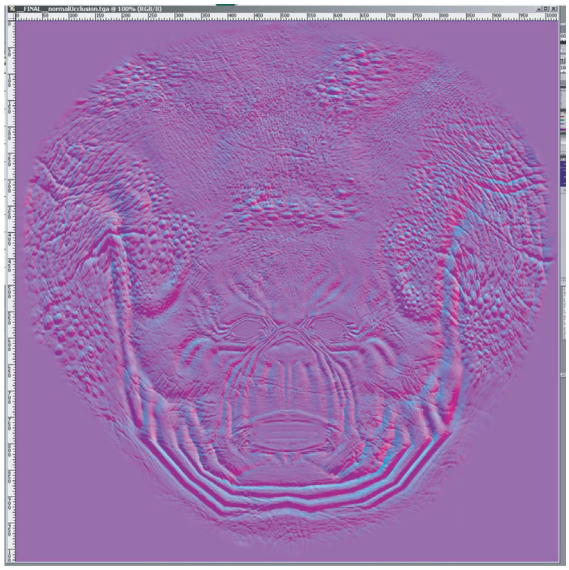

Figure 25.9. The normal map used in Figure 25.8. In this image, the red, green, and blue channels of the texture contain the X, Y, and Z coordinates of the surface normals. *Image courtesy Keith Bruns.*

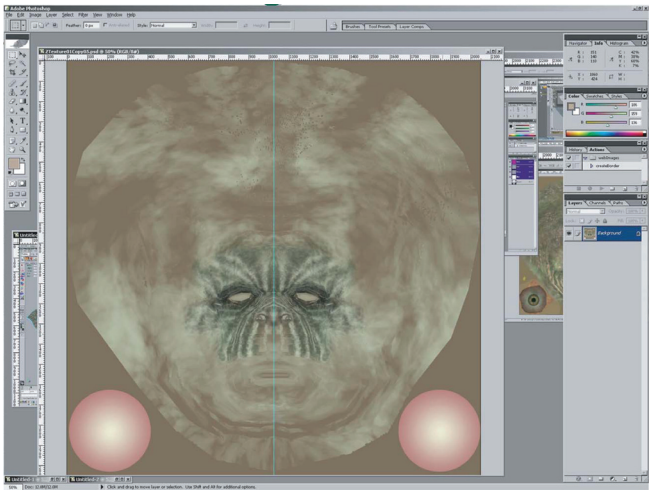

Figure 25.10. An early version of a diffuse color texture for the mesh from Figure 25.8, shown in Photoshop. *Image courtesy Keith Bruns.*

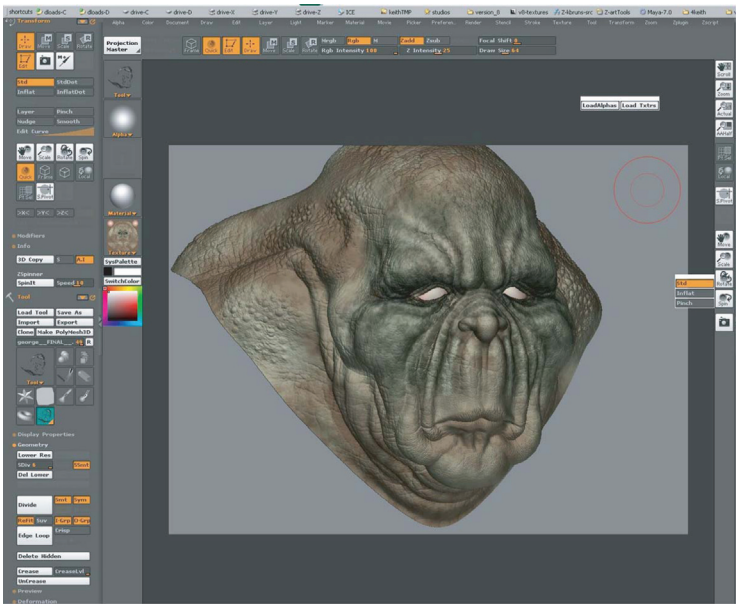

Figure 25.11. A rendering (in ZBrush) of the mesh with normal map and early diffuse color texture (from Figure 25.10) applied. *Image courtesy Keith Bruns.*

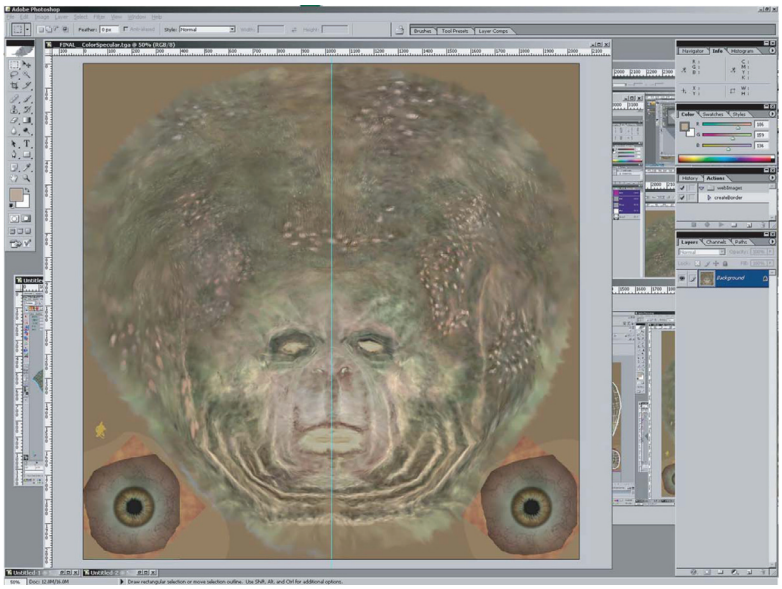

Figure 25.12. Final version of the color texture from Figure 25.10. *Image courtesy Keith Bruns.*

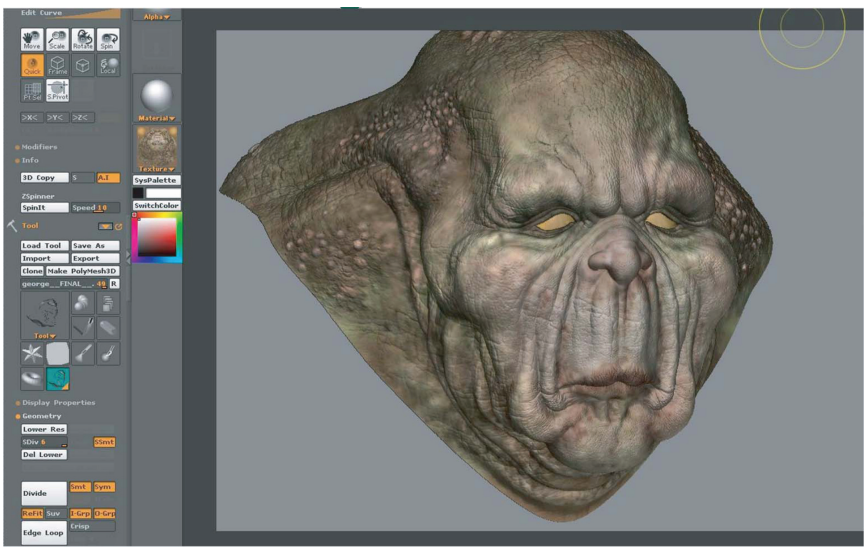

Figure 25.13. Rendering of the mesh with normal map and final color texture (from Figure 25.12) applied. *Image courtesy Keith Bruns.*

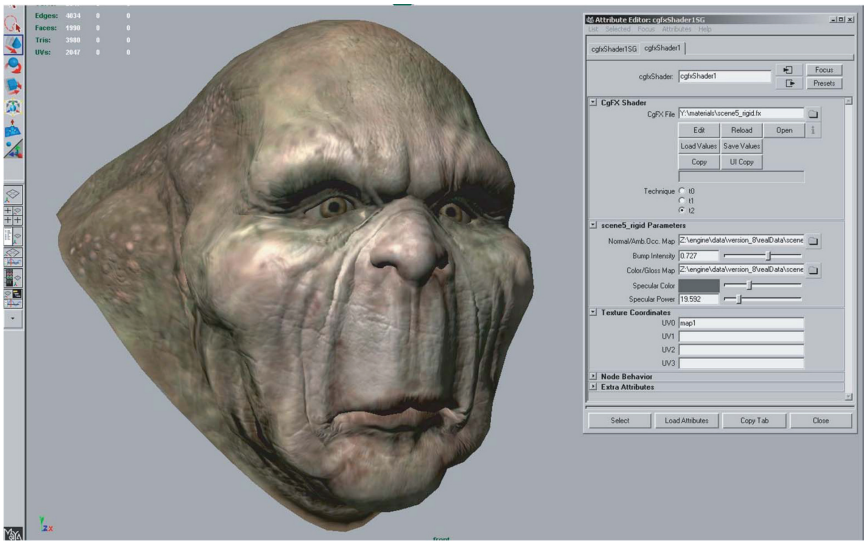

Figure 25.14. Shader configuration in Maya. The interface on the right is used to select the shader, assign textures to shader inputs, and set the values of non-texture shader inputs (such as the "Specular Color" and "Specular Power" sliders). The rendering on the left is updated dynamically while these properties are modified, enabling immediate visual feedback. *Image courtesy Keith Bruns.*

Shading

Shaders are typically applied in the same application used for initial modeling. In this process, a shader (from the set of shaders defined for that game) is applied to the mesh. The various textures resulting from the detail modeling stage are applied as inputs to this shader, using the surface parameterization defined during initial modeling. Various other shader inputs are set via visual experimentation ("tweaking"); see Figure 25.14.

Lighting

In the case of background scenery, lighting artists will typically start their work after modeling, texturing, and shading have been completed. Light sources are placed and their effect computed in a preprocessing step. The results of this process are stored in lightmaps for later use by the rendering engine.

Animation

Character meshes undergo several additional steps related to animation. The primary method used to animate game characters is *skinning*. This requires a *rig*, consisting of a hierarchy of transform nodes that is attached to the character, a

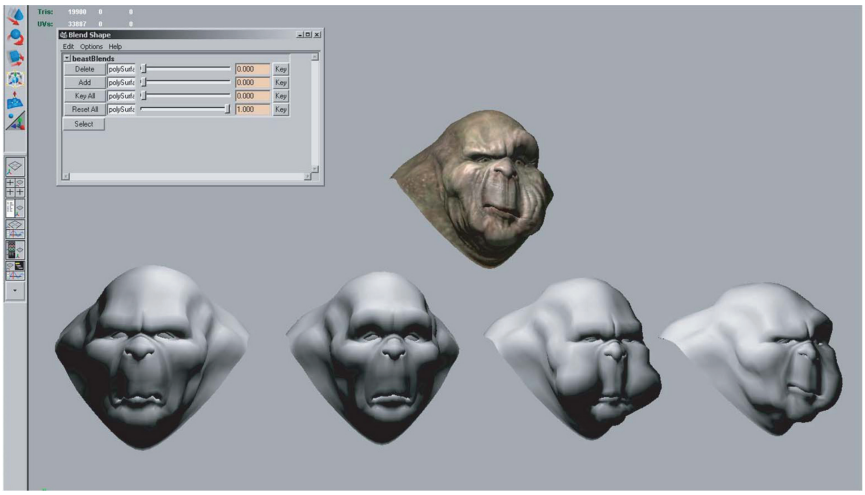

Figure 25.15. Morph target interface in Maya. The bottom row shows four different morph targets, and the model at the top shows the effects of combining several morph targets together. The interface at the upper left is used to control the degree to which each morph target is applied. *Image courtesy Keith Bruns.*

process known as *rigging*. The area of effect of each transform node is painted onto a subset of mesh vertices. Finally, animators create animations that move, rotate, and scale these transform nodes, "dragging" the mesh behind them.

A typical game character will have many dozens of animations, corresponding to different modes of motion (walking, running, turning) as well as different actions such as attacks. In the case of a main character, the number of animations can be in the hundreds. Transitions between different animations also need to be defined.

For facial animation, another technique, called *morph targets* is sometimes employed. In this technique, the mesh vertices are directly manipulated to deform the mesh. Different copies of the deformed mesh are stored (e.g., for different facial expressions) and combined by the game engine at runtime. The creation of morph targets is shown in Figure 25.15.

Notes

There is a huge amount of information on real-time rendering and game programming available, both in books and online. Here are some resources I can recommend from personal familiarity.

Game Developer Magazine is a good source of information on game development, as are slides from the talks given at the annual *Game Developers Conference* (GDC) and Microsoft's *Gamefest* conference. The *GPU Gems* and *ShaderX* book series also contain good information—all of the former and the first two of the latter are also available online.

Eric Lengyel's *Mathematics for 3D Game Programming & Computer Graphics*, now in its second edition, is a good reference for the various types of math used in graphics and games. A specific area of game programming that is closely related to graphics is collision detection, for which Christer Ericson's *Real-Time Collision Detection* is the definitive resource.

Since its first edition in 1999, Eric Haines and Tomas Akenine-Möller's *Real-Time Rendering* has endeavored to cover this fast-growing field in a thorough manner. As a longtime fan of this book, I was glad to have the opportunity to be a coauthor on the third edition, which came out in mid-2008.

Reading is not enough—make sure you play a variety of games regularly to get a good idea of the requirements of various game types, as well as the current state of the art.

Exercises

1. Examine the visuals of two dissimilar games. What differences can you deduce in the graphics requirements of these two games? Analyze the effect on rendering time, storage budgets, etc.

Tamara Munzner

26

Visualization

A major application area of computer graphics is *visualization*, where computer-generated images are used to help people understand both spatial and nonspatial data. Visualization is used when the goal is to augment human capabilities in situations where the problem is not sufficiently well defined for a computer to handle algorithmically. If a totally automatic solution can completely replace human judgment, then visualization is not typically required. Visualization can be used to generate new hypotheses when exploring a completely unfamiliar dataset, to confirm existing hypotheses in a partially understood dataset, or to present information about a known dataset to another audience.

Visualization allows people to offload cognition to the perceptual system, using carefully designed images as a form of *external memory*. The human visual system is a very high-bandwidth channel to the brain, with a significant amount of processing occurring in parallel and at the pre-conscious level. We can thus use external images as a substitute for keeping track of things inside our own heads. For an example, let us consider the task of understanding the relationships between a subset of the topics in the splendid book *Gödel, Escher, Bach: The Eternal Golden Braid* (Hofstadter, 1979); see Figure 26.1.

When we see the dataset as a text list, at the low level we must read words and compare them to memories of previously read words. It is hard to keep track of just these dozen topics using cognition and memory alone, let alone the hundreds of topics in the full book. The higher-level problem of identifying neighborhoods, for instance finding all the topics two hops away from the target topic *Paradoxes*, is very difficult.

Infinity - Lewis Carroll	Epimenides - Self-ref
Infinity - Zeno	Epimenides - Tarski
Infinity - Paradoxes	Tarski - Epimenides
Infinity - Halting problem	Halting problem - Decision procedures
Zeno - Lewis Carroll	Halting problem - Turing
Paradoxes - Lewis Carroll	Lewis Carroll - Wordplay
Paradoxes - Epimenides	Tarski - Truth vs. provability
Paradoxes - Self-ref	Tarski - Undecidability

Figure 26.1. Keeping track of relationships between topics is difficult using a text list.

Figure 26.2 shows an external visual representation of the same dataset as a node-link graph, where each topic is a node and the linkage between two topics is shown directly with a line. Following the lines by moving our eyes around the image is a fast low-level operation with minimal cognitive load, so higher-level neighborhood finding becomes possible. The placement of the nodes and the routing of the links between them was created automatically by the *dot* graph drawing program (Gansner, Koutsofois, North, & Vo, 1993).

We call the mapping of dataset attributes to a visual representation a *visual encoding*. One of the central problems in visualization is choosing appropriate encodings from the enormous space of possible visual representations, taking into account the characteristics of the human perceptual system, the dataset in question, and the task at hand.

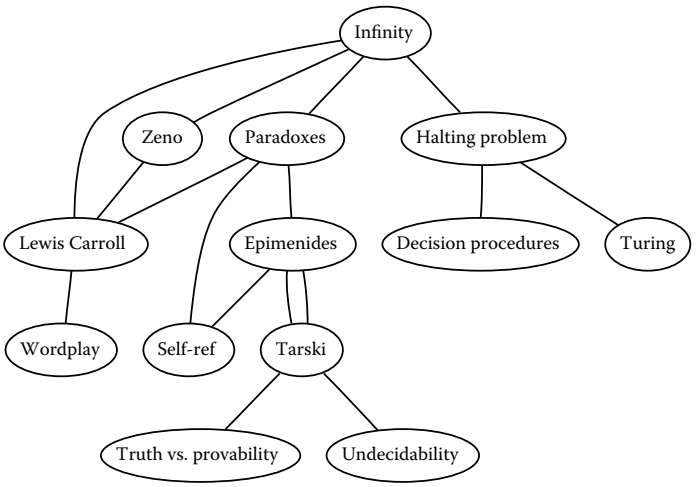

Figure 26.2. Substituting perception for cognition and memory allows us to understand relationships between book topics quickly.

26.1 Background

26.1.1 History

People have a long history of conveying meaning through static images, dating back to the oldest known cave paintings from over thirty thousand years ago. We continue to visually communicate today in ways ranging from rough sketches on the back of a napkin to the slick graphic design of advertisements. For thousands of years, cartographers have studied the problem of making maps that represent some aspect of the world around us. The first visual representations of abstract, nonspatial datasets were created in the 18th century by William Playfair (Friendly, 2008).

Although we have had the power to create moving images for over one hundred and fifty years, creating dynamic images interactively is a more recent development only made possible by the widespread availability of fast computer graphics hardware and algorithms in the past few decades. Static visualizations of tiny datasets can be created by hand, but computer graphics enables interactive visualization of large datasets.

26.1.2 Resource Limitations

When designing a visualization system, we must consider three different kinds of limitations: computational capacity, human perceptual and cognitive capacity, and display capacity.

As with any application of computer graphics, computer time and memory are limited resources and we often have hard constraints. If the visualization system needs to deliver interactive response, then it must use algorithms that can run in a fraction of a second rather than minutes or hours.

On the human side, memory and attention must be considered as finite resources. Human memory is notoriously limited, both for long-term recall and for shorter-term working memory. Later in this chapter, we discuss some of the power and limitations of the low-level visual attention mechanisms that carry out massively parallel processing of the visual field. We store surprisingly little information internally in visual working memory, leaving us vulnerable to *change blindness*, the phenomenon where even very large changes are not noticed if we are attending to something else in our view (Simons, 2000). Moreover, vigilance is also a highly limited resource; our ability to perform visual search tasks degrades quickly, with far worse results after several hours than in the first few minutes (Ware, 2000).

Display capacity is a third kind of limitation to consider. Visualization designers often "run out of pixels," where the resolution of the screen is not large enough to show all desired information simultaneously. The *information density* of a particular frame is a measure of the amount of information encoded versus the amount of unused space. There is a tradeoff between the benefits of showing as much as possible at once, to minimize the need for navigation and exploration, and the costs of showing too much at once, where the user is overwhelmed by visual clutter.

26.2 Data Types

Many aspects of a visualization design are driven by the type of the data that we need to look at. For example, is it a table of numbers, or a set of relations between items, or inherently spatial data such as a location on the Earth's surface or a collection of documents?

We start by considering a table of data. We call the rows *items* of data and the columns are *dimensions*, also known as *attributes*. For example, the rows might represent people, and the columns might be names, age, height, shirt size, and favorite fruit.

We distinguish between three types of dimensions: quantitative, ordered, and categorical. *Quantitative* data, such as age or height, is numerical and we can do arithmetic on it. For example, the quantity of 68 inches minus 42 inches is 26 inches. With *ordered* data, such as shirt size, we cannot do full-fledged arithmetic, but there is a well-defined ordering. For example, large minus medium is not a meaningful concept, but we know that medium falls between small and large. *Categorical* data, such as favorite fruit or names, does not have an implicit ordering. We can only distinguish whether two things are the same (apples) or different (apples vs. bananas).

Relational data, or *graphs*, are another data type where *nodes* are connected by *links*. One specific kind of graph is a *tree*, which is typically used for hierarchical data. Both nodes and edges can have associated attributes. The word *graph* is unfortunately overloaded in visualization. The node-link graphs we discuss here, following the terminology of graph drawing and graph theory, could also be called *networks*. In the field of statistical graphics, graph is often used for *chart*, as in the line charts for time-series data shown in Figure 26.10.

Some data is inherently spatial, such as geographic location or a field of measurements at positions in three-dimensional space as in the MRI or CT scans used by doctors to see the internal structure of a person's body. The information as-

sociated with each point in space may be an unordered set of scalar quantities, or indexed vectors, or tensors. In contrast, nonspatial data can be visually encoded using spatial position, but that encoding is chosen by the designer rather than given implicitly in the semantics of the dataset itself. This choice is one of the most central and difficult problems of visualization design.

26.2.1 Dimension and Item Count

The number of data dimensions that need to be visually encoded is one of the most fundamental aspects of the visualization design problem. Techniques that work for a low-dimensional dataset with a few columns will often fail for very high-dimensional datasets with dozens or hundreds of columns. A data dimension may have hierarchical structure, for example with a time series dataset where there are interesting patterns at multiple temporal scales.

The number of data items is also important: a visualization that performs well for a few hundred items often does not scale to millions of items. In some cases the difficulty is purely algorithmic, where a computation would take too long; in others it is an even deeper perceptual problem that even an instantaneous algorithm could not solve, where visual clutter makes the representation unusable by a person. The range of possible values within a dimension may also be relevant.

26.2.2 Data Transformation and Derived Dimensions

Data is often transformed from one type to another as part of a visualization pipeline for solving the domain problem. For example, an original data dimension might be made up of quantitative data: floating point numbers that represent temperature. For some tasks, like finding anomalies in local weather patterns, the raw data might be used directly. For another task, like deciding whether water is an appropriate temperature for a shower, the data might be transformed into an ordered dimension: hot, warm, or cold. In this transformation, most of the detail is aggregated away. In a third example, when making toast, an even more lossy transformation into a categorical dimension might suffice: burned or not burned.

The principle of transforming data into *derived dimensions*, rather than simply visually encoding the data in its original form, is a powerful idea. In Figure 26.10, the original data was an ordered collection of time-series curves. The transformation was to cluster the data, reducing the amount of information to visually encode to a few highly meaningful curves.

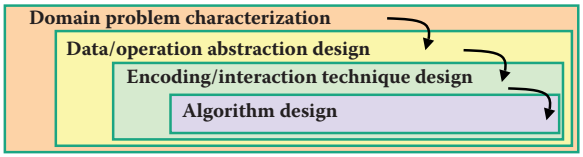

Figure 26.3. Four nested layers of validation for visualization.

26.3 Human-Centered Design Process

The visualization design process can be split into a cascading set of layers, as shown in Figure 26.3. These layers all depend on each other; the output of the level above is input into the level below.

26.3.1 Task Characterization

A given dataset has many possible visual encodings. Choosing which visual encoding to use can be guided by the specific needs of some intended user. Different questions, or *tasks*, require very different visual encodings. For example, consider the domain of software engineering. The task of understanding the coverage of a test suite is well supported by the Tarantula interface shown in Figure 26.11. However, the task of understanding the modular decomposition of the software while refactoring the code might be better served by showing its hierarchical structure more directly as a node-link graph.

Understanding the requirements of some target audience is a tricky problem. In a human-centered design approach, the visualization designer works with a group of target users over time (C. Lewis & Rieman, 1993). In most cases, users know they need to somehow view their data but cannot directly articulate their needs as clear-cut tasks in terms of operations on data types. The iterative design process includes gathering information from the target users about their problems through interviews and observation of them at work, creating prototypes, and observing how users interact with those prototypes to see how well the proposed solution actually works. The software engineering methodology of requirements analysis can also be useful (Kovitz, 1999).

26.3.2 Abstraction

After the specific domain problem has been identified in the first layer, the next layer requires abstracting it into a more generic representation as operations on

the data types discussed in the previous section. Problems from very different domains can map to the same visualization abstraction. These generic operations include sorting, filtering, characterizing trends and distributions, finding anomalies and outliers, and finding correlation (Amar, Eagan, & Stasko, 2005). They also include operations that are specific to a particular data type, for example following a path for relational data in the form of graphs or trees.

This abstraction step often involves data transformations from the original raw data into derived dimensions. These derived dimensions are often of a different type than the original data: a graph may be converted into a tree, tabular data may be converted into a graph by using a threshold to decide whether a link should exist based on the field values, and so on.

26.3.3 Technique and Algorithm Design

Once an abstraction has been chosen, the next layer is to design appropriate visual encoding and interaction techniques. Section 26.4 covers the principles of visual encoding, and we discuss interaction principles in Sections 26.5. We present techniques that take these principles into account in Sections 26.6 and 26.7.

A detailed discussion of visualization algorithms is unfortunately beyond the scope of this chapter.

26.3.4 Validation

Each of the four layers has different validation requirements.

The first layer is designed to determine whether the problem is correctly characterized: is there really a target audience performing particular tasks that would benefit from the proposed tool? An immediate way to test assumptions and conjectures is to observe or interview members of the target audience, to ensure that the visualization designer fully understands their tasks. A measurement that cannot be done until a tool has been built and deployed is to monitor its adoption rate within that community, although of course many other factors in addition to utility affect adoption.

The next layer is used to determine whether the abstraction from the domain problem into operations on specific data types actually solves the desired problem. After a prototype or finished tool has been deployed, a field study can be carried out to observe whether and how it is used by its intended audience. Also, images produced by the system can be analyzed both qualitatively and quantitatively.

The purpose of the third layer is to verify that the visual encoding and interaction techniques chosen by the designer effectively communicate the chosen

abstraction to the users. An immediate test is to justify that individual design choices do not violate known perceptual and cognitive principles. Such a justification is necessary but not sufficient, since visualization design involves many tradeoffs between interacting choices. After a system is built, it can be tested through formal laboratory studies where many people are asked to do assigned tasks so that measurements of the time required for them to complete the tasks and their error rates can be statistically analyzed.

A fourth layer is employed to verify that the algorithm designed to carry out the encoding and interaction choices is faster or takes less memory than previous algorithms. An immediate test is to analyze the computational complexity of the proposed algorithm. After implementation, the actual time performance and memory usage of the system can be directly measured.

26.4 Visual Encoding Principles

We can describe visual encodings as graphical elements, called *marks*, that convey information through visual channels. A zero-dimensional mark is a point, a one-dimensional mark is a line, a two-dimensional mark is an area, and a three-dimensional mark is a volume. Many *visual channels* can encode information, including spatial position, color, size, shape, orientation, and direction of motion. Multiple visual channels can be used to simultaneously encode different data dimensions; for example, Figure 26.4 shows the use of horizontal and vertical spatial position, color, and size to display four data dimensions. More than one channel can be used to redundantly code the same dimension, for a design that displays less information but shows it more clearly.

26.4.1 Visual Channel Characteristics

Important characteristics of visual channels are distinguishability, separability, and popout.

Channels are not all equally distinguishable. Many psychophysical experiments have been carried out to measure the ability of people to make precise distinctions about information encoded by the different visual channels. Our abilities depend on whether the data type is quantitative, ordered, or categorical. Figure 26.5 shows the rankings of visual channels for the three data types. Figure 26.6 shows some of the default mappings for visual channels in the Tableau/Polaris system, which take into account the data type.

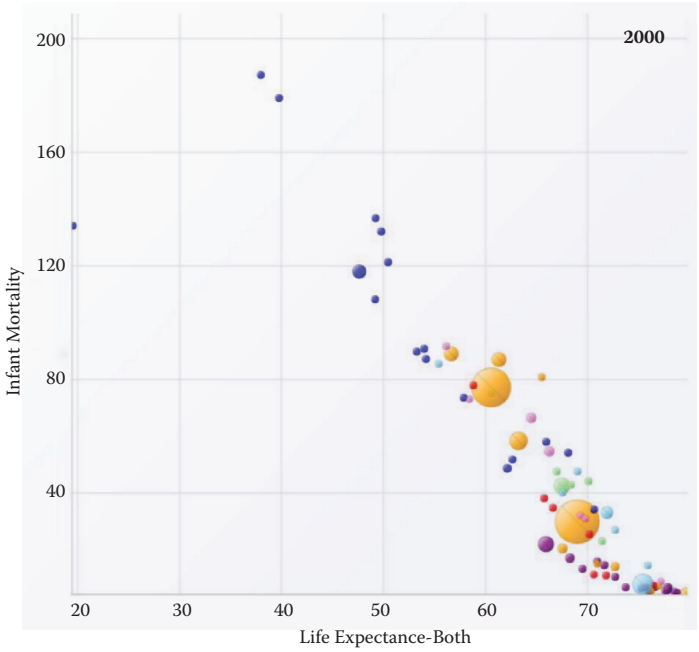

Figure 26.4. The four visual channels of horizontal and vertical spatial position, color, and size are used to encode information in this scatterplot chart *Image courtesy George Robertson* (Robertson, Fernandez, Fisher, Lee, & Stasko, 2008), © IEEE 2008.

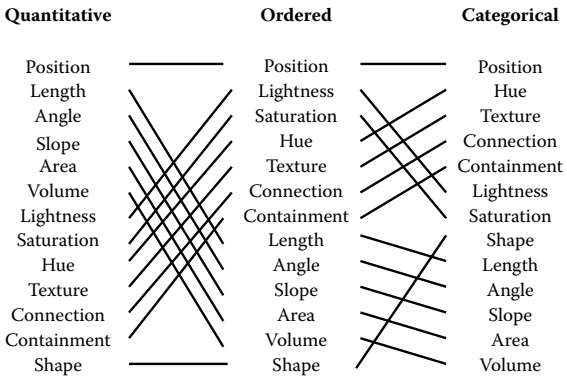

Figure 26.5. Our ability to perceive information encoded by a visual channel depends on the type of data used, from most accurate at the top to least at the bottom. *Redrawn and adapted from (Mackinlay, 1986).*

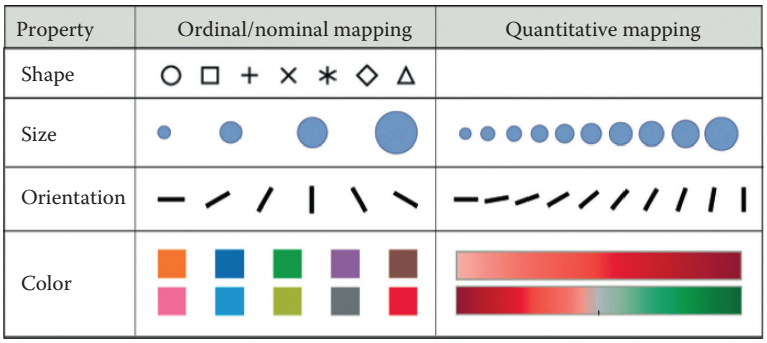

Figure 26.6. The Tableau/Polaris system default mappings for four visual channels according to data type. *Image courtesy Chris Stolte* (Stolte, Tang, & Hanrahan, 2008), © 2008 IEEE.

Spatial position is the most accurate visual channel for all three types of data, and it dominates our perception of a visual encoding. Thus, the two most important data dimensions are often mapped to horizontal and vertical spatial positions.

However, the other channels differ strongly between types. The channels of length and angle are highly discriminable for quantitative data but poor for ordered and categorical, while in contrast hue is very accurate for categorical data but mediocre for quantitative data.

We must always consider whether there is a good match between the dynamic range necessary to show the data dimension and the dynamic range available in the channel. For example, encoding with line width uses a one-dimensional mark and the size channel. There are a limited number of width steps that we can reliably use to visually encode information: a minimum thinness of one pixel is enforced by the screen resolution (ignoring antialiasing to simplify this discussion), and there is a maximum thickness beyond which the object will be perceived as a polygon rather than a line. Line width can work very well to show three or four different values in a data dimension, but it would be a poor choice for dozens or hundreds of values.

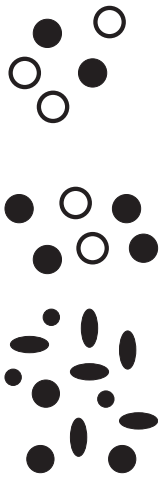

Figure 26.7. Color and location are separable channels well suited to encode different data dimensions, but the horizontal size and and vertical size channels are automatically fused into an integrated perception of area. *Redrawn after (Ware, 2000).*

Some visual channels are *integral*, fused together at a pre-conscious level, so they are not good choices for visually encoding different data dimensions. Others are *separable*, without interactions between them during visual processing, and are safe to use for encoding multiple dimensions. Figure 26.7 shows two channel pairs. Color and position are highly separable. We can see that horizontal size and vertical size are not so easy to separate, because our visual system automatically integrates these together into a unified perception of area. Size interacts with many channels: as the size of an object grows smaller, it becomes more difficult to distinguish its shape or color.

We can selectively attend to a channel so that items of a particular type "pop out" visually, as discussed in Section 20.4.3. An example of visual popout is when we immediately spot the red item amidst a sea of blue ones, or distinguish the circle from the squares. Visual popout is powerful and scalable because it occurs in parallel, without the need for conscious processing of the items one by one. Many visual channels have this popout property, including not only the list above but also curvature, flicker, stereoscopic depth, and even the direction of lighting. However, in general we can only take advantage of popout for one channel at a time. For example, a white circle does not pop out from a group of circles and squares that can be white or black, as shown in Figure 20.46. When we need to search across more than one channel simultaneously, the length of time it takes to find the target object depends linearly on the number of objects in the scene.

26.4.2 Color

Color can be a very powerful channel, but many people do not understand its properties and use it improperly. As discussed in Section 20.2.2, we can consider color in terms of three separate visual channels: hue, saturation, and lightness. Region size strongly affects our ability to sense color. Color in small regions is relatively difficult to perceive, and designers should use bright, highly saturated colors to ensure that the color coding is distinguishable. The inverse situation is true when colored regions are large, as in backgrounds, where low saturation pastel colors should be used to avoid blinding the viewer.

Hue is a very strong cue for encoding categorical data. However, the available dynamic range is very limited. People can reliably distinguish only around a dozen hues when the colored regions are small and scattered around the display. A good guideline for color coding is to keep the number of categories less than eight, keeping in mind that the background and the neutral object color also count in the total.

For ordered data, lightness and saturation are effective because they have an implicit perceptual ordering. People can reliably order by lightness, always placing gray in between black and white. With saturation, people reliably place the less saturated pink between fully saturated red and zero-saturation white. However, hue is not as as good a channel for ordered data because it does not have an implicit perceptual ordering. When asked to create an ordering of red, blue, green, and yellow, people do not all give the same answer. People can and do learn conventions, such as green-yellow-red for traffic lights, or the order of colors in the rainbow, but these constructions are at a higher level than pure perception. Ordered data is typically shown with a discrete set of color values.

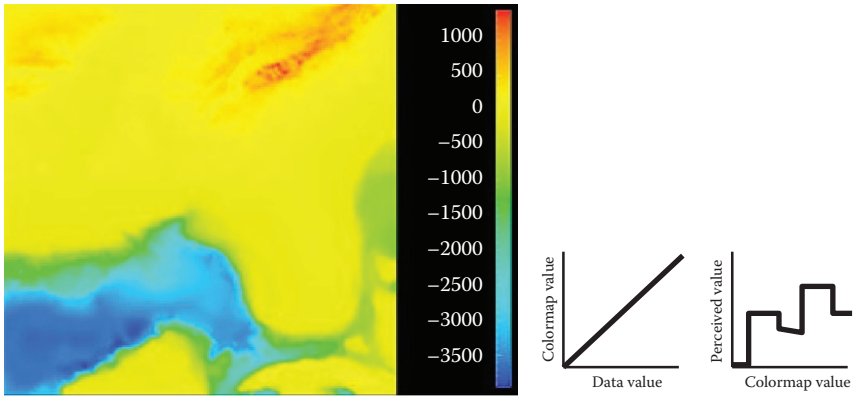

Figure 26.8. The standard rainbow colormap has two defects: it uses hue to denote ordering, and it is not perceptually isolinear. *Image courtesy Bernice Rogowitz.*

Quantitative data is shown with a *colormap*, a range of color values that can be continuous or discrete. A very unfortunate default in many software packages is the rainbow colormap, as shown in Figure 26.8. The standard rainbow scale suffers from three problems. First, hue is used to indicate order. A better choice would be to use lightness because it has an implicit perceptual ordering. Even more importantly, the human eye responds most strongly to luminance. Second, the scale is not perceptually linear: equal steps in the continuous range are not perceived as equal steps by our eyes. Figure 26.8 shows an example, where the rainbow colormap obfuscates the data. While the range from -2000 to -1000 has three distinct colors (cyan, green, and yellow), a range of the same size from -1000 to 0 simply looks yellow throughout. The graphs on the right show that the perceived value is strongly tied to the luminance, which is not even monotonically increasing in this scale.

In contrast, Figure 26.9 shows the same data with a more appropriate colormap, where the lightness increases monotonically. Hue is used to create a semantically meaningful categorization: the viewer can discuss structure in the dataset, such as the dark blue sea, the cyan continental shelf, the green lowlands, and the white mountains.

In both the discrete and continuous cases, colormaps should take into account whether the data is sequential or diverging. The ColorBrewer application (`www.colorbrewer.org`) is an excellent resource for colormap construction (Brewer, 1999).

Another important issue when encoding with color is that a significant fraction of the population, roughly 10% of men, is red-green color deficient. If a coding using red and green is chosen because of conventions in the target domain, re-

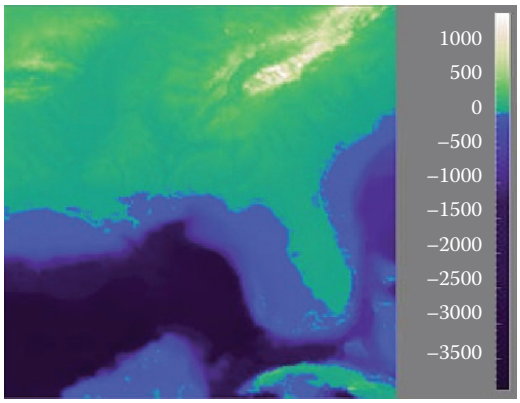

Figure 26.9. The structure of the same dataset is far more clear with a colormap where monotonically increasing lightness is used to show ordering and hue is used instead for segmenting into categorical regions. *Image courtesy Bernice Rogowitz.*

dundantly coding lightness or saturation in addition to hue is wise. Tools such as the website `http://www.vischeck.com` should be used to check whether a color scheme is distinguishable to people with color deficient vision.

26.4.3 2D vs. 3D Spatial Layouts

The question of whether to use two or three channels for spatial position has been extensively studied. When computer-based visualization began in the late 1980s, and interactive 3D graphics was a new capability, there was a lot of enthusiasm for 3D representations. As the field matured, researchers began to understand the costs of 3D approaches when used for abstract datasets (Ware, 2001).

Occlusion, where some parts of the dataset are hidden behind others, is a major problem with 3D. Although hidden surface removal algorithms such as z-buffers and BSP trees allow fast computation of a correct 2D image, people must still synthesize many of these images into an internal mental map. When people look at realistic scenes made from familiar objects, usually they can quickly understand what they see. However, when they see an unfamiliar dataset, where a chosen visual encoding maps abstract dimensions into spatial positions, understanding the details of its 3D structure can be challenging even when they can use interactive navigation controls to change their 3D viewpoint. The reason is once again the limited capacity of human working memory (Plumlee & Ware, 2006).

Another problem with 3D is *perspective distortion*. Although real-world objects do indeed appear smaller when they are further from our eyes, foreshortening makes direct comparison of object heights difficult (Tory, Kirkpatrick, Atkins, & Möller, 2006). Once again, although we can often judge the heights of familiar objects in the real world based on past experience, we cannot necessarily do so with completely abstract data that has a visual encoding where the height conveys meaning. For example, it is more difficult to judge bar heights in a 3D bar chart than in multiple horizontally aligned 2D bar charts.

Another problem with unconstrained 3D representations is that text at arbitrary orientations in 3D space is far more difficult to read than text aligned in the 2D image plane (Grossman, Wigdor, & Balakrishnan, 2007).

Figure 26.10 illustrates how carefully chosen 2D views of an abstract dataset can avoid the problems with occlusion and perspective distortion inherent in 3D views. The top view shows a 3D representation created directly from the original time-series data, where each cross-section is a 2D time-series curve showing power consumption for one day, with one curve for each day of the year along the extruded third axis. Although this representation is straightforward to create, we can only see large-scale patterns such as the higher consumption during working hours and the seasonal variation between winter and summer. To create the 2D linked views at the bottom, the curves were hierarchically clustered, and only aggregate curves representing the top clusters are drawn superimposed in the same 2D frame. Direct comparison between the curve heights at all times of the day is easy because there is no perspective distortion or occlusion. The same color coding is used in the calendar view, which is very effective for understanding temporal patterns.

In contrast, if a dataset consists of inherently 3D spatial data, such as showing fluid flow over an airplane wing or a medical imaging dataset from an MRI scan, then the costs of a 3D view are outweighed by its benefits in helping the user construct a useful mental model of the dataset structure.

26.4.4 Text Labels

Text in the form of labels and legends is a very important factor in creating visualizations that are useful rather than simply pretty. Axes and tick marks should be labeled. Legends should indicate the meaning of colors, whether used as discrete patches or in continuous color ramps. Individual items in a dataset typically have meaningful text labels associated with them. In many cases showing all labels at all times would result in too much visual clutter, so labels can be shown for a subset of the items using label positioning algorithms that show labels at a de-

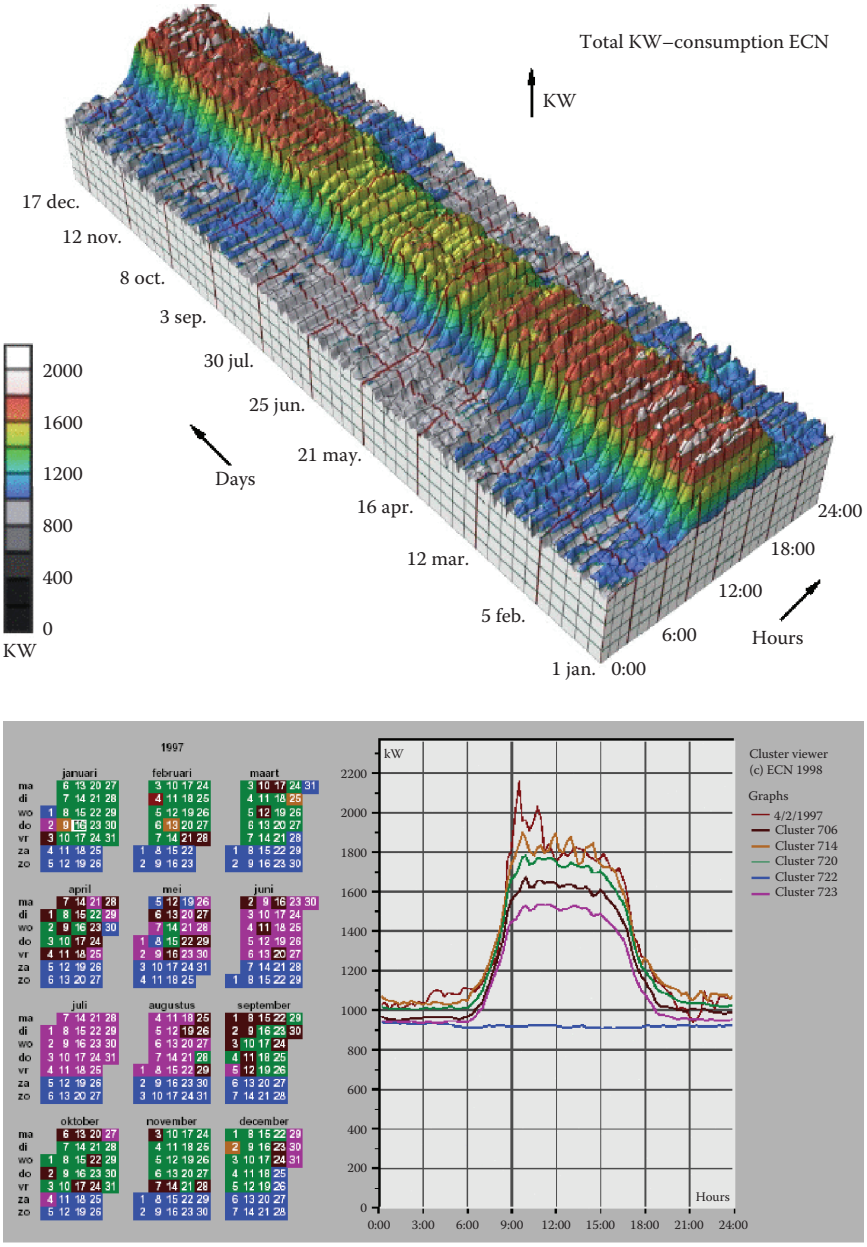

Figure 26.10. Top: A 3D representation of this time series dataset introduces the problems of occlusion and perspective distortion. Bottom: The linked 2D views of derived aggregate curves and the calendar allow direct comparison and show more fine-grained patterns. *Image courtesy Jarke van Wijk* (van Wijk & van Selow, 1999), © 1999 IEEE.

sired density while avoiding overlap (Luboschik, Schumann, & Cords, 2008). A straightforward way to choose the best label to represent a group of items is to use a greedy algorithm based on some measure of label importance, but synthesizing a new label based on the characteristics of the group remains a difficult problem. A more interaction-centric approach is to only show labels for individual items based on an interactive indication from the user.

26.5 Interaction Principles

Several principles of interaction are important when designing a visualization. Low-latency visual feedback allows users to explore more fluidly, for example by showing more detail when the cursor simply hovers over an object rather than requiring the user to explicitly click. Selecting items is a fundamental operation when interacting with large datasets, as is visually indicating the selected set with highlighting. Color coding is a common form of highlighting, but other channels can also be used.

Many forms of interaction can be considered in terms of what aspect of the display they change. Navigation can be considered a change of viewport. Sorting is a change to the spatial ordering; that is, changing how data is mapped to the spatial position visual channel. The entire visual encoding can also be changed.

26.5.1 Overview First, Zoom and Filter, Details on Demand

The influential mantra "Overview first, zoom and filter, details on demand" (Shneiderman, 1996) elucidates the role of interaction and navigation in visualization design. Overviews help the user notice regions where further investigation might be productive, whether through spatial navigation or through filtering. As we discuss below, details can be presented in many ways: with popups from clicking or cursor hovering, in a separate window, and by changing the layout on the fly to make room to show additional information.

26.5.2 Interactivity Costs

Interactivity has both power and cost. The benefit of interaction is that people can explore a larger information space than can be understood in a single static image. However, a cost to interaction is that it requires human time and attention. If the user must exhaustively check every possibility, use of the visualization system

may degenerate into human-powered search. Automatically detecting features of interest to explicitly bring to the user's attention via the visual encoding is a useful goal for the visualization designer. However, if the task at hand could be completely solved by automatic means, there would be no need for a visualization in the first place. Thus, there is always a tradeoff between finding automatable aspects and relying on the human in the loop to detect patterns.

26.5.3 Animation

Animation shows change using time. We distinguish animation, where successive frames can only be played, paused, or stopped, from true interactive control. There is considerable evidence that animated transitions can be more effective than jump cuts, by helping people track changes in object positions or camera viewpoints (Heer & Robertson, 2007). Although animation can be very effective for narrative and storytelling, it is often used ineffectively in a visualization context (Tversky, Morrison, & Betrancourt, 2002). It might seem obvious to show data that changes over time by using animation, a visual modality that changes over time. However, people have difficulty in making specific comparisons between individual frames that are not contiguous when they see an animation consisting of many frames. The very limited capacity of human visual memory means that we are much worse at comparing memories of things that we have seen in the past than at comparing things that are in our current field of view. For tasks requiring comparison between up to several dozen frames, side-by-side comparison is often more effective than animation. Moreover, if the number of objects that change between frames is large, people will have a hard time tracking everything that occurs (Robertson et al., 2008). Narrative animations are carefully designed to avoid having too many actions occurring simultaneously, whereas a dataset being visualized has no such constraint. For the special case of just two frames with a limited amount of change, the very simple animation of flipping back and forth between the two can be a useful way to identify the differences between them.

26.6 Composite and Adjacent Views

A very fundamental visual encoding choice is whether to have a single composite *view* showing everything in the same frame or window, or to have multiple views adjacent to each other.

26.6.1 Single Drawing

When there are only one or two data dimensions to encode, then horizontal and vertical spatial position are the obvious visual channel to use, because we perceive them most accurately and position has the strongest influence on our internal mental model of the dataset. The traditional statistical graphics displays of line charts, bar charts, and scatterplots all use spatial ordering of marks to encode information. These displays can be augmented with additional visual channels, such as color and size and shape, as in the scatterplot shown in Figure 26.4.

The simplest possible mark is a single pixel. In *pixel-oriented* displays, the goal is to provide an overview of as many items as possible. These approaches use the spatial position and color channels at a high information density, but preclude the use of the size and shape channels. Figure 26.11 shows the Tarantula software visualization tool (Jones et al., 2002), where most of the screen is devoted to an overview of source code using one-pixel high lines (Eick, Steffen, & Sumner, 1992). The color and brightness of each line shows whether it passed, failed, or had mixed results when executing a suite of test cases.

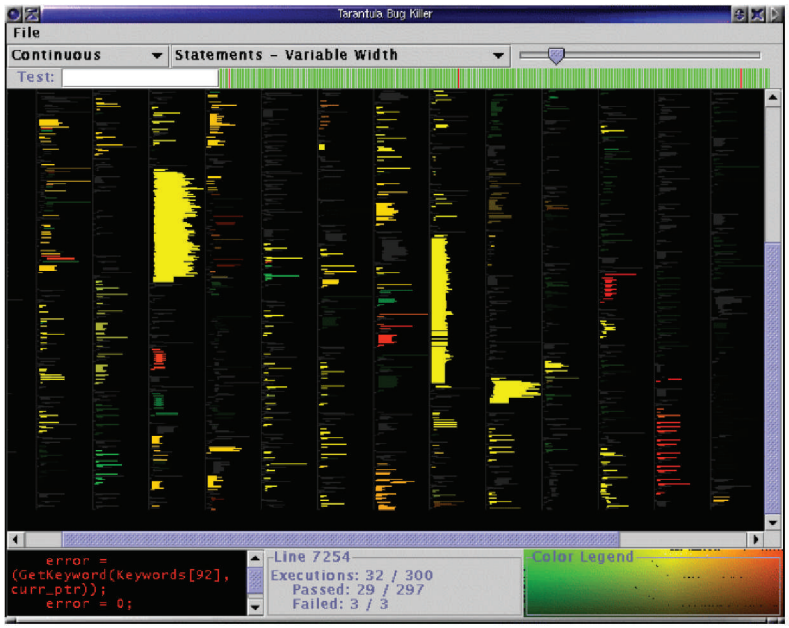

Figure 26.11. Tarantula shows an overview of source code using one-pixel lines color coded by execution status of a software test suite. *Image courtesy John Stasko* (Jones, Harrold, & Stasko, 2002).

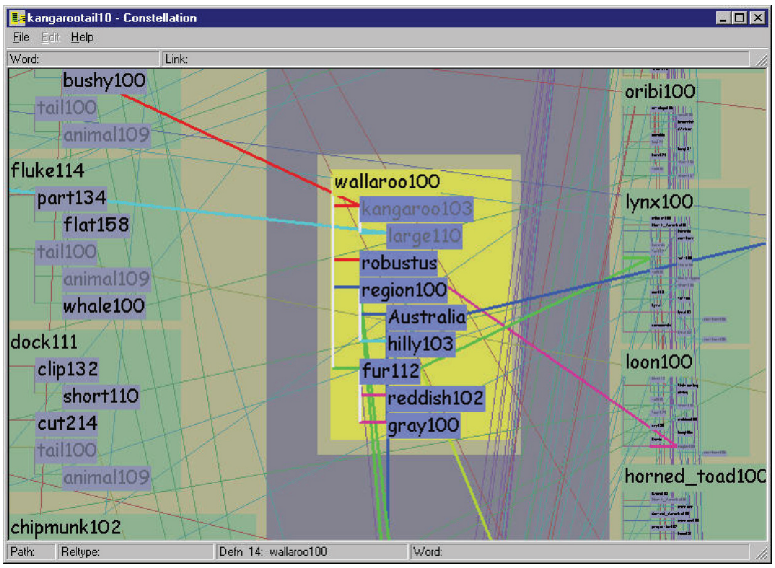

Figure 26.12. Visual layering with size, saturation, and brightness in the Constellation system (Munzner, 2000).

26.6.2 Superimposing and Layering

Multiple items can be superimposed in the same frame when their spatial position is compatible. Several lines can be shown in the same line chart, and many dots in the same scatterplot, when the axes are shared across all items. One benefit of a single shared view is that comparing the position of different items is very easy. If the number of items in the dataset is limited, then a single view will often suffice. Visual layering can extend the usefulness of a single view when there are enough items that visual clutter becomes a concern. Figure 26.12 shows how a redundant combination of the size, saturation, and brightness channels serves to distinguish a foreground layer from a background layer when the user moves the cursor over a block of words.

26.6.3 Glyphs

We have been discussing the idea of visual encoding using simple marks, where a single mark can only have one value for each visual channel used. With more complex marks, which we will call *glyphs*, there is internal structure where subregions have different visual channel encodings.

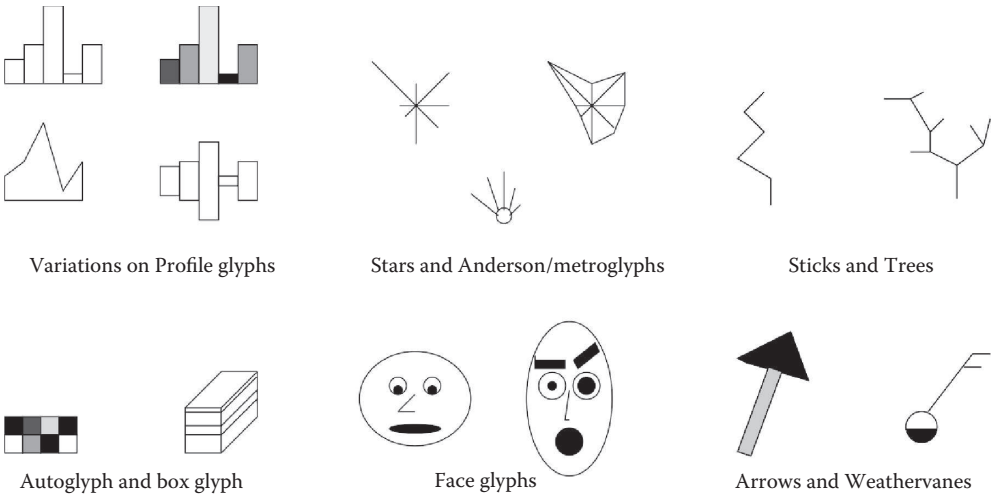

Figure 26.13. Complex marks, which we call *glyphs*, have subsections that visually encode different data dimensions. *Image courtesy Matt Ward* (M. O. Ward, 2002).

Designing appropriate glyphs has the same challenges as designing visual encodings. Figure 26.13 shows a variety of glyphs, including the notorious faces originally proposed by Chernoff. The danger of using faces to show abstract data dimensions is that our perceptual and emotional response to different facial features is highly nonlinear in a way that is not fully understood, but the variability is greater than between the visual channels that we have discussed so far. We are probably far more attuned to features that indicate emotional state, such as eyebrow orientation, than other features, such as nose size or face shape.

Complex glyphs require significant display area for each glyph, as shown in Figure 26.14 where miniature bar charts show the value of four different dimensions at many points along a spiral path. Simpler glyphs can be used to create a global visual texture, the glyph size is so small that individual values cannot be read out without zooming, but region boundaries can be discerned from the overview level. Figure 26.15 shows an example using stick figures of the kind in the upper right in Figure 26.13. Glyphs may be placed at regular intervals, or in data-driven spatial positions using an original or derived data dimension.

26.6.4 Multiple Views

We now turn from approaches with only a single frame to those which use multiple views that are linked together. The most common form of linkage is linked

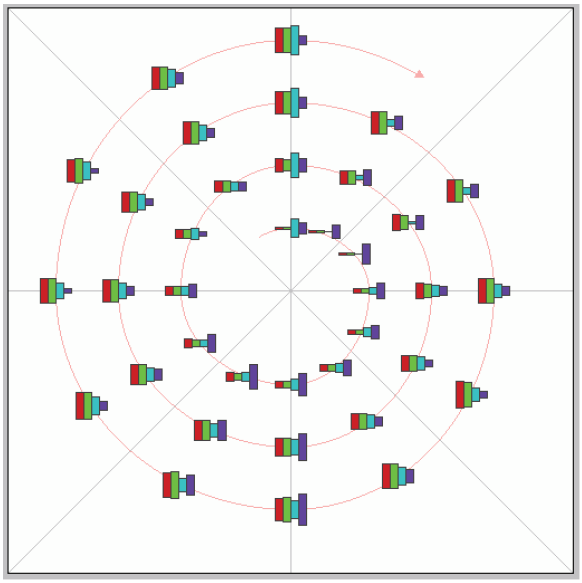

Figure 26.14. Complex glyphs require significant display area so that the encoded information can be read. *Image courtesy Matt Ward, created with the SpiralGlyphics software* (M. O. Ward, 2002).

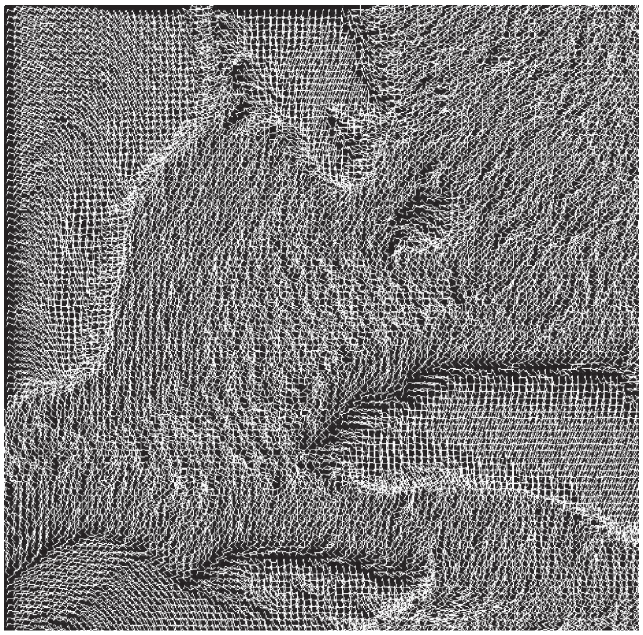

Figure 26.15. A dense array of simple glyphs. *Image courtesy Georges Grinstein* (S. Smith, Grinstein, & Bergeron, 1991), © 1991 IEEE.