Peter Willemsen

18

Using Graphics Hardware

Throughout most of this book, the focus has been on the fundamentals underlying computer graphics rather than on implementation details. This chapter takes a slightly different route and blends the details of using graphics hardware with the practical issues associated with programming that hardware.

This chapter, however, is not written to teach you OpenGL,™ other graphics APIs, or even the nitty gritty specifics of graphics hardware programming. The purpose of this chapter is to introduce the basic concepts and thought processes that are necessary when writing programs that use graphics hardware.

18.1 What Is Graphics Hardware

Graphics hardware describes the hardware components necessary to quickly render 3D objects as pixels on your computer's screen using specialized rasterization-based hardware architectures. The use of this term is meant to elicit a sense of the physical components necessary for performing these computations. In other words, we're talking about the chipsets, transistors, buses, and processors found on many current video cards. As we will see in this chapter, current graphics hardware is very good at processing descriptions of 3D objects and transforming them into the colored pixels that fill your monitor.

One thing has been certain with graphics hardware: it changes very *quickly* with new extensions and features being added continually! One explanation for the fast pace is the video game industry and its economic momentum. Essentially

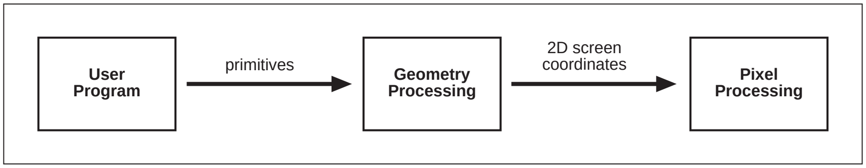


Figure 18.1. The basic graphics hardware pipeline consists of stages that transform 3D data into 2D screen objects ready for rasterizing and coloring by the pixel processing stages.

what this means is that each new graphics card provides better performance and processing capabilities. As a result, graphics hardware is being used for tasks that support a much richer use of 3D graphics. For instance, researchers are performing computation on graphics hardware to perform ray-tracing (Purcell et al., 2002) and even solve the Navier-Stokes equations to simulate fluid flow (Harris, 2004).

Most graphics hardware has been built to perform a set of fixed operations organized as a pipeline designed to push vertices and pixels through different stages. The fixed functionality of the pipeline ensures that basic coloring, lighting, and texturing can occur very quickly—often referred to as *real-time graphics*.

Real-Time Graphics: By real-time graphics, we generally mean that the graphics-related computations are being carried out fast enough that the results can be viewed immediately. Being able to conduct operations at 60Hz is considered real time. Once the time to refresh the display (*frame rate*) drops below 15Hz, the speed is considered more interactive than it is real-time, but this distinction is not critical. Because the computations need to be fast, the equations used to render the graphics are often approximations to what could be done if more time were available.

Figure 18.1 illustrates the real-time graphics pipeline. The important things to note about the pipeline follow:

- The user program, or application, supplies the data to the graphics hardware in the form of *primitives*, such as points, lines, or polygons describing the 3D geometry. Images or bitmaps are also supplied for use in texturing surfaces.

- Geometric primitives are processed on a per-vertex basis and are transformed from 3D coordinates to 2D screen triangles.

- Screen objects are passed to the pixel processors, rasterized, and then colored on a per-pixel basis before being output to the frame buffer, and eventually to the monitor.

18.2 Describing Geometry for the Hardware

As a graphics programmer, you need to be concerned with how the data associated with your 3D objects is transferred onto the memory cache of the graphics hardware. Unfortunately (or maybe fortunately), as a programmer you don't have complete control over this process. There are a variety of ways to place your data on the graphics hardware, and each has its own advantages which will be discussed in this section. Any of the APIs you might use to program your video card will provide different methods to load data onto the graphics hardware memory. The examples that follow are presented in pseudocode that is based loosely on the C function syntax of OpenGL,™ but semantically the examples should be applicable to other graphics APIs.

Most graphics hardware work with specific sets of geometric primitives. The primitive types leverage primitive complexity for processing speed on the graphics hardware. Simpler primitives can be processed very fast. The caveat is that the primitive types need to be general purpose so as to model a wide range of geometry from very simple to very complex. On typical graphics hardware, the primitive types are limited to one or more of the following:

- **points**—single vertices used to represent points or particle systems;
- **lines**—pairs of vertices used to represent lines, silhouettes, or edge-highlighting;
- **polygons**—triangles, triangle strips, indexed triangles, indexed triangle strips, quadrilaterals, general convex polygons, etc., used for describing triangle meshes, geometric surfaces, and other solid objects, such as spheres, cones, cubes, or cylinders.

These three primitives form the basic building blocks for most geometry you will define. (An example of a triangle mesh is shown in Figure 18.2.) Using these primitives, you can build descriptions of your geometry using one of the graphics APIs and send the geometry to the graphics hardware for rendering. For instance,

Primitives: The three primitives (points, lines, and polygons) are the only primitives available! Even when creating spline-based surfaces, such as NURBs, the surfaces are tessellated into triangle primitives by the graphics hardware.

Point Rendering: Point and line primitives may initially appear to be limited in use, but researchers have used points to render very complex geometry (Rusinkiewicz & Levoy, 2000; Dachsbacher et al., 2003).

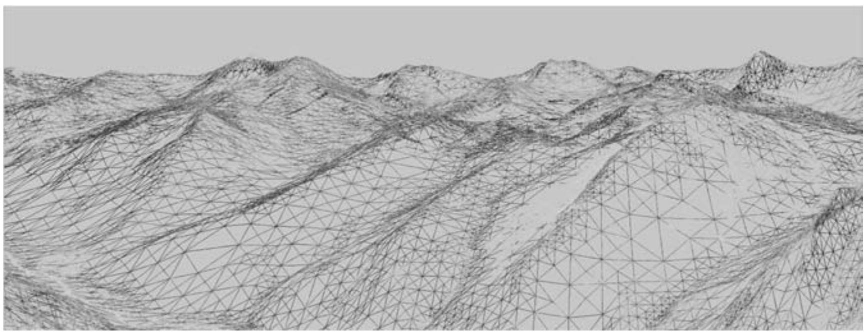


Figure 18.2. How your geometry is organized will affect the performance of your application. This wireframe depiction of the Little Cottonwood Canyon terrain dataset shows tens of thousands of triangles organized in a triangle mesh running at real-time rates. *The image is rendered using the VTerrain Project terrain system courtesy of Ben Discoe.*

to transfer the description of a line to the graphics hardware, we might use the following:

```
beginLine();
  vertex( x1, y1, z1 );
  vertex( x2, y2, z2 );
endLine();
```

In this example, two things occur. First, one of the primitive types is declared and made active by the `beginLine()` function call. The line primitive is then made inactive by the `endLine()` function call. Second, all vertices declared between these two functions are copied directly to the graphics card for processing with the `vertex` function calls.

A second example creates a set of triangles grouped together in a strip (refer to Figure 18.3); we could use the following code:

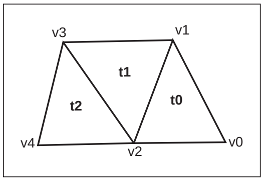


Figure 18.3. A triangle strip composed of five vertices defining three triangles.

```
beginTriangleStrip();
  vertex( x0, y0, z0 );
  vertex( x1, y1, z1 );
  vertex( x2, y2, z2 );
  vertex( x3, y3, z3 );
  vertex( x4, y4, z4 );
endTriangleStrip();
```

In this example, the primitive type, `TriangleStrip`, is made active and the set of vertices that define the triangle strip are copied to the graphics card memory for processing. Note that ordering does matter when describing geometry. In the triangle strip example, connectivity between adjacent triangles is embedded within the ordering of the vertices. Triangle $t0$ is constructed from vertices $(v0, v1, v2)$, triangle $t1$ from vertices $(v1, v3, v2)$, and triangle $t2$ from vertices $(v2, v3, v4)$.

The key point to learn from these simple examples is that geometry is defined for rendering on the graphics hardware using a primitive type along with a set of vertices. The previous examples are simple and push the vertices directly onto the graphics hardware. However, in practice, you will need to make conscious decisions about how you will push your data to the graphics hardware. These issues will be discussed shortly.

As geometry is passed to the graphics hardware, additional data can be specified for each vertex. This extra data is useful for defining *state* attributes, that might represent the color of the vertex, the normal direction at the vertex, texture coordinates at the vertex, or other per-vertex data. For instance, to set the color and normal state parameters at each vertex of a triangle strip, we might use the following code:

```
beginTriangleStrip();
  color( r0, g0, b0 ); normal( n0x, n0y, n0z );
  vertex( x0, y0, z0 );
  color( r1, g1, b1 ); normal( n1x, n1y, n1z );
  vertex( x1, y1, z1 );
  color( r2, g2, b2 ); normal( n2x, n2y, n2z );
  vertex( x2, y2, z2 );
  color( r3, g3, b3 ); normal( n3x, n3y, n3z );
  vertex( x3, y3, z3 );
  color( r4, g4, b4 ); normal( n4x, n4y, n4z );
  vertex( x4, y4, z4 );
endTriangleStrip();
```

Here, the color and normal direction at each vertex are specified just prior to the vertex being defined. Each vertex in this example has a unique color and normal direction. The `color` function sets the active color state using a RGB 3-tuple. The normal direction state at each vertex is set by the `normal` function. Both the `color` and `normal` function affect the current rendering state on the graphics hardware. Any vertices defined after these state attributes are set will be bound with those state attributes.

This is a good moment to mention that the graphics hardware maintains a fairly elaborate set of state parameters that determine how vertices and other components are rendered. Some state is bound to vertices, such as color, normal direction, and texture coordinates, while another state may affect pixel level rendering. The *graphics state* at any particular moment describes a large set of internal hardware parameters. This aspect of graphics hardware is important to consider when you write 3D applications. As you might suspect, making frequent changes to the graphics state affects performance at least to some extent. However, attempting to minimize graphics state changes is only one of many areas where thoughtful programming should be applied. You should attempt to minimize state changes when you can, but it is unlikely that you can group all of your geometry to completely reduce state context switches. One data structure that can help minimize state changes, especially on static scenes, is the scene graph data structure. Prior to rendering any geometry, the scene graph can re-organize the geometry and associated graphics state in an attempt to minimize state changes. Scene graphs are described in Chapter 12.

```
color( r, g, b );
normal( nx, ny, nz );
beginTriangleStrip();
  vertex( x0, y0, z0 );
  vertex( x1, y1, z1 );
  vertex( x2, y2, z2 );
```

```
      vertex( x3, y3, z3 );
      vertex( x4, y4, z4 );
    endTriangleStrip();
```

All vertices in this `TriangleStrip` have the same color and normal direction, so these state parameters can be set prior to defining the vertices. This minimizes both function call overhead and changes to the internal graphics state.

Many things can affect the performance of a graphics program, but one of the potentially large contributors to performance (or lack thereof) is how your geometry is organized and whether it is stored in the memory cache of the graphics card. In the pseudocode examples provided so far, geometry has been pushed onto the graphics hardware in what is often called *immediate mode* rendering. As vertices are defined, they are sent directly to the graphics hardware. The primary disadvantage of immediate mode rendering is that the geometry is sent to the graphics hardware each iteration of your application. If your geometry is static (i.e., it doesn't change), then there is no real need to resend the data each time you redraw a frame. In these and other circumstances, it is more desirable to store the geometry in the graphics card's memory.

The graphics hardware in your computer is connected to the rest of the system via a data bus, such as the PCI, AGP, or PCI-Express buses. When you send data to the graphics hardware, it is sent by the CPU on your machine across one of these buses, eventually being stored in the memory on your graphics hardware. If you have very large triangle meshes representing complex geometry, passing all this data across the bus can end up resulting in a large hit to performance. This is especially true if the geometry is being rendered in immediate mode, as the previous examples have illustrated.

There are various ways to organize geometry; some can help reduce the overall bandwidth needed for transmitting the geometry across the graphics bus. Some possible organization approaches include:

- **triangles**. Triangles are specified with three vertices. A triangle mesh created in this manner requires that each triangle in the mesh be defined separately with many vertices potentially duplicated. For a triangle mesh containing m triangles, $3m$ vertices will be sent to the graphics hardware.

- **triangle strips**. Triangles are organized in strips; the first three vertices specify the first triangle in the strip and each additional vertex adds a triangle. If you create a triangle mesh with m triangles organized as a single triangle strip, you send three vertices to the graphics hardware for the first triangle followed by a single vertex for each additional triangle in the strip for a total of $m + 2$ vertices.

- **indexed triangles**. Triangle vertices are arranged as an array of vertices with a separate array defining the triangles using indices into the vertex array. Vertex arrays are sent to the graphics card with very few function calls.

- **indexed triangle strips**. Similar to indexed triangles, triangle vertices are stored in a vertex array. However, triangles are organized in strips with the index array defining the strip layout. This is the most compact of the organizational structures for defining triangle meshes as it combines the benefits of triangles strips with the compactness of vertex arrays.

Of the different organizational structures, the use of vertex arrays, either through indexed triangles or indexed triangle strips, provides a good option for increasing the performance of your application. The tight encapsulation of the organization means that many fewer function calls need to be made as well. Once the vertices and indices are stored in an array, only a few function calls need to be made to transfer the data to the graphics hardware, whereas with the pseudocode examples illustrated previously, a function is called for each vertex.

At this point, you may be wondering how the graphics state such as colors, normals, or texture coordinates are defined when vertex arrays are used. In the immediate-mode rendering examples earlier in the chapter, interleaving the graphics state with the associated vertices is obvious based on the order of the function calls. When vertex arrays are used, graphics state can either be interleaved in the vertex array or specified in separate arrays that are passed to the graphics hardware.

Even if the geometry is organized efficiently when it is sent to the graphics hardware, you can achieve higher performance gains if you can store your geometry in the graphics hardware's memory for the duration of your application. A somewhat unfortunate fact about current graphics hardware is that many of the specifications describing the layout of the graphics hardware memory and cache structure are often not widely publicized. Fortunately though, there are ways using graphics APIs that allow programmers to place geometry into the graphics hardware memory resulting in applications that run faster.

Two commonly used methods to store geometry and graphics state in the graphics hardware cache involve creating *display lists* or *vertex buffer objects*.

Display lists compile a compact list representation of the geometry and the state associated with the geometry and store the list in the memory on the graphics hardware. The benefits of display lists are that they are general purpose and good at storing a static geometric representation plus associated graphics state on the hardware. They do not work well at all for continuously changing geometry and graphics state, since the display list must be recompiled and then stored *again* in the graphics hardware memory for every iteration in which the display list changes.

```
displayID = createDisplayList();
color( r, g, b );
normal( nx, ny, nz );
beginTriangleStrip();
  vertex( x0, y0, z0 );
  vertex( x1, y1, z1 );
  ...
  vertex( xN, yN, zN );
endTriangleStrip();
endDisplayList();
```

In the above example, a display list is created that contains the definition of a triangle strip with its associated color and normal information. The commands between the `createDisplayList` and `endDisplayList` function calls provide the elements that define the display list. Display lists are most often created during an initialization phase of an application. After the display list is created, it is stored in the memory of the graphics hardware and can be referenced for later use by the identifier assigned to the list.

```
// draw the display list created earlier
drawDisplayList(displayID);
```

When it is time to draw the contents of the display list, a single function call will instruct the graphics hardware to access the memory indexed through the display list identifier and display the contents.

Optimal Organization: Much research effort has gone into looking at ways to optimize triangle meshes for maximum performance on graphics hardware. A good place to start reading if you want to delve further into understanding how triangle mesh organization affects performance is the SIGGRAPH 1999 paper on the optimization of mesh locality (Hoppe, 1999).

A second method to store geometry on the graphics hardware for the duration of your application is through vertex buffer objects (VBOs). VBOs are specialized buffers that reside in high-performance memory on the graphics hardware and store vertex arrays and associated graphics state. They can also provide a mapping from your application to the memory on the graphics hardware to allow for fast access and updating to the contents of the VBO.

The chief advantage of VBOs is that they provide a mapping into the graphics hardware memory. With VBOs, geometry can be modified during an application with a minimal loss of performance as compared with using immediate mode rendering or display lists. This is extremely useful if portions of your geometry change during each iteration of your application or if the indices used to organize your geometry change.

VBOs are created in much the same way indexed triangles and indexed triangle strips are built. A buffer object is first created on the graphics card to make room for the vertex array containing the vertices of the triangle mesh. Next, the vertex array and index array are copied over to the graphics hardware. When it is time to render the geometry, the vertex buffer object identifier can be used to instruct the graphics hardware to draw your geometry. If you are already using vertex arrays in your application, modifying your code to use VBOs should likely require a minimal change.

18.3 Processing Geometry into Pixels

After the geometry has been placed in the graphics hardware memory, each vertex must be lit as well as transformed into screen coordinates during the geometry processing stage. In the fixed-function graphics pipeline illustrated in Figure 18.1, vertices are transformed from a model coordinate system to a screen coordinate frame of reference. This process and the matrices involved are described in Chapters 7 and 8. The modelview and projection matrices needed for this transformation are defined using functions provided with the graphics API you decide to use.

Lighting is calculated on a per-vertex basis. Depending on the global shading parameters, the triangle face will either have a flat-shaded look or the face color will be diffusely shaded (Gouraud shading) by linearly interpolating the color at each triangle vertex across the face of the triangle. The latter method produces a much smoother appearance. The color at each vertex is computed based on the assigned material properties, the lights in the scene, and various lighting parameters.

The lighting model in the fixed-function graphics pipeline is good for fast lighting of vertices; we make a tradeoff for increased speed over accurate illumination. As a result, Phong shaded surfaces are not supported with this fixed-function framework.

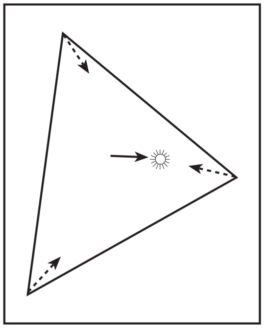


Figure 18.4. The distance to the light source is small relative to the size of the triangle.

In particular, the diffuse shading algorithm built into the graphics hardware often fails to compute the appropriate illumination since the lighting is only being calculated at each vertex. For example, when the distance to the light source is small, as compared with the size of the face being shaded, the illumination on the face will be incorrect. Figure 18.4 illustrates this situation. The center of the triangle will not be illuminated brightly despite being very close to the light source, since the lighting on the vertices, which are far from the light source, are used to interpolate the shading across the face.

With the fixed-function pipeline, this issue can only be remedied by increasing the tessellation of the geometry. This solution works but is of limited use in real-time graphics as the added geometry required for more accurate illumination can result in slower rendering.

However, with current hardware, the problem of obtaining better approximations for illumination can be solved without necessarily increasing the geometric complexity of the objects. The solution involves replacing the fixed-function routines embedded within the graphics hardware with your own programs. These small programs run on the graphics hardware and perform a part of the geometry processing and pixel-processing stages of the graphics pipeline.

18.3.1 Programming the Pipeline

Fairly recent changes to the organization of consumer graphics hardware has generated a substantial buzz from game developers, graphics researchers, and many others. It is quite likely that you have heard about *GPU programming*, *graphics hardware programming*, or even *shader programming*. These terms and the changes in consumer hardware that have spawned them primarily have to do with how the graphics hardware rendering pipeline can now be programmed.

Definition: *Fragment* is a term that describes the information associated with a pixel prior to being processed by the graphics hardware. This definition includes much of the data that might be used to calculate the color of the pixel, such as the pixel's scene depth, texture coordinates, or stencil information.

Specifically, the changes have opened up two specific aspects of the graphics hardware pipeline. Programmers now have the ability to modify how the hardware processes vertices and shades pixels by writing *vertex shaders* and *fragment shaders* (also sometimes referred to as *vertex programs* or *fragment programs*). Vertex shaders are programs that perform the vertex and normal transformations, texture coordinate generation, and per-vertex lighting computations normally computed in the geometry processing stage. Fragment shaders are programs that perform the computations in the pixel processing stage of the graphics pipeline and determine exactly how each pixel is shaded, how textures are applied, and if a pixel should be drawn or not. These small shader programs are sent to the graphics hardware from the user program (see Figure 18.5), but they are executed on the graphics hardware. What this programmability means for

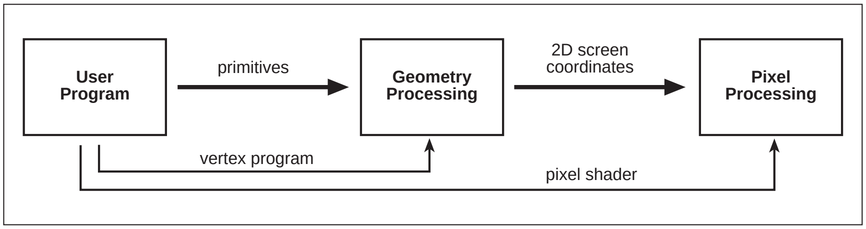


Figure 18.5. The programmable graphics hardware pipeline. The user program supplies primitives, vertex programs, and fragment programs to the hardware.

you is that you essentially have a multi-processor machine. This turns out to be a good way to think about your graphics hardware, since it means that you may be able to use the graphics hardware processor to relieve the load on the CPU in some of your applications. The graphics hardware processors are often referred to as *GPU*s. GPU stands for graphics processing unit and highlights the fact that graphics hardware components now contain a separate processor dedicated to graphics-related computations.

Historical: Programming the pipeline is not entirely new. One of the first introductions of a graphics hardware architecture designed for programming flexibility were the PixelFlow architectures and shading languages from UNC (Molnar et al., 1992; Lastra et al., 1995; Olano & Lastra, 1998). Additional efforts to provide custom shading techniques have included shade trees (Cook, 1984), RenderMan (Pixar, 2000), accelerated multi-pass rendering using OpenGL™ (Peercy et al., 2000), and other real-time shading languages (Proudfoot et al., 2001; McCool et al., 2004).

Interestingly, modern GPUs contain more transistors than modern CPUs. For the time being, GPUs are utilizing most of these transistors for computations and less for memory or cache management operations.

However, this will not always be the case as graphics hardware continues to advance. And just because the computations are geared towards 3D graphics, it does not mean that you cannot perform computations unrelated to computer graphics on the GPU. The manner in which the GPU is programmed is different from your general purpose CPU and will require a slightly modified way of thinking about how to solve problems and program the graphics hardware.

The GPU is a stream processor that excels at 3D vector operations such as vector multiplication, vector addition, dot products, and other operations necessary for basic lighting of surfaces and texture mapping. As stream processors, both the vertex and fragment processing components include the ability to process multiple primitives at the same time. In this regard, the GPU acts as a SIMD (Single Instruction, Multiple Data) processor, and in certain hardware implementations of the fragment processor, up to 16 pixels can be processed at a time. When you write programs for these processing components, it will be helpful, at least conceptually, to think of the computations being performed concurrently on your data. In other words, the vertex shader program will run for all vertices at the same time. The vertex computations will then be followed by a stage in which your fragment shader program will execute simultaneously on all fragments. It is important to note that while the computations on vertices or fragments occur concurrently, the staging of the pipeline components still occur in the same order.

The manner in which vertex and fragment shaders work is simple. You write a vertex shader program and a fragment shader program and send it to the graphics hardware. These programs can be used on specific geometry, and when your geometry is processed, the vertex shader is used to transform and light the vertices, while the fragment shader performs the final shading of the geometry on a per-pixel basis. Just as you can texture map different images onto different pieces of geometry, you can also write different shader programs to act upon different objects in your application. Shader programs are a part of the graphics state so you do need to be concerned with how your shader programs might get swapped in and out based on the geometry being rendered.

The details tend to be a bit more complicated, however. Vertex shaders usually perform two basic actions: set the color at the vertex and transform the vertex into screen coordinates by multiplying the vertex by the modelview and projection matrices. The perspective divide and clipping steps are not performed in a vertex program. Vertex shaders are also often used to set the stage for a fragment shader. In particular, you may have vertex attributes, such as texture coordinates or other application-dependent data, that the vertex shader calculates or modifies and then sends to the fragment processing stage for use in your fragment shader. It may seem strange at first, but vertex shaders can be used to manipulate the positions of the vertices. This is often useful for generating simulated ocean wave motion entirely on the GPU.

In a fragment shader, it is required that the program outputs the fragment color. This may involve looking up texture values and combining them in some manner with values obtained by performing a lighting calculation at each pixel; or, it may involve killing the fragment from being drawn entirely. Because operations in the fragment shader operate at the fragment level, the real power of the programmable graphics hardware is in the fragment shader. This added processing power represents one of the key differences between the fixed function pipeline and the programmable pipeline. In the fixed pipeline, fragment processing used illumination values interpolated between the vertices of the triangle to compute the fragment color. With the programmable pipeline, the color at each fragment can be computed independently. For instance, in the example situation posed in Figure 18.4, Gouraud shading of a triangle face fails to produce a reasonable solution because lighting only occurs at the vertices which are farther away from the light than the center of the triangle. In a fragment shader, the lighting equation can be evaluated at each fragment, rather than at each vertex, resulting in a more accurate rendering of the face.

18.3.2 Basic Execution Model

When writing vertex or fragment shaders, there are a few important things to understand in terms of how vertex and fragment programs execute and access data on the GPU. Because these programs run entirely on the GPU, the first details you will need to figure out are which data your shaders will use and how to get that data to them. There are several characteristics associated with the data types used in shader programs. The following terms, which come primarily from the OpenGL™ Shading Language framework, are used to describe the conceptual aspects of these data characteristics. The concepts are the same across different shading language frameworks. In the shaders you write, variables are characterized using one of the following terms:

- **attributes**. Attribute variables represent data that changes frequently, often on a per-vertex basis. Attribute variables are often tied to the changing graphics state associated with each vertex. For instance, normal vectors or texture coordinates are considered to be attribute data since they are part of the graphics state associated with each vertex.

- **uniforms**. Uniform variables represent data that cannot change during the execution of a shader program. However, uniform variables can be modified by your application between executions of a shader. This provides another way for your application to communicate data to a shader. Uniform data often represent the graphics state associated with an application. For instance, the modelview and projection matrices can be accessed through uniform variables. Information about light sources in your application can also be obtained through uniform variables. In these examples, the data does not change while the shader is executing, but could (e.g., the light could move) prior to the next iteration of the application.

- **varying**. Varying data is used to pass data between a vertex shader and a fragment shader. The reason the data is considered *varying* is because it is written by vertex shaders on a per-vertex basis, but read by fragment shaders as value interpolated across the face of the primitive between neighboring vertices.

Variables defined using one of these three characteristics can either be built-in variables or user-defined variables. In addition to accessing the built-in graphics state, attribute and uniform variables are one of the ways to communicate user-defined data to your vertex and fragment programs. Varying data is the only means to pass data from a vertex shader to a fragment shader. Figure 18.6 illustrates the basic execution of the vertex and fragment processors in terms of the inputs and outputs used by the shaders.

Another way to pass data to vertex and fragment shaders is by using texture maps as sources and sinks of data. This may come as a surprise if you have been thinking of texture maps solely as images that are applied to the outside surface of geometry. The reason texture maps are important is because they give you access to the memory on the graphics hardware. When you write applications that run on the CPU, you control the memory your application requires and have direct access to it when necessary. On graphics hardware, memory is not accessed in the same manner. In fact, you are not directly able to allocate and deallocate general purpose memory chunks, and this particular aspect usually requires a slight change in thinking.

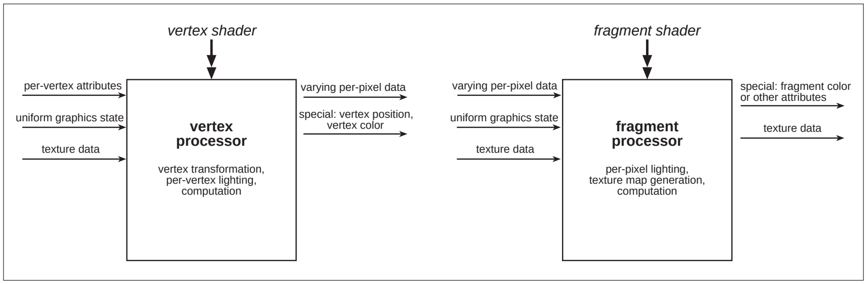


Figure 18.6. The execution model for shader programs. Input, such as per-vertex attributes, graphics state-related uniform variables, varying data, and texture maps are provided to vertex and fragment programs within the shader processor. Shaders output special variables used in later parts of the graphics pipeline.

Texture maps on graphics hardware, however, can be created, deleted, and controlled through the graphics API you use. In other words, for general data used by your shader, you will create texture maps that contain that data and then use texture access functions to look up the data in the texture map. Technically, textures can be accessed by both vertex and fragment shaders. However, in practice, texture lookups from the vertex shader are not currently supported on all graphics cards. An example that utilizes a texture map as a data source is bump mapping. Bump mapping uses a normal map which defines how the normal vectors change across a triangle face. A bump mapping fragment shader would look up the normal vector in the normal map "texture data" and use it in the shading calculations at that particular fragment.

Note: The shader language examples used in this chapter are presented using GLSL (OpenGL™ Shading Language). This language was chosen since it is being developed by the OpenGL™ Architecture Review Board and will likely become a standard shading language for OpenGL™ with the release of OpenGL™ 2.0. As of this writing, GLSL can be used on most modern graphics cards with updated graphics hardware drivers.

You need to be concerned about the types of data you put into your texture maps. Not all numerical data types are well supported and only recently has graphics hardware included floating point textures with 16-bit components. Moreover, *none* of the computation being performed on your GPU is done with double-precision math! If numerical precision is important for your application, you will need to think through these issues very carefully to determine if using the graphics hardware for computation is useful.

So what do these shader programs look like? One way to write vertex and fragment shaders is through assembly language instructions. For instance, performing a matrix multiplication in shader assembly language looks something like this:

```
DP4 p[0].x, M[0], v[0];
DP4 p[0].y, M[1], v[0];
DP4 p[0].z, M[2], v[0];
DP4 p[0].w, M[3], v[0];
```

In this example, the `DP4` instruction is a 4-component dot product function. It stores the result of the dot product in the first register and performs the dot product between the last two registers. In shader programming, registers hold 4-components corresponding to the x, y, z, and w components of a homogeneous coordinate, or the r, g, b, and a components of a RGBA tuple. So, in this example, a simple matrix multiplication,

$$\mathbf{p} = \mathbf{M}\mathbf{v}$$

is computed by four `DP4` instructions. Each instruction computes one element of the final result.

Fortunately though, you are not forced to program in assembly language. The good news is that higher-level languages are available to write vertex and fragment shaders. NVIDIA's Cg, the OpenGL™ Shading Language (GLSL), and Microsoft's High Level Shading Language (HLSL) all provide similar interfaces to the programmable aspects of graphics hardware. Using the notation of GLSL, the same matrix multiplication performed above looks like this:

```
p = M * v;
```

where $\mathbf{p}$ and $\mathbf{v}$ are vertex data types and $\mathbf{M}$ is a matrix data type. As evidenced here, one advantage of using a higher-level language over assembly language is that various data types are available to the programmer. In all of these languages, there are built-in data types for storing vectors and matrices, as well as arrays and constructs for creating structures. Many different functions are also built in to these languages to help compute trigonometric values (sin, cos, etc...), minimum and maximum values, exponential functions (log2, sqrt, pow, etc...), and other math or geometric-based functions.

18.3.3 Vertex Shader Example

Vertex shaders give you control over how your vertices are lit and transformed. They are also used to set the stage for fragment shaders. An interesting aspect to vertex shaders is that you still are able to use geometry-caching mechanisms, such as display lists or VBOs, and thus, benefit from their performance gains while using vertex shaders to do computation on the GPU. For instance, if the vertices represent particles and you can model the movement of the particles using a vertex shader, you have nearly eliminated the CPU from these computations. Any bottleneck in performance that may have occurred due to data being passed between the CPU and the GPU will be minimized. Prior to the introduction of vertex shaders, the computation of the particle movement would have been performed on the CPU and each vertex would have been re-sent to the graphics hardware on each iteration of your application. The ability to perform computations on the vertices already stored in the graphics hardware memory is a big performance win.

One of the simplest vertex shaders transforms a vertex into clip coordinates and assigns the front-facing color to the color attribute associated with the vertex.

```
void main(void)
{
    gl_Position = gl_ModelViewProjectionMatrix *
                     gl_Vertex;
    gl_FrontColor = gl_Color;
}
```

In this example, `gl_ModelViewProjectionMatrix` is a built-in uniform variable supplied by the GLSL run-time environment. The variables `gl_Vertex` and `gl_Color` are built-in vertex attributes; the special output variables, `gl_Position` and `gl_FrontColor` are used by the vertex shader to set the transformed position and the vertex color.

A more interesting vertex shader that implements the surface- shading equations developed in Chapter 10 illustrates the effect of per-vertex shading using the Phong shading algorithm.

```
void main(void)
{
    vec4 v = gl_ModelViewMatrix * gl_Vertex;
    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
    vec3 l = normalize(gl_LightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = gl_LightSource[0].diffuse;
    vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
        color = cr * (ca + cl * max(0,dot(n,l))) +
                  cl * pow(dot(h,n), p);
    else
        color = cr * (ca + cl * max(0,dot(n,l)));

    gl_FrontColor = color;
    gl_Position = ftransform();
}
```

From the code presented in this shader, you should be able to gain a sense of shader programming and how it resembles C-style programming. Several things are happening with this shader. First, we create a set of variables to hold the vectors necessary for computing Phong shading: $\mathbf{v}, \mathbf{n}, \mathbf{l}$, and $\mathbf{h}$. Note that the computation in the vertex shader is performed in *eye-space*. This is done for a variety of reasons, but one reason is that the light-source positions accessible within a shader have already been transformed into the eye coordinate frame. When you create shaders, the coordinate system that you decide to use will likely depend on the types of computations being performed; this is an important factor to consider. Also, note the use of built-in functions and data structures in the example. In particular, there are several functions used in this shader: `normalize`, `dot`, `max`, `pow`, and `ftransform`. These functions are provided with the shader language. Additionally, the graphics state associated with materials and lighting can be accessed through built-in uniform variables: `gl_FrontMaterial` and `gl_LightSource[0]`. The diffuse component of the material and light is accessed through the `diffuse` member of these variables. The color at the vertex is computed using Equation (10.8) and then stored in the special output variable `gl_FrontColor`. The vertex position is transformed using the func-

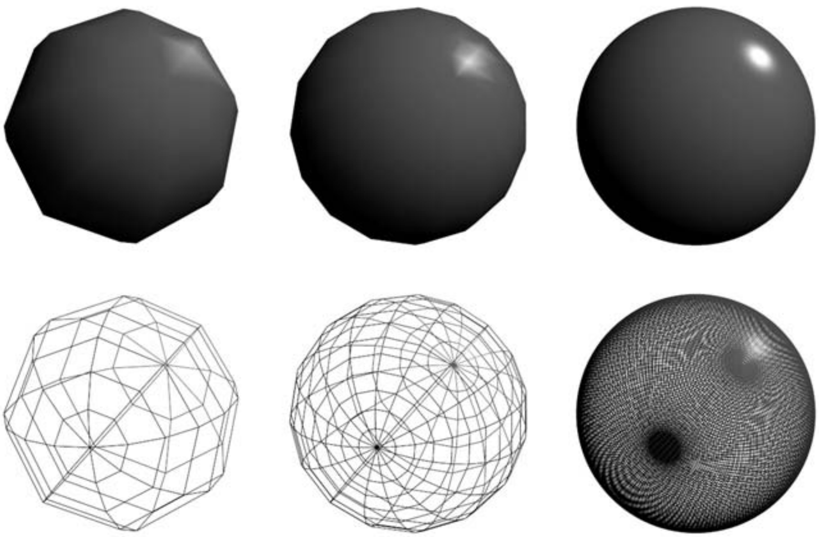


Figure 18.7. Each sphere is rendered using only a vertex shader that computes Phong shading. Because the computation is being performed on a per-vertex basis, the Phong highlight only begins to appear accurate after the amount of geometry used to model the sphere is increased drastically. (See also Plate VIII.)

tion `ftransform` which is a convenience function that performs the multiplication with the modelview and projection matrices. Figure 18.7 shows the results from running this vertex shader with differently tessellated spheres. Because the computations are performed on a per-vertex basis, a large amount of geometry is required to produce a Phong highlight on the sphere that appears correct.

18.3.4 Fragment Shader Example

Fragment shaders are written in a manner very similar to vertex shaders, and to emphasize this, Equation (10.8) from will be implemented with a fragment shader. In order to do this, we first will need to write a vertex shader to set the stage for the fragment shader.

The vertex shader required for this example is fairly simple, but introduces the use of *varying* variables to communicate data to the fragment shader.

```
varying vec4 v;
varying vec3 n;

void main(void)
{
    v = gl_ModelViewMatrix * gl_Vertex;
    n = normalize(gl_NormalMatrix * gl_Normal);

    gl_Position = ftransform();
}
```

Recall that varying variables will be set on a per-vertex basis by a vertex shader, but when they are accessed in a fragment shader, the values will *vary* (i.e., be interpolated) across the triangle, or geometric primitive. In this case, the vertex position in eye-space $\mathbf{v}$ and the normal at the vertex $\mathbf{n}$ are calculated at each vertex. The final computation performed by the vertex shader is to transform the vertex into clip coordinates since the fragment shader will compute the lighting at each fragment. It is not necessary to set the front-facing color in this vertex shader.

The fragment shader program computes the lighting at each fragment using the Phong shading model.

```
varying vec4 v;
varying vec3 n;

void main(void)
{
```

```
        vec3 l = normalize(gl_LightSource[0].position - v);
        vec3 h = normalize(l - normalize(v));

        float p = 16;
        vec4 cr = gl_FrontMaterial.diffuse;
        vec4 cl = gl_LightSource[0].diffuse;
        vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

        vec4 color;
        if (dot(h,n) > 0)
           color = cr * (ca + cl * max(0,dot(n,l))) +
                      cl * pow(dot(h,n),p);
        else
           color = cr * (ca + cl * max(0,dot(n,l)));

        gl_FragColor = color;
    }
```

The first thing you should notice is the similarity between the fragment shader code in this example and the vertex shader code presented in Section 18.3.3. The

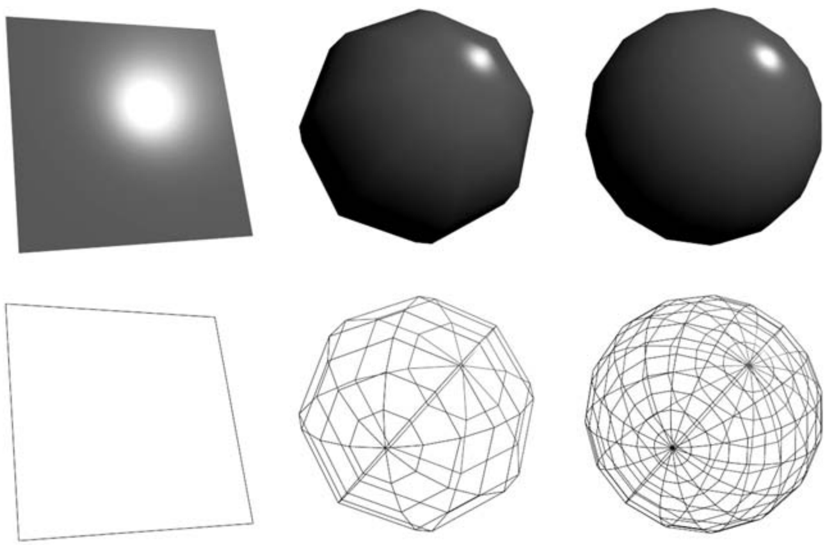


Figure 18.8. The results of running the fragment shader from Section 18.3.4. Note that the Phong highlight does appear on the left-most model which is represented by a single polygon. In fact, because lighting is calculated at the fragment, rather than at each vertex, the more coarsely tessellated sphere models also demonstrate appropriate Phong shading. (See also Plate IX.)

main difference is in the use of the varying variables, $\mathbf{v}$ and $\mathbf{n}$. In the fragment shader, the view vectors and normal values are interpolated across the surface of the model between neighboring vertices. The results are shown in Figure 18.8. Immediately, you should notice the Phong highlight on the quadrilateral, which only contains four vertices. Because the shading is being calculated at the fragment level using the Phong equation with the interpolated (i.e., varying) data, more consistent and accurate Phong shading is produced with far less geometry.

18.3.5 General Purpose Computing on the GPU

After studying the vertex and fragment shader examples, you may be wondering if you can write programs to perform other types of computations on the GPU. Obviously, the answer is yes, as many problems can be coded to run on the GPU given the various languages available for programming on the GPU. However, a few facts are important to remember. Foremost, floating point math processing on graphics hardware is not currently double-precision. Secondly, you will likely need to transform your problem into a form that fits within a graphics-related framework. In other words, you will need to use the graphics APIs to set up the problem, use texture maps as data rather than traditional memory, and write vertex and fragment shaders to frame and solve your problem.

Having stated that, the GPU may still be an attractive platform for computation, since the ratio of transistors that are dedicated to performing computation is much higher on the GPU than it is on the CPU. In many cases, algorithms running on GPUs run faster than on a CPU. Furthermore, GPUs perform SIMD computation, which is especially true at the fragment-processing level. In fact, it can often help to think about the computation occurring on the fragment processor as a highly parallel version of a generic `foreach` construct, performing simultaneous operations on a set of elements.

There has been a large amount of investigation to perform General Purpose computation on GPUs, often referred to as GPGPU. Among other things, researchers are using the GPU as a means to simulate the dynamics of clouds (Harris et al., 2003), implement ray tracers (Purcell et al., 2002; N. A. Carr et al., 2002), compute radiosity (Coombe et al., 2004), perform 3D segmentation using level sets (A. E. Lefohn et al., 2003), or solve the Navier-Stokes equations (Harris, 2004).

General purpose computation is often performed on the GPU using multiple rendering “passes,” and most computation is done using the fragment processor due to its highly data-parallel setup. Each pass, called a *kernel*, completes a portion of the computation. Kernels work on streams of data with several kernels strung together to form the overall computation. The first kernel completes the first part of the computation, the second kernel works on the first kernel's data, and so on, until the calculation is complete. In this style of programming, working with data and data structures on the GPU is different than conventional programming and does require a bit of thought. Fortunately, recent efforts are providing abstractions and information for creating efficient data structures for GPU programming (A. Lefohn et al., 2005).

Using the GPU for general purpose programming does require that you understand how to program the graphics hardware. For instance, most applications that perform GPGPU will render a simple quadrilateral, or sets of quadrilaterals, with vertex and fragment shaders operating on that geometry. The geometry doesn't have to be visible, or drawn to the screen, but it is necessary to allow the vertex and fragment operations to occur. This focus on graphics does make the learning curve for general purpose computing on this hardware an adventure. Fortunately, recent efforts are working to make the interface to the GPU more like traditional programming. The Brook for GPUs project (Buck et al., n.d.) is a system that provides a C-like interface to afford stream computations on the GPU, which should allow more people to take advantage of the computational power on modern graphics hardware.

Frequently Asked Questions

- How do I debug shader programs?

On most platforms, debugging both vertex shaders and fragment shaders is not simple. There is very little runtime support for debugging graphics applications in general, and even less available for runtime debugging of shader programs. However, this is starting to change. In the latest versions of Mac OS X, Linux, and Windows, support for shader programming is incorporated. A good solution for debugging shader programs is to use one of the shader development tools available from various graphics hardware manufacturers.

Notes

There are many good resources available to learn more about the technical details involved with programming graphics hardware. A good starting point might be the OpenGL™ Programming Guide (Shreiner et al., 2004). The OpenGL™ Shading Language (Rost, 2004) and The Cg Tutorial (Fernando & Killgard, 2003) provide details on how to program using a shading language. More advanced technical information and examples for programming the vertex and fragment processors can be found in the GPU Gems series of books (Fernando, 2004; Pharr & Fernando, 2005). A source of information for learning more about general purpose computation on GPUs (GPGPU) can be found on the GPGPU.org web site (http://www.gpgpu.org).

Exercises

1. How fast is the GPU as compared to performing the operations on the CPU? Write a program in which you can parameterize how much data is processed on the GPU, ranging from no computation using a shader program to all of the computation being performed using a shader program. How does the performance of you application change when the computation is being performed solely on the GPU?

2. Are there sizes of triangle strip lengths that work better than others? Try to determine the maximum size of a triangle strip that maximizes performance. What does this tell you about the memory, or cache structure, on the graphics hardware?

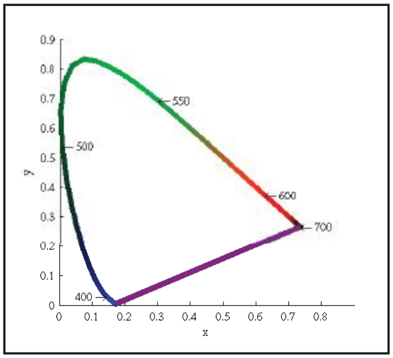


Plate XXVIII. The spectrum locus for the CIE 1931 standard observer. (See also Figure 21.6).

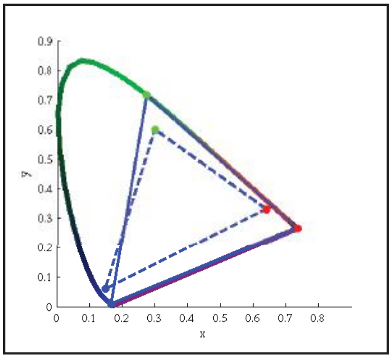


Plate XXIX. The chromaticity boundaries of the CIE RGB primaries at 435.8, 546.1, and 700 nm (solid) and a typical HDTV (dashed). (See also Figure 21.7.)

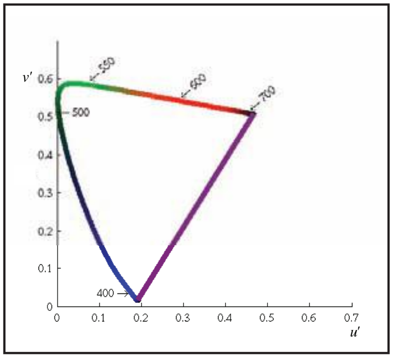


Plate XXX. The CIE $u'v'$ chromaticity diagram. (See also Figure 21.8.)

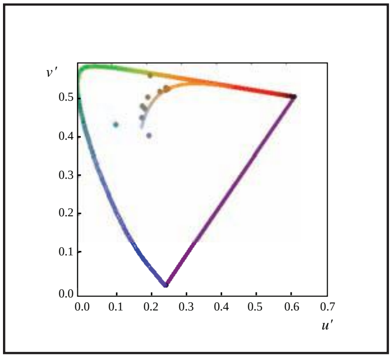


Plate XXXI. A series of light sources plotted in the CIE $u'v'$ chromaticity diagram. A white piece of paper illuminated by any of these light sources maintains a white color appearance. (See also Figure 21.11.)

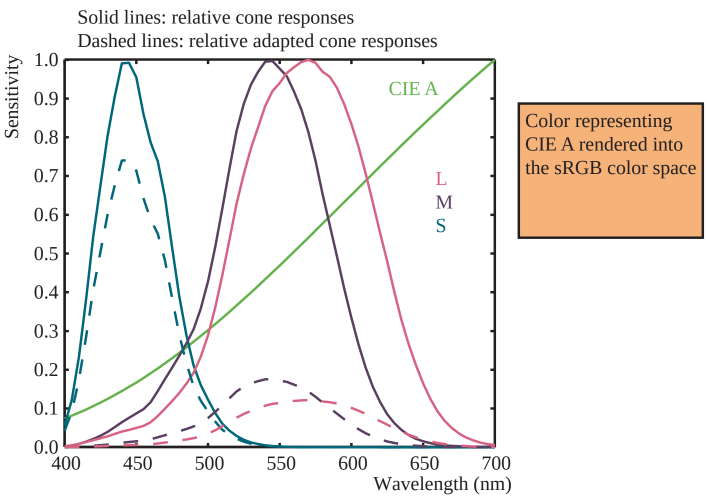


Plate XXXII. An example of von Kries–style independent photoreceptor gain control. The relative cone responses (solid line) and the relative adapted cone responses to CIE illuminant A (dashed) are shown. The separate patch of color represents CIE illuminant A rendered into the sRGB color space. (See also Figure 21.12.)

Plate XXXIII. *Crysis* exemplifies the realistic and detailed graphics expected of first-person shooters. *Image courtesy Crytek.* (See also Figure 26.2.)

Plate XXXIV. An example of highly stylized, nonphotorealistic rendering from the game *Okami*. *Image courtesy Capcom Entertainment, Inc.* (See also Figure 26.3.)

Plate XXXV. 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.* (See also Figure 26.4.)

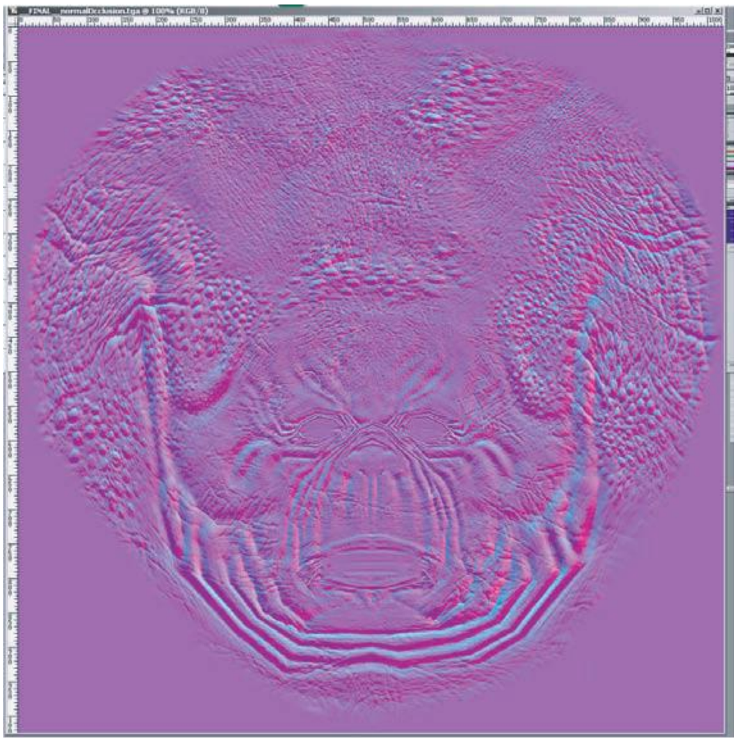


Plate XXXVI. The normal map used in Figure 26.8. In this image, the red, green and blue channels of the texture contain the X, Y, and Z coordinates of the surface normals. *Image courtesy Keith Bruns.* (See also Figure 26.9.)

Plate XXXVII. An early version of a diffuse color texture for the mesh from Figure 26.8, shown in Photoshop. *Image courtesy Keith Bruns.* (See also Figure 26.10.)

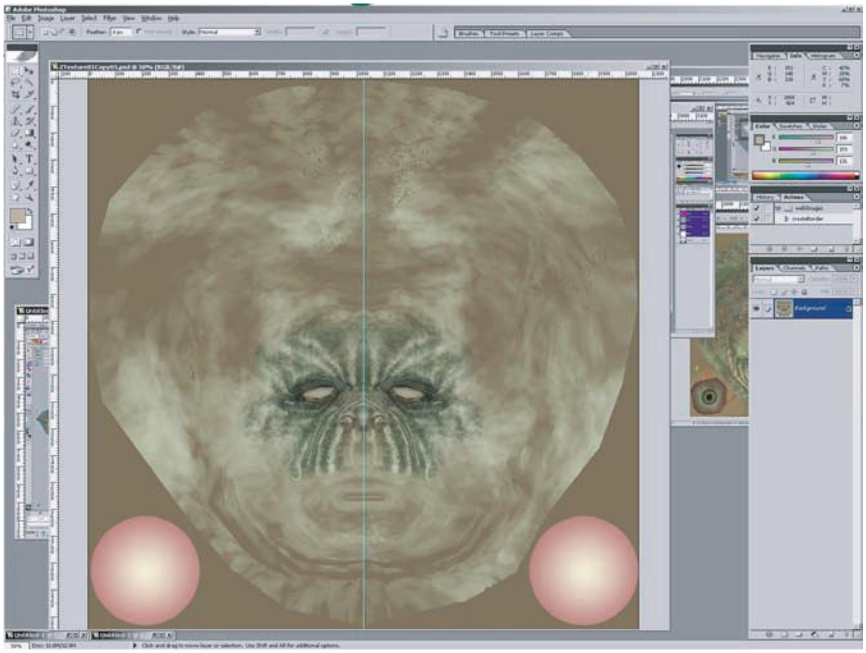
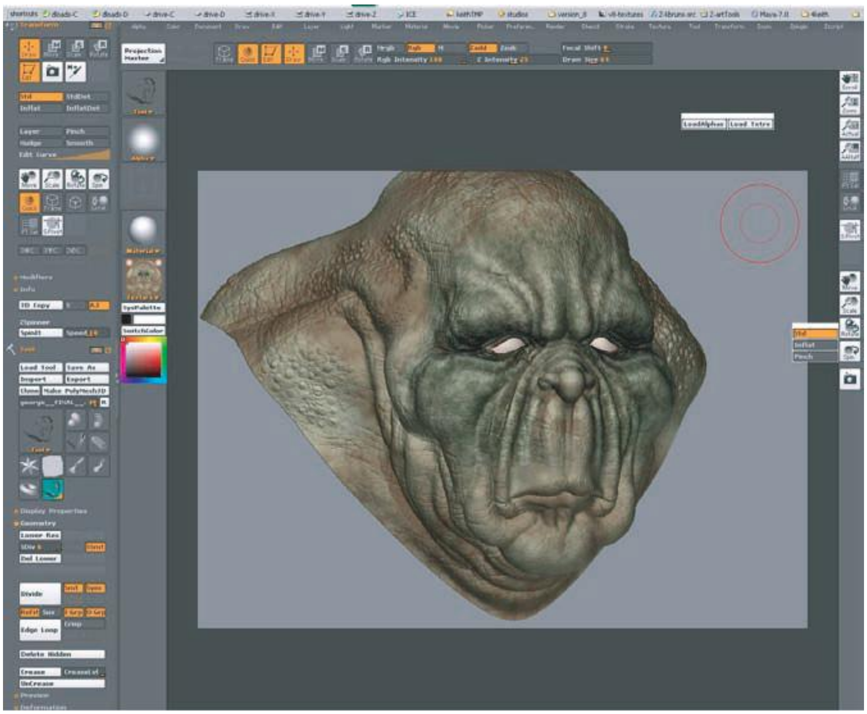


Plate XXXVIII. A rendering (in ZBrush) of the mesh with normal map and early diffuse color texture (from Plate XXXVII) applied. *Image courtesy Keith Bruns.* (See also Figure 26.11.)



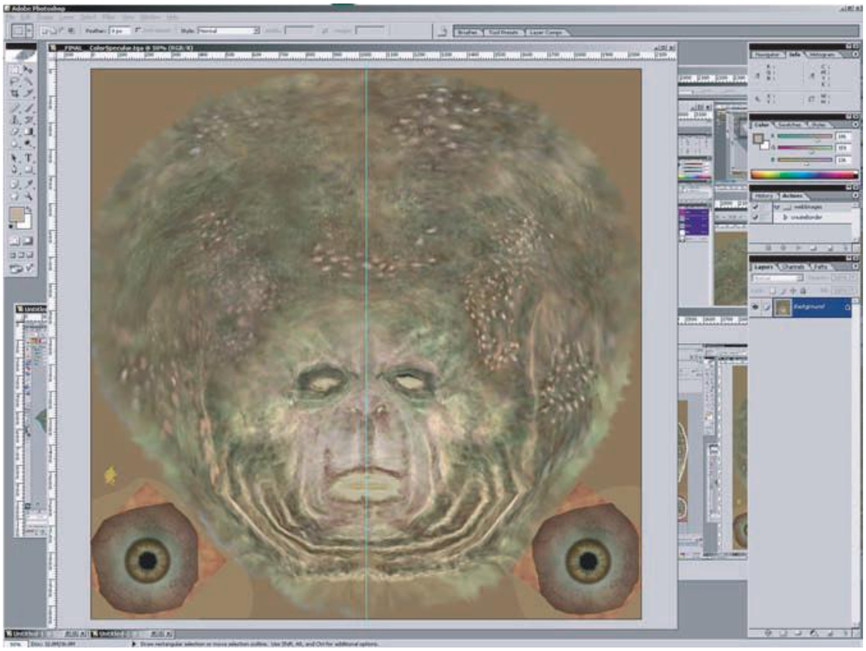


Plate XXXIX. Final version of the color texture from Plate XXXVII. *Image courtesy Keith Bruns.* (See also Figure 26.12.)

Plate XL. Rendering of the mesh with normal map and final color texture (from Figure 26.12) applied. *Image courtesy Keith Bruns.* (See also Figure 26.13.)

Plate XLI. 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.* (See also Figure 26.14.)

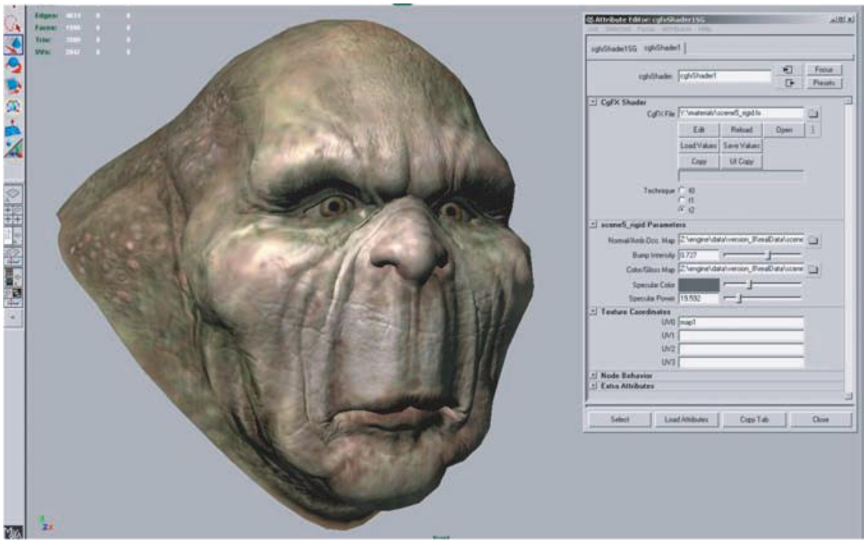


Plate XLII. The Tableau/Polaris system default mappings for four visual channels according to data type. *Image courtesy Chris Stolte* (Stolte et al., 2008), © 2008 IEEE. (See also Figure 27.6.)

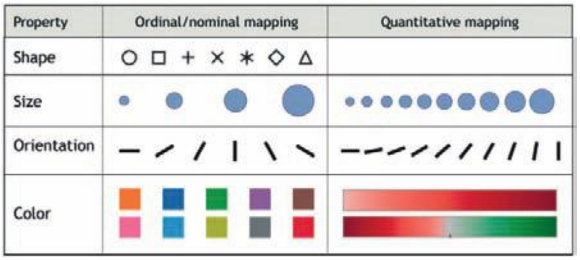
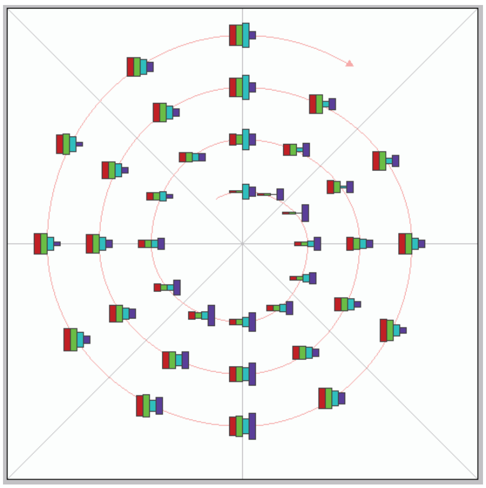


Plate XLIII. Complex glyphs require significant display area so that the encoded information can be read. *Image courtesy Matt Ward, created with the SpiralGlyphics software* (M. O. Ward, 2002). (See also Figure 27.14.)



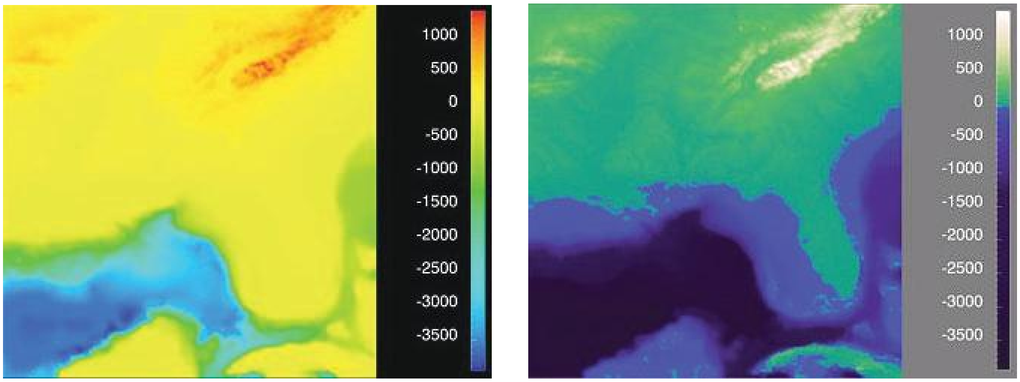


Plate XLIV. Left: The standard rainbow colormap has two defects: it uses hue to denote ordering, and it is not perceptually isolinear. (See also Figure 27.8.) Right: The structure of the same dataset is far more clear with a colormap where monotonically increasing lightness is used to show ordering and hue is used instead for segmenting into categorical regions. (See also Figure 27.9.) *Courtesy Bernice Rogowitz.*

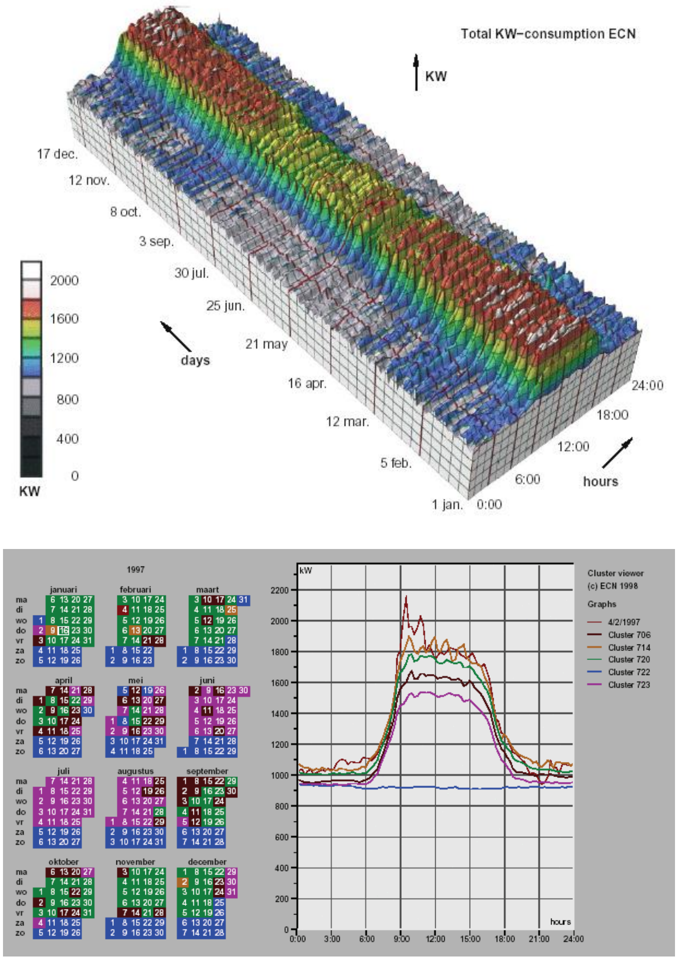


Plate XLV. Top: A 3D representation of this time series dataset introduces the problems of occlusion and perspective distortion. Bottom: The linked 2D views of derived aggregate curves and the calendar allow direct comparison and show more fine-grained patterns. *Image courtesy Jarke van Wijk* (van Wijk & van Selow, 1999), © 1999 IEEE. (See also Figure 27.10.)

Plate XLVI. 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 et al., 2002), © 2002 ACM, Inc. Included here by permission. (See also Figure 27.11.)

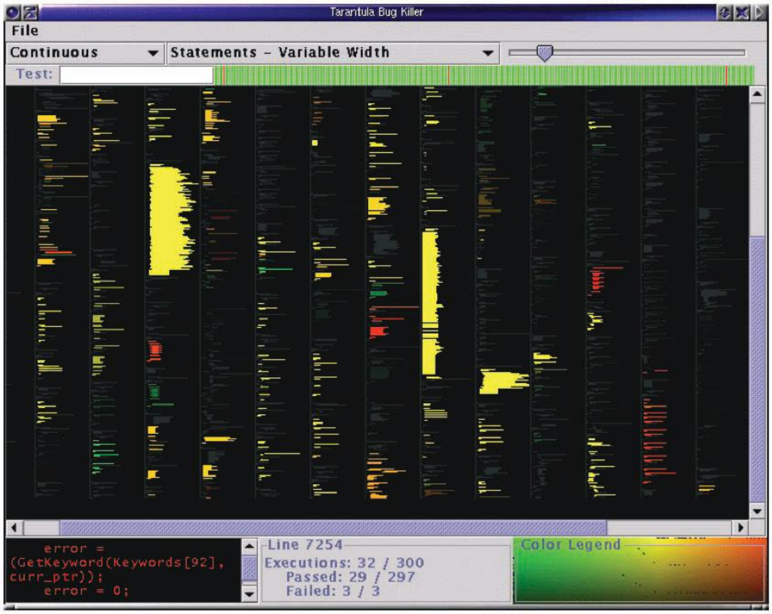
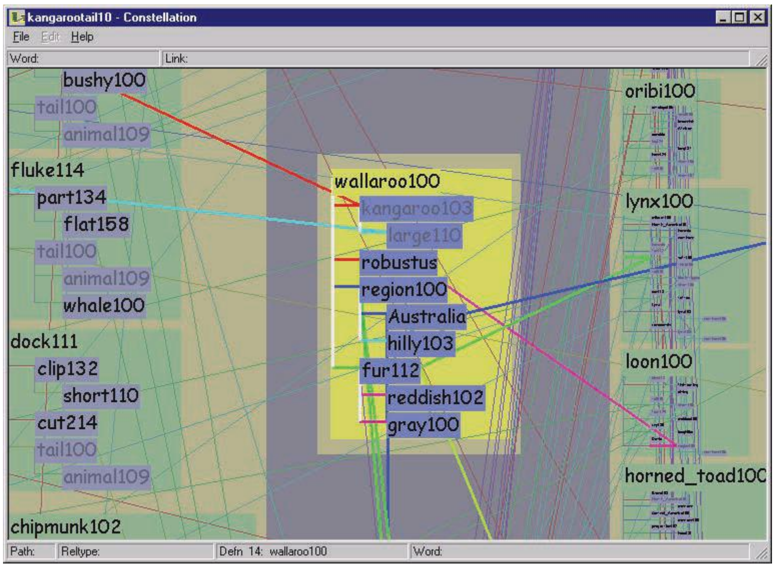


Plate XLVII. Visual layering with size, saturation, and brightness in the Constellation system (Munzner, 2000). (See also Figure 27.12.)



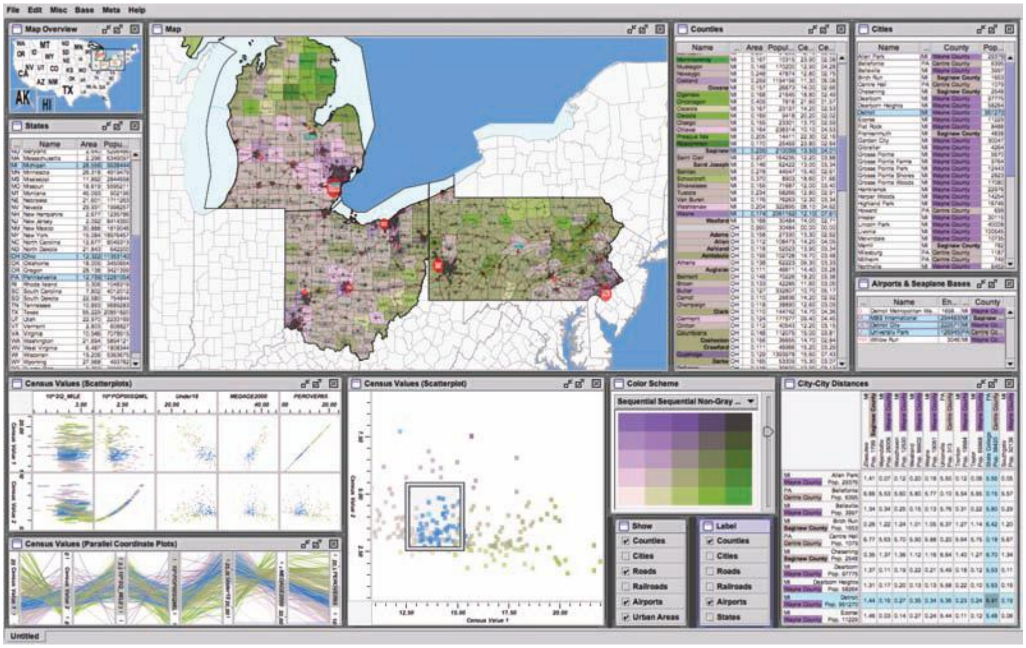


Plate XLVIII. The Improvise toolkit was used to create this multiple-view visualization. *Image courtesy Chris Weaver.* (See also Figure 27.16.)

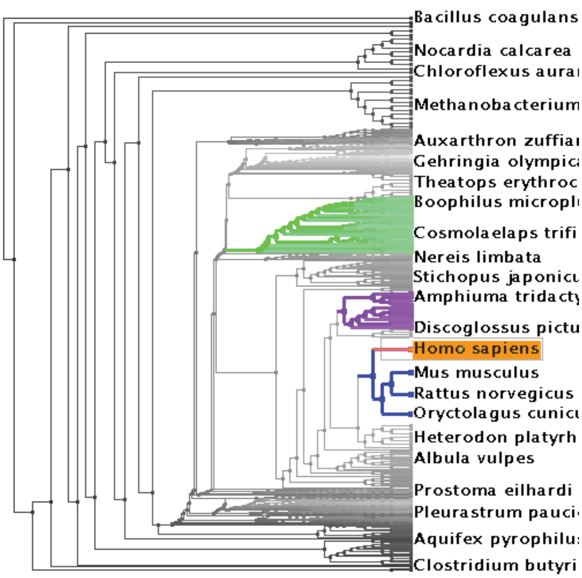
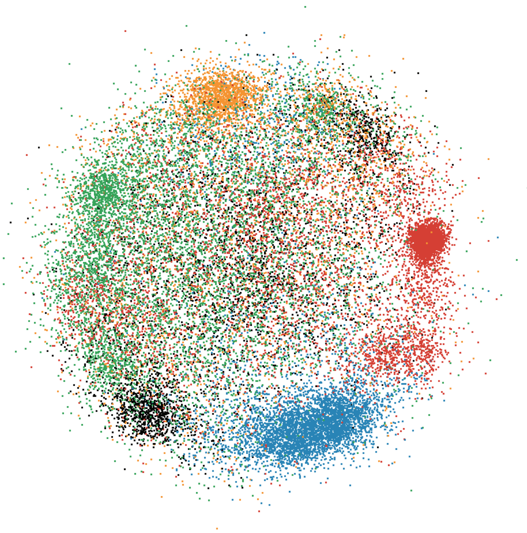


Plate XLIX. The TreeJuxtaposer system features stretch and squish navigation and guaranteed visibility of regions marked with colors (Munzner et al., 2003). (See also Figure 27.17).

Plate L. Dimensionality reduction with the Glimmer multidimensional scaling approach shows clusters in a document dataset (Ingram et al., 2009), © 2009 IEEE. (See also Figure 27.19.)



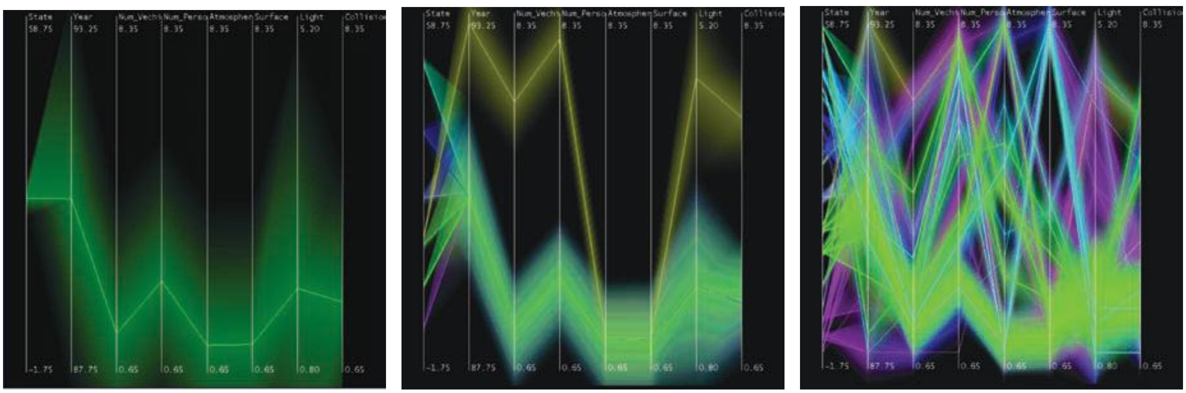


Plate LI. Hierarchical parallel coordinates show high-dimensional data at multiple levels of detail. *Image courtesy Matt Ward* (Fua et al., 1999), © 1999 IEEE. (See also Figure 27.21).

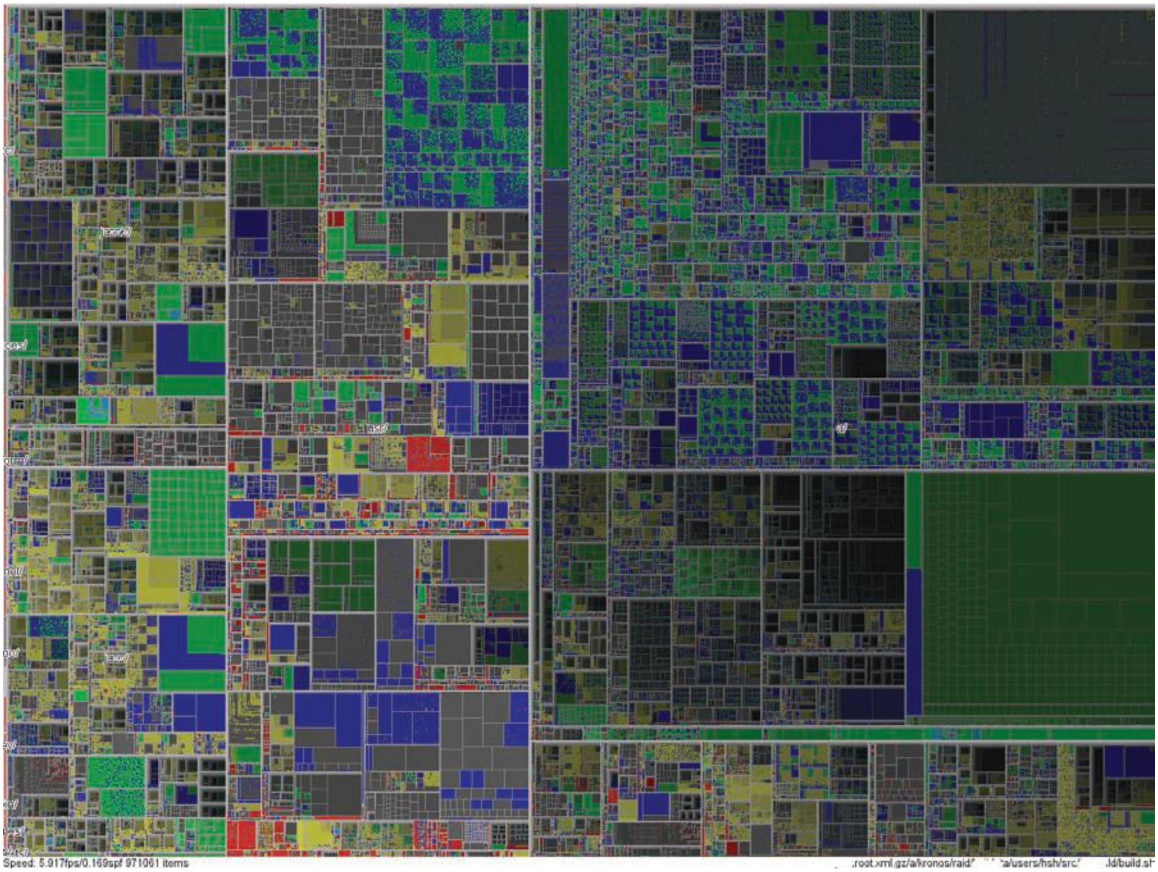


Plate LII. Treemap showing a filesystem of nearly one million files. *Image courtesy Jean-Daniel Fekete* (Fekete & Plaisant, 2002), © 2002 IEEE. (See also Figure 27.25.)

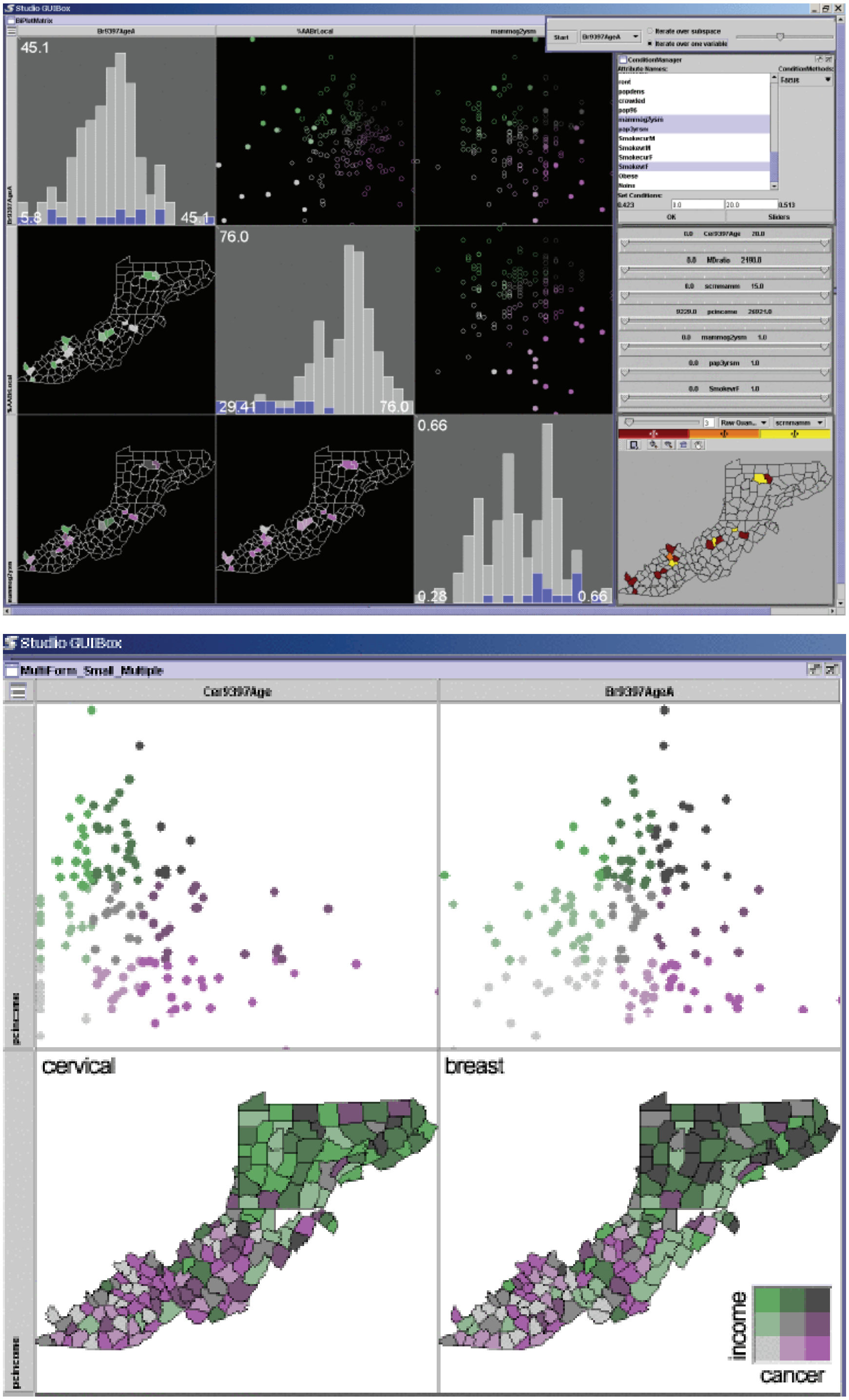


Plate LIII. Two matrices of linked small multiples showing cancer demographic data (MacEachren et al., 2003), © 2003 IEEE. (See also Figure 27.26).