The Render Graph: A Data Structure to Aid in the Interactive Display of Scene Graph Data

by
Jonathan Scott Hofmann

B.S. Computer Science, University of Illinois at Urbana-Champaign

A Thesis submitted to

The Faculty of

The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Master of Science

January 31, 2003

Thesis directed by

James Kwangjune Hahn
Professor of Engineering and Applied Science
Department of Computer Science
The George Washington University

Abstract

Interactive display of three-dimensional geometry has become increasingly popular over the past decade. This geometry is usually organized into a directed graph structure called a scene graph, with display of that geometry implemented by walking that graph. This is insufficient to properly display a wide variety of scenes, especially if there are transparent elements within the scene that require display in a specific order. In addition, to provide for the fastest display of scene-graph geometry, render state changes should be minimized, and reconciling the requirements for display of transparent geometry with the minimization of state changes can be difficult.

This thesis presents a data structure built from the scene graph which is structured to make the rearrangement of the scene-graph data for correct and fast display simple. The render graph is designed to reorganize the scene-graph data in order to minimize the number of render state changes, take advantage of spatial and temporal coherency within the stream of graphics instructions, and also to provide for the correct display of elements with camera-space order dependencies (such as transparent elements). As a side effect it also provides an abstraction layer that simplifies porting the display system to new graphics hardware interfaces (e.g. OpenGL, Direct3D).

Table of Contents

Table of Figures

1. Introduction

Efficient, correct polygon display has been the focus of almost every graphics library since Ivan Sutherland's seminal Sketchpad system [Suth63]. Polygon display is necessary in order to present graphical information in a wide variety of applications ranging from entertainment and the visual arts to scientific visualization to advertisement and electronic commerce. These polygons are defined by an artist, or generated by a computer program, as an ordered set of points (each point is defined as an ordered tuple of floating-point numbers). Each polygon also contains a number of attributes including (but not limited to) color, material properties like specular, diffuse, and emissive color, an image called a *texture* (multiple overlapping textures are not uncommon), and/or a short program (called a *shader*) that determines the visual properties of each polygon. In conjunction with the polygon's intrinsic properties, its visual properties may be affected by external factors such as lights, diffuse or specular reflection, or fog and other atmospheric effects.

Each polygon is originally defined in a coordinate system called *local space*. This is an arbitrarily defined, usually three-dimensional Euclidean space defined by an orthonormal set of axes. Each local space is defined relative to a parent coordinate system, which may be itself a subspace, until a root coordinate system called *world space* is defined. If a local space has no parent coordinate system, then it is equivalent to the world space (i.e. there is no local space, just world space).

1.1 Polygon Rasterization

These polygons are commonly displayed onto a raster-graphics monitor, which is a device capable of changing the image displayed on its surface rapidly (on the order of milliseconds – the update rate of such a device is usually at least 30Hz, and can top 100Hz).

A raster-graphics monitor displays a rectangular grid of pixels, where each pixel can be displayed in a different color selected from a palette ranging from just two colors (black and white) to well over sixty-five thousand different colors, depending on the features of the graphics hardware being used. When a polygon is displayed, it is first projected onto an idealized mathematical model of that rectangular grid through an algorithm called *rasterization* (see Figure 1). Where the polygon intersects a grid line (the mathematical representation of a pixel on the monitor), a *fragment* of that polygon is created and assigned a color.

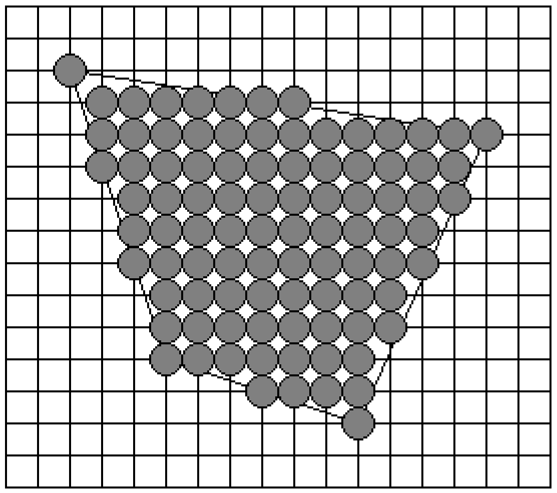

Figure 1. A rasterized polygon. The circles represent each polygon fragment. Whether the circles are located at the intersections of grid lines, or at the center of each square, is a matter of convention that varies from device to device.

When each fragment is generated, it is assigned a color derived from the polygon's attributes. This color is usually specified as red, green, and blue values, from which many colors can be derived [Fole90]. This may not be the color that was originally assigned to the polygon! It is derived from all polygon attributes that contribute to the visual appearance of the generated fragment. One such component that may be associated with the color's red/green/blue values is an *alpha* value, which is a measure of the fragment's transparency, and therefore the color of the fragment will be derived from both the polygon's color (if any) and the color of any other polygon behind the fragment being processed. The fragment's color may also be affected by external factors such as lights, diffuse or specular reflection, or fog and other atmospheric effects. Where multiple polygons all intersect a single pixel, their respective fragments are combined together (usually through an antialiasing filter; see [Fole90] or [Watt92] for more details) to generate that pixel's color. If one or more of the fragments include an alpha value, then an additional blending calculation is performed [Port84]. Note that when fragments are transparent, the order in which they are drawn is important – some blending mode equations are not commutative, so if the polygons that each fragment is generated from change order (e.g. the polygon in front is moved behind the polygon in back) then the output color will (erroneously) change. This polygon-order dependency is known as a *camera-space dependency*, since the most common way that the depth order is changed is to move the camera around a static scene.

Given that each polygon fragment has a (potentially) unique combination of attributes, the rendering algorithm can be viewed as the execution of a state machine, where the attributes that define each state include the current color, current texture, current material, current local-to-world-space transformation, and so on. In fact, the OpenGL graphics library is built around the execution of such a state machine [Sega01]. When a scene is displayed, the graphics library must spend some time managing the state machine, particularly when the state is changed due to the current fragment having a different state (i.e. set of attributes) from the previous fragment. Time spent managing the state machine is time not spent processing fragments, so minimizing the number of state changes is necessary to draw a set of polygons in the minimum time possible.

This rasterization process has to be executed for every polygon in the display. Given that the average display has hundreds of thousands to millions of pixels (An NTSC television set has 640 x 480 = 307,200 pixels; a 1280 x 1024 computer display has 1,310,720 pixels) this algorithm is usually implemented in hardware. Modern libraries such as OpenGL [Sega01] or Microsoft's Direct3D [Micr00] provide low-level interfaces to the graphics hardware, or implement the polygon display process in software if hardware is not available.

1.2 Scene Graphs

Managing and organizing the thousands of polygons displayed in each frame is difficult, and much research has gone into the data structure known as a *scene graph* (see Figure 2), which manages the polygons and other items necessary to generate the display. A scene graph is a hierarchical set of nodes connected by edges usually in a directed fashion. Interior nodes are used to specify child-to-parent-space transformations, define polygon and scene properties, or group related nodes together. Leaf nodes specify geometric primitives (see Figure 3).

The scene graph is used by many real-time graphics applications, including most scientific visualization programs, visual simulation programs, and computer games. Traversing this data structure and drawing each of these polygons in the correct order while making use of the graphics library and/or hardware in the most efficient fashion is a difficult task, however.

In Figure 2, the wireframe cube moves in a circle around the torus. The cube represents a point light source that illuminates the torus, so that the cube's motion illustrates the motion of the light source. This is why the light's scene graph node is attached to the cube's scene graph node, so that when the cube is moved the light will also be moved as a side effect. Since the torus remains stationary, its node cannot be attached to the cube's node, or it would also move along with the cube as well. Also note that the torus is actually implemented as a number of triangle strips, with each individual strip wrapped around the torus' axis.

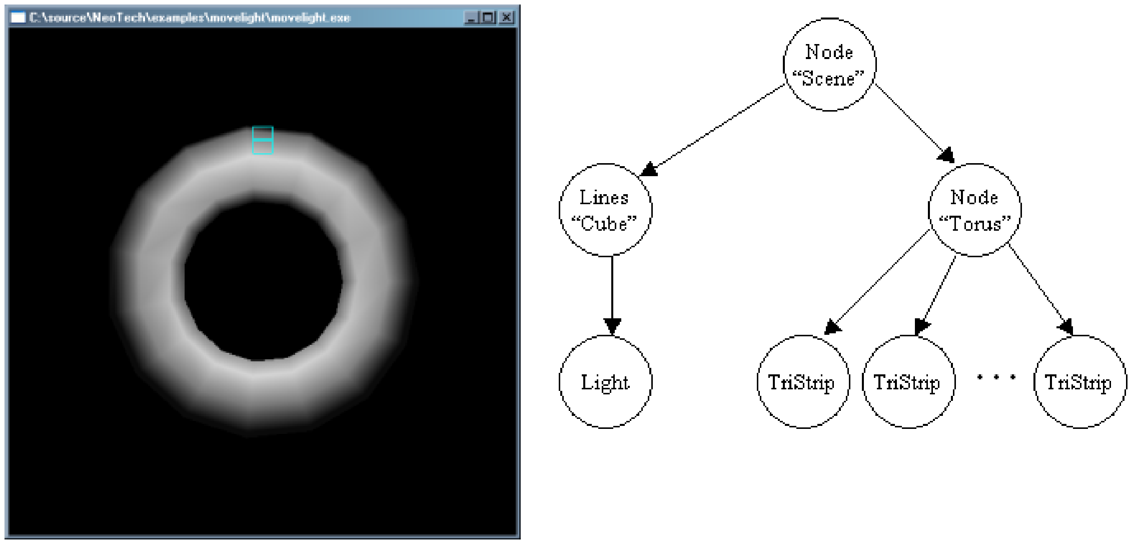

Figure 2. A sample scene and its scene graph.

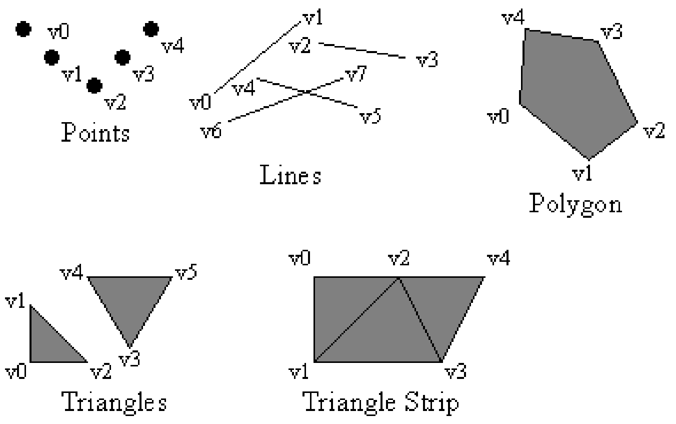

Figure 3. Geometric Primitives.

The display is then generated through a traversal of this data structure. Given that the scene graph has a hierarchical structure, its traversal can be implemented through a stack-based algorithm such as an inorder traversal [Knut97]. The stack then encodes the current state of the graph traversal, including several attributes of the graphics state such as current local-to-world coordinate transform, current color, material, or current texture. This stack is also how child nodes inherit state from their parent nodes.

Some graphics state cannot be expressed via the hierarchical nature of the scene graph, however, especially when polygons are moving relative to each other. It may be infeasible to locate every polygon illuminated by a light under a scene graph node representing that light as illustrated by Figure 2 above. The scene graph also does not support operations that have camera-space dependencies, such as the sorting of transparent polygons in back-to-front order, since that order changes as the camera moves around the polygons in question (or the polygons move around each other).

Scene-graph display libraries take one of two different approaches to solve these problems. They either attempt to solve everything within the scene graph, or they implement one or more auxiliary data structures to aid in the display process.

If an auxiliary data structure is not used, complex traversal strategies are required to correctly display a scene graph's contents. These libraries do not always visit the scene-graph nodes in a parent-to-child, child-to-parent, or sibling-to-sibling order, but they are visited (and frequently re-visited) in the order required to generate a correct display. These traversal strategies require large amounts of code to implement and are difficult to debug and maintain.

The other approach is to develop auxiliary data structure(s) to manage the bookkeeping necessary to generate a correct display. Of the libraries implementing this approach, they can be sorted into libraries that use just one auxiliary data structure to aid in the scene-graph traversal, and libraries that use multiple auxiliary data structures.

Several libraries introduce one data structure per problem (camera-space dependencies, effects that affect some arbitrary subset of the scene-graph nodes, and the use of the graphics hardware in the most efficient fashion). Coordinating the use of each data structure can be tricky, and the overlapping nature of some of these problems (e.g. transparent geometry illuminated by several lights) can make it difficult to use the graphics hardware efficiently.

The render graph is in the category of libraries that implement one unified auxiliary data structure that solves all of the above problems.

1.3 Render Graphs

This thesis introduces a data structure called a *render graph*. The render graph is a data structure designed to aid in the display of scene graph data at interactive frame rate. It address the problems of render state management, polygon sorting for blending operations, and configuration of per-vertex or per-fragment shader programs, and it decouples the order in which the polygons are rendered from the order in which the scene graph nodes are traversed, which eliminates the need for complex scene-graph traversal algorithms to accommodate the camera-space dependencies of effects like alpha-channel transparency. Its nature provides a mechanism by which the number of render state changes will be minimized, providing for efficient use of the graphics hardware.

The render graph also provides an abstraction layer that implements a measure of platform independence, simplifying the support of a wide variety of graphics-acceleration hardware.

2. Polygon Display and the Graphics Pipeline

There have been many libraries, academic, personal, and commercial, developed to address the problem of displaying as many polygons as possible in as little time as possible given the limits of the hardware used. The first widely known library developed for interactive computer graphics was Ivan Sutherland's Sketchpad system developed in 1963 [Suth63]. This library introduced the use of data structures to store graphical primitives (i.e. polygons with specific shapes like triangles, squares, and so on), and several other features which appear in modern graphics libraries [vanD88, ISO88, SGI91, HP91, ARB93, Micr00].

These libraries can be sorted into two categories: low-level immediate-mode libraries which retain only minimal state between polygon specifications, and high-level retained-mode libraries built around the scene graph and the encapsulation of polygon meshes into higher-level objects (e.g. the definition of a car-shaped collection of polygons as a car). Most retained-mode libraries use an immediate-mode library to interface with the graphics hardware; they treat the immediate-mode library as a sophisticated device driver controlling the graphics pipeline.

Every graphics library is concerned with the query and display of the *scene database*, which is the collection of polygons, their attributes (e.g. color, texture, and other per-vertex or per-polygon features), and anything else required to implement the desired display (lights, cameras, fog, and other items). Data structures such as the scene graph is then derived from the contents of the scene database.

2.1 Polygon Display

Every graphics library implements the display (called the *rendering*) of polygons into a framebuffer, that is, a region of memory which contains the pixel array displayed on a computer monitor (in certain rare cases, there is no frame buffer and the pixels are written directly to the display device). The process of rendering a polygon to the framebuffer consists of passing the vertices of the polygon through a pipeline which consists of transforming the vertex coordinates of each polygon to camera space, projecting those transformed vertices to the screen, clipping the polygons to the view frustum, and then performing hidden-surface elimination, rasterization, and shading (see Figure 4).

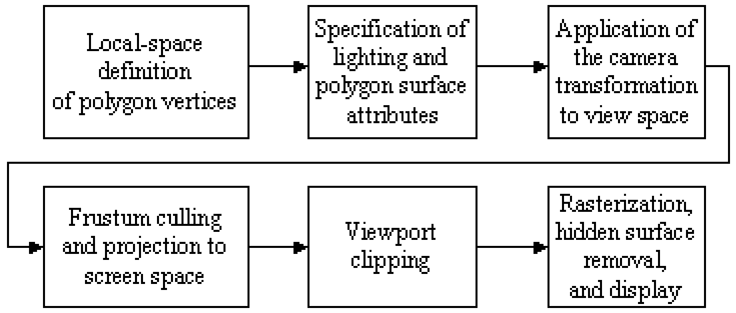
Local-space definition of polygon vertices
Specification of lighting and polygon surface attributes
Application of the camera transformation to view space
Frustum culling and projection to screen space
Viewport clipping
Rasterization, hidden surface removal, and display

Figure 4. The transformation and display pipeline.

Much research has been done into the process of efficiently processing polygons for display, and this pipeline is frequently implemented in hardware. The graphics library (e.g. OpenGL [Sega01] or Direct3D [Micr00]) then acts as a device driver mediating the application's use of the graphics hardware.

2.2 Immediate-mode Interfaces

The simplest form of graphics library renders each polygon as the application completes its definition. Only a minimal amount of state is retained between library function calls; this state is typically limited to current color, current transformation matrix, and similar attributes. Because each polygon is drawn as soon as it has been defined, these libraries are called “immediate mode”. Due to their relative simplicity, most of the earliest graphics libraries were immediate-mode systems, and a lot of modern systems include an immediate-mode interface to better support applications which don’t need scene-database management or other sophisticated support in the graphics library. Some of the more advanced immediate-mode libraries are Starbase [HP91], Iris GL [SGI91] and its derivative OpenGL [Sega01], and Direct3D’s immediate mode interface [Micr00].

The function calls implemented by these libraries execute quickly because they are small, simple, and rarely need to allocate memory or other permanent or semi-permanent resources. Due to this lack of resource management, they necessarily foist off scene-database management and render-state management responsibility onto the user, so system performance is almost entirely based upon the quality of the application’s design and data structures.

Immediate-mode libraries include no to minimal support for interaction, as the library contains no information about the scene being rendered. Another weakness of the immediate-mode library is that it typically is coupled very tightly to a specific piece of hardware- IRIS GL does not exist outside of the Silicon Graphics hardware, Starbase does not exist off of the HP hardware, and Direct3D is limited to the Intel hardware supported by Microsoft. This weakness is not an attribute of all immediate-mode libraries, as OpenGL has been ported to more platforms than any other graphics library in existence.

2.2.1 Display Lists

The next generation of graphics libraries included rudimentary support for retaining geometry through a data structure called a "display list". A display list is a static list of graphics commands that have been organized in the most efficient fashion possible. This is why the list is static – any changes may result in the list being sub-optimal, so the library does not allow the user to change the list *in situ*, if a change is desired the old list must be destroyed and a new list created.

Two of the earliest toolkits that included display list support were GKS [ISO88] and PHIGS+ [vanD88]. Many modern libraries include support for display lists, most notably OpenGL.

Use of display lists can improve frame rates, as the graphics library is able to bundle rendering commands together and transmit them to the rendering pipeline in bursts, therefore getting better use of the system bus and other graphics hardware. For example, on PC hardware supporting the Accelerated Graphics Bus (AGP) feature [Inte98], the NVIDIA OpenGL driver allocates AGP memory for display list storage. AGP memory is a part of the machine's main memory which has a separate high-bandwidth connection to the video card.

Display lists are still merely an optimization of an immediate-mode library, and contain no support for scene database management, or interaction with the rendered scene.

2.3 Retained-Mode Interfaces

The most sophisticated libraries that have been created to date are those libraries that include scene database management facilities, typically through a data structure known as the *scene graph*. A scene graph is a graph structure (usually a directed acyclic graph) of objects called *nodes* that are connected by *edges*. The node objects contain the scene-database information, including geometry, materials, textures, lights, cameras, and anything else necessary to implement the display.

The scene graph was first popularized in Silicon Graphics' IRIS Inventor system [Stra92] (later known as OpenInventor [Wern94]), and then refined in their IRIS Performer product [Rohl94]. Many other systems implement the scene graph data structure in some form, including VRML97 by the VRML Consortium [ISO97], NetImmerse from Numerical Design Labs [Bish98], and WildMagic from Magic Software [Eber00].

Scene-graph libraries provide detailed support for scene-database management and operations upon that database, such as interaction, culling, and rendering. The performance of the graph traversal algorithm becomes paramount in producing a system that can provide a real-time frame rate, and traversal strategies are many and varied, from a simple tree walk (VRML, NetImmerse) to an "action" framework (OpenInventor, IRIS Performer).

2.4 Procedural Graphics

Not all polygons are artist-defined. In many cases, the polygons are algorithmically produced. For example, the water effect employed by Bethesda Softworks' recent title The Elder Scrolls 3: Morrowind [Beth02] uses a cellular automaton to generate a normal map (i.e. a texture whose texels are normal vectors, not colors, that are then used in the lighting equations) to produce ripples in a simple polygon simulating the water's surface.

This allows a minimum number of polygons to be used, with the fine details produced by a bumpmapping effect, rather than the use of large numbers of polygons (Figure 5).

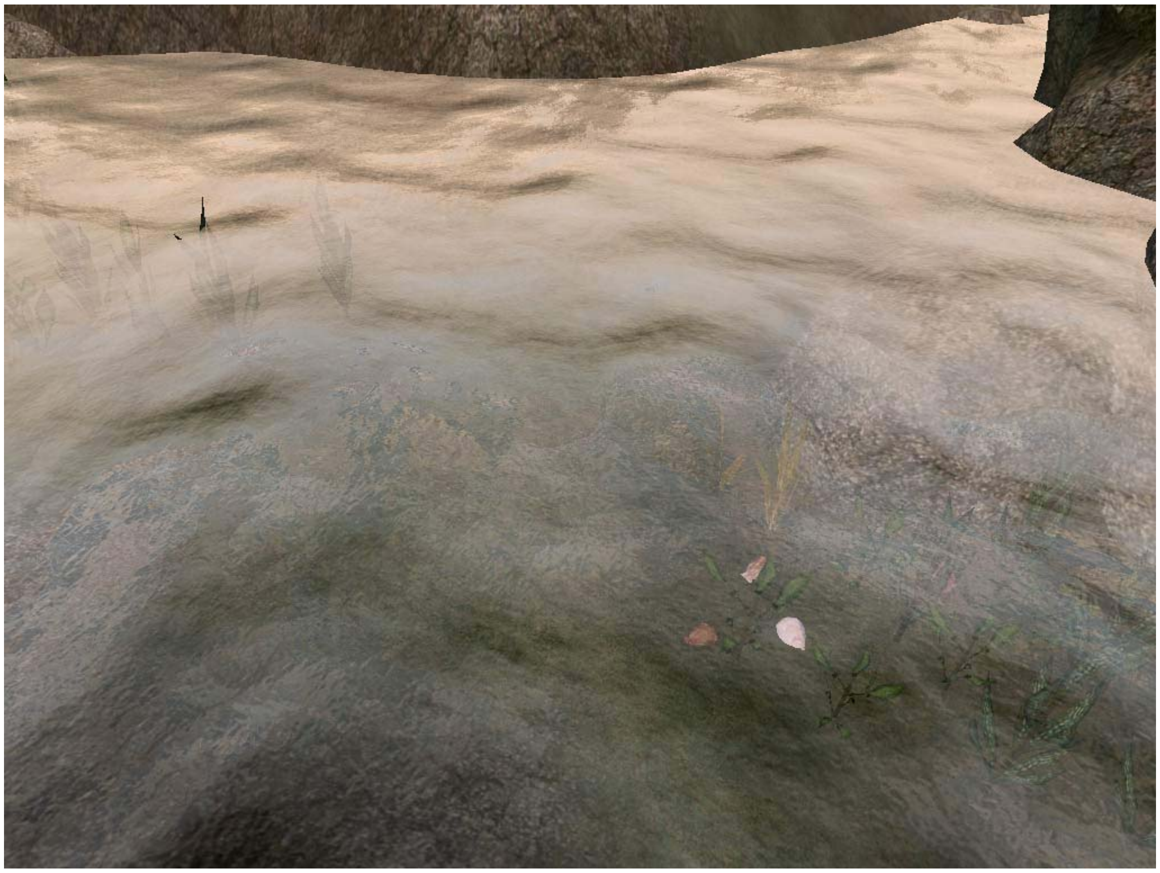

Figure 5. A procedurally generated water effect, from The Elder Scrolls 3: Morrowind. Image copyright 2002 Bethesda Softworks, Inc., a ZeniMax Media company.

In this case, both the water's polygons (a flat plane consisting of tiled triangles) and the texture applied to the water's surface are algorithmically generated. If the ripples were implemented using large numbers of polygons instead of the normalmap, the performance of the game would be unacceptable, as the frame rate would drop below 30Hz as the computer spent its time processing the extra polygons. One thing to note is that the above water mesh is only a few polygons (on the order of a couple of hundred polygons), with the polygons being far larger than the ripples displayed.

Another important use of procedural graphics is the addition of shadows to a scene. Shadows are an important cue to emphasize that meshes are located on the surface they're resting on, and not hovering slightly above that surface. They also provide more information on the shape and location of the shadow-casting object relative to the other objects in the scene (see Figure 6).

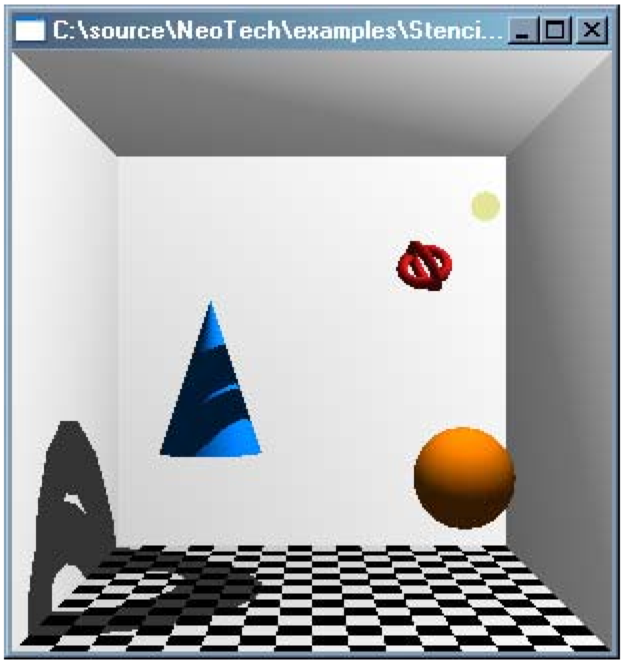

Figure 6. Real-time shadow casting.

3. Graphics Instruction Set Design

Graphics hardware was originally merely an application-specific integrated circuit (ASIC) that aided the CPU in the display of graphical information. Over the years, and especially with the growth in proceduralism in computer graphics, the graphics hardware has become a general-purpose computer in its own right. NVIDIA has even begun to call its GeForce line of graphics hardware a "GPU", short for Graphics Processing Unit, to emphasize its similarity with the computer's CPU chip.

The computer graphics pipeline is essentially a single-instruction multiple-data (SIMD) process, where the frame buffer is the data path and the host application is the control path. Since every pixel in the frame buffer is conceptually written at the same time, it is effectively a parallel process. Modern chips (e.g. NVIDIA's GeForce3 [Lind01], ATI's Radeon 8500 [ATI02]) have many attributes of vector processors with a vector size of 4, since the geometry transformation pipeline works on a stream of homogenous points or RGBA color values.

3.1 Issues in Graphics Instruction Set Design

The design of the instruction set supported by the graphics hardware reflects the challenges of polygon display. Those challenges include, but are not limited to: coordinate transformation, polygon rasterization, interpolation of per-vertex attributes across a polygon face, evaluation of the Phong lighting equation, texture sampling, antialiasing, and the execution of application-defined shader programs.

The graphics instruction set is not an instruction set in the traditional sense, that is, it is usually not a traditional assembly language or microcode. The graphics instruction set is usually a library of functions (also known as an application-programmer interface, or API) expressed in a medium- or high-level language such as C or C++. Some modern examples of these APIs are OpenGL [Sega01], which is a C language API, and Microsoft's Direct3D [Micr00], which is a C++ language API. Other groups have produced wrappers for the OpenGL API in other languages such as Java, Ada, Python, and FORTRAN.

These libraries then interface with a system-level device driver to control the hardware. This interface is usually an implementation detail of the graphics library, and invisible to the application using the library.

This thesis will focus on the use of OpenGL as an ISA, even though almost everything OpenGL does has been duplicated in whole or in part by Microsoft's Direct3D and other more obscure libraries. Researchers have focused on OpenGL due to its open and well-documented nature – Microsoft considers much of Direct3D's internal structure and implementation details proprietary trade secrets, and does not disclose them to researchers. Some other research projects have proposed APIs that attempt to address OpenGL 1.3's deficiencies [McCo00, Rost02] but none of these projects have published any results yet.

3.1.1 How is OpenGL an ISA?

The key observation to why OpenGL (and related or similar APIs such as Direct3D) is an ISA is that same operation(s) are performed for every pixel in the frame buffer. One control path (the execution of the program using the OpenGL API) acts on the elements (i.e. pixels) of the frame buffer in a conceptually parallel fashion, much as a SIMD computer operates. The frame buffer acts as an accumulator, texture memory acts as per-texel memory, the blending functions implement basic arithmetic operations, lookup tables support function evaluation, and the conjunction of alpha testing and the stencil buffer provide conditional execution.

An alternative approach to programmable hardware places the programmability within each pass. Rather than executing a large number of passes each performing one complex operation, each pass executes a number of simple instructions. This is the approach taken by Proudfoot's Real-Time Shading Language [Prou01].

Over the past couple of years, a couple of shading-language compilers have been written that target the OpenGL API [Peer00, Prou01]. All of these compilers support languages similar to the RenderMan shading language [Upst90] – one [Peer00] makes RenderMan support explicit. Each language differs in how it abstracts the graphics hardware.

More recent work adds high-level shading language support to the OpenGL specification itself. NVIDIA's Cg language compiler [NVID02] is a conventional compiler that emits vertex and fragment program code, while 3DLabs' OpenGL 2.0 Shading Language [Bald02] works directly with the graphics hardware device driver, compiling straight to the hardware with no intermediate instruction set.

3.1.2 OpenGL as SIMD Machine

Peercy et al's OpenGL compiler [Peer00] makes the SIMD model of computation explicit. Each element of the OpenGL framebuffer is a 4-byte quantity usually organized into four numbers, one each for the red, green, blue, and alpha (i.e. opacity) values. The interpretation of those numbers as color and opacity values is merely convention, however, so they can be redefined to correspond to whatever the application requires – a four-element homogenous point, a triple such as a point or normal and a single number, two two-element tuples, or four independent values. Values stored in the framebuffer can vary on a per-pixel basis, since each element of the framebuffer is independent of the other framebuffer values. In addition, the program can set a write mask controlling which framebuffer element(s) are written to, through the glColorMask() function.

Variable storage and manipulation is managed by copying portions of the framebuffer to a texture image, via the glCopyTexSubImage2D() function. These images need not contain all four floating-point values per texel; texture images may contain just one channel of information (i.e. a heightmap or greyscale image), three channels (an RGB image), or all four channels (an RGBA image). To use a variable a textured polygon is drawn, using a projective texture so that texel (i.e. variable) interpolation can be strictly controlled.

Arithmetic operations are performed using framebuffer blending operations. All blending operations have two arguments: a source (i.e. a fragment) and a destination (i.e. the framebuffer). The fragment can either be produced by drawing some geometry, or by copying from a texture or framebuffer with glCopyPixels(). The functions glBlendEquation(), glBlendFunc(), and glLogicOp() support copying, addition, multiplication, alpha blending, and logical operations. Subtraction is done by addition of negative values, and division is done through multiplication by the reciprocal.

To implement monadic mathematical functions (such as many trigonometric functions), a texture is created whose texels (i.e. texture elements) contain a lookup table. A pixel copy from one texture to another, or from the framebuffer to a texture, then evaluates the function.

Flow control is done through the stencil buffer and the glStencilFunc() and glStencilOp() functions. Various bits of the stencil buffer are then used to identify the object being operated upon, and the conditions that are satisfied. Only one bit is required per condition; adjacent bits represent nested conditions. If the condition varies across the frame buffer, the alpha test function is used to select the frame buffer pixels that satisfy the condition. Because a SIMD computation is being implemented, both branches of the if-then-else conditional are being evaluated; complementing the stencil operation provides the other execution path.

3.1.3 Abstract Programming Pipelines

The SIMD model is not the only theoretical model that a shading language compiler can support. Proudfoot et al's real-time shading language compiler [Prou01] separates shader execution into several categories of pass: constant, per-primitive, per-vertex, and per-fragment (see Figure 7).

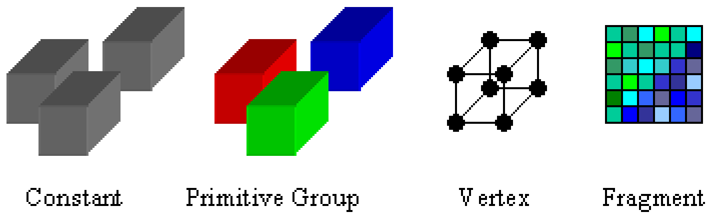

Figure 7. Computation Frequencies. Different colors denote independent elements within each frequency.

The fundamental difference between Peercy's approach and Proudfoot's system is that while Peercy treats each pass as a single complex SIMD instruction, Proudfoot treats each pass as a sequence of many simple instructions. Proudfoot then points out several fundamental differences between the two theoretical models:

- Vertex programmability (as opposed to fragment programmability) provides a more natural interface to properties that vary slowly across a polygon (such as lighting).
- Vertex programmability provides a more natural map to the programmer's conceptual model of the graphics pipeline.
- There are bandwidth bottlenecks between the graphics chip and framebuffer, and between the host computer and graphics chip. Each rendering pass crosses both of these bottlenecks, so the number of passes should be minimized to improve performance. Proudfoot also notes how VLSI technology trends indicate that the ratio of graphics chip performance to external memory bandwidth will continue to increase, favoring an approach that minimizes the number of passes.
- The SIMD model breaks down as each pass grows more complex. Higher-level, more complex instructions could be developed, but such instructions would be so application-specific as to be practically unused by all other programs that require a different combination of operations.

Proudfoot defines an abstract programmable pipeline (Figure 8) that his compiler then targets. This pipeline is designed to be easily mapped to OpenGL 1.3 (using the NVIDIA OpenGL extensions [NVID00]) or DirectX 8.0's Direct3D library. The stages are ordered in increasing frequency of computation, that is, there are more vertices than primitives in the geometry being rendered, and far more fragments than primitives or vertices. Between each stage of the programmable pipeline are fixed-function pipeline stages (i.e.

the conventional OpenGL pipeline) that implement the non-programmable parts of the graphics display pipeline. In particular, the polygon rasterization step that produces the fragment set and the interpolation of color and coordinate information across a polygon occurs between programmable pipeline stages.

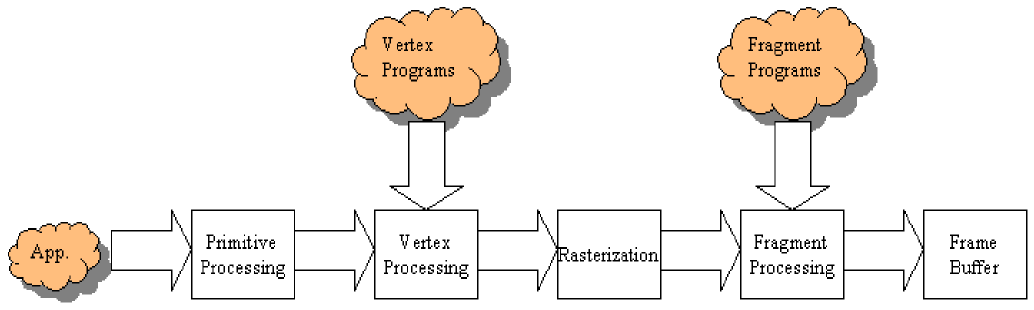

Figure 8. The abstract programmable pipeline.

Conceptually, the abstract programmable pipeline executes all shaders in a single pass. In practice, many large shaders are executed using more than one pass, although the multipass nature of the shaders is considered an implementation detail.

3.1.4 Other Graphics Instruction Sets

OpenGL is not the only graphics API that has been developed to support programmable hardware. Michael McCool at the University of Waterloo has developed the Simple Modeling and Shading API, also known as SMASH [McCo00]. SMASH makes the observation that there is a continuum of ways to implement programmable shading:

- Pure multipass
- Multipass with multitexturing
- Multipass with per-fragment shaders
- Multipass with per-vertex and per-fragment shaders

- Single pass with fallback to multipass

For example, Peercy's OpenGL Shading Language uses a multipass with multitexturing model, while Proudfoot's Real-Time Shading Language uses a single-pass model. McCool notes that the single-pass approach offers several advantages over the multipass model: it keeps all relevant information in one place, the shaders can be unrolled to a multipass implementation if performance is an issue, and trying to combine an already multipass shader into a single-pass shader can be very difficult. More importantly, single-pass shaders cross the bottlenecks of host computer – graphics chip and graphics chip – frame buffer only once, while a multipass shader crosses these bottlenecks once per pass.

3.2 Per-Vertex and Per-Fragment Instruction Sets

A recent development in graphics hardware architecture has been the introduction of per-vertex and per-fragment programmability [Micr00, Lind01]. These capabilities have been exploited to support real-time shading languages [Peer00, Prou01, NVID02, Micr02, Bald02, McCo00].

Hardware support for per-vertex and per-fragment programmability has been introduced into the market by NVIDIA with their GeForce3 series of graphics acceleration chips [Lind01] and ATI with their Radeon 8500 series of graphics accelerators [ATI02]. The vertex program model is displayed in Figure9. Vertex attributes are stored in the input registers, and the processed vertex is stored in the result registers. The parameter banks contains lighting and shading parameters, and the variable banks provides temporary storage. A function unit implements the 17 instructions in the vertex program instruction set. Each instruction's parameters can either be negated, or *swizzled* (i.e. permuted), to simplify the implementation of common operations such as the cross product R1 = R0 x R2, as displayed in:

```
MUL R1, R0.zxyw, R2.yzxw;
MAD R1, R0.yzxw, R2.zxyw, -R1;
```

Algorithm 1. Computing a cross product using the vertex program instruction set.

`MUL` performs a component-wise multiplication of the second and third arguments and stores the result in the first argument. `MAD` implements a component-wise multiplication of the second and third arguments, adds the fourth argument to the product, and stores the sum in the first argument (hence the name “MAD”, short for multiply and add). Note the use of swizzling and negation to perform the source vector rotations.

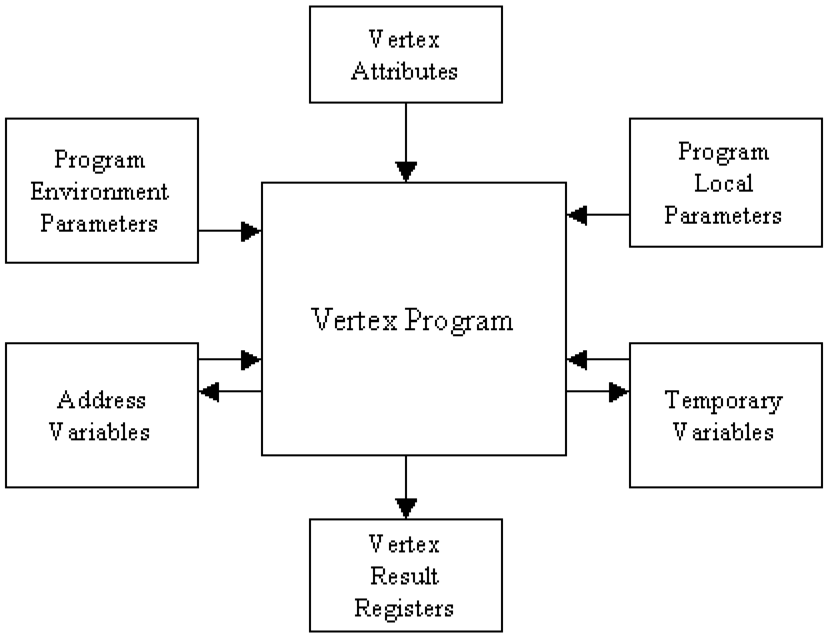

Figure 9. The vertex program model.

Table 1 displays the instruction set. Note the absence of branching instructions. The fixed-function pipeline, that is, the nonprogrammable portions of the graphics library, is controlled by a global state that does not depend on each vertex’s data. Driver-level optimizations of this pipeline preclude branching within vertex programs, as branches may affect the global state and therefore violate assumptions made by the drivers.

Instruction	Description
MOV	Move
MUL	Multiply
ADD	Add
MAD	Multiply and Add
DST	Euclidean Distance
MIN	Minimum
MAX	Maximum
SLT	Set on Less-Than
SGE	Set on Greater-Than or Equal
RCP	Reciprocal
RSQ	Reciprocal Square Root
DP3	3 element dot product
DP4	4 element dot product
LOG	Logarithm (base 2)
EXP	Exponentiation (base 2)
LIT	Phong Lighting
ARL	Address Register Load

Table 1. The vertex program instruction set.

3.3 High-level Shading Languages

The Cg language announced by NVIDIA Corporation [NVID02], and its sister D3DX High-Level Shading Language (HLSL) announced by Microsoft [Micr02], do not target a graphics API like OpenGL (targeted by Cg) or Direct3D (targeted by both Cg and HLSL), but rather they are part of their respective libraries. When a shader program is compiled, it results in a stream of vertex program or texture shader instructions that are exactly analogous to the compilation of ANSI C code to (for example) Intel Pentium 4 assembly-language instructions. These instruction streams are then downloaded into the graphics hardware for execution in the same manner as if the shading language was not employed.

The OpenGL 2.0 Shading Language recently proposed by 3D Labs [Bald02] also does not target OpenGL in the traditional sense, but rather it is part of OpenGL itself. Compilation of shader programs results in platform-specific object code that directly interfaces with the system-level graphics device drivers, not a set of functions in a high-level language such as C that use the OpenGL library to interface with the graphics hardware.

The OpenGL shading language does not abstract the graphics hardware in terms of the OpenGL library; it uses the abstraction already defined as part of the OpenGL 2.0 Proposal [Rost02], which models the graphics hardware as a state machine (Figure 0).

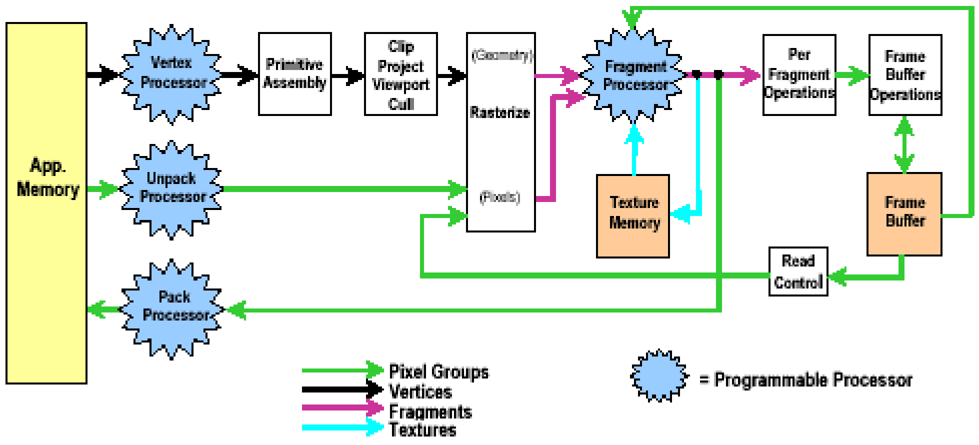

Figure 10. The OpenGL 2.0 Logical Diagram (Figure 2 in [Rost02]). The white boxes are components that are unchanged from OpenGL 1.3.

4. The Render Graph

The *render graph* is a data structure designed to aid in the display of scene-graph data. It is a graph structure whose nodes, called *render actions,* each represent one operation executed by the underlying graphics library. These actions are then reorganized to implement the optimal execution of their effects and executed by traversing the graph, producing the contents of the frame buffer that is then displayed to the user.

There are many similarities between the render graph's stream of render actions and the stream of instructions emitted by a compiler such as a conventional C language compiler or a shading language compiler such as Cg. One important advantage of the render graph is that over the years since the initial implementation of the render graph in 1999 it has been simple to incorporate technologies such as vertex and fragment programs, and shading language support for languages like Cg.

4.1 Related Work

This data structure was developed after making the critical insight that the display of scene-graph data depends on the execution of a stream of graphics-library function calls in a specific order, and the order of these function calls is critical both for the correctness of the frame-buffer contents at display time, and to make the most efficient use of the graphics hardware. This graphics instruction stream is generated through traversal(s) of a scene graph. Simple traversal of the scene graph, where each node is visited exactly once, is insufficient to correctly display the scene graph contents if the graph contains sophisticated information such as transparency. Existing scene-graph libraries, such as NetImmerse [NDL99], OpenInventor [Wern97], or Wild Magic [Eber00], address this problem through complex traversals of the graph nodes, where nodes are usually visited multiple times and the traversal order is not linear (i.e. traversal may not proceed from the parent node to its child nodes or vice versa, but rather may jump to non-adjacent nodes). Recent work at SGI has produced a sequence of libraries that further develop and expand the idea of scene-graph transformation as a fundamental operation to provide support for advanced features [Birc97, SGI98].

This stream of instructions is similar to the stream of assembly language instructions generated by a compiler parsing a program written in a high-level language. A similar observation was made by [Peer00], where they develop the insight that the functions of the OpenGL library [ARB93] can be considered as an "assembly language" for the SIMD computer the OpenGL state machine implements. We can use this similarity to apply algorithms developed in the compiler and interpreter literature to optimize this stream of graphics-library instructions for targets like fastest execution, minimal use of resources such as texture or frame buffer memory, or elimination of repeated operations through common subexpression evaluation passes.

The RenderWare library produced by Criterion Software [Crit00] also disassociates the traversal of the scene graph from the execution of graphics library instructions, through their PowerPipe data structure. Criterion's PowerPipe is similar to the render graph in that iterating over the elements of the PowerPipe generates the display, but the PowerPipe exists at a higher level than the render graph, where each element of the PowerPipe represents more than one graphics-library call. Because of this, it is possible to perform optimizations upon the render graph that aren't possible with PowerPipe.

The real-time procedural shading system developed by Proudfoot et al [Prou01] decomposes the display commands into a directed acyclic graph (DAG) that is then transformed into graphics library function calls in a manner similar to [Peer00]. The DAG nodes are operators that are interpreted by an abstract programmable pipeline representing the shading language's computational model. While Proudfoot's system is concerned specifically with procedural shading of geometry, the render graph abstracts the entire display process, including any procedural shaders that may be necessary. Many optimization techniques employed by both [Peer00] and [Prou01] can be implemented within the render graph structure.

4.2 Render Actions

The fundamental component of the render graph is the render action structure (see Figure 71). The render action is an atomic (i.e. indivisible) operation which implements one operation of the display process. This operation may be the execution of one function, a sequence of functions, or the setup of an argument to a function to be executed later. The implementation of each action is dependent upon both the graphics library the render graph is built upon and the hardware it's executing on; for example, a render action that draws a polygon would have different implementations if OpenGL were used versus Direct3D, or if it were executing on Nintendo's GameCube versus a Sony PlayStation 2.

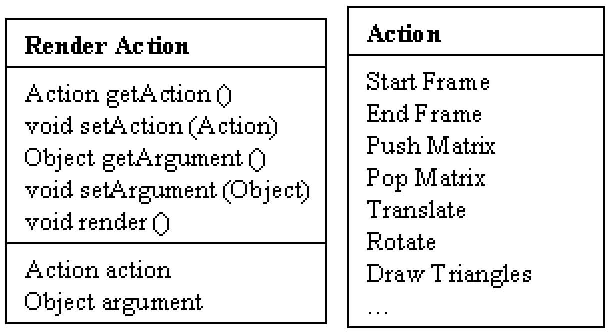

Figure 71. Render Action structure and some sample actions.

A stream of render actions is emitted during the traversal(s) of a scene graph structure. This stream is then parsed to implement the display. The stream is implemented as a directed graph that is the render graph itself, which is frequently acyclic although the compilation of procedural shaders may introduce cycles into the graph structure. It is at this point that several optimizations may be performed, and these optimizations will be discussed below.

4.2.1 Scene Graph Traversal

Render actions are created and attached to the render graph during the traversal of the scene graph. This traversal need only be a simple traversal of the scene graph, such as an infix traversal [Knut97], where each scene-graph node is visited exactly once. This is an important practical advantage of the render graph, in that the scene graph structure is traversed exactly once (per rendering pass) without the need for complex and difficult-to-maintain traversal mechanisms.

```
function CreateRenderGraph (SceneGraph scene_graph)
{
    RenderGraph render_graph;

    BuildRenderGraph (GetRootNode (scene_graph), render_graph);

    return render_graph;
}
```

```
function BuildRenderGraph (
    SceneGraphNode node, RenderGraph render_graph)
{
    Previsit (node, render_graph);
    Visit (node, render_graph);
    VisitChildren (node, render_graph);
    Postvisit (node, render_graph);
}
```

Algorithm 2. Render Graph Creation. Render Actions can be created in each visitation function.

Algorithm 2 displays the algorithm used to create the render graph. When a scene graph node is visited, one or more render actions may be instantiated and attached to the render graph. For example, Algorithm 3 displays the function used to draw a textured triangle. In that function, the following actions would be generated: translate, rotate, scale, enable texturing, set texture modulation function (these five actions are only emitted if necessary), bind texture (if it was not already bound during the traversal of a previously visited scene-graph node), draw triangles (see Figure 2). The reason the first five actions may not be emitted is to optimize performance- if an action is not necessary it is not emitted, so that no time is spent executing an action that would result in an identity transform of the graphics state (especially when emitting such actions would only complicate the optimization algorithms).

```
function VisitTriangles (SceneGraphNode node, RenderGraph graph)
{
    // Precondition: node is a scene graph leaf node containing
    // a set of triangles to be displayed

    if (node has a translation applied to it)
    {
        graph.AddRenderAction (TRANSLATE);
    }

    if (node has a translation applied to it)
    {
        graph.AddRenderAction (ROTATE);
    }

    if (node has a translation applied to it)
    {
        graph.AddRenderAction (SCALE);
    }
    for (each property attached to node)
    {
        // Properties are attributes that affect the render state
        // like polygon mode (e.g. solid or wireframe), blend function,
        // cull mode, depth buffering, and texture modulation function.
        graph.AddActionsForProperty (property);
    }

    for (each shader attached to node)
    {
        // Shaders are attributes that affect the appearance
        // of the triangles like materials, textures, vertex or
        // fragment programs, or high-level shader language programs.
        graph.AddActionsForShader (shader);
    }

    graph.AddRenderAction (DRAW_TRIANGLES);
}
```

Algorithm 3. The Render Graph Visitation function for a set of triangles.

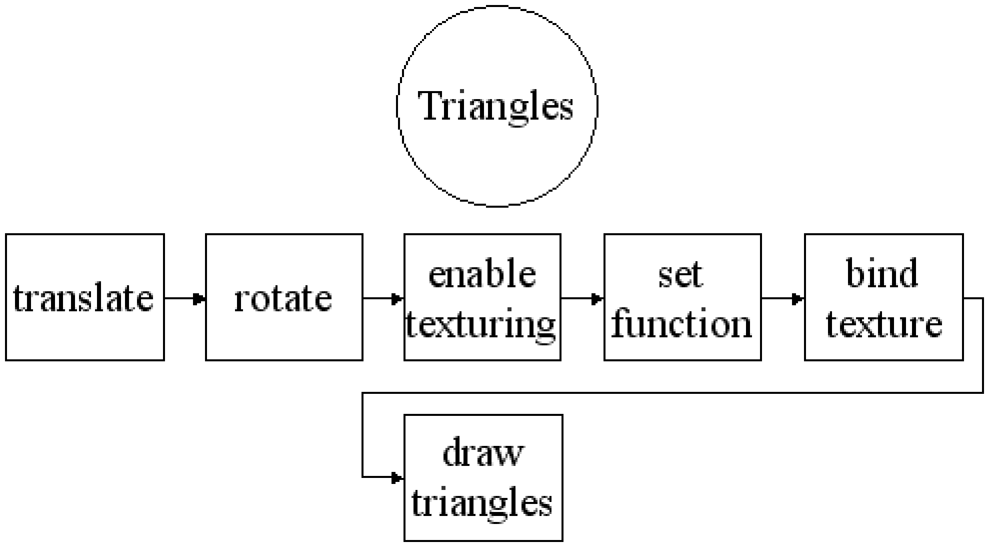

Figure 12a and Figure 12b. A simple scene graph (not shown is that the triangles have a texture applied to them) and the render graph generated from that scene graph.

As Algorithm 2 displays, node visitation is split into three phases: previsitation, visitation, and postvisitation. This is done to accommodate the order dependency inherent in many of the rendering operations, and also to simplify the allocation and deallocation of limited resources such as textures and lights. Phase order is fixed: previsitation is guaranteed to occur before visitation, and postvisitation is also guaranteed to occur after visitation. The children of the node being visited will be visited after the current node’s

Visit() function has executed, and before its Postvisit() function is executed. These functions need not be member functions of the scene graph library, but adding such support to the scene graph greatly simplifies render graph generation.

During previsitation, graphics hardware resources such as lights are allocated and enabled, the matrix stack is pushed if necessary to allow the local coordinate system to be modified without affecting the parent coordinate system, vertex, fragment, and/or shader programs are allocated, loaded, and enabled, and the alpha blending function is configured and enabled.

Visitation is where the bulk of the rendering process happens. During the execution of the Visit() function, lighting and material parameters are assigned, textures are bound, the local coordinate system is set up if necessary, and the geometry is downloaded to the graphics hardware. The traversal process then proceeds to the children of the node being visited (if it has any children) through the execution of the VisitChildren() function, which calls the BuildRenderGraph() function of Algorithm 2 on each child node.

Postvisitation is used to deallocate all resources no longer needed and to disable features no longer needed such as lighting and texturing so that the graphics pipeline is only performing the minimal number of calculations necessary to generate the display. An important feature of postvisitation is that it is done *after* the subgraph rooted at the node being visited has been traversed, so that inherited attributes such as the local coordinate system behave correctly.

4.2.2 Group Actions

Subgraphs within the render graph can be organized into *group actions*, which have several uses. Group actions are useful for implementing multipass effects (see section 4.2.3 below), performing optimizations such as common action elimination (similar to the common subexpression elimination performed by compilers), and as the integration point of the output of an interactive shading language compiler (such as the OpenGL shading language described in [Peer00] or the Interactive Shading Language described by [Prou01]). The shading language compiler could either be architected to directly emit a stream of render actions, i.e. a back-end would be added to that compiler that targets the render graph as opposed to another graphics library, or a post-processor can be written to transform the calls to a graphics library the shader compiler emits into render actions.

4.2.3 Multipass Effects

Computer games such as id Software's Quake 3 [Jaqu99] and Bethesda Softworks' The Elder Scrolls 3: Morrowind [Beth02] make extensive use of visual effects built through the composition of multiple passes over the scene's geometry, with different colors, lights, and/or textures applied to that geometry. Use of these techniques are now quite common in the game development industry, and are instrumental in the implementation of effects such as more complex lighting and shadow effects, the simulation of natural phenomena like water ripples or clouds and other gaseous effects, or the use reflection and refraction effects to better simulate transparent and/or translucent surfaces.

These effects can be implemented in one of two ways given the architecture of the render graph: Either the scene-graph nodes can be visited several times, once per render pass, and the necessary render actions will be generated upon each visitation, or they can be visited only once, with each pass represented by a separate group action containing the render actions necessary to implement that pass. Note that the render graph is not recreated upon each traversal! Since each pass is usually substantially similar to the previous pass, many render actions need not be recreated, only referenced avoiding expensive memory allocation operations.

Given that each pass of a multipass effect is encapsulated within a render group, this implies that each frame of the display is then composed of a set of at least one render group containing the necessary actions to generate the frame buffer contents.

4.2.4 Multiplatform Support

Each render action's semantic meaning is implemented through its render() function. The implementation of this function can be varied to provide support for different graphics libraries (e.g. OpenGL or Direct3D) or different hardware (e.g. Microsoft's Xbox or Nintendo's GameCube). This is a result of the render action being an instance of the Facade pattern [Gamm94]. Indeed, the hardware supported can be varied independently of the software (i.e. graphics library) supported, with the caveat that if you want to support *m* hardware platforms and *n* graphics libraries there must be $m * n$ implementations of the render() function.

Further platform independence is achieved through the use of standard technologies such as ANSI/ISO C++ to implement the algorithms described within this thesis.

4.3 Render Graph Optimization

A key benefit of the render graph is the ease of which the graph actions can be reorganized to improve display speed, take advantage of machine-specific hardware and/or software features, and provide for the most efficient implementation of sophisticated features such as transparency or programmable shading in the context of a retained-mode polygon display library.

4.3.1 Coherency

The organization of the render graph into an ordered set of render actions allows the implementation to take advantage of the temporal coherency of the scene graph. Between any two frames of the displayed content, there frequently has not been a lot of change in the structure of the scene graph, or in the related parameters used to generate the display. We can take advantage of this similarity to reduce the number of render actions that need to be created, deleted, or changed. This is especially important because memory allocation and deallocation can be time-expensive operations, so any memory management that can be avoided should be avoided to reduce the amount of time necessary to generate the frame buffer's contents.

4.3.2 Interpreter Design

Due to the similarity between the render graph's stream of render actions and the stream of instructions executed by a bytecode interpreter, we can take advantage of some optimization techniques developed by the interpreter-implementation community. For example, the core routine used to implement each render action is a switch statement whose predicate is the render action type. This is no different from the core routine of a bytecode interpreter, where it switches based upon the opcode type. We can take advantage of implementation techniques such as threaded code to improve the performance of the core routine.

4.3.3 Hardware-Specific Optimization

Each render action is implemented via its display() function. The implementation of this function is affected both by the target graphics library (e.g. OpenGL or Direct3D) and by the target hardware (e.g. NVIDIA's GeForce3 or ATI's Radeon 2). The current implementation of the render graph in the NeoTech content presentation engine [Zeni00] uses several OpenGL extensions as defined and implemented by NVIDIA [NVID00] to use NVIDIA's GeForce 3-class hardware [Lind01] in the most efficient way possible. If these extensions are not available (for example, if the machine has a graphics card not manufactured by NVIDIA) then the NeoTech engine uses slower routines that the OpenGL 1.3 specification guarantees will be available in every standard-compliant implementation of OpenGL.

Note that the render action's display() function is an instance of the Facade design pattern [Gamm94]. The use of the Facade pattern provides both a uniform interface between the low-level graphics library and the render graph management functions and a path to provide for graceful degradation of graphics features if the underlying hardware does not support a necessary function through software implementation of that function.

4.3.4 Analysis

Do these optimization algorithms offer a benefit over existing scene graph libraries? No other library takes the "graphics library as instruction set" paradigm to its logical conclusion and applies that abstraction to the entire display system. This allows the render graph to use algorithms from the interpreter design literature to execute the action stream efficiently.

No other library fully decouples the execution of graphics library instructions from the traversal of the scene graph, which allows the render graph to reorganize the execution sequence of the graphics library instructions to solve all of the problems detailed above.

More importantly, these two attributes of the render graph allows optimizations that take advantage of spatial and temporal coherency. Since the scene graph is a relatively static structure (it only changes if the scene changes) other approaches find it more difficult to take advantage of these kinds of coherency, involving manipulation of the scene graph itself or increasingly complicated traversal algorithms. Since the render graph is in effect the sequence of graphics-library instructions that must be executed to generate the display, it provides everything necessary to do analysis of the run-time complexity of the scene-graph display. To perform this analysis in another scene-graph display package a data structure remarkably similar to the render graph would be required.

The decoupling of the execution from the traversal also allows the render graph to be applied to any scene-graph library (although, as discussed above, a little support from the scene graph greatly simplifies the implementation of the render graph). Clearly defining the execution interface simplifies the support of multiple hardware platforms though the implementation of hardware-specific back ends via the Facade pattern.

5. Results

The render graph structure was implemented as part of the NeoTech content presentation engine developed for ZeniMax Technology, Inc. [Zeni00], and used in several technology demonstration projects there.

5.1 Implementation Details

The render graph was implemented in ANSI/ISO C++ for the NeoTech project at ZeniMax Technology, Inc. Graphics library interfaces were implemented for OpenGL 1.2 (requiring the extensions GL_ARB_multitexture , GL_ARB_texture_compression , GL_NV_vertex_array_range , and GL_NV_fence for basic support and GL_NV_vertex_program , GL_NV_texture_shader , GL_NV_register_combiners, and GL_NV_register_combiners2 for NVIDIA GeForce3 support [NVID00]) and the DirectX 8.0 version of Direct3D [Micr00].

5.1.1 Memory Management

Memory management is crucial to the practical viability of the render graph. Since so many render actions are generated each frame, their allocation and deallocation overhead can have a huge impact upon the performance of the rendering loop.

Each render action is allocated from a memory pool, which is a specialized allocator tuned to provide constant-time allocation and deallocation of identically sized elements with no resulting fragmentation of the pool memory.

The memory pool can be easily be replaced by a garbage collector, such as the mostly-copying conservative collector implemented by Boehm and Weiser for the C language [Boeh88]. Use of a C language garbage collection library has the advantage of being able to be used for all of NeoTech’s memory management, not just the allocation and deallocation of render actions.

6. Conclusions and Future Directions

In this thesis, I have described in detail a new method of implementing the display of scene graph data. This method consists of a data structure called the render graph, which is built during a traversal of the scene graph. The render graph is a single data structure that addresses all of the problems encountered during the display of scene-graph data.

The easily rearrangeable nature of the render actions address the problems of camera-space dependencies, as actions can be reorganized as necessary so that a simple linear traversal of the render graph visits each action in the correct order. Aggregate actions such as the group action simplify the organization of the render actions, and simplify the implementation of multipass effects where a set of actions need to be evaluated several times in sequence.

The render graph addresses the problem of using the graphics hardware in the most efficient fashion through analysis and modification of the stream of render actions. By carefully tracking the render state during traversal of the scene graph, the emission of redundant render actions can be suppressed, and the render action stream can be reorganized to expose further redundancies and otherwise minimize the number of render-state changes.

The render graph has an important side effect of providing an abstraction layer that offers a measure of platform independence. This has been demonstrated through support of both the OpenGL and Direct3D graphics libraries in the NeoTech content presentation engine.

Future development of the render graph will concentrate on support for new rendering technologies, such as the programmable geometry hardware now being sold by companies such as NVIDIA and ATI in their GeForce 4 and Radeon 2 products respectively, and support for real-time shading languages like the ones being developed at SGI [Peer00], NVIDIA [NVID02], and Stanford [Prou01].

Reorganization of the render actions offers many more possibilities for further optimization of the display process, and for support of technologies like the programmable hardware mentioned above.

6.1 Acknowledgements

ZeniMax Technology, Inc., a division of ZeniMax Media, Inc., and the Institute for Computer Graphics at the George Washington University has supported this work. I would like to thank in particular my manager at ZeniMax during much of NeoTech's design and implementation, Gregory Pierce, and the members of the NeoTech team, O'Dell Hicks, Guy Carver, and Scott Franke, for their insights into some of the more difficult problems. I would also like to thank my thesis advisor at GWU, Dr. James Hahn, for his patience and assistance.

7. References

[Akel93] Akeley, K., "RealityEngine Graphics," *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp. 109-116, August 1993.

[ATI02] ATI Corporation, Radeon 8500 Technical Specifications, 2002. http://www.ati.com/technology/hardware/radeon8500/index.html.

[Bald02] Baldwin, D. OpenGL 2.0 Shading Language White Paper, 3D Labs, 2002. http://www.3dlabs.com/support/developer/ogl2/whitepapers/.

[Beth02] Bethesda Softworks, Inc. The Elder Scrolls 3: Morrowind. May 2002.

[Birc97] Birch, P., D. Blythe, B. Grantham, M. Jones, M. Schafer, M. Segal, and C. Tanner, *An OpenGL++ Specificatio*n. SGI, March 1997.

[Bish98] Bishop, L., D. Eberly, T. Whitted, M. Finch, and M. Shantz, "Designing a PC Game Engine," *IEEE Computer Graphics and Applications*, pp. 46-53, Jan./Feb. 1998. http://www.ndl.com/bishop.pdf.

[Blin96] Blinn, J. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1996.

[Boeh88] Boehm, H.-J., and M. Weiser, "Garbage Collection in an Uncooperative Environment," *Software – Practice and Experience* 18(9), pp. 807-820, 1988.

[Crit00] *RenderWare Graphics*, Criterion Technologies, Ltd., Guildford, Surrey, UK, 2000.

[Eber00] Eberly, D., *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann, San Francisco, CA, 2000.

[Fole90] Foley, J., A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice*, Addison Wesley Publishing Company, Reading, MA, 1990.

[Gamm94] Gamma, E., R. Helms, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Publishing Company, Reading, MA, 1994.

[SGI91] *Graphics Library Programming Guide,* Silicon Graphics Computer Systems. Mountain View, CA, 1991.

[Inte98] Intel Corp., *Accelerated Graphics Port Interface Specification Revision 2.0*, May 1998, http://developer.intel.com/technology/agp/agp_index.htm.

[ISO88] International Standards Organization, "International Standard Information Processing Systems – Computer Graphics - Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description," ISO Document Number 880.5: 1988(E), American National Standards Institute. New York, NY, 1988.

[ISO97] International Standards Organization, "Virtual Reality Markup Language," International Standard ISO/IEC 14772-1:1997. http://www.vrml.org/.

[Jaqu99] Jaquays, P., and B. Hook. "Quake 3: Arena shader manual, revision 10." *Game Developer's Conference Hardcore Technical Seminar Notes*, C. Hecker and J. Lander, Eds., Miller Freeman Game Group, December 1999.

[Knut97] Knuth, D., *The Art of Computer Programming Volume 1: Fundamental Algorithms*, Third Edition, Addison Wesley Longman, Reading, MA, 1997.

[Lind01] Lindholm, E., M. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," *Computer Graphics (SIGGRAPH '01 Proceedings)*, August 2001.

[McCo00] McCool, M., *SMASH: A Next-Generation API for Programmable Graphics Accelerators*. Technical Report CS-2000-14, University of Waterloo, August 2000.

[Micr00] Microsoft Corporation, *Microsoft DirectX 8.0 DirectX Graphics,* October 2000, http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/directx/dx8_c/hh/directx8_c/_dx_directx_graphics_graphics.htm.

[Micr02] Microsoft Corporation, *Microsoft DirectX 9.0 Direct3D Shading Language,* June 2002.

[Moll99] Moller, T., and E. Haines, *Real-Time Rendering*, A.K. Peters Ltd., Natick, MA, 1999. http://www.realtimerendering.com.

[Moln92] Molnar, S., J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics (SIGGRAPH '92 Proceedings)*, pp. 231-240, July 1992.

[Mont97] Montrym, J., D. Baum, D. Dignam, and C. Migdal, "InfiniteReality: A Real-Time Graphics System," *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 293-302, August 1997.

[NDL99] *NetImmerse Programmers Manual,* Numerical Design Ltd., Chapel Hill, NC, 1999. http://www.ndl.com.

[NVID00] NVIDIA Corporation, *NVIDIA OpenGL Extension Specifications*, October 2000. http://www.nvidia.com/developer.

[NVID02] NVIDIA Corporation, *Cg Language Specification*, June 2002. http://developer.nvidia.com/cg.

[ARB93] OpenGL Architecture Review Board, J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, Reading, MA, 1993. http://www.opengl.org.

[Peer00] Peercy, M., M. Olano, J. Airey, and P. Ungar, "Interactive Multi-Pass Programmable Shading," *Computer Graphics (SIGGRAPH '00 Proceedings)*, pp. 425-432, July 2000.

[Port84] Porter, T., and T. Duff, "Compositing Digital Images," *Computer Graphics (SIGGRAPH '84 Proceedings)*, pp. 253-259, July 1984.

[Prou01] Proudfoot, K., W. Mark, S. Tzvetkov, and P. Hanrahan, "A Real-Time Procedural Shading System for Programmable Graphics Hardware," *Computer Graphics (SIGGRAPH '01 Proceedings)*, pp. 159-170, August 2001. http://graphics.stanford.edu/projects/shading/.

[vanD88] PHIGS+ Committee, Andries van Dam, chair, "PHIGS+ Functional Description, Revision 3.0," *Computer Graphics*, vol. 22, no. 3, pp. 125-218, July 1988.

[Rohl94] Rohlf, J., and J. Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," *Computer Graphics (SIGGRAPH '94 Proceedings)*, pp. 381-394, July 1994. http://oss.sgi.com/projects/performer.

[Rost02] Rost, R. OpenGL 2.0 Overview, 3D Labs, 2002. http://www.3dlabs.com/support/developer/ogl2/whitepapers/.

[SGI98] SGI Technical Publications, *Cosmo3D Programmer's Guide*, SGI Technical Publications, 1998.

[HP91] *Starbase Graphics Techniques and Display List Programmer's Guide*. Hewlett-Packard Company, Fort Collins, CO, 1991.

[Sega01] Segal, M., and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.3)*. http://www.opengl.org.

[Stra93] Strauss, P., and R. Carey, "An Object-Oriented 3D Graphics Toolkit," *Computer Graphics (SIGGRAPH 93 Proceedings)*, pp. 341-349, August 1993. http://oss.sgi.com/projects/inventor.

[Suth63] Sutherland, I.E., "Sketchpad: A Man-Machine Graphical Communication System," *Proceedings of the Spring Joint Computer Conference*, Spartan Books, Baltimore, MD, 1963.

[Torb96] Torborg, J., and J.T. Kajiya, "Talisman: Commodity Realtime 3D Graphics for the PC," *Computer Graphics (SIGGRAPH '96 Proceedings)*, pp. 353-363, August 1996.

[Upst90] Upstill, S., *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.

[Watt92] Watt, A., and M., Watt, *Advanced Animation and Rendering Techniques--Theory and Practice*, Addison-Wesley, Workingham, England, 1992.

[Wern97] Wernecke, J., *The Inventor Mentor*, Addison-Wesley, Reading, MA, 1994. http://oss.sgi.com/projects/inventor.

[Zeni00] *NeoTech Content Presentation Engine,* ZeniMax Technology, Inc., Rockville, MD, 2000. http://www.zenimax.com..