CMPS 203: Final Project

A Survey Of Graphics Programming Languages

Jerry Yee
Department of Computer Science
University of California, Santa Cruz
yee@soe.ucsc.edu
December 2, 2004

Edited By: Oliver Wang

Abstract

While digital graphics have increased in complexity in many industries such as scientific visualization and entertainment, developing graphics software and 3D models have become quite difficult. To simplify the development process, the industry has developed new graphics languages to accommodate various requirements and extending these languages as needs arise. OpenGL is the most widely used, supported, and documented 2D and 3D graphics programming language. In this paper, I survey five graphic programming languages (GPLs) and attempt to discover why OpenGL is the most popular GPL. I will explore the purpose, advantages, and disadvantages of OpenGL, Direct3D, Cg, VRML, and Fran.

1 Introduction

In *The Lord Of The Rings: The Two Towers*, 920 digital effect shots were used to create elaborate 3D environments. It took ten months to render the graphics on 1000 SGI and IBM Linux workstations [4]. As movies require more elaborate and complex digital graphics, graphics hardware architectures and programming languages evolve to accommodate these needs. Not only is this occurring in film, but also in other applications such as 3D web browsing, scientific visualization, and computer games. Various industry groups and researchers have been extending current graphics programming languages (GPLs) and creating new ones at the same time.

Over the past several years, dramatic advances in computer architecture and graphics hardware have opened doors to more complex 3D scenes that were unimaginable a decade ago. By utilizing these hardware resources, computer graphics have become more realistic. But in order to describe these complex 3D scenes in a general purpose programming language, programs have grown in complexity and size. They have become unmanageable and difficult to maintain, reuse, and enhance. This has brought forth the need to develop domain-specific languages (DSL) for computer graphics.

Through appropriate abstractions and notations, these domain-specific languages have enhanced productivity and made it possible to produce more readable, compact, portable, and reusable code. They provide a language that uses the theory and vocabulary of the graphics domain, and most filter out concepts of the general purpose programming languages

that are not needed for computer graphics programming. Therefore GPLs are generally considered less comprehensive that general purpose languages, but more expressive in the graphics domain.

This layer of abstraction was created to help solve some problems of past 3D computer modeling techniques. In the past, each object in a scene was programmed separately. However modeling often requires recreating objects that are similar to existing objects with some minor alterations. Modeling also requires applying the same repetitive steps or processes to each object in the scene, often with different parameters (e.g., moving each object in different directions). Since designing and rendering an object is quite expensive and time consuming, a GPL has to allow reuse of previously designed objects and the ability to modify those objects with various parameters. Those parameters may change according to time for animations, or they may change the point of view as the viewer moves about in the scene. Modeling is inherently a functional process operating on objects as known in object oriented programming. And like function application in functional programming, operations are applied to objects to produce new objects.

In real-world 3D modeling, objects and animation are viewed as being smooth and continuous. For example, lines are straight, spheres are round, and animation sequences are continuous. However computers and general programming languages view things in discrete terms. Lines are a sequence of colored pixels on the monitor's display forming a jagged line. Spheres consist of triangles to approximate the shape of a sphere. Animation sequences consist of static images, one image for each discrete moment in time of the animation. GPLs are able to integrate these discrete concepts into the language, so that they become transparent to programmers. This allows programmers to think about their scenes or animations in continuous terms, rather than change their natural understanding of modeling to fit the computer architecture's handling of modeling.

Various groups have developed these abstractions in various new GPLs. These languages can be divided into two distinct types. One type is freestanding. These are new languages, totally independent of any existing general programming language and not extensions to any host language. This allows for total control of the language, from the design phase all the way through the deployment of the new language. When one programs in such a language, only the functionality defined in the languages specifications may be used. Another type is embedded. These languages are dependent on an existing general programming language and are extensions of a host language. These languages cannot exist apart from its host language. Often these embedded languages provide an extra layer of abstraction between the graphics programmer and the underlying general programming language, such as an application programming interface (API). The abstraction can be a superset of an existing language, with extra syntax, keywords, and functions built on top of the existing language. Embedded languages are easier to develop and maintain, and therefore are more widely available. An example of a freestanding GPL is VRML. Some examples of embedded GPLs are OpenGL, Direct3D, Cg, and FRAN.

However researchers have encountered several disadvantages of DSLs, because of their added design, implementation, and maintenance costs. When designing a graphics language,

researchers have to define the scope of the new language clearly. Will the GPL be used for 2D or 3D modeling? Will it support animation? Will the language be transmitted through Internet applications? When implementing a language, researchers must also develop debugging tools and documentation for the new language. As with any new language, it is difficult for software developers to become familiar and productive in the new language. And as with any higher level of abstraction, there is always a potential lost of performance when using computer generated code versus hand-coded software at a lower level of abstraction (e.g., in an assembly language).

In this paper, I will explore these languages and attempt to answer these questions: Why is OpenGL the most widely used graphics language today? Why was the domain-specific language created? What were the design principles? How does it affect the software engineering process? What are the tools and environments needed to support the language? What are the technical advantages and disadvantages? Do the abstractions make programming easier?

2 OpenGL: Open Graphics Library

OpenGL is a software interface to graphics hardware. More specifically, it is an API for 2D and 3D graphics, providing abstractions and operations for modeling, animation, and visualization. It was created to provide high-quality and high-performance capabilities to graphics software applications in all industries, such as broadcasting, computer aided design, computer aided manufacturing, entertainment, and medical imaging. To support such a wide range of applications, OpenGL is supported across most operating systems, and the API is available for C, C++, Java, Fortran, and Ada programming languages. OpenGL has become the most widely used, supported, and documented GPL since its introduction in 1992.

OpenGL is maintained by the OpenGL Architecture Review Board (OpenGL ARB), which consists of industry representatives, providing a broad range of experience and knowledge to the development of OpenGL. Not only did they design the original OpenGL standard, they also ensure that future enhancements to the language will adhere to the original goal - to make OpenGL an industry standard across all platforms - by reviewing OpenGL enhancement requests and defining conformance tests. OpenGL is a truly open standard.

In order to establish OpenGL as an open standard, OpenGL must be stable, reliable, portable, scalable, evolvable, easy to use, and well documented. To provide a stable language, software developers must have time to adopt any language modifications and backward compatibility with older software applications must be kept. Thus the OpenGL ARB carefully controls language enhancements and modifications to the OpenGL specifications.

OpenGL made several key design requirements in order to provide a reliable language. First, the modeling state must be consistent with the complete execution of all previously invoked OpenGL commands. OpenGL is a sequential language, as it processes the commands one at a time as they are received. So the previous object must be drawn completely, before it is queried or the next object is drawn by subsequent commands. Second, OpenGL binds data on call. Function arguments are interpreted when the function is called. Even if the argument is a pointer, the pointer data is interpreted at function call, so any subsequent

changes to the pointer data will not effect the function. Third, floating-point errors should not cause the application to terminate, since OpenGL performs a considerable amount of floating-point operations. Any number is acceptable to any OpenGL floating-point function argument. Non-floating numbers will produce unspecified results, but will not cause the application to terminate. Likewise division by zero will produce unspecified results, but will not cause termination.

To be a portable language, the OpenGL ARB has required that all implementations of OpenGL deliver the same display results regardless of the operating system, computer architecture, or API version. Supported operating systems include MAC OS, UNIX, Windows, and Linux. API versions for C, C++, Java, Fortran, and Ada are thoroughly tested to ensure that they conform to the language specifications and produce correct results and behaviors.

To provide a scalable language, OpenGL applications can run on systems of all classes, from portable electronics to super computers. For this purpose, two different API implementations do not always have to agree on a pixel-to-pixel basis, but rather they have to approximate the same output.

To provide an evolving language, OpenGL has an extension mechanism to allow software developers to access new hardware capabilities. This mechanism allows software developers to step outside of the current OpenGL standard in order to take advantage of new hardware and technologies without waiting for an OpenGL specification enhancement. Thus allowing software developers to tailor OpenGL to their specific needs, which may include extracting higher hardware performance or porting the application to a consumer electronic device. Different OpenGL implementations may divide the processing time between the graphics processing unit (GPU) and the central processing unit (CPU), depending upon the function. So this mechanism also allows software developers to customize and explicitly executed certain OpenGL commands on the GPU or CPU.

To be an easy to use language, OpenGL has been designed to be a modeling language, containing efficient and simple routines to develop graphics software. It includes functionality for creating simple geometric shapes (e.g., lines and polygons) to highly complex lighting models. It allows for easy manipulation of how those objects are rendered through matrix transformations, coloring and blending functions, and other features. It is structured and designed to be intuitive to any graphics programmer. The language hides any operating system details (e.g., creating windows) and hardware specific information (e.g. handling frame buffers that store the current scene), so the programmer can concentrate on the graphical portion of the application and produce programs with less lines of code. However, OpenGL scope covers only what is rendered into the frame buffer. It does not support peripherals that are often associated with graphics hardware such as mouse and keyboard input devices. Developers must use other methods to handle those devices. And while OpenGL does have commands to control how complex geometries should be rendered (e.g., color, transformations), it does not have commands to create complex geometric shapes such as spheres or toruses. Developers must use OpenGL library extensions to create these object.

And finally, to provide a well documented language, numerous manuals, books, and program samples have been publish, and many are freely available on-line.

3 Direct3D: A Component Of DirectX

Microsoft DirectX is a suite of technologies for designing and implementing Windows-based applications. It contains C, C++, and Visual Basic APIs for graphics, video, music, and sound programming. The graphics API are included in the Direct3D component of the DirectX suite. Direct3D was originally created by 3D game developers and was later purchased by Microsoft in 1996.

Direct3D is a high-level 3D modeling language with low-level APIs to access graphics hardware capabilities. Direct3D has many of the same design goals as OpenGL, mainly ease of use, portability, composability, performance, extensibility, and backward compatibility. Microsoft wanted to encourage developers to make applications for its Windows operating system. Because Microsoft Windows originally did not support 3D graphics, software vendors had a difficult time writing 3D applications and games for Windows. So Microsoft released Direct3D to provide an easier development environment, by providing a language consistent with their COM (Component Object Model) APIs in DirectX. Direct3D also supports portability between computer architectures. Prior to Direct3D, software vendors had to customize their applications for different hardware configurations. However Direct3D allows software developers to access the graphics hardware through a single API, regardless of the underlying hardware architecture. With Direct3D, software developers can compose high-performing graphics on any Windows-based system. It also exposes some of the underlying low-level, hardware interfaces and provides extension mechanisms for developers who need extra performance. With each release, Direct3D has remained backward compatible, by continuing to support all interfaces and objects included with previous releases.

In contrast to OpenGL, Direct3D is owned and maintained by Microsoft for Windows development. Although Windows is the most widely used operating system for home computing, it is only one among several operating systems. Direct3D does not support users on Unix, Linux, and MAC OS. And currently, there is no Direct3D API for Java programmers. The audience of Direct3D is not as diverse as OpenGL.

A great benefit of Direct3D is that it is packaged with the DirectX suite of technologies, allowing Direct3D software to easily interface with other Windows features. With DirectX technologies, programmers may access sound cards, memory, input devices, and networking capabilities. By itself Direct3D may not be as powerful as OpenGL, but packaged with DirectX, it is the premier graphics language for Windows-based software.

4 Cg: C for Graphics

Cg is a shading language used to generate real-time graphics on a programmable graphics processing unit (GPU). A shading language tells the GPU how to shade the display pixels. Cg was developed and released by NVIDIA in 2002 to take advantage of the performance and technology of their latest real-time graphics architectures that consist of programmable vertex and fragment stream processors. It was also designed to support future generations of computer graphics hardware.

The Cg language reflects similar objectives as those of the C programming language. Both are hardware-oriented languages. But while C was designed to be a general programming language for the CPU, Cg was designed to be a general purpose programming language for the GPU. In fact the necessity of Cg rose from the fact that graphics hardware has evolved from being merely configurable to being highly programmable. Also current high-level languages are insufficient for GPU programming and programming in assembly code is tedious. So the Cg language was developed to support all the unique architectural features of the GPU hardware, such as fast floating-point operations and texture mapping capabilities.

Many of Cg's language design goals mirror that of OpenGL, such as high performance, ease of use, and portability. Cg data types and operators map directly to GPU hardware operations, so the run-time cost of each operation is apparent and not hidden in some abstraction. Also Cg uses a "multi-program" model. This requires one program to process the object's vertices and pass the resulting fragments to a second program to be processed. This closely matches the underlying GPU architecture (consisting of a vertex processor and a fragment processor), making it easier for programmers to estimate the performance of the code and write high performance code. Furthermore the Cg language and compiler are designed to out-perform any hand-written GPU assembly code.

Cg provides a high-level language, with syntax and semantics similar to C, to allow GPU programmers to be more productive and to generate compact and portable code. Cg is basically C modified in certain areas to support a multi-program model used by programmable GPUs. Like C, Cg provides structures and arrays, arithmetic operators, booleans and boolean operators, function definitions, and most C flow constructs (i.e., do, while, for, if, break, continue). Cg does not permit recursive function definitions, pointers, bitwise operations, and lacks most C++ object oriented features. Current GPUs are not capable of indirect addressing and cannot dereference pointer types, and so all function parameters are passed call-by-value. For the same reason, it impossible to create a runtime stack to store temporarily variables for recursive function calls, so recursion is prohibited. In addition to the C datatypes mentioned above, Cg also has vector, matrix, and boolean datatypes. Unlike C, Cg constants are typeless for the purposes of type promotion. This is important when promoting numeric constants, which may unintentionally promote an integer into a floating-point, resulting in a runtime performance penalty.

The designers want programs to be portable across hardware implementations and generations and across operating systems. However instead of mandating that all implementation support the complete set of Cg features, Cg keeps profiles of different implementations on various hardware architectures. These profiles indicate the subset of Cg features that are supported by a particular implementation. These profiles allow functions to be overloaded for each computer architecture, allowing for highly optimized functions. Cg is currently supported on eighteen different profiles using Windows, Linux, and MAC OS operating systems.

The Cg designers' goals also include support of graphics hardware functionality, extensibility for future graphics hardware, and support for non-shading use of the graphics hardware. The Cg language supports all of the functionality that the GPU assembly language can support. The language is extensible to support future GPU implementations, while maintaining

backward compatibility. And the language is general enough to support other tasks of the GPU, although shading and transformation are the most commonly used operations of the GPU. In this sense, its designers do not consider Cg to be a DSL but rather a general purpose programming language for graphics architectures.

However because Cg was design to be a more general purpose programming language, it does not support many of the abstractions that OpenGL supports. Cg does not provide support for surface and light shaders or use domain-specific information to perform optimization. This avoids the situation in which the language abstractions do not match the user's desired abstraction. And it avoids the situation when the language abstraction does not easily match that of the underlying graphics hardware. Cg does not hide the low-level functionality and therefore does not hide the underlying run-time costs of language operators. Therefore a graphics programmer still needs to utilize the OpenGL API to model their scenes. Cg actually operates above the OpenGL layer. The Cg compiler produces assembly code and passes it to the OpenGL extension mechanisms to be executed on the GPU.

5 VRML: Virtual Reality Modeling Language

VRML is a language for describing interactive 3D worlds distributed on the Internet and is interpreted by a VRML browser. The VRML designers' main goal is to unite and standardize the Web technology to generate 3D worlds. VRML was originally created and released by Silicon Graphics Inc (SGI) in 1995 as the standard 3D format for static scenes on the Internet. The VRML Architecture Group (VAG) was formed in 1995, consisting of researchers from various companies and universities who oversee the development of VRML. They released VRML 2.0 in 1996, which included new features for enhanced static worlds, object interaction, animation and behavior scripting, special effects, and object prototyping. VRML can represent static and animated objects and may have hyperlinks to sounds, movies, and other images.

VRML provides a high-level modeling abstraction to describe 3D scenes. It organizes the 3D scene as a hierarchical graph of objects called nodes. The 3D scene is rendering by traversing nodes, starting at the top and traversing down the graph breadth-first. Nodes may describe a shape, property, group of nodes, or a hyperlink. Shape nodes describe the geometry of physical objects in the scene. Unlike OpenGL, VRML supports complex 3D objects like spheres and cones. Property nodes describe the appearance of all successive shape nodes in the graph. Properties include material, color, textures, camera, and lights. At any node, the state may be changed (such as changing the drawing color property), thus affecting how successive shape nodes are rendered. Group nodes allow grouping of property and shape nodes into one unit, and thus limiting the property change to the shapes within the group.

The design goals for VRML includes performance, portability to various computing platforms, scalability to support both simple and very complex 3D scenes, composability and reuse of 3D objects within a scene, authorability allowing development of application editors and importation of other industry formats, multi-user environment capability, open stan-

dardization of the language, backward compatibility, completeness to support the needs of the industry, extensibility to add new elements when necessary, and backward compatibility. Using VRML, viewers may navigate freely throughout a 3D scene in a VRML browser. Browsers for VRML are widely available for many different platforms. VRML authoring tools are also available for the creation VRML files.

VRML 1.0 contained several problems because it was based on SGI's Open Inventor that was laden with ambiguity. The original specification also did not ensure unique object names or provide a mechanism for object prototyping - to describe an object without actually instantiating it in the scene. Also because changing a single property node could influence the remainder of the graph, it is impossible for the interpreter to optimize the scene graph. Performance and portability are key requirements in the VRML specifications, because 3D scenes may be transferred across slow network connections and to various operating systems and hardware configurations on the Internet. In VRML 2.0, they improved performance by allowing the interpreter (i.e., VRML browser) to optimize the 3D scene graph.

6 Fran: Functional Reactive Animation

Fran is graphics domain-specific language embedded in the Haskell functional programming language. Haskell supports first class functions, assignments, static typing, polymorphism, and lazy evaluation. Fran is essentially a Haskell library that provides 3D modeling, animation, and reactive/interactive functionality. It consists of a collection of types, constants, and functions for 3D modeling. Fran was developed by Microsoft and released in 1999. It is available on Microsoft Windows platforms.

Fran was created to hide the low-level details of the underlying programming language, and to allow software developers to focus on the shapes, appearance, and behavior of their 3D models. Fran provides many of the same benefits as OpenGL. However Fran was primarily designed for reactive animation. For animation, Fran also provides ease of construction of animation sequences, composability of animation sequences into longer sequences, and regulation of animation sampling rates based upon the hardware architecture to ensure that animation time is consistent across platforms.

Like all imperative programming languages, Haskell is good for presentation-oriented programming [5]. Presentation-oriented programming focuses on how to present a 3D object (as a set of pixels), rather than what the object is (a set of shapes). So the designers of Fran really wanted to create a model-oriented programming language that describes the object as a collection of shapes, colors, textures, and materials. In the real world, objects have infinite resolution and movement is a continuous sequence of events. However, in the virtual world, objects are defined by a finite set of pixels and movements are animated by a discrete set of images displayed in sequential order. Fran's abstractions tie together the two worlds, allowing software developers to think more in terms of the real world when designing their 3D models.

Fran provides two basic abstractions - one for events and one for behaviors. They are defined as polymorphic types. Events occur at specific moments in time, whereas behaviors

evolve over a continuous period of time. A behavior is of a time-dependent type, such as time-dependent images (i.e., object image at a specific moment in time) and time-dependent integers (e.g., timers). An event is a list of pairs of a time-dependent type and a time value. Events represent a sequence of occurrences in time. These abstractions provide facilities to interact with the objects as they evolve in time and to program the objects to react to stimuli.

7 Conclusion

The five graphics domain-specific languages surveyed in this paper all have similar design goals like ease of use, composability, portability, scalability, and performance. However each was designed for a specific purpose that none of the other languages currently address. Of the five languages, OpenGL was designed for 3D modeling over a large spectrum of operating systems and hardware configurations. Direct3D was designed to make Windows-based 3D graphics development easier. Cg is a shading language, designed primarily to program NVIDIA GPUs. VRML was designed to model 3D scenes over the Internet for which performance and portability becomes a big issue. Fran was designed to make it easier to program 3D reactive animations.

While OpenGL has many great benefits, it is set apart from the rest due to its openness and extensibility, making it the most widely used language today. It is also the oldest of the five languages surveyed in this paper. From the get-go, OpenGL was designed to meet the needs of a wide audience and be available on every major platform. The OpenGL ARB consists of representatives from industry, universities, and private research groups, thus they are able to gather a wide range of feedback and to mold OpenGL according to industry needs. The extensibility mechanisms allow software developers to access new hardware features and permit other programming languages (such as Cg) to tap into the OpenGL language. This provides for an even broader range of capabilities.

OpenGL has a great future in computer graphics. Industry leaders such as NVIDIA are constantly working with the OpenGL ARB to transform widely used extensions into new OpenGL standards. As computer architectures and systems evolve, so does OpenGL. As graphics, modeling, and animation become more complex, the OpenGL abstractions will also grow to make those functionalities easier to use, more portable, and higher performing.

References

[1] Hideyuki Ando, Akihiro Kubota, and Takashi Kiriyama. Study on the collaborative design process over the internet: A case study on vrml 2.0 specification design. *Design Studies*, 19(3):289–308, July 1998.

[2] Wolfgang Broll and Tanja Koop. Vrml: Today and tomorrow. *Computer and Graphics*, 10(3):427–434, 1996.

[3] Helen Cameron and Peter King. Modeling reactive multimedia: Events and behaviors. *Multimedia Tools and Applications*, 19:53–77, 2003.

[4] Audrey Doyle. The two towers. *Computer Graphics World*, 26(2):28–32, February 2003.

[5] Conal Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May/June 1999.

[6] D. W. Fellner, S. Havemann, and G. Muller. Modeling of and navigation in complex 3d documents. *Computer and Graphics*, 22(6):647–653, 1998.

[7] VRML Consortium Incorporated. An overview of the virtual reality modeling language. *The Virtual Reality Modeling Language Specification*, August 1996.

[8] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.

[9] Microsoft Corporation, http://msdn.microsoft.com/library/en-us/dndxgen/html/directxovrvw.asp. *Microsoft DirectX Overview*, 2000.

[10] OpenGL Architecture Review Board, http://rush3d.com/reference/opengl-bluebook-1.0/front.html. *OpenGL Reference Manual*, 1994.

[11] Rich Thomson. *Direct3D vs. OpenGL: A Comparison.* http://roxen.xmission.com/ legalize/d3d-vs-opengl.html, 2001.