Automated Shader Generation using a Shader Infrastructure

Chris Coleman, Kevin Harris, Anthony Hinton, Anton Ephanov
MultiGen-Paradigm
Richardson, TX
chris.coleman@multigen.com, kevin.harris@multigen.com, anthony.hinton@multigen.com, anton.ephanov@multigen.com

ABSTRACT

Programmable rendering on the GPU (Graphics Processing Unit) utilizing “shader” technology has become a recognized benefit to visual simulation, providing unprecedented realism and fidelity to synthetic environments. However, effective use of shaders is a technological challenge, where implementation of even a trivial vertex or fragment shader requires manual re-implementation of the fixed-function pipeline. Furthermore, generation of efficient shaders for existing data is a complicated undertaking that is compounded by the number of state permutations possible under a modern graphics API, such as OpenGL, and the desire to extend fixed function state with advanced rendering techniques.

Therefore, system integrators and run-time providers face significant challenges in incorporating shader technology while hiding complexity and maintaining backwards compatibility with existing data. Preserving the massive investment in existing database and model libraries, while enhancing system capabilities, is a fundamental concern. Though some shader techniques are required for modeling, others are typically supplied by the run-time. Fundamentally, it is clearly disadvantageous to have run-time code contained in shared modeling assets.

As a solution to these problems, this paper introduces a Shader Infrastructure that automates the building of vertex and fragment shaders by analyzing the OpenGL state machine. The Shader Infrastructure is capable of not only dynamic generation of highly efficient shaders for rendering any legacy data considered valid by the OpenFlight™ standard, but also extending the rendering pipeline via Advanced Rendering Techniques. Techniques allow for implementation and merging of novel GPU-based rendering approaches by injecting small snippets of shader code into the Shader Infrastructure. The Shader Infrastructure allows designers to customize portions of the rendering pipeline and to automatically combine that customization with other rendering techniques, either fixed-function or shader-based. Therefore, it significantly simplifies the problem of content management and reuse while taking full advantage of the advances in the programmable PC graphics hardware.

ABOUT THE AUTHORS

Chris Coleman is a Senior Software Engineer at MultiGen-Paradigm. He is currently working on shader technology in both the Vega Prime run-time product and the OpenFlight™ format. Chris received a B.S. in Computer Science and a M.S. in Visualization Sciences from Texas A&M University (College Station, TX).

Kevin Harris is a Senior Software Engineer at MultiGen-Paradigm where he works on the Vega Prime run-time product. He has six years' experience as a 3D graphics programmer in both the game and simulation industries. He also served in the U.S. Navy as an Operations Specialist with an emphasis on Anti-Submarine Warfare and navigation.

Anthony Hinton is a Software Engineer at MultiGen-Paradigm where he works on the Vega Prime run-time product. He has nine years' experience in the software industry in the 3D graphics, cryptographic, and satellite communication fields. Anthony received a B.S. in computer science engineering from University of Texas at Arlington (Arlington, TX).

Anton Ephanov received B.S. and M.S. degrees in Mathematics and Mechanics from Moscow State University (Moscow, Russia) and a Ph.D. degree in Mechanical Engineering specializing in Robotics from Southern Methodist University (Dallas, TX). He is currently the Principal Architect for the Vega Prime run-time product at MultiGen-Paradigm.

Automated Shader Generation using a Shader Infrastructure

Chris Coleman, Kevin Harris, Anthony Hinton, Anton Ephanov
MultiGen-Paradigm
Richardson, TX
chris.coleman@multigen.com, kevin.harris@multigen.com, anthony.hinton@multigen.com, anton.ephanov@multigen.com

INTRODUCTION

Higher level shading languages, such as Cg from NVIDIA, the DirectX Higher Level Shading Language (HLSL), and the OpenGL Shading Language (GLSL), are the key to unlocking the potential of modern graphics processing units (GPUs). These languages are extremely expressive and allow for everything from applying environment reflections on surfaces to processing advanced physics simulations on the GPU. While the breadth of computational possibilities is impressive, there are clear advantages to using programmable hardware for typical vis-sim applications where simulated worlds are rendered photo-realistically. Creative use of shaders yields better lighting and shadowing cues (Stamminger & Drettakis 2002), helps better define the shape of a surface through bump mapping (Fernando & Kilgard 2003), makes realistic water simulation possible (Multigen-Paradigm 2006), and can include further details that help to immerse the viewer in a virtual world, or avoid artifacts present in past simulations. Rather than addressing general purpose computing using programmable graphics hardware, often called GPGPU, this paper will remain focused on shaders which render surfaces.

Writing dedicated vertex and fragment shaders that customize the rendering of selected geometries is commonplace in modern graphics applications, but what if we need to implement a new lighting model that requires all geometry to use vertex and fragment shaders? Writing these shaders for every piece of geometry in a scene is a daunting task, especially if one is forced to handle the vast number of rendering state permutations possible under modern graphics APIs such as OpenGL.

MOTIVATION

The promise of shaders is the ability to perform rendering techniques above and beyond what the fixed-function pipeline can do. New lighting models, shadows, light maps, environment maps, layered fog and many more advanced rendering techniques are possible with shaders. Implementation of such techniques either bypasses or overwrites large parts of the default OpenGL implementation that is referred to as the Fixed-Function Pipeline (FFP). The Fixed-Function Pipeline was hard-coded for efficiency in the previous generation of graphics hardware (and hence the name "fixed").

Therefore, the ability to modify and add to the FFP makes it necessary to first re-implement required portions of the FFP so that matching geometry can render correctly before new rendering techniques can be implemented. A failure to do so can result in undesired visual artifacts. Consider an example where a user adds a new light source to the scene during run-time. Shaders for all geometries affected by the light need to change accordingly. If a terrain skin and a building layer were modeled with a shader that only handles a single directional light source, other GL lights would not render. Figures 1 and 2 below illustrate the situation.

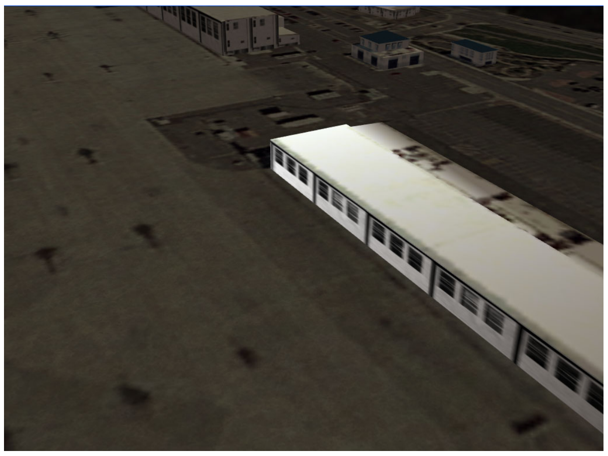

Figure 1. FFP rendering with directional light_0 and spot light_1

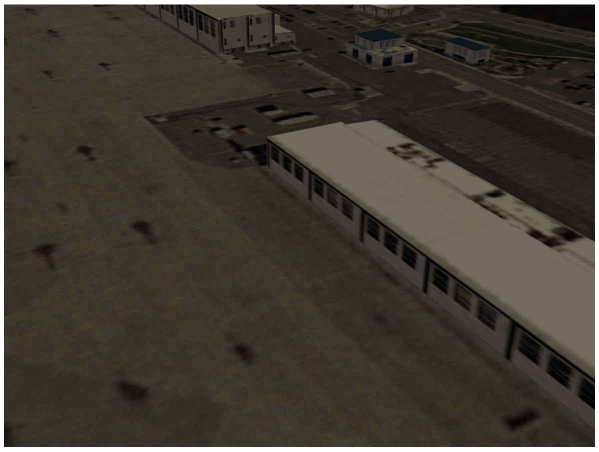

Figure 2. A basic shader provides no support for the spot light_1 so the building remains dark

Not only does the shader code need to account for all light sources in the scene, it also must implement their types correctly. In our example, even the light_0 light will not render correctly if the light is configured as a spot or positional light in the run-time. This creates a dilemma for the run-time. On one hand, implementing all of the OpenGL lights in a shader is computationally expensive, especially if only some of the lights are going to be used. On the other hand, adding or changing a light source requires re-generation of all affected shaders or maintaining a shader for each permutation of light configuration to account for the potential different lighting conditions encountered at run-time. Neither alternative is attractive. This introduces a content management challenge.

Fog is another FFP feature that is often problematic to combine with shaders. Mismatching fog is immediately obvious when the fog's visibility range is small and a different fog equation is running on different parts of the scene. Figures 3 and 4 illustrate the problem.

Lighting and fogging are visual effects which should alter virtually everything in the scene. If we applied a new lighting model to just a few pieces of geometry and used fixed-function lighting on the remainder, the shader-lit objects would look out of place in contrast to the scene's overall fixed-function lighting. In a similar fashion, if we applied a new fog model to some objects but not all, the realism of the fog effect would suffer if two objects, which are very close in proximity, received differing amounts of fog.

Nullifying visual anomalies caused by inconsistency in rendering state is typically non-trivial and in many cases the solutions tend to lower the quality of the newer technique for the sake of reconciling with the older technique.

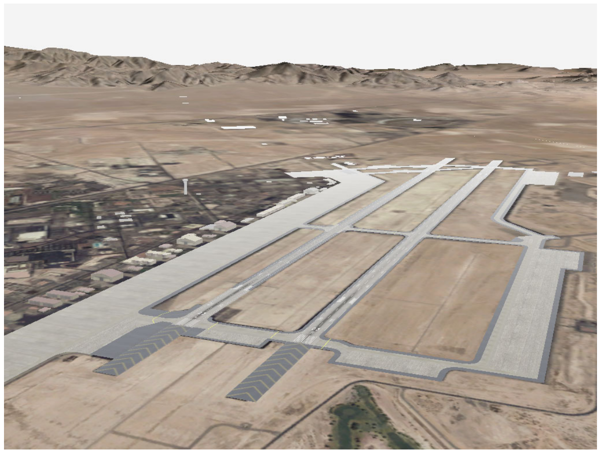

Figure 3. Shader rendering bump-mapped terrain without fog, but the airfield and buildings use FFP with vertex linear fog

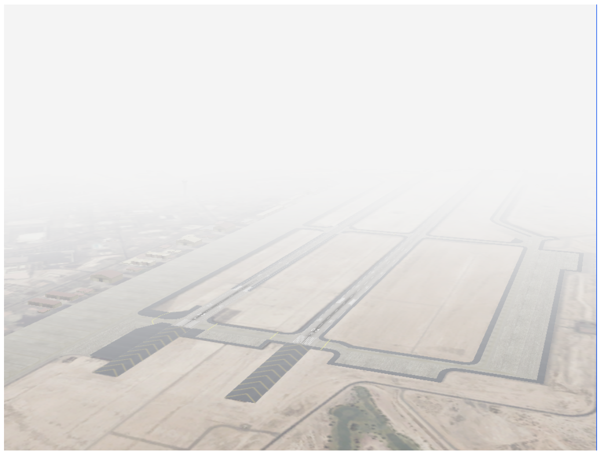

Figure 4. Shader rendering bump-mapped terrain matches the FFP vertex linear fog on the airfield and buildings

PREVIOUS WORK

Shading languages, such as Pixar's RenderMan, have long provided mechanisms to describe transformation of geometry, displacement of geometry, behavior of light sources, effects in the atmosphere, shading of a surface, and sampling of an image – often as separable shading components (Upstill 1989). Shaders and shading languages have also been broken into "building blocks" which are assembled through "shade trees." A visual editor to link various sub-components of a shader through a shading network was proposed such that execution of a shading network would be similar to the execution of an interpreted language (Abram & Whitted, 1990). Today, tools common in the film industry allow for artists to use graphical tools to create shaders executed in another shading language on the CPU.

These topics are gaining renewed interest in real-time graphics now that shading languages allow programmability directly to specialized hardware (the GPU). One advantage of breaking a shader into smaller pieces is that it allows more code re-use. Another advantage is that it abstracts the algorithm-development part of shader programming away from the syntax of any particular shading language so that artists can construct shaders using visual editors. Modern GPU shading languages are in need of the building block approach.

On the fly generation, manipulation, and specialization of shader programs for programmable graphics hardware is described in Shader Metaprogramming, where the Sh shading language is described (McCool, Qin, Popa, 2002). This source explains that C++ wrappers on assembly language shaders can tame some of the complexity of shader writing. In addition, the real-time program is the only place where the shader is actually generated, rather than files containing strings on disk. The Sh shading language allows for modularity of shaders that unfortunately is not found in Cg, HLSL, or GLSL.

Without modular shaders, system integrators and run-time providers face an explosion in shader permutations (discussed in the next section). Many game companies have treated shader application as a content management problem to be handled by the modeler. A few have built tools around the idea of generating shaders in the run-time. Some have even built über-shaders for a particular game engine that can perform any function, utilizing dynamic branching and allowing some code execution that does not contribute to the final rendered result. But this can degrade run-time performance.

The "Supershader" (McGuire, 2005) is an attempt to improve upon traditional über-shaders. A shader that does everything is hand-coded with many pre-processor conditionals (#if statements) that remove sections of the code at run-time when the shader is compiled for a particular surface or run-time condition. The supershader resolves some of the permutation problems, but is not very extensible and maintenance is more burdensome.

Current research on "Abstract Shade Trees" (McGuire, Stathis, Pfister, & Krishnamurthi, 2006) provides a framework for shade tree construction, which can result in vertex and fragment shaders written in modern GPU shading languages. In this approach, links among blocks of code are simplified through inferred connectivity of parameters and conversion of types to resolve type mismatching. This accomplishes the goal of modularizing the modern real-time shading languages and is a good starting point for this paper's research into constructing an extensible FFP from building blocks.

The shader infrastructure presented in this paper directly re-uses the "abstract shade tree" concept of breaking each shader into "atoms" and "weaving" a complete shader, so it is important to understand a few concepts. In an abstract shade tree, nodes of shader operations called "atoms" are connected in order of execution until reaching the "root" of the tree which is the output of the shader, typically a color value. By treating atoms as building blocks, many portions of various shaders can be mixed together by dragging and dropping atoms into the shader and connecting them to describe the flow of execution. The "weaver" is the program which translates the abstract shade tree into shader code in a particular shading language, such as Cg or GLSL.

The "Shader Infrastructure" described in this paper does not include a GUI for editing shade trees, but instead covers reconstruction of the fixed-function pipeline and automated merging of multiple shade trees. The final shader sent to the graphics pipe then, becomes a combination of the shading techniques created by an artist when modeling, and the shading techniques provided by the run-time. On the fly specialization of shader programs is accomplished by keeping the shader in its parametric, or "building block" form and weaving the shader itself during loading by the run-time. To facilitate this merging, an über-shader is effectively constructed from "atomic" building blocks.

CHALLENGES AND REQUIREMENTS FOR REAL-TIME SHADER MANAGEMENT

In the Motivation section, we presented only a few examples that demonstrate fundamental difficulties inherent in real-time visualization applications that use shaders. Further analysis of use-case scenarios netted a complete set of high-level requirements and challenges the Shader Infrastructure must solve. We discuss them in the following subsections.

Handling of OpenGL Rendering State Permutations

Fixed-function pipeline rendering as implemented by standard OpenGL offers a huge variety of ways to render content, even without shaders. Industry-standard formats, such as OpenFlight™ or Collada, are capable of storing all of the rendering nuances, while content creation tools allow modelers to take full advantage of the visualization options. On the receiving end, the run-time engine is responsible for loading, interpreting, and rendering the content via the OpenGL API. As mentioned in the previous section, the introduction of shaders forces developers to re-implement much of the FFP logic in one or more shaders. This requirement exposes us to the problem of programmatic handling of all state element permutations, previously done by the OpenGL driver and hardware. Let us take a closer look at state elements implemented in the FFP to fully appreciate the complexity of the problem. Consider the OpenGL state elements that represent vertex lighting (see Table 1).

Table 1: Permutations introduced by vertex lighting

Aspect of vertex lighting	Number of permutations
Lit or unlit	2
glColorMaterial for each component of lighting material (ambient, diffuse, ambient and diffuse, none)	4
Light source type (directional, point, spot, or none) for up to 8 hardware lights: 4*4*4*4*4*4*4*4	65536
Specular highlight On or OFF	2

Vertex lighting alone introduces 1,048,576 permutations. Omitted from this tally are permutations introduced by fog, texture coordinate transformations, texturing and texture blending, to mention just a few. Obviously, hand-coding a shader for every permutation appears infeasible. Additionally, managing a large number of shaders leads to other weighty questions:

- How will I store and load these shaders into my rendering system?
- How will my rendering system manage these shaders and apply them to my geometry?
- How will I setup the parameters my shaders need to run correctly?
- How do I protect my intellectual property in the shader code?

A brute-force, über-shader approach is also going to fall short simply because of limitations in the number of allowed instructions, which in turn are due to code complexity and potentially unacceptable run-time performance characteristics caused by dynamic branching. This leads us to consider an approach capable of dynamic, yet optimal, shader code generation at run-time, just to handle the FFP permutations alone if nothing else. Therefore, the question is: how do I automatically generate shaders for any piece of geometry?

At first, this question seems ridiculous. We've been taught to hand-code our shaders for highly narrow and specific purposes. We even tailor shaders to specific geometry with specific state settings. In a sense, we view vertex and fragment shaders as state attributes in the same way we view texture maps or materials, and for the most part, this view works well and is very intuitive. We write vertex and fragment shaders in text files and associate them with geometry just like we associate a texture file that contains texels. And when we render that geometry we simply bind those shaders just like we bind textures. However, as illustrated by the examples, treating shaders as just another state attribute becomes problematic if we're using shaders to implement a new lighting or fog model.

The naïve and most common approach to the problem of render-state inconsistency is hand-coding a collection of shaders that support only the minimum sub-set of features a particular rendering application requires. In other words, if we restrict what state settings can be applied by the run-time and force our modelers to obey a series of rules concerning lighting and texturing usage, we can manage a limited solution through hand-coding. In fact, this approach is widely popular in console and PC games.

But if a FFP-like shader could be auto generated based on OpenGL state, the need for hard-coded shaders is removed. Each element of OpenGL state implies a set of shader code. For example, light_0 configured as a directional light generates one set of shader code, while light_0 configured as a spot light generates a different set.

Additionally, the naïve hand-coding approach is too restrictive for use with modern graphics APIs that include a wide array of potential settings. This aspect of content management is particularly important if you are already invested in the content. A developer or system integrator may already have a library of models and, even more importantly, may not have the capability to upgrade the content. This brings us to the next challenge that needs to be addressed by the Shader Infrastructure.

The Need for Parametric Definitions of Techniques Applied to Models

We live in a world where a variety of content creation and management tools, both COTS and custom, are used simultaneously. Industry-standard data exchange formats, such as OpenFlight^{TM} or Collada, are used to survive in this heterogeneous world. However, each tool operates and renders differently. Additionally, modelers may be unaware of the run-time environment where the model is to be deployed. The run-time engine may or may not apply bump maps to a model, or cast cloud shadows onto it. Realizing that the rendering engine may be undefined during model creation leads one to the conclusion that it's impractical to use complete shader code as a means of providing the model's rendering definition.

Additionally, the world of advanced rendering effects is essentially open-ended. An introduction of a parametric definition of a rendering technique into the data format makes it easier to manage the assets associated with both render-state and shader techniques. Keeping the shader in its parametric form allows for each advanced technique's assets to be communicated appropriately with the model, but without requiring the complete shader attached to the model. For instance, a model may come with a bump technique attached. However, it would be up to the run-time engine to resolve the details, such as what shader code to use for the lighting computation, what texture stage to place the bump texture on, etc. This puts even more stress on the Shader Infrastructure, which would then need to be able to interpret parametric definitions of rendering techniques and generate shader code that makes this technique compatible with other pieces of the rendering puzzle. In certain cases, a rendering technique may even come with a snippet of code that represents a certain aspect of the simulation. For instance, consider a tree model that uses a vertex shader snippet to encode a tree's reaction to the wind. The Shader Infrastructure should be capable of incorporating the snippet seamlessly into the synthesized shader.

The shader auto-generation mechanism has other advantages. It could be extended not only to allow for new rendering techniques on a render-state for a model, but also for a global application of the technique to the whole scene. This is the subject of the next section.

Applying Global Rendering Techniques

Certain types of advanced rendering techniques such as light models, shadows, light maps, and cloud shadows require modifications to a shader for every geometry and state in the scene. Figure 5 demonstrates a cloud shadow technique, where the shadow is visible both on the terrain skin and the buildings.

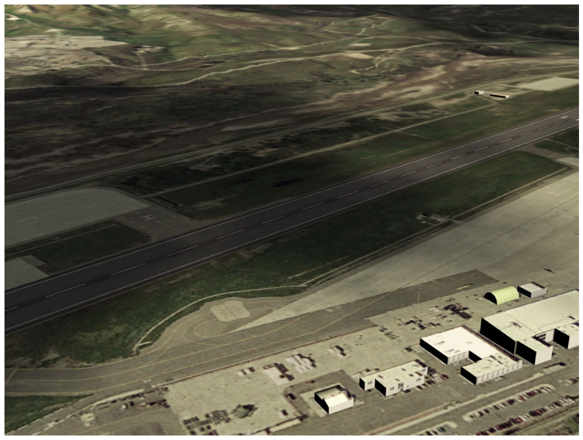

Figure 5. Global cloud shadow technique applied to all scene objects

The "apply to all" requirement introduces several fundamental difficulties for the Shader Infrastructure.

1. The number of permutations in the shader code expands even further (refer to the previous subsection for more details).
2. Global effects significantly complicate state management for the run-time. Consider cloud shadows as an example. A typical implementation requires that a cloud shadow texture is available for sampling in a fragment shader. However, the texture was not initially present in a geometry and, furthermore, the geometry maybe already using multi-texturing for implementing other effects. As a result, the Shader Infrastructure needs to find a way to incorporate an

additional texture into an already packed multi-texturing state.

3. A technique may also require additional vertex attributes to be generated. A projected texture based technique provides an example, where eye-linear texture coordinate generation needs to be enabled for it to work.

In principle, global techniques require a dynamic approach to shader code generation primarily because globally applied techniques are typically application specific. Cloud shadows are a perfect example in this context, where it is never considered during the database design or modeling phase due to its highly dynamic and application specific nature. Every run-time might have a different shader for applying cloud shadowing effects.

THE SHADER INFRASTRUCTURE

In order to tackle the problems of shader permutations, reproducing the fixed-function pipeline, and allowing for an extensible system, the *Shader Infrastructure* was born. The idea is not to manually write out shaders during content creation, which is tedious and error prone, but to dynamically create shaders at run-time. More precisely, we want to move away from guessing up front what rendering state our shaders will need to support, and toward analyzing the rendering state itself in order to dynamically build supporting shaders.

This ability to convert state into shaders is not only useful in handling the permutation problem, but is crucial for mixing older data, initially designed for the fixed-function pipeline, with newer shader based rendering techniques. If we ever hope to leverage these new rendering techniques, we will need a way to recreate the fixed-function pipeline which was the initial target during modeling. Only then can users apply new shader techniques to these old models.

As a technology, the Shader Infrastructure basically involves the sub-division of vertex and fragment shaders into a series of well defined "stages" in which indivisible snippets of shader code called "atoms" are stored. When a model needs shaders, the model's state is passed to the Shader Infrastructure and an algorithm uses the state to find atoms, which when combined, will construct vertex and fragment shaders suitable for rendering the model in question.

Defining Features as "Atoms" of Shader Code

In the Shader Infrastructure, a snippet of shader code is called an "atom." Atoms are the indivisible building blocks used to build dynamic shaders and they may comprise several lines of code, a single line of code, or even a single operation within a line of code. As an example, see Figure 6.

Atom "NdotL"

Snippet:

```
float NdotL =
 max(dot(normal,
lightdir),0);
```

Parameters:

Inputs:

```
float3 normal, lightdir;
```

Output:

Figure 6. A Shader Atom

The primary goal of the Shader Infrastructure is managing all of the atoms required to implement the features of the fixed-function pipeline and combine them dynamically to create valid vertex and fragment shaders. For example, if a model requires lighting and texturing, the Shader Infrastructure will mix together the atoms that implement lighting and texturing and auto generate vertex and fragment shaders suitable for that model. If the next model doesn't require lighting but uses multi-texture, the Shader Infrastructure will not bother with any lighting atoms but will use atoms concerned with blending two or more textures. In this way, any permutation possible by state settings can be implemented through dynamically created vertex and fragment shaders.

Defining the "Stages" of Shader Processing

Shader technology itself is actually broken into two units of computation: vertex shaders and fragment shaders. Within these two units, processing is possible in any order. However, it is useful to define further sub-units or "stages" of shader processing shown in Figure 7 below. The shader infrastructure supports nine stages of processing. The four Vertex Stages, "transformation," "per-vertex lighting," "per-vertex fog," and "texture coordinate generation," order and group the work done in a vertex shader, while the five

Fragment Stages, "texel lookup," "per-fragment lighting," "fragment combiners," "per-fragment fog," and "post processing," order and group the work done in a fragment shader. These stages are useful for defining exactly the type of processing the vertices and fragments will undergo and their processing order.

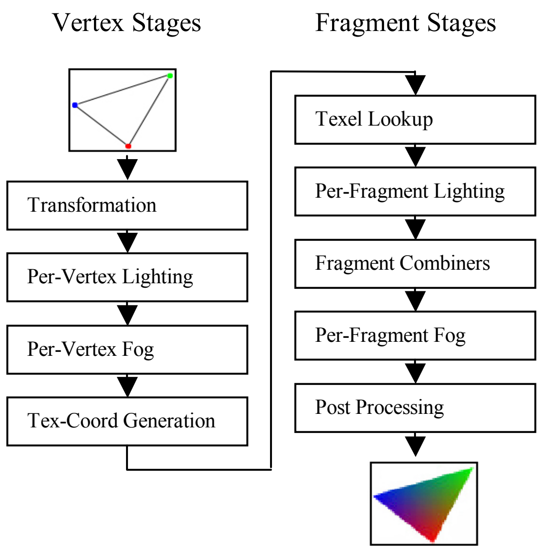

Figure 7. Vertex and Fragment Shader Stages

By registering atoms with a specific stage, the Shader Infrastructure has a way to define the type of vertex or fragment processing that is considered appropriate for each atom. More importantly, the registration of each atom to a stage amounts to the creation of an über-shader of atoms, or "Meta-Shader", which fully defines the feature set that the Shader Infrastructure supports.

Defining the "Meta-Shader"

Due to conflicts, such as order-of-execution or mutual-exclusion between the atoms of a particular stage, it's important not only to register an atom with the correct stage, but to define its relationship to at least one other atom within that stage. A "Meta-Shader," which is an über-shader of atoms, is used for this task. For example, the per-vertex fog stage will contain separate atoms for the calculation of linear, exponential, and exponential squared fog, but only one of those atoms should be active at a time. In this case, these three atoms are mutually-exclusive and it's important to record this relationship while building the meta-shader. As an example of atoms that are not mutually-exclusive but are order dependant, imagine two atoms within the fragment combiner stage that re-implement texture blending. One atom will be needed to handle the blending of the first texture stage while another is needed to handle the blending of the second texture stage to the result of the first. These two atoms are clearly not mutually-exclusive but if they are allowed to perform their calculations out of order the blending will be incorrect.

Once each atom has been registered to a stage and its relationship to the other atoms defined via the meta-shader, it becomes possible to analyze the rendering state of a model and to literally automate the weaving of atoms into vertex and fragment shaders. The atoms define functionality, the stages define global structure and flow, and the meta-shader resolves conflicts concerning the two most important relational aspects of atoms which are order and mutual-exclusion.

CREATING NEW "TECHNIQUES"

"Techniques" are simply collections of one or more atoms that work together to perform an identifiable task. Creation of new atoms and techniques allows for extension beyond fixed-function pipeline rendering. "Advanced Rendering Techniques" (ART) are techniques which provide capabilities beyond the FFP. Techniques are new attributes for render-state that can be provided during modeling or run-time. Therefore, techniques are the method for extension of render-state.

Commonly, ART is provided by the run-time rather than a modeler. Cloud shadows, HDR lighting, physics based atmospheric effects, and many other techniques require that shader snippets be inserted into the shaders for every model in the scene.

In addition, modelers should no longer be limited to the feature set of the FFP. Bump mapping, emissive light maps, and environment reflections are all ART that should be specified by an artist when creating a model. Common techniques' atoms should have their snippets of shader code specified by the run-time's Shader Infrastructure. Having the run-time provide atoms for ART eliminates the need for the artist to understand the technical details behind a technique. The Shader Infrastructure presented in this paper is currently a part of the commercially available Vega Prime run-time toolkit, which contains atoms for all of the above techniques. While the shader code itself is fairly universal and can be implemented in any run-time, it should be noted that many techniques require additional data, such as a texture, texture coordinates, or shader parameters as well. The modeling tools and formats need to grow to encompass this new data.

The specification of the FFP render state on a polygonal face of a model allows the Shader Infrastructure to select some atoms. The model and/or the run-time may specify ART on the render state as well. When all of the atoms are collected and ordered through the meta-shader, a vertex and fragment shader can be automatically generated. Thus, the Shader Infrastructure can automatically generate a shader for the FFP render state plus additional "techniques" provided by either the modeler or the run-time.

FUTURE WORK AND CONCLUSIONS

The Shader Infrastructure allows designers to customize portions of the rendering pipeline and to automatically combine that customization with other rendering techniques, covering both modeling and run-time techniques. Reproduction of the fixed-function pipeline can be seamlessly merged with advanced rendering techniques.

For simulation and training, leveraging the modeling assets that have already been created is very important. Being able to load those models in new run-time engines that continue to take advantage of programmable graphics hardware is also important.

Being able to extend those models through advanced modeling Techniques will require new standards for the additional data and methods. OpenFlight™ is one mechanism through which modeling assets are currently communicated. New data fields for the most common Techniques are currently being added to OpenFlight™.

Run-time supplied techniques will also continue to expand. To be extensible, the full list of atoms and techniques in the Shader Infrastructure would have to be exposed to the user. This would allow the user to define atoms and techniques that could be merged with existing modeling and run-time techniques.

The Shader Infrastructure significantly simplifies the problem of content management and reuse while taking full advantage of the advances in programmable PC graphics hardware. As we move forward with nearly infinite possibilities for rendering in hardware, utilization of a shader infrastructure to combine multiple techniques into executable shaders will become increasingly important.

REFERENCES

Abram, G. D., & Whitted T. (1990). Building Block Shaders. *SIGGRPAH '90 Conference Proceedings, Computer Graphics, Vol 24:4*, pp 283-288.

Fernando, R., & Kilgard, J. (2003). *The Cg Tutorial*, New York: Addison-Wesley.

McCool, M., Qin, Z., & Popa, T. (2002). Shader MetaProgramming (Revised). *Eurographics Graphics Hardware Workshop, September 2-3, 2002, Saarbruecken, Germany*, pp. 57-68. Retrieved June 13, 2007 from http://www.cgl.uwaterloo.ca/Projects/rendering/Papers/metaAPIpaper.pdf

McGuire, M., Stathis, G., Pfister, H., & Krishnamurthi, S. (2006). Abstract Shade Trees. *Symposium on Interactive 3D Graphics and Games*. Retrieved June 13, 2007 from http://www.cs.brown.edu/research/graphics/games/AbstractShadeTrees/index.html

McGuire, M., (2005). The SuperShader. *ShaderX^4*. Hingham, Massachusetts: Charles River Media, Inc. pp. 485-498.

MultiGen-Paradigm. (2006). *Vega Prime Marine Datasheet*. Retrieved June 13, 2007 from http://www.multigen.com/support/dc_files/VP_Marine.pdf

Stamminger, M., & Drettakis, G. (2002). Perspective Shadow Maps. *Proceedings of ACM SIGGRAPH 2002*.

Upstill, Steve. (1989). *The RenderMan Companion*. New York: Addison-Wesley.