Buffy, an SL Development Environment

Ian Stephenson

National Centre for Computer Animation
Bournemouth University
istephen@bournemouth.ac.uk

Abstract

A development environment for RenderMan shaders has been implemented, allowing the SL programmer to observe the bahaviour of variables in both time and space. By allowing both novices and experts to visualise the execution of their code, the creation of shaders is much simplified without restricting the flexability of the SL language.

1 Introduction

The RenderMan standard incorporates programmable shaders[Hanrahan and Lawson 1990]. These are written in "SL" or Shading Language, which attempts to simplify the development of custom shaders by providing appropriate hooks into the renderer, and a rich environment of operators and functions[Upstill 1989]. Despite this, the widespread respect for the standard, and the recent increase in the number of RenderMan compliant renderers, the development of shaders is still seen as a challenging task, beyond the abilities of most animators.

A number of tools, such as Cinema Graphics' ShadeTree, Houdini "TOPS", and most recently Pixar's SLIM editor attempt to make the development of procedural textures more accessable by allowing the user to describe a shader through a graphical UI. These tools generally operate by building up a network of operations, in a form of visual programming. However it appears that this approach is limited - it fails to provide the flexibility required by professional shader writers, yet novice users still lack the understanding required to use the tool effectively. Worse still, in attempting to protect the user from the harsh details of programming, little opportunity is provided to acquire the skills required for more advanced work.

In addition to the standard problems of programming, shader writers need to visualise the execution of their code over a surface. Shaders take the form of a function, which is evaluated at every point of a patch. This massively parallel approach makes traditional debugging techniques such as diagnostic printouts impractical - one printf statement in the shader applied to a single patch could produce in excess of 10,000 lines of text.

It is proposed that the aim of an SL development tool is that it should assist the SL programmer to produce shaders, supporting the process rather than attempting to hide it. The user should be able to see how variables change both over the shaded surface, and over time, as their code is single stepped. In this way, the programmer may be both more efficient, and develop his skills from novice through to expert.

This paper discusses the development, and application of such a tool, known as Buffy. Buffy is built around an SL runtime environment refered to as SLander. A brief review of RenderMan shading is provided, before the SLander engine is discussed, followed by the extensions required to implement Buffy. Finally the user experience is illustrated by considering the application of Buffy to a typical SL code fragment.

```
Ci=(specularcolor
        *specular(Nf,normalize(I),roughness)
        *Ks
    +(diffuse(Nf)*Kd+ambient()*Ka)*Cs)*Os;
```

Figure 1: Plastic shading Model in SL

2 RenderMan Shaders

Within a RenderMan renderer, all properties of a surface are specified by the application of shaders[Apodoca and Gritz 1999]. These take the form of short programs written in a C like language, known as SL. A number shader types exist, each of which performs one step of the imaging process. Though not all renderers need support all shader types, the available types typically include displacement, surface, atmosphere, and imager. Only one shader of each type may be applied to a particular surface.

Displacement is first applied to a surface, allowing a surface to be distorted. As with all shaders, by applying this modification at render time, the level of detail may be adapted dynamically to ensure that the rendered surface is free from artifacts. Having estabilshed the true location of each surface point, the Surface shader is executed. This typically forms the core of the texturing process, as its role is to calculate the colour of each surface element, for a given observer, and lighting contitions. In order to assist this task, a number of variables are passed into the code, either as parameter from the modeller, or calculated by the renderer. Functions allow the lighting conditions to be interogated, as shown in Figure 1, the lights themselves being controlled by Light shaders.

All forms of texturing take place within this shading pipeline, texture maps being support indirectly by the provision of functions within the shading language. This allows the placement of the texture map to be controlled procedurally, and the value so obtained may be used for any purpose, rather than simply being applied drectly as the surface colour.

Once the surface colour has been established, an atmosphere shader controls the propogation of the surface colour to the camera (implementing fogging, or other depth cues). An imager shader allows the colour so far calculated to be modified, for arbitary colour correction prior to recording the image.

This procedural approach to shading affords maximum flexability, as any part of the shading pipeline may be modified. As the code may be evaluated at any point, a well written shader can provide a non-repeating texture over any size area, at an arbitary level of detail. A shader should also control its high frequency components, removing detail as appropriate, to avoid aliasing without recourse to excessive supersampling. Unfortunatly shaders of this quality are difficult to build. Improved development tools would therefore prove invaluable.

3 SLander

The core of the Buffy application is the SLander runtime engine. Shaders are typically compiled from SL to a virtual machine(VM). This makes compiled shaders portable across hardware platforms, say in a renderfarm which may be composed of whatever machines are available (though the compiled shaders are not typically compatable between renderers, each of which has its own VM). The use of a virtual machine also simplifies the generation of code, as the VM will generally be designed to facilitate the running of shaders with none of the quirks typically found in real hardware. Though VM machines such as the Java VM, INTCODE[Richards and Whitby-Stevens 1979], or the P-machine have a (justified) reputation for being slow this is not the case when implementing SL for reasons which shall now be discussed.

The aim of the shading engine is to shade not simply a set of points, but a surface. That is a patch is passed to the engine for shading, and all of its component points are typically shaded at the same time rather than shading one point fully before progressing to the next (this is an important concept which Buffy is able to make clear to users). The overhead in interpreting a noise function call (for example), which will be a single op-code within the compiled shader will therefore be incurred once per patch. The noise function will then be evaluated at every point on the surface, making the cost of interpreting the machine code insignificant.

This approach (known as SIMD[Flynn 1996] - single instruction, multiple data), not only makes the use of a VM efficient, it also enables certain operations to be implemented which would otherwise be very difficult. These "area" operations rely on the values of expressions being known over the surface rather than just at a single point. The most obvious of these is "calculatenormal" which handles bumping of the surface, and requires the deformation of the local surface to be known. However even reading a texture map requires area information if anti-aliasing is to be done correctly. Further benefits of the SIMD approach are that uniform expressions (whose value is the same accross all points of a surface) can easily be optimised. Finally the style of programming scales well to high-end parallel hardware[Hockney et al. 1981; MasPar Computer Corporation 1991]. An SIMD VM is therefore used by both SLander, and PRMan (RenderDotC uses native code incorporating SIMD techniques).

Rather than design and implement a full VM and supporting environment from scratch, the VM design was borrowed from Blue Moon Rendering Tools[Gritz 1999] - an excellent shareware RenderMan compliant renderer. The BMRT VM is a simple stack based system, which stores its code in an ASCII format (Figure 2). This allowed the development of SLander to progress rapidly using pre-compiled shaders as test cases, the implementation of a shader compiler being postponed until much later in the project. Compatibility with BMRT is an added benefit, though not a design criteria - as such it may not be preserved in future releases of either BMRT or SLander.

It has been said that though RenderMan claims to be renderer independant, certain aspects of the SL standard makes it very difficult to implement within a ray tracing renderer. In calculating secondary rays, a ray tracer must deal with the shading of points rather than complete surfaces. This precludes the use of the SIMD approach. BMRT being a ray tracing/radiosity renderer must go to great lengths to correctly deal with area operators[Gritz and Hahn 1996]. As a result the BMRT VM is not designed for SIMD execution. It is however possible to execute BMRT object code relatively efficiently in an SIMD fashion, the main loss being the lack of support for uniform variables which are treated in an identical fashion to varying operations.

SLander has been implemented as a library written in C. The client program passes in a UV patch, with the SL variables defined

```
pushf roughness
pushv  I
normalize
negv
pushv  Nf
specular
mulCF  specularcolor Ks
mulcc
pushv Nf
diffuse
mulFc  Kd
ambient
mulFc  Ka
addcc
mulC  Cs
addcc
mulC  Os
popc  Ci
```

Figure 2: Plastic shading in BMRT object code

across the surface. The engine runs a shader program using the predefined global variables, and a set of lights as input. Multiple shaders (displacement follwed by surface for example), may be run by simply executing one shader after the other without reseting the VM.

Each instruction in turn is read from the compiled shader file, and applied to an array of nodes, each representing one point of the surface. Upon encountering a forward branch instruction processor nodes which are not required to execute the following code block are temporarily suspended, being returned to the active set when the destination of the branch is reached. The instructions themselves are always parsed, even though no procesors may be active. A backards branch rewinds the shader file only if the active set contains at least one node. Special care must be taken to correctly handle the stack pointer which is shared by all processors.

Upon completion of a shader the client program can extract values of Ci: the output colour for each point on the patch. In a full renderer this would then be mapped onto a surface. A standalone version of SLander has been developed which operates very well as a shader previewer, displaying the results of shading a simple linear patch, or sphere. As no geometry is used this program is far faster than a normal renderer, and even in this limited form, can dramatically reduce shader development times.

4 Buffy

Once the SLander engine was operational, it was relatively trivial to add a UI. This consists of a window for each shader document containing a text editor allowing the SL code to be developed with buttons to trigger the compilation, and execution of the shader. Selecting the compile option saves the file, runs an SL compiler to generate object code, and initializes the SLander engine to execute the newly compiled shader. Selecting "run" allows the current VM to run to completion. "reset" discards the current execution, and resets the VM back to its initial state, ready to run the re-run the shader.

The results of a run can be viewed by creating an inspector (using the inspect button). Normally a shader's task is to calculate Ci, the surface colour, and this may be displayed in the inspector, as if the shader had been applied to a simple planer surface. However, any other variable used in the shader may equally be extracted and displayed (Figure 3). Scalar values are displayed as grayscale, while vector values are displayed as RGB. Further, by clicking

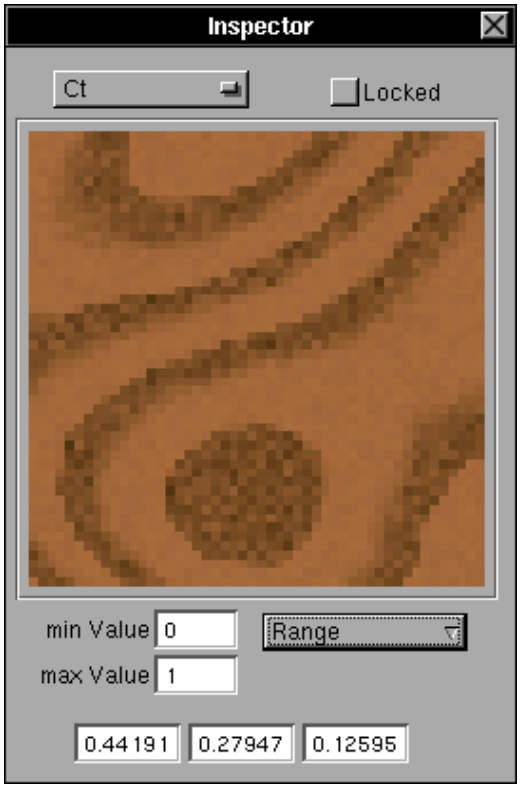

Figure 3: The Buffy Inspector Panel

on the surface, the numerical values at a particular point may be obtained. Multiple inspectors allow any number of variables to be viewed simultaneously.

In order to facilitate single stepping of the code, hooks were added to the SLander engine, such that each call to the library would execute only a single VM instruction. Following each call, the front end is able to extract and display the updated state of any variables being inspected. However single stepping at the object level is of limited use, as relating the state of the machine back to the high-level source code is difficult. Unfortunately the BMRT object code provides no way of identifying which line of source code the current instruction relates to.

In order to facilitate source level single stepping, the SLander VM was extended beyond the BMRT VM to include one new operation. This construct allowed lines of the form:

```
# 13 "fog.slc"
```

to be inserted into the object code, allowing SLander to record progress through the source code. Buffy could then inspect the state of SLander to identify the current line, highlight it, and allow single stepping by running the machine until the line number changes.

Though the debugging environment could now support single stepping, it is reliant upon the compiler to generate the information it needs to synchronise execution. The standard BMRT compiler "SLC", of course generates no such information. A simple debugging compiler "Giles" was therefore developed. Though the code produced by Giles is less efficient than that of SLC, and Giles currently has a number of limitations with respect to some of the more obscure aspects of the SL syntax, it is able to handle the vast majority of shaders, and when combined with Buffy allows single stepping at the source code level, as originally desired. Buffy will still function correctly when using SLC as its compiler, but single stepping is not possible.

```
for(i=0;i<6;i++)
{
mag += abs(snoise(P*freq))/freq;
freq *= 2;
}
```

Figure 4: Turbulence Code in SL

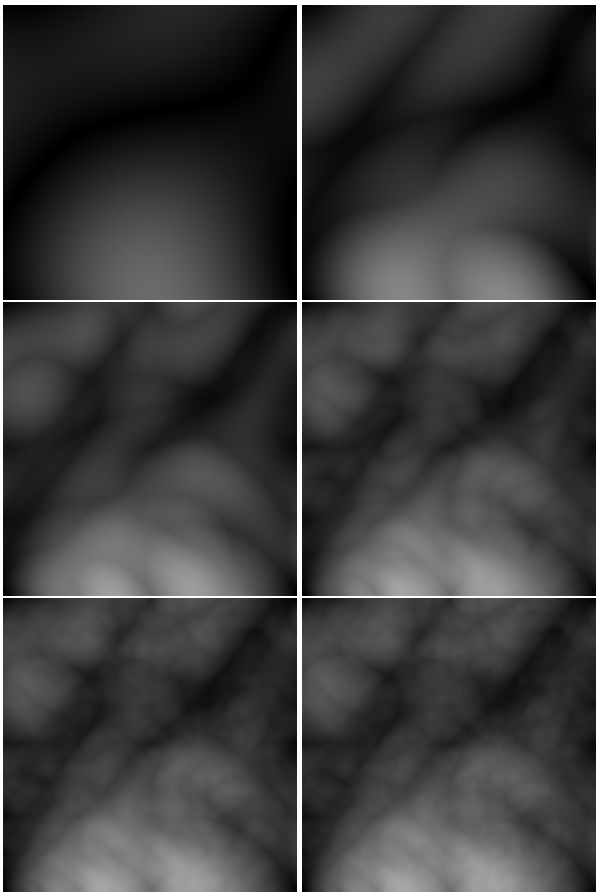

Figure 5: A turbulence function

5 Application

The insight into the development process afforded by Buffy is perhaps best illustrated by considering a simple turbulence construct[Ebert et al. 1998]. This takes the form of noise summed over a range of frequencies, as in figure 4.

"mag" may then be used, to modify the colour, displacement, or any other feature of the surface. Though this construct is familiar to any experienced shader writer, and forms the basis of a great many shaders, the technique can be difficult to grasp. Learning how to adapt and apply turbulence, and the related fBM functions is a skill which the SL programmer must acquire.

An understanding of the turbulence code can rapidly be obtained by single stepping the code in Buffy. The value of "mag" on successive iterations of the loop can be seen in figure 5. On the first

iteration we see that mag is simply 2D noise. This view alone makes clear the difference between noise as used in the context of shading, and simple random values. It can be seen that noise has a frequency, and that it varies continuously over the surface.

On further iterations it is possible to se how the texture is built up in layers, each adding finer detail of smaller amplitude and higher frequency. Once viewed in Buffy, this result appears trivial, but without a suitable visualisation tool is very difficult for the inexperienced programmer to associate the code with the final image produced. It is precisely this link from code to image which the shader writer must learn to make. While this ability can only come from experience, the oppotunity to examine the results of the code as the texture is constructed is an invaluable aid.

For the more experienced user, the ability to inspect the shader can also prove valuable. Shaders often contain "magic numbers" in order to bring the results of functions into a suitable range (for example the value of "mag" following the above calculation would typically be scaled to suit its final use). The ability to view the actual values produced by the calculation can remove much of the guesswork required to arrive at these values.

6 Further Work

In comparison with the debugging tools available for traditional programming languages, Buffy is still relatively primitive. Breakpoints, and watching of variables are standard features which could be incorporated into Buffy. The nature of shader writing, where code is usually short in length reduces the need for these options, but they would still be of benefit.

Buffy currently shades a flat plane. While this is the most useful shape to work with when developing a simple surface shader, it is also helpful to work with a range of shapes. Extending Buffy to support a choice of more complex geometry would be a powerful addition. Work is underway to allow Buffy to operate on a complete RenderMan scene file (RIB). It is hoped that the end user will be able to select an object within the rendered scene, and inspect the execution of a shader within the context of the intended scene.

It may also be useful to provide tools for performing statistical analysis on a variable to identify how it changes over space. Simple metrics such as maximum, minimum, mean, and standard deviation would be of great value, and information about the spectral properties of a variable could prove to aid in the challenging task of anti-aliasing.

The SLander engine has great scope for future application. It has already been used to process and generate 2D images for use in compositing applications, where SL can be a powerful scripting language. Integration into one or more compositing tools would allow SL programmers to transfer their skills from 3D tasks to 2D tasks. A full renderer making use of the SLander engine is under development. It is also believed that SLander itself may be used as a procedural shader within a non-RenderMan renderer, allowing SL textures to be used, for example, in Mental Ray.

7 Conclusion

A tool has been developed which supports shader writers in learning and practising their craft. The tool, known as Buffy makes use of the SLander shading engine, and Giles SL compiler to allow the user to view variables as they vary over both time and space. The facilities have been found to be helpful to both novice, and more experienced programmers, making clear many of the concepts of shader writing which may be difficult to grasp.

References

APODOCA, A., AND GRITZ, L. 1999. *Advanced RenderMan:Creating CGI for Motion Pictures.* Morgan Kaufmann Publishers.

EBERT, ET AL. 1998. *Texturing and Modeling, a Procedural Approach.* AP Profesional.

FLYNN, M. 1996. Some computer organizations and thei r effectiveness. *IEEE Trans. Computing C*, 21, 948–60.

GRITZ, L., AND HAHN, J. 1996. Bmrt: A global illumination implementation of the renderman standard. *Journal of Graphics Tools 1*, 3, 29–47.

GRITZ, L. 1999. *Blue Moon Rendering Tools*. www.bmrt.org.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Proceedings of SIGGRAPH 1990 24(4)*, ACM Press, ACM SIGGRAPH, 289–298.

HOCKNEY, R., ET AL. 1981. *Parallel Computers.* Adam Hilger Ltd.

MASPAR COMPUTER CORPORATION. 1991. *Data Parallel Programming Guide.* MasPar Computer Corporation.

RICHARDS, AND WHITBY-STEVENS. 1979. *BCPL - the language and its compiler*. Cambridge University Press.

UPSTILL, S. 1989. *The RenderMan Companion.* Addison Wesley.