Persistent Background Effects for Realtime Applications

Greg Lane, April 2009

0. Abstract

This paper outlines the general use of the GPU interface defined in shader languages, for the design of large-scale effects with per-pixel clarity. At this point, the stability of GPU programming has matured, and effects can be used reliably on most consumer hardware. Coupled with the low efficiency reduction and short code segments, the incorporation of shader effects seems to have a high benefit/cost ratio.

Algorithms of persistent animated effects are also detailed. These outline the general process used in the development of pixel shaders, and describe concepts such as render to texture, the framebuffer, and distortion effects.

1. Introduction

Programmable GPUs were introduced commercially six years ago, in high-end graphics cards. Initially, there were many standards and upgrades that offered important innovations, such as stencil and depth buffers, but reduced the general portability and reliability of shader code.

Recently, the incorporation of programmable GPUs into modern gaming consoles has established standard guaranteed results, and opened up the possibilities for realtime pixel level manipulations in commercial software. These effects are visually stunning, and generally run at relatively low overhead on the CPU.

In this paper, I explore the early use of the programmable GPU in the computer market, and examine some fundamental concepts for taking advantage of the hardware interface. I will discuss the use of textures as general data structures, which offer persistent information over multiple frames, and the use of the rendering pipeline as a general pixel processor. An analysis will cover the benefits and disadvantages of the current widely supported shader languages, and future extensions to GPU programming, such as CUDA.

I will conclude with a few algorithms implementing the previously explained methods of persistent animation. The first is a rippling effect that follows a position on screen and stacks the displacement of previous frames. The second is a radial light that changes hue, intensity, and saturation by a moving point on screen, and smoothly interpolates its animation.

2. General Shader Overview

Shaders are compiled programs executed on GPU, rather than the CPU. They are fairly low-level instruction sets that mimic assembly language and C. As their original intention was to offer visual computation, much of the syntax is oriented around geometry, textures, and colors.

Because the GPU is typically composed of many simple processors running in tandem, shaders are limited to a relatively small number of instructions, and generally perform transformations on one discrete unit of the scene.

Modern graphics hardware offers two distinct types of shaders, which are handled at different stages of the rendering pipeline. Vertex (or Geometry) shaders are designed to transform individual geometry in a scene, and are handled early in the rendering pipeline. Pixel (or Fragment) shaders are designed to transform the buffer information of individual screen coordinates, and are handled late in the pipeline, when the scene has been rasterized from geometry to pixel color values.

Pixel shaders operate on the individual pixels of a render pass, and are not given access to their neighbors due to the massively parallel processing. However, they are provided with common texture information, which typically represents a set of pixel color values. By using the framebuffer contents for the last frame as texture data for the current frame, simple persistent effects can be achieved.

For effects requiring more information than the color values of the previous frame, textures can be used beyond their general application. Since each pixel, or execution of the pixel shader program, is given 24 or 32 bits of information storage per screen-sized texture, this space can be used to represent arbitrary data accumulated over previous frames. This allows persistent distortion and other effects beyond the realm of basic color interpolating.

The use of textures as indeterminate data structures allows pixel shaders to perform general computation at each pixel, based on certain common parameters. This is the fundamental method for designing persistent full-resolution effects in a visual application.

3. The General Process

When rendering pixel effects on the GPU, the traditional rendering pipeline does not offer much towards their processing. Typically, the effect will be handled entirely in the pixel shader, and the other steps of the pipeline are sent simple tasks, purely to access the power of the pixel shaders.

For a simple fragment-independent effect, a pixel shader is a single additional step in the process. Shaders occur immediately after rasterization and texturing, which means that the current framebuffer contains the rendered display. Using these color values, simple effects like desaturation can be applied to the scene.

Figure 1: A simple desaturation pixel shader effect, applied to a quad in the center of the scene.

For a slightly more complicated distortion effect, two additional steps are added. First, the framebuffer of the rendered scene is copied to a texture. Then, a single screen-sized quad is passed to the rendering pipeline with a pixel shader effect attached, so that each pixel runs the shader program. This is not ideal, as time is spent on the fake geometry in lighting, transformations to screen space, and rasterization, for no reason but to set up the pixel shading stage properly. However, this is the only interface supplied to pixel shaders.

This is the simplest form of multipass rendering, where the rendering pipeline is run more than once each frame. For techniques like distortion, where the entire framebuffer must be accessible from each pixel, it is necessary overhead.

For even more complicated effects, an additional pass can be added at the beginning of the process to render data, rather than a color buffer. The data can then be copied to a texture and used to generate the framebuffer, or to define distortion and other effect parameters on a per-pixel basis.

4. Future Extensions

The current use of the GPU to handle general programming is restricted by the overall design towards graphics processing. There are no general data structures, and most persistent information must be broken up to fit in a set of textures. Additionally, each pass of rendering can only produce a framebuffer-sized amount of new information, as this is the only write-access structure.

While these limitations and the overall obfuscation of the interface reduce the extent to which the GPU can be easily programmed, recently general-purpose GPU programming languages have become available. CUDA, which is supported by newer high-end Nvidia cards, is a large improvement on non-graphics oriented processing.

CUDA provides an interface to main memory that does not involve textures. This allows each thread to allocate an arbitrary amount of memory, not bound to the sizes of any graphics structures. It also resolves the issue of storing new information, by allowing threads to write back to all of main memory.

CUDA still enforces certain limitations in its design, and introduces it own new limitations. First, threads are still prevented from running recursive calls. Second, the more general nature of CUDA programming reduces the optimization procedures of the graphics hardware, and introduces slowdown in more graphics-oriented functionality. Third, performance penalties result if threads are not handled in a convoluted way, where programs should be split into thousands of processes, and blocks of these threads must branch in a similar way. These limitations seem to arise from the general structure of graphics hardware devices, which are designed for a very specific operational flow.

5. Sample Algorithms

5.1 Persistent Ripple Distortion

A rippling effect is a distortion effect, which modifies the contents of the screen by selecting pixels in the framebuffer texture nearby the current position, through a mathematical algorithm. A simple example would be to calculate the distance from

the current point to the center point of the ripple effect, add the current time, and take the cosine. Code Segment 1 describes this method.

(1)
First Pass:
//no shader, just render the scene normally
copyframebuffer(tex);

Second Pass:
length = sqrt((pos_X-$cpos_X$)^2 + (pos_Y-$cpos_Y$)^2);
dir = (pos-cpos)/length;
return tex[pos + dir*cos((pos-cpos) + time)];

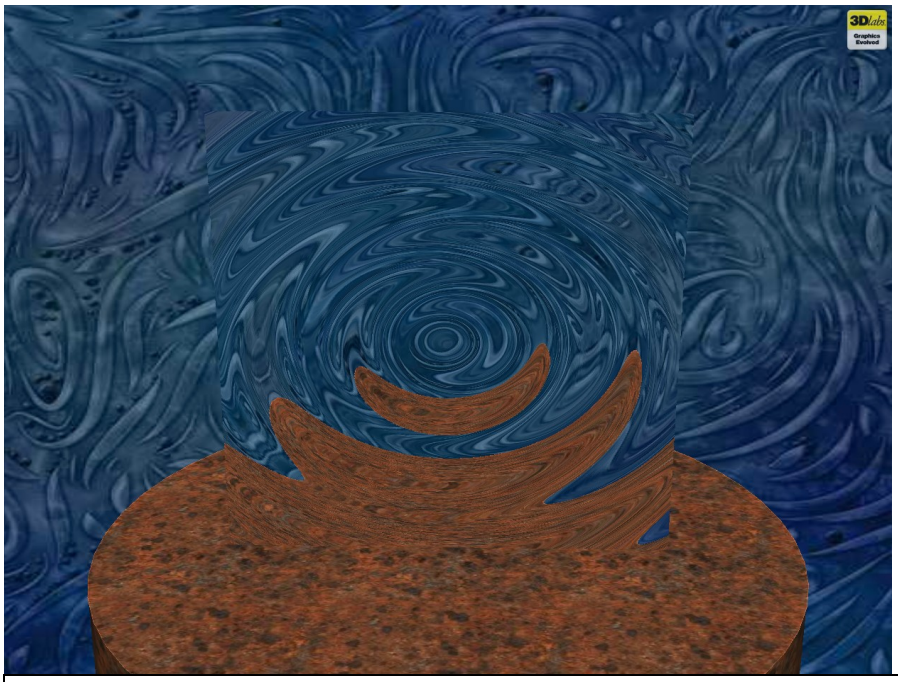

Figure 2: A simple ripple effect, which does not take the distortion of previous frames into account. It is applied only to the quad in the center of the scene, and would not have a defined edge if the quad was the size of the full scene.

Where the returned value is the color at position **pos** in the final scene, **cpos** is the center position of the ripple, **dir** is the normalized vector direction from cpos to pos, and **tex** is the framebuffer texture created in the first pass. By adding or subtracting time, the effect can ripple outward or inward.

However, this simple effect does not have a concept of earlier frames, and only bases the effect on the current center position. The perceived persistent effect is achieved by incorporating time in a periodic function, which changes the distortion at each pixel only slightly each frame.

To extend this algorithm, it is necessary to maintain information at each pixel about the previous frame. To prevent the scene from becoming distorted beyond recognition, this data must degrade as time passes, and eventually return to its default value.

Two additions are made to the previous algorithm, which add an additional pass in rendering. The first is to produce a screen size texture which contains offset information, indicating how much the pixel was distorted in the past. By encoding this as two floats in x and y, a 32 bit texture buffer can contain the information.

The second addition is to set a radius on the rippling effect, via a linear interpolation function. By maintaining a maximum radius property, the influence of the cosine distortion will decrease as the position is further away from the center point of the effect. Code Segment 2 shows the updated algorithm, reflecting both these changes.

(2)
First Pass:

```
length = sqrt((posX-cposX)^2 + (posY-cposY)^2);
amt = max(0,radius-length)/radius;
dir = (pos-cpos)/length;
distortion = amt*dir*cos((pos- cpos) + time);
return distortion + max((0,0),datatex[pos] – deltaTime);
//this creates a buffer of data, with 2x 16 bit floats for (x,y) distortion
copyframebuffer(datatex);
```

Second Pass:

```
//no shader, just render scene normally
copyframebuffer(tex);
```

Third Pass:

```
newpos = datatex[pos];
return tex[pos + newpos];
```

5.2 An Animated Radial Light

This is a pixel independent effect, where all animation is determined completely by common parameters and the previous framebuffer value at the same location. The effect is defined in HSL color space, and then converted to RGB color space for display. This technique requires only one pass, as it moves the previous framebuffer contents into a texture before rendering, and applies the shader effect on the scene directly.

The basic idea is to generate a light from a point on the screen, where the color and saturation are respectively based on the angle from that point to the current pixel, and the speed at which the screen point moved in the last frame. Therefore, parameters are **cpos**, the position on the screen, **opos**, the old position on the screen last frame, and **deltaTime**, the change in time since the last frame.

(3)

```
copyframebuffer(tex);
First Pass:
    oldhsl = rgb2hsl(tex[pos]);
    angle = abs(atan2(pos))/(2*pi); //0.0-1.0 range for angle
    speed = sqrt((cposX-oposX)^2+(cposY-oposY)^2);
    radius = max(0.0,min(n,speed/constprop);
    hsl = hsl(angle,speed,0.5); //half brightness gives fully saturated colors
    hsl.hs = max(0.0,min(1.0,(hsl.hs + oldhsl.hs)/2));
```

hsl.l = max(0.0,min(1.0,(hsl.l + oldhsl.l – deltaTime)/2));

Code Snippet 3 outlines this effect, where hsl components are described with access similar to rgb colors, changed to hsl for clarity. This effect is less intensive to compute, and still produces interesting visual results.

6. Analysis and Results

My initial effort in the implementation of the algorithms was unproductive, due to the use of the 3dLabs GLSL demo. When my research began, I was not fully aware of the differences between framebuffer access and full render to texture functionality. As the GLSL demo supports the former and not the latter, only simple distortion is supported, and persistent effects cannot be maintained. Coupled with the recent collapse of 3dLabs, which eliminated all documentation of the program, the development of the described sample effects proved impossible.

I am currently working on another version of the shaders, written in HLSL and XNA. This development environment completely supports render to texture functionality, and therefore allows the use of textures as general data structures. With this functionality, implementing the aforementioned algorithms should be trivial. Hopefully, the screenshots and analysis will be added soon, in an addendum to this paper.

7. Conclusions

The GPU is capable of handling advanced visual effects in realtime applications, despite strict limitations in current hardware interfaces. The shader languages have matured into stable, reliable processes, which produce expected results on a variety of hardware products. Despite the interface complexity, the GPU offers efficient solutions to problems that are not easily handled in a serial design.

Unfortunately, it seems future improvements to the programmable GPU interface will always remain convoluted, because the graphics hardware is primarily designed to improve graphics processing tasks. The internal structure of the hardware is wired to offer communication between graphical components, and the parallel processors are simply a secondary method of improving the performance of these tasks.

However, because graphical processing is a very complicated problem, a significant amount of power has been put into these devices. In addition, the general requirement for graphics in computers has made this hardware cheap and commonplace. Because of this, GPU programming will continue to improve overall performance of tasks, as it is widely supported and capable of quickly resolving complex problems.

8. Acknowledgements

I would like to thank 3dLabs, for the GLSL demo application provided freely online. It offered an interface for developing shader effects, and allowed me to start working on “the important stuff”.

I would also like to thank Tim Rowley, for his index on graphics papers. It provided invaluable resources on shader effects, and explained how the interface was used to develop certain effects.

Lastly, I would like to thank Barb Cutler, for teaching Advanced Computer Graphics. Most of the gray areas of my self-taught knowledge were cleared up in this class, and I only wish I had had more time to dedicate towards the concepts and assignments.

9. Bibliography

Purcell, Timothy J. et al. "Ray Tracing on Programmable Graphics Hardware". ACM Transactions on Graphics. 21.3 (2002): 703-712.

Rong, Guodong and Tan, Tiow-Seng. "Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform". i3D 2006. 14 Mar. 2006.

Wei, Li-Yi. "Tile-Based Texture Mapping on Graphics Hardware" SIGGRAPH2004. 2004.

Persistent Background Effects for Realtime Applications – Addendum

Greg Lane, April 2009

6. Analysis and Results

After porting the shader code from GLSL and the 3D Labs demo to HLSL and XNA, the results were very positive. However, the original plan for the radial light exceeded the maximum math instruction count on Shader Model 2 (64), reducing general portability. I was able to reduce the process by blending previous frames purely in RGB color space, rather than HSL.

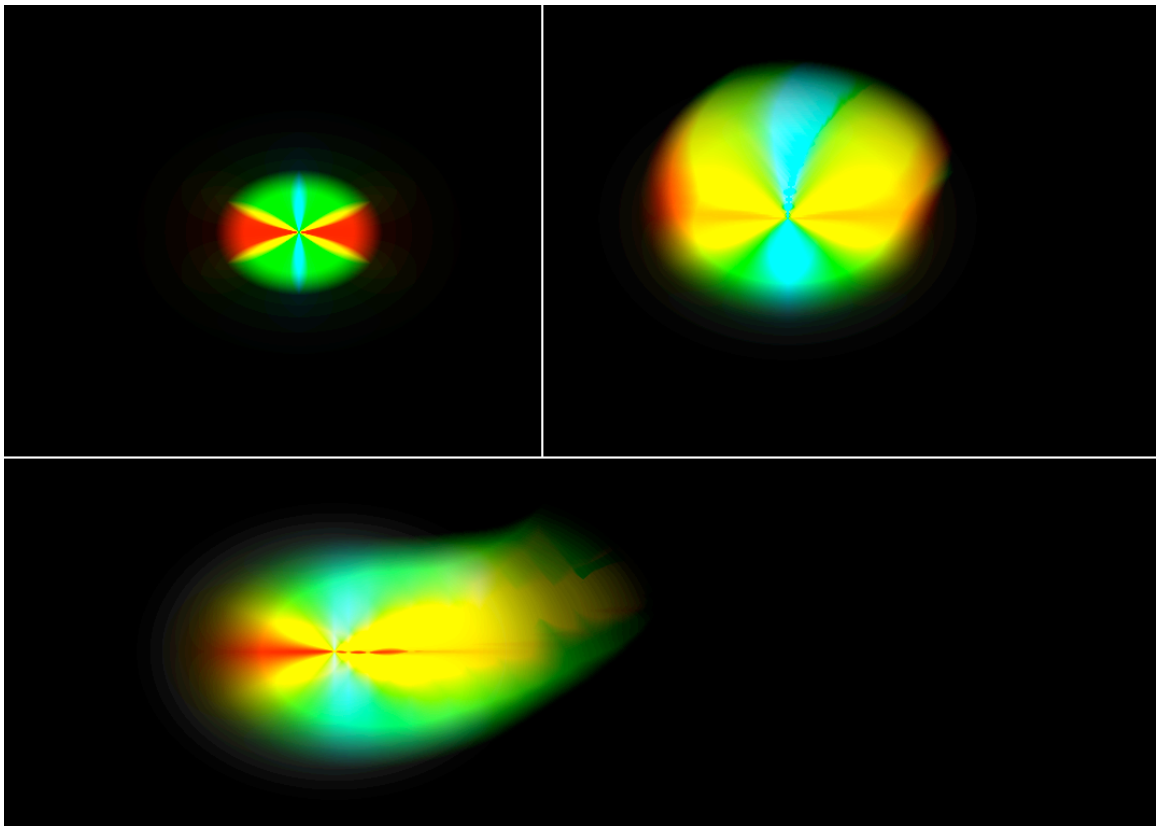

Figure 1: The radial light effect. When the center point is moved quickly, the intensity of the light increases, and previous frames appear as trails.

In the process of developing the persistent ripple effect, a bug introduced a new concept, of distorting the distortion buffer itself before applying it to the scene. This allowed for more consistent water trails, and removed some of the artifacts of my initially described method. However, both effects seem to work best in certain situations. The older effect feels more solid, and works better on photo-like images, while the newer ripple effect works best in drawn or high-contrast scenes.

The produced effects did not reduce the frame rate noticeably. As most of the calculations were handled on the GPU, the CPU was able to dedicate its time to other tasks. The effects tested the limits of current hardware, as the radial light effect used nearly all the math instruction slots, and the most complex ripple effect used four passes. Because of the efficiency found in using these effects, it seems reasonable to conclude that modern hardware is capable of handling persistent realtime effects.

Figure 2: The ripple effect described in the original document, on two different backgrounds to show the distortion in scenes with different contrast levels.

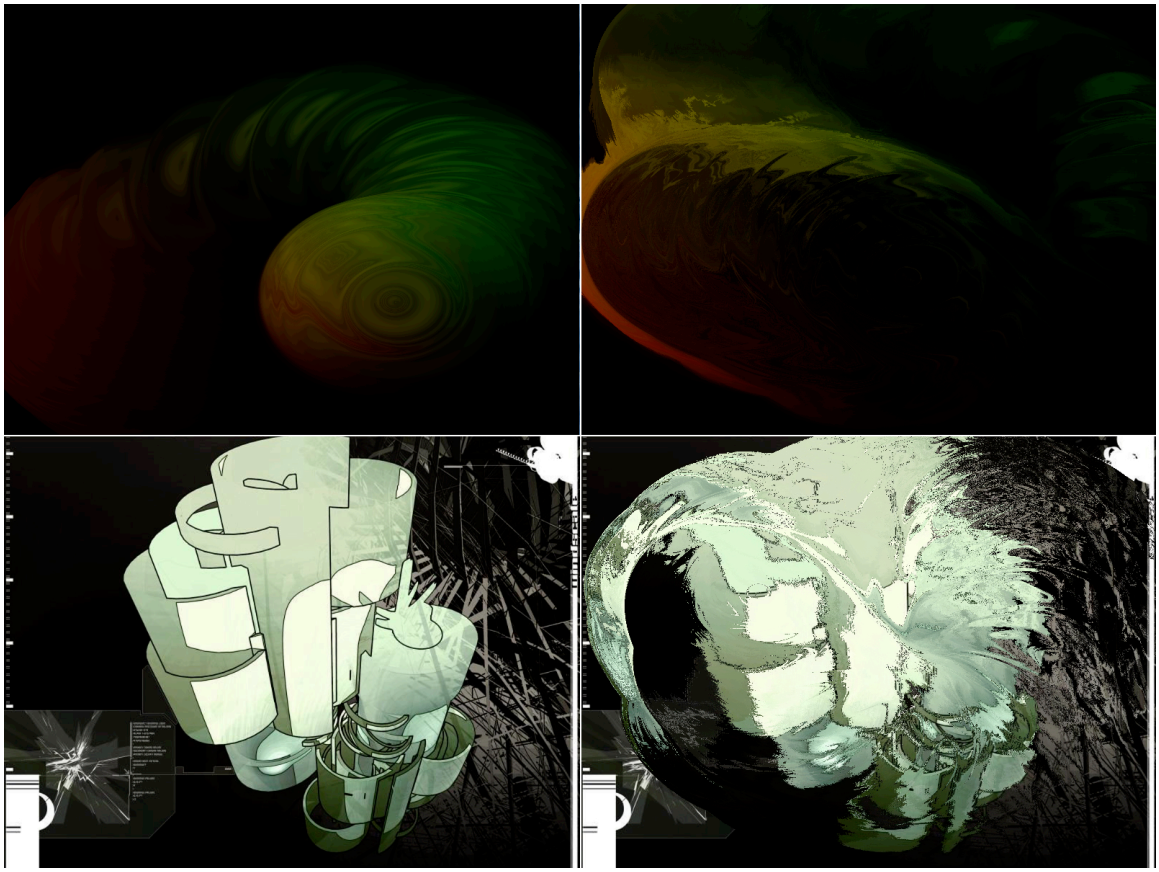

Figure 3: The newer ripple effect. The upper images show the visual representation of the initial distortion buffer (left), and the new distortion buffer (right). The trails in the old distortion buffer simply faded out, which would move the distortion back to zero in a fixed amount of time. In the new distortion buffer, trails blend and ripple more like water. Using the new distortion buffer, the bottom image (left) was distorted as shown (right).