5.5

Texture Masking for Faster Lens Flare

Chris Maughan, NVIDIA Corporation

cmaughan@nvidia.com

This gem introduces a novel way in which to generate texture information from pixels already rendered to the frame buffer. The technique can be used in several different ways, but is presented here as a solution to the common problem of occluded lens flare. Many games attempt to read back pixels generated in the frame buffer in order to determine exactly what was visible in the final scene. I will present a technique that works without CPU assistance, and does not require reading data from the graphics card. I will also outline the reasons why reading back information from the graphics card can be a costly operation and should be avoided if possible.

Lens Flare Occlusion

Many modern games add lens flare to the scene to increase realism. The lens flare is usually applied as the last item in the scene, using a 2D texture map rendered as a billboard over the frame. A complication with this technique is that objects in the scene can occlude the sun image, and in this case, the correct visual result is a lessening of the lens flare intensity. A good way to visualize this is to imagine yourself driving along a road lined with trees on a sunny day; if the sun is below the tree line, you will experience a flickering as your viewpoint through the trees changes and the amount of light reaching your eyes varies.

The usual approach to detecting the occlusion of the sun is to first render the objects in the scene, including the sun itself. Then we read back the frame buffer data around where the sun pixels would be and deduce the amount of sun visible in the scene. We can do this in two ways: we can read back the color buffer and look for values that match our sun color, or we can read back the Z-buffer and look for Z values that are as far away as the sun. The ratio of visible to occluded sun pixels gives us a handy approximation of the intensity of the lens flare. We are now ready to draw our lens flare by blending it onto the final scene using an alpha value to set its intensity.

Hardware Issues

The preceding approach is generally taken when rendering lens flare in games. While it can work well, it causes unnecessary stalls in the graphics pipeline that can seriously impair performance.

Modern graphics pipelines are very deep. Polygon data is not only queued inside the graphic chip pipeline, but it is also queued in a large "staging" buffer. Typically, many thousands of polygons will be queued in the staging buffer by the game, and the graphics chip will drain the buffer as it draws polygons into the frame buffer. In a good parallel system, the game can be doing useful work on the CPU, such as physics, AI, and so forth, while the graphics chip (GPU) is draining the staging buffers. Indeed, this is to be encouraged if the maximum performance of the system is to be achieved. Figure 5.5.1 illustrates a system with optimal parallelism.

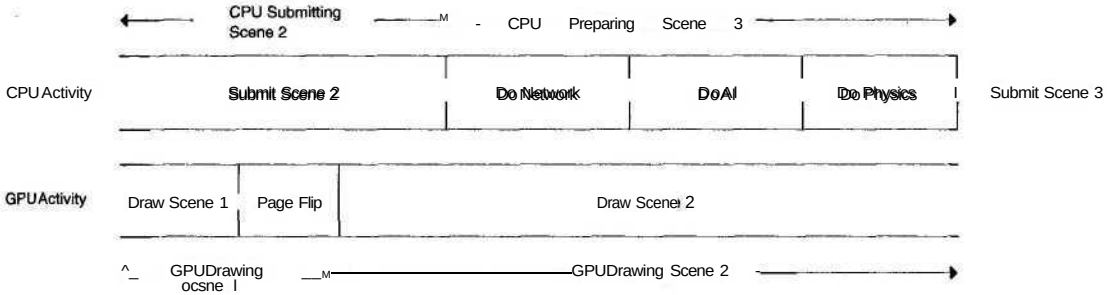

FIGURE 5.5.1 *Parallelism of the GPU I CPU in an ideal game engine.*

Now consider the situation that occurs when the game has submitted all the polygons for the current scene to the graphics chip. Much of the scene may still be queued in staging buffers and will not be drawn for some time. The next thing our game wants to do is to read back the contents of the scene to determine the sun visibility for the lens flare calculation. Here lies the first part of our problem. In order to read the completed scene from the frame buffer, we must wait for the rendering to complete, and in order for this to happen, the whole pipeline must be flushed. While waiting for this to occur, the CPU is effectively unused, as it is simply idle inside the graphics card driver waiting for the rendering to complete. Ideally, we want the GPU and the CPU to be concurrently active at all times. One potential solution to this problem is to insert our physics/AI code after the scene render, but before the lens flare calculation. In this way, the CPU will be busy while the GPU is rendering the polygons.

In this case, if our CPU work is more than the GPU work, then we do not have a problem—although arguably the system is unbalanced because the GPU is waiting for the CPU to finish its work, when it could be doing more rendering (Figure 5.5.2).

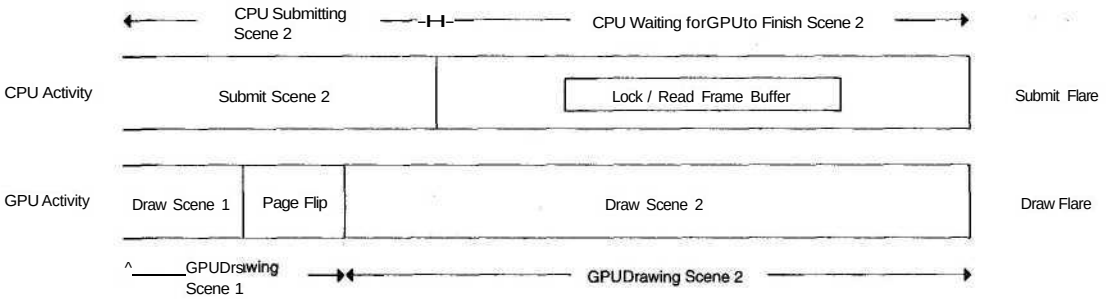

FIGURE 5.5.2 *Pipeline stall caused by flush at end of rendering.*

If our CPU work is less than the GPU work, we will again stall when we try to read back the lens flare data. This is undesirable because ideally we would like to be preparing the next frame of the game on the CPU while the last one completes on the GPU (Figure 5.5.3). Parallelism is the key here, and inserting any stalls can hurt it considerably.

The story does not, however, end here. After losing our concurrency, there is another problem. Reading back data from a current generation graphics card is a slow operation, and in some cases cannot be done at all.

While most graphics cards run at AGP 2x/4x speeds when written to, reading back from them is still limited to PCI speeds—this is inherent in the nature of the AGP bus design as it stands today. The result is that for a 32-bit frame buffer, we can only read pixels at a maximum rate of 66Mhz, with no caching or bursting behavior. Therefore, reading an area of 256 * 256 from the graphics card will take

1 / (66,000,000 / (256 x 256)) = ~1 millisecond.

This assumes a 256-square pixel area of the sun projected onto the screen. If we are standing nearer to the light source—for example, a street-lamp—then the projected area can vary quite considerably until the read back becomes very significant. Even if we are willing to take this hit in performance, there is no guarantee that the

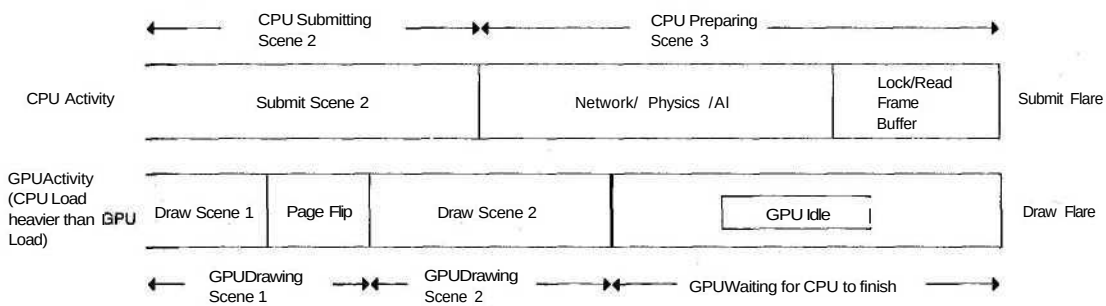

FIGURE 5.5.3 *Pipeline stall caused by flush before end of frame, after CPU work.*

graphics card will allow us to read back the data at all; in fact, many do not. Most will not allow reading of the Z-Buffer, many do not allow reading of the frame buffer, and none will allow reading of an antialiased buffer.

Texture Masking

How can we alleviate the stalls caused by reading the frame buffer? We arrive at the solution through a bit of lateral thinking. We know that we cannot pass a value down from the CPU to modulate the lens flare brightness, for the reasons outlined previously. This leaves only one possibility: that we modulate the lens flare value with data already generated in the graphics memory on the card. Such a value must be in the form of a texel in a texture map, and this texel needs to contain either a luminance color value or an alpha value. We know that we can render to a texture map in Direct3D or OpenGL. The remaining problem is how we actually generate this texel based on a set of source pixels. The solution lies in the alpha blending capabilities of the GPU. The alpha blending unit is basically an adder, which adds a value in the GPU to one on the frame buffer, and can be used to accumulate color information. Add to that the realization that objects in our scene are rendered in a color of our choosing, and a suitable solution presents itself. The following steps show how to achieve the result we need.

Step 1

Create a texture map of 16x16 texels. Call this the *sun map.* We will be rendering the sun itself and the occluding geometry onto this map. The surface format should have at least 8 bits per color component. On typical graphics hardware, this equates to a 32-bit ARGB surface. The sun map is chosen to be 16x16 because it contains 256 texels. As we will see later, we are not limited to a 16x16 size, and can play tricks with filtering to reduce it if necessary.

Step 2

Create a 1x1 texture map. This is the *intensity map.* This is the final destination of our intensity data. We only need a single texel of information, but of course, this texture can be as big as needed to satisfy particular hardware requirements. Again, the format of this surface should be at least 8 bits per color component.

Step 3

Render the portion of the scene containing the sun into the sun map. There are many ways to do this, but we choose a simple approach. We orient the viewer to look directly at the center of the sun. We also set the projection matrix so that the sun fills the viewport. The front plane is just beyond the eye; the back plane is just behind the sun. The sun map is cleared to black. We render one of two colors to the sun map. The image of the sun itself is rendered in white. The occluding geometry is rendered in black, thus

covering any of the sun values already rendered. If the sun is rendered first, Z-BufFering is not required, as we are just interested in the occlusion, not the depth ordering. The sun map now contains 256 values, some of which are white, some of which are black. In 32-bit color, this means that the frame buffer contains OxFFFFFFFF or 0x00000000. Note that for consistency, I set up the alpha channel in the same way as the color channels, enabling the option of alpha blending or color blending.

Step 4

Render 256 pixels to the 1x1 intensity map. Draw them using point-lists or point-sprites if the API/hardware supports them, as they require less transform work than full triangles or quads. Set texture coordinates so that each sun map texel is referenced once. Set the alpha blend unit to add the color information of each of the source sun map texels to the destination intensity map. We also want to sample only one texel at a time from the sun map, so we set texture sampling to point sampling mode. Note that we are writing to a single destination pixel multiple times, with different source texels. One further step is that we blend each sun map texel with a constant value of 0x01010101. This is actually 1/255 in each of the color channels. The result is that the modulated sun map values will either be 0x01010101 if the source sun map texel is white, or 0 if the source sun map texel is black. I choose white and black as values in the sun map because these are generally easy to achieve during rendering, and can be displayed for diagnostic purposes in the demo.

The trick here is that we are adding 256 values from the sun map, to form a possible range of 256 values in the intensity map. The brighter the pixel in the intensity map, the more visible sun pixels there are in the sun map, and hence the scene itself. We have "masked" out the texture results of a previous rendering and added them together. As we will see later, this texture masking approach can be very useful for various applications.

The whole process has used the GPU to generate lens flare intensity from information in the frame buffer. Because the GPU pipeline is serialized, we know that the intensity information will be written before we generate our lens flare, and that the pipeline does not require a flush during the operations discussed earlier. Further, because we have decoupled lens flare intensity generation from scene rendering, we can perform these steps at any point, and potentially use the resulting value during rendering of the actual scene rather than at the end. We could perhaps use this value to modify the ambient intensity of the scene to simulate the viewer's pupils' reaction to the light—an option that is not open to us with the lock method unless we want to render the entire scene twice.

Performance Considerations

The presented algorithm may seem like a lengthy procedure, but is in fact a relatively inexpensive operation when compared to reading the frame buffer and flushing the pipeline. Consider that we are adding a little GPU work at the end of the frame,

which can continue to run in parallel with the rest of our engine. The caveat is that the rendering of the sun map must be quick. We can ensure that this is the case by considering that the sun is usually above ground, and that the field of view to the sun is actually very small. This considerably reduces the number of objects we have to consider, and allows quick rejection of those that we do via frustum culling. We can do early-out rejection if we determine that any of our objects completely occlude the sun, and hence we do not draw the lens flare at all. A further option would be to render the potential occluders with less geometric detail.

Another performance concern may be the rendering of the *256* blended polygons we use to find the intensity result, but this isn't a problem because a modern GPU can render about 40 million points per second—a fraction of the time it will take to read back a potentially large amount of data from the frame buffer, issues of concurrency aside. In addition, reading back data from the frame buffer involves the CPU, which we are trying to free up for nongraphics tasks.

Improvements

The preceding scheme works well, and it gives a good approximate result. One obvious criticism of the technique is that the sun is typically a circular object and the sun map we are using is a square texture. In fact, this is not a problem, as we can sample any texels we like from the source sun map, including a circular sampling pattern. Of course, we need to make the sun map larger in order for it to cover the required number of samples.

If we wish, we can only sample a selection of the sun map texels. To do this, we simply sample the required number of texels in the sun map, and change the modulate operation we use when writing to the intensity map to scale up our results.

Note that we are solving the lens flare problem as an example, but many of the techniques are reusable in other situations. Perhaps the intensity value could be used to darken the scene to mimic overexposure effects, or to add silhouettes or halos to characters. In fact, once we realize that the GPU has great flexibility as a mathematical solver, we can modify our algorithm in a number of ways, using the GPU as a general parallel mathematics engine. For example, we can use texture filtering to sample four texels at a time, giving an average result for four samples. In our previous example, we only need to draw 64 points to get our intensity map results if we rely on bilinear filtering to get the average intensity of the four samples. We can also do several mathematical operations by varying the blending operation, such as using modulation to scale values. This is useful if we wish to accumulate more samples in the intensity map, because we can use the modulation to add scaling to our intensity map results at the expense of dynamic range.

Sample Code

if •;.«& "*§ *ON me co*

The sample demo on the companion CD-ROM gives a good idea of the performance difference between the locking and texture masking techniques, and shows how to

implement the algorithm as described. Options are also available on the menu to enable the display of the sun map and the intensity map.

When running the demo on my target system, I take about a 50-percent frame-rate hit for using the lock method versus the texture masking method. The performance delta also increases the larger the frame buffer becomes. Typical numbers for the opening scene are as follows:

Texture Masking (600x500x32bits) = 53.5 fps
Frame Buffer Locking (600x500x32bits) = 27.8fps

The source is heavily commented and should be easy to follow. I used Direct3D version 8 to write the demo, but the concepts are identical for OpenGL. See the README included with the application on the CD for more information on how to run the demo and analyze the results. Detail is also included on how to use the "Increase CPU Work" option in the demo to study the hit to system parallelism from using the lock call versus the texture masking technique.

s~~z- B^
<-ffii^
ON THE CD

Alternative Approaches

There are two alternative approaches to lens flare that should be mentioned for completeness:

- Sometimes, a geometry-based approach is taken to determine the visibility of the flare. The approach is to scan the geometry and do ray intersection tests to determine visibility of a grid of sun pixels. While this works, it can be costly in terms of CPU compute time, and is not a useful approach when using billboards, as the source textures have to be scanned for "holes."
- Some graphics cards have the ability to do asynchronous readback of the frame buffer data. In this case, the light source area is uploaded in parallel with the rendering of the rest of the scene. This can be useful assuming that there is a suitable point in the scene after occlusion objects are drawn in which to start the readback, but before the rest of the scenery or lighting work is done. This method does of course rely on the support from the API to read back the data asynchronously, and support from the hardware to upload the data. At the time of writing, this support is not available in Direct3D or OpenGL, although extensions to both APIs have been proposed.

References

[KingOO] Yossarian King, "2D Lens Flare," *Game Programming Gems*, Charles River Media Inc. 2000: pp. 515-518.