The F-Buffer: A Rasterization-Order FIFO Buffer for Multi-Pass Rendering

William R. Mark and Kekoa Proudfoot

Department of Computer Science*
Stanford University

Abstract

Multi-pass rendering is a common method of virtualizing graphics hardware to overcome limited resources. Most current multi-pass rendering techniques use the RGBA framebuffer to store intermediate results between each pass. This method of storing intermediate results makes it difficult to correctly render partially-transparent surfaces, and reduces the performance of shaders that need to preserve more than one intermediate result between passes. We propose an alternative approach to storing intermediate results that solves these problems. This approach stores intermediate colors (or other values) that are generated by a rendering pass in a FIFO buffer as the values exit the fragment pipeline. On a subsequent pass, the contents of the FIFO buffer are fed into the top of the fragment pipeline. We refer to this FIFO buffer as a fragment-stream buffer (or F-buffer), because this approach has the effect of associating intermediate results with particular rasterization fragments, rather than with an (x,y) location in the framebuffer. Implementing an F-buffer requires some changes to current mainstream graphics architectures, but these changes can be minor. We describe the design space associated with implementing an F-buffer, and compare the F-buffer to recirculating pipeline designs. We implement F-buffers in the Mesa software renderer, and demonstrate our programmable-shading system running on top of this renderer.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1 Introduction

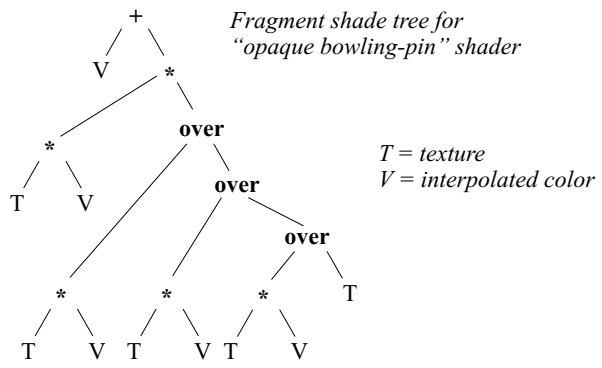
Real-time graphics hardware and applications are rapidly switching from fixed shading algorithms to fully programmable shading. Programmable shading is valuable because it can realistically model the enormous variety of materials and lighting effects that exist in the real world.

Over the past few years, each new generation of graphics hardware has supported programmable shading better than the previous generation by increasing the number of operations provided in each rendering pass. The fall 1999 NVIDIA GeForce chip supported two texture lookups and two pixel instructions in one pass; the spring 2000 ATI Radeon chip supported three textures and three instructions; and the spring 2001 NVIDIA GeForce3 chip supports four textures and eight instructions. However, the complexity of the shading programs (*shaders*) that users want to write continues to grow as well – it is easy to write shaders that exceed the capability of a single GeForce3 rendering pass.

We expect that this situation will continue. Even if a programmable pixel/fragment pipeline could support a large number of instructions, shaders would be able to exhaust other resources, such as texture units, vertex-to-fragment interpolants, or registers.

The standard solution to the problem of limited hardware resources is to virtualize the hardware using some combination of hardware and software techniques. For example, limited CPU DRAM is virtualized using disk storage (i.e. virtual memory).

On graphics hardware, multi-pass rendering has been the preferred technique for virtualizing limited hardware resources. Figure 1 shows an example of the multi-pass technique; the top of the figure illustrates a complex shader in the form of a shade tree [3], and the bottom of the figure shows the mapping of this shader onto multiple passes of a dual-texture OpenGL pipeline. As is the case for most multi-pass shaders, this shader uses the framebuffer to store intermediate results between rendering passes. Each rendering pass uses the same viewpoint and geometry, but a different pipeline configuration.*



The shade tree above maps to five passes of a dual-texture rendering pipeline, as shown below. Only the dark-colored units are used by the shader.

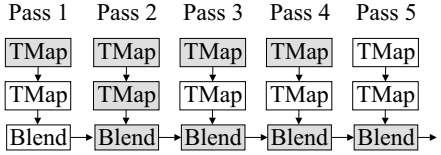


Figure 1: *A shade tree and its mapping onto five dual-texture OpenGL rendering passes.*

Application programmers, particularly game developers, have made increasing use of multi-pass rendering techniques to achieve sophisticated visual effects [1, 5]. However, this strategy of using

*Gates Building; Stanford, CA 94305 USA.
email: {billmark | kekoa}@graphics.stanford.edu
www: http://graphics.stanford.edu/~{billmark | kekoa}

*In this paper, we are only concerned with multi-pass rendering used for hardware-resource virtualization, which uses a single viewpoint for all passes. Multi-pass rendering can also be used to generate images from different viewpoints (e.g. to produce shadow maps), but we are not concerned with this second type of multi-pass rendering.

the framebuffer to store intermediate results between rendering passes has several problems:

- Overlapping, partially-transparent surfaces are rendered incorrectly.
- Each pass can only store one RGBA per-pixel result in the framebuffer, even though current programmable pipelines can generate more than one intermediate result in a single pass.
- If texture memory is used to hold per-pixel intermediate results (using frame-to-texture-copy or render-to-texture), some of this texture memory is typically wasted, because it is allocated but not used.
- Geometry must be re-specified and re-transformed for every pass.

These problems are a key obstacle to correct and efficient execution of complex, multi-pass shaders on current graphics hardware.

In this paper, we describe an alternative method for graphics hardware to store the intermediate results produced by each pass of a multi-pass shader. This alternative storage method eliminates many of the problems with the current framebuffer-based approach to storing intermediate results.

This new method is conceptually simple. It stores intermediate colors (or other values) that are generated by a rendering pass in a first-in, first-out (FIFO) buffer as the values exit the rasterization pipeline. On a subsequent pass, the contents of the FIFO buffer are fed into the top of the rasterization pipeline. We refer to this FIFO buffer as a fragment-stream buffer (or F-buffer), because this approach has the effect of associating intermediate results with particular rasterization fragments, rather than with an (x,y) location in the framebuffer.

In the next section, we will discuss the difficulties with the conventional approach to multi-pass rendering in more detail. These difficulties led us to develop the F-buffer approach. In sections 3 and 4, we explain the F-buffer approach in more detail and discuss its advantages. In section 5, we discuss a variety of approaches to incorporating F-buffers into a graphics architecture. We describe our demonstration implementation of the F-buffer in section 6. In the last two sections of the paper, we compare the F-buffer to other approaches and conclude.

2 Problems with Framebuffer Storage of Intermediate Results

Using the framebuffer for temporary storage in multi-pass rendering presents efficiency problems for several classes of shaders. In particular, shaders that require more than one simultaneously-live intermediate result run inefficiently and shaders that represent partially-transparent surfaces run either incorrectly or inefficiently. We address these two cases in turn.

First, consider a multi-pass shader that requires two simultaneously-live intermediate results. That is, at some point in the shader's execution there are two temporary per-pixel variables that will be read at least once more. Because the framebuffer can only hold one intermediate result at a time, the framebuffer must be "spilled" to other storage – typically texture memory [10] – to make room for the second intermediate result. Figure 2 illustrates a shader that requires this spilling. Normally, this spill texture must be the same size as the framebuffer, although it can be smaller if the software maintains a screen-space bounding box geometry.

If the graphics hardware lacks render-to-texture capability, then a spill to texture memory requires an expensive framebuffer-to-texture copy. On graphics hardware with a unified texture/framebuffer memory, this cost can be avoided, but other problems remain.

Newer hardware with long programmable pipelines, such as the GeForce3, can easily generate *more than one* intermediate result in a single pass. Currently, only one of these intermediate RGBA results can be saved; any other temporary results must be discarded and re-computed on subsequent passes. This one-live-register bottleneck at the end of a rendering pass is difficult for shading compilers to deal with – as an analogy, consider the difficulty of compiling to an x86 processor that invalides all but one of its registers every eight instructions. This bottleneck also makes it difficult to dynamically combine shader fragments at run time, and this difficulty is one of the reasons that our programmable shading system [11] requires that surface and light shaders be combined at compile time rather than render time.

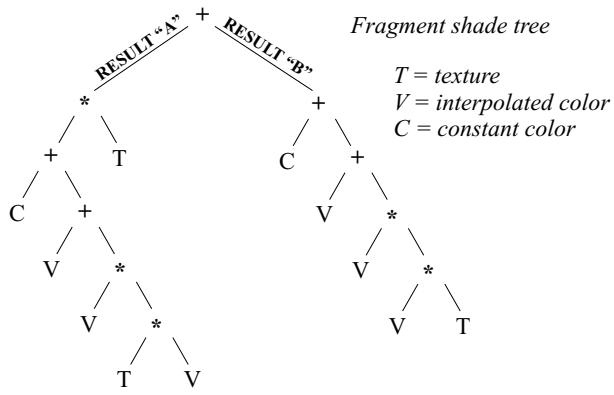


Figure 2: *This shader requires at least one framebuffer spill on a two-texture rendering pipeline. Once "Result A" is computed, the resulting image must be saved to texture memory so that the framebuffer can be used to compute "Result B".*

A second problem with the current approach to multi-pass rendering is that overlapping partially-transparent surfaces are rendered incorrectly. Consider what happens at a single pixel when two such surfaces are rendered using the same shader: The intermediate result for the rear surface is overwritten by the intermediate result for the front surface. If these two values are different (e.g. due to different texture coordinates), then subsequent passes will compute incorrect results for the rear surface, causing the final blended color to be incorrect as well.

None of the work-arounds for this transparency problem on current hardware are very satisfactory. The simplest solution is to render each partially transparent polygon completely independently (i.e. render all of its passes before proceeding to the next polygon). This solution is usually prohibitively expensive due to the state-change cost which is incurred. Alternatively, the application or shading library can group polygons to ensure that only non-overlapping polygons are rendered together. In most cases this solution is not attractive either, because it requires the software to perform screen-space analysis of polygons. Finally, Everitt's *depth peeling* technique [4] can solve the overlapping-transparency problem, but only at the cost of a multiplicative increase in the number of rendering passes.

3 The F-Buffer

The F-buffer provides an alternative method for storing intermediate results during multi-pass rendering. As fragments are rasterized in the first pass of a shader, the fragment data generated by the pass is stored in a FIFO buffer (an F-buffer). This data will generally include one or more RGBA colors (or other temporary values) for each fragment. For some variants of the F-buffer approach, this data also includes each fragment's X, Y, and Z values. In a subsequent pass of the shader, this stored data is read from the FIFO buffer and fed into the top of the graphics pipeline (e.g. as an input to texture combiners), where it is used for the pass's computations (Figure 3).

Generally, every rendering pass except the first reads from one or two F-buffers, although if a pass represents a leaf of the shade tree it does not need to read from an F-buffer. Every shader pass except the last always writes to one F-buffer. The last pass of the shader writes to the framebuffer in the usual manner. Although only one F-buffer can be written at a time, multiple F-buffers can be stored in memory, thus allowing multiple intermediate results to be saved and re-used as needed.

Some rendering passes must combine results from two earlier passes. In conventional multi-pass rendering, one such result is in the framebuffer, while the other is read from its spilled location in texture memory. When using F-buffers, this situation is handled by simultaneously reading from two F-buffers, so that there are two sets of colors (or other data) for each fragment. The shader shown earlier in Figure 2 must read two F-buffers to perform the addition operation at the root of its shade tree. Hardware that can simultaneously read two F-buffers and write one F-buffer is sufficient to render any shade tree composed of binary operations.

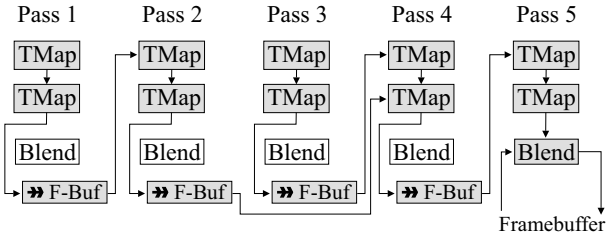


Figure 3: *F-buffers are used to hold results in between rendering passes.*

The most important property of the F-buffer is that it associates a unique storage location with each rasterized fragment. In contrast, a framebuffer can associate more than one fragment with a single pixel, if there are overlapping polygons rendered using the same shader. The F-buffer can be considered to be a special case of the stream buffers used in more general-purpose stream-oriented architectures [8, 12]. The F-buffer is also related to an A-buffer [2, 13], in the sense that the A-buffer can also store information for multiple fragments associated with a single pixel.

4 Advantages of F-Buffers

F-buffers solve the problems with conventional multi-pass rendering that we discussed earlier, and provide other advantages as well.

An F-buffer associates each fragment with its own storage location. As a result, the reads from an F-buffer are not subject to read-after-write hazards as framebuffer reads are, removing one of the major obstacles to making previous results available at the top of the fragment pipeline. Feeding an F-buffer into the top of the fragment pipeline allows intermediate results to be used as inputs to any of the functional units in the fragment pipeline, rather than to just the blend unit. These advantages are similar to those provided by render-to-texture capability, although render-to-texture is still subject to limited read-after-write hazards.

An F-buffer's association of each fragment with its own storage location eliminates the transparent-surface-rendering difficulties of conventional multi-pass rendering. There is no longer a storage-location conflict between multiple polygons covering the same pixel, although partially-transparent surfaces must still be rendered in back-to-front order.

F-buffers use graphics memory more efficiently and flexibly than auxiliary framebuffers (deep framebuffers) would. An F-buffer uses just enough memory to hold the fragments produced by the current shader. In contrast, an auxiliary framebuffer also uses memory for all of the pixels that are *not* touched by the current shader. Thus, it is usually more efficient to use several F-buffers for a multi-pass algorithm than it is to use several auxiliary framebuffers. The render-to-texture or copy-to-texture technique falls in between these two extremes, because a screen-space geometry bounding box can be used to reduce the size of the texture. However, maintaining this bounding box can be a serious imposition on the rest of the system, and unless the object(s) being shaded are perfectly rectangular, some memory is still wasted.

Unlike a deep framebuffer, an F-buffer can store an essentially arbitrary number of values with each fragment without increasing the total amount of graphics memory that is required for a given screen resolution. For example, an F-buffer might store two RGBA colors and a floating-point s,t texture coordinate with each fragment. We expect that this capability will become more useful as individual rendering passes become more powerful. There is of course a tradeoff between the size of a single fragment's entry in an F-buffer and the total number of fragments than can be stored in an F-buffer of a particular size. But as long as the rendering system can handle overflow of an F-buffer (an issue that we discuss in Section 5.5), it is useful to have this tradeoff available. In contrast, with the conventional framebuffer approach, the only available tradeoff is between framebuffer depth and resolution. This depth/resolution tradeoff can be made only once, for all shaders in the scene, rather than separately for each shader.

The writes to an F-buffer and reads from an F-buffer are perfectly coherent, since F-buffer accesses are FIFO rather than random. For an off-chip F-buffer, this property allows memory reads and writes to efficiently use large-granularity transfers.

An F-buffer allows multi-pass rendering algorithms to work correctly even if the framebuffer uses a lossy compression format. An example of such a format is the Z^3 A-buffer format [6], which approximates full multi-sample storage with only a few stored values at each pixel. If such a framebuffer is used with the conventional multi-pass approach, some shaders may not work correctly, because they assume that data will not be modified by compression. Difficulties are particularly likely to occur when an intermediate value represents non-color data such as texture coordinates, because compression errors in this data can be non-linearly magnified in the final color image. These problems can be avoided if intermediate results are stored uncompressed in an F-buffer, with one entry in the F-buffer for every multi-sample.

5 Design Alternatives

There are many variants of the basic F-buffer idea. In the next few sections, we discuss several important dimensions of the design space. For each major design-space dimension, we list the major alternatives, with some of their advantages and disadvantages. We usually do not indicate a preferred alternative, because the choice of a best alternative depends other hardware-design choices, and on implementation costs that we can not precisely estimate.

5.1 Where are F-buffers stored?

F-buffers can be stored on-chip, in graphics DRAM, in host DRAM, or in some combination of these three. The most promising configurations are a moderately-sized on-chip F-buffer, and an off-chip F-buffer in graphics DRAM. The former has the advantage of conserving memory bandwidth, but is likely to be small enough that it fills up when a shader's geometry covers a large percentage of the screen. A hybrid which uses on-chip storage until it fills up, then uses off-chip storage, provides the benefits of both approaches.

5.2 Are polygons rasterized on every pass?

The F-buffer approach to multi-pass rendering provides the opportunity to skip polygon rasterization on the 2nd through Nth passes of a multi-pass shader. This single-rasterization variant of the F-buffer approach requires that the screen-space x,y location and depth be stored in the F-buffer with each fragment, since these values will not be re-generated. It also requires that all polygon interpolants (e.g. colors, texture coordinates) be generated during the first pass, and stored in the F-buffer if they are needed on subsequent passes. This requirement increases the size of F-buffers, and the memory bandwidth needed to access them. It also makes the F-buffer implementation more complex. Some shaders may not run if the hardware places a limit on the number of polygon interpolants available in a single pass. If programmable vertex hardware is available, other shaders may not run if they exceed the limit on the number of vertex instructions that can be executed in a single pass.

An advantage of this single-rasterization approach is that it avoids the need for the shading library or graphics driver to feed geometry data to the graphics pipe multiple times. This change reduces the demand for CPU-to-graphics-card bandwidth, as compared to a multiple-rasterization approach which buffers geometry on the host. The single-rasterization approach also greatly simplifies the shading library or graphics driver's task of providing applications with the useful illusion that all shaders run in one pass.

If the alternative multiple-rasterization approach is used, there is an important requirement that is imposed on the hardware: Rasterization of the same geometry data must always produce the same set of fragments, in the same order, even if the fragment-pipeline configuration is different. This restriction is removed for the single-rasterization approach.

5.3 When are conventional framebuffer operations performed?

The natural time at which to perform conventional framebuffer tests (depth test, alpha test, stencil test, etc.) is at the end of the last pass of a shader. However, under some circumstances it is possible to perform some or all of these tests at the end of the first pass of the shader, thus allowing some fragments to be discarded before they are written to the first F-buffer. We refer to such a test as an *early test*. Note that the early test only guarantees rejection of fragments that are occluded by pixels *already* in the framebuffer; another test on the second or final pass is still required to resolve all occlusions by other fragments in the same F-buffer.

In order to perform an early test, F-buffer writes must be deferred until after the framebuffer tests, and all of the values upon which the framebuffer test depends (e.g. Z for the depth test) must be computed during the first shader pass. If all of the framebuffer tests are performed in the first pass, the Z value and any other values which are already computed can be written to the framebuffer at the end of the first pass. The hardware must be able to write to both an F-buffer and the Z-buffer on the same pass.

Early tests look less attractive for the multiple-rasterization approach to F-buffering than they do for the single-rasterization approach. The multiple-rasterization approach would require a method to synchronize incoming F-buffer streams (which are missing discarded fragments) with rasterized fragments in the 2nd through Nth passes. The complexity and cost of this resynchronization using a technique such as fragment ID's is likely to outweigh the performance gains from early testing.

5.4 Where does F-buffer data enter the fragment pipeline?

In our earlier overview of F-buffering, we were deliberately vague about exactly where the data from an F-buffer entered the fragment pipeline. Two major options are available, which we discuss in terms of the OpenGL fragment-pipeline model [7].

The first option is for the F-buffer data to replace a texture color when the pipeline is configured for F-buffer reads. A dual-texturing pipeline then provides the needed capability to read from two F-buffers simultaneously. This option has the potential to re-use the texture-read hardware for F-buffer reads, but reduces the number of textures that can be accessed when F-buffer(s) are being read.

A second option is for the F-buffer data to appear as a new input to the programmable fragment hardware (e.g. as a new register-combiner register). This option does not disable any texture-read units when an F-buffer is being read, but may require new hardware that is dedicated to F-buffer reads. If the F-buffer architecture supports more than one RGBA value for each fragment, each RGBA value from the F-buffer can appear in a separate register.

5.5 Strategies for handling F-buffer overflow

The most difficult issue associated with F-buffers is the problem of buffer overflow. If there is any maximum size imposed on an F-buffer, it is possible for a shader to overflow the F-buffer if it is used to rasterize a sufficiently large number of fragments. Even if the F-buffer can grow to the size of a framebuffer, overflow will still occur if enough overlapping polygons are rendered. Thus, any architecture which uses an F-buffer must provide a hardware and/or software method for dealing with this issue. It is important to note that if the complete system is designed so that F-buffer overflows are rare, the technique(s) used for handling overflow do not need to be particularly efficient.

If the total amount of available F-buffer memory is fixed, the maximum allowed size of *each* F-buffer varies depending on the number of simultaneously-live F-buffers required by a shader. Making an analogy to conventional CPU compilation, this number is equivalent to the maximum number of live registers required by a block of code. If n live F-buffers are required by a shader, then the maximum number of fragments that can be held in each F-buffer is $1/n$ of what it would be for the one-live-F-buffer case.* Since the application or shading library knows how many F-buffers are required by a shader, it can adjust its notion of the maximum allowed size for each F-buffer accordingly.

There are several major approaches to the F-buffer overflow problem:

- Allow the F-buffer to "overflow" to higher-capacity memory (e.g. off-chip graphics memory, then main memory) to provide the illusion of an infinite-sized buffer. Because F-buffers are FIFO buffers, not random-access buffers, providing a multi-level memory hierarchy for them is fairly

*This statement assumes that the framebuffer tests (e.g. depth test) are performed at the tail of the last rendering pass, so that all F-buffers used by a shader will require storage for exactly the same number of fragments.

simple. This solution is ideal in the sense that it minimizes the impact on software of dealing with buffer overflows.

- Advertise the F-buffer size limit to the software, and shift the burden of avoiding overflows entirely to software. The software must process polygons in batches that fit into the F-buffer, by executing all shading passes for one batch before proceeding to the next. The major problem with this solution is that buffer size is measured in fragments, but software manipulates polygons or other higher-level geometry. Expensive (and redundant) computations are required for software to estimate the size of a polygon in fragments.

- In a modification of the above approach, the hardware makes the current F-buffer fragment count available to the software. However, in most reasonable implementations this count will always be out-of-date in the sense that it will not include polygons that are currently in the geometry portion of the graphics pipeline.

- The hardware allows the F-buffer to fill up, but provides the software with information about exactly when this overflow occurred, so that rendering can be restarted at the correct point for the next batch of geometry. The hardware may need to provide support for this rendering restart.

The first three options are straightforward, so we will not discuss them further. The last option is more complex, with several possible variants. In particular, the rendering restart can be designed to occur one of at several different granularities. We will discuss fragment granularity, although other granularities are also possible. Some of the other possibilities include triangle granularity, span granularity, primitive-group granularity (i.e. glBegin), or application-defined granularity (inserting checkpoints in the graphics stream). Hybrids of these are possible as well.

For fragment-granularity restart, the hardware notifies the software of the fragment-number (f) that caused an F-buffer overflow. The hardware discards any in-flight fragments and polygons after the overflow. When the software detects this overflow, it stops issuing geometry, and completes the 2nd through Nth passes of the current batch of geometry. Rendering of the 2nd and subsequent batches of geometry is done in a slightly different manner. For these batches of geometry, the software tells the hardware to discard the first f fragments of geometry, then starts rendering geometry *from the beginning*. This approach is costly when buffer overflows occur; if B batches of geometry are rendered, the cost is $\frac{1}{2}B$ times the non-overflow cost. Thus, this approach is designed to provide correctness on buffer overflow, rather than high performance.

For fragment-granularity restart, the hardware must have the following relatively-simple capabilities:

- Ability to discard output fragments less than f', and greater than f' + sizeof(FBuffer), on output to either an F-buffer or to the framebuffer.

- Ability to inform software (even if only by polling) that some fragments greater than f' + sizeof(FBuffer) have been discarded. From the software's point of view, overflow detection may be be delayed (due to pipeline latency); however, the graphics API should also provide the option for a non-delayed F-buffer status query, with the understanding that using this option may require an expensive partial or complete pipeline flush.

- Ability to avoid reading from an input F-buffer until after f' fragments have been processed.

Hardware designers considering implementation of an F-buffer will want to know how frequently overflows will occur for a given buffer size. Figure 4 plots relevant data gathered from tracing and analysis of Quake III's demo001 sequence, with OpenGL extensions disabled. We used heuristics to identify the start and end of shaders within the OpenGL command stream, and spot-checked these heuristics against QuakeIII's built-in logging capability, which identifies shaders. Surprisingly, a few one-pass shaders produce more fragments than there are pixels on the screen. In-depth examination of the traces showed that these shaders were primarily full-screen "explosion" shaders and "blood-splat" shaders.

If the geometry rendered using a particular shader is likely to overflow the F-buffer, the software (application, library, or driver) can improve performance by cycling through the passes of the shader more than once per shader change. In other words, the software can improve performance by batching large geometry that is rendered using complex shaders. If the driver manages multi-pass shading, it can perform this batching, thus hiding it completely from the application.

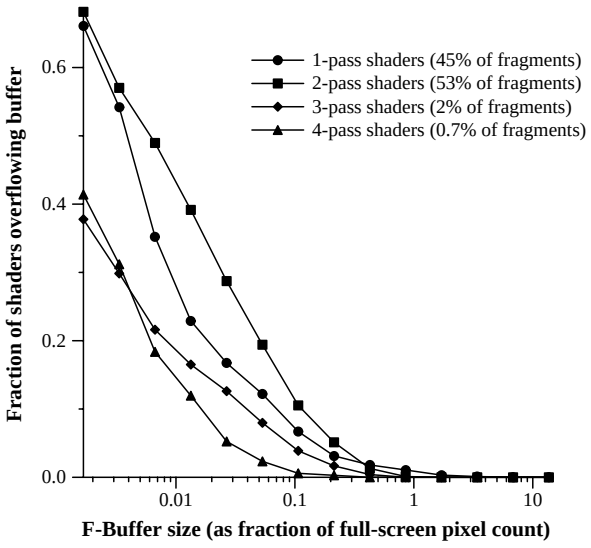


Figure 4: *Fraction of QuakeIII 'demo001' shaders that encounter overflow as a function of F-Buffer size*

6 Demonstration Implementation

We have built a demonstration software implementation of F-buffers by modifying version 3.1 of the Mesa renderer [9]. We used Mesa's software-only rendering path for this implementation, and added our own OpenGL extensions to the Mesa API to support F-buffers. We also added the GL_EXT_texture_env_combine extension to Mesa, because it is necessary for effective use of F-buffers.

We have modified our real-time programmable shading system [11] to run on top of this version of Mesa. Our goal in this effort was to demonstrate that F-buffers could be conveniently used by a programmable-shading system. In particular, we wanted to verify that we could adequately address the F-buffer-overflow issue from a software architecture standpoint. Note that our "real-time" programmable shading system is no longer truly real-time when running on top of any software renderer.

For our F-buffer implementation, we made design choices which we believe correspond to a minimization of the changes to existing

mainstream graphics hardware, while still satisfying the needs of a programmable-shading system. We made the following choices:

- F-buffers hold a single RGBA value for each fragment.
- F-buffers are stored in a generic memory space, but have a pre-determined maximum size.
- We handle F-buffer overflows using the fragment-granularity restart technique discussed earlier. Our programmable-shading library performs the software portion of overflow management, so that application programmers can completely ignore the issue of F-buffer overflows when they use our shading library.
- For multi-pass shaders, triangle rasterization occurs on every pass, rather than on just the first pass.
- F-buffers are written just prior to the OpenGL alpha test; all conventional framebuffer tests (e.g. alpha, stencil, depth tests) are performed at the end of the last pass of a shader.
- F-buffer data enters the fragment pipeline in place of a texture. Mesa has two texture units, so up to two F-buffers can be read in a single pass. If a texture unit is not used to read an F-buffer, it is available for texturing as usual.

Figure 5 shows two screenshots from our real-time programmable-shading system running on top of our modified version of Mesa. Table 1 lists the memory bandwidth required by different types of passes with and without F-buffers.

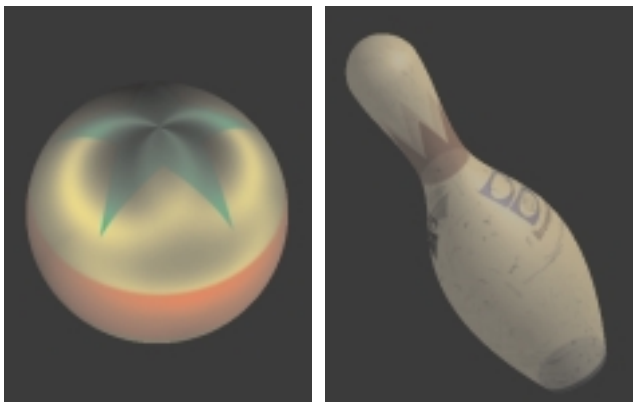


Figure 5: *Screenshots from our programmable-shading system running on top of our software F-buffer implementation. Without F-buffers, the image on the left would require two copy-to-texture (or render-to-texture) operations. The image on the right is a partially-transparent bowling pin, which requires seven rendering passes. All of the geometry is rendered together, demonstrating that F-buffers support correct rendering of overlapping, partially-transparent polygons.*

7 Comparison with Recirculating Pipeline

One alternative to the F-buffer for graphics-hardware virtualization is what we refer to as a *recirculating pipeline*. A recirculating pipeline allows fragments to make multiple trips through the hardware pipeline without being stored in any type of intermediate buffer. Enlarged pipeline-state tables allow a different set of operations to be performed on fragments during each pipeline trip. This type of architecture does not have to be directly exposed at the API level – it can instead be exposed as a very long pipeline.

	F-buffers		Std. Multi-Pass	
	Read	Write	Read	Write
First Rasterization Pass	0	$1f$	$1f$	$0/2f$
2..N-1 Rasterization Pass	$1f$	$1f$	$1/2f$	$0/1f$
Last Rasterization Pass	$2f$	$0/2f$	$1/2f$	$0/1f$
Copy-to-texture (if used)	—	—	$1S$	$1S$

Table 1: *Bandwidth comparison between F-buffering and standard multi-pass, measured in 32-bit words, assuming 32-bit color and 32-bit depth. The variable f represents the number of fragments generated by the geometry. The variable S represents the number of pixels covered by the rectangular area that could contain rendered geometry. Two results separated by a slash indicate the cost for surface-visible / surface-not-visible, with the assumption that color is not read on a failed depth test. The following assumptions are used for this table: The F-buffer variant is the one described in Section 6; the surface being rendered is opaque; and only one F-buffer is used for input on each pass.*

The recirculating pipeline addresses most of the same problems that the F-buffer addresses. It has the advantage that it uses less memory bandwidth than an off-chip F-Buffer and is simple to support in software. It also has the disadvantage that it may require a larger texture cache, because more textures are simultaneously active.

The advantages of a recirculating pipeline are sufficiently compelling that we believe some variant of it is already being used in current graphics architectures. However, there are several limitations of recirculating pipeline designs. First, the maximum number of pipeline trips is limited by the state-table size. Second, the maximum number of live temporary variables (including vertex-to-fragment interpolants) is limited by the number of registers carried with each fragment. Finally, the maximum number of vertex-program instructions is limited by the size of the vertex-program storage memory. In summary, a recirculating pipeline has all of the problems that a longer non-recirculating pipeline would have, so a recirculating pipeline provides only partial hardware virtualization.

All of these limitations can be overcome by providing support for F-buffers in conjunction with a recirculating pipeline. An F-buffer allows a complex shader to be broken into multiple hardware passes, with each of these passes consisting of several trips through the recirculating pipeline. Support for very complex shaders is particularly valuable for applications such as previewing of off-line rendering, where performance should be interactive, but does not need to be 60 frames/sec.

8 Discussion and Conclusion

F-buffers efficiently support multi-pass shaders that require more than one live intermediate result, and multi-pass shaders that describe partially-transparent surfaces. They also allow more than one result to be saved from each pass of a programmable fragment pipeline. By removing some of the fundamental application-visible differences between single-pass and multi-pass rendering, F-buffers assist in providing a virtual fragment pipeline with an *arbitrary* number of instructions and registers. This virtual fragment pipeline gracefully and generally supports shaders of arbitrary complexity.

Although we anticipate that the capabilities of a single rendering pass will continue to improve, we believe that users will always want to be able to execute even more complex shaders by

using multiple hardware passes. This expectation is likely to be particularly strong if users access programmable hardware via high-level shading languages. The ability to support complex shaders is especially valuable for applications such as scientific visualization, game prototyping, and previewing of off-line rendering.

The most serious difficulty in implementing F-buffers is the issue of buffer overflow. We believe this issue is solvable with any one of several approaches that we have suggested. The multi-level memory approach minimizes the impact on software; but the fragment-granularity-restart approach provides an alternative that we believe minimizes the necessary changes to current hardware designs. Because F-buffers should be able to draw from a common pool of graphics memory, we expect that their size can be chosen such that buffer overflows are a relatively rare event that only occurs when performance is already poor due to a high rendered-pixel count. Therefore, any additional performance degradation caused by the system's technique for handling F-buffer overflow is likely to be tolerable.

If each rendering pass re-rasterizes polygons, the use of F-buffers requires that the rasterizer generate identical fragments on each pass, in a consistent order. In a hardware design which meets this restriction, we believe that F-buffers can be implemented with relatively minor changes to the design, primarily by adapting the texture-read and framebuffer-write units to read and write F-buffers.

We also believe that F-buffers can be implemented on rendering architectures which perform screen space subdivision in time (i.e. tiling), or across parallel processors. On such architectures, each screen region must maintain its own subset of each F-buffer.

9 Acknowledgments

This work was conducted as part of the Stanford real-time programmable shading project, which is sponsored by ATI, NVIDIA, SONY, and Sun. Our meetings with individuals at these companies have contributed substantially to our understanding of graphics hardware; we thank Roger Allen and Matt Papakipos in particular for their comments on this work. We also thank the other members of the Stanford Graphics Hardware group, in particular Pat Hanrahan, for ongoing discussions about the ideas in this paper.

References

[1] SIGGRAPH 1999 course 29: Advanced graphics programming techniques using OpenGL, August 1999.

[2] Loren Carpenter. The A-buffer, an antialiased hidden surface method. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):103–108, July 1984.

[3] Robert L. Cook. Shade trees. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):223–231, July 1984.

[4] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, May 2001. Available at http://www.nvidia.com/.

[5] Paul Jaquays and Brian Hook. *Quake 3: Arena Shader Manual, Revision 10*, September 1999.

[6] Norman P. Jouppi and Chun-Fa Chang. Z^3: An economical hardware technique for high-quality antialiasing and transparency. In *Eurographics/SIGGRAPH workshop on graphics hardware*, pages 85–93, Los Angeles, CA, August 1999.

[7] OpenGL ARB, M. Woo, J. Neider, T. David, and D. Shreiner. *OpenGL programming guide*. Addison-Wesley, third edition, 1999.

[8] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. In *SIGGRAPH/Eurographics workshop on graphics hardware*, pages 23–32, Interlaken, Switzerland, August 2000.

[9] Brian Paul. The Mesa 3D graphics library. Available at http://www.mesa3d.org.

[10] Mark Peercy, Marc Olano, John Airey, and Jeff Ungar. Interactive multi-pass programmable shading. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 2000)*, pages 425–432, New Orleans, LA, July 2000.

[11] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 2001)*, Los Angeles, CA, August 2001.

[12] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, Dallas, TX, November 1998.

[13] Andreas Schilling and Wolfgang Strasser. EXACT: Algorithm and hardware architecture for an improved A-buffer. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 93)*, pages 85–91, Anaheim, CA, August 1993.