Stanford Tech Report - CSTR 2005-05 (2005), pp. 1–6

A Hardware F-Buffer Implementation

Mike Houston †, Arcot J. Preetham and Mark Segal ‡

Stanford University, ATI Technologies Inc.

Abstract

This paper describes the hardware F-Buffer implementation featured in the latest ATI graphics processors. We discuss the implementation choices made in each chip and the various implementation challenges faced like overflow handling. The F-Buffer was originally intended as a solution for multi-pass shading. We demonstrate this functionality, comparing it to traditional multi-pass rendering techniques, and show performance results. Given hardware F-Buffer support, we describe extended uses like order independent blending. We also show how a future F-Buffer implementation might be extended to allow more advanced operations like data filtering.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I.3.7 [Computer Graphics]: Color, shading, shadowing, and texture

1. Introduction

Multi-pass rendering is the traditional method for handling large shaders that need to be refactored into smaller shaders to fit within the resource limits of hardware. Even though hardware resources are increasing, there are still limits on the number of interpolants, program length, texture fetches, and available memory that still make it necessary to use multi-pass rendering for large shaders [CNS*02,FHH04,RLV*04]. However, using standard render-to-texture techniques to store intermediate values can create visible artifacts when shading transparent objects because overlapping fragments overwrite previously stored temporary values for that screen location. A rasterization order FIFO buffer, an F-Buffer, is an enhanced method of storing fragment data which assigns a unique storage location to each rendered fragment [MP01]. Using an F-Buffer for intermediate results allows for multi-pass rendering with correct transparency.

This paper presents the F-Buffer implementation available in ATI 9800 [ATI03b] and X800 [ATI04] series graphics processors. The primary contributions of this paper are:

- An explanation of the different implementation choices made in each generation of hardware and their ease of use.
- Exploration of functionality and features made possible by F-Buffer support.
- Proposed extensions to the current F-Buffer implementation to allow greater flexibility and performance.

2. Overview of F-Buffer

Mark and Proudfoot's introduction of the F-Buffer [MP01] presents a thorough explanation and overview of the basic functionality and implementation options of an F-Buffer. We briefly review the basic concepts here.

The F-Buffer provides an enhanced 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 include, for each output fragment, one or more RGBA colors, used for temporaries in intermediate passes. In subsequent passes, this stored data is read from the FIFO buffer as input data, where it is used for that pass's computations.

Generally, every rendering pass except the first reads from one or more F-Buffers. If a pass represents a leaf of the shade tree it does not need to read from an F-Buffer since there are no temporaries to be restored. Every shader pass except the last always writes to at least one F-Buffer. The last pass of the shader writes to the framebuffer as in normal rendering. Hardware that can simultaneously read two F-Buffers and write one F-Buffer is sufficient to render any shade tree

† mhouston@graphics.stanford.edu

‡ {preetham, segal}@ati.com

Standard Method F-Buffer

Figure 1: *Here we use a simple example to show the difference between traditional multi-pass rendering using the framebuffer and using an F-Buffer when rendering transparent objects. We draw a cyan triangle behind a yellow triangle, each with an opacity of 0.5. We use a two pass shader that saves off the triangle colors in the first pass and restores the colors in the second pass. As is shown, the traditional method restores the values of the last fragments drawn for both triangles yielding incorrect results.*

composed of binary operations, with more F-Buffer inputs supporting more complex shade trees.

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 storage location, if there are overlapping polygons rendered. An F-Buffer's association of each fragment with its own storage location eliminates the transparent-surface-rendering difficulties of conventional multi-pass rendering. With an F-Buffer, there is no longer a storage conflict between multiple fragments covering the same pixel, although partially-transparent surfaces must still be rendered in back-to-front order. See Figure 1.

F-Buffers use graphics memory more efficiently and flexibly than auxiliary framebuffers (deep framebuffers) would. An ideally sized 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. An example of the space savings of F-Buffer over using the framebuffer for storage can be seen in Figure 2. The reads and writes to 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.

3. Implementation

The original F-Buffer paper describes several possible hardware implementations. We discuss the F-Buffer implementation in the ATI 9800 and X800 series graphics processors and why certain design choices were made. The ATI 9800 series implements what is referred to as ATI Fbuffer in the marketing literature, and the X800 series has an improved F-Buffer implementation, mostly dealing with overflow handling, referred to as ATI Fbuffer2.

3.1. Common implementation

We discuss the common implementation choices between the two generations of processors. These choices are the core of the F-buffer implementation dealing with where and how F-Buffers are stored, rasterized to, and how previous passes' data are restored. Most of the design decisions were made to have minimum impact on the rest of the processor design, and to reuse as much of the standard pipeline functionality as possible.

F-Buffers are stored in graphics DRAM for processors with onboard memory, and in host memory for processors without onboard memory. The storage requirements for on-chip F-buffers are too large, and would therefore be too expensive, for the number of fragments many applications need to store. For example, storing a float4 value per fragment at 1024×1024 would require up to 16MB. Even worse, using multiple F-Buffer targets, the user could output to four float4 values through the use of multiple render targets, requiring up to 64MB. Similar to writing to a standard framebuffer, the latency of writing to an F-Buffer is hidden by the rendering parallelism of the hardware.

F-Buffers are treated in much the same way as standard 2D textures. At creation, F-buffers are defined to have a square size in powers of 2 (e.g. 32x32, 64x64,..., 2048x2048). This allows us to reuse all of the texture machinery already available in the driver and processor, allowing the use of standard texture lookups to restore values from the F-Buffer. When the F-buffer is in read mode, the address of the fragment in the F-Buffer is calculated from a global counter maintained by the scan converter and passed to the fragment shader via the fragment color interpolant. This value needs to be scaled by the size of the F-buffer being used. The user binds the previous F-buffer to one of the texture units and performs a texture lookup to restore the previous values. The disadvantage of this approach is that it adds the cost of one dependent texture lookup and requires the use of one interpolant (fragment color), but it provides a flexible method for the restoration of values and the use of multiple previously stored F-buffers, up to 16. An example of the output of a triangle rasterized to F-Buffers of different sizes is bound to texture and displayed is provided in Figure 2.

Polygons are rasterized on every pass. The original paper concentrates on a single rasterization approach, but this can drastically increase the amount of storage required for an F-Buffer and limit the shading flexibility. All interpolants (e.g. color, texture coordinates, etc) generated in the first pass would have to be stored if needed for subsequent passes. Since only the first pass can generate interpolants, the programmer is limited to the interpolants available. As noted

Figure 2: *This figure shows how an F-Buffer is laid out in memory for different sizes. The fragments are output into the F-Buffer in rasterization order. On the left is the rainbow triangle to be rasterized to a 512x512 window. For the 64x64 F-Buffer, we are only able to output the first 4096 fragments, the top corner of the triangle. For 128x128, we get a little bit further before we overflow the F-Buffer. At 256x256, we have successfully rasterized the triangle, and have a small amount of wasted space. At 512x512, we can easily fit all of the fragments into the F-Buffer. Most of the buffer is unused, corresponding to the amount of wasted space when using traditional multi-pass rendering.*

in [CNS*02, RLV*04, FHH04], a lack of interpolants in itself can create the need for multi-pass rendering. Rasterizing polygons on each pass requires less storage and enables more flexible use of interpolants. The disadvantage of this approach is that multi-pass shading a scene with many vertices can become bound by vertex processing instead of fragment processing. This will be explored in more detail in section 4.

Conventional framebuffer operations (depth test, alpha test, stencil, etc) are performed at the end of the last shader pass, when F-Buffers are only used as input. Therefore, each pass using an F-Buffer *must* generate the exact same order of fragments for the previous values to be restored correctly. All fragments that may be rasterized must be accounted for in the F-Buffer. This means that many operations that might normally mask a fragment output, e.g. using the KIL instruction in a fragment program, setting write masks, or using alpha/stencil/depth tests, are disabled when an F-Buffer is bound as output. However, the advantage of forcing consistent fragment generation is that the same F-Buffer can be used as both input and output since we can avoid read-modify-write hazards as we always read and write to the same location. This guarantee cannot be made if a user accesses an F-Buffer with general texture addressing.

3.2. Overflow Handling

As mentioned in the original F-Buffer paper, overflow handling is one of the more complex aspects of an F-buffer implementation. This is where the two generations of hardware differ in implementation.

With the 9800 series, the programmer creates an F-Buffer of a certain size and is responsible for handling overflow. The user can query the hardware to test whether overflow has occurred after their submitted geometry's fragments have committed to the F-Buffer. Fragments overflowing the F-Buffer generate undefined results. If overflow occurs, the user must restart the current shading pass by submitting a smaller batch of geometry. After the successful batch is fully shaded and output to the framebuffer, the next batch will need to be submitted for shading. Fortunately, since overflow can be detected at the end of the first shading pass, only one pass of wasted work can occur.

Although this is a functional solution, it puts a large burden on the user to achieve correctness and a large performance premium on overflow. To avoid overflow altogether, the user is forced to estimate how many fragments an object will generate and batch geometry accordingly. This may force the user to be overly conservative in their rendering unless they have intimate knowledge of the rasterization properties of the hardware. Encountering overflow with this implementation can lead to redundant computation and inefficiencies in shading.

The X800 series provides hardware overflow handling, which greatly eases the burden on the programmer. The hardware allows the F-buffer to fill up, and provides the user feedback that an overflow has occurred as well as the number of F-Buffers of the allocated size needed to handle the overflowing fragments. The overflow handling implementation provides a user-controlled fragment window. This window specifies the range of fragments, which is an F-buffer size number of fragments offset by multiples of the F-Buffer size, allowed to be written to the F-buffer. All fragments outside of the specified fragment window will be discarded early, before entering the fragment processors. For example, if the user defines a 32x32 F-buffer and generates 2048 fragments, this will overflow the buffer exactly one time. The user will shade the first 1024 fragments with their multi-pass shader, and the remaining 1024 fragments of overflow will be discarded. The user can then shift the F-buffer window by one, and the first 1024 fragments will be discarded and the second 1024 will be shaded. A longer example in pseudo code is available in Appendix A. This functionality allows for every fragment to be shaded only once, regardless of the amount

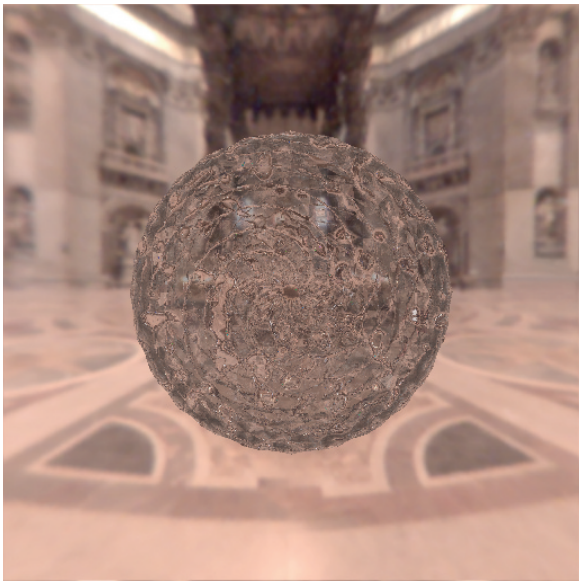

Figure 3: *A 3 pass pitted thin glass shader applied to a sphere and lit with the St. Peter's Basilica illumination data [Deb98] rendered to a 512x512 window using a 128x128 F-Buffer. Rendering this object requires 23 F-Buffer windows, or 69 passes total. Correct transparency is maintained because each fragment rendered gets its own storage in the F-Buffer during intermediate passes.*

of overflow, but may require the geometry to be submitted many times, once for each shading pass for each F-Buffer window.

4. Results

In this section, we will concentrate on the F-Buffer support in the X800 series and later hardware since they have a more flexible implementation. Using ASHLI [ATI03a], we have created a pitted glass shader that exceeds the resource limits available on all current hardware. This shader is a combination of the glass and stucco Renderman shaders. Using RDS [CNS*02], ASHLI chooses to split the shader into three rendering passes. Since this shader relies on blending and our test objects generate overlapping fragments, traditional multi-pass techniques using render-to-texture fail to shade the object correctly. As can be seen in Figure 3, because each fragment gets a unique storage location in the F-Buffer, correct shading is preserved. The difference in rendering transparent objects with an F-Buffer and traditional framebuffer rendering can best be seen with the simple example in Figure 1. Because of the implementation decisions made, the performance of rendering to F-Buffers with no overflow is equivalent to using traditional framebuffer methods.

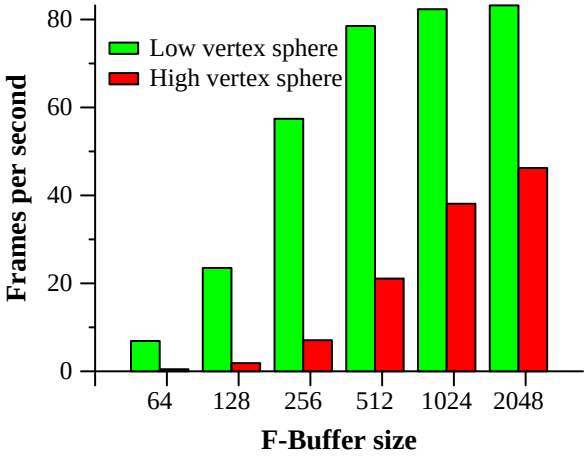

Figure 4: *Performance results from running the 3 pass pitted thin glass shader on a low vertex and high vertex count sphere of the same size with different F-Buffer sizes.*

Even though fragments outside of the current F-Buffer window are discarded before fragment shading, there is still the cost of sending the geometry through the vertex units each pass. For each F-Buffer window, we must resend the geometry. If the F-Buffer size is chosen small enough, geometry processing will dominate the cost of rendering each F-Buffer window. We expect the performance of shading to increase linearly with a reduction in the the number of F-Buffer windows required, until we become fragment processing bound, in which case performance should stay roughly the same since the same number of fragments are rendered regardless of F-Buffer window size. The effect of this can be seen in Figure 4 where we show the rendering performance of a simple and highly tessellated sphere, with 12,288 and 196,608 vertices respectively, rendered to a 1024x1024 framebuffer using different sized F-Buffers. We are using the same three pass shader as above for these tests. The low vertex sphere is vertex processing bound for small F-Buffer sizes, but becomes fragment processing bound for larger F-Buffer sizes. For the 64x64 F-Buffer, we overflow 162 times, so we must submit geometry a total of 489 times for our 3 pass shader. For the 256x256 F-Buffer, we overflow only 10 times and have to submit the geometry 33 times to complete our shading. In both cases, we shade the same total number of fragments, but the larger F-Buffer has *many* fewer vertices sent through the vertex processors to complete the shader. For the more tessellated sphere, we are heavily vertex bound and you can just start to see the performance curve knee over when using very large F-Buffers.

5. Discussion

F-Buffers provide an elegant mechanism to support multi-pass rendering while being able to maintain correct transparency. With compiler and runtime support for F-Buffer added to systems like RTSL [PMTH01] and ASHLI [ATI03a], very large shaders can be used with objects requiring transparency.

However, given hardware support for F-Buffer, there are other rendering problems that can be solved. In this section we talk about extended uses of F-Buffer for shading operations. We also explore several possible implementation changes to optimize F-Buffer support and to make it more flexible.

5.1. Extended shading operations

We can implement order-independent blending by storing the final fragment colors as well as their xyz screen space location into two separate F-Buffers. Instead of forcing the user to sort the geometry, we will use the F-Buffer to sort the fragments into the correct order for blending. Since we have xyz values and the final color for each fragment, we can sort the fragments to achieve order-independent blend. Using a stable sort, like bitonic sort, we can sort fragments with the same xy screen space location by z value. This can be done in $O(log^2 n)$ passes. When the F-Buffer is rebound and restored in the final pass, we generate the fragments in the correct order for blending. In the case of overflow, we have to store each of the overflows in separate F-Buffers and sort between and within each buffer, which is non-trivial and expensive.

There are many papers and researchers that have commented on the need for a method which uniquely stores each fragment rendered. The F-Buffer provides a solution to this need. Many algorithms traditionally relying on the sorting of geometry can be reimplemented by sorting the fragments stored in the F-Buffer after rendering has occurred. For example, unstructured volume rendering often relies on the visibility sorting of tetrahedral. Callahan et al. [CICS05] describe a k-buffer implementation to handle resorting fragments in the correct order, but it is limited to small number of overlapping fragments. Using an F-Buffer and sorting it similar to the above, it may be possible to handle larger values of k as well as to improve the efficiency of the algorithm. There are also interesting possibilities for CSG applications by modifying Goldfeather's algorithm [GMTF89] to make use of F-Buffers.

5.2. Hardware extensions

As previously discussed, the current implementation does not allow for some of the conventional framebuffer operations to be performed when using F-Buffer. We also do not allow late discard functions from preventing output to the F-Buffer. For example, if a shader executes a KIL instruction on a fragment, that fragment still needs to be accounted for in the output of the F-Buffer. If it were possible to prevent the fragments from being output to the buffer and still produce a FIFO, interesting data filtering operations could be performed. For example, if the user was generating a vertex array with the fragment shader, they could kill fragments to prevent certain vertices from being output into the buffer. The main difficulty in this addition is finding a way to keep the coherent output properties of F-Buffer in maintaining performance.

One of the performance issues not yet discussed with the current implementation is the inability to use early tests to prevent fragment generation. For example, if a very complex shader is applied to an object that is partially occluded, we would like to be able to use early-z tests to prevent the rendering of the occluded fragments. The difficulty with the current implementation is that the culling of fragments because of the F-Buffer window is done prior to the early discard units. It would be interesting to explore which early tests could be supported by F-Buffer. Any implementation must ensure that the same fragments *must* be generated for each pass, and if the output will not be included in the F-Buffer, how to maintain coherent output into the F-Buffer.

6. Conclusion

We have demonstrated the first available commodity hardware implementation of F-Buffer and discussed the various design choices made. With these design choices, F-Buffer support was able to be incorporated with minimal impact on the rest of the processor design. Using F-Buffer, multi-pass rendering with correct transparency can now be achieved with full hardware acceleration. We have also discussed several uses of F-Buffer for extended rendering tasks. Now that hardware accelerated support is available, we hope that the graphics community will explore other uses and extensions to F-Buffer.

Appendix A: F-Buffer Pseudo Code

```
DisplayLoop
  while remaining F-Buffer windows

    Set the F-Buffer window

    for pass 0 to 2

      if pass 0
        Attach F-Buffer1 to offscreen framebuffer
        Bind offscreen framebuffer

      if pass 1
        Attach F-Buffer2 to offscreen framebuffer
        Bind offscreen framebuffer
        Bind F-Buffer1 to texture

      if pass 2
```

```
    Bind normal framebuffer
    if first F-Buffer window
        Draw background objects
    Enable blending
    Enable test functions (depth,stencil,alpha)
    Bind F-Buffer2 to texture

Setup transforms
Bind vertex and fragment programs for pass
Render geometry

if first F-buffer window and pass is 0
    Get remaining number of F-Buffer windows to render
```

References

[ATI03a] ATI: ASHLI - advanced shading language interface, 2003. http://www.ati.com/developer/ashli.html. 4, 5

[ATI03b] ATI: Radeon 9800 technical specification, 2003. http://www.ati.com/products/radeon9800/radeon9800pro/specs.html. 1

[ATI04] ATI: Radeon x800 technical specification, 2004. http://www.ati.com/products/radeonx800/index.html. 1

[CICS05] CALLAHAN S. P., IKTIS M., COMBA J. L., SILVA C. T.: Hardware-assisted visibility sorting for unstructed volume rendering. In *IEEE Transactsions on Visualization and Computer Graphics* (may 2005), vol. 11. 5

[CNS*02] CHAN E., NG R., SEN P., PROUDFOOT K., HANRAHAN P.: Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Proceedings of the conference on Graphics hardware 2002* (2002), Eurographics Association, pp. 69–78. 1, 3, 4

[Deb98] DEBEVEC P.: Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of SIGGRAPH 98* (July 1998), Computer Graphics Proceedings, Annual Conference Series, pp. 189–198. 4

[FHH04] FOLEY T., HOUSTON M., HANRAHAN P.: Efficient partitioning of fragment shaders for multiple-output hardware. In *Graphics Hardware 2004* (Aug. 2004), pp. 45–53. 1, 3

[GMTF89] GOLDFEATHER J., MOLNAR S., TURK G., FUCHS H.: Near real-time csg rendering using tree normalization and geometric pruning. In *IEEE CG&A* (may 1989), vol. 9, pp. 20–28. 5

[MP01] MARK W. R., PROUDFOOT K.: The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (2001). 1

[PMTH01] PROUDFOOT K., MARK W. R., TZVETKOV S., HANRAHAN P.: A real-time procedural shading system for programmable graphics hardware. *ACM Transactions on Graphics* (August 2001). 5

[RLV*04] RIFFEL A., LEFOHN A. E., VIDIMCE K., LEONE M., OWENS J. D.: Mio: Fast multipass partitioning via priority-based instruction scheduling. In *Graphics Hardware 2004* (Aug. 2004), pp. 35–44. 1, 3