Front-to-Back Blending with Early Fragment Discarding

Adam Vlček*
Graph@FIT
Brno University of Technology

Jiří Havel†
Graph@FIT
Brno University of Technology

Adam Herout‡
Graph@FIT
Brno University of Technology

Abstract

Alpha-blending is a frequently used technique implemented in graphics hardware to achieve effects like transparency, volume rendering, displaying particle systems and others. A common approach to using blending is to sort objects from the furthest to closest (back-to-front) and render them in this order. However, the inverse order can be used with some small modifications to the blending process. This paper introduces an approach to use the front-to-back rendering order to discard fragments of rendered polygons that would not influence the visual result and thus speed the rendering up. The speed gain and visual compromises are measured in several scenarios simulating practical situations and the measured results are discussed and conclusions are drawn.

CR Categories: I.3.1 [COMPUTER GRAPHICS]: Hardware Architecture—Graphics processors; I.3.3 [COMPUTER GRAPHICS]: Picture/Image Generation—Display algorithms;

Keywords: blending, OpenGL, real-time, transparency

1 Introduction

When dealing with transparent objects in real-time computer graphics, hardware accelerated alpha-blending is usually used. While it is not suitable for simulating effects like refraction, it is still used for a wide variety of scene objects like glass and crystals, simple water surfaces, particle systems simulating dust, clouds, fog and fire [Cantlay 2007], smoothing the edges of vegetation leaves [Pelzer 2004], or drawing hair and fur [Nguyen and Donnelly 2005]. A particularly interesting use of alpha-blending is volume rendering. It is possible to render a volumetric object stored in a 3D texture by drawing a series of overlaid polygons using appropriate texturing coordinates slicing the volume and combining results for each slice by the blending operations [Ikits et al. 2004]. All such techniques are highly demanding on per-fragment operations and can easily become bottlenecks.

The classical approach to alpha-blending, using for example OpenGL, is to sort polygons in back-to-front order and draw all of them successively with enabled blending and correctly set blending factors. While producing good visual results and being considered the "standard way", this approach has several disadvantages [Akenine-Möller et al. 2008].

*e-mail: ivlcek@fit.vutbr.cz
†e-mail: ihavel@fit.vutbr.cz
‡e-mail: herout@fit.vutbr.cz

The main concern of this paper is the fact that in many cases it is being drawn more than it is really needed. For example, volume rendering of a human head showing the skull by slicing the volume data [Krüger and Westermann 2003] would produce a lot of fragments which will not be visible in the final image. To determine the visibility and thus spare some amount of drawing we must render in the opposite direction, front-to-back. While being slightly more complicated and counter-intuitive, this approach is used in some works today [Ikits et al. 2004] [Krüger and Westermann 2003]. Now the rendering system can determine when the opacity reaches such a level that no additional rendered fragments might change the visual result. The problem is, how to make such decision effectively on current graphical hardware and how to prevent the fragments from being drawn.

Since the opacity can vary on a per-fragment basis thanks to texturing, it is usually impossible to determine occluded regions before the rasterization stage. One approach to preventing the fragments from being drawn is using the depth buffer by inserting an invisible polygon covering the whole screen which only writes into the depth-buffer, once for several semi-transparent polygons drawn by the system [Krüger and Westermann 2003]. These invisible helper polygons constitute a notable overhead and the program gets complicated by them. In our approach presented in this paper we utilize reading from an alpha texture which is also bound as a render target at the same time. This produces some nondeterministic behaviour due to incoherence of caches, but we demonstrate that when used properly and with some help from a recent OpenGL extension, it can still serve well to speed up rendering.

The following Section 2 summarizes the algorithm of front-to-back alpha blending and presents the new mechanism for fragment discarding based on an auxiliary texture rendered into. Section 3 presents measurements of the speed and visual quality of the algorithm. Based on the measurements, Section 4 discusses the potential of the algorithm, its advantages and drawbacks, and speculates about possible small changes to the graphics hardware which would enable using the presented technique even more efficiently without further complicating the hardware. Section 5 concludes the paper and states some hints for future work.

2 Front-to-Back Blending Algorithm

Transparent surfaces are usually blended from back to front as described by Equation (1)

$$\mathbf{C} = \alpha_n\mathbf{C}_n + (1-\alpha_n)(\ldots(\alpha_1\mathbf{C}_1 + (1-\alpha_1)\alpha_0\mathbf{C}_0)\ldots), \quad (1)$$

where $\mathbf{C}$ is the framebuffer RGB colour, $\mathbf{C}_0, \ldots, \mathbf{C}_n$ are RGB fragments ordered from the furthest ($\mathbf{C}_0$) to the closest ($\mathbf{C}_n$) to the camera, and $\alpha_0, \ldots, \alpha_n$ are alpha values associated with the fragments.

Every blended pixel updates the value in the framebuffer so

$$\mathbf{C}' = \alpha_n\mathbf{C}_n + (1-\alpha_n)\mathbf{C}; \quad (2)$$

$\mathbf{C}'$ replaces the previous value $\mathbf{C}$. In OpenGL, the command for proper pipeline setting is `glBlendFunc(GL_SRC_ALPHA,` GL_ONE_MINUS_SRC_ALPHA). This simple blending can be modified using premultiplied alpha [Smith 1995] and inverting the alpha value itself (Equation (3)):

$$\begin{aligned} \mathbf{C}^* &= \alpha\mathbf{C} \\ \alpha^* &= 1-\alpha. \end{aligned} \tag{3}$$

Using premultiplied alpha has some advantages over classical alpha. Probably the most important one is that when using premultiplied alpha, the blending is associative. This property simplifies building of impostors [Forsyth 2001; Risser 2007] and is crucial for front-to-back blending. Fully transparent premultiplied colours also "do not have any colour", so linear interpolation between solid and fully transparent pixel does not alter the colour of the solid one. Inverting of the alpha value is not necessary, but it simplifies the expressions and our experiments show that it also slightly improves the rendering performance. Equation (4) shows the blending equation using premultiplied alpha.

$$\begin{aligned} \mathbf{C} &= \mathbf{C}_n^* + \alpha_n^*(\mathbf{C}_{n-1}^* + \alpha_{n-1}^*(\ldots(\mathbf{C}_1^* + \alpha_1^*\mathbf{C}_0^*)\ldots)) \\ &= \mathbf{C}_n^* + \alpha_n^*\mathbf{C}_{n-1}^* + \ldots + \alpha_n^*\alpha_{n-1}^*\ldots\alpha_1^*\mathbf{C}_0^* \end{aligned} \tag{4}$$

The expanded form in Equation (4) clearly shows the associativity, as every two neighbouring layers can be merged together using

$$\begin{aligned} \mathbf{C}^* &= \mathbf{C}_2^* + \alpha_2^*\mathbf{C}_1^* \\ \alpha^* &= \alpha_1^*\alpha_2^*. \end{aligned} \tag{5}$$

This is also the principle of front-to-back blending, as every transparent layer is merged with the framebuffer this way. In contrast with Equation (1), the transparent layers can be merged in both directions, so front-to-back blending is now possible, but the classical back-to-front order can be used, too. In contrast to back-to-front blending, the framebuffer must also contain the accumulated alpha value. The colours and alphas are mixed by different factors so in OpenGL, `glBlendFuncSeparate` must be used. The factors for colour component are GL_DST_ALPHA and GL_ONE. The factors for the alpha component can be either GL_DST_ALPHA, GL_ZERO, or GL_ZERO, GL_SRC_ALPHA.

When rendering both opaque and transparent objects in one scene, the classical approach is:

1. Render all solid objects, if possible in front-to-back order to minimize overdraw.
2. Set up blending, disable depth write.
3. Render transparent surfaces in back-to-front order.

With front-to-back blending, the process is slightly more complicated:

1. Render all solid objects.
2. Set up blending. Use the depth-buffer from step 1. Set the colour to be rendered to RGBA texture. The initial values in this texture are 0 for RGB and 1 for alpha.
3. Render transparent objects front-to-back to the texture with blending factors from the previous paragraph.
4. Blend the filled texture to the framebuffer. Multiply the destination value with texture alpha and then add texture value (`glBlendFunc(GL_ONE, GL_SRC_ALPHA);`).

This process can also be perceived as construction of one large impostor.

Switching framebuffer objects between texture rendering and classical framebuffer takes some time as well as the final blending of rendered texture over the opaque objects. However, the speedup achieved by the front-to-back rendering overcomes these overheads and in most tests, the front-to-back approach was still performing slightly faster than back-to-front, and the rendering to texture has one important advantage which we are exploring in the next section.

2.1 Early Alpha-Test

During the front-to-back rendering into a texture, the texture can be easily mapped as a sampler input of fragment shaders, which can use the accumulated alpha to determine whether the surfaces closer to the camera make a composition so opaque, that blending of further fragments will not change the result visibly. OpenGL does not prevent such usage of textures but the results of such an action are undefined. In practice, this means that it is not guaranteed for the values written by one fragment to be immediately visible for a succeeding one. This is due to cache incoherency and it usually takes some time to propagate the new values from the blending circuits back to the cache of the rendered texture.

As the texture cache cannot hold all the texturing data, it gets updated and the new values are being read. Luckily, this happens quickly enough to be useful for determination of opaque areas during the rendering. In fact, later in the frame, caches are becoming more useful as larger areas are being marked opaque – there are less texture fetches for actual texturing and larger parts of unchanged framebuffer can stay in the cache.

In our approach to discarding fragments during front-to-back blending, such cache incoherencies are not very harmful. If the opacity accumulated in the framebuffer exceeds a threshold so that all succeeding fragments are discarded, but the cache returns older value and causes the shader to continue, the result is that some fragment processing is done that is not necessary. However, with sufficiently low alpha-threshold, no visible difference will be in the final image thanks to the correct blending phase, which works exactly the same as for pure front-to-back rendering. With the threshold set unreasonably high, image appearance will start to change and later artifacts will start to pop out, usually forming a pattern of rectangles within drawn triangles, as seen in Figure 2.

The only real problem is the initialization of the alpha texture. Simple `glClear` sets all texels to specified values, but the old values might stay in caches, which occurs frequently and produces artifacts shown in Figure 2. Simple `glFlush` or `glFinish` does not help. NVIDIA recently released an OpenGL extension `NV_texture_barrier` [Bolz 2009], which is designed exactly for the purpose of invalidating all texture caches. However, it is not widely supported. It was also possible to acquire the same results in a very similar time by simply switching the framebuffer object after clearing the texture by `glClear` back to the default framebuffer (`glBindFramebuffer(GL_FRAMEBUFFER, 0)`) and then immediately back to the texture rendering one.

In scenes with a heavy overdraw and where high occlusion is expected, multiple render targets can be used to render the image into one RGBA texture and use one additional texture with just a single alpha channel for reading back into the shader. This means that more blending computation needs to be done and slightly more data written when the fragment passes, but it also means that only $1/4$ of data need to be read.

2.2 GLSL Shader Design

Suppose that the usual work of a given fragment shader is done by function `RealWork` which returns the final colour in a 4-component RGBA vector. The fragment shader code for back-to-front blending can look like this

```
void main(void)
{
  gl_FragColour = RealWork();
}
```

while the front-to-back alternative has to add the blending precomputation mathematics (3)

```
void main(void)
{
  vec4 rw = clamp(RealWork(), 0.0, 1.0);
  rw.rgb *= rw.a;
  rw.a = 1.0 - rw.a;
  gl_FragColour = rw;
}
```

Notice the `clamp` function to restrict all RGBA values to the interval $\langle 0,1\rangle$. Values outside this range could produce very different results from back-to-front blending, because similar clamping is performed automatically when colour is exported from the shader `gl_FragColour` variable into the blending stage, but in this case a part of the blending is performed before any automatic modifications.

The alpha discarding shader simply adds the testing for alpha read back from a 2D texture sampler `sAlpha` on the screen location passed from vertex shader in 2D vector `vScreen`. The acquired value is compared to the uniform float `uAlphaThresh` and fragments evaluated as unimportant are discarded.

```
void main(void)
{
  if(texture2D(sAlpha, vScreen).a < uAlphaThresh)
    discard;
  else{
    vec4 rw = clamp(RealWork(), 0.0, 1.0);
    rw.rgb *= rw.a;
    rw.a = 1.0 - rw.a;
    gl_FragColour = rw;
  }
}
```

An interesting detail is that sometimes the `if-else` construction produces significantly faster shaders than simple `if-discard` on hardware used for testing.

2.3 Sources of Speedup

Since we rely on the GLSL `discard` command in the fragment shader, the speedup is limited because the shader has to be started for each fragment. Section 4 discusses possibilities of using techniques based on early Z-test or similar, but the version immediately usable on current hardware cannot count with them. The presented solution, however, brings interesting speedups; their sources are explained in the following paragraphs.

To understand the potential speedup and its limits, it is necessary to use information about the way current hardware processes fragments. A good description for NVIDIA hardware can be found in materials describing the CUDA architecture [NVIDIA 2009]. The basic idea is running a group of threads for adjacent pixels called a *warp*, in which all threads execute the same code (*Single Instruction Multiple Threads*, SIMT) and all threads within the warp finish at the same time. In case of branching, the different execution paths are serialized, increasing the total instruction count for the whole warp. Sometimes the compiler performs branch prediction and loop unrolling, in which case no thread can diverge, and each instruction is associated with a predicate signalling the thread's condition. Instructions with false predicate are scheduled for execution, but they do not write results, evaluate addresses or read operands.

The sources of speedup are:

1. More favourable blending factors.
2. Discarded fragment is not required to be blended, which relieves memory bandwidth and the blending hardware. Even a simple shader assigning only constant colour showed noticeable speedup on a heavily overdrawn scene.
3. Discarded fragment shader thread is doing all the work as the longest running thread in the warp. But very expensive texture fetches from global memory are not executed, which relieves memory bandwidth and texturing units. The best speedup therefore occurs for memory intensive shaders, but depending on the compiled shader structure there can be other benefits.
4. If the whole warp is discarded, execution stops and resources can be used for another piece of data. This means that scenes with lower spatial frequencies in opacity will perform better.

The main limitations:

1. If only a single fragment of the warp passes the early alpha-test, all the resting threads in the warp are idle, potentially wasting a large portion of the computational power. However, there is still potential for speedup as described in item 3 above.
2. A simple test could be performed before scheduling individual fragments for shading, which could produce much higher speedup; however, using early Z-test is currently rather unpractical.

3 Measured Results

This section reports the measurements performed to describe the influence of the presented algorithm on the speed of rendering. Section 3.1 reports measurements targeted on the speed of the front-to-back blending algorithm with and without fragment discarding. Section 3.2 shows comparisons of the resulting frames rendered by different methods to report the quality of the rendering with fragment discarding. Section 3.3 measures the overhead related to the fragment discarding caused by rendering to a texture.

All experiments were conducted in a window of 512×512 pixels. Textures are 2D squares with real-life contents (photographs of leaves with transparency, smoke clouds for particle systems, etc.), using trilinear interpolation.

Throughout the tests, the following abbreviations are used for different methods experimented with:

B2F – back-to-front (classical) rendering

F2B – front-to-back rendering with precomputed alpha

F2BD – front-to-back with fragment discarding

F2BX – front-to-back with fragment discarding, maximal efficiency

The F2BX tests estimate the maximal achievable speedup by fragment discarding by clearing the alpha texture to maximal opacity ($\alpha^* = 0$), which means all fragments are discarded. This estimate is really the upper bound not achievable in real situations – real situations should be covered by the F2BD test, which is designed to be as realistic as possible, but the nature of used textures can shift results in both ways.

The tested polygons were axis-aligned quads parallel to the projection plane with different relative size to the screen and with random position. The whole quad is always visible on the screen. The polygon generation methods were as follows:

Random – random position, random size ($26 \sim 256$),

Small – random position, size $30\% \times 512 = 154$,

Big – random position, size $60\% \times 512 = 307$, and

Full – quad occupies always the whole screen, i.e. the same position.

The described algorithm was tested mainly (and the results are measured) on NVIDIA GeForce 9800 GTX+ with 1 GB, CPU Intel Core2 Quad Q9650 @ 3.0 GHz, 4 GB RAM, OS Microsoft Windows Vista. Similar results were measured also on Intel Core2 Duo E8300 @ 2.83 GHz, 2GB RAM, WinXP, with different graphics cards: NVIDIA GeForce 8600GT, NVIDIA GeForce GTX295, and ATI Radeon HD 4770, 512MB, DDR5.

3.1 Speed: Back-to-Front vs. Front-to-Back vs. Front-to-Back with Fragment Discarding

The speeds are highly dependent on the shader structure, the sizes of textures, the scene structure, and most of all on the distribution of opacity in textures. The use of caches has a significant impact on performance. Since we use a texture for reading back the alpha value, the size of the viewport has similar influence as texture sizes – the bigger it is, the worse is the utilization of caches. Shaders with intensive reading from large textures and a minimal amount of mathematics have better potential for speedup when most of the fragments of the warp are discarded. The best results are achieved when the variance in opacity is low over the whole warp of threads. When all shader threads within a warp are terminated, even a very complex shader with a high amount of computation will benefit significantly.

First, Table 1 reports the speedup compared to the common back-to-front approach, depending on the complexity of work done in the fragment shader. An extremely simple shader (constant colour) was tested, followed by shaders applying one or more textures and including some expensive mathematical functions, such as vector arithmetic or using goniometric functions for modifications of all sampled positions in the textures. Note that for example “4 textures” means reading 4 values from 4 different textures, while for example “4×4 textures” means reading 16 values from 4 different textures, 4 from each.

The results show that the front-to-back approach is faster even without any fragment discarding (see line “Constant colour” in the table). This can be explained by the mathematics of precomputed alpha. Equation (5) indicates that some value ($\alpha_2^* \mathbf{C}_1^*$) is added to the framebuffer contents ($\mathbf{C}_2^*$) while in the back-to-front case (Equation (2)), the contents of the framebuffer $\mathbf{C}$ must be read to the computational core, altered by multiplication and addition and written back. The speedup by mere front-to-back rendering – despite its computational overhead related to alpha pre-multiplication ($\alpha^* = \alpha_1^* \alpha_2^*$ in (5)) – indicates that current graphics chips support addition to frame-buffer as a single operation

Shader Type	Texture Size	F2B	F2BD	F2BX
Constant colour	—	1.03	1.88	1.90
1 texture	64	1.02	1.43	1.57
	1024	1.21	2.49	2.81
4 textures	64	0.96	1.44	1.71
	1024	1.15	6.03	9.38
4 textures + math	64	0.99	1.47	1.74
	1024	1.16	5.93	9.31
4×4 textures	64	0.95	3.45	5.06
	1024	1.11	15.16	39.39
4×4 textures + math	64	0.98	3.40	4.80
	1024	1.10	16.68	41.55

Table 1: Results for a random combination of 8192 randomly sized quads.

Note that for a small texture size (64×64), the front-to-back approach without discarding performs slightly worse than the back-to-front one. This should be explained by texture caching: when the texture cache is used in every single pass of the fragment shader, the texture contents remain in the cache, while when some fragments do not use the texture, the cache can be reused for another purpose. In case of front-to-back rendering, the texture cache is heavily used for the alpha texture which serves as an extended framebuffer. For larger textures (1024×1024) the texture does not fit into the cache and front-to-back rendering is always faster. Front-to-back rendering is also always faster than back-to-front when the proposed fragment discarding is engaged (for reasonably large amount of data rendered – see Table 4, one quad drawn).

Table 2 reports the speedup gained by different versions of the algorithm as it depends on the size of the textured polygons.

Distribution	F2B	F2BD	F2BX
Random	1.11	15.16	39.39
Small	1.07	26.15	83.61
Big	1.08	27.52	83.75
Full	1.14	2.58	12.15

Table 2: Speedup depending on the scene structure. Drawing 8192 quads with different distributions; 4×4 texture shader, texture size 1024×1024.

The next experiment was designed to investigate the influence of overlapping the same amount of equally sized quads. Three scene setups are presented: Thick, Medium, Scarce. All the setups are basically the Small quads with different placement randomness from the screen center: Thick: ± 0.1, Medium: ± 0.2, Scarce: ± 0.35 (full screen coverage).

Distribution	F2B	F2BD	F2BX
Thick	1.091	5.37	15.09
Medium	1.087	5.20	15.09
Scarce	1.078	3.95	15.09

Table 3: Speedup depending on the spatial density of the drawn primitives. 1024 quads, texture size 512×512.

Table 4 reports the speedups for the “Big” generation scheme for different number of primitives drawn.

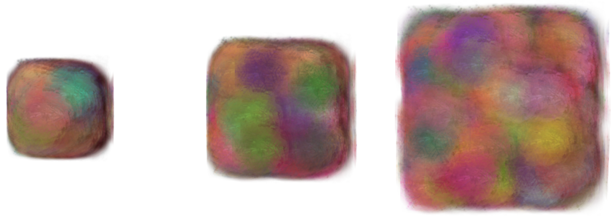

Figure 1: Three scene setups with the same amount of produced fragments: Thick, Medium and Scarce. All composed of 1024 Small quads.

Quad count	F2B	F2BD	F2BX
1	0.65	0.64	0.98
16	0.94	1.06	6.35
64	1.02	1.86	12.23
256	1.07	3.50	15.17
1024	1.10	6.39	16.42
8192	1.11	12.03	17.97

Table 4: Speed for various levels of overdraw. “Big” quads, 4×4 texture shader, texture size 1024×1024.

Table 5 reports the variation of speedup based on the value of alpha-threshold.

α_{thresh}	s-up	Comment
0.0	0.89	All drawn, alpha texture overhead
0.01	2.33	Practically no visual differnece to F2B
0.02	2.56	Roughly optimal
0.05	2.87	Small change in colours, artifacts hard to spot
0.25	3.78	Image structure changes, artifacts well visible
0.5	4.58	Totally different scene, heavy artifacts
0.9	5.82	Even worse
1.0	6.51	Only alpha equal to 1 passes, heavy artifacts
F2BX	8.00	All discarded

Table 5: Speedups for various alpha-thresholds. 1024 “Random” quads, 4×4 texture shader, texture size 1024×1024.

3.2 Quality: A Good Case and a Bad Case

Images generated by back-to-front rendering can be slightly different from those generated in front-to-back manner. This is caused by performing different mathematical operations over data types with limited precision. However, it is not possible to tell which case is the correct one and the differences usually are rather hard to identify by a naked eye as it is shown in Figure 2.

If used incorrectly, higher alpha-thresholds can also introduce additional distortions similar to alpha testing. Higher threshold will also almost certainly introduce cache incoherency errors similar to clearing render targets without the succeeding cache synchronization. Cache artifacts are especially disturbing for their rapid flickering due to nondeterministic manifestation in each frame. Such cases are illustrated in Figure 3.

Table 6 gives some information about the differences of the “common” back-to-front blending to our front-to-back with fragment discarding for various alpha-thresholds. The given values are for

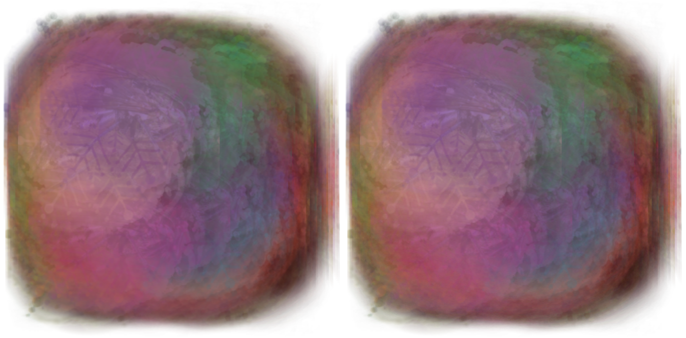

Figure 2: A comparison of the classical back-to-front (left) to our method (right). In most cases the results are nearly identical. This scene consists of 512 “Big” quads with 4×4 shader, 512×512 textures, alpha-threshold 0.02.

the scene from Figure 2. The difference $B2F - F2BD$ for each component of each pixel was computed.

α_{thresh}	Average	Min	Max
0.0	1.52	-8	15
0.01	1.36	-9	15
0.02	1.88	-10	15
0.25	20.7	-56	15
0.5	43.0	-113	14
0.9	80.8	-202	7
1.0	94.1	-226	5

Table 6: Differences of back-to-front to front-to-back with discarding. **Average** is the average absolute difference through all RGB components of all pixels. **Min** is the biggest negative difference and **Max** is the biggest positive.

3.3 Overhead: Time Wasted by Rendering to Alpha Texture

Due to the rendering to a texture, the initialization and finalization of front-to-back rendering introduces some overhead over the back-to-front as discussed in Section 2. It is necessary to switch the framebuffer objects, clear the additional texture (initialization), and blend the result over opaque objects (finalization). These overheads are reported in Table 7 in microseconds.

Method	Initialization	Finalization
B2F	13-32	0.81
F2B	80-100	120-600

Table 7: Time spent in extra stages of the rendering [μs].

Although measured by the `EXT_timer_query` extension, which should measure exactly the time spent by enclosed OpenGL commands, the times spent by initialization and finalization parts showed high variability for different shaders and frames. The most constant time is the finalization of back-to-front, which simply disables rendering arrays, shader, and blending. On the other hand, finalization of front-to-back with or without discarding appears to be the wildest part. The blending to backbuffer is roughly comparable with the time required to draw a single non-occluded “Full” quad with a simple shader.

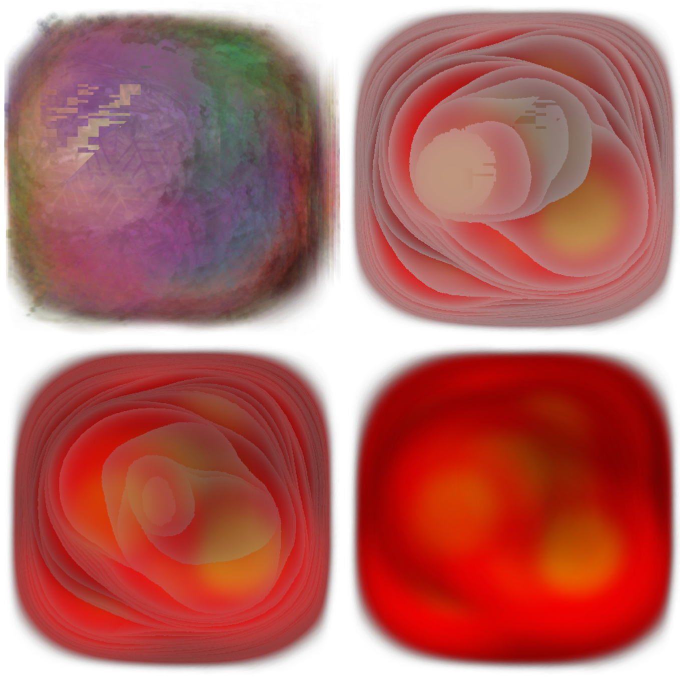

Figure 3: Problems and errors. **top-left**: Scene from Figure 1 rendered without cache synchronization by `NV_texture_barrier` showing artifacts. **top-right**: a different scene "red" rendered with too high alpha-threshold 0.5 showing incorrect image with sharp edges, and cache artifacts. **bottom-left**: "red" scene with alpha-threshold 0.25 still showing incorrect image lesser chance for visible cache artifacts. **bottom-right**: "red" scene with aplha-threshold 0.01 showing the correct image.

4 Discussion: Early Alpha-Test Wanted

Even for optimal conditions with all threads within a warp terminated, the speedup is still limited by the need to execute the fragment shader for each fragment. This could be avoided by using early Z-test (early depth-test). Early Z-test is implemented in all contemporary graphics chips and speeds up rendering notably by discarding invisible fragments without executing the fragment shader or performing other fragment operations and tests. Early Z-test is automatically disabled for fragment shaders that alter the depth of the fragment. This is correct for general rendering because such combination would result in unpredictable behaviour and visibility artifacts.

One approach is to insert depth modifying polygons always after a certain number of transparent ones [Krüger and Westermann 2003]. The speedup for favourable scenes is good but the potential overhead of additional geometry, significant amount of additional processed fragments without early Z-test, and frequent switching of the pipeline state are obvious.

A better solution would be to allow modification of the depth in the fragment shader and keep the early Z-test on – "risky early Z-test". In our approach there is no need to run the shader once the fragment is marked as occluded because the visual impact of such fragment is none or negligible; the visual artifacts would thus be eliminated. Such use of the early Z-test would not suffer even from any problems caused by hardware optimization and Z-buffer caching. Similarly to the texture incoherency mentioned in Section 2.1 and Section 3.2, any incoherencies in the Z-buffer would not influence the visual quality visibly.

A theoretical alternative to having the "risky early Z-test" could be an early α-test. In contrast to simply turning the "risky" mode of the already existing early Z-test, the early α-test would have to be newly implemented in the graphics hardware or firmware, which is not justified by the relatively limited use of the presented algorithm.

5 Conclusions

This article proposes an approach to alpha-blending on graphics hardware where transparent objects are rendered in front-to-back order using a RGBA texture as a rendering target. This way, the alpha value in the frame-buffer can be used for early fragment discarding.

Measurements show that the proposed approach speeds up rendering even without fragment discarding. With fragment discarding operational, the speedups are attractive with invisible image distortions. Since the speedup is achieved by skipping fragment shader's code, usage of the presented algorithm is desirable especially for complicated shaders with computations and multiple accesses to textures. Although complicated shaders are speeded up the most, the algorithm introduces some speedup even for simplistic shaders.

Our approach would benefit from a "risky early Z-test", i.e. early Z-test operational even for a fragment shader that alters the depth value. Though generally, early Z-test on such fragments can cause indeterministic image distortions, in the case of our algorithm it would behave predictably and correctly. As future work, we intend to experiment with open-source drivers for graphics chips to turn the "risky early Z-test" on and potentially propose such OpenGL extension.

Among the applications benefiting the most from the presented approach would certainly be volume rendering, which is often done by rendering thousands of slices through a 3D texture, where significant amount of fragments could be occluded for example by a bone. Another area could be particle systems, but relatively simple shaders are usually used there, so the speedup without early Z-test would be limited. Good results could be also achieved for all overlapping impostors like vegetation and distant objects.

Acknowledgements

This work was partially supported by the BUT FIT grant FIT-10-S-2, by the research program LC-06008 (Center for Computer Graphics), and by the research project "Security-Oriented Research in Information Technology" CEZMSMT, MSM0021630528.

References

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

BOLZ, J., 2009. NV_texture_barrier. OpenGL extension, August. URL: http://developer.download.nvidia.com/opengl/specs/GL_NV_texture_barrier.txt.

CANTLAY, I. 2007. High-speed, off-screen particles. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, ch. 23, 513–528.

FORSYTH, T. 2001. Impostors: Adding clutter. In *Game Programming Gems 2*, M. DeLoura, Ed. Charles River Media, Inc., ch. 5.7, 488–496.

IKITS, M., KNISS, J., LEFOHN, A., AND HANSEN, C. 2004. Volume rendering techniques. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, R. Fernando, Ed. Addison-Wesley, ch. 39.

KRÜGER, J., AND WESTERMANN, R. 2003. Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, IEEE Computer Society, Washington, DC, USA, 38.

NGUYEN, H., AND DONNELLY, W. 2005. Hair animation and rendering in the nalu demo. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, Ed. Addison-Wesley, ch. 23, 361–380.

NVIDIA, Ed. 2009. *NVIDIA CUDA C Programming Best Practices Guide*. NVIDIA Corporation. URL: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf.

PELZER, K. 2004. Rendering countless blades of waving grass. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, R. Fernando, Ed. Addison-Wesley, ch. 7, 107–121.

RISSER, E. 2007. True impostors. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, ch. 21, 481–490.

SEGAL, M., AND AKELEY, K. 2009. *The OpenGL Graphics System: A Specification (Version 3.2, Core Profile).*

SMITH, A. R. 1995. Image compositing fundamentals. Tech. rep.