Eurographics Symposium on Rendering (2006), pp. 1–12
Tomas Akenine-Möller and Wolfgang Heidrich (Editors)

An Efficient Multi-View Rasterization Architecture

Anonymous

Abstract

Autostereoscopic displays with multiple views provide a true three-dimensional experience, and full systems for 3D TV have been designed and built. However, these displays have received relatively little attention in the context of real-time computer graphics. We present a novel rasterization architecture that rasterizes each triangle to multiple views simultaneously. When determining which tile in which view to rasterize next, we use an efficiency measure that estimates which tile is expected to get the most hits in the texture cache. Once that tile has been rasterized, the efficiency measure is updated, and a new tile and view are selected. Our traversal algorithm provides significant reductions in the amount of texture fetches, and bandwidth gains on the order of a magnitude have been observed. We also present an approximate rasterization algorithm that avoids pixel shader evaluations for a substantial amount (up to 95*%) of fragments and still maintains high image quality.*

1. Introduction

The next revolution for television is likely to be 3D TV [MP04], where a *multi-view autostereoscopic* display [Dod05, JFO02] is used to create a true three-dimensional experience. Such displays can be viewed from several different viewpoints, and thus provide motion parallax. Furthermore, the viewers can see stereoscopic images at each viewpoint, which means that binocular parallax is achieved. Displays capable of providing binocular parallax without the need for special glasses are called *autostereoscopic*. This is in contrast to ordinary displays, where a 3D scene is projected to a flat 2D surface.

Analogously, for real-time graphics, the next revolution might very well be the use of autostereoscopic multi-view displays for rendering. Possible uses are scientific & medical visualization, user interfaces & window managers, advertising, and games, to name a few. Another area of potential great impact is stereo displays for mobile phones, and companies such as Casio and Samsung have already announced such displays.

Stereo is the simplest case of multi-view rendering, and APIs such as OpenGL [SA92], have had support for this since 1992. To accelerate rendering for stereo, a few approximate techniques have been suggested [PK96, SHS00]. Furthermore, efficient algorithms for stereo volume rendering [AH94, HK96] and ray tracing [AH93] have been proposed. The PixelView hardware architecture [SBM04] is used to compute and visualize a four-dimensional ray buffer, which is essentially a lumigraph or light field. The drawback is the expensive computation of the ray buffer, which makes supporting animated scenes difficult.

Surprisingly, to the best of our knowledge, only one research paper exists on rasterization for multiple viewpoints. Halle [Hal98] presents a method for multiple viewpoint rendering on existing graphics hardware, by rendering polygons as a multitude of lines in epipolar plane images. His system and ours can be seen as complementary, since his algorithm works well for hundreds of views, but breaks down for few views (<10). Our algorithms, on the other hand, have been designed for few views ($\leq$ 16). Another difference is that we target a new hardware implementation, instead of using existing hardware.

The inherent difficulty in rendering from multiple views is that rasterization for n views tends to cost n times as much as a single view. For example, for stereo rendering the cost is expected to be twice as expensive as rendering a single view [Ake03]. Rendering to a display with, say, 16 views [MP04] would put an enormous amount of pressure on even the most powerful graphics cards of today. However, since the different viewpoints are relatively close to each other, the coherency between images from different views can potentially be exploited for much more efficient rendering, and this is the main purpose of our multi-view rasterization architecture.

Our architecture is orthogonal to a wide range of bandwidth reducing algorithms, such as texture compression [BAC96, KSKS96], texture caching [HG97, IEH99], prefetching [IEP98], color compression [SMM*04], depth compression & Z-max-culling [Mor00], Z-min-culling [AMS03], and delay streams [AMN03]. In addition to using such techniques, our architecture directly exploits the inherent coherency when rendering from multiple views (including stereo) by using a novel triangle traversal algorithm. Our results show that our architecture can provide

significant reductions in the amount of texture fetches when rendering exact images. We also present an algorithm that approximates pixel shader evaluations from nearby views. This algorithm generates high quality images, and avoids execution of entire pixel shaders for a large amount of the fragments.

2. Motivation

Here we will argue that texturing is very likely to be the dominating cost in terms of memory accesses, now and in the future. For this, we use some simple formulae for predicting bandwidth usage for rasterization. Given a scene with average depth complexity d, the average overdraw, $o(d)$, is estimated as [CH93]:

$$o(d) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{d}. \quad (1)$$

Without using any bandwidth reducing algorithms, the bandwidth required by a single pixel is [AMS03]:

$$b = d \times Z_r + o(d) \times (Z_w + C_w + T_r), \quad (2)$$

where Z_r and Z_w are the cost, in bytes, for reading and writing a depth buffer value, respectively. Furthermore, C_w is the cost for writing to the color buffer (assuming no blending is done, since the term C_r is missing) and T_r is the total cost for accessing textures for a fragment. Standard values for these are: $Z_r = Z_w = C_w = 4$ bytes and a texel is often stored in 4 bytes. Trilinear mipmapping [Wil83] is commonly used for reducing aliasing, and this requires eight texel accesses, which makes $T_r = 8 \times 4 = 32$ bytes for a filtered color from one texture. If a texture cache [HG97] with miss rate, m, is used, and we have a multi-view display system with n views, Equation 2 becomes:

$$\begin{aligned} b(n) &= n \times [d \times Z_r + o(d) \times (Z_w + C_w + m \times T_r)] \\ &= n \times [\underbrace{d \times Z_r + o(d) \times Z_w}_{\text{depth buffer, } B_d} + \underbrace{o(d) \times C_w}_{\text{color buffer, } B_c} + \underbrace{o(d) \times m \times T_r}_{\text{texture read, } B_t}] \\ &= n \times [B_d + B_c + B_t] \end{aligned} \quad (3)$$

Now, we analyze the different terms in this equation by looking at two existing example shaders (not written by the authors). First, we assume the scene has a depth complexity of $d = 4$ ($\Rightarrow o \approx 2$). This is quite reasonable, since occlusion culling algorithms and spatial data structures (e.g. portals) efficiently reduce depth complexity to such low numbers, or even lower. For example, the recent game S.T.A.L.K.E.R [Shi05] has a target depth complexity of $d = 2.5$. Using $d = 4$, the depth buffer term will be $B_d \approx 4 \times 4 + 2 \times 4 = 24$ bytes and the color buffer term becomes $B_c \approx 2 \times 4 = 8$ bytes. Furthermore, we assume a texture cache miss rate of 25% ($m = 0.25$). This figure comes from Hakura and Gupta [HG97], who found that each texel is used by an average of four fragments when trilinear mipmapping is used. We have observed roughly the same behavior in our tests scenes.

Example I: Pelzer's ocean shader [Pel04] uses seven accesses to textures using trilinear mipmapping. The texture bandwidth alone will thus be $B_t = 2 \times 0.25 \times 7 \times (8 \times 4) \approx 112$ bytes. □

Example II: Uralsky [Ura05] presents an adaptive soft shadow algorithm, which uses more samples in penumbra regions. Eight points are first used to sample the shadow map, and if all samples agree, the point is considered as either in umbra or fully lit. Otherwise, 56 more samples are used. With percentage-closer filtering [RSC87] using four samples per texture lookup, we get, $B_t = 8 \times (4 \times 4) = 128$ bytes in non-penumbra regions, while in penumbra, $B_t = 64 \times (4 \times 4) = 1024$ bytes. Assuming only 10% of the pixels are in penumbra, the estimated cost becomes $B_t = 2 \times 0.25 \times (0.9 \times 128 + 0.1 \times 1024) \approx 109$ bytes. □

Compared to $B_c = 8$ and $B_d = 24$, it is apparent that texture memory bandwidth is substantially larger ($B_t = 112$ and $B_t = 109$). This term can also easily grow much larger. For example, current hardware allows for essentially unlimited texture accesses per fragment, and with anisotropic texture filtering the cost rises further.

In addition to this, both B_d and B_c can be reduced using non-lossy compression [Mor00, SMM*04]. Morein reports that depth buffer compression reduces memory accesses to the depth buffer by about 50%. B_d can also be further reduced using Z-min-culling [AMS03] and Z-max-culling [Mor00]. The advantage of buffer compression and culling is that they work transparently—the user do not need to do anything for this to work. For texture compression, however, the user must first compress the images, and feed them to the rasterizer. Both Example I and II contain textures that cannot be compressed, since they are created on the fly. Furthermore, none of the ordinary textures were compressed in the example code.

Texturing can easily become the largest cost in terms of memory bandwidth, as argued above. This fact is central when designing our traversal algorithm (Section 4). In computer cinematography, pixel shaders can be as long as several thousand lines of code [PVL*05], and this trend of making longer and longer pixel shaders can be seen in real-time rendering as well. This means that many applications are often pixel-shader bound. Thus, in a multi-view rasterization architecture, it may also be desirable to reduce the number of pixel shader evaluations, which is the topic of Section 4.3.

3. Background: Multi-View Rasterization

This section briefly describes a brute-force architecture for multi-view rasterization and multi-view projection.

3.1. Brute-Force Multi-View Rasterization

Here, we describe a brute-force approach to multi-view rasterization. This is used when rendering stereo in OpenGL and DirectX, and therefore we assume that this is the norm in multi-view rasterization. The basic idea is to first render the entire scene for the left view, and save the resulting color buffer. In a second pass, the scene is rendered for the right

```
BRUTEFORCE-MULTIVIEWRASTERIZATION()
1  for i ← 1 to n                          // loop over all views
2     create M^i                           // create projection matrix
3     for j ← 1 to t                       // loop over all triangles
4        RASTERIZETRIANGLE(M^i, Δ_j)
5     end
6  end
```

```
NEW-MULTIVIEWRASTERIZATION()
1  create all M^i, i ∈ [1,...,n]           // create projection matrices
2  for j ← 1 to t                          // loop over all triangles
3     RASTERIZETRIANGLETOALLVIEWS(M^1,...,M^n, Δ_j)
4  end
```

Figure 1: *Hi-level pseudo code for brute-force multi-view rasterization (top), and for our new multi-view rasterization algorithm (bottom). Assume rendering is done to n views, and that the scene consists of triangles, Δ_j, $j \in [1,\ldots,t]$. Note that the core of our algorithm lies in the* RASTERIZETRIANGLETOALLVIEWS() *function.*

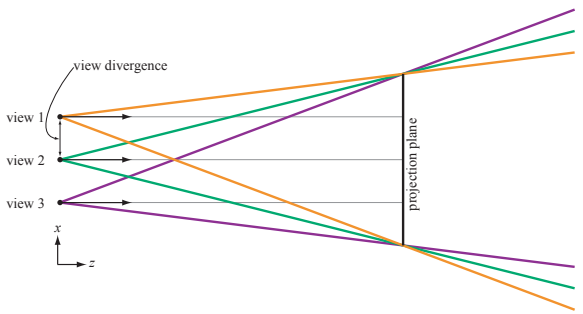

Figure 2: *Off-axis projections for three views. As can be seen, all views share the same z-axis. We use the term "view divergence" for the distance between two views' viewpoints.*

view, using another projection matrix. The resulting color buffers form a stereo image pair. Extending this to $n \geq 2$ views is straightforward. Pseudo code for brute-force multi-view rasterization is given in the top part of Figure 1, where it is assumed that we have a scene consisting of t triangles, Δ_j, $j \in [1,\ldots,t]$.

3.2. Multi-View Projection

We concentrate on projections with only horizontal parallax, since the great majority of existing autostereoscopic multi-view displays can only provide parallax along one axis. In this case, off-axis projection matrices are used. In Figure 2, this type of projection is illustrated for three views. Note that the distance between two views' camera viewpoints is denoted *view divergence*. In the following, we assume that n views are used, and we use an index, $i \in [1,\ldots,n]$, to identify a particular view. Furthermore, we have a vertex, $\mathbf{v}$, in object space, that should be transformed into homogeneous screen space for each view. These transformed vertices are denoted $\mathbf{p}^i$, $i \in [1,\ldots,n]$. For each view, a different object-space to homogeneous screen-space matrix, $\mathbf{M}^i$, must be created. Since parallax is limited to the x-direction, only the components of the first row of the $\mathbf{M}^i$ are different—the remaining three rows are constant across all views. Thus, the y, z, and w components of $\mathbf{p}_i = (p_x^i, p_y^i, p_z^i, p_w^i)^{\mathrm{T}} = \mathbf{M}^i\mathbf{v}$ will be exactly the same. We will use this fact when designing our traversal algorithm (next section). This could also be exploited for implementing an efficient vertex shader unit for a multi-view rendering architecture, but that is beyond the scope of this paper.

4. New Multi-View Rendering Algorithms

Since it is likely that texturing (B_t) is the largest consumer of memory bandwidth, our strategy is to devise a traversal algorithm that reduces the $n \times B_t$-term of Equation 3 as much as possible. Our hypothesis is that this should be possible if the texture cache can be exploited by all views simultaneously. Ideally, all views use exactly the same texels, and that would reduce $n \times B_t$ to B_t, which is a substantial improvement. Obviously, the best case will not occur, but very good results can be obtained as we will see. To make this possible, our approach is to rasterize a triangle to all views before starting on the next triangle. Pseudo code for this is given in the bottom part of Figure 1. Since the same texture is applied to a particular triangle for all views, one can expect to get more hits in the texture cache with this approach compared to the brute-force variant (Section 3.1).

The number of texture cache hits will also vary depending on how a triangle for the different views is traversed. Therefore, the core of our architecture lies in RASTERIZETRIANGLETOALLVIEWS (Figure 1), and in the following two subsections, we describe two variants of that algorithm. The general idea of our traversal algorithms is to traverse a small part (e.g., a tile or even just a pixel) of a triangle for a given view, and then determine which view to traverse next. To do this, we maintain an *efficiency measure*, E^i, for each view. With respect to the texture cache, the efficiency measure estimates which view is the best to continue traversing. Thus, E^i guides the traversal order of the different views.

We start by describing our algorithm for scanline-based traversal, and then show how it can be generalized to tiled traversal. In Section 4.3, we extend the traversal algorithm so that pixel shader evaluations can be approximated from one view to other views. Finally, in Section 4.4, our architecture is augmented in order to generate effects, such as depth of field, which require many samples.

4.1. Scanline-Based Multi-View Traversal

As many different parameters are often interpolated in perspective over a triangle, it is beneficial to compute normalized, perspective-correct barycentric coordinates (PBs),

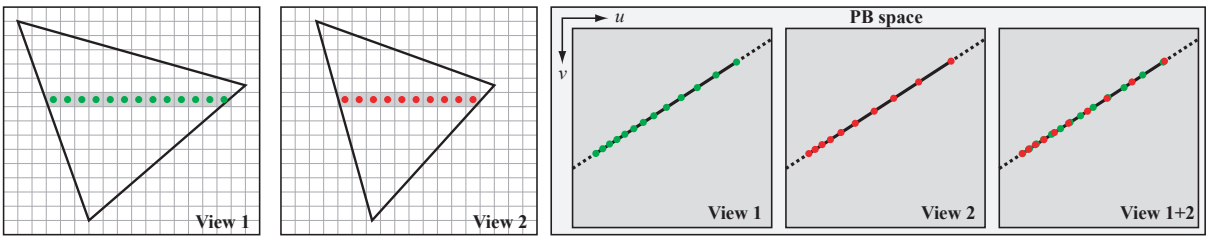

Figure 3: *The two views in a stereo system. A single scanline is highlighted for both the left and right view. In the right part of the figure, we show the PB space for view 1 and view 2, where the green and red samples in PB space correspond to the samples along the selected scanlines. To the very right, the texture sample locations from the two views are placed on top of each other. With respect to the texture cache, the best order to traverse the pixels in the two views is the order in which the samples occur along the PB traversal direction (fat line in PB space). The reason that the PB traversal directions in the two views are identical, is the multi-view projection, described in Section 3.2.*

$\mathbf{t} = (t_u, t_v)$ once per fragment, and then use these to interpolate the parameters in a subsequent step. The PBs can be expressed using rational basis functions [MWM02], and we will refer to the coordinate space of the PBs as *PB space*. Note that each view and pixel has its own PB, $\mathbf{t}^i = (t_u^i, t_v^i)$, where i is the view number.

The goal of our algorithm is to provide for substantial optimization of texture cache performance by roughly sorting the rasterized pixels by their respective PBs, and thereby sorting all texture accesses. In order to motivate this statement, we assume that a pixel shader program is used to compute the color of a fragment. The color will be a function, $color = f(\mathbf{t}^i, S^i)$, of the PB, t^i, and some state, S^i, consisting of constants for view i. If we assume that the shader contains no view dependencies, then all states, S^i, will be equal and we can write $color = f(\mathbf{t}^i, S)$, meaning that the only varying parameter will be the PBs. Since pixel shader programs are purely deterministic, the exact same PB will yield the same texture accesses. Therefore, it is reasonable to assume that roughly sorted PBs will give roughly sorted texture accesses. This applies to all texture accesses, including nested or dependant accesses, as long as they do not depend on the view. Note in particular that a shader containing view-independent texture access followed by view-dependent shading computations will be efficiently handled by our algorithm. An example of this is a shader that applies a bump map to a surface in order to perturb the normal, and after that, specular shading is computed based on the normal.

The rationale for our traversal algorithm is illustrated in Figure 3. For simplicity, only a stereo system is shown, but the reasoning applies to any number of views. We focus on a single scanline at a time. As can be seen, evenly spaced sample points in screen space are unevenly distributed in the PB space due to perspective. Using multi-view projection, the sample points for both views are located on a straight line in PB space. The direction of this line is denoted the *PB traversal direction.*

To guide the traversal, we define a signed efficiency measure, E^i, for each view, i, as:

$$E^i = \mathbf{d} \cdot \mathbf{t}^i, \tag{4}$$

where $\mathbf{d} = (d_u, d_v)$ is the PB traversal direction, which is computed as the difference between two PBs located on the same scanline. Thus, E^i is the projection of $\mathbf{t}^i$ onto the PB traversal direction.

In order to sort the pixel traversal order, we simply chose to traverse the pixel, and view, with the smallest efficiency measure E^i. When a pixel has been visited for view, i, the $\mathbf{t}^i$ for the next pixel for that view is computed, and the efficiency measure, E^i, is updated. The next view to traverse in is selected as before, and so on, until all pixels on the scanline have been visited for all views. Then, the next scanline is processed until the entire triangle has been traversed. An optimization of Equation 4 is presented in Section 5.

Below, pseudo code for our new traversal algorithm is shown. A single scanline is only considered since every scanline is handled in the same way.

TRAVERSESCANLINE(scanlinecoord y)

```
1  compute coordinate, x^i, for the leftmost pixel inside triangle
     for each view on scanline y
2  compute l^i =no. of pixels on current scanline for all views i
3  compute t^i for all views, i, for the leftmost pixel, (x^i, y)
4  compute d, and compute E^i = d · t^i for all views, i
5  while (pixels left on scanline for at least one view)
6     find view, k, with smallest E^k and l^k > 0
7     visit pixel (x^k, y) using t^k for view k
8     x^k = x^k + 1, l^k = l^k − 1
9     update t^k and E^k
10 end
```

In the algorithm presented above, we have used only the perspective-correct barycentric coordinates (PBs) to guide the traversal order. This does not take mipmapping into account. It would be very hard to optimize for mipmapping as it depends on a view-dependent level-of-detail parameter, λ^i, which is usually computed from the pixel shader program state using finite differences. It may be possible to optimize for mipmapping in the simple case of linear texture mapping, but it is next to impossible to generalize this to dependent accesses. Furthermore, we believe it is not worth

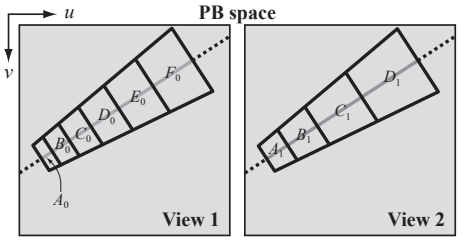

Figure 4: *Tiled traversal, shown in PB space, for a row of tiles overlapping a triangle (not shown). The PB traversal direction is the gray line. Our algorithm visits the tiles in the following order: A_0, A_1, B_0, B_1, C_0, D_0, C_1, E_0, D_1, and finally F_0. This is the order in which the tiles appear on the PB traversal direction.*

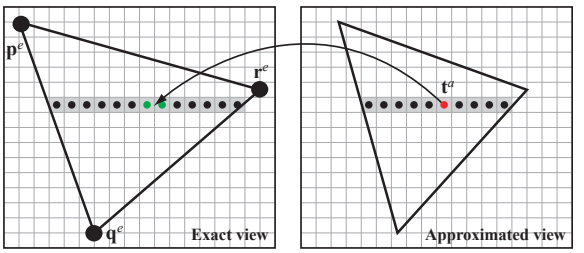

Figure 5: *Illustration of how a fragment (red circle) with perspective-correct barycentric coordinates $\mathbf{t}^a$, in an approximated view, can be mapped onto a screen-space position in the exact view. The color of two nearby fragments in the exact view (green circles) are interpolated to compute the color of the fragment in the approximated view.*

complicating our algorithm further for an expectedly slight increase in performance.

4.2. Tiled Multi-View Traversal

While scanline-based traversal works fine, there are many advantages of using a tiled traversal algorithm, where a tile of $w \times h$ pixels is visited before moving on to the next tile. For example, it has been shown that texture caching works even better [HG97], that simple forms of culling [Mor00, AMS03] can be implemented, and that depth buffer and color buffer compression [Mor00, SMM*04] can be employed. These types of algorithms are difficult or impossible to use with scanline-based traversal.

Fortunately, our scanline-based traversal algorithm can be extended to work on a per tile basis. This is done by imagining that the pixels in Figure 3 are tiles rather than pixels. A rectangular tile in screen space projects, in general, to a convex quadrilateral in PB space. In Figure 4, a triangle is assumed to have been rasterized, and for a particular row of tiles, the projection of the tiles are shown in PB space. As can be seen, the projected tiles overlap the same area in PB space for the two views, and therefore texture cache hit ratio can be expected to be high. This is especially true if the tiles are traversed in the order in which they appear along the PB traversal direction, as we suggest in our algorithm in Section 4.1.

Practically, this amounts to two computational and algorithmic differences compared to scanline-based traversal. First, we compute the efficiency measure, E^i, and perform sorting for the each tile rather than each pixel. As reference point for the computations, we use the center of the tile, but any point inside the tile would do. The second difference is that the traversal algorithm is designed so that all tiles (overlapping a triangle), on a row of tiles, are visited before moving on to the next row of tiles.

4.3. Approximate Pixel Shader Evaluation

In this section, we present an extension for tiled traversal algorithm (Section 4.2), which adds approximated pixel shader evaluation. The general idea, inspired by the work of Cohen-Or et al. [COMF99], is that *exact* pixel shader evaluation is done for a particular view, *e*, and when rasterizing to a nearby view, *a*, the pixel shader evaluations from view *e* are reused if possible. In Section 4.1, we motivated that if the pixel shader program contains no view dependencies, then it will be a deterministic function, $color = f(\mathbf{t}^i, S)$, of the PB, $\mathbf{t}^i$, of a pixel. We used this to motivate that for a given PB coordinate, the shader will always issue the same texture accesses. However, it is also true that the shader will always return the same color given the same PB as input. This implies that we should be able to reuse the results of the pixel shader programs.

In the following, we present an algorithm that exploits this assumption to provide *approximate* pixel shader evaluations, and our results show that we can obtain high-quality renderings. Since the approximation may produce incorrect results for view-dependent shaders, we suggest that the application programmer should have fine-grained control over this feature in order to turn it off/on as desired.

We initially divide our views into sets where the view divergences of the cameras in each set are considered small enough. When a triangle is rasterized, we select an *exact view* from each set. This can be done by either setting a fixed exact view, or by choosing the view that maximizes the triangle's projected area. We refer to the remaining views in the set as *approximated views*.

Figure 9 illustrates our approximate pixel shader technique. When evaluating the pixel shader for the exact view, we execute the full pixel shader, which may depend on the camera position. Hence, the exact view will render the triangle without any approximations. Looking at a single fragment in an approximated view, we will use its perspective-corrected barycentric coordinates (PBs), $\mathbf{t}^a = (t_u^a, t_v^a)$, to compute the position of the fragment in the exact view's ho-

mogeneous screen space. Assume the homogeneous coordinates of the triangle's vertices in the exact view are denoted $\mathbf{p}^e$, $\mathbf{q}^e$, and $\mathbf{r}^e$. Now, to find out which pixel in the exact view that correspond to $\mathbf{t}^a$ in an approximated view, we can use interpolation of the homogeneous screen-space coordinates as shown below:

$$\mathbf{c} = (c_x, c_y, c_z, c_w)^T = (1 - t_u^a - t_v^a)\mathbf{p}^e + t_u^a \mathbf{q}^e + t_v^a \mathbf{r}^e. \quad (5)$$

The screen-space x-coordinate in the exact view is found by homogenization: $x = c_x/c_w$, and the y-coordinate is implicitly known from the scanline being processed due to the findings in Section 3.2. The position, (x,y), will rarely map exactly to a sample point of a fragment in the exact view, but we can compute the color[†] of the approximated fragment by interpolating between the neighboring fragments in the exact view.

In order to approximate the pixel shader evaluation, we introduce a *shader output cache*, shown in Figure 6, which holds the pixel shader outputs (color and possibly depth) for a number of tiles rendered in the exact view. When a new triangle is being rasterized, we start by clearing the shader output cache to ensure that we only use fragment outputs of the same triangle during approximation. Each time a tile is being rasterized for the exact view, we allocate and clear the next tile-sized entry in the shader output cache. The next entry is selected in a cyclic fashion, so that the least recently traversed tile is retired from the cache. The pixel shader outputs of the fragments in the current tile are then simply written to that cache entry.

Each time we render a fragment in an approximated view, we compute the corresponding fragment position in the exact view as outlined above. The fragments to the left and right of the true fragment position are queried in the cache, and if both fragments are found, we compute the approximated pixel shader output by linear interpolation. If only one of the fragments exists, we simply set the approximated output to the value of that fragment.[‡] Finally, if none of the fragments are found, we execute the full pixel shader to compute the exact output.

Our main reason for choosing two shaded fragments and weighting them to form an approximated fragment is that we can use existing hardware interpolators to do the computations. This makes the implementation very inexpensive, and in Section 6, we show that this approximation generates high-quality results for view-independent shaders. However, there is a more precise way of performing the filtering. Instead of projecting the center of the pixel, we project the endpoints of a filter kernel into the exact view. This is shown in Figure 7. When the horizontal parallax difference between

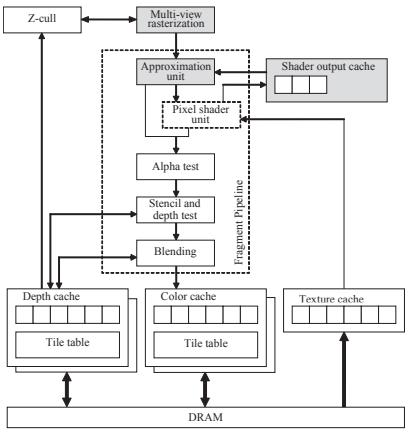

Figure 6: *To support approximate pixel shader evaluation, a shader output cache and an approximation unit (AU) are introduced into the rasterization pipeline. Before the pixel shader is executed for an approximated view, the AU checks with the shader output cache whether the fragment color can be computed from existing fragment colors in the cache. If so, the pixel shader unit is bypassed. Otherwise, the pixel shader is executed as usual.*

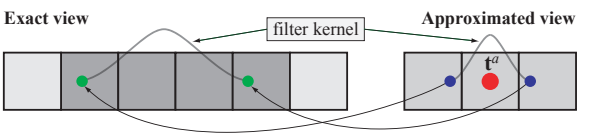

Figure 7: *Higher quality filtering is obtained by projecting the blue end points of the filter kernel in the approximated view into the exact view. These projected points (green) span a number of pixels (dark gray), which are weighted using the filter kernel to form the approximated fragment.*

the exact view and the approximated view is relatively large, the projected endpoints may span more than two pixels. To obtain a higher quality of the approximated fragments, a larger filter (e.g., tent, or Gaussian) is applied to all the pixels in the span. Note that this technique is also approximate.

The approximation method requires that the pixels in the different views are traversed in an ordered fashion, so the shader output cache is filled with the appropriate results before it is being queried. This is exactly what our algorithms (described in Section 4.1 & 4.2) do, and hence the approximation works well when used together with our traversal algorithms. However, in order to make sure that the shader output cache is filled before we start processing approximated views, we must delay the approximated views slightly. In our implementation we do this by computing the efficiency measure, E^i, for a tile located $k-1$ tiles away along the currently traversed row of tiles, where k is the number of entries in the shader output cache. For exact views, we compute the efficiency measure as usual.

[†] If the pixel shader also computes the depth of the fragment, our algorithm can be used to approximate the depth as well.

[‡] Alternatively, one could choose to execute the full pixel shader for such fragments. This would increase the quality slightly.

Discussion One possibility we explored was to use the already existing depth and color buffer caches to do the job of the shader output cache. This requires a small extension of one bit per cache entry to be able to flag if a color or depth value was written while rasterizing the current triangle. As it turned out, this works satisfactory. However, we cannot perform any approximations when blending is enabled, because using the blended values for approximation may cause distortion of geometry seen through a transparent triangle. Since blending is popular for particle effects, and crucial to multi-pass techniques in general, it is advantageous to be able to support blending in our approximate algorithm as well. We therefore recommend using the specialized shader output cache. It should be noted that the shader output cache can be kept small. For our tests, we use a cache size of five 4×4 pixel tiles and achieved a cache hit ratio of 95%. This includes all unavoidable cache misses that occur due to differing depth test outcomes or fragments that map to points outside the exact view.

4.4. Accumulative Color Rendering

A very simple and worthwhile extension of our architecture is to allow all views to access a single common color buffer, while each view has its own depth and stencil buffers. This allows for acceleration of some forms of multi-sampling effects, such as, depth of field for a single view. Recall that we compute the traversal order based on a texture-cache efficiency measure in our architecture. Therefore, the rendering order will be correct in the multiple depth buffers, but we cannot make any assumptions about the rendering order in the common color buffer. However, many multi-sampling algorithms can be described on an order-independent form:

$$\mathbf{m} = \sum_{i=1}^{n} w_i \mathbf{c}_i,$$

where n is the number of samples, $\mathbf{c}_i$ is the color of a sample, w_i is the weight of the sample (typically $w_i = 1/n$), and $\mathbf{m}$ is the color of the multi-sampled fragment. This equation can be implemented by using additive blending for the summation, and the factor w_i can be included in the pixel shader.

If the programmer is aware of that a multi-view rasterizer is being used, he/she can accelerate multi-sampling in the cases mentioned above. For instance, if we have hardware capable of handling four views, an image with depth of field using 4×4 samples can be rendered as follows:

```
RENDERSCENEDEPTHOFFIELD()
1  for y ← 1 to 4
2     create all M^i, i ∈ [1,4]
3     disable color writes
4     RASTERIZESCENETOALLVIEWS(M^1,...,M^4)
5     enable color writes
6     enable additive blending
7     enable pixel shader that includes the w_i = 1/16 scaling term
8     RASTERIZESCENETOALLVIEWS(M^1,...,M^4)
9  end
```

It is important to initialize the depth buffer (line 3 & 4) prior to color rendering, otherwise incorrect results will be obtained using additive blending. Note that this is also the case when using a single-view architecture, and so is not a disadvantage that stems from our architecture. An alternative approach is to render into 16 different color buffers, and combine the result in a post-process. However, our approach yields much better color buffer cache performance, since the triangles rendered simultaneously are coherent in screen space, and the post-processing stage is avoided.

In the pseudo code above, a deferred shading approach is used. Alternatively, the result could be blended directly into an accumulation buffer, but we have found that this uses more bandwidth. It is also worth pointing out that a brute-force implementation of depth of field using n samples per pixel sends the geometry n times to the graphics hardware. Our implementation sends the geometry $2n/v$, where v is the number of views (e.g. 16) the system can handle at a time.

5. Implementation

To benchmark and verify our algorithms, we have implemented a subset of OpenGL 2.0 in a functional simulator in C++. The core of the traversal algorithm and the approximation technique were implemented in less than 300 lines of code.

In our implementation, the efficiency measure E^i that is used to select the next tile to rasterize is computed in a less expensive way, compared to directly evaluating Equation 4. Since E^i is only used to sort the points, $\mathbf{t}^i$, we can evaluate this expression along the axis that corresponds to the largest of $\text{abs}(d_u)$ and $\text{abs}(d_v)$. Thus, E^i is simply the component of $\mathbf{t}^i$ that corresponds to this axis, and the computation of E^i is therefore almost for free.

Clipping a triangle against the view frustum may result in at most seven triangles. Since the projection matrices are different for each view, the number of resulting triangles may not be the same for all views. Hence, it is important to traverse unclipped triangles. We have adapted McCool et al's. [MWM02] traversal algorithm in homogeneous space, so that we traverse tiles along the horizontal viewport direction. In order to do so, we first find a screen space axis-aligned bounding box for each triangle, using binary search and a box-triangle overlap test [AMA05]. We then traverse the bounding box using a simple horizontal sweep. A more advanced traversal algorithm would yield higher performance, but this is outside the scope of this paper. The efficiency of the traversal algorithm will not affect bandwidth utilization in our simulator, since we detect tiles not overlapping the triangle and discard them.

In terms of culling, some special cases may occur. For instance, triangles can be backface culled or outside the view frustum in one view, while remaining visible in others. This

is easily solved if our algorithms are robustly implemented. A culled triangle can simply be given a scanline width of zero pixels, which will make sure it is never rasterized. For our approximate algorithm, a culled triangle will not generate any fragments, which means that the shader output cache will not be filled, and approximation will not be done. Thus, the visual result is not compromised.

It should be noted that it may be difficult to compute the PB traversal direction and efficiency measure in the context of a tiled rasterizer. We have favored simple code, and compute the PB traversal direction from the start and end points of a row of tiles. We also evaluate the efficiency measure in the center of each tile, regardless of whether it lies inside the triangle or not. This implementation suffers from a weakness that appear when a row of tiles cross the "horizon" of the plane that pass through the current triangle. When crossing this horizon, the PBs behave similarly to an $1/x$ function in the vicinity of the origin. This results in sign changes in the efficiency measure, and ultimately in incorrect sorting. However, it should be noted that this is a very rare occurrence. Furthermore, it will only affect the texture cache efficiency, and not the correctness of the result. In the future, we would like to investigate if there is an elegant way to extend our implementation so that correct sorting is guaranteed even in these extreme cases.

6. Results

In this section, we present the results for our exact and approximate algorithms (Section 4.2 and 4.3, respectively). The results were obtained from our functional simulator, using the test scenes summarized in the top row of Figure 8.

The Quake3 scene is a game level using multi-texturing with one texture map combined with a light map for every pixel. A *potentially visible set* is used during rendering, so the overdraw factor is similar to that of most modern games.

For our Soft Shadows test scene, we implemented Uralsky's soft shadow mapping algorithm [Ura05] in combination with bump-mapped and gloss-mapped per-pixel Phong shading. This scene is meant to model a modern or next-generation graphics engines, targeted for real-time graphics, which makes heavy use of complex shaders containing many texture accesses.

The Ocean scene is our implementation of Pelzer's ocean shader [Pel04]. This scene is a nightmare scenario for our algorithms, as it contains bump-mapped reflections and refractions, both view-dependent and highly diverging due to the bump mapping and high view divergence of the cameras. Thus, this scene was designed to contradict all assumptions made in our algorithms.

All scenes have an animated camera, and statistics were gathered for at least 200 frames. The Ocean scene also has an animated water surface. We chose to render at a resolution of 640×480 pixels per view, which is reasonable considering the current 3D display technology. For example, Philips has built a 3D display capable of either nine views at 533×400 pixels, or seven views at 686×400 pixels [vB04]. We have investigated the behavior of our algorithms with respect to rendering resolution, and conclude that they both behave robustly. Both total bandwidth and texturing bandwidth increase slightly sub-linearly with increasing resolution. This effect is due to all caches getting slightly more cache hits at higher resolutions, and the compression ratios of color and depth buffers were either constant or became slightly better. Similar behavior was observed for a brute-force architecture.

Even though there are computations that can be shared among different views in the vertex-processing units, we focus our evaluation only on the rasterizer stage, since it is very likely that it will become the bottleneck. In the following, we refer to a *conventional rasterizer* (CR) as a modern rasterizer architecture with the following bandwidth reducing algorithms: fast depth clears, depth buffer compression, depth buffer caching, Z-max culling, texture caching, color buffer compression and color buffer caching. A multi-view rasterization architecture that is implemented by rendering the scene n times using a single CR is called a *brute-force* (BF) multi-view architecture (see also Section 3.1).

For all architectures, we use a fully associative 6 kB texture cache with least-recently used (LRU) replacement policy. Since our architecture rasterizes to all views simultaneously, it needs an increasing amount of depth and color buffer cache with an increasing number of views. For all our tests, our architecture uses $n \times 512$ bytes for the depth buffer caches, and $n \times 512$ bytes for the color buffer caches. Thus, a stereo system will use 8 kB cache memory in total.

For a fair comparison, we ensure that our architecture and the BF architecture use exactly the same amount of cache memory. The extra 1 kB of cache memory that we need per view, can be spent on either the depth and color buffer caches, or on the texture cache in a BF architecture. We call these architectures BF DC and BF TX respectively. In all our tests, we have observed that the total amount of bandwidth is reduced most if the texture cache is increased. We have therefore chosen to omit the BF DC architecture from the results.

We present statistics gathered from our test scenes in Figure 8. As can be seen in those diagrams, both our exact and our approximate rasterization algorithms perform far better than the brute-force architecture. For the Quake3 scene, the majority of bandwidth usage is spent on the color and depth buffer. However, our algorithm provides major reductions in terms of texture bandwidth. In fact, it remains almost constant over an increasing number of views. The same holds for the Soft Shadow scene, but the results are even better since the texture bandwidth is more dominating compared to the Quake3 scene.

In the case of the Ocean scene, our algorithm performs

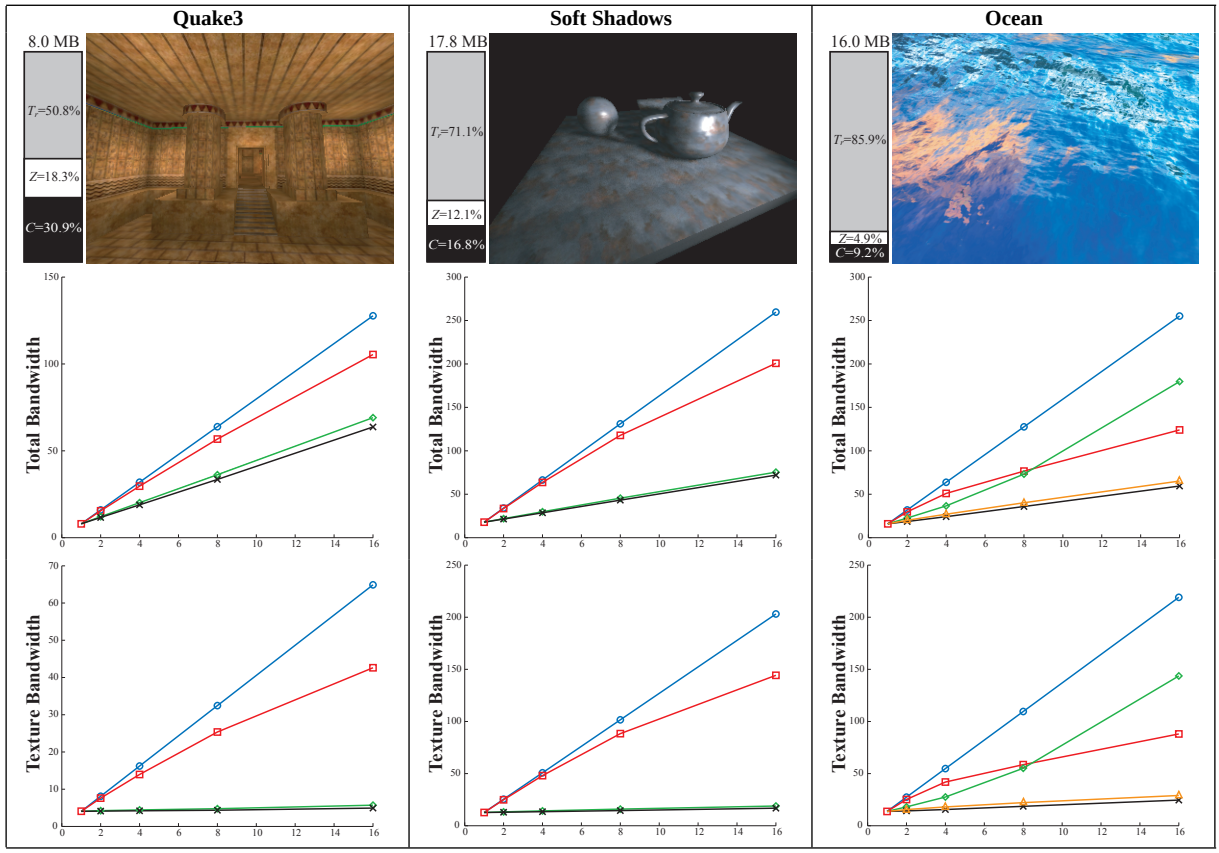

Figure 8: The first row shows a summary of the test scenes rendered at 640×480 pixels. The bandwidth (BW) figures, located on top of each bar, is the BW in megabytes (MB) per frame for a single view using a conventional rasterizer, as described in the text. Each bar is divided into texturing BW (gray), depth buffer BW (white), and color buffer BW (black). The second and third rows show total BW and texture BW per frame as a function of the number of views for each frame.

worse than BF TX when the number of views is greater than eight. This is not very surprising considering that the scene was designed as a worst case for our algorithm. The scene contains very little view coherency in the texture accesses, and the BF TX architecture will have a much bigger texture cache at 16 views. We think that the results are very good considering the circumstances, and substantial bandwidth reduction is achieved up to four views. In fact, our algorithm performs better even for 16 views, if we render four passes with a four-view architecture. It should be noted that all benchmarks once again turned completely to our favor simply by increasing the texture cache size to 12 kB. This means that for every scene, there is a texture cache size "knee" that makes our algorithm perform extremely well. The same applies to a BF TX architecture: when the texture cache size is decreased, performance will degrade gracefully. We have also included bandwidth measurements for a version of the Ocean scene that used a shared camera position for the shaders (Figure 6), and these indicate how disadvantageous the view dependencies of this scene are.

It should be noted that even though the bandwidth measurements for our approximate algorithm is only marginally better than the exact algorithm, the approximate algorithm completely avoids a very large amount of pixel shader program executions. Hence, computational resources are released and can be used for other tasks. Our tests show that about 95% of the pixels, in the approximated view in a stereo system, can be approximated. When the number of views increases, fewer pixels can be approximated due to increased view divergence, and with 16 views, our approximation ratio has dropped to approximately 80%. See Figure 9 for a visualization of the approximation in a five-view system. A major advantage of our approximate algorithm is that it always generates correct borders of the triangles and correct depth—only the content "inside" a triangle can be subject to approximation.

It is also important to measure image quality of our approximate algorithm. For the Quake3 scene, the peak-signal-to-noise-ratio (PSNR) was about 40 dB for the entire animation. This is considered high even for still image compression. When the number of views increased, the

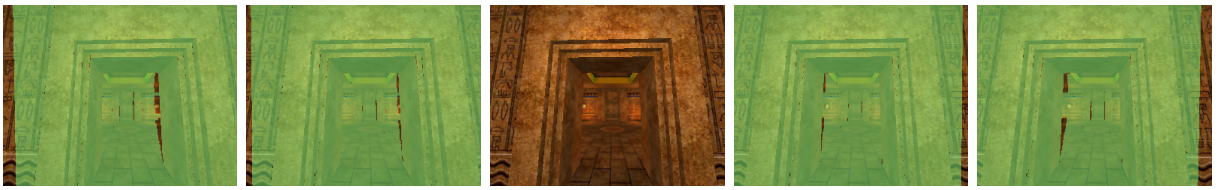

Figure 9: *Visualization of approximation in the Quake3 scene. Green pixels have been approximated from the exact central view.*

PSNR remained relatively constant. This was not expected by us, since using linear interpolation for approximation gives worse results for a larger view divergence between an exact view and an approximated view. However, fewer pixels can be approximated when the view divergence is high between the approximated and exact view, and hence the quality increases.

The Soft Shadows and Ocean scene are harder cases for the approximation algorithm since they both contain view dependencies. The Soft Shadows scene contain view dependencies in the form of specular highlights, and the Ocean scene has nested view-dependent texture lookups (bump-mapped reflections and refractions). For those scenes, we must use a unified camera position for shading when the approximate algorithm is used. Otherwise visible seams may appear between approximated pixels and "exact" pixels. In our Soft Shadows scene, the unified camera position is hardly visible, and the PSNR is between 36 and 40 dB. In our Ocean scene, the differences are easily spotted when comparing to the exact solution, but the approximated version still looks good.As previously stated, we believe the application programmer should be given the appropriate control over when approximation should be used. Approximation could be turned off for surfaces with view-dependent shaders, or be controlled by a user-tweakable setting for quality or performance. In some applications, such as games, it may be more reasonable to use approximation. However, when a graphics hardware architecture is used for scientific computing, e.g. fluid dynamics on the GPU, the application programmer would probably want to turn off approximation, and only use our exact algorithm, which would still give a performance advantage.

Interestingly, for our approximate algorithm, we observed that compression of the color buffer works better than for our exact algorithm. On average, the compression ratio improved by 5–10%, most likely because of the slight low-pass effect introduced by the approximation filter.

To summarize, our results show that our multi-view rasterization architecture gives substantial reductions in terms of total bandwidth usage. The texture bandwidth remains close to constant with an increasing number of views, and texture bandwidth reductions on the order of a magnitude are possible. Furthermore, our approximate technique can render high-quality images without executing the pixel shader for up to 95% of the fragments.

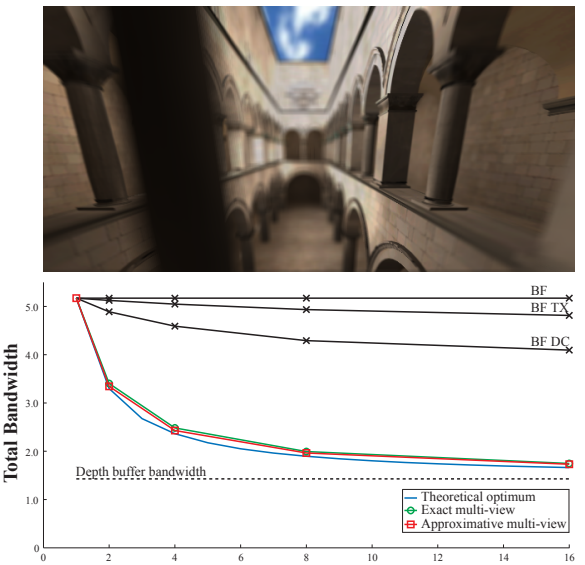

Figure 10: *The Sponza atrium rendered with a depth of field (DOF) effect. The diagram shows bandwidth measurements in gigabytes per frame as a function of the number of views supported by the rasterizer. The curve named "Theoretical optimum" shows the best possible performance for our architecture at a given number of supported views. In short, we assume that the texture and color buffer bandwidth are zero for all views but one during each render pass. The BF DC architecture has been included because it performs better than the BF TX architecure in this particular benchmark. This is due to the two-pass nature of the DOF algorithm, and since the scene does not use any complicated shaders.*

6.1. Accumulative Color Rendering

Figure 10 shows our final test scene, which is a *depth of field* (DOF) rendering of the Sponza atrium, using multi-texturing with decal textures and global illumination light maps. In contrast to the Ocean scene, which was designed as a nightmare scenario, this test hits the very sweet spot of our algorithm. Here, we benchmark the performance of accumulative color rendering (Section 4.4). The tests were made using the same configurations of the rasterizer as in the previous benchmarks. However, this time we rendered a 16×16 samples DOF, where each configuration rendered the scene in as few passes as possible. For instance, a 16-view multi-view rasterization architecture would need to render

16 passes, while a 4-view system would need $4 \times 16 = 64$ passes. The BF algorithms always require all 256 passes. Our results in Figure 10 show a major reduction, not only in texture bandwidth, but also in color buffer bandwidth. This is to be expected, since all color buffer cache memory can be spent on a single color buffer, and since a projected primitive will be relatively coherent in screen space across all views when rasterizing to the same buffer. It is worth noting that the performance of our architecture is very close to its theoretical limit.

6.2. Small triangles

In this section, we will shed some light on how our architecture performs when rendering very small triangles. As we perform sorting on a per-tile level, the behavior of our algorithm will approach an architecture that renders a triangle to all views without any sorting at all (called "Tri-by-Tri" below) when rendering small triangles. In order to test sub-pixel triangle rendering, i.e., where the average number of pixels per triangle (ppt) is less than one, we rendered the Quake3 scene at low resolution using various four-view systems. In the following table, we present the texture bandwidth relative to the BF TX architecture:

	80×60, 0.8 ppt	640×480, 48 ppt
BF TX	100%	100%
Tri-by-Tri	31.3%	88.8%
Our	28.5%	27.3%

As can be seen, our algorithm handles small triangles very robustly. A view-independent texture access will be very coherent for a small triangle no matter what point in the triangle we choose to sample from, and drawing a triangle to all views will provide sufficient sorting of the texture accesses. Our opinion is that the Tri-by-Tri algorithm is much less robust since it fails horribly when the triangle area increase. Large triangles are still frequently used in games, architectural environments & particle systems, and it is therefore crucial to be able to handle them as well.

7. Discussion

Our multi-view rasterization architecture has been designed with the current technological development in mind: computing power grows at a much faster rate than memory bandwidth, and DRAM capacity is expected to double every year [Owe05]. Hence our focus has been on reducing usage of memory bandwidth at a cost of duplicating the depth and stencil buffers. The BF architecture would only need to duplicate the color buffer, if the depth and stencil buffers are cleared between rendering different views.

From a cost/performance perspective, a reasonable solution would be to implement a multi-view rasterizer with our approximate pixel shader technique and our tiled traversal for two, three, or four views. A multi-pass approach can be used when rendering to more views than supported by the architecture. For example, a system for four views can be used to render to 12 different views by rendering the scene three times.

Our traversal algorithm requires the ability to change the view which is currently being rasterized to. The same pixel shader program is used for all views, so switching views only amounts to changing the current active view. This can be done by enumerating all views, and changing the currently active index. This index points to view-dependent information, such as view parameters, and it also points to output buffers, for example. Therefore, we are confident that view switching can be efficiently implemented in hardware.

8. Conclusion and Future Work

We have presented a novel multi-view rasteriziation architecture, and shown that it is possible to exploit a substantial amount of the inherent coherency in this context. It is our hope that our work will renew interest in multi-view image generation research, a field that has received relatively little attention. Furthermore, it is our belief that our architecture may accelerate the acceptance of multi-view displays for real-time graphics.

With our current architecture, it is apparent that the bottlenecks from texturing and complex pixel shaders have moved to color and depth buffer bandwidth usage. For future work, we would therefore like to investigate whether these buffers can be compressed simultaneously for all views. This can potentially lead to higher compression ratios. Some kind of differential encoding might be a fruitful avenue for this type of problem. Currently, we are making an attempt at augmenting our algorithms so that parallax in more directions can be obtained. This would allow us to have, for example, 2×2 viewpoints, and thus achieve both horizontal and vertical parallax. The parameter space (texture cache size, tile size, etc) involved in our architecture is large, and in future work, we want to explore various configurations in more detail.

References

[AH93] ADELSON S., HODGES L.: Stereoscopic Ray Tracing. *The Visual Computer, 10*, 3 (1993), 127–144.

[AH94] ADELSON S. J., HANSEN C. D.: Fast Stereoscopic Images with Ray-Traced Volume Rendering. In *Symposium on Volume Visualization* (1994), pp. 3–9.

[Ake03] AKELEY K.: The Elegance of Brute Force. In *Game Developers Conference* (2003).

[AMA05] AKENINE-MÖLLER T., AILA T.: Conservative Tiled Rasterization Using a Modififed Triangle Setup. *Journal of graphics tools, 10*, 2 (2005), 1–8.

[AMN03] AILA T., MIETTINEN V., NORDLUND P.: Delay Streams for Graphics Hardware. *ACM Transactions on Graphics, 22*, 3 (2003), 792–800.

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics, 22*, 3 (2003), 801–808.

[BAC96] BEERS A., AGRAWALA M., CHADDA N.: Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96* (August 1996), pp. 373–378.

[CH93] COX M., HANRAHAN P.: Pixel Merging for Object-Parallel Rendering: a Distributed Snooping Algorithm. In *Symposium on Parallel Rendering* (November 1993), pp. 49–56.

[COMF99] COHEN-OR D., MANN Y., FLEISHMAN S.: Deep Compression for Streaming Texture Intensive Animations. In *Proceedings of ACM SIGGRAPH 99* (1999), pp. 261–268.

[Dod05] DODGSON N. A.: Autostereoscopic 3D Displays. *IEEE Computer, 38*, 8 (2005), 31–36.

[Hal98] HALLE M.: Multiple viewpoint rendering. *Computer Graphics 32*, Annual Conference Series (1998), 243–254.

[HG97] HAKURA Z. S., GUPTA A.: The Design and Analysis of a Cache Architecture for Texture Mapping. In *24th International Symposium of Computer Architecture* (June 1997), pp. 108–120.

[HK96] HE T., KAUFMAN A.: Fast Stereo Volume Rendering. In *Proceedings of the 7th Conference on Visualization '96* (1996), pp. 49–56.

[IEH99] IGEHY H., ELDRIDGE M., HANRAHAN P.: Parallel Texture Caching. In *Graphics hardware* (1999), pp. 95–106.

[IEP98] IGEHY H., ELDRIDGE M., PROUDFOOT K.: Prefetching in a Texture Cache Architecture. In *Graphics Hardware* (1998), pp. 133–142.

[JFO02] JAVIDI B., F. OKANO E.: *Three-Dimensional Television, Video, and Display Technologies*. Springer-Verlag, 2002.

[KSKS96] KNITTEL G., SCHILLING A., KUGLER A., STRASSER W.: Hardware for Superior Texture Performance. *Computers & Graphics, 20*, 4 (1996), 475–481.

[Mor00] MOREIN S.: ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings* (August 2000), ACM Press.

[MP04] MATUSIK W., PFISTER H.: 3D TV: A Scalable System for Real-Time Acquisition, Transmission, and Autostereoscopic Display of Dynamic Scenes. *ACM Transactions on Graphics, 23*, 3 (2004), 814–824.

[MWM02] MCCOOL M. D., WALES C., MOULE K.: Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware* (2002), pp. 65–72.

[Owe05] OWENS J.: Streaming Architectures and Technology Trends. In *GPU Gems 2*. Addison-Wesley Professional, 2005, pp. 457–470.

[Pel04] PELZER K.: Advanced Water Effects. In *Shader X2*. Wordware Publishing Inc., 2004, pp. 207–225.

[PK96] PROFFITT D. R., KAISER M.: Hi-Lo Stereo Fusion. In *ACM SIGGRAPH 96 Visual Proceedings* (1996), p. 146.

[PVL*05] PELLACINI F., VIDIMČE K., LEFOHN A., MOHR A., LEONE M., WARREN J.: Lpics: A Hybrid Hardware-Accelerated Relighting Engine for Computer Cinematography. *ACM Transactions on Graphics, 24*, 3 (2005), 464–470.

[RSC87] REEVES W. T., SALESIN D. H., COOK R. L.: Rendering Antialiased Shadows with Depth Maps. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)* (1987), pp. 283–291.

[SA92] SEGAL M., AKELEY K.: The OpenGL Graphics System: A Specification.

[SBM04] STEWART J., BENNETT E. P., MCMILLAN L.: Pixelview: a view-independent graphics rendering architecture. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2004), pp. 75–84.

[Shi05] SHISHKOVTSOV O.: Deferred Shading in S.T.A.L.K.E.R. In *GPU Gems 2*. Addison-Wesley Professional, 2005, pp. 143–166.

[SHS00] STOEV S. L., HÜTTNER T., STRASSER W.: Accelerated Rendering in Stereo-Based Projections. In *Third International Conference on Collaborative Virtual Environments* (2000), pp. 213–214.

[SMM*04] SCHNEIDER B.-O., MOLNAR S., MONTRYM J., DYKE J. V., LEW S.: System and Method for Real-Time Compression of Pixel Colors. US Patent 6,825,847, 2004.

[Ura05] URALSKY Y.: Efficient Soft-Edged Shadows Using Pixel Shader Branching. In *GPU Gems 2*. Addison-Wesley Professional, 2005, pp. 269–282.

[vB04] VAN BERKEL C.: Philips Multiview 3D Display Solutions. 3D Consortium, http://www.3dc.gr.jp/english/domestic_rep/040617a.php, 2004.

[Wil83] WILLIAMS L.: Pyramidal Parametrics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)* (July 1983), pp. 1–11.