

3D Rasterization – Unifying Rasterization and Ray Casting

Carsten Dachsbacher^{†1} Philipp Slusallek^{‡2} Tomas Davidovic³ Thomas Engelhardt¹ Mike Phillips³ Iliyan Georgiev³

¹VISUS, University of Stuttgart ²DFKI, Saarland University ³Excellence Cluster M2CI, Saarland University

Abstract

Ray tracing and rasterization have long been considered as two very different approaches to rendering images of 3D scenes that – while computing the same results for primary rays – lie at opposite ends of a spectrum. While rasterization first projects every triangle onto the image plane and enumerates all covered pixels in 2D, ray tracing operates in 3D by generating rays through every pixel and then finding the first intersection with a triangle. In this paper we show that, by making a slight change that extends triangle edge functions to operate in 3D instead of 2D, the two approaches become almost identical with respect to primary rays, resulting in an efficient rasterization technique. We then use this similarity to transfer rendering concepts between the two domains. We generalize rasterization to arbitrary non-planar perspectives as known from ray tracing, while keeping all benefits from rasterization. In the reverse we transfer the concepts of rendering consistency, which have not been available for ray tracing thus far. We then demonstrate that the only remaining difference between rasterization and ray tracing of primary rays is scene traversal. We discuss a number of approaches from the continuum made accessible by 3D rasterization.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Picture/Image Generation—Display algorithms; Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—;

1. Introduction

The two main algorithms used in computer graphics for generating 2D images from 3D scenes are rasterization [Pin88] and ray tracing [App68, Whi80]. While they were developed at roughly the same time, rasterization has quickly become the dominant approach for interactive applications because of its initially low computational requirements (no need for floating point in 2D image space), its ability to be incrementally moved onto hardware, and later by the ever increasing performance of dedicated graphics hardware [Ake93, LKM01, NVI08]. The use of local per triangle computation makes it well suited for a feed-forward pipeline. However, the handling of global effects, such as reflections, is intricate.

Ray tracing, on the other hand, works directly in 3D world space, typically requiring floating point operations throughout rendering with recursive visibility queries for global ef-

fects, which require almost random memory access. This has resulted in the failure of many attempts to map ray tracing to hardware. Software-only ray tracing for interactive applications was generally assumed to be non-competitive, and ray tracing research essentially ceased during the 1990s. However, recent interactive Whitted-style ray tracing approaches achieved significant performance gains [PMS*99, WSB01, RSH05] and ray tracing has efficiently been implemented in hardware [WSS05]. Similar to many of these works, we focus on primary rays from a camera where rasterization and ray tracing compute identical results. Casting such coherent rays has been the focus of much research in ray tracing in recent years. In this spirit, we consider higher-order rays, and efficient global illumination algorithms, such as instant radiosity, or photon mapping, as orthogonal to our approach.

Rasterization and ray tracing seemed to be fundamentally different even for the case of primary rays. In this paper we show that with just a slight generalization – rasterizing through 3D edge functions defined in world space instead of 2D edge functions – we can fold ray casting, i.e. ray tracing of primary rays, and rasterization into a single 3D rasteriza-

[†] dachsbacher@visus.uni-stuttgart.de

[‡] slusallek@dfki.de



Figure 1: With our approach, ray tracing and rasterization become almost identical with respect to primary rays. Now rasterization can directly render to non-planar viewports using parabolic and latitude-longitude parameterizations (left images), and we can transfer rendering consistency and efficient anti-aliasing schemes from rasterization to ray tracing. The center image shows the Venice scene consisting of 1.2 million triangles. Our 3D rasterization bridges both approaches and allows us to explore rendering methods in between. The right images show the number of edge function evaluations per pixel for two different 3D rasterization methods (3DR-BIN and 3DF-FULL, see Sect. 5).

tion algorithm. As a result we can apply many techniques that have been limited to one approach in the context of the other. We demonstrate direct rasterization of non-planar views, as well as efficient anti-aliasing and fully consistent intersection computation for 3D rasterization.

In this new setting the only difference between the two algorithms lies in the way the scene is traversed. Our approach combines the 2D and 3D acceleration structures from rasterization and ray tracing: While the 2D grid of pixels is used to (hierarchically) test tiles of pixels for overlap with a triangle, we can use the same concept to test frusta against the bounding boxes of spatial index structures as used in ray tracing. Similarly, we can make use of an existing 3D spatial index structure by using frustum culling and occlusion culling in the core rasterization algorithm. Our approach allows for freely, and continuously, exploring the space between traditional ray tracing and rasterization.

Because 3D rasterization operates in world space it uses floating point computation throughout the pipeline. Our main target platform is a fully software-based graphics pipeline on highly parallel and programmable many-core processors, such as Intel’s Larrabee. On such platforms there is little, if any, drawback when using floating point compared to fixed-point computation.

2. Previous Work

Rasterization is currently the dominant rendering technique for real-time 3D graphics, and is implemented in almost every graphics chip. The well-known rasterization pipeline [FvDFH90] (see Fig. 4 for a contemporary design) operates on projected and clipped triangles in 2D. The core of the rasterization algorithm, coverage computation, determines, for all pixels (and possibly several sub-samples in each pixel), whether they are covered by a given triangle. The coverage test typically uses linear 2D distance functions in image space, one for each triangle edge, whose sign determines which side of an edge is inside the triangle [Pin88]. These functions can be evaluated in parallel, and hierarchi-

cally, to quickly locate relevant parts of the screen. Many extensions to this basic algorithm have been proposed including the hierarchical Z-buffer [GKM93], efficient computation of coverage masks [Gre96], hierarchical rasterization in homogeneous coordinates [OG97], and the irregularly sampled Z-buffer [JLBM05].

The idea of ray tracing was introduced to graphics by Appel [App68], while Whitted [Whi80] developed the recursive form of ray tracing. Since then, the two main trends in ray tracing have been the development of physically correct global illumination algorithms (see the overview in [PH04]), and trying to reach real-time performance comparable to rasterization. The latter is most often achieved by simultaneously tracing packets of coherent rays to increase performance on parallel hardware [WSB01]. Recent approaches use large ray packets and optimized spatial index structures, such as kd-trees [RSH05], BVHs [WBS07], interval trees [WSS05, WK06], and 3D grid structures [WIK*06]. A recent survey [WMG*07] gives an overview of build and traversal algorithms for spatial index structures used for ray tracing. Related to this paper, Hunt et al. [HM08] describe a continuum of *visibility algorithms* between tiled z-buffer systems and ray tracing by introducing acceleration structures that are specialized for rays with specific origins and directions. Our work goes further, folds ray casting and rasterization into a single algorithm, and thus allows full exploration of the continuum of rendering algorithms between the two.

While rasterization has benefited tremendously from being implemented in dedicated hardware, ray tracing was almost exclusively limited to software implementations, even when executed on the same graphics hardware [PBMH02, HSHH07, PGSS07] (we consider GPUs as a programmable hardware dedicated to graphics that runs software implemented as shaders). However, the increasing parallelism and programmability of graphics processors make it very likely that both rendering algorithms will be mostly implemented in software in the foreseeable future [Mar08, SCS*08].

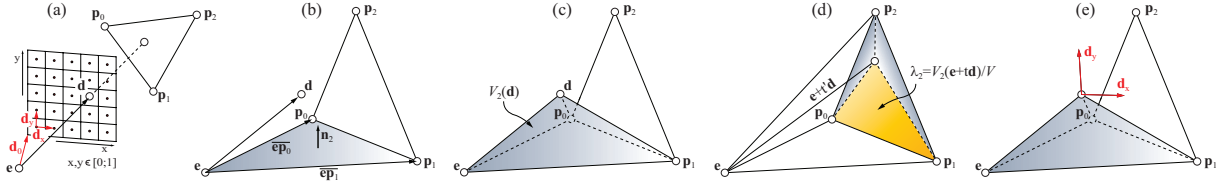


Figure 3: 3D edge functions for a triangle $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$ and a ray starting at the eye \mathbf{e} going through a pixel (x, y) with coordinates \mathbf{d} in world space. Intersection tests and computation of the barycentric coordinates are based on the signed volumes of the spanned tetrahedra.

3. From 2D to 3D Rasterization

In the following we first briefly review the traditional 2D rasterization technique, as it is implemented in current graphics hardware, before extending it to 3D. We discuss the modifications that would be required for integration into existing rendering pipelines, and illustrate benefits and new possibilities.

3.1. 2D Rasterization

Any linear function, u , on a triangle in 3D, e.g. colors or texture coordinates, obeys $u = aX + bY + cZ$, with $(X, Y, Z)^T$ being a point in 3D, and the parameters a , b , and c can be determined from any given set of values defined at the vertices [OG97]. Assuming canonical eye space – where the center of projection is the origin, the view direction is the Z-axis, and the field of view is 90 degrees – dividing this equation by Z yields the well-known 2D perspective correct interpolation scheme [Hec89] from which we observe that u/Z is a linear function in screen-space. During rasterization both u/Z and $1/Z$ are interpolated to recover the true parameter value, u .

Likewise, we can now define three linear edge functions in the image plane for every triangle (Fig. 2): Their values are equal to zero on two vertices and one on the opposite vertex [OG97]. A pixel is inside the triangle if all three edge functions are positive at its location and thus coverage computation becomes a simple evaluation of the three edge functions, which is well suited for parallel architectures. Hierarchical testing of pixel regions for triangle coverage, called *binning*, is a major factor for rendering performance. Typ-

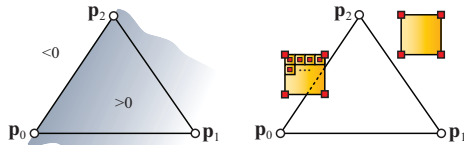


Figure 2: Left: The 2D edge function for the edge $\overline{\mathbf{p}_0\mathbf{p}_2}$ which can be evaluated in parallel. A hierarchical search can quickly locate pixels covered by the triangle (right).

ical implementations use either a quad-tree subdivision in image space starting from the entire screen or the triangle’s bounding box, or they locate relevant parts by sliding a larger bin over the screen, followed by further subdivision or per-pixel evaluation. Binning can be seen as similar to culling a triangle from a frustum of primary rays in ray tracing. Note that there are other triangle rasterization algorithms as well, but hierarchical rasterization has proven to be the most efficient and hardware-friendly algorithm and we thus restrict our discussion to it.

3.2. 3D Rasterization

Testing whether a pixel sample is covered by a 2D triangle is equivalent to testing if a ray, beginning at the eye, going through that pixel, intersects the triangle. Fig. 3a depicts this using the following notation: \mathbf{e} is the eye location, and the ray goes through $\mathbf{d} = \mathbf{d}_0 + x\mathbf{d}_x + y\mathbf{d}_y$, for pixel coordinates (x, y) , while $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ are the vertices of the triangle.

We can now formulate 3D linear edge functions using the signed volume idea well-known as the Pluecker ray-triangle intersection test [O’R98, KS06]. We first consider the triangle formed by the first edge $\overline{\mathbf{p}_0\mathbf{p}_1}$ and the eye (see Fig. 3b). For notational simplicity in this explanation, but without loss of generality, we assume that \mathbf{e} is in the coordinate origin, and thus the normal of the triangle is $\mathbf{n}_2 = \mathbf{p}_1 \times \mathbf{p}_0$. For better numerical stability with small triangles we use, mathematically equivalent, $\mathbf{n}_2 = \mathbf{p}_1 \times (\mathbf{p}_0 - \mathbf{p}_1)$ in practice, and for the cross-product we also use a consistent ordering of vertices of adjacent triangles. We define the corresponding 3D edge function as $V_2(\mathbf{x}) = \mathbf{n}_2 \cdot \mathbf{x}$, which is equivalent to computing the scaled signed volume of the tetrahedron with vertices $\mathbf{p}_0, \mathbf{p}_1, \mathbf{x}$, and \mathbf{e} (here the origin). The scaling factor of 6 can be ignored as intersection tests are based on the signs and ratios of volumes. $V_2(\mathbf{x})$ is positive for all \mathbf{x} in the half-space containing \mathbf{p}_2 (Fig. 3c). In analogy we define V_0 and V_1 using:

$$\begin{aligned} \mathbf{n}_i &= \mathbf{p}_{(i+2) \bmod 3} \times \mathbf{p}_{(i+1) \bmod 3} \\ V_i(\mathbf{x}) &= \mathbf{n}_i \cdot \mathbf{x}, \text{ with } i = 0, 1, 2. \end{aligned} \quad (1)$$

We denote the scaled volume of the tetrahedron spanned by the triangle and the origin by $V = \mathbf{p}_j \cdot \mathbf{n}_j$ (for any $j = 0..2$). To determine if the triangle is hit by a ray in the positive direction we only need to consider the signs of the volumes

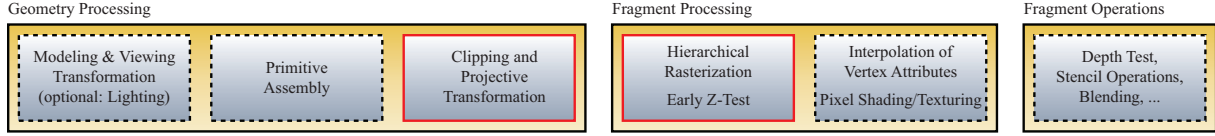


Figure 4: A depiction of the rasterization pipeline. When using 3D rasterization only the last stage of geometry processing and the first step of fragment processing (shown in red), requires modifications.

V_i and V : The ray $\mathbf{e} + t'\mathbf{d}$, $t' > 0$, hits the triangle if all four volumes have the same sign. If the sign is positive, it hits the triangle's front face, otherwise it hits the back face.

At the intersection point, $t\mathbf{d}$, the sum of the tetrahedra volumes $V_i(t\mathbf{d}) = \mathbf{n}_i \cdot (t\mathbf{d})$ equals the volume $V = \mathbf{p}_j \cdot \mathbf{n}_j$ (for $j = 0..2$) of the tetrahedron spanned by the triangle and the origin. Solving for t we get:

$$V = V_0(t\mathbf{d}) + V_1(t\mathbf{d}) + V_2(t\mathbf{d}) \\ t = V / (V_0(\mathbf{d}) + V_1(\mathbf{d}) + V_2(\mathbf{d})). \quad (2)$$

We can also write the intersection point, $t\mathbf{d}$, using barycentric coordinates as $t\mathbf{d} = \lambda_0\mathbf{p}_0 + \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2$ (Fig. 3d), which are in turn defined by the ratio of the scaled signed volumes, V_i , with $\lambda_i = V_i(\mathbf{d}) / (V_0(\mathbf{d}) + V_1(\mathbf{d}) + V_2(\mathbf{d}))$ [KS06]. Note that we do not need to compute the ray parameter t to test for intersections or to determine the barycentric coordinates, but we use it to compute the intersection coordinate and for the depth test.

Similar to 2D edge functions, we can compute the derivatives of the 3D edge functions with respect to screen space. Not only are $\mathbf{d}_x = \partial\mathbf{d}/\partial x$ and $\mathbf{d}_y = \partial\mathbf{d}/\partial y$ constant for planar views, but also $V_{i,x} = \partial V_i/\partial x$ and $V_{i,y} = \partial V_i/\partial y$:

$$V_i(\mathbf{d}) = \mathbf{n}_i \cdot \mathbf{d} = \mathbf{n}_i \cdot (\mathbf{d}_0 + x\mathbf{d}_x + y\mathbf{d}_y) \\ V_{i,x} = \frac{\partial V_i(\mathbf{d})}{\partial x} = \mathbf{n}_i \cdot \mathbf{d}_x \quad \text{and} \quad V_{i,y} = \frac{\partial V_i(\mathbf{d})}{\partial y} = \mathbf{n}_i \cdot \mathbf{d}_y. \quad (3)$$

These derivatives can be used as in 2D for defining rendering consistency (Sect. 4.2) and for incremental evaluation of the edge functions. However, the derivatives of the barycentric coordinates, λ_i , are not constant due to the perspective projection. Once an intersection is found we can easily compute t and λ_i for per-pixel texturing and shading.

Using 3D rasterization It is important to note that although 3D rasterization is based on 3D edge functions it still requires 2D evaluations only. This is because the edge evaluations per pixels boil down to computing $V_i(\mathbf{d}_0) + x \cdot V_{i,x} + y \cdot V_{i,y}$ for each edge given a pixel (x, y) .

Whether a direct or incremental evaluation is preferred depends on the renderer's architecture and binning strategy. In our implementations we used the direct evaluation throughout as it is more flexible, only marginally slower, and less prone to rounding errors.

3.3. Comparison of 2D and 3D Rasterization

In this section we briefly summarize the similarities and differences of hierarchical 2D rasterization (commonly used in software and hardware renderers) and 3D rasterization:

- Both use edge functions which can be incrementally or directly evaluated in image space to define the interior and exterior of triangles.
- 3D rasterization can have lower cost; Although the edge functions are defined in 3D, the evaluation takes place in 2D. Perspective correct computation of barycentric coordinates requires less operations than with 2D rasterization. Figure 10 in the appendix compares 2D and 3D rasterization with respect to arithmetic operations required for setup and evaluation of edge functions.
- No projection and clipping is required for 3D rasterization and thus it is not limited to planar viewports (see Sect. 4.1); Setup computation, i.e. computing V , $V_i(\mathbf{e} + \mathbf{d}_0)$, $V_{i,x}$, and $V_{i,y}$, only depends on the camera position.
- Due to operation in world space, 3D rasterization operates on floating point data as in ray tracing. Some of the consequences are discussed below.

3.4. A Rendering Pipeline with 3D Rasterization

Integrating 3D rasterization into the established rendering pipeline requires some modifications. 2D and 3D rasterization mainly differ in how they handle projection and viewing. Note that traditional viewing transformations can remain unchanged and, in this case, only modifications to the projective transformation and the rasterizer stage are necessary.

Projective Transformation 3D rasterization is very similar to generating primary rays in ray tracing. The projective matrix would be replaced by a new per-frame setup computing \mathbf{d}_x and \mathbf{d}_y (depending on the eye position \mathbf{e} and the field of view determined by \mathbf{d}_0) for the evaluation of edge functions.

Homogeneous Coordinates The traditional rendering pipeline uses homogeneous coordinates, primarily for perspective transformations, but other geometric calculations can be expressed elegantly using these coordinates. 3D rasterization as explained above works in Euclidian space and vertex coordinates need to be dehomogenized first. If we allow vertices, or the camera (for orthographic projections), to lie at infinity, an evaluation of the edge functions in homogeneous coordinates is necessary. In the appendix we demonstrate how to extend our method accordingly.

Hierarchical Rasterization 3D rasterization requires a different setup step for triangle rendering: A view-dependent computation of \mathbf{n}_i from its vertices and the eye position, \mathbf{e} , and computing the signed volumes and derivatives thereof.

Viewing Transformation Optionally we can also replace the canonical eye space transformation by a typical ray tracing camera. By this we can exploit the fact that setup of the triangle edge functions (Eq. 1) only depends on the eye position and not on the view direction. This can be beneficial when rendering multiple viewports for one camera position, e.g. when rendering cube environment maps. Note that when this modification is made the view and projective transformations can be combined into a single pipeline stage.

4. Benefits of 3D Rasterization

Our generalization allows us to use the same core routine for rendering images in rasterization and ray tracing of primary rays. As a consequence we can transfer rendering concepts from one to the other. We demonstrate rasterization with non-linear projections, and introduce rendering consistency and anti-aliasing techniques known from rasterization to ray tracing. Lastly, we discuss the continuum of rendering algorithms that can be explored using 3D rasterization.

4.1. Non-planar Viewports

The key to efficient rendering is a cheap computation of V_i for quickly testing for intersections and computing barycentric coordinates. We will discuss the hemispherical view obtained from the well-known parabolic mapping [HS98] as an example. The paraboloid function is $f(x, y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2)$, with $x^2 + y^2 \leq 1$, the direction vectors of the hemispherical viewport are $\mathbf{d} = (x, y, f(x, y))^T$, and the ray origin is $\mathbf{e} = (0, 0, 0)^T$. Both \mathbf{d} and $V_i(\mathbf{d}) = \mathbf{n}_i \cdot \mathbf{d}$ can be evaluated directly, but as they are quadratic functions we can also use a double incremental scheme to compute their values. For an incremental evaluation of a 3D edge function, V_i , we initialize the computation for the pixel, (x_0, y_0) , and the corresponding direction vector, \mathbf{d}_0 , with $\mathbf{V} = V_i(\mathbf{d}_0)$, $\mathbf{V}_x = V_{i,x}(\mathbf{d}_0)$ and $\mathbf{V}_y = V_{i,y}(\mathbf{d}_0)$. When going from a pixel, (x, y) , to another pixel, $(x + \Delta x, y + \Delta y)$, we update:

$$\begin{aligned} \mathbf{V} &\leftarrow \mathbf{V} + V_i(\Delta x, \Delta y) - \frac{n_z}{2} + \Delta x(\mathbf{V}_x - n_x) + \Delta y(\mathbf{V}_y - n_y) \\ \mathbf{V}_x &\leftarrow \mathbf{V}_x - n_x \Delta x \quad \text{and} \quad \mathbf{V}_y \leftarrow \mathbf{V}_y - n_y \Delta y. \end{aligned} \quad (4)$$

Note that parabolic rasterization has also been demonstrated with GPUs [GHFP08] by determining 2D bounding shapes for the curved triangles followed by per-pixel ray-triangle intersections. 3D rasterization naturally handles non-linear projections (Fig. 1 shows two images generated with our method), but hierarchical binning is based on the assumption that edges are straight lines between the projected vertices (Fig. 5a). We avoid computing a 2D bounding shape,

as in [GHFP08], as this requires an additional set of edge functions in image space. Instead we propose to modify the locations where the edge functions are evaluated for binning. This is based on simple observations: The parabolic projection of a line $\overline{\mathbf{p}_0\mathbf{p}_1}$ is a circle arc with radius $r = \|\mathbf{n}_z^{-1}\| \geq 1$, with $\mathbf{n} = (\mathbf{p}_0 \times \mathbf{p}_1) / \|\mathbf{p}_0 \times \mathbf{p}_1\|$; the center is at $(n_x/n_z, n_y/n_z)^T$ [GHFP08]. A bin, of size $m \times m$, fails to detect the intersection of a *convex edge* if the circle intersects one side of the binning region only. We can compute the maximum penetration depth of the circle which would still not be detected (Fig. 5b) by $p = r - \sqrt{r^2 - m^2}/4$ (a computationally simpler, tight bound for p is $p \leq m(1 - \sqrt{3}/2)$, as $r \geq 1$). By virtually shifting the bin corners towards the circle center by p we obtain a conservative intersection test with the curved edge. Note that we only need to shift one side of the bin, determined by the largest magnitude coordinate of the vector \mathbf{s} going from the bin center to the circle center. Lastly, if $r < m/2$ the circle might lie entirely inside the bin and we always need to split and recursively test the bin for triangle coverage.

We estimated the overhead for binning with this algorithm for paraboloid mapping by rendering randomly generated triangles. Approximately 18% of the bins have been refined, although they did not intersect a triangle. We used the conservative upper bound for p precomputed for fixed bin sizes, thus introducing only marginal computational overhead. Similar binning strategies can be developed for other non-linear projections by geometric reasoning.

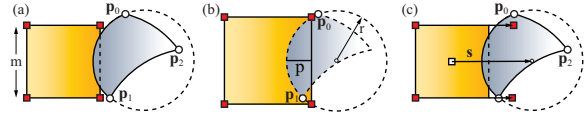


Figure 5: Binning evaluates the edge functions at the four corners (red squares) and thus misses intersections with curved edges which are circular arcs (a). We compute the maximum penetration of convex edges (b), and shift the locations where each of the edge functions is evaluated (c).

4.2. Rasterization Consistency

One important feature of 2D rasterization is that *consistency rules* can be defined. They ensure that each pixel intersecting adjacent triangles is rasterized exactly once. This is important to avoid holes and incorrect blending when rendering with semitransparent materials. A common rule, which is used by OpenGL and Direct3D, is the top-left filling convention with the pixel center as the decisive point. If it resides on an edge then it belongs to the triangle to its right, or below if the edge is horizontal [Mic06].

For 3D rasterization we adopt the same consistency rules. We determine the triangle a pixel belongs to based on the barycentric coordinates. If a pixel center lies on a triangle edge, i.e. $\lambda_i = 0$, we base the decision on the derivatives

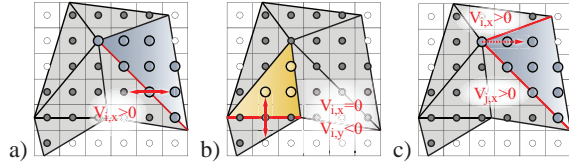


Figure 6: Rasterization consistency: we adopt the top-left filling convention from OpenGL and Direct3D for 3D rasterization based on the barycentric coordinates and the derivatives of V_i .

of V_i . For an edge, as shown in red in Fig. 6a, the derivative $V_{i,x}$ is positive for one triangle only, and we assign the pixel to that triangle. If the edge is horizontal ($V_{i,x} = 0$), we use the derivative $V_{i,y}$ as the secondary criterion of the top-left convention (Fig. 6b). Similarly, we decide for pixels centered on vertices: The pixel belongs to the triangle to its right (Fig. 6c), or to the one below if it resides on a horizontal edge, i.e. it belongs to the triangle for which $(V_{i,x} > 0 \wedge V_{j,x} > 0) \vee (V_{i,x} = 0 \wedge V_{j,x} > 0)$, with $\lambda_i = \lambda_j = 0$. Note that both derivatives are zero only for edges collapsed to a point, and thus for degenerated triangles which are not rasterized.

So far no fully consistent ray-triangle intersection algorithm is known for ray tracing: A ray-triangle intersection is assumed to happen if $0 \leq \lambda_i \leq 1$ for the three barycentric coordinates. This can lead to inconsistencies at shared edges, or vertices, of adjacent triangles. If multiple intersections are to be avoided, a scene-dependent epsilon distance along the ray is chosen by the user such that only one intersection may occur. By using 3D rasterization as a replacement for ray-triangle intersection of primary rays, the consistency rules naturally transfer to ray tracing and enable fully consistent rendering. Using these consistency rules we did not spot any precision problems with direct evaluation or incremental evaluation for small bin sizes.

4.3. Anti-aliasing

Anti-aliasing is crucial for high-quality image synthesis, but it is computationally expensive and requires a lot of memory and bandwidth. Most renderers use super-sampling, i.e. they actually compute the image at a higher resolution and down-sample the color buffer. Modern graphics hardware goes one step further and decouples coverage sampling (determining what fraction of a pixel is covered by a triangle) from shading computation: The coverage of sub-samples can be efficiently computed in 2D and 3D rasterization while shading is often the bottleneck. In this spirit, we added multi-sampling and anti-aliasing with coverage samples (CSAA) [NVI08] to our 3D rasterization framework (see Fig. 7).

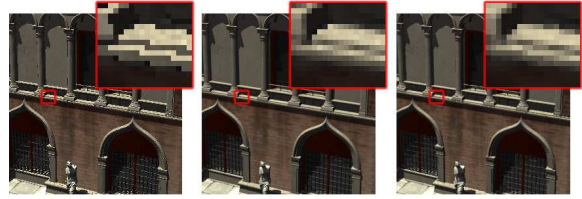


Figure 7: We also integrated anti-aliasing techniques into 3D rasterization: The rendering time using the 3DR-FULL algorithm (see Sect. 5) was 930 ms without anti-aliasing (left), 10270 ms for $16 \times$ MSAA (center), and 4022 ms for CSAA [NVI08] with 4 shading and 12 coverage samples (right). Shading is computed using recursive ray tracing with 1 shadow ray, and 1 reflection ray for every sample.

4.4. The 3D Rasterization Continuum

With 3D rasterization, the scene traversal is left as the main difference between rasterization and ray tracing of primary rays. This forms a continuous space with each at an opposite end. While pure rasterization must enumerate all triangles in a scene in arbitrary order, traditional ray tracing traverses the scene for every ray, each time enumerating only the minimum number of possible overlapping triangles.

A known intermediate sample point is frustum traversal of a spatial index structure [RSH05], enumerating all primitives of the scene overlapping the frustum. On the other hand, given a frustum of rays (e.g. the entire screen, a tile, or a single pixel), we must efficiently compute their intersections with the triangles by the frustum (i.e. coverage computation).

This leads us to the following combined approach (see Fig. 8), where F is a frustum and N is a node of the spatial index structure (typically starting with the entire viewport and the root node, respectively). The blue keywords denote oracles which control the behaviour of the algorithm. For ex-

```

isOccluded or isOutside ) return; // Fig. 9a
  if ( splitFrustum ) { // Fig. 9b
    split F into sub-frusta Fi // Fig. 9c
    foreach ( Fi ) traverse( Fi, N )
  } else
  if ( generateSamples ) {
    rasterize( N, binning ) // Fig. 9d
  } else {
    foreach ( child of N )
      traverse( F, child of N )
  }
}

```

Figure 8: This continuous rendering algorithm allows us to explore the continuum of methods between rasterization and ray tracing. Note that the `rasterize` function can be replaced by a ray-triangle intersections, or 2D rasterization for linear projections.

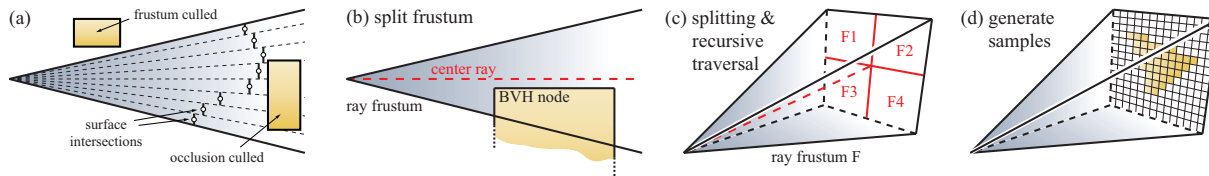


Figure 9: The continuous rendering algorithm traverses frusta through a spatial index structure, and determines the frustum samples covered by triangles. It consists of three operations: frustum and occlusion culling (a), frustum splitting (b, c), and sample generation (d).

ample, the algorithm behaves like a standard 2D rasterizer if we fix `generateSamples` and `binning` as true, and all other oracles as false. To obtain a coherent ray tracer, `isOccluded` and `isOutside` perform a test to determine if a spatial index node intersects a ray frustum or is occluded by other geometry; `splitFrustum` is false; The `generateSamples` oracle controls the traversal of the spatial index structure and is true for leaves only, where it starts a per-pixel ray-triangle intersection computation instead of rasterizing the triangles.

We can now combine concepts from both ends of the continuum and use occlusion and frustum culling based on the spatial index hierarchy with binning at the rasterization level as an additional 2D acceleration structure. We also experimented with adaptively splitting ray frusta, which can be beneficial for scenes with an irregular distribution of detail.

5. Evaluation, Results and Discussion

For a meaningful evaluation of 3D rasterisation (3DR) we conducted numerous experiments running on CPUs and GPUs, and compare 3DR to highly optimized 2D rasterization (2DR) and ray tracing implementations.

First, we developed a prototype CPU implementation of 3D rasterization and all components required to explore the 3D rasterization continuum outlined above. In order to assess the performance of the resulting rendering algorithms we compare this implementation to state-of-the-art highly-coherent ray tracing on CPUs [WBS07] implemented in RTfact [GS08], and to a SSE hand-optimized 2D rasterization algorithm. For the latter we implemented two versions, one evaluating all pixels inside a triangle’s 2D bounding box (2DR-BB), and one with hierarchical binning starting from the triangle’s bounding box and using early-z-culling (2DR-BIN).

In between these two extremes, we examined the following sample points of the continuum:

- Pure 3D rasterization with brute-force evaluation of all pixels inside a triangle’s bounding box (3DR-BB).
- Pure 3D rasterization with early-z-culling and binning identical to the 2DR binning strategy (3DR-BIN).

- 3DR with frustum culling: We subdivide the screen into frusta of 256^2 pixels and use a bounding volume hierarchy (BVH) to facilitate view frustum culling. Rasterization uses binning when leaf-nodes of the BVH are selected for rendering (3DR-FC).
- 3DR without binning, but instead using occlusion culling and frustum splitting (3DR-FS): A frusta is recursively split, down to 8^2 pixels, if a frustum’s center ray misses a BVH node’s bounding box (Fig. 9b, c), or if the box is small compared to the frustum.
- Combining the two previous options using occlusion culling and adaptive frustum splitting first, followed by 3DR with binning and early-z-culling (3DR-FULL).

Table 1 summarizes our benchmarks. We give measured frame times in milliseconds (ms), for the frame shown as inset, in order to compare the relative performance in a CPU software implementation. To allow a prediction of hardware speed we also report the number of triangle setups, edge function evaluations, and frustum vs. BVH-node intersections. The BVH was generated off-line using a SAH metric [WBS07] taking between 1 second to 1 minute for our test scenes. Building such acceleration structures for dynamic scenes is an active, yet orthogonal, research area; for recent work see Lauterbach et al. [LGS*09].

We observe that 3DR-BB is faster than 2DR-BB in most scenes, although we typically evaluate edge functions more often. The latter is because we cannot always determine tight screen space bounding boxes in 3DR for triangles that would be clipped in 2D rasterization, and we fall back to binning starting from the entire screen in these cases. This also explains why 2DR-BB is faster in scenes with fewer, large triangles, such as the Sponza and the Conference scenes. The overhead of hierarchical binning (3DR-BIN and 2DR-BIN) apparently does not amortize in our CPU implementations for the average triangle size in our test scenes. 3D rasterization with BVHs achieves a performance comparable to the highly optimized RTfact implementation and 3DR-FS/3DR-FULL even beat it for the Buddha and Teapots scenes which exhibit a large number of very small triangles. This is due to the adaptive frustum splitting; for the Teapots2 scene the early-z culling in 3DR-FULL provides an additional small speed-up.







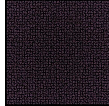
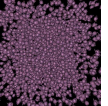
								
	Venice T=1.236k	Conference T=191k	Soda Out T=2.169k	Soda In T=2.169k	Sponza T=67k	Buddha T=1.088k	Teapots 1 T=2.332k	Teapots 2 T=2.332k
RTfact	203ms	135ms	142ms	105ms	97ms	512ms	1796ms	857ms
2DR-BB	519ms S=438k E=9817k	97ms S=45k E=2340k	994ms S=1444k E=10008k	786ms S=631k E=11546k	92ms S=26k E=3912k	464ms S=563k E=2376k	962ms S=1231k E=4763k	1053ms S=1176k E=9547k
2DR-BIN	535ms S=438k E=5975k	105ms S=45k E=1586k	1021ms S=1444k E=9345k	818ms S=631k E=8839k	93ms S=26k E=2000k	467ms S=563k E=2376k	957ms S=1231k E=4763k	1052ms S=1176k E=9547k
3DR-BB	393ms S=633k E=9044k	138ms S=87k E=2599k	618ms S=1082k E=8107k	591ms S=1064k E=14625k	166ms S=38k E=6806k	310ms S=532k E=1912k	598ms S=1217k E=3964k	699ms S=1181k E=7894k
3DR-BIN	1310ms S=633k E=17441k	318ms S=87k E=3367k	2084ms S=1082k E=19720k	1609ms S=1064k E=14259k	235ms S=38k E=2067k	1470ms S=532k E=20873k	2963ms S=1217k E=46685k	3135ms S=1181k E=41542k
3DR-FC	452ms S=380k E=4775k N=257k	186ms S=46k E=3648k N=34k	915ms S=1107k E=4443k N=821k	340ms S=500k E=1519k N=377k	130ms S=29k E=1415k N=21k	831ms S=540k E=11051k N=393k	1751ms S=1234k E=28439k N=829k	1793ms S=1221k E=16321k N=842k
3DR-FS	381ms S=595k E=12223k N=622k	293ms S=475k E=8709k N=566k	281ms S=370k E=6139k N=646k	258ms S=156k E=4214k N=279k	216ms S=195k E=4923k N=233k	387ms S=808k E=12935k N=598k	907ms S=1959k E=31347k N=1778k	796ms S=1543k E=24694k N=1958k
3DR-FULL	287ms S=266k E=9085k N=252k	218ms S=168k E=6136k N=184k	259ms S=370k E=3327k N=646k	164ms S=57k E=1877k N=102k	142ms S=69k E=1721k N=83k	403ms S=808k E=10187k N=598k	960ms S=1959k E=26368k N=1778k	768ms S=1543k E=14492k N=1958k

Table 1: Rendering statistics for the algorithms explained in Sect. 5 measured using one core of an Intel C2Q 9300 at 2.67GHz. Timings are given in milliseconds (ms) for the frames shown above, however we render primary rays with constant colors at 1024×1024 resolution without anti-aliasing only (lighting and secondary rays are not included in the timings as they are independent from the method used for primary rays). We also give the number of triangles (T), triangle setups (S), edge evaluations (E), and nodes visited during BVH traversal (N). Note that numbers are given in thousands (k) and all methods use backface-culling.

With regard to future graphics hardware, such as Intel’s Larrabee, we ran a comparison of the basic 2D and 3D rasterization algorithms “close to the metal” (see also Appendix and Figure 10). We implemented both algorithms as Direct3D shaders running on a GPU, and emulate the software-rasterization of a triangle by letting the GPU render a square. In both cases a vertex shader carries out the setup (we omitted clipping for 2DR), and a pixel shader performs per-pixel edge function evaluations and computes perspective correct depth and barycentric coordinates. In order to avoid GPU intricacies and to compare pure rasterization performance we disabled depth-buffering and output all results color-coded. The shaders were compiled using vertex/pixel shader 3.0 profiles. On an NVIDIA GeForce 8800GTX 3DR outperforms 2DR (homogeneous 2DR) for 262144 triangles with 153.9 to 120.6 (120.7) frames per second (fps), and with 1155 to 1073 (1079) fps for 256 triangles (both rendered at 1920×1200 resolution, triangles placed next to each other such that they fill the screen). Note that 2DR normally requires an additional clipping step that we omitted.

6. Conclusions and Future Work

Ray tracing and rasterization have long been considered as two distinct rendering approaches. We showed that by making a slight change that extends triangle edge functions to operate in 3D, the two approaches become almost identical with respect to primary rays. This yields a new rasterization technique which is faster than traditional 2D rasterization, and requires less operations for setup and evaluation. The accomplished similarity further allows us to transfer rendering concepts between both rasterization and ray tracing. We presented a generic algorithm that bridges both worlds and opens up a continuum with numerous possibilities for future research, allowing us to explore and compare new rendering methods.

We also believe that 3D rasterization makes the rendering pipeline – in future unified software rendering pipeline architectures – simpler and more elegant, but we aim for further generalization. In particular, a parameterization which allows for incremental computation, not only for the ray direction, but also the ray origin. This has a vast number of

applications in rendering, such as the rendering of global effects requiring secondary rays that are intricate to handle in rasterization, and expensive to compute in ray tracing.

Acknowledgements

We would like to thank Veronica Sundstedt for the model of the Ancient Egyptian Temple of Kalabsha.

Appendix

In this appendix we demonstrate how 3D rasterization, as introduced in Section 3, can be extended to homogeneous coordinates. Again, we start from the tetrahedron spanned by the triangle itself, and the origin of the rays, i.e. the camera or eye position. Given a triangle with vertices $\mathbf{V}_i = (w_i V_i, w_i)^T$, $i = 1, 2, 3$, and $w_i \neq 0$, an eye position $\mathbf{V}_e = (w_e V_e, w_e)^T$, and a view direction $\mathbf{d} = (d, 0)^T$, a point $\mathbf{V}(t) = \mathbf{V}_e + t\mathbf{d}$ lying in the plane of the triangle (in view direction, i.e. $t > 0$) can be written as:

$$\mathbf{V}(t) = \gamma_1(t)\mathbf{V}_1 + \gamma_2(t)\mathbf{V}_2 + \gamma_3(t)\mathbf{V}_3 + \gamma_e(t)\mathbf{V}_e, \quad (5)$$

with $\gamma_e(t) = 0$.

This is equivalent to a linear system of equations:

$$\begin{pmatrix} w_e V_{e,x} + t d_x \\ w_e V_{e,y} + t d_y \\ w_e V_{e,z} + t d_z \\ w_e \end{pmatrix} = \begin{pmatrix} w_1 V_{1,x} & w_2 V_{2,x} & w_3 V_{3,x} & w_e V_{e,x} \\ w_1 V_{1,y} & w_2 V_{2,y} & w_3 V_{3,y} & w_e V_{e,y} \\ w_1 V_{1,z} & w_2 V_{2,z} & w_3 V_{3,z} & w_e V_{e,z} \\ w_1 & w_2 & w_3 & w_e \end{pmatrix} \begin{pmatrix} \gamma_1(t) \\ \gamma_2(t) \\ \gamma_3(t) \\ \gamma_e(t) \end{pmatrix}. \quad (6)$$

We denote the determinant of a 4×4 matrix with column vectors \mathbf{V}_i , \mathbf{V}_j , \mathbf{V}_k , and \mathbf{V}_l as \mathcal{D}_{ijkl} :

$$\mathcal{D}_{ijkl} = \begin{vmatrix} \mathbf{V}_i & \mathbf{V}_j & \mathbf{V}_k & \mathbf{V}_l \end{vmatrix}.$$

Solving the system of equations with Cramer's rule yields:

$$\begin{aligned} \gamma_1(t) &= t \frac{\mathcal{D}_{d23e}}{\mathcal{D}_{123e}}, & \gamma_2(t) &= t \frac{\mathcal{D}_{1d3e}}{\mathcal{D}_{123e}}, \\ \gamma_3(t) &= t \frac{\mathcal{D}_{12de}}{\mathcal{D}_{123e}}, & \gamma_e(t) &= \frac{(\mathcal{D}_{123e} + t\mathcal{D}_{123d})}{\mathcal{D}_{123e}}. \end{aligned} \quad (7)$$

From $\gamma_e(t) \stackrel{!}{=} 0$ follows that $(\mathcal{D}_{123e} + t\mathcal{D}_{123d}) \stackrel{!}{=} 0$, and thus:

$$t = -\frac{\mathcal{D}_{123e}}{\mathcal{D}_{123d}}.$$

By substituting $t = -\frac{\mathcal{D}_{123e}}{\mathcal{D}_{123d}}$ into Eqs. 7 we get:

$$\gamma_1 = -\frac{\mathcal{D}_{d23e}}{\mathcal{D}_{123d}}, \quad \gamma_2 = -\frac{\mathcal{D}_{1d3e}}{\mathcal{D}_{123d}}, \quad \gamma_3 = -\frac{\mathcal{D}_{12de}}{\mathcal{D}_{123d}}. \quad (8)$$

Using γ_1 , γ_2 , and γ_3 with the bottom row of Eq. 6 yields:

$$w_1 \gamma_1 + w_2 \gamma_2 + w_3 \gamma_3 = w_e, \text{ and}$$

$$\mathcal{D}_{123d} = -\frac{1}{w_e} [w_1 \mathcal{D}_{d23e} + w_2 \mathcal{D}_{1d3e} + w_3 \mathcal{D}_{12de}]. \quad (9)$$

Using Eq. 8 we can compute the barycentric coordinates (in homogeneous space) using four determinants. Through simple modifications, we obtain a reformulation using only three determinants which reduces the triangle setup cost during rasterization:

$$\begin{aligned} \gamma_1 &= w_e \frac{\mathcal{D}_{d23e}}{w_1 \mathcal{D}_{d23e} + w_2 \mathcal{D}_{1d3e} + w_3 \mathcal{D}_{12de}} \\ \gamma_2 &= w_e \frac{\mathcal{D}_{1d3e}}{w_1 \mathcal{D}_{d23e} + w_2 \mathcal{D}_{1d3e} + w_3 \mathcal{D}_{12de}} \\ \gamma_3 &= w_e \frac{\mathcal{D}_{12de}}{w_1 \mathcal{D}_{d23e} + w_2 \mathcal{D}_{1d3e} + w_3 \mathcal{D}_{12de}}. \end{aligned} \quad (10)$$

Since all operations are carried out on homogenous coordinates, γ_1 , γ_2 , γ_3 , and t are given in homogenous space as well. The corresponding barycentric coordinates γ'_i , $i = 1, 2, 3$, and the depth t' in real space are obtained by:

$$\gamma'_1 = w_1 \gamma_1, \quad \gamma'_2 = w_2 \gamma_2, \quad \gamma'_3 = w_3 \gamma_3, \quad t' = \frac{t}{w_e}. \quad (11)$$

Orthographic Projection

3D rasterization also handles orthographic projections. We rewrite Eq. 5 using the view direction \mathbf{d} instead of the eye position and solve:

$$\mathbf{V}(t) = \gamma_1(t)\mathbf{V}_1 + \gamma_2(t)\mathbf{V}_2 + \gamma_3(t)\mathbf{V}_3 + \gamma_d(t)\mathbf{d}, \quad (12)$$

with $\gamma_d(t) = 0$.

Again $\mathbf{V}(t) = \mathbf{V}_e + t\mathbf{d}$, but in this case \mathbf{V}_e is a point on the image plane, i.e. $\mathbf{V}_e = (x, y, 1)^T$. Analogous to the perspective projection we apply Cramer's rule and obtain:

$$\begin{aligned} \gamma_1(t) &= \frac{\mathcal{D}_{e23d}}{\mathcal{D}_{123d}}, & \gamma_2(t) &= \frac{\mathcal{D}_{1e3d}}{\mathcal{D}_{123d}}, \\ \gamma_3(t) &= \frac{\mathcal{D}_{12ed}}{\mathcal{D}_{123d}}, & \gamma_d(t) &= \frac{(\mathcal{D}_{123e} + t\mathcal{D}_{123d})}{\mathcal{D}_{123d}}. \end{aligned} \quad (13)$$

We now observe that $t = -\frac{\mathcal{D}_{123e}}{\mathcal{D}_{123d}}$, but expectedly γ_i , $i = 1, 2, 3$, no longer depend on the depth t . Analogously to Eq. 10 we obtain:

$$\begin{aligned} \gamma_1 &= \frac{w_e \mathcal{D}_{e23d}}{w_1 \mathcal{D}_{e23d} + w_2 \mathcal{D}_{1e3d} + w_3 \mathcal{D}_{12ed}} \\ \gamma_2 &= \frac{w_e \mathcal{D}_{1e3d}}{w_1 \mathcal{D}_{e23d} + w_2 \mathcal{D}_{1e3d} + w_3 \mathcal{D}_{12ed}} \\ \gamma_3 &= \frac{w_e \mathcal{D}_{12ed}}{w_1 \mathcal{D}_{e23d} + w_2 \mathcal{D}_{1e3d} + w_3 \mathcal{D}_{12ed}}. \end{aligned} \quad (14)$$

Rasterization

For rasterization of perspective views, we parameterize our direction vector over the image plane for pixels (x, y) as:

$$\mathbf{d} = \mathbf{d}_0 + x\mathbf{d}_x + y\mathbf{d}_y$$

and hence obtain:

$$\begin{aligned} \mathcal{D}_{d23e} &= \mathcal{D}_{d_023e} + x\mathcal{D}_{d_x23e} + y\mathcal{D}_{d_y23e} \\ \mathcal{D}_{1d3e} &= \mathcal{D}_{1d_03e} + x\mathcal{D}_{1d_x3e} + y\mathcal{D}_{1d_y3e} \\ \mathcal{D}_{12de} &= \mathcal{D}_{12d_0e} + x\mathcal{D}_{12d_xe} + y\mathcal{D}_{12d_ye}. \end{aligned} \quad (15)$$

For orthogonal views the observer position \mathbf{V}_e is parameterized over the image plane instead, and analogously we can derive expressions for the incremental update

$$\mathbf{V}_e = \mathbf{V}_{e_0} + x\mathbf{V}_{e_x} + y\mathbf{V}_{e_y}$$

and obtain:

$$\begin{aligned} \mathcal{D}_{e23d} &= \mathcal{D}_{e_023d} + x\mathcal{D}_{e_x23d} + y\mathcal{D}_{e_y23d} \\ \mathcal{D}_{1e3d} &= \mathcal{D}_{1e_03d} + x\mathcal{D}_{1e_x3d} + y\mathcal{D}_{1e_y3d} \\ \mathcal{D}_{12ed} &= \mathcal{D}_{12e_0d} + x\mathcal{D}_{12e_xd} + y\mathcal{D}_{12e_yd}. \end{aligned} \quad (16)$$

Homogeneous 3D Rasterization

The following pseudo code (following the Direct3D HLSL syntax) handles 3D rasterization for perspective views. Note that the setup code is generic and is not optimized for fixed camera positions, viewing directions etc. The same code can be using for orthographic projections. In this case \mathbf{VE} is replaced by the view direction, and D_0 , DX , and DY are replaced by VE_0 , VE_X , VE_Y (identical to Eqs. 15 and 16).

```
float det3( in float3 v0, in float3 v1, in float3 v2 ) {
    return dot( v0, cross( v1, v2 ) );
}

float det4( in float4 v0, in float4 v1,
            in float4 v2, in float4 v3 ) {
    float d = 0;
    d += v3.w * det3( v0.xyz, v1.xyz, v2.xyz );
    d -= v2.w * det3( v0.xyz, v1.xyz, v3.xyz );
    d += v1.w * det3( v0.xyz, v2.xyz, v3.xyz );
    d -= v0.w * det3( v1.xyz, v2.xyz, v3.xyz );

    return d;
}

// per-triangle rasterization setup for perspective 3D rasterization
// Input: camera position VE
// triangle vertices V1, V2, V3
// image parameterization D0, DX, DY
// Output: determinants in VV0, VV1, VV2
// homogeneous-w in W
void setup(
    in float4 VE, in float4 V1, in float4 V2, in float4 V3,
    in float4 D0, in float4 DX, in float4 DY,
    out float4 VV0, out float4 VV1, out float4 VV2,
    out float4 W
) {
    VV0 = float4(
        det4( DX, V2, V3, VE ), // D_{dx,2,3,e}
        det4( DY, V2, V3, VE ), // D_{dy,2,3,e}
        det4( D0, V2, V3, VE ), // D_{d0,2,3,e}
        det4( V1, V2, V3, VE ) ); // D_{1,2,3,e}

    VV1 = float4(
        det4( V1, DX, V3, VE ), // D_{1,dx,3,e}
        det4( V1, DY, V3, VE ), // D_{1,dy,3,e}
        det4( V1, D0, V3, VE ), // D_{1,d0,3,e}
        0 );

    VV2 = float4(
        det4( V1, V2, DX, VE ), // D_{1,2,dx,e}
        det4( V1, V2, DY, VE ), // D_{1,2,dy,e}
        det4( V1, V2, D0, VE ), // D_{1,2,d0,e}
        0 );

    W = float4( V1.w, V2.w, V3.w, VE.w );
}
```

```
// Per-pixel operations
// Input: outputs from setup
// pixel position vector xy1=(x,y,1)
void rasterize(
    in float4 VV0, in float4 VV1, in float4 VV2,
    in float4 W, in float3 xy1 )
{
    float4 gad; //= (gamma_1,gamma_2,gamma_3, depth)
    gad.x = dot( xy1, VV0.xyz ); // D_{d,2,3,e}
    gad.y = dot( xy1, VV1.xyz ); // D_{1,d,3,e}
    gad.z = dot( xy1, VV2.xyz ); // D_{1,2,d,e}
    gad.w = VV0.w; // D_{1,2,3,e}

    if( v.x < 0 && v.y < 0 && v.z < 0 )
    {
        float D123d = dot( gad.xyz, W.xyz );

        // gamma and depth values in homogenous space
        gad = ( gad * W.w ) / D123d;

        // project barycentrics and depth into real space
        gad = gad.xyzw * float4( W.xyz, 1.f / W.w );

        SetPixel( x, y, gad.x, gad.y, gad.z, gad.w );
    }
}
```

Side-by-side Comparison to 2D Rasterization

In Figure 10 we show a side-by-side comparison of the 3D rasterization method optimized for dehomogenized coordinates, 2D rasterization, and homogeneous 2D rasterization in Direct3D HLSL syntax. We give the number of scalar instruction required for setup, and per-pixel evaluation, and also the number of instruction slots when compiling the code using the vertex/pixel shader 3.0 profiles.

References

- [Ake93] AKELEY K.: Reality engine graphics. In *SIGGRAPH '93* (1993).
- [App68] APPEL A.: Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference* (1968).
- [FvDFH90] FOLEY J. D., VAN DAM A., FEINER S. K., HUGHES J. F.: *Computer Graphics: Principles and Practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1990.
- [GHFP08] GASCUEL J.-D., HOLZSCHUCH N., FOURNIER G., PÉROCHE B.: Fast non-linear projections using graphics hardware. In *S13D '08: Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games* (2008).
- [GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-buffer visibility. In *SIGGRAPH '93* (1993).
- [Gre96] GREENE N.: Hierarchical polygon tiling with coverage masks. In *SIGGRAPH '96* (1996).
- [GS08] GEORGIEV I., SLUSALLEK P.: Rtfact: Generic concepts for flexible and high performance ray tracing. In *Proceedings of the IEEE / EG Symposium on Interactive Ray Tracing 2008* (2008).
- [Hec89] HECKBERT P. S.: *Fundamentals of Texture Mapping and Image Warping*. Tech. rep., Berkeley, CA, USA, 1989.
- [HM08] HUNT W., MARK W. R.: Ray-specialized acceleration structures for ray tracing. In *IEEE/EG Symposium on Interactive Ray Tracing 2008* (Aug 2008).
- [HS98] HEIDRICH W., SEIDEL H.-P.: View-independent environment maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (1998).

2D Rasterizer	2D Homogeneous Rasterizer	3D Rasterizer (for cartesian coordinates)
Triangle Setup (assuming that the eye position is in the origin; a, b, c denote the triangle vertices):		
<pre>float4 v0 = mul(projMat, a); float4 v1 = mul(projMat, b); float4 v2 = mul(projMat, c); float w0 = v0.w; v0 /= w0; float w1 = v1.w; v1 /= w1; float w2 = v2.w; v2 /= w2; // 3 RCP, 9 MUL float3 D12 = cross(v1.xyw, v2.xyw); float3 D20 = cross(v2.xyw, v0.xyw); float3 D01 = cross(v0.xyw, v1.xyw); // 3*(3 MUL,3 MADD) float DET = dot(v0.xyw, D12); // 1 MUL, 2 MADD float3x3 InterpolationMatrix = // 9 MUL, 1 RCP float3x3(D12, D20, D01) / DET; float3 DEPTH = float3(v0.z, v1.z, v2.z) * InterpolationMatrix; // 3 MUL, 6 MADD float3 ONE_OVER_W = float3(1.0 / w0, 1.0 / w1, 1.0 / w2) * InterpolationMatrix; // 3 RCP, 3 MUL, 6 MADD</pre>	<pre>float4 v0 = mul(projMat, a); float4 v1 = mul(projMat, b); float4 v2 = mul(projMat, c); float3 D12 = cross(v1.xyw, v2.xyw); float3 D20 = cross(v2.xyw, v0.xyw); float3 D01 = cross(v0.xyw, v1.xyw); // 3*(3 MUL,3 MADD) float DET = dot(v0.xyw, D12); // 1 MUL, 2 MADD float3x3 InterpolationMatrix = // 9 MUL,1 RCP float3x3(D12, D20, D01) / DET; float3 DEPTH = float3(v0.z, v1.z, v2.z) * InterpolationMatrix; // 3 MUL, 6 MADD float3 ONE_OVER_W = float3(1.0, 1.0, 1.0) * InterpolationMatrix; // 6 ADD</pre>	<pre>// d0, dx, dy defines the camera float3 n0 = cross(b - c, c); // 3 MUL, 3 MADD float3 n1 = cross(c - a, a); // 3 MUL, 3 MADD float3 n2 = cross(a - b, b); // 3 MUL, 3 MADD float V = dot(n0, a); // 1 MUL, 2 MADD float V0 = dot(n0, d0); // 1 MUL, 2 MADD float V1 = dot(n1, d0); // 1 MUL, 2 MADD float V2 = dot(n2, d0); // 1 MUL, 2 MADD float V0x = dot(n0, dx); // 1 MUL, 2 MADD float V0y = dot(n0, dy); // 1 MUL, 2 MADD float V1x = dot(n1, dx); // 1 MUL, 2 MADD float V1y = dot(n1, dy); // 1 MUL, 2 MADD float V2x = dot(n2, dx); // 1 MUL, 2 MADD float V2y = dot(n2, dy); // 1 MUL, 2 MADD</pre>
Output of the setup stage:		
InterpolationMatrix(I12, I20, I01) DEPTH, ONE_OVER_W	InterpolationMatrix (I12, I20, I01) DEPTH, ONE_OVER_W	V0 = float4(V0x, V0y, V0, V); V1 = float3(V1x, V1y, V1); V2 = float3(V2x, V2y, V2);
Per-Triangle Arithmetic Operations / Instruction Slots		
23 MADD, 0 ADD, 34 MUL, 4 RCP = 55 Ops 1 DP3, 12 DP4, 7 MAD, 6 MOV, 11 MUL, 4 RCP = 41 1 DP3, 7 MAD, 3 MOV, 11 MUL, 4 RCP = 27	17 MADD, 6 ADD, 22 MUL, 1 RCP = 46 Ops 1 DP3, 12 DP4, 7 MAD, 6 MOV, 7 MUL, 1 RCP = 34 1 DP3, 7 MAD, 5 MOV, 7 MUL, 1 RCP = 21	29 MADD, 0 ADD, 19 MUL = 48 Ops 3 ADD, 10 DP3, 3 MAD, 2 MOV, 3 MUL = 21
Per-Pixel Operations for a pixel with coordinates (x,y), with xyl = float3(x, y, 1.0)		
<pre>float E01 = dot(I01, xyl); float E12 = dot(I12, xyl); float E20 = dot(I20, xyl); // 3*2 MADD if (E12 < 0 && E20 < 0 && E01 < 0) { float Z, W, u, v; W = 1.0 / dot(ONE_OVER_W, xyl); // 2 MADD, 1 RCP Z = dot(DEPTH, xyl) * W; // 2 MADD, 1 MUL u = E12 * W // 1 MUL v = E20 * W // 1 MUL result = float4(u, v, Z, 0.0); }</pre>	<pre>float E01 = dot(I01, xyl); float E12 = dot(I12, xyl); float E20 = dot(I20, xyl); // 3*2 MADD if (E12 < 0 && E20 < 0 && E01 < 0) { float Z, W, u, v; W = 1.0 / dot(ONE_OVER_W, xyl); // 2 MADD, 1 RCP Z = dot(DEPTH, xyl) * W; // 2 MADD, 1 MUL u = E01 * W; // 1 MUL v = E12 * W; // 1 MUL result = float4(u, v, Z, 0.0); }</pre>	<pre>float4 v = float4(dot(xyl, V0.xyz), dot(xyl, V1.xyz), dot(xyl, V2.xyz), V0.w); // 3*2 MADD if (v.x > 0 && v.y > 0 && v.z > 0) { float invV = 1.0f / dot(v.xyz, 1); // 1 RCP, 2 ADD v.xyw *= invV; // 3 MUL result = float4(v.x, v.y, v.w, 0.0); }</pre>
Per-Pixel Arithmetic Operations / Instruction Slots		
10 MADD, 3 MUL, 1 RCP = 14 Ops 1 DIV, 5 DP3, 4 CMP, 2 MUL, 1 MOV = 13	10 MADD, 3 MUL, 1 RCP = 14 Ops 4 CMP, 5 DP3, 1 MUL, 2 MOV, 1 RCP = 13	6 MADD, 2 ADD, 3 MUL, 1 RCP = 12 Ops 4 CMP, 4 DP3, 2 MOV, 1 MUL, 1 RCP = 12

Figure 10: Relative costs of 3D rasterization optimized for dehomogenized coordinates and (homogeneous) 2D rasterization [OG97] including perspective correct computation of depth and barycentric coordinates for interpolation: additions (ADD), multiplications (MUL) and reciprocals (RCP). Multiply-add (MAD) operations are used where possible to replace sequent MUL and ADD operations. Red numbers denote the number of scalar operations, blue denotes the number of instruction slots when the code is compiled using Direct3D's HLSL compiler for the vertex/pixel shader 3.0 profiles. The green number denote the setup cost for 2D (homogeneous) rasterization if the projection matrix multiplication is ignored. We assume that a camera transformation is applied that transforms the observer into the origin before the setup. Clipping cost is also ignored in this comparison.

- [HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (2007).
- [JLBM05] JOHNSON G. S., LEE J., BURNS C. A., MARK W. R.: The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. on Graph.* 24, 4 (2005).
- [KS06] KENSLER A., SHIRLEY P.: Optimizing ray-triangle intersection via automated search. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2006).
- [LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. *Computer Graphics Forum* 28, 2 (2009).
- [LKM01] LINDHOLM E., KILGARD M. J., MORETON H.: A user-programmable vertex engine. In *SIGGRAPH '01* (2001).
- [Mar08] MARK W.: Future graphics architectures. *Queue* 6, 2 (2008).
- [Mic06] MICROSOFT: Direct3D 10 Reference. Direct3D 10 graphics, <http://msdn.microsoft.com/directx>, 2006.
- [NVI08] NVIDIA: NVIDIA GPU programming guide. <http://developer.nvidia.com>, December 2008.
- [OG97] OLANO M., GREER T.: Triangle scan conversion using 2D homogeneous coordinates. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (1997).
- [O'R98] O'ROURKE J.: *Computational Geometry in C*. Cambridge University Press, New York, NY, USA, 1998.
- [PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. In *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2002)* (2002).
- [PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum (Proceedings of Eurographics)* 26, 3 (Sept. 2007).
- [PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
- [Pin88] PINEDA J.: A parallel algorithm for polygon rasterization. *Computer Graphics (Proceedings of SIGGRAPH '88)* 22, 4 (1988).
- [PMS*99] PARKER S. G., MARTIN W., SLOAN P.-P., SHIRLEY P., SMITS B., HANSEN C.: Interactive ray tracing. In *Proceedings of the 1999 symposium on Interactive 3D graphics and games* (1999).
- [RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2005)* 24, 3 (2005).
- [SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2008)* 27, 3 (2008).
- [WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2007)* 26, 1 (2007).
- [Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM* 23, 6 (1980).
- [WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2006)* 25, 3 (2006).
- [WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the Eurographics Symposium on Rendering* (2006).
- [WMG*07] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007* (2007).
- [WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001 (Proceedings of the 12th EUROGRAPHICS Workshop on Rendering)* (2001).
- [WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A programmable ray processing unit for realtime ray tracing. In *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2005)* (2005), vol. 24.