3D Rasterization: A Bridge between Rasterization and Ray Casting

Tomáš Davidovič[*]
Saarland Excellence Cluster MMCI
Saarland University

Thomas Engelhardt[†]
Karlsruhe Institute of Technology

Iliyan Georgiev[‡]
Saarland Excellence Cluster MMCI
Saarland University

Philipp Slusallek[§]
Saarland University
DFKI Saarbruecken

Carsten Dachsbacher[¶]
Karlsruhe Institute of Technology

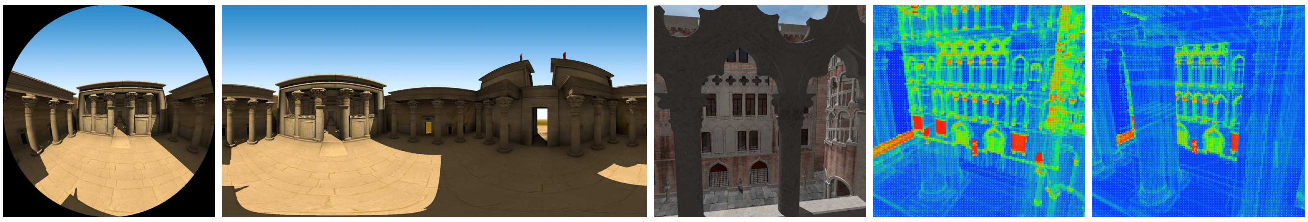

Fig. 1: With our approach, ray tracing and rasterization become almost identical with respect to primary rays. Now rasterization can directly render to non-planar viewports using parabolic and latitude-longitude parameterizations (left images), and we can transfer rendering consistency and efficient 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 binning and 3DR full, see Sect. 5).

Abstract

Ray tracing and rasterization have long been considered as two fundamentally different approaches to rendering images of 3D scenes, although they compute the same results for primary rays. Rasterization projects every triangle onto the image plane and enumerates all covered pixels in 2D, while ray tracing operates in 3D by generating rays through every pixel and then finding the closest intersection with a triangle. In this paper we introduce a new view on the two approaches: based on the Plücker ray-triangle intersection test, we define 3D triangle edge functions, resembling (homogeneous) 2D edge functions. Then both approaches become identical with respect to coverage computation for image samples (or primary rays). This generalized “3D rasterization” perspective enables us to exchange concepts between both approaches: we can avoid applying any model or view transformation by instead transforming the sample generator, and we can also eliminate the need for perspective division and render directly to non-planar viewports. While ray tracing typically uses floating point with its intrinsic numerical issues, we show that it can be implemented with the same consistency rules as 2D rasterization. With 3D rasterization the only remaining differences between the two approaches are the scene traversal and the enumeration of potentially covered samples on the image plane (binning). 3D rasterization allows us to explore the design space between traditional rasterization and ray casting in a formalized manner. We discuss performance/cost trade-offs and evaluate different implementations and compare 3D rasterization to traditional ray tracing and 2D rasterization.

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1 Introduction

The two main algorithms used in computer graphics for generating 2D images from 3D scenes are rasterization [30] and ray tracing [3, 40]. 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. The use of local, per-triangle computation makes it well suited for a feed-forward pipeline. However, 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 effects, which results in largely unpredictable memory accesses. Software-only ray tracing for interactive applications was assumed to be non-competitive in the beginning, such that ray tracing research essentially ceased during the 1990s. However, recent interactive ray tracing approaches achieved significant performance gains on both CPUs and GPUs [1, 6, 27, 28, 33, 39, 43] and ray tracing has been efficiently implemented in hardware [42].

In this paper we focus on primary (camera) rays, i.e. rays with a common origin or parallel rays, because only these are also covered by rasterization. We consider secondary rays and efficient global illumination algorithms, such as path tracing or photon mapping, as orthogonal to our approach. Even for the simple case of primary rays, rasterization and ray tracing seemed to be fundamentally different. In this paper, we show that a different view on both algorithms based on a slight generalization – rasterizing through 3D edge functions defined in world space instead of 2D edge functions on the image plane – we can fold ray casting, i.e. ray tracing of primary rays, and rasterization into a single *3D rasterization* algorithm. In this unified context we can now apply techniques that have been limited to one approach also in the context of the other.

In this new setting, the only difference between the two algorithms is the way potentially visible triangles and potentially covered image samples are enumerated. Traditionally, both approaches use different acceleration structures, e.g. ray tracing uses a 3D spatial index structure for approximate front to back traversal of the scene with implicit frustum and occlusion culling to only enumerate potentially visible triangles for coverage testing, and rasteriza-

[*] e-mail: davidovic@cs.uni-saarland.de
[†] e-mail: thomas.engelhardt@kit.edu
[‡] e-mail: georgiev@cs.uni-saarland.de
[§] e-mail: slusallek@dfki.de
[¶] e-mail: dachsbacher@kit.edu

tion uses a 2D grid of image samples that enables fast (hierarchical) identification of samples that may be covered by a triangle (cf. binning). Combinations of both worlds have been introduced before, e.g. Benthin et al. [4] use parallel traversal and binning in a multi-frustra ray tracing setting, and Hunt et al. [15] describe a continuum of *visibility algorithms* between tiled z-buffer systems and ray tracing. 3D rasterization enables us to freely explore many more combinations in a formalized manner and analyze the different trade-offs between these techniques. Recently, Laine and Karras [19] demonstrated that software 2D rasterization algorithms can be efficiently implemented on GPUs. We believe that their work creates a basis for rendering using unified 3D rasterization.

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 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 edge functions in image space, one for each triangle edge, whose sign determines which side of an edge is inside the triangle [30]. These functions can be evaluated in parallel, and hierarchically, to quickly locate relevant samples on the screen. Many extensions to this basic algorithm have been proposed including different traversal strategies [23], the hierarchical z-buffer [10], efficient computation of coverage masks [9], hierarchical rasterization in homogeneous coordinates [26], and the irregularly sampled z-buffer [16].

The idea of ray tracing was introduced to graphics by Appel [3], while Whitted [40] 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 [29]), and trying to reach performance comparable to rasterization. The latter is most often achieved by simultaneously tracing packets of coherent rays to increase performance on parallel hardware [39]. Recent approaches use large ray packets and optimized spatial index structures [1], such as kd-trees [33], bounding volume hierarchies (BVHs) [6, 27, 36], interval trees [42, 35], and 3D grid structures [17, 37]. A recent survey [38] gives an overview of construction and traversal algorithms for spatial index structures used for ray tracing.

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 [14, 31, 32] (we consider GPUs as programmable hardware dedicated to graphics that runs software implemented as shaders). However, the increasing parallelism and programmability of graphics processors make it likely that rendering algorithms will be mostly implemented in software in the future [19, 22, 34].

Implementations of specialized rasterization algorithms have been presented recently. Fatahalian et al. [5] discuss methods to efficiently rasterize micropolygons for high-quality rendering. Loop and Eisenacher [21] present a sort-middle rasterizer implemented in CUDA. Two papers are most closely related to this paper: Benthin et al. [4] perform synchronized traversal and binning in a multi-frustra tracing approach. Hunt et al. [15] 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 than both, as we fold ray casting and rasterization into a single algorithm enabling many more optimizations as well as exploration of the continuum of rendering algorithms between the two.

3 From 2D to 3D Rasterization

In the following we first briefly review the traditional 2D rasterization (2DR) technique [23, 30, 34] as it is implemented in current graphics hardware, before making the step to 3D. We then discuss the benefits and illustrate new possibilities.

3.1 2D Rasterization

Any linear function, u, on a triangle in 3D, e.g. colors or texture coordinates, obeys $u = aX + bY + cZ + d$, 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 [26]. 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 [12] 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.

We can now define three linear edge functions in the image plane for every triangle (Figure 2): Their values are equal to zero on two vertices and to one on the opposite vertex [26]. 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* [34], is a major factor for rendering performance. Typical 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-sample evaluation. Binning is analogous to culling a triangle from a frustum of primary rays in ray tracing.

Note that other rasterization algorithms exist, 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. Figure 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 method well-known as the Plücker ray-triangle intersection test [2, 4, 18]. We first consider the triangle formed by the first edge $\overline{\mathbf{p}_0\mathbf{p}_1}$ and the eye (see Figure 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 $\mathbf{n}_2 = \mathbf{p}_1 \times \mathbf{p}_0$. (In practice we use $\mathbf{n}_2 = (\mathbf{p}_1 - \mathbf{e}) \times (\mathbf{p}_0 - \mathbf{p}_1)$ which is mathematically equivalent but numerically more stable for small triangles.) We define the corresponding 3D edge function as $V_2(\mathbf{d}) = \mathbf{n}_2 \cdot \mathbf{d}$, which is equivalent to computing the scaled signed

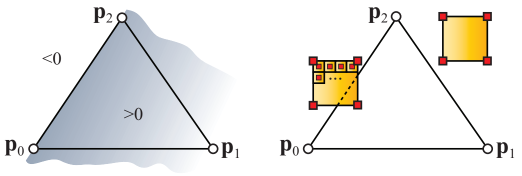

Fig. 2: Left: the 2D edge function for the edge $\overline{\mathbf{p}_0\mathbf{p}_2}$ which can be evaluated in parallel. A hierarchical search can quickly locate pixels covered by the triangle (right).

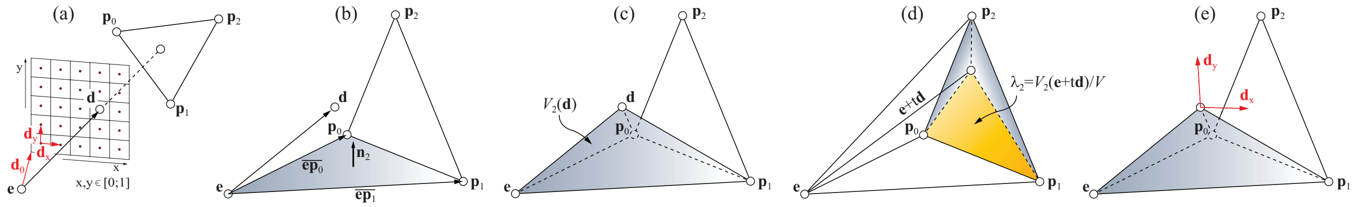

Fig. 3: 3D edge functions for a triangle $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$ and a ray starting at the eye $\mathbf{e}$ going through pixel (x, y) with direction $\mathbf{d}$ in world space. Intersection tests and computation of the barycentric coordinates are based on the signed volumes of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

volume of the tetrahedron with vertices $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{d}$, and $\mathbf{e}$ (the origin). The scaling factor of 6 (a factor of $1/6$ from tetrahedra volume formula) can be ignored as intersection tests are based on the signs and ratios of volumes. $V_2(\mathbf{d})$ is positive for all $\mathbf{d}$ in the half-space containing $\mathbf{p}_2$ (Figure 3c). In analogy we define V_0 and V_1:

$$\mathbf{n}_i = \mathbf{p}_{(i+2)\ mod\ 3} \times \mathbf{p}_{(i+1)\ mod\ 3}$$
$$V_i(\mathbf{x}) = \mathbf{n}_i \cdot \mathbf{x}, \text{ with } i = 0, 1, 2. \tag{1}$$

We denote the scaled volume of the tetrahedron spanned by the entire 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 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.

We can also write the intersection point, $t\mathbf{d}$, using barycentric coordinates as $t\mathbf{d} = \lambda_0\mathbf{p}_0 + \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2$ (Figure 3d), which are in turn defined by the ratio of the scaled signed volumes, V_i, with $\lambda_i = V_i(\mathbf{d})/(V_0(\mathbf{d}) + V_1(\mathbf{d}) + V_2(\mathbf{d}))$ [18]. Note that we do not need to compute the ray parameter t to test for intersections or to determine the barycentric coordinates (see below).

While the above has been widely used in ray tracing already, we can also compute the derivatives of the 3D edge functions with respect to screen space, similar to the derivatives of the 2D edge functions. 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 so are $V_{i,x} = \partial V_i/\partial x$ and $V_{i,y} = \partial V_i/\partial y$:

$$V_i(\mathbf{d}) = \mathbf{n}_i \cdot \mathbf{d} = \mathbf{n}_i \cdot (\mathbf{d}_0 + x\mathbf{d}_x + y\mathbf{d}_y) \tag{2}$$
$$V_{i,x} = \frac{\partial V_i(\mathbf{d})}{\partial x} = \mathbf{n}_i \cdot \mathbf{d}_x \quad \text{and} \quad V_{i,y} = \frac{\partial V_i(\mathbf{d})}{\partial y} = \mathbf{n}_i \cdot \mathbf{d}_y.$$

Where $\mathbf{d}_0$ is the base and $\mathbf{d}_x$, $\mathbf{d}_y$ are the derivatives of ray direction. These derivatives can be used as in 2D for defining rendering consistency (Section 3.4) 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.

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 pixel boil down to computing $V_i(\mathbf{d}_0) + xV_{i,x} + yV_{i,y}$ 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, fully parallel, and only marginally slower.

Discussion This intersection test is well-known as the Plücker test in ray tracing, which has been utilized by Woop [41] who pointed out the equivalence of Plücker tests and 2D homogeneous rasterization. Despite the fact that our results resemble 2D homogeneous rasterization (Fig. 10 provides a comparison chart), it should be noted, that homogeneous 2D rasterization requires transforming triangle vertices explicitly into the camera space. This step can be omitted in 3D rasterization, because viewing and perspective is folded directly into triangle setup. This might seem cumbersome for traditional rasterization pipelines, but is an inherent benefit for the formulation of the 3D rasterization continuum (Section 4).

3.3 Homogeneous Coordinates

The traditional rasterization pipeline uses homogeneous coordinates primarily for perspective transformations, but other geometric calculations can be also expressed using these coordinates. 3D rasterization, as described above, works in Euclidian space and thus vertex coordinates have to be dehomogenized first.

While it is possible to define the corresponding operations also using homogeneous coordinates [11], they are more expensive. As ray tracing has never used homogeneous coordinates without ill effects, we see little disadvantage in not supporting them for transformations. Note that projective texturing is completely independent from the rasterization of geometry. 3DR can handle homogeneous texture coordinates and projective texturing without any limitations.

Homogeneous coordinates can also be used to handle orthographic projections by moving the camera to infinity along the projection direction. In 3D rasterization we handle this case explicitly by computing the cross product between the projection direction $\mathbf{d}_p$ and the vector along the edge: $\mathbf{n}_i = \mathbf{d}_p \times (\mathbf{p}_{(i+1)\ mod\ 3} - \mathbf{p}_{(i+2)\ mod\ 3})$.

3.4 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 missing an intersection with either triangle along shared edges, or incorrect blending when rendering with semi-transparent materials and accounting for intersections with both triangles. 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 the one below in case the edge is horizontal [24].

The Plücker test as described by Amanatides and Choi [2] ensures consistency by counting rays hitting edge/vertex towards all triangles sharing the edge/vertex. While this works well for opaque materials, it can cause artifacts with semi-transparent materials. To ensure only one triangle is hit by each ray, we adopt the same consistency rules as 2D rasterization.

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 of V_i. For an edge, as shown in red in Fig. 4a, 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. 4b). Similarly, we decide for pixels centered on vertices: The pixel belongs to the triangle to its right (Fig. 4c), 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 two edges collapsed to a point and thus for degenerated triangles which should not be rasterized at all.

So far, no fully consistent ray-triangle intersection algorithm has been 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 edges or vertices shared between adjacent triangles. Today, the most widely used solution is the use of edge-based intersection algorithms (like the original Plücker test) that avoid most issues but cannot handle samples that intersect a point or edge exactly. The use of edge derivatives in 3D rasterization allows us to transfer the consistency rules from rasterization to ray tracing and enables fully consistent rendering.

Numerics Using these consistency rules we did not observe any sampling problems, neither with direct nor with incremental evaluation (for reasonably small bin sizes). Note that the formulations for computing the normal vector $\mathbf{n}_i$ are mathematically symmetric in the vertices. However, due to numerics, commutative operations do not necessarily yield the same results, e.g. $(\mathbf{e}-\mathbf{b}) \times (\mathbf{a}-\mathbf{b}) \neq -(\mathbf{e}-\mathbf{a}) \times (\mathbf{b}-\mathbf{a})$. This is because of matching the exponents that takes place in floating point addition and subtraction. Although rendering artifacts are very rare with 3D rasterization (as they are with ray tracing), this may lead to inconsistencies in edge function evaluations of adjacent triangles.

One solution, rather practical for hardware implementations, is to order vertices consistently using any criterion, e.g. always choosing the one with the smaller x-component as the first one in the computation. An alternative is to compute the cross-product for the normal vectors using the edge midpoints, which yields $(\mathbf{e}-(\mathbf{a}+\mathbf{b})/2) \times (\mathbf{a}-\mathbf{b}) = -(\mathbf{e}-(\mathbf{b}+\mathbf{a})/2) \times (\mathbf{b}-\mathbf{a})$. This approach is cheaper for software implementations compared to the consistent ordering. This ensures bitwise identical and thus consistent results even when using floating point arithmetic.

3.5 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 [13] 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:

$$\begin{aligned} \mathbf{V} &\leftarrow \mathbf{V} + V_i(\Delta x, \Delta y) - \frac{n_z}{2} + \Delta x(\mathbf{V}_x - n_z) + \Delta y(\mathbf{V}_y - n_z) \\ \mathbf{V}_x &\leftarrow \mathbf{V}_x - n_x \Delta x \text{ and } \mathbf{V}_y \leftarrow \mathbf{V}_y - n_y \Delta y. \end{aligned} \quad (3)$$

Note that parabolic rasterization has also been demonstrated with GPUs [7] by determining 2D bounding shapes for the curved triangles followed by per-pixel ray-triangle intersections. 3D rasterization naturally handles non-linear projections (Figure 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 (Figure 5a). We avoid computing a 2D bounding shape, as in [7], 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 $(n_x/n_z, n_y/n_z)^T$ [7]. 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 (Figure 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.

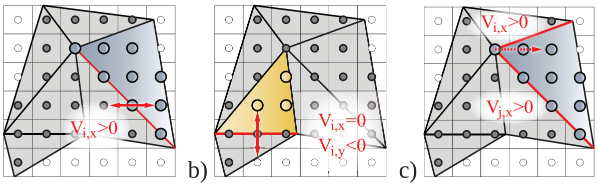

Fig. 4: Rasterization consistency: We adopt the top-left filling convention from OpenGL and Direct3D for 3D rasterization based on the barycentric coordinates and the derivatives of V_i.

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.

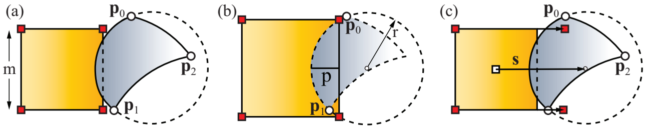

Fig. 5: Binning evaluates the edge functions at the four corners (red squares) and thus misses intersections with curved edges which are circular arcs (a). We compute the maximum penetration of convex edges (b), and shift the locations where each of the edge functions is evaluated (c).

4 The 3D Rasterization Continuum

With the unified evaluation of coverage for both ray tracing and rasterization via 3D rasterization, scene traversal and sample binning remain the only differences in handling primary rays.

Scene Traversal The design space for scene traversal is spanned by brute-force rasterization that simply processes every triangle in the scene on one side and traditional ray tracing of individual rays on the other one. In the latter case performance is optimized by building a spatial index structure, e.g. [39], and only computing coverage information (intersections) with triangles that are contained in cells intersected by the rays.

Note that the traditional rasterization pipeline does not comprise scene traversal, even if the scene data might already be stored on the graphics hardware, e.g. as OpenGL vertex buffer objects. However, frustum culling (using the entire view frustum, or smaller view frusta aligned with tiles for tile-based rasterization) are typically used to prevent triangles from being processed by the pipeline.

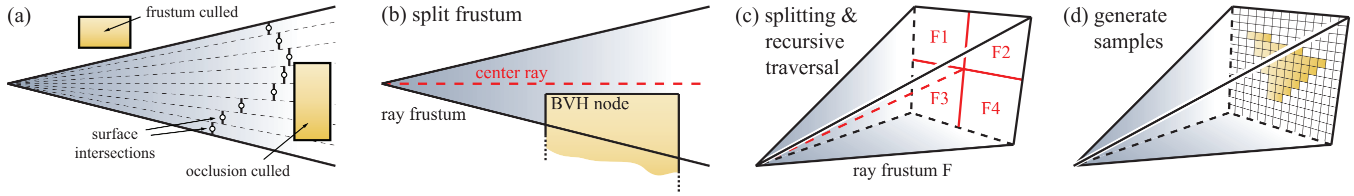

Fig. 6: The continuous rendering algorithm traverses frusta through a spatial index structure and determines the frustum samples covered by triangles. It consists of three operations: Frustum and occlusion culling (a), frustum splitting (b, c), and sample generation (d).

Newer OpenGL extensions provide means to query occlusion information from the GPU, or perform conditional submission of scene content, leaving the scene traversal to the application. A unified rendering approach like 3D rasterization suggests that scene traversal should be integrated more tightly with the core rendering algorithm on the GPU.

Binning Once a triangle has been chosen for rendering, we must efficiently identify the image samples/rays that it covers. For traditional ray tracing only triangles that are likely to be intersected are enumerated by the traversal. However, in case of larger frusta, a triangle may only cover a small fraction of the samples and it may be advantageous to use binning to accelerate finding the covered ones. For the same reason, rasterization has always used binning to quickly locate the covered samples on the screen.

```
traverse( frustum F, node N ) {
 if ( isOccluded or isOutside ) return;  // Figure 6a
 if ( splitFrustum ) {                   // Figure 6b
   split F into sub-frusta Fi            // Figure 6c
   foreach ( Fi ) traverse( Fi, N )
 } else
 if ( generateSamples ) {
   rasterize( N, binning )               // Figure 6d
 } else {
     foreach ( child of N )
       traverse( F, child of N )
} }
```

Fig. 7: 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 intersection, or 2D rasterization for linear projections.

Combined Use of Traversal and Binning A generic formulation of combined traversal and binning (not explicitly aligned) is shown as pseudo-code in Figure 7. Here 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 behavior of the algorithm, and allow us to reproduce the aforementioned rendering algorithms and to explore new avenues.

For example, the algorithm behaves like a standard 2D rasterizer if we set `generateSamples` and `binning` to true, and all other oracles to false. To obtain a coherent ray tracer, `isOccluded` and `isOutside` are activated to perform a test that determines 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 of the spatial index only, where it initiates the per-sample coverage computation instead of rasterizing the triangles.

This approach allows us to flexibly combine concepts from both ends of the continuum and use occlusion and frustum culling based on the spatial index hierarchy with binning at the coverage level using the sample grid additional 2D acceleration structure. We also experiment with adaptively splitting ray frusta, which can be beneficial for scenes with an irregular distribution of details.

More distinct combinations from this continuum have already been explored. For example, Benthin et al. [4] build on a 16-wide SIMD architecture traversing 16 frusta in parallel through a scene acceleration structure, each bounding a set of highly coherent rays. They also cull back-facing and missed triangles prior to ray-triangle intersection computation. We believe that our formulation of the full continuum allows us to explore other promising algorithms in a systematic way.

Anti-aliasing Many renderers use super-sampling for high-quality image synthesis, but 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. 3DR can be used with any such technique and we added multi-sampled anti-aliasing (MSAA) and anti-aliasing with coverage samples (CSAA) [25] to our framework (see Figure 8).

Anisotropic Texture-filtering On current GPUs screen-space derivatives for texture filtering are typically approximated by simple differencing in 2×2 pixel blocks (e.g. called ddx_coarse/ddy_coarse in Direct3D HLSL intrinsics). We can use exactly the same approach in 3DR without any additional cost compared to 2D rasterization. Exact derivatives would be marginally more expensive than with 2DR requiring one additional dot product per interpolated component per triangle; the per-pixel cost remains identical to 2DR.

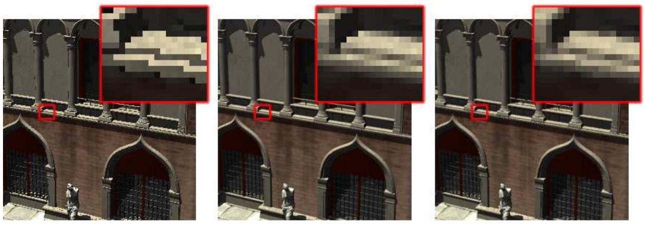

Fig. 8: We integrated anti-aliasing techniques into 3D rasterization: The rendering on the left uses the 3DR full algorithm (see Sect. 5) without any anti-aliasing, using 16× MSAA (center) takes 11× longer, while CSAA with 4 shading and 12 coverage samples (right) takes only 4.3× longer. Shading is computed using recursive ray tracing with 1 shadow ray, and 1 reflection ray for every sample.

5 Evaluation, Results and Discussion

For a meaningful evaluation of 3D rasterization (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 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 [36] implemented in RTfact [8], and to an 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 brute-force), and one

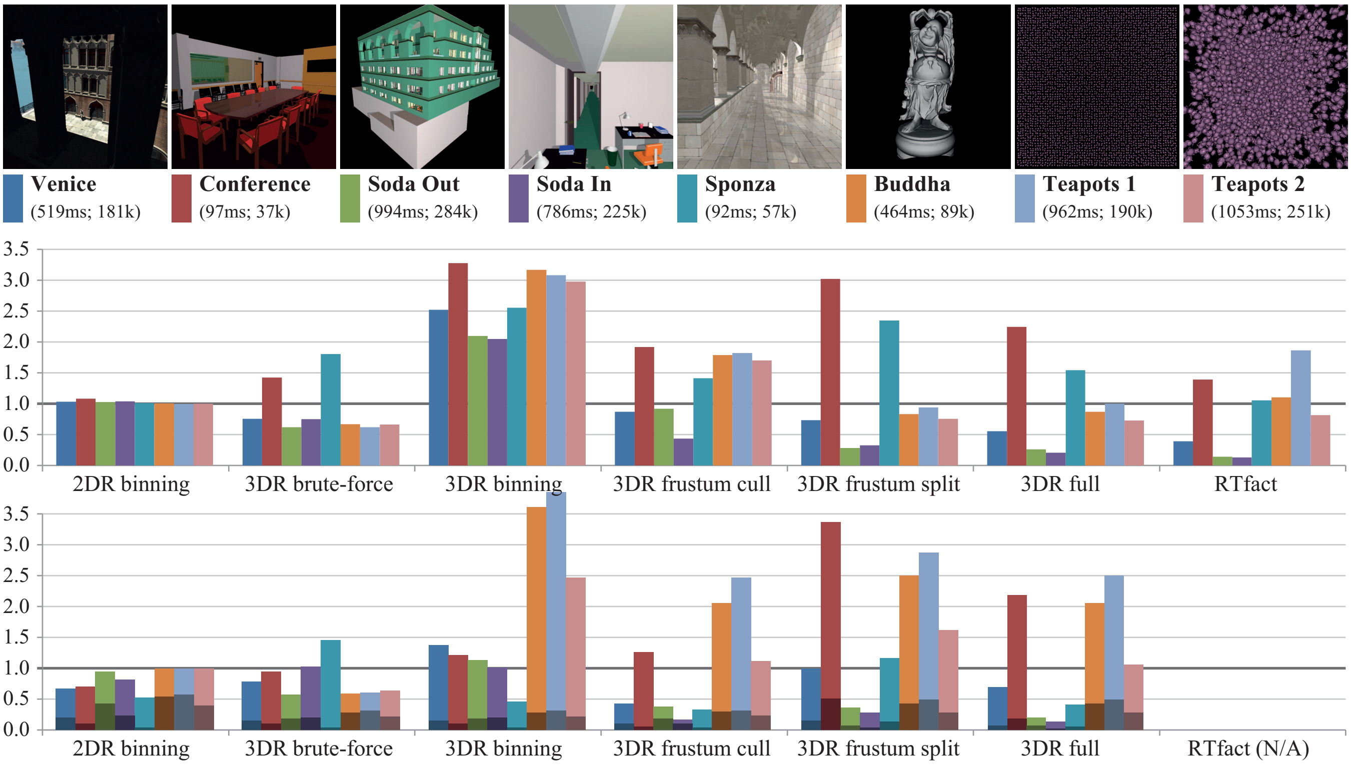

Fig. 9: Top graph: measured rendering time relative to 2DR brute-force (> 1 means slower). Bottom graph: Instructions per frame relative to 2DR brute-force computed from the instruction counts given in Figure 10 and the number of triangle setups and edge evaluations during rendering. The dark shade in each bar indicates the fraction of instructions due to triangle setups. The numbers in brackets for each scene are absolute performance and instructions per frame for 2DR brute-force.

with hierarchical binning starting from the triangle's bounding box and using early-z-culling (2DR binning) with a w-buffer. Please note that all our implementations use w-buffering for depth testing and do not rely on interpolated z-values. We use 2DR brute-force as our reference method.

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 brute-force).
- Pure 3D rasterization with early-z-culling (using a hierarchical w-buffer) and binning identical to the 2DR binning strategy (3DR binning).
- 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. 3D rasterization uses binning when leaf-nodes of the BVH are selected for rendering (3DR frustum cull).
- 3DR without binning, but instead using occlusion culling and frustum splitting (3DR frustum split): A frustum is recursively split, down to 8^2 pixels, if its center ray misses a BVH node's bounding box (Figure 6b, 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).

Figure 9 summarizes our benchmarks. For each frame in the inset we provide time in milliseconds and number of instructions it takes our reference method, 2DR brute-force, to render the frame. The upper bar graph then shows relative times for each method, in a CPU software implementation (anti-aliasing turned off). The lower bar graph shows relative instructions per frame for each method, to allow a prediction of hardware speeds. Instructions per frame are computed by counting the number of triangle setups and edge evaluations and multiplying them by scalar instruction count from Figure 10. Where required, we used BVHs constructed off-line using the SAH metric [36], taking between 1 second and 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. [20].

We observe that 3DR brute-force is faster than 2DR brute-force in most scenes, with the instructions per frame closely matching the actual speedup. The 2DR brute-force is faster only in the Sponza and the Conference scenes. Here, for a few triangles, the 3DR cannot determine tight screen-space bounding boxes, and falls back to testing on the whole screen. The overhead of hierarchical binning (3DR binning and 2DR binning) apparently does not amortize in our CPU implementations for the average triangle size in our test scenes. 3DR with BVHs achieves a performance comparable to the highly optimized RTfact implementation and 3DR frustum split/3DR full even beat it for the Buddha and Teapots scenes, which exhibit large numbers of very small triangles. This is due to the adaptive frustum splitting; early-z culling in 3DR full provides an additional small speed-up for the Teapots2 scene.

In Figure 10 we show a side-by-side comparison of 3D rasterization, traditional 2D rasterization, and homogeneous 2D rasterization [26]. The pseudo-code is laid out for scalar computing architectures and we report the number of scalar instructions required for setup and per-pixel evaluation. On scalar architectures, factoring out of constants is beneficial, and our 2D rasterization takes advantage of such an optimization, as we know $w = 1$ for all projected vertices (2nd for-loop). However, such optimizations can be impedimental on architectures that support vector instructions [19].

2D Rasterization	2D Homogeneous Rasterization (Olano [26])	3D Rasterization
Setup Input vertices: $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v_2}$ projection matrix: $\mathbf{M}_P$	**Setup Input** vertices: $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v_2}$ projection matrix: $\mathbf{M}_P$	**Setup Input** vertices: $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v_2}$ camera: $\mathbf{e}, \mathbf{d}_0, \mathbf{d}_x, \mathbf{d}_y$
for $i \leftarrow 0$ to 2 **do** $\mathbf{v}'_i \leftarrow \mathbf{M}_P \cdot \mathbf{v}_i$ $w_i \leftarrow \mathbf{v}'_i.w$ $\mathbf{v}'_i \leftarrow \mathbf{v}'_i / w_i$ **end for** →3 RCP, 12 MUL, 36 MADD **for** $i \leftarrow 0$ to 2 **do** $\mathbf{c}_{i,x} \leftarrow \mathbf{v}'_{(i+1)\%3,y} - \mathbf{v}'_{(i+2)\%3,y}$ $\mathbf{c}_{i,y} \leftarrow \mathbf{v}'_{(i+2)\%3,x} - \mathbf{v}'_{(i+1)\%3,x}$ $\mathbf{c}_{i,z} \leftarrow \mathbf{v}'_{(i+1)\%3,x}\mathbf{v}'_{(i+2)\%3,y} - \mathbf{v}'_{(i+1)\%3,y}\mathbf{v}'_{(i+2)\%3,x}$ **end for** →3 SUB, 3 MUL, 3 MADD $D \leftarrow \mathbf{c}_0 \cdot \mathbf{v}'_0.xyz$ 1 MUL, 2 MADD $\mathbf{M}_I \leftarrow \frac{1}{D} \cdot [\mathbf{c}_0\ \mathbf{c}_1\ \mathbf{c}_2]^T$ 1 RCP, 9 MUL $\mathbf{w} \leftarrow \left(\frac{1}{w_0}, \frac{1}{w_1}, \frac{1}{w_2}\right)^T$	$\mathbf{v}'_0 \leftarrow \mathbf{M}_P \cdot \mathbf{v}_0$ $\mathbf{v}'_1 \leftarrow \mathbf{M}_P \cdot \mathbf{v}_1$ $\mathbf{v}'_2 \leftarrow \mathbf{M}_P \cdot \mathbf{v}_2$ →36 MADD $\mathbf{c}_0 \leftarrow \mathbf{v}_1.xyw \times \mathbf{v}_2.xyw$ 3 MUL, 3 MADD $\mathbf{c}_1 \leftarrow \mathbf{v}_2.xyw \times \mathbf{v}_0.xyw$ 3 MUL, 3 MADD $\mathbf{c}_2 \leftarrow \mathbf{v}_0.xyw \times \mathbf{v}_1.xyw$ 3 MUL, 3 MADD $D \leftarrow \mathbf{c}_0 \cdot \mathbf{v}'_0.xyz$ 1 MUL, 2 MADD $\mathbf{M}_I \leftarrow \frac{1}{D} \cdot [\mathbf{c}_0\ \mathbf{c}_1\ \mathbf{c}_2]$ 1 RCP, 9 MUL $\mathbf{w} \leftarrow \mathbf{M}_I \cdot (1,1,1)^T$ 6 ADD	$\mathbf{n}_0 \leftarrow (\mathbf{v}_2 - \mathbf{e}) \times (\mathbf{v}_1 - \mathbf{e})$ $\mathbf{n}_1 \leftarrow (\mathbf{v}_0 - \mathbf{e}) \times (\mathbf{v}_2 - \mathbf{e})$ $\mathbf{n}_2 \leftarrow (\mathbf{v}_1 - \mathbf{e}) \times (\mathbf{v}_0 - \mathbf{e})$ → 9 ADD, 9 MUL, 9 MADD $V \leftarrow \mathbf{n}_0 \cdot (\mathbf{v}_0 - \mathbf{e})$ 1 MUL, 2 MADD $\mathbf{M}_I \leftarrow [\mathbf{n}_0\ \mathbf{n}_1\ \mathbf{n}_2]^T \cdot [\mathbf{d}_x\, \mathbf{d}_y\, \mathbf{d}_0]$ → 9 MUL 18 MADD
Setup Output matrix: $\mathbf{M}_I$ inverse depth: $\mathbf{w}$ 41 MADD, 25 MUL, 3 SUB, 4 RCP = 72 total 1 DP3, 9 DP4, 10 MADD, 5 MOV, 9 MUL, 4 RCP = 39 total	**Setup Output** matrix: $\mathbf{M}_I$ inverse depth: $\mathbf{w}$ 47 MADD, 19 MUL, 6 ADD, 1 RCP = 73 total 1 DP3, 9 DP4, 7 MADD, 3 MOV, 6 MUL, 1 RCP = 27 total	**Setup Output** matrix: M_I total volume: V 29 MADD, 19 MUL, 9 ADD = 57 total 10 DP3, 5 MADD, 1 MOV, 3 MUL, 3 ADD = 22 total
Per-Pixel Operations $\mathbf{p} \leftarrow (x,y,1)^T$ $\mathbf{E} \leftarrow \mathbf{M}_I \cdot \mathbf{p}$ 6 MADD **if** $\mathbf{E}.x < 0 \wedge \mathbf{E}.y < 0 \wedge \mathbf{E}.z < 0$ **then** $W \leftarrow (\mathbf{w} \cdot \mathbf{E})^{-1}$ 2 MADD, 1 MUL, 1 RCP $u \leftarrow W * \mathbf{w}.x * \mathbf{E}.x$ 2 MUL $v \leftarrow W * \mathbf{w}.y * \mathbf{E}.y$ 2 MUL **return** $(u,v,W)^T$ **end if**	**Per-Pixel Operations** $\mathbf{p} \leftarrow (x,y,1)^T$ $\mathbf{E} \leftarrow \mathbf{M}_I^T \cdot \mathbf{p}$ 6 MADD **if** $\mathbf{E}.x < 0 \wedge \mathbf{E}.y < 0 \wedge \mathbf{E}.z < 0$ **then** $W \leftarrow (\mathbf{w} \cdot \mathbf{p})^{-1}$ 1 RCP, 2 MADD $u \leftarrow W * \mathbf{E}.x$ 1 MUL $v \leftarrow W * \mathbf{E}.y$ 1 MUL **return** $(u,v,W)^T$ **end if**	**Per-Pixel Operations** $\mathbf{p} \leftarrow (x,y,1)^T$ $\mathbf{E} \leftarrow M_I^T \cdot \mathbf{p}$ 6 MADD **if** $\mathbf{E}.x < 0 \wedge \mathbf{E}.y < 0 \wedge \mathbf{E}.z < 0$ **then** $W \leftarrow \mathbf{E}.x + \mathbf{E}.y + \mathbf{E}.z$ 2 ADD **return** $\frac{1}{W}(\mathbf{E}.x, \mathbf{E}.y, V)^T$ 3 MUL, 1 RCP **end if**
8 MADD, 2 MUL, 1 RCP = 14 total 4 CMP, 4 DP3, 1 MOV, 2 MUL, 1 RCP = 12 total	10 MADD, 3 MUL, 1 RCP = 14 total 4 CMP, 4 DP3, 2 MOV, 1 MUL, 1 RCP = 12 total	6 MADD, 2 ADD, 3 MUL, 1 RCP = 12 total 4 CMP, 4 DP3, 2 MOV, 1 MUL, 1 RCP = 12 total

Fig. 10: Costs of 3D rasterization and (homogeneous) 2D rasterization, including perspective correct interpolation of camera space depth and barycentric coordinates for interpolation: additions (ADD), multiplications (MUL) and reciprocals (RCP). Multiply-add (MADD) 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 corresponding implementation is compiled using Direct3D's HLSL compiler for the vertex/pixel shader 3.0 profiles.

Thus we also report the number of instruction slots when compiling the code using a 4-wide SIMD architecture using the vertex/pixel shader 3.0 profiles of Direct3D.

With regard to future programmable graphics hardware, we ran a "close to the metal" comparison of the basic 2D and 3D rasterization algorithms. To measure the pure rasterization performance we implemented both algorithms as Direct3D shaders running on a GPU and emulate the software-rasterization of triangles by invoking the computation through rendering a full-screen quad. In both cases a vertex shader carries out the setup (we omitted clipping for 2DR, as most hardware rasterizers do not need to clip often) 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.

We measured performance on an ATI HD5870 rendering at 1920×1200 resolution. For 262144 triangles, 3D rasterization runs at the same speed as homogeneous 2DR with 465 frames per second (fps); both slightly outperform 2DR with 464 fps. The same trend is visible with significantly less primitives: for 144 triangles, the difference of homogeneous 2DR (2558 fps) and 3DR (2559) is negligible, while 2DR is roughly 2% slower with 2503 fps. Recall that 2DR normally requires an additional clipping step that we omitted.

Discussion As shown by Woop [41], Olano's homogeneous 2D rasterization can be further optimized, ultimately resembling 3D raterization. Then per-pixel operations of Woop's variant and our 3DR are identical and vertex setup only differs in how perspective and viewing transformation is applied. While homogeneous 2DR uses matrix transformations pre-hand, 3DR folds projection and viewing directly into vertex setup.

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, somewhat similar to homogeneous 2D rasterization, which is faster than traditional 2D rasterization, 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. Clearly, the next step would be to implement 3D rasterization using CUDA within the software-based rasterization pipeline of Laine and Karras [19]; depending on the scene, 3DR could then benefit from the avoidance of vertex transformation and possibly existing 3D acceleration structures. However, 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 simulation of global effects requiring secondary rays that are intricate to handle with rasterization, and expensive to compute in ray tracing.

Acknowledgements The Temple of Kalabsha model (Fig. 1) is courtesy of the University of Bristol.

References

[1] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics*, pages 145–149, 2009.

[2] J. Amanatides and K. Choi. Ray tracing triangular meshes. In *Western Computer Graphics Symposium*, pages 43–52, 1997.

[3] A. Appel. Some Techniques for Shading Machine Renderings of Solids. In *AFIPS '68 (Spring): Proceedings of the spring joint computer conference*, pages 37–45, 1968.

[4] C. Benthin and I. Wald. Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing. In *Proceedings of High Performance Graphics*, pages 135–144, 2009.

[5] K. Fatahalian, E. Luong, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan. Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur. In *Proceedings of High Performance Graphics*, pages 59–68, 2009.

[6] K. Garanzha, J. Pantaleoni, and D. McAllister. Simpler and Faster HLBVH with Work Queues. In *Proceedings of High Performance Graphics*, pages 59–64, 2011.

[7] J.-D. Gascuel, N. Holzschuch, G. Fournier, and B. Péroche. Fast Non-Linear Projections using Graphics Hardware. In *Proceedings of Symposium on Interactive 3D Graphics and Games*, 2008.

[8] I. Georgiev and P. Slusallek. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *Proceedings of Symposium on Interactive Ray Tracing*, pages 115–122, 2008.

[9] N. Greene. Hierarchical Polygon Tiling with Coverage Masks. In *SIGGRAPH '96*, pages 65–74, 1996.

[10] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *SIGGRAPH '93*, pages 231–238, 1993.

[11] P. Hanrahan. Ray-Triangle and Ray-Quadrilateral Intersection in Homogeneous Coordinates. Technical Note, `http://graphics.stanford.edu/courses/cs348b-04/rayhomo.pdf`, 1989.

[12] P. S. Heckbert. Fundamentals of Texture Mapping and Image Warping. Technical report, Berkeley, CA, USA, 1989.

[13] W. Heidrich and H.-P. Seidel. View-Independent Environment Maps. In *Proceedings of Workshop on Graphics Hardware*, 1998.

[14] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D Tree GPU Raytracing. In *Proceedings of Symposium on Interactive 3D Graphics and Games*, pages 167–174, 2007.

[15] W. Hunt and W. R. Mark. Ray-Specialized Acceleration Structures for Ray Tracing. In *Proceedings of Symposium on Interactive Ray Tracing*, pages 3–10, Aug 2008.

[16] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark. The Irregular Z-Buffer: Hardware Acceleration for Irregular Data Structures. *ACM Transactions on Graphics*, 24(4):1462–1482, 2005.

[17] J. Kalojanov and P. Slusallek. A Parallel Algorithm for Construction of Uniform Grids. In *Proceedings of High Performance Graphics*, pages 23–28, 2009.

[18] A. Kensler and P. Shirley. Optimizing Ray-Triangle Intersection via Automated Search. In *Proceedings of the Symposium on Interactive Ray Tracing*, pages 33–38, 2006.

[19] S. Laine and T. Karras. High-performance software rasterization on GPUs. In *Proceedings of High Performance Graphics*, 2011.

[20] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[21] C. Loop and C. Eisenacher. Real-time patch-based sort-middle rendering on massively parallel hardware. Technical Report MSR-TR-2009-83, Microsoft Research, May 2009.

[22] W. Mark. Future Graphics Architectures. *Queue*, 6(2), 2008.

[23] M. D. McCool, C. Wales, and K. Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterizatione. In *Proceedings of Workshop on Graphics Hardware*, pages 65–72, 2001.

[24] Microsoft. Direct3D 10 Reference. Direct3D 10 graphics, http://msdn.microsoft.com/directx, 2006.

[25] NVIDIA. NVIDIA GPU Programming Guide. http://developer.nvidia.com, December 2008.

[26] M. Olano and T. Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Proceedings of Workshop on Graphics Hardware*, pages 89–95, 1997.

[27] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95, 2010.

[28] S. G. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive Ray Tracing. In *Proceedings of Symposium on Interactive 3D Graphics and Games*, pages 119–126, 1999.

[29] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 2004.

[30] J. Pineda. A Parallel Algorithm for Polygon Rasterization. *Computer Graphics (Proceedings of SIGGRAPH '88)*, 22(4):17–20, 1988.

[31] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum (Proc. of Eurographics)*, 26(3):415–424, Sept. 2007.

[32] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *ACM Transactions on Graphics (Proc. of SIGGRAPH'02)*, pages 703–712, 2002.

[33] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2005)*, 24(3):1176–1185, 2005.

[34] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transaction on Graphics (Proc. of SIGGRAPH 2008)*, 27(3):1–15, 2008.

[35] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Proceedings of Eurographics Symposium on Rendering*, pages 139–149, 2006.

[36] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transaction on Graphics (Proc. of SIGGRAPH 2007)*, 26(1), 2007.

[37] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transaction on Graphics (Proc. of SIGGRAPH 2006)*, 25(3):485–493, 2006.

[38] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *STAR Proceedings of Eurographics 2007*, pages 89–116, 2007.

[39] I. Wald, P. Slusallek, and C. Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001 (Proceedings of the 12th EUROGRAPHICS Workshop on Rendering)*, pages 277–288, 2001.

[40] T. Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6), 1980.

[41] S. Woop. A Fast Scanline Micro-Rasterization Kernel and its Application to Soft Shadows. Technical report, Intel Corporation, 2010.

[42] S. Woop, J. Schmittler, and P. Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *ACM Transactions on Graphics (Proc. SIGGRAPH 2005)*, volume 24, pages 434–444, 2005.

[43] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-Time KD-Tree Construction on Graphics Hardware. *ACM Transactions on Graphics*, 27(5):1–11, 2008.