Efficient GPU Screen-Space Ray Tracing

Morgan McGuire Michael Mara
Williams College

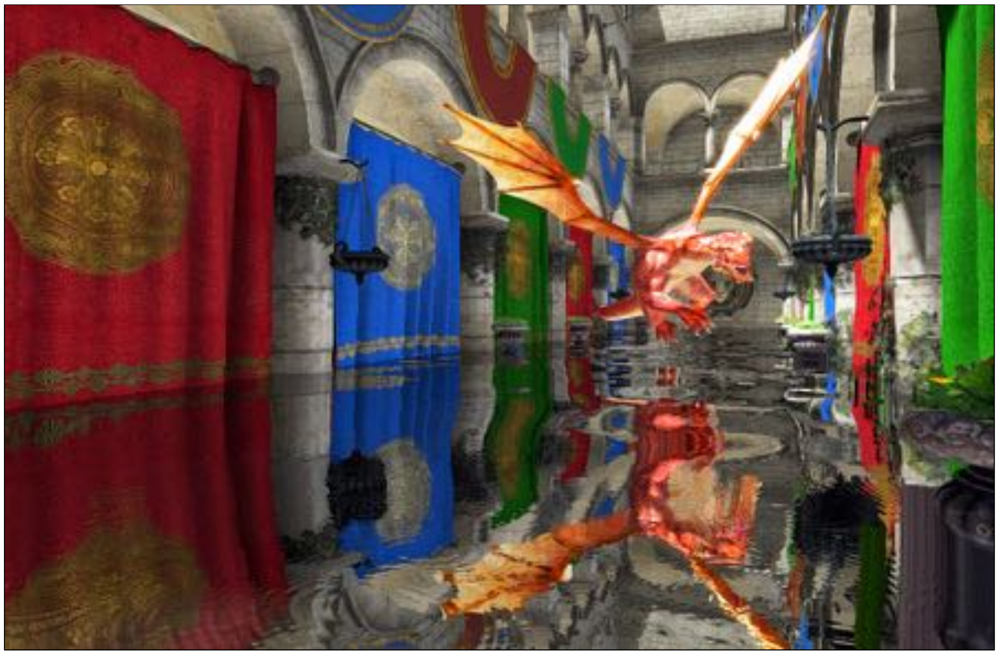


Figure 1. Reflections in ripply water approximated by a screen space ray tracer in 2 ms at 1080p on GeForce 650M.

Abstract

We present an efficient GPU solution for screen-space 3D ray tracing against a depth buffer by adapting the perspective-correct DDA line rasterization algorithm. Compared to linear ray marching, this ensures sampling at a contiguous set of pixels and no oversampling. This paper provides for the first time full implementation details of a method that has been proven in production of recent major game titles. After explaining the optimizations, we then extend the method to support multiple depth layers for robustness. We include GLSL code and examples of pixel-shader ray tracing for several applications.

1. Introduction

Ray *casting* is a fundamental computer-graphics primitive for determining visibility between two points and for discovering the next surface intersection on a path that transports light through a scene. (Ray *tracing* is technically the process of recursively casting rays based on discovered hit points initially described by Turner Whitted. However, so many ray-based algorithms have since been developed that "casting" and "tracing" are now often used interchangeably, with the latter perhaps more popular.)

The inputs to a ray-tracing algorithm are a ray and a scene. Different tracing algorithms are preferred for different scene data structures. Important structures include dynamic triangles in a bounding-volume hierarchy [Wald et al. 2007], implicit surfaces [Blinn 1982; Hart 1994], dense voxels [Amanatides and Woo 1987], sparse voxels [Laine and Karras 2011], and regular heightfields [Musgrave 1988; Henning and Stephenson 2004]. Different algorithms are also preferred for different architectures. Today all high-performance processors are concurrent vector processors, which primarily differ only in vector width, thread count, and ALU/bandwidth ratio.

Ray tracing against the scene data structure of a depth buffer, perhaps with multiple layers, is at the core of many new research and already-deployed industry algorithms. The depth buffer naturally gives a strong bound on scene complexity, allows fully-dynamic scenes, is efficiently built as a natural side-effect of rasterization, and captures all visible surfaces. Ambient occlusion, reflection, refraction, and even full screen-space path tracing can be implemented using only a depth buffer, with varying quality of approximation versus a geometric scene representation. Of course, the depth buffer fails to represent areas of high depth complexity and those far outside the viewport. However, a guard band around the viewport and fallback outside the view frustum to some other data structure, such as a voxel grid, improve trace quality.

1.1. Linear 3D Iteration

The current state of the art for screen-space ray tracing on a GPU was presented by Sousa et al. [2011] of Crytek, for the application of mirror reflection tracing in games. This widely-deployed method linearly ray marches a sample point along a 3D ray for a bounded distance. It projects each 3D point into screen space and classifies it as a ray hit if the point is behind the depth buffer at that pixel. Some extensions are binary search to refine the final hit point and heuristics to detect depth discontinuities and hits against "backfaces." Sousa et al. described the properties of this method but not specifics, which are given in the supplement of Mara et al.'s paper [2014]. Crytek's *Crysis 3* and the game *Just Cause 2* by Avalanche Studios, published by Eidos Interactive in 2010, are major commercial games that appear to use this technique, based on artifacts visible in reflections. Ganestam and Doggett [2014] extend this method for refraction as well as reflection. Their method uses a combination of screen-space data and a geometric bounding-volume hierarchy. Their observations on representing finite surface thickness are particularly relevant for robustness, and we incorporate those ideas later in this paper.

Linearly marching in 3D is reasonable for low-quality reflections on legacy game consoles. Those machines are limited by memory bandwidth to only about five sam-

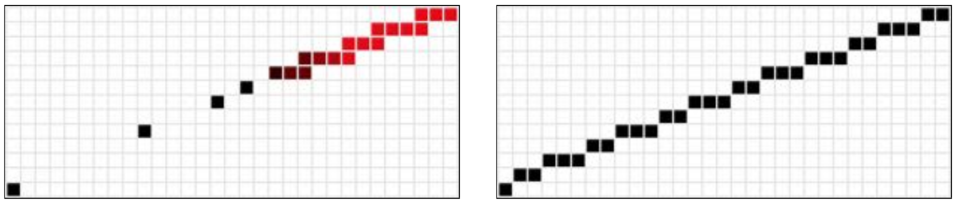


Figure 2. Sampled pixels along a ray, where more red is more oversampled. Linear 3D ray marching (left) skips some pixels and oversamples others. The linear 2D iteration (right) that we describe addresses both drawbacks.

ples per pixel at 30 fps. Yet, taking so few samples misses many intersections, due both to large steps across the scene and termination before finding any intersection. Higher quality through more samples is desirable on faster GPUs with more bandwidth. Sousa et al.'s method extends to this case but becomes inefficient because linear 3D marching is not equivalent to linearly marching in 2D for most rays (figure 2). So, even with a high number of steps, it may still miss many depth-buffer pixels (which is a source of error) and will sample the same pixels multiple times (which is inefficient).

1.2. Linear 2D Iteration

Recent advances for tracing a ray linearly in 2D across a screen-space depth buffer address the limitations of the linear 3D approach and provide the following properties:

1. Places each sampled pixel adjacent (in x, y, or diagonally) to the previous one.
2. Samples each pixel at most once.
3. Clips the ray to the view frustum.
4. Efficiently uses GPU resources by minimizing register consumption, divergent execution, and expensive operations.

The first property addresses quality. The last three address efficiency. Algorithms with these properties have been proved in practice through games available in the last two years. The underlying method of 2D iteration is well known and about 20 years old; the evolution of 2D line rasterization from Bresenham's algorithm through DDA and Musgrave's heightfield applications are covered in standard textbooks today. What the recent games brought was the details needed for reflection tracing and optimizations for modern GPU (or equivalently, CPU SIMD) architectures. Wronski [2014a, 80][2014b] and Valient [2014b, 90][2014a] sketched the algorithms in recent presentations. In this paper, we provide practical details needed to implement such algorithms. We developed these in our own production use and through discussions with those authors. We then explain why the implementation that we provide is efficient in the context of a GPU architecture.

We also offer some extensions. These include relaxing the adjacency constraint (to support controlled spacing between samples for efficiency), jittering (to compensate for the banding artifacts that result from such spaced samples), and application to multiple depth layers (for robustness). These depth layers are those that are produced by depth peeling. However the depth peeling algorithm is slow, so on modern GPUs we prefer faster single-pass methods (e.g., [Mara et al. 2014]) based around reverse reprojection, multiple viewport, and multiple rasterization.

We observe that marching along a 3D ray with samples placed at unique and adjacent 2D pixels is equivalent to perspective-correct rasterization of the portion of the ray that lies within the view volume. So, we adapt a thin-line rasterization algorithm to the application of ray casting. This can step diagonally between pixels. If that is undesirable, then we suggest extending it to conservative line rasterization but have not done so ourselves.

2. Challenges of 2D Iteration

Current GPUs support both integer and floating-point operations, but only floating-point is at full speed. The preferred floating-point line-rasterization algorithm is a Digital Differential Analyzer (DDA). Listing 1 gives pseudocode for a line-segment DDA *for the first (+x, +y) quadrant of the plane*.

```
def drawLine($x_0$, $y_0$, $x_1$, $y_1$):
    if $x_1 - x_0 > y_1 - y_0$:
        slope = $\frac{y_1 - y_0}{x_1 - x_0}$
        for $t = 0$ to $x_1 - x_0$: setPixel($x_0 + t$, $y_0 + t \cdot \text{slope}$)
    else:
        slope = $\frac{x_1 - x_0}{y_1 - y_0}$
        for $t = 0$ to $y_1 - y_0$: setPixel($x_0 + t \cdot \text{slope}$, $y_0 + t$)
```

Listing 1. Pseudocode for a 2D thin-line DDA rasterizer.

The arithmetic operations in the DDA are well-suited to the GPU because the inner loop contains simple fused multiply-add (FMUL) instructions. The major performance bottleneck is the branch. A typical application of screen-space ray-tracing has each pixel trace its own ray, e.g., for a reflection. GPUs group pixels into "warps" or "wavefronts" (e.g., of $8 \times 4 = 32$ pixels on today's architectures) of adjacent pixels, each of which runs on a single core of a superscalar processor. At a statement where pixels within a group branch in different directions, the GPU must compute both sides of the branch, issuing no operation to a portion of the vector lanes for each branch. So, while branch instructions themselves are often relatively expensive compared to arithmetic instructions, their most important cost is that **divergent execution will reduce throughput by one half** for each nested branch.

We gave pseudocode for the first quadrant. To build a practical DDA implementation, one must flip iteration directions based on the quadrant, clip to the viewport, and handle single-pixel lines. Those transformations are tedious but straightforward to implement. Unfortunately, they also mean that **a practical DDA requires a significant number of branches in the classic form**, making it inefficient on a GPU[1].

Rasterizing a 3D line segment is very similar to rasterizing a 2D one. Rasterization can trivially interpolate any property linearly along the iteration direction. If the property is in homogeneous space, then the result will be linear interpolation of the corresponding value in 3D. Specifically, let point Q be a 3D point on the ray and $H = \mathrm{M} \cdot (Q, 1)$ be its homogeneous perspective projection by matrix M. Properties $k = 1/(H \cdot (0,0,0,1))$ and $Q \cdot k$ interpolate linearly in 2D. So, treating the point and reciprocal of the homogeneous w as functions along the 2D line, $\frac{\partial(Q \cdot k)}{\partial x}$, $\frac{\partial k}{\partial x}$, $\frac{\partial(Q \cdot k)}{\partial y}$, and $\frac{\partial k}{\partial y}$ are constant in screen space. At any 2D point (x, y), the corresponding 3D point is

$$Q'(x,y) = \frac{(Q \cdot k)(x,y)}{k(x,y)}. \tag{1}$$

3. An Efficient GPU DDA Solution

Listings 2 and 3 are our GPU-optimized ray tracer based on a DDA, with support for between one and four depth layers. We break it into two listings and format it specially for readability in this paper. The full code as a standalone file with additional comments is available on the *JCGT* website, including a separate version with specific optimizations for the common single depth-layer case.

The classic DDA suffered divergence from eight cases (4 quadrants $\times$ 2 iteration directions). We unified these (line 48) by permuting axes in lines 30-34 and taking derivatives relative to `sign(delta.x)` so iteration is always on the x-axis. Matrix `permute` compensates for the permutation in the vertical case, so all code after line 34 can assume iteration in x.

The practical DDA also required expensive frustum clipping. We observe that for screen-space ray tracing, the caller can trivially ensure the ray origin lies within the view frustum. So, we clip only the far end of the ray against the near plane in line 12. We implicitly clip to the viewport by structuring our test so that the zero value returned by `texelFetch` for an out-of-bounds depth-buffer sample will terminate iteration (we also provide efficient viewport clipping code in listing 4 for reference.)

From this optimized structure, we then applied GPU programming best practices: minimize peak register count, use conditional assignment instead of branches, multiply by precomputed inverses instead of dividing, use floating point instead of integer, minimize instructions within the inner loop (for the instruction cache), and favor

[1] A GPU already has a hardware implementation of parallel rasterization, although it is not accessible to software for 2D ray-tracing. Ignoring access, one might ask why *that* implementation is efficient on a GPU if a classic DDA is not? Part of the answer is that hardware rasterization is fixed point and may not use DDA. But more importantly, it rasterizes a single primitive across multiple processing units, whereas for pixel-shader ray tracing, we need to process multiple primitives (the rays for all pixels in the group) on the single GPU core handling that group. That is, the difference is deep vs. wide parallelism.

FMUL over other operations.

Preprocessor loops (e.g., line 4) are a feature of the G3D Innovation Engine (`http://g3d.sf.net`) and not standard GLSL. Many frameworks support these; for others, the loops may be expanded manually. We use them to generalize the ray trace to multiple depth layers. They are unnecessary for single depth-buffer tracing.

The function input arguments are: camera-space ray origin `csOrig` and direction `csDir`; the matrix `proj` that projects from camera-space position to pixel screen position, along with explicit (negative) `nearPlaneZ`; the depth buffer `csZBuffer`, its dimensions `csZBufferSize`, and encoding (`csZBufferIsHyperbolic`); the camera-space `zThickness` to ascribe to each point in a depth buffer; the `maxDistance` in camera-space and `maxSteps` to trace, and stride information. Setting `stride = 1.0` produces thin-line rasterization. Larger values space samples for efficiency, at a loss of quality. That also introduces banding, which can be concealed by setting the `jitter` $\in [0,1]$ amount to bump the ray in pixels. We use right-handed camera space where $+z$ is out of the screen.

The function returns `true` if the ray hits a surface, with output parameters giving the `hitPixel` coordinates, the `hitLayer` in the depth buffer (useful when `numLayers` > 1), and the camera-space `csHitPoint`. To reduce the loop body and increase instruction caching, we break out of the loop on a hit but don't check if it was a legal screen coordinate until line 84.

Within the function, `P0` and `P1` are the screen-space pixel endpoints of the line segment. None of the divisions will encounter a zero denominator: by line 19, the ray was already clipped to the near plane so there is no $w = 0$ singularity; and line 27 guarantees that `delta.x` later will be nonzero even for the case of a single-pixel line. The optional viewport clipping code also avoids division by zero because the extent is never zero when clipping is required.

It is likely that the application will trace many rays that touch the same pixels. Converting the hyperbolically encoded values in a standard depth buffer to camera-space z values for each iteration at each pixel will then be expensive. For convenience, if `csZBufferIsHyperbolic = true`, then our implementation *will* accept a standard depth buffer and use lines 66-70 to convert each value to camera-space. However, it prefers a floating-point buffer of precomputed (negative) camera-space values.

The code tracks up to four depth-buffer layers in `sceneZMax` at each pixel. It assumes that each represents the front of a frustum-shaped voxel that extends away `zThickness` in z. Within a pixel, the ray itself covers the z interval [`rayZMin`, `rayZMax`]. The ray hits the depth buffer at a pixel if this overlaps a voxel in any layer.

Our implementation assumes that pixel (0, 0) is the upper-left corner of the screen. To switch to the lower-left corner convention, modify the final hit point by `hitPoint.y = csZBufferSize.y - hitPoint.y`.

```
#define point2 vec2
#define point3 vec3

#for (int numLayers = 1; numLayers < 5; ++numLayers)
bool traceScreenSpaceRay##numLayers(point3 csOrig, vec3 csDir, mat4x4 proj,
    sampler2D csZBuffer, vec2 csZBufferSize, float zThickness,
    const bool csZBufferIsHyperbolic, vec3 clipInfo, float nearPlaneZ,
    float stride, float jitter, const float maxSteps, float maxDistance,
    out point2 hitPixel, out int hitLayer, out point3 csHitPoint) {

    // Clip to the near plane
    float rayLength = ((csOrig.z + csDir.z * maxDistance) > nearPlaneZ) ?
        (nearPlaneZ - csOrig.z) / csDir.z : maxDistance;
    point3 csEndPoint = csOrig + csDir * rayLength;
    hitPixel = point2(-1, -1);

    // Project into screen space
    vec4 H0 = proj * vec4(csOrig, 1.0), H1 = proj * vec4(csEndPoint, 1.0);
    float k0 = 1.0 / H0.w, k1 = 1.0 / H1.w;
    point3 Q0 = csOrig * k0, Q1 = csEndPoint * k1;

    // Screen-space endpoints
    point2 P0 = H0.xy * k0, P1 = H1.xy * k1;

    // [ Optionally clip here using listing 4 ]

    P1 += vec2((distanceSquared(P0, P1) < 0.0001) ? 0.01 : 0.0);
    vec2 delta = P1 - P0;

    bool permute = false;
    if (abs(delta.x) < abs(delta.y)) {
        permute = true;
        delta = delta.yx; P0 = P0.yx; P1 = P1.yx;
    }

    float stepDir = sign(delta.x), invdx = stepDir / delta.x;

    // Track the derivatives of Q and k.
    vec3  dQ = (Q1 - Q0) * invdx;
    float dk = (k1 - k0) * invdx;
    vec2  dP = vec2(stepDir, delta.y * invdx);

    dP *= stride; dQ *= stride; dk *= stride;
    P0 += dP * jitter; Q0 += dQ * jitter; k0 += dk * jitter;
    float prevZMaxEstimate = csOrig.z;
```

Listing 2. Optimized GLSL screen-space ray trace (setup).

```
    // Slide P from P0 to P1, (now-homogeneous) Q from Q0 to Q1, k from k0 to k1
    point3 Q = Q0; float  k = k0, stepCount = 0.0, end = P1.x * stepDir;
    for (point2 P = P0;
        ((P.x * stepDir) <= end) && (stepCount < maxSteps);
        P += dP, Q.z += dQ.z, k += dk, stepCount += 1.0) {

        // Project back from homogeneous to camera space
        hitPixel = permute ? P.yx : P;

        // The depth range that the ray covers within this loop iteration.
        // Assume that the ray is moving in increasing z and swap if backwards.
        float rayZMin = prevZMaxEstimate;
        // Compute the value at 1/2 pixel into the future
        float rayZMax = (dQ.z * 0.5 + Q.z) / (dk * 0.5 + k);
        prevZMaxEstimate = rayZMax;
        if (rayZMin > rayZMax) { swap(rayZMin, rayZMax); }

        // Camera-space z of the background at each layer (there can be up to 4)
        vec4 sceneZMax = texelFetch(csZBuffer, int2(hitPixel), 0);

        if (csZBufferIsHyperbolic) {
#           for (int layer = 0; layer < numLayers; ++layer)
                sceneZMax[layer] = reconstructCSZ(sceneZMax[layer], clipInfo);
#           endfor
        }
        float4 sceneZMin = sceneZMax - zThickness;

#       for (int L = 0; L < numLayers; ++L)
            if (((rayZMax >= sceneZMin[L]) && (rayZMin <= sceneZMax[L])) ||
            (sceneZMax[L] == 0)) {
                hitLayer = layer;
                break; // Breaks out of both loops, since the inner loop is a macro
            }
#       endfor // layer
    } // for each pixel on ray

    // Advance Q based on the number of steps
    Q.xy += dQ.xy * stepCount; hitPoint = Q * (1.0 / k);
    return all(lessThanEqual(abs(hitPixel - (csZBufferSize * 0.5)),
                         csZBufferSize * 0.5));
}
#endfor
```

Listing 3. Optimized GLSL screen-space ray trace (inner loop).

```
float xMax=csZBufferSize.x-0.5, xMin=0.5, yMax=csZBufferSize.y-0.5, yMin=0.5;
float alpha = 0.0;

// Assume P0 is in the viewport (P1 - P0 is never zero when clipping)
if ((P1.y > yMax) || (P1.y < yMin))
    alpha = (P1.y - ((P1.y > yMax) ? yMax : yMin)) / (P1.y - P0.y);

if ((P1.x > xMax) || (P1.x < xMin))
    alpha = max(alpha, (P1.x - ((P1.x > xMax) ? xMax : xMin)) / (P1.x - P0.x));

P1 = lerp(P1, P0, alpha); k1 = lerp(k1, k0, alpha); Q1 = lerp(Q1, Q0, alpha);
```

Listing 4. Optional viewport clipping.

4. Parameters and Image Quality

Adjust the layer thickness to match the expected object thickness to improve image quality. This allows rays to pass behind objects. It is most useful for a multi-layered depth buffer. Figure 3 compares varying thickness in a simple scene with a single buffer, where near-infinite thickness is also a reasonable choice.

Some rays will never hit a surface captured by the depth-buffer because those rays travel farther than `maxDistance` or towards the camera. In this case, one can fall back to tracing a different data structure, such as sparse voxels, or simply fetch from an environment cube map based on the ray direction. Figure 4 (left) shows water reflection and refraction for a box of teapots. The right side highlights pixels at which ray misses occurred.

Several parameters allow the programmer to trade the image quality for performance. Figure 5 shows the impact of adjusting `stride` and `jitter`. We recommend `stride`>1 only for rough surfaces, where details conceal artifacts. Reducing the ray length by camera-space `maxDistance` and screen-space `maxSteps` gives the obvious effects of shortening the ray and bounding worst-case computation time.

Figure 6 shows four applications of screen-space ray tracing. Reflection and refraction each cost one ray cast per pixel and are reasonable for real-time applications. The ambient occlusion and radiosity examples each cast 25 uniformly distributed rays per pixel. We chose 25 rays because that was the highest count possible for real-time performance on our desktop. These results are academically interesting, but since they spend about 30 ms on just ray casts for global illumination, they are not viable techniques in a real-time pipeline containing other graphics tasks. However, for solving the specific problem of real-time global illumination today, we recommend a much faster ($\approx$ 3ms) approximation [2014]. That method is so fast because it casts no rays...it instead samples within a ball and assumes that each point can transport light directly to the center with no further occlusion.

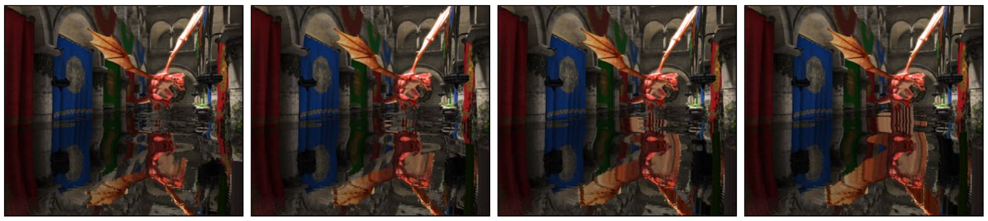


Figure 3. Varying thickness. From left to right, `zThickness` = { 1, 2, 5, 1000} meters.

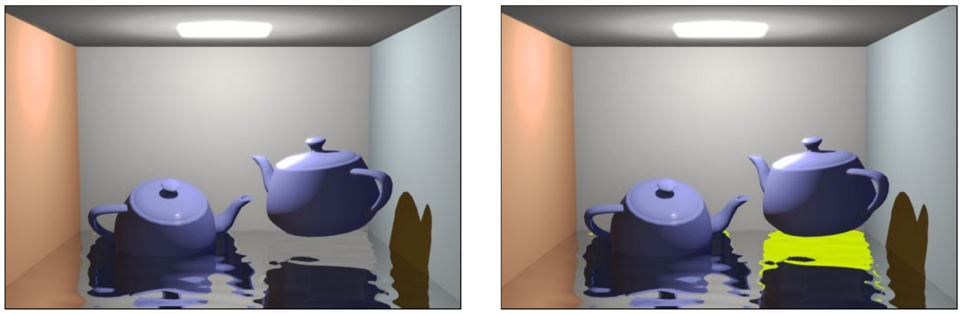


Figure 4. Left) Teapots reflecting in water. Right) The left image with origins of rays that missed the depth buffer and instead sampled the cube map are marked in yellow.

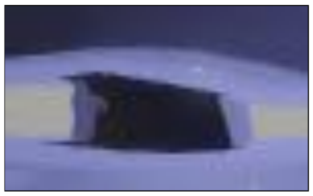
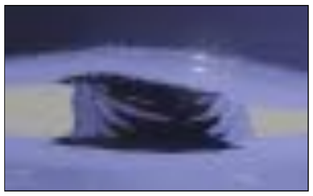
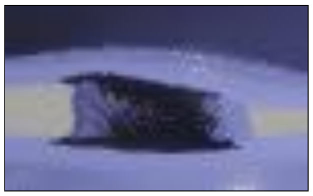
Image			
`stride`	1	4	4
`jitter`	0.0	0.0	`float((c.x+c.y)&1)*0.5`

Figure 5. An enlarged region of the reflection of the left teapot's knob in figure 4. Stride samples trades quality for performance; jittering can help. `c = ivec2(gl_FragCoord.xy)`.

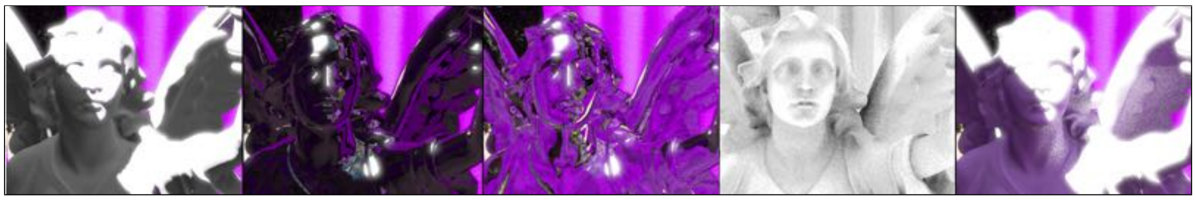


Figure 6. The "Lucy" angel statue rendered with global illumination terms computed by ray tracing in screen space plus a cube map of areas outside the viewport. a) Direct illumination b) reflective chrome c) refractive glass d) ambient occlusion e) radiosity.

Ray tracing performance of course varies with the scene because it is linear in the number of iterations before a hit is encountered or the `maxDist` is reached. At 1920×1080 resolution, casting a diagonal ray for 25 iterations of the inner loop at every pixel costs 1.2 ms on a desktop NVIDIA GeForce Titan and 6.4 ms on a MacBook NVIDIA GeForce 650M. (Here, we chose 25 iterations as the minimum needed for reasonable image quality; it is a coincidence that this is the same constant as the number of rays per pixel in the previous paragraph.)

The method from this paper has been both suitably fast and flexible for our applications. In closing, we note three natural extensions others might wish to explore as future work. First, the trace could test *all* pixels that the ray passes through, instead of allowing diagonal adjacency. This requires adapting a conservative rasterization algorithm [Amanatides and Woo 1987; Wu 1991]. Second, when using `stride > 1`, one might wish to refine the final hit point location by binary search over the final interval. Third, subpixel refinement allows bilinear sampling of the screen color at the hit point, which gives smoother indirect lighting effects.

Acknowledgements

We thank Bart Wronski, Michal Valient, and Michal Drobot for presenting their work publicly and discussing it with us privately, and Ulf Assarsson for the many improvements he contributed to this paper's exposition.

References

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, 3–10. URL: `http://www.cse.yorku.ca/~amana/research/grid.pdf`. 74, 83

BLINN, J. F. 1982. A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (July), 235–256. URL: `http://doi.acm.org/10.1145/357306.357310`. 74

GANESTAM, P., AND DOGGETT, M. 2014. Real-time multiply recursive reflections and refractions using hybrid rendering. *The Visual Computer*, 1–9. doi:10.1007/s00371-014-1021-7. 74

HART, J. C. 1994. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 527–545. URL: `http://graphics.cs.illinois.edu/papers/zeno`. 74

HENNING, C., AND STEPHENSON, P. 2004. Accelerating the ray tracing of height fields. ACM, GRAPHITE '04, 254–258. URL: `http://doi.acm.org/10.1145/988834.988878`. 74

LAINE, S., AND KARRAS, T. 2011. Efficient sparse voxel octrees. *IEEE TVCG 17*, 1048–1059. URL: `http://doi.ieeecomputersociety.org/10.1109/TVCG.2010.240`. 74

MARA, M., MCGUIRE, M., NOWROUZEZAHRAI, D., AND LUEBKE, D. 2014. Fast global illumination approximations on deep G-buffers. Tech. Rep. NVR-2014-001, NVIDIA Corporation, June. URL: http://graphics.cs.williams.edu/papers/DeepGBuffer14. 74, 76, 81

MCGUIRE, M. 2013. *The Graphics Codex*, 2.7 ed. URL: http://graphicscodex.com. 84

MUSGRAVE, K. F. 1988. Grid tracing: Fast ray tracing for height fields. Tech. Rep. YALEU/DCS/RR-639, Yale University. 74

SOUSA, T., KASYAN, N., AND SCHULZ, N. 2011. Secrets of CryENGINE 3 graphics technology. In *SIGGRAPH Courses*, ACM, New York, NY, USA. URL: http://www.crytek.com/cryengine/presentations/secrets-of-cryengine-3-graphics-technology. 74

VALIENT, M., 2014. Reflections and volumetrics of Killzone Shadow Fall. Presentation at SIGGRAPH Advances in Real-Time Rendering in Games course. URL: http://advances.realtimerendering.com/s2014/valient/Valient_Siggraph14_Killzone.pptx. 75

VALIENT, M., 2014. Taking Killzone Shadow Fall image quality into the next generation. Presentation at GDC'14. URL: http://www.guerrilla-games.com/presentations/GDC2014_Valient_Killzone_Graphics.pdf. 75

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (Jan.). URL: http://doi.acm.org/10.1145/1189762.1206075. 74

WRONSKI, B., 2014. Assassin's Creed 4: Black Flag, road to next-gen graphics. Presentation at GDC'14. URL: http://bartwronski.files.wordpress.com/2014/03/ac4_gdc.pdf. 75

WRONSKI, B., 2014. The future of screenspace reflections, January. Blog post. URL: http://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/. 75

WU, X. 1991. An efficient antialiasing technique. *SIGGRAPH 25*, 4 (July), 143–152. URL: http://doi.acm.org/10.1145/127719.122734. 83

Index of Supplemental Materials

Our supplemental materials are a fully-commented GLSL implementation of the ray tracing algorithm (`raytrace.glsl`), and C++ and GLSL code (from the *Graphics Codex* [McGuire 2013]) for recovering z from a depth buffer value and other utilities (`util.*`).

Author Contact Information

Morgan McGuire and Michael Mara
Williams College
47 Lab Campus Drive
Williamstown, MA 01267
morgan@cs.williams.edu
http://graphics.cs.williams.edu

McGuire and Mara, Efficient GPU Screen-Space Ray Tracing, *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 4, 73–85, 2014
http://jcgt.org/published/0003/04/04/

Received:	2014-08-05		
Recommended:	2014-08-10	Corresponding Editor:	Ulf Assarsson
Published:	2014-12-09	Acting Editor-in-Chief:	Ulf Assarsson