A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering

Alexander Majercik[1,2] Cyril Crassin[1] Peter Shirley[1] Morgan McGuire[1,2]
[1]NVIDIA [2]Williams College

Figure 1. A massive Minecraft world rendering at 3.2 Mvoxels/ms (53 Mvoxels/frame at 60 fps) on GeForce 1080 via our fast ray-box intersection. This *fully dynamic* example is 7× faster than rasterization of the equivalent *static* mesh with precomputed visibility. The full model can be downloaded from https://www.planetminecraft.com/project/apterra---huge-custom-survival-map/.

Abstract

We introduce a novel and efficient method for rendering large models composed of individually-oriented voxels. The core of this method is a new algorithm for computing the intersection point and normal of a 3D ray with an arbitrarily-oriented 3D box, which also has non-rendering applications in GPU physics, such as ray casting and particle collision detection. We measured throughput improvements of 2× to 10× for the intersection operation versus previous ray-box intersection algorithms on GPUs. Applying this to primary rays increases throughput 20× for direct voxel ray tracing with our method versus rasterization of optimal meshes computed from voxels, due to the combined reduction in both computation and bandwidth. Because this method uses no precomputation or spatial data structure, it is suitable for fully dynamic scenes in which every voxel potentially changes every frame. These improvements can enable a dramatic increase in dynamism, view distance, and scene density for visualization applications and voxel games such as *LEGO*® *Worlds* and *Minecraft*. We provide GLSL code for both our algorithm and previous alternative optimized ray-box algorithms, and an Unreal Engine 4 modification for the entire rendering method.

 ISSN 2331-7418

1. Introduction

This paper deals with rendering large models composed of voxels, while contributing to related tasks. The core of our method is a new GPU algorithm for efficiently finding the intersection point and the normal at that point where a ray hits a box.

In this context, we use "box" as conventional shorthand for the surface of a hollow rectangular parallelepiped. Boxes may be oriented (OBox) or axis-aligned (AABox) with the Cartesian axes. They can be outright modeling primitives as well as bounding-box (AABB/OBB) volumes for more detailed geometry.

While much of the literature on ray-box intersection is concerned with bounding volume hierarchies for visible surface ray tracing, in practice the applications of ray-box intersection are more diverse:

- AABB for bounding volume hierarchy (BVH) traversal to support ray-triangle intersection for many rendering, physics, and AI applications;
- OBox for intersections between boxes and thin objects such as wires, where the ray models the wire;
- OBox for continuous collision detection on particles (e.g., bullet or rain droplet in a game, molecule in a chemical simulation);
- AABox for physics forces on proxy objects, such as downward exploration rays on a car or for planting character feet;
- OBox for AI line of sight and pathfinding;
- OBox for conservative precomputed visibility via "stabbing";
- AABox for sparse voxel indirect light and ambient occlusion;
- AABox and OBox for direct ray tracing of massive *dynamic* voxel models (e.g., brick/voxel-video games, microchip visualization, map rendering of buildings and roads, Navisworks civil engineering models of pipes and wires); we demonstrate this application explicitly.

The first several of the above applications arise in most interactive 3D applications. The last one, which we emphasize in this paper, is specialized to rendering tasks for scientific/medical/engineering visualization (see, for example, Figure 3) and a specific niche of brick/voxel video games–albeit a niche which includes the best-selling game of all time, *Minecraft*, the massive brand *LEGO*®, many smaller titles inspired by these, and the indie art style of small voxel models popularized in part by *MagicaVoxel* and *Voxatron*.

We use the voxel-rendering application as motivation for a voxel ray tracer that exceeds the performance of GPU rasterization, but pose the core of our method as a

Figure 2. Millions of independently-oriented dynamic voxels in our real-time ray-traced explosion of the science-fiction city scene.

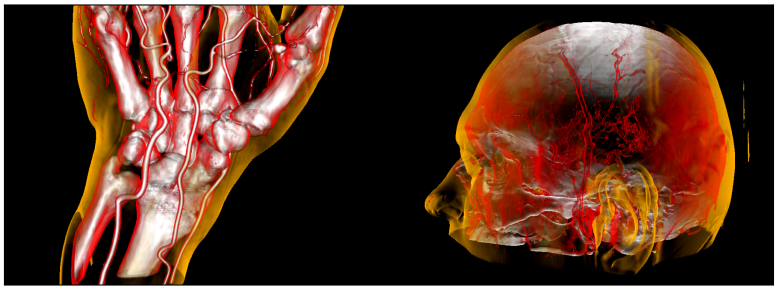

Figure 3. Voxel computational tomography data rendered by Csébfalvi et al. [2012].

new algorithm that significantly outperforms the previous state of the art for many of the above cases.

Specifically with regard to rendering voxels, there are many optimizations (e.g., [Crassin et al. 2009; Lysenko 2012; Barrett 2015]) that can be made for important but restricted cases, such as chunking static geometry at low precision, compressing axis-aligned surfaces, and computing implicit texture coordinates. Our method is compatible with some of those optimizations as dynamism becomes restricted, but we do not pursue that in this paper. We specifically evaluate the unrestricted case of arbitrary, dynamic voxels that have the potential to change independently without a complex data structure (as shown in Figure 2 and our video). For cases of rasterizing restricted, mostly static meshes, e.g., as done in *Minecraft*, previous methods likely achieve higher performance at the expense of those limitations and algorithmic complexity.

Our ray-box algorithm is also suitable for solving the other problems listed above with large numbers of potentially-oriented boxes on a GPU. Note that most of those are not rendering applications; however they are commonplace in engineering software (e.g., CAD, pipe layout, processor design) and video games (see, for example, Figure 4).

An exception is the ray-AABB intersection specifically for bounding volume hierarchy (BVH) traversal. That special case is better addressed via optimization techniques that exploit axis alignment and the relatively small number of boxes in a BVH. We give code and performance results for a previous industry method that we call

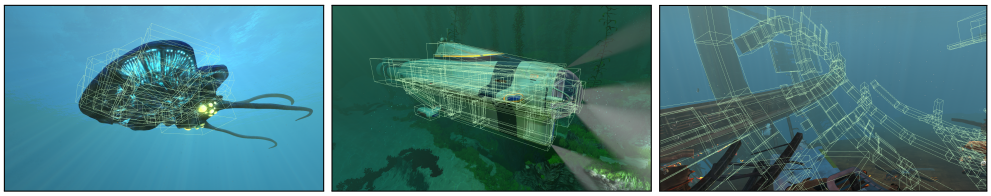

Figure 4. Oriented boxes used for collision detection against particles and projectiles in the 2018 PC video game *Subnautica*. A typical scene might contain millions of instanced oriented boxes for such non-rendering applications. Our ray-box algorithm is suited to this application as well as rendering. Images courtesy of and ©2018 Unknown Worlds.

"efficient slabs" that does this. Our new solution is designed for large numbers of boxes and oriented boxes, and it is generally less efficient in the BVH AABB case.

2. Related Work

Much effort goes into ray-AABox tests for culling. It is a kernel operation for BVH-based ray intersection, on which modern high-performance ray-casting algorithms are built. For over a decade, the Kay and Kajiya [1986] slabs method has been the dominant algorithm, but the details of the code for implementing this algorithm have changed significantly. A variety of other methods have been used on the CPU. Woo [1990] used backface culling to halve the number of tests, and our method extends this concept to GPUs in a way that efficiently computes the surface normal. Eisemann et al. [2007] used a set of three 2D tests to intersect AABBs. Mahovsky and Wyvill [2004] used Plücker coordinates to determine intersections for AABBs. Their approach should also work for arbitrary convex shells defined by eight vertices.

The most popular tests have been variants of the Kay-Kajiya slabs method. Williams et al. [2003] improved both its precision and performance. The graphics website Scratchapixel [2016] (which has intentionally anonymous industry authorship) gives a further-optimized version for the CPU. Most fast SIMD implementations, such as the ones in OptiX and Embree [Laine et al. 2013; Áfra et al. 2016], take advantage of vector min/max intrinsics on both CPUs and GPUs. Listing 1 is a representative implementation. (All code listings in this paper are in the GLSL language with boilerplate removed. See the supplement for complete executable shaders for OpenGL and UE4.)

To determine the normal and hit point, common practice is to adapt these previous methods by tracking *which* face of which slab was hit and then solving for the intersection data. For oriented boxes, one first transforms the ray into box space and then the hit point back into world space.

Games and visualization applications that use explicit voxels convert them to polygon meshes by eliminating all faces between opaque voxels. An entirely en-

```
bool slabs(vec3 p0, vec3 p1, vec3 rayOrigin, vec3 invRaydir) {

   vec3 t0 = (p0 - rayOrigin) * invRaydir;
   vec3 t1 = (p1 - rayOrigin) * invRaydir;
   vec3 tmin = min(t0,t1), tmax = max(t0,t1);

   return max_component(tmin) <= min_component(tmax);
}
```

Listing 1. Efficient slab test for a ray intersecting the AABB with corners `p0` and `p1` representative of BVH traversal in the NVIDIA OptiX and Intel Embree triangle ray tracing APIs.

closed (e.g., underground) voxel generates zero triangles and one exposed on one face (e.g., within a wall) generates two triangles. The worst case is a completely isolated voxel, which produces 12 triangles and requires 24 unique vertices because position, normals, and texture coordinates are indexed in parallel arrays on GPUs.

3. Voxel Rendering Algorithm

We use the rasterizer as a potentially-visible set optimization to iterate only over pixels for which rays might intersect a voxel, and then execute a small ray tracer in the pixel shader. That is, we "splat" billboards that give coarse visibility and compute exact visibility in a pixel shader. This works for any pinhole perspective projection, including eye rays and shadow rays, so we use it for the shadow map rendering pass as well (see Listing 2).

```
Host:
 Optionally frustum cull objects composed of voxels based on their aggregate bounding boxes
 Submit voxels as points in OpenGL and as indexed triangle billboards in DirectX

Vertex shader:
  Read the voxel transformation and material (from an attribute stream or a texture)
  Cull against the near plane conservatively

  If the voxel projected area is subpixel:
     Stochastically cull based on a stable hash of the voxel index [Cook et al.] and the area
     If the voxel passes the cull, enlarge its radius proportionally

  If the voxel projected area is small (few pixels):
     Compute the AABB covering the bounding sphere
  Else:
     Compute the line segments of the edges
     Clip each edge against the near plane
     Compute the AABB of the vertices after clipping

  Move the vertices of the covering quad in homogeneous clip space to the closest vertex,
      maintaining coverage under perspective projection
  If GL: change the point sprite center and size to fit the covering quad
  If DX: move the vertex to one of the four quad corners based on the vertex index

Pixel shader:
 Compute the ray through the current pixel or MSAA sample
 Use our fast ray-box text to find the intersection point
 If no intersection: discard/texkill
 Else: shade the intersection and conservatively adjust the depth-buffer value to match the intersection
```

Listing 2. Voxel pseudocode.

OpenGL is a little different because it supports point sprites, which on NVIDIA hardware have an optimized axis-aligned rectangle rasterization path in the GPU (other GPUs may accelerate this as well). This is not exposed in DirectX, so under that API we explicitly move the vertices to form the billboard in the vertex shader.

Our implementation in Unreal Engine 4 (UE4) is built on the UE4 particle system. Voxels are drawn on screen-aligned quads emitted from a GPU emitter. We extend the GPU particle state by adding a `FVector4 Color` field to the `struct FNewParticle` to be read into a new `ColorTexture` (similar to the `Position Texture` and `Velocity Texture`) when the particles are injected into the scene. To render the particles, we submit one instanced quad. In `ParticleGPUSprite VertexFactory.ush`, the method "FVertexFactoryIntermediates GetVertex FactoryIntermediates(FVertexFactoryInput Input)", returns per-particle state information (position, velocity, etc.) in an `FVertexFactoryIntermediates` struct. A particle index computed from an instance ID is used to read the particle state textures, and this information is propagated to the `FVertexFactoryIntermediates` struct. The vertex positions for each instance are computed from particle state data using our projected voxel bounds. In `BasePassPixelShader.usf`, we compute the world space normal and depth of each pixel using our ray-box intersection algorithm, discarding no-hit pixels. Explicit control over the billboard in DX allows us to generate non-square sprites, which more tightly bounds the projected voxel.

Our implementation allows us to simulate millions of particles in realtime by using the UE4 particle simulation in `ParticleSimulationShader.usf`. However, for static models such simulation is unnecessary, and the algorithm could be optimized by not running the particle simulation at all. Further, as the particles are static, the velocity texture (and other textures that store particle state) may be done away with entirely. We did not implement these optimizations in order to maintain the voxel particles in full generality. We could use the `QUAD_FILLMODE` which is exposed through an NVAPI (`https://docs.nvidia.com/gameworks/content/gameworkslibrary/coresdk/nvapi/group__dx.html`) in DX and on NVIDIA hardware, and which allows rasterizing the AABB of each submitted triangle. This would have allowed saving one vertex load and transform for each quad. We did not use it in the UE4 implementation in order to maintain compatibility with any hardware configuration.

```
// Square area
float stochasticCoverage = pointSize * pointSize;
if ((stochasticCoverage < 0.8) &&
    ((gl_VertexID & 0xffff) > stochasticCoverage * (0xffff / 0.8))) {
    // "Cull" small voxels in a stable, stochastic way by moving past the z = 0 plane.
    // Assumes voxels are in randomized order.
    position = vec4(-1,-1,-1,-1);
}
```

Listing 3. GLSL implementation of the stochastic pruning.

4. Screen-Space Bounds Calculation

An important aspect of our technique is the efficient calculation of a screen-space AABB for each billboard, which tightly fits the perspective-projected oriented 3D box and allows for minimizing of the number of fragments emitted for each of them. To achieve that goal, we approximate the oriented 3D boxes with bounding spheres, and we rely on the elegant and very efficient quadric-based implicit formulation by [Sigg et al. 2006] to compute the screen-space bounding box. The general idea is to express the bounding box as the root of a bilinear form corresponding to the implicit definition of the quadric surface of the bounding sphere projected in screen space. The quadric is defined using homogeneous coordinates, which allows applying arbitrary linear transformation, including perspective projection (see Listing 4).

The AABB of large voxels that are close to the camera, and thus project to many pixels ($> 20 \times 20$ pixels) on the screen, need to be computed more precisely. This is especially important because those voxels are also more likely to be clipped by the frustum's planes, reducing their screen-space footprint even more when tightly bounded.

For those large voxels, we actually clip each edge of the 3D box against five of the frustum planes (excluding the far plane, which is unlikely to be crossed), project those clipped segments, and then greedily compute their screen-space bounds.

```
//Fast Quadric Proj:  "GPU-Based Ray-Casting of Quadratic Surfaces" http://dl.acm.org/citation.cfm?id=2386396
void quadricProj(in vec3 osPosition, in float voxelSize, in mat4 objectToScreenMatrix, in vec2 halfScreenSize,
                                     inout vec4 position, inout float pointSize) {

    const vec4 quadricMat = vec4(1.0, 1.0, 1.0, -1.0);
    float sphereRadius = voxelSize * 1.732051;
    vec4 sphereCenter = vec4(osPosition.xyz, 1.0);
    mat4 modelViewProj = transpose(objectToScreenMatrix);

    mat3x4 matT = mat3x4( mat3(modelViewProj[0].xyz, modelViewProj[1].xyz, modelViewProj[3].xyz) * sphereRadius);
    matT[0].w = dot(sphereCenter, modelViewProj[0]);
    matT[1].w = dot(sphereCenter, modelViewProj[1]);
    matT[2].w = dot(sphereCenter, modelViewProj[3]);

    mat3x4 matD = mat3x4(matT[0] * quadricMat, matT[1] * quadricMat, matT[2] * quadricMat);
    vec4 eqCoefs =
            vec4(dot(matD[0], matT[2]), dot(matD[1], matT[2]), dot(matD[0], matT[0]), dot(matD[1], matT[1]))
            / dot(matD[2], matT[2]);

    vec4 outPosition = vec4(eqCoefs.x, eqCoefs.y, 0.0, 1.0);
    vec2 AABB = sqrt(eqCoefs.xy*eqCoefs.xy - eqCoefs.zw);
    AABB *= halfScreenSize * 2.0f;

    position.xy = outPosition.xy * position.w;
    pointSize = max(AABB.x, AABB.y);
}
```

Listing 4. GLSL implementation of the quadric projection and screen-space bounds calculation.

5. Ray-Box Intersection Algorithm

Our algorithm proceeds as follows (Listing 5):

1. transform ray into box local coordinate system;
2. determine the planes of the three potential front faces;
3. compute a bitmask of whether each ray-plane intersection lies within the box;
4. set the distance and normal from the bitmask;
5. return whether any bit is true.

This has several advantages over the slab method. First, recognizing that the box is convex allows back-face culling as well as recognizing that any hit is the first hit. Second, the bitmask (the `bvec3` in Listing 5) used for intersection computation implicitly contains the normal and ray-hit parameter information. Third, it is branchless by use of conditional moves instructions. Because it is branchless, it can vectorize with perfect occupancy across GPU lanes of scalar ALUs. On vector ALUs, the distance and bitmask computations operate in parallel across all three faces.

```
float max(vec3 v) { return max (max(v.x, v.y), v.z); }

// box.rotation = object-to-world, _invRayDir unused if oriented
bool ourIntersectBox(Box box, Ray ray, out float distance, out vec3 normal,
  const bool canStartInBox, const in bool oriented, in vec3 _invRayDir) {

  ray.origin = ray.origin - box.center;
  if (oriented) { ray.dir *= box.rot; ray.origin *= box.rot; }

  float winding = canStartInBox && (max(abs(ray.origin) * box.invRadius)
         < 1.0) ? -1 : 1;
  vec3 sgn = -sign(ray.dir);
  // Distance to plane
  vec3 d = box.radius * winding * sgn - ray.origin;
  if (oriented) d /= ray.dir; else d *= _invRayDir;

# define TEST(U, VW) (d.U >= 0.0) && \
   all(lessThan(abs(ray.origin.VW + ray.dir.VW * d.U), box.radius.VW))
  bvec3 test = bvec3(TEST(x, yz), TEST(y, zx), TEST(z, xy));
  sgn = test.x ? vec3(sgn.x,0,0) : (test.y ? vec3(0,sgn.y,0) :
        vec3(0,0,test.z ? sgn.z:0));
# undef TEST

  distance = (sgn.x != 0) ? d.x : ((sgn.y != 0) ? d.y : d.z);
  normal = oriented ? (box.rot * sgn) : sgn;

  return (sgn.x != 0) || (sgn.y != 0) || (sgn.z != 0);
}
```

Listing 5. GLSL implementation of our algorithm for the general case. The `const bool` arguments produce compile-time reduction to the exact features required. See supplementary code for hand-optimized special cases.

The code for the ray distance and the normal is shown in Listing 5. This code can handle the axis-aligned or oriented case as indicated by the boolean input argument.

Note that in the oriented case a check for zeros in `ray.dir` is not needed in this algorithm because the conditionals implicitly take care of these cases.

In addition to the algorithmic concepts, our implementation has been carefully optimized using GPU programming best practices and profiling. The specific GLSL implementation given adjusts the order of operations in some non-obvious ways in order to reduce the peak register count. It largely avoids the cost of branches by preferring conditional move operations.

6. Results

6.1. Ray-Box Robustness

As noted by Williams et al. [2003], geometric algorithms are subject to numerical instability from finite precision in real implementations, and even errors with very low probability of occurrence tend to arise in situations such as path tracing where trillions of intersection computations are invoked per frame. We ran a full path tracer on a glass Cornell Box (Figure 5) to stress-test intersections in tricky cases such as at or inside of the box, and on a complex voxel model (Figure 6), where many boxes and many reflections were involved.

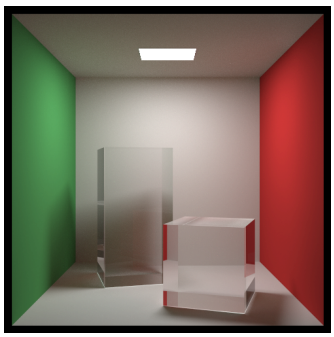

Figure 5. Path-traced Cornell Box with glass showing the numerical robustness of our intersector.

Figure 6. Path-traced voxel scene showing the numerical robustness of our intersector.
City courtesy of Sir_Carma, characters by ephtracy

6.2. Ray-Box Performance

We measured GPU ray-box performance with a test harness that cast millions of rays at random points within twice the radius of random boxes (to stress both the hit and miss branches). We report results on NVIDIA Titan V and GeForce 1080 using driver 391.01, both on Windows 10. These GPUs have different processor architectures (Volta and Pascal) and different memory (685 GB/s HBM2 on a 3072-bit bus and 320 GB/s GDDR5 on a 256-bit bus). Analysis of a suite of algorithms on such radically differently-provisioned algorithms gives a sense of their general suitability and scaling on vector processors, rather than limiting analysis to the characteristics of one specific machine.

Performance results are shown in Tables 1–3. Table 1 shows a hit-only AABox test as would be used in a BVH efficiency structure, where the efficient slabs method wins by a wide margin. To our knowledge, this important efficient slabs method used by industry ray tracers is not widely known.

Table 2 shows AABoxes with normal and distance computations. Our method outperforms all previous methods in this case. Table 3 shows application in the general case, to OBoxes with normal and distance computations. The new method also outperforms all others here.

Algorithm	Titan V	GeForce 1080
Mahovsky and Wyvill [2004]	0.012 ns/ray	0.079 ns/ray
Woo [1990]	0.022 ns/ray	0.059 ns/ray
Kay and Kajiya [1986]	0.010 ns/ray	0.030 ns/ray
Williams et al. [2003]	0.0007 ns/ray	0.014 ns/ray
Scratchapixel [2016]	0.002 ns/ray	0.017 ns/ray
efficient slabs [Laine et al. 2013; Áfra et al. 2016]	**0.0002** ns/ray	**0.007** ns/ray
ours	0.013 ns/ray	0.011 ns/ray
ours outside*	0.013 ns/ray	0.011 ns/ray

Table 1. Axis-aligned box hit only.

Algorithm	Titan V	GeForce 1080
Mahovsky and Wyvill [2004]	0.267 ns/ray	0.293 ns/ray
Woo* [1990]	0.062 ns/ray	0.103 ns/ray
Kay and Kajiya [1986]	0.034 ns/ray	0.062 ns/ray
Williams et al. [2003]	0.034 ns/ray	0.051 ns/ray
Scratchapixel [2016]	0.037 ns/ray	0.052 ns/ray
efficient slabs [Laine et al. 2013; Áfra et al. 2016]	0.008 ns/ray	0.033 ns/ray
ours	**0.006 ns/ray**	**0.026** ns/ray
ours outside*	0.005 ns/ray	0.021 ns/ray

Table 2. Axis-aligned box with normal and distance.

Algorithm	Titan V	GeForce 1080
Mahovsky and Wyvill [2004]	0.288 ns/ray	0.297 ns/ray
Woo* [1990]	0.051 ns/ray	0.104 ns/ray
Kay and Kajiya [1986]	0.030 ns/ray	0.070 ns/ray
Williams et al. [2003]	0.027 ns/ray	0.057 ns/ray
Scratchapixel [2016]	0.030 ns/ray	0.055 ns/ray
efficient slabs [Laine et al. 2013; Áfra et al. 2016]	0.012 ns/ray	0.041 ns/ray
ours	**0.008** ns/ray	**0.030** ns/ray
ours outside*	0.007 ns/ray	0.028 ns/ray

Table 3. Oriented box with normal and distance. (* produces incorrect intersection when the ray origin is in the box)

6.3. Native Voxel Rendering

From the fast intersection algorithm, we now build a method for native rendering of dynamic voxels, without first converting them to meshes or a spatial data structure. GPU capabilities are exposed differently under OpenGL and Vulkan/DirectX. We describe an OpenGL version appropriate for scientific visualization applications and a slightly different Unreal Engine 4 version appropriate for games, for which we provide full source code in the supplement (`http://www.jcgt.org/published/0007/03/04/supplement.zip`)

For the OpenGL implementation, we rasterize as a `GL_POINT` (axis-aligned square) the 2D bounding box (computed in a vertex shader) of each 3D box and then ray trace against (in a pixel shader) each pixel in that 2D box. This avoids rasterizing all triangles that make up a box (2×-12× improvement) and also reduces bandwidth to the vertex shader. Some care has to be taken to clip properly at the $z = 0$ singularity.

Given those independent speedup factors, it is not surprising that the combination of these optimizations gives approximately a 2× net speedup for voxels near the camera and 20× for distant voxels. We gain a further modest 1.2× speedup by stochastic pruning [Cook et al. 2007] of voxels with subpixel projected area as implemented in Listing 3. The net result is a 7× speedup in rendering over the full depth range for a game like *Minecraft* (Figure 7); this is closer to the best case than the worst, because under perspective projection most visible voxels are far from the camera. Figure 8 compares the performance of mesh rasterization and our direct voxel rendering, scaling with resolution and voxel count (both methods are linear in both parameters). Table 4 shows performance in the large scenes of Figures 1 and 9 for varying viewpoints.

The video results (from our OpenGL implementation, captured on GeForce 1080 at 1920 × 1080 resolution at 60 fps, including shading and post-processing) demonstrate the stability and robustness of the full rendering method. The first scene is the

Figure 7. *King's Landing*, a well-known epic Minecraft model of eight Mvoxels, rendered by our method in the Unreal Engine 4 with the full effects pipeline from https://www.planetminecraft.com/project/showcase---kings-landing-an-epic-city-1843386/.

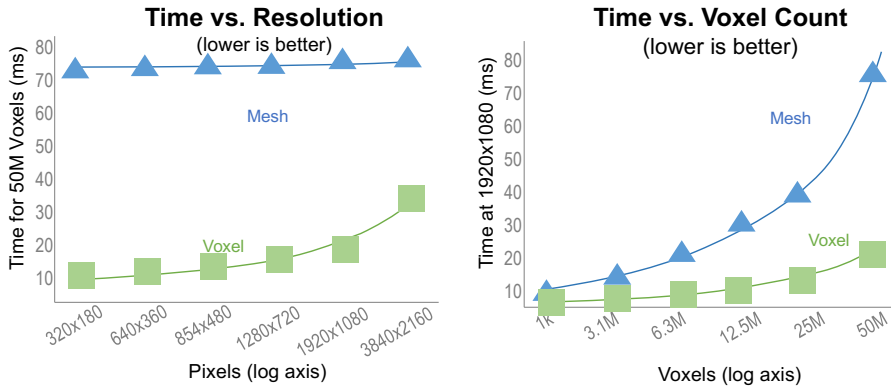

Figure 8. Render time for rasterization and our voxel renderer vs. voxel and pixel count.

Method	Fig. 1 Best	Fig. 1 Worst	Fig. 9 Best	Fig. 9 Worst
Mesh	75.1 ms	80.8 ms	15.2 ms	38.5 ms
Ours	8.3 ms	13.9 ms	2.5 ms	6.1 ms

Table 4. Voxel rendering performance on GeForce 1080 for various large voxel models and viewpoints at 1920×1080.

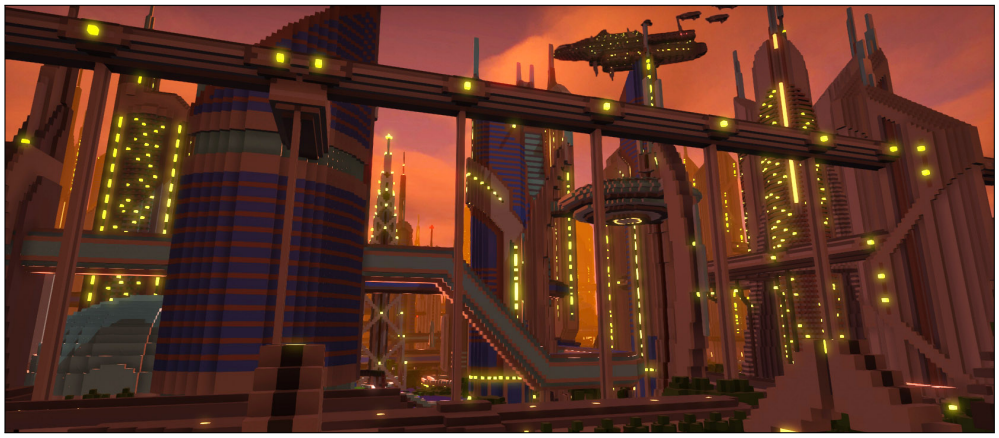

Figure 9. Real-time voxels rendered for a large animated (see the video) science fiction city that appears in several of our performance measurements. Every voxel moves independently when the city disintegrates. The full model can be downloaded from `https://www.planetminecraft.com/project/future-city-3149015/`.

large 53 Mvoxel scene from Figure 1. Note that clipping is correct near the camera, and that the stochastic pruning is stable in the distance (particularly where buildings are silhouetted against the sky). There is some color flicker in the grass from the H.264 compression in the video which is not present when the program's output is viewed directly. The second scene is the science-fiction city from Figure 9. We used a vertex shader to independently move each voxel in the entire scene to create a disintegration animation. Because, unlike previous fast voxel methods, our method uses no precomputation, this extreme case of animation has zero impact on rendering performance.

In addition to performance, direct voxel rendering by our method has the advantage that each voxel can be independently animated as there is no static mesh or complex data structure. Because we still test and write to the depth buffer and submit as a normal draw call, this technique also integrates into scenes with meshes, rasterization, and post-processing effects. To demonstrate robustness of our direct voxel ray-tracing method for primary rays under different rendering strategies, we show it with phenomenological transparency (Figure 10), deferred deep G-buffer radiosity (Figure 11), and forward rendering (Figure 1).

We measured the impact of our optimizations for screen-space bounding box calculation. On a viewpoint which exhibits both large voxels close to the camera, and small voxels further-away, our approach is $\approx 17\%$ faster (for the G-buffer generation pass) than a naive bounding box calculation, projecting the eight vertices individually.

7. Summary

We have presented an algorithm for computing ray intersection and normal computation with oriented boxes. Our algorithm is not well-suited for the "does the ray hit at all" axis-aligned bounding volume query where the slabs method from Listing 1 is likely to remain dominant. This algorithm maps well to hardware and renders very large voxel models at interactive speeds; it renders large voxel models faster than any method we are aware of, and in particular is faster than using the hardware rasterizer for tessellated boxes. Our algorithm is a good example of how efficient GPU programs are constructed now. We combined a core geometric/algorithmic observation with a hardware-aware implementation that avoids branches and their divergence. We took that kernel and extended it with known algorithmic optimizations to make

Figure 10. Real-time voxels rendered with emission and fog by order-independent transparency, showing the interaction of our voxel renderer with an algorithm developed for triangles. Model courtesy of Thibault Simar `http://ex-machina.fr`

Figure 11. Caverns with deep G-buffer radiosity, showing the interaction of our voxel renderer with a rendering algorithm developed for triangle meshes. Model courtesy of Sir_Carma

a complete and efficient GPU renderer. Finally, we tested this on two, very different GPU architectures to make sure the implementation wasn't "overfit" to a specific one and provided implementations for one of the dominant game engines and suitable for inclusion in custom engines or research code.

Acknowledgements

Thanks to Ingo Wald for making us aware of the efficient slabs test and discussion of the literature on the subject, and for advising us on how to best compute the normal in previous algorithms; Max McGuire for supplying us with the Subnautica images and data; SirCarma, Thibault Simar, EphTracy for sharing their voxel models and advising us; Sabbas Apterus and Zeemo_2 for the Minecraft models in Figures 1 and 9; Andrew Willmott for careful editing; and Eric Haines for Minecraft model support.

References

ÁFRA, A. T., WALD, I., BENTHIN, C., AND WOOP, S. 2016. Embree ray tracing kernels: Overview and new features. In *ACM SIGGRAPH 2016 Talks*, ACM, New York, NY, USA, 52:1–52:2. 69, 75, 76

BARRETT, S., 2015. stb_voxel_render.h. GitHub, C++ Library. URL: https://github.com/nothings/stb/blob/master/stb_voxel_render.h. 68

COOK, R. L., HALSTEAD, J., PLANCK, M., AND RYU, D. 2007. Stochastic simplification of aggregate detail. *ACM Transactions on Graphics 26*, 3. 76

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 15–22. 68

CSÉBFALVI, B., TÓTH, B., BRUCKNER, S., AND GRÖLLER, M. E. 2012. Illumination-driven opacity modulation for expressive volume rendering. In *Proceedings of Vision, Modeling & Visualization 2012*, Eurographics Association, Goslar Germany, 103–109. URL: https://www.cg.tuwien.ac.at/research/publications/2012/Csebfalvi-2012-IOM/. 68

EISEMANN, M., MAGNOR, M., GROSCH, T., AND MULLER, S. 2007. Fast ray/axis-aligned bounding box overlap tests using ray slopes. *Journal of Graphics Tools 12*, 4, 35–46. 69

KAY, T. L., AND KAJIYA, J. T. 1986. Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '86, 269–278. 69, 75, 76

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, ACM, New York, NY, USA, HPG '13, 137–143. 69, 75, 76

LYSENKO, M., 2012. Meshing in a minecraft game, june. Blog post. URL: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/. 68

MAHOVSKY, J., AND WYVILL, B. 2004. Fast ray-axis aligned bounding box overlap tests with Plücker coordinates. *Journal of Graphics Tools 9*, 1, 35–46. 69, 75, 76

SCRATCHAPIXEL, 2016. A minimal ray-tracer: Rendering simple shapes. http://www.scratchapixel.com. Accessed: 2016-09-15. 69, 75, 76

SIGG, C., WEYRICH, T., BOTSCH, M., AND GROSS, M. 2006. GPU-based ray-casting of quadratic surfaces. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, SPBG'06, 59–65. 72

WILLIAMS, A., BARRUS, S., MORLEY, K., AND SHIRLEY, P. 2003. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools 10*, 54. 69, 74, 75, 76

WOO, A. 1990. *Graphics Gems*. Academic Press Professional, Inc., San Diego, CA, USA, ch. Fast Ray-box Intersection, 395–396. 69, 75, 76

Author Contact Information

Alexander Majercik
NVIDIA
2788 San Tomas Expressway,
Santa Clara CA 95051, USA
amajercik@nvidia.com

Cyril Crassin
NVIDIA
2788 San Tomas Expressway,
Santa Clara CA 95051, USA
ccrassin@nvidia.com

Peter Shirley
NVIDIA
2788 San Tomas Expressway,
Santa Clara CA 95051, USA
pshirley@nvidia.com

Morgan McGuire
NVIDIA
2788 San Tomas Expressway,
Santa Clara CA 95051, USA
mcguire@nvidia.com

Majercik, Crassin, Shirley, McGuire, A Ray-Box Intersection Algorithm, *Journal of Computer Graphics Techniques (JCGT)*, vol. 7, no. 3, 66–81, 2018
http://jcgt.org/published/0007/03/04/

Received:	2018-07-01		
Recommended:	2018-08-27	Corresponding Editor:	Andrew Willmott
Published:	2018-09-20	Editor-in-Chief:	Marc Olano


BY ND