Real-time Rendering of Voxels Using Ray Tracing

Emil Olofsson

Computer game programming, Luleå University of Technology

S0007E: Specialization project in game development

March 17, 2025

Abstract

Rendering highly detailed voxel-based worlds in real time presents significant challenges, particularly in efficiently storing and accessing dense voxel data. This project explores the use of brick maps to optimize memory usage and provide real-time performance. The brick map consists of a three-dimensional grid where each cell either references a brick containing voxel data or remains empty to conserve memory. This approach balances performance and memory efficiency, making it suitable for high-detail voxel environments.

1. Introduction

The rendering of voxel-based worlds has traditionally relied on constructing triangle meshes from voxel data and rendering them using rasterization, as seen in games like Minecraft (2009) and Cube World (2013). However, these games utilize large voxels, approximately one cubic meter in size. Modern voxel-based games, such as Teardown (2022), employ significantly smaller voxel sizes, around one cubic decimeter, requiring a thousand times more voxels to represent the same volume as a single block in Minecraft. To achieve real-time rendering with such dense voxel grids, ray tracing is utilized. The primary challenge in this approach is not rendering the voxels themselves but efficiently storing them. This project explores the use of brick maps as an efficient storage method for voxel data while ensuring real-time rendering performance.

2. Background and Related Work

2.1 Ray-Stepping Algorithm

Voxel rendering in this project is based on the ray-stepping algorithm introduced by John Amanatides and Andrew Woo[1]. This algorithm efficiently traces rays through a voxel grid by incrementally stepping from one voxel to the next along the ray's path.

2.2 Voxel Storage Techniques

The most straightforward method for storing voxels is using a uniform 3D grid where each cell corresponds to a voxel. While sufficient for small scenes, this approach becomes impractical for larger environments due to its exponential memory requirements.

A more efficient method involves the use of brick maps, a two-level hierarchical grid structure. Instead of storing voxel data directly in the top-level grid, it only contains pointers to bricks (sub-grids) that store the actual voxel information. This method reduces memory usage by eliminating the need to store empty regions of space. Thijs van Wingerden[2] has previously implemented brickmaps as a means of storing voxel data efficiently. His work serves as a reference point for this project, particularly in evaluating memory consumption and rendering performance.

Another widely used approach for efficient voxel storage is the Sparse Voxel Octree (SVO), as implemented by Laine and Karras[3]. SVOs work by recursively dividing the scene, allowing large homogenous sections to be stored as a single node, significantly reducing memory consumption. While SVOs offer high compression rates, they introduce excessive overhead in managing dynamic updates since they need to be either partially, or entirely rebuilt; additionally, they require more complex traversal logic. In contrast, brick maps provide a balance between storage efficiency and real-time modification flexibility, making them well-suited for interactive voxel environments.

3. Implementation

3.1 Data Structure Implementation

The top level of the brick map, the brick grid, is a three-dimensional grid that contains the entire scene. It is composed of 32-bit unsigned integers, representing pointers to the individual bricks where the raw voxel data is stored, a value of *0xFFFFFFFF* means that the brick is empty. The bricks are split into two components, a bitmask representing solid voxels, and the color data. The bricks have the dimensions $D = (8, 8, 8)$ as defined by Wingerden. The color data is stored separately to minimize strain on the memory bandwidth when traversing, it also allows multiple bricks to share the same texture.

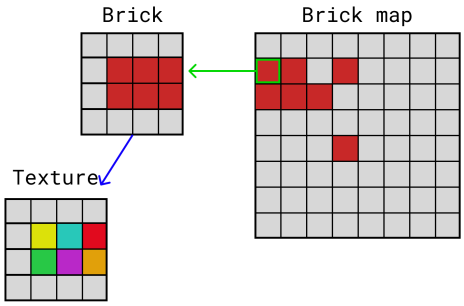

Figure 1: *Data structure visualization.*
Brick map: *filled cells represent a brick that contains a pointer to a brick.*
Brick: *filled cells represent a solid voxel.*
Texture: *All cells represent a color.*

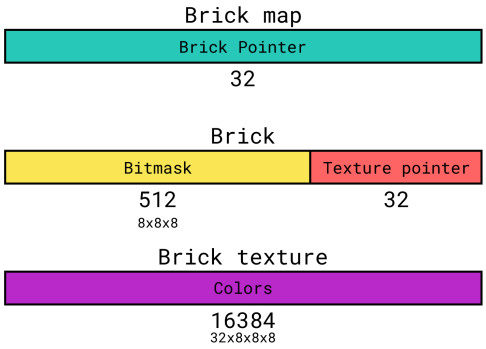

Figure 2: *Data structure bit representation.*

3.2 Rendering

The rendering uses a 3D DDA algorithm to find voxels in the grid. The algorithm begins by performing a ray-bounding box intersection test to find the first cell in the brick map, then it steps through the grid in the direction of the ray one cell at a time. Once a brick containing a brick pointer is found, the same algorithm is performed on the brick, if the ray exits the brick without finding a voxel, we continue stepping through the brick map. Voxel normals can be calculated during rendering by keeping track of which direction the last step was made. A more detailed explanation of the algorithm can be found in the appendix.

3.3 Editing

The voxel data is stored on both the CPU and GPU. To select where in the scene to modify, a ray is cast from the camera, which uses the same algorithm as the rendering to traverse the grid. Once a voxel is found we take one step back to find where the new voxel should be placed. When inserting a voxel where a brick already exists the brick's bitmask and texture need to be modified, if a brick does not exist at the location a new one has to be created. Once the brick map data has been updated on the CPU we send the data to the GPU.

3.4 Model loading

To convert meshes into voxel scenes, the mesh is first split into an octree, the depth of the octree is defined by the user. After the mesh has been subdivided depth-first search is performed and we find all octree nodes that contain one or more triangles. The center of the node is projected onto one of the triangles and a texture coordinate is calculated by interpolating between the coordinates defined in the triangle vertices, then a voxel with the color sampled from the mesh's texture is inserted into the brick map.

4. Results

The goal of this project was to create a system that can render large voxel scenes in real-time within reasonable memory constraints. Since the entire scene is always stored, and the size of the brick map components is known, the calculations for memory consumption are simple. The equation for memory consumption (in bytes) is:

$$M = 4(D_x \cdot D_y \cdot D_z) + 20\,B_b + 2048\,B_t$$

where D is the dimensions of the top-level grid, B_b is the number of bricks, and B_t is the number of brick textures. The resulting memory consumption will be compared to Wingerden's brick maps, and Laine & karass' sparse voxel octrees. The experiments are run on three different scenes: Hairball, Sibenik, and Conference room (Figure 3, 4, 5), all of which can be found at *McGuire Computer Graphics Archive*[4].

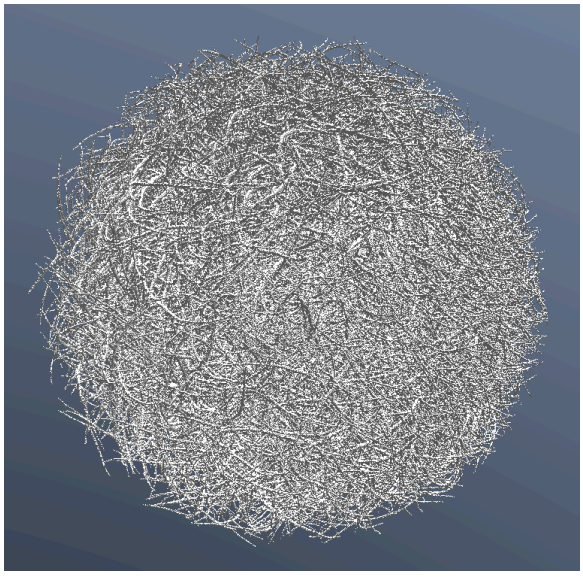

Figure 3: *Hairball 1024³*

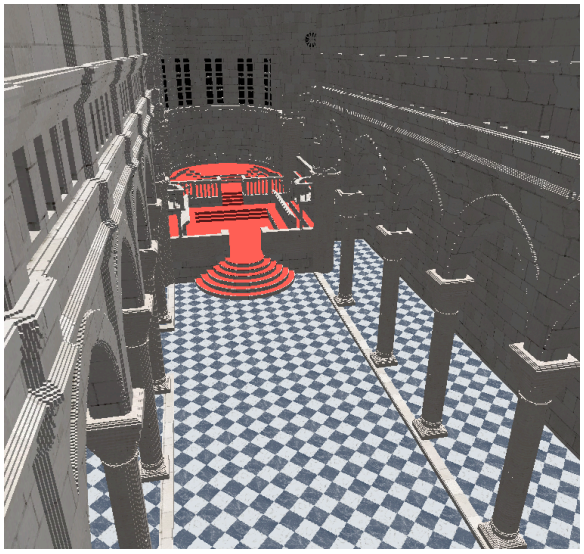

Figure 4: *Sibenik 2048³*

Figure 5: *Conference room 2048³*

Scene	Brick map (our)	Brick map (Wingerden)	SVO
Hairball 1024	1074	23	262
Sibenik 2048	749	-	132
Conference 2048	484	100	89

Table 1: *GPU memory consumption in MB for each test scene in different implementations.*

The reason for the large discrepancy between our implementation and Wingerden's is that they employ a streaming system, which allows them to not store the entire scene on the GPU at once, instead requesting only what is visible. They also store the voxel data as compressed RGB values, as opposed to our uncompressed RGBA voxels, this in and of itself reduces the size of their raw voxel data by 87.5% compared to ours.

5. Conclusion and Future Work

This project demonstrates that brick maps offer an efficient solution for storing and rendering voxel worlds in real-time, although this implementation leaves room for improvement. The system is adequate for medium-sized scenes but struggles for larger ones. Future improvements could include adding a grid above the current top level to accelerate ray traversal, particularly when stepping over large empty areas. Implementing voxel compression would enable support for larger scenes by reducing memory requirements. Additionally, enhancing visual fidelity through ray-traced reflections, shading, ambient occlusion, and global illumination would significantly improve the overall rendering quality.

References

1. Amanatides, J., & Woo, A. (1987). A Fast Voxel Traversal Algorithm for Ray Tracing
2. van Wingerden, T. (2015). Real-time Ray Tracing and Editing of Large Voxel Scenes
3. Laine, S., & Karras, T. (2010). Efficient Sparse Voxel Octrees
4. McGuire, M. (2018). https://casual-effects.com/data/

Appendix

Ray marching algorithm

Ray-box intersection test

The first step in the algorithm is to determine if the ray intersects with the voxel grid. This is done by computing an intersection test with an axis-aligned bounding box (AABB) that covers the entire grid. The intersection test returns true if a collision is found, as well as the values of *t* for the entry and exit points of the ray.

A ray is defined by an origin point and a direction vector, and the equation for it is $p = \vec{o} + \vec{d} \cdot t$ for $t \geq 0$, where *o* is the origin and *d* is the direction. An AABB is defined by a minimum- and maximum point.

```
bool Intersect(Ray ray, BoundingBox box, out float tNear, out float tFar) {
  vec3 t1 = (box.min - ray.origin) / ray.direction;
  vec3 t2 = (box.max - ray.origin) / ray.direction;
  vec3 vMin = min(t1, t2);
  vec3 vMax = max(t1, t2);
  tNear = max(vMin.x, max(vMin.y, vMin.z));
  tFar = min(vMax.x, min(vMax.y, vMax.z));
  return tNear < tFar && tFar > 0;
}
```

DDA initialization

The ray marching itself uses a variant of the digital differential analyzer (DDA) line drawing algorithm. The first step of the initialization phase is to determine the starting voxel. If the ray begins inside of the grid, the starting voxel is equal to the floored value of the ray origin divided by the size of the voxels. If the ray origin is outside of the grid bounds the ray is first projected onto the grid by using the *tNear* value from the intersection test, then divided by the voxel size, and lastly floored. The second step in the initialization is determining the step direction for each axis. Each component of the step direction vector will be assigned a value of either 1, 0, or -1, determined by the sign of $\vec{d}$.

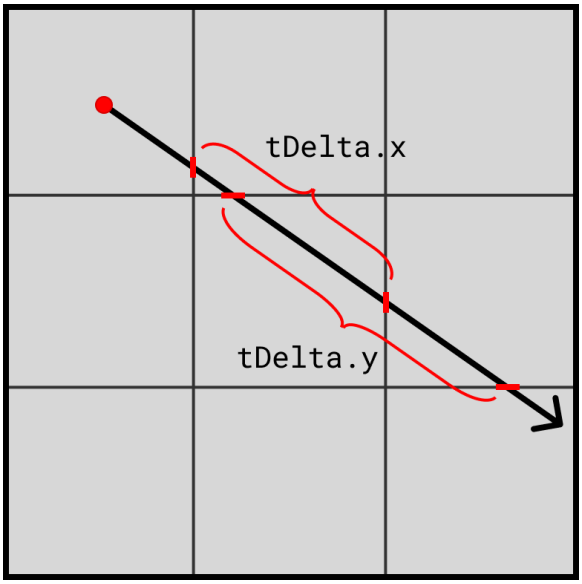

Figure 6: *Visual representation of the tDelta variable.*

The third step is to initialize the vector *tDelta*, where each component indicates how far is needed to travel along the ray for such a movement to be equal to the width of a voxel. *tDelta* is calculated by taking the step direction and dividing it by $\vec{d}$.

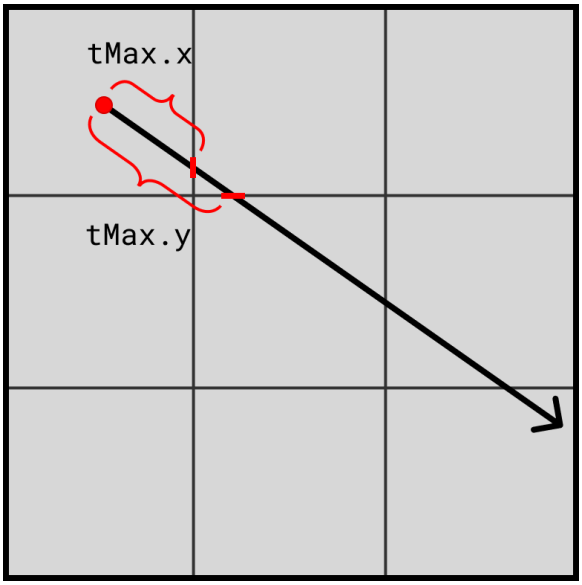

Figure 7: *Visual representation of the tMax variable.*

The last step is to calculate the value of *t* where the ray crosses the next voxel boundary in each axis and the result is stored in the vector *tMax*, e.g. the *x* component of *tMax* represents the distance along the ray from $\vec{o}$ to the next vertical voxel boundary. To find *tMax* we perform a ray-plane intersection.

```
bool RayMarch(Ray ray, BoundingBox box) {
  float tNear, tFar;
  if (!Intersect(ray, box, tNear, tFar)) return false;

  // By using the max value of 0 and tNear we account for the ray
  // starting inside the grid, and outside of it.
  vec3 rayStart = (ray.origin + ray.direction * max(0, tNear)) / voxelSize;
  ivec3 voxelPosition = floor(rayStart);
  ivec3 stepDirection = ivec3(sign(ray.direction));
  vec3 tDelta = stepDirection / ray.direction;
  vec3 tMax = tDelta * (stepDirection * ((rayStart - voxelPosition) + 0.5));

  ...
}
```

DDA traversal

The ray marching begins by first checking if the starting voxel contains any data, if it does, we can exit the function, otherwise need to continue stepping through the grid. To determine in which axis we should step, we need to find the smallest component of *tMax*. After the smallest value in *tMax* is found we increment it with the corresponding component from *tDelta*, we also increment *voxelPosition* by *stepDirection*. Below is the full ray marching function.

```
bool RayMarch(Ray ray, BoundingBox box) {
  float tNear, tFar;
  if (!Intersect(ray, box, tNear, tFar)) return false;

  // By using the max value of 0 and tNear we account for the ray
  // starting inside the grid, and outside of it.
  vec3 rayStart = (ray.origin + ray.direction * max(0, tNear)) / voxelSize;
  ivec3 voxelPosition = floor(rayStart);
  ivec3 stepDirection = ivec3(sign(ray.direction));
  vec3 tDelta = stepDirection / ray.direction;
  vec3 tMax = tDelta * (stepDirection * ((rayStart - voxelPosition) + 0.5));

  while (InBounds(voxelPosition)) {
    if (VoxelAt(voxelPosition)) {
      return true;
    }
    int stepAxis =
      tMax.x < tMax.y ? (tMax.x < tMax.z ? 0 : 2) : tMax.y < tMax.z ? 1 : 2;
    tMax[stepAxis] += tDelta[stepAxis];
    voxelPosition[stepAxis] += stepDirection[stepAxis];
  }
  return false;
}
```