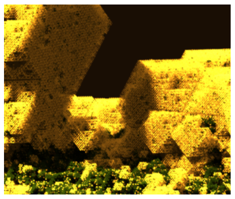
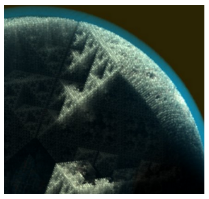
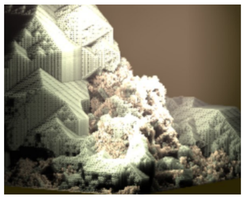
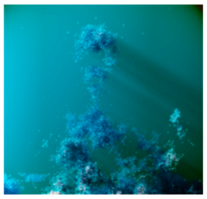
Voraldo v1.1:
A GPU-Based Voxel Editor with Support for Volumetric Lighting Effects

Jonathan A. Baker[*]

Ohio University
jb239812@ohio.edu

January 15, 2021

Abstract

Volume graphics in realtime applications had seen some development in the 1990's before being largely superceded by rasterized polygons due to memory limitations. Today, commodity graphics hardware provides large, fast memory buffers and a highly parallel model for computation. This paper presents a demonstration of volume modeling functions and a realtime rendering method which traces rays through RGBA volume data for each pixel. Also presented is a realtime lighting model which allows for volumetric lighting effects, by considering how light rays traverse this volume.

I. Introduction

Voxels, or volume elements, are a discrete means to represent volume data. Here, they are regular cubes, arranged orthogonally, and can be thought of as a 3D raster. A single value, or set of values, exists and is stored for each volume element. This value can be associated with any point in the space defined by the voxel's extents. This concept has seen applications in physical simulation as well as computer graphics.

This project explores applications for this representation in the domain of realtime graphics. In the 1990s, there was contention on best practices for realtime hardware. As the market settled, polygonal representations would become the de facto standard for producing realtime 3D graphics. The math governing their use is well understood. Transforms exist which allow arbitrary rotation, scaling, shearing, flipping, and many other types of operations to be performed on models defined as a set of triangles. Due to the simplicity of this shape – three vertices – there is an unambiguous definition of geometry: these three points define a plane, as long as they are noncollinear.

Competing rendering technologies existed, though many have largely fallen out of use. Voxels are one example of this. It is common today to see polygonal representations of voxels, but until very recently, it was not common to see them used directly as a realtime rendering method [4]. The following is a list of games which have made use of this representation for realtime rendering purposes. This list is not complete, but it represents a diverse set of approaches to getting an image to the screen.

- Commanche (Novalogic, 1992)
- Blade Runner (Westwood, 1997)
- Shadow Warrior (3DRealms, 1997)
- Blood (Monolith, 1997)
- Hexplore (Heliovision, 1998)

[*] `https://jbaker.graphics/` for more information.

- Outcast (Appeal, 1999)
- RollerCoaster Tycoon (Chris Sawyer, 1999)
- Command and Conquer: Tiberian Sun (Westwood, 1999)
- Dwarf Fortress (Bay 12, 2006)

More recently, with approaches like marching cubes [1][1] and dual contouring [5] being applied to realtime applications, there exist relatively general methods to convert from a volume representation to a polygonal surface representation. This project centers around potential benefits of the direct volume representation, and why it might be considered as an alternative.

II. Methods

i. Representation

The project almost entirely eschews the use of OpenGL geometry, in order to avoid problems with the order dependence in alpha blending[2]. Two triangles are used to cover the screen, in order to get a fragment and a corresponding shader invocation for every pixel on the screen [7]. The voxel data itself is kept using OpenGL's 4.2 Image Load/Store functionality [9]. This provides a GLSL interface called images, which allow for read/write access where texture samplers provide read-only access. Three sets of image3D objects are used: color, mask, and lighting, the use of which are detailed in the following sections. Color holds four-channel RGBA values, while mask and lighting are single-channel images with only one value per voxel. The use of this type of buffer has allowed almost all computation to be moved to the GPU.

ii. Rendering

Logically, the rendering process is done per-pixel in a fragment shader. The reality is a little more complicated, in order to implement a two-fold optimization that makes the program much more efficient. A framebuffer image is used to hold the result and present it to the user.

First, this process decouples the display resolution from the screen resolution, using a scale factor defined at compile time. A value greater than one will allow for supersampling, while a value less than one can be used for environments with limited performance, such as an integrated GPU in a laptop. The display treats the framebuffer image as a texture, using multisampling and linear texture filtering to see the benefit from this scale factor.

The second part of this optimization comes from the fact that unless the content, orientation, or scaling of the voxel block changes, there is no need to produce a new image. There is a redisplay flag set by the functions which perform these operations. When a framebuffer update is called for, a tile-based renderer in a compute shader is invoked. The use of tiles keeps the operation flexible enough to deal with varying framebuffer image sizes, and any portion of a tile that falls outside of the image's extents is ignored. The result is then sampled in order to produce the image output.

The renderer itself is based on compositing samples through the volume to derive final pixel color. This process starts by computing a ray origin and ray direction for each pixel. Starting with a set of basis vectors, two user-controlled rotations are applied to represent a pair of euler angles. By stepping back from the origin along the rotated Z vector, you arrive at a point at the center of the image. The X and Y vectors are used to compute each individual pixel's offset from this point. Adding this offset to the point at the center of the image, the ray origin is known. For orthographic rendering, the ray direction is simply the rotated Z vector.

The next step considers a parametric form of this ray, and uses it to test against an AABB which represents the extents of the voxel block. If the ray misses, this shader invocation has no more work to do. If the ray hits, near and far intersection points are established in the form

[1]Originally developed for visualization of medical data like CAT, MRI and PET scans.

[2]For more information on past approaches, including methods for overcoming alpha blending limitations in OpenGL geometry [6] see `https://jbaker.graphics/writings/voraldo_history.html`

of two floating point numbers, `tnear` and `tfar`.

By sampling through the volume from the farthest intersection point to the nearest intersection point, simple 'over' style alpha blending can be performed [12]. Using the raw alpha channel proved to be limiting[3] as very little of the range was usable. To counter this limitation, the value held in the alpha channel is raised to a user-adjustable power to get the number that will be used in the computation.

In this computation, two buffers are referenced – one containing the current color and opacity, and one containing the current lighting data. The value in the lighting buffer is used to scale the intensity of the R, G and B color channels' contributions to the blending operation. The pixel color, C, is initialized with the OpenGL clear color, and is updated using a color sample, T, and a light sample, L, according to the following recurrence relation:

$$\begin{aligned} C_{n+1}.rgb = {} & (T.rgb \cdot 4.0 \cdot L) \cdot T.a^{\alpha power} \\ & + C_n.rgb \cdot C_n.a \cdot (1 - T.a^{\alpha power}) \end{aligned}$$

$$C_{n+1}.a = T.a^{\alpha power} + C_n.a \cdot (1 - T.a^{\alpha power})$$

A series of samples is composited between the intersection points, based on some given step size and a maximum number of allowed steps. The final value of C.rgb is stored in the framebuffer image.

The factor of four was determined experimentally to give a usable range, with enough headroom to create highlights. A value of zero will zero out the color contribution of the voxel under consideration. A value of 0.25 will cancel the scale factor to represent a neutral level of lighting. Higher lighting values will begin to saturate the color at values of 1.0 for all three channels, making the contribution go to white[4]. The use of this separate lighting buffer is to decouple the logic governing the display from the logic governing the computation of the volumetric lighting.

[3]GLSL normalizes these 8-bit values, and that assumption is carried through the equations in this paper – when referencing a texture, the value returned is the value stored in memory, divided by 256. The alpha power is applied to this floating point number, in the range 0.0–1.0.

[4]Note that if there are small values or zeroes in the R, G or B channels, it will not go to white, but it will eventually move towards saturation of the nonzero channels.

iii. Modeling

There are a number of moving parts to the modeling operation in order to provide smooth interaction to the user and handle GPU syncronization. There are two sets of buffers involved: color and mask. Inspired by the double-buffering technique used by most graphics APIs, two copies of each are kept. This ensures that for any given drawing operation, one image representing the current state is read from and the other is written to, which represents the new state of the data. The display code only references the current color buffer. With appropriate use of memory barriers, and due to the single-threaded nature of OpenGL, there is no ambiguity as to which buffer is which.

The drawing functions handle a number of different primitives, more than is appropriate to list here. They fall into two classes: those which use a boolean `is_inside()` function to determine if a given voxel's location is inside the shape and those which use a separate load buffer to get data from the CPU. Both of these treat the mask the same way. As a 1-bit value, it determines whether or not that voxel can be written to – if a cell is masked, it will retain its contents, even if it would be affected by the modeling functions. The shader associated with each drawing operation has independent toggles for drawing and masking. Also included are several functions to manipulate this mask, such as inverting, clearing, or masking ranges in the color channels[5].

The functions which use the `is_inside()` function could easily be extended to handle arbitrary signed distance fields. The best way to handle this interface has not yet been identified, but this would be a good place to incorporate in-engine scripting to define an arbitrary `is_inside()` function. There are only a few load buffer functions including loading of voxel data from a PNG image on disk and

[5]E.g. mask all voxels with a value in the red channel greater than 0.5.

Brent Werness' Voxel Automata Terrain (VAT) [11]. VAT is an algorithm converted from the original Processing implementation to C++. It generates highly varied voxel forms based on a sort of 3D analog to the Diamond-Square algorithm. This has been very useful in testing the implementation of the lighting functions. In both of these cases, the data is acquired and buffered to the GPU, before a compute shader is invoked to copy the data with a toggle to either respect or ignore the state of the mask.

The combination of drawing and masking functions has proved to be a very powerful way to produce complex voxel models. By keeping this 3D raster data on the GPU and manipulating its contents with these various compute shaders, many operations can be applied over one another in interesting and creative ways.

iv. Lighting

The lighting concepts are where we begin to see more benefit from the 3D raster. By using this representation, it becomes easier to talk about what is happening along a ray which traverses the volume. At any point along the ray's length the volume data can be queried to determine the opacity of the space.

There are a number of lighting functions that are defined: point and directional lighting, ambient occlusion, and an approximation of global illumination.

Point lights are computed using a shader invocation for each voxel. The inputs to this shader are light position, initial light intensity, decay power, and distance power. The relative position of the light with respect to the voxel is used to determine the vector along which the light will travel, with the light position also serving as the origin point for that ray. There is code to handle the cases in which the light is inside the voxel under consideration or if the light position falls outside the volume.

Similar to how the display function operates, there is a process of sampling along the ray in discrete steps. In this case, it is done forwards, rather than backwards, and tracks the ray intensity as it is attenuated from a user defined initial value. The value of decay power determines how much the intensity of the ray is attenuated by the alpha value at sampled points. This also can be expressed as a recurrence relation:

$$Intensity_{n+1} = Intensity_n \cdot (1 - alpha^{decaypower})$$

This is iterated until the ray reaches the location of the current voxel. The value of distance power is then used to compute a scale factor that approximates the inverse square law, at the default value of 2.0. It can be set to zero to have the scale factor go to one, or raised to a higher power to decay more in a shorter distance. The final result of the computation is then added to the existing value in the voxel in the lighting buffer associated with the shader invocation.

Directional lights are computed in a very similar fashion, but have no divergence. They use a uniform vector for all shader invocations, as if the light was at infinite distance. The inputs are a pair of euler angles and decay power. The euler angles are used to express the direction of the light vector. The shader does a ray-box intersection to determine near and far intersection points, and an almost identical process is followed from the near intersection point to the voxel's location. Distance is not considered in this computation.

Ambient occlusion is based on a variable radius neighborhood and is invoked per voxel the same as the previous two. There is logic which does a weighted sum of the neighbohring cells' alpha values according to something like a gaussian kernel, and considers a ratio of this versus the maximum possible value. The value in the lighting buffer is written as the existing value times one minus this ratio.

The structure for global illumination is a little different. The original implementation [11] was done serially on the CPU, and made some assumptions based on this sequential evaluation. This required a 2D invocation structure, which would go in slices down the y axis. For each voxel in the current slice, vectors to the nine voxels above it are considered[6]. The inputs are an alpha threshold and a sky intensity.

[6]These are the voxels which share an edge or corner with the top face of the current voxel.

The vectors make discrete steps through the volume, until one of two things happens. If they hit a voxel whose alpha channel is above the threshold, their contribution is established by some portion of the lighting value at that point. If they escape the volume before encountering an opaque enough voxel, their contribution is established by the sky intensity. The contributions of the nine rays are averaged, and the value is added to the existing value in the lighting buffer for this voxel.

The last lighting feature to consider is called 'mash'. In the same way that the display shader will apply the lighting value as a scale factor to the color data, this same operation is destrucively applied to the RGB data stored in the current color buffer. The lighting buffer can then be reset to neutral values, so the appearance of the voxel data is maintained. The primary idea for this feature was to be able to save voxel data with the lighting applied; however, interesting creative applications for this exist as well.

v. Other Manipulation Functions

Most of these features have come from using the editor. Among them are gaussian and box blurs, shifting, loading and saving, clearing and loading operations that will respect the mask, and a few functions to manipulate the mask buffer. The blurs are applied with variable radii and with the option to respect the mask as well. They diffuse color and alpha values into neighboring cells, which can create interesting effects around masked, opaque voxels. Shifting takes integer arguments for each axis, and moves each voxel to a new location, with the option to loop off the edges.

Loading and saving are used to get data to and from the GPU. The file format is a simple PNG image, which contains all the slices of the volume data enumerated out one after another, in a very tall image. Using a lossless compressed format allows for small files that still contain all the data[7]. Using some loader code for PNG images [2], the format requires almost no processing before being passed directly to OpenGL as 3D texture data.

vi. Interface Elements

There are several other improvements to the interface which made the editor much easier to use. A convenient GUI has been implemented using dearImGUI [3], which presents all the functions to the user with a tabbed layout to switch between them. It provides input widgets to set parameters and buttons that can be used to call functions. Another of these features is to deal with the fact that it is easy to become disoriented when rotating the block, with respect to where each axis is pointing. To counter this, a small orientation widget rendered with OpenGL geometry has been put together that sits down in the corner of the screen and tracks where the X, Y, and Z basis vectors are pointing[8]. This is useful so that you can achieve predictable placement of geometry and lights.

III. Conclusions and Future Directions

One of the largest benefits that comes from this voxel representation of space is a constant-time reference to a global representation of all the geometry. This makes it easier to talk about some parts of how light behaves, but lacks the normal data that would be required in order to determine accurately how it would bounce or refract. Another buffer could be added to keep the normal data when you draw new shapes, but that may not correlate well with functions like blurring operations.

The current system has a limitation in that it uses a single channel representation of light intensity. One direction for future develoments is to convert to an RGB representation, to allow for arbitrarily colored lights. Also related to

[7]For a voxel block 256 on an edge, this is a 256 by 65535 image. By comparison, a BMP image with an alpha channel would be more than 50 megabytes where these are rarely more than a few.

[8]See the appendix to for an example.

lighting, cone lights would be specified similarly to point lights, but with direction and a solid angle, perhaps something to express how it falls off towards the edges of this angle. With a current desktop GPU, the lighting functions are fast enough to support realtime animation. By caching an ambient level in one of the lighting buffers and applying point or directional lights with varying parameters each frame, this could open up more creative possibilities.

There are a few other features for which a usable interface has yet to be determined. The first is a scriptable interface for the use of things like signed distance fields, which would allow for fractals and other complex shapes to be defined by the user. Another is a copy/paste operation which would take volume data from one location and apply it elsewhere. By using 3D texture samplers, this could be extended to allow for rotation, scaling, mirrored repeats and more.

References

[1] Lorenson, W. & Cline, H. (July, 1987). Marching Cubes: A High-Resolution 3D Surface Construction Algorithm. ACM Computer Graphics, vol. 21, no. 4, pp. 163–169. Retrieved from `http://academy.cba.mit.edu/classes/scanning_printing/MarchingCubes.pdf`.

[2] Vandevenne, L. LodePNG. `https://lodev.org/lodepng/`.

[3] Cornut, O. DearImGUI. `https://github.com/ocornut/imgui`.

[4] Panteleev, A. Practical Real-Time Voxel-Based Global Illumination for Current GPUs. *GPU Technology Conference 2014*. Retrieved from `https://on-demand.gputechconf.com/gtc/2014/presentations/S4552-rt-voxel-based-global-illumination-gpus.pdf`.

[5] T. Ju, F. Losasso, S. Schaefer, and J. Warren. (July, 2002). Dual Contouring of Hermite Data. ACM Transactions on Graphics, vol. 21, no. 3, pp. 339–346, (Proceedings of ACM SIGGRAPH2002). Retrieved from `https://www.cse.wustl.edu/~taoju/research/dualContour.pdf`.

[6] Quilez, I. (2006). Volumetric Sort. `https://www.iquilezles.org/www/articles/volumesort/volumesort.htm`.

[7] Quilez, I. (2008). Rendering Worlds With Two Triangles. *NVScene 2008*. Retrieved from `https://www.iquilezles.org/www/material/nvscene2008/nvscene2008.htm`.

[8] Mendoza, N. (2013). GLSL Rotation About an Arbitrary Axis. `http://www.neilmendoza.com/glsl-rotation-about-an-arbitrary-axis/`

[9] Image Load/Store. (2019). `https://www.khronos.org/opengl/wiki/Image_Load_Store`

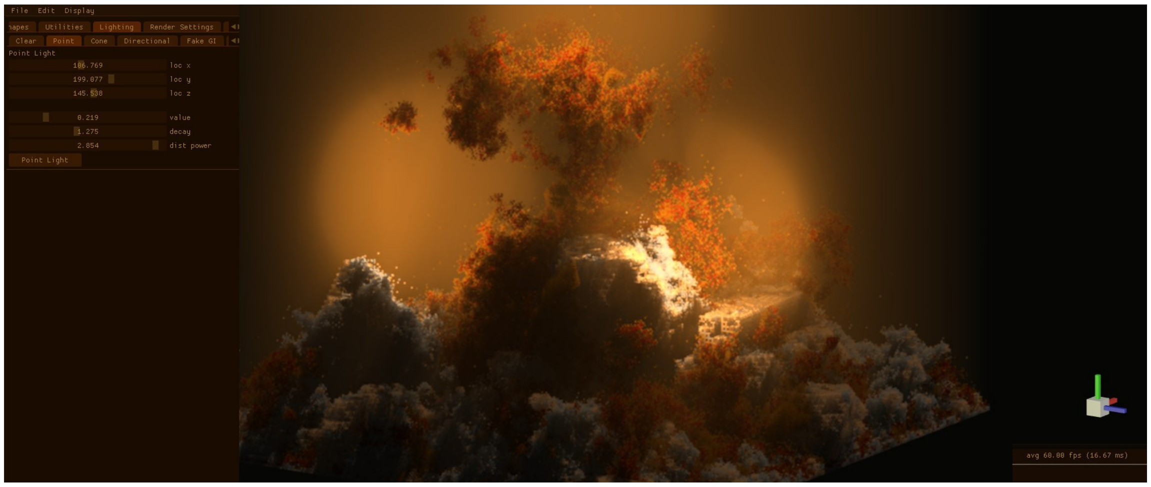
[10] Williams, Barrus, Morley, Shirley. (July, 2005). An Efficient and Robust Ray-Box Intersection Algorithm. ACM SIGGRAPH 2005 Courses. Retrieved from `https://dl.acm.org/doi/10.1145/1198555.1198748`

[11] [9] Werness, B. (April, 2017). Voxel Automata Terrain. `https://bitbucket.org/BWerness/voxel-automata-terrain/src/master/`

[12] Duff, T. & Porter, T. (July, 1984). Compositing Digital Images. *Computer Graphics*, Volume 18, Number 3, 253-259. Retrieved from `https://keithp.com/~keithp/porterduff/p253-porter.pdf`

[9] I wish to express appreciation to Brent Werness for helping me understand his Voxel Automata Terrain and fake Global Illumination code in order to convert it for my application.

IV. Appendix: Images

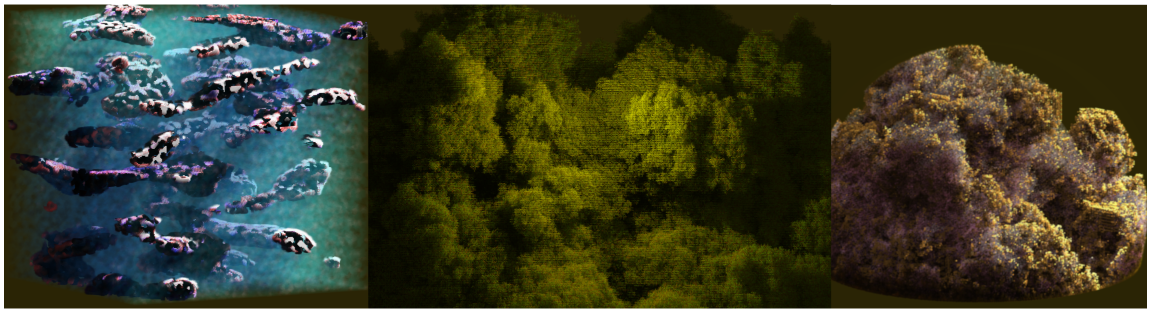


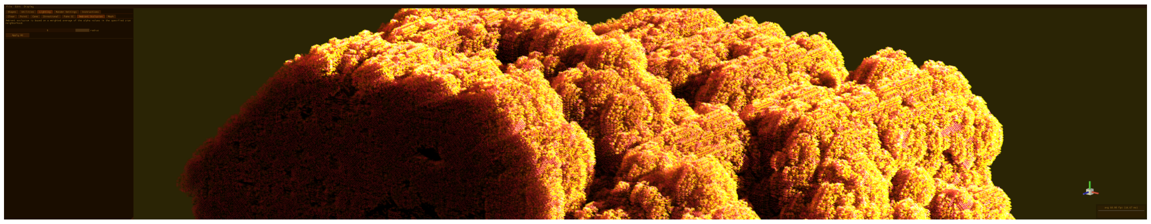
```
Point Light
106.769   loc x
199.077   loc y
145.538   loc z

0.219     value
1.275     decay
2.854     dist power
```

```
avg 60.00 fps (16.67 ms)
```

Showing the editor interface, with the orientation widget in the lower right corner.


Showing some example data made with the editor and lighting functions.


Tile based renderer showing a model at 5760x1080, 8x MSAA, with 512 voxel resolution at 60fps.