DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF ENGINEERING, LTH

Parallax Voxel Ray Marcher

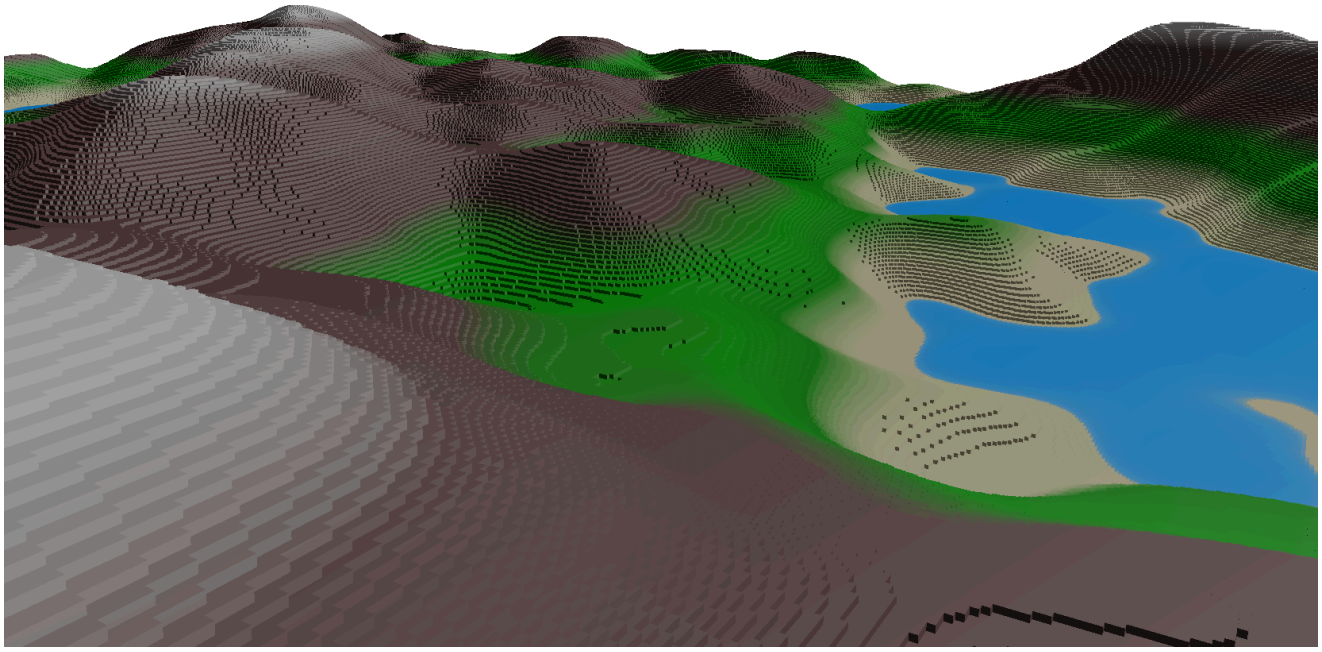
Project Report in High Performance Computer Graphics, EDAN35

Theodor Lundqvist *
Department of Computer Science
Lund University
Sweden

Jintao Yu †
Department of Design Sciences
Lund University
Sweden

Jiuming Zeng ‡
Department of Design Sciences
Lund University
Sweden

December 2023



* e-mail: theodor.lundqvist@gmail.com
† jintaoyuemail@gmail.com
‡ jiuming.zeng.0720@student.lu.se

***Abstract*—This report aims to examine the practicability and usefullness of an alternative voxel rendering method. A raytracing traversal algorithm described by Amanatides & Woo [1] was implemented together with two different acceleration structures. Different use cases like 3D cellular automata, terrain generation and editing were explored. The report briefly discusses techniques that have been used to improve visual fidelity like Blinn shading and ambient occlusion. The engine runs on OpenGL and uses some convenient classes like the FPSCamera from the Bonobo framework [2].**

I. Introduction

Totally destructible and editable worlds can be found in games like Minecraft. A common approach is to build a mesh of the terrain surface. However, during a live stream, Tuxedo Labs showed of their engine for the game Teardown [3]. The game boosts with totally destructible and physically simulated environments. A large difference being that the number of voxels per cubic meter of the game is multiplied a thousand fold compared to Minecraft, which would take the number of triangles closer to or past the number of pixels.

During the livestream the developers showed off that the voxels where not actually made up of triangles but instead raytraced through volumetric bounding boxes. They also refered to an algorithm by Amanatides & Woo, the Fast Voxel Traversal Algorithm. [1].

When investigating the approach further we found a number of small youtube channels that where doing something similar [4]. However, Teardown is the only AA game we could find to be using the technique.

II. Algorithms

Parallax Mapping

The live stream and youtube videos did not provide too much detail on the rendering except for the general concept. But somehow, they projected a 3D texture onto its bounding box. To do that, we have used a technique similar to parallax mapping. Where each surface pixel is given a color corresponding to the perspective corrected projected geometry [5]. The geometry in this case being a 3D texture of bytes where each byte represents a material. We started off by projecting the 3D texture on to a single quad by taking fixed size steps through the texture in the direction of the view vector, starting from the interpolated uv coordinate.

```
for i < max_steps:
  vec3 P = vec3(uv, 0) + t * V
  if(!insideAABB(P)) discard
  int mat = int(texture(volume, P).r * 255)
  if(mat != 0) return mat
  t += 0.1 * voxel_size
discard;
```

Listing 1: Fixed step traversal pseudo code

Later, in order to be able to move around and into the volume, we choose to instead project the geometry onto the back faces of the cube as in Figure 1. This approach was mentioned by Tuxedo Labs in the live stream [3]. The traversal must still begin from the first intersection with the volume, which we found by doing an AABB intersection test.

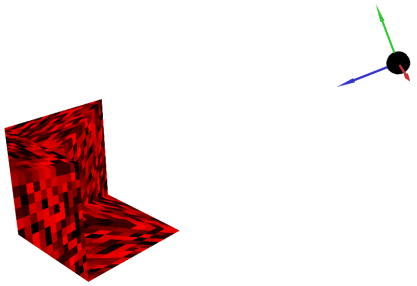


Figure 1: Perspective aware parallax projection

Grid Traversal

In order to decide which texels are to be rendered, we implemented two different algorithms in the fragment shader, fixed step traversal and Fast Voxel Traversal Algorithm, FVTA for short, by Amanatides & Woo [1].

Traversing the grid with a fixed sized step is very easy to implement. After setting the starting point, direction and step size, the program can traverse the entire texcoord space to find the point to be rendered. Stepping a fixed distance every time creates a poor approximation of the volume. This is due to some voxel corners being stepped over. It is also very slow since the step size must be small in order to remove inaccuracies. We have set the step size to 1/10 of the voxel size.

To improve the performance and accuracy of the shader, the Fast Voxel Traversal Algorithm (FVTA) was implemented. Unlike the fixed size step method, each iteration will step from one voxel to the next voxel along the view direction. To do that, the algorithm calculates the distance to the next voxel on each axis. Then it will choose the direction of the axis with the smallest distance and step in that direction. The FVTA in two dimensions is illustrated in Figure 2. In three dimesions there is just one more axis to keep track off, the basics of the algorithm are the same.

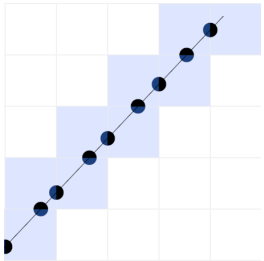


Figure 2: FVTA 2D Illustrated [6]

Figure 3 compares the results of the two methods. It can be seen that FVTA has better accuracy at voxel boundaries.

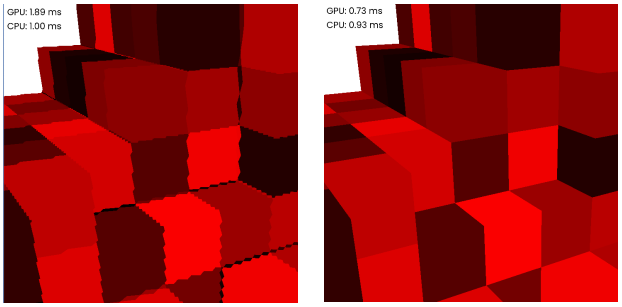


Figure 3: Fixed step traversal (left) and FVTA (right).

A performance comparison between the algorithms can be found in Section III - Performance.

Blinn Shading

For shading, we choose to use Blinn shading which is an alternative to Phong shading [7]. Blinn shading changes the calculation of highlights, using half-range vectors. This makes the effect more similar to how the human eye experiences highlights.

The results of the Blinn shading are shown in figure 4. Without shading it is very hard to see the contours of the object. With shading, the results are more realistic.

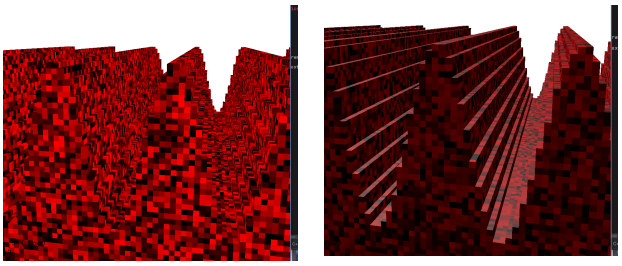


Figure 4: No shading (left) and Blinn shading with reinhard-jodie tone mapping (right).

Ambient Occlusion

In order to add some further depth to the rendered image we added a crude ambient occlusion. For every pixel that is close to the edge of a voxel face, we take one sample in each of the four single-axis directions orthogonal to the normal. The sample point also contains a small offset along the normal.

The approach is not at all optimal since it will result in two points always being sampled inside the neighbour laying in the direction of the face normal. A better approach would be possible by doing smarter sample selection or by using deferred rendering and screen space ambient occlusion.

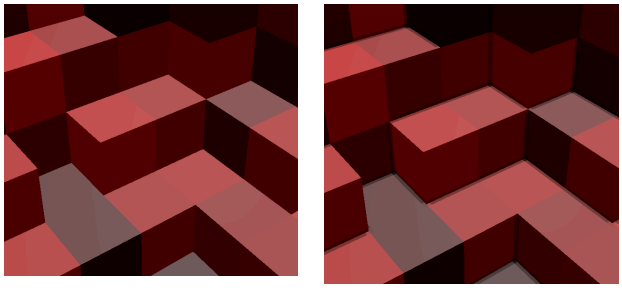


Figure 5: Standard shading (left) and hard ambient occlusion shadowing (right).

Cellular Automata

Cellular automata in three dimensions is different from Conway's game of life in 2D [8], [9], the rules for updating the cells are much more complicated. Instead of having a binary state and eight neighbours as in game of life, different rules can have different sets of its 26 neighbours being counted and the number of states is variable. The basic of a 3D CA rule can be divided into 4 parts .

{survival, spawn, state, neighbour method}

Survival indicates that alive cells will survive if they have a specific number of neighbours which can be found in the survival rule, the corresponding is true for the spawn rule and dead cells. Unlike the possible states in game of life, dead and alive, when a living cell in 3D cellular automata is determined to be dead in the next round, it will not disappear immediately, but will perform a state-1 operation every frame until state goes down to 0, then it will be removed. Moore and Von Neuman are two common neighbour patterns where Moore takes all 26 neighbours into consideration and Von Neuman only considers the 6 adjecent neighbours.

Since most of the possible rules will make cells expand rapidly or die out, we have to be very careful with the rule settings. Many interesting CA rules can be found on the Internet. The following images are some of rules that we have tested and used in the project.

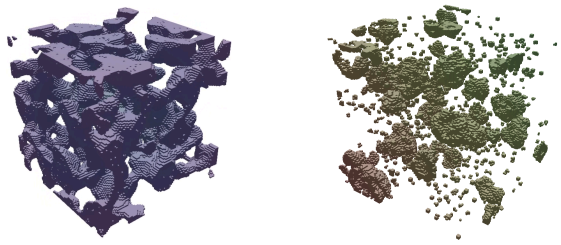


Figure 6: Two CA rulesets, Cloud (left) and Pyroclastic (right).

Since it is interesting to gain some more insight into the state of the cells, we added 4 different coloring modes: dis-

tance from center, density, state and position. Basically the cell information is converted into a color index and sent to the voxeldata along with a color palette. The color indices are used to sample the color palette on the GPU in order to store one byte per voxel instead of 3 floats.

Distance Functions

Distance functions are commonly used in ray marching to describe different mathematical shapes. In this case, we have used unsigned distance functions in order to fill the shapes with material.

By iterating over each voxel in the volume and determining whether it is within the bounds of a model defined by a distance function, we can create blocky representations of some different shapes.

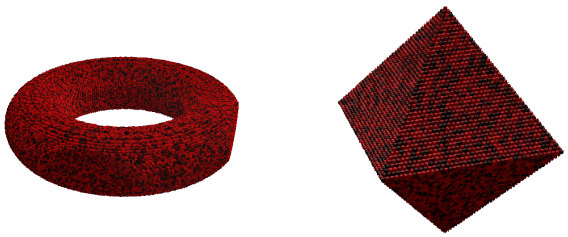


Figure 7: SDF of a torus (left) and of an octahedron (right).

Fractal Brownian Motion

To generate more realistic terrain we use Fractal Brownian Motion (FBM) which was introduced by Mandelbrot and van Ness in 1968 [10] to add more detail to the perlin noise. In difference to Brownian Motion, there now exists a correlation between the iterations. Hurst index, H for short, takes values ranging from 0.0 to 1.0, which control the raggedness of the motion, the higher the value, the smoother the path.

When $H > \frac{1}{2}$, the covariance funtion is greater than zero, which means that the changing trends of these two variables are the same, thus current and previous increments are positively correlated. When H < $\frac{1}{2}$, increments are negatively correlated since the covariance E is less than zero. For $H = \frac{1}{2}$ we have the normal Brownian Motion.

The basic idea of FBM in procedural terrain is to continuously accumulate noise with smaller and smaller amplitude but greater and greater frequency. We decided to set H to 1 not only because higher H will give a more smooth and detailed shape, but also construct a self-similarity when using G to scale the amplitude.

```
float G=exp2(-H)
float frequency = 1.0
float amplitude = 1.0
float sum = 0
for i < octaves
  sum += amplitude * noise(frequency * position)
  frequency *= 2.0
  amplitude *= G
```

(2)

Listing 2: Fractal brownian motion pseudo code

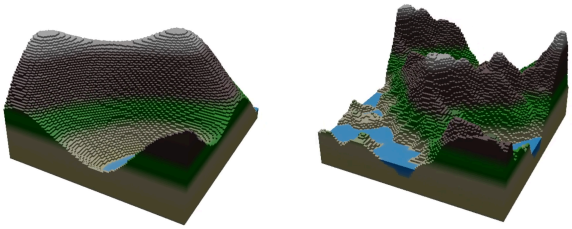


Figure 8: Without FBM (left) and with FBM (right).

Level of Detail

After we had implemented the FVTA we found that it was very easy to just let the GPU skip two, three or four blocks at a time in order to traverse volumes faster. This can be beneficial when rendering distant volumes. Note that we still upload the full resolution texture to the GPU. Objects that are further away will automatically be less taxing to render since there are fewer fragments visible. However, this approach also makes for a quicker z-axis traversal.

The level of detail is dynamically picked based on the distance to the voxel volume. The selection is aggressive in the way that we lower the render resolution very early, one should probably only lower the resolution when a voxel is the same size as a pixel in order to keep visual fidelity.

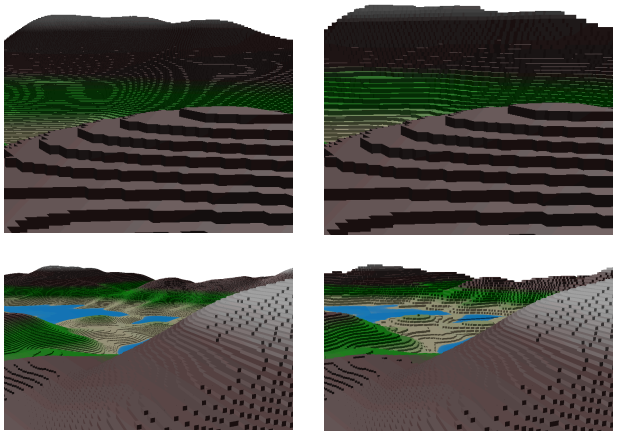


Figure 9: Comparison of distant terrain at full resolution (left) and with aggressive dynamic LOD (right).

III. Results

Performance

We have found that the render time is lower bounded by the number of memory reads. Since a trace is executed for each pixel on the volume bounding-box the render time increases linearly with the screen space area covered by the volume. The render time is also scene dependent since the number of memory reads per trace increases drastically when traversing empty space.

Scene	Size	Voxels	FVTA	Fixed
Sphere	128^3	2.1M	3 ms	10 ms
Waves	128^3	2.1M	4 ms	15 ms
Terrain	128^3	2.1M	4 ms	15 ms
Torus	128^3	2.1M	6 ms	17 ms

Table 1: Performance comparison, worst case, (M1 pro 16-core GPU @ 1000x1000).

Scene	Size	Voxels	FVTA	Fixed
Empty	128^3	2.1M	7 ms	25 ms
Empty	256^3	16.8M	19 ms	65 ms
Filled	128^3	2.1M	0.2 ms	0.2 ms
Filled	$128^3 \cdot 100$	209.7M	6.5 ms	6.5 ms

Table 2: Performance comparison, worst case, (M1 pro 16-core GPU @ 1000x1000).

It should also be noted that there is an overhead only from uploading the data to the GPU. On M1 this accounts for 6.5ms when rendering 100 volumes at 128^3 as seen in Table 3 'Filled'. When using a second 128^3 texture for storing safe step distance. The overhead increases to approximatly 10ms. This seems to be the case for other GPUs as well. There is also a problem with `glTexImage3D` taking 150+ms per frame on M1, while total CPU time on other computers we have tried are 0-1ms.

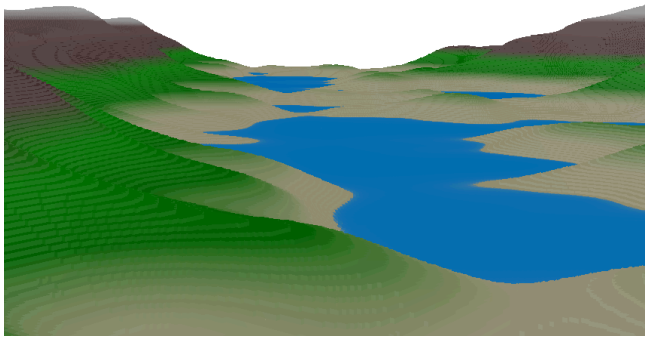


Figure 10: (Terrain*), a long stretch of low-land/valley covering the diagonal of a 10x10 volume grid resulting in a lot of empty space.

Performance measurements will refer to Figure 10 as (Terrain*) in order to visualize the worst case scenario for large terrain.

Scene	Size	Voxels	FVTA	Fixed
Terrain	256^3	16.8M	8 ms	28 ms
Terrain	$128^3 \cdot 100$	209.7M	30 ms	85 ms
Terrain*	$128^3 \cdot 100$	209.7M	46 ms	110 ms
Empty	$128^3 \cdot 100$	209.7M	53 ms	150 ms

Table 3: Performance comparison, worst case, (M1 pro 16-core GPU @ 1000x1000).

It is noticable that the number of texture reads have a large impact on the rendering performance. However, since the number of pixels does not increase, we can see a relatively small performance decline when going from 16.8M to 209.7M voxels.

Acceleration Structures

1) *Level of Detail:*

The dynamic level of detail discussed in Section II.H does provide a decent performance benefit in the worst case scenario. However, in the average case, it does not appear to have any significant impact. Note that this is with an aggressive level of detail, rendering distant volumes at a resolution low enough to sacrifice visual richness.

Scene	Level of Detail	Render time
Terrain* (worst)	Full	46 ms
Terrain* (worst)	Dynamic LOD	34 ms
Terrain (Figure 9 bottom)	Full	20-21 ms
Terrain (Figure 9 bottom)	Dynamic LOD	19-20 ms

Table 4: Performance comparison of using dynamic LOD

2) *Distance Fields:*

A simple approach to accelerate the traversal is to store one additional datapoint for each voxel, specifiying how many steps we can safely take from that position. This can lower the number of texture reads when traversing through empty space. However, the complexity of keeping the distance fields up to date when editing the volume is high. For this reason it can be beneficial to only store whether the closest neighbours are empty or not. Currently, the distance fields are calculated by iterating over each voxels neighbours. They could be generated much faster using dynamic programming or by letting distance spread out from surface voxels. The last approach was described in [11].

Renderer	Neighbours	Empty 256^3	Terrain* $128^3 \cdot 100$
Fixed Step	-	65 ms	110 ms
FVTA	-	22 ms	46 ms
FVTA + 1 step DF	8	14 ms	37 ms
FVTA + 2 step DF	124	11 ms	32 ms
FVTA + 3 step DF	342	10 ms	28 ms

Table 5: Performance comparison of storing DF in separate texture. (M1 pro 16-core GPU @ 1000x1000)

3) *Mipmaps:*

Distance fields are hard to keep up to date and require twice the space as regular FVTA. Another approach we tried were mipmaps. For the entire 3D texture we manually create a copy with half the resolution on each axis. The mipmap is conservative in the sense that a texel may only be air if there is no material inside it. This allows us to jump ahead many indexes at once without missing any material. Each time the ray hits material in a mipmap, we will go up one step in resolution until we hit material in the highest resolution grid. If the ray did not intersect any material in that part of a higher resolution mipmap, we will once again go down to a lower resolution mipmap.

This approach drastically lowers the render complexity as can be seen in Table 6. Unlike distance field acceleration which in the complexity increases exponentially for higher orders, each consecutive mipmap is easier to calculate than the last, making for a logarithmic time complexity. The same is true for the memory complexity, where mipmaps require at most 14% more memory independent of the number of levels.

Renderer	Resolution	Empty 256^3	Terrain* $128^3 \cdot 100$
Fixed Step	1	65 ms	110 ms
FVTA	1	22 ms	46 ms
FVTA + 1 mipmap	$1 + \frac{1}{8}$	12 ms	32 ms
FVTA + 2 mipmaps	$1 + \frac{1}{8} + \frac{1}{64}$	7 ms	26 ms
FVTA + 3 mipmaps	$1 + \frac{1}{8} + \frac{1}{64} + \frac{1}{512}$	5 ms	22 ms

Table 6: Performance comparison, using multiple mipmaps in separate textures. (M1 pro 16-core GPU @ 1000x1000)

Some further testing showed that removing the highest resolution mipmap (1/8) did not affect performance noticably. Resulting in a memory usage only 1.7% higher than without mipmaps. We also tried adding three even lower resolution mipmaps, essentially simulating a dense octree, this took us down to 20ms on the Terrain* scene at a very low additional cost.

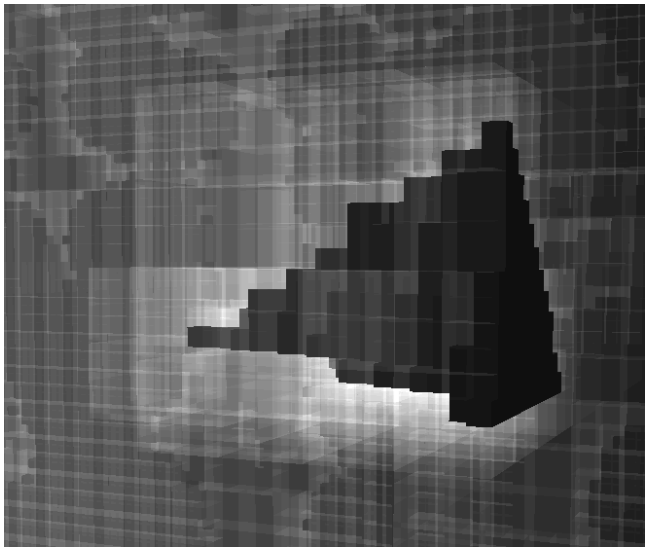


Figure 11: Illustration of the traversal complexity with 3 mipmaps, lowest resolution mipmap invisible.

IV. Discussion

Rendering

Rendering using parallax projection and grid traversal was very tricky to get right. Especially the fast voxel traversal algorithm required a lot of tinkering to get working properly. Even after looking at some different working implementations, we could not get it to work. However, perseverance was the key in this case.

Cellular Automata

After not getting cellular automata to work either we found an article explaining that cellular automata in three dimesions requires parameters not needed in two dimensions [9]. Bays, C. describes how each cell needs more that just 2 states, dead and alive. So after implementing a variable number of states, the automata worked fine. Though rules that did not just collapse were hard to find and get right.

Unpolished Algorithms

Some algorithms that we implemented were done as a spontaneous afterthought and are unpolished. The ambient occlusion could be done much more efficiently and produce a softer result. The CPU side traversal for editing is still done in a fixed-step manner on each volume it intersects. Editing therefore has some bugs and would require some more tinkering.

Performance

The acceleration structures that we built increases the GPU performance significantly but are not production ready. The structures are rebuilt every time a volume is edited, which is very slow for distance fields of high order but quite fast for mipmaps. However, it is very rewarding to go from the fixed step rendering pipeline to FVTA and fi-

nally to FVTA with acceleration structures and seeing that the result is rendered in realtime.

At first we had a problem with FBM heightmap generation being very slow and therefore built a threaded terraingenerator that could run in the background. However, when turning debug mode off, generation was understandably way faster.

V. Further Work

1) *Octrees:* It would be very interesting to implement a sparse voxel octree instead of the full resolution texture in order to further increase performance but also drastically decrease memory usage. However, this was not possible in the provided timeframe.

2) *Deferred rendering:* It would be possible to switch from a forward renderer to a deferred rendering pipeline in order to draw shadows and do some smoothing of the normals to make the voxels smoother at the edges.

3) *Regular Mesh:* Another interesting task would be to compare the performance of the pipeline with a regular triangulated mesh.

4) *Overhead:* The 6 ms overhead (with $128^3 \cdot 100$ voxels) from data transfer may be lowered from using sparse octrees, but it would also be a good idea to implement frustum culling. With frustum culling, volumes that are not to be rendered would not contribute to memory bandwidth. The limitation on PCI-E x16 of 32GB/s (gen 4) and 64GB/s (gen 5) may not be enough if we upload the textures every frame. Consider the following example.

$$\begin{aligned} \frac{128^3 \cdot 100}{1e9} &= 0.2 \text{ GB} \\ \frac{32 \text{ GB/s}}{0.2 \text{ GB}} &= 160 \text{ fps} \end{aligned} \tag{3}$$

With one byte per voxel, we put a theoretical limit at 160 frames per second, which is exactly eqvivalent to the 6 ms overhead we are seeing. We can see how this would go with even larger worlds or more bytes per voxel. However, we are not sure that there isn't any GPU side caching going on. All we really know is that the overhead increases with total data bandwidth. In any case, only sending updated chunks to the GPU would be a good approach as long as they fit in memory.

VI. Contributions

Feature	**Originator**
FVTA, Shading	Zeng, Lundqvist
CA, SDFs, Noise	Jintao Yu
Pipeline, Scenes, Parallax, Acceleration structures, Performance measures, Ambient Occlusion, Level of Detail,	Lundqvist

References

[1] J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing", *Proceedings of EuroGraphics*, vol. 87, p., 1987.

[2] Michael Dogget, "CG_Labs". [Online]. Available: https://github.com/LUGGPublic/CG_Labs

[3] Tuxedo Labs, "Teardown Technical Teardown - Twitch vod 26/11". [Online]. Available: https://youtu.be/0VzE8ROwC58?si=gHZ6JLK0d4aEeGfO&t=602

[4] Douglas, "Drawing MILLIONS of voxels on an integrated GPU with parallax ray marching". [Online]. Available: https://www.youtube.com/watch?v=h81I8hR56vQ

[5] T. Kaneko *et al.*, "Detailed shape representation with parallax mapping", *In Proceedings of the ICAT*, vol. 2001, p., 2001.

[6] Leonard Ritter, "A Fast Voxel Traversal Algorithm". [Online]. Available: https://www.shadertoy.com/view/XddcWn

[7] J. F. Blinn, "Models of light reflection for computer synthesized pictures", *SIGGRAPH Comput. Graph*, vol. 11, pp. 192–198, 1977, [Online]. Available: https://doi.org/10.1145/965141.563893

[8] M. Gardner, " MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life" ", *Scientific American*, no. 224, pp. 120–123, 1970, Accessed: Dec. 20, 2023. [Online]. Available: http://www.jstor.org/stable/2027184

[9] C. Bays, "Candidates for the Game of Life in Three Dimensions", *Complex Syst.*, vol. 1, 1987, [Online]. Available: https://api.semanticscholar.org/CorpusID:44960664

[10] B. B. Mandelbrot and J. W. V. Ness, "Fractional Brownian Motions, Fractional Noises and Applications", *SIAM Review*, vol. 10, no. 4, pp. 422–437, 1968, Accessed: Dec. 20, 2023. [Online]. Available: http://www.jstor.org/stable/2027184

[11] Douglas, "GPU-generated DISTANCE FIELDS". [Online]. Available: https://www.youtube.com/watch?v=REKcTBgkrsE