Introduction and structure

This project will be focused on implementing and comparing various voxel acceleration structures for ray marching voxel scenes. Voxel ray marching is the process of determining the onscreen pixel colour based on traversing a ray through a 3D grid. This traversal can be accelerated through the use of these acceleration structures, where the distance stepped along the ray can be non uniform depending on the geometry of the scene, decreasing the number of intersection tests required by the ray, and possibly reducing the memory usage of the scene data.

For the ray-marching one possible approach is to implement a wavefront[3] based approach. This approach involves separating the individual stages of the ray-casting into individual shaders, instead of all stages occurring in the same 'mega kernel'. This should result in performance improvements at the cost of some additional memory requirements for the additional buffers.

The simplest structure is a 3D grid, in which DDA (digital differential analyser)[1] can be used to traverse the grid. DDA can be expanded to traverse non uniform space, which leads to octrees[2] and contrees. These are tree structures where each node has either 8 or 64 children respectively, and the leaves represent the individual voxels. These two methods can be combined to form brick maps[4], where each node of the tree is actual a 'brick' of voxels, an $n * n * n$ collection. This combines some of the benefits of the two styles.

This project aims to compare each of the methods against each other. This comparison can be done both quantitatively and qualitatively. The methods can be compared quantitatively by measuring the memory consumption, render times, GPU register pressure and intersection count of each method. Qualitatively the methods can be measured based on the how 'easy' the structures are to modify, generate and serialize. 'easy' being subjective for how much additional work is required.

While comparison of these structures has been done previously, the comparison of these structures with regards to split rendering has not. Another aim is to compare these structures in areas that are not directly quantitative, in particular with what may be required to modify and regenerate the structures during runtime.

The results of this project should help inform others on which acceleration structure may be suitable for their own project, this may come from the direct metrics obtained from the acceleration structures or from the qualitative analysis of the methods. The application should also provide an interactive way to observe these comparisons directly by allowing a user to change between differing scenes and structures and directly see the performance results on their own machines.

The project will be written in C++, with Vulkan as the chosen rendering API, less for its performance and more for its flexibility. Slang will be used as the shader language as this can allow for hot reloading of shaders as the compiler can be implemented with the rest of the project.

A summary of each structures is below

- 3D grids
  Each voxel is individual within a 3D grid. DDA can be used to trace through the grid and check for collisions. This can be represented directly through a buffer on the GPU. Indexing of the buffer can be done by either flattening the grid index or using a Z-order curve to improve spatial location of the indexing.
- 3D textures
  Uses the GPU architecture to help with memory lookups. Mipmaps can be used to implement

level of detail and occupancy checks.

- Octrees
  A tree where each node has 8 children. In particular sparse octrees will be used allowing the tree to have a varying depth. The voxels themselves are represented by the leaf nodes.
- Contrees
  Similar to octrees except each node of the tree has 64 children instead of 8.
- Brick map
  A mix between octrees and 3D grids. Similar to the octree in being a tree structure, except each leaf node is composed of a 'brick', an $n * n * n$ collection of voxels.

Core project

The core element of the project involves implementing the ray marcher for each structure, and implementing a system that should allow for easy swapping between them. This system should also allow for different scenes to be loaded, so that each structure can be compared in various layouts. These scenes can either be generated programmatically, or alternatively by generating voxels based on a mesh. This skin of a mesh can be generated by checking if any triangle of a mesh intersects a grid of cubes.

Along with the rendering of these structures another aim is to allow modification to them. The modification of the structures should allow for addition, replacement and removal of voxels, while also allowing for multiple voxels to be affected as apposed to a single voxel with each operation.

The final goal of the core project is to evaluate the structures against each other based on the render time, memory usage and register pressure on the GPU, along with evaluating the structures based on qualitative features.

Extensions

The extensions of this project aim to add additional functionality, with the main extension being split rendering, which is the process of rendering the scene across multiple compute nodes. In particular this may involve partial rendering or pre-processing of the scene on one node, and then completing the rendering and displaying of the scene on a separate client-node. This reduces the computation power required on the client-node as some of the computationally intensive work can be moved to a node with more computation.

Another extension could be generation of larger then memory scenes, requiring streaming of data from disk to the GPU, while at the same time offloading data from the GPU to disk in order to reserve memory as needed.

Starting Point

Apart from the relevant courses (Introduction to graphics, Further Graphics, C and C++), I have prior experience using C++ with OpenGL, Vulkan, and GLSL with regards to implementing various graphics techniques and projects.

Success Criteria

The core project will be deemed a success if:

- Shaders are reloaded during runtime after the source file is modified
- The 5 acceleration structures, (3D grid, 3D texture, Octree, Contrees, Brickmaps) are implemented
  - Structures can be used for ray-marching
  - Structures can be modified
  - Structures can be created during runtime from files
- The used acceleration structure can be changed during runtime
- Metrics of the structures can be measured
  - Memory consumption
  - Render time
  - GPU register pressure
  - Intersection count
- Voxel scenes can be generated from meshes via a separate script
- Scenes can be loaded and changed during runtime

Work Plan

Table 1:

Sprint	Dates	Plan	Milestones
1	Oct 13 – Oct 27	- Project setup - Basic Vulkan rendering to the screen via Compute shaders - Shader hot reloading - ImGui setup - Camera movement - Event system	- Able to render a sphere through compute shaders - Reposition around the sphere via the camera - Modify properties of the sphere via ImGui - Modify the sphere from the shader without closing the window
2	Oct 27 – Nov 10	- Start implementing ray marching structures - 3D grid structure - Octree structure	- Render a scene with the 3D grid and octree - Switch between structures within ImGui - Generation of structures
3	Nov 10 – Nov 24	- Continue with implementing basic methods - Brickmap support - Contree support	- Render a scene with Brickmaps and contrees

Continued on next page

Table 1: (Continued)

4	Nov 24 – Dec 08	- Additional structures - 3D textures - Modification for flat grid	- Render a scene with 3D textures
5	Dec 08 – Dec 22	- Modification for octree / contree - Modification for brickmap	- Octree and brickmap can be modified - Cubes and spheres of voxels can be added and removed - Voxels in various colours can be added
6	Dec 22 – Jan 05	- Mesh to Voxels - Slack time	- Be able to convert meshes to voxels - Load files for scenes - Change scene during runtime - All previous milestones met
7	Jan 05 – Jan 19	- Start split rendering - Split software into client and server - Establish connection between client and server	- Be able to load the application as either a server or client - Connect the client to the server - Data can be shared between server and client
8	Jan 19 – Feb 02	- Implement 3D grid for split rendering	- Data for 3D grid can be shared from server to client - Data can be requested from server
9	Feb 02 – Feb 16	- Implement structures across devices - Implement Octree/Contree for split rendering	- Octree can be shared between server and client - Nodes of the octree can be loaded as needed from server
10	Feb 16 – Mar 02	- Octree addition - Implement loading for tree nodes	- Octree can be modified on client and updated server side
11	Mar 02 – Mar 16	- Octree across devices - Allow for nodes to be individually loaded between devices	- Only visible children are loaded on the client - Newly visible children requested from server
12	Mar 16 – Mar 30	- Refactor / extra time	- Project should be completed
13 →	Mar 30 – May 15	- Dissertation	

By the end of Michaelmas the core project should be finished to a sufficient degree. This leaves Lent to work on the more complicated extensions. In particular lent will be focused on the split rendering, with an interest in implementing the octree/contree as this structure, along with selective inclusion of the tree nodes that are out of level of detail range, and those that are obscured would

require more communication between the client and server, and so a more interesting challenge.

There are also some additional tasks, that if time allows can be included. Notably this may include better lighting calculations. These are not planned for as the time requirement for split rendering is not easy to judge. This is why the last sprint is left open, either for additional tasks, or for continuing work on the previous tasks.

Resource Declaration

I will be using my personal laptop for most of the project. This has the specs of AMD ryzen 7 5800H with IGPU, 16GB Ram, 1TB NVME SSD.

I also have access to a desktop PC with the following specs: Intel I7-11700KF, Nvidia RTX 3070, 32GB ram, 1TB SSD, 4TB HDD.

I will be using git for version control, along with github for backups.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

Bibliography

[1] John Amanatides, Andrew Woo, et al. “A fast voxel traversal algorithm for ray tracing.” In: 1987.

[2] Samuli Laine and Tero Karras. “Efficient sparse voxel octrees”. In: 2010. URL: https://doi.org/10.1145/1730804.1730814.

[3] Samuli Laine, Tero Karras, and Timo Aila. “Megakernels considered harmful: wavefront path tracing on GPUs”. In: 2013. URL: https://doi.org/10.1145/2492045.2492060.

[4] TL van Wingerden. “Real-time ray tracing and editing of large voxel scenes”. MA thesis. 2015.