Introduction and structure

This project will be focused on implementing and comparing various acceleration structures that are targeted towards voxel ray marching. These structures are designed to increase the efficiency at which a ray can be traversed through space in order to check for collisions with voxels.

For the ray-marching one possible approach is to implement a wavefront[1] 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)[2] can be used to traverse the grid. DDA can be expanded to traverse non uniform space, which leads to octrees[3] 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 these methods to each other, comparing particularly for the rendering efficiency in the methods. This can be expanded to compare the memory efficiency, ease of generation, and ease of modification. The modifications can be compared through animated scenes. With specific key-frames the entire scene can be changed every couple frames to stress test the modification of the scene.

The goal is to hit performance suitable for real-time rendering, from each of the methods. However, this may not be suitable for some of the structures for all scenes. Performance can be additionally compared based on the theoretical intersection count and the actual observed count. This can be pushed further by looking at the register pressure on the GPU for the shaders that are used. This requires external tooling, either Nsight graphics for Nvidia cards or AMD's developer tool suite for AMD graphics.

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 few external libraries will be required, these are used to reduce the boilerplate required and to help with the setup of Vulkan, along with some additional tooling. These libraries are:

- GLFW: A cross platform windowing library that can also handle inputs.
- GLM: A maths library targeted towards OpenGL but also functional with Vulkan.
- (Dear) ImGui: Immediate mode GUI, simple to use, and flexible UI library.
- slang: Compiler for the slang shader language.
- vk-bootstrap: Simplifies initialization of Vulkan.
- VulkanMemoryAllocator: Helps with memory allocation within Vulkan.

[1] Laine, Karras, and Aila, "Megakernels considered harmful: wavefront path tracing on GPUs".
[2] Amanatides, Woo, et al., "A fast voxel traversal algorithm for ray tracing."
[3] Laine and Karras, "Efficient sparse voxel octrees".
[4] Wingerden, "Real-time ray tracing and editing of large voxel scenes".

- Tracy: A frame profiler

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, while still hitting the real time rendering constraints. The modification of the structures should allow for addition, modification and removal of voxels, while also allowing for multiple voxels to be affected as apposed to a single voxel with each operation.

The aim is the compare the structures for the efficiency in rendering, memory, and register pressure on the GPU. The render time and memory usage can be measured directly through Vulkan, while external tools can measure the register pressure of a shader by inspecting the generated SPIR-V shader code.

Extensions

The extensions of this project aim to add additional functionality. One possible extension is larger then memory scenes. This would put focus on ensuring only visible voxels are possibly rendered, and those voxels that are not visible are left out of GPU memory. This should also lead to scenes that are near infinite in size to be used as those areas that are not visible can be offloaded, and those far from the view are put through a level of detail.

Another extension is split rendering 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 finishing the rendering and display 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.

Starting Point

Apart from the relevant courses (Introduction to graphics, Further Graphics, AGIP), previous work has been done in graphics and shader programming. In particular work with Vulkan and GLSL has been done before. Slang will need to be learnt, however it is designed to be similar to HLSL which somewhat follows GLSL.

Success Criteria

The project will be deemed a success if:

- It is possible to change the acceleration structure with minimal changes
- The performance of an acceleration structure can be measured
- The performance of the acceleration structures follow what is expected
- The implementations are performant to enable real-time rendering, where it applies
- The scenes can be modified, either by a user or programmatically (animations)
- Scenes can be loaded to test the structures in various ways

Work Plan

Sprint	Dates	Plan
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
2	Oct 27 – Nov 10	Start implementing ray marching structures Flat grid structure Octree structure
3	Nov 10 – Nov 24	Continue with implementing basic methods Brickmap support Contree support
4	Nov 24 – Dec 08	Additional structures 3D textures Modification for flat grid
5	Dec 08 – Dec 22	Modification Modification for octree / contree Modification for brickmap
6	Dec 22 – Jan 05	Break
7	Jan 05 – Jan 19	Start split rendering Split software into client and server Establish connection between client and server
8	Jan 19 – Feb 02	Data Implement 3D grid for split rendering
9	Feb 02 – Feb 16	Implement structure across devices Implement Octree/Contree for split rendering
10	Feb 16 – Mar 02	Octree addition Implement loading for tree nodes
11	Mar 02 – Mar 16	Octree across devices Allow for nodes to be individually loaded between devices
12	Mar 16 – Mar 30	Refactor / extra time

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, and serialization and loading of the voxel data. 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.

Github projects will be used to split these tasks into further subcomponents and to organise those tasks within the sprints. This will also allow for bug tracking and any additional features that

may be needed to be included and planned for.

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.