GPGPU Acceleration of Memworld, a Raycasting Engine for Direct-Memory Representation of 3D Spaces

Final Report

Team 18
Client: Dr. Wymore
Adviser: Dr. Wymore

William Blanchard	Parallelization Lead
Mason DeClercq	Team Lead
Jay Edwards	Documentation Lead
Cristofer Medina Lopez	Integration Lead
Dalton Rederick	Communications Lead
Collin Reeves	Game Development Lead

sddec22-18@iastate.edu
https://sddec22-18.sd.ece.iastate.edu

Revised: 12/7/2022 : Version 2

Executive Summary

Development Standards & Practices Used

730-2014 - IEEE Standard for Software Quality Assurance Processes | IEEE Standard | IEEE Xplore

- We will use the IEEE standard for software quality assurance processes because we are making a software product that needs testing . This standard talks about how we should make a software quality assurance process in order to confirm that our product meets the established requirements that were given.

1219-1998 - IEEE Standard for Software Maintenance | IEEE Standard | IEEE Xplore

- We will adhere to this standard in how we will handle maintenance of Memworld. This standard shows the groundwork for testing and adhering to client given requirements for software that is undergoing development. The standard goes into detail about having a control group of requirements that needs to be maintained as development on the project continues.

24748-5-2017 - ISO/IEC/IEEE International Standard - Systems and Software Engineering--Life Cycle Management--Part 5: Software Development Planning | IEEE Standard | IEEE Xplore

- This standard talks about systems and software engineering with life cycle management. We would be focusing on the software development planning section of this standard. This talks about using peer reviews and testing during development as well as coding standards in order to have a clear understanding that our project is working and how it works.

Summary of Requirements

Requirements/specifications:

- Must use "direct-memory" representation and rendering
- Must parallelize rendering algorithm using GPGPU
- Preferably use a portable GPGPU framework, both in terms of hardware and OS
- Must characterize performance in terms of FPS and determine speedup from GPGPU parallelization
- Performance target is 30 FPS or higher with the following settings:
    - 1024 x 768 resolution or higher
    - Voxel density of 5 or higher
    - Max draw distance of 100 voxels or higher

Test application requirements:

- Must use parallelized engine
- Should be portable
- Should highlight advantages/strengths of engine
- Visually pleasing UI
- May incorporate major new features such as physics, destructible environment, lighting, etc.

Table of Contents

List of figures/tables/symbols/definitions

Figures:

Tables:

Definitions:

GPGPU - General-purpose computing on graphics processing units, uses the GPU to perform computation normally done by the CPU.

Voxel - A basic unit of a three-dimensional digital representation of an image or object.

Parallelization - A program or system that processes data in parallel threads.

Octree - A tree data structure where each internal node has eight children.

1 Introduction

1.1 Problem Statement

The goal of our Senior design project is to update and improve the direct memory rendering model created by Dr. Wymore, including features by his request or to our interest. These features include, but are not limited to, increased resolution, improved frame rate or run speed, and inclusion of miscellaneous object interactions such as physics and collision. At the end of the project, the team will create a simple game to show off these improvements and additional features.

1.2 Requirements & Constraints

Final deliverables are:

1. GPGPU-parallelized Memworld engine
2. Memworld test/demonstration application

Requirements/specifications:

1. Must use "direct-memory" representation and rendering (**constraint**)
2. Must parallelize rendering algorithm using GPGPU (**constraint**)
3. Preferably use a portable GPGPU framework, both in terms of hardware and OS (**constraint**)
4. Must characterize performance in terms of FPS and determine speedup from GPGPU parallelization (**constraint**)
5. Performance target is 30 FPS or higher with the following settings:
6. 1024 x 768 resolution or higher (**constraint**)
7. Voxel density of 5 or higher (**constraint**)
8. Max draw distance of 100 voxels or higher (**constraint**)

Test application requirements:

1. Must use parallelized engine (**constraint**)
2. Should be portable (**constraint**)
3. Should highlight advantages/strengths of engine
4. Visually pleasing UI
5. May incorporate major new features such as physics, destructible environment, lighting, etc.

1.3 Engineering Standards

730-2014 - IEEE Standard for Software Quality Assurance Processes | IEEE Standard | IEEE Xplore

- We will use the IEEE standard for software quality assurance processes because we are making a software product that needs testing . This standard talks about how we should make a software quality assurance process in order to confirm that our product meets the established requirements that were given.

1219-1998 - IEEE Standard for Software Maintenance | IEEE Standard | IEEE Xplore

- We will adhere to this standard in how we will handle maintenance of Memworld. This standard shows the groundwork for testing and adhering to client given requirements for

software that is undergoing development. The standard goes into detail about having a control group of requirements that needs to be maintained as development on the project continues.

24748-5-2017 - ISO/IEC/IEEE International Standard - Systems and Software Engineering--Life Cycle Management--Part 5: Software Development Planning | IEEE Standard | IEEE Xplore

- This standard talks about systems and software engineering with life cycle management. We would be focusing on the software development planning section of this standard. This talks about using peer reviews and testing during development as well as coding standards in order to have a clear understanding that our project is working and how it works.

1.4 Intended Users and Uses

The main beneficiaries for this project are Game Developers, specifically those that fit the niche of wanting a voxel-based game engine with a focus on performance. They will use it as a game developer would use any other game engine, as a tool to develop games. The difference with Memworld is that it will give a direct memory representation of what it is rendering.

- Use Case 1: Render an object/world
- Use Case 2: Physics simulation on voxels
- Use Case 3: Create a visual representation of current memory storage

2 Project Plan

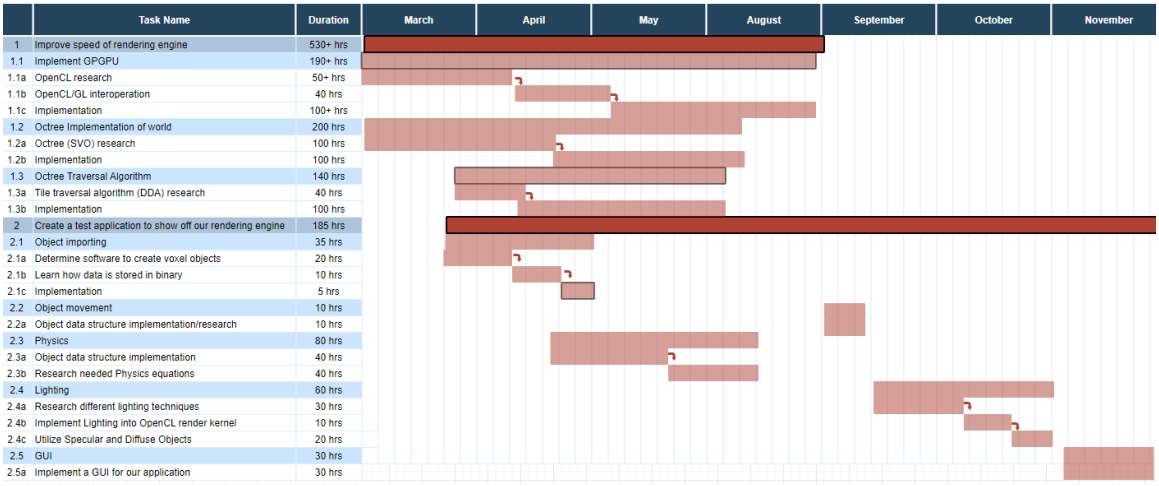
2.1 Project Proposed Milestones, Metrics, and Evaluation Criteria

Milestones:

- Performance:
  - 30 FPS or higher with the following settings
    - Voxel density of 5 or higher
    - 1024 x 768 resolution or higher
    - Render distance of 100 voxels or higher
- Physics (all require under 2 fps slowdown from current implementation):
  - Objects can fall
  - Objects can collide and stop movement
- Simple rigid body physics
  - Lighting
  - Ray trace from object to light source (1 bounce from object)
- Object importing:
  - Can import a world of 256 x 256 x 256
  - Can import individual objects up to 64 different objects
- Portability:
  - 2 platforms (Windows and Mac)

2.2 Project Timeline/Schedule

Project Timeline



	Task Name	Duration
1	Improve speed of rendering engine	530+ hrs
1.1	Implement GPGPU	190+ hrs
1.1a	OpenCL research	50+ hrs
1.1b	OpenCL/GL interoperation	40 hrs
1.1c	Implementation	100+ hrs
1.2	Octree Implementation of world	200 hrs
1.2a	Octree (SVO) research	100 hrs
1.2b	Implementation	100 hrs
1.3	Octree Traversal Algorithm	140 hrs
1.3a	Tile traversal algorithm (DDA) research	40 hrs
1.3b	Implementation	100 hrs
2	Create a test application to show off our rendering engine	185 hrs
2.1	Object importing	35 hrs
2.1a	Determine software to create voxel objects	20 hrs
2.1b	Learn how data is stored in binary	10 hrs
2.1c	Implementation	5 hrs
2.2	Object movement	10 hrs
2.2a	Object data structure implementation/research	10 hrs
2.3	Physics	80 hrs
2.3a	Object data structure implementation	40 hrs
2.3b	Research needed Physics equations	40 hrs
2.4	Lighting	60 hrs
2.4a	Research different lighting techniques	30 hrs
2.4b	Implement Lighting into OpenCL render kernel	10 hrs
2.4c	Utilize Specular and Diffuse Objects	20 hrs
2.5	GUI	30 hrs
2.5a	Implement a GUI for our application	30 hrs

2.3 Other Resource Requirements

- Research Time
- Internet access for research
    - Google Scholar
- Access to a computer
    - Windows
    - Mac
    - Linux (if time permits)
- Required programs
    - MinGW
    - GLFW
    - an IDE (member's choice)
    - Cmake
    - Vulcan
    - MagicaVoxel

3 Security

3.1 Cyber Security Concerns

The main cyber security concern that our project contains is that we are reading from a file that users are able to modify. Ideally, we would have a fixed set of inputs that the user can't deviate from, but due to time constraints, we use a plain text file for settings. We must ensure that we are reading the files in a secure manner and that buffer overflows are not possible when using input files. We also must make sure that the inputs are in the range of normal operation to make sure that our program doesn't crash unexpectedly.

The next security concern that our project contains relates to reading in MagicaVoxel files. The application only allocates enough memory for 512 x 512 x 512 voxels. This is plenty for most use cases, but if by chance a user forgets about this limitation or an attacker would want to go past the bounds that have been set, they could possibly modify other pieces of memory on the heap.

3.2 Cyber Security Countermeasures

In order to avoid the concerns that we have listed above, we have included checks as to what values of the input file are being read. For example, if we are reading in the value for window height, the only acceptable values are in the range of 768 and 1080. If the values aren't in the range, the application exits and gives an error message as to why it exited. We are also only reading in numbers, so even if the user enters anything besides a number, the characters will be converted to numbers. If the numbers end up overflowing, that is fine because we check that the numbers are in the correct range for the program to run correctly.

In order to fix the concern relating to the voxel memory, we check to make sure that the files that we are reading in will fit in memory.

4 Design

4.1 Prior Work

4.1.1 Prior Work/Solutions

This is a paper from NVIDIA demonstrating how they implemented a sparse voxel octree. This is more complicated than we need ours to be. It is an efficient way to traverse a world to find voxels. We also are restricted by not being able to use a sparse version of the world (i.e. we have to represent empty space as a zero in memory).

> Laine, Samuli, and Tero Karras. "Efficient sparse voxel octrees–analysis, extensions, and implementation." *NVIDIA Corporation* 2.6 (2010).

We found a thesis paper done on a similar idea. While they are able to get high fidelity static models working with their design, real time animation appears to be difficult to accomplish with their work. This means that for our purposes, game development, it is likely not a feasible strategy to follow.

> Crassin, C., Neyret, F., & Sillion François X. (n.d.). *Gigavoxels: Un pipeline de Rendu Basé Voxel pour l'exploration efficace de scènes larges et détaillées* (thesis).

Here is a 3D voxel engine,

This is Ken Silverman's personal project which is a similar 3D voxel engine. He explains part of his process on the website and in the readme for the project, mainly just having tips for certain parts of the implementation. The issues with this source is that the notes and codes are not outsider friendly. The notes read like personal memos and the code is hard to understand for someone other than the dev.

> Silverman, K. (2018). PND3D demo and source code. Retrieved 2022, from http://advsys.net/ken/voxlap/pnd3d.htm

This is a tutorial that talks about modern rendering techniques that are used for triangle based meshes. It goes over many different optimization techniques, different types of shading, and the math behind all of it. Even though our project doesn't use triangle based meshes, some of the concepts still apply.

> "Computer Graphics for the 'Rest of Us'." *Scratchapixel*, 31 Aug. 2022, https://www.scratchapixel.com/index.php?redirect.

4.1.2 Comparison to Prior Work/Solutions

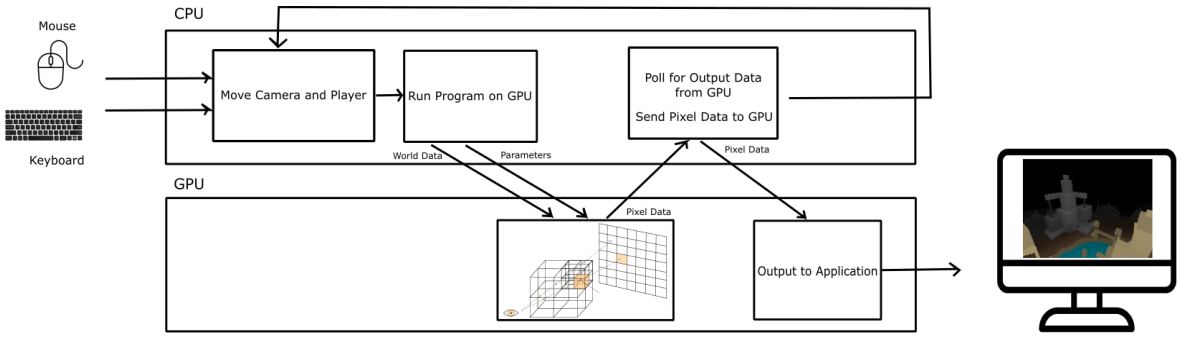
Raycasting is an existing concept. The purpose of this project is to take the problem of raycasting and optimize it using the physical resources available to the computer. In this way, the scope of our problem, while challenging, does aim to improve upon currently existing standards if it doesn't at least try to find a different way of implementing them.

When looking at the other voxel-based rendering implementations, ours is different because of our direct memory representation of the world. Octrees are widely used across these voxel-based implementations, but octrees are useful for representing a sparse world allowing for many layers in the octree. Having a direct memory representation means that we store the zeros in memory. This gives us the ability to add and remove voxels faster, at the cost of more memory, and the voxel octree that we implemented added even more memory to store the meta-data.

In terms of the tutorial about modern rendering techniques, it helped us in determining how we could improve our voxel based rendering based on previous techniques and what would be possible during our time in this class. Primarily the use of bounding boxes and chunk based rendering was inspired by this tutorial. The solutions to these problems are still quite different because the direct memory representation of the voxels in our model aren't being rendered the same way that triangles would be rendered.

4.2 Design Evolution

System Interaction Diagram



4.2.1 Renderer

491 Design:

Since we started 491, the renderer has changed a lot. At the end of 491, the rendering algorithm was based on using a voxel octree. This was efficient because we could traverse large empty distances quickly as it cut down on the amount of cache misses that we were seeing. There were still problems with objects having jagged edges depending on how high the voxel density was. If the object was only one voxel thick, there would also be flickering in the render. This was due to the nature of how we were skipping along the ray in order to get to a voxel. Our client wanted us to use a direct memory representation of the world, so bigger worlds required large amounts of memory. The extra metadata that was required to support the octree didn't help either.

492 Design:

Since we started 492, we started using the 3DDDA (3D Digital Differential Analyzer) algorithm. This removed the need for voxel density as we were able to skip along the ray exactly the right amount until a new voxel was hit instead of using an approximation. We were able to convince the

client to change from a direct memory representation of the world to a direct memory representation of the objects. This allowed us to reduce the amount of memory that we were using, but would also allow us to change how the renderer worked. So, instead of using the octree that we had implemented, we started using bounding boxes that surrounded the objects. The renderer would then check for ray to bounding box intersections and then traverse the objects at the intersection. This became expensive if we were to check bounding box intersections of every object. We then implemented a way to separate the world into chunks. Each chunk would contain the number of objects and an array of object indexes that it contains. The renderer would traverse the chunks along the ray, checking the objects that are contained within the chunks. Since the renderer now works by checking ray to bounding box intersections, we can move objects by changing the position of the bounding box, and the voxels will follow.

4.2.2 Physics

491 Design:

During 491, our design for physics was primarily voxel-based physics where we would move individual voxels and have voxels collide with each other. When moving large objects, the cost of moving all the voxels becomes quite expensive. The need to check from collisions on all of the voxels adds more complexity to it as well. The player would only collide 1 voxel at a time where the camera was located.

492 Design:

Since we started 492, the change of the rendering algorithm led to an increase in performance of the physics implementation because of the addition of bounding boxes. The addition of bounding boxes allowed us to check bounding box collisions, which is cheaper, and if there is a bounding box collision, then, we check for voxel based collisions. The renderer uses the bounding boxes to display the objects. This allows us to move the bounding box which "moves" the voxels around with the bounding box without having to move them around in memory. The camera now has a bounding box associated with it to give the player some height. The camera has another smaller bounding box as well that determines if the player can jump over small changes in voxel height. We can tell if the player can jump over small voxel objects if there is a bounding box collision with the larger bounding box and no bounding box collision with the smaller bounding box. There is also a way to create moving platforms for our 3D-platform where you can set points in the world and the platform will interpolate between the points by the amount of steps specified.

4.2.3 Memworld

491 Design:

During 491, objects contained in memworld weren't fully distinguishable and were merged with the rest of the world. There was also only one world that we could create to move around in, and there was no way to change worlds.

492 Design:

Since we started 492, the objects are now represented by their bounding boxes. In order to not have to check each bounding box, we separated the world into chunks as was described in the renderer section. When importing an object into the world, the object would be placed in the chunks that contain it. These chunks get updated every time the object moves. This isn't too expensive as objects on average are only contained by one chunk. There is now a starting room in our game with 4 portals. Each of these portals lead to a different world. The world is able to change by writing over the previous voxels and updating the indexes of each object to know where the voxels are located. Since there aren't many objects in each world, world loading is not noticeable by the player.

4.2.4 OpenCL Implementation

491 Design:

In 491, the voxels were represented in the 3D world space. For example, if a voxel was located at (400, 500, 100), the voxel would be in that place in memory. Changing voxel objects from the CPU was done by updating a large chunk of the world. For example, if the world was 1024 by 1024 by 1024 and you had a voxel object that was 50 by 50 by 50 voxels, you would have to enqueue a write to the GPU of 50 by 1024 by 1024 which is a lot more memory than could have possibly been changed.

492 Design:

Most of the design for the OpenCL implementation is the same from 491. Since we started 492, the new implementation of the renderer allowed us to represent the voxels in a different way that was less wasteful of memory. Voxels are now represented as their location within a bounding box. For example, if the third object was added, and the bounding box was (50, 50, 50), the voxels contained within the bounding box are added to the voxel array at the index where the last object ended. Instead of representing the whole world, the individual objects are fully represented in memory, empty spaces included. Representing individual objects also allows us to update voxel objects from the CPU a lot faster by only changing 50 by 50 by 50 voxels in a 1024 by 1024 by 1024 world.

5 Testing

5.1 Testing Process

5.1.1 Unit Testing

- File importer: Load .vox files made in MagicaVoxel into Memworld to test the bounds of the program and ensure consistent results with what was created in MagicaVoxel.
- Create OpenCL program: Make sure all necessary files are working on the system. Have all API calls for OpenCL defined for the kernel, work groups and enqueueing. Make sure the application can be built with the setup by running the makefile and watching for errors.
- Run OpenCL program: Have defined inputs with expected outputs for the application. Establish these inputs/outputs for the serial application and compare inputs/outputs for the OpenCL implementation.
- Physics: Have a predefined input of objects in the world, have an expected outcome of how they will interact with the world, and check their coordinates to make sure the results match up with our expectations.

Tools:

- We will write our own unit tests in a separate c file that we create.
- MagicaVoxel for quick and testable world and object files

5.1.2 Interface Testing

- Internal API Calls: As our project is effectively a rendering API, we want to ensure that it is easy for developers to use it to render scenes defined in memory. In order to do this, we have a test program with a main function that shows the uses of the functions in the API. If anything seems counter to how it should be used or difficult to set up, we will be able to reorganize the API calls and functions to ensure the usage pipeline is smooth and simple.

5.1.3 Integration Testing

- File and world importing:
    - File importing and world importing are critical to the project as they make sure that the world bounds and voxel array are properly generated. If they have any issues then the program will quickly crash when running, so it's vital that these are tested together to confirm they work.
    - These will be tested with our own created functions to confirm they work together without causing any issues in the generation.
- OpenCL and renderer:
    - OpenCL is the framework that is tasked with making this application more efficient. The renderer will work on generating the world that we define. The renderer will be parallelized using the OpenCL framework, so it will be critical that these two are integrated correctly. Possible issues that can arise from poor integration would be poor efficiency or the world not rendering correctly.

    - These will be tested by checking inputs and outputs for the renderer and verifying correct definitions for the OpenCL api calls/ setup for what we want to run for on the kernel.
- Physics and renderer:
    - Physics will involve many objects moving around the world and interacting with one another. With this, we need to make sure that the renderer is properly displaying the objects as they move and won't have any overlapping or graphical bugs.
    - This will be tested visually to confirm that no graphical errors are happening.
- Lighting and renderer:
    - Lighting will allow the application to produce a light source to introduce shading. The renderer is tasked with displaying the world which will include lighting components. Integration will be important to ensure that there are no abnormalities with how lighting is functioning in the world and having the application run efficiently.
    - This will be tested visually to confirm that no graphical errors are happening.

5.1.4 System Testing

- For our system level testing strategy, we will be taking all of the components from integration testing and making sure that each of the components is working properly in accordance with our requirements.
- We will focus on changing the world size, window size, and render distance to make sure that we are meeting the input and output requirements that are given.
- System level testing is to go through full use cases and check that each system is behaving as expected. Interface testing would involve going through normal inputs and verifying that the system reacts appropriately for expected and unexpected inputs. Unit testing would involve verifying results of specific modules during a use case, an example would be checking physics based collisions. Integration tests would be for use cases that touch multiple components of the system. An example would be importing an object and making sure the world is correctly set up with said object.

- Tools:
    - Dr. Memory for memory leaks

5.1.5 Regression Testing

- We are ensuring that new additions do not break the old functionality by running our unit, interface, integration, and system tests before submitting a merge request for the code that is being created for the task assigned. We will also be creating the necessary tests needed for future regression testing of the new code.
- Some critical features that we do not want to break in this project are:
    - The kernel rendering function because that is what we use for displaying the scene to the user. If this is broken, the user wouldn't be able to use the program properly.
    - The file importer function because that is what we use for loading our scene with objects. If this is broken, the user wouldn't be able to see the world correctly.

    - Creating the OpenCL program pipe as well as running the OpenCL program needs to work. If OpenCL doesn't work, we won't be able to meet the FPS requirements as well as the world won't be rendered correctly.
    - Physics needs to be close to be correct. There is not a necessity for accurate physics, but we would like objects to fall and jumping to work for our demo application.
- The regression testing is driven by our requirements, but there will also be extra testing to make sure that modules needed to meet requirements are implemented correctly.

5.1.6 Acceptance Testing

- We have a list of requirements from the client such as a target frame rate, target voxel density, and target world size. Voxel density and world size are configurable variables inside our project, so we can simply set those variables to our requirements and run the program to make sure that those requirements are met. For frame rate, we are using a formula to calculate the frame rate of the project, and it is output to the application window, so we can take the average frame rate and compare that to our requirements to ensure that it meets expectations.
- We involve our client by showing memworld running to different specifications and ensuring that he is satisfied with the progress we are making. We run our ideas for new modules that we want to add by him to make sure that we are meeting what he invisions.

5.2 Testing Results

Using Dr. Memory in order to look for memory leaks, we were able to clear up all but two bytes of leaked memory. The last 2 bytes of memory that were leaked were not under our control.

In terms of testing the performance of our application, listed below are the target settings that were given to us. The application has changed over time which allows us to test using the testing settings listed below.

Target Specifications

Voxel Density	Window Size	Draw Distance	World Size	Average Frame Rate
5	1024 x 768	100	Height: 16 * Voxel Density Width: 25 * Voxel Density Depth: 40 * Voxel Density	Result of testing for FPS

Testing Specifications

Voxel Density	Window Size	Draw Distance	World Size	Average Frame Rate
N/A	1024 x 768	1024	Height: 1024 Width: 1024 Depth: 1024	Result of testing for FPS

Systems of Each Team Member

	GPU used	Operating System	CPU	RAM	Average Frame Rate (World 1, 2, 3, 4)
Collin	NVIDIA GeForce GTX 1060	Windows 10	Intel i7-7700HQ	16 GB	45, 70, 55, 100
Cristofer	Radeon RX 580 Apple M1	Windows 10 MacOS	Ryzen 5 2600 Apple M1	16 GB 8 GB	33, 33, 50, 100 213, 238, 233, 226
Dalton	NVIDIA GeForce GTX 980	Windows 10	Intel i5-4690k	16 GB	70, 100, 80, 111
Jay	NVIDIA GeForce GTX 1070	Windows 10	Intel i7-6700k	16 GB	90, 130, 100, 140
Mason	NVIDIA GeForce GTX 1060 with Max-Q Design Intel(R) UHD Graphics 630	Windows 10	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz	16 GB	67, 91, 71, 143 56, 59, 57, 77 (pixel density 2)
Wil	NVIDIA GeForce RTX 3070 NVIDIA GeForce GTX 970M	Windows 10 Manjaro Linux	Intel 10th Gen i7 Intel 6th Gen i7	16 GB 12 GB	250, 275, 225, 275 N/A

Our design for testing is useful because we get to have our code tested on a variety of platforms and see what the performance is based on the different hardware that is available. This helped us determine if we were moving in the right direction or if we needed to change how we are rendering and/or taking inputs from the user. The one downside of testing an application like ours is that we have to manually run systems tests which is harder to do because humans are not perfect at replicating actions. With everyone testing, we have been able to detect what is wrong and what needs to be changed with most of the bugs that occurred.

6 Implementation

6.1 Memworld Implementation

This is where our main function lies. We first read in the settings file and determine if we should use the default settings of the application or what is contained in the settings file. We then initialize the memory for the voxels in the world as well as the texture that we are displaying to the screen in order to render the world. After the buffers are set up, OpenCL is initialized (described in the OpenCL Implementation section), voxel objects for the world are read in, and the chunks containing those objects are updated. OpenGL is then initialized to be able to display the texture that we are generating to a plane. Then, the main loop starts running until there is a signal to the application that it should close (pressing Q or the X button in the top right corner). In the loop, mouse and keyboard inputs are processed, and then, the rendering kernel is sent to the GPU to run. While the renderer is running, physics is updated for all the objects. The application then waits for the renderer to finish processing and return the texture data for OpenGL to use to display the texture to the application window.

6.1.1 World Chunks

The world size that is supported by the application is 1024 x 1024 x 1024 voxels. This world is split up into 8 x 8 x 8 chunks where each chunk contains an array of indexes of the objects in the objects array. Each chunk also contains the number of objects in the chunk. To determine if an object is in a chunk, we take the bounding box of each object and if the bounding box is inside of the chunk, the object is added to the chunk. Using chunks speeds up the rendering algorithm as it is only checking objects that could intersect with a ray instead of checking all of the objects for every ray.

6.1.2 Moving Objects

In order to move objects, the bounding box of the object is moved by the amount that the object is intending to move. Every time that an object is moved, there is a possibility that the object will move out of its chunk and into another chunk. This is the reason why the object is removed from its chunk, and then, it is moved. Once it has moved, the chunks are updated based on the new position.

6.1.3 Input Handling

The player is represented as a camera (i.e. the player doesn't contain any voxels). The default camera speed is 1. Powerups can increase the speed of the player. GLFW allows us to get inputs from the player. WASD keys move the player using sin and cos of the azimuth of the player. In other words, it allows the player to move where they are facing. The player is able to jump using the space key. Based on these inputs, the player's position is updated by the change indicated by the inputs. Collision detection of the player is then calculated after the position is updated. First, the bounding box collision is checked, and then, if there is a collision, the voxel based collision is checked. If there is a collision the player is moved until there are no more collisions and the player's y velocity is set to 0. For detecting mouse inputs, there is a callback function that is created for GLFW to use. It takes the difference between the current mouse position and the previous mouse position and updates the altitude and azimuth accordingly. There are checks for max and min azimuth positions, so the player won't be able to look up so far that the camera inverts itself.

6.2 OpenCL Implementation

6.2.1 Setup

OpenCL is what enables the rendering algorithm to run on the GPU using multiple threads. First, the application grabs the different GPU platforms that are available on the user's device. That is when the user chooses the platform they would like to use or the setting file automatically chooses the platform for the user. A compute context is then created along with a command queue. The application then reads in the render algorithm stored in a file. There are certain defines at the top of the renderer which get modified based on some settings the user selects. This is done in order to speed up the renderer by removing some of the source code that isn't necessary based on the settings. The kernel program is then built and the memory needed to be sent to the GPU is allocated.

6.2.2 Adding Objects to the World

OpenCL can only take in one dimensional arrays, so we have an array that holds all of the voxels for the application. When the application reads the voxel data in from a file for the world, the application sets the memory on the GPU to match that read in from the file. When the next object gets read in, the application sets the memory starting after the first object's length of memory. Each object keeps track of the index in the array that the data is stored at in order for the renderer to know which data belongs to which object. When changing worlds, the offsets for where the next object is going to be stored is reset, allowing us to overwrite the previous data in the array.

6.2.3 Main Loop

For every frame that gets rendered the application sets the arguments to the kernel that was compiled, and then, the kernel is enqueued to the command queue. When the kernel is enqueued to the command queue, the work size is set. This determines how many threads are used for running the kernel. We set the amount of threads to equal the amount of pixels in the image (one per pixel). The GPU runs the kernel. To get the data back to show the image to the application window using OpenGL, the application waits for the command queue to be finished executing tasks. The application enqueues a read buffer to the command queue in order to read a certain amount of data from GPU memory.

6.2.4 Teardown

After the application has finished running the memory that has been reserved gets released along with the context, command queue, and kernel.

6.3 Renderer Implementation

Our implementation for the renderer has been changed from what was stated in the design section from the previous semester. We now encapsulate the objects with bounding boxes that contain all of the voxels in the object. Each of these objects are then put into an array of chunks that make up the world. The world is 8x8x8 chunks split by the world size that is indicated. If an object's bounding box is contained in a chunk, the object is added to that chunk's list. Using the chunk array, a ray gets sent and traverses the chunk array using the 3DDDA algorithm. If there is an object in the chunk. The objects in the list of that chunk are all checked to see if the ray interests the bounding box of the object. If it intersects with the object, the distance is stored to that object. Once all of the objects in the chunk have been checked, the shortest distance is picked to traverse. The object is traversed from the intersection point using the 3DDDA algorithm. If a voxel is hit, the color is set for that pixel of the image. If a voxel is not hit, the next closest object is checked the same way. Once all of the objects in a chunk have been checked, the chunk array is traversed to the

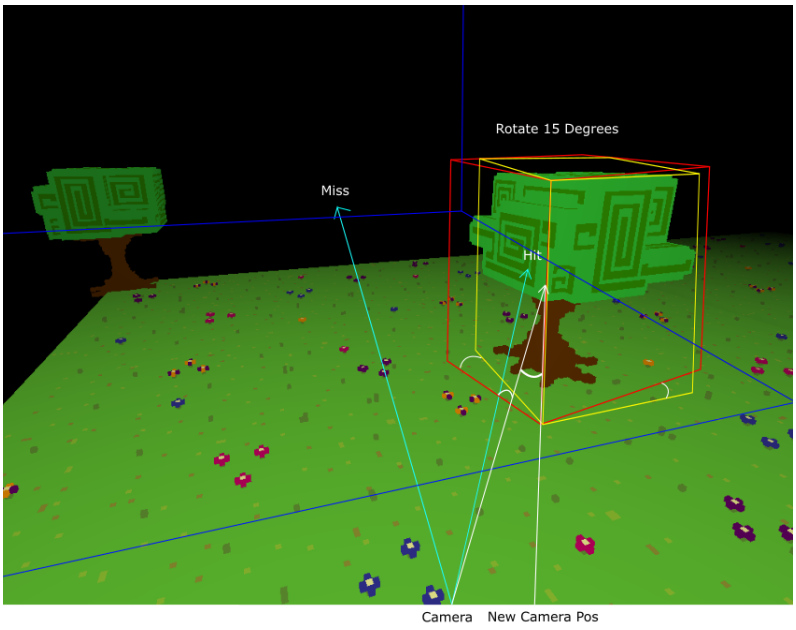
next chunk. If no voxels are hit by the time the ray reaches the bounds of the chunk array, the pixel is set to black.

6.3.1 Object Rotation

Each object is able to rotate around the y-axis. This is just a visual rotation, and the collision is still axis aligned. This works by rotating the ray that we are sending from the camera by the angle of the object. This is done by using a rotation matrix where the y part stays the same and the x part is ray_x * cos(ANGLE) + ray_z* sin(ANGLE) and the z part is ray_z * cos(ANGLE) + ray_x * sin(ANGLE). Then, the origin of the ray is rotated around the center of the object which lines the ray up correctly to enter the bounding box. Since we are using Euler angles, there is a possibility for Gimbal lock, but nobody has experienced it yet, and it might not be possible by the way the user's input is translated to the ray direction.

Shown below is a diagram depicting how the rotation works. The blue lines represent the chunk that contains the objects. The teal lines are the rays from the camera. The red bounding box is the standard bounding box and the yellow bounding box represents the rotated object.

Object Rotation



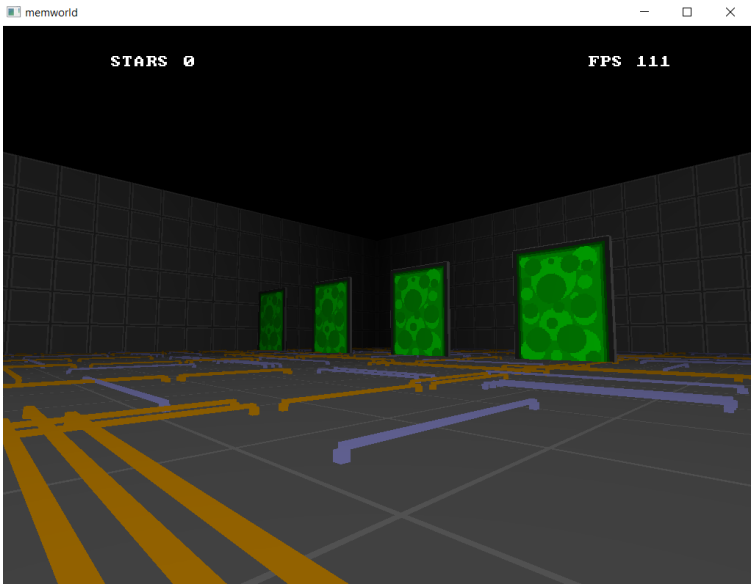
6.4 Lighting

6.4.1 Distance based

The distance that we get from the point of intersection of the ray and the object, from the start of the ray, is divided by a view distance set by the user in the settings file. We take one minus that percentage in order to get the percentage in which we reduce each r, g, and b component of the color.

Shown below is an image of distance based lighting. The further the voxel is in the world, the darker it becomes.

Distance Based Lighting



6.4.2 Ray based

The render kernel is compiled at runtime, so ray based lighting has a define in order to reduce the amount of code in the kernel to speed up slower GPUs. If ray based lighting is defined, once a voxel is hit by the initial implementation, a second ray in the direction of the far upper corner from the point (0, 0, 0) is traversed to determine if a voxel is hit. If no voxel is hit within the current object, the pixel is shaded normally. Otherwise, the pixel is shaded by reducing the color by 50%.

Ray Based Lighting

6.4.3 Pixel Density

If the pixel density is 2. Then, for each ray from the camera, a 2x2 pixel is drawn to the screen. This helps decrease the amount of threads that are needed to run on the GPU which speeds up smaller GPUs like the integrated graphics chip some computers have installed.

Shown below is an image of a pixel density of 3. The performance increases, but the lines aren't as crisp.

Pixel Density

6.5 Physics Implementation

Only objects that are moving are checked for bounding box and voxel based collision. That could be if they are affected by gravity or have their x or z velocities not equal to 0. For objects affected by gravity, there is a dampening effect, so they don't bounce forever. When the velocity in the y direction is close to 0, then the y velocity is set to 0.

6.5.1 Bounding box based collision

Every object already has a bounding box associated with it for the rendering of the objects to work properly. The objects bounding boxes are axis aligned bounding boxes, so we are able to check the intersection of two boxes by comparing the side positions to each other along the x, y, and z axis. If there is an intersection, the objects get moved in the opposite direction in which they came by the intersection amount.

6.5.2 Voxel based collision

In order to save time, we first check if there is a bounding box collision between 2 objects. If there is a collision, the object is then checked for voxel collisions by using the size of the moving objects bounding box and checking voxels of the object against voxels in the same position of that of the object's voxels. In order to move the object out of the other object, the object moves up, checks for collisions, then right, checks for collisions, and finally left. The player would get stuck on small ledges because of this because there would be a voxel collision. To remedy this, the player has 2

bounding boxes. One is the full bounding box and the other is a smaller height bounding box that is raised up. If there is a voxel collision in the first and not the second, then the player will move up. This acts like an automatic jump feature to get over small bumps in objects or allows the player to flow up ramps.

6.5.3 Moving platforms

We are making a 3D platformer for our game which usually includes moving platforms. Our implementation of the moving platforms involves setting 4 different points that the platform will interpolate between where the last position is the position in which the platform starts. We define the amount of steps that the platform takes between points to indicate how long the platform will take to move. If the player collides with a platform (is standing on top of the platform), the player will move along with the platform. This is done by moving the player the same amount as the platform is moving.

6.6 Worlds Implementation

6.6.1 Memworld Test Application Design

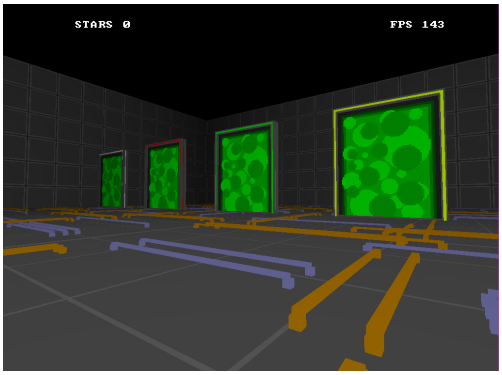
Gameplay Overview:

- The type of test application that we are planning to make will be a 3D platformer.
- There will be 4 different levels that increase in size as you progress through the levels up to a max size of 1024 x 1024 x 1024.
- The overarching goal in these levels will be to acquire various collectables
- There will be an area that will be the hub in which you will be able to choose which level you want to enter.
- The user will be able to press the "H" key if they want to go back to the hub area.
- The application will attempt to utilize the full range of our 1024 x 1024 x 1024 voxel worlds by incorporating verticality in the level design.
- The application will provide hazards or failure states in the level design (lava, pits, enemies, etc...) to provide some kind of challenge.
- The application will put a major focus on displaying the strengths of the engine by highlighting working features (pillars/windows to show off shadows, falling objects/moving objects for physics, etc).

6.6.2 Hub World

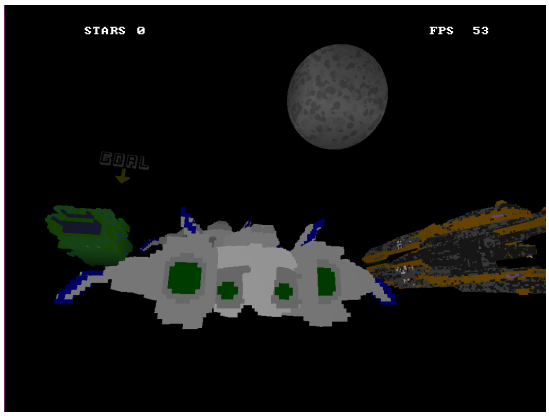
Hub World

In order to allow players to select which world they want to play in, we incorporated a simple hub world for them to start in. This world is a simple room with 4 color coded portals located within it. Walking into any of the four portals teleports you to the world associated with it. World 1 is yellow, world 2 is green, world 3 is red, and world 4 is white. in order to allow for players to swap worlds at any time, the hub world can be accessed by pressing the H key during gameplay.



6.6.3 World 1

World 1

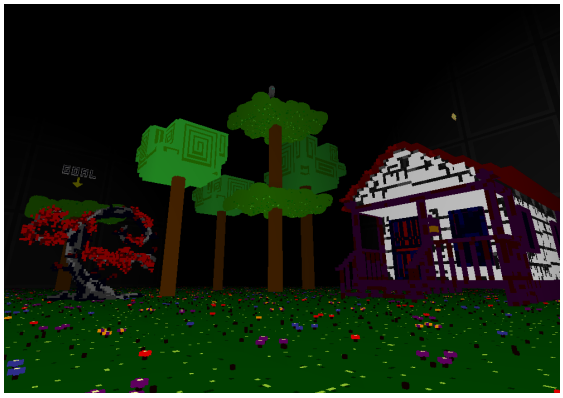


The first world takes place in space, where you are jumping from spaceship to spaceship attempting to reach the large green ship in the distance to access its biosphere, an artificial forest and the setting for world 2. As it is the first world, it starts mechanically simple to ease the player into the game, requiring only simple platforming to progress. The first collectible can be found here as well, hidden on the lower wing of the larger, more detailed ship.

On the technical side, the world contains only 19 objects with only two of those objects containing any special properties, the star for collecting and the goal for world traversal. Two of the objects, the goal and the goal warship, have been scaled up and the goal post has been rotated to be visible from the start of the map. While the world only contains 19 objects with these simple additions, the objects used are highly detailed and quite large, which can lead to performance issues if more were to be added.

6.6.4 World 2

World 2



The second world takes place in a biosphere, which is full of trees and a large cabin. The collectible for this level is another star, this time taking place on top of the cabin, which is normally not a required area in order to progress through the level. This will require players to progress through the level as normal by going from treetop to treetop, eventually reaching the largest tree. This large tree contains a powerup that will allow the player to do a double jump, and during this time they will have a much easier time reaching the cabin to get the optional collectible. They must then utilize the double jump to get to a tree near the corner of the map that, while not as tall as previous trees, is still too high to be accessible without the double jump being acquired. Upon reaching this tree, they will find the goal which will teleport them to the third world.

From a technical standpoint, this world required several workarounds to keep the overall world running smoothly. For one, we created several different types of trees, and utilized the scaling feature in order to scale trees up to various heights. This allowed us to have interesting interactions with jumping from different tops of different trees in order to achieve a sort of verticality with our level design. This scaling came at a cost however, as this resulted in a lot of empty space taking up memory with our bounding boxes for these trees, as the main trunk was connected to the leaves, so

the box would be as wide as the leaves and thus go through the empty space near the trunk of the tree. This required us to modify the assets used for our trees to separate the trunks from the leaves and count them as two separate objects, which greatly increased the performance of the world.

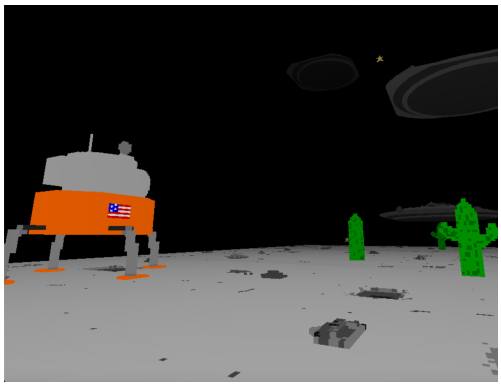
6.6.5 World 3

World 3

The third world is back in space similarly to world 1. There are different spaceships that function as platforms for the player to jump on as well as meteors that continuously fall to act as an obstacle for the player. The player is expected to go from one spaceship to the next until they reach the final platform. When the player is hit by a meteor they are sent back to the starting platform to try again. The meteors also are sent to the top of the world to fall again whenever they hit the bottom of the world. This level was kept relatively short so that the player won't have to spend too much time getting to where they were when they got hit. This level was relatively straightforward to implement. The only limitation was that the meteors needed to be fairly small voxel count wise else it would greatly reduce performance since there are a lot of them falling at once.

6.6.6 World 4

World 4



The fourth and final world takes place on a Moon-esque world with moving rocks floating above the surface. The goal for this world is that the player must collect a handful of stars that are placed throughout the world. The player must use moving platforms (massive rocks floating above) to reach the star collectables. Given that this world is based off the moon, the gravity for this level has also been changed to -3, as opposed to -9.8 in the other worlds. This will give the player some help with traversing each platform and with reaching the stars that are placed in harder to reach locations. Once all stars are collected, the player must make their way toward the highest platform in the world using the moving platforms. This platform is not moving and contains a teleporter that will send the player back to the Hub. Effectively, ending the game.

6.7 UI Implementation

The application displays the current FPS in the top right corner and the number of stars the player has collected in the top left corner. In order to display the text and numbers to the screen, characters are encoded into 8 x 8 bitmaps. The 8 x 8 bitmaps are provided by Daniel Hepper (Hepper). We are able to do this for strings or numbers by indexing the bitmap array where the number or character is located. Using these bitmaps we change the color of the texture at a location if the bit in the bitmap is a 1.

Shown below is the UI that we have for the application.

UI Implementation

6.8 File Importer Implementation

6.8.1 Type of Files

The files that our group is importing to our application are .vox files. These are in the format given by MagicaVoxel (see "Voxel Model Github Repository" by Ephtracy in the appendices). They contain color and location information for each voxel in the model along with the total length, width and height of the model.

6.8.2 How it Works

Our program opens and reads through the .vox file to gather the information required to incorporate it into the world. First, it finds the number of voxels within the object and checks that it doesn't exceed the maximum number of voxels our system can handle. We then select a for loop to run depending on the rotation style selected, as the rotation will necessitate a translation to occur during every iteration of the loop. Within the loop, we assign every single voxel within the bounding box an RGB value which is selected from the default palette. If we are scaling up the object, we assign a cube of voxels with dimensions equal to the scaling factor for each voxel in the original .vox file. This allows us to import objects larger than the 256 x 256 x 256 limit of the .vox format, though with the trade off of less fidelity.

6.9 Unit Tests Implementation

We are not using a library for unit tests, but instead made our own in order to run unit tests. We chose to do this because we wouldn't have to learn a new library and instead could make something that made sense to the group. The tests are run off of one function that takes in a function pointer to a test function and totals up all the passed tests that are associated with a certain file. The results are then printed out. If the tests fail, the user will know which test failed by the error printed out.

7 Closing Material

7.1 Discussion

This project is a mix of a product and an experiment. Our group was assigned this project to see if having a direct-memory representation of a voxel would increase the speed of different aspects of an application such as collision detection or rendering. The client already had a proof-of-concept ray casting engine and wanted us to improve the speed with certain requirements that needed to be met. As of this moment, the new renderer is roughly 300x faster (on certain computers) than the proof-of-concept given to us in a much bigger world with a longer range of vision. We are meeting the FPS requirements of 30 frames per second on almost all of the GPUs. The integrated graphics chip struggles and hovers around 20-30 depending on the world. This can be improved by using a pixel density of 2 which averages around 60 FPS. Collision detection is faster compared to that of modern game engines, but our group's rendering struggles to compete on a high texture density level because of the hardware acceleration for modern rendering on GPUs.

7.2 Conclusion

Throughout these two semesters that we have been working on this project, we have learned a lot. Most of the team has never worked with or understood how games were rendered. Given that this project's main objective was to improve the performance of a renderer for direct memory based voxels, we all learned something that is valuable as most applications have to deal with some sort of rendering.

Another important lesson of this project is learning how to work in a larger group. We had a total of 6 people in our group. Throughout college the most people classes allowed in a group was up to 4. A group of 6 people adds complexity to making sure that everyone has something of value to do and that everyone is understanding how the application works. This is an important lesson to learn because in industry there is a high probability that you will work in larger groups.

The last big lesson that was taken away from this project was how to work on and maintain a project for a long period of time. Projects in industry can last for years and keeping a clean repository, commented code, and documentation about the project is something that is crucial in keeping high productivity of the current team and new members that could join in the future.

Overall, this capstone project has aided us in getting a glimpse into what working in industry will look like, and what we should expect when we graduate.

7.3 References

C. Crassin, F. Neyret, and Sillion François X., “Gigavoxels: Un pipeline de Rendu Basé Voxel pour l'exploration efficace de scènes larges et détaillées,” thesis.

“Computer Graphics for the 'Rest of Us'.” *Scratchapixel*, 31 Aug. 2022, https://www.scratchapixel.com/index.php?redirect.

Ephtracy, H. Castaneda, DimasVoxel, and J. Kirch, “Voxel Model Github Repository,” GitHub, 17-Jan-2022. [Online]. Available: https://github.com/ephtracy/voxel-model.

Hepper, Daniel. “font8x8/font8x8_basic.h At Master · Dhepper/font8x8.” *GitHub*, 11 June 2019, https://github.com/dhepper/font8x8/blob/master/font8x8_basic.h.

K. Silverman, “PND3D Demo and Source Code.” [Online]. Available: http://advsys.net/ken/voxlap/pnd3d.htm.

Laine, Samuli, and Tero Karras, "Efficient sparse voxel octrees–analysis, extensions, and implementation." NVIDIA Corporation 2.6 (2010).

Wilhelmsen, Audun, “Efficient Ray Tracing of Sparse Voxel Octrees on an FPGA” Norwegian University of Science and Technology, Trondheim

7.4 References - Art Assets

GameStudioJukaaa. (2019, January 18). Voxel Space Ship. OpenGameArt.org. Retrieved 2022, from https://opengameart.org/content/voxel-space-ship-0

edited sizing and color

Parata, M. (n.d.). 10+ voxel spaceships assets for free by maxparata. itch.io. Retrieved 2022, from https://maxparata.itch.io/voxel-spaceships

edited sizing and color

Parata, M. (n.d.). Free Voxel Graveyard asset by Maxparata. itch.io. Retrieved 2022, from https://maxparata.itch.io/voxelgraveyard

edited sizing and color

Parata, M. (n.d.). Voxel environment : Counrty side by maxparata. itch.io. Retrieved 2022, from https://maxparata.itch.io/counrty-side

edited sizing and color

Texnist. (n.d.). Voxel spaceships by Texnist. itch.io. Retrieved 2022, from https://technistguru.itch.io/voxel-spaceships

edited sizing and color

8 Appendices

8.1 Operational Manual

Setup for Windows:

Part 1: Downloads

The following items will need downloaded in order to properly run this program:

GLFW (available at https://www.glfw.org/),

Vulkan SDK (available at https://vulkan.lunarg.com/sdk/home#windows, get the windows SDK installer. Run the installer as normal, default options should be fine.)

Doxygen (available at https://www.doxygen.nl/download.html, scroll down to Sources and Binaries, select the binary distribution for Windows. Run the installer as normal, default options should be fine.)

CMake (available at https://cmake.org/download/, select the Windows x64 Installer)

MinGW-W64 (available at https://sourceforge.net/projects/mingw-w64/files/, scroll down to the MinGW-W64 GCC-8.1.0 section, download the "x86_64-posix-seh" version)

OpenCL (available at https://github.com/KhronosGroup/OpenCL-Headers)

Part 2: Setup

Extract the "mingw64" folder of your posix-seh zip file you downloaded earlier to the root of your drive (ex: C:\mingw64). Next, rename this folder to "MinGW" for easy use in the future.

Next, make sure that all of these programs have been added to your PATH. To check, Press the windows button, and search for "Edit the system environment variables". This should open up a "System Properties" window. Click the "Environment Variables..." button. Under the "System variables" tab, select the "Path" variable, then click edit.

Confirm you have the following in your path (if you don't, add it now):

"C:\VulkanSDK\1.2.198.1 (or some other version)\Bin",

"C:\Program Files\doxygen\bin",

"C:\MinGW\bin"

Part 3: Installing glfw

Make a folder on your desktop called "glfw". Open this folder, and make another folder named "glfw" inside this folder. Next, Take the downloaded glfw zip from part 1, and extract the contents to

your desktop. You should have on your desktop a folder called "glfw", and a folder called something like "glfw-3.3.6".

Next, open the CMake program you downloaded earlier. In the "Where is the source code:" option, navigate to your "glfw-3.3.6" folder. In the "Where to build the binaries:" option, navigate to your "glfw" folder. Press "Configure" down at the bottom, and in the specified generator, look through the dropdown and select the "MinGW Makefiles" option. Make sure that "Use default native compilers" is selected, then press finish. The following options should be checked, the rest should be unchecked:

"GLFW_BUILD_DOCS", "GLFW_BUILD_EXAMPLES", "GLFW_INSTALL"

Next, change the value of "CMAKE_INSTALL_PREFIX" to point to the folder "glfw", specifically the folder "glfw" inside your "glfw" folder on your desktop. Finally, press "Generate".

Open your terminal (Like cmd), and navigate to your desktop "glfw" folder. Type "mingw32-make" and press enter. Wait for this to finish, then run "ming32-make install" and press enter.

Now, you need to copy some files into MinGW. Navigate to the following directory: "C:\users\username\Desktop\glfw\glfw\lib", and copy the libglfw3.a file to both your "C:\MinGW\lib" directory, and your "MinGW\x86_64-w64-mingw32\lib" folder. Next, copy the "GLFW" folder inside "glfw\glfw\include" to "C:\MinGW\include" and "MinGW\x86_64-w64-mingw32\include".

Part 4: Installing OpenCL

Extract the OpenCL zip that you downloaded earlier. Copy the "CL" folder to both the "MinGW\include" directory, and the "MinGW\x86_64-w64-mingw32\include" directory. Next, navigate to the directory "C:\Windows\System32", and copy the file "OpenCL.dll". Paste this in your "MinGW\lib" and "MinGW\x86_64-w64-mingw32\lib" folders.

Part 5: Running the project

With a terminal, navigate to your memworld project. Run "mingw32-make", then run the command "./memworld.exe" to start the project.

Go to the “Using the Application” section for instructions on how the application works while it is running.

Setup for Mac:

Part 1: Downloads:
GLFW (available at https://www.glfw.org/)

Part 2: Setup:
After downloading GLFW, open the folder and place it somewhere you can find it.
Proceed to open the Terminal and use the ‘cd’ command to navigate to the downloaded GLFW folder.

Once the user is in this folder enter the following commands:

cmake .

sudo make install

After doing this GLFW will be installed on your Mac system.

OpenCL is implemented as a framework, OpenCL.framework, on Mac which contains the OpenCL API, runtime engine and compiler. This framework is installed by default on most modern Mac systems.

Part 3: Run the program

Navigate to Memworld project directory using the terminal, build the program using **make mac**.

After building the project, an executable file named *memworld* will be added to the project directory.

To run memworld, run **./memworld** in the terminal

*NOTE: recommended to avoid running program on IDEs (i.e. XCode), there may be additional framework requirements that can make running memworld unnecessarily complicated and can spawn additional building issues, particularly with OpenCL.

Using the Application (Applies to Both Platforms):

Once you have the application running, you will be given a prompt to choose the GPU that you would like to run the application on. Once you choose the GPU, the application will load up. The player will be spawned in a portal room. Going through each of the portals will take the player to a different level in the game. This is a 3D platforming game, so the player will be exploring the level in order to collect stars and reach the exit.

Keys W, A, S, and D are used for movement. The spacebar is used to jump. The left shift key is used to sprint. The player can press H if they want to go back to the hub area. Pressing T will bring the player to the testing area. This was used for development purposes. The application can be closed by pressing Q.

The application includes a settings file called “settings.txt” in the folder that includes all of the sources. The user can change a few different settings using this file:

- **Voxel Density**: The values for this can be 1, 2, or 3. There is no visible difference when changing this value. In previous versions this made the render clearer, but now it renders at max clarity all of the time.
- **Max Draw Distance**: The values for this can be between 0 and 2000. This will determine how far from the player objects will fade to black. Where 0 means the player can’t see anything and 2000 means the player can see objects far into the distance.
- **Window Width**: The values for this can be between 1024 and 1920. This will determine the width of the application window. The larger the value, the worse performance the user will experience.
- **Window Height**: The values for this can be between 768 and 1080. This will determine the height of the application window. The larger the value, the worse performance the user will experience.

- **Pixel Density**: The values for this can be 1, 2, or 3. This will pixelate the render of the application in exchange for better performance. The larger the value, the better the performance the user will experience.
- **Enable Ray Lighting**: The values for this can be 0 or 1. This will enable or disable ray based lighting in the application. Where 1 is for enabling it and 0 is for disabling it. Enabling it will produce hard shadows in the application, but reduce performance.
- **GPU Index**: The values for this can be -1, 0, or 1. This indicates which GPU will be chosen to run the application on. The value -1 indicates that the user will be asked to input the GPU index at startup.

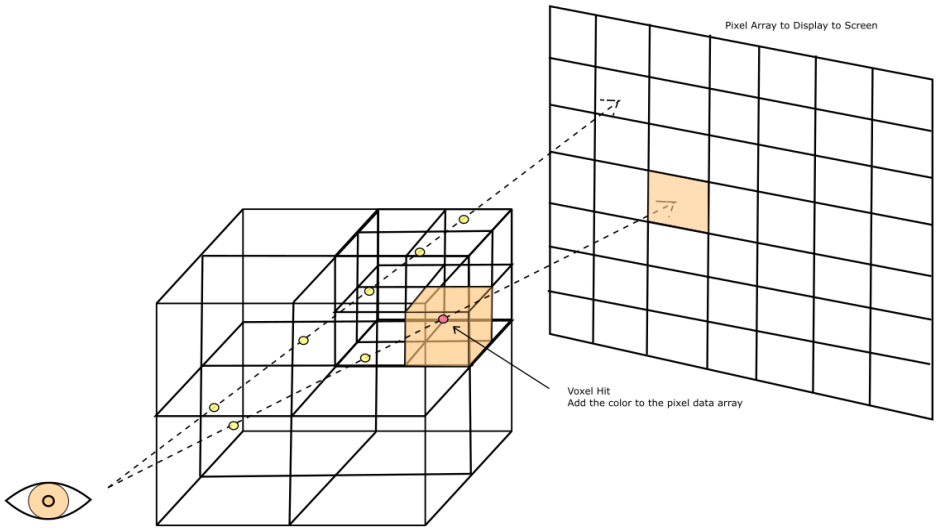
8.2 Alternative Versions of the Design

During the first semester of senior design, most of the work was done on the render, trying to increase the speed from what we were given. Eventually, we landed on the idea of having a voxel octree because that is what most products with voxels utilize. Our project uses a direct memory representation of the world, so the octree would come in terms of metadata describing which cubes contained voxels. This required a lot of memory if we wanted to make large worlds (1024 x 1024 x 1024 voxels would require around 4.3 gigabytes to be sent to the GPU along with the metadata which added around 200 megabytes. Some GPUs can't handle this much memory which limits world development.

Voxel Octree

In the diagram listed below, there is a demonstration of how the octree works and how it translates the rays to the output screen. The eye is the camera of the character that moves around the world. Rays are cast from the eye to an imaginary grid in the distance. For each ray, there is a work item that is being run on the GPU. This work item computes the magnitude of the ray and travels along the ray until it reaches a voxel that it hits. There is meta-data that keeps track of whether a voxel is contained within a larger cube. If there isn't a voxel within the larger cube, the ray is traversed until it reaches the next larger cube in order to skip unneeded memory accesses. Once it hits a voxel, the color stored in that position of the world array is transferred to the pixel array to be displayed to the screen.

Octree Ray Casting Diagram



Limitations:

For our project, we were also tasked with creating an application to demonstrate the capabilities of the renderer. Making a game to go along with the renderer seemed to be the best way to test the renderer. In games, objects have to move a majority of the time. Moving voxel objects is quite slow, especially if the object has hundreds of millions of voxels. Moving objects would also have to be done on the GPU which can be multithreaded, but is still limited by the large amount of cache misses. Physics would also need to be done on the GPU, but we would have extra time on the CPU where nothing is happening while the application waits for the texture to be returned from the GPU.