

Computer Games Development CW208

Software Functional Specification

Year IV

Robin Meyler

C00231699

04/05/2021

|  |
| --- |
|  |

# Contents

# 

[Acknowledgements 3](#_heading=h.30j0zll)

Overview 4

Define Application 5

Who is going to use the application? 5

Metrics 6

Is there precedent for this project? 6

[Project Abstract 7](#_heading=h.3znysh7)

[Project Introduction and Research question 8](#_heading=h.2et92p0)

Project Description 9

[References](#_heading=h.3dy6vkm) 12

[Appendices 1](#_heading=h.1t3h5sf)3

# Acknowledgements

I would like to thank the following people who assisted in completing this project including;

IT Carlow and all of the lectures I have had over the last 4 years for providing me with the knowledge and foundation to be able to take on a project like this.

To my supervisor Philp Bourke for assisting me through the process of creating this project and providing guidance where possible.

**Overview**

**My design goals**

The main goal of this project is to investigate and hopefully indicate that the use of GPU hardware for computation of integral game features is worthwhile. Game development naturally leads to many problems for which the current solutions are workarounds, brute force or special techniques to compute them in such a time sensitive environment. The rate of improvement in GPU hardware is currently noticeable faster than CPU hardware due to current limitations at and micro hardware level. Due to this fact, I believe that use of a massively parallel computational device is a future solution to problems such as pathfinding, collision detecting and other costly aspects to game development. In the past the ability to interface with the GPU in a non-rendering aspect was difficult but with the introduction of Vulkan and a general mindset shift, I believe that compute shaders and general GPU computation will become an integral part of game development separate from just rendering.

My specific design goal is to prove that compute shaders can be used in Vulkan through a compute pipeline alongside a rendering pipeline, executing both during runtime synchronizing each of them via inbuilt Vulkan semaphores and fences. Part of this goal is to show that the speed of execution of pathfinding is exponentially faster than CPU calculation for large sets and at large numbers that it is possible to fragment the calculation between a few frames to have it calculated without performance hiccups.

*What is the project?*

The project will display a grid of variable sizes defined before compile time from 4x4 to 500x500 of nodes up to 250,000 and possibly up to 1,000,00. The grid will be shown in 3D with 3D cubes created with Vertex buffer objects and drawn with an Index buffer to reduce vertex processing time. The outer limit of the grid will be populated with cubes and then at random the inner of the grid will be filled with cubes as obstacles on non-passable nodes. A number of agent “AI” cubes will then be spawned at random passable nodes from 1 – 1000 agents. Upon input from the user, the project will dispatch a compute shader that has memory already mapped to a storage buffer with start and node information. This dispatch will execute the A\* compute shader which will use the mapped GPU memory to calculate the fastest route for each agent created at initialization and fill a second storage buffer an array of INTS which are the Path IDs for the specific agent. The Project will capture this information and begin to move the cubes along their paths to the goal at a user designed speed. It can then be reset and repeated.

*What can the user do?*

Upon loading the program, the user can see the grid from directly above the centre of the grid. The user then has the ability to move the camera left, right, up and down to scan over the map depending on the size of the grid. The user also has the ability to zoom into and away from the grid to get a better or worse view of the detail below. Finally the user has the ability to tilt the camera up by some degrees to have an angle of the cubes in work, this gives a much more 3D effect to the program. The user can then press space to dispatch the A\* pathfinding. After a short time based on the grid size and number of agents, agent cubes will begin to move along their path. The user can then press input to increase the speed at which the agents move as well as decrease it. Once the agents reach the goal, the user can input to reset the agents to their original position.

**Define the Application**

The demo will showcase GPU accelerated A\* path finding being computed in a compute pipeline alongside the rendering pipeline and synchronized to avoid any errors in GPU usage. To do this it will use a square grid of nodes which will be represented with 3D blocks in a 3D environment. Impassable nodes will be filled with blocks and movable agent blocks as well as a goal block. This will update over time to show the results of GPU accelerated pathfinding.

# What is the application supposed to do/ Main Features

1. **Path find:** Path finding on variable grid sizes with variable numbers of agents using A\* via a compute shader to accelerate it with the GPU
2. **Render:** Renders a 3D environment with up to thousands of objects at a fast and reliable framerate using the verbose Vulkan API to explicitly set up a rendering pipeline and synchronize frames via a swap chain to draw and present.
3. **Integrate:** Integrate the rendering and computing pipeline into one project using Vulkan’s unifying nature to allow both operations to coexist while synchronized.
4. **Demo:** Demonstrate the results of the aforementioned path finding by updating buffer objects over time in the 3D world to navigate a constructed grid.
5. **Camera:** Movable camera that allows the user to scan around the grid during and before pathfinding operations to explore the scale of the project This camera will also allow for zooming functionality.
6. **Timing:** The speed of Compute shader dispatch till the GPU goes idle will be measured and compared as a means of determining the viability of the concept. Other metrics will be taken for context like workgroup size difference, mapping speed, buffer allocation speed and buffer allocation size allowances.

**Who is going to use the application?**

This project is merely a demo for GPU accelerated A\* path finding, It shows off the benefit of such an approach could bring to a specific aspect of game development. The project itself will not be designed to be used in any other aspect or product as a whole but the concept itself is designed to be portable into any game project where pathfinding is a feature. The A\* Compute shader itself will be designed to be directly usable in any project that wishes to employ the tactic as it will be neutral to the program using it and as long as the storage buffer information is matched it could be ported into any project big or small.

**Metrics**

The main metric to be attained in this project is the time it takes to execute a certain amount of A\* pathfinding incidents in microseconds to display them in seconds. To gather this information, I will use a metric of the number of agents (10 - 1000) and the size of the Grid (50x50 - 500 x 500) which is a node count of (2,500 - 250,000). The combination of these two metrics will give context to the time taken to dispatch the computer shader to the GPU and execute it. The time take will then be referred to as the speed of execution.

# Project Abstract

**The Motivation:**

Since the late 1990’s the GPU has become an extremely important component of a video game experience and in many cases the more crucial part of the experience than the CPU as the term “GPU” was popularized by Nvidia in 1999. It uses many threads, which differ from the thread on a CPU, to execute many invocations of code at once in parallel. This parallel execution fits perfectly with the current most popular form of graphics rendering of polygons. Polygons consist of vertices and each of these vertices are operated at the same time using shader scripts. Shader scripts are programmable scripts that take information specified by the creator and apply transformations. A vertex shader operates on vertices and a fragment shader operates on pixels but there is another shader the GPU allows for called a compute shader. The compute shader allows the user to specify buffers with sets of information that can each be operated on independently and is regularly used in modern games as a means of calculating particles information or image pre/post processing. The information can then be directly sent to a vertex shader or accessed again via SSBO.

Pathfinding is present in almost all modern games in one form or another and is very commonly used as a means of navigating AI agents around obstacles to a target. Determining the fastest route to a target in the shortest amount of time is extremely valuable to the developer as the number of AI agents increases, the execution time increases. A\* Pathfinding is considered one of the best and fastest pathfinding algorithms in modern programming as it uses a distance heuristic and a priority queue to ensure the best nodes are processed next. Since the GPU executes in parallel, if the correct information is provided to it theoretically could process the path for each specific AI agent in tandem and the execution time would be the same as the slowest single execution. The main motivation of this project is to determine what speed of execution benefit would be achieved by this approach compared to general CPU unthreaded approach.

Vulkan is a modern graphics API released for public use in 2016 by the group Kronos, the creators of OpenGL. Vulkan was designed as a cross platform abstraction over the GPU that requires the programmer to explicitly declare every part of the rendering pipeline and memory allocation on modern multicore computers. The API is much more verbose than OpenGL but it unifies the graphics and compute functionality into a single API as it expects general purpose usage.

**The Aim:**

My goal for this project is to examine the speed of execution of the various amounts of agents pathfinding on various amounts nodes using A\* pathfinding on the GPU via a compute shader compared to a non-threaded CPU implementation. I will be using the Vulkan API to synchronize the compute operation with general rendering, for controlling the exact memory allocation needed for each A\* execution and to map to, write to and read from device local VRAM memory that is optimized to operate as fast as possible. Upon capturing the speed of execution of both approaches, I will attempt to determine if and when this approach might be viable as a pathfinding solution in a modern game.

# Project Introduction and Research question

*“GPU acceleration of A\* pathfinding with the Vulkan API”*

**Question:** “What execution speed increase can be achieved from A\* pathfinding with GPU acceleration compared to traditional CPU execution?”

This project objective is to use GPU accelerated execution of the A\* pathfinding algorithm to determine the performance benefits that may occur compared to a traditional non CPU threaded approach.

I am using the A\* Pathfinding algorithm because pathfinding is one of the most common problems to be solved with all modern games big and small and A\* is considered one of the best and fastest methods of finding the best path from one position to another in an obstacle rich environment by using a distance heuristic to evaluate and set specific node information during execution that, with the help of a priority queue, allows for fast resolving of paths.

The reason I am using the GPU to accelerate the pathfinding is because the design of GPU architecture means that it executes many invocations at the same time in parallel. This will theoretically mean a drastic improvement in execution time and the difference, if any, is what I wish to measure.

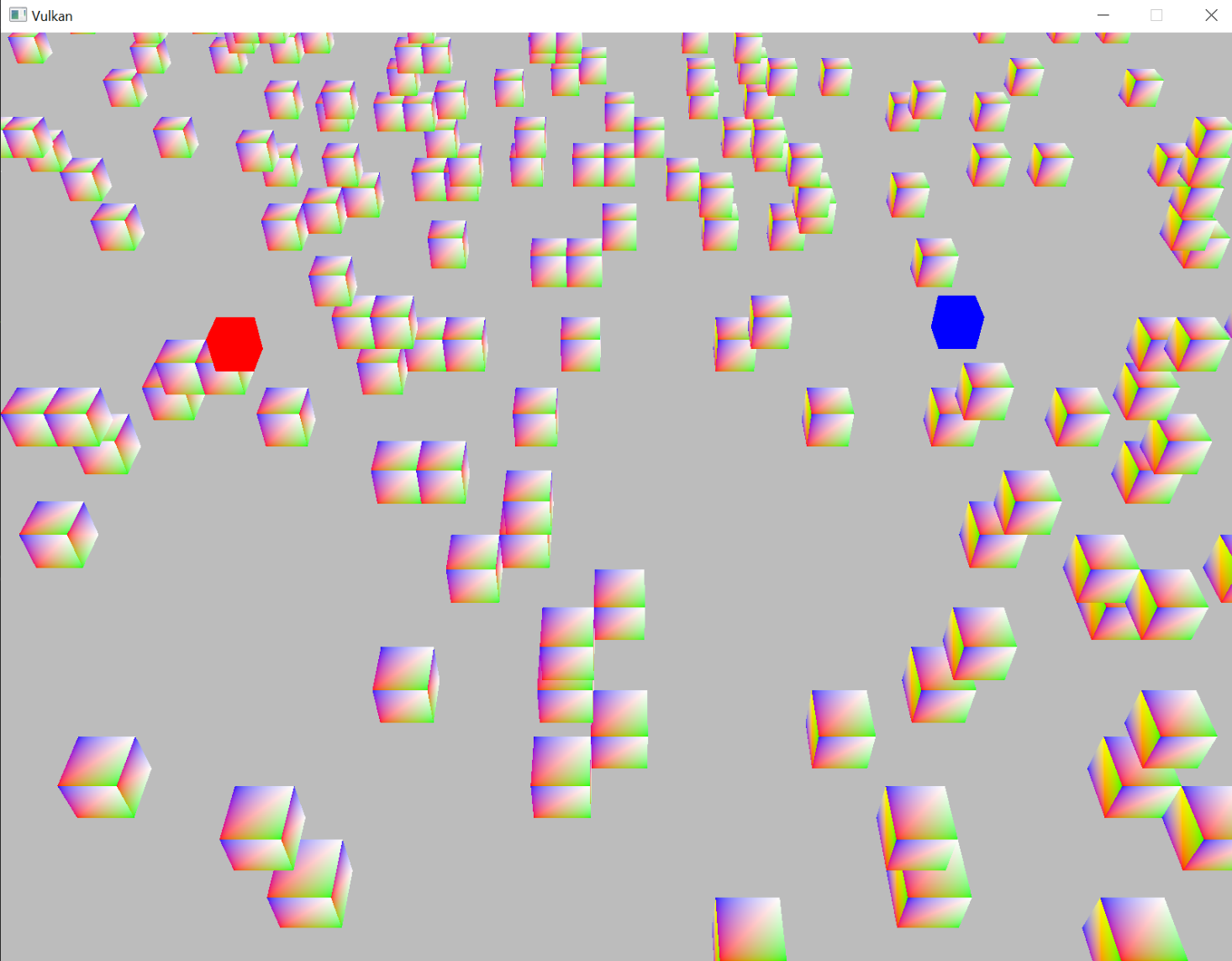
I am using the Vulkan API to interface with the GPU as it is the most modern, verbose and explicit GPU that allows for specific memory allocation. Vulkan unifies the rendering and compute functionality to allow the developer to both render to a window as well as use compute execution in tandem by synchronising the GPU with fences. Vulkan’s general approach to rendering is to decide if you are using double or triple buffering and then create a swap chain with many images, it is then your job to synchronize each of these images so that while you are presenting one image to the screen you are writing to another image. Each stage of drawing and presenting happen at indeterminate speeds so semaphores and fences allow for minimizing time lost. This synchronizing fundamental allows for the use of compute shading operations that don’t interrupt the rendering.

The potential impact of this project would drastically increase pathfinding execution time in such a way that would leave CPU threads available for more dynamic processing such as networking that may change the nature of its execution more often and need a lot more custom creation. Pathfinding can simply be a brute force execution that happens as fast the CPU will allow it.

**Project Description**

Before starting the program, the code must be adjusted to decide how many nodes and how many agents are wanted for the demonstration. The “Astar.comp” compute shader must be updated to expect the same amount of nodes for the memory to match on both the CPU and GPU. Once the shader is updated, you will need to compile GLSL shader code into SPIR-V using the provided compiler.

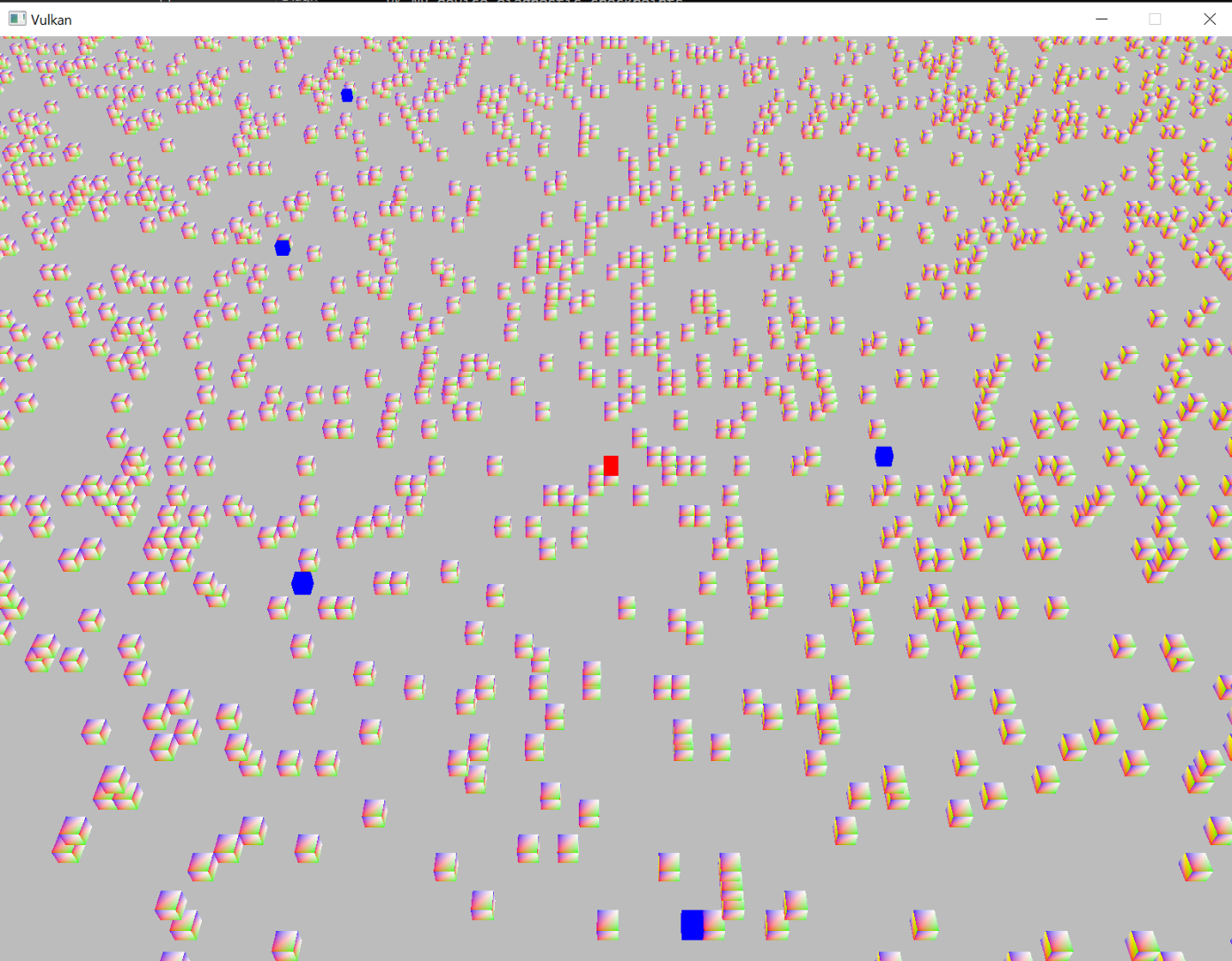
Once the user decides on the amount of nodes and agents the program will attempt to create the requirements and allocate specific GPU memory to accommodate the nodes in a storage buffer, at the same time the program will create Vertex buffer objects of cubes for the outer walls of the grid. The program will create a cube for agents of the number selected and then randomly place them on a passable node on the map. The program will create a Goal cube at the selected goal position.



*Up close view of a grid with obstacle cubes, the goal cube and one of the many agent cubes.*

Once all of the storage and buffer object objects are allocated on HOST\_VISIBLE memory as staging memory and copied to DEVICE\_LOCAL memory on the GPU, the program runs and the user is shown an above view of the grid of the desired size. The rendering pipeline renders to images and swaps them for presenting for triple buffering using synchronization fences to ensure the GPU is finished with a draw and that it is finished presenting the last image before we try to draw the next image. Via the nature of Vulkan, it is possible to start drawing to an unused image while the last image is still presenting, this can provide performance gains.

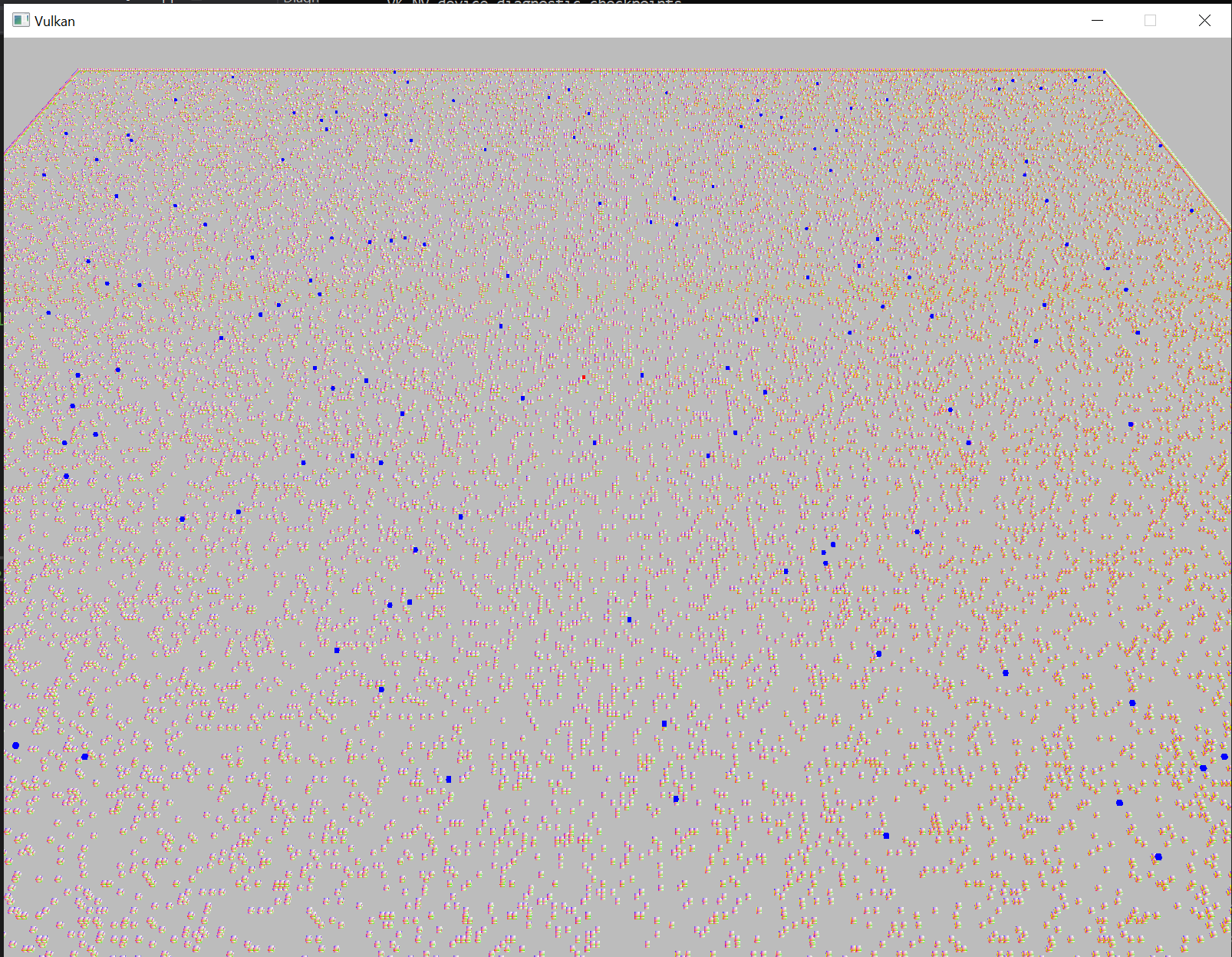
The user has the ability to move the camera forward, back, left and right with W,S,A,D. In addition to this the user has the ability to zoom in to the grid and back out with Q,E. Finally the user has the ability to change the perspective to be at an angled view seen in the example images with F,C. The user has the ability to scan over the whole grid of their chosen size and observe any specific agent moving along their path to ensure they are avoiding obstacles or zoom out and watch all agents move along their path together toward the goal.



*Somewhat far out image to show the scale of the grid and angled to demonstrate the 3D nature of the elements*

The user then can press the space key to activate the pathfinding, where all the agents that are drawn to the grid will have their paths computed on the GPU with the compute shader by submitting the compute command buffer specified at initialization. This shader begins a series of calls which prepares the dispatch using the Compute pipeline created in initialization. This pipeline is specified to avoid the rendering aspects of the GPU by avoiding vertex shader, rasterization and each unneeded stage that is required to bring an image to fruition. This allows faster and direct connection to the GPU.

The speed of the dispatch is measured by using the Standard Library’s Chrono time management include, starting a precise clock just before dispatch and using a fence that waits for the GPU to be idle. Once the GPU returns it is idle, the time is taken and compared. At this point, the blocks will begin to move at a slow speed from block position to block position along their path resolved by the compute shader. The speed of the moving blocks can then be increased and decreased with UP and DOWN keys. Once the blocks have reached their final location, the user can press R to return the positions to their starting places and pressing Space will begin the Dispatch again.



*200 AI agents pathfinding on 250,000 grid of nodes zoomed out*

# References

|  |  |  |
| --- | --- | --- |
| **Referenced Publication** | **Citation** | **Reference** |
| Website | (Avi 2008) | Avi .B (2008), GPU Accelerated Pathfinding, NVIDIA Corporation  <https://folk.idi.ntnu.no/elster/tdt24/tdt24-f12/presentations/lars-espen-GPUAcceleratedPathfinding.pdf> (Date Accessed: 12 February 2021) |
| Website | (Giuseppe, 2012) | Giuseppe .G (2012) GPU Accelerated Multi-agent Path Planning based on Grid Space Decomposition, SciVerse  <https://core.ac.uk/download/pdf/82615384.pdf> (Date Accessed: 3 March 2021) |
| Website | (Anon, n.d | Anon . A (n.d) Dynamic Search on the GPU, PeopleCS  <https://people.cs.umass.edu/~fmgarcia/Papers/Dynamic%20Search%20on%20the%20GPU.pdf> (Date Accessed: 12 April 2021) |
| Book |  | Akeine-Moller A. Haines E. Hoffman N. Pesce A. Iwanicki M. Hillaire S. (2018), Fourth Edition, *Real Time Rendering.*  CRC Press |
|  |  |  |

# Appendices