CGP3011M
Game Engine Architectures
Assessment 1

Ryan Docherty

Part A: Log File System

In this section, I will describe how the design of the code interface meets the requirements stated in the workshop.

1. Two files named “altlog.c” and “altlog.h” have been implemented. The .h file contains various components, such as the method identifiers. The .c file contains most of the code related to the function of the log system. These files make it very easy to keep track of data.
2. There are two methods which allow for the initialising and de-initialisation of logging, named ‘beginLog’ and ‘stopLog’ respectively. They simply open and close the text file. These are beneficial to quickly stop and start the system.
3. The log system has implemented variadic arguments which allow the logger method to pass in different types of arguments.
4. The system creates a file named ‘altlog.txt’, which is used to store logged data.
5. Instead of logging data to the file, there is an option to log to the console instead.
6. For this requirement, I have used enums, which allow the logged data to be tagged easily. They have proven to be beneficial, and have been used to facilitate the tagging of ‘Performance Info’, ‘Debug Info’, ‘Error Info’ and ‘Position Info’. However, these tags cannot be filtered at run-time. I decided against this because I feel that the programmer would know the type of data they wish to tag before run-time.
7. The programmer can switch off the logging system by selecting the option when prompted. This is a useful feature as it doesn’t affect performance.
8. I believe that this system does not impact the computational overhead of the Quake engine, so I didn’t put a significant emphasis on minimising its impact.
9. The interface that has been implemented is a simple input interface using the command prompt. I have used messages which appropriately guide the user on which options to choose depending on which data they wish to log. I believe that it is usable and clear.

Part B: Gameplay Analytics Tool

For my analytics tool, I have implemented seven visualisations altogether. The visualisations include individual trajectories of four players, an overlay of all four trajectories, a heatmap of player deaths and then an overlay of the heatmap onto players' trajectories. The data was gathered from three different text files or coordinates, and each file contains over five minutes of gameplay data.

The visualisations were implemented using OpenGL, GLFW and GLEW libraries. These helped dramatically with the basic maths and graphics functionalities which were required in order to achieve the visualisations. OpenGL provides vertex and fragment shading functionality, GLFW helped with the creation of windows and contexts, and GLEW helped with the runtime mechanisms. I also used the GLM header, which provides very useful mathematical support for OpenGL. I chose these libraries and headers as they have direct compatibility with OpenGL, and drastically make programming in C/C++ easier and more user-friendly.

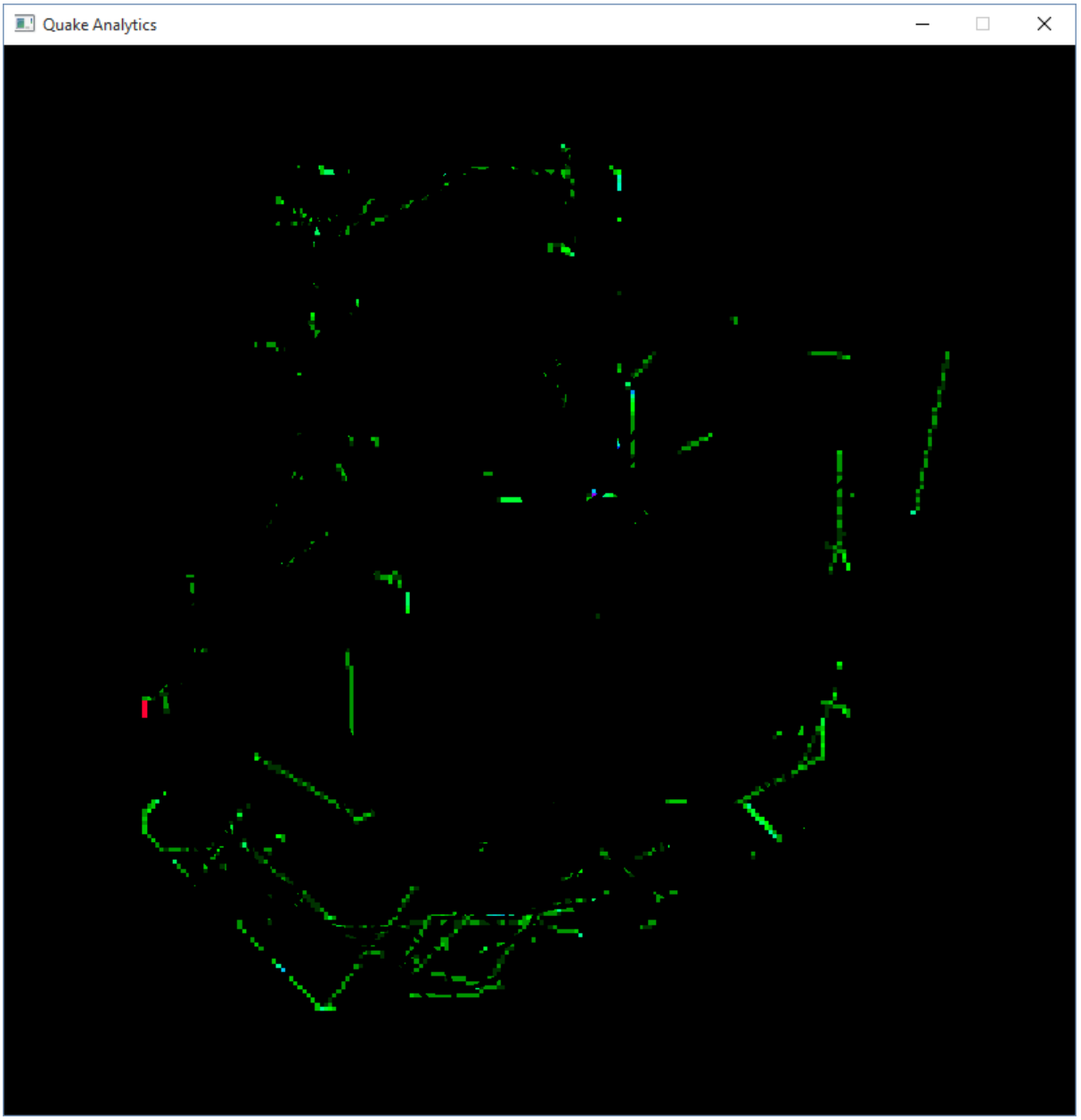
My technique of creating the heat-map initially involved creating an algorithm that splits the screen into 400 by 400 squares. The reasoning behind this is that in order to draw the squares onto the screen, it needs to be set up like a grid. I then created an algorithm that searched through the player positions file and gathered the coordinates of the location where players died. The difficult part was then creating an algorithm to select a colour based on the frequency of deaths at any given point. The colour would get progressively more red as the frequency of deaths increased, which would give it its heat-map look. I didn't manage to implement this feature completely the way I hoped for, as I was hoping to gather more information of death coordinates to produce a much more substantial heat-map. See appendix one to view the finished product.

To create the trajectory mapping, player positions were gathered from several games of Quake 3, which were then stored in text files. I believe that text files are suitable for this program, as they are easy to create and have lots of methods of accessing from the C++ language. I then created several algorithms which search through the text files and gather relevant information regarding the player number and their corresponding x and y coordinates at each frame. These coordinates were then stored in an array for easy access. I then created an algorithm that loops through this array and spawns points which correspond to the number of x and y coordinates, then the points are spawned in their actual position on the screen, subsequently creating a trajectory. I did this for each of the four players. I then assigned a unique colour to the player trajectories so that they would be easy to distinguish from one another. Appendix two, three, four and five show each player's' colour coded trajectory, appendix six shows every trajectory.

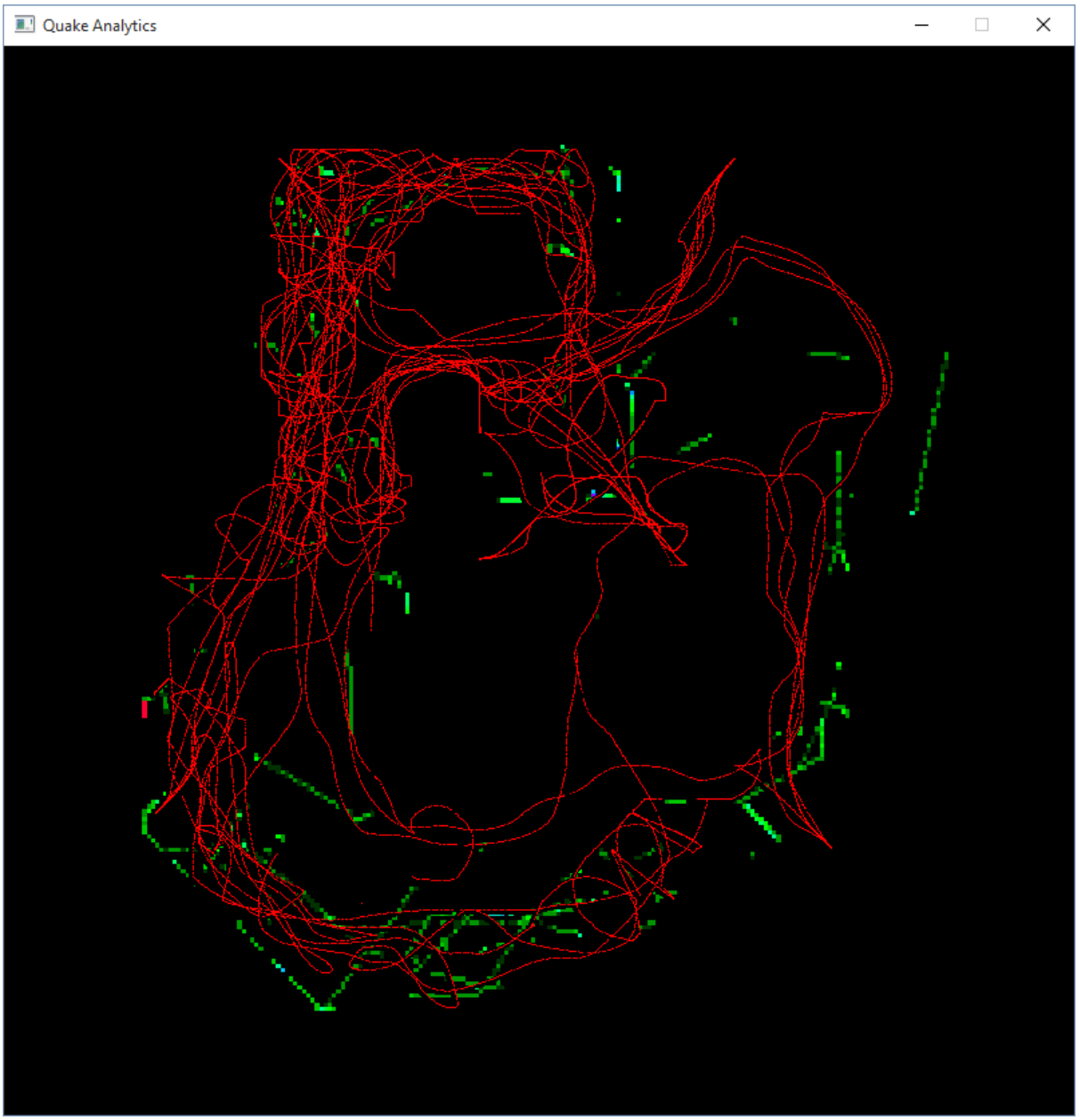
The next feature that I implemented was the ability to pan and zoom. This feature firstly involved setting up a camera using a model and a view. These are defined in the vertex shader and are multiplied together to give a position. I then created three vec3 definitions for a camera position, camera front and camera up. In my main game loop, I then assigned the view a value, which was a ‘lookAt’ of the resulting addition of camera position, camera front and camera up. I then defined a projection in respect to the window size, and set the perspective to a high number, so that the trajectories can be in full view of the window.

The next thing was to set up a facility to control the camera, which simply involved creating two key press methods. The first method would check to see if a key has been pressed or released, setting a boolean value to true or false. The reason behind this method is that the program is running too fast to detect a normal key press, so a key release is needed. After this, it was just a matter of telling the program which keys correspond to which function. I used a simple switch statement to decide on which trajectories to draw, and set the case in a ‘do movement’ method. I believe this was the simplest way to implement this feature. The documentation for how to control the visualisations is in the readme.txt file in the Visual Studio folder . See appendix seven and eight for these features.

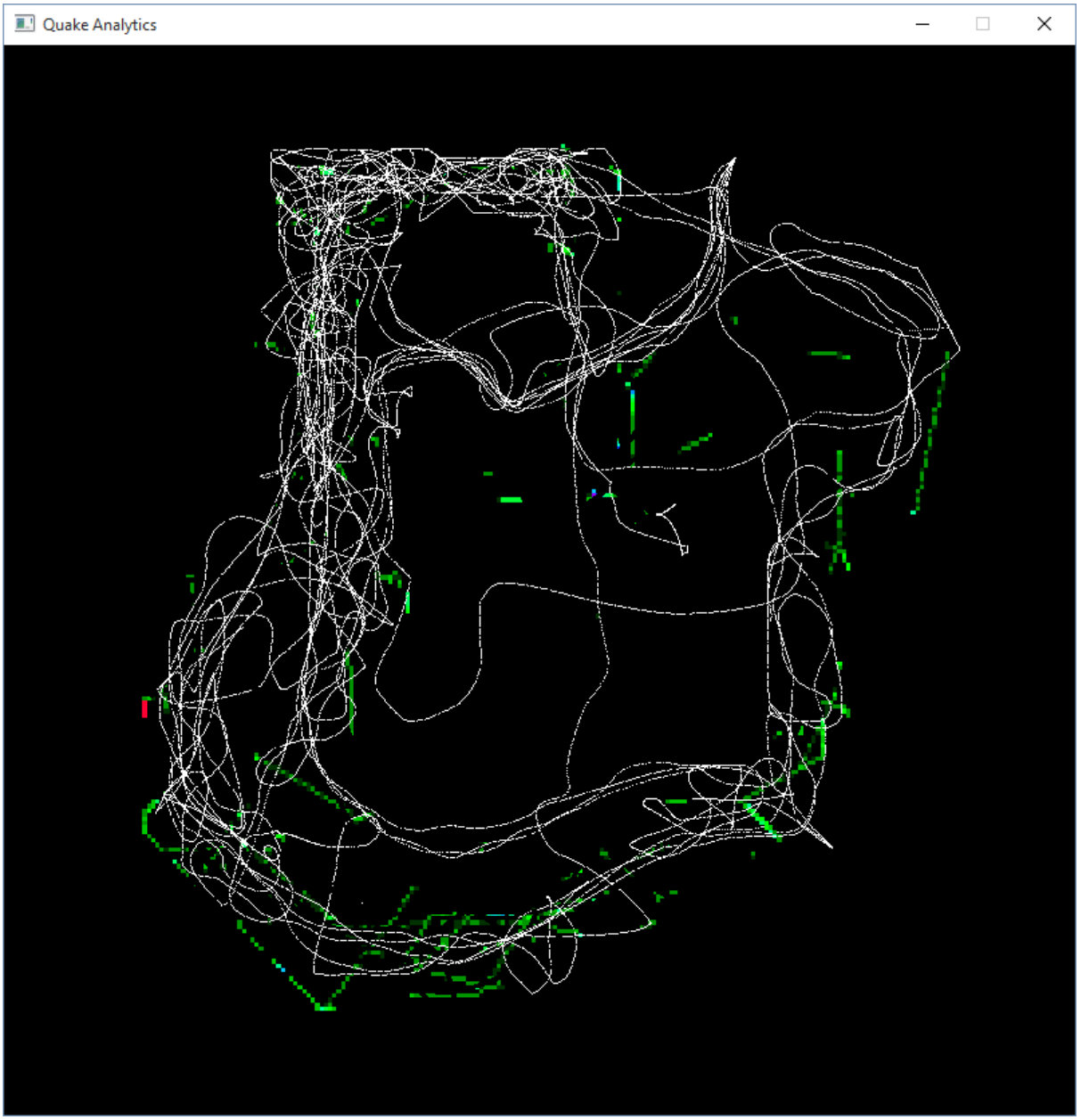
Appendix



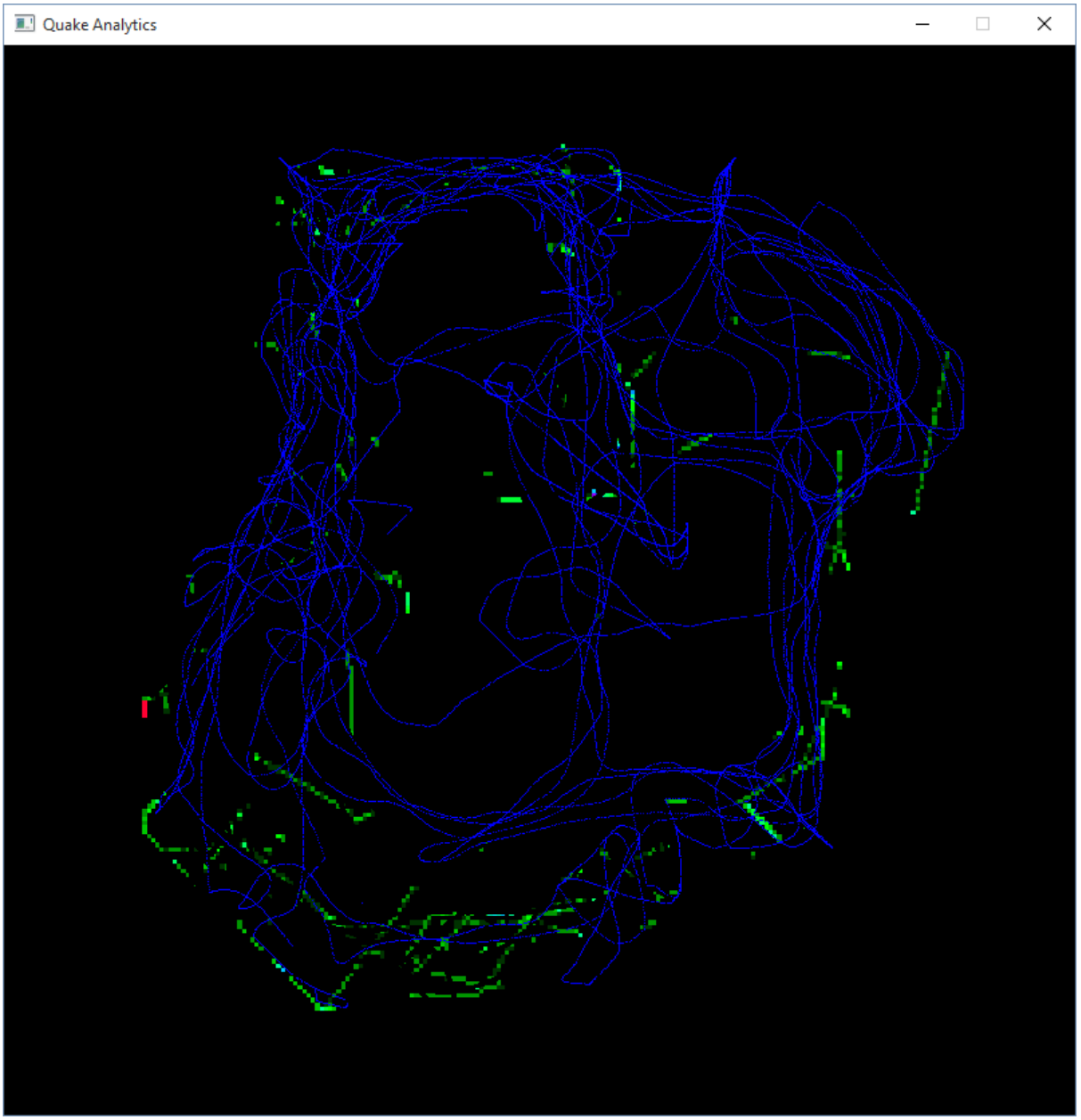
1. Death Heat-Map



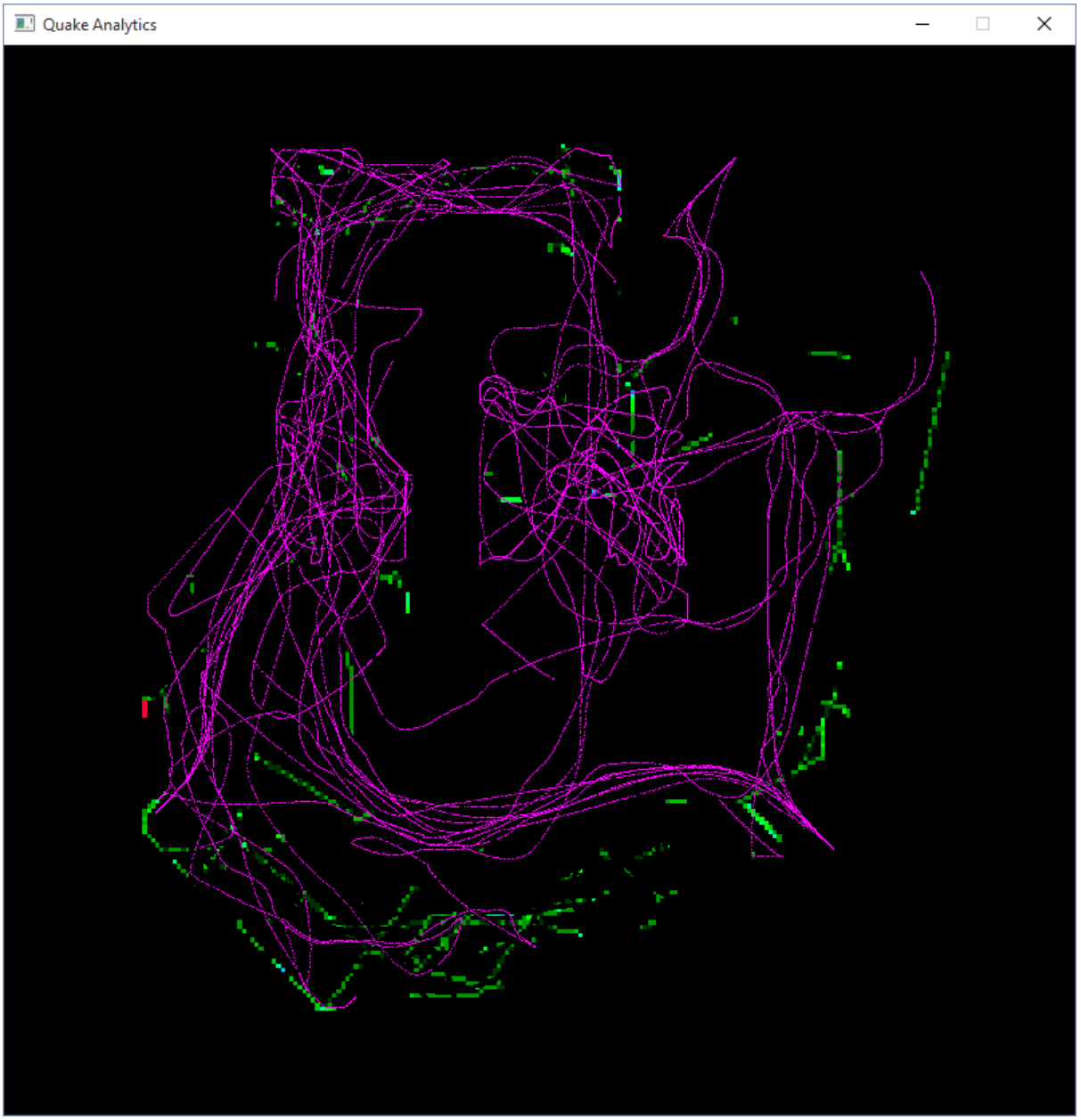
2. Player 1 Trajectory



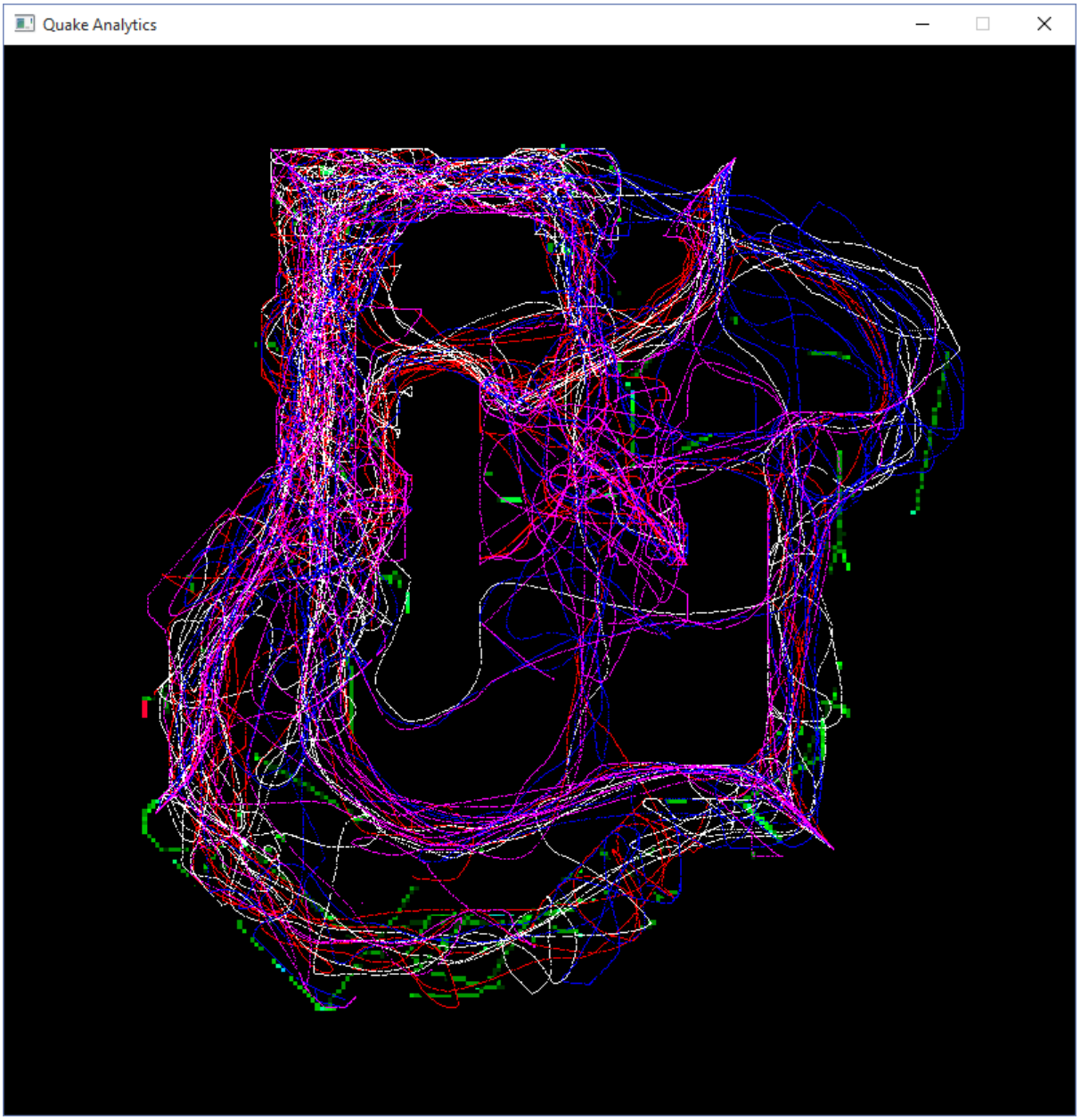
3. Player 2 Trajectory



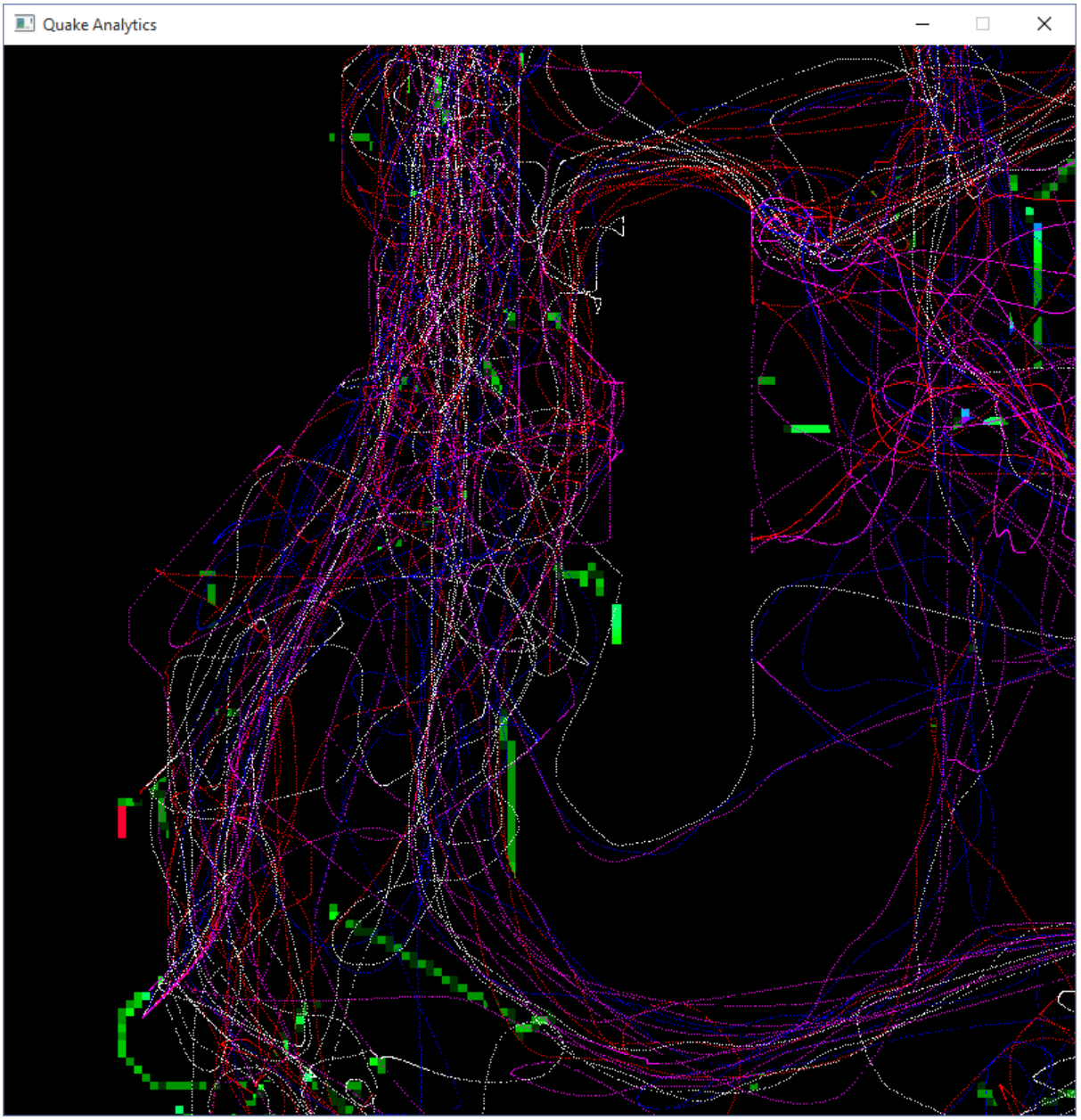
4. Player 3 Trajectory



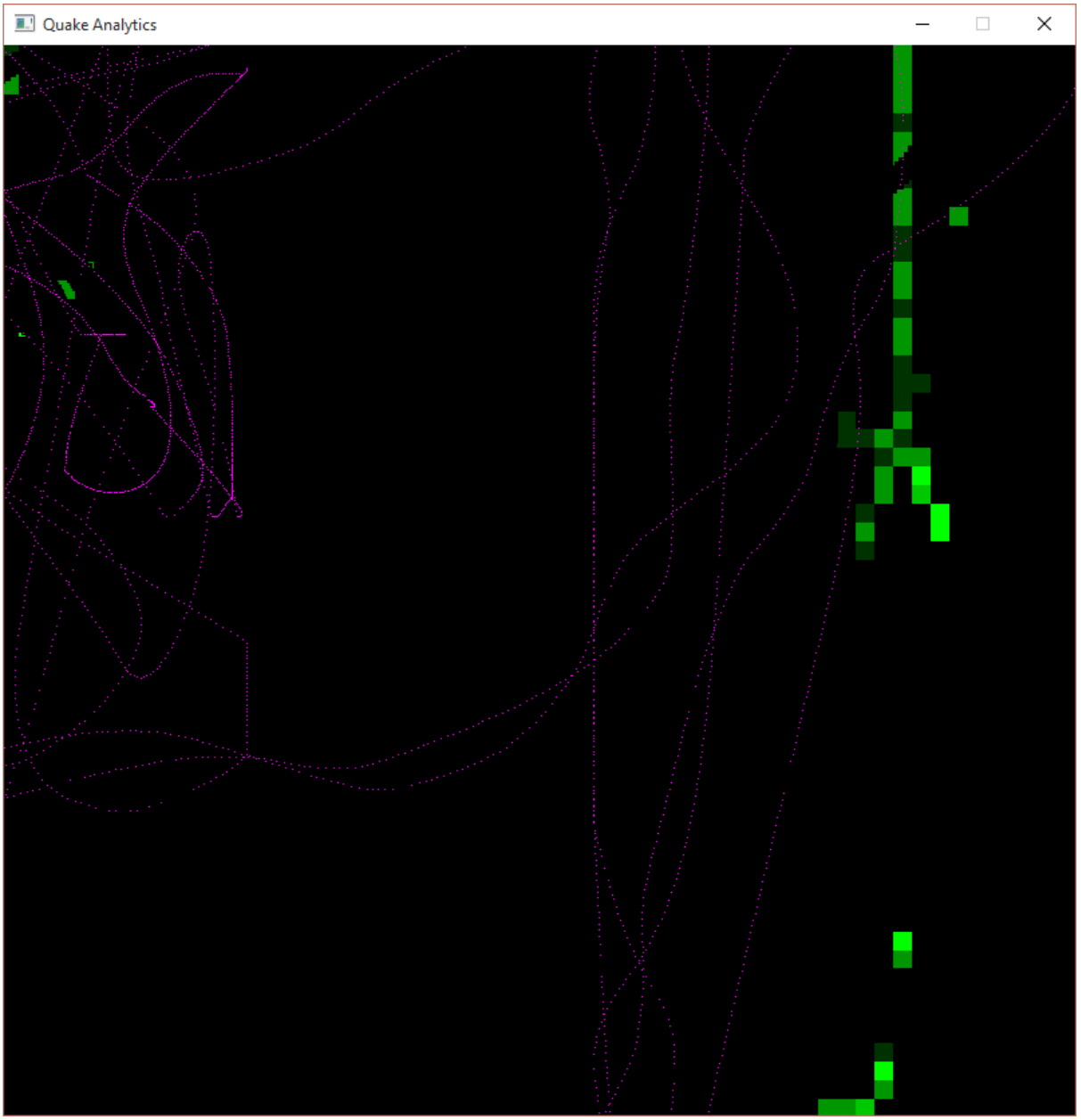
5. Player 4 Trajectory



6. Every Player Trajectory



7. Pan and Zoom



8. Pan and Zoom

Altlog Code: .h file

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#ifndef __altlog_h__
#define __altlog_h__

#define none            0
#define txtfile         1
#define console         2
#define both            3



//enums will help with tagging
typedef enum
{

        debugInfo = 0,
        errorInfo = 1,
        PerformanceInfo = 2,
        positionInfo = 3

} LogMode;
LogMode logInfo;

//methods
void selection();
void beginLog();
void endLog();
void logger(char *message);
void loggerConsole(char *message);
int logCatagory;

#endif
```

Altlog Code: .c file

```
#include "altlog.h"
#include <stdarg.h>

//set up a file
FILE *fp = NULL;

//Begin log and set up file
void beginLog()
{
#ifdef _DEBUG
	fp = fopen("altlog.txt", "a");

	//error check
	if (fp == NULL)
	{
		printf("Error: Couldn't open file\n");
		exit(1);
	}
#endif
}

void stopLog()
{
	fclose(fp);
}

void selection()
{
#ifdef _DEBUG

	int userInput;

	//Allow user to select options
	printf("AltLog Initiated\n");
	printf("Press 0 to turn logging off\n");
	printf("Press 1 to log to file");
	printf("Press 2 to log to console\n");
	printf("Press 3 to log to both\n");

	//basic error checking
	//doesn't check for none-numeric values however
	userInput = scanf("%d", userInput);
	if (userInput < 0 || userInput > 3)
	{
		printf("Invalid Input");
	}

	else
	{
		printf("%d,", userInput);
```

```
		}

		logCatagory = userInput;

#endif

}

void logger(LogMode logLevel, const char *message, ...)
{

#ifdef _DEBUG //this will stop logging while in release

		va_list args;
		va_start(args, message); //start variadic arguments
		const char *text = "";

		switch (logLevel)
		{
		case 0: //all
			if (logCatagory == console || both)
			{
				printf("%s", "(All)");
				if (logCatagory == txtfile || both)
				{
					vfprintf(fp, message, args);
				}

		case 1: //performanceInfo
			if (logCatagory == console || both)
			{
				printf("%s", "(Performance)");
				if (logCatagory == txtfile || both)
				{
					vfprintf(fp, message, args);
				}
			}

		case 2: //debugInfo
			if (logCatagory == console || both)
			{
				printf("%s", "(Debug)");
				if (logCatagory == txtfile || both)
				{
					vfprintf(fp, message, args);
				}
			}

		case 3: //errorInfo
			if (logCatagory == console || both)
			{
				printf("%s", "(Error)");
```

```
                        if (logCatagory == txtfile || both)
                        {
                                vfprintf(fp, message, args);

                        }
                }
                }
                va_end(args); //end of variadic arguments
        }
#endif
};



void loggerConsole(char *message)
{
        printf(message);
}
```

Bibliography

1. "Learn OpenGL, Extensive Tutorial Resource for Learning Modern OpenGL." *Learn OpenGL, Extensive Tutorial Resource for Learning Modern OpenGL*. N.p., n.d. Web. 15 Dec. 2015.