

Gary Sangwell

SAN12366949

CGP3011M

Game Engine Architectures

Assessment 1

Table of Contents

Part A.....	3
Introduction	3
Requirements.....	3
Self-contained Source, Altlog.txt	3
Functions, Variadic Arguments, Log Levels.....	4
Overhead, Release Builds.....	5
Useable Interface	5
Additional Features.....	5
Part B.....	6
Introduction	6
Requirements.....	6
Heatmap.....	6
Trajectory Mapping.....	7
Death Heatmap.....	7
Combined Mode	7
Play Mode	7
UI.....	8
Controls.....	8
Possible Extensions	8
Appendix	9
Part A.....	9
Part B.....	12

Part A

Introduction

For the first part of the assignment, the task was to create an alternative logging system for the Q3A Engine, with some new and potentially useful features. A list of requirements for the logging system was given, and the solution presented here successfully implements all of the required features, as well as some additional features that may prove useful.

Requirements

1. It should be implemented in two new files named "altlog.c" and "altlog.h".
2. As a minimum, it should have functions to initialize and de-initialise the logging, and a function analogous to Com_Printf() which can be called from any .c code file at any point.
3. The logging function should follow the C standard for passing variadic arguments (just like Com_Printf).
4. Logged data should be stored in a text file "altlog.txt".
5. There also should be an option to direct the logged data to the console rather than the file, if needed.
6. There should be a facility to tag logged data in some way (eg "Debug Info", "Errors", "Performance data", etc), and to filter it at run time, so that the programmer can switch off and on different tags. For example, we might only want to log "Performance Data" on a specific test run of the game.
7. The programmer should be able to switch off all logging in such a way that programme performance is not affected (for a release build).
8. You should give general consideration to computational overhead, and minimize the impact of the logging system as much as possible.
9. This system will be used by other team members, so you should design a usable and clear interface.

Self-contained Source, Altlog.txt

The first requirement given was to implement the solution in two new files, name "altlog.h" and "altlog.c". All of the code and functionality required for the logging system is present within these files, as shown in appendix 4-5, and there are no dependencies on any of the Q3A source files. This allows the system to be portable, and therefore easily moved to another game with minimal, if any, changes.

The system logs all messages to a txt file named "altlog.txt", which was the fourth requirement given for the solution. A further extension to this solution that could prove valuable would be to allow the initialise, log message and de-initialise functions to all take a log file name, rather than use an internally hard-coded name. This would have allowed multiple different log files to be opened and written to within the same program; for example, error messages could have been written to one log, whilst performance messages were written to another. However, this was left out in order to meet the requirement given.

Functions, Variadic Arguments, Log Levels

The second requirement was for 3 functions to be present at minimum; one for initialising the logging system, one for de-initialising the logging system, and a third for logging messages, analogous to the `Com_Printf()` function already available in Q3A. In this implementation, all of these functions were implemented, along with additional functions to provide more functionality.

The initialise function opens the log file in append mode, appending any new log entries to the existing file (if present), rather than over writing it. It also prints a message to the log file to show the time the logging session was started; this is useful to help identify different logging sessions, especially if multiple are appended to a single file. The decision to use append mode rather than overwriting the log file was made to ensure that if the program was accidentally re-run after a logging session, the log data would not be deleted. If a programmer wishes to log into an empty file, the log file can be manually deleted and the solution will automatically recreate it the next time it is ran.

The de-initialise function closes the log file, in which any store writes are flushed and written to the file. The function for logging messages takes multiple parameters, including variadic arguments following the C standard as `Com_Printf()` does, which was a third requirement for the logging system. The signature for this function is shown in appendix 1; the first parameter takes the log level to tag the message with, the second parameter allows a programmer to direct the log message to the console, and the third parameter allows the user to pass in any number of arguments for logging. The decision was made to use Variadic arguments, not only to meet the requirements given, but also to provide greater functionality to a programmer.

The use of log levels allows a programmer to tag different log messages with different tags, including the following: `LOG_ERROR`, `LOG_DEBUG` and `LOG_PERFORMANCE`. These tags are shown in the log file, providing an easy way to identify which category a message belongs to when reviewing a log file manually through a text editor. An example of this is shown in appendix 2. In addition to this, these tags can be filtered at run time, so that only messages with a tag matching the set level are logged, and the rest are discarded. This would be useful when different testers require different information from log files; the messages can be filtered by simply changing one line of code, rather than removing any unneeded log calls. When the logging system is first initialised, the log level is set via a function parameter, and tags can be combined to allow multiple tags to be logged. An function also exists to allow a programmer to change what level messages are written to the log file and console window at run-time, and levels can again be combined together to allow more than one type of data to be logged at once.

This requirement was implemented through the use of enumerations and bit values, which allow the tags to be combined together, and therefore multiple tags can be logged at the same time. This could have been achieved through the use of a switch statement and an integer as the log level, but the decision was made to use an enumeration and bit values to make multiple tags possible, as well as to make code written using the log system clearer; an enumeration name is much clearer to understand than a simple integer.

Overhead, Release Builds

Another set of requirements for the logging system was to both minimise the impact the solution had on the performance of Q3A, as well as to provide a method to completely turn off logging if needed, for example when building release builds. This was achieved through the use of the `_DEBUG` define, which is only defined if the solution is built in debug mode; this is used with the `#if` macro within each function, and the compiler removes the code inside each function if not built in release mode. It was originally considered to use a macro function to wrap the C function, but the choice was made to use the `#if` macro instead to minimise the complexity of the solution, and to make the interface more clear to any programmers using the solution. Whilst this solution still leaves the function calls when built in release mode, any modern compiler should automatically optimise these empty function calls out, resulting in no additional computational overhead.

Useable Interface

A clear and useable was the last requirement for the logging system, so that other team members could easily use the system. This was achieved through the use of good coding standards, including relevant function and parameter names. The function and parameter names chosen clearly indicate both what the function does, as well as the data it is expecting. In addition to this, functions are clearly commented in a templated format, detailing exactly what a function does, what it returns and what parameters it expects. The solution also only exposes the minimal amount of functions needed, in order to decrease complexity for programmers.

Additional Features

In addition to the given requirements, some extra features were added that should prove useful to a programmer using the logging system. One extra feature implemented was to allow the user to flush the pending writes to the log file, allowing writes to be made before the de-initialise function is called. This is useful to avoid log data being lost if the program execution crashes before the log file is closed. For example, this could be called once a frame if use within a game environment, resulting in only a single frame of log data being lost if the program terminates before the writes were flushed through closing the log file.

A second additional feature that was added was the use of time-stamps within the log messages; every message that is logged is prefixed with the time the message was logged. This allows for a programmer to view the exact time during the programs execution when a message was logged; this could be useful when trying to debug errors within a game. This can also be seen in appendix 2.

A third additional feature that is provided is the use of colour when log messages are directed to the console. The different levels of log messages are shown in different colours; for example, errors are shown in red, whereas debug messages are shown in yellow. This can be seen in appendix 3. This could prove useful to a programmer using the logging system, as it allows for different types of log messages to be easily identified within the console window; for example, any errors that are logged will be immediately apparent as they are clearly visible in red font.

Part B

Introduction

For the second part of the assignment, the task was to build a standalone tool that could visualise log files generated from Q3A game-play. The tool could either read files generated from the logging system in Part A, or files generated by the Q3A engine. A list of requirements for the tool was given, including graphic visualisation of trajectories and heatmaps, as stated below. Most of the requirements were implemented, along with some additional features.

Requirements

1. It should be a standalone tool – it should be able to read Q3A log files, or the log files you wrote to in part A.
2. The log file(s) should be specified as command line parameters, with sensible defaults. You do not need to build a file opening UI.
3. The tool should have a “heatmap” mode where a heatmap of player positions should be displayed (https://en.wikipedia.org/wiki/Heat_map). The user should be able to explore the heatmap by zooming and panning
4. There should be a trajectory visualisation mode that shows the trajectories players take through the world.
 - a. The trajectory visualisation should be able to (on a mode switch) also visually illustrate the speed of the player along the trajectory
 - b. The trajectory visualisation should be able to be shown on top of the heat map (i.e. one, the other, or both)
 - c. Trajectories of different players should be visually distinct from each other.
5. For both modes, the user should be able to cycle through showing data for all players, player 1, player 2, player X, and back to all players.
6. There should be a sliding window feature that only shows the data for a section of the in-game time, and that can be paused and played (with key presses) to move through in-game time.
7. On-screen indication of the mode/state of the visualisation
8. The tool should function for game logs of at least 5 minutes of game play with at least 4 players. You should discuss this in your report.
9. The visualisation should be of a quality such that users can understand the activities of players that are in the log files.
10. Documentation for how to control the visualisation.

Heatmap

One of the first requirements for this solution was the visualisation of player position through a heatmap, which can be zoomed and panned around; this can be seen in appendix 6. In order to achieve this visualisation, the player position data collected from the Q3E log files generated in part A was used. As this data was very dense, early attempts resulted in a poor heatmap; due to the high precision of the stored position, positions were often only ever visited once. This was solved by down sampling the data, and instead using block areas and summing the total points inside the block. Whilst this resulted in a blockier heatmap, it meant that colour could be used to much better effect as a visual aid for player position.

Trajectory Mapping

Another requirement for the game analytic tool was visualisation of player trajectory, including visualisation of player speed. This was successfully implemented through the use of the logged player positions. The OpenGL line strip drawing mode was used in order to draw lines from the given positions, as can be seen in Appendix 7-10. The visualisation can cycle through the different players, the same as the heatmap mode, as well as show all of the player's trajectories on a single screen. This is shown in Appendix 7-11. The visualisation of player speed can also be toggled on and off, and is achieved through the use of line colour intensity; the colour is more intense in areas players were travelling fast, and less pronounced in areas the player was travelling slowing. Appendix 12-16 shows how this is visualised. One limitation of the current method of visualising player trajectory is the line jumps, shown as long straight lines, in which a player dies and then moves to a new spawn point. This could have been solved by filtering for player health, and omitting drawing any positions where the player was not alive.

Death Heatmap

In addition to the player position heatmap, a death heatmap mode was also created in order to visualise how player deaths were distributed throughout the map. This was implemented for all 4 players, as well as a combined map that shows the distribution of all player deaths within the game match; this is shown in Appendix 17-21. This feature was added to provide a user with a way of visualising player deaths from game data, which could be useful when analysing game play tactics. This was implemented by checking the player's health in the logged data, and making a note of their position as soon as their health dropped below 0. These points were then drawn onto the screen as translucent squares, which when overlaid provide a stronger colour. Which quite a primitive method of drawing heat maps, this method worked well enough to show the distribution of player deaths throughout the map.

Combined Mode

One of the requirements given was for the tool to be able to overlay player trajectory on top of the heatmap. This mode was implemented but using the death heatmap, rather than the positional heatmap. This decision was made as the trajectory map was almost unnoticeable when overlaid onto the positional heatmap. In addition to this, the combined death and trajectory maps allow a user to see how players have moved about the map, at the same time as seeing where their deaths occurred, which could be useful for many types of game play analysis. This was achieved by first drawing the death map, and then drawing the trajectory map over the top of it. This mode was also implemented for all 4 players, as well as a combined screen showing all the players combined deaths and trajectories, as can be seen in Appendix 22-26.

Play Mode

The sixth requirement given for the game analytic tool was a sliding window feature that allows a section of the game play data to be selected and played. This was almost fully implemented, and is available in both Death Map mode as well as Player Trajectory mode. The player can adjust the speed of the playback through the use of keyboard keys, as well as the window size and position. Playback is only played when the user is holding a keyboard key. Different players can be cycled through during playback, meaning the user can quickly switch between different player data and the

combined screen if needed. Appendix 27-28 show this mode being used in both the Trajectory and Death Heatmap modes.

UI

An onscreen UI in the top left corner shows the current mode the visualisation tool is current in, as well as what player data is being shown on screen. In addition, when in play mode, one extra feature added is the display of additional information, such as current game time, playback speed, start and end times, in the top right. This UI provides the user with additional information about the visualisation, and helps the user control the tool more easily.

Controls

Zoom: Mouse scroll wheel.

Panning: Hold the middle mouse button + mouse movement.

Mode selection: Keyboard keys 1 – 4.

Speed visualisation: Keyboard key 0.

Player cycling: Left or right mouse button.

Enable playback: P key.

Play playback mode: Hold enter/return key.

Adjust playback speed: Pageup / Pagedown keys.

Adjust playback window: Up and down keys.

Possible Extensions

One additional feature that could be implemented to provide further features to users would be visualisation of player kills. Whilst player deaths can be visualised through the death map mode, it is hard to determine which player killed whom. By logging player kills as well, further information could be portrayed to the user. This would be especially useful when using the tool to analyse game play as a team, and for visualising exactly when, where and by whom a player was killed.

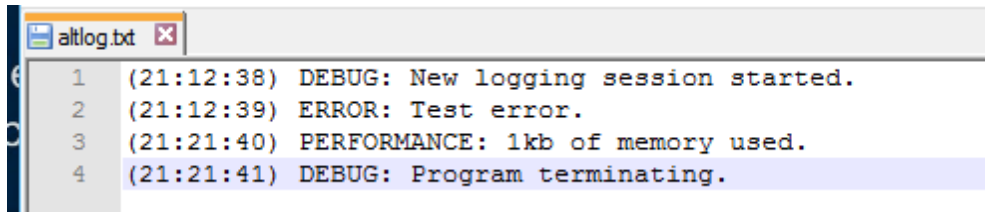
Another feature that could have been implemented would be visualisation of player pickups. This would allow the user to visualise how the player travelled through the map, what pickups they picked up and how this led to interactions with other players. This again could be useful for analysing game play, as it would allow for the user to see exactly where and when a player picked up what pickup.

Appendix

Part A

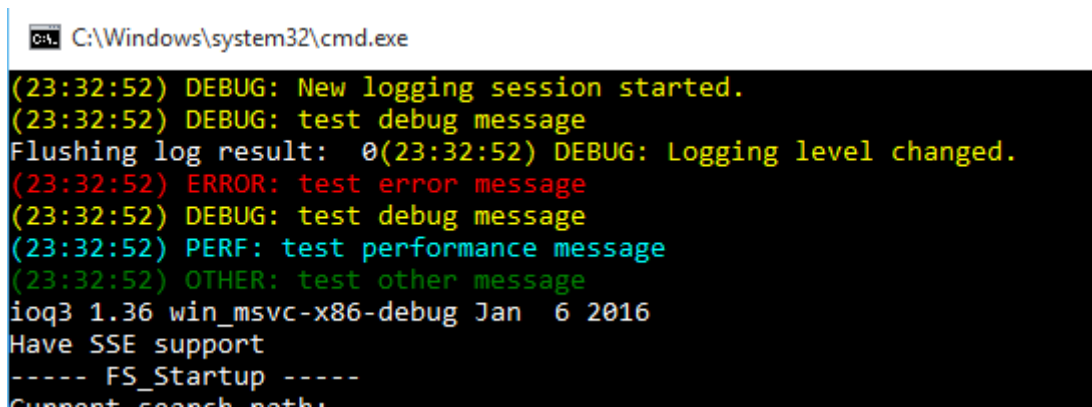
```
void logMessage(const t_logLevel level, bool consoleLog, char *format, ...)
```

Appendix 1: The signature for the logMessage() function, showing the parameters the function takes.



```
altlog.txt
1 (21:12:38) DEBUG: New logging session started.
2 (21:12:39) ERROR: Test error.
3 (21:21:40) PERFORMANCE: 1kb of memory used.
4 (21:21:41) DEBUG: Program terminating.
```

Appendix 2: Messages written to the log file are automatically prefixed with their tag and the time the message was logged, useful when reviewing log files at a later date.



```
C:\Windows\system32\cmd.exe
(23:32:52) DEBUG: New logging session started.
(23:32:52) DEBUG: test debug message
Flushing log result: 0(23:32:52) DEBUG: Logging level changed.
(23:32:52) ERROR: test error message
(23:32:52) DEBUG: test debug message
(23:32:52) PERF: test performance message
(23:32:52) OTHER: test other message
ioq3 1.36 win_msvc-x86-debug Jan 6 2016
Have SSE support
----- FS_Startup -----
Current search path:
```

Appendix 3: Messages directed to the console are colour coded for easy identification.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  // Windows colour code defines.
6  #define COLOUR_BLACK 0
7  #define COLOUR_BLUE 1
8  #define COLOUR_GREEN 2
9  #define COLOUR_CYAN 3
10 #define COLOUR_RED 4
11 #define COLOUR_MAGENTA 5
12 #define COLOUR_BROWN 6
13 #define COLOUR_LIGHTGRAY 7
14 #define COLOUR_DARKGRAY 8
15 #define COLOUR_LIGHTBLUE 9
16 #define COLOUR_LIGHTGREEN 10
17 #define COLOUR_LIGHTCYAN 11
18 #define COLOUR_LIGHTRED 12
19 #define COLOUR_LIGHTMAGENTA 13
20 #define COLOUR_YELLOW 14
21 #define COLOUR_WHITE 15
22
23 // Enumeration for log levels
24 #define enum {
25     LOG_ERROR = 0x01,
26     LOG_DEBUG = 0x02,
27     LOG_PERFORMANCE = 0x04,
28     LOG_OTHER = 0x08
29 } t_logLevel;
30
31 extern FILE *logFile;
32 extern t_logLevel loggingLevel;
33
34 /*
35 * Function:    startLogging
36 *
37 * Purpose:    Opens the log file for writing in append mode.
38 *
39 * Parameters:
40 *     level:    The initial level for messages to be logged.
41 *               Takes type t_logLevel; these can be combined to allow for multiple log levels at once.
42 *               For example, to log both error and debug messages: LOG_DEBUG | LOG_ERROR
43 *
44 * Returns:    void
45 */
46 extern void startLogging(const t_logLevel level);
47
48 /*
49 * Function:    logMessage
50 *
51 * Purpose:    Logs a message to the log file, and optionally the console.
52 *
53 * Parameters:
54 *     level:    The log level to tag the message with, for example: LOG_ERROR
55 *     consoleLog: Pass true to also direct message to the console window, false to only log to the log file.
56 *     *format:    Variadic parameter format.
57 *                 For example, to pass two strings: "%s %s"
58 *     ...        Additional parameters of any type can be passed, and are used based on the format parameter.
59 *
60 * Returns:    void
61 */
62 extern void logMessage(const t_logLevel level, bool consoleLog, char *format, ...);
63
64 /*
65 * Function:    flushLog
66 *
67 * Purpose:    Flushes pending writes to the log file.
68 *
69 * Parameters:
70 *     none
71 *
72 * Returns:    void
73 */
74 extern void flushLog();
75
76 /*
77 * Function:    changeLogLevel
78 *
79 * Purpose:    Changes the log level during runtime.
80 *
81 * Parameters:
82 *     level:    The new level for messages to be logged.
83 *               Takes type t_logLevel; these can be combined to allow for multiple log levels at once.
84 *               For example, to log both error and debug messages: LOG_DEBUG | LOG_ERROR
85 *
86 * Returns:    void
87 */
88 extern void changeLogLevel(const t_logLevel level);
89
90 /*
91 * Function:    endLogging
92 *
93 * Purpose:    Closes the log file, and flushes any pending writes to the log file.
94 *
95 * Parameters:
96 *     none
97 *
98 * Returns:    void
99 */
100 extern void endLogging();
101
102 /*
103 * Function:    getAltTime
104 *
105 * Purpose:    Gets the current time in a custom format.
106 *
107 * Parameters:
108 *     *string: The char pointer to populate with the current time string.
109 *
110 * Returns:    void
111 */
112 void getAltTime(char *string);
113
114 /*
115 * Function:    setColorAndBackground
116 *
117 * Purpose:    Sets the windows console foreground and background colour.
118 *
119 * Parameters:
120 *     ForgC: The foreground colour to set; use the defines COLOUR_RED etc.
121 *     BackC: The background colour to set; use the defines COLOUR_BLACK etc.
122 *
123 * Returns:    void
124 */
125 void setColorAndBackground(int ForgC, int BackC);

```

Appendix 4: altlog.h header file.

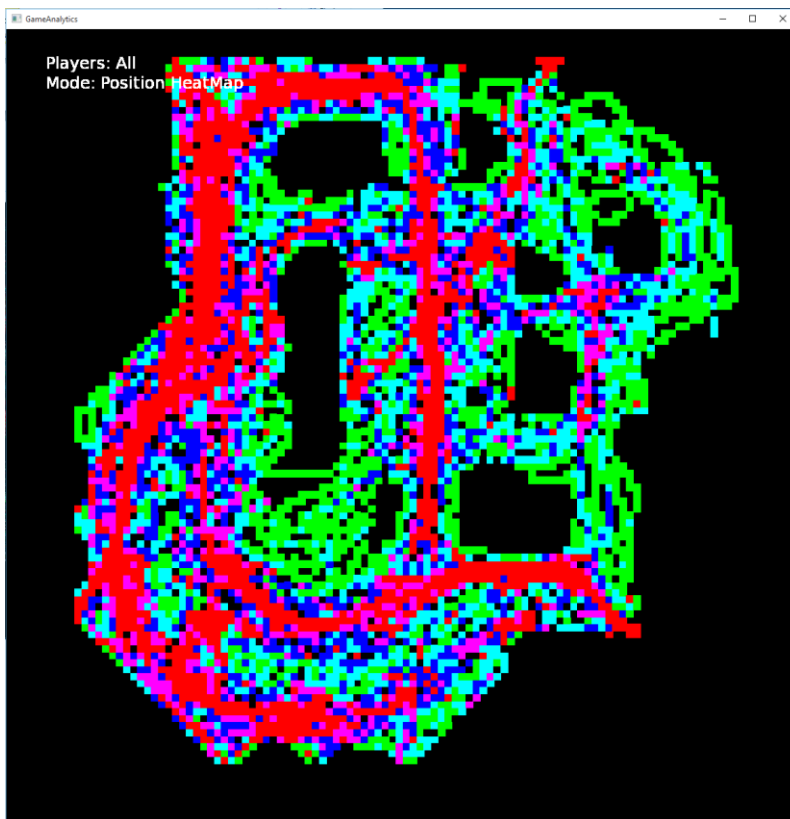
```

1  #include "altlog.h"
2  #include <time.h>
3  #include <stdarg.h>
4
5  #include <stdio.h>
6  #include <windows.h>
7
8  FILE *logfile = NULL;
9  t_logLevel loggingLevel;
10
11 void startLogging(const t_logLevel level)
12 {
13     #if _DEBUG
14
15         // Store logging level and open the log file
16         loggingLevel = level;
17         logfile = fopen("altlog.txt", "a");
18
19         // Information about the logging session
20         logMessage(LOG_DEBUG, "%s", "New logging session started.\n");
21
22     #endif
23 }
24
25 void logMessage(const t_logLevel level, bool consoleLog, char *format, ...)
26 {
27     #if _DEBUG
28
29         va_list args;
30
31         // Check if the message level flag is set to be logged
32         if(loggingLevel & level)
33         {
34             const char* prefix;
35
36             // Colour and prefixes based on message level
37             switch (level)
38             {
39                 case LOG_ERROR:
40                     setColorAndBackground(COLOUR_LIGHTRED, COLOUR_BLACK);
41                     prefix = "ERROR:";
42                     break;
43                 case LOG_DEBUG:
44                     setColorAndBackground(COLOUR_YELLOW, COLOUR_BLACK);
45                     prefix = "DEBUG:";
46                     break;
47                 case LOG_PERFORMANCE:
48                     setColorAndBackground(COLOUR_LIGHTCYAN, COLOUR_BLACK);
49                     prefix = "PERF:";
50                     break;
51                 default:
52                     setColorAndBackground(COLOUR_WHITE, COLOUR_BLACK);
53                     prefix = "OTHER:";
54                     break;
55             }
56
57             // Formatted time
58             char formattedTime[10];
59             getAltTime(formattedTime);
60
61             // Varadic arguments
62             va_start(args, format);
63
64             // Show log time and message level
65             fprintf(logfile, "(%) %s ", formattedTime, prefix);
66
67             // Print variadic argument using given format and argument list
68             vfprintf(logfile, format, args);
69
70             // Print to console?
71             if (consoleLog)
72             {
73                 // Prints to standard console, not quake console. Portability. Decoupled design.
74                 printf("(%) %s ", formattedTime, prefix);
75                 vprintf(format, args);
76
77                 setColorAndBackground(COLOUR_WHITE, COLOUR_BLACK);
78             }
79
80             prefix = NULL;
81
82             va_end(args);
83         }
84     #endif
85 }
86
87 void flushLog()
88 {
89     printf("%s %i", "Flushing log result: ", fflush(logfile));
90 }
91
92 void changeLogLevel(const t_logLevel level)
93 {
94     #if _DEBUG
95         loggingLevel = level;
96         logMessage(LOG_DEBUG, true, "%s", "Logging level changed.\n");
97     #endif
98 }
99
100 void endLogging()
101 {
102     #if _DEBUG
103         // Flush any pending writes
104         flushLog();
105
106         // Close log file
107         fclose(logfile);
108         logfile = NULL;
109     #endif
110 }
111
112 void getAltTime(char* string)
113 {
114     // Format time to a custom format for logging
115     time_t currentTime = time(NULL);
116     strftime(string, 10, "%X", localtime(&currentTime));
117 }
118
119 void setColorAndBackground(int ForgC, int BackC)
120 {
121     // Windows console colour
122     // Note: add #ifdef macro to support linux too.
123     WORD wColor = ((BackC & 0x0F) << 4) + (ForgC & 0x0F);
124     SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), wColor);
125     return;
126 }
127

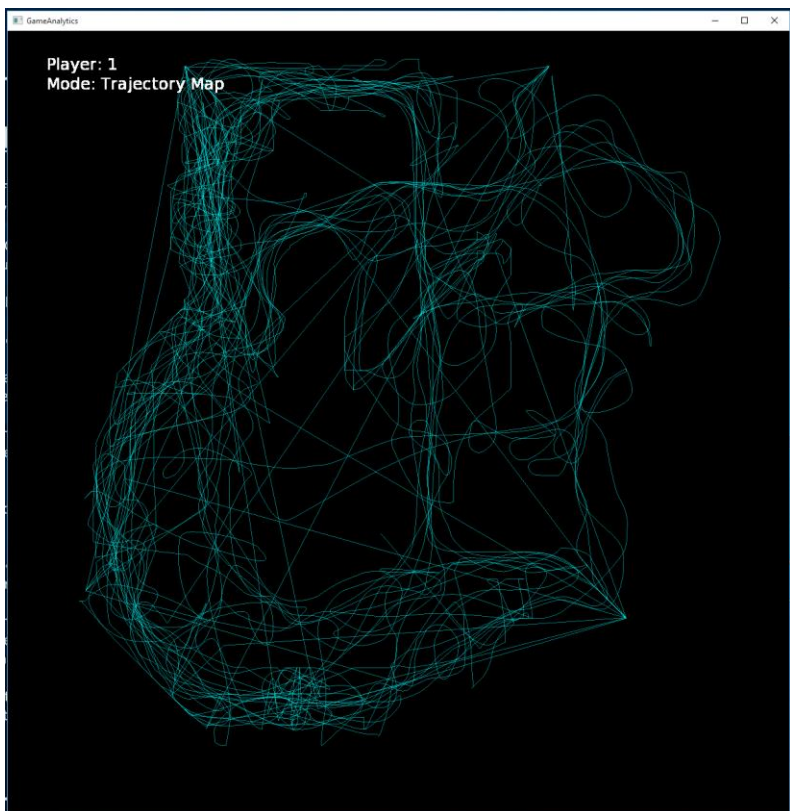
```

Appendix 5: altlog.c header file.

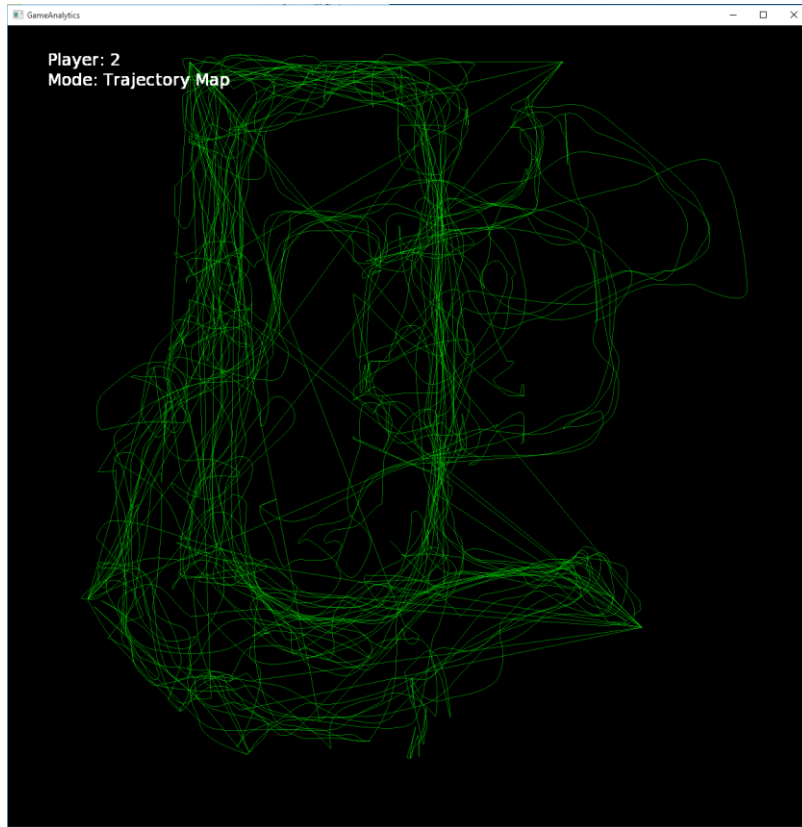
Part B



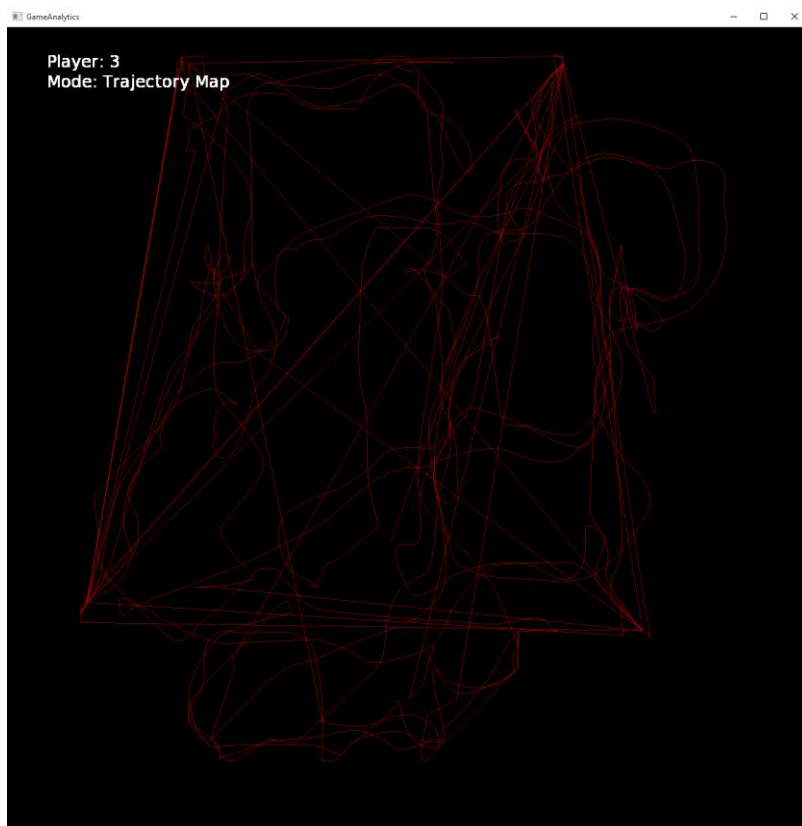
Appendix 6: Position Heatmap mode, showing a coloured heatmap of player position throughout the game.



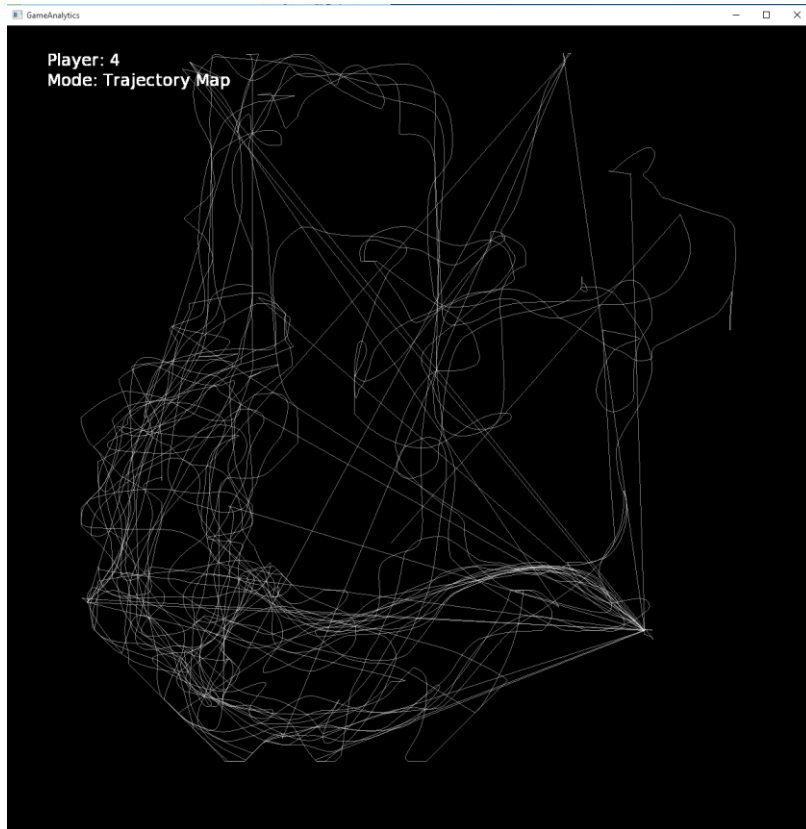
Appendix 7: Trajectory map for player 1.



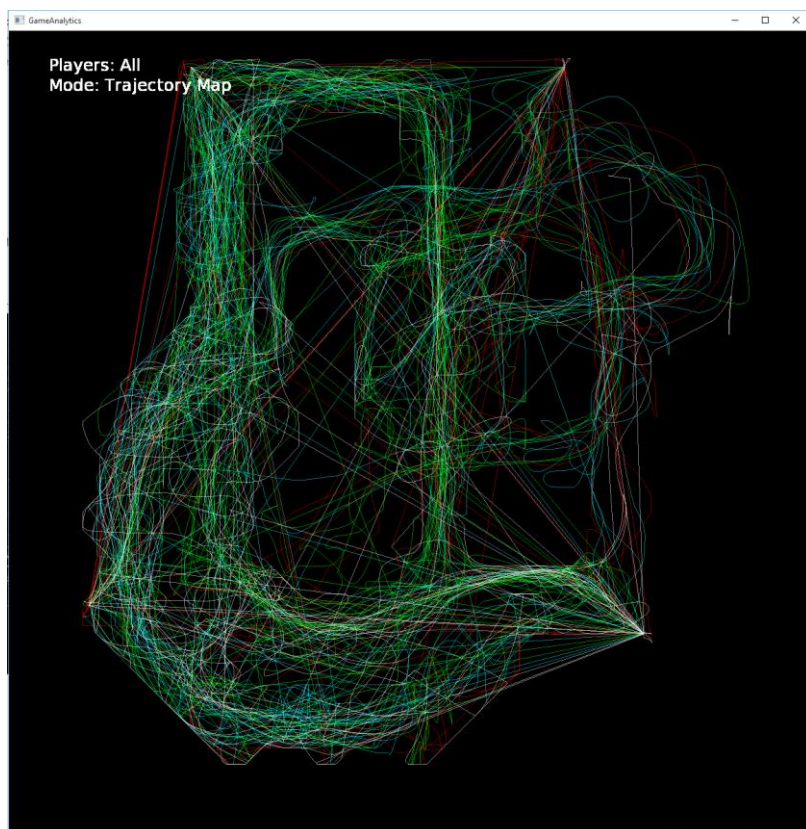
Appendix 8: Trajectory map for player 2.



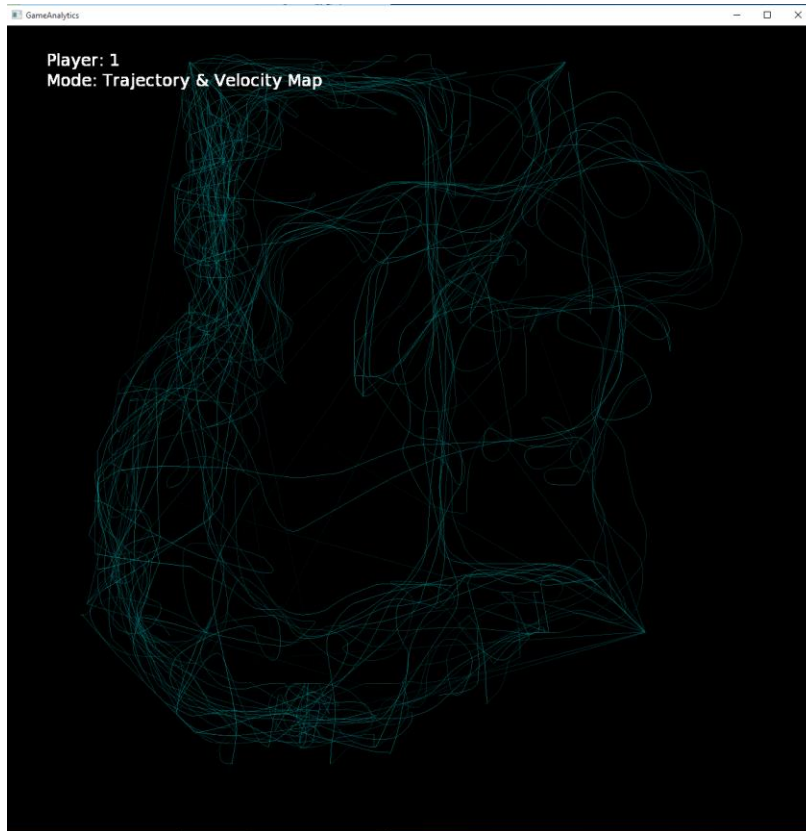
Appendix 9: Trajectory map for player 3.



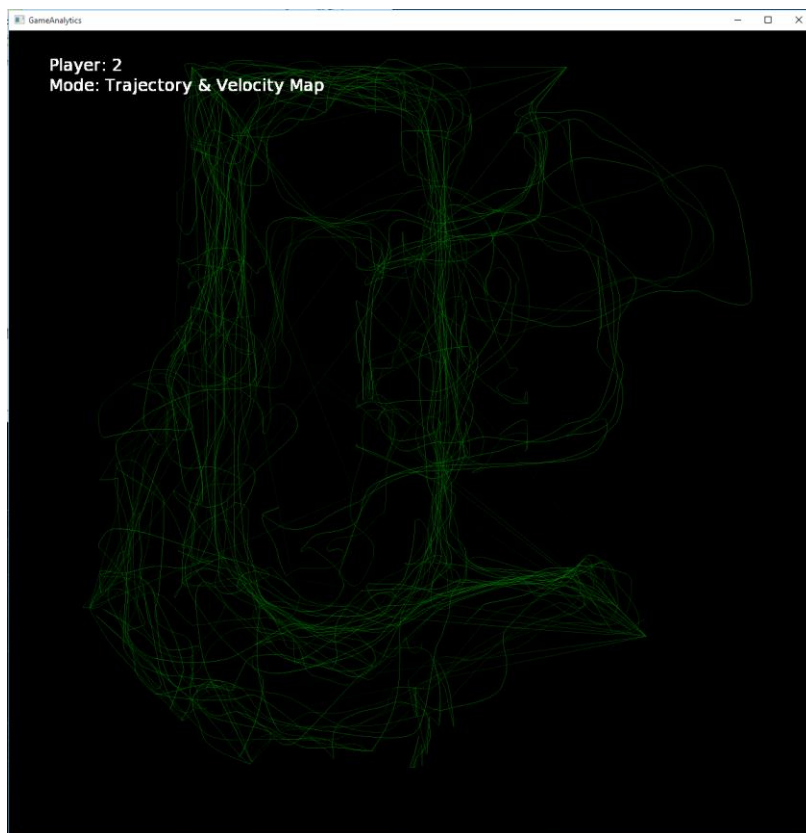
Appendix 10: Trajectory map for player 4.



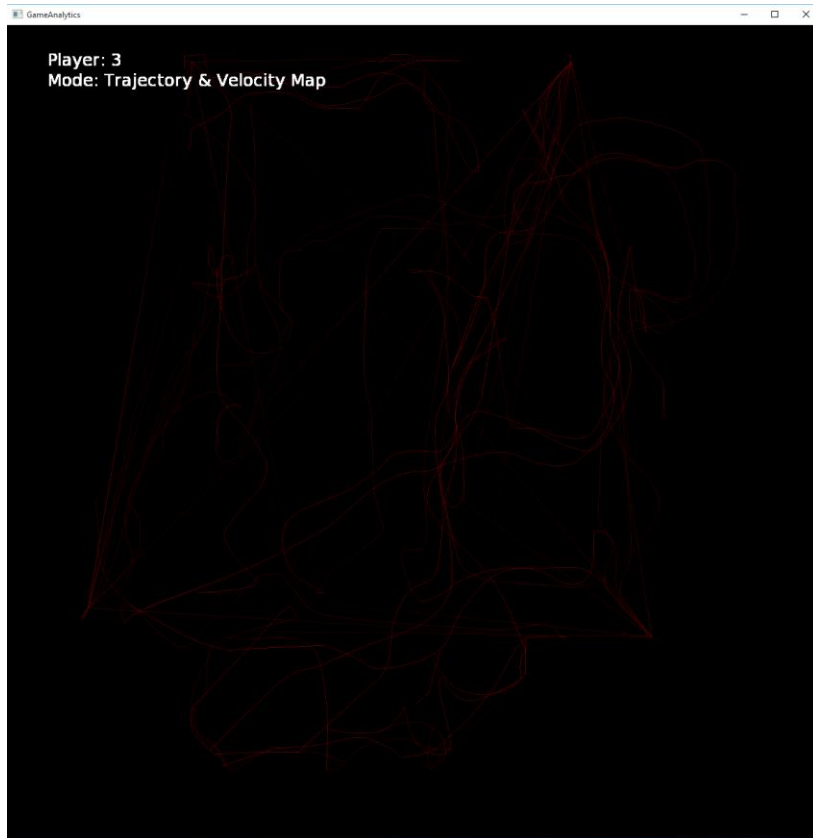
Appendix 11: Combined trajectory map for all players.



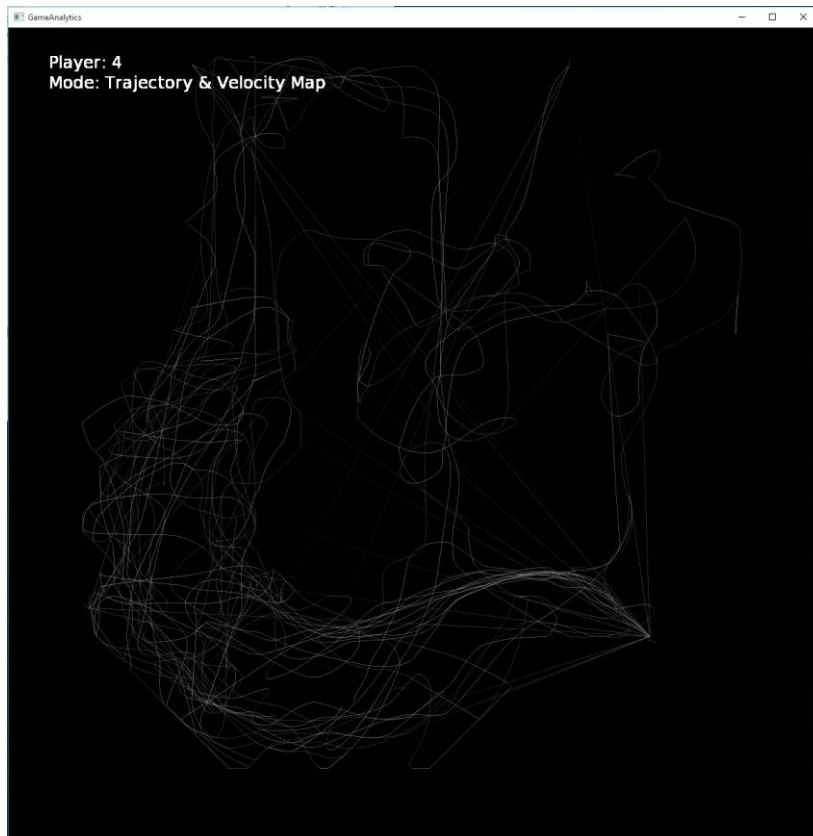
Appendix 12: Trajectory map showing speed through colour intensity for player 1.



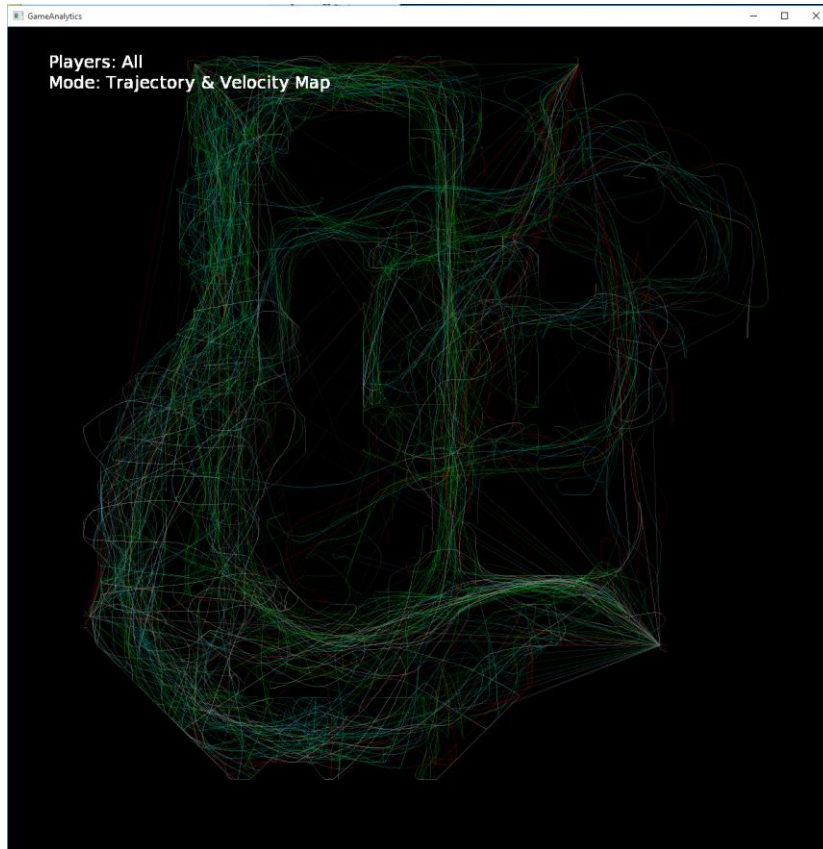
Appendix 13: Trajectory map showing speed through colour intensity for player 2.



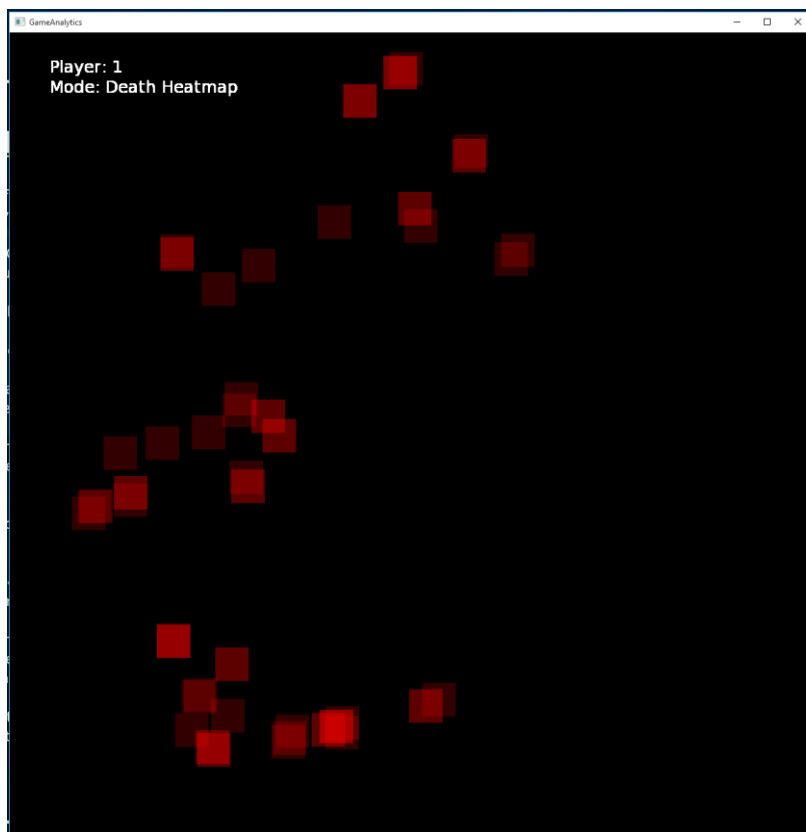
Appendix 14: Trajectory map showing speed through colour intensity for player 3.



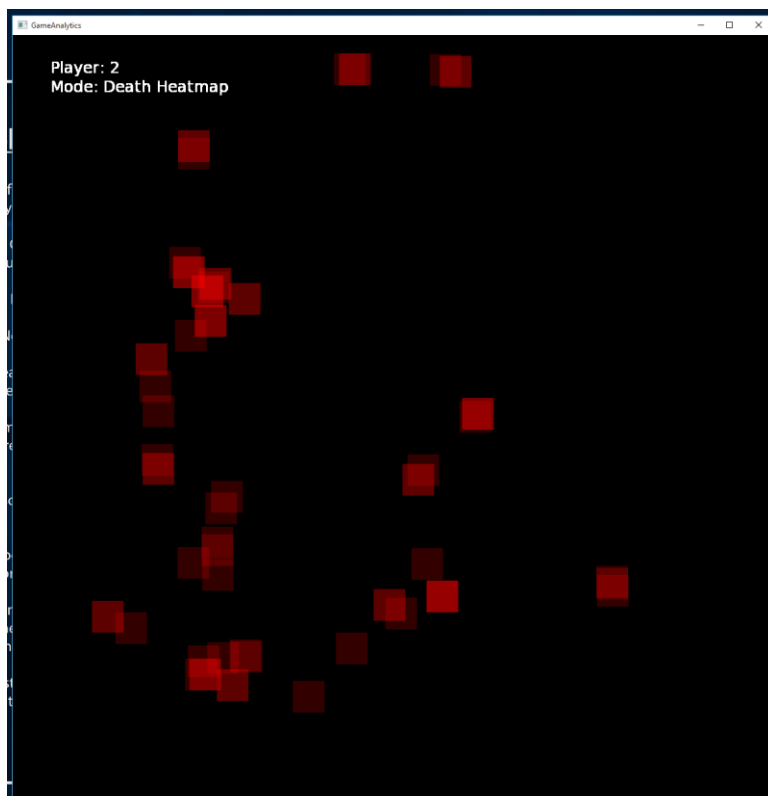
Appendix 15: Trajectory map showing speed through colour intensity for player 4.



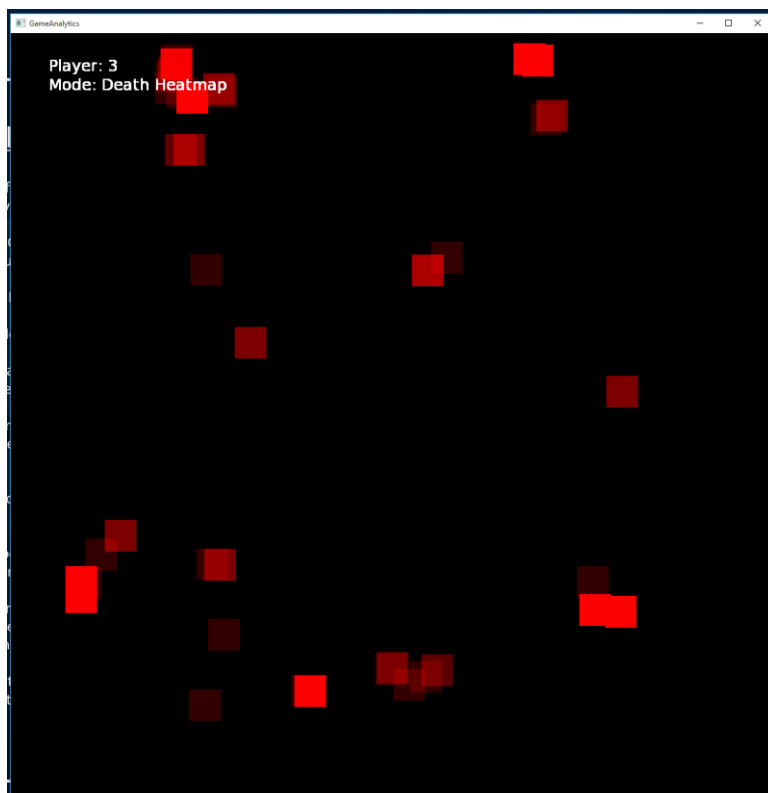
Appendix 16: Trajectory map showing speed through colour intensity for all players



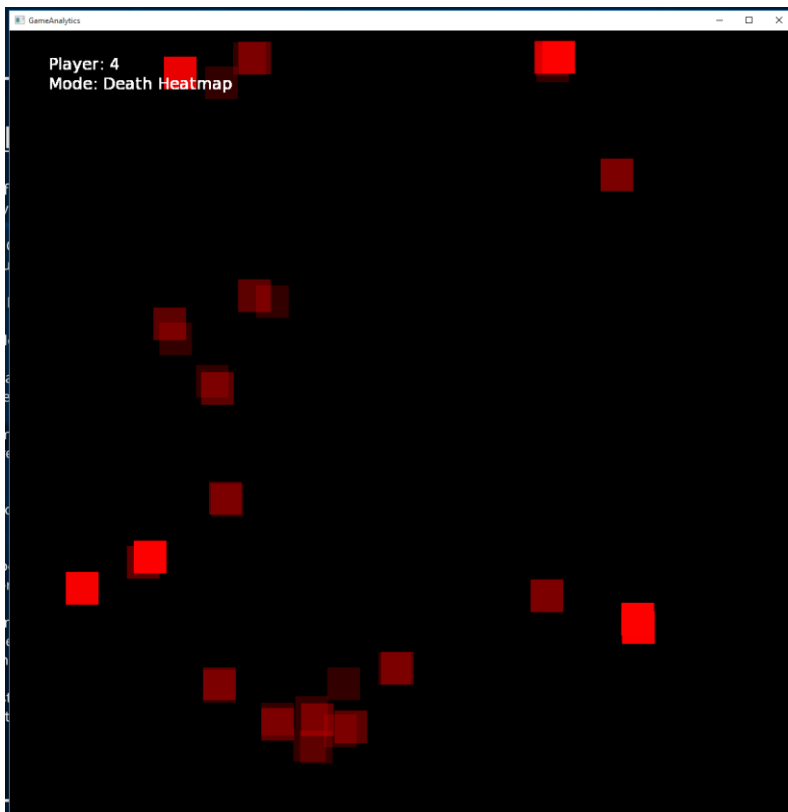
Appendix 17: Death heatmap for player 1.



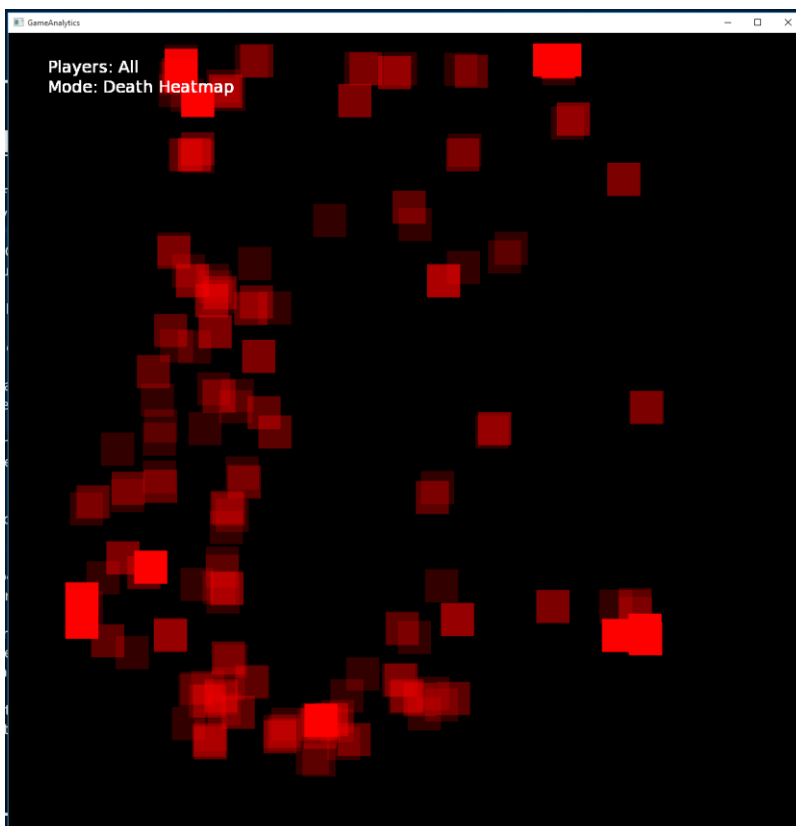
Appendix 18: Death heatmap for player 2.



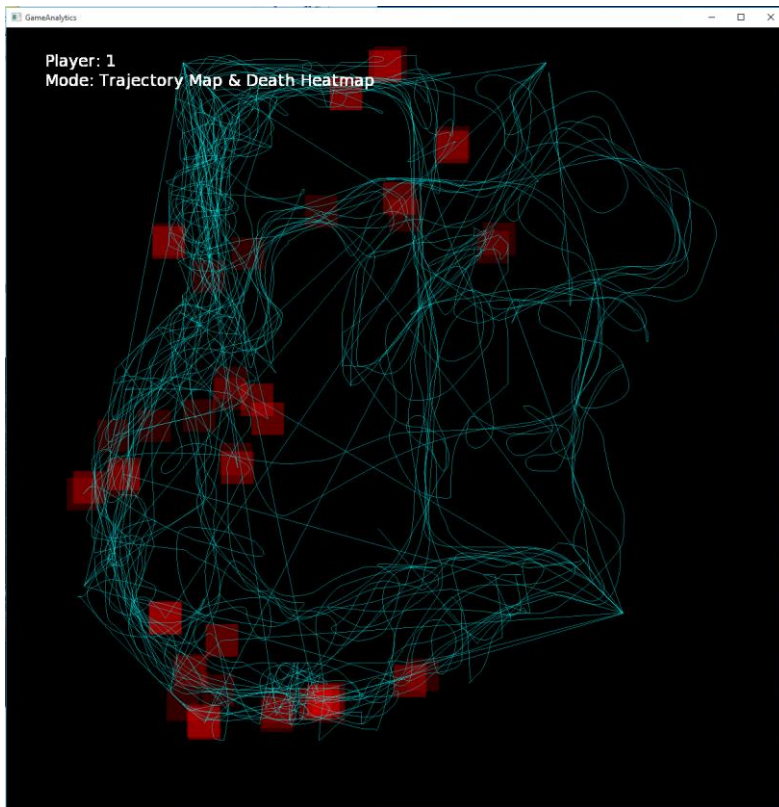
Appendix 19: Death heatmap for player 3.



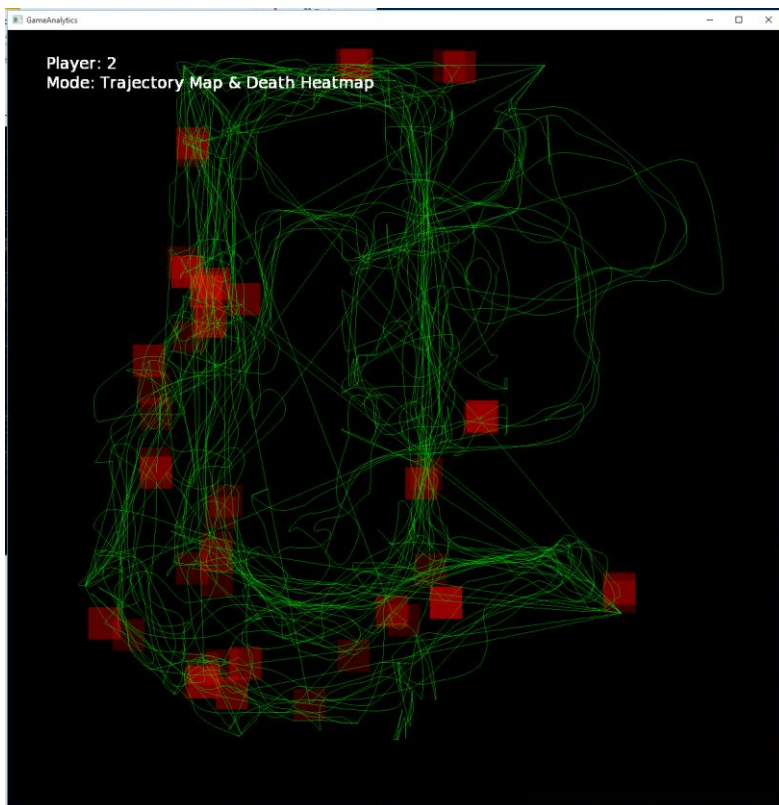
Appendix 20: Death heatmap for player 4.



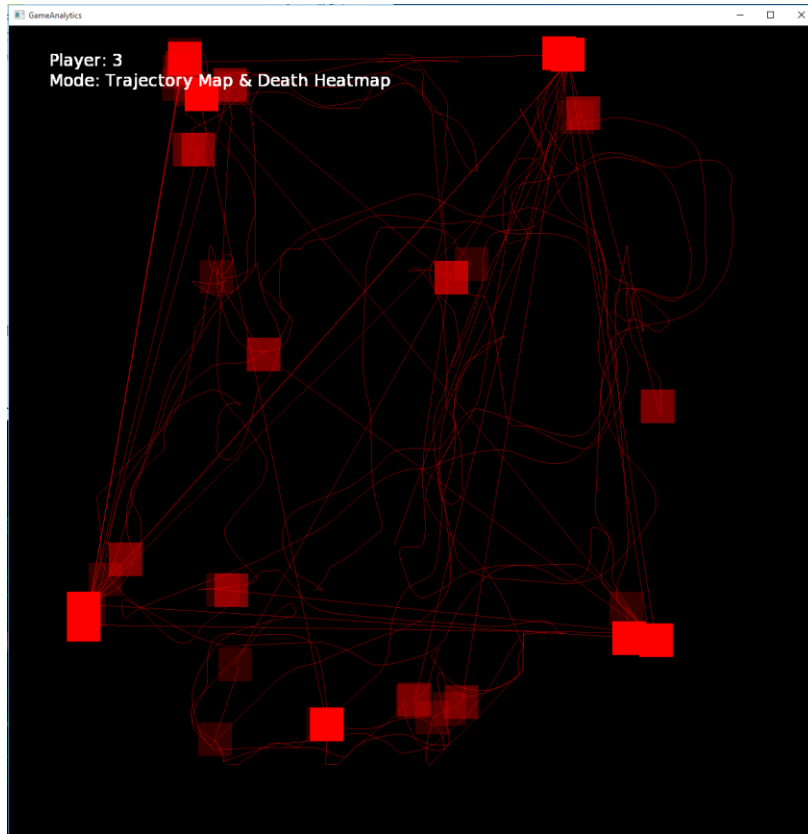
Appendix 21: Combined death heatmap for all players.



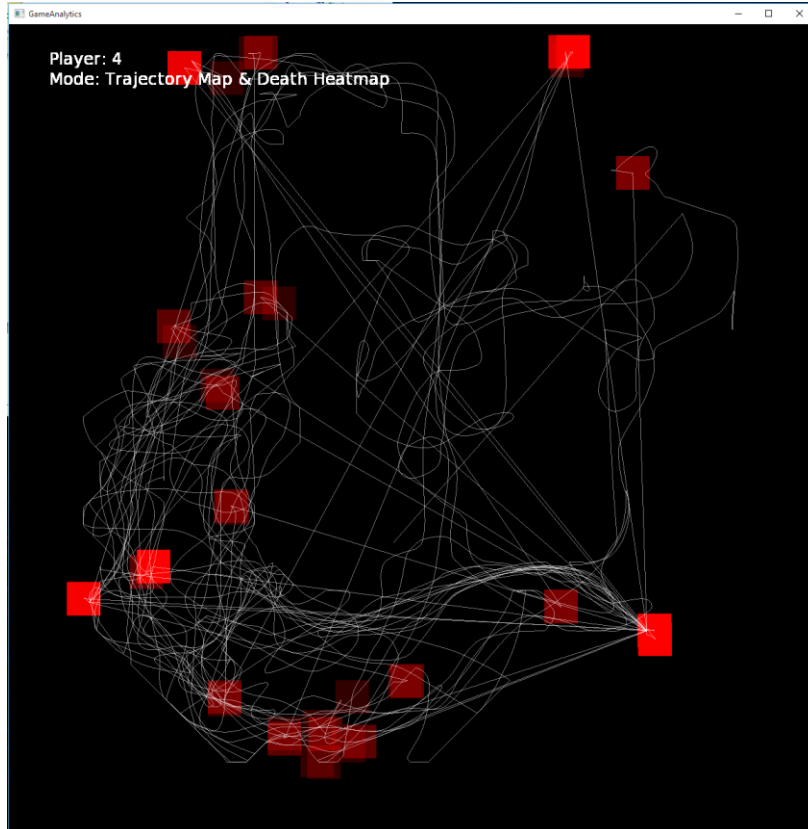
Appendix 22: Combined mode for player 1.



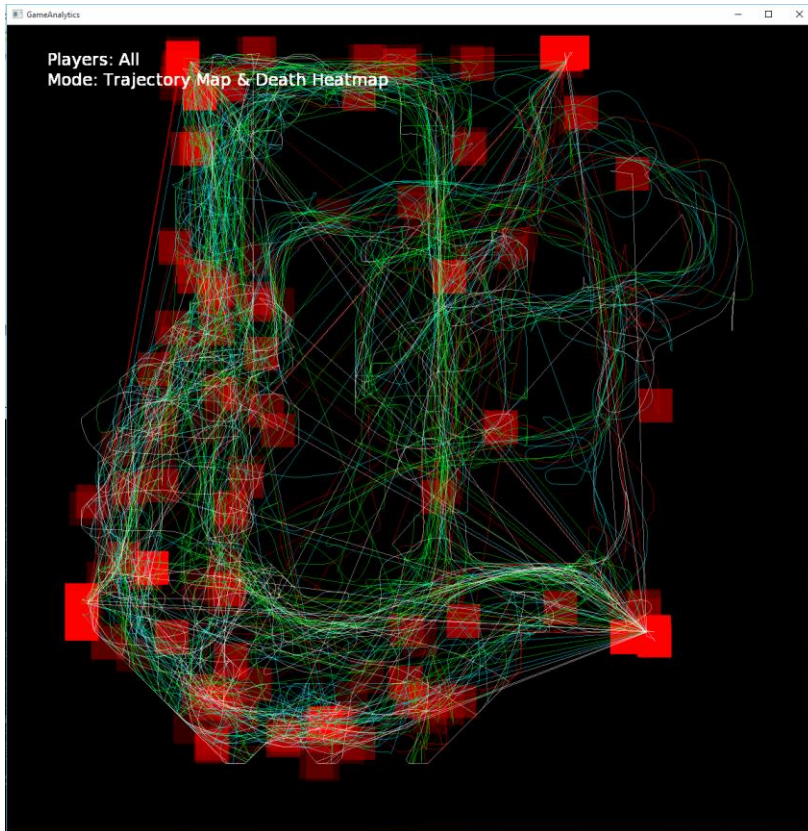
Appendix 23: Combined mode for player 2.



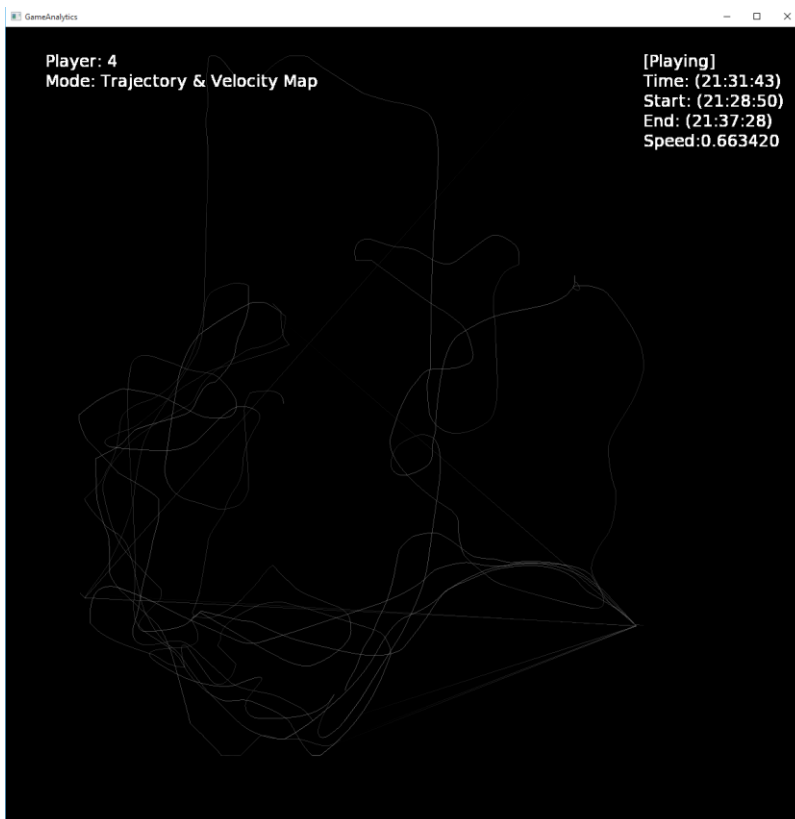
Appendix 24: Combined mode for player 3.



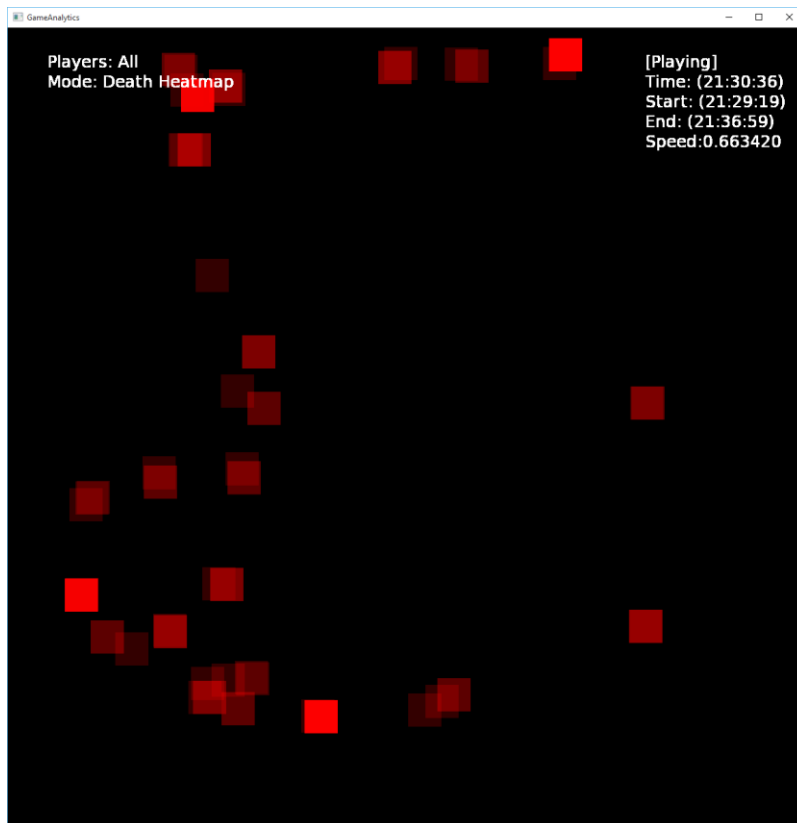
Appendix 25: Combined mode for player 4.



Appendix 26: Combined mode for all players.



Appendix 27: An example of using playback mode to follow the trajectory of player 4.



Appendix 28: An example of using playback mode to see how player deaths progressed through the game.