Live stream automation and remote management for video game tournaments

University of
St Andrews

March, 2020

David Wood
160013746

Supervisor: Tristan Henderson

Abstract

Video games as a spectator sport grows increasingly popular. With live streaming websites such as Twitch.tv playing host to millions of unique broadcasts every month, coupled with the growth of large esports events for games such as Counter Strike and League of Legends, it is clear that online broadcasting of these events is an important way to grow popularity for your video game.

Publicly available tools to help with these type of events for less popular games are scarce and in high demand. This project creates an application that processes the data of video games and uses it to automatically control a live video broadcast, keeping track of scores and displaying relevant information. Additionally, a remote configuration interface has been created which further reduces the labour required to run a video game live stream. The system is extensible, supporting almost every PC game available.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 13,958 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

1 Introduction

For the past 30 years, people have watched others play video games as a form of entertainment. The Nintendo World Championships in 1990 are seen as the beginning of video games as a spectator event [4], where attendees watched players compete to set high scores in a variety of Nintendo games. The proliferation of internet users led to the development of online games with built in multiplayer. Online tournaments such as Quake Red Annihilation took place as early as 1997 with over 2000 competitors playing in the tournament, where the finals were broadcast live online from the Electronic Entertainment Expo [5].

In recent years, sites such as Twitch.tv and YouTube have brought video gaming as a spectator sport to the masses [4]. These sites allow users to broadcast live video, known commonly as live streams, on the internet for anyone to watch. Viewers can choose between hundreds of thousands of users currently broadcasting, a majority of whom play games. From people casually playing single-player games, to online multiplayer games, to video game tournaments with 1,000s of attendees, these live streaming websites cater to a large, diverse audience of video game enthusiasts. The most popular of these live streams had a peak of 1.3 million live viewers [7], broadcasting the finals of a large video game tournament.

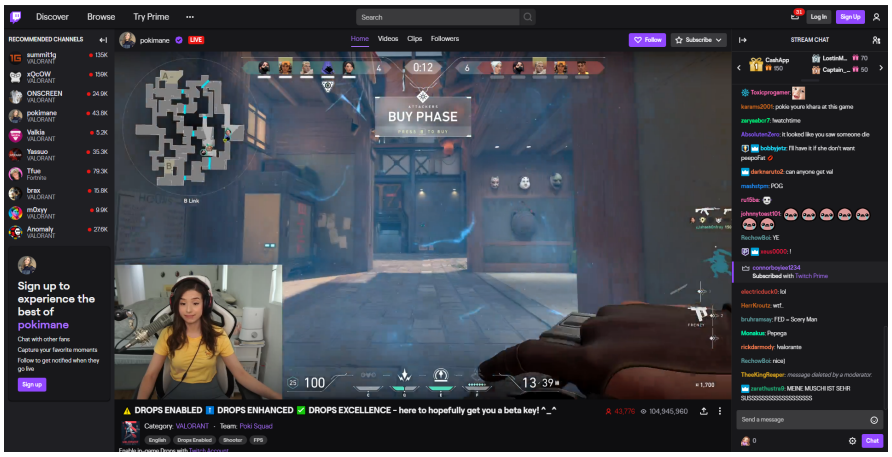


Figure 1: Live stream of the game *VALORANT* on Twitch.tv

The issue with live streaming websites today is that the market is dominated by the most popular games. Using data obtained from the website SullyGnome [19], a website which tracks Twitch.tv statistics, it was found that in February, 2020, over 50% of all game viewership was attributed to just 10 video games, with the remaining half being split amongst over 15,000 other video games. Figures 2 and 3 show cumulative distribution functions of total hours watched for all 15,600 games live streamed

on Twitch.tv during the month of February. The graphs display how the most popular games greatly skew the distribution. Within the first 10 games, over 50% of hours watched has already been reached, with 90% being reached at just after the 200th game.

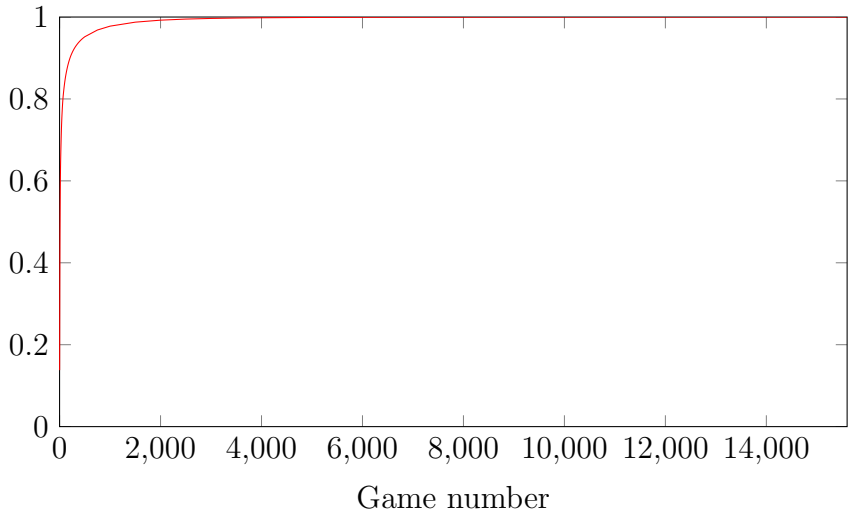


Figure 2: Cumulative distribution function of total hours watched of games watched on Twitch.tv in February, 2020

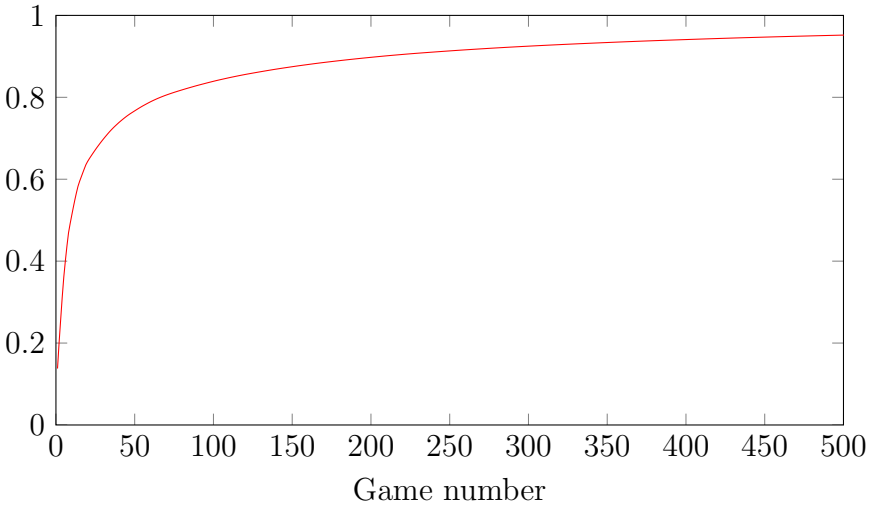


Figure 3: Graph from Figure 2 restricted to the first 500 games

Cha *et al* related Anderson's "the Long Tail", a business hypothesis that large opportunities exist in selling small amounts of large number of unpopular items, to the creation

of user generated content [2][1]. The paper postulates that entertainment videos on YouTube stood to gain up to 45% improvement in pursuing a Long Tail approach due to the large number of YouTube videos that go under the radar. Though the paper was written in 2007 when YouTube and Twitch.tv were a mere fraction of the size they are today, the ideas are still prominent today as the idea of the Long Tail can still be applied to video game live streaming.

The motivation for this project is to provide tools that will make it easier for smaller live streaming channels to grow, and provide support for video games that do not often get the spotlight. Large video game tournaments for games such as *League of Legends* and *Fortnite* are organized by the game publishers themselves, using proprietary software specific to the individual game. As a result they have a massive head-start compared to less popular games. Our specific focus will be on providing tools for tournaments of video games that do not have the infrastructure available to them and rely on grassroots organization. The software will make it easier for video game tournament organizers to run live streams for these events, providing automatic score tracking as the game progresses, and a web user interface that will allow users to remotely manage the application for things that cannot be automatically tracked.

To automate score tracking of a video game, it is necessary to first obtain data from a video game. This data will be vary game-to-game but largely will be simple things such as 'is a game in progress' or 'who won the last game'. The back end of the application will obtain this data and process it into usable information that reflects the current state of the game, which can then be used to modify aspects of the live stream, such as a score display.

1.1 Objectives

1.1.1 Primary

1. Create the back-end for the update mechanism that will interpret raw data from the video game.
    - The back end will extract data from the video game and present it in a format that can be used by the live stream software.
2. Create a customizable overlay for the video game that can be used to display changeable elements.
    - In order to easily display information on the stream, an HTML overlay will be displayed over the game containing all the relevant information for the game.

3. Create an automation system to track game progress and player information, and update the stream using the back end.
    - Using data extracted from the game in real-time, keep track of the game score and other relevant information. If a match has finished the winner will be displayed and the program will wait for the next match.
4. Create a remote management application to manually change elements of a live stream.
    - Some aspects of the live stream cannot be automated such as inputting player names. Create a remote management application that is accessible on any device such as mobile phones so stream organizers can manage the stream on the go.

1.1.2 Secondary

1. Use image recognition to extract information from the game.
2. Extend the system to support other video games using image recognition.
    - The system is currently limited to one video game, using image recognition and computer vision it could be expanded to support other video games.
3. Further automate the system by collecting player names through QR or RFID scanner.
    - One of the last remaining aspects of the stream that cannot be automated is the player names. By giving each competitor a QR code or an RFID tag we can automatically fill out their names as they are playing on the live stream.

1.1.3 Tertiary

1. Integrate the software with the Twitch desktop app.
    - To increase accessibility, have the system deployable through the Twitch.tv Windows app.

1.2 Disruption due to COVID-19

Due to disruptions caused by the COVID-19 pandemic, some minor aspects of the project were unable to be completed. By the time the university had closed, all of the primary objectives had been completed, as well as majority of the secondary objectives, and I had the necessary resources at home to finalize the project in its current working state.

The only objective that was not possible due to COVID-19 was secondary objective 3. I was unable to gather the resources necessary to pursue this objective due to the closure of the university. This objective would have fully automated the application, requiring even less input from the user. This objective isn't a necessity for the basic function of the overall application, however it would have been an important step in full live stream automation, the original motivation for the project.

Additionally, user testing could not be performed due to school closures. As a result, some aspects of the final application might produce unexpected errors due to reduced testing capacity. What tests were able to be performed are detailed in Section 11

2 Context survey

2.1 Motivations of live streaming

As briefly discussed in the introduction, live streaming websites are increasingly popular forms of entertainment. As such, it has a significant impact on the video game industry. Johnson *et al* discuss how live streaming has increased the lifespan of older video games that have long since lost the spotlight [11]. Particularly, the 'speedrunning' community, a group of video game players who aim to complete games as fast as possible, have brought back the spotlight to many older video games, such as *Super Mario 64* (1998) and *The Legend of Zelda: Ocarina of Time* (1998) [15]. Live streaming has allowed these games to remain a source of entertainment even 20 years after their initial release.

Live streaming can also be used as an effective form of advertisement. Twitch offers paid sponsorship opportunities to content creators through the use of their "Bounty Board program", whereby advertisers can offer live streamers money to play their video games and promote their products [20]. Live streaming gives potential customers a way to experience a game more wholly than a simple review. To quote a live streamer interviewed by Johnson *et al*, "if a big game streamer streamers a game, that's amazing advertising, because you see the game rather than reading the review" [11].

2.2 Developments in live streaming

Before looking at video game tournaments specifically, we must first discuss existing technologies that enable live streaming in the first place, and what services they offer to the user. The obvious choice is the widely popular Open Broadcaster Software (OBS) [13]. It is often the recommended option for live video broadcasting [8] as it is free, open-source, and works on Windows, macOS, and Linux. The program supports the composition of various elements, such as video sources, images, and text.

OBS supports the creation of scenes which are a composition of different input sources, such as video cameras, audio input devices, computer display captures, and more. The composition of these elements allows the user to very easily build simple video graphics such as displaying a video game with a webcam overlay. The creation of multiple scenes is an important part of live streams as it allows users to have scenes dedicated to a variety of scenarios. For example, users could swap to another scene which displays a video camera showing the players when a game isn't in progress, or swap to another scene when technical difficulties may occur. The flexibility of software like OBS is an essential part of live streaming.

Further developments have been made using OBS as a framework to improve ease of use, such as Streamlabs [18] which supports a variety of widgets that seek to aid

interaction with viewers. Streamlabs has a remote control app which allows users to manually change elements of the stream from their mobile phones. This type of system could help with certain goals of this project, however the remote control is not verbose enough for this project, and would likely further complicate things.

2.3 Developments in video game tournament broadcasting

As discussed in the introduction, there exists proprietary software used at video game tournaments that is specific to individual video games. For example, tournaments for the popular video game *Overwatch* use a tournament client that was created by the game developers. This client has additional capabilities coded into it to output raw information from the games for the broadcast teams to use, such as individual player statistics. This method of obtaining information from games is not scalable as it relies solely on game developers providing access to the data itself.

A similar method of raw information gathering is offered by the framework Project Slippi [16]. Slippi is a data extraction framework developed for use with the game *Super Smash Bros. Melee*, a console game released in 2001. The framework provides a variety of tools to extract data from the video game, send it to a computer, and process it into usable information. Software runs on the Nintendo Wii console which is able to output data in real-time, which is then sent across a network connection to a node running on a nearby computer which processes the data into a usable format. The framework is open-source and available to the public which makes it ideal for this project, but again scalability is a big concern which would limit the extensibility of the application.

In terms of scalable solutions, a company called Metascouter [12] is developing a system to extract data from a variety of video games using computer vision. The program is capable of reading stats from video games that are being played using image processing and presenting them in a variety of ways on live streams. The system currently only supports a handful of games but claims it can support any game if developed to support them. It is currently still in development and has not been released to the public as of yet. This project looks to take the idea of data extraction and processing that Metascouter offers and apply is to live streaming directly. The project will use the data extracted from video games to automate aspects of live streaming, making it easier for people to set up and run live streams for video game events.

3 Requirements Analysis

3.1 Final requirements

3.1.1 High-priority requirements

1. Functional: The system must be able to extract data in real time from a video game.
2. Functional: The system will store the data in an appropriate format. The data must be able to be read and edited easily.
3. Functional: Users must be able to alter parts of the data through an interface.
4. Non-functional: The interface must be accessible through the internet; all major mobile browsers (Safari, Google Chrome, etc.) must be able to use the system.
5. Functional: The system must automatically track the score as the game progresses, using the data extracted, resetting when the game has ended.
6. Non-functional: The system must display the relevant data in real-time format on the live stream.

3.1.2 Medium-priority requirements

7. Functional: The system will be extended to support other video games.

3.1.3 Low-priority requirements

8. Non-functional: The interface must be styled appropriately and must be easy to use.
9. Functional: The system shall make use of an RFID or QR code reader to read the names of the players are they are playing the game, displaying their name on the live stream. Specifically, players are assigned an RFID tag or a QR code and will scan it as they sit down to play a match.
10. Non-functional: The system must be easily deployable, potentially using a Docker container.

3.2 Removed requirements

Due to changes in the project specification due to ethical considerations (discussed later in Section 5), parts of the requirements had to be revised. The requirements listed here are the requirements that were removed from the original requirements analysis.

3.2.1 High-priority requirements

1. Functional: The system must be able to extract data in real time, in a readable format, from the video game *Super Smash Bros. Melee*.

    - This was removed as the project shifted to no longer focus on console video games due to ethical considerations. The requirement was instead replaced with the more generalised requirement 1. listed above.

3.2.2 Medium-priority requirements

3. Functional: The system shall be able to use image recognition to extract information from a video game being played.

4. Functional: The system shall support various other video games through the use of image recognition.

    - The new chosen implementation for reading data from a video game no longer required the use of image recognition to implement extensibility as it supported it by default. These two requirements were replaced with requirement 6. listed above.

4 Software Development Process

The creation of requirements made it relatively simple to create a general plan for software development. It was clear from the beginning that the project is composed of a variety of systems that can be built up from the core requirement of extracting data from a video game. As such, we decided to pursue an agile development process with 1 week sprints in accordance with our weekly supervisor meetings, coupled with weekly progress report so we could keep track of progress. This allowed for any obstacles in the software development to be dealt with quickly which proved to be important throughout the project. Meetings were useful to discuss potential ethical concerns with the project, show demonstrations of the application, and planning for future sprints.

In our first few meeting we discussed the most important aspects of the project design, such as data storage methodologies and what various technologies could be employed in the project. The most notable benefit to using agile development was presented to us just before the end of the first semester. After complications with the ethics application, part of the project had to be reconsidered and a portion of the coding done towards the project had to be scrapped. A different approach was then followed from then on which will be discussed in detail later on. Without agile development we would not have been able to catch these issues early, and many more issues could have arose throughout the development process.

Continued development of the application presented us with a working prototype for data extraction in February. From there I built up integration with the live stream software for automation, and created the remote management interface to allow remote configuration of the application. It was around this time that the high-priority requirements were mostly fulfilled and development could begin on lower-priority requirements and further testing. The final development sprints were spent on polishing the application, making it more user-friendly, and ensuring it is in working order.

5 Ethics

The main ethical consideration for the project was that 3rd party software, the Project Slippi Framework, must be installed on the Nintendo Wii console to enable the extraction of data that can be used for managing live streams [16]. Such installation violates the End-user license agreement for the Nintendo Wii console, which reads:

> "Your Wii Console and the Wii Network Service are not designed for use with unauthorized software, services, or devices or non-licensed accessories, and you may not use any of these with your Wii Console."

Because of this, there was a small chance that Nintendo may have shut down the project. Historically 3rd party software running on games consoles has been a legal grey area; whilst not explicitly illegal, the legality of the software is not confirmed either.

However, recently the 'Right to Repair' movement has challenged this idea [6]. The EFF has been fighting to give users control of their devices, hopefully allowing people to legally repair their own appliances, modifying existing software, and installing 3rd party software on owned hardware. One of their main arguments is that it stifles innovation, which was especially relevant in this case. Though there is a chance that it breaks the law, it could be said that breaking this law is justified given that it would stifle this project.

Due to this, a full ethical application form was submitted for this project. The original proposal was discussed by the University Ethics Committee and was deemed to be to be infeasible due to potential legal concerns. Following this, the project was revised to not require the use of a video game console or the use of Slippi Framework. Instead the project focusses on extracting data from PC video games, reading the data directly from the computer's memory. The shift away from console video games meant that a framework would need to be built from the ground up rather than using existing technologies. This method does not have any associated End-user license agreements or any further ethical concerns.

As a result there are no further ethical considerations with the project. All data that is collected by the application is anonymous and contains no personal information. In terms of data storage, all data is stored in a secure database on the host computer.

6 Design

6.1 Project Overview

Before designing specific components of the application, it is important to consider the overall system design. Figure 4 shows the general data flow throughout the system. A configuration file will describe certain aspects of the system, such as the IP and port used to host the web server. The system reads data in from the video game and processes it into usable information, keeping track of the current match score. This information is combined with the player information manually input by the user on the web interface hosted, and is used to display the match information on the live stream. Additionally, the game data will also be used to control the live stream itself by changing what scene is currently being displayed on the broadcast, displaying sources such as video cameras, audio devices, images, etc.

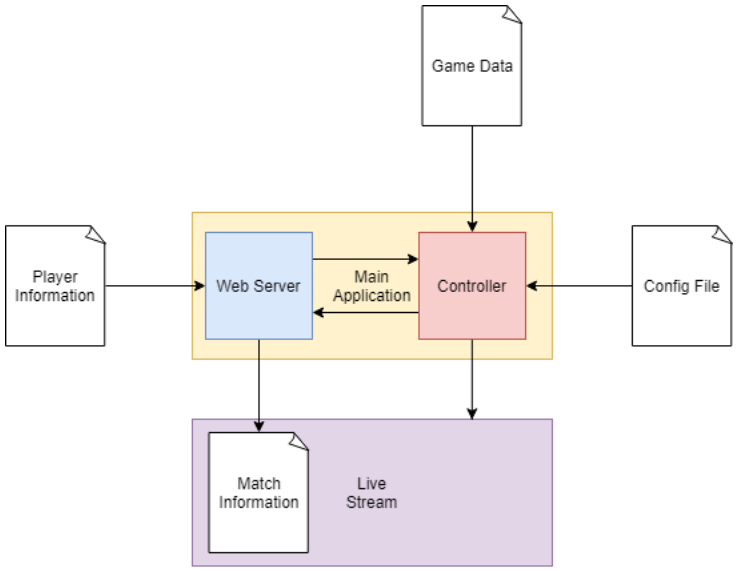


Figure 4: System Design

6.2 Data Extraction

The biggest design decision in the project was how data was going to be obtained from a video game. This decision would shape the scope of the whole project as a method that was extensible to multiple games would be ideal. Initially we planned to use the Slippi Framework as described in the context survey (Section 2), as it provided a simple infrastructure that worked out of the box. As a lower-priority requirement I chose to use image analysis similar to Metascouter as a means of providing extensibility through

supporting multiple games. However, due to ethical considerations regarding breaking the Nintendo Wii's End-user license agreement, we decided to no longer pursue using the Slippi Framework.

Instead we decided to focus on PC games, specifically open-source PC games. The idea was that it would be possible to write a plugin or modify the source code to output data in real time from the application. This type of system would again not be very scalable but would give a means to gather data which could be expanded later. Fortunately however, this gave me inspiration to pursue a different path. Instead of relying on games themselves outputting data, I decided instead to simply read the data from the computers memory myself. This solution could be applied to any game that can run on a computer and it would be extensible by default, not requiring any additional programming to extend the system to different games. Games could be specified by lightweight configuration files instead of requiring reprogramming each time.

There is one issue with this method of approach. It is unclear whether or not this method of reading memory will work with online games with aggressive cheat detection policies. Though the method used by this project doesn't modify the memory of a video game, the practice of reading and writing memory is a common way to cheat in online video games. Often times developers will opt to use cheat detection programs such as 'Punkbuster' which aim to detect software that is used to give players unfair advantages in online multi-player games. There is a chance that games that employ these strategies will not work with this software and may result in players getting banned from online play. As such, this project will focus mainly on local multi-player games that do not have cheat detection measures.

6.3 Operating System

Though both Microsoft Windows and Linux support video games, the majority of video games are developed for Windows. Using data taken from the popular video game distribution service Steam [17], as of 25th April, 2020, Microsoft Windows supports 75,101 games offered by the platform, compared to Linux supporting only 14,833 games. Data extraction from games on each OS would require separate implementations, and a variety of tools useful to development only work on Windows (discussed in Section 7.2). As such, the system will be designed and implemented only for Windows.

6.4 Application interfaces

6.4.1 Web configuration interface

The web interface is the main way to interface with the application. The interface can be accessed by any device with access to a web browser, allowing modifications to be

made remotely. The page allows for users to change what scene is currently used by the live streaming software, change what game is currently being processed as offered by the local application interface, and input the names of each player currently playing the game. Buttons are associated with each element to submit changes made to each.

Figure 5: Initial web configuration GUI

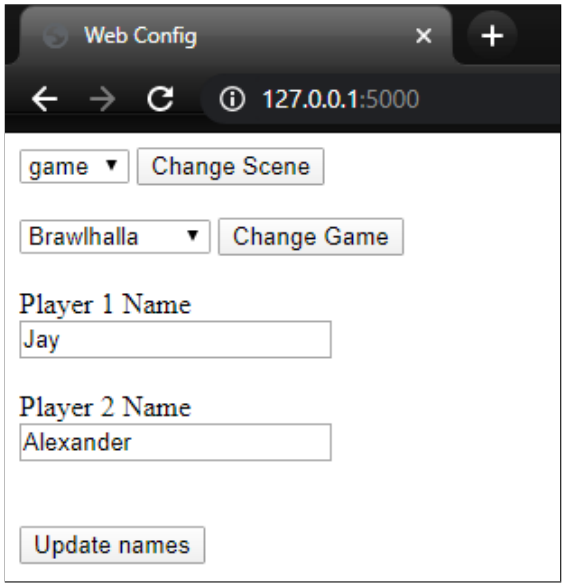


Figure 6: Web application GUI final implementation

6.4.2 Controller application interface

The controller application interface is an interface that is displayed on the computer running the application. It is not the main way to interface with the application and is less verbose than the web interface. It provides a basic way to configure the application and a way to exit the application when necessary. The interface only requires the user to select what game the application should track so the interface need not be very complicated. Initially, the application was controlled via the console and while it functioned as needed, it was not as streamlined as a graphical user interface. As a result I scrapped the console application design and decided to opt for a simple GUI.

The local application interface offers each game supported as a button that will start processing that game if clicked. An indicator is displayed at the side to indicate which game is currently in progress. From this interface users can swap between games and will be notified if a game has been closed.

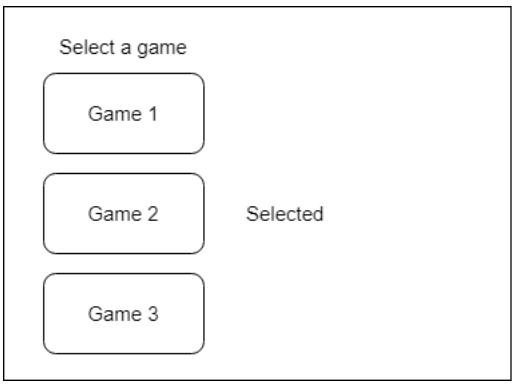


Figure 7: Initial controller application GUI design

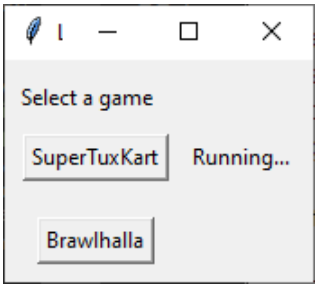


Figure 8: Controller application GUI final implementation

6.5 Live stream overlay layout

Shown below is an example overlay that can be displayed on the live stream. The box on the right side of the screen is used to display the scores of the players currently playing. A banner is also displayed across the top of the screen showing the winner of the match when a match has ended. This overlay will be displayed on top of a video output of the video game similar to the score overlays used in TV broadcasts of sports games. This design was chosen as it can be used between video games as the supported elements (score, winner) are consistently used with most video games.

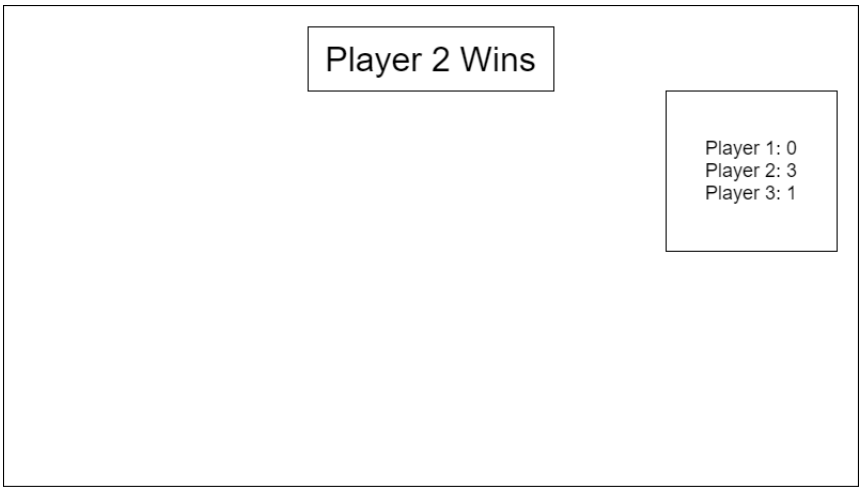


Figure 9: Example live stream overlay layout

Figure 10: Final live stream overlay, shown live on Twitch.tv

7 Tools and Technologies

7.1 Open Broadcaster Software (OBS)

As discussed in the Context Survey (Section 2), Open Broadcaster Software (OBS) is an open-source live streaming application. I chose to use regular OBS over Streamlabs OBS due to OBS's flexibility with plugins. Streamlabs OBS supports a variety of widgets but none catered to what was required by the project. Instead we use OBS with the obs-websocket plugin which allows the application to be controlled through a socket-based API, which we use to swap between different scenes in OBS. OBS supports the use of browser sources, where live web pages can be displayed on the live broadcast. The use of browser sources allows information to be displayed on top of the game being played, such as player names and player scores.

7.2 Cheat Engine

To read the data from a games memory, you must first know what memory addresses to read from. The memory structure of video games is vast, complicated, and constantly changing. Memory addresses are very rarely static and will continuously change location depending on when allocated and freed. Fortunately, there are tools available that make it possible to find the location of specific values in memory, and ensure that they can be read from regardless of their actual memory address.

One of these applications is called Cheat Engine [3]. It is a low-level, open-source memory scanning and debugging application created for Windows. The primary function of Cheat Engine is modifying memory values to cheat in video games, such as increasing a players score artificially. However, the application is a perfect fit for finding relevant the memory addresses for this project. It is worth clarifying that the application does not rely on Cheat Engine to run, the application is used to find memory addresses that are specified in a game's configuration file.

The application is able to scan through an applications memory space looking for specific values. Figure 11 shows an example where the value 8 is searched for in a process' memory space. Each memory address is checked and if it stores the value 8, it is output to the list displayed on the left side of the screen. Subsequent scans can be used to narrow down the list to a small set of addresses. For example, if a player has a score of 5, you initially scan for the value 5 in the program and get a large list of results. When the players score changes to 6, you again scan for the value 6 which narrows down the long list of results to a smaller list. Eventually, through tracking various changes in a value, it is possible to find the exact memory address used to track a player's score.

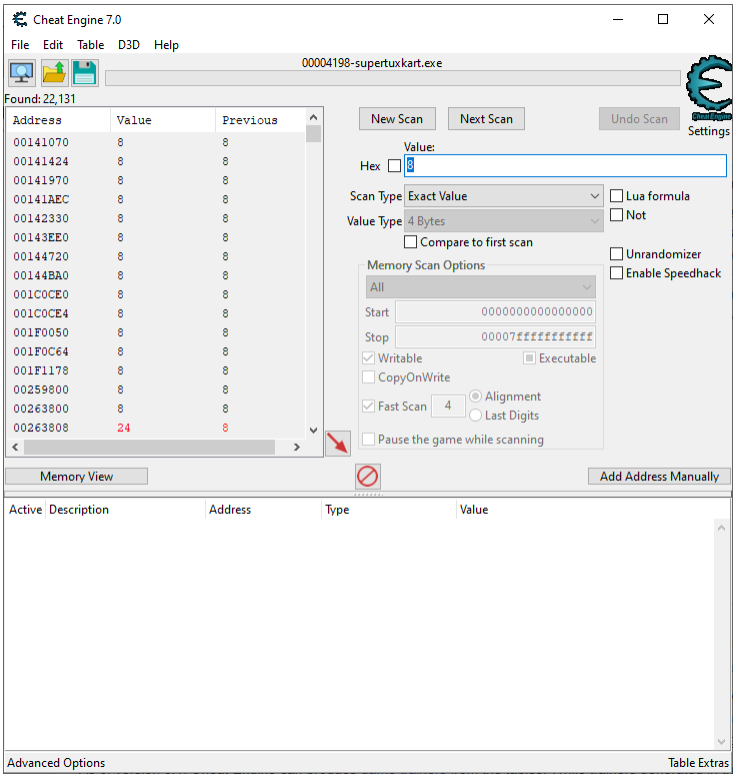
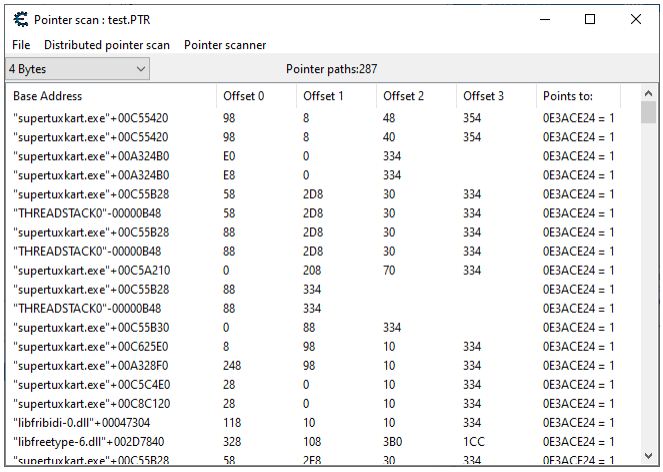


Figure 11: A Cheat Engine scan for the value 8

The issue is that the addresses found by Cheat Engine are still not static addresses. The memory addresses can still change on subsequent runs of the game and will bear no relation to the address found initially. However, there is a way to remedy this. Memory addresses are allocated in one of three ways; static allocation that is assigned when the program starts so is constant between runs of an application, automatic allocation which occurs for non-static variables declared inside functions, and dynamic allocation that occurs when memory is allocated manually and freed. Automatic and dynamically allocated variables that we are trying to find can always be found by adding various offsets to static memory addresses and reading the pointer values at those offsets. The only issue is finding the correct set of offsets.

Cheat Engine's pointer scanner feature recursively scans from each modules static base address. Each process has a variety of modules that are loaded into memory when the application starts, such as `.dll` and `.exe` files. Conveniently, it is possible to find the static base address of these modules programmatically between executions. Cheat Engine iterates through a variety of offsets that are added to these module base addresses, checking the value of the resultant memory address. For example, if the max offset was set to 1000 and the base address was at 0x00004000, it would check

the values at 0x00004000 through 0x00005000. If the value at the memory address is a pointer to the address we wish to find, it is added to a list and it continues. The pointer scanner also accepts a max depth, which refers to how deep the tree search will go. For example, it will check the values at all 1000 offsets from a module's base address initially, then from each of those 1000 offsets it will search another 1000 offsets. If the depth is set to a large value (typically greater than 5), then the output list of values can be expected to be many gigabytes or terabytes in size, increasing exponentially. Both space and time complexity can be represented by $\mathcal{O}(b^m)$ with $b = max\ offset$ and $m = max\ depth$.



Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Points to:
"supertuxkart.exe"+00C55420	98	8	48	354	0E3ACE24 = 1
"supertuxkart.exe"+00C55420	98	8	40	354	0E3ACE24 = 1
"supertuxkart.exe"+00A324B0	E0	0	334		0E3ACE24 = 1
"supertuxkart.exe"+00A324B0	E8	0	334		0E3ACE24 = 1
"supertuxkart.exe"+00C55B28	58	2D8	30	334	0E3ACE24 = 1
"THREADSTACK0"-00000B48	58	2D8	30	334	0E3ACE24 = 1
"supertuxkart.exe"+00C55B28	88	2D8	30	334	0E3ACE24 = 1
"THREADSTACK0"-00000B48	88	2D8	30	334	0E3ACE24 = 1
"supertuxkart.exe"+00C5A210	0	208	70	334	0E3ACE24 = 1
"supertuxkart.exe"+00C55B28	88	334			0E3ACE24 = 1
"THREADSTACK0"-00000B48	88	334			0E3ACE24 = 1
"supertuxkart.exe"+00C55B30	0	88	334		0E3ACE24 = 1
"supertuxkart.exe"+00C625E0	8	98	10	334	0E3ACE24 = 1
"supertuxkart.exe"+00A328F0	248	98	10	334	0E3ACE24 = 1
"supertuxkart.exe"+00C5C4E0	28	0	10	334	0E3ACE24 = 1
"supertuxkart.exe"+00C8C120	28	0	10	334	0E3ACE24 = 1
"libfribidi-0.dll"+00047304	118	10	10	334	0E3ACE24 = 1
"libfreetype-6.dll"+002D7840	328	108	3B0	1CC	0E3ACE24 = 1
"supertuxkart.exe"+00C55B28	58	2E8	30	334	0E3ACE24 = 1

Figure 12: A Cheat Engine pointer scan of an address

Figure 12 shows an example pointer scan of an address, with max offset set to 5000, and max depth set to 4. The first result shows the base module "`supertuxkart.exe`" + an initial offset `0x00C5542C` and 4 more offsets. The value at the base module address + the initial offset is read, then 'Offset 0' is added to the result and then subsequently read. Eventually after all 4 offsets have been processed, it points to the address listed in the rightmost column with the value 1. Note that with a max offset of 5000 and a max depth of 4, a total of 287 results were found. Most values can be found at a depth of 4-5 but certain values may require depths of 9-10+ which, as shown earlier, requires a lot of storage and a lot of time to find. Furthermore, to ensure that these values are universal and work on any computer, it is recommended that the same process be executed on a different computer for each value obtained to narrow the results down further. A value can then be chosen from the resultant list and it will work as indented.

7.3 C++

After obtaining a list of addresses and offsets to read using Cheat Engine, a lightweight program must be created to read the values from the games memory in real time. Reading the memory of a process requires low-level programming which cannot be offered by languages such as Java or Python. There were only two main languages to choose between to create such a program, C++ or C#. C# is higher level than C++ but still has libraries available that can be used for reading a process's memory. In the end I decided to use C++ as I was more familiar with it than C# and thanks to it being lower level than C#, the program could be optimized further.

7.4 Python

Implementation of the controller application, including the web server, was done in Python. The application itself didn't need to be particularly low-level so I chose Python as I was very comfortable using the language. It makes the creation of web servers trivial, it supports a variety of modules that can be easily installed, and is very portable between different machines by the use of virtual environments. Python can interface with external applications using the built-in subprocess library which allows it to run the memory reader application written in C++. The system uses a variety of external libraries:

Flask: Flask is a micro web framework developed for Python. It provides a lightweight web application framework which is used to host the web configuration and the match information displayed on the live stream. As the web applications does not need to be accessed by many people, a lightweight framework such as Flask is ideal for this project as it is simple to implement and supports robust security measures such as SSL.

obs-websocket: Python neatly integrates with the obs-websocket plugin discussed earlier. The library serves as an abstraction over the socket requests API exposed by the OBS plugin and neatly allows for connections to be established and requests to be sent and received.

Tkinter: Tkinter is the standard GUI used by Python. To create the local application interface, I use Tkinter to draw the window and elements, and handle any events that occur.

SQLite: To store the player information, I opted to use SQLite for data storage. SQLite is a relational database system that, unlike traditional database management systems, is embedded into the application rather than using a client-server model. I

chose to use SQLite over other database formats as it is well supported in Python and is lightweight in comparison to other database systems.

7.5 Javascript

To create the web configuration and the match information display, a combination of HTML, CSS, and Javascript were used. Javascript is used to populate both pages with information, such as player names, scores, winners, etc. Additionally, integration with the obs-websocket plugin is supported by Javascript and has been added to the web configuration.

8 Implementation

8.1 System architecture

The structure of the final implementation of the system can be seen below. It is a developed version of the design shown earlier in Figure 13. Each box represents a different component in the system. Provided interfaces are represented by circles, such as the 'Live video' interface provided by the live streaming software. Required interfaces are represented by semi-circles, such as the 'User input' interfaces offered by the controller. Finally, dependencies are specified by dotted arrows, such as `database.py` depending on the `database.db` file, and the live streaming software depending on the `overlay.html` site.

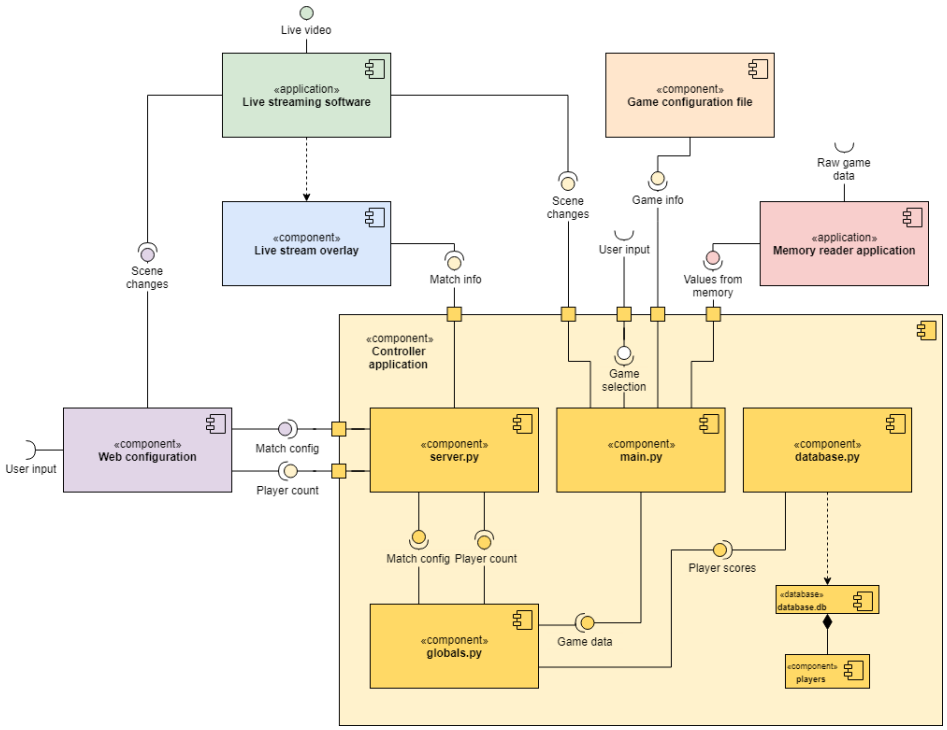


Figure 13: Component diagram of the system

The diagram highlights the individual components that cooperate to make up the whole system. The controller component handles the values and logic described in the game configuration files, processes the information output by the memory reader application, and runs the web server to handle the web configuration and the live stream overlay. The structure is very modular due to the contrast between low-level code required by the memory reader, the higher-level implementation of the controller, and the separate

web components provided by the controller. A monolithic approach would mean the controller would be more complicated to implement and the web pages would be less dynamic due to the focus on the server-side implementation.

The diagram shows that the only interfaces available to the user are on the web configuration and the controller application itself, whereas the memory reader application requires data from the video game. The culmination of the application is the output of live video by the live streaming software which can be controlled by the controller and by the web configuration.

The following sections detail the specific implementation of each component, including the various changes made throughout the development process.

8.2 Game configuration files

The game configuration files are implemented using the JSON file format. The file structure of JSON lends itself to supporting the various attributes required by the controller application.

Figure 14 shows an example JSON configuration file. The `platform` indicates which memory reader executable should be used (x86 or x64). The `pointers` list indicates the list of memory addresses to read from. Each entry contains a variety of variables that are used by the memory reader application to read the correct memory address of the value it is trying to track, identified by the pointer's `id`. The `display` indicates whether this value should be added to the output JSON file that is used to display data on the live stream. If `true`, it is possible to display this value on the live-stream if the overlay supports it.

Finally, the `controller` object is where some basic logic is described that is used to control certain elements of the live stream. There are 4 actions supported by the system.

1. `player_count` — the number of players currently playing the game
2. `match_finished` — an indicator to check if a match has concluded
3. `winner` — the player number of the match winner
4. `paused` — an indicator to check if the game is paused

Each element is described using some basic logic. Strings such as `no_of_ai` indicate variables described in the pointers list. Integers act as regular integers. Objects are operators denoted by the key with the parameters passed as a list object of length 2 as the value. Permitted operators are `+`, `-`, `*`, `/`, `&&`, `||`, `==`, and `!=`. This basic logic allows control to be specified on a per-game basis, rather than requiring hard-coding of

commands as offered by other technologies discussed earlier in the context survey, such as Metascouter.

```
{
    "platform": 64,
    "controller": {
        "player_count": "no_of_players",
        "match_finished": {
            "==": ["results_screen", 1]
        },
        "winner": {
            "+": [1, {"-": ["first_place", "no_of_ai"]}]
        },
        "paused": {
            "&&": [{"!=": ["results_screen", 1]},
                   {"==": ["race_in_progress", 0]}]
        }
    },
    "pointers": [
        {
            "id": "race_in_progress",
            "window": "SuperTuxKart",
            "module": "supertuxkart.exe",
            "initial_offset": "0x00C625E0",
            "offsets": ["0x58", "0x48", "0x28"],
            "display": false
        },
        {
            "id": "results_screen",
            "window": "SuperTuxKart",
            "module": "supertuxkart.exe",
            "initial_offset": "0x00A325F0",
            "offsets": ["0x1B0", "0x0",..., "0x1BC"],
            "display": false
        },
        {
            ...
        }
    ]
}
```

Figure 14: Example game configuration file. *n.b. some information has been omitted*

8.3 Memory reader application

As discussed in the design section, the memory reader application is implemented in C++. As this project is built for Windows, we will use the standard header `Windows.h` which provides a variety of functions useful for memory reading. Additionally, the header `Tlhelp32.h` is used to help with finding specific modules within a process.

The application takes in a variety of parameters which are found using the Cheat Engine application. It takes the window name, which is the name of the application's as it appears when running in Windows (e.g. the process `chrome.exe` has the window name 'Google Chrome'); the module name of the base address, which is usually an `.exe` or `.dll` file; the initial offset to add to the base address; and finally an optional list of additional offsets to add to the calculated address. To aid the loosely coupled nature of the system, there is no connection established with the Python controller, the output of the application is simply output to `stdout`.

To find the memory address specified by the pointer offsets, we first need to attach to the running process itself. We do this by creating a handler to the game's top-level window using the function `FindWindow()` offered by the `windows.h` header. Using this handler we can then get a handler to the process itself using `OpenProcess()`. The next step is to get the base address of the module provided in the parameters. We use the `Tlhelp32.h` header to take a snapshot of the running process which stores the heaps, modules, and threads used by the process specified. We iterate through the list of modules in the snapshot, comparing the names to the one we want, and then store the base address of that module.

Once the base address has been found, we first add the initial offset supplied in the parameters. The resultant address is the first memory address that we read the value of. To do this, we pass the address into a function alongside the list of optional offsets provided in the parameters. We then iterate through this list of offsets. We read the memory address using the `ReadProcessMemory()` function; passing the process handle found earlier, the address to read from, and a pointer to the current address variable where the resultant pointer address will be stored. The next offset is then added to the current address and then processed again until every offset value has been added. After all the offsets have been processed, the value of the final address is read, which corresponds to the actual value we wish to track, which is then output to `stdout`.

To compensate for differences in x86 and x64 memory addressing, two versions of the application are compiled, one for each system architecture. x86 applications cannot read the memory of x64 processes, but x64 applications can read the memory of x86 processes. However, to implement x86 reading in x64, a variety of conditionals are required which is more complex than simply having two executables. The application is very lightweight so the overheads in compiling two versions are negligible.

8.4 Controller application

The controller application is split into a variety of separate components. The application is written in Python and uses a variety of the technologies discussed earlier.

8.4.1 Controller initialisation

The main controller facilitates processing of the game configuration files and the associated logic, drawing of the local application GUI, interfacing with the memory reader, and some basic communication with OBS. The main controller is also used to start the web server and to initialise the database connection. To handle multiple tasks the application makes use of multi-threading, with each important task being assigned its own thread.

Internal data structures To aid interoperability in the system, a Python file called `globals.py` is used to store a variety of variables that are needed by the main controller, the server, and the database. This file reads the application configuration file which specifies the host, port, and password for both the web configuration and the OBS connection. Additionally it stores what current game is being played, the reference to the database connection, a mutex which is used to ensure thread-safe database operations, and the dictionary object that is sent to both the web configuration and the live stream overlay. The object has the following structure:

```
{
    "player_count": int,
    "winner": int,
    "pointers": {},
    "players": {},
    "games": []
}
```

Figure 15: Dictionary object of information delivered by the web server

The web configuration needs to know how many players to offer inputs for, hence `player_count` is required and the live stream overlay needs to know whether or not to display the `winner`. Any pointers marked with `display: true` in the game configuration file are stored in the `pointers` object, the player names are stored in the `players` object, and the list of `games` is also required by the web configuration to switch between available games.

OBS connection First, a connection to OBS is established using the obs-websocket module. This socket is used to send scene change requests to OBS. To reduce the number of requests sent over the connection, a handler is set up to keep track of the current OBS scene. When scene change requests are sent to OBS it checks first whether the requested scene is the same as the current scene, and if so it doesn't send the request.

Games list Next, the list of supported games is loaded. The list is contained in a JSON file and an example entry is shown in Figure 16. The object contains the `name` of the game that is displayed to the user and the `file_name` of the associated configuration. The database connection is then established.

```
{
    "name": "SuperTuxKart",
    "file_name": "supertuxkart.json"
}
```

Figure 16: Object in the list of games

Controller GUI Next, the GUI of the controller is drawn using the Tkinter module. As shown earlier in the Section 6, the UI is very simple, featuring buttons for each game and an indicator for which game has been selected. Tkinter supports adding labels and buttons to the UI window, where buttons can be assigned an associated function to execute when pressed. When a button is pressed it changes the current game ID variable which is then detected by the main thread. Additionally, when a game that is being tracked is closed, an alert is displayed to the user informing them that the application is no longer tracking any game.

8.4.2 Main controller loop

The main loop is where the information is extracted from the game, processed, and displayed. The loop makes use of two threads, one which continually checks which game has been selected by the user, and another we will call the 'game loop' which continually scans the memory of the selected video game and processes the data received. The game loop is controlled by a global boolean variable `loop` which dictates whether or not to continue processing the selected video game. When a new game is selected by the user, the variable is temporarily set to `False` which stops the processing from continuing and closes the thread. A new thread is then created which processes the newly selected game and `loop` is then set back to `True`.

When a game is selected, the corresponding configuration file is parsed and the loop begins. The user can specify how fast the game loop should run in the configuration file, where setting it to 0 will poll at the max speed possible and setting it higher will rate-limit the system. This is to allow the users to select how much system resources should be used by the application so as to not overload a system. The loop will continue until a new game is selected, the game is closed, or the application is closed. At the start of each loop, the value each pointer specified in the configuration file is checked. The `subprocess` module module is used to run the memory reader application and to pipe the output of the application's `stdout` into a dictionary variable. The platform listed in the game's configuration is used to determine what memory reader executable should be opened (x86 or x64).

After all the memory values have been read and stored, the loop then deals with the 3 static functions discussed earlier: gathering the player count; checking if a match is finished, and displaying the winner; and checking if the game is paused. Each of these are described using logic in the configuration file which needs to be parsed. As described above there are three possibilities that can be represented in the object to be parsed; an integer which represents just an integer, a string which represents the name of a pointer, and a dictionary object which represents some operator between two additional objects.

Two functions were created to help with this. `check_json_object()` is used to check what type of variable the object is. If it is an integer, it will simply return the integer value, if it is a string it will find the corresponding pointer value, and if it is a dictionary object it will call `check_logic()` with both the values of the list given in the dictionary value as parameters. `check_logic()` calls `check_json_object()` on both of the parameters passed to it and performs the operator described in the key of the object. The recursive calling allows for nested operations to be implemented with a variety of different operators, giving additional expressiveness to the configuration files.

The program initially calls `check_json_object()` on the first object it finds, and from there it will process the full logic statement written in the configuration. The function for player count then updates the player count; the match finished check then updates the winner variable to be reflective of the games winner; and the game paused checker will send a request to OBS to change the scene to the `paused` scene if it is paused, if not it will then swap back to the `game` scene.

8.4.3 Web server

The Flask web server is simple in design but has a few design quirks that are important to discuss. The first is the use of the module Flask CORS. To properly display HTML elements on the live stream using the live stream overlay, Cross Origin Resource Sharing

(CORS) must be enabled. Without CORS, additional script files and other resources such as images and CSS will be rejected by most web browsers. Additionally, caching is disabled in Flask, web browsers tend to cache the JSON files that are served by the Flask application which means that updates won't be displayed properly on the live stream. The web server also switches between SSL and regular HTTP depending on whether or not a certificate and key had been provided.

The web server serves all the overlay files listed in the overlay directory, as well as the web configuration page at the index of the server. A login page is displayed by default when trying to access the configuration, the user must first input a password which is specified in the application configuration. The server processes requests made by the web configuration, such as storing the newly entered player names or changing the game currently being processed.

8.4.4 Database

The database implementation is relatively simple as the system doesn't require much persistent storage. It uses only one table with two entries, a primary key `username`, and `score`. When a winner is declared, the score is incremented by 1, or if the user doesn't exist, an entry is created with the score initialized to 1. Database operations are done across multiple threads, hence the SQLite connection is told not to ensure queries are made on the same thread. To ensure thread safety, each transaction obtains a mutex prior to executing.

8.5 Web configuration

The web configuration is implemented as a HTML page with Javascript elements. It communicates with the web server directly and with OBS using obs-websocket-js, another OBS web socket API almost identical to the Python module. JQuery is also used to simplify element selections and sending of additional requests. Before accessing the configuration page, the user must first input a password on the login page.

The configuration displays drop-down menus that allow users to manually change the scene in OBS and also to change the game currently being tracked. OBS scenes are changed by sending a `SetCurrentScene` request over the connected OBS socket. Both game change requests and player name updates execute a POST request sent to the web server which will update the controller application with the respective information. The configuration requests the web server for the most up-to-date data when the page is loaded. This data contains the current player count and is used to dynamically create the amount of text inputs required.

8.6 Live stream overlay

The live stream overlay is a simple HTML page with functionalities implemented in Javascript. The live stream overlay also makes use of JQuery for selecting and modifying elements on the page. To stay up-to-date, the overlay must constantly poll the web server for new data. Player names and scores are appended to a container on the right-hand side of the page when available. Every time the server is polled the box is emptied and repopulated so that if the player count decreases, old entries are removed. If a winner has been declared a container is filled at the top of the screen displaying the winner, and after a short period of time is emptied again.

9 Evaluation

9.1 Primary

All four primary objectives have been achieved.

1. Create the back-end for the update mechanism.
    - The memory reader application is capable of reading data from reading data from video games and the outputs are piped into the controller application to be further used. The implementation of the back-end communication is discussed in Sections 8.3 and 8.4.
2. Create a customizable overlay for the video game that can be used to display changeable elements.
    - An overlay was created in HTML with Javascript which ultimately is used to display the information as it is produced. The implementation is shown in Section 8.5 and a demonstration of the overlay is shown in Section 10, Figure 27.
3. Create an automation system to track game progress and player information, and update the stream using the back end.
    - The controller automatically keeps track of the games progress, tracking who wins, when the game is paused, how many players are playing, and each player's information. A description of how this works is available in Section 8.4, and an example of the automatic tracking of score is shown later in Section 10, Figure 28.
4. Create a remote management application to manually change elements of a live stream.
    - The remote management application was created in the form of the web configuration which is accessible on any internet connected device with a web browser. A demonstration of the web configuration working can be found in Section 10, Figures 21 and 25.

9.2 Secondary

1. Use image recognition to extract information from the game.
2. Extend the system to support other video games using image recognition.
    - Due to the ethical considerations discussed earlier in the report, instead of initially supporting one game then branching off and supporting others, the

design decision to use memory reading gave us the framework to support a variety of games from inception. As a result, image recognition was no longer required to fulfil the goal of supporting multiple games. An example of multiple games being supported is shown in Section 10, specifically Figures 28 and 33.

3. Further automate the system by collecting player names through QR or RFID scanner.

    - As discussed in Section 1.2, due to lack of resources and complications due to the closure of the university, I was unable to begin work on a QR or RFID scanner. Work could have been done theoretically on static images of QR codes, but would not serve much of a purpose to the application as in practice it would require an attached camera to actually scan physical QR codes.

9.3 Tertiary

1. Integrate the software with the Twitch desktop app.

    - The Twitch desktop app is tailored towards specifically games and game mods, rather than desktop applications. As such it is not possible to deploy the application on the app. Instead, it is easier to use Python virtual environments with the pre-compiled versions of the memory reader application. The memory reader application is not easily compiled compared to installing the Python requirements and running the scripts, hence the pre-compiled memory reader executable files have been provided.

9.4 Quantitative evaluation

As the user can specify how often the application should read the memory values of a video game, the performance of the system varies based on configuration. However, assuming that the system is running at the maximum rate, it is possible to ascertain roughly how well the main application loop performs.

To test how well the application performs, one of the largest potential bottlenecks is how the controller application obtains the data from the game. The controller application runs the external memory reader process using the built-in subprocess module which creates additional overheads in the application.

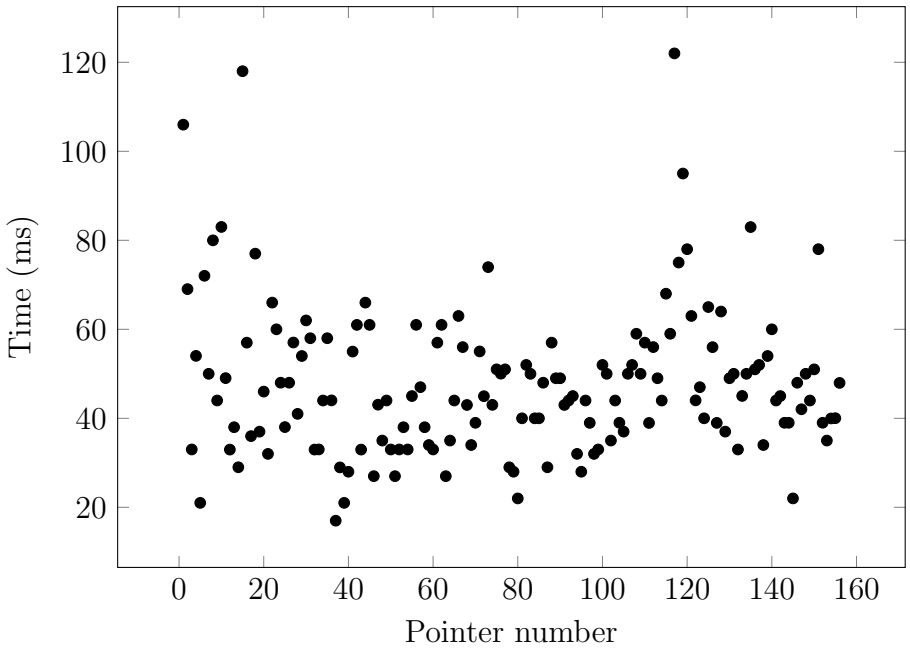


Figure 17: Time taken to read a pointer value in the controller application

Figure 17 above shows the time taken for a variety of calls to the memory reader application. The mean time taken to read one pointer address is 48ms and has a relative standard deviation of 35%. This means that on average it can read and store 20 pointer addresses per second. Conversely, Figure 18 below shows the time taken for a variety of memory reader application executions. This time, the mean time taken is only 1ms with a relative standard deviation of only 6.3%, meaning that calls using the subprocess module increase the overhead by a factor of 43.

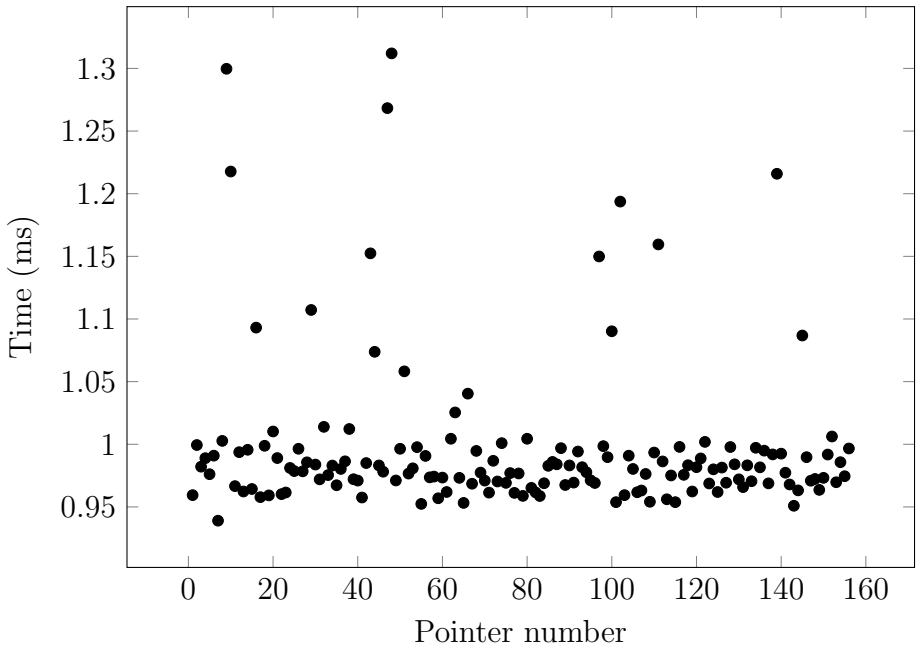


Figure 18: Time taken to read a pointer value in the memory reader application

The consequences of implementing the controller application in a high-level language like Python were clear from the start and were expected from the initial design phase. Implementing the full application in C++ would have made the application more efficient, meaning more values could be read in a shorter time window. The original goal for the project was to read data in real-time from a video game and use this data to automate a live stream. Though a large bottleneck is introduced by the use of Python, the application can still read memory values in real-time and respond to those changes within a reasonable time-frame. An alternative to implementing the controller application in C++ could have been to assign a priority to each pointer value to be read, ensuring that the most important values were read in as real-time as possible, though this is likely just a patch on a much larger problem.

10 Demonstration

This section will take the place of the in-person demonstration initially planned for the project. I will display evidence of the system working as intended, as well as discussing how testing was performed, with various use cases. Written statements with images cannot show off the full functionality of the project, such as showing the automation system working in real-time, however they can give a good indicator as to how the program executes and can show important events as they occur.

Earlier in the Section 7 of this report, we discussed how to obtain pointer values from Cheat Engine that are used by the application to read points in memory. In the in-person demonstration an example of finding these pointer values would have been shown, so it is recommended to follow that write-up written earlier in its place.

Prior to starting the application, it is important to first open OBS so that the application can interface with the software. The image below shows what is displayed when the controller application is run for the first time. The console informs the user that a connection with OBS has been established, and that the Flask web server is running. The GUI draws the list of games as a column of individual buttons, asking the user to select one.

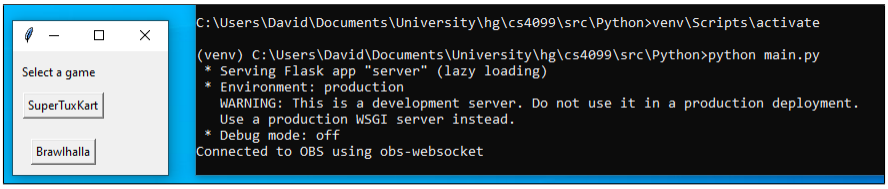


Figure 19: Entering the virtual environment and running the application

After the application has been run for the first time, we can set up OBS to display the overlay we created to display information on the live stream. In OBS we create a scene called 'game' which will display the game running and the information overlay. In this scene we add a 'Browser Source' which is used to display an HTML webpage. Figure 20 below shows the created Browser source with its URL being the web overlay hosted by the application. Additionally we can create a 'Game Capture' source which will capture the video games being played, and a second scene called 'pause' which will be switched to when the game has been paused.

It is worth noting here that the display outlined in red in Figure 20 is what is actually broadcast live online. In OBS it can be configured to output the live video onto any public live streaming site, such as YouTube, Twitch.tv, Facebook, etc. Later in the demonstration we will put the stream live on Twitch.tv to show how it can be easily broadcast around the world.

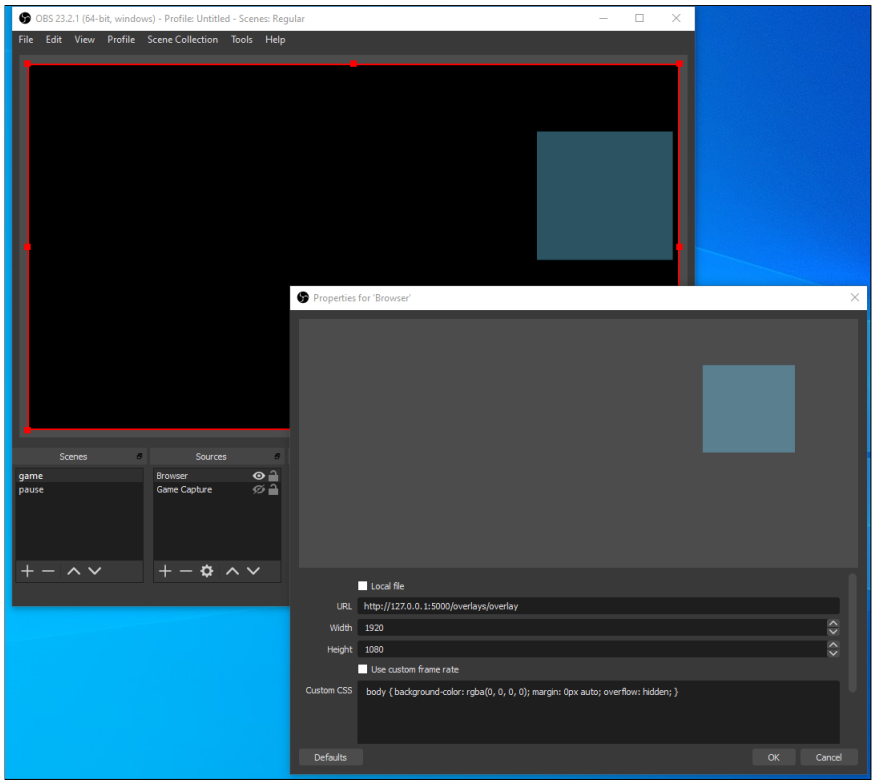


Figure 20: OBS application with a Browser Source displaying the web overlay

At this point, the web configuration page is already accessible. By going to the host and port specified in the `config.ini` file, the user can configure the application remotely. In this example we will access the configuration page from a second computer with access to the first computer's network. The web page can be accessed from mobile devices as well. When accessed for the first time, the page requires a password from the user to access the page. Failed attempts at password entry are rejected with an error message displayed.

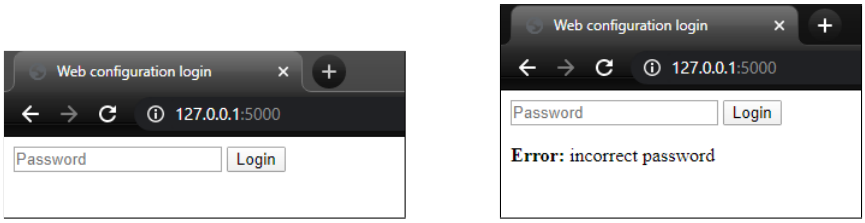


Figure 21: The login page with a failed login attempt

After successfully logging in, the configuration page is displayed. From here both the current OBS scene and the game to track can be changed. As no game is currently being tracked, the page doesn't know how many players are currently playing so the inputs are not displayed.

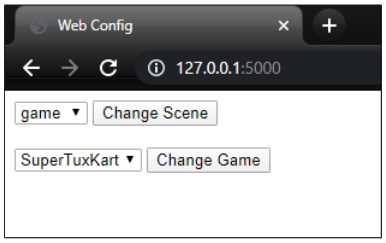


Figure 22: The default web configuration page

To read data from a game, the game itself must first be open. First I will demonstrate the game 'SuperTuxKart'. For this example I will open the game and begin with two players in a locally hosted multiplayer game. Figures 23 and 24 below show two players being selected and the game being started. For SuperTuxKart, the number of players is determined after the race is started so for this example we will first begin the race then make changes to the game configuration.

n.b. the race does not need to be in progress for the controller application to function properly, it is only done so in this example so that the web configuration can detect the correct number of players

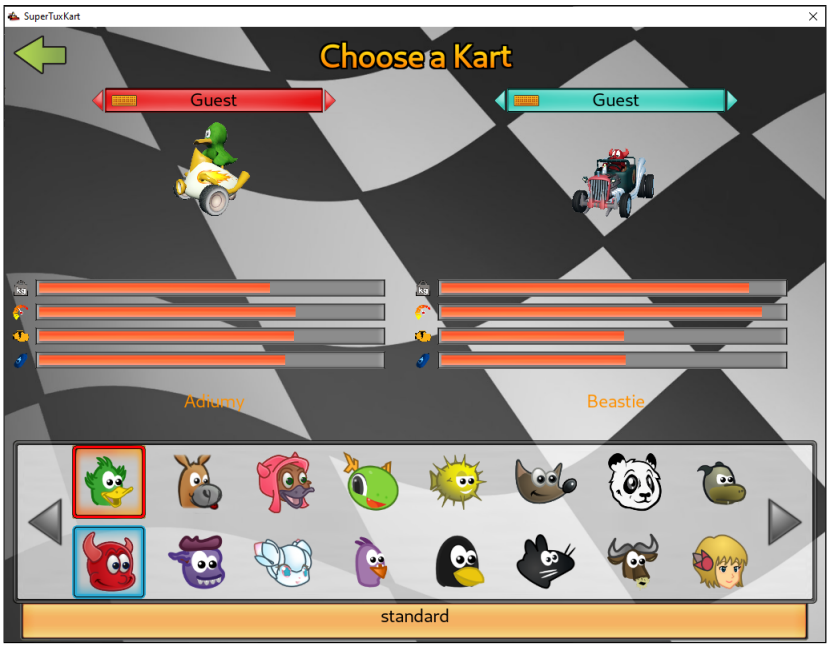


Figure 23: A multi-player game of SuperTuxKart with two players

Figure 24: The game being started with two players

Now that the game has been started, we can tell the controller application to begin

processing. Selecting 'SuperTuxKart' from the game drop-down menu and pressing 'Change Game' selects the SuperTuxKart game in the controller application and starts reading data from the game. The configuration page automatically updates the page with inputs for the player's names, and the controller GUI marks the game as being selected. We can also select the game directly from the controller GUI itself.

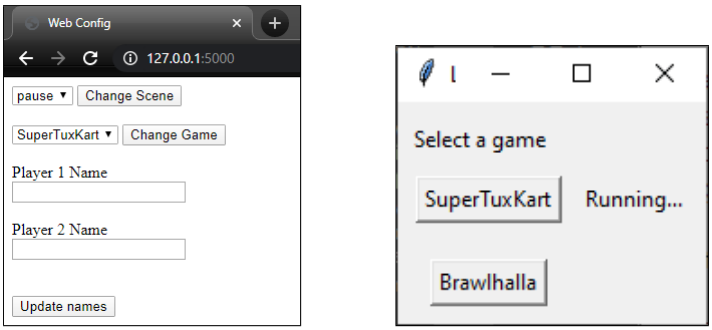


Figure 25: The web config showing inputs for both players, and the GUI displaying the running game

Using the web configuration we can now input the names of each player and have them displayed on the live stream. In this example will name Player 1 'Alexander' and Player 2 'Martin'. The use of the web configuration page demonstrates that primary objective 4 has been met.

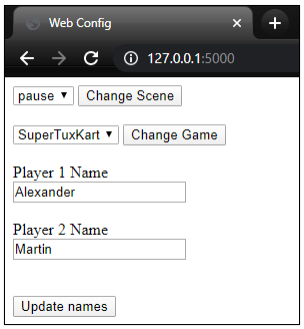


Figure 26: Inputting the names of each player on the web config

To show the outcome of changing the names above, we will now start actually live streaming with OBS. I have configured the application so that the video output is broadcast live to my profile on Twitch.tv. Figure 27 below shows the game being broadcast live on Twitch.tv. On the right side of the screen you can see the blue

box containing the names of each player as well as their respective score. The overlay displayed here on the live stream shows that primary objective 2 has been met.

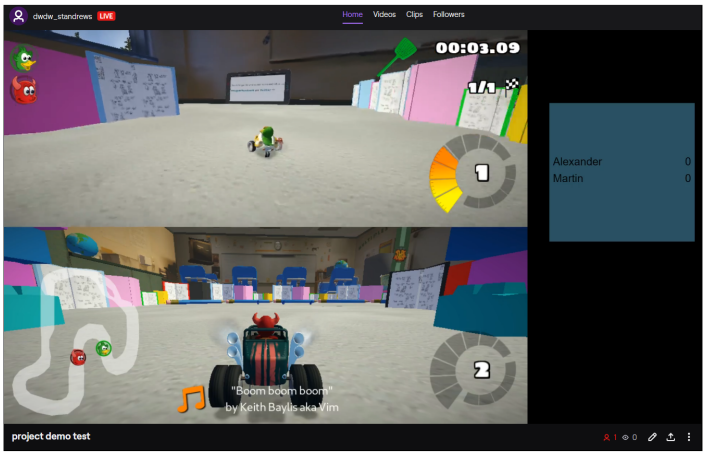


Figure 27: The game being streamed live on Twitch.tv with the overlay displayed

To show the winner being displayed and the score being updated, the race will be completed with 'Alexander' finishing first and 'Martin' coming second. Figure 28 shows the live stream on twitch showing Alexander having won the race, and updating his score total on the right hand side. Being able to extract useful data from the video game and use the data in some meaningful fashion demonstrates that primary objective 1 has been met, and using this data to keep track of the game's score shows that primary objective 3 has also been met. Henceforth, all primary objectives have been achieved.

Figure 28: The winner being displayed and the score being updated

When a game is not in progress, for example if the game has been paused or the users

are changing settings, then the OBS scene will be changed to 'pause' and the pause screen will be displayed. Figure 29 shows the live stream when the game has been paused. For this demonstration the scene is basic, displaying only static text, but the scene itself can be modified to however the user sees fit; for example, if they wished to change to a webcam showing the two players, or display a graphic or a video.

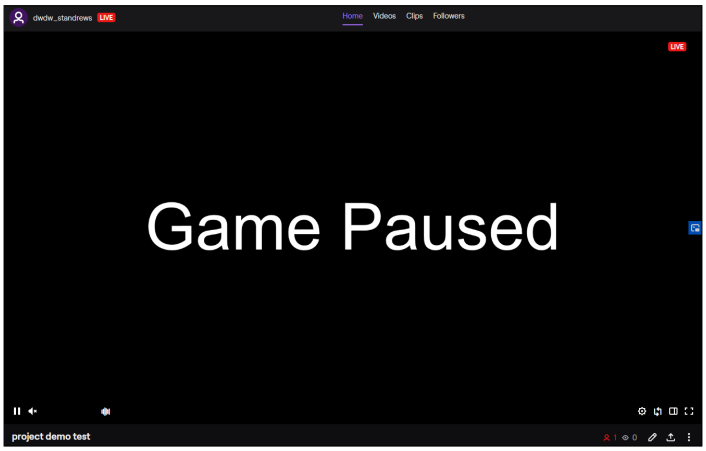


Figure 29: The winner being displayed and the score being updated

Now we will show how the application handles the game being closed, and swapping to a new game as described in secondary objective 2. In Figures 30 and 31 we close the SuperTuxKart game and show how the controller application displays a message to the user informing them that the process has been lost. From here we boot up the next game, Brawlhalla, and start tracking it using the application.

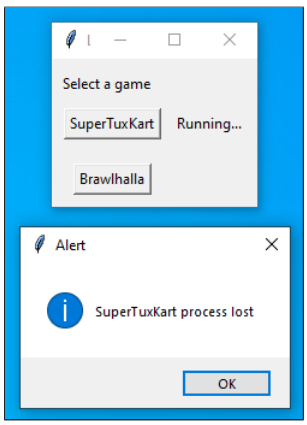


Figure 30: Alert being shown when the game process has been closed

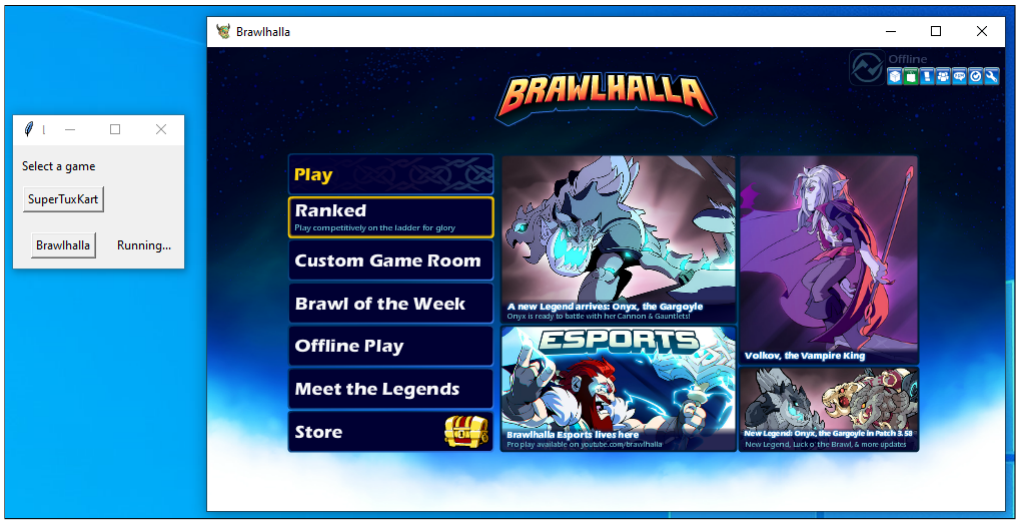


Figure 31: The application tracking the new game ‘Brawlhalla’

Again we will change the names on the web configuration, this time to ‘Jay’ and ‘Alexander’. These changes will then be reflected on the live stream. Note that Alexander’s score has been saved despite changing between games and changing from Player 1 to Player 2. This time we will have Player 2 win the game, and as you can see again the winner is displayed on the live stream and the score updated. The application is able to track multiple games, henceforth secondary objective 2 has been met without requiring image recognition as described in secondary objective 1.

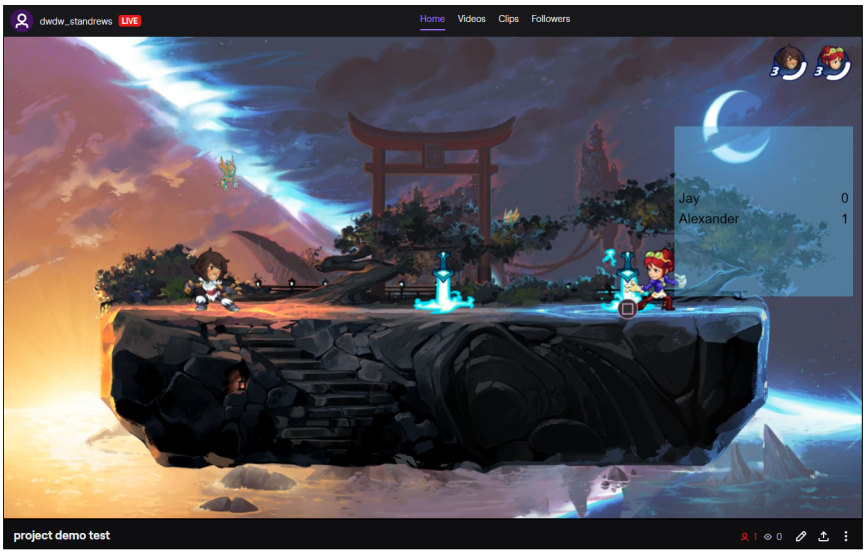


Figure 32: The Twitch stream displaying the new game Brawlhalla and displaying the score

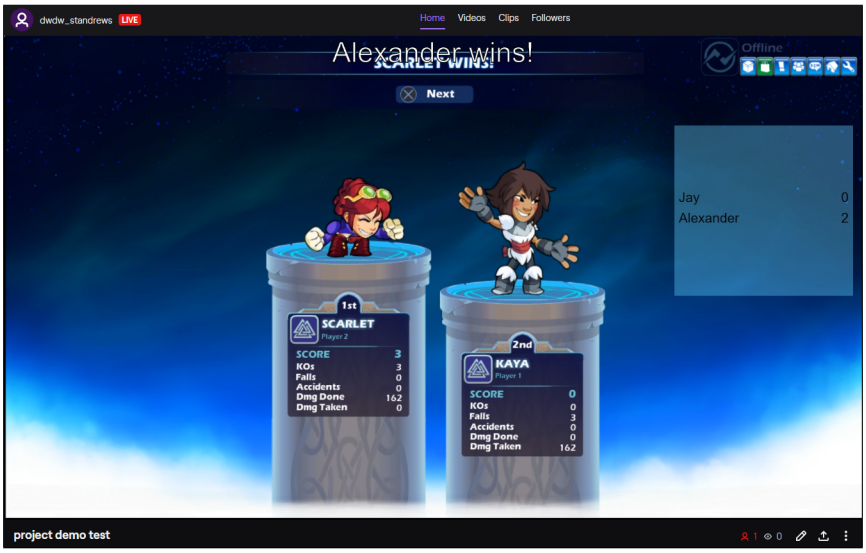


Figure 33: Display of the winner, Alexander, and the updated score

I have demonstrated that all components of the system work as expected and that all the primary requirements have been met. The application can read data from multiple video games, can control a web overlay to display information on a live video broadcast, can be configured remotely using a web server, and can make changes to the live streaming software itself.

11 Testing

The system relies on very loose coupling between components. As such, it is important to test extreme and exceptional cases in both the memory reader application and the controller application, to ensure that both systems can handle potentially erroneous data.

11.1 Memory reader application

The memory reader application has checks in place to make sure that the correct amount of parameters has been given to the application to ensure that no crashes occur. If a process cannot be found the application will exit and return an error. The application makes use of the function `strtoul()`, which converts the type `string` to an `unsigned long` to process the parameter inputs. The program expects inputs of the form `0x00000000` to represent memory addresses, but the function `strtoul()` can accept any input.

In Figure 34 we can see that we have given the `memoryreader32.exe` application some bad inputs that do not represent a valid memory address or offset. As a result of this, the memory reader application doesn't crash but merely spits out nonsense numbers as it attempts to convert the strings to memory addresses. This means that the application can deal with failure, and puts the control in the hands of the game configuration files which specify these offsets. Malformed configuration files wouldn't work regardless, but they wont crash the system.

```
C:\Users\David\Documents\University\hg\cs4099\src\Python>memoryreader32.exe SuperTuxKart supertuxkart.exe bad_string1 bad_string2
0

C:\Users\David\Documents\University\hg\cs4099\src\Python>memoryreader32.exe demonstration explorer.exe test dra erte
-1754022121
```

Figure 34: Giving the memory reader application badly formed inputs

11.2 Controller application

One of the potential issues with regular operation of the application comes from how memory is allocated in certain games. Some values that are tracked might not be assigned to memory until a certain point in execution, leaving the existing pointer values specified in the configuration files to point to random values. For example, when SuperTuxKart is initially ran, the values of the following pointers are shown in Figure 35. The seemingly random values for a variety of values could potentially result in some false-positives being detected by the controller application, such as if `results_screen` was somehow set to 1 and the controller interprets that as the game ending. However, during the limited user testing that was able to be performed, no

such error has occurred. The values are all set to their expected values when the match begins.


```
{
    "first_place":"-1116527396",
    "no_of_ai":"-1116527396",
    "no_of_players":"1",
    "p1_position":"-1116527396",
    "race_in_progress":"0",
    "results_screen":"-1116527396"
}
```


Figure 35: SuperTuxKart pointer values when initially ran

As far as I can see there is no way to remedy this issue. The way that the games are implemented is outside of our reach and we must make use of the values that the game outputs. It is possible to prevent the application from tracking erroneous values such as the negative numbers shown in Figure 35 by adding more logic statements to the game configuration files, but it would overcomplicate the files greatly and impede the readability of the files.

To demonstrate how the application handles bad JSON configuration files, I created 4 malformed files and added them to the list of games.

1. File not found — the file specified in the list of games doesn't exist
2. Malformed JSON — the JSON file is malformed, it cannot be decoded properly
3. Bad pointers — Essential data is missing from the pointer list
4. Bad platform — Platform is not 32 or 64 bit

Figure 36 shows the errors displayed to the user for each of the situations described above. The application can handle the four types of malformed configuration files and displays an appropriate alert to the user. The application can continue execution after the fact and doesn't crash.

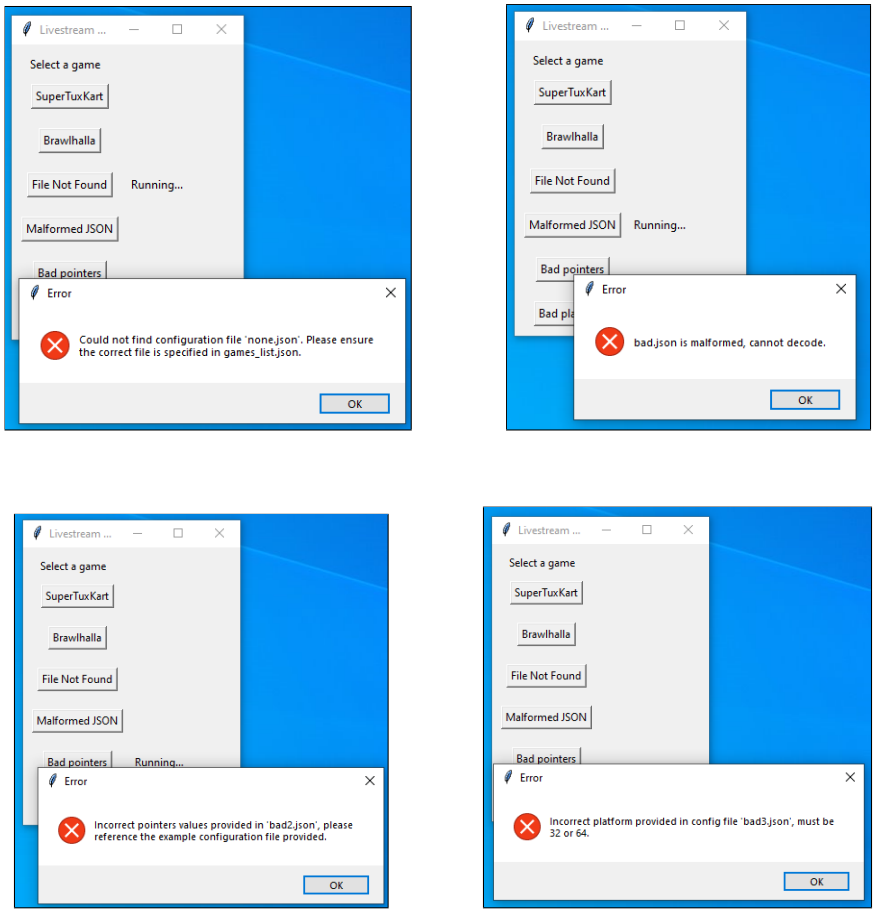


Figure 36: Results of the four tests

11.3 Web configuration

Since the web configuration is relatively simple and largely decoupled from the rest of the system, not many errors can occur. The configuration page is able to handle instances where it cannot connect to OBS using obs-websocket. Instead of crashing it prevents the user from sending any scene change requests by hiding the input elements. Another possible interaction between the controller application and the web configuration is that the erroneous values for 'Player Count' could result in thousands of input boxes being created for each player on the web configuration page which would likely crash the page. To prevent this, I put a cap on how many player inputs there could be on the configuration page to 16 inputs. This provides enough inputs for larger team-based video games without going off the deep end.

11.4 User testing

Due to the closure of school facilities, I was unable to test the application with different participants and ascertain useful feedback. As discussed earlier in Section 6.2, the application isn't designed to be used with online multiplayer games, hence remote user testing was not a viable option either. Limited testing was done locally with myself controlling two players at one time which provided ample results, but doesn't reflect the system being used to its fullest capacity.

12 Discussion

12.1 Reflections

This project was a foray into the world of software development for myself. It gave me an opportunity to put into practice the skills and techniques I have learned over the last 4 years, as well as learning new ones. I got the opportunity to display my knowledge of a variety of different programming languages and architectures and got to apply it to an project that is deeply rooted in my interests.

The project gave me experiences with familiar technologies, such as Python, Javascript, and both front-end and back-end web development. It also gave me a chance to delve into new areas, such as memory reading and using C++ largely for the first time, as well as using external tools like Cheat Engine to search for relevant memory addresses. I am glad I went with a familiar language like Python for the base implementation as it gave me a comfortable platform to implement my ideas upon without having to worry too much about learning new notation. Python gave access to a variety of libraries and modules that let me implement the basics easily, such as a web server and socket communication with OBS, so that I could worry more about interactions between components and the specific implementation of my ideas.

I found the software development process that me and my advisor followed to be very effective. The weekly sprints coupled with weekly meetings allowed for ideas to be refined, issues to be caught early, and the clear setting and achievement of goals. The issues raised early regarding ethics gave me an insight into the challenges faced by real software developers and gave me room to explore ideas I never had considered before.

Overall, I am satisfied with the outcome of the application. The application meets all the major goals I wanted to achieve with the project; a fully working automated system which can be controlled remotely and supports a variety of different video games. I am confident that the application will have an impact on video game live streaming in future, especially for those who lack the tools required to compete with the most popular games.

12.2 Critical appraisal and future ideas

Though the primary and most secondary objectives have been met, it is worth discussing the benefits and drawbacks of the system, including comparisons drawn to other similar systems.

12.2.1 Complexity

The first thing that is important to discuss is the inherent complexity of reading data from the games themselves. To find the corresponding multi-level pointers for each item you wish to track is difficult and time-consuming owing to the complexity involved with sifting through a program's memory. It is time consuming and challenging to find the memory address of a value on its own, but in addition it can take a long time to process an appropriate multi-level pointer in Cheat Engine using the 'Pointer Scan' feature. As discussed earlier, the space and time complexity of the search is $\mathcal{O}(b^m)$ with $b = max\ offset$ and $m = max\ depth$ which can make it infeasible to find extremely specific values in a game's memory.

There is a trade-off between complexity and extensibility in regards to using the memory scanning approach. One one hand, it is complex to work out values for specific video games, but on the other hand it is possible to support any video game using this approach and doesn't require any re-programming. The alternate approach used by Metascouter [12] which uses computer vision to read data also suffers from the same trade-off as the approach employed in this project, as it is also challenging to implement a vision system for each video game the project aims to support. Overall, I would say that the trade-off is worth it by the virtue that extracting data from specific video games also requires some complicated system be built on top of the video games themselves. For example the Slippi Framework [16] offers a simple API to interface with the framework itself, but the implementation of the framework is complicated and took many years to develop and it only applies to one specific game.

12.2.2 Domain-specific language

Building upon the complexity of the games configuration files, it is worth taking a look at how logic is implemented in the JSON configuration files. The JSON format is not normally intended for the declaration of logic statements, it is by definition an 'Object Notation' and is used for data interchange, not for programming. The implementation of the logic using JSON as a specifier suffers from the issues that most other configuration files suffer; unclear syntax, semantics, lack of debug capability, and lack of clear documentation aside from examples [9]. The implementation of logic should be done using a programming language or a domain-specific language. Martin Fowler states that "a Domain-Specific Language (DSL) is a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem" [10] which is exactly what is required by the application. To aid readability and reduce complexity, it would have been beneficial to implement a domain-specific language that more naturally supported logic statements including pointer values and integers. Though the JSON system works, the notation is quite confusing and could benefit from such a change.

12.2.3 Security

Security is an important part of any application and there are a few points related that must be discussed. As mentioned in Section 8.4.3, the web server supports both HTTP and HTTPS depending on whether the appropriate certificates have been provided. However, the OBS web socket password is sent unencrypted over the connection and is stored in plaintext. A fix to this could be sending OBS update requests to the web server itself and having it forward them on to OBS rather than implementing it unsecured in Javascript. The login info input on the configuration page is protected by SSL (if enabled) but the actual local storage of the password is merely stored inside the configuration file for the system. It could be an idea to store the password information inside a secure database and ask the user themselves to login when the application is started, rather than storing it in an unprotected plaintext file.

13 Conclusion

Online video broadcasting is a massive market. In the second quarter of 2019 alone it was reported that 3.77 billion hours of content was watched across the four main live streaming websites [14]. Yet of these hours watched, over 50% goes towards the same 10 video games [19], and a whopping 74% of Twitch viewership goes towards the top 5,000 channels [14].

This project aims to provide a system that aids smaller channels and smaller video games, giving them the necessary infrastructure to run tournaments for these video games. The application gives users a means to automate a large portion of video game tournament live streaming by keeping track of scores automatically and making necessary changes to the live stream itself. For parts that cannot be automated, the application offers a web configuration which allows users to modify elements of the live stream remotely from their mobile devices. The user is decoupled from the menial tasks of inputting scores and can instead focus on other activities related to running video game tournaments.

The application has made use of a variety ideas and technologies that exist today but has built them together to create a system that is not readily available to the masses. My hope is that this application will be useful to those who wish to run video game tournaments in future, but previously lacked the means to show them off. Future aspirations for the project would mainly include reducing the complexity of configuration files so that the system can be more easily expanded to more games.

References

[1] Chris Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, 2006. ISBN: 1401302378.

[2] Meeyoung Cha et al. "I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System". In: *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*. IMC '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 1–14. ISBN: 9781595939081. DOI: 10.1145/1298306.1298309. URL: https://doi.org/10.1145/1298306.1298309.

[3] *Cheat Engine*. Online; accessed 24-April-2020. URL: https://www.cheatengine.org/.

[4] Frank Cifaldi. *The Story of the First Nintendo World Championships*. Online; accessed 14-October-2019. 2016. URL: https://uk.ign.com/articles/2015/05/13/the-story-of-the-first-nintendo-world-championships.

[5] Dave Consolazio. *The History Of Esports*. Online; accessed 14-October-2019. 2018. URL: https://www.hotspawn.com/guides/the-history-of-esports/.

[6] *Defend Your Right to Repair!* Online; accessed 25-April-2020. URL: https://www.eff.org/issues/right-to-repair.

[7] *ELEAGUE Major 2018 detailed stats — Esports Charts*. Online; accessed 14-October-2019. 2018. URL: https://escharts.com/tournaments/csgo/eleague-major-2018.

[8] Cat Ellis. *The best free streaming software 2020*. Online; accessed 24-April-2020. URL: https://www.techradar.com/uk/news/the-best-free-streaming-software.

[9] James Fisher. *Configuration files suck*. 2014. URL: https://hackernoon.com/configuration-files-suck-6daa9812f601.

[10] Martin Fowler. *DomainSpecificLanguage*. 2008. URL: https://martinfowler.com/bliki/DomainSpecificLanguage.html.

[11] Mark R Johnson and Jamie Woodcock. "The impacts of live streaming and Twitch.tv on the video game industry". In: *Media, Culture & Society* 41.5 (2019), pp. 670–688. DOI: 10.1177/0163443718818363. eprint: https://doi.org/10.1177/0163443718818363. URL: https://doi.org/10.1177/0163443718818363.

[12] *Metascouter*. Online; accessed 07-March-2020. URL: https://metascouter.gg/.

[13] *Open Broadcaster Software*. Online; accessed 24-April-2020. URL: https://obsproject.com/.

[14] Sarah Perez. *Twitch continues to dominate live streaming with its second-biggest quarter to date*. Online; accessed 10-March-2020. 2019. URL: https://techcrunch.com/2019/07/12/twitch-continues-to-dominate-live-streaming-with-its-second-biggest-quarter-to-date/.

[15] Rainforest Scully-Blaker. "Re-curating the Accident: Speedrunning as Community and Practice". Sept. 2016. URL: https://spectrum.library.concordia.ca/982159/.
[16] *Slippi*. Online; accessed 11-March-2020. URL: https://slippi.gg/.
[17] *Steam*. Online; accessed 25-April-2020. URL: https://store.steampowered.com/.
[18] *Streamlabs*. Online; accessed 24-April-2020. URL: https://streamlabs.com/.
[19] *SullyGnome - Twitch statistics and analysis*. Online; accessed 23-April-2020. URL: https://sullygnome.com/games/2020february/.
[20] *Twitch Bounty Board Program*. Online; accessed 25-April-2020. URL: https://help.twitch.tv/s/article/bounty-board-program-information-and-faq.