Blade II Online: Development of an Online Multiplayer Adaptation of Blade II

by

James Einosuke Stanton

K1611060

Supervised by

Dr. Jarek Francik

A dissertation submitted in partial fulfilment of the requirements for the degree of

BSc (Hons) Games Technology

Presented to the faculty of the

School of Computer Science and Mathematics

Kingston University

2019/2020

Contents

Glossary of Terms

Bolting (Bolt, Bolted)
The act of deactivating the card that was most recently added to a player's field. The card with this effect is known as a *Bolt* card, and a card that was deactivated in this fashion is referred to as being *bolted*.

CCG
Collectible Card Game – A card game that features a range of different cards, that can be strategically combined into a deck. Typically features a way of pitting one deck against another in a game with a particular set of rules.

DDOS attack
Distributed denial of service attack – A malicious attempt to disrupt normal traffic to and from a web service, network, or server, by flooding the target, or its surrounding network infrastructure, with internet traffic such as requests for information, or with malicious traffic patterns.

Edge server
A server that exists at the logical, and often physical, edge (the point at which one particular network ends) of a network.

Game (In relation to Blade II Online)
Blade II Online itself – the game, and all its local and online components in their entirety.

Match (In relation to Blade II Online)
A match is a single instance where a player plays against either an online opponent, or an AI opponent.

NGINX
NGINX is a web server that offers functionality such as a reverse proxy, or file serving.

Nihon Falcom
A Japanese Game Developer, established in 1981, famous for its RPG's.

Origin server
A server that receives, processes, and responds to traffic from clients on the internet.

Reverse proxy
A server that receives traffic on behalf of a web server, before forwarding it to the intended recipient.

The Legend of Heroes
A series of single player story RPG's developed by Nihon Falcom. Includes games such as Trails in the Sky, and Trails of Cold Steel.

Tokyo Xanadu eX+
A single player story ARPG developed by Nihon Falcom.

Traffic

Internet communications, such as HTTP requests for web pages.

Trails of Cold Steel I/II

A single player story RPG developed by Nihon Falcom.

Introduction

Summary of the Game

Blade II Online is an online multiplayer card game designed to offer fast-paced gameplay with minimal downtime. With a rapid play cycle and an Elo based high score system that updates after every match, Blade II Online presents an exciting gameplay experience with a tight feedback loop and easy to understand mechanics regardless of player ability. When playing online, players will quickly find themselves competing against other players of similar skill levels and will be able to track their progress in real-time as they continue to play the game.

Aim of the Game

During a match, players take turns selecting a card, while attempting to end the game with a higher score than their opponent.

Outside of a match, players will be able to interact with the launcher application, where they will be able to perform tasks such as creating accounts, resetting passwords, viewing the ranked leaderboard, and viewing their match statistics (wins, losses etc.).

Objective of the Game

When a match begins, each player is provided with a random set of cards. These cards are placed in the respective player's decks, from which each player then draws cards into their hand. Depending on the state of the game, they must take turns selecting a card from either the deck or their hand, in an attempt to end the game with a higher score than their opponent.

There are many triggers that can cause a game to end, such as when a player overcomes their opponents score, on their own turn, while simultaneously putting their opponent in a position where are no legal moves for them to play, or when both players run out of cards to play.

Personal Aims and Objectives

My main goal for the development of Blade II Online, was to produce a fully functional prototype that functions as a proof of concept for an online version of Blade II. This will become the centrepiece of my portfolio, that I will use to demonstrate my skill and knowledge as a game developer when I start job hunting in the near future.

I plan on applying for jobs at multiple game development companies in Japan, and there is one particular company that I consider to be my first choice. In attempt to boost my chances of success, I wanted to create something that I could present to them during the application and/or interview stage, in order to demonstrate both my ability, and my passion for game development.

The company in question, Nihon Falcom, is the company that developed the games within which Blade II minigame is featured. After reading an interview with the company president, where they discussed how they ended up working for the company after creating a fan site for one of Nihon Falcom's games (Kondo, 2019), I began to believe that being able to present a portfolio piece that directly relates to their most popular franchise will greatly increase my chances of being selected, serving as sincere proof of my desire and drive to work for them.

Of course, fervour alone will not get me a job, and so along with the skills, knowledge, and experience that I have accumulated throughout my time with Kingston University, I strove to make full use of the time I spent developing Blade II Online.

Lastly, I wished to improve my C++ programming skills. During my year-long placement at Sony, I found myself writing mainly JavaScript and Python code during development. While these languages both have their merits and use-cases, they are uncommon when it comes to console and PC game development, where C++ and C# are in the majority.

I already have a few years of experience using C# and consider myself to be relatively proficient. I began my journey of developing games with online tutorials that focused on Unity and C#. Since then I have had many opportunities to develop games using Unity at university and feel confident that I can implement anything that I might need with Unity's C# scripting interface.

Conversely, I have had very little opportunity to develop games using Unreal Engine 4 and have had only a few chances to write C++ code since the end of the first year of university. This is essentially my last chance to do so before I start job hunting, and I desperately wanted to give myself the opportunity to developing a major C++ project, as well as being able to add it to my portfolio.

Document Summary

This document aims to present details regarding the planning, development, and analysis of Blade II Online in a clear and concise manner. At its core, this is a technical document, with the purpose of highlighting the intricate details of Blade II Online, whilst drawing attention to particularly interesting or difficult to implementation details of the entire project, from the desktop launcher, to the online services used to power the back-end infrastructure. However it also covers topics such as game design, look, and feel, as well as looking back on, and critically reviewing the project and its development.

Literature Review

Review of Similar Games

There are many comparable products available on the market today. Many of them are digital versions of existing card-games (such as Magic: The Gathering Arena, and Pokémon TCG Online). Some are based on existing characters and lore (such as Hearthstone) whilst others are largely original concepts (such as Shadowverse). Other notable examples include Microsoft Solitaire, which was introduced in Windows 3.0 to familiarize users with various methods of interacting with a PC using a mouse (Caldwell, 2019).

Digital trading card games have existed in various forms since the late 90's, with some of the earliest examples being Magic: The Gathering, released in 1997 (Blevins, 2000) and Pokémon Trading Card Game, released in 1998 (Nintendo, 1998). Online Digital trading card games have also existed since the late 90's, with notable examples being Sanctum (Jebens, 1998), and Chron X (Bregger, 2019), both of which were released in in 1997. Though these are all examples of games that feature card-collection mechanics, Blade II Online will feature a single, static set of cards in order to stay true to the original game upon which it is based.

Hearthstone

Hearthstone is an online CCG developed and published by Blizzard Entertainment. It was released on March 11th, 2014 (Zeriyah, 2014) for Windows and MacOS (Yin-Poole, 2014) and has since been made available on iOS and Android (Kollar, 2015). The game is built using Unity (Wilson, 2014).

Hearthstone is 'fast paced' and 'is a game that's designed for anyone to come up, to pick up and play; to be a very accessible experience' (Brode and Chayes, 2013), and Blade II Online will attempt to echo these sentiments.

Fast, easy to learn gameplay with simple rules is a key design goal for Blade II Online, providing a low barrier to entry that requires no knowledge of the game universe from which it draws inspiration (The Legend of Heroes series), and very little understanding of card

games in general, whilst rewarding those that invest time into learning more advanced ways of playing, and the intricacies of the games rules and interactions.

In addition, Hearthstone, while presented as a 2D top-down card game, is in fact comprised of a combination of 2D and 3D assets and rendered entirely in 3D (Hearthstone, 2017, 04:02). 2D Paintings are mapped onto 3D models using Maya (Brode, 2013), and after angling models so that they appear natural in the single, top-down POV (Zwicker, 2013) are added into the game.

By presenting what would typically be a 2D game in three dimensions, Hearthstone is able to employ a variety of 3D effects without the need for complex shaders. Z-depth and layers are handled by world-space positioning, paving the way for cards with layers and inherent parallax effects, and indicators (such as arrows or special effects) that stand out from the background with their physical presence and accurate perspective effects.

In a similar fashion, Blade II Online makes heavy use of 3D assets to represent objects within the game. The cards, the arena, player avatars, and in-match indicators (such as the cursor for selecting cards) are all 3D models that are moved around in 3D space.

League of Legends

League of Legends is an online MOBA developed and published by Riot Games. It was released on October 27th, 2009 for Windows (Paah, 2010), and later other platforms. The game makes use of a custom engine built specifically for League of Legends, with the core game being written in C++. The game client, a separate application that acts as a homepage/game launcher, is built around the CEF (Chromium Embedded Framework) and uses a combination of JavaScript/HTML/CSS and C++ (Lima Beans, 2009).

Being a MOBA, League of Legends gameplay is completely different from the card-game that is Blade II Online. However, the ranked matchmaking system employed by League of Legends is a framework upon which many of the design decisions for the online multiplayer components of Blade II Online were based.

Players can queue up to find a game and will be automatically placed into games with other players of similar skill levels, earning or losing Elo rating (commonly referred to as MMR (Match Making Rating) or LP (League Points)) based on the outcome of the game.

The act of playing ranked games and testing one's mettle versus other players is both exhilarating and stressful. Earning and losing Elo rating, being promoted, or demoted between ranks, and knowing that every game contributes to one's personal rating creates an incredibly fun, though possibly addictive gameplay experience that is joined by millions of players worldwide (Lol Smurfs, 2018). Blade II Online aims to capture some of this thrill and excitement, whilst remaining true to the core themes of the Legend of Heroes series.

The second feature of League of Legends from which Blade II Online draws inspiration is the division of players by Elo rating into different tiers.

A tier is a range of Elo ratings that are grouped by a named category. In League of legends, players are divided up into nine separate groups based on their Elo ratings (from lowest to highest) – Iron, bronze, silver, gold, platinum, diamond, master, grandmaster, and challenger, of which most iron through diamond are further split into divisions.

Blade II Online implements a similar system, but due to the considerably smaller scale, only implements three tiers with no subdivisions. The tiers are Junior Bracer, Senior Bracer, and Grandmaster, and are based on the lore of the Legend of Heroes universe.

Tools and Technologies

Game Engine and Language

During the initial planning state, it became immediately clear that the main game client would have to be built using either Unity and C#, or Unreal Engine 4 and C++. Other engines do indeed exist, but they either do not offer as wide a range of tools as the two above, or do not offer first class support for both 2D and 3D workflows.

Also, due to the scale of the project, it was determined at an early stage that a familiar engine should be used, to avoid time being spent on tasks such as learning about the game's API, or trying to figure out which engine features are suitable for which purpose.

Network and Server Technologies

The Go Programming Language

Go is an open source programming language developed by Google.

Go was the language of choice for writing the Blade II Online REST API, and the game server, as it is well suited for building web servers due to its simple and efficient first-class support for parallelism and multi-processing. The Go runtime handles all the underlying logic in such a way that is completely transparent to the host OS. Context switching is very cheap (over 10 times cheaper than traditional threads) and inter-thread communication is lightweight (Hiwarale, 2018). This means that it's relatively straight forward to develop Golang servers that are capable of serving many concurrent users, even on limited hardware, as well as being very easy to scale.

Go has an excellent standard library, offering many high-quality tools for a vast range of applications, including a full http library. Furthermore, there are many well maintained open-source libraries available for free, such as the chi router, which is used in production by many large companies including Cloudflare and Heroku (*go-chi/chi: lightweight, idiomatic and composable router for building Go HTTP services*, 2019).

The open source Go libraries used by the Blade II Online game server and REST API are listed in the Go Libraries section of the appendices.

Amazon Web Services

AWS is a cloud platform with millions (Lunden, 2015) of users accessing services across the globe.

It was chosen due to the incredibly generous free tier offered to new users, for the duration of a year. Within this year, a user can effectively maintain 24/7 uptime for both a single low spec virtual machine and a single relational database instance in a flavour of their choice, as well as 1 million requests lambda functions and API gateways each month.

It can also be assumed that the reliability of these services will be high, as AWS will 'use commercially reasonable efforts to make the included services each available for each AWS

region with a Monthly Uptime Percentage of at least 99.99%' (AWS, 2019), which will significantly reduce the risk of issues related to network connectivity.

Cloudflare

Cloudflare is a cloud platform that offers various security, availability, and performance tools, by acting as a reverse proxy; rerouting all traffic so that it first enters, and eventually passes through their edge server network, before reaching the origin server. It is free for personal use.

By rerouting internet traffic to an origin server in this fashion, Cloudflare is able to offer multiple security, performance, and reliability benefits. For example, Cloudflare issues a free SSL certificate that can be used for all inbound traffic, effectively allowing all proxied services to serve over HTTPS, encrypting the data and ensuring its privacy and data integrity (Cloudflare, 2019).

In a similar fashion, Cloudflare also provides world-class DDOS attack protection, reducing the risk of protected services being knocked offline by attackers.

UEWebsocket

UEWebsocket is a free, cross platform, open source WebSocket plugin for use with Unreal Engine 4. While it is primarily designed for use with Blueprints, it is implemented in C++, therefore allowing the WebSocket implementation to be called directly from game code. Under the hood, UEWebsocket is a wrapper for Libwebsockets, a free C library designed for implementing network protocols, which is included in the plugin's source as a dll.

Elo Rating Calculations

While Blade II Online bases its Elo system on the system found in League of Legends, the exact formulae used to calculate Elo in said system have never been publicly released. Therefore, the formulae below are largely based on the standard Elo formula described by James Grime (2019, 00:52), with some assumptions made based on observations of League of Legends Ranked system.

There are two main values that must be calculated for each player after a match has been concluded.

1. The expected score for the player. This can be described also as the chance that the player had to win the game.
2. The new Elo rating for the player.

The formula for calculating the expected score of player A is

$$E_A = \frac{1}{1 + 10^{\frac{R_A - R_B}{400}}}$$

where E_A is the expected score for player A, and R_A and R_B are the current ratings for player A and B, respectively.

The constant value of 400 describes the Elo rating difference that would indicate that one of the players is 10 times better than the other.

The resultant value is always a value between 0 and 1, making it plain that the formula for calculating the expected score for player B is

$$E_B = 1 - E_A$$

Therefore, the formula for calculating the new Elo R'_A for player A is

$$R'_A = R_A + K(S_A - E_A)$$

where R_A is the current Elo rating for player A, K is the K-factor (akin to the maximum shift in Elo that can occur), and S_A is the actual score[i] for player A.

Thus, the formula for calculating the new Elo R'_B for player B is

$$R'_B = R_B + K(S_B - E_B)$$

where R_B is the current Elo rating for player B, K is the K-factor, and S_B is the actual score for player A.

[i] The "actual" score for a player; a value of 0, 0.5, or 1, for a loss, draw, or a win, respectively.

The constant K, known as the K-factor, defines the sensitivity to change – how wins and losses impact a player's Elo rating. Blade II Online makes use of a varying K-factors depending on a player's current Elo Rating.

Making use of a smaller K-factor for higher ranked players should in theory avoid Elo inflation at higher ratings, where players in the very top percentile will often find themselves playing either players of equal or lower ratings.

Unrated players will have a significantly higher K-factor, with a goal of being able to quickly reach a rating that closely matches their apparent skill level – The potential for large shifts will quickly shift their rating towards their "true" rating.

AI Opponent

Blade II online features an AI opponent against which matches can be played. The AI's behaviour is not designed to emulate a human player – in fact, Anguelov *et al.* (2014, p. 5) suggest that 'even some behaviours that actual humans would display should be avoided, because those behaviours *appear* inhuman when performed by an AI-controlled character'.

The Blade II Online AI opponent aims to provide an experience that players will quickly become accustomed to. This not only creates a smooth learning experience (with no surprises), but also quickly puts players in a position where they will most likely find more stimulation, and hopefully enjoyment in playing games against real opponents. This will hopefully lead to players relying on AI games to practice or strategy testing, as they will be able to effectively disregard the AI opponent as the obstacle between them and a desirable outcome for the game once they reach a point at which they can reliably beat the AI.

The AI will be implemented as a finite state machine that shifts between states based on the current state of the match, and moves (a card being played) by the player. Anguelov *et al.* (2014, p. 48) explain that 'An FSM breaks down an NPC's overall AI into smaller, discrete pieces known as *states*. Each state represents a specific behaviour or internal configuration, and only one state is considered "active" at a time.'.

While it could be tempting to implement the AI using a sequence of conditional statements that slowly filter out moves until one has been selected, it is perhaps be better, and definitely more interesting to implement a depth-limited Monte Carlo tree search algorithm that determines the best move to make. The Monte Carlo tree search algorithm is a modified version of the one described by Cowling et al. (2012), where valid moves are rated based on their potential effect on the state of the game should they be played. Applying a depth limit both ensures that the AI cannot see too far into the future (and therefore cannot reliably determine the best move), and ensures that the AI does take too much CPU time in order to determine a move to make, as this could negatively impact the performance of the game client.

Analysis

Though many of the tools and technologies to use were determined during the literature review stage, some choices still remained to be made. Namely, the choice of game engine and launcher framework.

Following this, a comprehensive list of requirements and their priorities was created, becoming the foundation of the development methodology, dictating what tasks needed to be completed, and the timeframe in which they were expected.

Game Engine and Language

Ultimately, the decision was made to develop the Blade II Online game client using Unreal Engine 4 and C++. An overwhelming majority of the game client features were implemented using C++ rather than using Blueprints, though the user interface makes heavy use of the UMG widget tools, which strongly encourages the use of the built-in editor GUI. Even then, all the widget logic is handled by C++ classes, from which the widget blueprints derive.

The main reasoning for choosing Unreal Engine 4 and C++ are detailed below, along with considerations of the main alternative, Unity and C#.

Performance

Unreal Engine 4 runs on, and supports development using C++, which is typically more performant than equivalent code written in C#.

Though the article is not focused on comparing performance, Warren's (2019) investigation of the low-level features of C# reliably show that even well optimized C# runs slower than comparable C++ code, and with higher memory requirements.

Selecting the most performant engine and language combination was of utmost importance, with the aim of keeping the game client lightweight and fast on all modern machines, regardless of hardware.

Unreal Engine 4 also supports development with Blueprints, a visual scripting tool. While this offers many of the safety features that are also available for unity (protection against

editor crashes due to errors, for examples), it often becomes unwieldy and hard to manage as the size of a project increases, as can be seen in the examples in the UE4 Blueprints From Hell blog[ii].

Blueprints are also significantly less performant than C++ code due to having to run in a virtual machine, and, even though it is possible to nativize Blueprints (automatically generate C++ code from a project's Blueprints that will be compiled during the process), Cpt. Trips (2017, 05:06) shows the generated C++ code is not as performant as equivalent handwritten C++ code.

Postgraduate employment opportunities

Due in part to the exposure of being used by high profile Japanese game franchises such as Street Fighter, Tekken, and Kingdom Hearts, Japanese adoption of Unreal Engine 4 is steadily increasing. Increasing numbers of developers in Japan are gaining experience with Unreal Engine 4, which should naturally lead to increased awareness and rates of adoption of the engine for other projects as time passes (Kawasaki, 2017). Combined with first class Japanese language support for official resources (Suzuki, 2014), experience with Unreal Engine 4 is a valuable asset for those looking to work in the Japanese games industry.

For this reason, combined with Epic setting up a Japanese branch in 2010 (Famitsu, 2010), a lot of existing C++ talent, and 'a great number of universities and professional schools in Japan [adopting] UE4 into their curriculum' (Kawasaki, 2013), Unreal Engine 4 is more popular in Japan than ever, and being able to demonstrate proficiency will increase the employability of anyone wishing to apply for jobs in the Japanese gaming industry.

Features

Whilst both Unreal Engine 4 and Unity have a fairly similar set of tools, the former often has a richer, more powerful equivalent. For example, both engines support rich 2D UI's, but Unreal features a dedicated UI editor, along with a fully featured, modular, extensible UMG library. UMG is implemented using C++ code, and can be extended by creating a class

[ii] UE4 Blueprints From Hell blog: https://blueprintsfromhell.tumblr.com/.

derived from a UserWidget. The visual aspect of a UMG Widget is typically managed by using the UMG UI designer, a visual UI authoring tool featured within the Unreal Engine 4 Editor.

Unity does offer similar modular capabilities with its prefab system, but UI classes cannot be inherited from and must therefore be wrapped in a helper script that can only access the public interfaces of the UI class that it is handling. UI prefabs can be modified in their own prefab editor, but the majority of UI development is handled directly in the editor scene window.

While Unity does have some significant advantages over Unreal Engine 4, such as significantly lower compile times, and protection against compile and runtime errors (Unreal Engine 4 will typically crash when encountering an error, while Unity will often throw an error, but remain responsive), these were determined not to be significant enough to warrant using Unity over Unreal Engine 4.

UEWebsocket was selected for use in this project due to the highly permissive license, and the fact that it is compatible with multiple build targets including Windows, macOS, and Android.

Launcher Application

The launcher was originally planned as an integrated component of the game client, but during development a decision was made to split the launcher into a separate application.

Developing a visual aspect of the game such as the UI requires frequent iterations, often with minor changes. When C++ code is modified, the entire game module must be recompiled, leading to a lot of downtime when fine-tuning and tweaking UI elements.

Many frameworks exist for creating non-game applications for desktop, such as Xamarin.Forms[iii], though the choice to use Electron[iv] was made fairly soon after the in-game launcher was abandoned.

Developing applications with Electron is 'fast, efficient, and cost-effective' (Warcholinski, 2020), and while cost is not an issue, being able to write all the program logic in JavaScript completely removes any downtime that would otherwise be spent waiting for Unreal Engine 4 to compile. JavaScript code is also fairly easy to write, and due to it being one of the most commonly used programming languages as shown in the results of the most recent Stackoverflow Developer Survey (2019), there are plenty of resources available online to refer to when problems occur.

Furthermore, JavaScript provides first class support for interacting with devices on the internet, making it simple to develop applications that interact with web servers, or communicate via Websockets (and in the case of Blade II Online, both).

Lastly, Electron is cross platform, meaning that Blade II Online could very easily be ported to another operating system such as macOS if the game client were also built for this target.

Requirements

All the requirements for Blade II Online are listed in the table here, prioritised using the MoSCoW method (Clegg and Barker, 1994). The finished product will ideally have all of the features in the first two columns (as long as the requirements / goals do not change in a way that would make the feature impractical or irrelevant), with features in the third column being implemented as and when any spare development man hours are discovered. Due to a lack of experience developing games of this type, there is a considerable chance that the requirements will change over the course of development. Features identified as irrelevant, impractical, or too costly will most likely be removed, or moved down in priority.

[iii] Xamarin.Forms is an open source framework for building iOS, Android, and Windows apps: https://dotnet.microsoft.com/apps/xamarin/xamarin-forms/.

[iv] Electron is an open source framework developed and maintained by GitHub, which allows desktop GUI applications to be built using web technologies (JS / HTML / CSS): https://www.electronjs.org/.

Development Methodology

Throughout this project, development was carried out in accordance with the Agile Kanban methodology. Agile Kanban, an extension of traditional Kanban, is a lightweight method that places a heavy focus on the visual aspect of organising and maintaining tasks. Ongoing tasks are displayed as cards or tiles on a kanban board[v], which are further organised into columns that indicate their current state, such as to-do, in-progress, or done (Muslihat, 2018).

There are no strict time limits, or requirements for constant iteration – which was ideal when it came to balancing the development time of Blade II Online with deliverables for other University modules throughout the school year. Rather, tasks were completed as required, and when possible, with iteration being carried in the form of continuous improvement, as opposed to the bi-weekly iterations of the Agile Scrum method described by Knight (2015).

A study undertaken by Padmanabhan (2018) shows that user responses to Kanban are generally favourable, with many considering it to be 'one of the best agile mechanisms to track the actions (both strategic and run-the-function) in a seamless way'. They then go on to describe how users feel 'empowered due to the full visibility of backlog and the ability to pull the actions to design stage by themselves'.

Each task was created and prioritised in accordance with the MoSCoW requirements table detailed above, as well as the project Gantt chart (included in the appendix here). The Gantt chart visualises a logical overview of the tasks that needed to be carried out during the develop of Blade II Online, and the creation of the associated dissertation, along with their approximate scheduling and estimated man-days required.

During development, the MoSCoW requirements were implemented roughly in order of priority, whilst considering other factors such as parallelising tasks where possible, and batching related tasks to reduce downtime between changing development environments.

[v] A kanban board is an agile project management tool that visualises tasks: https://www.atlassian.com/agile/kanban/boards/.

Where tasks were completed ahead of schedule, related, lower priority tasks were inserted, though this was rarely the case.

Conceptual Design

Blade II Online is, in essence, a proof of concept for an online multiplayer version of a minigame found in various Nihon Falcom games, such as *The Legend of Heroes: Trails of Cold Steel II* (where it appears as Blade, and Blade II in newer versions) and *Tokyo Xanadu eX+* (where it appears as Gate of Avalon). In an attempt to remain faithful to its original form, it was implemented as a carbon copy with the addition of online features, drawing its gameplay, appearance, and themes directly from versions of Blade found.

As an aside, special care and attention was paid to ensure that the gameplay was identical to the version found in the aforementioned games, due in part to a passionate, and highly vocal player base that would almost certainly raise concerns if this was not the case.

Theme

Blade II Online draws all of its themes from the version of Blade II found in *The Legend of Heroes: Trails of Cold Steel II*. Therefore, Blade II Online is a card game that can be played by anyone, featuring cards that represent different weapons and spells found within the game. Some of the cards also have special effects which are loosely based on special abilities of certain characters. For example, the bolt card has the ability to disable the most recently played card on a player's field, while the rod card has the ability to re-enable, a disabled card.

These details would of course be lost to players of Blade II Online that are not familiar with above works, but as they have no direct impact on the gameplay itself, and do they do not contribute to the effectiveness of the eventually proof of concept prototype, building upon these themes was relegated to being a low priority task at an early stage.

Storyline

When Blade / Gate of Avalon appears in Nihon Falcom's games, it is presented as a minigame, in the context of the world within which it exists. It is not clear whether there is any specific story or lore attached to the minigame itself. However, by observing the

appearance of the cards, which are decorated with spells and weapons found in the game, it can be assumed that it exists more as a self-reference as seen in this screenshot.

For this reason, Blade II Online was not designed with any particular lore or story in mind, and rather aims to act as a proof of concept that shows that an online version of Blade II can indeed be created.

Game Mechanics

The actual gameplay component of Blade II Online is presented as a game client within which matches are played. Matches involve two players, one of which may be an AI opponent, and remains active until either the game ends, or a player forfeits or disconnects.

Match Overview

Each match starts with the source set of forty cards, of which thirty are randomly selected per match. Each player is dealt a random selection of fifteen cards, that become the player's deck (a set of cards that cards can be drawn from under certain circumstances). From the deck, ten cards are drawn into their hand (cards that can played on their turn), and then the match begins in earnest.

1. Once the match begins and the cards have been dealt, each player must draw a card from the deck and place it on the field.
2. The player that has the lower score starts first and must place a card from their hand onto their side of the field.
3. Players then take turns selecting a card to play, selecting a valid move such as:
    a. Selecting a card with a value that allows them to surpass or match their opponents current total or,
    b. Playing a special effect card and observing its effect. Depending on the effect, they may need to play another card. Otherwise, their turn will end, and it becomes the other player's turn.
4. Should the scores be equal after a player finishes their turn, the field is cleared, and the cards are discarded, and both players must draw and play new cards.
5. This process repeats until either player can no longer surpass their opponent's total value, they run out of legal moves, fail to beat, or match their opponents score, or run out of cards to play.

6. If both players run out of cards after a turn ends with a tied score, the game is considered to be tied.

An illustration of the layout of the Blade II Online board is included below, in the form of an early wireframe that was used as a guideline for the development of the game client. Note that the wireframe illustrates the view of the game from the perspective of a single player, and thus, their opponent's deck and hand cards are not visible.

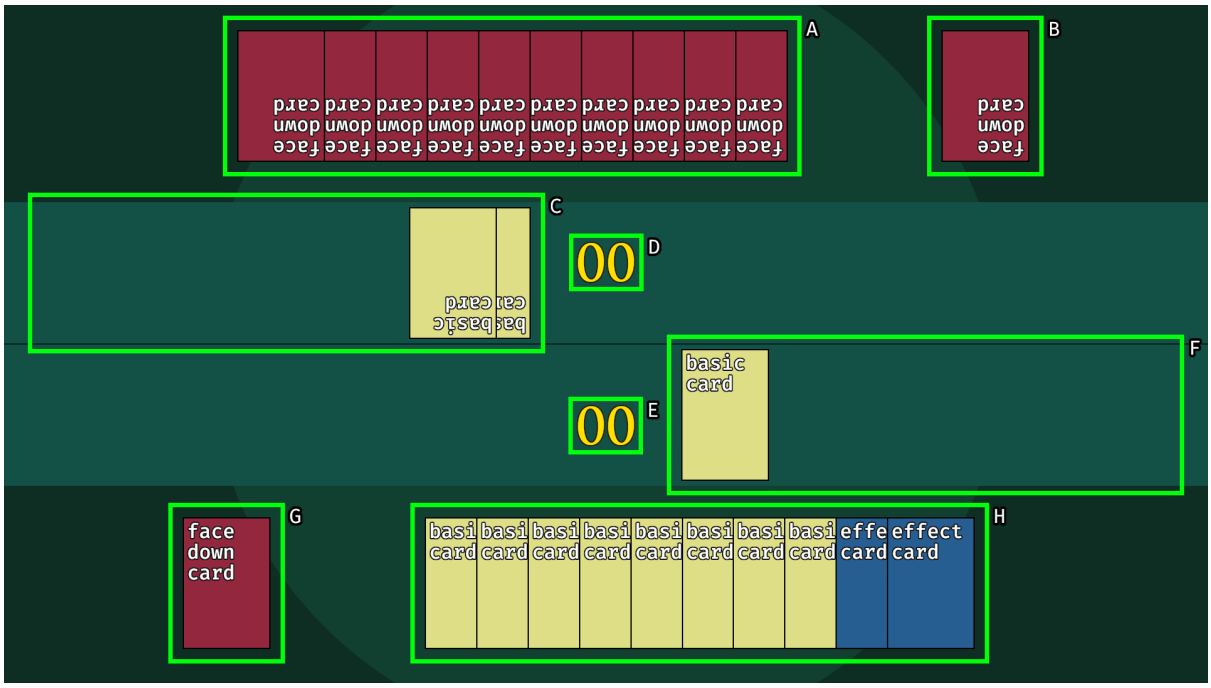

Figure 1. An early wireframe illustrating the layout of the board for a Blade II Online match.

Keys and Definitions	
A	Opponent's hand
B	Opponent's deck
C	Opponent's field
D	Opponent's score
E	Player's score
F	Player's field
G	Player's deck
H	Player's hand
basic card	A card that increases the player's score
effect card	A card that can also trigger a special effect under certain conditions

face down card	A card that is not yet revealed

Table 1. Descriptions of the keys in the wireframe shown above.

Card overview

All cards have a value that contributes to the player's score - even cards with special effects. Special effect cards that are not used for their effect (such as when played onto an empty field) will increase the score by their base value of one. Otherwise they will trigger an effect, such as bolting (flipping over and removing the points gained) a card.

ID	Formal Name	Count	Value	Special Effect
0	Elliot's Orbal Staff	2	1	*Un-bolt* a *bolted* card on the player's side of the field
1	Fie's Twin Gunswords	5	2	None
2	Alisa's Orbal Bow	5	3	None
3	Jusis' Sword	5	4	None
4	Machias' Orbal Shotgun	4	5	None
5	Gaius' Spear	3	6	None
6	Laura's Greatsword	2	7	None
7	Bolt	4	1	*Bolt* the opponent's last played card, reducing its value to zero
8	Mirror	4	1	Switches the field around, effectively swapping the played cards for each player, as well as their current scores
9	Blast	4	1	Removes a card from the opponent's hand (selected by the player)
10	Force	2	1	Doubles the player's score instead of increasing the score by its value of 1, when placed on the field
Total number of cards: 2 + 5 + 5 + 5 + 4 + 3 + 2 + 4 + 4 + 4 + 2 = 40				

Table 2. A description of all the cards that can be found in a Blade II Online match.

Special rules

1. Effect cards cannot be played as the last card. If a player's hand consists of only effect cards at any point during the game, they will lose.
2. Effect cards placed on an empty field, such as at the start of a match after being drawn from the deck, add their value to the players score, and the effect is ignored.

3. Should the scores be tied after a player ends their turn, and a player has no cards left in their deck, this player must select a card from their hand instead of their deck.

End conditions

1. A player ends their turn without successfully matching or surpassing their opponent's score.
2. All the cards in the game are exhausted in some way.
3. A player begins their turn, and none of their cards would allow them to beat or match their opponent's score.
4. A player ends their turn with only effect cards in their hand.
5. A player disconnects from, or chooses to forfeit a match.

Appearance

The Blade II Online game client was designed with the intention of faithfully recreating the minigame found in the aforementioned Nihon Falcom titles, and as such, its appearance is almost identical. However, the original version was implemented as a fully 2D game, whereas Blade II Online replaces some aspects of the game with 3D models, including cards, and the board.

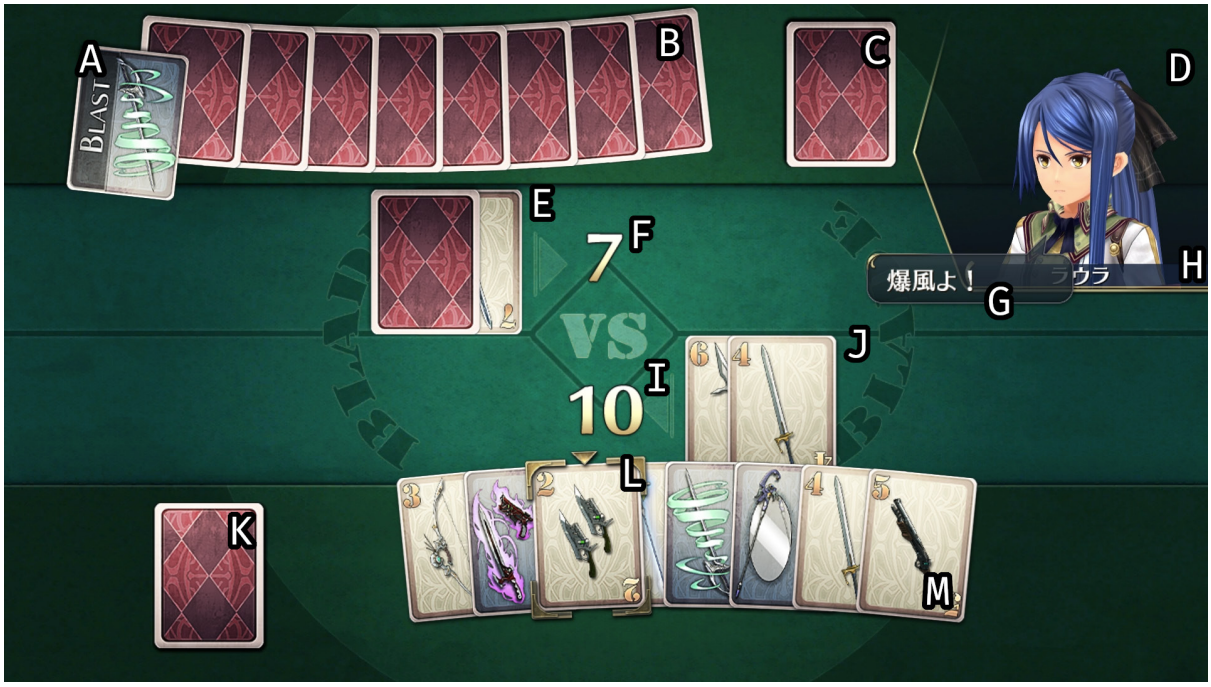

Figure 2. A screenshot of Blade II gameplay from Nihon Falcom's *The Legend of Heroes: Trails of Cold Steel II*, with the language set to Japanese. Used as a reference image when developing the visual aspects of the game.

Keys and Definitions	
A	An effect (Blast) card that is currently being played by the opponent.
B	The opponent's hand
C	The opponent's deck
D	The opponent's avatar (a 3D model, rendered in real-time)
E	The opponent's field, containing one basic (Laura's Greatsword) card, and one bolted card
F	The opponent's score
G	A context-specific message from the opponent
H	The opponent's handle (obscured by the message G)
I	The player's score
J	The player's hand, containing two basic cards (Gaius' Spear, and Jusis' Sword)
K	The player's deck
L	A card that is currently selected by the cursor. In this instance, the opponent is selecting a card to destroy after activating a Blast card
M	The player's hand

Table 3. Descriptions of the keys in the above screenshot.

Where possible, textures were extracted from a purchased copy of *The Legend of Heroes: Trails of Cold Steel II,* in order to recreate the game as faithfully as possible.

Model Design

The 3D models created for use in Blade II Online are very simple, and the models themselves are not particularly of note. Both the cards, and the board/arena are literal interpretations of the 2D version found in the original game.

It was required, however, to define the physical dimensions of each card, as this would be used to determine the scale of the board, and the world space area within which the game would operate.

The dimensions of the cards are based on the dimensions of the cards featured in the Pokémon TCG. Pokémon trading cards are also very similar to standard poker cards in size, and due to the prominence of both, their size proved to be a natural looking, aesthetically pleasing size that allowed for the maximum number of cards for the hand (ten), and field

(fifteen) to fit within the dimensions of a screen with a 16:9 aspect ratio. Standard Pokémon cards are a 63mm wide, and 88mm tall, according to BGG user Grant Allen (2014).

The exact values were not used, but a card with the same width and height ratio was then created using Maya. These cards were then laid out in world space, and used to determine the dimension of the board model.

User Interface Design

Game Client

The user interface for the original game is fairly minimal, with very little information being displayed on the screen at one time. Each card clearly shows what type of card it is, and its value, meaning that very little information is left to be shown to the user by the UI.

What remains are the elements that display the following (refer to this screenshot for a visual guide).

1. Each player's score.
2. The position of the cursor.
3. The opponent's avatar.
4. The opponent's handle display.
5. The opponent's message display.

The Blade II Online game client replicates this UI a near 1:1 relationship, though some changes were made in due to the original being slightly asymmetrical, most likely due to the player not having a visible avatar or handle display.

In addition to the basic UI, the game client also features multiple additional UI layers, which implement features such as the special effects that play when an effect card is activated, and the victory/draw/defeat messages that play at the end of a match.

Lastly, a new UI element was designed that displays the current state of the game to the player, informing them of whether it is their turn, their opponent's turn, or if the game is currently waiting for an animation to finish, etc. This element was created using image

assets extracted from a purchased copy of *The Legend of Heroes: Trails of Cold Steel II*, to ensure that it felt thematically correct within the context of the game.

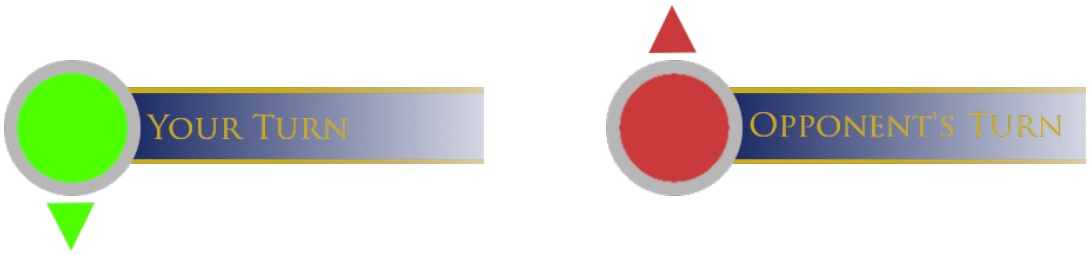

Figure 3. Early prototypes for the game state indicator.

Launcher

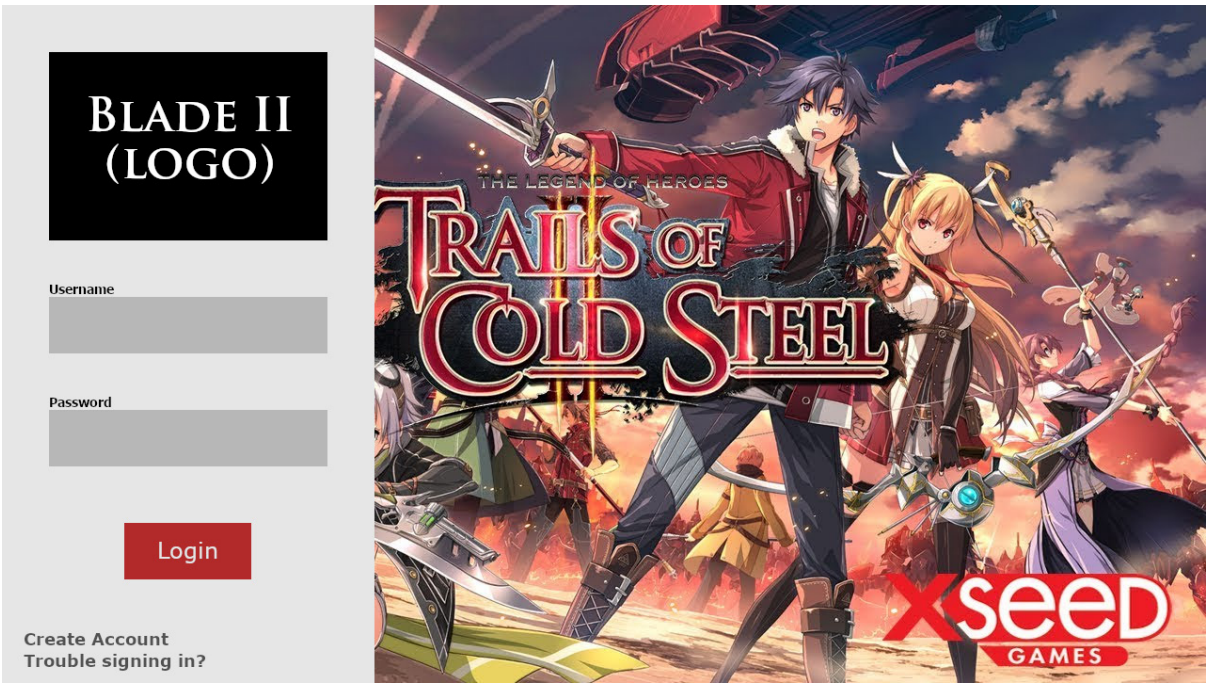

Figure 4. An early wireframe prototype for the Blade II Online launcher.

The Blade II Online launcher is where the vast majority of non-gameplay interactions take place. Each player must make use of the launcher to perform tasks such as creating an account, logging in, viewing player information, viewing the leaderboard, and finding matches to play.

The launcher is the user's interface for interacting with the entirety of the Blade II Online ecosystem. It was designed with the intention of delivering a rich, fully featured, easy and intuitive to use, responsive, interface with which players could carry out every single related task, aside from playing an actual match.

The design of the launcher drew inspiration from two main sources. The first, the League of Legends launcher, which provided a robust starting point that was used to develop the initial wire frames. The minimalistic design of the League of Legends Launcher allowed for a clear artistic vision to be developed from an early stage, quickly cementing the core design patterns and principles that would be revisited throughout the planning and development stage of the Blade II Online launcher.

Changes were made to the colour scheme, however, in order to better align the launcher with the visual aspects of the actual game of Blade II. The main light and dark colours were changed slightly, mostly due to personal preference, and the choice of the primary colour, as well as the various highlight colours, were changed based on the colour of particular aspects of the original version of the minigame found in the version of Blade found in *The Legend of Heroes: Trails of Cold Steel II*.

Figure 5. The various primary colours used throughout the Blade II Online launcher, with in-game screenshots from *The Legend of Heroes: Trails of Cold Steel II* that inspired them.

The second inspiration for the launcher was the battle menu found in one of Nihon Falcom's older titles; *The legend of Heroes: Trails in the Sky SC Evolution* (though it appears in many forms, in many of their other titles). The layout, and functionality of the battle menu served as a foundation when designing and prototyping the lobby view, which offers functionality such as viewing the leaderboards, and searching for a match. In short, users can select options by clicking on buttons that rotate the selector wheel clockwise, or counter-clockwise.

Each rotation changes the button that, when pressed, opens another interface where the user is interact with the online features of Blade II Online.

The battle menu in question can be seen in this screenshot.

Technical Design

Blade II Online costs of four major components – the launcher, the game client, the REST API, and the game server. The overall architecture of, and typical communications between each component can be seen in this diagram.

For each component, a class diagram has been included – though due to the size of the applications, only publicly accessed members are shown, and only the most important relationships are illustrated.

Note that, diagram for the REST API and game server omits descriptions of parts of the server that are implemented without exposing their internal objects to other packages. For example, the database package implemented using the single pattern, and exposes only functions instead of exposing types that can be instantiated. This was done to avoid having more than one instance of a connection to the database, in order to avoid race conditions between reads and writes.

Game Launcher

The class diagram for the Blade II Online launcher can be seen in the appendix here.

It was developed using the Electron framework.

Game Client

The game client has grown to a stage where it is no longer viable to attempt to fit an overall class diagram into an image of reasonable size. Instead, a basic relationship diagram has been included here, which highlights the relationship between the main manager class, and the other engine and game components throughout Blade II Online's C++ source code.

In addition, a link to the auto-generated documentation has been included in the appendices here.

The game client was developed using Unreal Engine 4 and C++.

REST API

The class diagram for the Blade II Online REST API can be seen in the appendix here.

It was developed using Go, for use as an AWS Lambda[vi] function.

Game Server

The class diagram for the Blade II Online game server can be seen in the appendix here.

It was developed using Go, built for Linux, and hosted using an AWS EC2[vii] instance.

[vi] AWS Lambda enables code to be run in the cloud without a server. See: https://aws.amazon.com/lambda/.

[vii] AWS EC2 (Elastic Compute Cloud) is a virtual machine service. See: https://aws.amazon.com/ec2/.

Implementation Details

Blade II Online is a full stack project, built from the ground up using a wide range of technologies. Each component is a fully functional implementation of one of the cogs in the Blade II Online ecosystem, which come together to deliver an entire online game.

A selection of technical details of some of the most noteworthy features of each component have been curated and included below. The full source code for each component is also included in the appendices here.

Some screenshots of the launcher and game client can be seen in the appendix here.

Game Launcher

The launcher was designed in accordance with the MVP (model-view-presenter) pattern. This allows for a strict separation of program logic and interface logic, which is especially useful in an application that features a lot of different user interfaces.

Network Interactions

`NetModel` (`netmodel.js`) is a JavaScript class that implements network communication methods, allowing the launcher to communicate with both the REST API (over HTTPS) and the game server (over WebSocket Secure). It inherits from `BaseModel` (`basemodel.js`), the parent class from which all Model classes derive.

A number of functions are available that implement preconfigured HTTP requests, which can be called by other model classes. Once one of these functions is called, `NetModel` packages any required data into a JSON format string, adds any required headers (such as Authorization), and then sends the request to the REST API via HTTPS using the node module request.

As can be seen in the `sendCreateAccountRequest` function, calling this function and passing in a handle, email, and password, will prompt `NetModel` to send an account creation request to the REST API. The API will then process this request, returning a response depending on the outcome of the operating.

`NetModel` then asynchronously waits for either a response from the server, or for the request to time out. One of the events members of `NetModel`, such as `onCreateAccountResponse`, will then be called, broadcasting the result of the request to any registered event listeners.

`onCreateAccountResponse` is an instance of the `B2Event` class, which implements an event that broadcasts out to all registered functions, as well as various helper functions for configuration, registering, and deregistering etc.

WebSocket connections are made by calling the `startMatchMaking` function, and using a new instance of the `B2WS` class, initialises a WebSocket connection to the matchmaking server, sends authentication data (the username and password that would have been received and stored by `NetModel` upon successful login), and then listens for messages. When messages are received, the `onWebsocketEvent` function is called, which then broadcasts an event depending on the message contents.

Account Creation

`CreateAccountModel` (`createaccountmodel.js`) is a JavaScript class that processes user updates from the account creation view interface. It inherits from `BaseModel` (`basemodel.js`), the parent class from which all Model classes derive.

`CreateAccountModel` is responsible for tasks such as validating usernames and passwords that are chosen by the user, as well as passing account creation requests to the `NetModel`, and processing the result of these requests.

Validation is performed locally, as can be seen in `determineUsernameWarning` function, which responds to changes in the password input field in the account creation form. On change, the current value is passed down from the view, and then validated using various `RegExp`[viii] variables defined in `validation.js`. The outcome of the validation tests are stored in an instance of the `PasswordWarningState` container class.

[viii] RegExp is a native JavaScript object used for matching text with a regular expression. See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/.

Once validation is complete, the results are passed back up to the presenter (`CreateAccountPresenter`) via events, and processed by the view (`CreateAccountView`) when the presenter calls the `updatePaswordWarnings` function, which inspects the contents of the `PasswordWarningState` parameter, and then styles the various predefined password requirement hints appropriately. The resultant output is an error message that details exactly what is wrong (or right) about the user's choice of password.

Figure 6. The account creation password field, showing an invalid, and then password, respectively.

Localization Module

The entire Blade II Online launcher is fully localized in both Japanese and English, through the JavaScript class `Localization`. This class is responsible for storing the current locale, and for updating all the localised elements when the locale is changed.

When initialised, the `load` function is called and localization data is read from `localizations.yaml`, and stored in a local `Map`[ix] instance after being parsed. This data persists throughout the lifetime of the app, and is accessible via the `get` function.

During initialisation of the `settings module`, a `call` is made to `setLocale`, a function that updates the current locale, and then iterates over all of the localised elements of the application, updating the text according to the value that was defined in `localizations.yaml`. This function can be called at any point during the application's lifetime, meaning that the display language can be changed by the user should they wish to do so. This is one of the many configurable settings found in the options menu, a modal popup that is accessed by clicking the cogwheel icon on the title bar of the launcher.

[ix] Map is a JavaScript object that stored key-pair values while maintaining the insertion order. See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/.

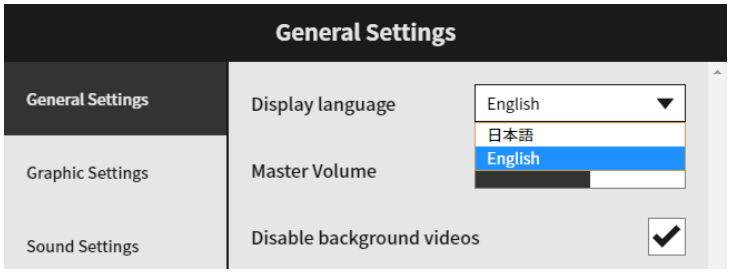

Figure 7. The display language selector in the options menu.

Game Client

The GameMode Class

`BladeIIGameMode` (`BladeIIGameMode.h`, `BladeIIGameMode.cpp`) is the core of the Blade II Online game client, acting as both an entry point, and a central manager throughout the runtime of a match. It is responsible for multiple tasks, some of which are outlined below:

Discovery and initialisation of Blueprint child classes

Though all program logic is written in C++, In certain cases Blueprint classes deriving from C++ classes were used in order to simplify the configuration of objects that have a visible representation in the game world, such as cards, the user interface, and the avatar capture rig. Some of these classes are placed in the game world before the game starts, though the majority of them are initialised when the match begins.

Performing this task in blueprints is simple, due to the fact that Blueprint types are immediately available for use without any additional processing. With C++, however, one must jump through hoops in order to get a reference to a subclass of the derived blueprint, which must then be created and added to the world when the match begins.

An example of this process can be seen here. The `GetUIAvatarWidgetClass` function is called during construction of the `BladeIIGameMode` instance, and the blueprint class is stored as a member variable of type `TSubclassOf`[x]. This is then used to create a new instance of the UI avatar widget (when the `SetupUIAvatarLayer` function is called during initialisation), which is then added to the player's screen and stored as a pointer.

[x] TSubclassOf is a template class that stores a derived Blueprint of a particular UObject type. See: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/TSubclassOf/index/.

This pattern can be seen multiple times throughout the `BladeIIGameMode` class, finding and initialising man other UI components, as well as some in-game actors.

Handling Events

`BladeIIGameMode` is also responsible for receiving and processing events from various engine components, including the opponent instance, the card dealer instance, and the user input module.

When the match begins, the `RegisterEventListeners` function is called, which assigns handlers to various event delegates found throughout various engine. These handlers are marked with the `UFUNCTION()`[xi] macro, so that they can be registered using the `AddDynamic()`[xii] helper macro, allowing them to be registered by function pointer rather than having to register them using a function name.

These handlers then process any events that they receive, an example of which can be seen in the `HandleDealerEvent` function, where the event payload (an enum class of type `EDealerEvent`) is checked by a switch statement, determining how to handle the event.

Initialisation of engine components

Engine components are initialised either at construction time, or when a match begins. In this case, engine components refer to instances of classes that provide some sort of core functionality.

One of the first components that is initialised is the settings class, `USettings` (`Settings.h`, `Settings.cpp`). This is initialised with an instance of the launch config reader/parser class, `B2LaunchConfig` (`LaunchConfig.h`, `LaunchConfig.cpp`). `B2LaunchConfig` reads and parses the launch config file when constructed. This file is created during the launcher bootstrapping phase, and contains the data required to correctly initialise the current match

[xi] UFUNCTION() is a macro used to mark functions as a UFunction, a function that is recognised by the UE4 reflection system. See: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Functions/index.html.

[xii] A helper macro for registering functions to delegates by pointer instead of by name. See: https://github.com/EpicGames/UnrealEngine/blob/release/Engine/Source/Runtime/Core/Public/Delegates/Delegate.h#L374.

of Blade II Online, as well as the most recent settings set in the launcher application. Default values are loaded when an error is encountered, to ensure that the game does not crash – instead, players are entered into a match against the AI opponent, which they are free to forfeit without penalty.

The aforementioned `USettings` instance processes these launch config values, and applies relevant settings when the `ApplyAll` function is called. This function is responsible for both updating the games locale based on the one defined in the launch settings, as well as updating the games graphics settings based on the configuration defined in the launcher's options menu. Note that sound settings are not applied here, as this task is left to an instance of `AGameSound` (`GameSound.h`, `GameSound.cpp`) responsible for handling all in-game sounds.

Providing Access to Game and Engine Component Instances

`BladeIIGameMode` also exposes multiple public functions (declared and defined in the header file here) that allow other classes to get pointers to various engine and game components that are stored as private variables.

Match State Machine

The vast majority of the match logic is contained within a state machine implementation written specifically for Blade II Online's game client. A single class, `B2GameStateMachine` (`GSM.h`, `GSM.cpp`) is responsible for its operations, and will either execute or change the current state based on calls to `Tick` and `ChangeState`. All state changes are handled by the `BladeIIGameMode`, in order to reduce complexity and avoid situations where race conditions cause an incorrect state to be entered (as mentioned earlier, `BladeIIGameMode` responds to events from various engine components, and in certain circumstances will update the state machine in response).

The state machine can be swapped over to any state that derives from `GSM_State` (`GSM_State.h`, `GSM_State.cpp`), a base class that implements various virtual functions that implement common functionality such as `Tick`, called by the state machine when requested to do so by `BladeIIGameMode`, as well as utility functions that are used by states to streamline interaction between the state and the match, such as `UpdateCursorPosition`.

Derived classes, representing different possible states of a match, interact directly with various engine and game components, progressing the match based on inputs from the player, and the opponent. The different possible states for the state machine can be seen here. Each state performs specific tasks based on the current state of the game, and runs until either its own internal state prevents it from doing so (such as setting a flag to prevent further processing of a particular event, as can be seen in the `Tick` function of the `DrawToEmptyField` state).

Each state is relatively simple, performing a few related tasks in order to handle a particular mechanic or phase of a match. For example, the `PlayerTurn` state (`GSM_State_PlayerTurn.h`, `GSM_State_PlayerTurn.cpp`) implements a handler for user input in its `Tick` function, moving the game cursor across the player's hand when specific inputs are detected by the `ALocalPlayerInput` (`LocalPlayerInput.h`, `LocalPlayerInput.cpp`) instance. When it detects the select input, it identifies the card on which the cursor currently is positioned, and requests for the card to be moved by the `Dealer` class instance (details below).

Card State and Card Animations

When a state detects that a particular card has been selected, or if the state of the game requires a card to be manipulated (both physically and logically), the state can make use of the `Dealer` (`Dealer.h`, `Dealer.cpp`) instance in order to do so. This class is responsible for executing predefined sequences of animations that both moves cards around in the game world, whilst also updating their logical representation within the `AArena` (`Arena.h`, `Arena.cpp`) instance.

As can be seen in the `Mirror` function, the dealer creates sets of `B2Transition` (`Transition.h`, `Transition.cpp`) instances, a class that is initialised with the details of an animation, and a `Tick` function that progresses the animation based on the current `DeltaTime`. These individual transitions are added to an instance of `FB2CardAnimationGroup` (`CardAnimator.h`) which forms a logical group of transitions that will be run simultaneously, whilst also ensuring that, each animation in the group is

finished before the next group can begin execution. These are then passed to the `UB2CardAnimator` (`CardAnimator.h`, `CardAnimator.cpp`) instance.

The card animator then iterates every `Tick`, updating each animation within the current animation group until they have all reached completion. After each individual animation is updated, `UB2CardAnimator` updates the position and rotation for the associated card, and then continues, until all the animations have been updated. When all the animations in a group are complete, the group will be discarded, and either the next group will be executed, or the animator will remain idle until it receives a new batch of animations.

Playing Cards

The playing cards for Blade II Online are simple 3D meshes created using Maya. For each card, the art from the original game was recreated using GIMP. In order to do this, the 3D models for the illustrations on each card where extracted from a purchased copy of *The Legend of Heroes: Trails of Cold Steel II,* and their textures where upscaled. Using Maya and the Arnold renderer, each model was rendered with a transparent background, imported into GIMP, and then combined with the card background. The finished product can be seen here.

Player Avatars

Each player avatar started life as an FBX file extracted from a copy of *The Legend of Heroes: Trails of Cold Steel II* by GitHub user uyjulian. The models were then important into Maya, where they were smoothed and adjusted (multiple meshes making up the face were combined and the UV maps layered onto a single face), before being important into Unreal Engine 4. An animation was created using Mixamo.

These were put together in a capture rig actor derived by the `AAvatarCaptureRig` (`AvatarCaptureRig.h`, `AvatarCaptureRig.cpp`) class, which implemented eye and mouth animations, as well as the loading of appropriate model/texture files based on the selection of model. The capture rig renders out to a render texture in real-time, which is displayed on the UI by a user widget derived by the `UAvatar` (`Avatar.h`, `Avatar.cpp`) class.

Opponent

For each Blade II Online match, an instance of one of the opponent classes is initialised, and used as an interface between the player and the opponent. The opponent system was designed to be entirely transparent to the `BladeIIGameMode` instance. Regardless of whether the opponent was controlled by a local AI, or if it was another player somewhere else in the world, the opponent base class does not change its functionality, nor the fashion in which it handles inputs and outputs. The opponent acts as a front end to a server – either a local state machine that processes, and responds to moves from the player, or an actual remote server, connecting via a WebSocket connection.

Both opponent classes inherit from the base class `UB2Opponent` (`Opponent.h`, `Opponent.cpp`) which implements the inbound/outbound message pumps, and though they have different their initialisation functions (`NetOpponent::Configure`, `AIOpponent Configure`), pass exactly the same kind of messages back to the player in response to moves and other inputs.

At the heart of each opponent, is a pseudo server that inherits from the `UB2Server` (`Server.h`, `Server.cpp`) base class. This base class implements the input and output interfaces through which messages are sent to, and received from, as well as including virtual functions that control the lifecycle of the server.

AI Server

UB2AIServer (`AIServer.h`, `AIServer.cpp`) implements an AI that responds to moves made by the player and, depending on the state of the match, attempts to find and make moves that prevent the AI from losing. It is also responsible for generating the set of cards that will be used for the current game, sending the generated set of cards to the player after it is initialised. This is handled transparently – the first call to `GetNextUpdate` will trigger the generation of a set of cards, which is then packages as a message and added to the inbound queue (the queue that is read by the `BladeIIGameMode` instance).

Every update `Tick`, the AI checks the outbound message queue (in this case, outbound refers to messages that would in theory be forwarded to a remote server, but are intercepted and handled by the AI instead). When an update is detected, the AI uses it to update its internal

state of the match using the `UpdateState` function, and then attempts to find a set of actions that will result in the AI's turn ending without causing the AI to lose by calling `ResolveAITurn`.

`ResolveAITurn` iterates until a valid state is reached, performing actions until the turn switches back to the player, or the AI ends up in a state where there are no more valid moves to make (that would not cause the AI to lose). While iterating, it either calls `HandleTie`, if the scores are tied, or `ExecuteTurn` if not. Both of these functions attempt to discover a valid move to make given the current state of the board.

Net Server

UB2NetServer (`NetServer.h`, `NetServer.cpp`) implements a WebSocket wrapper that facilitates the forwarding of messages from the `BladeIIGameMode` instance to the Blade II Online game server, whilst also handling, processing, and packaging messages from the server before adding them to the inbound queue.

Messages received by the `UB2NetServer` instance are repackaged into a format defined by the game server, as can be seen in the `Tick` function. Serialised into an appropriate JSON format string, using the `GetSerialised` helper function of the `FB2ServerUpdate` (`ServerUpdate.h`) class, before being sent down the WebSocket.

Messages received from the game server via the WebSocket connection are also handled in a similar fashion by the `HandleMessageReceivedEvent` function, which is registered as a callback for when messages are received from the WebSocket. Here, message payload is parsed into a FB2WebSocketPacket (`WebSocketPacket.h`) using the built-in helper `FromJSONString`, and is then repackaged for use within the Blade II Game, depending on the server code that was parsed from the contents of the payload. This also carries out the handling of non-WebSocket error messages, which will typically lead to the WebSocket connection being terminated, and a message being sent to the game mode indicating what happened. WebSocket specific events are handled by their specific callback handlers, such as `HandleConnectionErrorEvent`, which handles WebSocket errors such as a loss of connectivity, or the inability to initialise a connection.

REST API

Serverless Function and API Gateway

The Blade II Online REST API implements various HTTP endpoints that allow networked clients to interact with back-end infrastructure in a safe, strictly defined way. The following endpoints have been made available for use by both the Blade II Online launcher and the game client, though they can in theory be used by anyone that has an internet connection, and suitable tools such as curl[xiii].

Method	Outcome on Success
`POST /auth`	An authentication token, and a refresh token, for the authenticated user
`GET /leaderboards?from&count[&pid]`	The leaderboards, starting at rank `from`, with `count` rows. Optional user specific row if `pid` specified.
`GET /matches/{pid}`	The match history for the user specified by `pid`
`PATCH/profiles/{pid}`	Update the match stats for the user specified by `pid`
`GET /profiles/{pid}`	Get the profile data for the user specified by `pid`
`POST /users`	Create a new account with the handle, email address, and password specified in the request body

Table 4. The REST API HTTP endpoints that expose various back-end functionality for Blade II Online.

The API itself is made available to the internet through an Amazon API Gateway[xiv], which receives requests from clients, forwards them onto an AWS function, and then returns the response to the client.

The lambda functions are written in Go, and are spun up whenever a forwarded request is received from the API gateway. Once received, the function inspects the message object, and after validating the request, its contents, and any additional arguments (such as query string parameters), attempts to perform whatever action it was designed for.

[xiii] Curl is a command line tool and library for transferring data with URL's. See: https://curl.haxx.se/.
[xiv] Amazon API Gateway is a managed service that acts as a front end for applications within the AWS ecosystem. See: https://aws.amazon.com/api-gateway/.

Taking a look at the lambda function `blade2_create-account`, it is revealed that the internal logic of the REST API is relatively straightforward. Upon receiving a message, the lambda is spun up, and the `main` function is called by the lambda runtime. A connection to the Blade II Online database is initialised, and the `lambda.Start` function (a helper function provided by the aws-lambda-go module) is called with the bootstrapper function `functionWrapper` as an argument. The bootstrapper then makes a call to the relevant function in the routes package (where endpoints are defined), which in this case is `routes.CreateAccount`.

`routes.CreatAccount` inspects the message body of the events.APIGatewayProxyRequest[xv] argument, validates them, and then attempts to create an account. If any of the validation functions returns a non-`nil` error at point during this process, or if one of the validation checks fail, the function will exit early, returning an error response with a suitable HTTP response code (typically 4xx for handled errors, or 500 for unhandled errors).

While it varies between the different lambda functions, the account creation function performs multiple validation steps for the values sent in the body of the request, such as ensuring that all the required fields were included in the request, ensuring the password meets the minimum complexity requirements (by calling `validatePasswordFormat`), and checking for profanity within the handle.

Once the request contents are validated, a call is made to the `database.CreateUser` function, which performs all the tasks required when creating a new user account such as hashing and salting the users choice of password, creating a new row in the users, profile and tokens table, and generating an email confirmation token.

Worth noting is that, in the case of this, and the other multi-query database functions, the queries are performed within a transaction, rather than being performed one at a time. This ensures that:

[xv] APIGatewayProxyRequest is a struct that represents the data that is received from an API Gateway proxy. See: https://github.com/aws/aws-lambda-go/blob/master/events/apigw.go#L5-L19.

1. The state of the database does not change during the query, invalidating or causing a discrepancy in a key column such as the index.
2. If any of the preliminary steps, or if any of the queries fail or result in an error, the state of the database can be rolled back with relative ease

One may also notice the use of `defer` keyword, a form of syntactic sugar that prevents the execution of the associated line of code until the function returns. This is used throughout the database package, to ensure that prepared statements and transaction rollbacks are properly carried out in the event of an error.

On success, the function returns, with the previously generated email confirmation token, and an error with a value of `nil`, passing control back to the `routes.CreateAccount` function. Then, after requesting for the email confirmation token to be sent to the email address that was specified in the request, the handle is packaged in a `types.CreateAccountResponsePayload`, serialized as a string, packaged into a `types.HTTPResponse`, which is then also serialized and packaged in a `types.LambdaResponse` – the final transformation required before the response can be returned to the API gateway. The API gateway then returns the response to the client, providing there was no error message returned by the lambda.

Note that errors are almost entirely opaque, meaning that they are returned with no indication as to what happened – in order to avoid this, the Go lambdas never return an actual error, opting instead to return a suitable HTTP status code to be handled by the client instead.

Database

Blade II Online makes use of a single instance of a MySQL database, within which all the data is stored. It was hosted, managed, and monitored using the Amazon Relational Database Service.

There are four tables, though as they are not particularly impressive, and for the most part, simply store data, they will not be elaborated upon, and have been relegated to a section in the appendix here.

Interactions with the database are carried out using prepared statements, which allows for safe interactions with a database when including user defined values in queries.

Most of the prepared statements used for the REST API are simple, though there are a few that may be worth highlighting. For the sake of brevity, however, an explanation of just one of them is included, and can be seen in the appendix here.

Game Server

The Blade II Online game server is a single Go program running on an AWS EC2 instance. The application is controlled and supervised by `systemd`[xvi] (essentially, as a service), and serves on local port 20000. NGINX is used as a reverse proxy, serving on port 443, passing external traffic onto either the matchmaking server, or game server route. Access to this server is restricted to a strict subset of IP addresses – the range specified by Cloudflare[xvii], in order to reduce the vector for external attacks.

It is immediately obvious that the naming convention of these components is confusing. For clarity, please see the following:

Game server: The portion of the Blade II Online game server that implements the /game endpoint, to which game clients can connect via Websockets in order to play networked games of Blade II Online.

Matchmaking server: The portion of the Blade II Online game server that implements the /matchmaking endpoint, to which the launcher can connect via WebSockets in order to search for a game against other players also connected via their launcher.

Both endpoints, /game and /match are served through the same port, using functions defined in the `routes` package, and when receiving a WebSocket connect request, will attempt to

[xvi] systemd is a system and service manager that daemons on Linux operating systems. See: https://www.freedesktop.org/wiki/Software/systemd/.

[xvii] Cloudflare IP's as defined on their website. See: https://www.cloudflare.com/ips/.

upgrade the connection to a WebSocket connection (which starts off as a standard HTTP request).

For brevity, only the game server will be covered, as the matchmaking server was implemented using very similar design patterns. For those interested, the source code for the entire server can be seen here in the appendix.

Successful connections to the game server are passed into the transactions package, where the `HandleGSConnection` function asynchronously waits for the connected client to send authentication data by opening up a channel into which messages from the WebSocket are passed.

When the first message is received, the `checkAuth` function is used to verify the contents of the message – whether it was properly formatted, contains authentication data, and whether provided credentials could be confirmed to be valid.

Once this is complete, the function continues to wait until a second message is received. This message is expected to contain the ID of the match that the client is attempting to join, and is validated using the `validateMatch` function. The order of the messages is important – receiving match data before auth data will render the connection attempt invalid, and the connection will be terminated. This was a conscious design decision, in an attempt to remove the ability for a client without valid credentials being able to find out information about matches that may, or may not exist.

If both validation steps are successful, the connection is passed onto the game server itself, by calling the `AddClient` function available on the `game.Server` pointer that is passed in as an argument to the `HandleGSConnection` function.

The game server handles matches, as well as new client connections, and the removal clients that have disconnected. It is also responsible for updating the state of each match based on updates from the connected clients, detecting invalid moves, and detecting when a match enters a state from which no more moves can be played (and therefore the match must be ended).

The entirety of the game server's logic is handled by the `MainLoop` function which is run inside of a goroutine when the server is created. A single synchronous loop iterates until the server is shut down, or something causes the server to panic[xviii]. The server has a minimum tick rate of 250ms, which may increase if the server is experiencing unusually high loads. Each tick, three channels (thread safe communication channels through which data can be passed) are examined to see if they contain any data. These channels are then drained, on a first come first serve basis, until they are all empty.

The four channels (connect, disconnect, broadcast, and commands) receive messages from both the server, and individual clients.

The broadcast channel is used to inform the server that a particular message needs to be sent out to all clients, while the command channel is used to control, or request information from, the server itself.

The connect channel contains new client connections, that the server must attempt to place in a match. If the match that the client is requesting to join does not yet exist, the match will be created – otherwise, they will be added to the one that exists. There are a number of edge cases that are handled, such as when a client reconnects to a match from which their previous connection is yet to be removed, and when a client joins a match where the other client is currently disconnected after having connected already.

When a match (a `Match` that contains both client connections, and is responsible for updating itself based on updates from the players) has been successfully filled with the two designated players (defined by a row in the database that was generated by the matchmaking server), the `match.SetMatchStart` is called, and the server will begin calling the match's `Tick` function.

The `disconnect channel` is used to inform the server that a particular client, and in some cases the match they are associated with, needs to be terminated. The channel accepts

[xviii] A panic is when something unexpectedly goes wrong, such as an unhandled error, or when something tries to access something to which it does not have permission. See: https://gobyexample.com/panic/.

messages of type `DisconnectRequest` that describe which client needs to be removed from the server. The disconnection process can be seen here in the `handleDisconnectRequests` function. It's worth noting that the disconnect channel is handled separately to the other channels – this is to avoid race conditions, such as when a client attempts to join a match that already existed, with a player that is pending disconnection and has been removed from the match.

When a match is determined to be complete, the server updates the match information for the corresponding match in the database by calling `match.SetMatchResult` which sends a request to the Blade II REST API, in order to update each players match stats and Elo rating.

Both the Matchmaking, and the game server both interact with the database using the database package, though due to similarities with the database package found in the REST API, this will not be covered.

Testing and Evaluation

Operational Testing

Testing was an ongoing process throughout the development of Blade II Online, typically in the form of iterative, operational testing. It was initially planned to make use of the TDD methodology when implementing features that were expected to return certain outputs, or behave in certain ways based on particular inputs, but it quickly became clear that this was taking too much time. In a larger project, it would almost certainly be worth doing, or at the very least, implementing tests post-implementation in order to detect regression of features, but in this instance it was deemed an unnecessary step that wasted more time in the short term – valuable time that was better used implementing more features, rather than polishing existing ones.

Formal testing was reserved for the completion of major features, and therefore the majority of tests were carried with the following pattern:

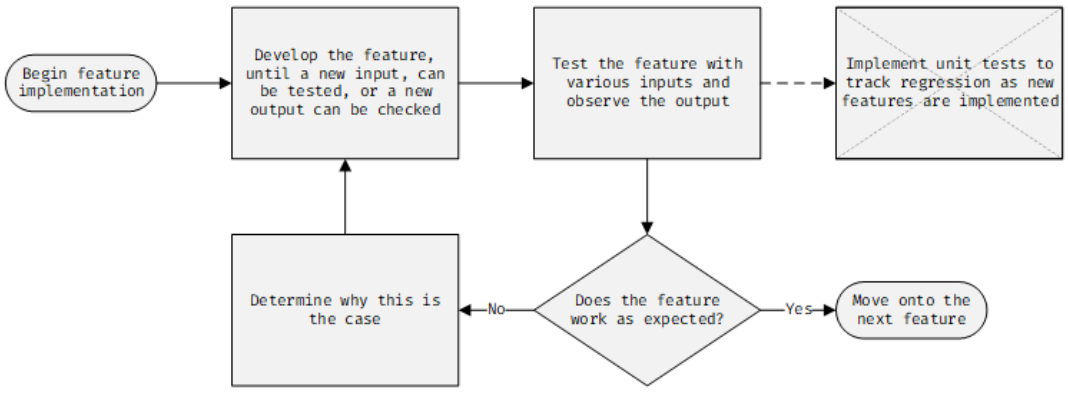

Figure 8. A flow chart that illustrates the iterative operational testing method utilised during the development of Blade II Online.

Note the top right tile, a step which, in an ideal world, would be carried out using automated unit and integration testing tools, but was omitted during the development of Blade II Online.

This testing method tied in favourably with the Agile Kanban development methodology described in the analysis section, ensuring a tight iteration loop with immediate feedback upon change, whilst also maintain a temporal proximity between the features being implemented, and the tests that were performed. Temporal proximity is especially important in an environment where multiple different projects are being handled at a time, such as for a university student. By immediately testing features as they were implemented, bug fixing and troubleshooting was made significantly more efficient: An engineer who has recently been involved with the development of a particular feature, and has spent a lot of their recent time engrossed in the task, can be considered to be *in the zone*, described by Hill (2017) as 'a hyper-focused, sometimes spiritual, state of mind where anything is possible. It's where we become our most productive, creative, and powerful selves'.

A significant portion of the testing carried out in this fashion was all white box testing. Using the debugging tools included with Visual Studio and Visual Studio Code, logical, flow, and functional testing was carried out.

When developing the game client, for example, encountering program halting errors was fairly common. In these instances, if the problem were not immediately obvious, the program would be restarted by launching it using Visual Studio, so as to enable breaking on exceptions, and to allow manual breaking. Inspecting variables, and following the flow of the program at runtime proved to be an invaluable technique for squashing bugs.

Play Testing

Near the end of the development, the game client reached a stage where a match could be fully played, from start to finish. Initially, only AI matches were supported, but eventually network matches became possible as well.

Due to time constraints, AI matches were not tested by third parties, instead being tested in real time, each time a new feature was being implemented. Network games, however, required testing with two instances of the game client, and while this was possible to emulate by running two instances of Unreal Engine 4, a few colleagues were invited to play test the game, to try and identify bugs, and features that were in some way unsatisfying.

Colleagues were then asked to fill in a SUS test form, as well as discussing their thoughts over Discord. One of the completed SUS forms has been included in the appendix here. The overall result of the test was a SUS score of 87.5 which, is a SUS grade of A (somewhere in the 95th percentile of scores). Worth noting is the selection of neutral for the question "I think that I would like to play this game frequently", which was a common trend amongst play testers. This is not entirely unexpected, however, as the game was designed and developed as a proof of concept, rather than an actual game that was expected to be played, and therefore much of the development time was spent focusing on implementing core features, rather than attempting to make the game more fun (this would also draw the game away from its original inspiration which, as stated in the introductory section, was a undesirable outcome).

Some of the verbal feedback received from, play testers has been included in the appendices here as well. The feedback was used to move around various tasks in terms of priority, and also helped to shape the development plan for Blade II Online in the near future, in preparation for presentation to potential employers.

Stress Testing

Once both the matchmaking, and the game server had been implemented, and the majority of the major bugs ironed out, stress tests were carried out to find out how each server would behave under high load. Stress tests were not carried out for the REST API, as breaching the free tier usage limits may have financial repercussions.

Each server endpoint was tested to see how many connections it could handle before the server either stopped responding, or crashed. Tests were carried out locally, so as not to trigger Cloudflare's DDOS and/or anti-spam protection, which could lead to the source IP address being blacklisted.

Tests were carried out using Artillery[xix], and run using Mingw-w64[xx]. Some of the results can be seen here in the appendix.

[xix] Artillery is a tool for load/functional testing. See: https://artillery.io/.

[xx] Mingw-64 is a development environment for Windows that supports GCC. See: http://mingw-w64.org/doku.php

Requirements Review

A review of the MoSCoW Requirements table was carried out once Blade II Online was determined to have reached a point where it could be considered to be a minimal viable product for the original purpose. The results of this analysis can be seen in the appendix here.

As can be seen, 90% of the Must Have requirements were either achieved or partially achieved, along with 64% of the Should Have requirements. The majority of the partially completed features will be implemented at a later date, in order to fully realise the vision of Blade II Online. Best endeavours will be used in order to include as many of the missing features as possible before the VIVA presentation for this project.

Legal, Social, Security, and Ethical Issues

Age Rating and Target Audience

First and foremost, the main target audience for Blade II Online are fans of the Legend of Heroes series. At its core, Blade II Online is a simple card game, with easy to learn rules and a shallow learning curve. While thematically, and aesthetically, it is intended for fans of the aforementioned series, its shallow learning curve, and immediate access to playing games online, should be appealing for casual gamers and card game enthusiasts regardless of whether they are fans of the aforementioned series.

The age rating for Blade II Online is the lowest rating available for all relevant ratings agencies. It does not feature content that typically warrants a less-than-minimum content rating for games, such as violence, bad language, or gambling, and is intended to be suitable for players of all ages.

Ratings board	Rating
PEGI	3
ESRB	Everyone
CERO	A 全年齢対象 (Zen Nenrei Taishō / “For All Ages”)

Table 5. Blade II Online age rating targets.

One thing worth noting is that online interactions will not be rated at all. Blade II Online will eventually include a communication channel to be used during matches through which players can communicate via preselected messages, similar to Rocket League’s quick chat (Khan, 2016). In the future, it may be possible to add real-time chat or voice communications which could be used to communicate with their opponent. This system will be optional, however, and in an ideal world would require age verification to enable – though this is out of scope.

Ethical Considerations

Blade II Online includes player avatars which will be selectable from a predetermined set of characters. These characters feature in the Legend of Heroes universe and are included purely for visual purposes. While these are indeed humanoid characters, there is no

discrimination, abuse, violence, or sexualization portrayed. All other objects in the game are inanimate, and the only interactions that can occur are the rule-based interactions that dictate gameplay, such as selecting a card, configuring the games settings, or observing the animation and evolution of the game state after a card was played.

Some of the cards feature images of fictional weapons that may be rendered in such a way that they appear to have a physical form in the game. This is offset by the fact that they are cell shaded and use textures to provide most of the detail, and do not look particularly realistic.

Some of the sound effects imitate explosions-like sounds or aim to sound like thunder and lightning, and other natural phenomenon such as whirlwinds. There is a chance that this could trigger a PTSD response for certain players, and so the game offers the option to mute all sound effects.

Intellectual Property

As this game is based on a minigame from select Nihon Falcom games, it contains IP that belongs to them. In order to ensure that Blade II Online does not infringe on any copyright laws, special attention will be payed to ensure that all usage falls under the 'Non-commercial research and private study' clause described in the Exceptions to copyright guide published by the Intellectual Property Office (2019). Under no circumstances will the game be commercialized, without first either stripping out all the copyrighted content, or after receiving appropriate permission from the owner of the IP.

Due to the non-commercial, educational nature of the game, Unreal Engine 4 can be used without paying any fees. 3D models were created using Maya 2018, with a student license.

Any assets (media, software etc.) from external sources are used either with permission, or in accordance with the aforementioned 'Non-commercial research and private study' clause, and clearly credited in the third-party licenses section of the in-launcher settings menu, as well as in this document. Where possible, an appropriate license has been requested from the third-party, and/or prior consent for use of the asset has been sought.

The music used in the game is also Nihon Falcom IP. According to 'Falcom's Free Music Use Declaration' it is entirely legal to use any song published by them as long as, amongst other things, the use case is non-commercial and an appropriate credit is included in along with the work within which the song is featured (Falcom, 2010).

Sound effects were either created using Bfxr, a free tool with an Apache 2.0 License that can be used to create effects, or extracted directly from a purchased copy of *The Legend of Heroes: Trails of Cold Steel II*. All sounds created with Bfxr are free to use for any purpose (increpare, no date).

All code written for this project will eventually be uploaded to a public GitHub repository and released under an MIT license, and open sourced for the benefit of all.

Data Protection

As the Blade II Online back-end infrastructure stores various data in a remote database, it must abide by various data protection laws. When creating an account, users will be informed exactly what data will be stored, and what will be done to said data.

In accordance with the Data Protection Act 2018 (Department for Digital, Culture, Media & Sport and Home Office, 2018), special care has been taken to ensure that data is:

1. Used fairly, lawfully, and transparently.
2. Used for specified, explicit purposes.
3. Used in a way that is adequate, relevant, and limited to only what is necessary.
4. Accurate and, where necessary, kept up to date.
5. Kept for no longer than is necessary.
6. Handled in a way that ensures appropriate security, including protection against unlawful or unauthorised processing, access, loss, destruction, or damage.

The sensitive data that is stored is limited to just an email address, a username, and a password. The email address will be required for account validation and password resets/recovery, whilst the username and password will be used for authentication, and as a display name in matches and in leaderboards. The password itself is never stored – rather, a

salted, hashed version is stored in its place, which is sufficient for checking if a password is the same as the one that was used to generate the salted hash, but is sufficiently hard to crack.

Critical Review

As the development reaches its final stages, it appears as though Blade II Online has realised its primary goal – that is, to serve as a proof of concept for an online version of Blade II, as covered in detail in the personal aims and objectives section. All four major components are functional, and through their interactions, it is possible to open the launcher, create an account and/or sign in, and either search for an online match, or start a match against an AI opponent. With some polish, bug squashing, and implementation of the missing features, it will certainly reach a state that fully realises its objective of acting as a proof of concept.

Known Issues

Issue	Comment
The game occasionally crashes when the opponent plays a blast card	The cause of this crash is unknown and, as it happens only sporadically, is proving very hard to pin down.
The timings for some of the cards are inaccurate relative to the original game, due to the animation system being reworked	The animation system rework was a necessary resolution to a problem that stemmed from animations being unable to detect the state of other animations in the same animation group. This led to situations where either the first or last animation in a group would end up out of sync by a single frame. While this may not sound like a problem and it is not a game breaking bug, it was a very obvious visual bug that could be seen every time the cards were initially brought into play.
Match stats are not updating properly	It was observed that a match may have been recorded with the incorrect winner. However, this may not be the case, as it is not clear exactly which match this record related to as the issue was not realised until a while after the play test was carried out.
For some reason, the launcher is unable to detect certain types of WebSocket disconnection. When the matchmaking server fails, the launcher is often unaware.	A keepalive message pump should be implemented, that checks to see if the WebSocket connection is still alive.

The AWS free tier offer will run out in around 9 months, and hosting Blade II Online's networked features will start to cost money.	It is likely that job hunting will be successful before this becomes an issue. If not, each component can be migrated over to a fresh AWS account (this has already been carried out once before).
Client-side win condition checking has some inconsistencies	The win condition logic will be re-examined after submission of this document.

Time Management

As discussed in the development methodology section, the initial development plan involved a set of tasks, laid out chronologically in the form of a Gantt chart. However, as development progressed, it became apparent that the allocation of development time that was initially laid out was inaccurate – development was rarely completed within the allotted time, typically overrunning due to unexpected bugs, or having to divert from the main task in order to implement a dependency that was previously unknown.

Combined with the additional workload of other university subjects, by the end of 2019 development was significantly delayed. A major reason for this was the decision to remove the majority of the game's non-game related interface to a separate application – in hindsight, this may not have been the best decision in terms of workload, but from an aesthetic and usability perspective, has helped to better realise the aim of this project.

Due in part to the unexpectedly high volume of work from other university subjects, and in combination with the development of a crippling addiction to a certain MOBA during the second semester, there were large gaps in development during which other work and activities were prioritised. This led to the temporary abandonment of the original Gantt chart in late October 2019, followed by it being discarded shortly after reading week in 2020.

Instead, the Agile Kanban methodology was fully embraced, with a shift towards managing time in an entirely ad-hoc fashion. Using Trello, requirements were added as tiles, and internal TODO lists were used to keep track of tasks required to implement them, as can be seen in this image. Each task was added with the end goal of creating a working proof of concept that implemented at least one possible user journey, before the submission date in May. Once a task was complete, the next feature that was required was determined, and the

tasks that were immediately required in order to begin implemented said task were added to the Kanban board. This process would continue until the feature was complete.

Lessons learned

All things considered, the biggest factor that led to poor time management throughout development was the sheer scale of the project. What started as a plan to develop two components (the game, and a server to support it) eventually spiralled out of control, into a full-stack implementation with a back-end that was, in theory, both horizontally and vertically scalable. The development of the launcher application was no small feat either, being built from the ground up specifically for the requirements of Blade II Online.

Included below is a selection of noteworthy pain points and takeaways that became learning experiences, over the course of the development of Blade II Online.

Start Small, and Extend

Due to the scale of the project, and the sheer volume of features that needed to be implemented and tested for each component, it became difficult to determine how much time could be allocated to a particular task at any time. It was fairly common for the time taken to complete a task to differ from the original plan, even by multiple days in some cases. Eventually, time planning was almost entirely abandoned, in favour of iteratively checking to see what was still required in order to develop the minimal viable product – a prototype that implemented the whole user journey from application launch, through to playing a match.

In hindsight, it would have been significantly better to begin planning with an end goal of achieving a minimal viable product, which could then be iteratively improved until time ran out.

The Network is (Not) Reliable

A common fallacy surrounding the development, an opinion often held by those not yet familiar with the development of cross network applications, is that *the network is reliable*. In reality, this is usually not the case, as described by Chircu (2018). While initial plans for the network components of Blade II Online did include error checking, with appropriate

outputs that could be handled by the client, it failed to properly define a plan for the development of systems that would ensure that clients could reliably perform the cross-network tasks that they were attempting. For example, when a player disconnects from a match that has commenced play, the match is immediately terminated and the player that disconnected is determined to have forfeited the match.

When play testing, testers with less reliable internet connections occasionally found themselves booted out of a match, especially during initialisation of the game client. As a temporary measure, various server-side timeouts were increased in duration, though this has a negative impact on games where a player force closes the application, as the other player will often be unaware of this until the timeout period has elapsed.

It may have been better to design the system with the opinion that the network was always unreliable, and instead handle a reliable connection as the edge case.

Nothing Ever Goes According to Plan

There was not a single feature implemented throughout development that went exactly as initially planned. It was very common for the development of a feature to branch off into the development of a particular requirement for said feature, or for development to be temporarily suspended due to bugs being discovered, or when a particular aspect of a feature had to be redesigned due to the original plan being inadequate or unsuitable.

Whilst it was originally thought that this ongoing problem stemmed from a lack of developer experience, it appears as though it may be a common trend that surrounds software development. ‘Our plans are typically based on best-case scenarios and “yield overly-optimistic predictions of completion times.”’ describes Collingwood (2018), and as originally stated by Hofstadter (1980), ‘It always takes longer than you expect, even when you take into account Hofstadter's Law’.

In the future, it may be better to plan projects with overall targets and goals in mind, and handle the allocation of resources as and when the requirement of a new task becomes apparent. This lends itself well to various Agile methodologies, which is by far the most

common development pattern for modern software development, according to Kukhnavets (2018).

Always Verify Data

When receiving a message from a remote client, it is paramount that the message is checked to ensure that it contains the expected data, and that the data is within the expected range.

Each network facing endpoint on the REST API, and the game server, required the addition of multiple stages of verification and error checking to ensure that only valid requests were handled, and invalid ones were returned with a suitable error response.

Games are Deceptively Complicated

It was initially believed that the implementation of the game client would be one of the least demanding aspects of development. However, it soon became apparent that this was not the case.

For example, when developing an FPS game with Unreal Engine 4, it is possible to make use of a vast selection of prebuild actors and components that implement common features and, in combination with the physics engine, make it possible to implement common mechanics with relative ease. However, in the case of a card game, no such building blocks exist, and in the end almost all of the engine and gameplay logic was implemented from scratch.

Developers often take these readily available tools and components for granted, without realising, or even consciously thinking about how much time an effort was required to implement them.

Personal Achievements

Aside from the experience gained throughout development, and the creation of a working proof of concept prototype that can be presented to potential employers, there were a few additional achievements that are worth highlighting. Blade II Online was more an exercise of personal growth, rather than an attempt to create a fun game, and in this sense it has been a roaring success.

The main goal of this project was to increase employability, and this will hopefully bear fruit when job hunting starts in earnest once the final deliverable for the current university year is submitted.

A Standout Portfolio Piece

The development of Blade II Online has been successful in realising its main goal of creating a working proof of concept cum prototype that can be presented to potential employers, in order to demonstrate the skills and knowledge gained during the time spent at Kingston University.

With Blade II Online, and along with various other projects creating before and during the time spent at Kingston University, it has been possible to compile a strong portfolio of work that effectively displays the wide range of applicable skills and technologies that will hopefully be attractive to potential employers.

Go Development Experience

Go was successfully used to implement three core aspects of the Blade II Online ecosystem, serving as a tremendous learning experience, and providing a valuable augmentation to the range of languages that can be included in a curriculum vitae or mentioned during an interview.

According to Hired (2020) Go is currently the most sought-after programming language in the world, and therefore having experience with coding in Go should significantly bolster the chances of finding a good graduate job.

C++ Development Experience

Valuable experience was gained in planning, writing, refactoring, and troubleshooting C++ code – one of the major aims of this project. Though Unreal Engine 4 enforces many non-standard coding patterns, and also encourages practices that don't fit particularly well into modern C++ coding standards (such as requiring pointers to classes derived from UObject to be stored as a raw pointer instead a variant of a smart pointer), the experience has led to a significant increase in confidence when it comes to developing software using C++.

Unreal Engine 4 Experience

Valuable experience working with, and writing C++ code for Unreal Engine 4 was gained throughout the development of Blade II Online. While this was touched on during the second year of university, it was not covered in much detail, and development of the project at the time was carried out almost exclusively using Blueprints.

This experience acts as not only a proof of being able to work with Unreal Engine 4, but also as proof of being able to work with any engine that utilises C++.

Future work

For the immediate future, the plan is to clean up and polish existing features, clean up and refactor code bases, and then implement some of the missing features that were not implemented due to time constraints. These missing features have been listed below.

Launcher Features

1. The profile page in the lobby, where players can view their profile, see their ranking, and change their profile information (including avatar and handle). This page will also be used to display the profiles of other players, which can be accessed by clicking their name in the rankings page.
2. The Rankings page in the lobby, where players can view a list of the top Blade II Online players.
3. The home page in the lobby, that will display appropriate, and contextual information related to the game, which is updated when new information becomes available (akin to the home page in the League of Legends launcher for example).
4. Onboarding hints for first time players, including popups that guide players through the account creation process, and guide them towards the tutorial.
5. The tutorial interface, a set of illustrated pages that describe the rules for Blade II Online, as well as eventually launching the game client with a tutorial game.

Game Client Features

1. A tutorial game, a preconfigured match that follows a set path, will be added. The player interacts with the match according to hints and messages that appear on the screen, learning about the game's rules in an interactive manner.
2. The card animations will be cleaned up and adjusted so that they match the animations found in the original version of Blade II.
3. A bug, that causes the entire game client to freeze when the client is moved around the screen, will be investigated and hopefully fixed.

Best endeavours will be made in order to implement all these features in time for the VIVA.

Appendices

Third Party Code

Go Packages (Libraries)

1. Gorilla WebSocket, a Go implementation of the WebSocket protocol. See: https://github.com/gorilla/websocket/.
2. Argon2id, a Go library that implements a convenient interface for interacting with Go's argon2 implementation. See: https://github.com/alexedwards/argon2id/.
    a. Argon2 is a password hashing algorithm that won the Password Hashing Competition that ran from 2013 to 2015. See: https://password-hashing.net/#argon2.
3. AWS Lambda for Go, a Go library designed to facilitate the development of AWS lambda functions using Go. See: https://github.com/aws/aws-lambda-go/.
4. xid, a Go library that implements a GUID generator that produces sortable, 20-character ID's. See: https://github.com/rs/xid/.
5. Go MySQL Driver, a MySQL driver for Go's database/sql package. See: https://github.com/go-sql-driver/mysql/.

Node Modules (additional JavaScript libraries)

1. electron-settings, a user settings framework for Electron. See: https://github.com/nathanbuchar/electron-settings/.
2. JS-YAML, an implementation of the YAML spec. See: https://github.com/nodeca/js-yaml/.
3. markdown-it, a markdown parser. See: https://github.com/markdown-it/markdown-it/.
4. Request, an http client. See: https://github.com/request/request/.

Unreal Engine 4 Plugins

1. UEWebsocket, a WebSocket plugin for Unreal Engine 4. See: https://github.com/feixuwu/UEWebsocket/.

Images

https://drive.google.com/file/d/1VE8xWBn-QTuPw0sn6ON-rlJb8qNabMzm/.

Figure 9. The Gannt chart that was designed to assist in the organisation of developer time throughout the course of development.

https://drive.google.com/file/d/1HrxpQB_fenZqUA8HVEKpt8FgIM6jl4oJ/.

Figure 10. A screenshot of Blade II gameplay from Nihon Falcom's title *The Legend of Heroes: Trails of Cold Steel II*.

https://drive.google.com/file/d/1-Si-EkHHXLO8Xpm7n3Ms51RmBpma-FWZ/.

Figure 11. A screenshot of the battle menu from Nihon Falcom's title *The Legend of Heroes: Trails in the Sky FC Evolution.*

https://drive.google.com/file/d/1Vnee8ikP-e6QABQZ4rEO36HASs-J-Qce/.

Figure 12. Architecture diagram for Blade II Online.

https://drive.google.com/file/d/1D_jbagya-2W0wCxjRnycNqLcNoJKRNR3/.

Figure 13. Class diagram for the Blade II Online launcher.

https://drive.google.com/file/d/1jACuyOa8jqB3VJQWxfBJTsflwkhDH-oi/.

Figure 14. Rows returned as a result of the match history query.

https://drive.google.com/file/d/1_9hLthHSJG2ZNe-xuywuxlhnI4yB0dQm/.

Figure 15. An example of the TODO list for one of the tiles on the Trello board used to keep track of tasks and development time.

https://drive.google.com/file/d/1Gt6ngS6n9gcmptOCqOKa2-BwJOU4FTHF/.

Figure 16. A comparison of an original Blade II card (Bolt) and the recreated for use in Blade II Online.

https://drive.google.com/file/d/1M5daPvpuszmEaSHk9-t2A3zaCaCrVqX9/.

Figure 17. Class diagram for the Blade II Online game server.

https://drive.google.com/file/d/1-xYqadSEsqnrxVro7lEwKyxsODRhDAMr/.

Figure 18. Class diagram for the Blade II Online REST API.

https://drive.google.com/file/d/13pz_a027KRko3_d_HCuPNJgPl3tqYit0/.

Figure 19. Relationship diagram for the Blade II Online game client's source code.

https://drive.google.com/file/d/1ObrrnXWjF_95qIHRjaHf_r-3dt5TVu5o/.

Figure 20. Blade II Online launcher: login Screen.

https://drive.google.com/file/d/1Pni2M_5njmUHpxnC5J6UJj8cH8SoD9CZ/.

Figure 21. Blade II Online launcher: account creation screen.

https://drive.google.com/file/d/1B9qQNnd2KZZfoQFqvIAUZ93qibYZIm8d/.

Figure 22. Blade II Online launcher: lobby.

https://drive.google.com/file/d/1L5n-bUywcItTFXI5xfak_O6OAnTYFzFX/.

Figure 23. Blade II Online launcher: match type selection.

https://drive.google.com/file/d/13N16g-47NSCz_pk3zYdV0We4L-oX7o0S/.

Figure 24. Blade II Online launcher: match type selection (Japanese).

https://drive.google.com/file/d/1toqhecbt62i1rrTMhDwwYHPcQl0UPWiU/.

Figure 25. Blade II Online launcher: options menu.

https://drive.google.com/file/d/1ay-mFRcNAkhs-XvWuWy08FJlrXY05JAR/.

Figure 26. Blade II Online game client: loading screen.

https://drive.google.com/file/d/1h4B_QZD_ODPKV6_ESDJWVGGa5wazxUBh/.

Figure 27. Blade II Online game client: gameplay screenshot 1.

https://drive.google.com/file/d/1pz8wSBxELGv_v0PWBTy3-wH7-5EU-qQq/.

Figure 28. Blade II Online game client: gameplay screenshot 2.

https://drive.google.com/file/d/1Twy5RtiJt4pK3kCABfvPJGHqG6Ra4gLu/.

Figure 29. Blade II Online game client: gameplay screenshot 3.

Tables

Must Have	Should Have	Could Have	Won't Have
In-client account creation	Game controller compatibility	In-game messaging with pre-set text	PS4 build (lacking appropriate license)
Login with username and password	Server-side validation for player actions	Animated 3D avatars in menus and player profiles	Microtransactions
Login, settings, queue, game, post-game, leaderboard game states	SSL/TLS encryption for all network communications	Ability to reconnect to unfinished games	Mobile app for Elo leaderboards, user profile lookup etc.

In-client password reset via email and front-end webpage	Multi-platform builds (PC, mobile, consoles if available)	Add other players as friends and start games with them directly	In-game chat with unrestricted text input
Offline AI opponent	In-client Elo rating high score tables	Unranked queue with no visible Elo rating	Paid DLC
Optional tutorial vs. AI opponent	Classifications for each Elo range	Web-app Elo leaderboard	Card collecting mechanics
Multiplayer matchmaking queue	Username profanity filter	Web-app game client	Dynamic matchmaking queue that adapts to players based on recent win-percentage
Token based authentication for the game server	UI animations	Free DLC	
Persistent data stored in remote SQL database	Customisable options such as volume and graphics quality	Optional advertisements	
Mouse & Keyboard compatibility	Option to remember username	Customizable card design	
WebSocket network communication with a central relay server built with Go	Cross-platform multiplayer	Customizable playing table design	
REST API server built with Go (auth, queries etc.)	Credits and license information section	Animated Login screen	
Customizable player avatars	Post-account creation email confirmation	Game launcher built with Electron	
Elo leaderboard with divisions that group players by percentile	Initial seeding period for new players where Elo shift is inflated	3D parallax effect for cards	

Client-side validation for player actions		Optional AR mode	
Fully featured UI, including window controls		Optional in-game voice chat	
Fully Localized for Japanese & English		Theme and story introduction	
Background music and sound effects		Lore and lore explanation	
Match History		Flavour text for cards	
Animated 3D avatars in-game			

Table 6. Blade II Online feature priorities organised using the MoSCoW Method.

	Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	I think that I would like to play this game frequently			✓		
2	I found the game unnecessarily complex	✓				
3	I thought the game was easy to play					✓
4	I think that I would need the support of a technical person to be able to play this game	✓				
5	I found the various functions in this game were well integrated					✓
6	I thought there was too much inconsistency in this game		✓			
7	I would imagine that most people would learn to play this game very quickly					✓
8	I found the game very cumbersome to play	✓				
9	I felt very confident interacting with the various controls of the game				✓	
10	I needed to learn a lot of things before I could get going with this game		✓			

Table 7. Results of one of the SUS tests carried out by a colleague during the play testing phase of Blade II Online.

Must Have	Should Have	Could Have	Won't Have
In-client account creation	Game controller compatibility	In-game messaging with pre-set text	PS4 build (lacking appropriate license)
Login with username and password	Server-side validation for player actions	Animated 3D avatars in menus and player profiles	Microtransactions
Login, settings, queue, game, post-game, leaderboard game states	SSL/TLS encryption for all network communications	Ability to reconnect to unfinished games	Mobile app for Elo leaderboards, user profile lookup etc.
In-client password reset via email and front-end webpage	Multi-platform builds (PC, mobile, consoles if available)	Add other players as friends and start games with them directly	In-game chat with unrestricted text input
Offline AI opponent	In-client Elo rating high score tables	Unranked queue with no visible Elo rating	Paid DLC
Optional tutorial vs. AI opponent	Classifications for each Elo range	Web-app Elo leaderboard	Card collecting mechanics
Multiplayer matchmaking queue	Username profanity filter	Web-app game client	Dynamic matchmaking queue that adapts to players based on recent win-percentage
Token based authentication for the game server	UI animations	Free DLC	
Persistent data stored in remote SQL database	Customisable options such as volume and graphics quality	Optional advertisements	
Mouse & Keyboard compatibility	Option to remember username	Customizable card design	

WebSocket network communication with a central relay server built with Go	Cross-platform multiplayer	Customizable playing table design	
REST API server built with Go (auth, queries etc.)	Credits and license information section	Animated Login screen	
Customizable player avatars	Post-account creation email confirmation	Game launcher built with Electron	
Elo leaderboard with divisions that group players by percentile	Initial seeding period for new players where Elo shift is inflated	3D parallax effect for cards	
Client-side validation for player actions		Optional AR mode	
Fully featured UI, including window controls		Optional in-game voice chat	
Fully Localized for Japanese & English		Theme and story introduction	
Background music and sound effects		Lore and lore explanation	
Match History		Flavour text for cards	
Animated 3D avatars in-game			
% Completed or partially completed			
18 / 20 = 90%	9 / 14 = 64%	3 / 19 = 16%	

Table 8. Blade II Online feature priorities organised using the MoSCoW Method, this time with completed, partially completed, incomplete, and abandoned features highlighted (in the colours shown here).

	Connections							
Server	10	50	100	200	400	800	1600	3200
Game	✓	✗	✗	✗	✗	✗	✗	✗
Matchmaking	✓	✗	✗	✗	✗	✗	✗	✗

Table 9. A table showing the results of the first batch of stress tests.

Server	Connections							
	10	50	100	200	400	800	1600	3200
Game	✓	✓	✓	✓	✓	✗	✗	✗
Matchmaking	✓	✓	✓	✗	✗	✗	✗	✗

Table 10. A table showing the results of the second batch of stress tests, after the maximum channel size was increased to 256.

Server	Connections							
	10	50	100	200	400	800	1600	3200
Game	✓	✓	✓	✓	✓	✓	✓	✗
Matchmaking	✓	✓	✓	✓	✓	✓	✓	✗

Table 11. A table showing the results of third batch of stress tests, after numerous changes were made including the inclusion of mutex locks for critical sections, and a general overhaul of the client removal system.

Text

> "I wasn't sure what to do once the launcher loaded. I didn't know what account to use to login, and I didn't notice the create button for a while. Maybe it needs to be bigger? Or I guess if there was a popup the first time you play or something..."

Figure 30. Feedback received from a play tester when asked about the initial onboarding process.

> "I couldn't even tell it was working online. It wasn't laggy or anything. I guess it's not a fast-paced game like [popular FPS game], so maybe you wouldn't know that there is lag unless it's really bad... What happens if I lag so hard that I disconnect? Do I lose the game?"
>
> "... I guess that makes sense, though it's annoying that you can't reconnect."

Figure 31. Feedback received from a play tester when asked about the network performance of a match.

> "It was really confusing at first, I had no idea what the cards do, so I just kept choosing the ones that looked cool or had the highest scores. But after a few games I realised how to play, and honestly it was pretty fun for a bit."

Figure 32. Feedback received from a play tester when asked if they found the game intuitive and easy to learn.

Third-Party Assets

Note that, aside from a select few, the majority of the textures used in Blade II Online's various components were either obtained third party assets, or adapted from textures extracted from a purchased copy of The Legend of Heroes: Trails of Cold Steel II.

A list of original textures is included in the original assets section.

1. Assets extracted from, or screen-shot from a purchased copy of The Legend of Heroes: Trails of Cold Steel II, a game created by Nihon Falcom: http://www.trailsofcoldsteel.com/cs2/.
    a. Multiple textures.
    b. In-game screenshots, including those taken during cutscenes, videos, and during gameplay.
    c. Multiple effect sounds.
    d. Models extracted by GitHub user uyjulian: https://gist.github.com/uyjulian/6c590476819bf3bfde6fc78aa3765698/.
2. Blade II Online logo, commissioned for creation by fiver seller aishdesigner: https://www.fiverr.com/aishdesigner/design-modern-logo-design/.
3. Breathing animation, generated using Mixamo: https://www.mixamo.com/.
4. Icons downloaded from FLATICON.
    a. Icons by Freepik: https://www.flaticon.com/authors/freepik/.
    b. Icon by Those Icons: https://www.flaticon.com/authors/those-icons/.
    c. Icon by Kiranshastry: https://www.flaticon.com/authors/kiranshastry/.
    d. Icon by Pixel perfect: https://www.flaticon.com/authors/pixel-perfect/.
    e. Icon by Good Ware: https://www.flaticon.com/authors/good-ware/.
5. Fonts from Google Fonts.
    a. Noto Sans JP: https://fonts.google.com/specimen/Noto+Sans+JP/.
    b. Noto Serif JP: https://fonts.google.com/specimen/Noto+Serif+JP/.
    c. Fira Code: https://fonts.google.com/specimen/Fira+Code/.
6. Fonts from Adobe Fonts.
    a. Trajan: https://fonts.adobe.com/fonts/trajan/.

7. Card shuffling sound, created by freesound user tymau: https://freesound.org/people/tymaue/sounds/79617/.
8. The Legend of Heroes: Trails of Cold Steel II Japanese logo: https://www.falcom.co.jp/page/wp-content/uploads/2019/02/jan4956027126925_logo.png/.

Original assets

This is a list of the original texture, model, and sound assets created for use in Blade II Online. If an asset is not listed here, it was either taken directly from a third-party source, or adapted from a copy of an asset taken from a third-party source (listed in the third-party assets section).

1. Game client textures
    a. All of the textures in the /Textures/Sprites/Endgame folder in the contents directory of the Blade II Online game client
    b. All of the textures in the /Textures/Cards/ folder in the contents directory of the Blade II Online game client
    c. T_Blast_Whirl.tga
    d. T_DownArrow_24x24.tga
    e. T_ComboBoxOptionBorder.tga
    f. T_Mirror_Glow.tga
    g. T_Mirror_SpecularSpritesheet.png
    h. T_SpinningCard_Spritesheet.png
    i. T_Up_Test.tga
    j. T_Utility_1x1_White.tga
    k. T_Arena_Edge.tga
2. Launcher textures
    a. app.png
    b. 1x1-transparent.png
    c. 880x160-transparent
    d. scanlines.png

3. Game client models
    a. SM_Arena.fbx
    b. SM_Card.fbx
    c. SM_Card_Cursor.fbx
    d. SM_Card_Highlighter.fbx
    e. All of the models in the /Meshes/Digits folder in the contents directory of the Blade II Online game client
4. Game client sounds
    a. S_Callout_TextPulse.wav
    b. S_Cursor_Navigate.wav

Blade II Online Game Client Docs

The documentation for the Blade II Online game client, generated using Doxygen, can be seen here:

https://6a.github.io/b2o-game-client-docs/index.html.

Not all of the pages work, but the class list does, providing an overview of all the classes and their public interfaces, as well as class and relationship diagrams.

Source Code

Repository Links

Blade II Online Game Launcher
https://github.com/6a/blade-2-online-client-electron/tree/dfs/.

Blade II Online Game Client
https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/tree/dfs/.

Blade II Online REST API
https://github.com/6a/blade2-serverless-api/tree/dfs/.

Blade II Online Game Server
https://github.com/6a/blade-ii-game-server/tree/dfs/.

Code Snippets

Blade II Online Launcher

1. `basemodel.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/model/basemodel.js/.
2. `netmodel.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/model/netmodel.js/.
    a. `sendAccountRequest`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/model/netmodel.js#L108-L131.
    b. `onCreateAccountResponse`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/model/netmodel.js#L65.
    c. `startMatchMaking`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/model/netmodel.js#L156-L167.
3. `event.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/event.js/.
    a. `B2Event`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/event.js#L16-L49.
4. `websocket.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/utility/websocket.js/.
    a. `B2WS`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/utility/websocket.js#L1-L58.
    b. `onWebsocketEvent`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/model/netmodel.js#L177-L208.
5. `createaccountmodel.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/model/createaccountmodel.js/.
    a. `determinePasswordWarning`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/model/createaccountmodel.js#L71-L91.
6. `containers.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/containers.js/.
    a. `PasswordWarningState`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/containers.js#L1-L36.

7. `validation.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/validation.js/.
8. `createaccountpresenter.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/presenter/createaccountpresenter.js/.
9. `createaccountview.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/view/createaccountview.js/.
    a. `updatePasswordWarning`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/view/createaccountview.js#L282-L295.
10. `localization.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/localization.js/.
    a. `load`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/localization.js#L37-L64.
    b. `setLocale`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/localization.js#L66-L97.
    c. `get`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/mvp/utility/localization.js#L66-L97.
11. `localizations.yaml`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/docs/localizations/localizations.yaml/.
12. `settings.js`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/utility/settings.js/.
    a. `setInternal`: https://github.com/6a/blade-2-online-client-electron/blob/dfs/assets/js/utility/settings.js#L108-L112.

Blade II Online Game Client

1. `BladeIIGameMode.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2GameMode/BladeIIGameMode.h/.
    a. Engine and game component pointers: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2GameMode/BladeIIGameMode.h#L90-102.

2. `BladeIIGameMode.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2GameMode/BladeIIGameGameMode.cpp/.
    a. `GetUIAvatarWidgetClass`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2GameMode/BladeIIGameGameMode.cpp#L334-341.
    b. `SetupUIAvatarLayer`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2GameMode/BladeIIGameGameMode.cpp#L499-509.
    c. `RegisterEventListeners`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2GameMode/BladeIIGameGameMode.cpp#L406-434.
    d. `w`
3. `DealerEventEnum.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Enum/DealerEventEnum.h/.
4. `Settings.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/Settings.h/.
5. `Settings.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/Settings.cpp/.
    a. `ApplyAll`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/Settings.cpp/#L136-202.
6. LaunchConfig.h: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/LaunchConfig.h/.
7. LaunchConfig.cpp: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/LaunchConfig.cpp/.
    a. `LaunchConfig`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/LaunchConfig.cpp/#L29-53.

8. GameSound.h: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Game/GameSound.h/.
9. GameSound.cpp: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Game/GameSound.cpp/.
10. `GSM.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/GameStateMachine/GSM.h/.
    a. `ChangeState`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/GameStateMachine/GSM.h#L11-22.
11. `GSM.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM.cpp/.
    a. `Tick`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM.cpp#L11-14.
12. `GSM_State.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/GameStateMachine/GSM_State.h/.
13. `GSM_State.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM_State.cpp/.
    a. `Tick`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM_State.cpp#L15-36.
    b. `UpdateCursorPosition`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM_State.cpp#L104-132.
14. `Match state machine states`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/tree/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/.
15. `GSM_State_DrawToEmptyField.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-

game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM_State_DrawToEmptyField.cpp/.

    a. `Tick`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM_State_DrawToEmptyField.cpp#L155-191.

16. `GSM_State_PlayerTurn.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/GameStateMachine/GSM_State_PlayerTurn.h/.
17. `GSM_State_PlayerTurn.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM_State_PlayerTurn.cpp/.
    a. `Tick`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/GameStateMachine/GSM_State_PlayerTurn.cpp#L48-141.
18. `LocalPlayerInput.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dissertation-final-state/Source/BladeIIGame/Public/B2Engine/LocalPlayerInput.h/.
19. `LocalPlayerInput.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dissertation-final-state/Source/BladeIIGame/Public/B2Engine/LocalPlayerInput.h/.
20. `Dealer.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/Dealer.h/.
21. `Dealer.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/Dealer.cpp/.
    a. `Mirror`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/Dealer.cpp#L812-879.
22. `CardAnimator.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/CardAnimator.h/.
23. `CardAnimator.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/CardAnimator.cpp/.

    a. `Tick`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/CardAnimator.cpp#L32-69.

24. `Opponent.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/Opponent.h/.
25. `Opponent.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/Opponent.cpp/.
26. `NetOpponent::Configure`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/NetOpponent.cpp#L8-16.
27. `AIOpponent.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/AIOpponent.cpp#L8-13.
28. `Server.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/Server.h/.
29. `Server.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/Server.cpp/.
30. `AIServer.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/AIServer.h/.
31. `AIServer.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/AIServer.cpp/.
    a. `GetNextUpdate`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/AIServer.cpp#L13-35.
    b. `ResolveAITurn`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/AIServer.cpp#L591-622.
    c. `Tick`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/AIServer.cpp#L37-60.
    d. `UpdateState`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/AIServer.cpp#L230-293.
    e. `HandleTie`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/AIServer.cpp#L144-228.
    f. `ExecuteTurn`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/AIServer.cpp#L104-126.

32. `NetServer.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/NetServer.h/.
33. `NetServer.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/NetServer.cpp/.
    a. `Tick`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/NetServer.cpp#L106-151.
    b. `HandleMessageReceivedEvent`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/NetServer.cpp#L309-419.
    c. `HandleConnectionErrorEvent`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Engine/NetServer.cpp#L281-307.
34. `ServerUpdate.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/ServerUpdate.h/.
    a. `GetSerialised`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/ServerUpdate.h#L22-31.
35. `WebSocketPacket.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/WebSocketPacket.h/.
    a. `FromJSONString`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Engine/WebSocketPacket.h#L37-47.
36. `AvatarCaptureRig.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2Game/AvatarCaptureRig.h/.
37. `AvatarCaptureRig.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2Game/AvatarCaptureRig.cpp/.
38. `Avatar.h`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Public/B2UI/Avatar.h/.

39. `Avatar.cpp`: https://gitlab.kingston.ac.uk/K1611060/blade-ii-online-game/blob/dfs/Source/BladeIIGame/Private/B2UI/Avatar.cpp/.

REST API

1. `create_account.go (main package)`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/endpoints/create_account/create_account.go/.
    a. `main`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/endpoints/create_account/create_account.go#L24-L31.
    b. `functionWrapper`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/endpoints/create_account/create_account.go#L19-L22.
2. `create_account.go (routes package)`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/routes/create_account.go/.
    a. `CreateAccount`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/routes/create_account.go#L20-L100.
3. `helpers.go (routes package)`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/routes/helpers.go/.
    a. `validatePasswordFormat`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/routes/helpers.go#L125-L170.
4. `database.go (database package)`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/database/database.go/.
    a. `CreateAccount`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/database/database.go#L133-L211.
5. `payloads.go (types package)`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/types/payloads.go/.
    a. `CreateAccountResponsePayload`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/types/payloads.go#L20-L23.
6. `responses.go (types package)`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/types/reponses.go/.
    a. `HTTPResponse`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/types/reponses.go#L15-L19.

   b. `LambdaResponse`: https://github.com/6a/blade2-serverless-api/blob/dfs/internal/types/reponses.go#L21-L25.

Game Server

1. `routes package`: https://github.com/6a/blade-ii-game-server/tree/dfs/internal/routes/.
2. `handleconnection.go (transactions package)`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/transactions/handleconnection.go/.
   a. `HandleGSConnection`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/transactions/handleconnection.go#L25-L111.
3. `checkAuth.go (transactions package)`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/transactions/checkauth.go/.
   a. `checkauth`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/transactions/checkauth.go#L28-L60.
4. `validatematch.go (transactions package)`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/transactions/validatematch.go/.
   a. `validateMatch`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/transactions/validatematch.go#L20-L48.
5. `server.go (game package)`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go/.
   a. `Server`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go#L25-L43.
   b. `AddClient`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go#L74-L81.
   c. `MainLoop`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go#L97-L261.
   d. `broadcast channel`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go#L225-L232.
   e. `command channel`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go#L233-L238.

   f. `connect channel`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go#L112-L224.
   g. `disconnect channel`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go#L242-L243.
   h. `handleDisconnectRequests`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/server.go#L263-L435.
6. `match.go (game package)`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/match.go/.
   a. `Match`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/match.go#L54-L76.
   b. `SetMatchStart`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/match.go#L309-L344.
   c. `SetMatchResult`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/match.go#L346-L384.
   d. `Tick`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/match.go#L78-L119.
7. `disconnectrequest.go (game package)`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/disconnectrequest.go/.
   a. `DisconnectRequest`: https://github.com/6a/blade-ii-game-server/blob/dfs/internal/game/disconnectrequest.go#L12-L24.
8. `database package`: https://github.com/6a/blade-ii-game-server/tree/dfs/internal/database/.

Database

Database Schemas

```
CREATE TABLE `matches` (
  `id` bigint(8) unsigned NOT NULL AUTO_INCREMENT,
  `player1` bigint(8) unsigned NOT NULL,
  `player2` bigint(8) unsigned NOT NULL,
  `winner` bigint(8) unsigned NOT NULL,
  `register` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `start` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  `end` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  `phase` tinyint(1) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`,`player1`,`player2`),
  UNIQUE KEY `id_UNIQUE` (`id`),
  KEY `id_idx` (`player1`,`player2`,`end`)
) ENGINE=InnoDB AUTO_INCREMENT=100 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

Figure 33. Schema for the matches table in the Blade II Online database.

```
CREATE TABLE `profiles` (
  `id` bigint(8) unsigned NOT NULL,
  `public_id` varchar(20) NOT NULL,
  `avatar` smallint(8) unsigned NOT NULL DEFAULT '0',
  `mmr` int(4) NOT NULL DEFAULT '1200',
  `wins` int(5) unsigned NOT NULL DEFAULT '0',
  `draws` int(5) unsigned NOT NULL DEFAULT '0',
  `losses` int(5) unsigned NOT NULL DEFAULT '0',
  `winratio` float GENERATED ALWAYS AS ((`wins` / ((`wins` + `losses`) +
`draws`))) VIRTUAL,
  `ranked_total` bigint(6) GENERATED ALWAYS AS (((`wins` + `losses`) + `draws`))
VIRTUAL,
  `created` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `modified` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`,`public_id`),
  UNIQUE KEY `id_UNIQUE` (`id`),
  UNIQUE KEY `public_id_UNIQUE` (`public_id`),
  CONSTRAINT `id_profiles` FOREIGN KEY (`id`) REFERENCES `users` (`id`) ON DELETE
CASCADE ON UPDATE CASCADE,
  CONSTRAINT `public_id_profiles` FOREIGN KEY (`public_id`) REFERENCES `users`
(`public_id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Figure 34. Schema for the profiles table in the Blade II Online database.

```
CREATE TABLE `tokens` (
  `id` bigint(8) unsigned NOT NULL,
  `email_confirmation` varchar(32) NOT NULL DEFAULT '""',
  `email_confirmation_expiry` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `password_reset` varchar(32) NOT NULL DEFAULT '""',
  `password_reset_expiry` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `auth` varchar(32) NOT NULL DEFAULT '""',
  `auth_expiry` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `refresh` varchar(32) NOT NULL DEFAULT '""',
  `refresh_expiry` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `match` bigint(8) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `id_UNIQUE` (`id`),
  CONSTRAINT `id_tokens` FOREIGN KEY (`id`) REFERENCES `users` (`id`) ON DELETE
CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Figure 35. Schema for the tokens table in the Blade II Online database.

```
CREATE TABLE `users` (
  `id` bigint(8) unsigned NOT NULL AUTO_INCREMENT,
  `public_id` varchar(20) NOT NULL,
  `handle` varchar(60) NOT NULL,
  `email` varchar(320) NOT NULL,
  `salted_hash` varchar(100) NOT NULL,
  `banned` tinyint(1) unsigned NOT NULL DEFAULT '0',
  `email_confirmed` tinyint(1) unsigned NOT NULL DEFAULT '0',
  `modified` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  `created` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `privilege` tinyint(1) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`,`public_id`,`handle`),
  UNIQUE KEY `id_UNIQUE` (`id`),
  UNIQUE KEY `public_id_UNIQUE` (`public_id`),
  UNIQUE KEY `handle_UNIQUE` (`handle`),
  UNIQUE KEY `email_UNIQUE` (`email`)
) ENGINE=InnoDB AUTO_INCREMENT=100 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

Figure 36. Schema for the users table in the Blade II Online database.

Prepared Statement: Match History Query

`SELECT`	Return a row with the following columns:
`m`.`id`,	id
`p1`.`handle` as `player1handle`,	player1handle
`p1`.`public_id` as `player1pid`,	player1pid
`p2`.`handle` as `player2handle`,	player2handle
`p2`.`public_id` as `player2pid`,	player2pid
`w`.`handle` as `winnerhandle`,	winnerhandle
`w`.`public_id` as `winnerpid`,	winnerpid
`m`.`end`	end
`FROM `Blade_II`.`matches` `m``	From the matches table, where values that with matching keywords in the users table are used to

	overwrite values in the matches table
WHERE ? IN(`player1`, `player2`) AND `phase` = 2	For the player with the id ?, and where the matches phase is 2
ORDER BY `end` DESC, `id` DESC;	Ordered by end time and id in descending order.

Figure 37. The prepared statement used to get the match history for a player.

References

Allen, G. (2014) *Card Dimensions for CCGs and LCGs magic, Pokemon, Future Card Buddyfight, Kaijudo, Cardfight Vangaurd, Yugioh, Game of Thrones, Android Netrunner, Call of Cthulu, Lord of the Rings, Star Wars*. [Online discussion group] 11 March. Available at: https://boardgamegeek.com/thread/1137261/card-dimensions-ccgs-and-lcgs-magic-pokemon-future/ (Accessed: 09 May 2020).

Anguelov, B. *et al*. (2014) *Game AI Pro*. Edited by S. Rabin *et al*. Boca Raton, FL: CRC Press.

AWS (2019) *Amazon Computer Service Level Agreement*. Available at: https://aws.amazon.com/compute/sla/ (Accessed: 09 May 2020).

Bizreach (2018) *Puroguramingu Gengobetsu Nenshuu Chuuōchi Wo Happyō, Kyuujin Kensaku Enjin「Sutanbai」Shirabe*. Available at: https://www.bizreach.co.jp/pressroom/pressrelease/2018/0807.html (Accessed: 22 October 2019).

Blevins, T. (2000) *Magic: The Gathering Review*. Available at: https://www.gamespot.com/reviews/magic-the-gathering-review/1900-2542422/ (Accessed: 27 September 2019).

Knight, C. (2015) *3 Differences Between Scrum and Kanban You Need to Know*. Available at: https://www.cprime.com/2015/02/3-differences-between-scrum-and-kanban-you-need-to-know/ (Accessed: 09 May 2020).

Blizzard Entertainment (2019) *Lightning Storm*. 16 May. Available at: https://d2q63o9r0h0ohi.cloudfront.net/videos/npe/videos/LightningStorm-final-be597a7d03990bcf124753dcb4f60ba29c232ee531687c34d960aff589e40b8361e2026e742ac3562cc956f3fdf11c6673da087660beaee9ddc1ad9cc1f53147.mp4 (Accessed: 02 October 2019).

Bregger, P. (2019) *Chron X (1997) Windows release date*. Available at: https://www.mobygames.com/game/windows/chron-x/release-info/ (Accessed: 27 September 2019).

Brode, B. [@bbrode] (2013) @Ziethriel mixed - projected textures on 3d models [Twitter] 14 December. Available at: https://twitter.com/bbrode/status/411927244827684866/ (Accessed: 02 October 2019).

Brode, B., Chayes J. (2013) Interviewed by Miaari for *Blizz Planet*, 24 March. Available at: https://www.youtube.com/watch?v=FRVSsVCC5v8 (Accessed: 02 October 2019).

Caldwell, J. (2019) *Microsoft Solitaire enters the World Video Game Hall of Fame*. Available at: https://www.onmsft.com/news/microsoft-solitaire-enters-the-world-video-game-hall-of-fame/ (Accessed: 27 September 2019).

Chircu, V. (2018) *Understanding the 8 fallacies of Distributed Systems*. Available at: https://www.simpleorientedarchitecture.com/8-fallacies-of-distributed-systems/ (Accessed: 11 May 2020).

Clegg, D. and Barker, R. 1994. *CASE Method Fast-Track: A Rad Approach*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc..

Cloudflare (2019) *What Is SSL? | SSL and TLS*. Available at: https://www.cloudflare.com/learning/ssl/what-is-ssl/ (Accessed: 09 May 2020).

Collingwood, J. (2018) *Hofstadter's Law and Realistic Planning*. Available at: https://psychcentral.com/lib/hofstadters-law-and-realistic-planning/ (Accessed: 11 May 2020).

Cowling, P. et al. (2012) 'Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering', *IEEE Transactions on Computational Intelligence and AI in Games*, 4(4), p. 243. doi:10.1109/TCIAIG.2012.2204883.

CPT. Trips (2017) *UE4 Performance - BP vs C++ vs Blueprint Nativization | feat. Multithreaded Tick*. Available at: https://www.youtube.com/watch?v=8gVixDglpQ4 (Accessed: 09 May 2020).

Department for Digital, Culture, Media & Sport and Home Office (2018) *Data Protection Act 2018*. Available at: https://www.gov.uk/government/collections/data-protection-act-2018 (Accessed: 25 October 2019).

Duncan, I. (2019) *Build a RESTful JSON API with Golang*. Available at: https://medium.com/the-andela-way/build-a-restful-json-api-with-golang-85a83420c9da/ (Accessed 05 October 2019).

Ellingwood, J. (2017) *How to create a Self-Signed SSL Certificate for Nginx on CentOS 7*. Available at: https://www.digitalocean.com/community/tutorials/how-to-create-a-self-signed-ssl-certificate-for-nginx-on-centos-7/ (Accessed: 23 October 2019).

Falcom (2010) *Falcomu Ongaku Furii Sengen - Gakkyoku Riyō Kiyaku*. Available at: https://www.falcom.co.jp/music-use/rules/ (Accessed: 25 October 2019).

Famitsu (2010) *Epic Games No Nihon Jōriku Ga Kettei, Unreal Engine No Nihongo Sapōto Wo Kyōka*. Available at: https://automaton-media.com/articles/interviewsjp/20180524-68476/ (Accessed: 22 October 2019).

Glazer, J. and Madhav, S. (2016) *Multiplayer Game Programming*. Crawfordsville, IN: Pearson. The Addison-Wesley Game Design and Development Series.

go-chi/chi: lightweight, idiomatic and composable router for building Go HTTP services (2019) Available at: https://github.com/go-chi/chi/tree/master/ (Accessed: 22 October 2019).

Google (no date) *Package json*. Available at: https://golang.org/pkg/encoding/json/#Unmarshal (Accessed 23 October 2019).

Grime (2019) *The Elo Rating System for Chess and Beyond*. 15 February. Available at: https://www.youtube.com/watch?v=AsYfbmp0To0 (Accessed: 22 October 2019).

Håkonsen, E. (2015) *System-and AI design of a digital collectible card game*. PhD thesis. University of Copenhagen. Available at: https://www.pdf-archive.com/2017/12/20/hearthstone-ai-thesis-ejnar-h-konsen/hearthstone-ai-thesis-ejnar-h-konsen.pdf (Accessed: 24 October 2019).

Hearthstone (2017) *Behind the Card | Amara: Warden of Hope*. 13 April. Available at: https://www.youtube.com/watch?v=l0qqFROfHWE (Accessed: 02 October 2019).

Hill, E. (2017) *What is 'Being in the Zone'? – the Fascinating Psychology of Super Productivity*. Available at: https://www.huffpost.com/entry/what-it-really-means-to-b_b_10300610/ (Accessed: 11 May 2020).

Hiwarale, U. (2018) *Achieving concurrency in Go*. Available at: https://medium.com/rungo/achieving-concurrency-in-go-3f84cbf870ca/ (Accessed: 22 October 2019).

Hofstadter, D. R. (1980) *Gödel, Escher, Bach : an eternal golden braid*, Harmondsworth: Penguin.

IGN (2014) *Card Rarity - Hearthstone: Heroes of WarCraft Wiki Guide*. Available at: https://uk.ign.com/wikis/hearthstone-heroes-of-warcraft/Card_Rarity (Accessed: 23 October 2019).

increpare (no date) *Bfxr. Make sound effects for your games.* Available at: https://www.bfxr.net/ (Accessed: 25 October 2019).

Intellectual Property Office (2019) *Exceptions to copyright*. Available at: https://www.gov.uk/guidance/exceptions-to-copyright#non-commercial-research-and-private-study (Accessed 25 October 2019).

Jebens, H. (1998) *Sanctum Open for Business*. Available at: https://www.gamespot.com/articles/sanctum-open-for-business/1100-2464321/ (Accessed: 27 September 2019).

Johnsson, H. (2019) *REST-API with Golang, Mux and MySQL*. Available at: https://medium.com/@hugo.bjarred/rest-api-with-golang-mux-mysql-c5915347fa5b (Accessed: 23 October 2019).

Kawasaki, T. (2013) 'Interview with Taka KAWASAKI'. Interview with Taka Kawasaki. Interviewed by J. Szczepaniak for *The Untold History of Japanese Game Developers Volume 2*, 31 October, p. 391.

Kawasaki, T. (2017) Interviewed by Keigo Toyoda for *Famitsu*, 6 September. Available at: https://www.famitsu.com/news/201709/06140985.html (Accessed: 23 October 2019).

Khan, I. (2016) *OMG! Wow! Rocket League quick chat options expanding*. Available at: https://www.digitaltrends.com/gaming/rocket-league-quick-chat-update/ (Accessed 25 October 2019).

Kollar, P. (2015) *Play Hearthstone on your iPhone or Android phone starting today*. Available at: https://www.polygon.com/2015/4/14/8407107/hearthstone-ios-android-iphone-release/ (Accessed: 27 September 2019).

Kondo, T. (2019) Interviewed by A. Donaldson for *VG247*, 22 October. Available at: https://www.vg247.com/2019/10/22/fansite-owner-ended-president-falcom-one-japans-prestigious-rpg-houses/ (Accessed: 08 May 2020).

Krogh-Jacobsen, T. (2018) *2018.2 is now available*. Available at: https://blogs.unity3d.com/2018/07/10/2018-2-is-now-available/ (Accessed 23 October 2019).

Kukhnavets, P. (2018) *Why Agile is So Popular in Project Management World*. Available at: https://hygger.io/blog/why-agile-is-so-popular-in-project-management/ (Accessed: 11 May 2020).

Lima Beans (2009) *Re: Curious: What Programming Language?*. [Online discussion group] 7 October. Available at: http://forums.na.leagueoflegends.com/board/showthread.php?t=16318 (Accessed: 04 October 2019).

Lol Smurfs (2018) *Is League of Legends Finally Dying in 2019?*. Available at: https://www.lol-smurfs.com/blog/is-league-of-legends-dying/ (Accessed: 08 May 2020).

Lunden, I. (2015) *Amazon's AWS Is Now A $7.3B Business As It Passes 1M Active Enterprise Customers*. Available at: https://techcrunch.com/2015/10/07/amazons-aws-is-now-a-7-3b-business-as-it-passes-1m-active-enterprise-customers/ (Accessed: 09 May 2020).

Merril, M. (2009) Interviewed by Suzie Ford for *WarCry Network*, 24 January. Available at: https://web.archive.org/web/20131213123923/http://www.warcry.com/articles/view/interviews/5686-League-of-Legends-Marc-Merrill-Q-A/ (Accessed: 02 October 2019).

Mitchell, I. (2014) *Example of a Kanban Board separating internally and externally blocked work*. Available at: https://upload.wikimedia.org/wikipedia/commons/thumb/7/7b/Kanbanboardidentifyinginternalexternalblockages.jpg/536px-Kanbanboardidentifyinginternalexternalblockages.jpg (Accessed: 25 October 2019).

Muslihat, D. (2018) *Agile Methodology: An Overview*. Available at: https://zenkit.com/en/blog/agile-methodology-an-overview/ (Accessed: 09 May 2020).

Nintendo (1998) *Pokémon Kaado GB*. Available at: https://www.nintendo.co.jp/n02/dmg/acxj/index.html (Accessed: 27 September 2019).

Paah (2010) *HoN/LoL release dates?*. [Online discussion group] 12 July. Available at: http://forums.euw.leagueoflegends.com/board/showthread.php?t=78609 (Accessed: 2 October 2019).

Padmanabhan, V. (2018) 'Functional Strategy Implementation - Experimental Study on Agile KANBAN', *Sumedha Journal of Management*, 7(2), pp. 6-17. Available at: https://search-proquest-com.ezproxy.kingston.ac.uk/docview/2149601807?accountid=14557 (Accessed 25 October 2019).

PEGI (no date) *What do the labels mean?*. Available at: https://pegi.info/what-do-the-labels-mean (Accessed: 24 October 2019).

Rawdat, A. (2017) *Rate Limiting with NGINX and NGINX Plus*. Available at: https://www.nginx.com/blog/rate-limiting-nginx/ (Accessed: 23 October 2019).

Sookocheff, K. (2019) *How Do Websockets Work?*. Available at: https://sookocheff.com/post/networking/how-do-websockets-work/ (Accessed: 23 October 2019).

Stackoverflow (2019) *Developer Survey Results*. Available at: https://insights.stackoverflow.com/survey/2019#technology-_-programming-scripting-and-markup-languages (Accessed: 09 May 2020).

Suzuki, G. (2014) *Unreal Engine Pōtaru Saito Ga Kōshiki Nihongoka, Unreal Editor Honyaku Mo Shinkōchuu.* Available at: https://www.gamespark.jp/article/2014/05/01/48231.html (Accessed: 23 October 2019).

Sweeney, T. (2018) Interviewed by Minoru Umise for *Automaton Media*, 24 May. Available at: https://automaton-media.com/articles/interviewsjp/20180524-68476/ (Accessed: 22 October 2019).

Warcholinski, M. (2020) *What Can You Build with Electron Development?*. Available at: https://brainhub.eu/blog/electron-development/ (Accessed: 09 May 2020).

Warren, M. (2019) *Is C# a low-level language?*. Available at: https://mattwarren.org/2019/03/01/Is-CSharp-a-low-level-language/ (Accessed 22 October 2019).

Wilson, J. (2014) *Even Hearthstone runs on Unity — and that's why it's already on iPad*. Available at: https://venturebeat.com/2014/04/24/even-hearthstone-runs-on-unity-and-thats-why-its-already-on-ipad/ (Accessed: 27 September 2019).

Wu, L. (2018) *Setting-up mitmproxy on macOS to intercept https requests*. Available at: https://medium.com/what-i-talk-about-when-i-talk-about-ios-developmen/setting-up-mitmproxy-on-macos-to-intercept-https-requests-f3cba29ff003 (Accessed: 23 October 2019).

Yin-Poole, W. (2014) *Hearthstone leaves beta and launches proper on PC and Mac*. Available at: https://www.eurogamer.net/articles/2014-03-12-hearthstone-leaves-beta-and-launches-proper-on-pc-and-mac (Accessed: 3 September 2019).

Zeriyah (2014) *Welcome to the Hearthstone Launch!*. Available at: https://playhearthstone.com/en-us/blog/13154923/ (Accessed: 3 October 2019).

Zwicker, J. [@Zwickarr] (2013) @Ziethriel @bdbrode I take a painting made by Ben Thompson and project it on 3d models. Models are a bit distorted :) [Twitter] 14 December. Available at: https://twitter.com/Zwickarr/status/411939154990071808/ (Accessed: 3 October 2019).