Conducting Real-time Video Analysis on a 30-year-old Retro Game

Oustan Ding
2B Software Engineering
ID: 20840114

Relevant courses:
CS 137 Programming Principles
SE 212 Logic and Computation
ECE 417 Image Processing

Table of Contents

In this blog post, I discuss the engineering decisions that I made while building [nestris.online](), an online platform for Classic Tetris players to compete against each other. I worked on this project during the summer after my freshman year, so some of the code could be better written, but the ideas remain the same.

Background

Of the many iterations of Tetris that have been released over the past few decades, one of the most popular is the 1989 version, widely known as **Classic Tetris**, created for the Nintendo Entertainment System (NES) (Figure 1). Over the past few years, the game has surged in popularity due to its extremely addicting gameplay, its seemingly infinite creative possibilities, and internet memes. If you aren't familiar with the game, I encourage you to check out the epic gameplay at https://www.youtube.com/watch?v=5sxMqLjTv6k (Classic Tetris, 2020).

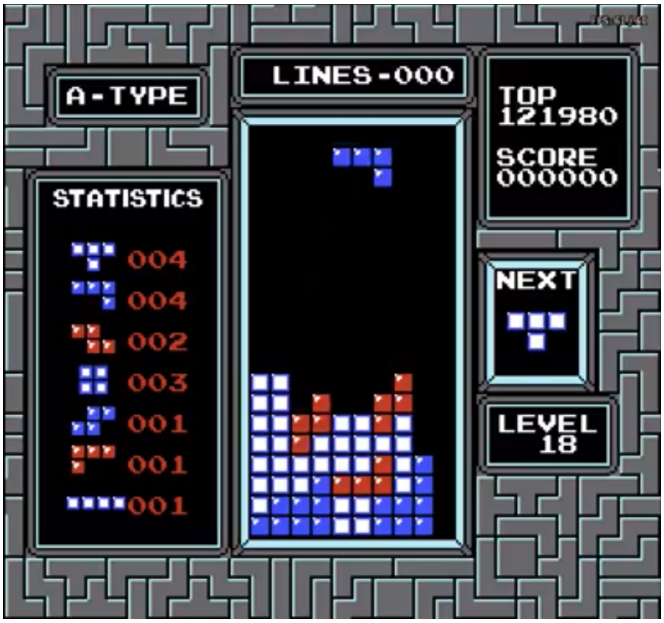

Figure 1. Classic Tetris game screen

When the world went online in 2020, the demand for online one-on-one gameplay peaked. Thus, in June 2020, I had the idea of building a platform to address this demand. Classic Tetris did not have a built-in multiplayer mode, so at the time, the most popular way to

compete online was to have a middleman aggregate the live videos of the players' game screens and referee them manually. This was tedious, so I explored ways to allow players to coordinate the games themselves and automate the refereeing process through building nestris.online.

In a nutshell, nestris.online helps players play Classic Tetris against each other by streaming their game videos to each other. Behind the scenes, the app would conduct real-time analysis on the players' game videos to automatically referee the game. Below, I reflect on two key aspects of the app: the **player-to-player video streaming** and the **identification of upcoming pieces** (a necessary piece of information for refereeing the game). I will focus on the latter as it was the more interesting topic of the two.

Challenges and Constraints

Retro games are difficult to work with because of how old and low-level they are. Today, many games are open-source and provide developer toolkits for anyone to contribute new features. However, retro games that were built for the NES such as Classic Tetris were written with the 6502 Assembly language, which is difficult to work with these days (Zurawel, n.d.). Furthermore, the NES is an 8-bit console with a chip much weaker than today's computers', so it wouldn't be easy to add complex functionalities. This limits the opportunities to modify the game itself.

Retro games also tend to involve "frame-perfect" gameplay, meaning that nearly every video frame rendered by the game is important (North, 2020). For example, on levels 19-28 in Classic Tetris, it can take as little as 0.8 seconds for a piece to fall and lock in place. At levels 29+, it takes half that time. This means that any game state analysis needs to analyze the game video at a rate greater than 1 Hz so that the game state and the analysis are synchronized as much as possible. The higher the rate, the better. Overall, for the piece identification, the following constraints had to be met:

- Identification of upcoming piece with a very high degree of accuracy (at most 1 incorrect identification per 100 pieces)

- Detection of upcoming piece multiple times per second (run detection algorithm at a rate greater than 1 Hz)
- Minimal per-game financial cost when identifying the upcoming pieces with the above constraints (less than 1 USD per game)

On the other hand, the game video streaming did not have as many requirements. When competing in Classic Tetris, the most important pieces of information to note from an opponent's screen are the opponent's score and level, and whether the stack is close to the top of the board. Since the above information is easily readable, I aimed to render the current player's and the opponent's game videos at **a frame rate greater than 30 Hz** to provide a pleasant gaming experience.

Real-time Game Video Streaming

As mentioned earlier, it was not feasible to modify the game itself to communicate with other players. Thus, the video had to be captured and sent between players. To be able to analyze the players' games, I first needed to capture their device screens and exchange either player's video with the other player.

Until I worked on this project, I had only worked with HTTP requests. In HTTP, a client (a player's browser) would make a request to a server which handles the request and sends back a response. Every HTTP request includes both data sent by the client and headers storing additional information about the request (Gamage, 2017). With the extra overhead, HTTP requests have high latency and are not suited for the rapid bi-directional data transfer needed for a game being played in real time between two players (Luecke, 2018).

The frontrunner solution for low-latency data exchange between clients is to use WebSockets (Ubl & Kitamura, 2010). Socket connections allow a client (a player's browser) to open and maintain a persistent connection with the server through the initial connection, which is called a **handshake** (Ubl & Kitamura, 2010). Then, both the client and the server can **emit** and **listen** for events, allowing them to execute logic without having to

establish a new connection. The data overhead in HTTP headers does not apply to sockets, and the latency is much lower, hence its popular use in games, messaging, and other applications. This was critical to allowing the players to watch each other's game screens in real-time. I used Socket.IO to manage the socket connections on the server and PeerJS to manage the video connections between players. Using these technologies, I was able to stream the players' game videos at a frame rate greater than 30 Hz, satisfying the constraint.

Analyzing the Upcoming Piece

In Classic Tetris, one of the most important pieces of information that can be retrieved from the game screen is the upcoming piece that will be dropped after the piece that is currently falling (Figure 2). With this information, one can analyze game statistics and player performance, including the player's tetris rate (percentage of line clears used for tetrises) and whether the player is "dead" (when the pieces pile up and reach the top of the board).

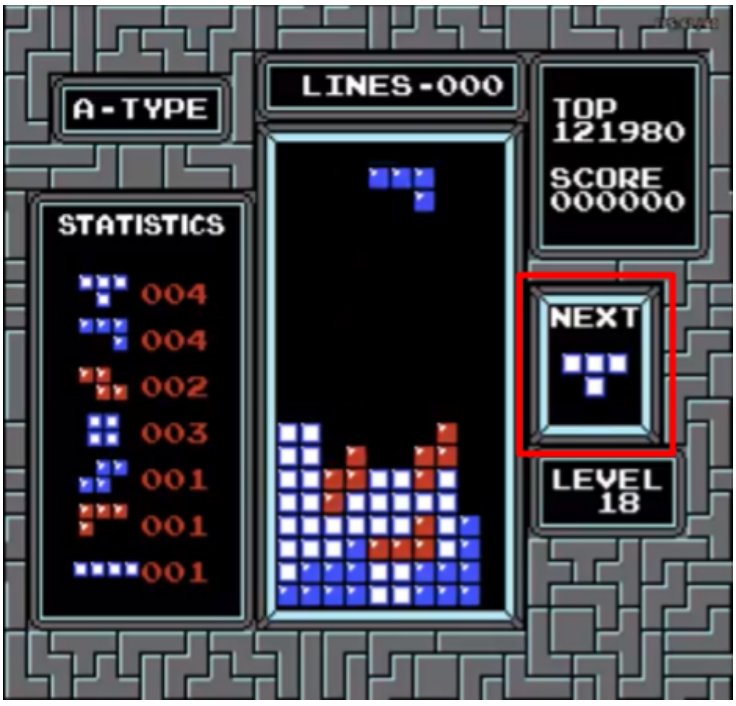

Figure 2. The "Next" box that displays the upcoming piece in Classic Tetris

This was the most crucial step to completing the automatic refereeing system and also the biggest challenge of the project. During initial scoping, I researched potential computer vision (CV) libraries capable of identifying the upcoming piece from the game video. Google Vision API was one of the most obvious choices, with its high degree of efficacy, but it had two major issues. First, it was too slow. Analyzing an image would require uploading it first to Google and then waiting for an API response before the results can be used. And this would need to be done multiple times per second. Second, considering the number of requests to the API, pricing is an issue. A Classic Tetris game between two players that lasts 10 minutes would require about 4,800 requests and cost about 7.20 USD if the game state was analyzed 4 times per second (Google, 2021). That would be too expensive, given the constraints.

Considering the speed constraint, it's much faster to take advantage of a player's browser's computing resources to conduct the image recognition locally, and then only send the processed data to the server via a socket connection. This would massively reduce the amount of data exchanged between clients and the server, and would eliminate the issue of high latency when communicating with external APIs. It would also reduce the computing resources needed on the server to process all the data, thereby lowering deployment costs.

At this point, I contacted my friend, Yash, for some advice (he specializes in machine learning and computer vision). After some insightful discussion and brainstorming, one suggestion that he made clicked: avoiding the use of computer vision libraries entirely. I ultimately utilized his suggestion in my solution.

Extracting the Next Piece from the Game Video

In Classic Tetris, there are 7 "tetrominoes", also known as "pieces" that drop from the top of the screen periodically (Figure 3).

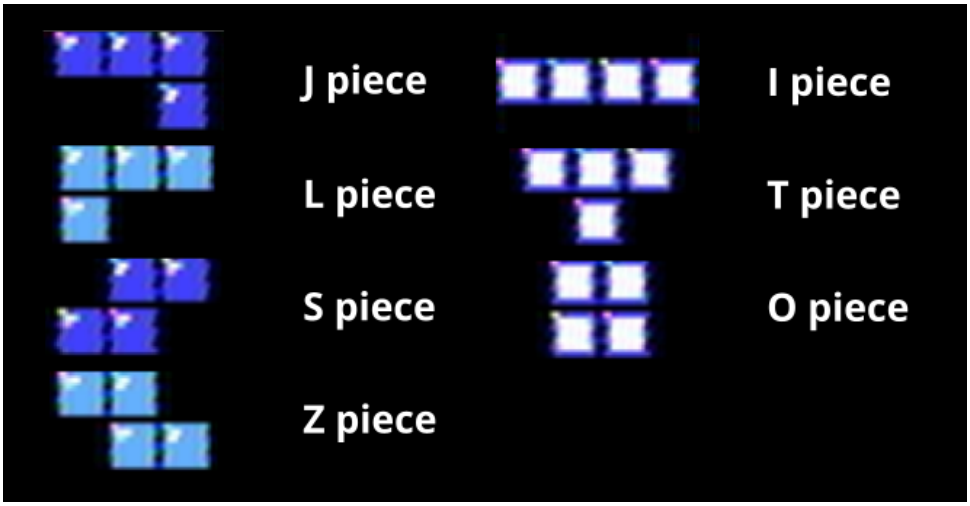

Figure 3. The 7 tetrominoes ("pieces") in Classic Tetris

As mentioned earlier, these pieces are displayed in the "Next" box before they drop from the top of the screen. The first step was to crop out the "Next" box from the game video. Using an online pixel ruler, I measured the pixel offset and dimensions of the "Next" box relative to the top-left corner of the game screen and stored them in a 4-dimensional vector with format `[<x-offset>, <y-offset>, <width>, <height>]`:

```
const NEXT_PIECE_DIMS = [1247, 539, 143, 70]
```

Using the Canvas API, I then applied these measurements to crop out the "Next" box from the game video (an HTML `<video>` element stored in `videoRef.current`) and "redraw" it on an HTML `<canvas>` element stored in `pieceRef.current` (Mozilla, 2021).

```
const canvas = pieceRef.current.getContext('2d')
canvas.drawImage(
    videoRef.current,
    ...NEXT_PIECE_DIMS,
    0, 0, NEXT_PIECE_DIMS[2], NEXT_PIECE_DIMS[3]
)
```

Now, I had a consistent image of each piece in the game as rendered in the "Next" box. This meant that all the images for I pieces, for instance, would look the same, with the same size and dimensions so that an identification algorithm could be applied consistently (Figure 4).

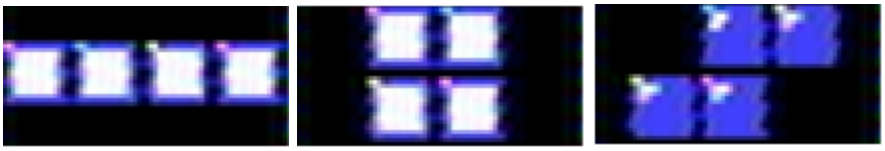

Figure 4. Cropped images of the I, O and S pieces from the "Next" box with equal dimensions

Design

The overarching idea of the piece detection algorithm in nestris.online involved strategically selecting a couple of pixels from the "Next" box, and checking whether the pixels were black. This data allowed the quick and accurate determination of the shape and thus the identity of the piece.

To determine which pixels to check, I first added the cropped pictures of the pieces shown in Figure 4 to a Google Drawing, layering those that had overlapping squares (where each piece is composed of 4 squares). To clarify, when the S, Z, T, L and J pieces are layered, their squares form a 2x3 grid without any partial overlaps (Figure 5).

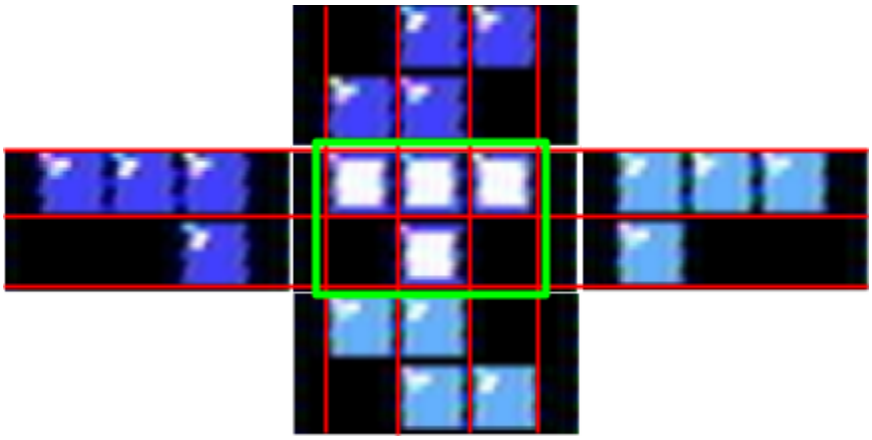

Figure 5. The squares that make up the S, Z, T, L and J pieces line up in a 2x3 grid (highlighted in green)

If we only consider these five pieces and the ten pixels marked by red dots in Figure 6 below, the combination of which pixels are non-black is sufficient for identifying the piece in the image.

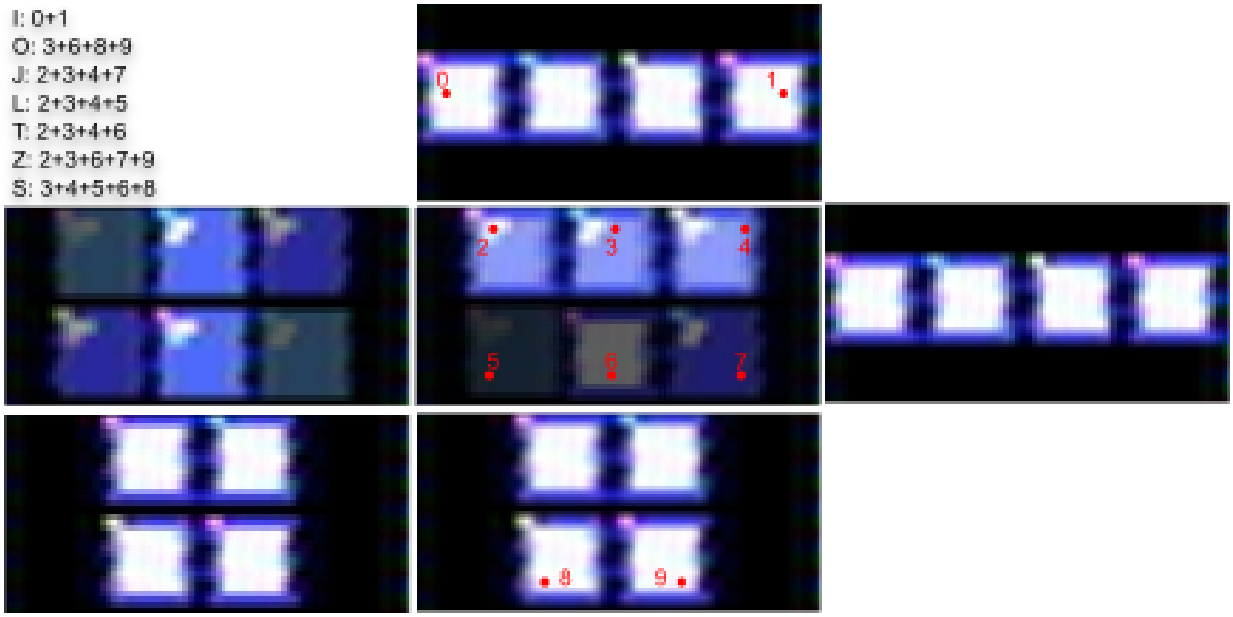

Figure 6. The 10 pixels (red dots) used to identify the "Next" piece. Top left corner: the combination of pixels that must be non-black to distinguish each piece.

For a simple example, let's pretend that the game only contains the S, Z, T, L and J pieces and only examine pixels 2-7 in our analysis. Consider the following pixel configuration:

Pixel index	2	3	4	5	6	7
Black?	Yes	No	No	No	No	Yes

Table 1. Example pixel configuration for an unknown piece

Since pixel 4 is not black, the piece cannot be a Z (which has non-black pixels 2,3,6,7). Subsequently, since pixel 6 is non-black, the piece cannot be an L or J. Thus, the piece can only be an S or a T, and the fact that pixel 5 is non-black narrows it down to an S.

The I piece is pretty straightforward - if pixels 0 and 1 in the image are non-black, then we can conclude that it is an I piece as no other piece is that wide. And for the O piece, we ensure that pixels 2 and 4 are black and pixels 8 and 9 are non-black. There are other possible configurations for detecting an O piece, but this one worked fine.

With this algorithm, it was possible to determine the next piece quickly based on the colours of the 10 pixels and some conditional logic, which was much faster and less resource-intensive than using computer vision libraries that utilize machine learning.

Implementation

Using the blueprint in Figure 6 and an online pixel ruler, I pinpointed the pixel coordinates of the 10 pixels relative to the top-left corner of the cropped pieces and stored them as 2-dimensional `[<x-offset>, <y-offset>]` vectors in a JavaScript object mapping the pixel indices to their coordinates. Note that no height or width values were needed because only singular pixels (1x1) were analyzed.

```
const PIECE_DATA = {
    0: [11, 33],
    1: [119, 33],
    2: [28, 9],
    3: [71, 9],
    4: [117, 9],
    5: [28, 52],
    6: [71, 52],
    7: [117, 52],
    8: [40, 61],
    9: [100, 61]
}
```

For each pixel analyzed, I used the Canvas API to extract the RGB values of the pixel's colour. The following function `hasColour` returns `true` if the pixel is non-black and `false` otherwise.

```
const hasColour = (r, g, b) => {
    return (
        r !== 0
        || g !== 0
        || b !== 0
    )
}
```

Finally, I used conditional logic to determine the next piece from the pixel configuration. The input was a 10-dimensional boolean vector, with `p0` being `true` if pixel 0 was non-black, `p1` being `true` if pixel 1 was non-black, and so on. The output was the identified piece.

```
const analyzePixels = ([p0, p1, p2, p3, p4, p5, p6, p7, p8, p9]) => {
    if (p0 && p1) { // Exclusive to I
        return 'I'
    } else if (p2 && p3 && p4) { // Narrow down to J, L, T
        if (p5) {
            return 'L'
        } else if (p6) {
            return 'T'
        } else if (p7) {
            return 'J'
        }
    } else if (p8 && p9) {
        return 'O'
    } else if (p2 && p3) {
        return 'Z'
    } else if (p3 && p4) {
        return 'S'
    }
    return 'Invalid piece'
}
```

You've probably noticed that some pixels don't actually need to be analyzed. For example, pixel 3 is non-black in the S, Z, T, L and J pieces, and it is not a factor in distinguishing between I and O. However, upon testing the function without considering pixel 3, the algorithm's accuracy decreased, particularly when it would detect Z and S pieces as O pieces. This was likely due to small discrepancies in pixel colours - some black pixels may have been rendered as off-black. It took some tweaking and testing to arrive at the conditional logic in the function above.

The last step was to run the image cropping and analysis functions on the game video at a rate of 4 times per second. The following is a condensed version of the code for better readability:

```
const getNextPiece = () => {
```

```
    // Crop next piece from game video
    const canvas = pieceRef.current.getContext('2d')
    canvas.drawImage(
        videoRef.current,
        ...NEXT_PIECE_DIMS,
        0, 0, NEXT_PIECE_DIMS[2], NEXT_PIECE_DIMS[3]
    )

    // Analyze 10 pixels
    const pixelData = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].map(pixel => {
        // Get pixel data
        const info = canvas.getImageData(
            ...PIECE_DATA[pixel], 1, 1
        )
        return hasColour(...info.data) // info.data is of the form [r,g,b]
    })

    const nextPieceString = analyzePixels(pixelData)
    setNextPiece(nextPieceString)
}

// Run the analysis code 4 times per second (every 250ms)
setInterval(getNextPiece, 250)
```

With this algorithm, I was able to correctly identify the upcoming piece at 4 times per second with a 99% success rate and zero financial cost per game. The implementation satisfied the original constraints of this engineering problem: a cheap, yet accurate algorithm to detect the upcoming Tetris piece at a rate greater than 1 Hz.

Potential Improvements

While the piece identification algorithm satisfied the original requirements, there is room for improving the project to further increase its frequency or reduce its resource consumption. Two methods include using **bit arrays** and using **WebAssembly**.

Bit Arrays

In my current implementation, I use a 10-dimensional vector of booleans to represent the configuration of black and non-black pixels in the “Next” box. When executing the

conditional logic for piece detection, several nested `if` conditions are run. While these conditions are usually so low-level that their optimization doesn't make a noticeable difference, they could impact the piece detection algorithm, which needs to run at a high frequency. Moreover, the faster this algorithm can be made, the higher the frequency it can be run at, and the closer the analysis is synchronized with the game.

Since the 10-D vector is just a sequence of `true`/`false` values, each pixel's black/non-black state could be represented as a bit in an integer. Then, instead of using several AND operations and nested conditions to check the various conditions, a **mask** can be applied to check the state of multiple bits (pixels) simultaneously. This may boost the performance of the piece detection function, but it would require some extensive testing to verify.

WebAssembly

A couple of weeks ago, I watched a YouTube video about WebAssembly (WASM), a relatively new technology that serves as a compilation target for code on the web (Fireship, 2020). WASM features many performance benefits compared to plain JavaScript, including being more compact, optimized and compatible with machine code (Clark, 2017). These features can drastically boost an app's performance on the web, hence why it powers some very well-known web technologies such as Figma and Tesseract.js.

By compiling the piece detection algorithm to WASM, it is possible to further increase its speed. This also opens the door to increasing the algorithm's accuracy by analyzing the entire squares that make up the pieces rather than individual pixels, accounting for potential discrepancies in the pixel colours. Best of all, WASM is backward-compatible with existing web technologies, making it a prime candidate for adding features and improvements that would normally be too resource-intensive.

Conclusion

Through the thorough analysis and evaluation of various approaches to solve the challenges of building nestris.online, I implemented socket connections and my

self-designed pixel analysis algorithm as the ideal solutions to allow players to watch their opponent's screens and to determine the upcoming piece for each player. These features were critical to the functionality of the project.

The decision to use socket connections over HTTP requests allowed quickly exchanging the players' game video data via the server. With the lack of data overhead and the consistent server-client connections, the data transfer speed using sockets was fast enough to fulfill the requirement of the game video frame rate being at least 30 Hz.

Analyzing the upcoming pieces was the most interesting challenge of this project, as it required thinking outside the realm of popular image recognition libraries. The pixel analysis algorithm that I designed and implemented satisfied the requirements of being accurate, fast and cheap. Through blueprinting the pieces and implementing some simple conditional logic, the algorithm ran with zero financial costs, and was 99% accurate at a rate of 4 Hz. This satisfied all three constraints defined in the project.

Overall, the engineering decisions satisfied the design constraints and achieved high execution frequencies to conduct real-time data transfer and analysis on a "frame-perfect" retro game (North, 2020). With Classic Tetris as a game being unlikely to change anytime in the next couple of years, the ideas presented in this blog post should remain viable going into the near future.

In addition, building the piece detection algorithm has further motivated me to create game analysis tools for competitive Classic Tetris players. In 2021, Classic Tetris' competitive scene is pushing players to improve and innovate more than ever, and providing quality analytics has the serious potential to transform the playing field.

References

Clark, L. (2017, February 28). *What makes WebAssembly fast?*. Mozilla Hacks. https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/

Classic Tetris. (2020, January 10). *Greatest Classic Tetris Match EVER! Greentea vs. Joseph EPIC 2019 CTWC Quarterfinal FULLSCREEN* [Video]. YouTube. https://www.youtube.com/watch?v=5sxMqLjTv6k

Fireship. (2020, October 26). *Web Assembly (WASM) in 100 Seconds* [Video]. YouTube. https://www.youtube.com/watch?v=cbB3QEwWMlA

Gamage, T. A. (2017, November 19). *HTTP and Websockets: Understanding the capabilities of today's web communication technologies*. Medium. https://medium.com/platform-engineer/web-api-design-35df8167460

Google. (2021, May 12). *Pricing*. Google Cloud. https://cloud.google.com/vision/pricing

Luecke, D. (2018, January 26). *HTTP vs Websockets: A performance comparison*. The Feathers Flightpath. https://blog.feathersjs.com/http-vs-websockets-a-performance-comparison-da2533f13a77

Mozilla. (2021, February 19). *Canvas API*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

North, C. (2020, June 10). *What Does It Mean To Be Frame-Perfect?*. ggn00b. https://ggn00b.com/for-noobs/frame-perfect-explained/

Ubl, M., & Kitamura, E. (2010, October 20). *Introducing WebSockets: Bringing Sockets to the Web*. HTML5 Rocks. https://www.html5rocks.com/en/tutorials/websockets/basics/

Zurawel, K. (n.d.). *Making NES Games in Assembly.* Strange Loop. https://www.thestrangeloop.com/2017/making-nes-games-in-assembly.html