AlphaJoust





Micah Vargas
IMGD+CS BS

Collin Shields
CS BS

Ellys Gorodisch
IMGD+CS BS

William Smith
IMGD BS

Emanuel Sweeney
IMGD+CS BS

Ever since the 1980s, gaming has exploded from a simple pastime to a massive, world-spanning community. Within this society of people brought together by a love of games, one group seeks to optimize their favorite games to the millisecond. Known as ‘speedrunners’ (Hay, 2020, p.1), these players use all manner of high-skill maneuvers, exploits, glitches, and a bit of luck to complete games as quickly as possible. Speedrunning is the endeavor of taking typically a video game and seeing how fast a person can complete it from start to end (Lafond, 2018). Speedrunning events have become massively popular, such as the non-profit Games Done Quick organization that has raised over 50 million dollars (GamesDoneQuick, n.d.) for charitable groups (Brewer et al., 2023, p. 289). These events often draw in large crowds, both online and in person.

This end product involves many hours of human effort. Searching for exploits and time saves usually requires heavy trial and error by a large group of people. Furthermore, speedrunners may want to compare their best times with the idealized, best possible versions of their runs. Enter Tool-Assisted Speedrunning (TAS). The term TAS was coined during speedrunning efforts for the original Doom game when Andy "Aurican" Kempling created a modified version of the game that allowed for recording demos in slow motion to create a more perfect speedrun. Over time, TAS systems have been refined to be more and more complex for the purpose of creating faster and faster speedruns. Today, the modern TAS is a machine that plays a game with its controls at given times pre-selected. This is to ensure that it does the same inputs for every run of the game and can be reprogrammed to display runs through a game with perfect precision. However, these still require manual pre-programming for every single input at every single period of time. Most video games play at sixty frames per second, and the length of speedruns can be measured in minutes or hours depending on the game played. This also doesn’t account for any failed TAS sequences that may require the creator to discover what went wrong and try again. As such, this is a rather tedious way of designing an optimal route – a better method must exist.

Rather than hard-program a machine to act frame-by-frame, we endeavored to use the power of artificial intelligence to design a genetic model that trains itself to play a video game. The advantages of this are two-fold. A learning AI can be left alone for a few hours and achieve the same or similar results without the tedium of creating a TAS. Secondly, this also has the possibility of exposing new speedrunning strategies for players. The AI may find a new exploit, a better path that players did not consider. This second benefit has been proven before in the realm of tabletop games. For example, AlphaZero has been used to learn new strategies in chess. Transplanting that idea to the realm of video games is not a far-fetched possibility. Mastering a game is quite a complex problem for an AI to solve – the plentiful amount of distinct situations for it to adapt to, all while it seeks its goal of maximizing a utility, ensures that. As such, AI programs like these may also be applicable to real-world situations such as a delivery robot that learns the optimal path and movement to its destination through deep learning.

Various tools have already been created to solve similar problems in the past, ranging from Tool-Assisted Speedruns to various forms of AI algorithms trained to accomplish a high-yield utility in a certain video game. These approaches come with their own strengths and weaknesses.

Many of these speedruns require almost superhuman timing or skill (Schmalzer, 2021). For example, speedrunning the original Super Mario Bros. for the Nintendo Entertainment system requires players to execute ‘frame-perfect’ inputs which require the player to accurately time their movements to 1/60th of a second multiple times throughout the speedrun. As such, developers have created TAS tools to play through games like Super Mario Bros. to speedrun the game perfectly using all the techniques and strategies that have been refined by speedrunners over the past decades. TAS systems are a very effective solution, as they remove human limitations and will perform

the best performance possible. However, TAS requires predetermined strategies and knowledge on how to create the ideal fastest route from start to finish of a video game. It also requires information on certain tricks that can be executed within the game, typically strange glitches that are utilized for a faster time. TAS bots will also always reproduce the same run that they were coded for which is both an advantage and disadvantage. As compared to an AI agent, TAS is non-exploratory as it will follow a predetermined path to recreate a best run that a human might intervene in. As such, many companies and researchers have developed agents to play different video games extremely well using different methods.

OpenAI created an agent called 'Five'. An agent player trained using machine learning to play the 5v5 multiplayer online battle arena video game, Dota 2. Through its large-scale deep reinforcement learning, the agent was able to defeat top players of the game (OpenAI et al., 2019). Dota 2 is an extremely competitive game with its highest prize pool at a competition being 40 million dollars. As such, players of the game are fiercely skilled and play for thousands of hours. In 2017 however, OpenAI Five was able to defeat some of these players, utilizing skills like long-term planning, situational awareness, and cooperative play (Raiman et al., 2019). While OpenAI might've had a full company's resources, producing highly competitive AI agents has been shown to be possible by small groups and individuals too. The well-known platform fighting game Super Smash Bros. Melee has had an extremely invested community since its release in 2001. A group of three individuals saw the opportunity to use deep learning to train an AI agent to play Melee at a high level. Using deep reinforcement learning, they succeeded in evolving the agent and eventually brought it to tournaments to play against well-known players (Firoiu et al., 2017). Although it wasn't as incredibly competitive as OpenAI Five it was still able to win and put on a great performance against such skilled players. These agents can be trained to reach high levels of competence, even without large, well-funded teams backing them. While both of these AIs are impressive in their own right, they are examples of narrow AI, only able to succeed in Dota 2 and Super Smash Bros. Melee respectively. There still exist many games that no one has developed agents for, and many strategies yet to be discovered.

While an agent specifically trained for a certain game will likely know how to play it best with enough training, there also exist agents that have been trained to generally play video games. They are trained on multiple games to find out how to adapt to new environments and succeed when moving from game to game. They have the advantage of being able to play multiple games, learning to adapt and use their commonly shared mechanics (Liébana et al., 2020). However, it is not known if they can reach the same depth of skills and explore as many new possibilities.

The solution we developed is a narrow AI that uses genetic algorithms to learn how to play the 1982 arcade game *Joust*. *Joust* is an action-platformer game where the player takes the role of a knight mounted on a flying bird, and the goal is to defeat other knights with your lance by flying into them. If two knights strike each other simultaneously, the one who survives is the one that strikes from above. Despite being physics-based, the game is incredibly simple, with the only actions being to flap up into the air or to move left or right. This grants the benefit of a simple baseline, allowing us to place our focus on the AI player instead.

The key difference between our program and other retro game optimization efforts is our implementation of artificial intelligence via a genetic algorithm. The use of any kind of AI is still generally uncommon in gaming, and past examples like *learnfun* by Tom7 are more primitive and simple (Murphy, 2013). Outside of the artificial intelligence space, humans optimize games through enormous amounts of time and effort in real-time speedruns, or with months of labor putting together a TAS. We planned to have our *Joust*-playing AI, AlphaJoust, to be able to go from no game experience and very limited knowledge to expert performance in a matter of hours, not months. Additionally, the agent may develop strategies that people would not have thought of, with no human biases to interfere with the training process.

Functionalities	Improvements Over Existing Methods
Learning	TASes can not learn how to play; every frame's input must be manually programmed ahead of time. AlphaJoust can start with very little information, and learn how to maximize its heuristic.
Training	Human speedrunners must develop muscle memory over many hours of practice to achieve optimal or near-optimal play. An AI program does not get tired or distracted and can improve significantly faster.

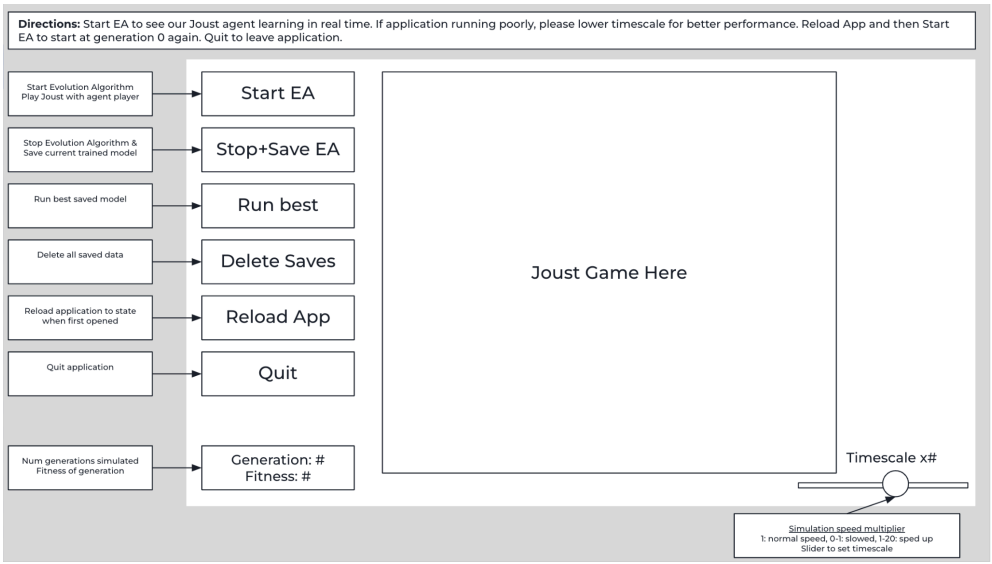
Creative Strategies	The extent of human gameplay is limited by people's biases and imagination. An AI does not know which strategies are "creative", it will find and expand upon any strategy that gives the best results.
Iteration	The means of iteration is inefficient among humans. Players can share strategies with each other, but in order to complete the best speedruns, each individual must build up skills and manage their mentality and other aspects of their lives. AlphaJoust has just one model to manage, with perfect execution, no distractions, and rapid genetic evolution.

We recreated the game *Joust* in the Unity game engine, using C#. C# is the primary programming language supported by the Unity game engine. Relying primarily on C# rather than Python, the more common choice for AI programs, made it much easier for us to integrate the artificial intelligence model with the Unity game so that we could extract the required values needed for training the model. If we did not use a language that can integrate with the game code in Unity, we would have had to write the values to a file or use a number recognition model on a screenshot of the screen, both of which would be highly impractical. For the artificial intelligence model itself we used a genetic algorithm technique called NeuroEvolution of Augmenting Topologies, or NEAT.

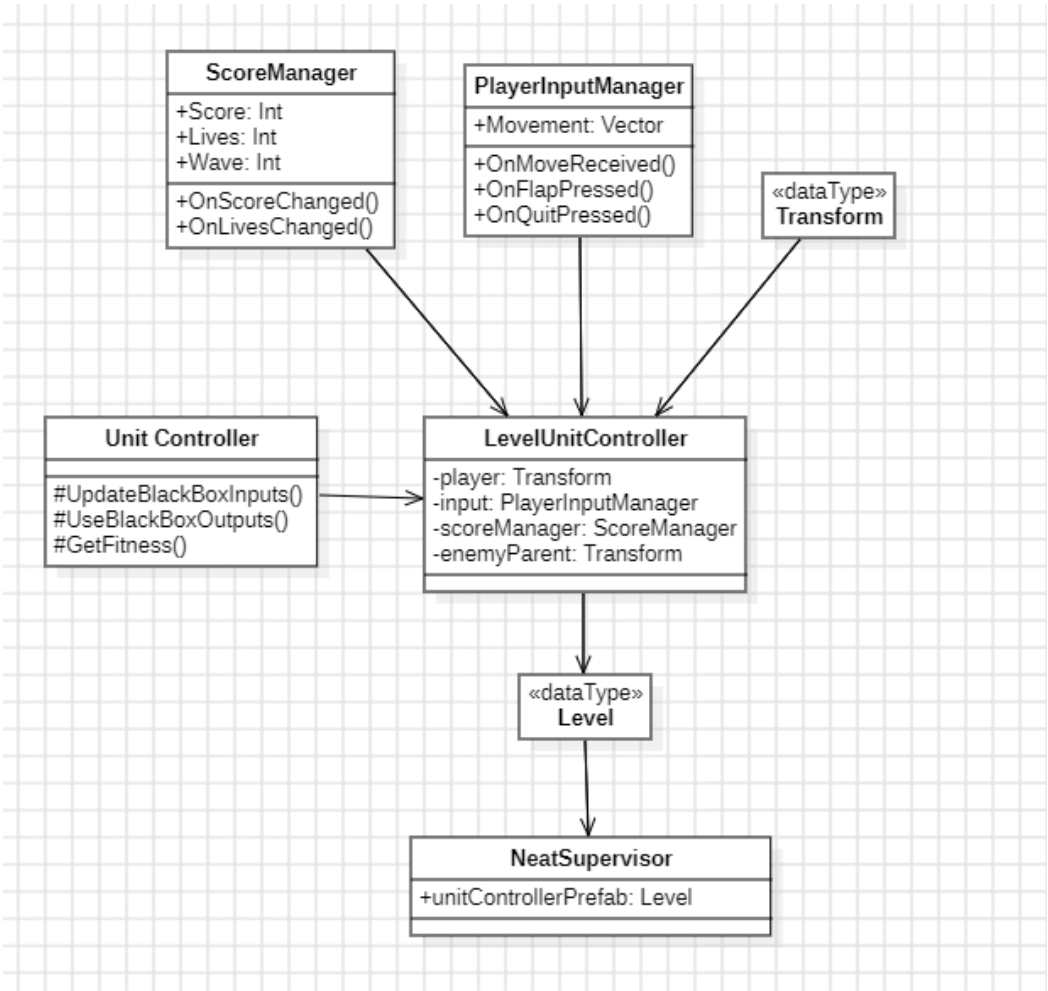
NEAT works by applying three key techniques: genetic encoding, tracking genes through historical markings, and protecting innovation through speciation (Stanley & Miikkulainen, 2002, p. 1). NEAT's first technique, genetic encoding, works through a system of node genes connected by connection genes. During mutation, both new node genes and connection genes can be added. The weights of the connection nodes as well as whether or not they are expressed can be also affected. Through this style of mutation, genomes of wildly varying sizes are created. Some of these genomes have the same node genes just with different connection genes. The second technique that NEAT utilizes, tracking genes through historical markings, works through a system the creators call innovation numbers. Whenever a new gene appears through mutation, it is given a new global innovation number. Then when comparing two genomes, NEAT can easily distinguish which genes share the same historical origin by simply checking the genes' innovation numbers. This allows any two structures to be combined without topological analysis (Stanley & Miikkulainen, 2002, p. 2). The third technique, protecting innovation through speciation, allows NEAT to divide the population into individual species based on topological similarities. This system uses the innovation numbers to compare how similar two genomes are. The more disjoint they are, the less likely that they will be grouped into the same species. This mechanic preserves diversity among the population of genomes by employing explicit fitness sharing where genomes in the same species must share the fitness of their niche. These three key techniques allow NEAT to dominate in many difficult tests. For example, NEAT far exceeded the capabilities of other leading neuroevolution artificial intelligence models on the double pole balancing without velocity information task (Stanley & Miikkulainen, 2002, p. 4).

Along with its high scores among testing benchmarks, there are numerous examples that already exist of NEAT being used to train artificial intelligence models to be able to play games well. Sanjay M. wrote on Medium about using the NEAT model for a game similar to *Flappy Bird* (Sanjay M., 2020). Boris Castellanos Matamoros at the University of the Andes in Columbia used a Python implementation of NEAT to train a model to play the arcade game *Galaxian* (Castellanos Matamoros, 2023). More researchers writing for *Personal and Ubiquitous Computing* used NEAT to train a model to decide enemy waves for a tower-defense game (Hind & Harvey, 2024).

Another major reason why we used NEAT is because there is a Unity plugin, publicly available on GitHub, that integrates SharpNEAT into Unity. SharpNEAT is a NEAT implementation built with C#.



UI Diagram



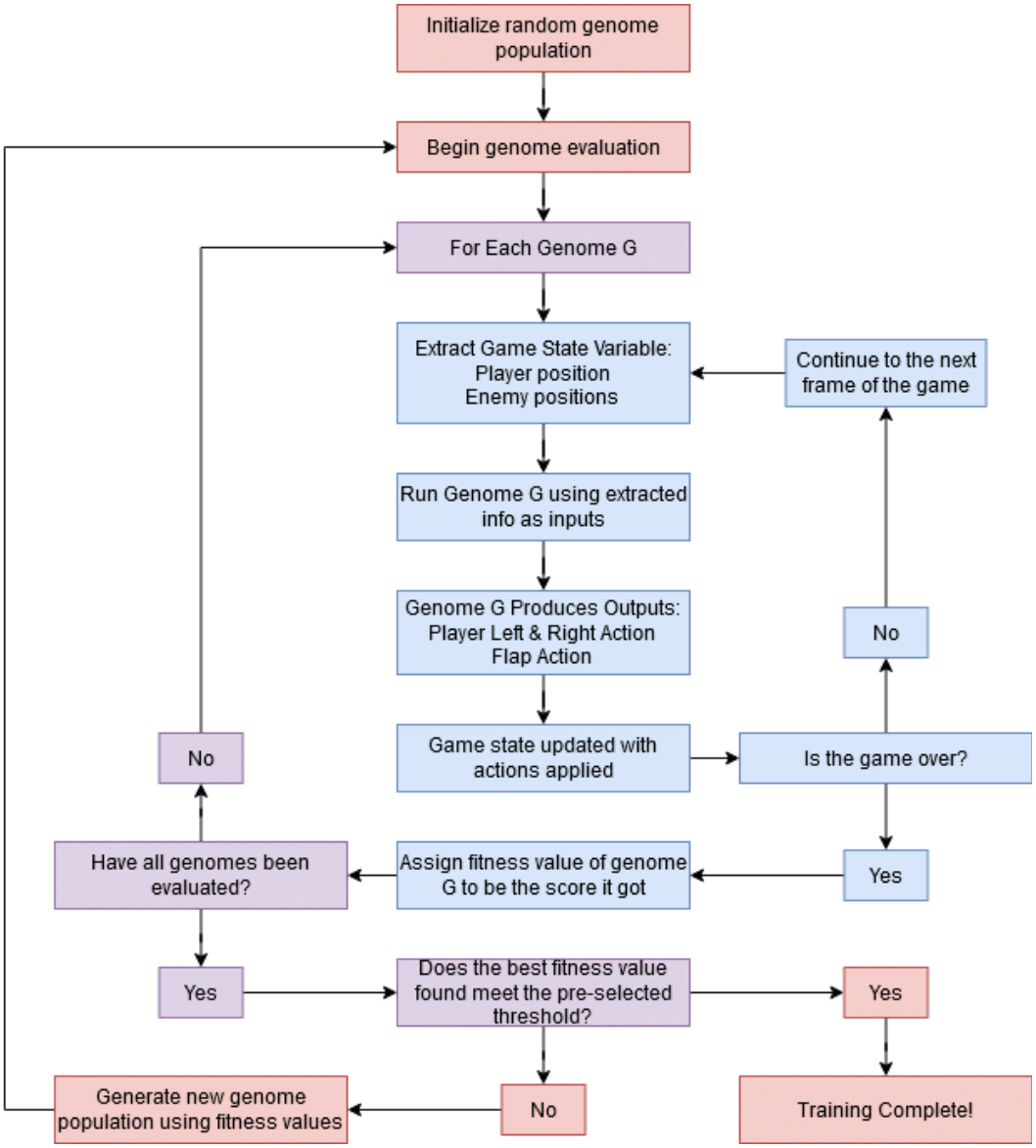
UML Class Diagram

Our implementation of the NEAT model using SharpNEAT uses a supervisor class called NeatSupervisor. This NeatSupervisor class takes a Level with a UnitController script attached to it. We created a subclass of the base UnitController class called LevelUnitController which controls a Level and the player within it. The NeatSupervisor creates one of these Levels for each genome in the population, and uses the LevelUnitController to control it according to the inputs and outputs that each genome receives and gives. For each generation, there was a population size of 30 with 6 unique species, and the model tested them for 120 seconds.

A video game is a complicated environment for an artificial intelligence model because there are many inputs and outputs. In addition, since the game *Joust* spawns enemies in waves, there are a dynamic number of enemies in the game at any given time. This makes it difficult to give a static number of inputs to the NEAT model. These enemies can also be of different types. The NEAT model needs to know these types as that would change how it approaches defeating them. The first two inputs for our NEAT model are the x and y coordinates of the player that the artificial intelligence model is controlling. Then the next inputs are the type, x coordinate, and y coordinate for each enemy in the level at each frame. Since the NEAT model requires the number of inputs to be defined beforehand, we gave it an arbitrarily high number so that we could increase and decrease the number of inputs that we are changing each frame.

We originally expected that we might have to provide more information to the agent. We were unsure if other details would be necessary for the model to learn effectively. Similar to the positions of enemies on screen, we thought we may need to include information about the layout of the level: the ground and the platforms that the characters move around in, but we did not. Additionally, we thought it may be beneficial to provide the agent knowledge about its current score and how long the game has been running. These factors are not necessary for the agent to know while it is playing. We use score to measure fitness throughout the evolution process, not as a value to track during one particular run. Similarly, we considered using time to measure fitness, but it would have introduced too many complications to have more than one utility to maximize.

The outputs of the system were simpler to implement. The agent controls player movement, moving the character back and forth using one output. The second output was used to determine if the agent should “flap” the player’s wings and move upwards.



High Level Solution Process

The main evaluator we are using to measure the success of our AI is score. The score is a direct measure of one's success in the game, so it is the value we used for fitness as well as our own evaluation of the AI's effectiveness. We were initially considering using time as an evaluator, such that a faster time would correlate to a better solution, but due to limitations in the NEAT library and our limited time working on the project we decided against it. The AI is able to prevent itself from dying, but is not very efficient in its time spent. Its strategy is also very simplistic and does not employ any advanced techniques. Notably, it would only flap occasionally to get height over an enemy, and opted for a strategy of predominantly running back and forth. This strategy was effective due to the altered level design we used. We found that the maximum score we could achieve after 30 minutes of training was 80250. Comparing this score and gameplay to a player we found that it underperformed compared to human players.

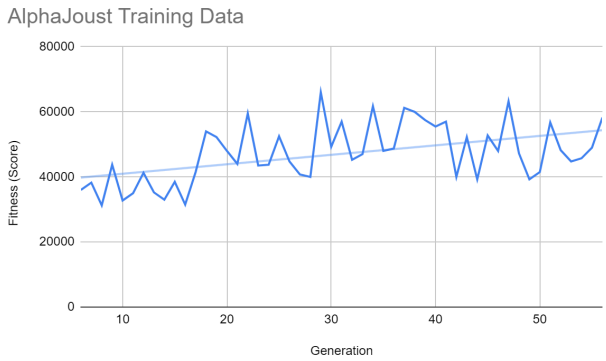


Chart showing the AI's best fitness score through each generation. It can be seen that AI gradually improves.

Throughout this project we learned about a variety of new topics and tools and built up our programming skills. Namely, we learned the inner workings of how genetic algorithms function, as well as how to best build around such an algorithm to maximize its effectiveness. For example, we learned that genetic algorithms thrive in situations where "success states" may not be as easily defined. This is an area that deep learning networks struggle with, as you have to define the optimal output set for deep learning networks to learn. We also learned that defining fitness evaluation is a complicated problem, as simplistic fitness evaluation can lead to simplistic solutions.

We developed our knowledge of Unity throughout the project, as we used a premade recreation of *Joust* in Unity for our project. On that note, we learned how to integrate NEAT into a Unity project using the C#/Unity NEAT implementation, SharpNEAT. SharpNEAT had built in UI systems for us to take advantage of, which allowed us to focus on integrating the game controls with the model. We used a Unity implementation of *Joust* as we found that old arcade games were built in a way that are not easy to port into modern systems. This forced us to use a recreation of the game, and the one we found was in Unity. Several team members were familiar with Unity, so we decided to use it.

One of the most important techniques we learned was building for scale. In order to push the training process to its peak, we learned how to create multiple instances of the game that run simultaneously. This sped up the training process as it allowed us to train the whole genome population at once.

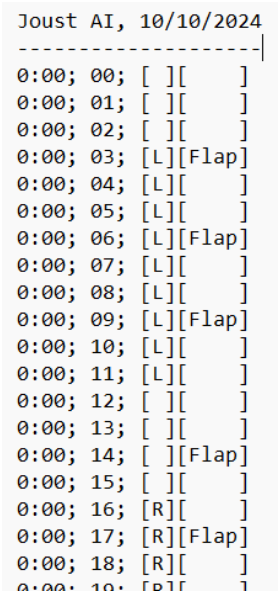
Overall, we were able to accomplish the vast majority of what was necessary for our project. Firstly, we set up the game in Unity, recreating its physics, mechanics – we wouldn't have much of a game-playing AI if there was no game to play it on. Additionally, we gave it the ability to play more than one frame at once. Then, we implemented NEAT into our program, configured the input and output, and configured it to have an adjustable timescale. This was important since it would allow us to train the AI at a much faster pace than it could be trained at a normal game speed.

Following this, we worked to optimize each frame so that we could run multiple games in parallel. From there, we turned up the number of frames to better train the agent. Finally, we did a test run where we gathered our data for our results. In short, we were successful in implementing most of what we had planned out at the beginning of the project.

We were able to accomplish most of what our team was planning to complete, but some features unfortunately were unable to be completed or simply were out of scope for the project. For instance, we initially intended to have the agent create a list of inputs as it played. Upon completion, all of these inputs would be written to a text file so that a hypothetical user could begin programming it into an actual TAS bot. More about this is written below, where hypothetical future contributions are discussed. We also intended to use time as a secondary utility, but that proved too complicated as the agent would have had to balance two different utilities and do complex prioritization calculations for them.

Meanwhile, some other features were implemented, but we did not get around to fully testing them. The most notable among these was the customizable set of parameters we displayed upon startup – namely, the generation duration, population size, and number of unique species are customizable. However, for the purpose of streamlined testing, we decided to focus on the agent's functionality with a few base parameters lest rampant experimentation distract us from our main goal. Furthermore, we also wished to train the agent for a longer period of time. Doing this would mean that our trial run would end up developing more generations. As such, we would have

seen a more clear picture of the agent learning and growing better at the game with time. However, we were unable to run training for a longer period of time.



Mock-up of the string of inputs to be plugged into a TAS.

While our project is well-developed, some additional features could be implemented to make this project even more viable for speedrunners as a tool to help create TAS bots. One potential contribution to this project would be for the AI to write its list of inputs to a file for the user to read later. This would likely be a simple feature to implement that would massively assist future speedrunners, as they would not only know the methods of the AI's play but the exact inputs for them to replicate, either for by-hand play or to input into a TAS bot. To do this, each frame would have the game's controller write the time of the gameplay, the direction of the stick input – this would be 'left', 'right', or 'none' as *Joust* does not include any meaningful vertical inputs – and whether or not it flapped the player's bird's wings. Furthermore, a second program can input this written function directly into a TAS. This would fulfill the design goal of making the creation of a TAS bot extremely expedient and easy. It would simply read these inputs at runtime and pre-program the player controller to execute all of these inputs, one at a time.

Another way that we could improve our AI work would be to increase the value of the display for potential users. We could display the simultaneous games in a manner that is easily usable. Currently, these games are stacked vertically on top of each other, which makes it hard to look at any more than one of them at a time. To fix this, the games could be visually stacked depth-wise. To make each game visually distinct, these games would have different opacities. We would set them at different z-depths to prevent any issue with the different games' objects interacting.

Other means of bettering user experience via UI would be to display the skill of each AI as time went on. To better portray the progression of the AI's skill, another viable implementation would be to show each generation's point values in a line graph updated throughout the training process. As time goes on and the AI player gets better, this line would steadily go up with time.

Our model is well-designed to play *Joust* specifically, but speedrunners take interest in all manner of games. Although this last possibility is far greater in project scope, a final contribution could be to implement the functionality of other, possibly similar, games into it. An example game would be the NES game *Balloon Fight*, another single-screen game where the goal is to fly into the top sides of enemies to disable them from attacking you. Unlike *Joust*, however, *Balloon Fight* requires that you follow the vulnerable enemies after the first attack and collide with them a second time. This would be rather simple to implement by way of giving more points for finishing off an enemy. By doing this, a future developer could make a finished project that allows speedrunners to create TAS bots for a wide array of games.

Contributions:

Name	Contribution
Micah Vargas	Set up Joust game in Unity game engine Set up NEAT to work with Joust game Configured model blackbox (inputs and outputs) Bug fixing LevelUnitController and model training Section 1. Introduction (10 Points)
Ellys Gorodisch	Set up agent environment in Unity game engine Configured NEAT supervisor Bug fixing Set up LevelUnitController with inputs and outputs Trained model Section 2. Your AI Application/Tool/Solution Development/Implementation (15 Points)
Collin Shields	Configured model supervisor Bug fixing Section 3. Final Test Results and Discussions (10 Points) Section 4: Lessons Learned (3 Points)
Will Smith	Set up agent environment in Unity game engine Configured model supervisor Bug fixing Edited and uploaded the Training YouTube videos Created the AlphaJoust Training Data Chart Section 2. Your AI Application/Tool/Solution Development/Implementation (15 Points)
Emanuel Sweeney	Configured model supervisor Debugging Assistance Presentation Work Section 1. Introduction (10 Points) Section 5. Conclusions and Future Work (5 Points)
Everyone	Background research Organization & Editing Consulting on the development of the model

Works Cited

Brewer, J., Ruberg, B., Cullen, A. L. L., & Persaud, C. J. (Eds.). (2023). *Real Life in Real Time: Live Streaming Culture*. The MIT Press. https://doi.org/10.7551/mitpress/14526.001.0001

Castellamos Matamoros, B. (2023). *NEAT for video game learning: Advancing agent intelligence through evolutionary algorithms*. https://hdl.handle.net/1992/73636

Firoiu, V., Whitney, W. F., & Tenenbaum, J. B. (2017). *Beating the World's Best at Super Smash Bros. With Deep Reinforcement Learning* (No. arXiv:1702.06230). arXiv. http://arxiv.org/abs/1702.06230

GamesDoneQuick. (n.d.). *All Events Tracker*. GamesDoneQuick. Retrieved September 13, 2024, from https://tracker.gamesdonequick.com/tracker/

Hay, J. (2020). Fully Optimized: The (Post)human Art of Speedrunning. *Journal of Posthuman Studies*, *4*(1), 5–24. https://doi.org/10.5325/jpoststud.4.1.0005

Hind, D., & Harvey, C. (2024). Using a NEAT approach with curriculums for dynamic content generation in video games. *Personal and Ubiquitous Computing*, *28*(3), 629–641. https://doi.org/10.1007/s00779-024-01801-z

Lafond, M. (2018). The complexity of speedrunning video games [Application/pdf]. *LIPIcs, Volume 100, FUN 2018*, *100*, 27:1-27:19. https://doi.org/10.4230/LIPICS.FUN.2018.27

Liébana, D. P., Lucas, S. M., Gaina, R. D., Togelius, J., Khalifa, A., & Liu, J. (2020). *General Video Game Artificial Intelligence*. Springer International Publishing. https://doi.org/10.1007/978-3-031-02122-0

Murphy, T. (2013, April 1). *The First Level of Super Mario Bros. Is Easy with Lexicographic Orderings and Time Travel...after that it gets a little tricky.* https://www.cs.cmu.edu/~tom7/mario/mario.pdf

OpenAI, :, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Pinto, H. P. d. O., Raiman, J., … Zhang, S. (2019). *Dota 2 with Large Scale Deep Reinforcement Learning* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.1912.06680

Raiman, J., Zhang, S., & Wolski, F. (2019). *Long-Term Planning and Situational Awareness in OpenAI Five* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.1912.06721

Sanjay.M. (2020, May 3). AI teaches itself to play a game. *AIKISS*. https://medium.com/aikiss/ai-teaches-itself-to-play-a-game-6442c9cef336

Schmalzer, M. (2021). Breaking The Stack: Understanding Videogame Animation through Tool-Assisted Speedruns. *Animation*, *16*(1–2), 64–82. https://doi.org/10.1177/17468477211025661

Stanley, K. O., & Miikkulainen, R. (2002). Efficient evolution of neural network topologies. *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, *2*, 1757–1762. https://doi.org/10.1109/CEC.2002.1004508