Playing RollerCoaster Tycoon with Reinforcement Learning

Jonathan Campbell
jonathan.campbell@mail.mcgill.ca
McGill University
Montréal, Québec, Canada

Clark Verbrugge
clump@cs.mcgill.ca
McGill University
Montréal, Québec, Canada

Abstract

Automating gameplay in a complex environment poses challenges for learning due to the large state and action spaces and the need for long-term planning. This paper presents a Gymnasium environment to allow for reinforcement learning research and experimentation in the video game RollerCoaster Tycoon, a popular amusement park simulation game with complex mechanics. We also present an approach to learn to play and win several game scenarios in this environment.

CCS Concepts

• **Applied computing → Computer games**; • **Theory of computation → Reinforcement learning**.

Keywords

game AI, reinforcement learning, deep learning

ACM Reference Format:
Jonathan Campbell and Clark Verbrugge. 2025. Playing RollerCoaster Tycoon with Reinforcement Learning. In *International Conference on the Foundations of Digital Games (FDG '25), April 15–18, 2025, Graz, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3723498.3723818

1 Introduction

The use of games in artificial intelligence (AI) research has long been a popular approach for developing and testing learning algorithms, motivating the development of simplified or custom-built game environments for AI experimentation [7]. In this paper, we present such an environment for the commercially-renowned video game RollerCoaster Tycoon, as well as detail a methodology to play and win large parts of the game.

RollerCoaster Tycoon (RCT) is a construction and management simulation game that offers a rich gameplay experience. Players design and manage amusement parks by placing rides, shops, and paths, with one end-goal being to make their park amusing for guests by the end of a certain in-game year. To this end, the game contains multiple metrics to measure park success, and an AI agent for the game must learn to understand these metrics and the spatial and temporal properties of the game, in addition to other gameplay mechanics such as ride unlocking.

In this paper, we introduce a new Gymnasium environment for AI research in RCT. Unlike prior limited-scope implementations, such as MicroRCT [3], this environment directly interacts with game code, offering a much more complex and realistic setting to test and verify our approach. AI agents can place rides, shops, and paths and receive statistics on park performance to guide future decisions in order to fully play the game. The full RCT game includes over 100 different scenarios, with diverse objectives and varying levels of difficulty, providing a broad and challenging landscape for AI learning.

Our experiments in this environment focus on a subset of these scenarios, with the goal being to 'win' the objective of each by attracting a certain number of guests to the park. To do so, we learn the choice and placement of rides and shops throughout the duration of a scenario using Proximal Policy Optimization with a custom state and action space. Unlike general video game AI models that rely on raw pixel data, our custom state representation efficiently isolates the most pertinent features of the environment, thereby reducing computational overhead. These experiments offer insights into the effectiveness of AI techniques when applied to this highly dynamic and intricate environment.

Specific contributions of this work include:

- a Gymnasium environment in which reinforcement learning (RL) experiments for ride and path placement and ride selection in the full RCT game can be conducted,
- an approach to play a full scenario in RCT by selecting and placing rides and shops, with the goal of attaining the in-game objective, and
- results that show the ability of the RL agent to train on and win multiple scenarios.

2 Related Work

Artificial intelligence algorithms have been applied to many types of games; we focus here on work done with respect to simulation games.

Rios and Chaimowicz [14] developed an AI for Transport Tycoon which focused on managing railroad routes using A$*$ with some heuristics. Their approach was able to defeat the in-game AI. Later, Konijnendijk [10] applied Monte Carlo Tree Search to the same environment with mixed results. In the domain of city-building simulations, Earle used fractal neural networks to play SimCity [4], and showed the ability of their model to generalize to varying board sizes.

With regards to reinforcement learning in particular, deep RL has shown great capability in playing video games over the past decade [15]. It has been used to achieve excellent performance in many arcade games [13], board games like Chess and Go [16] and 3D first-person shooters like Doom [11], amongst others.

The work most closely related to our own is by Cerny Green et al. [3], who introduced MicroRCT, a simplified environment inspired by RCT. Like in RCT, rides and shops can be selected and placed in order to optimize certain metrics for guests. The authors then use the MAP-Elites algorithm to explore the design space of their environment, considering combinations of metrics like excitement and intensity of rides. However, MicroRCT represents a very limited subset of the full RCT game environment. In MicroRCT, a fixed number of guests visit the park, not dependent on any metrics; the guests then follow a fixed path structure, around which rides and shops can be placed; their pathfinding and ride consideration behaviour is also simplified, and money is not considered. While their work introduced a valuable exploration of RCT mechanics and its design space, our approach goes further by utilizing the authentic mechanics of the game with only minimal omissions. Specifically, our model directly interfaces with game code and thus retains the original mechanics for guest spawning, paths, pathfinding and behaviour. These mechanics provide a much more detailed and accurate simulation and introduce a large number of complexities which is not modelled in MicroRCT. By consequence, our model can play official, complete game scenarios, each with varying path structures and ride availability. We note also that we initially tried to use the MicroRCT environment in initial tests, but the solutions learned by our agent did not transfer well to the real game.

In the same MicroRCT environment, Earle et al. [5] used reinforcement learning to demonstrate controllable procedural content generation. Their generators learned to build rides and shops such that specified levels of guest happiness were obtained.

Finally, in other work in RCT, Campbell and Verbrugge [2] presented an approach to procedural content generation of rollercoasters using reinforcement learning, with the ability to control various properties of a ride such as its excitement, intensity and nausea. Earlier efforts by Burke [1] explored genetic algorithms for the same purpose, while Ebert [6] applied recurrent neural networks (RNNs).

3 Background

RollerCoaster Tycoon (RCT) is a series of best-selling amusement park simulation games created by Chris Sawyer. In these games, players design and manage theme parks by constructing rides, concession stalls, paths and scenery. Simulated guests traverse the paths, go on rides and buy from the shops. Figure 1 shows a game screenshot.

The game includes many different theme parks, called scenarios, that that can be played. These scenarios each differ based on terrain, ride availability, initial ride placement and path structure (if any), objective and various other properties. We discuss some of these properties below, including rides, the number of guests, park rating and ride research.

Many different kinds of rides and shops can be built in the game. There are two general types of rides: flat rides and rides with tracks. Flat rides are those that comprise a single unit to be placed, e.g., a merry-go-round, haunted house or ferris wheel. By contrast, rides with tracks, such as rollercoasters and water rides, consist of multiple different pieces that must be individually placed, and are typically much larger in surface area. For our experiments, we limit selection to flat rides so that placement is easier. We allow placement of almost all flat rides (16 types), with shapes ranging from 3x3 to 7x1. We also allow placement of all 30 in-game shops, including food stalls, drink stalls, restrooms and other facilities. (We hope to extend to rollercoasters in future.)

Each ride has metrics that describe its effect on guests: an excitement, intensity and nausea rating. The more exciting the ride, the more likely a guest will want to ride it. Each guest also has a preferred intensity range, and will want to go on rides that fall in that range. Guests also have a variable nausea tolerance, refusing to go on rides over their threshold. Therefore, a ride can be more or less popular depending on its ratings, and certain rides will be more popular with certain guests. Another factor influencing the popularity of a ride is its price. Each ride has a secret ride value calculated by the game, based in part on its three ratings. If the set price exceeds this secret value, guests will not find it worthwhile to go on the ride. Therefore, a player must seek to optimize this hidden value so that they do not set the price too high. Rides also influence the number of new guests that visit the park, with each different kind of ride adding a flat number.

Indeed, the number of new guests that visit the park is a critical measure of its success. The more guests, the more money that can be made which can be used to create further rides. Further, and important for our experiments, the goal of many of the in-game scenarios is to attain a certain number of guests to the park by a certain in-game month (typically two or three in-game years) while maintaining a certain park rating. (There are other goals in other scenarios, but we omit them for the present work.)

Park rating is another measure of a park's success. It is a combination of many factors, including the number of highly exciting rides, the number of guests and guests whose happiness exceeds a certain threshold. Guests are considered happy if all of their needs are met, including going on rides that match their preferences, and not being hungry, thirsty, needing to go to the restroom amidst other criteria, which is where the importance of shops and facilities like restrooms comes into play.

Ride and shop selection is therefore critical as it influences park rating, number of guests and guest happiness. Other mechanics that affect such metrics include ride age and breakdowns, litter and weather, but we omit these for our experiments.

Finally, an important scenario property is ride availability. Each scenario has a different selection of rides and shops that can be built. Some can be built right at the start of the scenario, while others are unlocked by a research mechanic as time goes on. Thus, rides that are more exciting may appear soon in one scenario, but much later in another, forcing the player to adopt a different strategy.

4 Environment

In this section we present our Gymnasium environment for RollerCoaster Tycoon, whose code we release publicly[1].

As mentioned earlier, our environment directly interfaces with the game code, and specifically, the open-source reimplementation of RCT2 known as OpenRCT2 [9]. To do so, we have forked the game repository and made changes to a small number of files. In particular, we have added the ability to send commands to the game over a socket, and the code to process these commands.

[1]Our code can be found at https://github.com/campbelljc/rctrl.

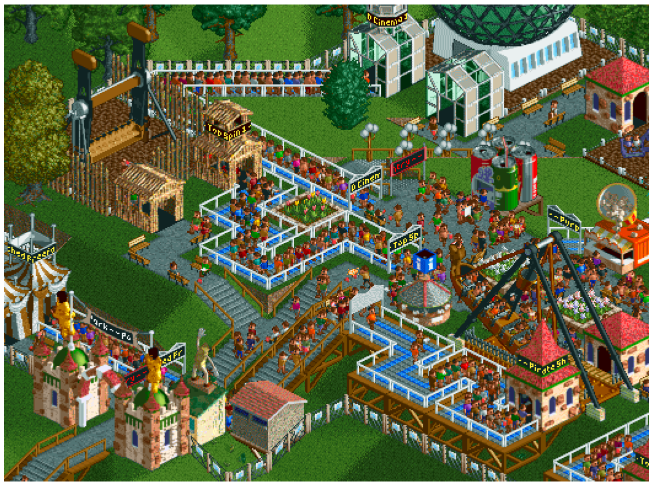

Figure 1: A screenshot of part of a player-created amusement park in RCT2. The path structure is in grey, with queue lines in blue leading up to the entrances of different rides, including a swinging pirate ship in the bottom right. Several food and drink stalls as well as a restroom and information kiosk are scattered around the paths.

We have implemented the following commands, which we sort into categories:

- park management: loading a scenario, opening and closing a park, pausing and unpausing the game, setting the game speed.
- ride management: placing a flat ride or path at a given position with a given price.
- information retrieval: the current path and ride structure, the list of currently researched rides, average guest happiness, path visit counts, park rating, company value, park value, cash on hand, ride statistics (excitement, intensity, nausea, popularity, profit), total guest thoughts on negative subjects and information on the scenario objective.
- simulation: running the game simulation for a specified number of game ticks (defaulting to a month).

One benefit of interfacing directly with the game is that the rides, paths and simulation can be seen graphically as a player would, which enables easier debugging and observation. To ensure the fastest speed possible, however, we turn off all animation while the simulation is running, so that the graphics will update only in small increments (e.g., at the conclusion of each month). Further, all commands are executed in pause mode, again to increase speed. Doing so has a small advantage compared to a human player, who cannot build while paused, but an efficient human could build quickly and then turn pause on again, so we do not foresee this difference to be impactful.

Finally, for our experiments, we also turn off certain game mechanics which would take longer to learn, but which can likely easily be solved algorithmically. Such aspects include those related to hiring staff such as handymen (no littering) and mechanics (no ride breakdowns).

There is also a financial element to scenarios. Players must manage their cash on hand and can build rides or shops only if they have enough money. However, players can take out loans in the game; for our experiments, we automatically increase the loan to a sufficient amount if we run out of money. We note however that in scenarios that the agent learns to win, it typically breaks even by the end since cash usually increases with number of guests. We leave a future exploration of cash (e.g., profit maximization) to future work.

5 Methodology

We present our approach for playing RCT below. We begin with a description and analysis of our dataset, followed by details on our reinforcement learning methodology.

5.1 Dataset

To validate our approach, we train on scenarios provided with the game. RollerCoaster Tycoon 2 has a total of over 150 scenarios, when including those from its expansion packs and from the first game in the series as well. Some scenarios have a difficulty: of those, 34 are considered at beginner level, 48 are challenging and 37 are expert. Each scenario also has an objective, of which there are 10. The most common objective (82 scenarios) is to achieve a certain number of guests by a certain year; this objective is the one considered for this paper, while the rest are left to future work.

We split these 82 scenarios them into their respective difficulties and also number of years until the objective must be met. We then use for this paper six of the beginner difficulty scenarios: Forest Frontiers (250 guests at the end of one year), Electric Fields (700 guests, two years), Bumbly Beach (800, two years), Barony Bridge (1200, three years), Crater Lake (1300 guests, three years) and Future World (1500 guests, three years).

Since the scenarios could be considered copyrightable game materials, we do not include them with our code, but instead provide a script that automatically sorts the parks into the categories listed above and provides statistics given the game's data folder.

Finally, some parks have a starting path structure, but a small number do not. Our RL agent currently does not build paths on its own, but requires paths so that rides can be placed adjacent to them. Therefore, we add to the latter parks a simple path structure so that there is enough space for rides. See Figure 2 for an example scenario which already has a path (and some rides) to start, and Figure 3 for a park where the path structure was augmented manually.

5.2 Reinforcement learning approach

We use Proximal Policy Optimization (PPO) as our RL algorithm. Below we discuss the specifics of our environment: state and action space, episode initialization and length, network architecture, termination condition and reward function.

5.2.1 *State space.* Our state space is comprised of a 2D grid with multiple channels containing all information that may be pertinent to the agent's decision-making. The grid shape is set to (87, 87) which fits the majority of game scenarios, omitting outliers. The grid channels include the following:

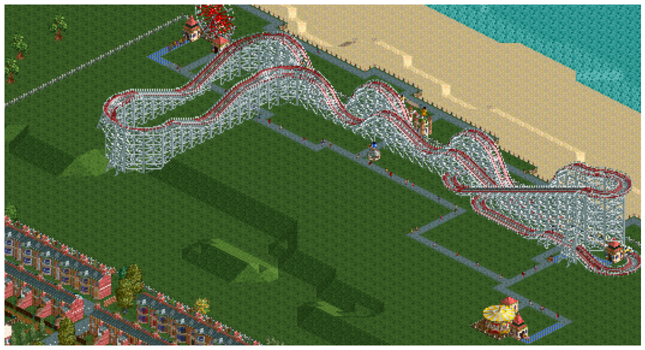

Figure 2: A screenshot of the Bumbly Beach game scenario, in which there is already a path structure, three rides and an information kiosk in the game's starting configuration.

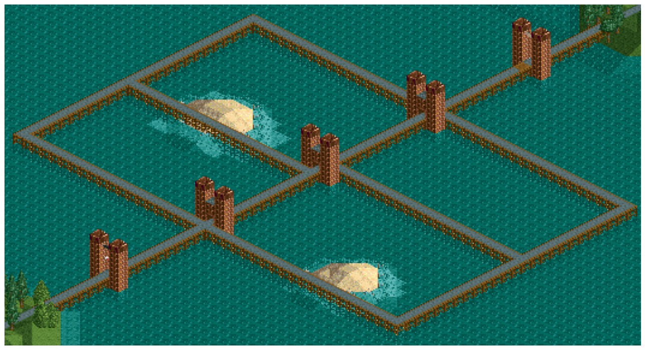

Figure 3: A screenshot of the Barony Bridge game scenario. Normally, the scenario starts with a single path from coast to coast. We augment the path structure with a square pattern to increase the number of rides that can be placed adjacent.

- one for each different ride type (boolean), indicating if a ride is at that position (either entrance, exit or other part),
- one for the price of a ride (float),
- one each for the excitement, intensity, nausea, popularity and profit of a ride (float),
- one for paths (boolean),
- one for sloped paths (boolean),
- one for queues (boolean),
- one for z-coordinate (float),
- one for path visit counts (float), and
- one for whether a square is otherwise occupied (e.g., by a ride that was present at the start of the scenario).

All values are normalized where appropriate. There are also three further channels to represent information about the entire current state instead of a single square; in these cases, the single value is repeated over the entire grid:

- one for the current month divided by the total number of months until the objective is met (float),
- one for the number of guests that must be met for the objective to be won, and
- one each for the current targets to be maximized, discussed further below (float).

We note that all the above information is public, that is, it is exposed to the player. There are other properties which could be useful for an RL agent, such as average guest happiness or ride values, but which a player cannot see directly, and we thus do not include those.

5.2.2 Action space. The action space consists of three parts: one to specify the (y, x) position to place the ride (a vector of probabilities, one for each grid square); one to specify the ride type (a vector, one for each ride type), and a third to specify the ride's price (a float).

Rides can be added at any position adjacent to a path, as long as there is sufficient space to place a ride at that location. Specifically, we check, for each path-adjacent coordinate, if there is space for a queue line of length 3, a ride entrance and exit, and then a 3x3 ride beyond that. We also check that the z coordinate of these squares is not higher than the path's z coordinate, nor is it too much lower, and that no square is currently occupied by a path or ride or out of bounds of the map, to satisfy the game's building constraints. The logits corresponding to invalid squares are masked so that these actions are not selected, which is a common practice to increase RL learning efficiency when there are many invalid actions [8].

We also add the possibility of replacing existing rides. Rides can be replaced with other rides as long as they are of the same size.

We also apply action masking to the ride type output. We mask all ride types that are not currently researched in the scenario. As the scenario unfolds and rides are researched, they are no longer masked; some rides are masked for the duration of a scenario if they are not present in it at all or not researched in time. (Each scenario has a different unlocking order and different selection of rides that can be researched which extends beyond the objective cut-off.)

We also add a 'skip' ride type to skip a timestep without placing anything. This action is only enabled when the map is full of rides and there are no more possible positions.

When any other action is taken, a command is sent to the game to place an entry queue line and exit path, followed by the ride entrance and exit, followed by the ride itself. Then, we then run the simulation for a certain number of ticks, until the next ride needs to be placed.

5.2.3 Episode initialization and length. As mentioned at the end of section 3, each scenario has a different selection of rides and shops that can be built – some at the start, some unlocking during the scenario and some that are not present in that scenario. To be able to choose the best rides/shops available at the start or during any scenario, we vary the ride research schedule at the beginning of each episode on a fixed scenario. Specifically, we make 10 random rides/shops available at the start of a scenario, and then unlock one new ride/shop at random at the beginning of each month; these numbers mimic the ones used in-game.

As to the number of actions per episode, we vary it based on the number of months until the objective is met. We place a number of rides equal to 1.5 times the number of months. This number attempts to mimic the gameplay of human players, who likely place rides throughout the duration of a scenario; it also modulates in-game cost and space concerns. Furthermore, since new ride types are continuously unlocked throughout the duration of a scenario, we anneal the number of rides to be placed per month over time, so that more rides are placed near the end when more are available to be chosen from. In particular, we use the formula $\frac{\sqrt{i}}{\sum_i \sqrt{i}} * X$ for every month i, where X is the total number of rides to place in the scenario.

5.2.4 Network architecture. The 87x87 grid is passed through a stack of four 2D convolutional layers (with 32 filters each and kernel size 3x3). It is then passed through four 2D de-convolutional layers, so that the output size is also 87x87 (but with one channel). This grid is flattened and then passed separately to the three dense layers, one for each component of the action space.

5.2.5 Termination condition & reward function. We terminate an episode at the end of the scenario's duration (typically two or three years), at which point the game reports whether the objective has been won or lost. We then reward the agent a flat bonus between 5 and 15 if it has won the objective, with the amount dependent on the scenario duration, with longer scenarios obtaining a higher reward. We found this difference to be important when training on multiple scenarios of differing lengths at the same time.

For shorter scenarios, this reward is sufficient to learn to win. However, in testing, it was found to be much harder to win scenarios of three or more years. Therefore, we introduce a partial reward at each timestep related to how much the number of guests in the park has changed since the last timestep. This reward for the target number of guests guides the agent towards winning the objective.

For longer duration scenarios which require more guests, this partial reward is still not sufficient. In these cases, special factors can arise that must be considered. For instance, after some time has passed, guests may complain about the absence of a restroom (or the high price of a restroom), causing attrition. Therefore, we also add partial rewards for the difference between timesteps of the percentage of guests who have a thought of needing to go to the restroom, being hungry, thirsty, thinking a facility fee is too high or that they can't pathfind to a ride. (Guests express their desires in the game through thoughts, which then go on to influence park rating and other metrics.)

We note that the number of guests in the park, as well as the number of guest thoughts per thought category, is public knowledge and accessible to players; indeed, it is often used by players to find out how to improve their park.

Finally, we give a reward of -1 if the game reports that the placed ride or shop cannot be built (and we do not run the simulation but instead immediately choose a new action). Although we have tried to ensure the agent chooses only valid actions by masking the invalid actions as discussed above, there can be certain small issues that arise. For example, there are many ride sizes, but we only verify that a 3x3 ride (the most common ride size) can be placed. It is possible that a 7x1 ride is chosen but cannot fit in the specified area. Since this issue can arise frequently, and since the proliferation of -1 rewards was found to disturb the learning process, we disable some of the game's construction checks. We note that this simplification does not give too much of an advantage over human players, since they can manage space much more effectively by varying the placement, length and slope of queue lines, which our agent does not support.

6 Experimental Results & Discussion

In this section we describe the training of our RL agent, followed by the results on our six dataset scenarios.

6.1 Model training

The RL model was trained using the RLlib implementation of PPO [12] on an M1 Max chip with TensorFlow 2 and eight rollout workers. Training was ended when the number of wins over all scenarios in the experiment reached a plateau, which came at about 130,000 timesteps or about 24 hours. Hyperparameter values included 5e-5 for learning rate, 0.3 for clip parameter, 10 for value function clip parameter, 1.0 for value function loss coefficient, 0.2 for KL coefficient and 0.01 for KL target, 0.01 for entropy coefficient, all determined through a grid search in a smaller-sized environment.

6.2 Results on scenario objectives

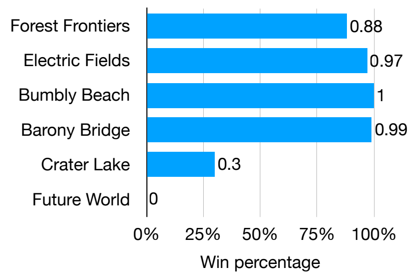

Figure 4: Win percentages for the four scenarios.

After training our model on a single scenario (Barony Bridge), we run it 100 times on each of the six scenarios. Win percentages for objectives are shown in Figure 4; the scenarios are ordered in terms of difficulty.

The figure shows that the RL agent was able to learn to win the scenario on which it was trained (Barony Bridge) as well as all the easier scenarios, with a very high success rate. Meanwhile, for the two harder scenarios which require a larger number of guests, the win rate is much lower.

Statistics about the kinds of rides favoured by the RL agent in different scenarios can be seen in Figure 5. We select Forest Frontiers and Barony Bridge as they are quite different in terms of scenario length (one vs. three years) and required number of guests (250 vs. 1200); it is thus expected that the rides selected by the agent will differ. Indeed, the figure shows that in Forest Frontiers, various gentle rides are chosen, while in Barony Bridge, the most popular ride is the Top Spin, a more exciting ride which attracts a larger number of guests to the park.

We also present the most frequent rides placed at the first and last timesteps for all four scenarios, as well as their average price, in Figure 6. The figure shows that the merry-go-round is favoured

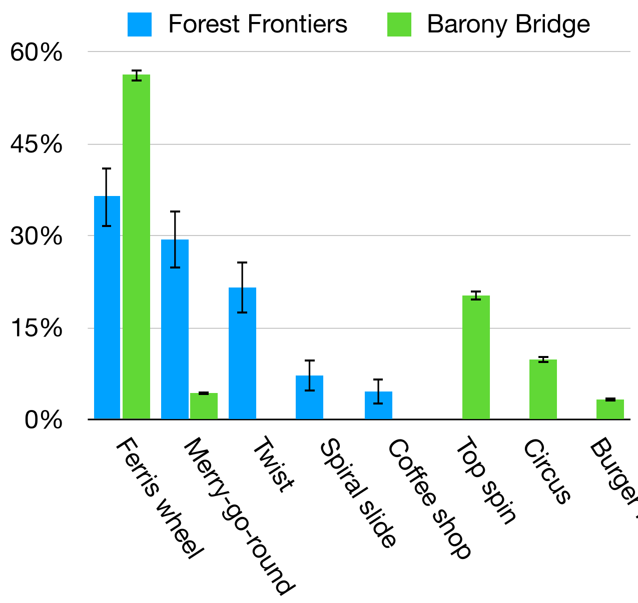

Figure 5: Most popular rides for two of the scenarios.

Rides placed at first timestep

Scenario	Ride	% of rides placed	Average price
Forest Frontiers	Ferris wheel	55.79%	$3.38
Electric Fields	Ferris wheel	63.74%	$2.50
Bumbly Beach	Ferris wheel	70.33%	$2.24
Barony Bridge	Ferris wheel	40.66%	$1.54
Crater Lake	Twist	87.50%	$4.09
Future World	Twist	38.20%	$3.67

Rides placed at last timestep

Scenario	Ride	% of rides placed	Average price
Forest Frontiers	Ferris wheel	39.56%	$1.31
Electric Fields	Ferris wheel	59.04%	$0.11
Bumbly Beach	Ferris wheel	27.27%	$0.19
Barony Bridge	Ferris wheel	46.59%	$0.16
Crater Lake	3D Cinema	85.11%	$0.11
Future World	3D Cinema	45.00%	$0.10

Figure 6: The rides placed the most at the first and last timestep for all six scenarios, and their average price.

as the first ride to be placed in most scenarios, likely because it is most common to be unlocked at the start of each, while the top-spin is favoured as the final ride in the harder scenarios. We also note that the average price of these rides is higher at the start and lower at the end. The agent was not trained to maximize profit, so it is likely that it has set prices lower at the end since it assigns more credit to actions taken at the end of an episode, and the lower the price, the more guests will ride it; it could also be that guests have less money after having been in the park for some time, so the prices must thus be lower.

Finally, Figure 7 shows the result of a Barony Bridge scenario played by the RL agent.

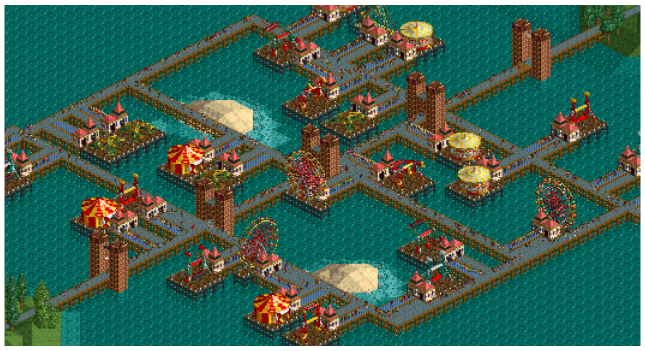

Figure 7: A screenshot of the Barony Bridge scenario near the ending month, after rides have been placed by an RL agent.

7 Conclusions and Future Work

In this paper, we introduced a new Gymnasium environment for RollerCoaster Tycoon, enabling reinforcement learning experiments in a rich simulation setting with the full spectrum of gameplay mechanics, including spatial planning, temporal decision-making and ride unlocking.

We also presented a methodology to play the game using reinforcement learning. Using a custom-defined state and action space, our agent learned to select and place rides and shops effectively, achieving the in-game objectives in several scenarios.

While our experiments focused on only a small subset of scenarios, the positive results indicate that RL agents can be trained to handle the diverse challenges of the game. Looking forward, we hope to expand our work to further, more difficult scenarios, including higher number of guest targets and other scenario objectives such as profit maximization. We also hope to explore the transferability of the trained models to other, unseen scenarios.

Another avenue to explore we would like to explore is the use of procedural content generation (PCG) in the game; for instance, the generation of a park that meets the scenario objective while also meeting other requirements, such as attaining a certain profitability, using a set number of unique rides, and so on. Similarly, PCG might be used to generate a path structure, which would help to create a more holistic approach to playing the game.

Acknowledgments

This work was supported by the Fonds de recherche du Québec, grant number 285499, as well as Natural Sciences and Engineering Research Council of Canada grant RGPIN-2019-05213.

References

[1] Kevin Burke. 2014. Hacking RollerCoaster Tycoon with Genetic Algorithms. https://kevin.burke.dev/kevin/roller-coaster-tycoon-genetic-algorithms/ Accessed on March 1, 2023..

[2] Jonathan Campbell and Clark Verbrugge. 2024. Procedural generation of rollercoasters. *IEEE Transactions on Games* (2024), 1–10. doi:10.1109/TG.2024.3404001

[3] Michael Cerny Green, Victoria Yen, Sam Earle, Dipika Rajesh, Maria Edwards, and L. B. Soros. 2021. Exploring open-ended gameplay features with Micro RollerCoaster Tycoon. *arXiv e-prints* (May 2021).

[4] Sam Earle. 2019. Using Fractal Neural Networks to play SimCity 1 and Conway's Game of Life at Variable Scales. In *Proceedings of the AIIDE Workshop on Experimental AI in Games*.

[5] Sam Earle, Maria Edwards, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. 2021. Learning Controllable Content Generators. In *Proceedings of the 2021 IEEE Conference on Games* (Copenhagen, Denmark) *(COG'21)*. IEEE Press, 9 pages. doi:10.1109/CoG52621.2021.9619159

[6] Dylan Ebert. 2017. Neural RCT: Using recurrent neural networks to generate tracks for RollerCoaster Tycoon 2. https://dylanebert.com/neural_rct/ Accessed on March 1, 2023..

[7] Chengpeng Hu, Yunlong Zhao, Ziqi Wang, Haocheng Du, and Jialin Liu. 2024. Games for Artificial Intelligence Research: A Review and Perspectives. *IEEE Transactions on Artificial Intelligence* (2024), 1–20. doi:10.1109/TAI.2024.3410935

[8] Shengyi Huang and Santiago Ontañón. 2022. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. In *Proceedings of the 35th International Florida Artificial Intelligence Research Society Conference (FLAIRS'22)*. doi:10.32473/flairs.v35i.130584

[9] Ted John. 2014. OpenRCT2. GitHub repo. https://github.com/openrct2/openrct2 Accessed on March 1, 2023..

[10] Geert Konijnendijk. 2015. *MCTS in OpenTTD*. Bachelor's Thesis. Maastricht, Netherlands. Advisor(s) Mark Winands.

[11] Guillaume Lample and Devendra Singh Chaplot. 2017. Playing FPS games with deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California). AAAI Press, 2140âĂŞ2146.

[12] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (PMLR'18, Vol. 80)*. 3053–3062.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533. http://dx.doi.org/10.1038/nature14236

[14] Luis Henrique Oliveira Rios and Luiz Chaimowicz. 2009. trAIns: An Artificial Inteligence for OpenTTD. In *2009 VIII Brazilian Symposium on Games and Digital Entertainment*. 52–63. doi:10.1109/SBGAMES.2009.15

[15] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. 2019. A Survey of Deep Reinforcement Learning in Video Games. *arXiv e-prints* (December 2019). http://arxiv.org/abs/1912.10944

[16] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.