Exploring the synergies between transfer in reinforcement learning and procedural content generation

Müller-Brockhausen, M.F.T.

Chapter 5

Scalable Procedural Content Generation via Transfer Reinforcement Learning

Procedural Content Generation algorithms that make use of Machine Learning have garnered significant attention from the general public due to their ability to generate text, images, or video content that is often indistinguishable from human-crafted artworks. These achievements typically necessitate a substantial quantity of data, which may be scarce in specialized domains such as game-level design. One machine learning technique, Reinforcement Learning (RL), can be employed to learn from trial and error to address this scarcity. However, the RL training process requires a substantial time investment and may prove unsuccessful in the case of sparse reward structures. This paper[a] empirically demonstrates the efficacy of curriculum learning as a viable solution to address scalability issues inherent to learning more complex tasks. Instead of trying to learn the whole space from scratch, we employ transfer learning on a curriculum from small to larger levels. We empirically validated this in a 3D vector-based environment, where the objective is to generate free-form tracks that facilitate a rider to move between designated points in the same vain as the flash game hit "Linerider".

[a]This chapter is based on the publication [172] Müller-Brockhausen, M., Khalifa, A., Preuss, M.: Scalable procedural content generation via transfer reinforcement learning. In: Data Science and Artificial Intelligence (DSAI). Springer (2024), https://doi.org/10.1007/978-981-97-9793-6

5.1 Introduction

PCG offers numerous algorithmic approaches to generate various content for games. Some involve the manual creation of intricate rulesets to generate diverse realistic Minecraft villages [225]. Other methods employ generative adversarial models [282] or language transformers [261] to create 2D Mario levels. Search-based and evolutionary algorithms are applicable to a variety of PCG-related tasks as well [271].

Most PCG work is focused on discrete 2D domains with a limited amount of actions. In this work, we explore the game "Linerider" [166]. It enables players to create new tracks by drawing free-form lines through mouse input. We expanded on that problem by extending the game to 3D space. This allowed for a complex continuous space that is different from generating images or text because training data is scarce, which makes it unfeasible to use supervised technique methods. Also, generating levels for games introduces the requirement that the generated content needs to be functional (the level has to be playable).

This data scarcity and the functional aspect of game content pushed researchers and developers [140] towards using the Machine Learning approach of RL. Although RL has achieved superhuman performance in game playing [28, 247, 279], it still has trouble with domains with sparse rewards. Due to that, successful training either requires a lot of training time or requires a small environment. In this work, we showcased that transfer learning for reinforcement learning helps in training agents on bigger vector-based domains in the same vein as Zakria et al. [300] work. Based on this, we think this paper contributes to the following under-explored areas in the combination of PCG and RL:

- Free-form content generation through vector-based action spaces
- Applying learning curricula to tackle RL's scalability issues

5.2 Related Work

Reinforcement Learning and PCG are a fruitful combination as the PCGRL [140] framework highlights. For example, it successfully balances competitive grid-based strategy game-levels [223]. Moreover, it enables the transfer of experience to generate Mario levels that optimize fun and historical deviation [242]. Furthermore, applied to Sokoban, it can generate more diverse and high-quality levels compared to other

approaches, such as GANs and VAEs [301]. It is also not limited to 2D worlds, as it can generate 3D Minecraft structures [129].

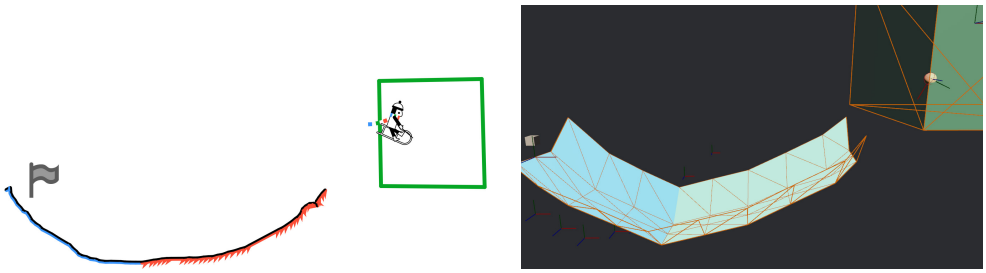
However, the key difference between our approach and PCGRL [140] is that PCGRL operates on an iterative basis, revising work generated by another algorithm and making incremental decisions regarding whether to adjust the current piece under consideration. This stepwise action space makes the agent learn to do repairs. For example, Gupta et al. [108] trained an RL agent to correct programming source code mistakes. However, we aim to harness RL for generating tracks from scratch, which aligns more with Campbell and Verbrugge's work [47], which generates new Rollercoaster Tycoon 2 (RCT2) tracks from scratch. A key distinction between Linerider and RCT2 is the requirement for rollercoaster tracks to loop back towards the starting point. This has been seemingly impossible to solve using exclusively RL, as Campbell and Verbrugge [47] resort to custom heuristics and the A* search algorithm to finish tracks initiated well by RL. In contrast, our tasks involve designing tracks connecting A and B but never looping back to point A.

Transfer Learning is an interesting avenue for PCG as it allows knowledge to be extracted from an existing model that then is effectively transferred to a similar game. Alternatively, multiple models can be combined into a single model, fostering collaboration and knowledge-sharing [228]. This knowledge could, for example, already be embedded in a LLM, which was not at all trained to generate content for a game. Nevertheless, with the right prompt, they can be led to generate Sokoban [269] levels. Alternatively, novel Mario levels can be generated through fine-tuning, which is also a sort of transfer [261].

In the realm of reinforcement learning, the integration of transfer learning with PCG remains an under-explored area. But it holds great promise as reusing existing models becomes even more important for sparse reward settings such as ours. Designing procedural curricula can facilitate the transition from simpler tasks to more complex ones, ultimately enhancing the agent's ability to generalize its learned skills [244]. Under Taylor and Stone's taxonomy [265], the varying reward structures across curriculum steps could be seen as a form of transfer learning.

5.3 Linerider

Originally Linerider (shown in Figure 5.1a) is a two-dimensional flash game released in 2006 [156]. It offers a basic yet engaging sandbox that can be applied in education to teach students fundamental physics concepts [6]. Linerider stands apart from



(a) Screenshot from the original Linerider game. Blue is a normal track and red is an accelerating track.

(b) Screenshot from our 3D version of Linerider. Dark blue is a normal track and light blue is an accelerating track.

Figure 5.1: A comparison of the original Linerider and our implementation. In both versions, a simple ramp that propels the rider into the goal is built. Note that in the original Linerider (5.1a) the green goal is purely decorative and has been added for a better visual comparison.

traditional gaming experiences as it lacks a predetermined objective. Instead, players create custom tracks by drawing lines using a mouse. Moreover, players can make the line accelerate the rider. At any point during the creation process, users can *simulate* their designs. This simulation drops a rider (a person riding a sleigh) at the starting point. From there on out, physics takes over, and the user can watch the rider roll down the created track. The second iteration, "Line Rider 2: Unbound" [166], released in 2008, introduces predefined levels with start and target points, requiring users to fill empty sections with appropriate track components to guide the rider successfully to the goal. Linerider still fosters an active community [156] that uses the game as a creative outlet to express themselves artistically. This expression can be in the form of audio-visualizers that match a specific song or apply orthographic projection to create seemingly 3D images that the rider drives through.

5.4 3D Linerider Framework

Our re-implementation of Linerider (illustrated in Figure 5.1b) emphasizes the development of solvable tasks that can be learned rapidly within the realm of reinforcement learning. To add to the original challenge, we move to 3D Space. Another change we introduced is to use a ball instead of a person riding a sleigh. The primary reason is that in early prototype testing, a rectangular shape gets stuck too easily in curves. Besides the influence on the rider's behavior, this change will allow for an alternative mode where the rider is controlled by a player/agent. Then, the game would be more

similar to Super Monkey Ball [83].

We developed our platform using a natively compilable language rather than Python. Research suggests that the primary obstacle in RL training stems from insufficient sample collection during environmental interaction [10]. Consequently, minimizing latency between agent actions and environmental feedback is essential. This aligns with other physics-based RL environments, such as Mujoco [270], where the physics calculations are done in C(++). By leveraging the Rust programming language [160] alongside the Bevy game engine [29] and Rapier physics simulator [71], we harness a powerful yet lightweight technology stack optimized for scientific exploration. This stack has a proven track record for scientific research in both PCG [31] and offloading physics computations to servers [146].

Our track design connects XYZ coordinates with flat track pieces with edge railings, enabling the rider to navigate curves without exiting the track. To mitigate the likelihood of the rider leaving the track, we have adjusted the restitution coefficient (which is used for allowing to simulate elastic behavior) to 0.5. This modification ensures that the ball bounces less vigorously upon impact, reducing the chances of unintended jumps above the edge railings.

5.4.1 Formalizing the Environment

We formulate our problem as MDP [26]. Software-side we use Gym [39] to allow RL-Agents to train in our environment. We rely on Pyo3 [212] for interoperability between Rust and Python, which allows us to use our Linerider Rust Gym like any other one implemented in pure Python. When designing an MDP, three crucial design decisions must be made: *observation space, action space, and reward design.* Each of these parameters makes or breaks the learning process.

To make Linerider 3D stand out from the related work, we opt for a continuous action space that enables vector-based PCG. This is in contrast to, e.g. [47], which has a static action space with ≈ 152 predefined pieces. The continuous action space expects RL to generate four values: The coordinates (XYZ) relative to the previous point and the type of track to generate. The track type can be one of the following: *Normal*, *Empty* (to allow building Jumps), and *Boost* (accelerates the rider in the direction of the track).

For the observation space, we use two variations: Full Observation and Sliding Window Observation. Both contain track coordinates and their type. In full, the observation contains all possible track pieces that can be put down (10 by default

Name	Description	Value
Rider distance to Goal	The closest, the rider has passed by the goal during simulation	$\frac{\iota}{1000}$
Track distance to goal	Encourages building towards the goal	$0.1 * (1 - \frac{\kappa}{0.5 \cdot \lambda})$
Goal reached by track	Given if the built track is within the range of the goal	0.5
Goal reached by rider	Given if the rider reaches the pre-defined goal	2
Scold premature end	Given if the rider falls outside of the building area	-1
Use of boost	If the chosen track type is of type acceleration	0.01

Table 5.1: Individual reward components that are used for the framework. In the equations we have the distance between the rider and goal (ι), the distance between the last placed track piece to the goal (κ) and the world size (λ).

but we also test larger sizes in Section 5.6.1). On the other hand, the sliding window only includes the last n placed track pieces (4 by default). Sliding window yields the possibility of transfer between arbitrary world sizes as it is not restricted to a specific world size. The sliding window observation is used for the transfer learning task as it is not restricted to a specific world size.

For the reward function, we use a simple sparse reward that is granted at the end of the episode (after the simulation is complete). The reward function focuses on making sure the ball and the track reach the final goal. It also rewards using boost tracks and punishes falling off track. Table 5.1 covers the individual reward parts and how they are calculated.

5.4.2 Environment Hyperparameters

The environment is configurable through a variety of parameters. Most are left unchanged after the initial test, but they can be relevant for more experiments, e.g., testing a different transition function for a transfer learning experiment. Unchanged parameters include: Rider mass (1.0), density (5.0), track width (0.5), track wall width (0.5), track wall height (0.75), maximum track piece length (1.5).

Parameters we do vary in experiments include: World Size ([10, 20, 30] with 10 as the default value, see section 5.6.1 for more details), steps to simulate before prematurely ending it (default is 1250, calculated via $\frac{1000}{80} * (80 * \lambda)$ where λ s the world

↓ Success Rate in % / → Size	10	20	30
Baseline-Ball	**100**	**98.92**	**96.55**
RL-Full-Ball	99.85	0.68	0
RL-Full-Track	0.15	55.26	0.14
RL-Sliding-Ball	99.99	0	0
RL-Sliding-Track	0.01	0	0

Table 5.2: A table comparing the success metric track reached goal and ball reached goal between the two RL observation spaces and our baseline heuristic. RL's reported success rate stems from the 10k evaluation episodes after training has concluded. In this table, the ball category also incorporates the case of both parts reaching the track.

size), and the task type that reflects the complexity of the problem ([up, same, down], see section 5.5 for more details). The world's size plays a role in determining several aspects of the environment. Firstly, it establishes the spatial boundaries within which track pieces can be placed, namely $[0, \lambda]$ for the XYZ-Directions, with λ being the world size. Secondly, the number of available track pieces and, consequently, the maximum number of steps per episode is directly proportional to the world size, as per step in the environment one track piece is laid down. Lastly, the rider's starting position and the goal location are randomly selected within the world size's coordinate range. Additionally, we ensure a minimum distance between these points to preclude scenarios where no actions are necessary for success.

5.5 Experiments

We apply Proximal Policy Optimization (PPO) [234] from the stable-baselines3 [215] library. All hyperparameters are left to their default values; thus the neural network size comprises 2 fully connected layers of size 64. Because experiments in deep reinforcement learning are notoriously difficult to reproduce [201], we provide *replay traces* [173] in the code repository[1] to allow our results to be at least verifiable.

To evaluate the success of an agent, we have the following metrics. The track reaches the goal (where the reward, in that case, is 0.5), the rider reaches the goal (where the reward, in that case, is 2), or both parts reach the goal (where the reward is 2.5). As the rider is uncontrolled, it can happen either by coincidence or through bad track design that the rider falls off the track without reaching the goal, while the track itself reaches the goal. An episode is considered a success if the rider reaches the goal. Hence, the track reaching the goal is only considered a step in the right

[1] `https://github.com/Hizoul/3dlineriderpcg`

direction. If both reach it, that is a nice feat, but as Figure 5.1b demonstrates, the rider can also reach the goal without the track doing so. To influence the agent to use the boost track pieces, we include the "use of boost" reward (Table 5.1).

We also have a heuristic agent that achieves a 100% success rate in both parts (making the ball reach the goal and the track). This is an agent based on a heuristic that calculates the direction vector between the last track piece and the goal. Due to its reliable success, we refer to this agent as *baseline* agent. While the heuristic reliably builds a straight line to the goal, the core of Linerider is to be creative. The heuristic enables us to train a baseline RL agent that can fulfill the goal. From this policy, we can then transfer to different reward functions to increase track traits such as jumps (Section 5.6.3).

For the scalability of our trained models, we increase the problem complexity by varying the **world size**. This influences the size of the world, the distance between the start and goal locations, as well as the number of placeable track pieces. Increasing this size ensures a delay in the reward signal making it more difficult for RL algorithm to learn the environment. Similar to the problem of late reward introduced by Bontrager et al. [36].

5.6 Results

The training regimen of all experiments consists of 100k steps. Every 2048 steps, an intermediate evaluation comprising 100 episodes is conducted. Upon completion of training, an end evaluation of 10k episodes is done. Each experiment configuration is executed five times, and all values presented in Figures and Tables represent the average of these runs. If a plot incorporates translucent colored areas behind a line, it denotes the confidence interval (95%) of the mean across the aforementioned five runs.

Before conducting the scalability experiments, we verified that RL can solve the basic task in the smallest size environment under default parameters (Section 5.4.2). We found out that when the goal is lower than the starting position or at the same height, RL achieves a 100% success rate. On the other hand, going upward is a little harder but our RL solution was able to reach 96.77% success rate. The remaining 3.23% are not complete failures, as the track still reached the goal. Looking at the built tracks we can see that the failures are due to not using the *boost*-track type.

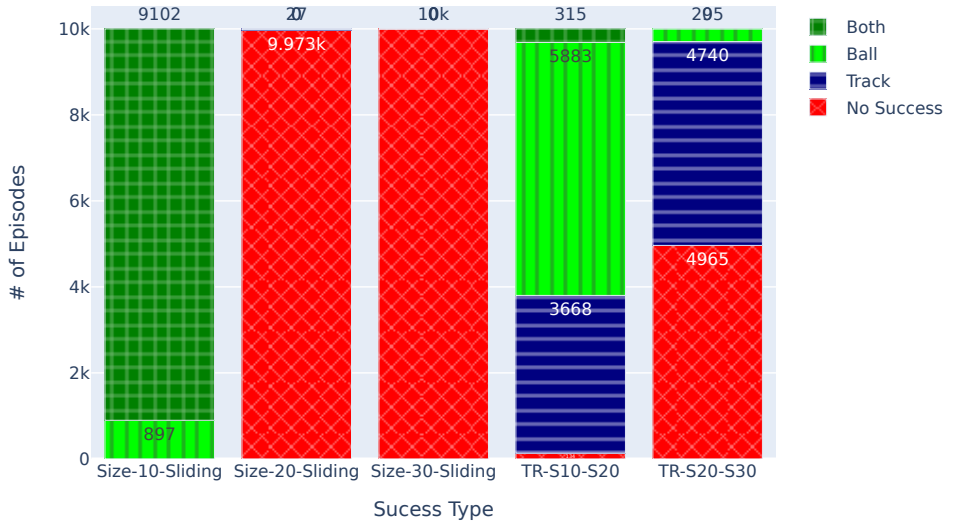


Figure 5.2: Comparison of performance on transferring between different world sizes. The necessity to transfer to be able to solve larger world sizes is clearly visible.

5.6.1 World Size Base Experiment

To investigate the impact of different observation spaces on learning efficiency, we conducted experiments on various world sizes ranging from 10 to 40. Note that the number of track pieces placed is equal to the world size (see Section 5.4.2). Our primary objective was to determine the minimum track size beyond which RL encounters significant difficulties in solving the problem. So, we decided to only focus on problems where the goal location is lower than the starting location because RL has solved this task in a world size of 10 with 100% accuracy. We compare both observation spaces (Full and Sliding Window) in this task, to see if it affects learnability. We also compared the results on the different sizes to our baseline agent.

Table 5.2 summarizes the outcomes of our experimentation. Looking at the numbers for the baseline, we can see the likelihood of the ball exiting the track prematurely before reaching the goal increases as the world size grows. However, even for the largest track size of 40, the success rate for the baseline agent remains consistently above 96%.

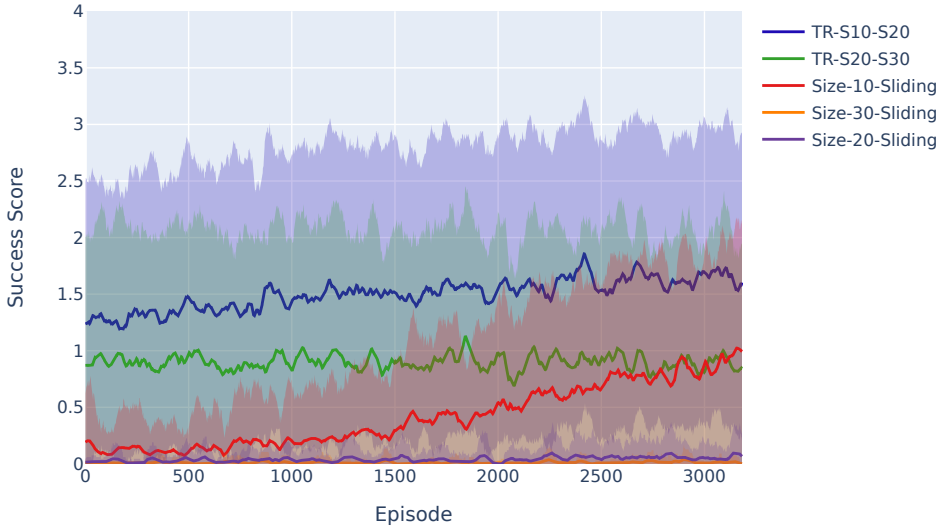


Figure 5.3: Visualization of success score (see Section 5.5) per episode during training. Lines are smoothed with a rolling window of 50. Lines prepended with "TR" are transferred policies. The first label after TR denotes the source task, while the second label represents the target task. For example, the TR-Down-Same policy is trained on the *A to B down* task and transferred to the *A to B same* task.

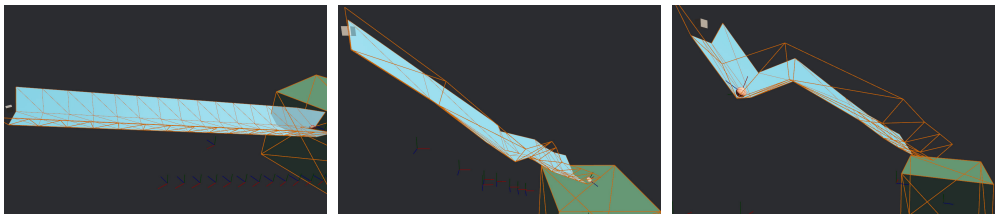
Examining RL in the two observation spaces yields mixed outcomes. While the sliding observation window of size four yields a marginal improvement of 0.14% in Ball success rate for a world size of 10, its efficacy counter-intuitively diminishes when learning larger world sizes from scratch. Despite the full observation space managing to at least build a track to the goal around half the time (55.26%), the sliding observation space proves unsuccessful and fails to generate any useful track. Conversely, the sliding window observation enables the transfer of a previously learned policy and is more successful in learning larger world sizes when training from scratch (see Section 5.6.2).

5.6.2 Transfer Learning Between World Sizes

In this experiment, we are using the sliding window observation as it will allow for easier transfer between sizes as its size is fixed between all the experiments. The

results from the previous section showcased the inability to facilitate generalization to larger world sizes. This prompted an evaluation of policy transfer techniques from smaller to larger ones. If learning tasks from scratch proves unsuccessful, employing a curriculum to introduce difficulty slowly can yield improved results [183]. More specifically, Zakria et al. [300] highlight that in PCG scenarios with a sparse reward, such as ours, an incremental increase in size during training improves performance on larger worlds.

Taylor & Stone [265] define four metrics of success for TRL: *Jumpstart*, denoting a higher initial reward than training from scratch. Time to threshold, signifying shorter training duration to achieve a comparable reward level. Total reward, indicating whether the transferred policy achieved a higher cumulative reward than training from scratch. Lastly, *asymptotic performance* means the agents' final performance has improved.



(a) Track built by the baseline in a world size of 20. Both track and ball reach the goal and the resulting track is a straight line from start to finish.

(b) Track built by "TR-S10-S20". The track does not quite reach the goal, but the ball does. The resulting track indicates that RL is close to an optimal solution.

(c) Track built by "Size-20-Sliding". Neither track nor the ball reach the goal, but despite a near zero success rate it underlines that something senseful is learnt.

Figure 5.4: A visual comparison of tracks built in a world size of 20 in the *A to B down* task by the baseline (5.4a), RL learning from scratch with the sliding observation space (5.4c) and RL after learning using a curriculum that transfers from world size 10 to 20 (5.4b).

Figure 5.2 clearly illustrates asymptotic performance, revealing that a transferred policy successfully manages some success in world sizes of 20 or 30, whereas training from scratch is completely unsuccessful in this scenario. The transfer policy is also more successful in making the ball reach the goal (61.98% vs 0.68%) than training RL from scratch using the Full Observation Space (see Section 5.6.1).

Additionally, Figure 5.3 showcases a jumpstart, lower time to the threshold, and a higher total reward compared to training from scratch on the tasks. These findings underscore the necessity of curriculum learning to address higher dimensional state spaces. However, with these improvements, a world size of 30 is not reliably solvable

Name	Description	Value
Touch	The amount of time the rider was touching the track during simulation	$\frac{\nu}{\mu}$
Air	The amount of time the rider was in the air during simulation	$\frac{\xi}{\mu}$
Speed	The overall speed during the whole simulation	$\frac{\sum \sigma}{\mu}$
End speed	The current velocity of the rider at the moment it reaches the goal	$\frac{\rho}{50}$

Table 5.3: Individual reward components that are used in reward shaping. In the equations we have the total time (μ), the time the ball is touching the track (ν), the time the ball is in the air (ξ), the velocity per step (σ) and the velocity at the end of the simulation (ρ)

through RL, whereas the baseline manages a success rate of 96.55%.

We provide a visual comparison of built tracks in Figure 5.4. It depicts the in-baseline solution: a straight line to the goal (Figure 5.4a). When transferring from size 10 to 20 RL finds a similar solution (Figure 5.4b). However, the track is more bumpy, which could explain why in 36.68% of the cases the track reaches the goal but not the ball (for "TR-S10-S20"). Lastly, we also include a failed track from the training from scratch on size 20 (Figure 5.4c) to underline that while the success rate is exceptionally low, the built tracks are not completely useless. If the second track piece would have less of an up inclination the rider would not get stuck and could reach the goal. We think that the reason for not learning to use the straight track instead of going up is due to the delayed reward which causes a problem in reward assignment with every piece in the track [36].

5.6.3 Reward Shaping

In this experiment, we let RL build tracks that fulfill the goal while also maximizing a specific trait. The traits we optimize for can be: Touch (rider contact with track), Air (opposite of touch), Speed (velocity of the rider), and End Speed (velocity of the rider when reaching goal). Table 5.3 details how these are calculated.

The experiments were conducted on the *A to B* Down variant in a world size of 10, as RL previously came close to a 100% success rate in this setup. As illustrated in Figure 5.5, reward shaping yields mixed results. We aim to normalize all values across 0 and 1. Hence, Air and Touch have an inverse relationship, and their sum will always be 1.0. Encouraging the rider to go faster proved to be effective, as evidenced by the highest speed attained during the simulation, and the highest end speed achieved upon

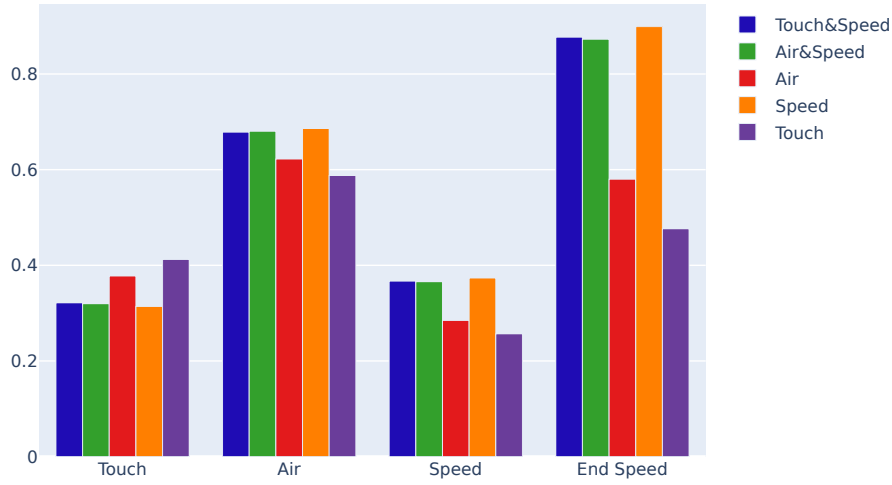
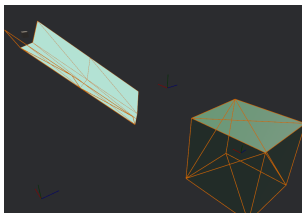
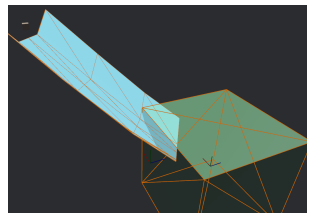


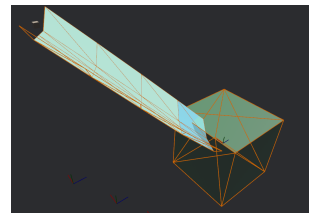
Figure 5.5: A comparison of certain track/simulation traits and reward shapes that try to optimize these traits. The color/label determines what the reward was attempting to optimize. The x-axis shows the performance per individual trait. Values in the Y-Axis have been normalized to 1.



(a) A track maximizing air time



(b) A track maximizing Touch-Time



(c) A track maximizing speed

Figure 5.6: A visual comparison of tracks built in a world size of 10 in the *A to B down* optimizing for different track traits through reward shaping.

reaching the goal was measured in the "Touch&Speed," "Air&Speed," and "Speed" runs.

The pure touch reward exhibits a comparable efficacy in achieving its intended objective, except for the combination "Touch&Speed" which results in a similar Air-time than the focus on Air-time alone. While pure air reward has slightly more air time than the touch experiment it is still lower than when optimizing for speed. We assume that this is due to the small world size not allowing for as much variation and that the effect would be more notable in a reward-shaping experiment with a larger world size. Hence, we can see a clear correlation between speed and air time. The disparities highlighted in Figure 5.5 are also evident in the built tracks portrayed in Figure 5.6.

For example, optimizing for Air-time results in the cessation of track additions shortly before the goal, ensuring that the rider will not engage with the track further, as evident in Figure 5.6a. Additionally, the utilization of accelerating track pieces, signified by their greenish-blue coloration, results in elevated velocities and ensures that the rider jumps far enough to reach the goal. When optimizing for track touches, as depicted in Figure 5.6b, the conventional non-boosting track pieces colored in light blue are used. Notably, the track does not exhibit a straight downward trajectory but rather a slight upward curvature towards the end to force more track touches. Lastly, when optimizing for speed, as illustrated in Figure 5.6c, the track maintains a straight line and predominantly incorporates accelerating pieces.

5.7 Conclusion & Future Work

We introduce a three-dimensional (3D) Linerider implementation that is capable of being learned through RL. Our experimental evidence suggests that RL can procedurally generate free-form vector-based tracks for 3D Linerider. This involves simple A to B tasks (section 5.6). While RL exhibits limited scalability to larger world sizes (section 5.6.1), we underline how curriculum learning is a promising direction to overcome these limitations (section 5.6.2). The usefulness of TRL is not limited to scaling to larger problem sizes. Moreover, we show that reward shaping enables the maximization of specific desired track traits such as rider speed (section 5.6.3).

Several promising avenues for further research on this problem exist. Regarding the reward shaping (section 5.6.3), it would be interesting to see if the effects are more pronounced in the data if the world size is larger. For the scaling experiment (section 5.6.2), an investigation into a more dense curriculum that increases the world size in increments of two or four rather than ten could alleviate the scalability issue to sizes of thirty. Moreover, the construction of pre-made levels with designated sections that require completion could be tested. This would enable our choice of action space and the PCGRL approach. Furthermore, we could build pre-made levels where only certain parts must be filled in. This could also be compared to using the PCGRL-Action space, where RL only decides if it wants to adjust a certain point or keep it as it is.

Additionally, as hinted in section 5.4, incorporating player control over the rider would bring the problem closer to that of Super Monkey Ball [83]. This modification would enable a generative adversarial approach to generate progressively challenging tracks. Considering our positive curriculum learning results, this might prove a successful extension to tackle the scalability issues to larger world sizes. Lastly, the track could be transformed into a pipe through which the rider would traverse, thereby decreasing the likelihood of the track alone reaching the goal without the rider's presence, as there would be no means of falling off the track.