Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Live Content Generation in Momentum-based Games

Semester Thesis

Benjamin Wolff

bewolff@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Oliver Richter
Prof. Dr. Roger Wattenhofer

July 26, 2021

Abstract

The goal of this project is to find a procedural content generation (PCG) method for online generation in momentum-based games. To this end, we make our own game called *Fast Painting*. The content should be solvable, varied, and it should exhibit the desired difficulty.

In our approach, two reinforcement learning agents, the solver and the generator, are trained in an alternating Markov game. We incentivize solvability and variety of the generated content by giving non-negative *variety rewards* to the generator when his actions produce recently unseen experiences for the solver. We sabotage the solver by limiting his vision, his ability to act and his training duration in an effort to create easier levels. In a procedure called *ghost mode* we give a reward for placing level components, that the solver only passes with initial momentum.

Our generator produces solvable levels without much training. The variety of its content increases very slowly across millions of training steps. Our attempts at enforcing difficulty have had little success. The achieved difficulty range is rather small and the more difficult generators still place many easy platforms. It is likely, that more training or an additional difficulty reward is needed to make the behaviors of the different generators diverge more drastically. The momentum reward seems to result in more interesting levels.

Contents

Chapter 1

Introduction

1.1 Motivation

The most obvious constraints a game level designer has to work around are

- Solvability: There must be a way to progress, otherwise there is no point in playing.
- Variety: Monotonous and only slightly differing experiences become boring very fast.
- Difficulty Fit: Beginners as well as advanced players should be able to enjoy the game. Offering levels of varying difficulty and labeling them, gives everyone a starting point and a goal to work towards.

That these properties are desirable is self-evident. But, what they mean concretely and how they can be enforced in a certain game is not straight-forward. This makes level design, also known as content generation, a very hard problem which creatives solve in an elaborate iterative process in the majority of games. In this semester thesis we will focus on PCG for momentum-based games i.e. games where momentum is the main game mechanic a player needs to master to do well. The nature of momentum in these games poses a significant challenge for level design. In a non-momentum platformer it is easy to ensure that gaps can be cleared because the maximum distance jump-trajectory is predictable. This changes completely when the character maintains momentum. Suddenly, the possibility of a jump depends on whether the player can build up the necessary momentum which can be a complex function of the surrounding environment. This often makes it infeasible to prove solvability. Someone has to perform the jump in order to show that it is indeed possible. Creating a hard level is therefore at least as hard as completing it. This means that games are often shipped with unrealized difficulty potential. In *Trials Fusion* for example, the hardest tracks made by players (e.g. Crown Control - Qkoosy [1]) are **much** harder than the most difficult tracks made by the level designers (e.g. Inferno IV [2]). Another

consequence of momentum is that the player needs to know about upcoming segments earlier to act properly. This can lead to situations where it is impossible for the player to know how to act without having seen the level before. If we generate the content as it enters the players vision, we should be able to enforce solvability with the players momentum in mind.

1.2 Goals

We want to find a method for procedural content generation in momentum-based games using our own game, called *Fast Painting*. The levels should adhere to the design philosophy described in section 1.1 i.e. they should be solvable, varied and exhibit the desired difficulty. On top of that, the generator should be able to iteratively generate while we are playing, making sure that the generated sections are solvable with the players current momentum. This creates a game mode that is simply not possible without PCG.

Chapter 2

Game Setup

To develop and test our PCG method, we made our own momentum-based game *Fast Painting* in Unity (Fig. 2.1). It is a two-dimensional motorcycle platformer and it can be played in a browser at: https://beb.vsos.ethz.ch [1]

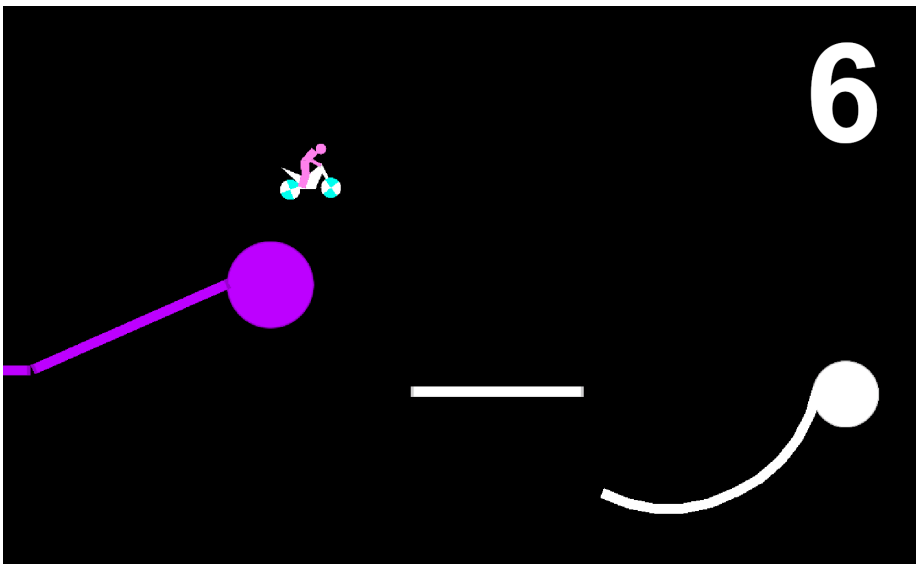


Figure 2.1: Screenshot of the game made for this semester thesis.

2.1 Gameplay

In *Fast Painting*, the player has to drive from left to right, coloring everything he touches purple. For every platform he passes, his score increases by one. The platforms are placed as the player goes along and the difficulty increases with

[1] If the link is not working in Jul-Aug 2021, please let me know.
Contact: bewolff@student.ethz.ch

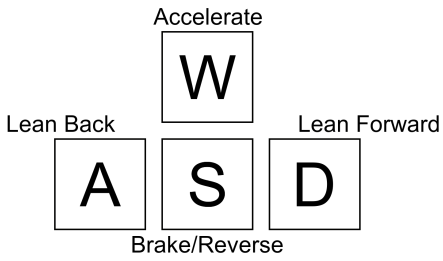


Figure 2.2: Keyboard controls for the game.

increasing score. If he falls or hits his head, the game restarts. The generated platforms will be different on every run. A player goal is to beat their friends' or their own high score.

2.2 Controls

The bike is controlled with 4 discrete inputs and can be played in a browser with a keyboard (Fig. 2.2).

- gas - accelerate the bike using the rear wheel motor
- brake/reverse - brake with both wheels or reverse if the rear wheel is in contact with the ground
- lean left/right - applies a torque to the rider, allowing him to stabilize or flip

Additionally, leaning forward moves the riders center of gravity over the front wheel and leaning backward shifts his weight back. His center of gravity remains in the respective position until the player leans in the other direction. Note that the mass of the motorcycle, which is twice the mass of the rider, does not shift. Transitioning into the opposite stance also gives the rider an initial torque thrust. This, combined with a very playful i.e. soft suspension, makes the motorcycle behave more intuitive with discrete inputs and allows for more interesting maneuvers like bunny hops (Fig. 2.3).

2.3 Platforms

The generated platforms can be parametrized by their shape

$$s \in \mathbb{S} = \{\text{Rectangle}, \text{Circle}, \text{Square}, \text{Ramp}\}, \tag{2.1}$$

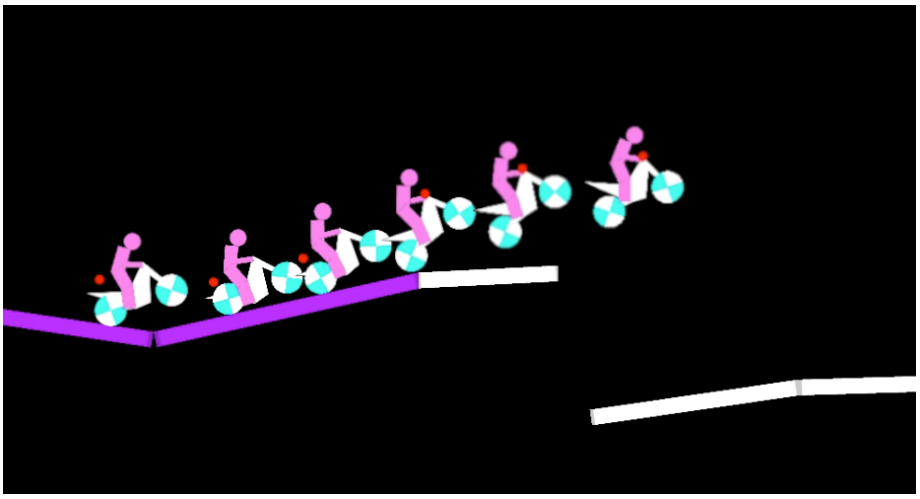


Figure 2.3: Overlapped frames at different stages of a bunny hop. The red dot indicates the center of mass of the rider since the stance of the rider is not animated.

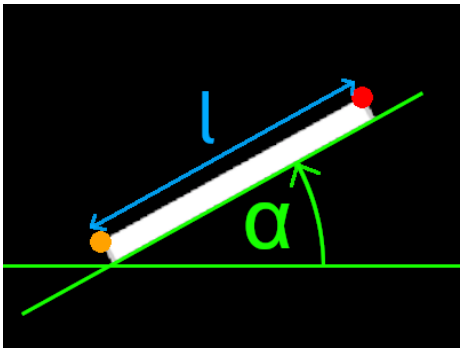
size $l \in \mathbb{R}_{>0}$ and angle α. The meaning of l and, due to different rotation symmetries the range of α, depend on the shape:

- rectangle with constant height - l denotes the length and $\alpha \in [-\frac{\pi}{2}, \frac{\pi}{2})$ (Fig. 2.4a)
- circle - l denotes the diameter, $\alpha = 0$ (Fig. 2.4b)
- square - l denotes the height and width, $\alpha \in [0, \frac{\pi}{2})$ (Fig. 2.4c)
- 45° ramp consisting of 5 trapezoidal segments - l denotes the length of the border on the inside i.e. concave side and $\alpha \in [-\frac{5\pi}{8}, \frac{3\pi}{8})$ (Fig. 2.4d)

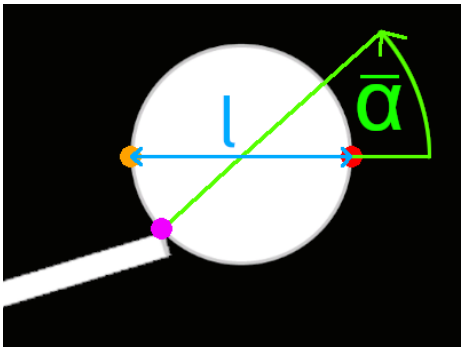
A platforms position is given by the position of its *anchor* $\mathbf{a} = (a_x, a_y)^T \in \mathbb{R}^2$, which marks where the platform begins. The *head* marks the end of a platform and its position is $\mathbf{h} = (h_x, h_y)^T \in \mathbb{R}^2$ (Fig. 2.4).

2.4 The Motorbike

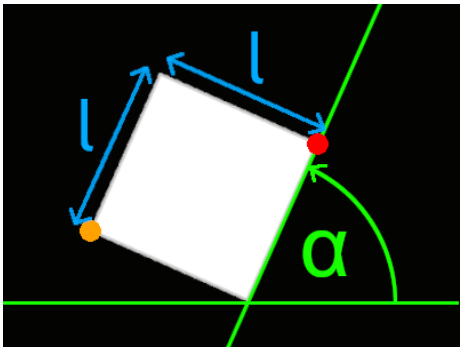
The distance between the wheel centers is 2u, where u is the length unit in the game, and the motorbike position $b = (b_x, b_y)^T \in \mathbb{R}^2$ is given by the center point between the wheel contact points (Fig. 2.5). We chose the larger gravitational acceleration $-15\frac{\mathrm{u}}{\mathrm{s}^2}$ to compensate for this scaling.



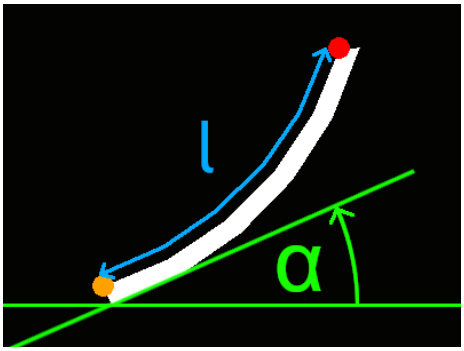
(a) Rectangular platform, $\alpha \in [-\frac{\pi}{2}, \frac{\pi}{2})$



(b) Circular platform, $\alpha = 0$. The generator can use $\bar{\alpha} \in [-\frac{\pi}{2}, \frac{\pi}{2})$ to set the anchor of the circle when connecting to the previous platform (pink dot).



(c) Square platform, $\alpha \in [0, \frac{\pi}{2})$



(d) Ramp platform, $\alpha \in [-\frac{5\pi}{8}, \frac{3\pi}{8})$

Figure 2.4: The meaning of l and α for different platform shapes. The relative anchor and head positions are shown by the orange and red dots respectively.

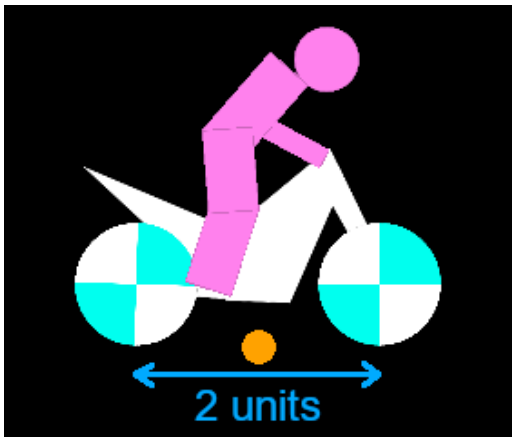


Figure 2.5: Screenshot of the motorbike at rest with a scale reference. Its position $\boldsymbol{b}$ is given by the position of the orange dot. The angle of the bike is $\theta = 0$.

Chapter 3

Content Generation

3.1 Related Work

In recent years there has been a lot of research on procedural content generation (PCG) for video games. The methods usually cover a subgenre of games and range from design tools to complete generators, as shown by an overview of puzzle generation methods [3]. Shaker *et al.* [4] generate a sequence of events through grammatical evolution, called *timeline*. Then the timeline is simulated, i.e. level components are placed whenever they appear in an event such that a solvable level is constructed [4]. Unfortunately, the approach does not suit our game, where there are no discrete events that can easily be mapped to level components. In another approach, Gisslén *et al.* [5] show, that adversarial reinforcement learning can be used for PCG (ARLPCG). One RL agent, called generator, learns how to iteratively construct a level while another, named solver, learns how to solve it. Special generator observations, referred to as auxiliary variables, are connected to its reward function and used to control difficulty as well as very specific behaviors, like airtime in a racing game [5].

3.2 Generator

3.2.1 Actions

Let $m_g \in \mathbb{N}$ be the *generator distance*. To allow the solver and player to react, platforms are requested such that there are always at least m_g platforms in front of the solver. Let p^k be the platform placed at step $k \in \mathbb{N}$ and let $\mathbf{a}^k$ and $\mathbf{h}^k$ be its anchor and head positions respectively (see section 2.3). Platform p^k is considered *passed* when

$$b_x \geq h_x^k \tag{3.1}$$

where $\boldsymbol{b} = (b_x, b_y)^T$ is the position of the bike (Fig. 2.5). Then the generator action for p^{k+1} is requested as soon as the bike passes p^{k-m_g+1} (Fig. 3.2).

The output of the generator has a discrete part

$$\boldsymbol{d} = (d_1, d_2)^T \in \mathbb{S} \times \mathbb{D}$$

where $\mathbb{S}$ as in eq. 2.1 and

$$\mathbb{D} = \{\text{Connect}, \text{Gap}\} \tag{3.2}$$

and a continuous part

$$\boldsymbol{c} = (c_1, c_2, c_3, c_4)^T \in \mathbb{R}^4$$

that are interpreted as follows. The shape of the platform is given by d_1. Let $\beta = 4\text{u}$ be a scaling factor, then the position of the anchor relative to the head of the previous platform i.e. the *offset* is

$$\boldsymbol{\delta} = \mathbf{a}^{k+1} - \mathbf{h}^k = \left\{ \begin{array}{ll} (0,0)^T, & \text{for } d_2 = \text{Connect} \\ \bar{\boldsymbol{\delta}}, & \text{for } d_2 = \text{Gap} \end{array} \right\} \in \mathbb{R}^2$$

where

$$\bar{\boldsymbol{\delta}} = \boldsymbol{\mu}_{\boldsymbol{\delta}} + \beta \cdot \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

and

$$\boldsymbol{\mu}_\delta = \begin{pmatrix} 2u \\ -3u \end{pmatrix} \tag{3.3}$$

is the mean intermediate offset of the heuristic generator (detailed in section 4.2). By choosing $d_2 = \text{Connect}$ the generator can therefore connect platforms seamlessly.
Let

$$\bar{\alpha} = ((\mu_\alpha + c_3 \cdot \frac{\pi}{2} + \frac{\pi}{2} \bmod \pi) - \frac{\pi}{2}) \in [-\frac{\pi}{2}, \frac{\pi}{2})$$

be the *intermediate angle* where

$$\mu_\alpha = 0 \tag{3.4}$$

is the mean intermediate angle of the heuristic generator. The angle then depends on the shape s of the platform according to

$$\alpha = \left\{ \begin{array}{ll} \bar{\alpha}, & \text{for } s = \text{Rectangle} \\ 0, & \text{for } s = \text{Circle} \\ \frac{\bar{\alpha}}{2}, & \text{for } s = \text{Square} \\ 2\bar{\alpha}, & \text{for } s = \text{Ramp} \end{array} \right\}.$$

Since for circles $\bar{\alpha}$ is not used to rotate the platform, we can allow the generator to use it to change where to anchor a circle when connecting it to the previous platform i.e. $d_2 = 0$ (Fig. 2.4b).
Finally, the size of the platform l is determined by

$$l = \max(1.5u, \mu_l + \beta \cdot c_4)$$

where

$$\mu_l = 7u \tag{3.5}$$

is the mean intermediate size of the heuristic generator. We limit the size to $[1.5, \infty)$, because tiny platforms are rather uninteresting.
Note, that the continuous part of the action $\boldsymbol{c}$ determines the normalized offset of the values $\bar{\boldsymbol{\delta}}$, $\bar{\alpha}$ and l from the heuristic mean intermediate values $\boldsymbol{\mu}_\delta$, μ_α and μ_l.

3.2.2 Observations

When the platform p^{k+1} is requested, the generator observation consists of a bike observation $\boldsymbol{o}_b \in \mathbb{R}^4 \times \{0, 1\}$ and platform observations

$$\boldsymbol{o}_p^i \in \mathbb{R}^4 \times \{0, 1, 2, 3\} \quad \text{for } k - m_g + 2 \leq i \leq k$$

i.e. one for each platform in front of the bike. Additionally, it receives one random input in $[0, 1)$. The total observation is then fed to the network as a floating-point vector of size $n_g = 5 + 5 \cdot (m_g - 1) + 1$.

The bike is observed as

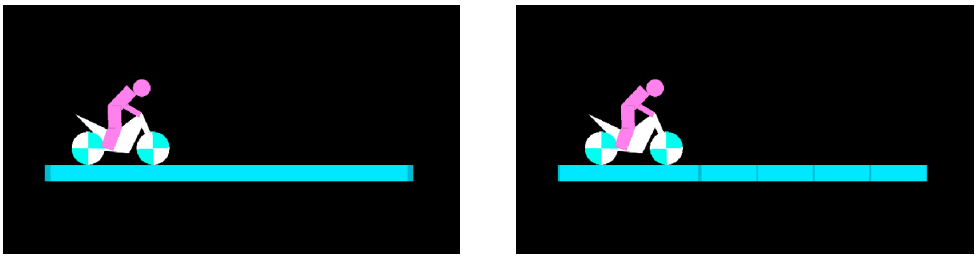
$$\boldsymbol{o}_b = \begin{pmatrix} \frac{v_x}{\beta} \\ \frac{v_y}{\beta} \\ ((\frac{\theta}{\pi} + 1) \bmod 2) - 1 \\ \frac{\omega}{2\pi} \\ \left\{ \begin{array}{ll} 0, & \text{for } q = \text{Back} \\ 1, & \text{for } q = \text{Forward} \end{array} \right\} \end{pmatrix}$$

where $\boldsymbol{v} = (v_x, v_y)^T \in \mathbb{R}^2$ is its linear velocity, $\theta \in [-\pi, \pi)$ is its angle, $\omega \in \mathbb{R}$ is its angular velocity, $q \in \{\text{Back}, \text{Forward}\}$ is the stance of the rider and $\beta = 4\text{u}$ is a scaling factor. Platforms are observed as

$$\boldsymbol{o}_p^i = \begin{pmatrix} \xi_x^i \\ \xi_y^i \\ (\frac{\alpha^i}{\frac{\pi}{2}} \bmod 2) - 1 \\ \frac{l^i}{\beta} \\ \left\{ \begin{array}{ll} 0, & \text{for } s^i = \text{Rectangle} \\ 1, & \text{for } s^i = \text{Circle} \\ 2, & \text{for } s^i = \text{Square} \\ 3, & \text{for } s^i = \text{Ramp} \end{array} \right\} \end{pmatrix}$$

where

$$\boldsymbol{\xi}^i = (\xi_x^i, \xi_y^i)^T = \frac{\boldsymbol{a}^i - \boldsymbol{b}}{\beta}.$$



(a) The appearance of the starting platform in game.

(b) The appearance of the starting platform to the generator and solver.

Figure 3.1: The starting platform in cyan and the bike in the starting position.

The starting platform (Fig. 3.1a) is perceived as split into m_g pieces, such that the generator can also make the observations at the beginning of an episode (Fig. 3.1b).

3.3 Solver

3.3.1 Actions

Every 100ms we request a discrete action

$$\boldsymbol{f} = (f_1, f_2)^T \in \mathbb{G} \times \mathbb{L} \tag{3.6}$$

from the solver where

$$\mathbb{G} = \{\text{Reverse}, \text{Coast}, \text{Gas}\} \tag{3.7}$$

$$\mathbb{L} = \{\text{Lean Back}, \text{Not Leaning}, \text{Lean Forward}\}. \tag{3.8}$$

The solver can therefore control his speed and rotation with f_1 and f_2 respectively (see section 2.2 for details).

3.3.2 Observations

The solvers observations are very similar to the observations of the generator. They consist of bike and platform observations $\boldsymbol{o}_b$ and $\boldsymbol{o}_p^i$ as defined in section 3.2.2. However, the solver does not observe the same platforms as the generator. The bike is considered to be *at* platform p^k if

$$h_x^{k-1} \leq b_x < h_x^k. \tag{3.9}$$

Let p^k be the platform the bike is currently at and let $m_s \in \mathbb{N}_{\leq m_g}$ be the *solver distance*, then the solver observes

$$\boldsymbol{o}_p^i \quad \text{for } k-1 \leq i \leq k+m_s-1 \tag{3.10}$$

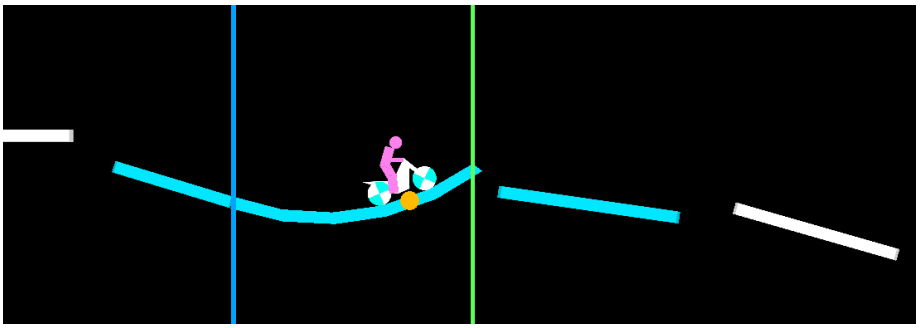


Figure 3.2: Screenshot of a scenario with $m_g = 3$, $m_s = 2$ and 5 platforms. The platforms that are observed by the solver are colored cyan and unobserved platforms are white. Once the bike position i.e. the orange dot passes the green line, a generator action is requested and the platform observations of the solver are shifted forward. Should the bike fall behind the blue line, the observations are shifted back but no generator action is requested.

i.e. it sees the platform behind itself, the platform it is on, as well as $m_s - 1$ platforms ahead. There is an unreachable out-of-sight platform behind the starting platform, such that the solver can also make his observation at the start of an episode. When the bike crosses the head of a platform in positive x-direction i.e. when $b_x \geq h_x^k$, the platform observations are shifted forward and if platform p^{k+m_s} does not exist, a generator action is requested. If the bike leaves the platform in the negative x-direction i.e. if $b_x < h_x^{k-1}$ the solver shifts his platform observations back (Fig. 3.2).

3.3.3 Rewards

At end of step t, the solver receives the reward

$$r_s = \frac{b_x^{\tau+1} - b_x^{\tau}}{\beta} \tag{3.11}$$

i.e. the unscaled progress in the positive x-direction, where $\boldsymbol{b}^{\tau} = (b_x^{\tau}, b_y^{\tau})^T$ is the bikes position at the beginning of step τ. The solver therefore wants to make progress in the positive x-direction as quickly as possible.

3.4 Solvability

Gisslén *et al.* [5] use the success rate of the solver to traverse platforms in *platform game* as a measure of difficulty and control it using auxiliary variables i.e. on a hard difficulty setting, the generator is rewarded when the solver fails. To

prevent impossible levels when setting a hard difficulty, they give a negative reward to the solver when it fails, so that it does not attempt a jump it deems impossible, therefore not failing [5].
We take a different approach to enforce difficulty (see section 3.7), i.e. our generator is not rewarded for making the solver fail. Because an episode ends when the solver crashes or leaves the track and the generator receives non-negative rewards with every step (see sections 3.5 and 3.6), it is directly incentivized it to create traversable platforms. This eliminates the need for a negative solver reward upon failing, which is convenient, because it means we do not have to tune this reward i.e. the courage of the solver. Our solver is therefore never discouraged and learns from every obstacle regardless of whether it deems it possible.

3.5 Variety

To enforce general variety in our levels, we reward the generation of recently unseen level segments. In our game it is possible that unnecessary platforms are generated. We consider a platform unnecessary when the solver passes it without touching it (Fig. 3.3). The variation in these platforms hardly impacts the player experience and should thus not be rewarded.
Let A^i be the event that platform p^i was touched by one or more wheels before p^i was passed for the first time and let p^k be the platform the bike is currently at (eq. 3.9). In the event of A^k, the *transition vector*

$$\boldsymbol{t}^k = (t_1^k, \ldots, t_9^k)^T \in \mathbb{T} \tag{3.12}$$

$$\mathbb{T} = \mathbb{R}^6 \times \mathbb{S}^2 \times \mathbb{D} \tag{3.13}$$

with $\mathbb{S}$ and $\mathbb{D}$ as in (eqs. (2.1) and (3.2)), is associated with the transition that the bike made i.e. the transition from the most recently touched platform

$$p^i : i = \max \{\, j \mid A^j,\ j < k\} \tag{3.14}$$

to p^k, and stored in the *transition buffer* $\mathbb{B}$. The transition vector t^k contains the normalized intermediate parameters of the two platforms

$$t_1^k = \left\{ \begin{array}{ll} 0, & \text{for } s^i = \text{Circle} \\ \frac{\bar{\alpha}^i}{\frac{\pi}{2}}, & \text{else} \end{array} \right\} \tag{3.15}$$

$$t_2^k = \left\{ \begin{array}{ll} 0, & \text{for } s^k = \text{Circle and } d_2^k = \text{Gap} \\ \frac{\bar{\alpha}^k}{\frac{\pi}{2}}, & \text{else} \end{array} \right\} \tag{3.16}$$

$$t_3^k = \frac{l^i}{\beta} \tag{3.17}$$

$$t_4^k = \frac{l^k}{\beta} \tag{3.18}$$

$$t_7^k = s^i \tag{3.19}$$

$$t_8^k = s^k \tag{3.20}$$

and information about how they are connected (Fig. 3.3)

$$(t_5^k, t_6^k)^T = \boldsymbol{\delta^{ik}} = \boldsymbol{a^k} - \boldsymbol{h^i} \tag{3.21}$$

$$t_9^k = \left\{ \begin{array}{ll} \text{Connect}, & \text{if } p^i \text{ and } p^k \text{ overlap} \\ \text{Gap}, & \text{else} \end{array} \right\}. \tag{3.22}$$

If p^i is a circle, $\bar{\alpha}^i$ has no effect on the transition, thus we set $t_1^k = 0$. Similarly, if p^k is a circle and is not connected to p^i, i.e. $i < k-1$ or $d_2^k = \text{Gap}$, the anchor $\boldsymbol{a^k}$ is unaffected by $\bar{\alpha}^k$ and we set $t_2^k = 0$.
To measure the similarity of two transitions we use the metric

$$d : \mathbb{T} \times \mathbb{T} \to [0, 3] \tag{3.23}$$

$$d(\boldsymbol{t}, \boldsymbol{t'}) = \left\{ \begin{array}{ll} d_H(\boldsymbol{t}, \boldsymbol{t'}), & \text{for } d_H(t, t') > 0 \\ d_A(\boldsymbol{t}, \boldsymbol{t'}), & \text{else} \end{array} \right\} \tag{3.24}$$

where the Hamming distance $(t_7, \ldots, t_9)^T$

$$d_H(\boldsymbol{t}, \boldsymbol{t'}) = \sum_{j=7}^{9} 1_{[t_j \neq t'_j]} \tag{3.25}$$

differentiates the types of transition vectors and dominates the asymptotic distance $(t_1, \ldots, t_6)^T$

$$d_A(\boldsymbol{t}, \boldsymbol{t'}) = 1 - \exp(-\sum_{j=1}^{6} (t_j - t'_j)^2) \tag{3.26}$$

which compares transition vectors of equal type. Let n_b be the current size of the buffer, whenever the bike passes a platform p^k for the first time, the generator

receives the *variety reward*

$$r_v^k = \left\{ \begin{array}{ll} 10 \cdot \min\{d(\boldsymbol{t}^k, \boldsymbol{t}) | \boldsymbol{t} \in \mathbb{B}\}, & \text{for } A^k \wedge n_b = n_{b,max} \\ 10, & \text{for } A^k \wedge n_b < n_{b,max} \\ 0, & \text{for } \neg A^k \end{array} \right\}. \tag{3.27}$$

This transition vector t^k is then stored in a buffer $\mathbb{B}$ of maximum size $n_{b,max} = 200$. In case $n_b > n_{b,max}$, old transition vectors are removed from the buffer $\mathbb{B}$ until $n_b = n_{b,max}$ before calculating r_v^k. The transition vector $\boldsymbol{t}^k$ is added to the buffer $\mathbb{B}$ after calculating r_b^k. The buffer has to fill only once. It is **not** emptied at the beginning of the episode or when resuming training. The buffer is also **not** observed by the generator and does not function as memory. It simply incentivizes the generator to use its random input to create different transitions. This reward function has several benefits:

- Placing platforms behind the solver in an attempt to gain instant and infinite reward, will not work, because unless the solver touches the platform the generator does not get a reward
- Unnecessary platforms earn no reward and thus, placing them is disincentivized through reward discounting
- Variations in unnecessary platforms do **not** affect the reward
- Variety in the transitions the solver and player experience are incentivized
- Because there are only

$$n_t = |\mathbb{C}| \cdot |\mathbb{S}|^2 = 2 \cdot 4^2 = 32 < 200 = n_{b,max} \tag{3.28}$$

  transition types, a decent policy should rarely receive $r_v^k \geq 1$. The reward signal is therefore dominated by d_A. We expect the boundary of the set of transitions the solver makes $\mathbb{T}_c$ to contain the most difficult transitions e.g. barely possible jumps or steep uphill sections. To maximize reward, the generator has to create some transitions at the boundary and therefore generate sections where the solver struggles. Note, that it would make sense to impose an upper bound on the normalized sizes of the platforms t_3^k, t_4^k and a lower bound on the normalized offset in y-direction t_6^k, to ensure, that $\mathbb{T}_c$ is closed.

3.6 Momentum

To incentivize the creation of *momentum-based levels* i.e. levels that require the player to keep his momentum across platforms in order to progress, we construct

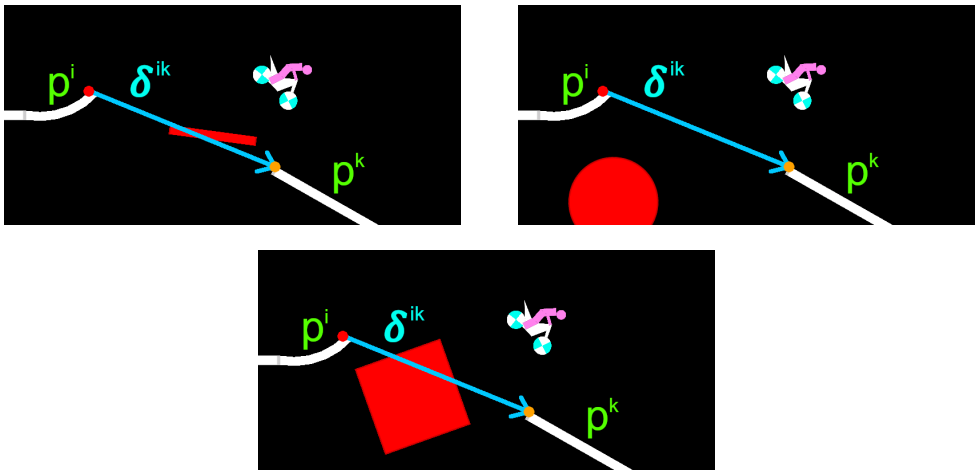


Figure 3.3: Screenshots of three similar level sections where the wheels of the bike touch the white platforms, but do not touch the red platform. The blue arrow visualizes that the reward function treats all situations equally.

a momentum reward. Let p^k be a platform the bike touched and just passed i.e. A^k, and let p^i be the most recently touched platform as in eq. 3.14. With probability p_{ghost}, the generator training enters a procedure called *ghost mode* (Fig. 3.4b)), where the generator has the opportunity to receive an additional reward for p^k. The bike is sent back to the position it had when it passed the center $\boldsymbol{c}^i = (c_x^i, c_y^i)^T \in \mathbb{R}^2$ of platform p^i for the first time i.e. when

$$b_x \geq c_x^i \tag{3.29}$$

and its linear and angular velocity is set to zero. The purpose of this mode is to see if the solver can make it past p^k, starting from p^i, **without** initial momentum. The experiment can end in three ways:

- The solver makes it past p^k (Fig. 3.4c): The generator receives **no** momentum reward.
- The solver crashes the bike (Fig. 3.4d): The generator receives the momentum reward r_m.
- The solver backs up into p^{i-1}: Its reward (eq. 3.11) does not incentivize it to do this unless the transition requires more momentum. The generator receives the momentum reward r_m.

When ghost mode ends, the bike is set back into the position it was in, before entering the mode (Fig. 3.4e) and a platform is requested. The bike also keeps its momentum from before entering the procedure. Note, that giving a reward

to the generator for making the *ghost rider* crash does **not** give an incentive, to create impossible platforms. This is, because ghost mode is not entered unless the rider has already made the transition successfully. The probability p_{ghost} and r_m are hyperparameters. A high value for p_{ghost} slows down training time per step significantly, since it mainly depends on simulation time.

3.7 Difficulty

Because the reward of the generator is tied to the success of the solver, we can force it to make easier levels by sabotaging the solver. We do this in three ways:

- sticky actions - Let $\bar{\boldsymbol{f}}^t$ be the action that is *actually* applied to the game, when the solver action $\boldsymbol{f}^t$ is requested, then

$$\bar{\boldsymbol{f}}^t = \left\{ \begin{array}{ll} \boldsymbol{f}^t, & \text{with probability } 1 - p_{sticky} \\ \bar{\boldsymbol{f}}^{t-1}, & \text{with probability } p_{sticky} \end{array} \right\}. \tag{3.30}$$

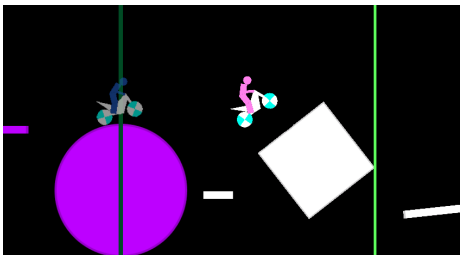
  Notice, that due to the recursive nature of eq. 3.30, one action can get stuck for several time steps. The solver will therefore develop a much more *careful* policy. When the solver makes inference during generator training, actions can still stick.

- solver distance m_s - limiting the amount of platforms the solver is able to see ahead, should make it difficult to act appropriately.

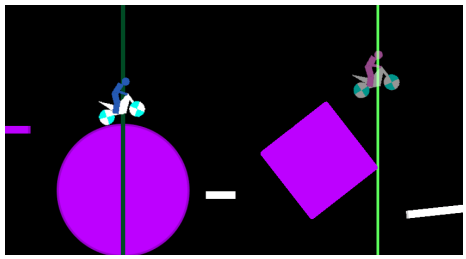
- training steps - shortening the training of the solver and generator, results in a suboptimal solver model and therefore in an easier generator.

3.8 Live Generation

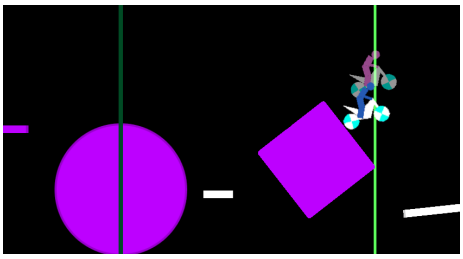
By running inference on the trained solver and generator models, it is possible to generate levels offline. However, we can also simply replace the solver with a player. This is, what allows for online content generation. In *Fast Painting*, we adjust the camera zoom such that the last $m_g = 5$ platforms are fully inside the players vision.



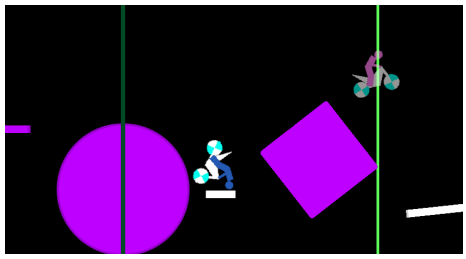
(a) The transparent blue rider on the left marks the saved rider position. If the solver crashes before touching another platform and passing it, ghost mode is **not** entered.



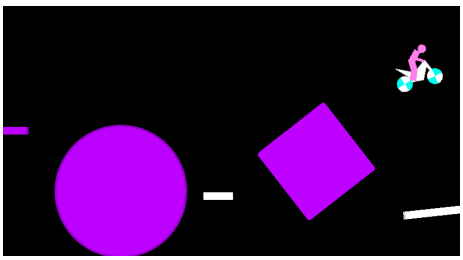
(b) The pink transparent rider on the right marks where the rider position and momentum was saved before the solver takes control of the blue *ghost rider*, which has no initial momentum.



(c) The *ghost rider* reaches the pink riders position, the generator receives **no** additional reward and the solver takes control of the pink rider (Fig. 3.4e) which kept his momentum and a platform is requested.



(d) The *ghost rider* crashes, the generator receives an additional reward and the solver takes control of the pink rider (Fig. 3.4e) which has kept his momentum and a platform is requested.



(e) The ghost disappears.

Figure 3.4: Step-by-step example of the *ghost mode* procedure. Platforms that were touched by a wheel are colored purple. Transparent bikes mark positions the bike might return to. The color of the rider indicates whether we are in ghost mode. Pink means regular training, blue means ghost mode.

Chapter 4

Training

This chapter highlights the training setup and technical difficulties with *Unity ML-Agents* [6].

4.1 The Algorithm

To train the generator and solver policies, we use Proximal Policy Optimization (PPO) [7]. Since the generator reward for placing a platform p^k depends on whether the player touches it (see section 3.5) and sometimes on the whether the ghost makes the transition to it (see section 3.6), we cannot give the reward to the generator in the same step k. Instead, the reward for platform p^k is paid in step $k + m_g - 1$. Because the solver acts on a much higher frequency, we set the reward discount factor γ to a higher value for it, than for the generator. We only experimented little with other hyperparameters. Table 4.1 lists all of them. We alternate between training the solver and generator every $3 \cdot 10^5$ steps and start by training the solver on a heuristic generator (see section 4.2). While training one model, we run inference on the latest version of the other model. Due to the time limit of this project we never trained for so long that the rewards no longer increased i.e. an optimum was reached. Still, our generator produces solvable content because its solvability does not rely on convergence as with the setup by Gisslén *et al.* [5]. The solver only needs to touch and make it past a few generated platforms in an episode before crashing in order to give the generator a reward signal.

4.2 Heuristic Generator

The heuristic generator places platforms according to

$$\begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \frac{1}{\beta} \begin{pmatrix} \max(-1u, \Delta\delta_x) \\ \max(-2u, \Delta\delta_y) \end{pmatrix} \tag{4.1}$$

	Generator	Solver
Buffer Size	2048	2048
Batch Size	512	32
Learning Rate Schedule	Linear	Linear
Learning Rate	$3 \cdot 10^{-4}$	$5 \cdot 10^{-4}$
Epochs	3	3
Layers × Units	3 × 128	3 × 256
Time Horizon	32	128
γ	0.9	0.99
λ	0.95	0.95
β	$5 \cdot 10^{-3}$	$5 \cdot 10^{-2}$
ϵ	0.2	0.2
Normalize Observations	No	No

Table 4.1: Hyperparameters used with the *Unity ML-Agents PPO Trainer* [6].

$$c_3 = \frac{\Delta\alpha}{\frac{\pi}{2}} \tag{4.2}$$

$$c_4 = \frac{\Delta l}{\beta} \tag{4.3}$$

where

$$\boldsymbol{\Delta\delta} \sim \mathcal{N}(0, I_2 \cdot u^2) \tag{4.4}$$

$$\Delta\alpha \sim \mathcal{N}\left(0, \left(\frac{\pi}{12}\right)^2\right) \tag{4.5}$$

$$\Delta l \sim \mathcal{N}(0, (2u)^2) \tag{4.6}$$

and

$$d_1 = \left\{ \begin{array}{ll} \text{Rectangle,} & \text{with probability } 0.7 \\ \text{Circle,} & \text{with probability } 0.1 \\ \text{Square,} & \text{with probability } 0.1 \\ \text{Ramp,} & \text{with probability } 0.1 \end{array} \right\} \tag{4.7}$$

$$d_2 = \left\{ \begin{array}{ll} \text{Connect,} & \text{with probability } 0.5 \\ \text{Gap,} & \text{with probability } 0.5 \end{array} \right\}. \tag{4.8}$$

The discrete and continuous parts $\boldsymbol{d}$ and $\boldsymbol{c}$ are interpreted as a regular generator action (see 3.2.1). The heuristic generator is supposed to generate easy levels that the initial solver can train on. Thus, the variance in the platform parameters is small. Admittedly, the clipping of $\boldsymbol{\Delta\delta}$ in eq. 4.1 is overcomplicated and unnecessary.

Chapter 5

Results

5.1 Experiments

To evaluate the impact of the rewards and the sabotaging of the solver, we look at 20 training runs.

In runs 1-10, we do **not** apply the ghost mode procedure described in section 3.6. In runs 11-20, we do give the momentum reward. Across the 10 runs within a batch, we try to achieve easier levels by sabotaging the solver as detailed in section 3.7. Tables 5.1 and 5.2 show the exact hyperparameters used. We will refer to models with a higher run number as *better* and *harder* for solvers and generators respectively, not because they are guaranteed to perform better, but because they have an advantage. Models with a lower run number will be referred to as *worse* and *easier*.

For runs 1-10 we see, that for the generator the episode length and the average step reward increase with training and are generally higher for the *harder* solver generator pairs (Figs. 5.1a and 5.1b). Because the generators only receives the variety reward r_v in these runs, it must be making progress towards the solvability and variety objectives. The fact that the reward is higher for the *better* solver generator pairs indicates, that forcing the generator to make easy levels, makes it harder to create variety, which is expected. Run 2 and 5 however do not follow this trend i.e. have significantly higher step reward than similar runs.

For runs 11-20 the results are similar (Figs. 5.1c and 5.1d). However, the momentum reward causes oscillation in training, especially at the start. Apparently randomly, generator 17 receives significantly higher step reward.

When putting the solvers up against different generators, we would expect, that *better* solvers outperform *worse* solvers on the same generator and themselves on a *harder* generator i.e. receiving more cumulative reward. To evaluate this, we put 4 solvers up against 4 different generators (Fig. 5.2).

For runs 1-10 our expectations are unfortunately not met (Fig. 5.2a). Although, solvers 4,7 and 10 perceive generator 10 has the most difficult, solver 10 is the **only** solver who perceives the generator difficulties in the expected order.

Ironically, Solver 1 does better against generator 10 than any other solver, despite having no training experience with it. This is likely due to the shortsighted solvers having a smaller observation space and thus learning much faster, which is a significant advantage, that is not compensated enough by the reduction in training steps.

Surprisingly, for runs 11-20, the results are less random (Fig. 5.2b). The reached distances are a lot smaller for the *harder* generators, which hints at more difficult tracks. For solvers 11 and 14, the generators difficulties are ranked as expected. The difference in the average steps between generators 11 and 14 is however much bigger than between generators 14 and 17. Furthermore, solvers 11, 14 and 17 perform very similar against generator 11, while solver 20 does extremely poor. Surprisingly again, the average horizontal bike velocity v_x is greater in the runs **without** momentum reward.

The variance in the distance un runs 1-10 is **extremely** high (Fig. 5.2a) and 200 samples are probably not enough to get a good estimate of the mean. We cannot properly evaluate the difficulty of the content using this data. In Runs 11-20 the variance is smaller (Fig. 5.2b). The additional training or the momentum reward or both, must be the cause of the less random results.

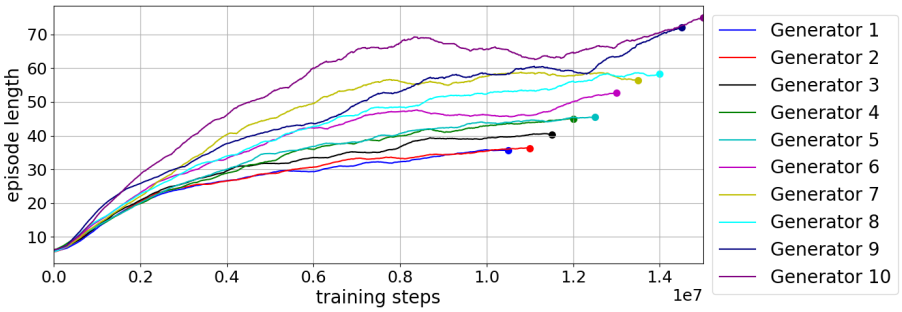
An episode is reset if no platforms are requested for 15 seconds. When the rider falls on his back or gets stuck without hitting his head, we get a very poor estimate of the average horizontal bike velocity v_x. Its relation to the momentum reward are therefore also unclear.

	1	2	3	4	5	6	7	8	9	10
p_{sticky}	0.4	0.36	0.32	0.28	0.24	0.20	0.16	0.12	0.08	0.04
m_s	1	1	2	2	3	3	4	4	5	5
steps/10^7	1.05	1.1	1.15	1.2	1.25	1.3	1.35	1.4	1.45	1.5
p_{ghost}	0	0	0	0	0	0	0	0	0	0
r_m	0	0	0	0	0	0	0	0	0	0

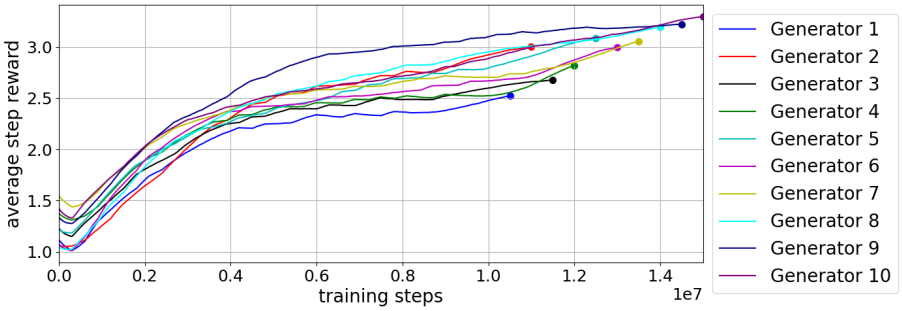
Table 5.1: Momentum and difficulty hyperparameters in runs 1-10.

	11	12	13	14	15	16	17	18	19	20
p_{sticky}	0.4	0.36	0.32	0.28	0.24	0.20	0.16	0.12	0.08	0.04
m_s	1	1	2	2	3	3	4	4	5	5
steps/10^7	1.55	1.6	1.65	1.7	1.75	1.8	1.85	1.9	1.95	2
p_{ghost}	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
r_m	5.5	3.6	3.03	2.8	2.7	2.67	2.67	2.7	2.74	2.8

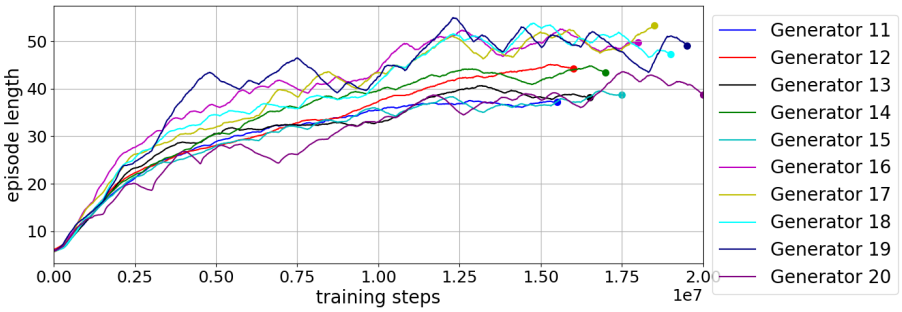
Table 5.2: Momentum and difficulty hyperparameters in runs 11-20.



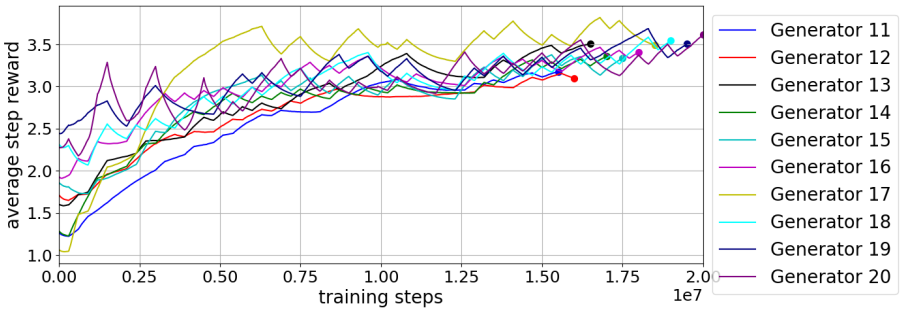
(a) The average generator episode length in runs 1-10.



(b) The average generator step reward in runs 1-10.

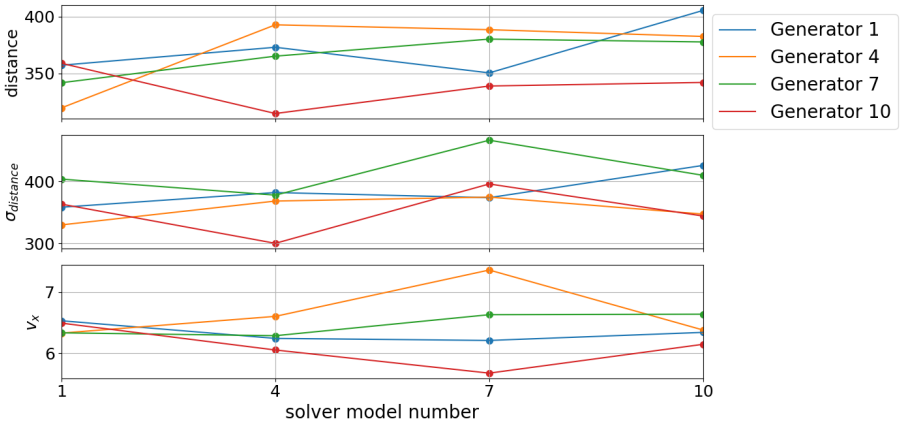


(c) The average generator episode length in runs 11-20.

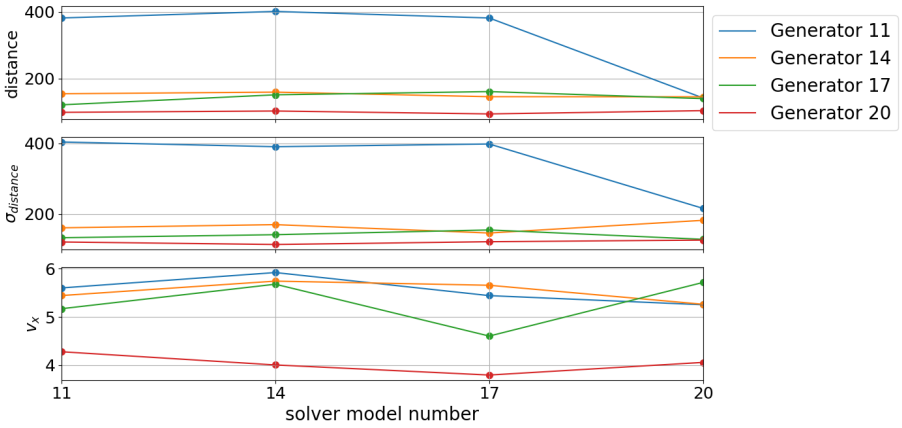


(d) The average generator step reward in runs 11-20.

Figure 5.1: The number of training steps increases with the run number. The values are exponentially smoothed with a halftime of $6.9 \cdot 10^5$ steps.



(a) Runs 1-10



(b) Runs 11-20

Figure 5.2: Distance reached by the solver and the horizontal bike velocity v_x averaged over 200 episodes, and the standard deviation of the distance $\sigma_{distance}$ for 16 different solver generator pairs.

5.2 Conclusion

The generator starts producing reliably solvable levels very early on during training. After that, the variety in the created levels increases slowly. The generator therefore successfully learns the solvability and variety objectives. From inspection of the content, it is very obvious however, that the generator overfits to the starting platform i.e. the first few platforms are always very similar and then they become more and more varied.

Our approach to generating content of different difficulty, had little success. There is no strong pattern in the performance of the solver models on the generated content. Still there is a noticeable difference in difficulty when switching from the easiest to the hardest generator. The generators which received momentum reward and trained longer, appear to generally produce more interesting and harder levels. Unfortunately, the overall difficulty range is quite small. The easiest generator should be easier and the hardest one should be more difficult. Also, the difficult generators do **not** exclusively place hard platforms.

More training is probably needed to see the difficulty categories of the generators diverge more drastically and to see the *better* solvers outperform the shortsighted ones. An additional reward to incentivize placing hard level components exclusively, could help make the generators' difficulty level more consistent.

The generators seem to behave the same when generating live, which means overfitting to a solver model is not an issue.

5.3 Future Work

Due to the setup used during this project (see section A.1), it was not possible to manipulate the experience buffer of agents after the fact. However, it might be a powerful tool to speed up the increase in variety reward. Unnecessary platforms as in Fig. 3.3 and their respective generator actions could be merged, such that the generator quickly learns to omit them.

To make even easier levels, one could try sabotaging the solver even more and in other ways.

One could also do something similar to our ghost mode procedure where different solvers are put in the same situation and a reward is given to the generator if only some solvers make the transition and others do not. This could help teach the generator to exclusively place hard level components.

Bibliography

[1] DarkMoon_, "Trials fusion - crown control (ninja level 8)," https://www.youtube.com/watch?v=Kz3DZ0Skv0Y, accessed: 07.07.2021.

[2] C. Trials, "Trials fusion - inferno iv wr," https://www.youtube.com/watch?v=Ga_esJO1344, accessed: 24.07.2021.

[3] B. De Kegel and M. Haahr, "Procedural puzzle generation: A survey," *IEEE Transactions on Games*, vol. 12, no. 1, pp. 21–40, 2020.

[4] M. Shaker, N. Shaker, J. Togelius, and M. Abou-Zleikha, "Aalborg universitet a progressive approach to content generation," 2015.

[5] L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar, "Adversarial reinforcement learning for procedural content generation," 2021.

[6] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," 2020.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[8] M. L. Littman and C. Szepesva´ri, "A generalized reinforcement-learning model: Convergence and applications," USA, Tech. Rep., 1996.

Appendix A

Using ML-Agents

A.1 Technical Difficulties

To train the solver and generator in an *alternating Markov game* [8], we need to run inference on one policy, while updating the other. Using the *Low-Level Python API for Unity ML-Agents*, it is possible to connect the Unity environment i.e. the game, to a custom training loop. We can then run the Unity environment together with the training loop on the cluster and the switch in the updating policy occurs automatically (Fig. A.1). However, there are two downsides to this approach:

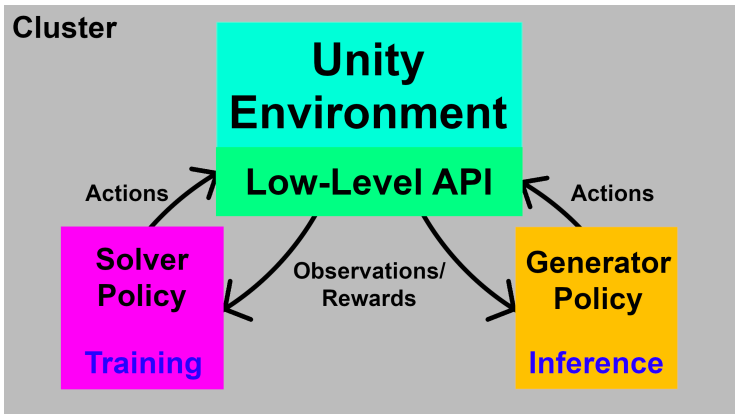
- Setting up the low-level API with a reinforcement learning library and writing a custom loop is rather big undertaking
- The trained network cannot easily be converted into one, that can be integrated into the game, and it therefore needs an external python process that supplies generator actions

For these reasons we decided to use the command line interface

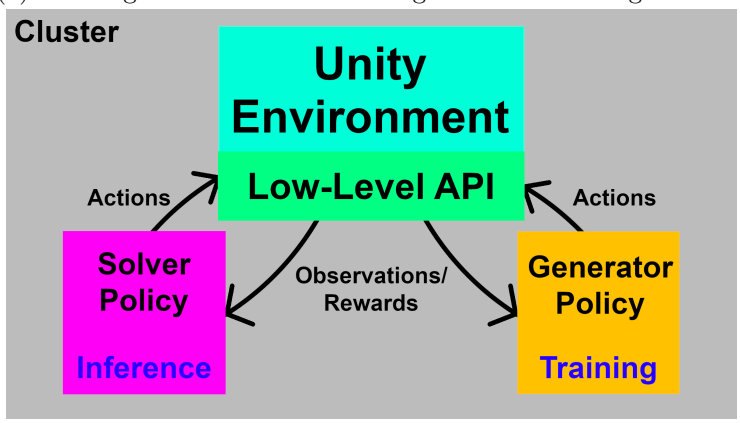
```
mlagents-learn
```

provided by the *ML-Agents Toolkit* and the PPO algorithm that comes with it [6]. Getting started with `mlagents-learn` is very straight-forward. However, our asymmetric adversarial setup creates the following challenges:

- Because `mlagents-learn` does not allow us to run inference on one behavior while training another, we have to integrate the updated model into the Unity environment when switching trainees, which requires rebuilding the environment.
- Rebuilding the environment is not possible on the cluster, because the *Unity Editor* does not support Debian Linux.



(a) Training the solver while running inference on the generator.



(b) Training the generator while running inference on the solver.

Figure A.1: An alternative training setup. The *Unity ML-Agents Low-Level Python API* accepts actions and returns observations and rewards for all behaviors in the Unity environment [6]. Switching between training the solver and the generator, can be done in a custom training loop directly on the cluster.

- Building the environments can be automated on supported operating systems, but only one instance of the *Unity Editor* can run on any machine at once.

To work around this issue, I run multiple *build servers*, that supply the cluster processes with new environments whenever the trainee is switched (Fig. A.2). The servers run on systems that support the *Unity Editor*.

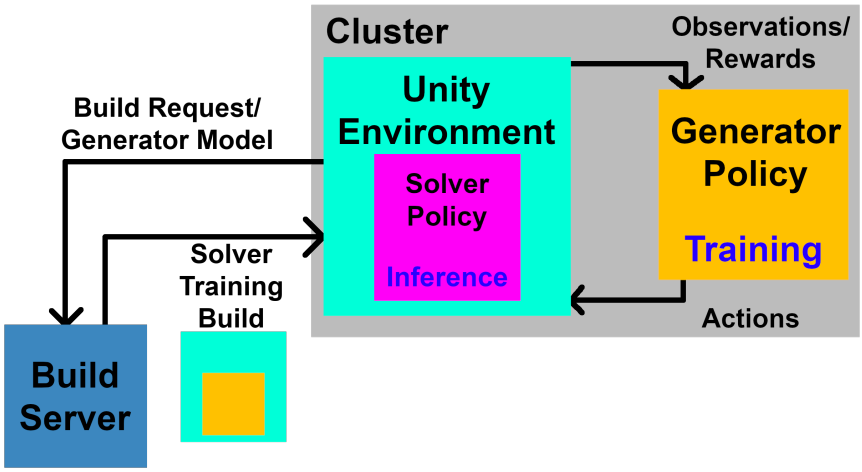


Figure A.2: The trainee is switched from generator to solver. The *build server* downloads the updated generator model and integrates it into a new environment build that the solver can continue training in.