Racing Track Levels as Procedural Content Generation using Unity3D Engine

Julia Alejandra Rodriguez-Abud

Introduction to Machine Learning

***Abstract*—This document describes the project developed for the 2021 Cinvestav Guadalajara Introduction to Machine Learning class given by Dr. Andres Mendez Vazquez. This project explored a practical implementation of machine learning techniques for the use in the videogame industry. More specifically exploring the use of ML-Agents for Procedural content generation (PCG) of levels. This document describes the routes taken and decisions made to be able to attend to the PCG objective as well as the things learned while developíng the project and the possible future approaches given the limitations discovered in the process.**

***Index Terms*—Machine Learning, Unity3D, Procedural Content Generation, Game Design**

I. Introduction

PROCEDURAL Content Generation is an important part of the video game industry, as it allows for improvement and optimization in the production pipeline. Manually creating content is an expensive and time consuming task that can be minimized to allow the art team to focus on other parts of the creative process and to let the computer create some of the assets.

Many Machine Learning techniques require previously accumulated data to use as priors and learn based on this information but because of the nature of video games, especially small indie productions we need to make sure to generate content on the go while in the development stage when there are not many created assets to work with. When a game is in its beginning stages there may not be a lot of content to learn from because it has not been made yet. But one of the prime reasons to want to train a content generator is in fact to not have to produce the whole content.[1]Many games use content generation for this purpose, some of them use it to create whole worlds such as No Man's Sky[2] and Minecreaft [3], others to create dungeons and levels such as The Binding of Isaac[4] and Spelunky[5], and others even use it to create assets such as the weapons in the Borderlands series[6].

The core dilemma to always keep in mind is that the Machine Learning agents with the purpose of generating procedural content need to be allowed to create the content by itself or with minumum help as early as possible in the video game production process.

For this project the plan is to use Machine Learning in the Unity3D Engine to help in the level design process of a **car racing game**. The generated racing tracks would need to be playable and have a proper 3D mesh with its collider. They also need to consider the functionality of these types of games: it should fully complete a lap and close within itself, should not be extremely short and should have a well defined main route.

The files for this project can be found in the following GitHub repository :

https://github.com/JuliaAbud/RacingTrackGenerator

Fig. 1. Unity's Training environment

A. Unity3D Engine

Unity is a cross-platform game engine used by a large community of game developers to create a variety of projects such as interactive simulations, from small mobile games to AAA high-budget console games and VR/AR experiences. Besides its traditional gaming background it has also found acceptance and practical use in the film, auto, and AEC (Architecture, Engineering, Construction) industries. Unity is a real-time 3D development platform that consists of a rendering and physics engine as well as a graphical user interface called the Unity Editor. [7]

B. ML-Agents Toolkit

The Unity Machine Learning Agents Toolkit [7] is an open-source project that enables the creation of environments for training intelligent agents to serve in the development of games and simulations. The Unity package ML-Agents allows the use of Machine Learning techniques such as Reinforncement Learning or Imitation Learning. For reinforncement learning PPO (Proximal Policy Optimization) is used and for the immitation learning BC (Behavioral Cloning) is used. ML-Agents allows the use of a learning environment that connects and communicates to a simple Python API. (Figure 1) The output of the trainings done in the Python API is an ONNX (Open Neural Network Exchange) file that can be used within the Unity engine as a brain to be used by the agents. (Figure 2)

Julia Alejandra Rodriguez-Abud is with the Department of Computer Science, Cinvestav, Guadalajara, México, e-mail: julia.rodriguez@cinvestav.mx.

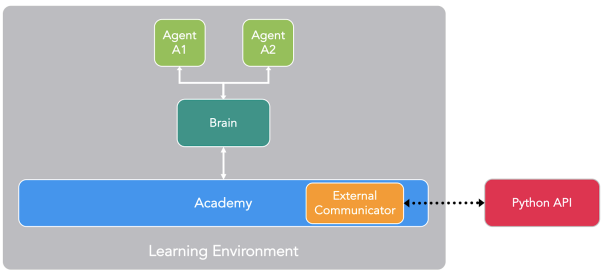

Fig. 2. Unity's Learning environment description

II. Suggested routes

To solve the objective of generating functional racing track levels, there are several constraints that needed to be considered for them to be used in an actual racing game.

In this section we will describe the various methods that may have been used to solve the racetrack generation and the decision process towards the route that was taken. This is important as it is the base to define the data that would be collected and fed to our ML Agent as input.

A. Assembled pieces vs Bezier curve

The first decision to make was how the path creation method would be handled, in other words the way it would be made even if it were done by a human. There were two routes explored for this: Assembled pieces and Generate a 3D mesh from a bezier curve.

The **assembled pieces** pipeline proposes having several n prefabs (prefabricated gameObjects that include mesh, the material with textures, the colliders, etc) that can be assembled together to form a pathway. We would have setups for all the possible turns, such as: forward, small turn, big turn, slope, etc. (Figure 3) In total we would have $4n$ possible setups for each type of tile, as for each piece we have four possible rotations: $0°, 90°, 180°$ and $270°$ degrees.

The original idea for this (not implemented) process was to have a matrix filled with integers that represent what element is going to be set in that space of the grid. The code 0 would be used to represent an empty space in the grid and the numbers from 1 to $4n$ for each one of the different setups (type of piece + rotation). Having these rules set, we could have filled the matrix with the help of a cellular automata and finally, use that data to assemble the tracks in a grid. The problem with this approach was that for it to be easily implemented, all the pieces would need to have a predetermined size for them to fit in their designated space in the grid.

Generating a 3D mesh from a **bezier curve** is a better solution that allows more freedom for the track design and route. A bezier curve is defined by a set of control points $P0$ through Pn, so the complete path is implemented with a list of Vector3's describing the control points positions that will form our curve. (Figure 4) This way we can make a loop out of the Vector3 list by connecting the last Pn and the first $P0$ control point.

Knowing that this is the data structure that would hold our racing track pathway we can develop an algorithm to generate

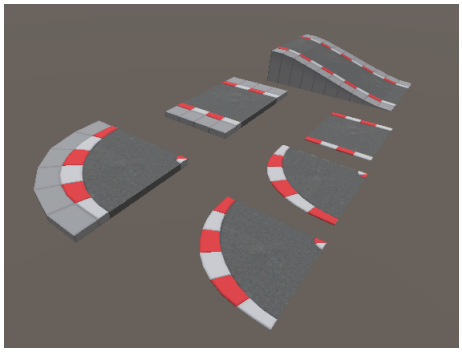

Fig. 3. Assembled pieces

n positions and then use a tool to extrapolate the 3D mesh so it can be rendered with a material and it can be used as a collider for it to be usable as a Racing Track floor. This is the process that appeared to give a nicer result, so it was the one explored with our Generator in Section 3.

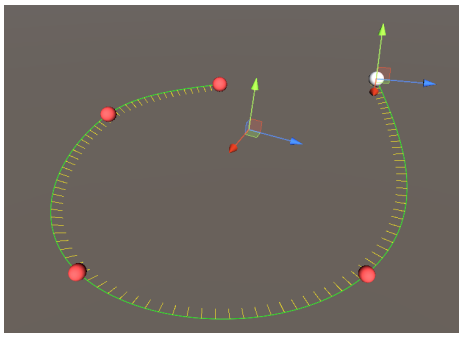

Fig. 4. Bezier curve tool. Consists of a series of control points positions described as Vector3 xyz floats.

B. XYZ-space vs XZ-space

Another decision to be made was the use of the 3D space, as there is the possibility to actually use the three axis X,Y and Z because we are working on a three dimensional space with our control points positions. Creating the racing track in a XYZ-space gives a far more interesting result as it can create more complex routes that go up and down, but at the same time it becomes a harder problem to define properly (Figure 5) as there are many more variables to consider as what qualifies as a "valid" racing track in the classification process. The full use of the 3D space would also add the complication of maknig sure the normals feel natural so when extrapolating the the 3D mesh it comes out correctly creating a shape that actually resembles a racing track and not a roller coaster.

The XZ-space is more simple, and is good enough for our objective as it can still be fully playable and easily set in the ground. The main constraint to focus on is the avoidance of overlapping sections in the racetrack path, which can be easily done with some computational geometry concepts as described in Section 3 in the "discriminator". The decision made was to use the XZ-space .

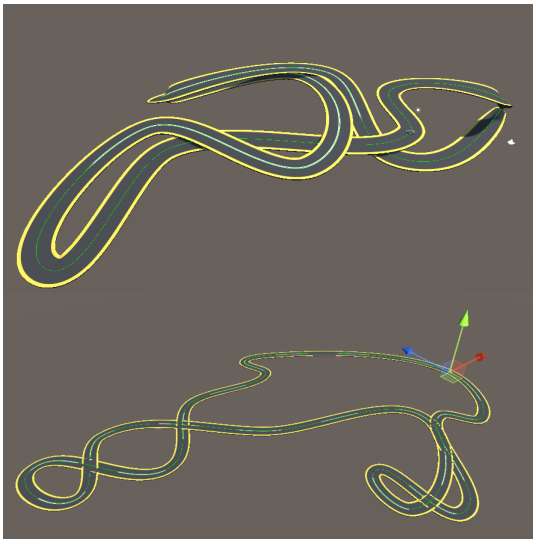

Fig. 5. 1) XYZ- space generated track. 2) XZ-space generated track.

C. Generative Adversarial Networks

The starting idea was to use Generative Adversarial Networks (GAN) to generate the racing tracks by developing a Generator and a Discriminator to keep improving the generation process. As [1] suggests, the PCG of 3D map levels and 2D map level generation in Machine Learning is usually approached with an Adversarial Learning . Altough some of the examples mentioned like a Doom level generator [8] seem to have been made with a research purpose and not used in a fully released video game.

A GAN consists of two Neural Networks competing with each other. The Generator G trained to minimize $log(1 - D(G(z)))$and a Discriminator D trained to maximize the probability of assigning a correct label. (Figure 6) D and G play the following two-player minimax game with value function $V(G, D)$: [9]

$$\min \max(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))]$$

Altough because of the nature of the GAN structure and our limitations with the ML-Agents algorithms, it was decided to not use GANs, and just take the generative nature of it for a Neural Network and for it to be trained with a "discriminator algorithm" that is fully predefined.

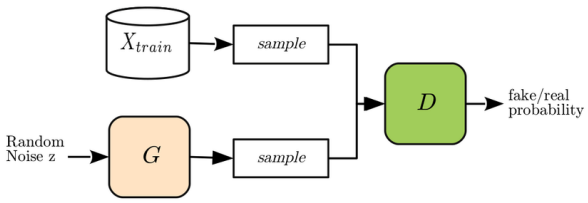

Fig. 6. Generative Adversarial Network structure showing the Generator G that works with some noise z, and the Discriminator D that work with the input of the prior data Xtrain and the generated data G(z) [10]

III. Implementation

Exploring the Procedural Content generation (PCG) of levels using ML-Agents in Unity limits the problem solving to the use of reinfoncement and/or imitation learning. And because of this limitations we will adapt the general idea and using a Generative Neural Network trained with Reinforcement Learning and a Discriminator algorithm designed for this problem (not a NN). By doing this we are no longer taking the Adversarial Learning route.

"Reinforcement learning is commonly used to learn to play games, which makes sense as the problem of playing a game can easily be cast as a reinforcement learning problem; the action space is simply the actions available to the agent, and most games have a score or similar which can be used to provide a reward signal. In contrast, problems of designing games or game content are most of- ten cast as optimization processes, where a measure of quality is used as an objective function, or sometimes as supervised learning problems" [11]

ML-Agents framework uses a reinforcement learning technique called Proximal Policy Optimization (PPO). PPO uses a neural network to approximate the ideal function that maps an agent's observations to the best action an agent can take in a given state.

A. Generator

The technique used to generate our racetrack is by generating the control points in the bezier curve to then extrapolate a mesh to be used to render the path and create the collider.

The data that would be used to train our generator will be a list of Vector 3 points that provide the positions.

1) Control points generation: The bezier control points are defined based on the last point generated x and z. The y pos will be set up to be on 0, as right now we are considering for our track to be on a plane ground.

max max distance from the last generated point
$prevP$ keeps track of the last Vector3 position
$newP = prevP + (rnd(-max, max), 0, rnd(-max, max))$

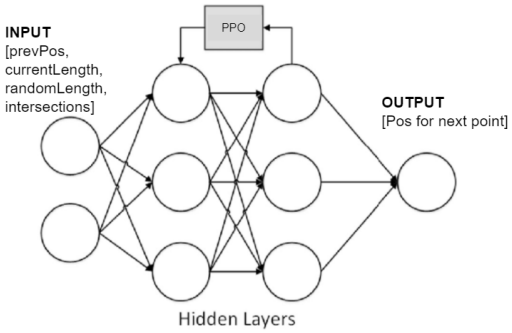

Fig. 7. Neural network proposed for the Generator. Its a PPO that receives 6 floats as input : prevPos.x, prevPos.y, prevPos.z, currentLength, randomLength and Intersections

2) Description of the Neural Network: A Proximal Policy Optimization using a Neural network that consists of a vector with 6 floats as inputs as described in (Figure 7) and an output with 3 floats that will help to position the next control point. This NN is trained with PPO and has 2 layers.

3) Description of the training environment: The training environment used had the following hyperparameters (Table I). And the PPO Reinforcement Learning structure is described with the figure (Figure 8) and the following information:

- **Set-up:** At the start of each episode we start with a point in Vector3 zero, and start gradually adding up new controls in random positions relative to the last point generated.
- **Goal:** Generate a racing track that is playable by not intersecting within itself
- **Agents:** The environment contains one agent.
- **Agent Reward Function:** (minimum of -1, and maximum of +1.8)
  - +0.1 when generating a new control point (more than 2 and less than 10) Adding of a bezier control point, causes to give a small reward (This motivates it to have more points and shapes)
  - +1.0 accepted final track
  - -1.0 rejected final track
- **Behavior Parameters:**
  - Vector Observation space: Vector3 variable corresponding to prevPos control position, Integers corresponding to randomLength currentLength and Intersections
  - Actions: random generation of 3 normalized numbers that will be used for defining the next position

TABLE I
HYPERPARAMETERS USED FOR THE TRAINING OF THE GENERATIVE MODEL TRACKGENERATOR

trainer_type: ppo	
hyperparameters:	batch_size: 1024
	buffer_size: 10240
	learning_rate: 0.0003
	beta: 0.005
	epsilon: 0.2
	lambd: 0.95
	num_epoch: 3
	learning_rate_schedule: linear
network_settings:	normalize: False
	hidden_units: 128
	num_layers: 2
	vis_encode_type: simple
	memory: None
	goal_conditioning_type: hyper
reward_signals: extrinsic:	gamma: 0.99
	strength: 1.0
	normalize: False
reward_signals: network_settings:	hidden_units: 128
	num_layers: 2
	vis_encode_type: simple
	memory: None
	goal_conditioning_type: hyper
init_path: None	
keep_checkpoints: 5	
checkpoint_interval: 500000	
max_steps: 500000	
time_horizon: 64	
summary_freq: 50000	
threaded: False	
self_play: None	
behavioral_cloning: None	

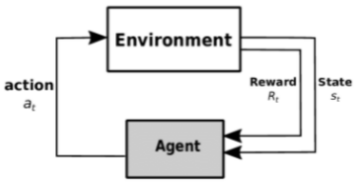

Fig. 8. Reinforcement Learning

B. Manual "Discriminator" Test

The objective of our discriminator is to classify if our racing track is a "valid one". But before trying out to do an automatic training, a manual classification by the user was implemented using **InputKey.Q** for accepting the track and **InputKey.W** for rejecting the track.

The objective was to quickly proove if the route taken with our generator was something feasible, and verify with a few trials if our Generator Neural Network is actually improving before implementing an algorithm to do the automatic decision.

This test was done for around 30 minutes and with approximately 1500 episodes. This tiny result showed that this route was worth pursuing. (Figure 10)

Fig. 9. Aerial view with B/W camera to facilitate the manual process of visualizing if there are any intersections

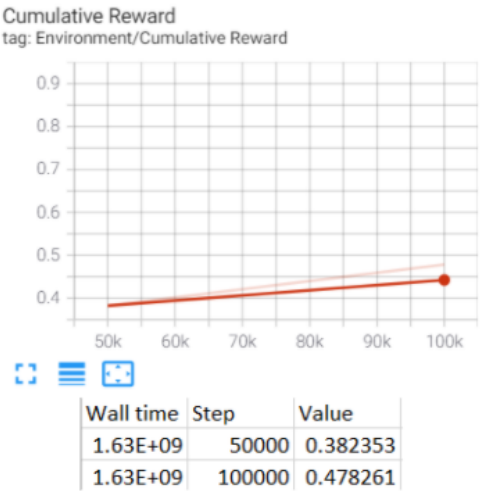

Fig. 10. Results of the manual classification

C. Automatic "Discriminator" Algorithm

This algorithm has the objective of defining if the given path doesn't have any intersections within itself, this nonintersecting polygon is considered a **Simple closed path.** For this we verify if there exists any line segment intersection given every pair of segments that form our path. The line segment intersection problem is one of the most fundamental problems in computational geometry. [12]

The approach for this algorithm is based in the notion of orientation of an ordered triplet of points in the plane. Two segments (a,b) and (c,d) intersect only if in the general case both ordered triplets have different orientations or in the special case all triplets are colinear and one of the points lies exactly on the other segment.

And with the function that checks if there is an intersection for any two given segments, we implemented a small function that counts the amount of intersections in the given path comparing once each pair of segments. The amount of intersections is an integer that is also fed to our neural network.

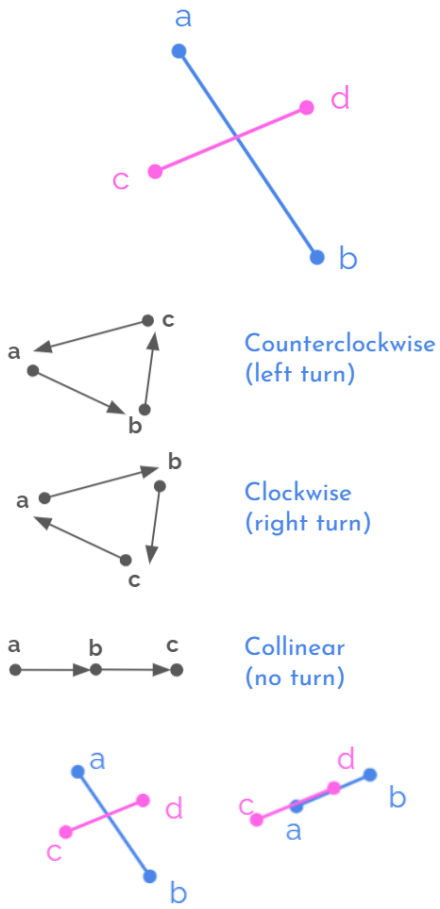

Fig. 11. **Top)** Line Segment Intersection. **Center)** The **orientation** of an ordered triplet of points in the plane can be: counterclockwise, clockwise or collinear **Bottom)** General and special cases of segment intersection

Algorithm 1 OnSegment

Require: Points p,q,r
Ensure: Given three colinear points p, q, r
the function checks if point q lies on line segment 'pr'

```
if ( q.x ≤ max(p.x, r.x) and q.x ≥
min(p.x, r.x) and q.y ≤
max(p.y, r.y) and q.y ≥ min(p.y, r.y) )
  return true
else
  return false
```

Algorithm 2 Orientation

Require: Points p,q,r
Ensure: To find orientation of ordered triplet (p, q, r)

```
val = (q.y − p.y) * (r.x − q.x) − (q.x − p.x) *
(r.y − q.y)
if (val == 0)
  return 0 // colinear
elif (val > 0)
  return 1 // clock wise
else
  return 2 // counterclock wise
```

Algorithm 3 DoIntersect

Require: Points p1 and q1 (First segment) and points p2 and q2 (Second segment)
Ensure: Return if there is an intersection between the two given segments

```
// Find the four orientations needed for
// general and special cases
o1 = orientation(p1, q1, p2)
o2 = orientation(p1, q1, q2)
o3 = orientation(p2, q2, p1)
o4 = orientation(p2, q2, q1)
// General case
if (o1 != o2 and o3 != o4)
  return true
// Special Cases
// colinear triplet and point lies on segment
if (o1 == 0 and onSegment(p1, p2, q1))
  return true
if (o2 == 0 and onSegment(p1, q2, q1))
  return true
if (o3 == 0 and onSegment(p2, p1, q2))
  return true
if (o4 == 0 and onSegment(p2, q1, q2))
  return true
// Doesn't fall in any of the above cases
return false
```

IV. Results

After letting the training of the NN run for the generation of over 35k racing tracks, (Table II) it was shown that at that point most of the tracks where **nonintersection paths** that were accepted by our discrimination algorithm.

The quality of the tracks generated was improved as showed by the cumulative mean reward over time, as it went from 0.585 to a 1.730 (Figure 12 and Table III).

The cumulative mean reward has a maximum reward value of 1.9 to be gained by a track; this is without any intersections and with more than 10 control points (this was favoured in the design of the training environment with the implementation of small rewards per control point as described in Section 3)

TABLE II
ACCUMULATIVE RESULTS OF GENERATED TRACKS

Total tracks generated	35267
Accepted tracks	29752
Rejected tracks	5514

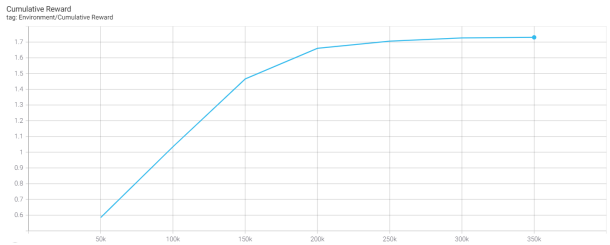

Fig. 12. Cumulative reward during the training session. Being 0.585 at the first 5k steps and finishing in 1.730 at 35k steps.

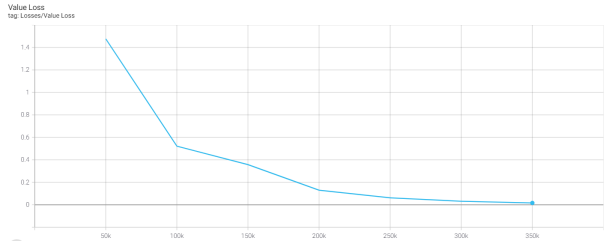

Fig. 13. Value loss over time

TABLE III
TRACKGENERATOR.ONNX TRAINING DATA

Step	Time elapsed	Mean reward	Standard Deviation of reward
5000	594.235 s	0.585	0.924
10000	1252.818 s	1.035	0.921
15000	1905.642 s	1.466	0.695
20000	2566.427 s	1.660	0.447
25000	3234.126 s	1.706	0.348
30000	3920.747 s	1.726	0.261
35000	4610.893 s	1.730	0.263

V. Conclusions

We have proven that the Reinforncement lraning process can be used to teach a NN with satisfiying results creating

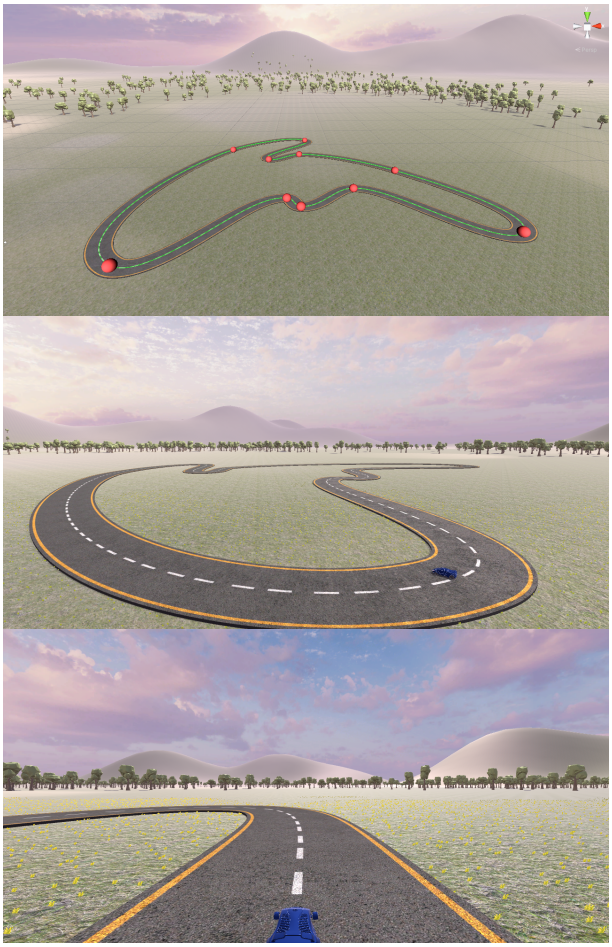

Fig. 14. **Top**) In-engine view showing the bezier curve and its control points. **Middle)** Full view of the track. **Bottom)** First person view of the track integrated in an environment.

PCG content. But some things to try out in the future would be to consider other constraints to improve the creation of this racing tracks. We could even have multiple “discriminators” to be used in different training sessions. For example, the Track Generator currently doesn’t consider the width when extrapolating the mesh, so there are tracks that get “approved” because they don’t intersect but they still touch each other ever so slightly.

Some other improvements would be to integrate the use of the XYZ space with the consideration of the usability constraints for it’s training. And even to implement some PCG in other parts surrounding the racing track as it would be having different environments that can suit the racing track.

Acknowlegment

I would like to thank Dr. Andres Mendez-Vazquez for being my mentor for the length of the development of this project. I would also like to thank my classmates Carlos Cardenas-Ruiz and Emilio Tonix-Gleason for their support, comments and suggestions for this development process.

REFERENCES

[1] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Computing and Applications*, vol. 33, no. 1, pp. 19–37, 2021.

[2] B. Gareth, "No man's sky," [Game], Hello Games, 2016. [Online]. Available: https://www.nomanssky.com/

[3] M. Persson and J. Bergensten, "Minecraft," [Game], Mojang Synergies AB., 2011. [Online]. Available: https://minecraft.net/en-us/

[4] M. Edmund and H. Florian, "The binding of isaac," [Game], 2011. [Online]. Available: https://store.steampowered.com/app/113200/The_Binding_of_Isaac/

[5] Y. Derek, "Spelunky," [Game], Mossmouth, 2008. [Online]. Available: https://spelunkyworld.com/

[6] D. Mark and C. Jeramy, "Borderlands series," [Game], 2K Games, 2009. [Online]. Available: https://borderlands.com/

[7] 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," pp. 1–28, 2018. [Online]. Available: http://arxiv.org/abs/1809.02627

[8] E. Giacomello, P. L. Lanzi, and D. Loiacono, "DOOM Level Generation Using Generative Adversarial Networks," *2018 IEEE Games, Entertainment, Media Conference, GEM 2018*, pp. 316–323, 2018.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[10] J. Hayes, L. Melis, G. Danezis, and E. De Cristofaro, "LOGAN: Membership Inference Attacks Against Generative Models," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 1, pp. 133–152, 2019.

[11] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "PCGRL: Procedural content generation via reinforcement learning," *Proceedings of the 16th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2020*, pp. 95–101, 2020.

[12] P. Giblin, M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2001, vol. 85, no. 502.

Julia Alejandra Rodriguez-Abud has a Digital Arts major. She has worked as a software developer and currently is pursuing her M.Sc. at Cinvestav Guadalajara.