Procedural Generation of Isometric Racetracks Using Chain Code for Racing Games

Erik Jhones F. do Nascimento
Department of Computer Science
Federal Institute of Ceará (IFCE)
Fortaleza, Brazil
erik.jhones06@aluno.ifce.edu.br

Tassiana M. Castro
Department of Computer Science
Federal Institute of Ceará (IFCE)
Fortaleza, Brazil
marinho.tassiana05@aluno.ifce.edu.br

Ana Carolina S. Abreu
Department of Computer Science
Federal Institute of Ceará (IFCE)
Maracanaú, Brazil
ana.carolina.silva05@aluno.ifce.edu.br

Filipe A. Lira
Department of Computer Science
Federal Institute of Ceará (IFCE)
Fortaleza, Brazil
filipe.almeida.lira04@aluno.ifce.edu.br

Amauri H. Souza
Department of Computer Science
Federal Institute of Ceará (IFCE)
Fortaleza, Brazil
amauriholanda@ifce.edu.br

***Abstract*—In this work, we propose a procedural generation method for railroad circuits based on the chain code algorithm. The process creates a list with directions in a two-dimensional space from a series of operations using the chain code and images of predefined real tracks. This list was used in the *Unity engine* to provide the positions, starting from a predefined starting point, which will insert the elements of the circuit during its creation. The race track format still goes through a preprocessing method to filter the ladder effect generated by the chain code and fine-tune details before being simulated in a game. This allows each track to have a unique pattern and good gameplay. Tests carried out using our generation method in a 2D (two dimensions) isometric kart game showed that in all runs, the tracks had an average of 98.5% difference among them and maintained the gameplay standards.**

***Index Terms*—racetracks, racing games, chain code, procedural generation, 2D games**

I. Introduction

Racing games are a prevalent genre and almost as old as the digital games and the early Arcades. The first examples of racing games date back to 1974, and the evolution of the genre in Arcades has provided unique experiences with steering wheels and cabins where the player can sit, giving an impression of immersion never before provided [15].

Since racing games represent a unique genre in the industry, its development also has unique characteristics [21]. Thus, the knowledge to develop them is a secret kept by companies that already have the know-how to produce them. To work with the limitation of lack of experience in this type of game, experimentation is an essential tool that allows the developer to create and test different backgrounds and to be able to choose which one best suits the situation. However, the cost of producing tracks can be prohibitive since each iteration of the experimentation requires a new circuit and a specific effort from the development team to elaborate the new layout, to change the geometry of the track, to modify the terrain, and adapt the adornments to the perimeter of the runway [8].

Current solutions for the games production, in general, are not suitable for the production of racing games. With tools right for other game genres, game engines are adopted with difficulty in the production of racing games, requiring adaptations in its use [16].

In this paper, we propose a procedural tracks generation method for 2D (two-dimensional) isometric racing games, with flat terrain based on chain code algorithm. Our approach considered focuses on 4 processing steps for the production of a list of cartesian directions in a bidirectional space, and this list was applied to know the position of the track elements in the game field. After these 4 steps, the race track can be generated by the *Unity engine*.

Our approach was inspired by the works of [1], [8], [10]. We tested our generative approach using a 2D isometric game, produced by the authors, based on the 1992 Mario Kart racing game. Empirical results have shown that the chosen approach managed to generate race tracks that differ 98.5% from each other, having a path size suitable for the type of game, and building truly playable tracks. We verified that our approach can be easily modified to run on other varieties of racing games.

II. Theoretical Reference

In this section, we introduce the concepts for understanding the methodology applied on the work.

A. What makes a racing game fun?

Togelius *et al.* (2006) [18], offer a hypothesis for the factors that make a racing game interesting. They list 5 factors that were used as metrics for evaluating the results. The first factor of fun would be the feeling of speed. People like to drive fast and the track must allow them to reach maximum speed; The second factor would be the challenge, that is, the player would be bored driving in an endless line, even at high speed; The next factor would be the right adjustment of the challenge, as

hitting all the time during the race, for example, would not be fun; The following factor would be the variation of challenges, with the track varying in characteristics and not repeating the same challenge all the time; The last factor would be drift or slippage in curves, and it seemed to be an important element for the authors. All the fun factors are associated with the track and its interaction with the vehicle.

Other authors offer different perspectives on the characteristics that a racing game must have to be considered fun. For example, Ralph Koster (2013) [19] offers a different perspective in his book "Theory of Fun For Game Design". Speaking of video games in general, he says that playing and learning are closely linked, and that for a game to be fun it needs to allow the player to learn continuously. Togelius *et al.* (2006) [18] states that one way to interpret this in a racing game context would be that a good race track is one where the player has very poor results the first time he plays, but quickly and consistently improves on subsequent races.

B. Procedural Content Generation (PCG)

According to Shaker *et al.* (2014) [17], Procedural Content Generation (PCG) consists of generating data through algorithms, data that can be interpreted in a game as scenarios, images, models, sounds, vehicles, characters, and other elements. The terms "procedural" and "generation" imply that algorithms that create something are being considered and these algorithms can be executed with or without human intervention. Overcoming storage limits in the first computer games was one of the main reasons for using procedural content generation. In the early 1980s, the storage constraint forced designers to look for other methods to generate and save content.

Hendrikx *et al.* (2013) [24] identify five groups of methods for Procedural Content Generation for Games (PCG-G) that were group into classes such as Pseudo-Random Number Generators (PRNG), Generative Grammars (GG), Image Filtering (IF), Spatial Algorithms (SA), Modeling and Simulation of Complex Systems (CS) and Artificial Intelligence (AI) ,with data augmentation [2], see Fig.1. They show in their work that methods belonging to these groups can be used in different types of content such as Textures, Sound, Vegetation, Buildings, Behavior, Fire, Water, Stone, and Clouds, Indoor Maps, Outdoor Maps, Bodies of Water and Other Map Features, Ecosystems, Road Networks, Urban Environments, Entity Behavior, Puzzles, Storyboards, Stories, Levels, System Design, World Design, and others.

In fact, PCG techniques can be used for a multitude of applications in the digital gaming world. In his article, Lima *et al.* (2019) [3], for example, developed a method of procedural generation of missions for games using genetic algorithms and automated planning. Combining planning with an evolutionary research strategy guided by arches history, the proposed method can generate coherent missions based on a specific narrative structure.

Serpa *et al.* (2019) [4] uses artist-generated similar images to create a shaded and a color-segmented version of an outline. His technique is capable of generating sprites similar to those generated by artists. Dam *et al.* (2019) [5] present a method that allows the creation, with limited data availability, of a wide, high-quality environment optimized for terrain simulations using Unity 3D (three dimensions). Pereira *et al.* (2019 [6] uses the procedural generation to tell stories in a serious game called Orange Care. This research proposes the application of such a technique to automate the construction of characters and scripts. Finally, Connor *et al.* (2019) [7] assesses the impact of content generated procedurally in games nowadays. The study assesses the impact on game quality. It concludes that there is not much noticeable negative impacts on games created procedurally compared to those created by classical methods.

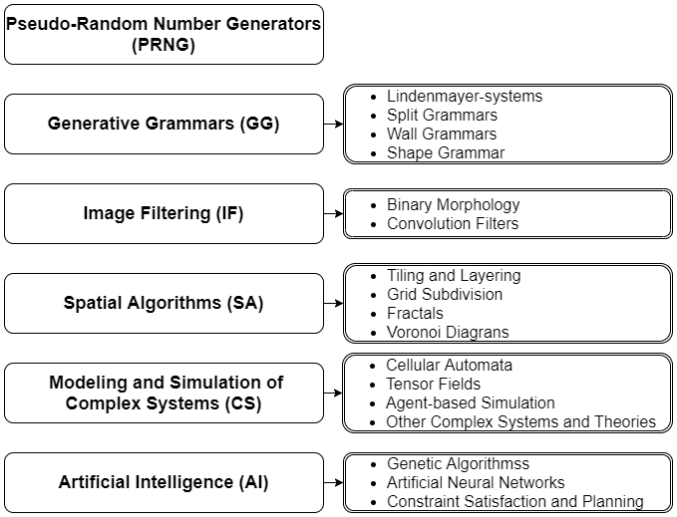

Fig. 1. Taxonomy of common methods for generation game content, Hendrikx *et al.* (2013) [24].

The use of procedural generation is a technique widespread in all forms of electronic games. Much of this success is mainly due to automation and reduction of production costs.

C. Chain code

Chain code is a lossless compression technique used for representing an object in images. The coordinates of any continuous boundary of an object can be represented as a string of numbers where each number represents a particular direction in which the next point on the connected line is present. The chain code was created in 1961 by Herbert Freeman [22]. This method encodes the boundary of a region into a sequence of octal digits, each representing a step of the boundary in one of the eight basic directions in relation to the current pixel position. Chain code is useful for reducing storage in relation to the number of bits needed to store a list of the coordinates of all boundary pixels (and also in relation to the number of bits needed to store a complete binary image), L. Yong and B. Zalik (2005) [11].

The shape of a region can be represented by quantifying the relative position of consecutive points on its boundary.

The chain code technique achieves this representation by analyzing each point on the boundary in sequence (e.g., counterclockwise) and assigning a code digit to the transition from each point to the next. The transition from one point to the next can be coded with 4-connectivity, considering the 4 nearest neighbors, or 8-connectivity, where transitions to all adjacent points are coded, L. Yong and B. Zalik (2005) [11]. Fig.2 shows how the chain code works.

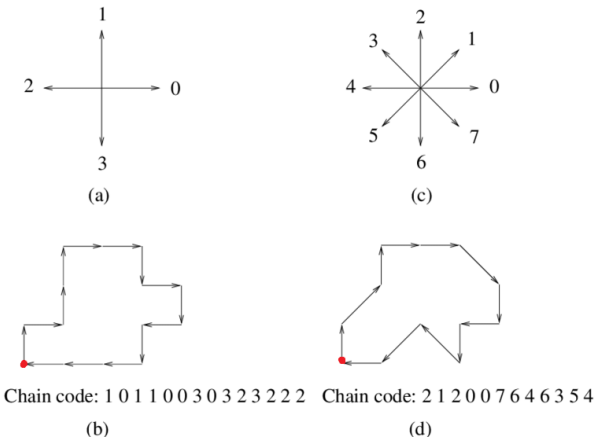

Fig. 2. The chain code has several types of neighborhood connections. However the most famous and basic ones are 4 and 8.

Efficient uses of chain code include content compression with the works of Zalik *et al.* (2018) [23] and K. Dhou (2019) [25], and feature extraction with the works of A. Azmi (2017) [26] and P. Karezmanek (2017) [27].

D. Ramer–Douglas–Peucker Algorithm

The Ramer–Douglas–Peucker Algorithm [14], [28] is an algorithm that reduces the number of points that is approximated by a series of points. It is also known as the Douglas–Peucker algorithm and iterative end-point fit algorithm. In simple words, it represent a complex line with fewer points in a visually proper way. See the Algorithm 1.

The main purpose of this algorithm is to find a similar curve with fewer points for a given curve composed of line segments (also called Polylines). This algorithm define 'dissimilar' based on the maximum distance between the original curve and the simplied curve, i.s. the Hausdorff Distance. The simplified curve consist of a subset of points that defined the original curve. The Ramer–Douglas–Peucker Algorithm is most commonly used in geospatial visualizations, like Google Maps, but also useful for other in-browser visualizations as well.

Algorithm 1: DouglasPeucker

```
Input: PointList[], epsilon
Output: ResultList[]
initialization
dmax = 0
index = 0
end = lenght(PointList)
for i to (end - 1) do
    d = perpendicularDistance(PointList[i],
    Line(PointList[1], PointList[end]))
    if d > dmax then
        index = i
        dmax = d
    else
ResultList[] = empty
if dmax > epsilon then
    recResults1[] =
    DouglasPeucker(PointList[1...index], epsilon)
    recResults2[] =
    DouglasPeucker(PointList[index...end], epsilon)
    ResultList[] = (recResults1[1...length(recResults1) -
    1], recResults2[1...length(recResults2)])
else
    ResultList[] = (PointList[1], PointList[end])
```

III. Related Work

In the academic sphere, the work most related to the theme of this article is the study of Cavalcante Junior (2017) [8]. His research aimed to use procedurally generated content, associated with AI, to assist the designer in the production of racing games. Its aim was to obtain an efficiency gain for the development team and cost reduction. The proposed tool intended to guide the designer in the creation of racing game content, using a concept of curves to simplify the design of the layout, in addition to correcting game design decisions, supporting creativity without removing the power of decision.

C. Gleidson (2018) [1] affirmed that the terrain is the most extensive part of a game. The author claimed that some games need to present different terrains on each level so that the game does not become repetitive to the players. His work presented a methodology for procedural terrain generation for 2D platform games. His approach developed was based on the use of Markov models, specifically the hidden Markov chains, to produce new terrain probabilistically. His results showed that the technique presented was able to generate a 2D platform game terrain.

E. Galin *et al.* (2010) [10] proposed an automatic method for generating roads based on a weighted anisotropic shortest path algorithm. Given an input scene, the authors automatically created a path connecting an initial and a final point. The trajectory of the road minimizeed a cost function that takes into account the different parameters of the scene including the slope of the terrain, natural obstacles such as rivers, lakes, mountains and forests. The road was generated by excavating the terrain along the path and instantiating generic parameterized models.

IV. Workflow

Given a flat terrain, created by a procedural method, manually or extracted from a set of real data, our generative approach works by following the steps listed below.

First, we created a database with images of racetrack circuits and a list with the pixel positions of the circuit for each image is extracted using the chain code. After that, we choose some elements from the generated list in order to create the new circuit. Next, we created a list with the directions that, from a starting point on the game ground, the elements of the new circuit would be added. Following, we perform a process of fine-tuning details and removing noise from this list to improve its appearance in the creation process. Finally, we use the list of directions to guide the *Unity engine* where the elements that make up the track must be created during the game execution. The track elements were loaded and assembled during this stage. Fig.3 shows a simple flow chart of the method.

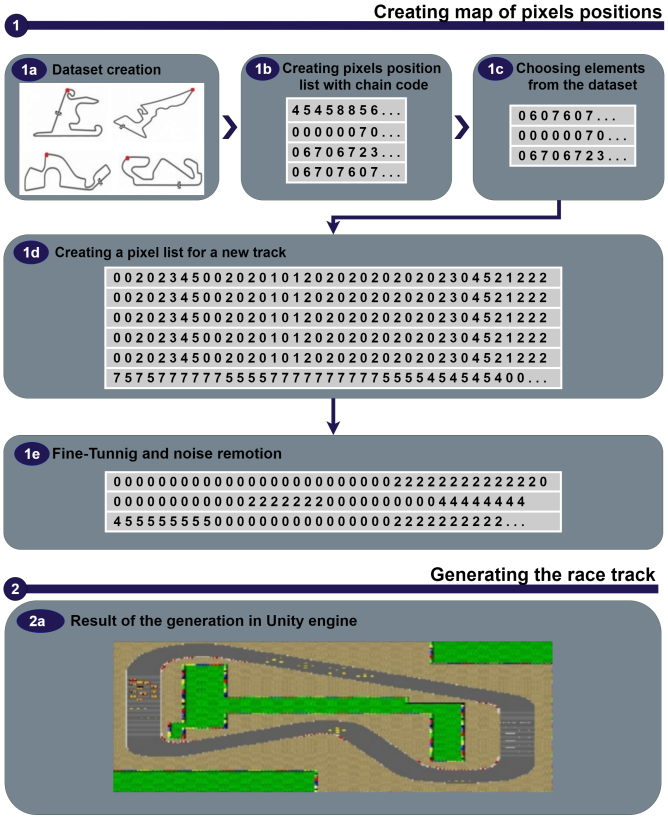

Fig. 3. Flowchart presenting the five steps taken by the method to generate the racetracks.

A. *Database and list of directions*

First, we acquire images from the internet containing real and playable racing circuits from different games. We also created other circuits manually in order to increase the diversity of forms in the dataset. The images formats and sizes were matched for Joint Photographic Experts Group (JPEG or JPG) and sized for 300×300 pixels. The images sizes directly affects the size of the race track created, due to the fact that the our approach works with the perimeter size of the objects in the images. The database contained 150 images, but it could be expanded indefinitely. Fig.4 shows an example of the data set tracks (See the appendix A if you want more information about the track in the images).

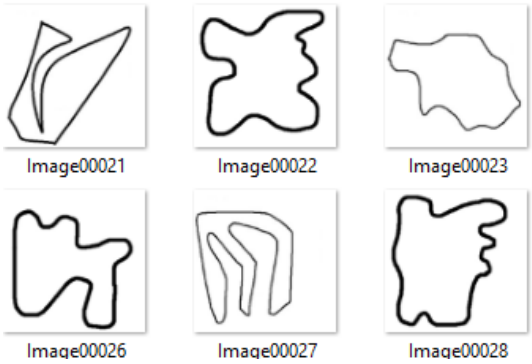

Fig. 4. Some race tracks that make up the dataset for this project. Most are tracks for Kart and Formula 1 games.

Since only the circuit outline was necessary to generate the race tracks, we developed an approach of low computational cost and relatively simple, based on the use of chain code algorithm.

For this work, we considered the chain code with 8 neighborhood. The reason we've chosen this 8-neighbor approach is due to our images have 8-pixel connected. This allowed for a more faithful capture of the shape of the race tracks, and thus a more faithful recreation of new tracks. To execute the chain code on the data set images, we've chosen first a point in the area of interest as the starting point, and the point chosen was the most upper left. Fig.5 exemplifies data set tracks and their chain code graph.

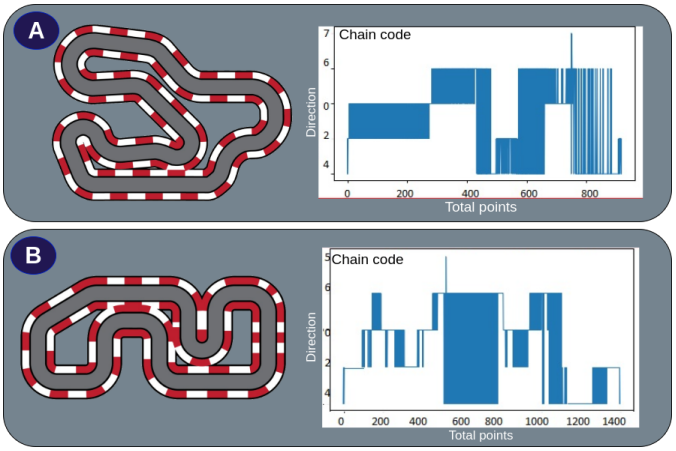

Fig. 5. Two data set racetracks and their respective chain codes obtained using 8 connections. The vertical axis indicates the direction value and the horizontal axis the number of values. The red dot represents the starting point of the algorithm.

After all the images were submitted to the chain code, a list containing all the lists with the chains was obtained.

B. Choosing elements from the list

One of the advantages of using the chain code was that the corresponding figure could be redrawn using the list with its chain code. We took advantage of this property to define how the result of the new race track would be. The track shape that would be generated depended a lot on this stage, since the strings chosen would directly affect the shape of the track.

To choose which elements of the chain list would be used to define the new circuit, we apply a stochastic criterion of choice. More precisely, we choose 10% of the samples at random to form a new chain list. we define this percentage value based on tests carried out by the authors in search of the best number of chains that could generate a totally different track and that could be playable, following the criteria of [18]. Very low percentage values resulted in tracks that were in the dataset itself, while higher percentage values resulted in tracks so simple and similar that it would leave any player unmotivated to play on it. Fig.6 exemplifies generated race tracks and the percentage of chains chosen.

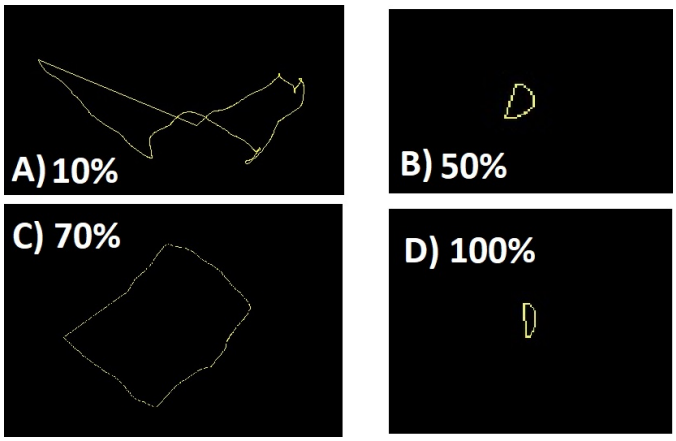

Fig. 6. Race track samples obtained using 10%, 50%, 70%, and 100% of the data present in the data set respectively.

The test to find which value would be most suitable was performed as follows: 100 iterations were performed for each determined number of chains (10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100%), then the number of iterations that were generated tracks with up to 50% difference from the previous ones. To compare the generated tracks, we calculate the *HU invariant moment* of each image. As this approach was invariant to scale, rotation and translation, it was the most suitable for the work. Images that had between 50% and 100% difference were counted as potential new circuits, while the others were disqualified. Fig. 7 shows a graph that relates the number of images / chains chosen with the degree of average difference for 100 iterations.

The percentage of 10% reaches an average of 98.7% of new and visually attractive tracks.

C. Creating the list with directions

Given a set of chain lists obtained by the method described in the previous subsection, the process of generating a single

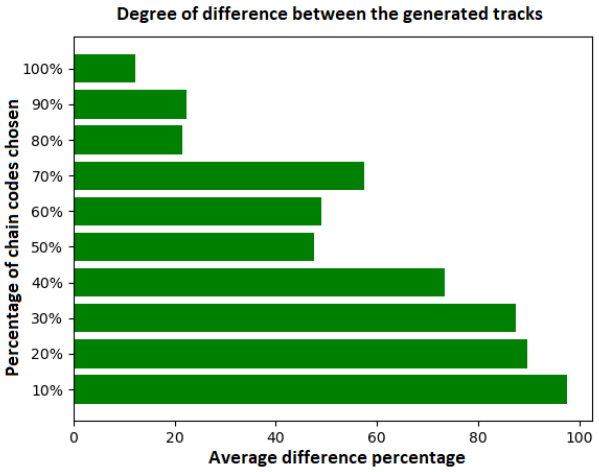

Fig. 7. Comparative graph between the number of images used to generate the new chain and the difference of the images generated for 100 iterations.

chain list derived from that set has followed a few steps. As the circuits considered to generate the data set had different shapes and sizes, the size of the chains generated by these circuits were also different. This means that after the choice step, it was necessary to use a method to match the size of these chosen chains. To normalize the size of the chains, we choose initially the chain with the smallest size (the goal was to make all other chains have the same size as the chosen chain). The normalization function was formally defined in Algorithm 2.

Algorithm 2: Normalize two chain lists

```
Input: A, B
Output: new_list
initialization
X = 0
P = |A| / |B|
p1 = P
R = int(P)
new_list = [1... |A| ]
while X < |B| do
    new_list = |B|
    P = P - R
    P = P + P1
    R = int(P)
    X= X + R
```

Where A and B are the biggest and smallest chain list respectively.

With the chains standardized, the resulting new chain list was created from a process of choosing the modes majority. This process should iterate through all the lists at the same time and, for each index, the value with the greatest trend must be chosen for the new list. In this way, important characteristics of the circuits used as base should be maintained. For example, if most circuits had a curve in a certain area, it means that

the resulting circuit also needed to have a curve, however its shape should be quite different from that of the others. This approach also maintained control over the new circuit shape, preventing its final form from being completely random and unplayable. Fig.8 exemplifies the result of treating the chains with algorithm 1. The difference in appearance between the strings before and after they are normalized can be noted.

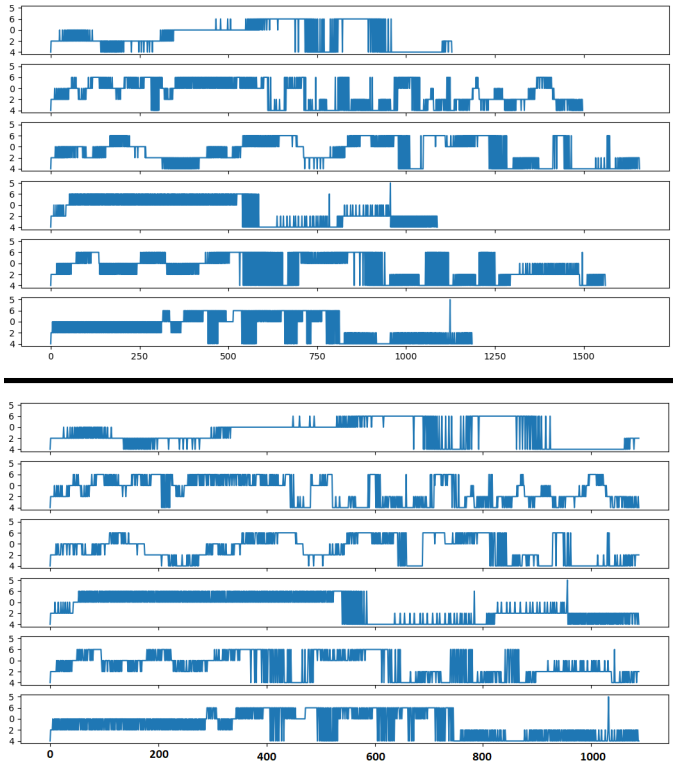

Fig. 8. A) Lists of chains before having their sizes normalized by algorithm 1. B) The same lists already standardized. It can be noted that the size of all of them is the same as the one at the fourth chain.

Fig.9 shows the newly created list.

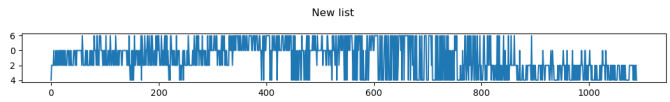

Fig. 9. A new chain list originated from a set of other lists using the most mode technique.

D. Thinning and noise removal process

The process of creating the new list may generate imperfection in the final circuit design. This was one of the side effects of using mode between chain lists to create the new list. This imperfection happened when a group of values in the lists had very different element values.

The result of this type of situation was a new list with groups of values that were not homogeneous, but very different. This kind of situation is known as the ”stair effect” because visually a figure that would normally be a ”smooth” straight line looks like a straight line with steps.

To eliminate these imperfections, we use the Ramer–Douglas–Peucker algorithm described in [14], [28], see the Algorithm 1. This technique smoothed the lines affected by the stair effect. The result of the ”stair effect” on the race tracks can be seen at the upper part of the Fig.10, and the result of the smooth can be seen at the botton of Fig.10.

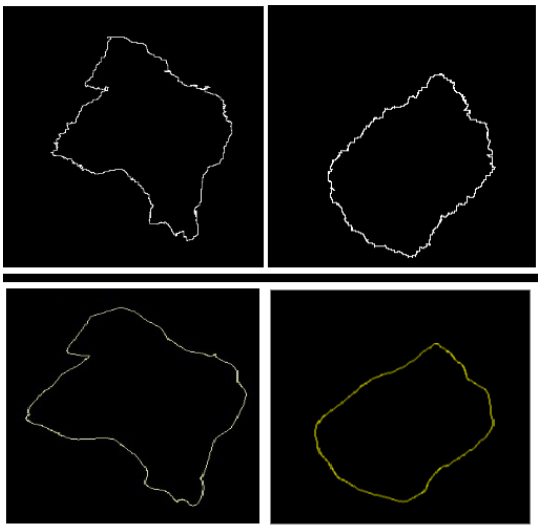

Fig. 10. Up side) The stair effect affects the appearance of contour lines making them look noisy. Down side) The result of eliminating the ladder effect on the tracks is clear.

E. Track generation in the UNITY Engine

The steps listed above were done with the sole purpose of generating a list containing directions in a two-dimensional space to serve as a guide for the allocation of the elements that would compose the circuit. First, we choose a starting point on the pitch. The chosen point was in the middle of the map. Every time a direction pointed out of the map, a routine was executed causing the elements generated so far to recede in a position in the opposite direction to the current one. It ensured that the race track was within the limits of the map. The automatic creation process occurred by iterating through that list. For each element of the list, its value was read (which was a direction to be followed from the current point), and the elements of the track in the direction read were added to the field of play. This process, in addition to creating a race track, also ensured that the sprites (like the ones that make up the asphalt) did not overlap. This was important because in practice, an asphalt sprite corresponds to a certain size of the total circuit. Therefore, a list that has a thousand elements and a sprite of asphalt that corresponds to 1 meter of road, will result in the end a race track with 1 km in length. The other elements of the race track could be added following predefined rules. For example, barricades on curves, which would only be added to locations whose values of directions in the list

corresponded to the beginning, middle and end of a curve. This process had the advantage of requiring little computational cost. Fig.11 shows one of the race tracks generated by the method defended in this article during one of the practical tests.

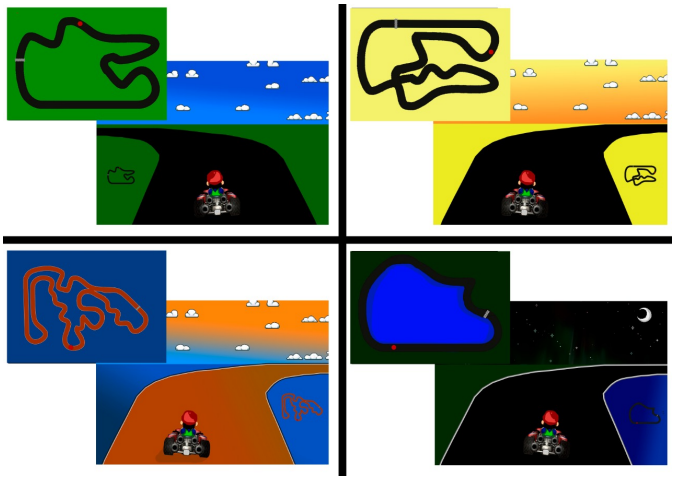

Fig. 11. Our approach managed to create several different race tracks as well as different race scenarios as well. In almost all cases, the five criteria of "what makes a racing game fun" were respected.

V. Results and Discussions

We implement our method with the C# language, used in the initial development of the code and adjustments of some parameters, and C# using *Unity Engine* for practical tests. The tests were performed on an Intel Core i5-3470 2.9GHz quadcore. A 2D kart game was developed based on the famous racing game Mario Kart [20] to test the gameplay of the generated tracks. In this test game, only the asphalt and the start line were added. It was done to simplify the tests, however more elements can be added, such as sideways and barricade tires.

We perform tests out by alternating the amount of chain samples, considered to form the final race track. We quantify the average generation and loading time for the tracks in each test. The number of samples was increased by 10% for every 10 runs of the game. We also evaluated the size of the race tracks generated by our approach. Kart tracks were on average 1150 meters long and 8 meters wide. Table 1 shows the results of our analyses. It can be noted that the size of the tracks, despite a certain variation, are in all cases within the allowed limit for go-kart race tracks.

Even though the average times were less than 13 seconds and the average length of the tracks was suitable for this kart game, increasing the number of chain samples to generate the tracks would not improve the method at all. As seen in the subsection 'Choosing Elements from the List' this would just make the tracks less challenging and repetitive.

We tested our approach with an external audience. We selected 271 volunteer participants from a gaming group called "1 Real a Hora" on Discord, a free, voice over IP app, designed primarily for gaming communities. Participants were quite diverse in their preferences in game types and racing game genres. This allowed the results to be less biased and ended up favoring more positively because participants "like kart games".

TABLE I
Results of average build times and average sizes for the race tracks considering x% percentage of dataset data.

Percentage of data	Average time/size	
	Time Creation (s)	*Size Track (m)*
10%	8.8±0.74	1044±33
20%	10.9±0.45	1185±62
30%	7.0±0.18	1220±95
40%	10.6±0.24	1191±74
50%	9.7±0.67	1022±66
60%	12.4±0.41	1028±92
70%	13.2±0.49	1042±30
80%	7.9±0.59	1025±85
90%	11.5±0.67	1174±69
100%	11.9±0.26	1169±29

We offered the Mario Kart game executable file, which contained our approach, and asked participants to play at 100 race tracks consecutively. At the end of each race, we asked them to fill out an evaluation form about it. We asked the participants to be neutral about the game's setting, its genre (Kart), visual elements in general, and we asked them to focus their attention only on 5 factors, namely the same ones advocated by Togelius et. al. (2016) [18]. The questions contained in the questionnaire were: 1 Did you feel the speed during the game? 2 Did you feel the race track was challenging? 3 Did you feel that the race tracks were getting harder as you played? 4 Do you think this race track was different from the previous ones? 5 Did you manage to perform any "drift" in corners?

With these five questions, we assess our approach to the factors that make a racing game fun. Remembering that our goal with this article is not to report the creation of a racing game, but to report the creation of a method of automatic generation of race tracks that is efficient, fast, and above all, simple. We present the results of these tests in table 2. Each question in the questionnaire was associated with a factor in the table, question 1 was associated with factor 1, and so on.

We separated the responses to the questionnaires into 3 groups, those that gave positive, neutral (i.e. "I don't know"), and negative responses. In fact, as can be seen in table 2, our racetrack generator method managed to give the vast majority of survey participants what was promised: Racetracks different from each other, with an adequate size, and challenging gameplay, but not impossible. Our method pleased a good part of the participants when the subject is "difference among race tracks". The factor 5, despite being very low, does not let us down. After all, it is a detail of the game mechanics, and does not come from our method.

VI. Conclusion and future works

This paper presented a method for generating race tracks that work in isometric 2D race games. The generation of circuits produced different shapes at each run using a simple algorithm and was able to maintain the proper level of playability on the track. By the practical tests, the 5 factors of a fun racing game defended by [18] and also by [19] were respected. The proposed solution can be easily modified to suit different types of application. Further research can be done to take into account obstacles in the circuits, such as tunnels, or ramps. One way to improve the generation of leads is to consider subsets of chains with a certain similarity. It would give more loyalty to certain characteristics of these chains in the final result, thus, making a track more realistic, if this is the goal.

TABLE II
Participants' responses to the tests of our model with a real racing game "Mario Kart". They focused their answers only on the race tracks and their quality.

response type	Quality Factors				
	factor 1	***factor 2***	***factor 3***	***factor 4***	***factor 5***
Positive	83.0%	81.3%	75.9%	96.7%	24.5%
Neutral	13.3%	17.8%	15.0%	1.4%	3.7%
Negative	3.7%	1.1%	9.1%	1.9%	71.8%

As a future work, it is being considered to change the method for procedural generation of race tracks advocated in this work to suit racing games in 3D. [12] applied the chain code to describe curves in 3D and [9] addressed criteria of difficulty on race tracks, so an approach using this principle could achieve satisfactory results in this project. A repository with the source code can be found in https://github.com/ErikJhones/racetrack.

References

[1] G. M. Costa and T. B. Borchartt, "Procedural terrain generator for platform games using Markov chain," in 17th Brazilian Symp. on Comp. G. and Digi. Ent., Foz do Iguaçu, PR, BR, Eds. Oct. 2018, pp. 671-674.

[2] E. J. F. do Nascimento, D. V. Cavalcante, A. C. S. Abreu, D. P. P. Mesquita and A. H. de S. Júnior, "Classificando Graus de Pterígio Utilizando Aprendizado de Máquina," in Journal of Health Informatics, Foz do Iguaçu, PR, BR, vol. 12, Dez. 2021, pp. 248-253, issn: 2178-2857.

[3] E. S. Lima, B. Feijó and A. L. Furtado, "Procedural Generation of Quests for Games Using Genetic Algorithms and Automated Planning," in 18th Brazilian Symp. on Computer G. and Digi. Ent., Rio de Janeiro, RJ, BR, Eds. Oct. 2019, pp. 495-504.

[4] Y. R. Serpa and M. A. F. Rodrigues, "Towards Machine-Learning Assisted Asset Generation for Games: A Study on Pixel Art Sprite Sheets," in 18th Brazilian Symp. on Computer G. and Digi. Ent., Rio de Janeiro, RJ, BR, Eds. Oct. 2019, pp. 533-542.

[5] P. Dam, F. Duarte and A. Raposo, "Terrain Generation Based on Real World Locations for Military Training and Simulation," in 18th Brazilian Symp. on Computer G. and Digi. Ent., Rio de Janeiro, RJ, BR, Eds. Oct. 2019, pp. 524-532.

[6] Y. H. Pereira, R. Ueda, L. B. Galhardi and J. D. Brancher, "Using Procedural Content Generation for Storytelling in a Serious Game Called Orange Care," in 18th Brazilian Symp. on Computer G. and Digi. Ent., Rio de Janeiro, RJ, BR, Eds. Oct. 2019, pp. 543-548.

[7] A. M. Connor, T. J. Greig, and J. Kruse, "Evaluating the Impact of Procedurally Generated Content on Game Immersion," The Computer Games Journal, vol. 6, pp. 1-17, Dec. 2017, doi:10.1007/s40869-017-0043-6.

[8] A. V. C. Junior, "Uma ferramenta de auxílio ao designer na Geração Procedimental de Conteúdo para jogos de corrida," M.S. thesis, Univ. Fed. de Pernambuco, Recife, PE, BR, 2017. [Online]. Available: https://repositorio.ufpe.br/handle/123456789/29387

[9] R. V. D. Ploeg, "Modeling Race Track Difficulty in Racing Games," M.S. thesis, Dept. Inf. Comp. Sci., Univ. Utrecht, Utrecht, NLD, 2015. [Online]. Available: encurtador.com.br/fpsLX

[10] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin, "Procedural generation of roads," Comp. Graph. Forum, vol. 29, no. 2, pp. 429-438, Jun. 2010, doi: 10.1111/j.1467-8659.2009.01612.x.

[11] Y. K. Liu and B. Žalik, "An efficient chain code with Huffman coding," Pattern Recognition, vol. 38, pp. 553-557, Apr. 2005, doi: 10.1016/j.patcog.2004.08.017.

[12] E. Bribiesca, "A chain code for representing 3D curves," Pattern Recognition, vol. 33, pp. 755-765, May. 2000, doi: 10.1016/S0031-3203(99)00093-X.

[13] P. Karczmarek, A. Kiersztyn, W. Pedrycz, and M. Dolecki, "An application of chain code-based local descriptor and its extension to face recognition," Pattern Recognition, vol. 65, pp. 26-34, Dec. 2016, doi: 10.1016/j.patcog.2016.12.008.

[14] T. Poiker and D.H. Douglas, "Reflection Essay: Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature," Class. in Cart.: Reflec. on Influe. Art. from Cartographica, vol. 10, pp. 29-36, Mar. 2011, doi: 10.1002/9780470669488.ch3.

[15] IGN Entertainment. "the history of racing games" ign.com. http://uk-microsites.ign.com/the-history-of-racing-games/ (accessed Jul. 1, 2021).

[16] K. Graft. "Game Tech Deep Dive: Reworking the Unreal Engine for racing." gamedeveloper.com. https://www.gamedeveloper.com/design/game-tech-deep-dive-reworking-the-unreal-engine-for-racing (accessed Jul. 1, 2021).

[17] N. Shaker, J. Togelius and M. Nelson, "Constructive generation methods for dungeons and levels" in Procedural Content Generation In Games, 1th Ed. Gewerbestr, CH: Springer Int. Pub., 2016, pp. 55–70.

[18] J. Togelius, R. De Nardi and S.M. Lucas, "Making racing fun through player modeling and track evolution," In Proc. of the SAB'06 W. on Adap. Appr. for Opt. P. Satis. in Comp. and Phys. G., Southern Denmark, DK, Nov. 2006.

[19] R. Koster, "Different Fun For Different Folks," in Theory of Fun for Game Design, 2th ed. Sebastopol, CA, USA: O'Reilly Media, 2013, pp. 102-111.

[20] E. A. D. Nintendo, "Super Mario Kart Instruction Booklet." nintendo.com. https://www.nintendo.co.jp/clvs/manuals/common/pdf/CLV-PSAAFE.pdf (accessed May. 12 2021).

[21] J. Togelius, R. De Nardi and S. M. Lucas, "Towards automatic personalised content creation for racing games," 2007 IEEE Symposium on Computational Intelligence and Games, 2007, pp. 252-259, doi: 10.1109/CIG.2007.368106.

[22] H. Freeman, "On the Encoding of Arbitrary Geometric Configurations," in IRE Transactions on Electronic Computers, vol. EC-10, no. 2, pp. 260-268, June 1961, doi: 10.1109/TEC.1961.5219197.

[23] B. Žalik, D. Mongus, N. Lukač and K. R. Žalik, "Efficient chain code compression with interpolative coding," in Information Sciences, vol. 439-440, pp. 39-49, May. 2018, doi:10.1016/j.ins.2018.01.045.

[24] M. Hendrikx, S. Meijer, J. Van Der Velden and A. Iosup, "Procedural content generation for games: A survey," in ACM Transactions on Multimedia Computing, Communications, and Applications, vol. 9, pp. 1-22, Fev. 2013, doi: 10.1145/2422956.2422957.

[25] K. Dhou and C. Cruzen, "An Innovative Chain Coding Technique for Compression Based on the Concept of Biological Reproduction: An Agent-Based Modeling Approach," in IEEE Internet of Things Journal, vol. 6, no. 6, pp. 9308-9315, Dec. 2019, doi: 10.1109/JIOT.2019.2912984.

[26] A. N. Azmi, D. Nasien and F. S. Omar, "Biometric signature verification system based on freeman chain code and k-nearest neighbor," in Multimedia Tools and Applications, vol. 76, pp. 1-15, Jul. 2017, doi: 10.1007/s11042-016-3831-2.

[27] P. Karczmarek, A. Kiersztyn, W. Pedrycz and M. Dolecki, "An application of chain code-based local descriptor and its extension to face recognition," in Pattern Recognition, vol. 65, pp. 26-34, Dec. 2016, doi: 10.1016/j.patcog.2016.12.008.

[28] U. Ramer, "An iterative procedure for the polygonal approximation of plane curves," in Computer Graphics and Image Processing, vol. 1, pp. 244–256, Jul. 1972, doi:10.1016/s0146–664x(72)80017–0.