2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

Procedural generation of tunnel networks for unsupervised training and testing in underground applications.

Lorenzo Cano, Danilo Tardioli and Alejandro R. Mosteo

Abstract— **Developing a robotic application requires thorough testing of the complete system to ensure its reliability. However, depending on the target environment, real-life testing can be difficult to carry out, which favors simulations. Also, some techniques like those based on machine learning, may require large varieties of sensor data, which can be gathered in simulation with ease, whereas doing the same in real environments can pose a great challenge.**

This work presents a flexible approach to the procedural generation of tunnel networks suitable for underground robotics simulations. The method starts with a graph representation of an underground environment, and applies a custom meshing strategy to generate tunnels that follow the graph structure. This mesh can then be imported into the desired simulation software. The ease of use of this method allows for the testing of robotic applications in an arbitrary number of different environments in completely automated workflows.

I. Introduction

The use of simulation in robotics research is a fundamental tool to reduce costs, improve thoroughness, enable repeatable testing, and generally prepare before attempting an experimental campaign with actual hardware. For this reason, its use is pervasive in many fields of robotics, including the domains of underwater, ground-based, and aerial vehicles.

Procedural generation origins can be traced back to the early developments on computer graphics and refers to the use of algorithms instead of manual techniques to generate content such as images, level maps in games (another early-adopter industry of procedural generation) or, in the case of mobile robotics research, simulation environments in which to deploy virtual robots.

Robotics in hostile environments particularly benefit from simulation, as otherwise humans have to be exposed to difficult or dangerous environments for preparation and testing. In extreme cases, like robotics for search and rescue, hazardous materials inspections, or space missions, extensive testing in the operational environments may be entirely impossible. Mock-up environments can be helpful to some extent, but their preparation may prove too expensive or limited in scope.

Another reason to resort to simulation arises when the need for a large variety of environments is also a necessity. In many recent applications of machine learning, the training of algorithms requires bulk amounts of data [1] devoid of biases, so having a diverse set of data sources becomes a new demand during development. In these cases, procedural generation of simulation scenarios alleviates the repetitive and time-consuming task of manually creating simulation environments. Also, in exploration applications there is a continuous need for new environments in order to be able to test the algorithms.

The contribution of this work is a Python library and a Graphical User Interface (GUI) tool to generate underground tunnel networks, ranging from simple cases —like straight or curved road tunnels— to complex maze-like environments such as the gallery networks found in mining operations. The modular nature of our system allows to define the topological structure of the tunnel network separately from the geometric features of each individual tunnel, which can be parameterized in regard to roughness, shape and diameter.

The graphical user interface (GUI) is provided to easily produce tunnel networks when a particular structure is required. The resulting environments are exported as a mesh file, ready for importing in the usual simulators used nowadays in robotics research, such as Gazebo or Unreal.

Finally, our library can be easily integrated in unsupervised testing or AI-based-systems training pipelines. The final mesh is augmented with data created during generation, enabling the use of synthetic ground-truth during training or evaluation.

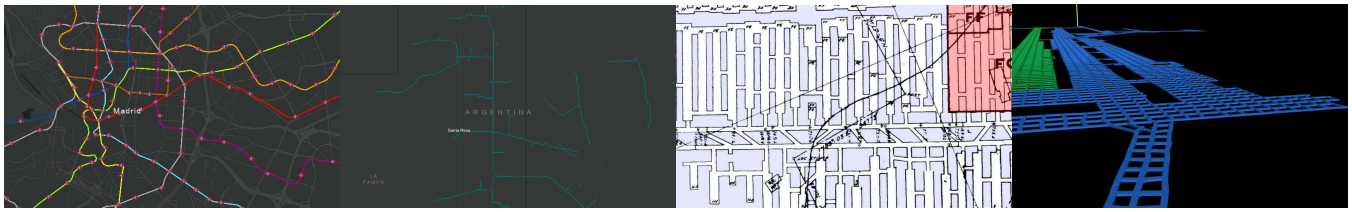
A. Motivation

The main reason for developing this tool was to further refine the work presented in [2]. Originally, this system —a framework for navigating in maze-like environments without the need for a geometric map— was trained in the tunnel tiles provided by DARPA for the SubT challenge [3]. It performed well in environments built with said tiles, but failed to generalize to more varied environments, as the use of a limited set of tiles narrows the training set.

By training the same system in the environments generated with this work, we have vastly improved the generalization and performance in more realistic environments. Later, this tool has proved extremely useful in training a fast tunnel traversal approach for drones and ground robots with limited sensing [4]. These tunnels can be generated with largely different parameters from one another (radius, roughness, curvature, inclination etc.), and this enables zero-shot learning from the generated environments to reconstructed real environments and real-world deployments.

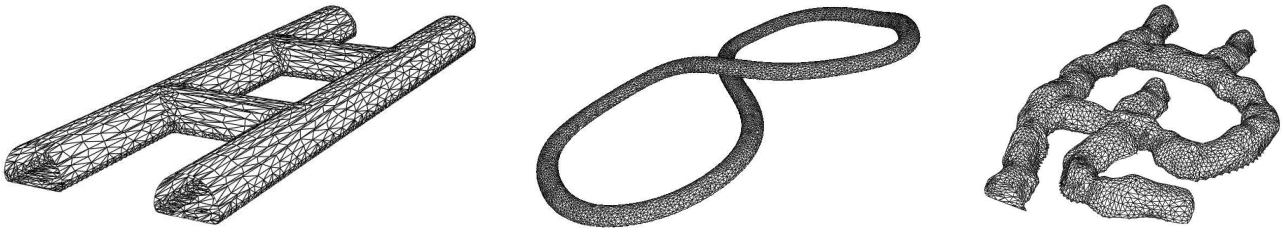
We strongly believe that this tool can be useful to the robotics community working in underground environments, given that these settings —especially human-made ones—

Lorenzo Cano is with the University of Zaragoza and Instituto de Investigación en Ingeniería de Aragón, D. Tardioli and A.R. Mosteo are with Centro Universitario de la Defensa, Zaragoza and Instituto de Investigación en Ingeniería de Aragón. Email: {lcano,dantard,amosteo}@unizar.es



(a) (b) (c) (d)

Fig. 1: Samples of human-made environments. a) Madrid's underground metro system. b) Underground water ways in Argentina. c) Mine in Pensylvania excavated using the Room and Pillar method. d) Model of a Coal mine in West Virginia.



(a) (b) (c)

Fig. 2: Three different meshes generated with the proposed method.

can be characterized as a set of tunnels that intersect with each other, as it is the case with civil infrastructure (like amenity tunnels or urban sewer networks), urban transportation systems (metro networks, underground roads) or mines. In Fig. 1 we illustrate this point with some examples.

Of course, simulation cannot totally replace actual in situ experimentation, but it can undoubtedly ease the transition to actual deployments. This same approach is the one used in the DARPA SubT challenge, where the organization provided extensive tools for robotics simulation in underground environments[1] as a testing platform previous to the real-world deployments. While this tool facilitates creating complex underground environments, it is limited to the provided set of tiles and does not allow the generation of randomized environments, which are essential for the generation of proper training sets for AI-based systems and thorough testing of robotic applications.

The paper continues with an overview of procedural generation in related fields in Sec. II. The main ideas and details about the method are given in Sec. III. The GUI is described next (Sec. III-B), and applications are presented in Sec. V, before the concluding remarks in Sec. VI.

II. Related Work

As advanced in the introduction, procedural generation is a well-known idea, although its particular execution is strongly tied to the application. Focusing on mobile robotics, we can find examples of procedural generation in the field of autonomous driving: complete urban environments for robotics are generated in, e.g., [5], [6], including simulation of crowds [7]. Closer to the topic of underground tunnel networks generation we find procedurally generated road networks in [8], [9], [10], [11].

Combining randomized generation with user input is explored in [12], where a procedural road generator using stochastic data from sample environments is combined with parametric control, enabling non-expert users to use the method.

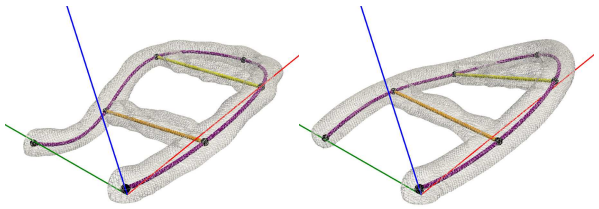
Underground robotics is a particularly challenging domain for experimentation [13], [14], [15], [16]. In the video game domain we can find examples of procedural generation of underground environments, but mainly focused on cave-like scenarios [17], [18], [19], [20], and which cannot be conveniently used by the robotics community as none include source code, a downloadable library or a readily usable API.

Generation of structures similar to tunnels, formed by linking straight and fixed-radius shapes, is also found in [21], which deals with bin picking of entangled objects. The authors considered infeasible to generate enough training data using real objects, and thus resorted also to procedural generation.

Early ideas for our underground tunnel generation can be found in our previous work [22]. Therein, we employed tiles from the DARPA Subterranean Challenge and assembled them procedurally in a puzzle-like manner.

To the best of our knowledge, our generator is the first of its kind in the robotics community specifically tailored to subterranean environments and supporting integration in automatic simulation pipelines from the outset. Furthermore, it incorporates a graphical user interface that can be used to guide the procedural generator and facilitate the replication of complex shapes by users without technical expertise.

[1] https://subtchallenge.gazebosim.org/world-builder



(a) (b)

Fig. 3: Meshes generated from code in Listing 1. The purple, orange and yellow splines correspond to `t1`, `t2` and `t3` respectively.

III. Generation of Tunnel Networks

As already mentioned, the goal of this work is to offer the means to generate underground tunnel networks that can be easily integrated in training or testing pipelines. Most underground environments (specially human-made ones, Fig. 1) can be understood as a set of galleries that intersect with each other. This conceptualization of underground environments is the main idea behind this proposal and allows the method to create scenarios like road tunnels (Fig. 2a), circuit-like tunnels (Fig. 2b) or mine-like environments (Fig. 2c)

The method is divided into two main steps. The first one is the creation of a *Tunnel Network Graph* (TNG), a graph which contains the topological and geometrical information of the final Tunnel Network environment. The TNG is defined by a set of *Tunnel Graphs* (TGs) which are, in turn, a collection of nodes in 3D space, that are interconnected in a chain-like manner. For two TGs to be in the same TNG, they must share at least one node, which is referred to as an Intersection node.

The basic structure of TGs and TNGs can be created specifying the 3D position of their Nodes or by using the GUI presented in Sec. III-B. Alternatively, it can be procedurally generated as explained in Sec.III-C.

Over the initial structure, alterations can be made in a consistent or randomized way. For example, given an arbitrarily complex network, it is possible to randomly discard some of the edges to create different scenarios for exploration (with the possibility of guaranteeing the presence of a path between entrance and exit).

The second step consists on generating a mesh that is geometrically and topologically consistent with the Tunnel Network Graph. To generate this mesh, we first generate a Point Cloud that outlines the shape of the Tunnel Network, and add distinct geometrical features using Perlin Noise, as outlined in Sec.IV-.2 and Sec.IV-.3. Finally, the point cloud is fed into a meshing algorithm (details in Sec.IV-.4), the output of which is the final mesh that represents an underground environment. In this second step, the aspect of the environment can be altered with a set of parameters as, for example, the roughness of each individual tunnel or that of the tunnel network as a whole.

```
from subt_proc_gen.tunnel import TunnelNetwork,
    Tunnel, Node
from subt_proc_gen.mesh_generation import
    TunnelNetworkMeshGenerator
from random import randrange

tn = TunnelNetwork(initial_node=False)
n1 = Node(0, 0, randrange(-5, 5))
n2 = Node(25, 0, randrange(-5, 5))
n3 = Node(50, 0, randrange(-5, 5))
n4 = Node(70, 20, randrange(-5, 5))
n5 = Node(50, 40, randrange(-5, 5))
n6 = Node(25, 40, randrange(-5, 5))
n7 = Node(0, 40, randrange(-5, 5))
t1 = tn.add_tunnel(Tunnel(nodes=(n1, n2, n3, n4, n5
    , n6, n7)))
t2 = tn.add_tunnel(Tunnel(nodes=(n2, n6)))
t3 = tn.add_tunnel(Tunnel(nodes=(n3, n5)))
tnmg = TunnelNetworkMeshGenerator(tn)
tnmg.compute_all()
```

Listing 1: Example Python script that generates environments from a set of pre-defined nodes.

A. Tunnel Network Graph from node Positions

The first option to generate a simple Network graph is the use of the plain library. For example, the code shown in List 1 will create a TNG with seven nodes (their position, in meters, is specified in the constructor) belonging to three TG. The first TG owns the seven nodes while the other TGs link two nodes each, creating the environments shown in Fig. 3.

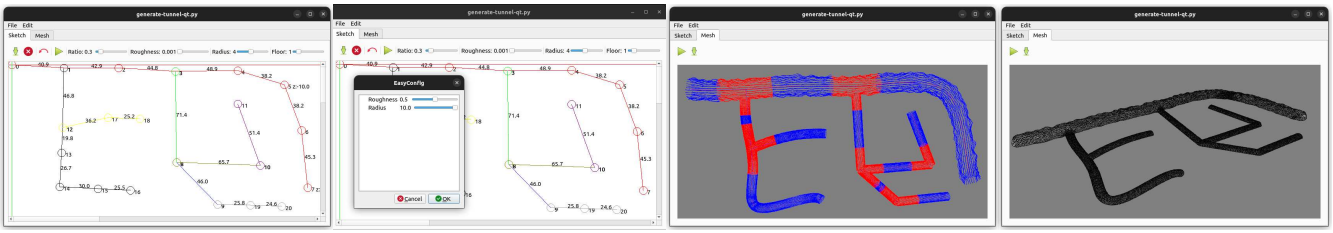
B. Graphical User Interface

A similar result can be obtained using the provided graphical user interface written in Python using the PyQt library. It allows for the TNG to be defined manually by the user through the creation of TGs specifying the position of the nodes. Such TGs can be sketched using the mouse (see Fig. 4a); each click will generate a new node that can subsequently be moved by dragging it while holding the Ctrl key. Right clicking one of the nodes will show a popup that will offer the possibility of specifying the z coordinate of the node (allowing the creation of complex $3D$ graphs). New TGs can be initiated from nodes already in the TNG holding the Shift key while clicking on any node. There is also the possibility of scaling the sketch to make it larger or smaller; the length of the edge between two connected nodes is shown in the segment that connects them. Also, each Tunnel's parameters (roughness and radius) can be set individually by right-clicking on any of its nodes (Fig. 4b).

Once the TNG has been sketched, clicking on the *Play* button will trigger the generation and visualization of the Tunnel Network Point Cloud in the adjacent tab (see Fig. 4c). Once satisfied with the result, the user can generate the final mesh by clicking on the *Render* button (see Fig. 4d). The process usually takes between a few seconds and less than a minute depending on the size of the Tunnel Network.

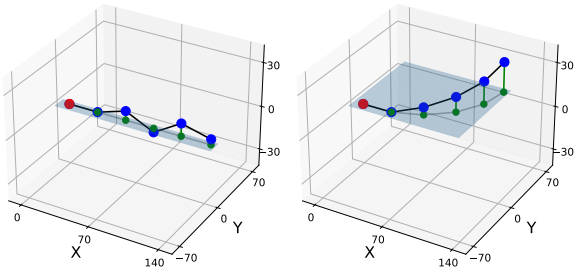
C. Procedural Tunnel Graph Generation

The third option allows to procedurally generate TNGs from iteratively adding randomly generated TGs. To this



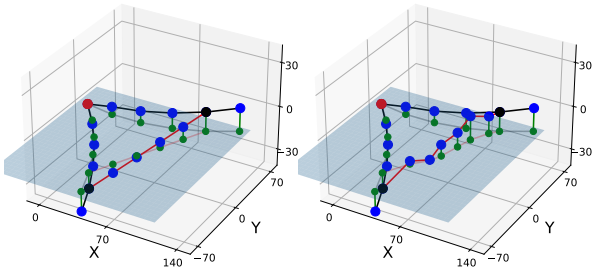
(a) (b) (c) (d)

Fig. 4: The GUI tool.(a) Sketching tab. (b) Individual tunnel configuration (increase of the roughness and size of the red tunnel). (c) Point cloud preview. (d) Mesh preview.



(a) (b)

Fig. 5: RGTG generation. The blue surface is in the xy plane, the red dot is the initial node, the blue dots are the nodes (projected to the xy plane as green), the black lines the edges (projected to the xy plane as gray). For all examples, L $= 150$, BSL $= 30$ and the other parameters are 0 unless specified otherwise. (a) VN $= 20$. (b) HT $= 20$, VT $= 10$.



(a) (b)

Fig. 6: CTG generation. The black dots are the IN and FN. The red lines are the edges of the connector TG (projected as light red to the xy plane). (a) MSL $= 30$ and NPN $= 0$. (b) MSL $= 20$ and NPN $= 10$.

end, the framework offers the possibility of generating and adding to a TNG the *Randomly Grown* TGs (RGTG) that grow randomly starting from an initial node, and *Connector* TGs (CTG) that connect two nodes of the same or different tunnels.

1) Generating Randomly Grown TG: The process of Generating Randomly Grown TG consists in adding nodes sequentially, creating segments that connect each node to the previous one following a set of parameters: the *Horizontal Tendency* (HT) and *Horizontal Noise* (HN) that control the relative yaw between two consecutive segments, the *Vertical Tendency* (VT) and *Vertical Noise* (VN) that control the inclination of the segments and the *Base Segment Length* (BSL) and *Segment Length Noise* (SLN) that control the average distance between nodes until reaching the desired length *Length* (L). Fig. 5 shows some examples of the resulting Tunnel Graphs.

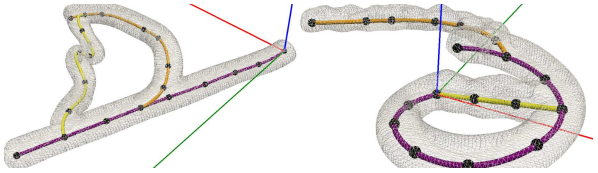
2) Generating Connector TG: In this case, the TG is created by generating nodes between the initial node IN and the final node FN. These are initially placed on the segment that links IN and FN (evenly separated from each other according to *Mean Segment Lenght* (MSL)). Then, their position is randomized by an amount dependent on the parameter *Node Position Noise* (NPN). Fig. 6 shows two examples of CTGs from the same two nodes, were it can be seen that the lower the MSL and higher the NPN, the more tortuous the Tunnel ends up being.

```
from subt_proc_gen.tunnel import TunnelNetwork,
    Tunnel, Node
from subt_proc_gen.mesh_generation import
    TunnelNetworkMeshGenerator

tn = TunnelNetwork(initial_node=False)
n1 = Node(0, 0, randrange(-5, 5))
while True:
    t1 = Tunnel.grown(i_node=n1)
    if not tn.check_collisions(t1):
        tn.add_tunnel(t1)
        break
while True:
    t2 = Tunnel.grown(i_node=tn.get_random_node())
    if not tn.check_collisions(t2):
        tn.add_tunnel(t2)
        break
while True:
    ni = tn.get_random_node()
    nf = tn.get_random_node()
    t3 = Tunnel.connector(inode=ni, fnode=nf)
    if not tn.check_collisions(t3):
        tn.add_tunnel(t3)
        break
tnmg = TunnelNetworkMeshGenerator(tn)
tnmg.compute_all()
```

Listing 2: Example Python script that generates environments with two connector tunnels and one grown tunnel. All parameters are assigned randomly at run-time.

Listing 2 shows a way of using these two primitives to generate a random TNG with two RGTG and one CTG.



(a) (b)

Fig. 7: Meshes generated from code in Listing 2. The purple, orange and yellow splines correspond to `t1`, `t2` and `t3` respectively.

This script begins with by creating node at from which the first RGTG is generated. Then a random node of the first RGTG is chosen and second one is generated from it. Then two random nodes are sampled from the TNG and a CTG is generated between them. Note how the script checks for collisions before adding a new tunnel, as the random nature of their generation could make two tunnels intersect at a point where it is not expected. This check ensures that the final mesh will be topologically consistent with the Tunnel Network. Examples of environments generated with this script can be seen in Fig. 7

IV. Point Cloud and Mesh Generation

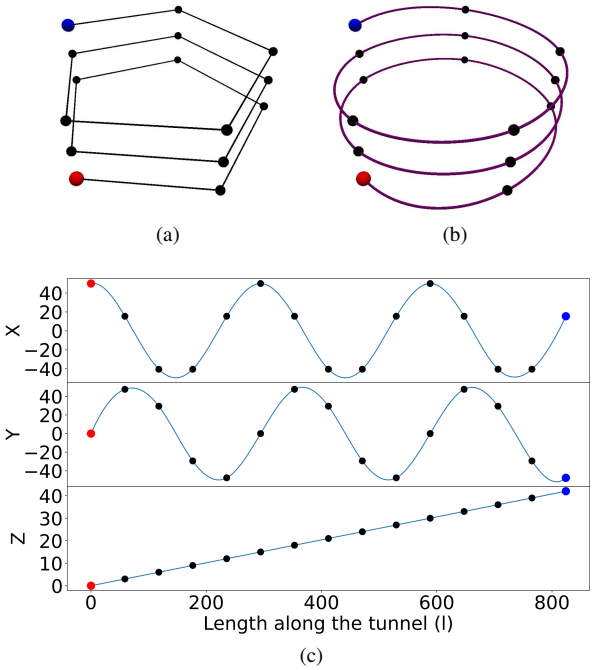
Once the topological structure of the final environment has been defined in the TNG, it is necessary to create a mesh around it, so that it can be used in simulation. This process is divided into four parts. Firstly, the Splines that represent the axes of the resulting tunnel networks are computed. Secondly, the point cloud is generated around the Splines. Then, this point cloud is refined at the intersections, removing excess points. Finally, the refined point cloud is fed into the mesh-generation algorithm.

1) Spline generation: At this point, the axis of a tunnel could be defined as the edges between its nodes. However, this would result in tunnels composed of straight sections (Fig. 8a), and achieving a smooth curve would be challenging. The axis of each tunnel is represented as a 3D Spline, fitted to the position of the nodes (Fig. 8b).

This 3D spline is defined as 3 separate B-Splines [23], each fitted to one of the xyz coordinates, and parameterized by the position along the tunnel l. To fit these B-splines, an l value has to be assigned to each node of the tunnel. In our case, the l coordinate of a node in a tunnel is the sum of the length of the segments from that node to the first node of the tunnel. This means that, the first node of the tunnel has $l = 0$ and the final node of the tunnel has $l = L$, being L the total length of the tunnel.

After each node in a tunnel has an l coordinate the three B-Splines can be fitted by using the lx, ly, lz pairs of coordinates, resulting in the curves shown in Fig.8c). Querying these 3 splines at the same l yields the xyz coordinates of the tunnel's axis along with its tangent direction at that point.

This parameterized representation makes straightforward to uniformly sample the spline by taking an equally spaced set of l coordinates, and obtaining an equally spaced set xyz coordinates, along with the tangent directions of the spline at those points.



(a) (b) (c)

Fig. 8: Tunnel Spline Illustration. (a) TG of the tunnel, initial node in red and final node in blue. (b) Spline of the Tunnel. (c) Individual x,y and z splines fitted to the coordinates of the nodes (black dots), and parameterized with the length along the tunnel in the horizontal axis.

2) Point cloud generation: The initial point cloud is the result of generating rings of points around the splines of the TGs.

To generate these rings for a given tunnel, a set of N equally-spaced points ($p_i, i \in [1, N]$) are selected from the TG's spline, along with the direction vector of the spline at said points ($v_i, i \in [1, N]$), as can be seen in Fig. 9a. Then, for each p_i, a set of M points ($q_{ij}, j \in [1, M]$) in the shape of a circumference with radius r (chosen by the user) centered at p_i and in the plane perpendicular to v_i are created as shown in Fig. 9b.

To add some texture to the final Tunnel (if required), Perlin noise [24] is added to the radius of the circumference (see Fig. 9c). The frequency of the noise determines how irregular the Tunnel will end up being. Given that Perlin noise is spatially coherent, it is necessary to map each of q_{ij} to a unique 2D coordinate that will be used to query the noise generator. In our case this 2D mapping is the equivalent of unwrapping the points around the axis into a rectangle of length equal to the length of the axis, and width equal to the circumference.

Finally, if required, a floor is created by setting all the points q_{ij} that are below p_i by more than a certain threshold, to the same height (see Fig. 9d). This step by itself creates perfectly smooth floors, but this is not realistic in many underground environments, so it is also possible to add

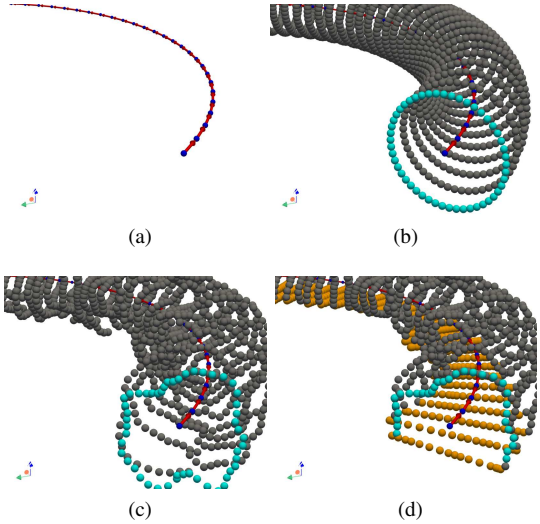


Fig. 9: Point cloud generation for a Tunnel. (a) Points of the Tunnel spline (blue dots) and corresponding spline direction (red arrow). (b) Point cloud obtained from placing points in a circumference around the spline points (first circumference in green for clarity). (c) Point cloud after adding the Perlin noise. (d) Final point cloud after flattening the floor points (orange).

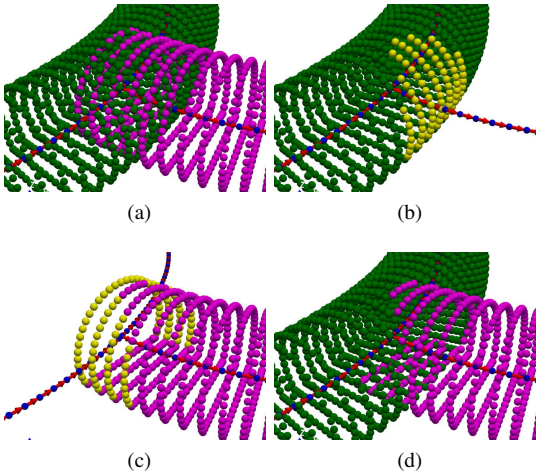


Fig. 10: Cleaning the point clouds of two intersecting tunnels. The Tunnels have no Perlin noise for clarity. (a) Intersection of two Tunnels before removing the excess points. (b) Points to remove from the green Tunnel. (c) Points to remove from the magenta Tunnel. (d) Intersection after removing the points.

random variation to the position of these floor points, adding irregularity to the floor.

3) Cleaning of the Intersections: The next step is to refine the point cloud. This step is necessary because in the intersections between tunnels there are excess points (highlighted in yellow in Fig. 10b and Fig. 10c). If these points were present at the mesh-creation step, a wall would appear where it is not supposed to (according to the TNG). To identify these points we use a simple rule: if a point belonging to tunnel is inside any other tunnel of the intersection, it is removed. In Fig. 10d, the point cloud that results from eliminating the highlighted points is shown.

4) Generation of the Mesh: After all the intersections in the network have been cleared, all the individual point clouds are combined into one. This final point cloud is then transformed into a mesh using the Poisson surface reconstruction method [25]. In our case, we use the implementation provided by the Open3D library [26].

V. Applications

Although the method proposed relies exclusively on mathematics methods to generate the environment and has been mainly conceived to create tunnels, the addition of Perlin noise allows the creation of both realistic tunnel-like and cave-like environments. Even if a comparison with other methods and with real-world ground truth is not an easy task, we noticed that AI models trained exclusively in environments generated using this framework have been able to easily generalize to real-world environments.

Our system produces a file in one of several common mesh formats (`.obj`, `.dae`, `.ply`), which is straightforward to import into any simulation software, allowing the creation of completely automated pipelines with on-the-fly generated training or testing environments (Fig. 11). It is also easy to modify the resulting meshes with external software to meet more specific needs. For example, Fig. 12 shows a mesh to which a texture has been added, along with some rocky obstacles.

For this reason, this tool has already proved very useful in our own research, and has been a key element of the dataset generation pipelines used in training deep-learning based approaches for robotic navigation in underground environments. The fact that the topological structure of the generated mesh, along with the geometric representation of the tunnel axis can be accessed, makes it possible to completely automate the labeling process, and because of this, very large and varied datasets can be generated with a small input from the user.

An example use-case can be found in [4]. It presents a high-speed tunnel traversal method based on predicting the heading of the robot that would allow the robot to advance centered in the tunnel itself. To do so, the robot is required to adjust its heading to reach the axis of the tunnel and then meet its direction while it advances. This is done by feeding the readings of a 2D LiDAR sensor to a Convolutional Neural Network (CNN) that is in charge of computing the current orientation of the robot with respect to the tunnel axis. To train such a network, we used an automated pipeline based on this framework as follows: in each iteration 1) a simple RGTG tunnel is generated and imported into Gazebo, then 2) a random point of the Spline is selected and the robot is artificially moved to that point with an orientation that matches the tangent in that point as shown in Fig.13a (notice that this information is provided by the framework itself

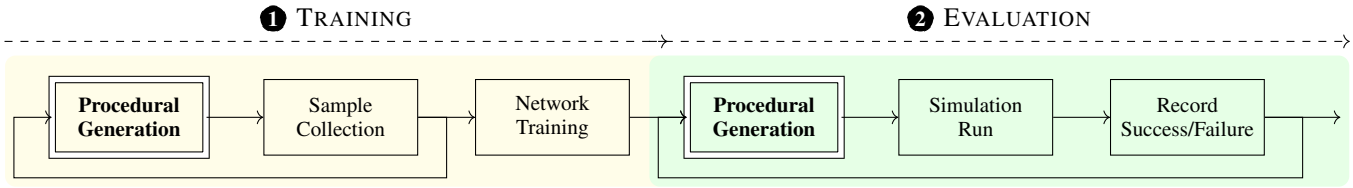


Fig. 11: Steps in which we have used procedural generation in some of our fully automatic training and evaluation workflows.

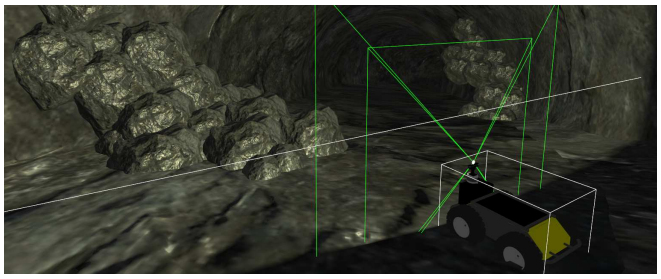


Fig. 12: Tunnel Mesh in Ignition Gazebo with added texture and obstacles.

since the Spline is the base on which the tunnel is built as explained in Sec. IV-.1). Then 3) the robot is rotated by a random amount of degrees (Fig. 13b); such an amount is the value we used as the label for the training process. Successively, 4) the robot is moved a random amount in the y-axis (Fig. 13c) and finally 5) a LiDAR reading is acquired (Fig. 13d); This information is fed to the CNN together with the label. Steps 2-4 are repeated automatically several hundred times before the next iteration where another tunnel is procedurally generated and the process continues.

Given the modular nature of our system, obtaining variations on the same Tunnel Network can be easily done by changing the meshing parameters. This is displayed in Fig. 14, where 2 different meshes have been generated from the same Tunnel Network.

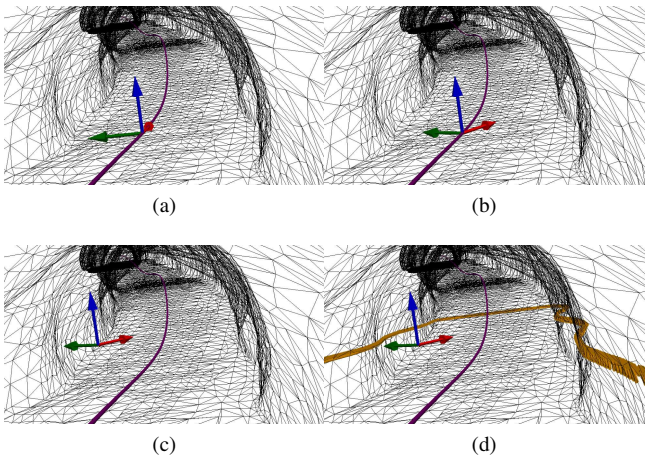


Fig. 13: Use of the proposed method to generate a dataset for fast tunnel traversal. (a) Selected Axis Point. (b) Applied yaw which is equal to the label. (c) Applied translation perpendicular to the axis. (c) Captured LiDAR reading.

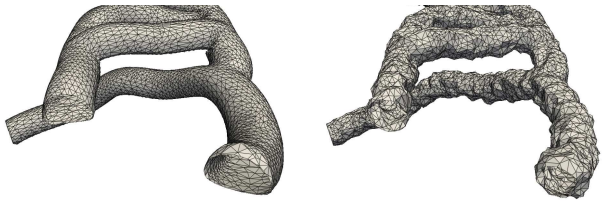


Fig. 14: Different meshes generated from the same graph. (a) No Perlin Noise, Radius of 3 and the floor is Flattened. (b) Perlin noise of high frequency and radius of 2 meters.

This modularity also facilitates the development of other procedural generation systems based on it. As an example we propose a system to benchmark different robotic applications for search-and-rescue missions in mines. In these scenarios it is feasible that the the environment changes after an accident, for example, in the case of a cave-in, some tunnels could get blocked. To emulate this with our system, it is possible to define the base structure of the Mine (in this example, a simple grid shown in Fig. 15a), an initial position and a final objective. For each test, a set of tunnels are removed while ensuring connJectivity between the initial position and final objective (very easy to check with the TNG), so that each trial would be different, ensuring the reliability of the solution.

VI. CONCLUSION

This work presented a flexible approach to generate tunnel networks suitable for underground robotics simulations, since experimental campaigns in such hostile environments are often challenging, costly, and even hazardous. These characteristics and the need for a large variety of data sources for machine learning applications prompted us to use procedural generation to be able to create as many different simulations as needed.

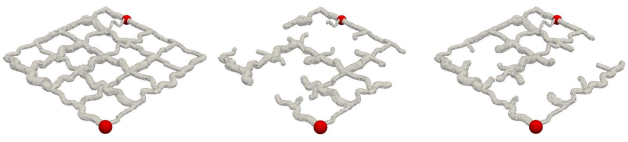


Fig. 15: Example of a possible use of the application. (a) Base structure of the environment. Final and initial positions are in red. (b)(c) Testing environments with tunnels removed.

The method starts with a graph representation of an underground environment. Users can create this graph programmatically or with the included graphical user interface, which allows quick and simple creation of Tunnel distributions. Additionally, graphs can be randomly grown one gallery at a time, to mimic human-made underground structures. This graph is then used to generate the Tunnel mesh, according to geometric information associated to the graph nodes. The Tunnels can be further randomized in shape and size through noise parameters. The resulting environments are ready for importing in simulators typically used in robotics research.

Examples of the kind of scenarios that can be obtained with our method are available through the paper. We are already applying these results in our research on high-speed tunnel navigation for drones and Unmaned Ground Vehicles with limited sensing capabilities, and Topological Navigation of ground robots in underground environments, where it has proven to be an effective way to generate scenarios to train and test our algorithms in simulation. The source code[2] is made available in the hope it can be useful to other members of the robotics community.

ACKNOWLEDGMENTS

This work was supported by the Spanish projects PID2019-105390RB-I00, and DGA FSE-T45 20R.

REFERENCES

[1] A. Loquercio, E. Kaufmann, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza, "Deep drone racing: From simulation to reality with domain randomization," *IEEE Transactions on Robotics*, vol. 36, no. 1, pp. 1–14, 2019.

[2] L. Cano, A. R. Mosteo, and D. Tardioli, "Navigating underground environments using simple topological representations," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 1717–1724.

[3] A. Koval, C. Kanellakis, E. Vidmark, J. Haluska, and G. Nikolakopoulos, "A subterranean virtual cave world for Gazebo based on the DARPA SubT challenge," http://arxiv.org/abs/2004.08452, 2020.

[4] L. Cano, D. Tardioli, and A. R. Mosteo, "Fast tunnel traversal for ground vehicles by relative yaw estimation with neural networks," in *ROBOT2023: Sixth Iberian Robotics conference*. Springer, 2023.

[5] D. González-Medina, L. Rodríguez-Ruiz, and I. García-Varea, "Procedural city generation for robotic simulation," in *ROBOT2015: Second Iberian Robotics Conference*. Springer, 2016, pp. 707–719.

[6] I. Paranjape, A. Jawad, Y. Xu, A. Song, and J. Whitehead, "A modular architecture for procedural generation of towns, intersections and scenarios for testing autonomous vehicles," in *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2020, pp. 162–168.

[7] O. Roglà Pujalt, "Procedural modeling of cities with semantic information for crowd simulation," Master's thesis, Universitat Politècnica de Catalunya, 2016.

[8] G. Muktadir, A. Jawad, I. Paranjape, J. Whitehead *et al.*, "Procedural generation of high-definition road networks for autonomous vehicle testing and traffic simulations," *SAE International Journal of Connected and Automated Vehicles*, vol. 6, no. 1, p. 2023, 2022.

[9] C. Campos, J. M. Leitão, J. P. Pereira, A. Ribas, and A. F. Coelho, "Procedural generation of topologic road networks for driving simulation," in *2015 10th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, 2015, pp. 1–6.

[10] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin, "Procedural generation of roads," in *Computer Graphics Forum*, vol. 29. Wiley Online Library, 2010, pp. 429–438.

[11] L. Z. Kelvin and B. Anand, "Procedural generation of roads with conditional generative adversarial networks," in *2020 IEEE Sixth International Conference on Multimedia Big Data (BigMM)*. IEEE, 2020, pp. 277–281.

[12] E. Teng and R. Bidarra, "A semantic approach to patch-based procedural generation of urban road networks," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017, pp. 1–10.

[13] C. Rizzo, D. Tardioli, D. Sicignano, L. Riazuelo, J. L. Villarroel, and L. Montano, "Signal-based deployment planning for robot teams in tunnel-like fading environments," *The International Journal of Robotics Research*, vol. 32, no. 12, pp. 1381–1397, 2013.

[14] D. Tardioli, L. Riazuelo, D. Sicignano, C. Rizzo, F. Lera, J. L. Villarroel, and L. Montano, "Ground robotics in tunnels: Keys and lessons learned after 10 years of research and experiments," *Journal of Field Robotics*, vol. 36, no. 6, pp. 1074–1101, 2019.

[15] K. Ebadi, Y. Chang, M. Palieri, A. Stephens, A. Hatteland, E. Heiden, A. Thakur, N. Funabiki, B. Morrell, S. Wood *et al.*, "LAMP: Large-scale autonomous mapping and positioning for exploration of perceptually-degraded subterranean environments," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 80–86.

[16] M. Li, H. Zhu, C. Tang, S. You, and Y. Li, "Coal mine rescue robots: Development, applications and lessons learned," in *International Conference on Autonomous Unmanned Systems*. Springer, 2021, pp. 2127–2138.

[17] J. Cui, Y.-W. Chow, and M. Zhang, "Procedural generation of 3D cave models with stalactites and stalagmites," *International Journal of Computer Science and Network Security*, 2011.

[18] R. Van Der Linden, R. Lopes, and R. Bidarra, "Procedural generation of dungeons," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 1, pp. 78–89, 2013.

[19] B. Mark, T. Berechet, T. Mahlmann, and J. Togelius, "Procedural generation of 3D caves for games on the GPU." in *Foundations of Digital Games*, 2015.

[20] I. Antoniuk and P. Rokita, "Procedural generation of underground systems with terrain features using schematic maps and L-systems," *Challenges of Modern Technology*, vol. 7, no. 3, pp. 8–15, 2016.

[21] G. Leão, R. Camacho, A. Sousa, and G. Veiga, "An inductive logic programming approach for entangled tube modeling in bin picking," in *ROBOT2022: Fifth Iberian Robotics Conference*, D. Tardioli, V. Matellán, G. Heredia, M. F. Silva, and L. Marques, Eds. Cham: Springer International Publishing, 2023, pp. 79–91.

[22] L. Cano, D. Tardioli, and A. R. Mosteo, "Procedural generation of underground environments for Gazebo," in *ROBOT2022: Fifth Iberian Robotics Conference*, D. Tardioli, V. Matellán, G. Heredia, M. F. Silva, and L. Marques, Eds. Cham: Springer International Publishing, 2023, pp. 314–324.

[23] C. De Boor and C. De Boor, *A practical guide to splines*. springer-verlag New York, 1978, vol. 27.

[24] K. Perlin, "Improving noise," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002, pp. 681–682.

[25] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *Proceedings of the fourth Eurographics symposium on Geometry processing*, vol. 7, 2006, p. 0.

[26] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.

[2] https://github.com/LorenzoCanoAn/procedural-subt-gen