Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare

LUCRARE DE DIPLOMĂ

Generare date sintetice folosind 3D rendering

Tudor-Gabriel Vîjială

Conducător Științific:
Conf. Dr. Ing. Iuliu Vasilescu

București, 2022

University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Synthetic data generation using 3D rendering

Tudor-Gabriel Vîjială

Scientific Adviser:
Conf. Dr. Ing. Iuliu Vasilescu

Bucharest, 2022

I had the pleasure of collaborating with project supervisor Conf. Dr. Ing. Iuliu Vasilescu, whom I thank for the unyielding support. I am also thankful to Alexandru Petre using my synthetic data generator when training his machine learning models.

This endeavor would not have been possible without the help of the Blender and Google Research open-source communities, given through their respective public developer forums.

Sinopsis

Antrenarea modelelor de învățare automată de tip "deep learning" necesită cantități mari de date de înaltă calitate. Cu toate acestea, obținerea de adnotări pentru date reale are un cost ridicat și prezintă probleme legate de diversitatea datelor și de licențiere. Generarea de date sintetice evită unele dintre aceste limitări, dar diversitatea datelor rămâne nesatisfăcătoare atât timp cât sunt utilizate modele 3D statice. Pentru a rezolva această problemă, prezentăm The Procedural Outdoors Scene Generator, un proiect Python care combină date din diverse Sisteme informaționale Geografice cu tehnici de modelare procedurală, pentru a genera scene realiste cu o înaltă diversitate și variație a datelor. Demonstrăm eficacitatea sistemului prin generarea unor seturi de date care să fie utilizate pentru antrenarea unui model de învățare automată care detectează șine de cale ferată.

Abstract

Deep learning requires vast amounts of high quality data. However, obtaining annotations for real data has a high cost, and presents issues with data diversity and licensing. Synthetic data generation avoids some of these limitations, but data diversity remains lacking as long as static 3D models and assets are used. To address this problem, we introduce The Procedural Outdoors Scene Generator, a Python framework that combines data from various Geographic Information Systems with procedural modeling techniques, to generate realistic scenes with high data variance and diversity. We demonstrate the effectiveness of the system by generating datasets to be used for training a machine learning model that detects railroad tracks.

Contents

Chapter 1

Introduction

Deep learning requires vast amounts of high quality data[15]. Obtaining quality annotations from real-world data is a high-cost endeavor, and presents issues with data diversity and licensing[1]. To overcome these limitations, we advocate the use of synthetic data for deep learning vision tasks.

To this end, we combined data from open access GIS (Geographic Information System) services with procedural modeling techniques, to develop The Procedural Outdoors Scene Generator[2], an open-source Python framework aimed at generating realistic synthetic video data of outdoor environments. We used this system to create a realistic scene with rail tracks, together with per-pixel annotation data. The resulting datasets are evaluated for usefulness in training a machine learning model for semantic segmentation of the tracks.

We discuss the general approach in Chapter 3, implementation details in Chapter 4, and evaluate the benefits of using data generated from this system for a specific machine learning segmentation task in Chapter 5. Finally, a long-term plan is made in Chapter 7 to extend the framework into a comprehensive ecosystem of freely licensed procedurally generated assets and scenes.

The next section explains the design considerations that were taken into account when deciding on the approach.

1.1 Design Requirements

Makes use of GIS data. Publicly available Geographic Information Systems (GIS) distribute a variety of data, such as satellite imagery, altitude maps, and data on streets, highways, railroads, buildings and urban zoning areas. The framework includes a method to access and combine multiple data sources into a single 3D environment object containing the raw data for generating a scene.

Capable of procedural 3D modeling. Specialized 3D artists can create and modify procedurally generated 3D assets, including their textures and shaders, independently of the research team which implements the simulation logic in Python. Procedural 3D modeling can also be used to create variations of existing static 3D objects and their texture and shader information.

Open. The framework must be based on open-source code, and enable the sharing, distribution and modification of both user code and procedurally generated 3D objects. The framework is based on the Kubric[9] library, which uses the Blender 3D computer graphics software

[2] https://github.com/gabriel-v/the_procedural_outdoors

as the rendering backend. Both these components are free and open-source, which makes it possible to analyze and modify any component, including the rendering engine.

Portable and Scalable. All components of the framework are easily deployable on cloud hardware. Rendering tasks can be parallelized by running multiple instances of the same parameter settings.

Interactive Rendering. The framework must support tasks in which the machine learning algorithm needs to effect changes in the scene. An example of this would be an online reinforcement learning model learning to guide a drone over rail tracks: at each frame, the training algorithm decides of a movement, and the simulation must update its scene based on that movement.

Chapter 2

Related Work

A common source of realistic synthetic data is found in video games. We discuss the use of video games for this purpose in Section 2.1.

2.1 Synthetic Data Generation with Video Games and Other Software

Video games present the lowest barrier of entry in the domain of synthetic video data generation. This is because they combine realistic graphics meant to suspend human disbelief with high-performance rendering engines that achieve fast render times in large resolutions, on commodity hardware.

Figure 2.1: LIDAR Simulation using GTA V[11]

The most popular video game platform for generating synthetic video for deep learning datasets is, by far, "GTA V" by Rockstar Games. This platform was used for semantic segmentation[21], LIDAR simulation [11], crowd counting [25], and pedestrian detection and tracking[8]. Other off-the-shelf software programs have also been used for image tasks, such as Google Earth [16].

Figure 2.2: Synthetic Aerial 3D Dataset Generation using Google Earth[16]

Using video games for this task, however, has a number of disadvantages. Firstly, all video games have a specific artistic style in their renditions of reality, which becomes another task that must be solved: domain adaptation [27, 25]. Other issues include lack of control in the simulation, inability to add or edit models and environments, and a need to reverse-engineer proprietary, closed source software. This led to the development of specialized software meant to extract data from specific games [7].

An alternative to closed-source programs and video games is the use of game engines. These afford the user complete control over what, and how, is being rendered. We discuss the use of game engines for synthetic video generation tasks in the next section.

2.2 Synthetic Data Generation with Game Engines

Game engines are software frameworks meant to design and implement video games. As such, they give the user total control over how the simulation renders and runs. Popular game engines used for synthetic dataset generation include the proprietary Unity[1] and Unreal[2] engines, as well as the open-source project Blender[3].

Applications of game engines in synthetic video dataset generation include UAV monitoring and detection [2], procedural urban and outdoor environments [22, 14, 23], and indoor environments [17]. Applications include tracking construction workers[18], self-driving car obstacle avoidance[5, 20], or more generally rendering generic environments with people[13].

Figure 2.3: Synthetic scenes for tracking construction workers using Blender[18]

Of the more general-use projects that build upon game engines, we bring attention to Kubric [9], a project concerned with realism, scalability and flexibility.

Since a scene is only as complex as its included 3D assets, the data diversity generated by game engines with static assets is limited in scope. To create the large variance needed to

[1] https://unity.com/
[2] https://www.unrealengine.com/en-US
[3] https://www.blender.org/

train a robust machine learning model, procedurally generated graphics are used. Various techniques for generating 3D objects procedurally are discussed in the following section.

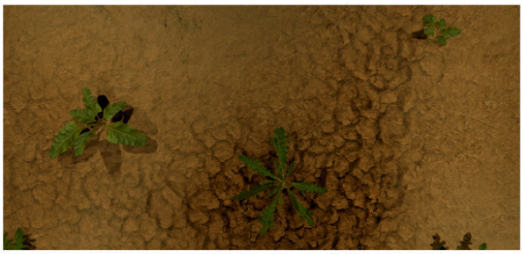
2.3 Procedurally Generated Graphics and Environments

Using procedurally generated models, geometries and materials in synthetic video data ensures a high degree of data variance, with no two objects looking exactly the same. This is achieved through modeling a randomization of object attributes, in a highly controllable way.

Projects that make use of procedurally generated objects include generative models for agricultural plant models [6, 3], trees [10], volumetric clouds [4], and even entire urban planning areas generated with probabilistic grammars[12].

Other projects use physical modeling of light to procedurally re-interpret existing images to create highly realistic images from existing synthetic datasets [24, 26].

All game engines mentioned in Section 2.2 have an implementation for procedural modeling. Unity has "Procedural Mesh Geometry"[1] and "Procedural Materials" [2], Unreal has "Procedural Mesh" [3], and Blender has "Geometry Nodes" [4] and "Shader Nodes"[5]. However, these features are not frequently used in synthetic data generation tasks to their fullest extent.



(a) RGB Image

(b) Artificial Ground truth

Figure 2.4: Procedural Generation of Agricultural Plants using Unreal Engine 4 [6]

Use cases for procedural modeling with game engines are mentioned in the next section.

2.4 Use Cases

Procedural modeling can be used to construct synthetic datasets of any type of outdoor scene. It requires, however, some data describing the layout of a scene: road positions, terrain maps, and building profiles. These types of data can be found in large quantities in Geographic Information Systems (GIS), which contain satellite imagery for the whole planet, as well as accurate terrain and street map information.

A notable use of procedural modeling together with GIS data is the video game "Microsoft Flight Simulator 2020", which realistically models the entirety of Planet Earth at a scale of 1 to 1, complete with trillions of trees, dynamic weather systems and accurate renditions of all major cities around the globe [6].

[1] https://docs.unity3d.com/560/Documentation/Manual/GeneratingMeshGeometryProcedurally.html
[2] https://docs.unity3d.com/560/Documentation/Manual/ProceduralMaterials.html
[3] https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/Components/ProceduralMesh/
[4] https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/index.html
[5] https://docs.blender.org/manual/en/latest/render/shader_nodes/index.html
[6] https://www.pcgamesn.com/microsoft-flight-simulator/planet-earth-simulation

Figure 2.5: Screenshot from Microsoft Flight Simulator 2020, showing volumetric clouds and procedurally modeled cityscape [4]

We propose combining GIS data with procedural modeling to obtain an outdoor scene generator with high data diversity and realistic graphics. While applications range from autonomous drone control to self-driving cars, we chose to focus our efforts on the problem of rail track semantic segmentation and anomaly detection.

The methodology used for building such a framework is outlined in Chapter 3, and the actual implementation is explained in Chapter 4.

Chapter 3

Approach

The Procedural Outdoors Scene Generator is a Python framework that enables researchers and procedural 3D modeling artists to create synthetic video data of realistic outdoor scenes. The main purpose is building a systematic way to collaboratively create outdoor scenes and use them in interactive machine learning tasks.

To generate data, the user of the framework chooses a geographical area to be rendered, defines the procedural graphics primitives to be used on the raw geographic data, and implements the logic of the simulation. The procedural graphics primitives can be re-used and shared as individual assets, alongside the Python code that controls the simulation.

3.1 Data Pipeline

The data pipeline contains three main stages: downloading geographic data from various web services; combining it with user-provided procedural geometry, materials and models into a generated 3D environment; and finally using the created environment to render full frames according to user-provided logic that drives the scenarios. This is outlined in the Data Flow Diagram in Figure 3.1.

After each of the three stages, the result is saved into a shared storage environment for repeated use in the stages below. For example, the scene generation algorithm can be run with different parameters to rearrange foliage, signage location, and all other randomized environment features, thus leading to more data variation from the same GIS data. In the same way, the render loop can be run multiple times in each scene, while varying the scene parameters for each run.

The design of each stage is discussed in the following sections.

3.2 Download Stage

A reproducible process is needed to obtain and combine GIS data from the various providers, including: satellite images from Google Maps and Bing Maps, terrain height maps from Shuttle Radar Topography Mission (SRTM), and road, highway, railway and building data from OpenStreetMap (OSM). The standardization and popularity of these services make it possible to find open-source software components for using them.

In our case, the Blender GIS plugin is used to combine multiple data sources into a single 3D Scene, containing ground texture and elevation, street and railway paths and building perimeters; see figure 3.2. The plugin also supports obtaining a variety of other data types, like urban zoning areas and water bodies.

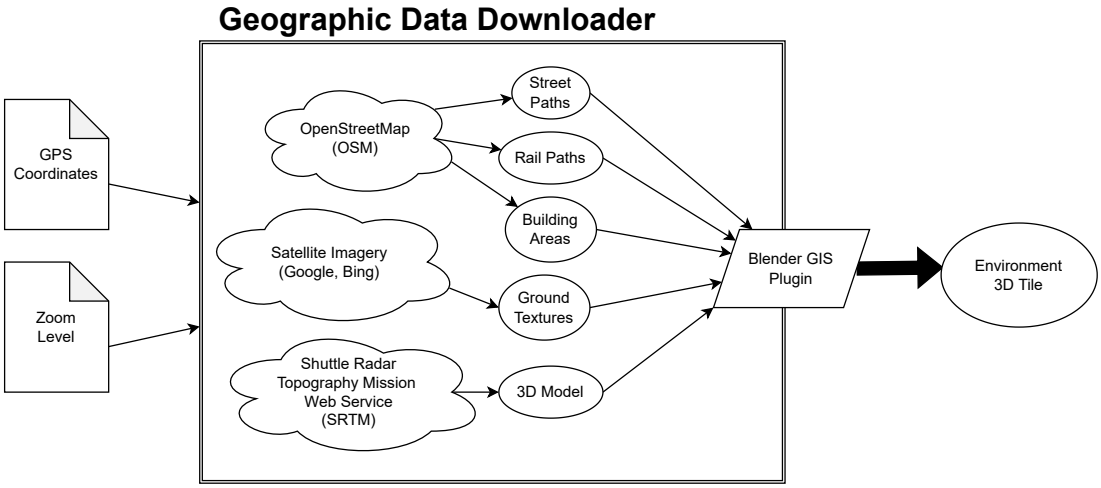


Figure 3.2: Geographic Data Downloader

Each application of the GIS data downloader requires a GPS coordinate and a zoom level, and produces a Blender file containing raw GIS data and metadata, called a 3D tile. For most GIS services, the data resolution is constant, regardless of zoom level. This means we can collect a number of tiles at different zoom levels for the same GPS location, to cover a large geographic area of the background without using a large amount of memory.

3.3 Procedural Scene Generation Stage

This stage collects the 3D tiles stored in the previous step, and combines them into a single terrain object. The data types come from different sources, they are not perfectly aligned with one another. For example, the road and railroad paths from OSM do not perfectly align with the SRTM terrain height maps, or with the Google Satellite ground textures. Because of these slight incompatibilities, this procedure also requires adjusting the terrain profile to be flat around roads, railroads and building foundations.

The parallelograms in Figure 3.3 represent user-customizable hooks in the framework, here exemplified for a railway environment generation task. The pipeline results in a single 3D environment scene, containing all static features of the scene, such as railways, roads, signage and electric poles, vegetation and buildings. The result can be stored as a Blender file prior to rendering.

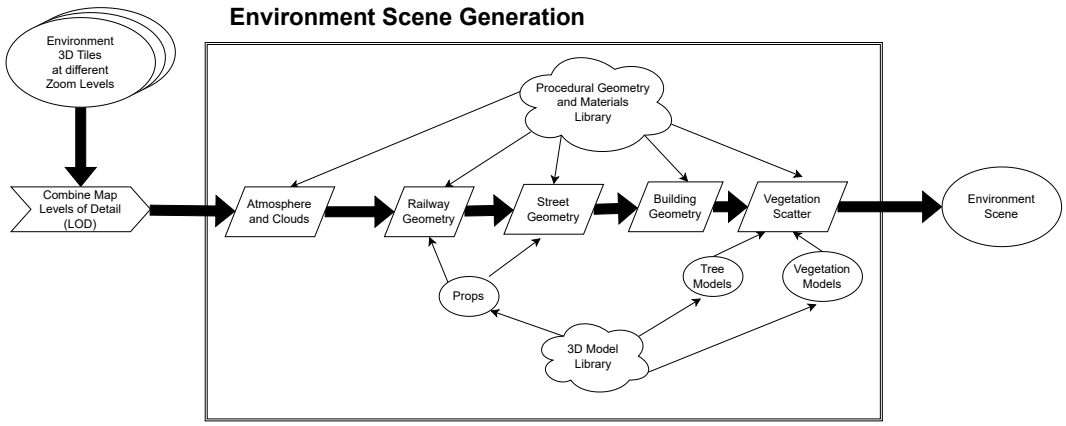


Figure 3.3: Scene Generation

3.4 Render Stage

The render stage can be used in two ways: with or without externally interacting with a training algorithm.

In the non-interactive case, this stage uses the user-supplied environment scene parameters and frame setup functions to configure each frame for rendering. Dynamic objects, such as obstacles and vehicles, are also added to the scene based on the setup function. The resulting images and metadata are kept in storage, resulting in a dataset that can be used for offline learning. The process is outlined in Figure 3.4.

In the case when some online training algorithm is used, the environment is affected by this algorithm at every step. In this case, the render stage sends its resulting images and metadata to the user-provided training algorithm. The result of this algorithm is used by the user-provided functions to update the scene parameters and then configure the next frame. The rendered images and metadata can optionally be stored for later use; see Figure 3.5.

The main difference between these two kinds of render loops is how scalability is achieved: in the non-interactive case, each scene frame can be rendered independently, provided the scene logic is deterministic. The interactive case, however, needs to scale the renderers together with the online training algorithm workers, with concurrent rendering of sequential frames not possible.

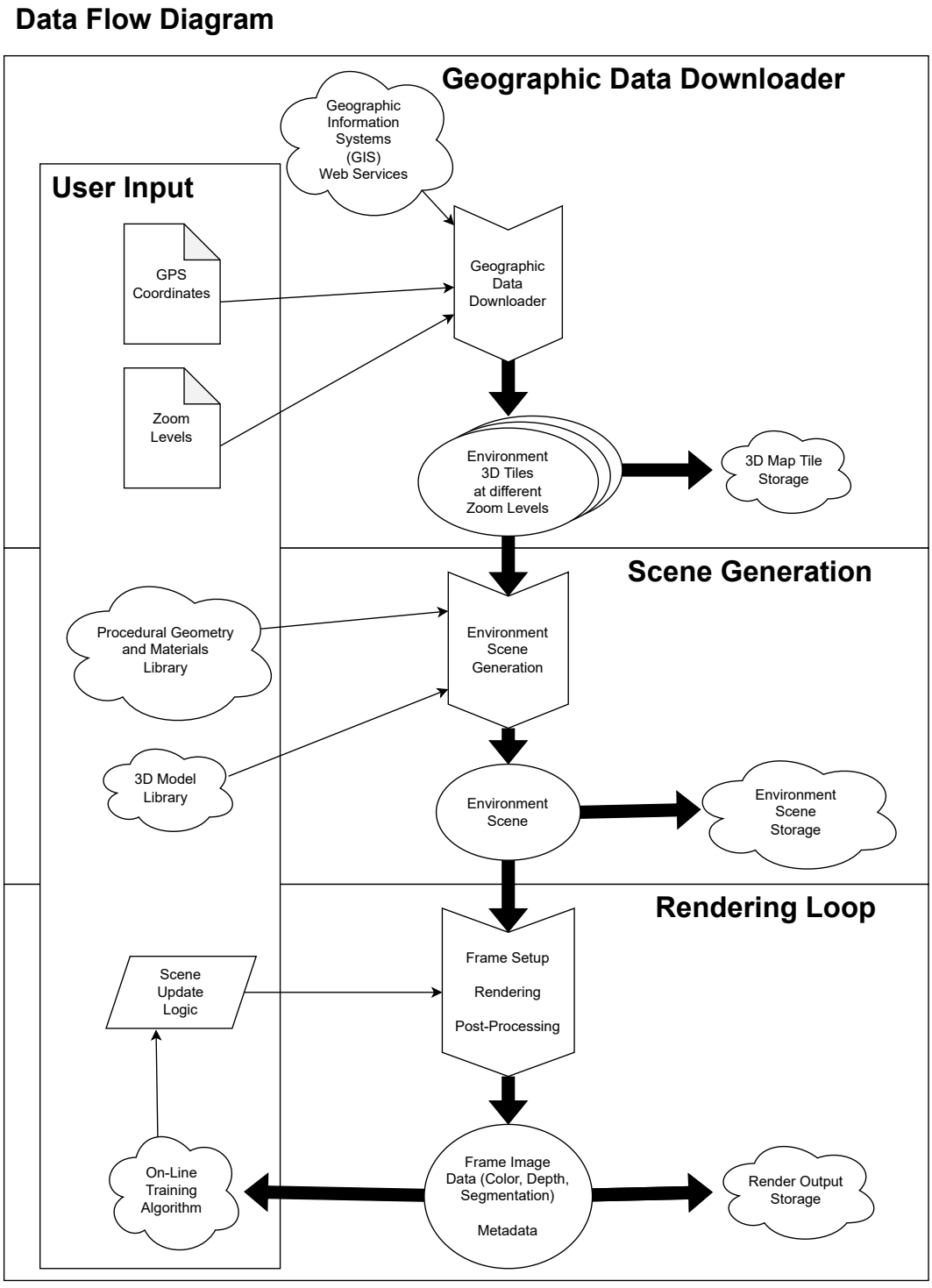


Figure 3.1: Data Flow Diagram

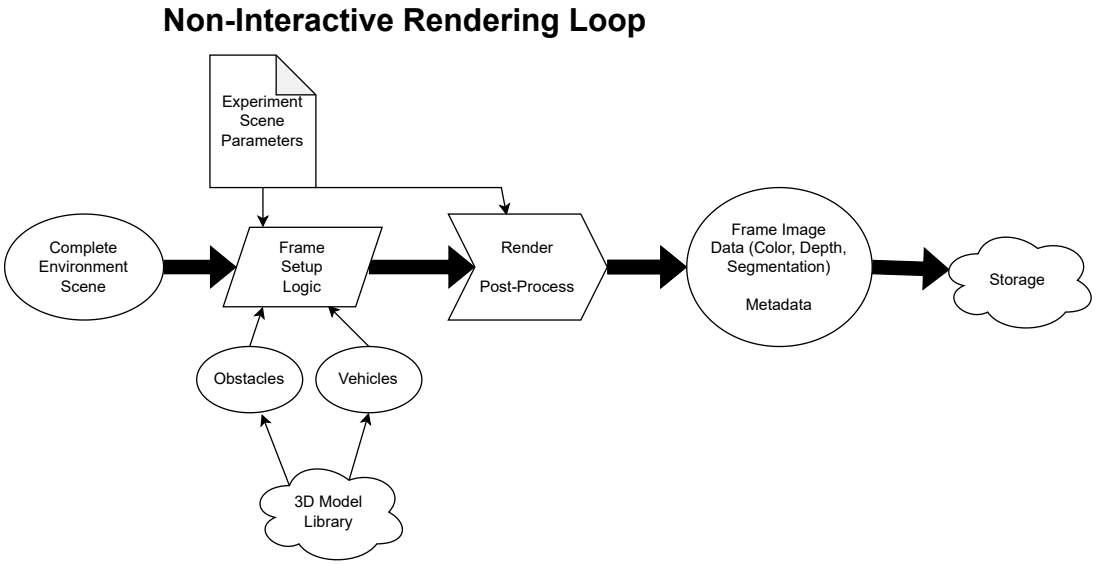


Figure 3.4: Non-Interactive Render Loop

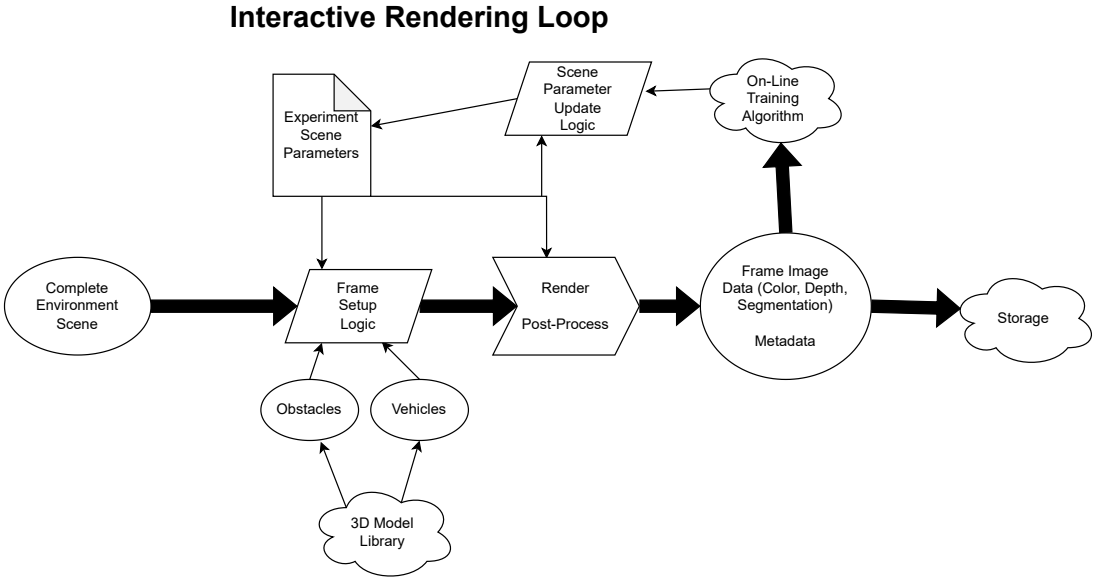


Figure 3.5: Interactive Render Loop

Chapter 4

Implementation

This chapter describes implementing the framework, as well as its first scene: an outdoors rail track segmentation task. The RailSem19 dataset with dashcam video from trains[28] is used as inspiration for designing details in the rail tracks and backgrounds, as seen in Figure 5.1. The use of the generator together with a Machine Learning Model is discussed in the next chapter, Section 5.1.

The 3D objects that the segmentation task is focused on, in this case the rail tracks and immediate area, are created using procedurally generated graphics. This ensures a great degree of variation and customization for the same object: one can randomize physical properties of each component of the object, such as changing the shape or material of the railroad ties, or randomly skipping missing planks or changing their distance.

4.1 Modifying Kubric to use Blender version 3.2

The procedurally generated graphics pipeline is implemented in Blender as a feature called "Geometry Nodes". It is a visual and node-based programming environment that allows 3D artists to create and manipulate different attributes of an object's geometry. Operations on geometry made in this way include arithmetic and vector math, conditional logic, curve resampling, mesh triangulation, material selection, ray tracing and object instancing.

The "Geometry Nodes" feature was added in Blender 2.92[2], reworked in Blender 3.0 to make designing Geometry Node Groups more abstract and functional[3]. Later releases have added more node types and performance improvements[4]. We chose to make use of Blender version 3.2[5] as the render engine. This requires updating the Kubric build process and resolving any breaking changes, since at the time of writing, Kubric uses Blender version 2.93.

The upgrade process begins with simple changes: the Python language version was upgraded to 3.10, so the base operating system for the image is increased to a newer version that packages it. Then, the Blender version is switched as mentioned above. These changes are outlined in Listing 4.1.

```
1 --- a/docker/Blender.Dockerfile
2 +++ b/docker/Blender.Dockerfile
3 @@ -13,7 +13,7 @@
4 - FROM ubuntu:20.04 as build
```

[2] https://wiki.blender.org/wiki/Reference/Release_Notes/2.92/Geometry_Nodes
[3] https://wiki.blender.org/wiki/Reference/Release_Notes/3.0/Nodes_Physics
[4] https://wiki.blender.org/wiki/Reference/Release_Notes/3.1/Nodes_Physics
[5] https://wiki.blender.org/wiki/Reference/Release_Notes/3.2/Nodes_Physics

```
5  + FROM ubuntu:21.10 as build
6
7  @@ -41,7 +41,7 @@  # RUN git clone https://git.blender.org/blender.git
8  - RUN git clone https://github.com/blender/blender.git --branch blender-v2
       .93-release --depth 1
9  + RUN git clone https://github.com/blender/blender.git --branch blender-v3
       .2-release --depth 1 blender
```

Listing 4.1: Start of the Blender Version Upgrade

Following the changes in Listing 4.1, more adjustments are required for the build process, namely to installed operating system packages, code directories and Python package version numbers. After the build succeeds, more issues follow: Blender function names have changed, and some default behavior has also changed. In total, about 400 lines of code have been added to the Kubric Library to support the new version.

The Docker Build process can also be used for other purposes, such as installing assets or Blender Plugins. An example of Blender Plugin installation is found in Section 4.3.

4.2 Saving and Restoring Blender Geometry Nodes

In the previous section of this chapter, we discussed how the "Geometry Nodes" feature in Blender is new and actively developed. This means some operations cannot be yet achieved in a straight-forward fashion. This includes saving, restoring and transferring Geometry Nodes between Blender scenes.

In Blender 3.2 there is no direct way to import or export "Geometry Node Groups" from another Blender project. However, imported Blender objects will keep all their linked Geometry Nodes objects, and all the Materials and Node Groups that are recursively linked by the first-order objects. This feature can be exploited to save and restore Geometry Node Groups, the building blocks of this system, by using a dummy object to store the geometry.

The solution for overcoming this limitation when exporting consists of two consecutive steps, as shown in Listing A.2. First, a dummy object is created in the scene, and all Geometry Node Groups present in the system are chained together on it. All the objects in the scene, apart from the dummy object, are removed. The resulting scene is saved in a temporary file, see Listing A.3.

We find that the resulted scene file has additional objects in it. This is because Blender uses reference counting for all objects, which means simply deleting an object referenced elsewhere does not release it from the scene. The additional objects in question have been referenced by the Geometry Nodes themselves, as input arguments to "Object" type parameters. In a second step, we initiate an empty Blender scene in a separate Python process, and import only the dummy object from the temporary file. This removes all "Object" type parameters from the Geometry Nodes, and removes the additional Blender objects from the scene. The result is saved as the Geometry Node library on disk. This second step is shown in Listing A.4.

Importing a Geometry Node library is trivial: the dummy object is imported, and hidden from rendering or editing interface (Listing A.1).

This logic is used in the project for incremental "Geometry Node Group" development: the user can open the Blender Geometry Node editor, make changes to Geometry Nodes in-place, save the file, and re-run the save/restore script to use the new version in the scene generation in the full scene.

4.3 Downloading Geographic Data

As stated in the previous chapter, the first stage of the pipeline is concerned with collecting all geographic data required for a 3D scene at a chosen geographic location. To aid in accessing public data from GIS services, we use the freely available BlenderGIS Plugin [1].

After the plugin is installed into Blender, it can be used to manually download all required data using the provided User Interface. An example of the manual download flow is available in the Appendix, Figure A.1.

The plugin is also installed in the Docker image at the standard location `scripts/addons` inside the Blender installation directory. Then, some Python code is executed at module load time to load it and configure its settings. In the example in Listing 4.2, we show loading and configuring the plugin with the required keys to access the data.

```
1 def load_gis_addon():
2     bpy.ops.preferences.addon_enable(module='BlenderGIS')
3     api_key = os.getenv('SRTM_API_KEY')
4     if 'API_Key' not in bpy.context.preferences.addons['BlenderGIS'].
          preferences.demServer:
5         bpy.context.preferences.addons['BlenderGIS'].preferences.demServer
              += f"&API_Key={api_key}"
6     bpy.ops.wm.save_userpref()
```

Listing 4.2: Activating and Configuring a Blender Plugin

The plugin code can then be called from a Blender Operator to ensure its correct execution context, as described by the project maintainer [2]. However, the steps for automating this feature were not implemented in this project, and all data used for this work have been obtained manually through the process in Figure A.1.

4.4 Combining Levels of Detail

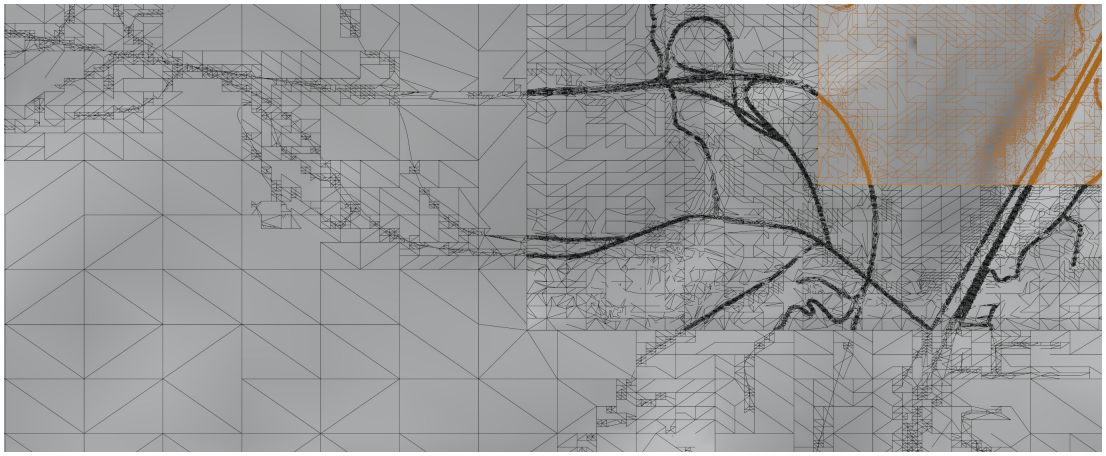


Figure 4.1: Different Levels of Detail

The geographic data aggregated by the BlenderGIS Plugin is stored in a number of Blender files, one for every zoom level. These assets must be combined into a single terrain object, with the scene action happening at the center of the smallest tile.

[1] https://github.com/domlysz/BlenderGIS
[2] https://github.com/domlysz/BlenderGIS/issues/430

In addition to the reconciliation of the different level of details for the same terrain, this step must also reconcile the different types of data obtained from different sources. For example, the SRTM height map information does not perfectly match the road position at every point. To resolve this, a height adjustment step is made on both objects: the road track is wrapped to the terrain on the Z axis, and the terrain is flattened in the area immediately around the road track. A similar computation is made in parallel for all the other assets: rail tracks, urban zones, and building perimeters.

Finally, this step ensures adequate tessellation of the ground object. More detailed polygons are generated in the areas immediately around the roads, rails and buildings, and less accurate geometry is used everywhere else. This helps keep the memory footprint low.

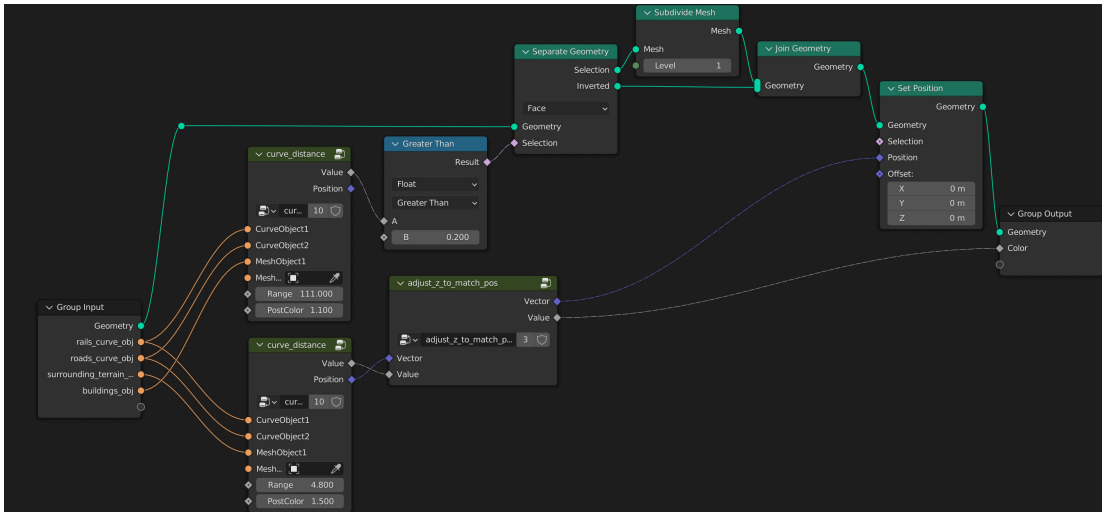


Figure 4.2: Geometry Node Implementation for Terrain Height Adjustment and Tessellation

The terrain adjustment and tessellation is implemented as a "Geometry Node Group", shown in Figure 4.2. It makes use of several subgroups: one for computing distance to a set of curves and meshes (Figure 4.3) and another for adjusting the terrain height based on that distance (Figure 4.4).

The procedural graphics designer can reuse any "Node Groups" as a part of a larger system, which justifies building a public procedural graphics primitive library to use across different projects. The technical method through such a library is managed is described in Section 4.2.

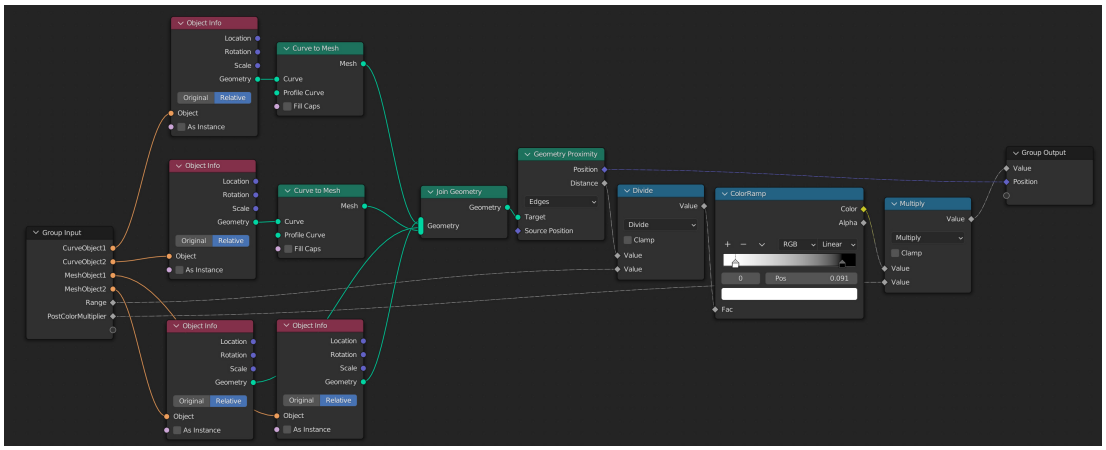


Figure 4.3: Geometry Node Implementation for Computing Distance to Curves

During this stage, additional "Geometry Node Groups" are used to create volumes and texture them to represent buildings. Detailed shapes are not implemented, since the scene is primarily concerned with the segmentation of the railway track.

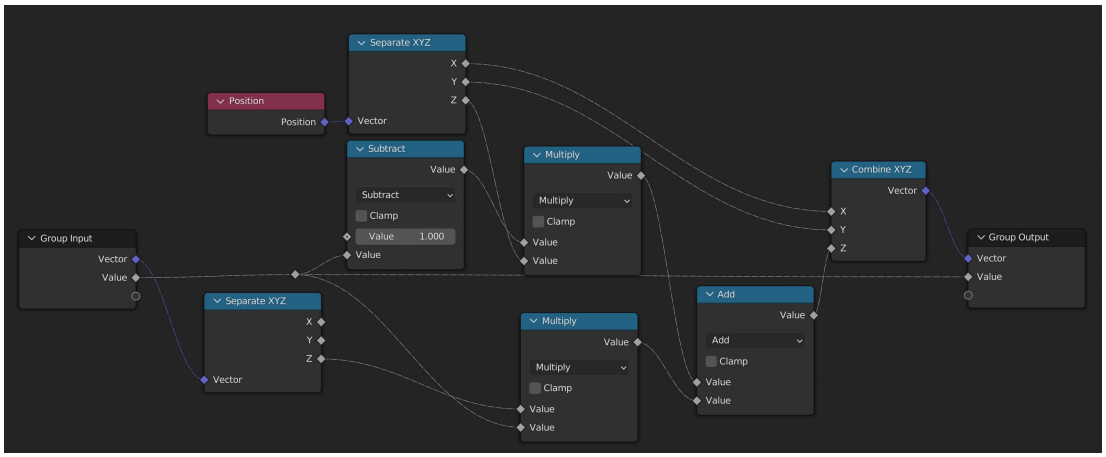


Figure 4.4: Geometry Node Implementation for Adjusting Geometry Height

After the ground objects are flattened, adjusted and the different levels of detail are combined, the scene building continues in Section 4.5 with the addition of the foreground object: rail tracks, railroad ties, the fasteners (bolts) and ballast.

4.5 Procedural Railway Generation

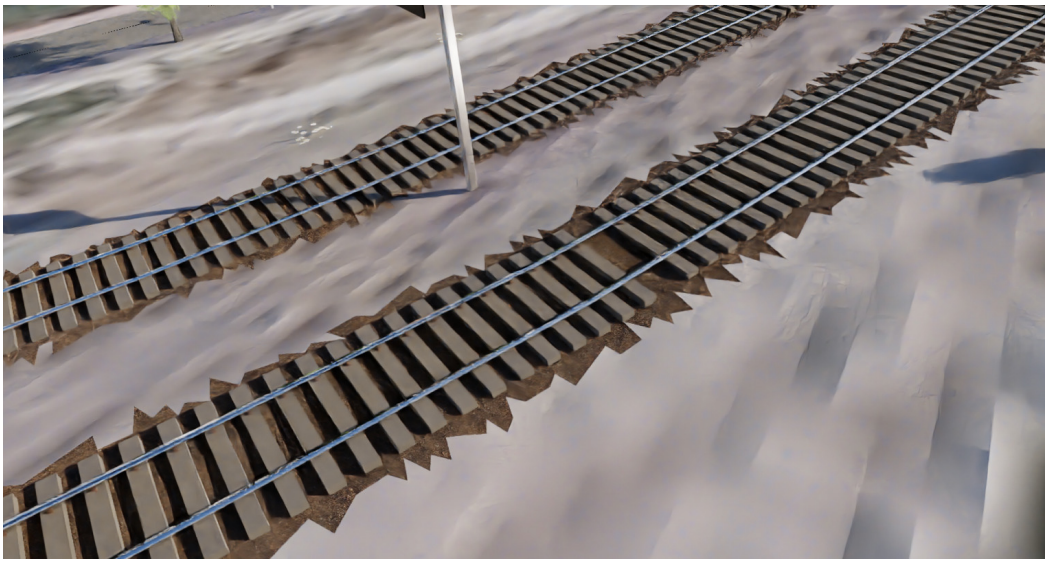


Figure 4.5: Rail Tracks - Final Result

This section explains in detail the creation of railway objects from the rail track curve. The rail track curve is obtained from OpenStreetMap as described in Section 4.3, and adjusted to fit the terrain as explained in Section 4.4.

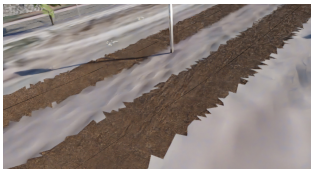


Figure 4.6: Track Ballast

First, the ground material immediately around the track is changed into a rock-like texture that represents the track ballast made of crushed stone (Figure 4.6). Then, objects are created using "Geometry Nodes" to procedurally model the metallic rail tracks, the wooden railroad ties, and the rusted metallic bolts that keep them together. The final result is shown in Figure 4.5.

The metallic rails are generated by using the "Curve to Mesh" primitive (Figure 4.7) to enwrap a profile curve over the track path. The profile curve is created by duplicating a small square to the left and right of the origin, to obtain the standard Romanian rail gauge of 1435 mm. A metallic material is set on the rail tracks geometry, with high Specular and Metallic factors.

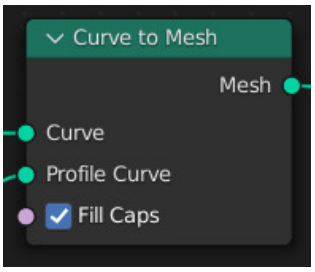


Figure 4.7: "Curve to Mesh" Geometry Primitive

To aid in creating the "rail-to-rail" segmentation mask used in the machine learning model from Chapter 5, we also create a transparent placeholder volume between the tracks. This volume is part of the track object, but does not appear in the rendered image. The placeholder volume is shown in Figure 4.9.

The wooden railroad ties (traverse planks) are similarly generated using only the railway curve. First, the wooden plank geometry is generated by scaling a cube to the standard wooden railroad tie size of 2600 mm×160 mm×220 mm. The planks are then instanced along the rail track curve, using the "Curve to Points" geometry primitive (Figure 4.8. Additionally, on each plank, we add two rust-colored hexagonal bolts. The bolts are created by using a Cylinder primitive with a setting of 6 sides, then instanced at the same time as the railroad ties.

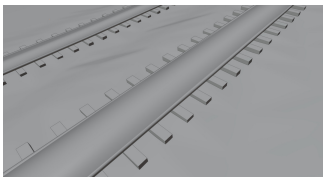


Figure 4.9: Transparent Segmentation Placeholder Volume

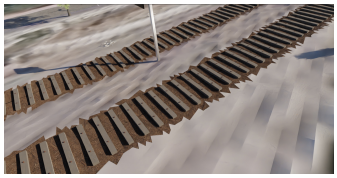


Figure 4.10: Railroad Ties in Far Configuration

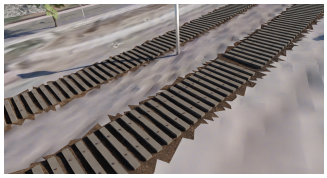


Figure 4.11: Railroad Ties in Close Configuration

Finally, railroad signage is generated by instancing a 3D model from the model library. The signage is placed alongside the tracks, at a set distance from the center.

To reduce the memory usage of the scene generation, only the visible objects are instanced during rendering. This includes rail tracks, ties, bolts, and signs. This is implemented by using ray tracing primitives to create a View Frustum Culling Geometry Node. This node is configured with a slightly larger view cone than the virtual camera, to ensure all objects from the camera are instantiated at the right time, and that objects from slightly outside the camera's view still cast shadows in the scene.

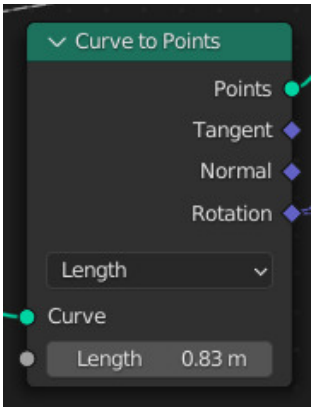


Figure 4.8: "Curve to Points" Geometry Primitive

A similar process happens with the generation of Roads: the ground area around the road curves is painted in asphalt material, and power lines are placed along the side of the road. Since the roads in this simulation are designed only as part of the background of the generated railroad footage, we did not implement finer details involving roads, such as parked or moving cars, road signage, semaphores or road barriers. The principles behind implementing these are identical to the railroad generation, and such we leave them as an exercise to the reader.

With the foreground objects finished, the scene generation continues with background details: vegetation and buildings.

4.6 Vegetation Scatter and Geometry Caching

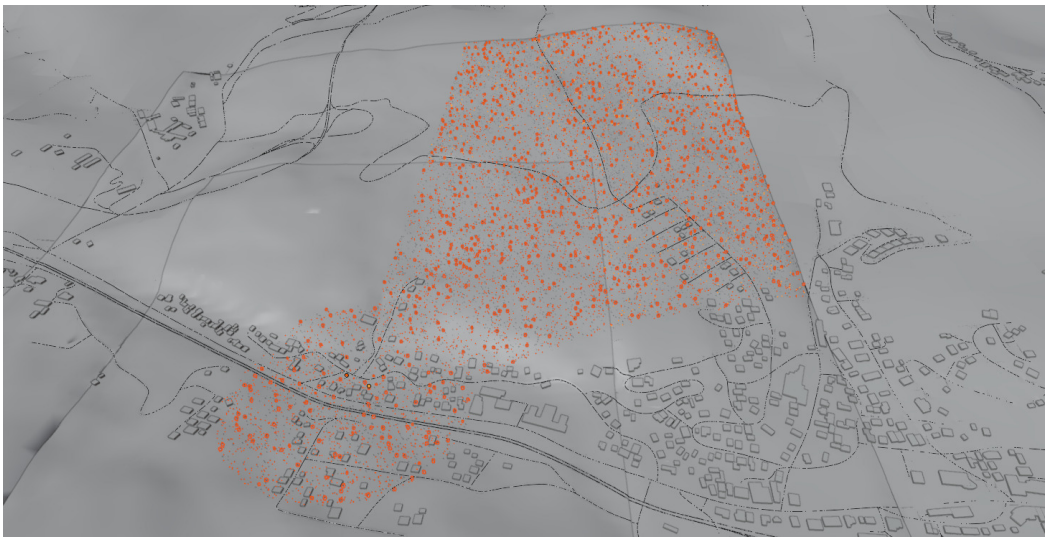


Figure 4.12: Vegetation Scattering and View Culling

After placing railroads, roads and buildings, we complete the background with instanced 3D objects representing vegetation. The process involves randomly distributing points on the terrain, while ensuring points are placed at adequate distance to roads, rail tracks, and buildings.

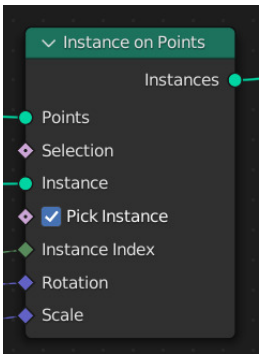


Figure 4.13: Object Instantiation

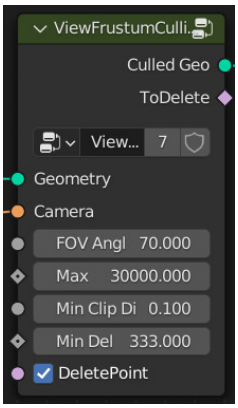


Figure 4.14: View Culling Node

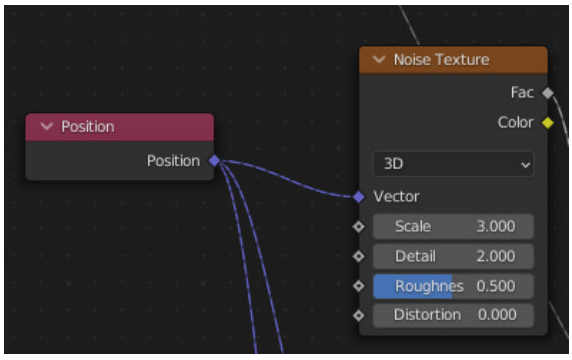


Figure 4.15: Noise Texture used with Location

To lower render times, only the visible objects are instanced. This is implemented by using a View Frustum Culling Geometry Node (Figure 4.14) that makes use of ray tracing to check if a given point is inside the camera's view. Points very close to the camera will also be added, to make sure shadows from objects behind the camera still appear in the scene. The result is visible in Figure 4.12.

The distances to various terrain features are calculated with a first Geometry Nodes Group and stored in the terrain as Vertex Group metadata called "rails proximity", "roads proximity" and "buildings proximity" (Figure 4.16). These attributes are sampled by the point scatter logic to make sure any points too close to the areas of interest are not created. This avoids situations with trees growing in the middle of the street, or through a building's roof.

Even through the Blender Geometry Nodes can be used as a functional, unidirectional data flow, it has no support for result caching. This means all Geometry Node logic must execute at every frame of the simulation, which greatly impacts the simulation frame rate.

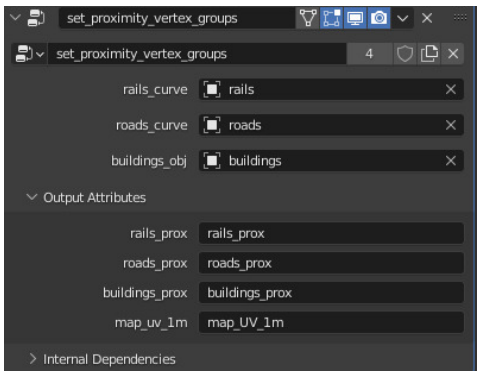


Figure 4.16: Output Vertex Groups in Ground Object

To work around this limitation, we store results of expensive computations in temporary objects, by "applying" the Geometry Nodes Modifier in Blender. This technique is used for vegetation scatter, as distributing tens of thousands of points to anchor vegetation elements is an expensive procedure.

A second Geometry Nodes Group is used to randomly generate points that respect the minimum distance to features. These points are stored in temporary objects, to avoid expensive re-computation at every frame of the render.

At instantiation time, additional random parameters are required, such as rotation, scale and instance to be picked. These extra parameters are obtained by sampling a deterministic random noise texture with the point location. Different settings for noise textures are used for different parameters. The View Frustum Culling node is shown in Figure 4.14.

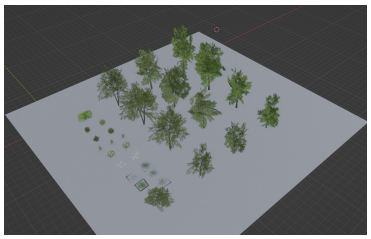


Figure 4.17: 3D Model Library for Vegetation

Finally, the points are instantiated into randomly selected objects of each vegetation type. The models for the vegetation objects are taken from the 3D model library. The "Instance on Points" primitive is shown in Figure 4.13, and the 3D models used in the simulation are shown in Figure 4.17.

With the vegetation instantiated, we can start rendering the scene. The rendering loop is described in the next section.

4.7 Scene Setup and Rendering Loop

First, scene parameter randomization happens, where random default values are set for all parameters. Parameters include both simulation settings (such as render resolution, and output video frames per second), and scene parameters (like cloud density and level, or sky texture sun orientation and strength).

Then, some custom frame setup logic is applied: user logic affects camera position and movement, changes simulation parameters, or adds and removes additional objects from the scene. This is controlled frame by frame, and can optionally sample output images from the render process of the next frame, in order to decide on how to affect the next frame.

The user-facing API (application programming interface) only contains three methods: a scene generation entry point, where the user generates the static background; a scene initialization method, to configure the initial parameters before rendering starts; and finally a frame callback that gets run on every frame. All these functions can access any Kubric or Blender internals, to reduce programmer friction. Listing 4.3 shows the interface, which in Python is called Abstract Base Class (ABC).

```
1  from abc import ABC, abstractmethod
2  class SceneGeneratorInterface(ABC):
3      @abstractmethod
4      def generate_background_scene(self, scene):
5          pass
6
7      @abstractmethod
8      def init_scene_parameters(self, scene):
9          pass
10
11     @abstractmethod
12     def frame_callback(self, scene, render_data=None):
13         pass
```

Listing 4.3: Scene Generation API

The demonstration rail track scene makes use of a relatively simple camera setup logic: the camera follows the track at a constant speed. From the multiple available rail track paths, we filter out the points too far to be used in the scene. The remaining points are checked for the longest continuous section with consecutive indexes. The camera then follows that longest path. Since the track points are not at a constant distance to each other, the path needs to be resampled, so the camera moves along the interpolated points at a constant rate. An excerpt of the track picking and path interpolation code is shown in Appendix Listing A.5.

Two implementations for the interface in Listing 4.3 exist: one generates a smooth video with a constantly moving camera, and slowly varying one parameter at a time, meant for presentation video generation; and the other one randomizes all parameters on every frame, and is meant for machine learning dataset generation.

As a predefined number of frames are rendered, the images and related metadata are stored in the output storage location.

Chapter 5

Testing and Evaluation

Synthetic video data generated from the project is created from multiple parameter configurations. This generated dataset was shared with [19] to evaluate its benefits for training and fine-tuning machine learning models for the task of detecting and segmenting rail tracks. The scene generator is also benchmarked in multiple configurations for run time and data output size, on consumer-grade CPU and GPU hardware.

5.1 Validating Generated Video using a Machine Learning Model

We have cooperated with [19] to design and optimize a scene generator for their rail track segmentation task. This resulted in the creation of two different datasets[2] of about 170 images each, containing image frame data, as well as auto-generated segmentation masks.

Example frames and model activations are shown in Table 5.1.

This synthetic data was used alongside real-world data from train dash cams[28], and drone footage [1] of rail tracks. It is shown in [19] that the use of synthetic data alongside real data during training is beneficial to the performance of the model. These results are reproduced in Table 5.2.

Two datasets have been created for this purpose. The first one, Dataset v1, makes use of all the features implemented for the scene generator, including volumetric clouds and vegetation scattering. At the request of the machine learning developers, we have removed volumetric clouds and all vegetation in a second version, Dataset v2. The second dataset also make use of Gamma normalization, to ensure the image exposure level is uniform across the entire dataset.

5.2 Evaluating Run Time and Data Size

This section evaluates the runtime impact of each setting with respect to render time. A discussion of the Blender scene sizes and intermediate data sizes is also made.

Since the rendering is done by a ray tracing algorithm, it is alike to a Monte Carlo Simulation: randomized rays of light are evaluated to obtain their impact on a specific pixel of the rendered image. In Blender, means the algorithm takes a variable time to finish, as early stopping is made when further iterations of the algorithm no longer affect the image in a meaningful way.

[2] https://github.com/gabriel-v/all-tracks-no-trains

[1] Obtained from an initiative of NETIO and Thales

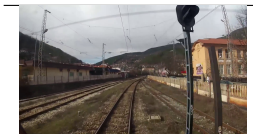
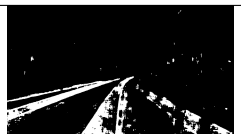
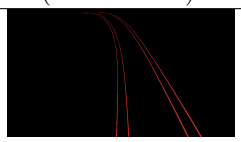
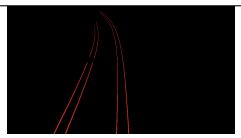
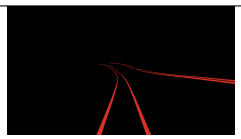
Frame	Ground Truth	Evaluation	Class
			rail-to-rail
(a) Training Dataset - RailSem19[28] (Train Dash Cam)			
			rail-to-rail
(b) Validation Dataset [1] (Drone Cam)			
			rails only
(c) Initial Attempt			
			rails only
(d) Improved Rail Track Materials			
			rails only
(e) Adjusted Camera Angle, Added Clouds			
			rail-to-rail
(f) Final Result			

Table 5.1: Evaluating the "unet" Machine Learning Model[19] on Real and Synthetic Frames

Data Used for Training	Model IoU Score
Real data only	87.39%
Real data and Synthetic Dataset v1 (with trees and clouds)	87.47%
Real data and Synthetic Dataset v2 (no trees, normalized Gamma)	87.49%

Table 5.2: "unet" Model Accuracy Improvement from use with Synthetic Data[19]

This means the render times are highly dependent on the scene: the number of objects visible, their complexity, but also local volumetric effects like clouds all affect the render time.

To evaluate scene build and render times in Table 5.3 we used 3D tiles of zoom levels 15, 17 and 18 to render train tracks in a resolution of 960x540. The rendering was done on a 6-core CPU model "Intel(R) Core(TM) i5-9400F @ 2.90GHz".

Render Trees?	Render Clouds?	Render Buildings?	Environment Generation Time (s)	Frame Render Time (s)
False	False	False	88.93	59.02
True	False	False	180.51	61.44
False	**True**	False	85.53	132.5
False	False	**True**	110.28	55.34
True	False	**True**	210.36	65.14
True	**True**	**True**	193.82	123.13

Table 5.3: Scene Generation and Render Run Times

As shown in Table 5.3, the cloud generation has the most impact on render time, and the vegetation generation has the most impact on scene generation time.

We have also used Blender's Nvidia CUDA renderer implementation with a GPU model "NVIDIA GeForce GTX 1660 Ti", and found a consistent 5-10% improvement in render times, at the cost of a much slower environment generation time. This was caused by the shader compilation step, which for our use-case takes approximately 320 seconds.

Memory use, on the other hand, is more predictable. Our implementation of the rail track segmentation task uses 8-12 GB of RAM for rendering the scene, depending on the render parameters and scene settings.

Asset Type	Asset Size (MB)	Observations
Downloaded 3D Tile	30 - 100	Larger or more densely urban tiles require more storage.
Generated Environment Scene	270 - 330	Depends on tile size, count, and the 3D model library size.
Rendered Frame Set	1 - 4	Depends on resolution. All outputs can be converted to JPEG to reduce storage requirements.

Table 5.4: Scene Generation and Render Run Times

Finally, we discuss the output data sizes for each stage of the project in table 5.4. As mentioned in Section 4.3, the raw GIS data is stored as 3D tiles. Then, a number of tiles centered on the same point are combined into an environment background scene. Finally, the environment scene is combined with more user assets and logic to render frames.

Chapter 6

Conclusions

6.1 Summary

We introduce The Procedural Outdoors Scene Generator, an open-source Python framework aimed at generating realistic synthetic video data of outdoor environments. The framework integrates data from various GIS services (satellite imagery, altitude maps, and OpenStreetMap) with a custom library of procedural graphics primitives and freely available 3D models.

The project explores novel procedural modeling techniques and addresses a number of problems present with the "Geometry Nodes" programming environment in Blender, including the import and export of "Geometry Node Groups" from Blender scenes, and the caching of intermediate data between different applications of procedural modeling primitives.

6.2 Results

We demonstrate the suitability of our framework in generating synthetic datasets for machine learning tasks by evaluating the benefits of using generated data in a machine learning segmentation task. This results in two publicly shared image datasets of rail tracks, annotated with semantic segmentation masks for the rail track.

We show that the addition of a relatively small number of synthetic images to a machine learning model's training dataset improves its IoU score for the rail track segmentation task by **0.1%**. Finally, we evaluate the scene generator's performance in terms of run time, memory use and output data quantity.

We include a discussion of the future of the project in Chapter 7, where we envision the creation of a free library of procedural 3D assets, alongside a web platform meant to facilitate open collaboration.

Chapter 7

Further Work

This section outlines recommendations for continuing the project. We describe plans for implementing more procedurally generated environment models, as well as a blueprint for an open web collaboration platform, where procedurally generated graphics - both geometry primitives and procedural textures - can be shared under a permissive license. Finally, we outline a one-year action plan with estimated budget and human resource requirements.

7.1 Free Library of Procedurally Generated 3D Assets

Procedural generation can be used to create virtually endless variations of the same base objects, which helps machine learning models be more robust in real-world situations. This project only designed the train tracks in this way, and such was only useful for detecting that specific object class.

We propose combining various techniques listed in this section to make most relevant classes for machine learning available in procedurally generated designs in a public, free library. This would enable machine learning researchers to easily put together realistic simulations for their synthetic data needs.

The related work in Chapter 2, Section 2.3 can be translated to the Blender Geometry Node environment. Where this is not directly possible, we propose the creation of Blender Plugins to implement novel functionality in Python. These plugins would have the ability to execute arbitrary code.

We propose building a comprehensive procedural geometry library with the following elements:

Vegetation. Trees[10] and plants[3] can be procedurally generated. The "Sapling Add-on"[2], included in Blender, can also be used. Geographical biome is to be queried from GIS, to select a specific ecosystem from which procedural models are chosen. Seasonal effects on vegetation can also be modeled.

Ground Features. procedural textures for asphalt, gravel. Procedurally generated stones.

Weather. Base weather on real-world data, similar to "Microsoft Flight Simulator 2020"[3].

Buildings. Different building volumes and faces for urban, rural and industrial zones.

Humans. Physical appearance, clothes and actions can all be synthetically generated using different methods.

[2] https://docs.blender.org/manual/en/latest/addons/add_curve/sapling.html
[3] https://www.pcgamer.com/heres-how-microsoft-flight-simulator-creates-its-realistic-weather/

Traffic. Road, pedestrian, and rail traffic should all be modeled in the simulation.

A collective effort to provide openly licensed assets for the above purposes will require specialized software where contributors can access, modify and share assets under permissive licenses. This web collaboration platform is discussed in the next section.

7.2 Open Web Collaboration Platform

To aid the building of a free, public and comprehensive graphics library, we propose implementing a specialized web collaboration platform where contributors can preview, access, modify and share specific assets.

User friction is to be eliminated through the use of a Blender Plugin to communicate with the web service, to ensure individual assets found on the platform are transferred to the local Blender client through a single click. The designer can then modify the asset, and re-upload it as a new asset.

The web platform would have the following features:

3D Tile Cache. Downloading OSM, SRTM and Satellite data is very time-consuming, with larger tiles taking upwards of 15 minutes to obtain. The maintainer of the web platform should handle the delivery of pre-processed 3D tiles.

3D Model Hosting. Models uploaded on the Internet with permissive licenses are to be re-uploaded on the platform, to be a part of the free library. They can also be modified to use procedurally-generated textures, to increase their variance. For example, car models could have their paint material changed, or dirtied by mud.

Geometry Primitive Hosting. We propose using the technique from Section 4.2 to create a public library of Geometry Node Groups. These would range from small building blocks, such as the "View Culling" Node Group from Figure 4.14, up to entire objects, such as the "rail tracks" Node Group which creates entire railway lines.

Scene Setup and Logic Hosting. Entire code bases that use the 3D models and geometry primitives mentioned above would be shared on the platform. Machine learning researchers can then browse through the public catalog and select a starting point for their synthetic data generation project.

Since this web platform must host and execute arbitrary code as either **Scene Setup and Logic** or **Geometry Primitives**, a comprehensive security team must monitor assets and implement safe execution environments where user code can run. Additionally, since its public nature, this library would need a moderation team to keep the community safe. The costs of these teams, and all the other resources required to implement this collaboration platform, are outlined in the next section.

7.3 One Year Action Plan

In the previous sections, we have presented a plan to create a public library for procedurally generated 3D assets and scenes, that would be used alongside the framework implemented in Chapter 4. In this section, we estimate the human resources and financial costs needed to implement this plan.

Foremost, a core team of engineers will develop the web platform. This team would be comprised of backend and front-end engineers, a security specialist and system administrator.

Secondly, procedural 3D artists are needed to build more of the world for demonstration purposes, to start producing video scenes of a more broad scope. This base content will be needed

to start a community around the free asset library, and to initiate the network effect of using the platform.

Additional experts are given medium-term contracts to implement interoperability between Blender, the web platform, and existing machine learning tooling. This includes Reinforcement Learning (RL) libraries, rendering infrastructure, and other miscellaneous software components.

Lastly, the task of community building requires two human aspects: outreach, and moderation. The outreach aspect is necessary to make the project be known and trusted, so the asset library eventually starts being contributed to by members of the public, free of charge. A well-placed social media campaign or high school outreach program could help the younger generation transition from developing "Roblox games"[1] to professional 3D artist careers. The moderation element is required to ensure trust in the platform is kept, by removing copyrighted or otherwise illegal content.

Issue	Person	Resource	Time Frame	Cost
Web Platform Implementation	Python engineer	Backend Implementation	long	RON 130,000
	JavaScript engineer	UI Design and Implementation	long	RON 110,000
	Linux security specialist	Isolated Execution Environment, Security Audit	short	RON 30,000
	Sysadmin	Hardware acquisition, deployment and maintenance	long	RON 100,000
Base Asset Library	Procedural 3D Artist	Freely-licensed procedural models	long	RON 75,000
Interactivity between Blender and Web Platform	Python Blender engineer	Blender Plugins	medium	RON 75,000
Interactivity between Scene Generator and ML tools	ML specialist	ML tooling and integrations	medium	RON 70,000
Community building	Marketing specialist	Articles, ads and social media posts	short	RON 17,000
Copyrighted or illegal content	Moderator	Community safety	long	RON 65,000

Table 7.1: One Year Action Plan

Table 7.1 breaks down the costs mentioned in this section. The average yearly net salary values for Romania have been obtained from the Economic Research Institute [2] web page in September 2022. The "Time Frame" column represents the total work period for the full-time member, more exactly 3 months for a "short" time frame, 6 months for "medium" and the entire year for "long".

By taking the current year-on-year inflation in Romania of about 15% [3] and projecting it into the future, then adding yearly hardware rental costs of RON 30,000 and finally doubling the resulting budget for safe measure, we arrive at a total estimate cost of RON 15,000,000, or about

[1] https://www.businessinsider.com/roblox-direct-listing-young-game-developers-2021-3
[2] https://www.erieri.com/
[3] https://tradingeconomics.com/romania/inflation-cpi

300,000 Euro, for one year of development. We note that this budget is small when compared to the average multi-million-Euro budget that realistic video games mentioned in Section 2.1 are known to have[1].

[1] https://www.ibtimes.com/gta-5-costs-265-million-develop-market-making-it-most-expensive-video-game-ever-produced-report

Appendix A

Code Excerpts and Manual Processes

The function in Listing A.1 imports a dummy object containing the Geometry Nodes in the procedural graphics library. See Section 4.2.

```
def import_geometry_cube(blend_path):
    cube = blend_append_object(blend_path, GEOMETRY_DUMMY_CUBE_NAME)
    cube.select_set(True)
    cube.hide_render = True
    cube.hide_viewport = True
```

Listing A.1: Geometry Nodes - Import Function

This functions in Listings A.2, A.3 and A.4 compile and export a Geometry Nodes Library. See Section 4.2.

```
def save_geometry(original_path, destination_path, skip_if_missing=True):
    if skip_if_missing and not pathlib.Path(original_path).is_file():
        return
    temp_path = GEOMETRY_TMP_FILE
    p = Process(target=_save_geometry_1, args=(original_path, temp_path,
        skip_if_missing))
    p.start()
    p.join()
    assert p.exitcode == 0

    p = Process(target=_save_geometry_2, args=(temp_path, destination_path,
        skip_if_missing))
    p.start()
    p.join()
    assert p.exitcode == 0
    try:
        os.unlink(temp_path)
    except Exception:
        pass
```

Listing A.2: Geometry Nodes - Export - Main Function

```
def _save_geometry_1(original_path, destination_path, skip_if_missing=True):
    if skip_if_missing and not pathlib.Path(original_path).is_file():
        return
```

```
4
5       scene = kb.Scene(resolution=(settings.RESOLUTION_X, settings.
            RESOLUTION_Y),
6                        frame_start=1, frame_end=settings.MAX_FRAMES)
7       renderer = Blender(
8           scene, custom_scene=original_path, custom_scene_shading=True,
9           adaptive_sampling=True, samples_per_pixel=settings.SAMPLES_PER_PIXEL
                ,
10      )
11      pre_init_blender(renderer)
12      cube_name = GEOMETRY_DUMMY_CUBE_NAME
13      for obj in bpy.data.objects:
14          if obj.name == GEOMETRY_DUMMY_CUBE_NAME:
15              bpy.data.objects.remove(obj, do_unlink=True)
16
17      scene += kb.Cube(name=cube_name, scale=(1, 1, 1), position=(0, 0, 0))
18
19      with make_active_object(cube_name) as cube:
20          cube.hide_render = True
21          cube.hide_viewport = True
22
23          already_added = set()
24          for node_group in bpy.data.node_groups:
25              # for each landmap item, set Geometry modifier
26              if node_group.name in already_added:
27                  continue
28              already_added.add(node_group.name)
29              mod = cube.modifiers.new('geometry_wrapper_' + node_group.name,
                    'NODES')
30              mod.node_group = bpy.data.node_groups[node_group.name]
31
32          # immediately delete the created objects, since we *really* only
                want the cube
33          for obj in bpy.data.objects:
34              if obj.name != GEOMETRY_DUMMY_CUBE_NAME:
35                  bpy.data.objects.remove(obj, do_unlink=True)
36              else:
37                  log.info('KEEP␣%s', obj.name)
38
39      save_blend(renderer, destination_path)
```

Listing A.3: Geometry Nodes - Export - First Step

```
1   def _save_geometry_2(original_path, destination_path, skip_if_missing=True):
2       if skip_if_missing and not pathlib.Path(original_path).is_file():
3           return
4
5       scene = kb.Scene(resolution=(settings.RESOLUTION_X, settings.
            RESOLUTION_Y),
6                        frame_start=1, frame_end=settings.MAX_FRAMES)
7       renderer = Blender(scene, adaptive_sampling=True, samples_per_pixel=
            settings.SAMPLES_PER_PIXEL)
8       pre_init_blender(renderer)
9       import_geometry_cube(original_path)
10      save_blend(renderer, destination_path)
```

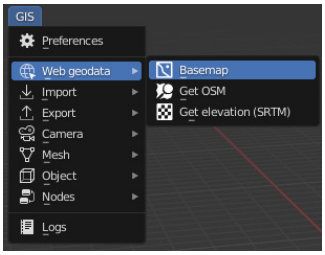
Listing A.4: Geometry Nodes - Export - Second Step

The code in Listing A.5 implements camera path interpolation as part of the scene setup process. See Section 4.7 for more details.

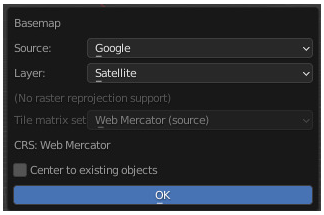
```
    anim_distance = (settings.MAX_FRAMES / settings.SIMULATION_FPS) *
        settings.CAMERA_ANIMATION_SPEED_M_S * 2 + 500
    animation_path = 'rails_center'
    with make_active_object(animation_path):
        path_vertices = [
            (d.index, (d.co.xyz.to_tuple()))
            for d in bpy.context.active_object.data.vertices
            if math.sqrt(
                d.co.xyz.to_tuple()[0]**2 + d.co.xyz.to_tuple()[1]**2
            ) < anim_distance
        ]
    index_count = [
        i for i, (x, y) in enumerate(
            zip([a[0] for a in path_vertices[:-1]],
                [a[0] for a in path_vertices[1:]])
        ) if x + 1 != y
    ]
    index_count = [0] + index_count + [len(path_vertices)]
    index_count = {index_count[k]: index_count[k + 1] - index_count[k]
                   for k in range(0, len(index_count) - 1)}
    start_idx, run_len = max(index_count.items(), key=lambda t: (t[1],
        random.random()))
    path_point = [path_vertices[k][1]
                  for k in range(1 + start_idx, start_idx + run_len + 1)
                  if k < len(path_vertices)]

    def _3d_dist(point, next_point):
        return math.sqrt((point[0] - next_point[0])**2 + (point[1] -
            next_point[1])**2 + (point[2] - next_point[2])**2)
    def _add_nan(points, avg_dist=1):
        log.info('adding␣NaN␣to␣%s␣points,␣interp␣dist␣%s', len(points),
            avg_dist)
        for point_id in range(0, len(points) - 1):
            point = points[point_id]
            next_point = points[point_id + 1]
            next_point_dist = _3d_dist(point, next_point)
            yield point
            yield from [(numpy.nan, numpy.nan, numpy.nan)] * int(
                next_point_dist / avg_dist)
        yield points[-1]

    start_to_end_dist = _3d_dist(path_point[0], path_point[-1])
    path_point = pandas.DataFrame(_add_nan(path_point, settings.
        CAMERA_ANIMATION_SPEED_M_S / settings.SIMULATION_FPS))
    path_point = path_point.interpolate(method='linear', limit_area='inside'
        )
    path_point = [tuple(t[1]) for t in path_point.iterrows()]
    assert len(path_point) > settings.MAX_FRAMES
```

Listing A.5: Camera Path Interpolation

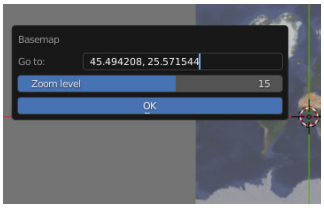
Figure A.1 describes the steps required to manually obtain geographic data using the GIS plugin. See more details in Section 4.3.



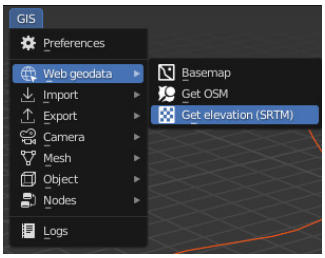
(a) Open Base Map Form



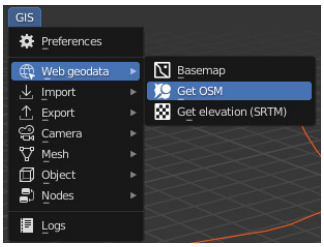
(b) Choose Satellite Provider



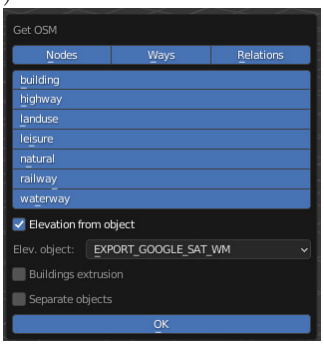
(c) Choose Location and Zoom



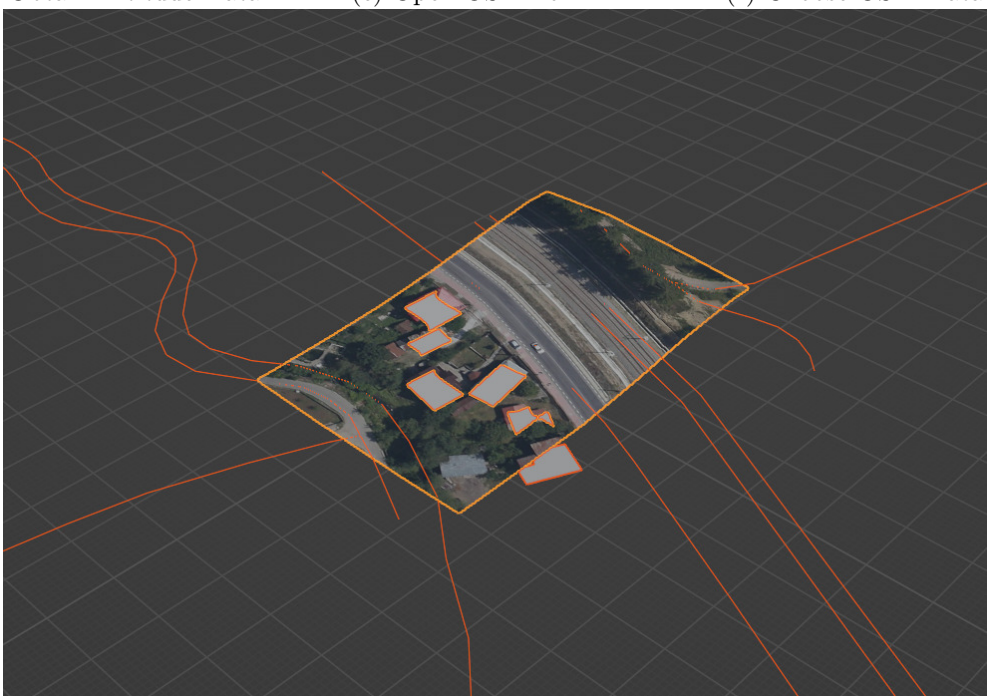
(d) Obtain Altitude Data



(e) Open OSM Form



(f) Choose OSM Data



(g) Save End Result

Figure A.1: Manual Process for Downloading Geographical Data

Bibliography

[1] Yuki M Asano, Christian Rupprecht, Andrew Zisserman, and Andrea Vedaldi. Pass: An imagenet replacement for self-supervised pretraining without humans. *arXiv preprint arXiv:2109.13228*, 2021.

[2] Antonella Barisic, Frano Petric, and Stjepan Bogdan. Sim2air-synthetic aerial dataset for uav monitoring. *IEEE Robotics and Automation Letters*, 7(2):3757–3764, 2022.

[3] Ruud Barth, Joris IJsselmuiden, Jochen Hemming, and Eldert J Van Henten. Data synthesis methods for semantic segmentation in agriculture: A capsicum annuum dataset. *Computers and electronics in agriculture*, 144:284–296, 2018.

[4] Adam Bengtsson. Efficient realistic cloud rendering using the volumetric rendering technique. 2022.

[5] Nishchal Bhandari. *Procedural synthetic data for self-driving cars using 3D graphics*. PhD thesis, Massachusetts Institute of Technology, 2018.

[6] Maurilio Di Cicco, Ciro Potena, Giorgio Grisetti, and Alberto Pretto. Automatic model based dataset generation for fast and accurate crop and weeds detection. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5188–5195. IEEE, 2017.

[7] Anh-Dzung Doan, Abdul Mohsi Jawaid, Thanh-Toan Do, and Tat-Jun Chin. G2d: from gta to data. *arXiv preprint arXiv:1806.07381*, 2018.

[8] Matteo Fabbri, Guillem Brasó, Gianluca Maugeri, Orcun Cetintas, Riccardo Gasparini, Aljoša Ošep, Simone Calderara, Laura Leal-Taixé, and Rita Cucchiara. Motsynth: How can synthetic data help pedestrian detection and tracking? In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 10849–10859, October 2021.

[9] Klaus Greff, Francois Belletti, Lucas Beyer, Carl Doersch, Yilun Du, Daniel Duckworth, David J Fleet, Dan Gnanapragasam, Florian Golemo, Charles Herrmann, Thomas Kipf, Abhijit Kundu, Dmitry Lagun, Issam Laradji, Hsueh-Ti (Derek) Liu, Henning Meyer, Yishu Miao, Derek Nowrouzezahrai, Cengiz Oztireli, Etienne Pot, Noha Radwan, Daniel Rebain, Sara Sabour, Mehdi S. M. Sajjadi, Matan Sela, Vincent Sitzmann, Austin Stone, Deqing Sun, Suhani Vora, Ziyu Wang, Tianhao Wu, Kwang Moo Yi, Fangcheng Zhong, and Andrea Tagliasacchi. Kubric: a scalable dataset generator. 2022.

[10] Charlie Hewitt. Procedural generation of tree models for use in computer graphics [ph. d. dissertation]: Cambridge. *University of Cambridge*, 2017.

[11] Braden Hurl, Krzysztof Czarnecki, and Steven Waslander. Precise synthetic image and lidar (presil) dataset for autonomous vehicle perception. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2522–2529. IEEE, 2019.

[12] Amlan Kar, Aayush Prakash, Ming-Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. Meta-sim: Learning to generate syn-

thetic datasets. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4551–4560, 2019.

[13] Abdulrahman Kerim, Cem Aslan, Ufuk Celikcan, Erkut Erdem, and Aykut Erdem. Nova: Rendering virtual worlds with humans for computer vision tasks. In *Computer Graphics Forum*, volume 40, pages 258–272. Wiley Online Library, 2021.

[14] Samin Khan, Buu Phan, Rick Salay, and Krzysztof Czarnecki. Procsy: Procedural synthetic dataset generation towards influence factor studies of semantic segmentation networks. In *CVPR workshops*, pages 88–96, 2019.

[15] Michalis Korakakis, Phivos Mylonas, and Evaggelos Spyrou. A short survey on modern virtual environments that utilize ai and synthetic data. In *MCIS*, page 34, 2018.

[16] Alina Marcu, Dragos Costea, Vlad Licaret, Mihai Pîrvu, Emil Slusanschi, and Marius Leordeanu. Safeuav: Learning to estimate depth and safe landing areas for uavs from synthetic data. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pages 0–0, 2018.

[17] Pablo Martinez-Gonzalez, Sergiu Oprea, John Alejandro Castro-Vargas, Alberto Garcia-Garcia, Sergio Orts-Escolano, Jose Garcia-Rodriguez, and Markus Vincze. Unrealrox+: An improved tool for acquiring synthetic data from virtual 3d environments. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[18] Marcel Neuhausen, Patrick Herbers, and Markus König. Using synthetic data to improve and evaluate the tracking performance of construction workers on site. *Applied Sciences*, 10(14):4948, 2020.

[19] Alexandru Petre and Iuliu Vasilescu. Optimizing ml model for embedded devices. *[Unpublished Bachelor's Thesis]*, pages 19–22, 2022.

[20] Samira Pouyanfar, Muneeb Saleem, Nikhil George, and Shu-Ching Chen. Roads: Randomization for obstacle avoidance and driving in simulation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[21] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European conference on computer vision*, pages 102–118. Springer, 2016.

[22] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3234–3243, 2016.

[23] Gregory J Stein and Nicholas Roy. Genesis-rt: Generating synthetic images for training secondary real-world tasks. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7151–7158. IEEE, 2018.

[24] Apostolia Tsirikoglou, Joel Kronander, Magnus Wrenninge, and Jonas Unger. Procedural modeling and physically based rendering for synthetic data generation in automotive applications. *arXiv preprint arXiv:1710.06270*, 2017.

[25] Qi Wang, Junyu Gao, Wei Lin, and Yuan Yuan. Learning from synthetic data for crowd counting in the wild. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8198–8207, 2019.

[26] Magnus Wrenninge and Jonas Unger. Synscapes: A photorealistic synthetic dataset for street scene parsing. *arXiv preprint arXiv:1810.08705*, 2018.

[27] Bichen Wu, Xuanyu Zhou, Sicheng Zhao, Xiangyu Yue, and Kurt Keutzer. Squeezesegv2: Improved model structure and unsupervised domain adaptation for road-object segmenta-

tion from a lidar point cloud. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 4376–4382. IEEE, 2019.

[28] Oliver Zendel, Markus Murschitz, Marcel Zeilinger, Daniel Steininger, Sara Abbasi, and Csaba Beleznai. Railsem19: A dataset for semantic rail scene understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.