University of the Philippines Manila
College of Arts and Sciences
Department of Physical Sciences and Mathematics

Procedural Modeling for Sustainable Urban Development and Planning: A Blender Plugin for 3D Modeling of Philippine Cities

A special problem in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science

Submitted by:

Adrian Neil P. Santos

June 2023

Permission is given for the following people to have access to this SP:

Available to the general public	Yes
Available only after consultation with author/SP adviser	No
Available only to those bound by confidentiality agreement	No

ACCEPTANCE SHEET

The Special Problem entitled "Procedural Modeling for Sustainable Urban Development and Planning: A Blender Plugin for 3D Modeling of Philippine Cities" prepared and submitted by Adrian Neil P. Santos in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science has been examined and is recommended for acceptance.

Ma. Sheila A. Magboo, Ph.D. (*cand.*)
Adviser

EXAMINERS:

	Approved	**Disapproved**
1. Avegail D. Carpio, M.Sc.		
2. Richard Bryann L. Chua, Ph.D. (*cand.*)		
3. Perlita E. Gasmen, M.Sc. (*cand.*)		
4. Vincent Peter C. Magboo, M.D., M.Sc.		
5. Marbert John C. Marasigan, M.Sc. (*cand.*)		
6. Geoffrey A. Solano, Ph.D.		

Accepted and approved as partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Vio Jianu C. Mojica, M.Sc.
Unit Head
Mathematical and Computing Sciences Unit
Department of Physical Sciences and Mathematics

Marie Josephine M. De Luna, Ph.D.
Chair
Department of Physical Sciences and Mathematics

Maria Constancia O. Carrillo, Ph.D.
Dean
College of Arts and Sciences

Abstract

This study presents a procedural modeling plugin for sustainable urban development and planning, specifically focusing on the 3D modeling of Philippine cities. The plugin integrates machine learning, procedural generation techniques, and interactive 3D visualization to provide urban planners with a powerful toolset for efficient and accurate urban modeling. By incorporating a classification model for identifying architectural styles unique to Philippine urban cities, the plugin enables the creation of digital assets and procedural systems that capture the essence of urban elements while allowing for flexible manipulation. Through a comprehensive case study on Taguig City, the effectiveness of the plugin is demonstrated in generating dynamic and realistic urban cityscapes. This study contributes to the advancement of urban planning practices by offering innovative solutions to the challenges faced in digital city replication, empowering urban planners to make informed decisions and foster sustainable development in Philippine urban cities.
Keywords: procedural modeling, plugin, sustainable urban development, urban planning, 3D modeling, Philippine cities, machine learning, architectural styles, digital assets, urban elements

Contents

List of Figures

List of Tables

I. Introduction

A. Background of the Study

Cities are complex systems that are shaped by various historical, cultural, and economic factors [1]. The built environment plays a crucial role in shaping the quality of life, social development, and well-being of the population [3]. However, rapid urbanization can lead to haphazard development that prioritizes growth and profitability over sustainability, inclusivity, and cultural preservation [4]. One of the key challenges in urban planning is preserving and promoting a city's unique architectural style, which refers to the characteristic features and design principles of buildings and structures in a specific region or period [1]. This can be particularly challenging in cities like those in the Philippines, which have a rich architectural heritage influenced by a variety of styles such as Chinese, Western, and Hindu [5].

Procedural modeling is a technique that addresses these issues by allowing the generation of 3D content, such as buildings, road networks, urban decorations, vehicles, and crowds, algorithmically based on a set of rules and control parameters [6]. One of the key advantages of this technique is that it can be programmed to generate assets that adhere to specific architectural styles by incorporating architectural style rules and architectural style vocabulary, which refers to the specific terms, phrases and language used to describe and classify architectural styles. This enables the efficient creation and updating of digital replicas of cities, and can support the creation of sustainable and inclusive urban environments [7].

However, determining architectural styles of the assets created using procedural modeling remains a challenging task as it often involves subjective judgments and can be influenced by various factors such as cultural influences, historical context, and personal preferences [1]. Machine learning, a subset of Artificial Intelligence, is a set of algorithms that can learn from data and make predictions or decisions without being explicitly programmed [4]. One solution to this problem is using

machine learning algorithms that can classify architectural styles and determine the percentage of different architectural styles present in a structure.

The primary aim of this study is to design and implement a procedural generation plugin for Blender, a widely used open-source 3D creation software, that aids in creating and updating digital replicas of cities and other urban elements such as trees and streetlights in the Philippines. The plugin is being evaluated through quantitative analysis of the generated models and a machine learning model for evaluating adherence to architectural styles.

B. Statement of the Problem

The study aims to address the inefficiency and scalability issues of traditional methods for creating digital replicas of urban cities for urban planning and development. These replicas serve as a visualization tool for understanding and designing the structural compositions of the city, including road networks, buildings, and urban decorations. However, traditional methods involve manual modeling and texturing, which are costly and time-consuming, and result in static replicas that are quickly outdated.

Furthermore, the study aims to address the problem of structural dissonance in cities caused by rapid urbanization and haphazard development. This can lead to a lack of sustainability and inclusivity in the built environment, negatively impacting the quality of life and well-being of the population.

Procedural techniques offer a solution to these issues by streamlining the digital creation process and allowing dynamic modification of the procedural components. This method can generate countless variations of urban cities almost instantaneously, which reduces the cost and effort required for creating digital replicas. Furthermore, it address the problem of structural dissonance in cities by allowing for the creation of multiple iterations and variations of the city that follows a particular architectural style. Hence, enabling urban planners and developers to visualize and plan for sustainable and inclusive urban environments.

C. Objectives of the Study

C..1 General Objectives

The study aims to create a Blender plugin for representing urban elements of Philippine urban cities. This plugin allows the urban planner to select an urban element and manipulate its parameters and get a 3D model for the 3D environment. The urban planner can also evaluate how well the generated model represent a particular urban element by a machine learning model, and evaluate the performance of the model in terms of geometry and storage space.

C..2 Specific Objectives

1. Perform a case study on Taguig city, a 1st class highly urbanized city in Metro Manila, Philippines and its residential buildings.
    (a) Gather images of urban residential buildings in Taguig city.
    (b) Identify and select common architectural style rules and vocabulary specific to Philippine urban residential buildings.
2. Create a digital asset for each urban elements including residential buildings, road networks, and trees.
3. Create procedural systems for residential buildings, road networks, trees, and the city.
    (a) Convert styles/rules to codes.
    (b) Establish control parameters for the procedural systems, which encompass the following aspects.
        i. Residential buildings.
            A. Height.
            B. Width.
            C. Length.

ii. Road networks.

A. Avenue count (parallel to the x-axis).

B. Road count (parallel to the y-axis).

C. Avenue spacing (distance between the avenues).

D. Road spacing (distance between the roads).

iii. Trees.

A. Main branches count.

B. Main branches height.

C. Sub branches count.

D. Branches length.

E. Leaves count.

F. Seed.

G. Tree height.

(c) Establish control parameters for the city, which includes the following.

i. Local Parameters.

A. Mimimum and maximum height, width, and length for residential buildings.

B. Mimimum and maximum avenue count, road count, avenue spacing, and road spacing for road networks.

C. Minimum and maximum tree height, branches count, branches length, and leaves count for trees.

ii. Global Parameters.

A. Amount of city's population which affects the height of the residential buildings.

B. Amounf of city's wealth which affects the height and length of the residential buildings.

C. Quality of city's transportation which affects the road and avenue count of the road networks.

D. Quality of city's environment which affects the tree height, leaves count, and branches count of trees.

4. Train a machine learning model using the gathered images in the case study.
   (a) Preprocess the data.
   (b) Train a machine learning model using various datasets in teachable machine to classify architectural styles.
   (c) Evaluate the trained models using the following metrics.
      i. Accuracy.
      ii. Precision.
      iii. Recall.
      iv. F1 Score.
      v. Receiver Operating Characteristic - Area Under the Curve (ROC AUC) score for multiclass classification.
      vi. Matthew's correlatiopn coefficient for multiclass classification.
   (d) Select the best performing model.
   (e) Deploy the best performing model.
5. Create a tool that measures performance of digital asset.
   (a) Create a script to retrieve polygon count.
   (b) Create a script to display storage space.
6. Develop a plugin in a 3D application that has the following functionalities.
   (a) Allows the urban planner to add a procedural system for an urban element to the scene.
   (b) Allows the urban planner to modify control parameters of a procedural system.
   (c) Allows the urban planner to measure architectural styles.

(d) Allows the urban planner to measure the performance of digital assets in terms of the following.

   i. Polygon Count.

   ii. Vertex Count.

   iii. Edges Count.

   iv. Estimated Storage Size.

(e) Allows the urban planner to export selected procedural systems in Filmbox (FBX) format for use in other 3D programs such as Maya, 3ds Max, and Microsoft 3D Viewer.

(f) Allows the urban planner to export all procedural systems in the scene in Filmbox (FBX) format for use in other 3D programs such as Maya, 3ds Max, and Microsoft 3D Viewer.

D. Significance of the Project

The study provides urban planners and developers with a tool that generates urban Philippine cities procedurally. This allows for a more cost-effective and scalable process for creating and modifying 3D cities for planning compared to traditional methods.

Due to the generated city's procedural nature, it also helps urban planners and developers design urban structures that achieve architectural harmony. This is because procedural systems use rules to generate structures, resulting in cohesive and harmonious content throughout the city. This greatly benefits urban planners and developers in creating sustainable and inclusive Philippine urban cities.

E. Scope and Limitations

This study aims to develop a procedural generation plugin for Blender, a widely-used open-source 3D creation software, with a specific focus on 3D modeling of Philippine cities. The plugin is designed to assist in creating and maintaining

digital replicas of cities and various urban elements. The study specifically concentrates on urban elements such as residential buildings, road networks, and trees.

However, it is important to acknowledge the following limitations of this study:

1. **Architectural Styles:** The generated urban residential buildings are constrained to the most prevalent architectural styles found in Taguig City, including Spanish colonial, contemporary, and vernacular styles. Other architectural styles may not be fully represented.

2. **Interior Details:** The generated urban buildings do not incorporate interior details. The focus is primarily on the external representation and structural aspects of the buildings.

3. **Tree Species:** The procedural trees are limited to portraying the most common tree species typically found in Philippine urban settings, such as acacia and mahogany. Other tree species may not be included in the generated models.

4. **Polygon Count and Detail Level:** The models generated by the procedural trees have a relatively low polygon count and may lack intricate details. They aim to provide a basic representation of trees rather than highly detailed and realistic models.

5. **City Elevation:** The elevation of cities, including variations in terrain and topography, is not considered in the generated models. The focus is primarily on the structural elements of the urban environment.

F. Assumptions

1. Blender, the open-source 3D creation software, is installed on the urban planners' workstations, as the procedural modeling plugin is an extension that integrates with Blender's functionality.

2. The urban planners using the system are familiar with navigating 3D applications, specifically Blender, as the procedural modeling plugin is designed to work within the Blender environment.

II. Review of Related Literature

This section provides a discussion of resources for this study's approach. It is divided into four sections namely urban planning, procedural systems, geographical visualizations, and active urban simulations.

A. Urban Planning

Urban planning drives urban development into a desired trajectory by utilizing various established instruments, practices, and modes of governance by public authorities [8]. Representation of urban settings has been a key instrument in the creation of plans for a particular place, which can be achieved through various means such as summary statistics, replicas, and maps.

Previous approaches to creating replicas involve the creation of physical models in miniature form. The advantage of this approach is the immersive nature of the replica, as it provides a grasp of how each element works together as a whole. These physical models play a crucial role during public consultations, which involve the transformation of an entire area. However, the downside of this approach is its cost in terms of time, money, and effort. Additionally, the generated physical model is static, making it difficult to modify and reuse.

Figure 1: Physical architectural model.

Digital replicas, on the other hand, are computer-generated representations of content. They significantly reduce the cost of creating a setting as they are created

using 3D authoring tools, which are less expensive than physical materials. However, the limitation of this approach is that the created models for each element are static, making them difficult to modify. This restricts the modifiability and reusability of the generated content, which can be addressed through procedural content generation.

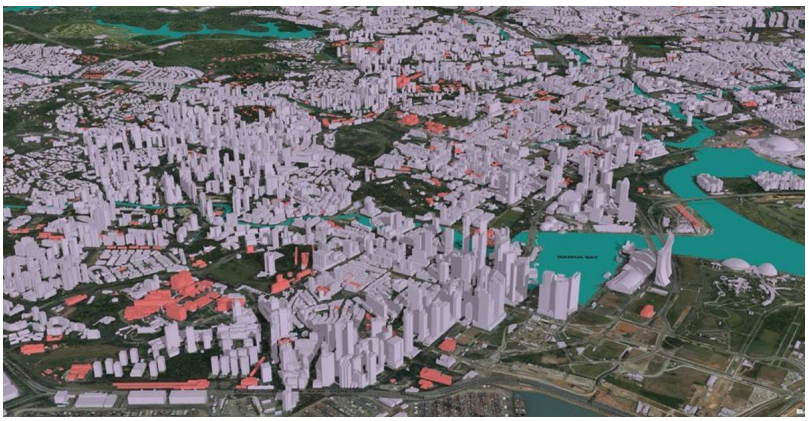

Figure 2: Digital replica of a city.

Roumpani [7] studied the usefulness of active urban city simulators, which involve a dynamic digital replica of a city for urban planning. These replicas use procedurally generated 3D city models that can easily be modified using simple controls. This enhances the interactivity and responsiveness of the represented setting, enabling the visualization of different versions of the city in different conditions. It is effective in supporting planning for urban cities.

Xu and Wang [9] showed a workflow involving procedural content generation of urban buildings to support sustainable city development. This workflow integrated a geographical information system (GIS) dataset to automatically generate models and settings. The built settings can estimate urban scale energy demand loads, which is beneficial in global climate change mitigation at the urban scale.

Hudson-Smith [10] also looked at using digital replicas for urban planning and found that they serve as a tool for urban planners in enhancing planning participation and design. They can also represent a building dynamically based on its purpose and time.

B. Procedural Systems

Procedural techniques take in parameters and in turn generate a model or effect algorithmically. This relies on pseudo-randomness heavily to create uncountable variations of generated content. This technique is applied in generating photo-realistic images and 3D geometry. Users provide procedures that are called to generate geometry [7]. In procedural approach, parameters through simple controls modifies geometry. This is significantly better than specifying details of a geometry manually.

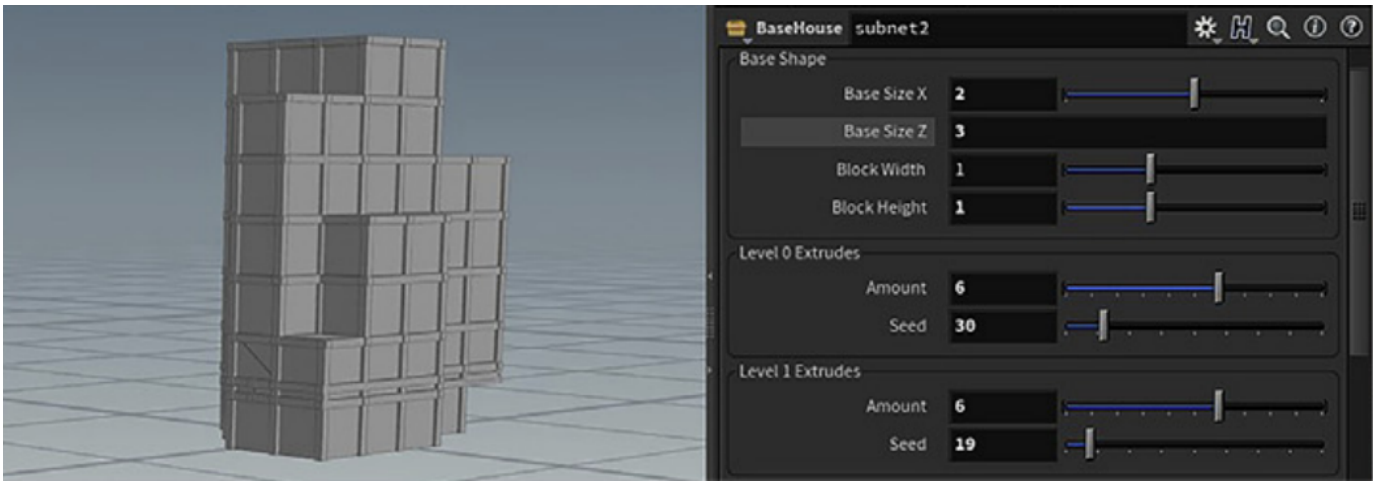

Figure 3: Procedural geometry with parameter controls. Adapted from AlFadalat and Al-Azhari [1].

Roumpani [7] used procedural methods to dynamically generate 3D city models for urban planning. The generated 3D city model depends on urban analytics such as population, employment, and growth rate. These analytics are changed with simple controls which demonstrates the interactivity and responsiveness of procedural cities. The study found that such urban modelling methods and simulations can support planning and provide a visualization of different versions of how the city would look like in different conditions.

The study presents the idea of maximizing the interactivity and usefulness of procedurally generated 3D cities for urban planning. This is done by using urban analytics such as population, employment, growth rate, and wealth of a city. These analytics can be toggled to show its distribution and effects in the

city. Another idea is the high degree of controllability of the generated 3D cities which allows for ease of customization for its users. This provides a framework on future procedural systems relevant to urban planning.

Alomía et al. [6] introduces a novel workflow for streamlining the urban 3D model creation process that involves procedural modeling. This workflow is tested by capturing urban data of the city of Bogota using a database containing geographic data. Procedural modeling techniques were then integrated with the urban data to create a 3D representation of the city in a short time. The generated city allowed for ease in the modification of its different elements. The study found that the workflow can be used effectively for urban planning activities and simulations.

Paranjape et al. [11] presented a system that can generate a huge volume of towns, intersections, and scenarios for training autonomous vehicles. The system uses a combination of various procedural techniques and supports the generation of procedural road mesh and navigation mesh based on road network data. These mesh data were used for simulations of vehicles and pedestrians which are controlled through behavior trees. The study found that the generated road networks, vehicle and pedestrian behaviors, and scenarios can create a simulation environment for autonomous vehicles.

AlFadalat and Al-Azhari [1] explored the challenge of integrating new and existing structures to achieve spatial congruence. The proposed framework of the study was to integrate a procedural modeling technique to create new structures based on the grammar of the existing structures to achieve harmony. The generated structure was integrated with augmented reality technology for visualization. The approach was then tested on a group of university students and the results were evaluated using a machine-learning model. The study found that the method is effective and can be regarded as a step toward achieving spatial congruence.

C. Geographical Information System

Geographical Information System (GIS) represents a map of geographical objects. These objects contain spatial information and are linked to reality. In three-dimension, GIS can represent houses, roads, and trees. Since this system also provides an overview of a particular location, it plays a significant role in urban and environmental management, and location routing [12].

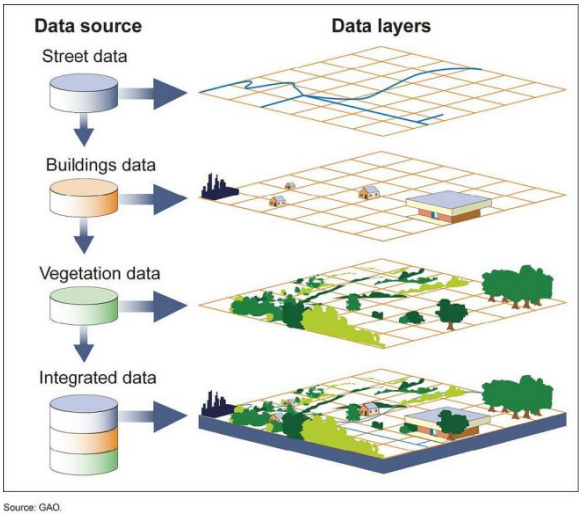

Figure 4: Geographic Information System.

Syafuan et al. [12] produced GIS maps to represent 3D models of a campus map accurately. OpenStreetMap (OSM), an open-source geographical database of the world, data was used to obtain a base map of the campus. The base map consists of geographical information such as height of buildings, trees, mountains, and road networks. These information were used to generate a base model of the elements of the campus for further processing.

Alomía et al. [6] introduced a workflow in generating urban 3D city model using OSM geographical data. The workflow captured urban elements of a city such as houses. buildings, parks, and roads. These data were processed and exported as digital terrain model (DTM) and digital surface model (DSM) formats. DTM was used for 3D urban buildings while DSM was used for the road network. This workflow was validated ans was able to reconstruct various classes of buildings and highway network in a short time.

Badwi et al. [13] created a 3D-GIS for virtual urban simulation using a game engine to support urban decision-making. This system takes 2D spatial data as inputs and generates dynamic 3D models as outputs. These models were generated through City Engine, a commercial 3D modelling software. The results of the study showed that it was able to represent urban elements realistically.

D. Active Urban Simulations

Active urban simulators are 3D models of cities that are able to evolve computationally [7]. It dynamically changes depending on a scenario or context. In turn, it enhances the responsiveness of the generated city. One method of creating these urban simulators is through procedural generation. In this approach, a content is generated by processing parameters.

Roumpani [7] created active urban simulators through procedural cities. Users can change the generated content by interactively modifying the parameters of its procedural elements. These parameters are urban analytics such as growth rates, unemployment rates, and city wealth. The created active urban simulators can support planning and communication of urban life.

Paranjape et al. [11] presents a system that creates road scenarios procedurally. It can generate huge scale road networks or intersections as a mesh. This can be used for generating various scenarios involving vehicles and pedestrians such as urban cities.

III. Theoretical Framework

A. Contextual Design

Architecture is a reflection of society, as it embodies context elements such as time, purpose, location, conditions, and surroundings. It reflects people's aspirations and a place's sense of identity at a specific point in time [14]. However, inconsistent architectural styles can disrupt this reflection, resulting in a loss of architectural character and continuity.

Contextualism theory promotes the idea that architectural styles should strive for harmony with their surroundings. Contextual design is an approach that achieves this harmony by ensuring that structures and their context elements in the environment establish a common identity, leading to cohesiveness [15]. Contextual elements include urban conditions such as building density, sidewalks, vegetation, and population.

Procedural modeling is a technique that addresses the issue of contextual design [1]. It ensures that all generated structures and context elements follow a set of rules which establish uniformity. However, this approach can result in generated content that is too uniform, reducing its believability. To address this, randomization can be introduced during some stages of 3D asset creation.

B. Architectural Styles, Rules, and Vocabulary

Architectural styles refer to the quintessential features and design principles of buildings and structures in a specific region or period. These are shaped by factors such as cultural influences and historical context [14]. Understanding architectural styles is important for determining the structural compositions of the city.

Architectural style rules refer to the guidelines that govern the design of buildings and structures in a particular architectural style. These are used to ensure that the buildings and structures adhere to the style. For example, an architectural style rule for a Gothic style building might be the use of pointed arches and

ribbed vaults in the design of the structure.

Architectural vocabulary refers to the specific terms, phrases and language used to describe and classify architectural styles [14]. It is an important tool for understanding architectural styles, and is used to communicate the design principles and characteristics of a particular style. For example, the vocabulary used to describe a Gothic style building might include words such as "pointed arch," "ribbed vault," "flying buttress," and "tracery."

C. Generative Design

Generative design is a design exploration process that uses computational algorithms to quickly generate a wide variety of high-performing design alternatives for a given problem [16]. This approach is useful in the field of architecture and urban planning since it allows for the consideration of possible design solution in a short time.

D. Procedural Modeling

Procedural modeling is a way of generating content algorithmically, using a set of rules and procedures. In urban development and planning, procedural modeling can be used to generate 3D models of cities, buildings, and other urban elements in a consistent and controlled manner [6].

One advantage of using procedural modeling is that it allow for the creation of large amounts of detailed and varied content in a relatively short amount of time. This can be especially useful in urban development and planning, where large numbers of buildings and other urban elements need to be created and placed in a realistic and coherent manner.

D..1 Parameter-Based Procedural Modeling

A parameter-based procedural modeling is a method for generating 3D models that allows for a high degree of control and flexibility in the characteristics of

the generated models [13]. In this approach, urban planners can specify a set of input parameters that control the properties and characteristics of the generated models. These parameters can be adjusted to control the size, shape, layout, and other properties of the generated models.

In this study's approach, a parameter-based procedural modeling will be used to generate a wide range of different types of urban environments, such as buildings, streets, and landscapes. This approach will take as input a set of parameters that specify the desired characteristics of the generated models, such as the width, height, and number of floors of a building.

To implement the parameter-based procedural modeling, Blender software and its geometry nodes will be used to create the algorithms that control the generation of the models. Additionally, we will be utilizing several libraries and tools for the implementation of this system, such as the Python programming language for the development of scripts that handle the system's logic, input, and output.

D..2 Procedural Algorithm

Blender's geometry nodes feature allows for the creation of complex 3D models through a node-based interface. The interface allows for the generation and manipulation of 3D geometry, which is crucial for the procedural algorithm in this study. Nodes can be interconnected to create networks, resulting in complex and detailed 3D models. Mathematical operations nodes will be utilized to generate the base shape of the building such as the floor plan, height, and number of floors. Subsequently, fractal noise nodes will be connected to introduce variations in shape such as the roof or window placement [17].

Noise nodes will be utilized to introduce variations in texture to simulate materials such as brick or stone. A displacement node will be utilized to create surface details such as bumps or ridges. Additionally, skinning nodes will be utilized to create complex shapes for features such as window frames and balcony railings. The specific set of nodes and their interconnection will be outlined in the following

sections.

1. Input Nodes

   Input nodes are data container, which allow various type of values, integers, colors, vectors, and strings, that is used for other nodes.

2. Geometry Nodes

   Geometry nodes are operations that are used to modify the geometry of 3D meshes and volumes.

3. Instances Nodes

   Instances nodes allow for efficient replication of 3D objects by linking multiple copies to a single shared set of data. It enables easy modification and updates of the replicated objects, which is particularly useful for procedural generation where many repeating elements are present.

4. Mesh Nodes

   Mesh nodes manipulate the 3D geometry of an object. It acess the vertices, edges, and faces of an object, and manipulate them using a wide variety of mathematical operations and functions.

5. Material Nodes

   Material nodes provide materials for a 3D object.

6. Utilities Nodes

   Utilities nodes are mathematical operations and functions that is used to modify object data.

7. Vector Nodes

   Vector nodes are functions that operate on vector quantities.

E. 3D Asset Creation Process

3D asset creation relies on a multitude of multidisciplinary processes. There are plenty of methods developed in various fields for creating assets. The approach used in this study follows basic workflow and optimization techniques commonly used in film-making and game development [2].

E..1 Pipeline/Workflow

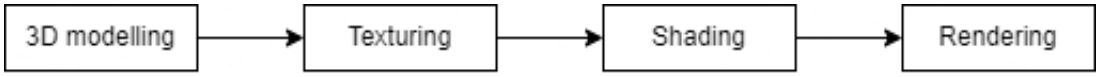

Figure 5: 3D asset creation pipeline. Adapted from [2]

1. 3D Modelling

   3D modelling is the process of representing 3D objects in a computer. A 3D model refers to a 3D object that is the final output of 3D modelling process. It is made within a computer-based 3D modelling software or a programming language.

   A mesh is the core of 3D model. It is made of a collection of vertices and polygons that define the shape of a 3D object. It also stores coordinate data which the software can use to identify the location of each vertical and horizontal point, relative to a reference point.

2. Texturing

   Texturing is the process of superimposing 2D images or maps in a 3D model. These images or maps represent information such as color, height, bump, normals, roughness, and other attributes of a 3D model. Texturing can portray three main properties of 3D object in a 3D environment such as the material, light effects, and tertiary details.

   The process of texturing is mainly composed of UV unwrapping and texture painting. UV unwrapping is the process of representing 3D mesh into 2D

coordinate system called UV. This process helps to superimpose 2D images into the 3D models. Texture painting is the process of creating 2D images that contains color information, surface details, and visual properties of a 3D model.

3. Shading

   Shading is the process of computing, simulating, or altering the color of objects in the 3D scene as seen from a viewpoint. This process is performed by a program called a shader.

4. Rendering

   3D rendering is the process of representing 3D models or 3D environments into a 2D image. It creates a 2D representation of a 3D environment based on its size, shape, texture, and shaders.

   Physically based rendering is a rendering approach that simulates the behavior of light in the real world. This approach is usually used to ahieve photorealism. It also have some advantages as compared to pure artistic techniques such as physical accuracy, intuitive parameterization, and portability.

F. Convolutional Neural Network

Convolutional neural network (CNN) is one of the most representative and crucial neural networks in the field of deep learning. It is able to harness a massive amount of data to achieve a promising result. It can be used for data in 1D, 2D, and multidimensional. 2D CNN is widely used in image classification and it produces good results in this field [4].

CNN is a kind of feedforward neural network that is able to extract features from data with convolution structures. First, it takes an input image then resize them before passing onto further layers for feature extraction. Then the convolution layer act as filters for the images. This obtains feature sets from images.

These extracted feature sets are passed to a pooling layer. This layer reduces the dimensions of the large images while maintaining the most important information in them. Then the rectified linear unit layer replaces every negative number of the pooling layer with 0. This helps the CNN to be mathematically stable. Lastly, the fully connected layers take the high-level filtered images and translate them into categories with labels [18]. Figure 6 shows this process.

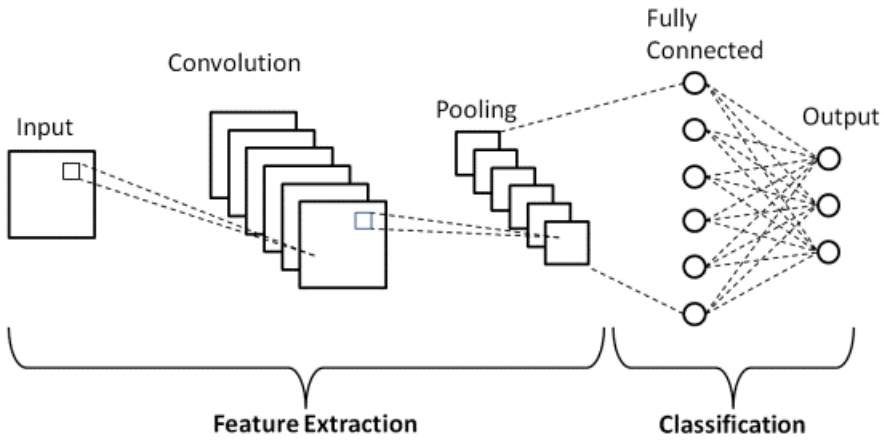

Figure 6: Basic Architecture of CNN

G. MobileNetV2

MobileNet is a type of CNN designed for efficient image classification on mobile and embedded devices. It uses depthwise separable convolution architecture to reduce the number of parameters and computations, making it well-suited for low-resource environments. The model is trained on large datasets to classify images into a set of predefined classes. Due to its efficiency and high accuracy, MobileNet is particularly well-suited for image classification tasks that require real-time processing, such as those found in augmented reality and camera-based user interfaces. The architecture allows for faster computation and low power consumption which makes it a good fit for mobile and embedded devices, as well as other edge devices [19].

H. Transfer Learning

Transfer learning is a machine learning technique that uses knowledge gained from a source task to improve the performance of a target task. This technique is useful when the availability of data for a specific task may be limited. By leveraging the knowledge gained from the source task, the model is able to make better use of the data available for the target task. This can potentially lead to imrpoved performance as compared to training a model from scratch. In addition, it can reduce the amount of data and computational resources required to train a model on the target task [19].

I. Performance Metrics

Performance metrics are used to evaluate the effectiveness of a model on a given task. On a multiclass classification problem, there are a variety of metrics that can be used to assess the performance of the model.

In this section, we will discuss several common metrics for evaluating the performance of multiclass classifiers, including confusion matrix, precision, recall, ROC AUC score, Cohen's Kappa score, Matthew's correlation coefficient, and log loss.

I..1 Confusion Matrix

A confusion matrix is a table that summarizes the model's prediction and it visualizes the model's performance across different classes. For a multiclassification problem with n classes, the confusion matrix is a $n \times n$ table. The rows typically correspond to the true classes of the samples while the columns correspond to the predicted classes. The entries in the confusion matrix are the counts of samples that fall into each combination of true and predicted classes.

In a confusion matrix, each sample can fall into four categories as shown below.

1. True Positive (TP). The number of samples that were correctly classified as

a particular class.

2. True Negative (TN). The number of samples that were correctly classified as not a particular class.

3. False Positive (FP). The number of samples that were incorrectly classified as a particular class.

4. False Negative (FN). The number of samples that were incorrectly classified as not a particular class.

I..2 Precision

Precision is defined as the number of true positive predictions made by the classifier, divided by the total number of positive predictions. Mathematically, precision is defined as $\frac{TP}{TP+FP}$.

I..3 Recall

Recall is a measure of the completeness of a classifier's predictions. It is defined as the number of positive predictions made by the classifier, divided by the total number of actual positive cases in the data. Mathematically, it is defined as $\frac{TP}{TP+FN}$.

I..4 F1 Score

F1 score is defined as the harmonic mean of the classifier's precision and recall. It is mathematically defined as $\frac{2\times Precision\times Recall}{Precision+Recall}$. In general, a higher F1 score indicates better performance of the classifier.

I..5 Receiver Operating Characteristic - Area Under the Curve (ROC AUC) score for multiclass classification

The ROC AUC score is defined as the average of the individual AUC scores for each class, where the AUC score is the area under the curve of the classifier's

true positive rate (TPR) versus false positive rate (FPR) for that class. It is mathematically defined as $\frac{AUC_1+AUC_2+...+AUC_n}{n}$ where AUC_i is the AUC score for class i, and n is the total number of classes.

I..6 Matthew's correlatiopn coefficient for multiclass classification

Matthews correlation coefficient (MCC) is defined as the geometric mean of the classifier's recall and specificity, normalized by geometric standard deviation of the two. MCC is defined mathematically as $\frac{Recall \times TNR}{\sqrt{(1-Recall)\times(1-TNR)}}$.

In general, a higher MCC score indicates better performance of the classifier. A MCC score of 1 indicates perfect agreement between the classifier and the ground truth labels, while a score of -1 indicates perfect disagreement. A score of 0 indicates no agreement beyond chance.

IV. Design and Implementation

A. Proposed Approach

In this section, we discuss a contextual design approach that uses procedural systems to represent Philippine urban cities accurately, and efficiently in terms of running time and storage space. This will be achieved by creating an interface for the parameters of procedural systems that will allow for ease in control for its users. To ensure that the generated city represents a Philippine urban setting, we will conduct a case study to analyze the most significant architectural vocabulary. This vocabulary will then be used as a rule that will be used in the procedural systems. The generated city will be rendered using a 3D application and will be analyzed with a machine learning model and performance indicators. The approach is shown in figure 7.

A..1 Case Study Selection

The study will focus on the urban elements in Taguig city, a 1st class highly urbanized city in Metro Manila, as the representative sample of current and future Philippine urban settings. A thorough analysis of the architectural features of these urban elements will be conducted to derive a set of architectural design principles, or vocabulary, that will serve as the basis for the procedural systems used in this study.

A..2 Digital Asset Creation

Blender 3D will be used as a 3D authoring software to create an asset that will be used by the procedural system. The process will begin by modularizing models of each urban elements. Then modular textures will be created that will be packed into trim sheets to reduce the usually high storage cost of representing a city. These textures will be used to create a physically based shaders to render cities in a realistic manner.

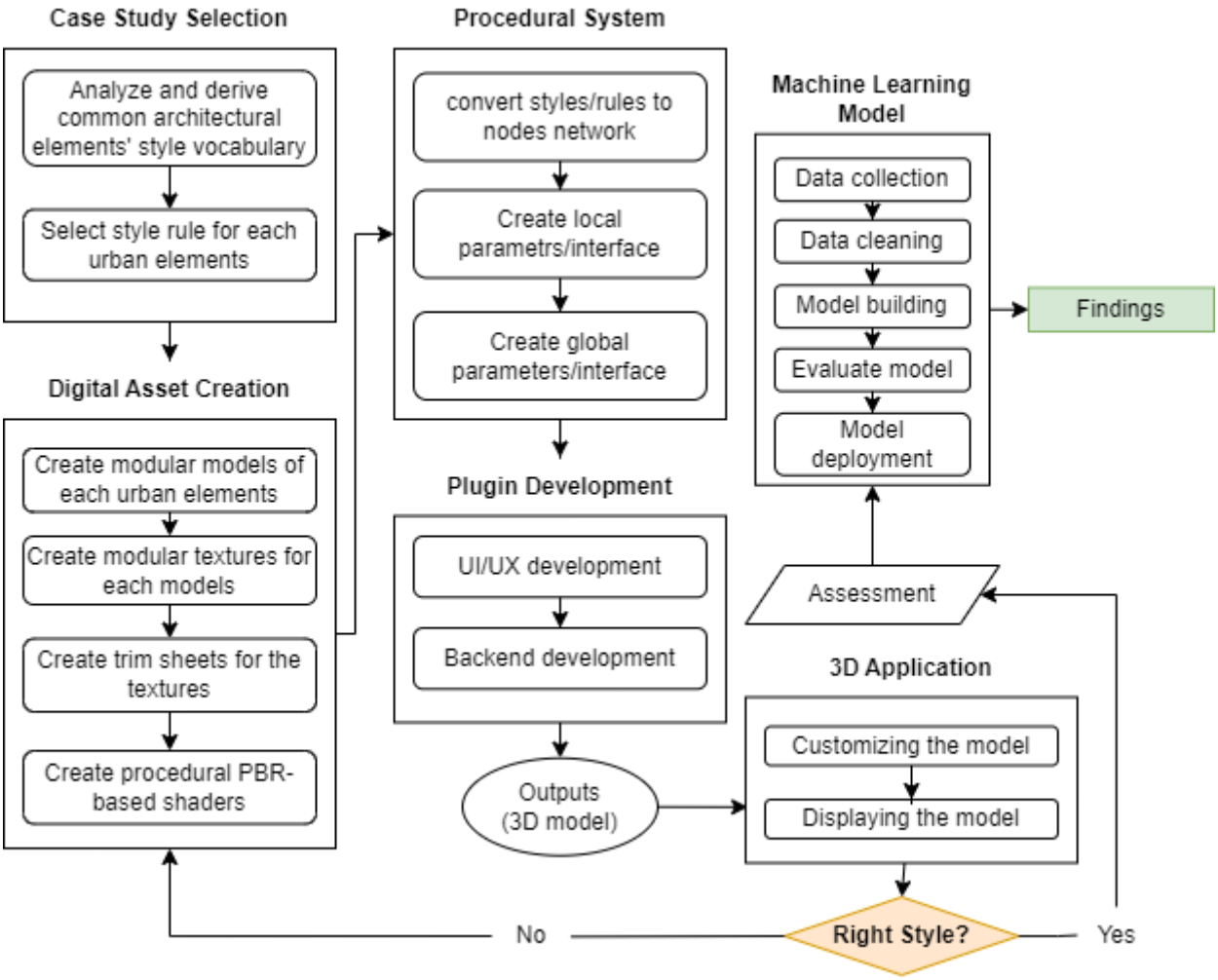

Figure 7: Proposed approach. Adapted from AlFadalat and Al-Azhari [1].

A..3 Procedural System

Geometry nodes inside Blender will be used to create procedural systems for each urban elements and one procedural system to control the city as a whole. It is a visual programming interface for modifying geometry of a 3D object [17]. This process will take the rules from the architectural vocabulary to ensure that the generated urban elements adhere to the style of Philippine urban cities. Local and global parameters will be developed to allow its users to easily modify the generated urban element.

A..4 Plugin Development

Python, which is a scripting language of Blender, will be used to create the plugin. The plugin contains three modules which are the procedural city module, procedural systems module, and evaluation module.

A..5 3D Application

Rendering engines inside Blender such as Eevee and Cycles will be used to render generated urban elements. Eevee will be used for rendering the generated urban elements and city in real-time. Cycles will be used for rendering similar content in a more realistic manner.

A..6 Machine Learning Model

Teachable machine will be used to create the machine learning model. The dataset that will be used to train the model are screenshots of urban elements in Taguig city which will be obtained from google maps street map view. It will be integrated in Blender using Python and TensorFlow. The model will be used to evaluate the similarity of rendered urban elements and cities to Philippine urban cities. Results from this process will determine how well the procedural systems represent Philippine urban cities as well as its urban elements.

A..7 Performance Evaluation

Polygon count and storage space will be monitored to evaluate the performance of the procedural system in terms of memory space and rendering time.

B. Use Case Diagram

There is one primary and one secondary actor in the system. The primary actor is the urban planner which initiates functionalities of the procedural system. The urban planner can initiate four main functionalities of the system such as selecting procedural content, evaluating generated content, rendering generated content, and inputting map coordinates. The system is the secondary actor which reacts on the interaction of the urban planner with the procedural system. The system is responsible for processes such as adding procedural content, evaluating generated content, verifying coordinates, and autopmatically generating a city. The interaction between the actors and the procedural system is shown in figure 8.

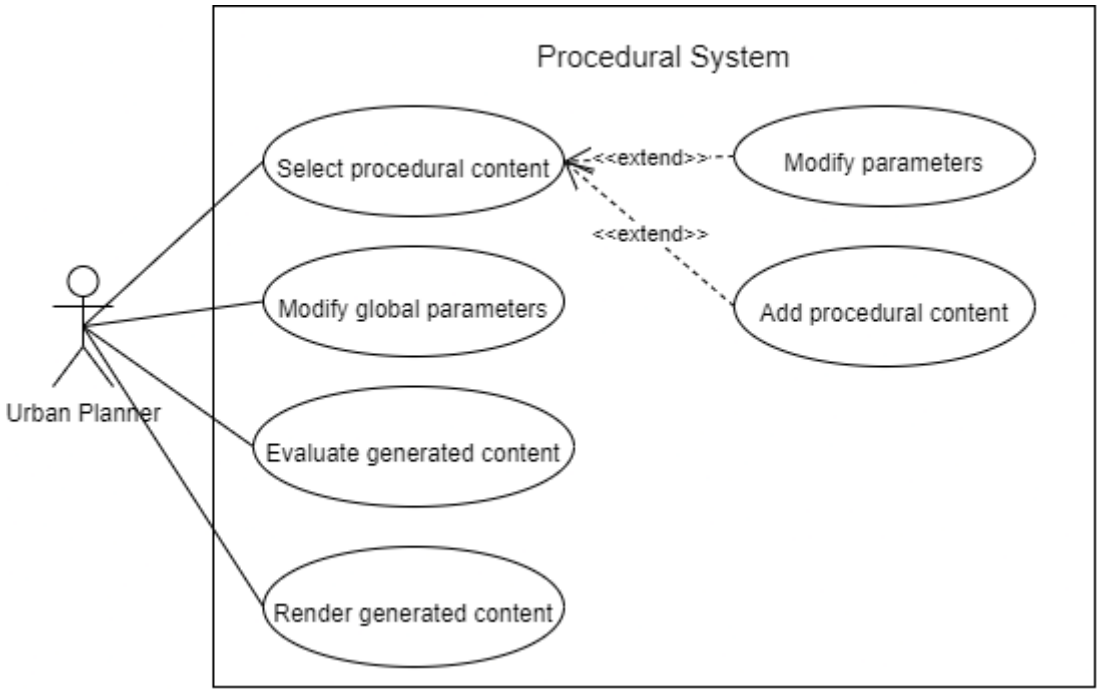

Figure 8: Use Case Diagram

C. User Flowchart

The flowchart in figure 9 describe the process of generating procedural cities using the procedurals system. It contains of two major processes with the first one being obtaining procedural content, and then followed by evaluating the generated content. The former process consists of selecting procedural content, addition of the content to the scene, and completely populating the scene with procedural contents to form a procedural city. The latter process encapsulates the rendering, evaluating, and generating performance report of the scene.

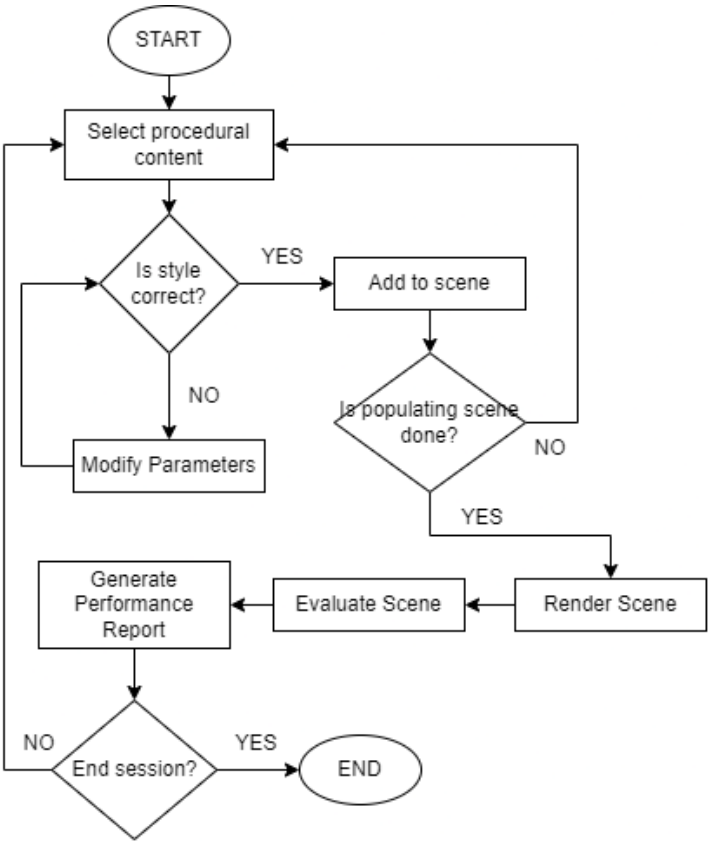

Figure 9: Flowchart of generating procedural city by the user.

V. Results

The results of the study on the procedural modeling plugin for sustainable urban development and planning, with a specific focus on the 3D modeling of Philippine cities, are presented in this section. The findings are organized based on the specific objectives of the study.

A. Case Study on Taguig City

A..1 Gathering Images

The case study involved gathering images of urban cities and architectural elements in Taguig City, Philippines. A comprehensive collection of images was obtained to represent the diverse architectural styles and urban elements present in the city.

Taguig City is politically subdivided into 28 barangays, which are further categorized into two districts: the first district with 15 barangays and the second district with 13 barangays. For this study, data were collected from three barangays in each district.

In the selection process, five barangays were chosen based on their high population numbers, namely Lower Bicutan, New Lower Bicutan, and Ususan from the first district, as well as Pinagsama and Western Bicutan from the second district. Additionally, Fort Bonifacio, a highly urbanized area from the second district, was selected as one of the barangays for data collection.

Barangay	**Spanish Colonial**	**Contemporary**	**Vernacular**	**Total Samples**
Lower Bicutan	0	2	34	36
New Lower Bicutan	2	4	15	21
Ususan	0	16	2	18
Fort Bonifacio	0	32	4	36
Pinagsama	0	0	13	13
Western Bicutan	0	2	10	12
Total	2	56	78	136

Table 1: Data Samples by Barangay

The data collection process involved utilizing Google Maps' Street View feature. A comprehensive set of 136 samples was obtained from various barangays in Taguig City. Among the collected samples, the distribution of architectural styles was as follows: Vernacular style accounted for the highest number of samples (78), followed by Contemporary style (56). However, the Spanish Colonial style was represented by only three samples, resulting in an imbalanced dataset.

	Spanish Colonial	**Contemporary**	**Vernacular**	**Total Samples**
Total	27	27	27	81

Table 2: Rectified Data Samples

To address this data imbalance, an additional 25 Spanish Colonial samples were gathered using Google Images. Subsequently, random sampling was performed on the Contemporary and Vernacular styles to obtain 27 samples each, ensuring an equal representation across all architectural styles. This approach aimed to rectify the initial data imbalance and foster a more balanced dataset for subsequent analysis.

In Figures 10, 11, and 12, sample images representing the Contemporary, Spanish Colonial, and Vernacular architectural styles are provided, respectively. These images serve as visual examples showcasing the distinct characteristics and design elements of each architectural style.

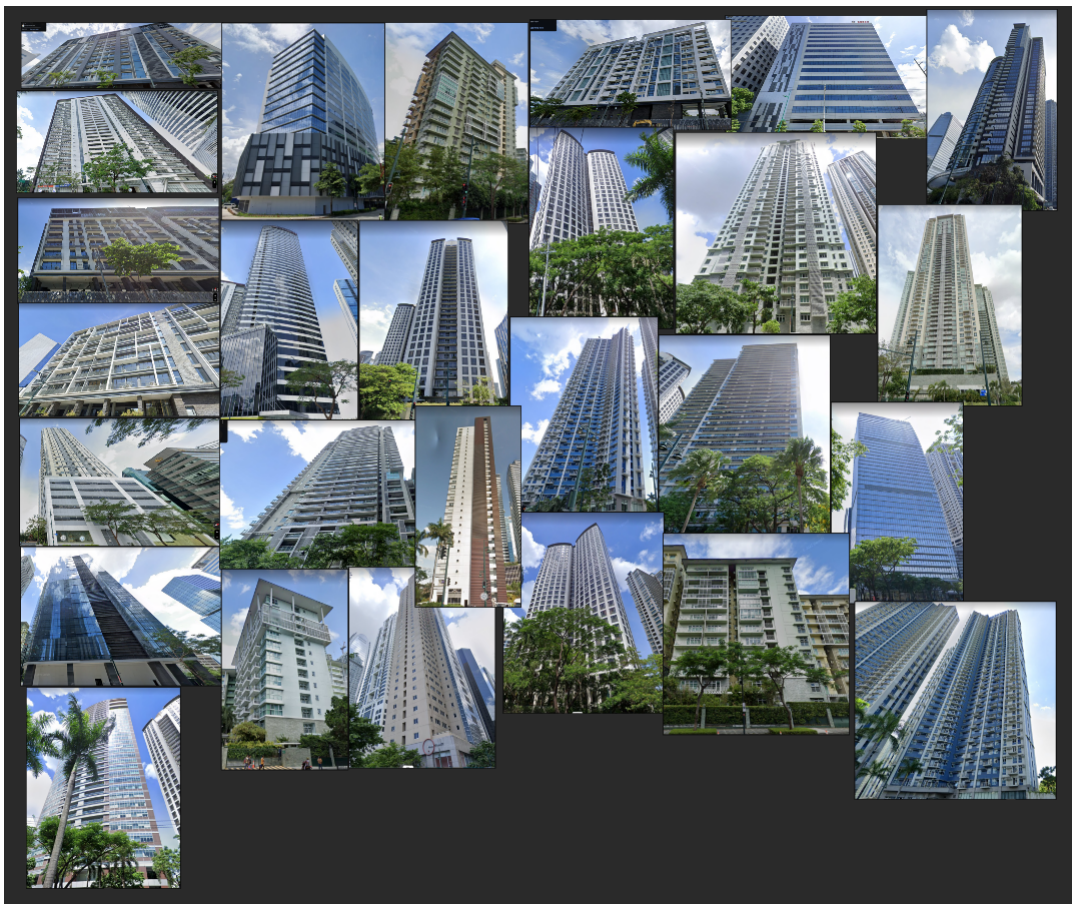

Figure 10: Contemporary Architectural Style Samples

Figure 11: Spanish Colonial Architectural Style Samples

Figure 12: Vernacular Architectural Style Samples

A..2 Selection of Architectural Style Rules

This section discusses the architectural style rules associated with the facades of each architectural style present in the collected dataset, focusing on the Philippine setting. The architectural styles analyzed include Spanish Colonial, Contemporary, and Vernacular.

Contemporary The Contemporary architectural style in the Philippine setting represents modern design trends. Some of the architectural style rules associated with Contemporary facades include:

- Clean lines and minimalist aesthetics
- Use of modern materials such as glass, concrete, and metal panels
- Large windows and floor-to-ceiling glass walls
- Integration of outdoor spaces and balconies
- Unique facade treatments and innovative design elements

Spanish Colonial The Spanish Colonial architectural style in the Philippines reflects the influence of Spanish colonization. The following architectural style rules are commonly observed in Spanish Colonial facades:

- Thick masonry walls or stucco exteriors
- Arcaded windows or balconies with wrought iron grilles
- Red tile roofs or clay roofs
- Decorative cornices and friezes
- Elaborate entrance doors with ornamental details

Vernacular The Vernacular architectural style in the Philippines encompasses local, traditional building practices. The following architectural style rules are commonly found in Vernacular facades:

- Use of locally available materials, such as bamboo, thatch, or wood
- Simple and functional designs with emphasis on practicality
- Protruding roofs or wide eaves for protection against the tropical climate
- Decorative elements showcasing regional craftsmanship
- Adaptation to local cultural practices and climate conditions

Analyzing the facades of buildings within these architectural styles allows for a focused examination of their distinct features and characteristics. By understanding the architectural style rules specific to facades, accurate classification and identification of architectural styles can be achieved, contributing to a comprehensive analysis of architectural diversity within the Philippine context.

B. Digital Asset Creation

The data gathered from the previous sections served as the foundation for the creation of digital assets. Through the use of reference materials and 3D modeling software, representations of the facades of different architectural styles in the Philippine setting were created.

B..1 Residential Buildings

Six residential buildings were created, consisting of three regular residential buildings and three high-rise residential buildings.

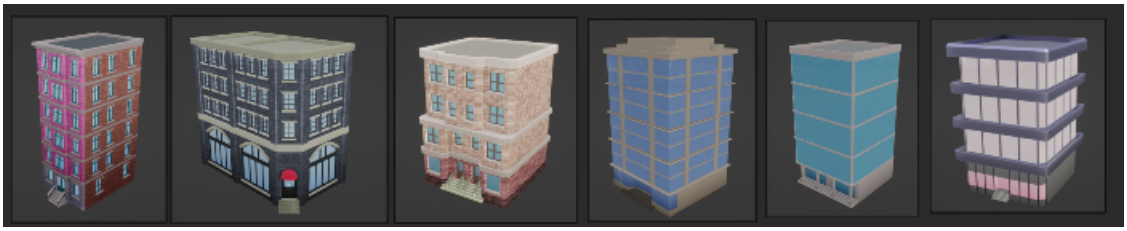

Figure 13: Digital Residential Buildings

B..2 Roads

Three components of roads were created: a 2-lane road, a 4-lane road, and an intersection featuring stoplights and pedestrian lanes.

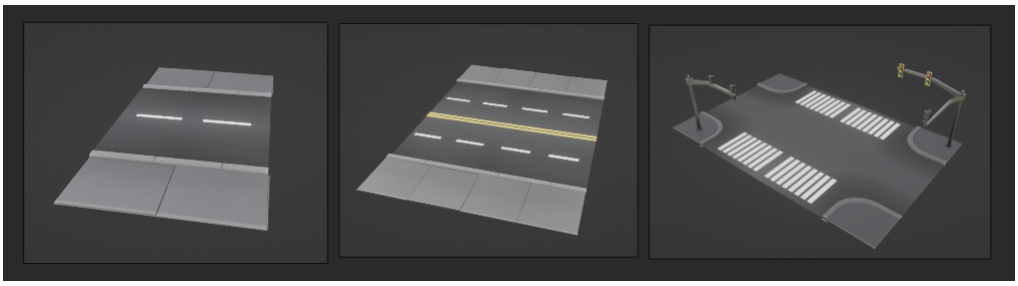

Figure 14: Digital Roads

B..3 Trees

A variety of tree models were created to enhance the realism of the digital environment.

Figure 15: Digital Trees

Figure 15 showcases the base models of trees that were created. These tree models were designed to mimic the appearance of different tree species commonly

found in the Philippine setting. The trees vary in size, shape, and foliage to provide visual diversity within the digital environment.

C. Procedural Systems Creation

The creation of procedural systems involved using Blender's Geometry nodes, a visual programming tool, to generate a wide variety of digital assets with adjustable control parameters. This allowed for interactive editing and customization of the assets created in the previous section. By leveraging Blender's Geometry nodes, the procedural systems provided a flexible and efficient way to generate diverse variations of digital assets, including buildings, roads, and trees, while ensuring realism and accuracy within the virtual environment.

C..1 Residential Buildings

Six static residential building models were used as a foundation for creating procedural systems. These models served as the base templates from which variations were generated. Control parameters such as height, width, and length were implemented to enable the creation of diverse variations of the static models. By adjusting these parameters, the procedural system could generate residential buildings with different dimensions and proportions, allowing for greater flexibility and customization in the digital asset creation process.

Figure 16: Procedural Residential Building A

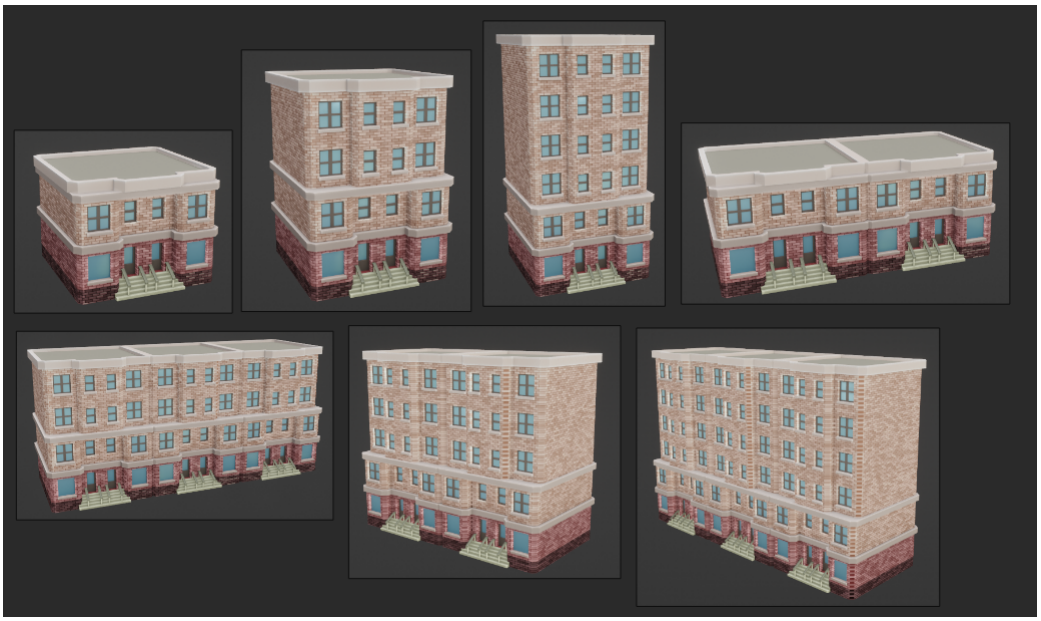

Figure 17: Procedural Residential Building B

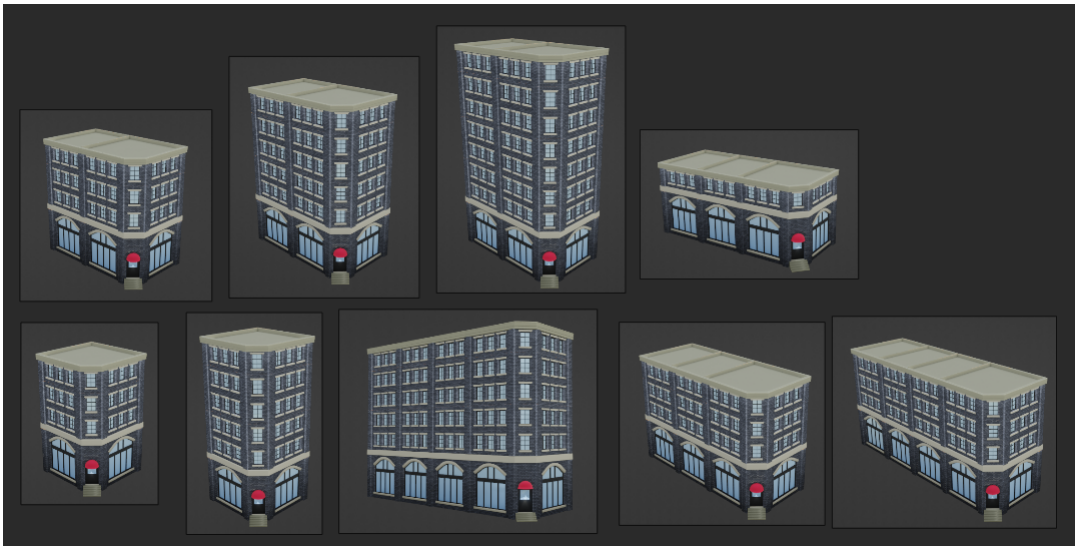

Figure 18: Procedural Residential Corner Building A

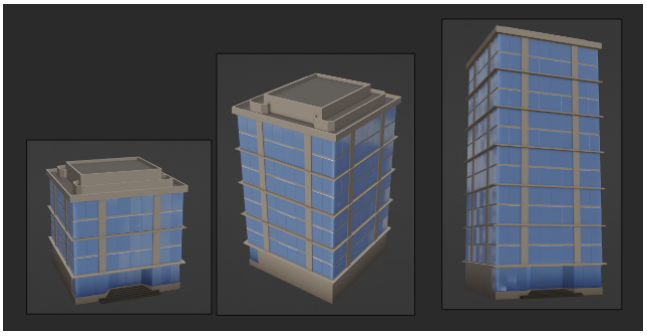

Figure 19: Procedural Skyscraper A

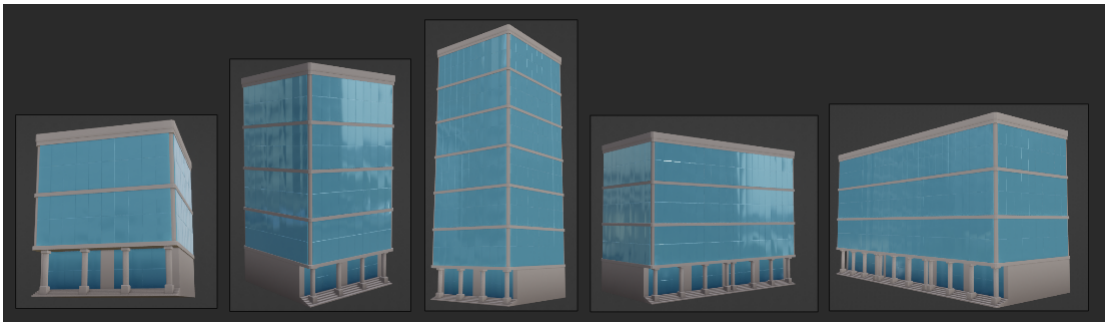

Figure 20: Procedural Skyscraper B

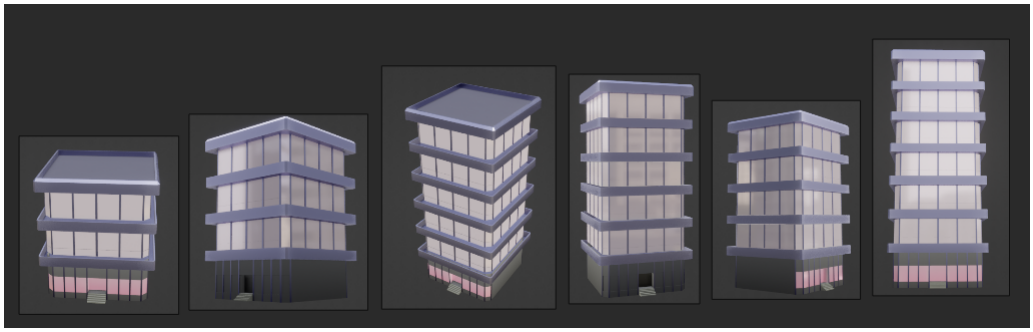

Figure 21: Procedural Skyscraper C

C..2 Roads

The procedural system includes the creation of roads with different configurations. Two-lane and four-lane roads were generated, along with an intersection that incorporates stoplights and pedestrian lanes.

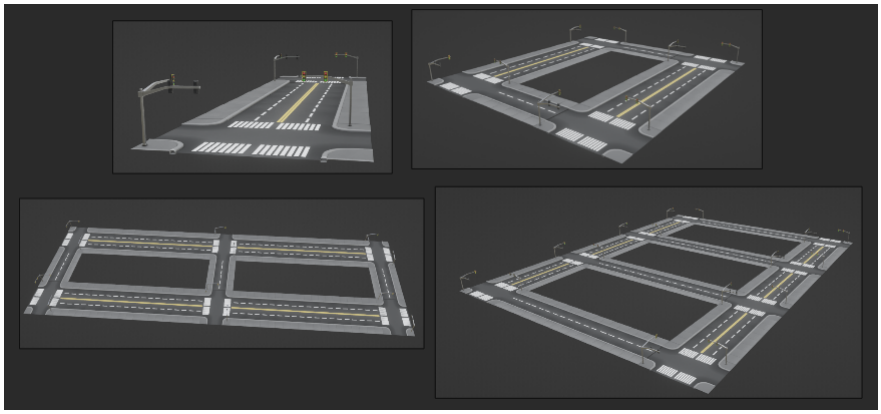

Figure 22: Procedural Road System

The generated road system, as depicted in Figure 22, showcases the layout and design of the roads, including their lane count, intersections, and pedestrian accommodations.

C..3 Trees

The procedural system includes the generation of tree models to enhance the visual environment. Various types of trees were created to add realism and diversity to the scene.

Figure 23: Procedural Tree Models

The generated tree models, as shown in Figure 23, showcase different tree types with variations in shape, size, and foliage. The control parameters, such as branch count, tree height, branch length, branch height, leaves count, and randomization, allow for the customization and creation of a wide range of tree variations. By adjusting the control parameters, users can create unique tree assets tailored to their specific requirements.

D. Machine Learning Model

In this subsection, the machine learning model utilized in the study is described. The model plays a crucial role in classifying architectural styles and assessing the adherence of generated models to specific styles.

The machine learning model employed in this study follows a supervised learning approach, where a labeled dataset is used for training and evaluation. The model is trained to classify architectural styles based on input images of urban elements.

D..1 Dataset

The dataset used in this study serves as the baseline for training and developing the architectural style detector in the procedural modeling plugin. It comprises a collection of architectural styles and their corresponding sample counts.

Table 3 presents the architectural styles included in the baseline dataset along with their respective sample counts. The dataset encompasses three architectural styles: Contemporary, Spanish Colonial, and Vernacular. Each architectural style is represented by a specific number of samples, as indicated in the "Sample Count" column of the table.

Architectural Style	**Sample Count**
Contemporary	27
Spanish Colonial	27
Vernacular	27

Table 3: Baseline Dataset

The baseline dataset provides a starting point for the development of the procedural modeling plugin by offering a representative collection of architectural styles. As the study progresses, this dataset can be augmented or expanded to include additional architectural styles and samples, allowing for more comprehensive training and enhancing the plugin's ability to generate diverse and accurate 3D models of Philippine cities.

D..2 Data Preprocessing

The images were uniformly resized to a resolution of 640x640 pixels. This resizing ensured consistency and facilitated efficient processing while preserving the overall content and visual characteristics of the original images. By standardizing the images through this preprocessing step, the dataset was appropriately prepared for subsequent stages, such as data augmentation and machine learning model training.

D..3 Data Augmentation

Two variations of the dataset were generated using different data augmentation techniques, and each training example produced three outputs to further enrich the dataset.

The data augmentation steps for each dataset variation are as follows:

1. **Baseline Dataset**
   (a) The original, unaltered images obtained during the case study served as the baseline for this variation.

2. **Dataset Variation 1**
   (a) Horizontal flipping
   (b) Cropping with a minimum zoom level of 15% and a maximum zoom level of 20%
   (c) Rotation between -20 degrees and 20 degrees
   (d) Shear transformations of up to ±15 degrees in both horizontal and vertical directions.

D..4 Machine Learning Model Training

The machine learning model utilized in this study underwent training using various datasets, as illustrated in Table 4.

Architectural Style	**Baseline**	**Variation 1**
Contemporary	27	81
Spanish Colonial	27	81
Vernacular	27	81
Total	**81**	**243**

Table 4: Samples per Data Variation

The dataset was divided into an 85% training set and a 15% test set. The model was trained using 100 epochs, a batch size of 32, and a learning rate of 0.001. These hyperparameters were chosen to optimize the training process and achieve desired performance.

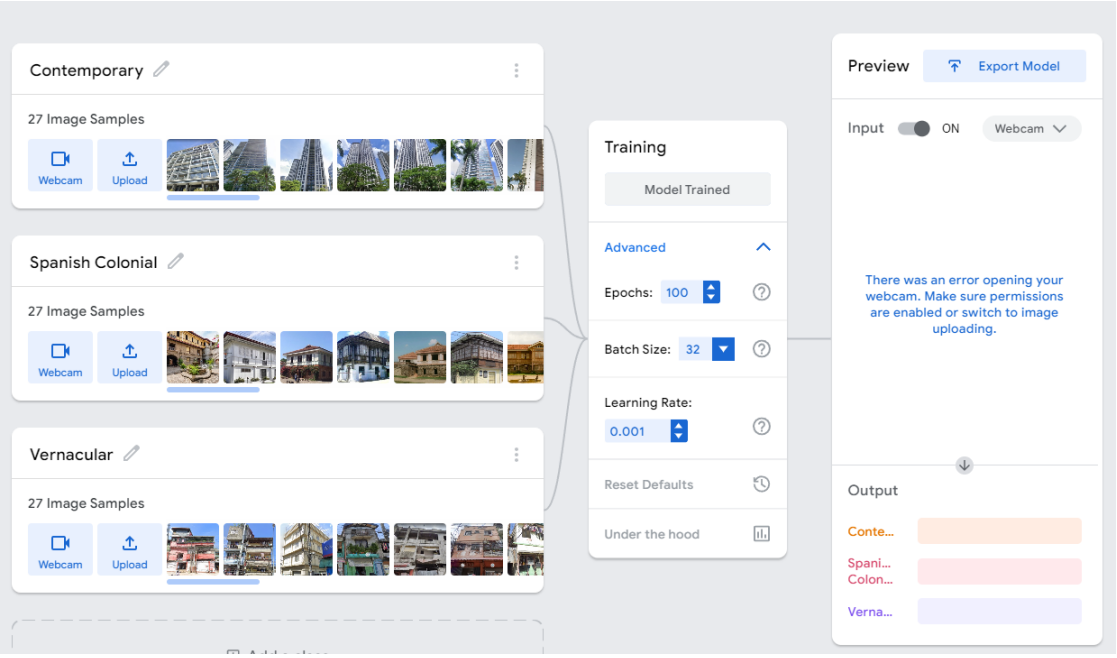

Figure 24: Training with Teachable Machine

To facilitate the training process, Teachable Machine, a user-friendly machine learning tool, was employed. The dataset was trained using Teachable Machine, which provides an intuitive interface for model training and evaluation as shown in 24.

D..5 Performance Evaluation

The performance of the machine learning model was assessed using various evaluation metrics. The evaluation results for the baseline dataset and data variation 1 are presented in Tables 5 and 6, respectively.

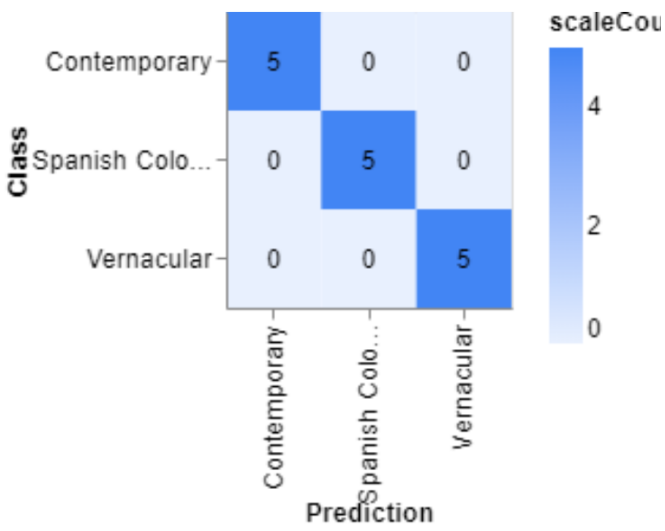

Figure 25: Confusion Matrix for Baseline Dataset

Baseline Dataset				
	Accuracy	**Precision**	**Recall**	**F1-Score**
Contemporary	1.00	1.00	1.00	1.00
Spanish Colonial	1.00	1.00	1.00	1.00
Vernacular	1.00	1.00	1.00	1.00
Micro avg	-	1.00	1.00	1.00
Macro avg	-	1.00	1.00	1.00
Weighted avg	-	1.00	1.00	1.00
Samples avg	-	1.00	1.00	1.00

Table 5: Evaluation Results for Baseline Dataset

In the baseline dataset, the model achieved perfect accuracy, precision, recall, and F1-score for all classes, including Contemporary, Spanish Colonial, and Vernacular styles.

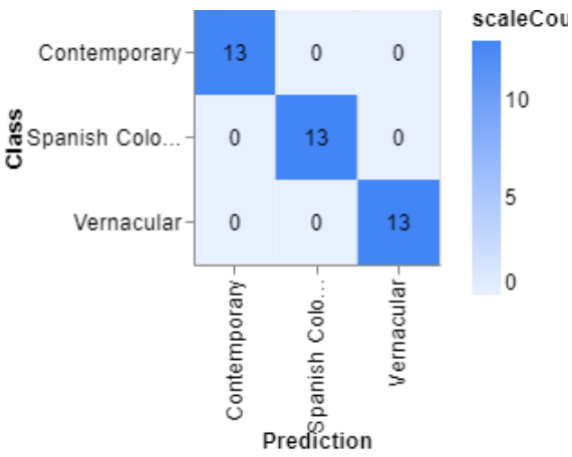

Figure 26: Confusion Matrix for Data Variation 1

Data Variation 1				
	Accuracy	**Precision**	**Recall**	**F1-Score**
Contemporary	1.00	1.00	1.00	1.00
Spanish Colonial	1.00	1.00	1.00	1.00
Vernacular	1.00	1.00	1.00	1.00
Micro avg	-	1.00	1.00	1.00
Macro avg	-	1.00	1.00	1.00
Weighted avg	-	1.00	1.00	1.00
Samples avg	-	1.00	1.00	1.00

Table 6: Evaluation Results for Data Variation 1

The evaluation results for data variation 1 show that the model achieved perfect accuracy, precision, recall, and F1-score for all classes, including Contemporary, Spanish Colonial, and Vernacular styles.

D..6 Selecting the Best Performing Model

The performance evaluation of the machine learning model involved comparing two dataset variations using various evaluation metrics. Table 7 summarizes the evaluation results for the different dataset variations.

	Accuracy	Precision	Recall	F1-Score	ROC AUC	MCC
Baseline	1.00	1.00	1.00	1.00	1.00	1.00
Variation 1	1.00	1.00	1.00	1.00	1.00	1.00

Table 7: Evaluation Results of Different Dataset Variations

Among the evaluated dataset variations, both the Baseline and Variation 1 demonstrated perfect accuracy, precision, recall, F1-Score, ROC AUC, and MCC values, indicating their strong performance in classifying architectural styles. However, Variation 1 exhibited slight improvements in terms of other evaluation metrics.

In addition to its exceptional performance, Variation 1 had the advantage of a larger sample size and data augmentation. The augmented dataset in Variation 1 allowed for a more comprehensive representation of architectural style features, potentially capturing a wider range of variations and patterns. This increased sample size and diversity likely contributed to its superior performance and generalization ability.

Based on these findings, Variation 1 emerges as the best performing model among the evaluated dataset variations, displaying exceptional accuracy, precision, recall, F1-Score, ROC AUC, and MCC values. The combination of perfect performance and the benefits of a larger sample size and data augmentation make Variation 1 the preferred choice for accurately classifying architectural styles.

D..7 Deploying the Best Performing Model

The best performing model, trained under dataset variation 1, was exported using Teachable Machine. The exported model can be integrated into various applications and workflows for architectural style classification tasks.

E. Plugin Development

E..1 Overview

This section provides a concise overview of the plugin's capabilities and functionalities. The plugin serves as a powerful tool for sustainable urban development and planning, offering a range of key features and benefits.

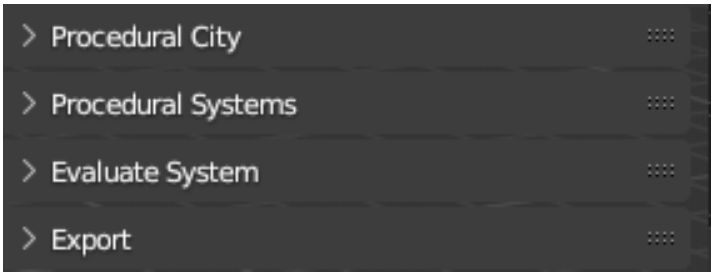

Figure 27: Plugin Overview

The plugin comprises four main panels: Procedural City, Procedural Systems, Evaluate Systems, and Export. These panels provide intuitive interfaces for creating, manipulating, and evaluating procedural systems within urban scenes. They enable users to generate variations of cityscapes, control parameters, measure architectural styles, inspect mesh properties, and export the generated content for use in other 3D applications.

E..2 Procedural Systems

1. The Procedural Systems Panel

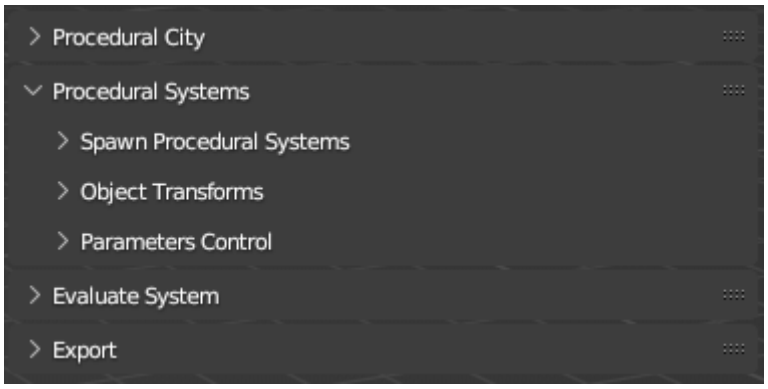

Figure 28: Procedural Systems Panel

The Procedural Systems panel offers a comprehensive suite of functionalities dedicated to the creation and manipulation of a wide array of procedural systems. Within this panel, users have the capability to generate and customize diverse urban elements, such as residential buildings, road networks, and trees. It comprises three distinct subpanels, each serving a unique set of functionalities: 'Spawn Procedural Systems,' 'Object Transforms,' and 'Parameters Control.' These subpanels provide users with granular control over the spawning of procedural systems, manipulation of object transforms, and fine-tuning of parameters to achieve desired outcomes.

2. Spawning Procedural Systems

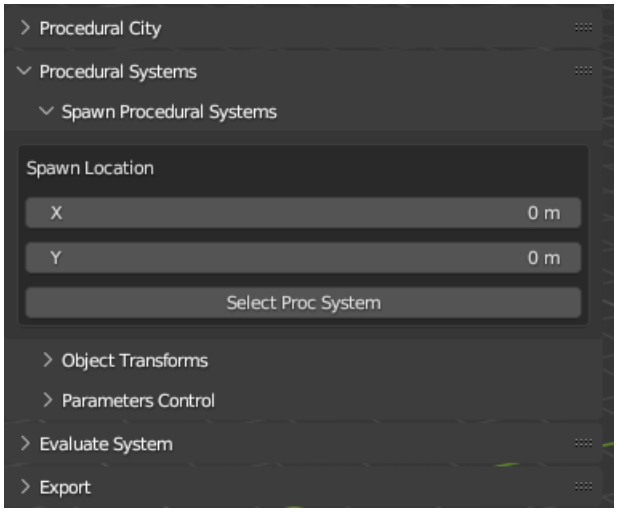

Figure 29: Spawn Procedural Systems Panel

The 'Spawn Procedural Systems' panel facilitates the configuration of the spawn location and the selection of the desired procedural system to be generated.

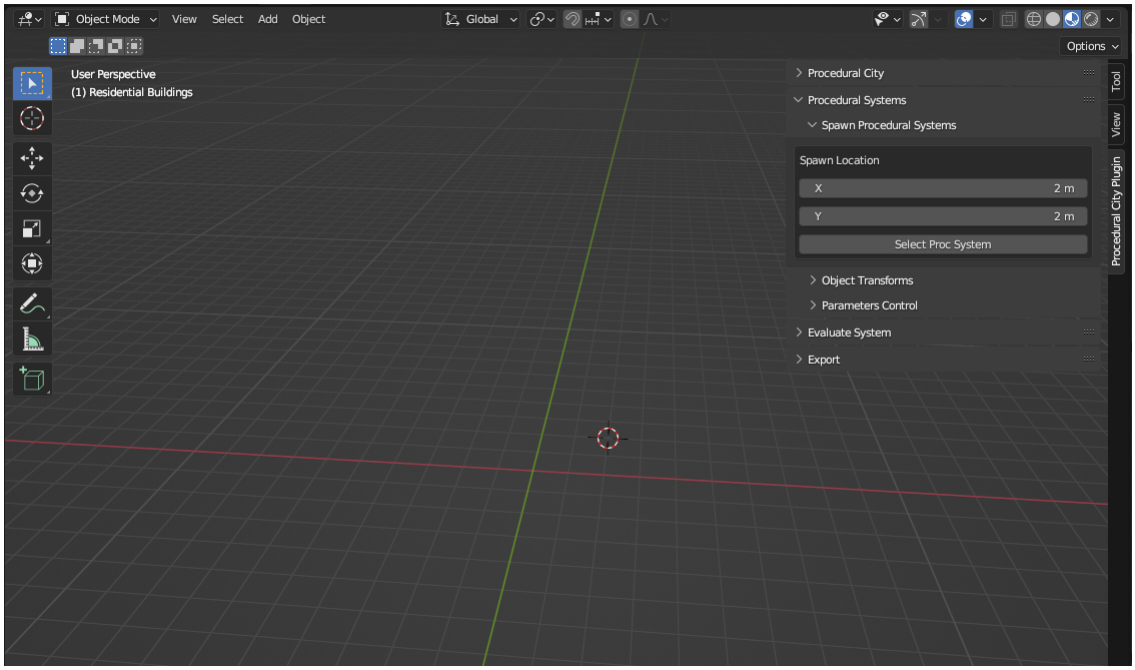

Figure 30: Editing Spawn Location

Users have the ability to adjust the spawn location along the x and y axes using a slider within the interface. Alternatively, they can manually input specific values for precise control over the spawn location. This location is visually represented within the scene by a circle referred to as the '3D cursor'.

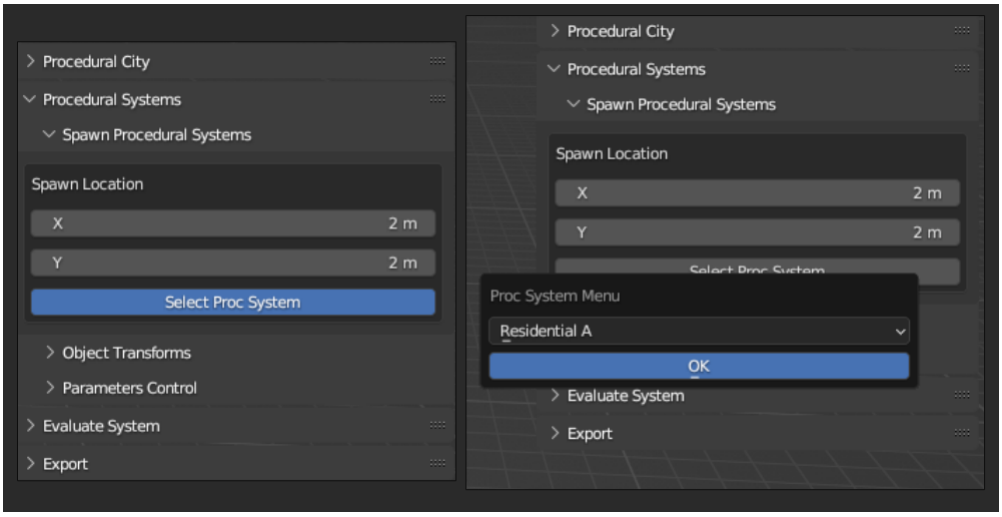

Figure 31: Modal Interface for Procedural System Selection

The button labeled 'Select Proc System' grants users access to a modal

interface featuring a dropdown menu.

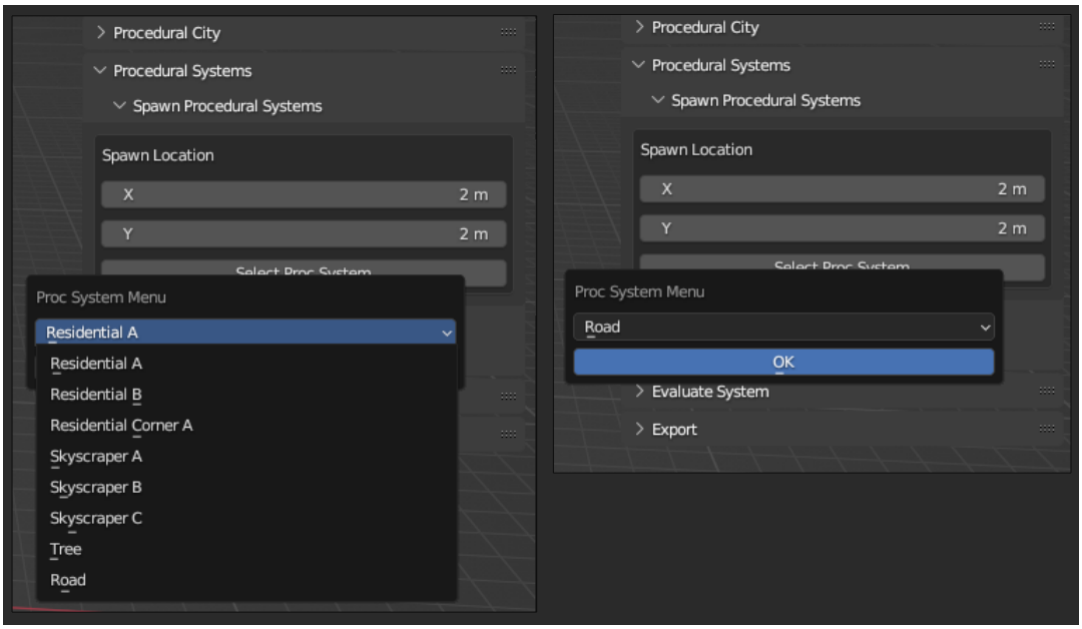

Figure 32: Dropdown Menu for Selecting Procedural System

The dropdown menu provides users with the option to create a diverse range of urban elements, including various types of residential buildings, a road network, and a tree.

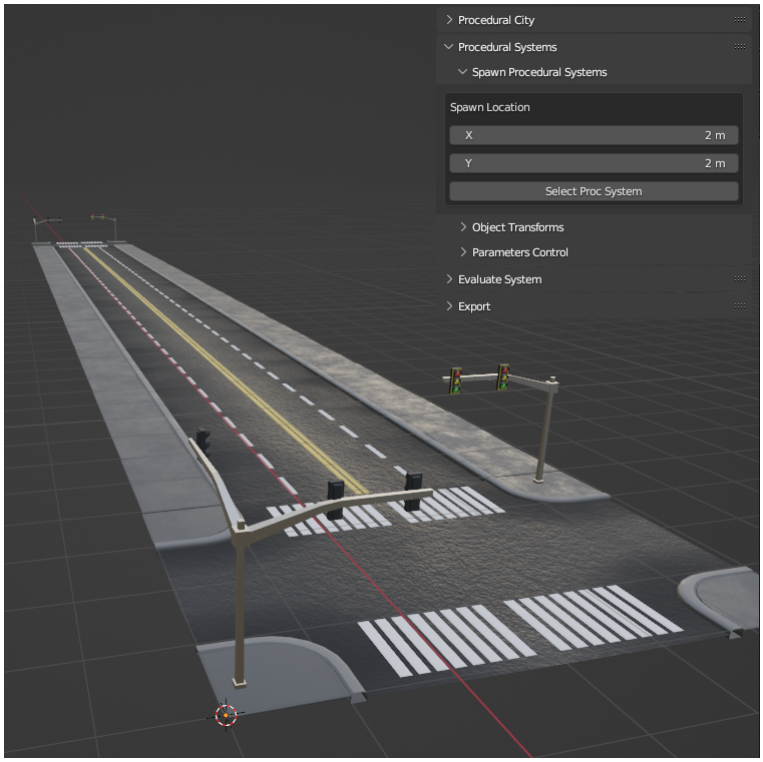

Figure 33: Creation of Selected Procedural System

After selecting the desired urban element from the dropdown menu, users can proceed by clicking the 'OK' button. This action triggers the creation of the selected urban element, precisely positioned relative to the 3D cursor.

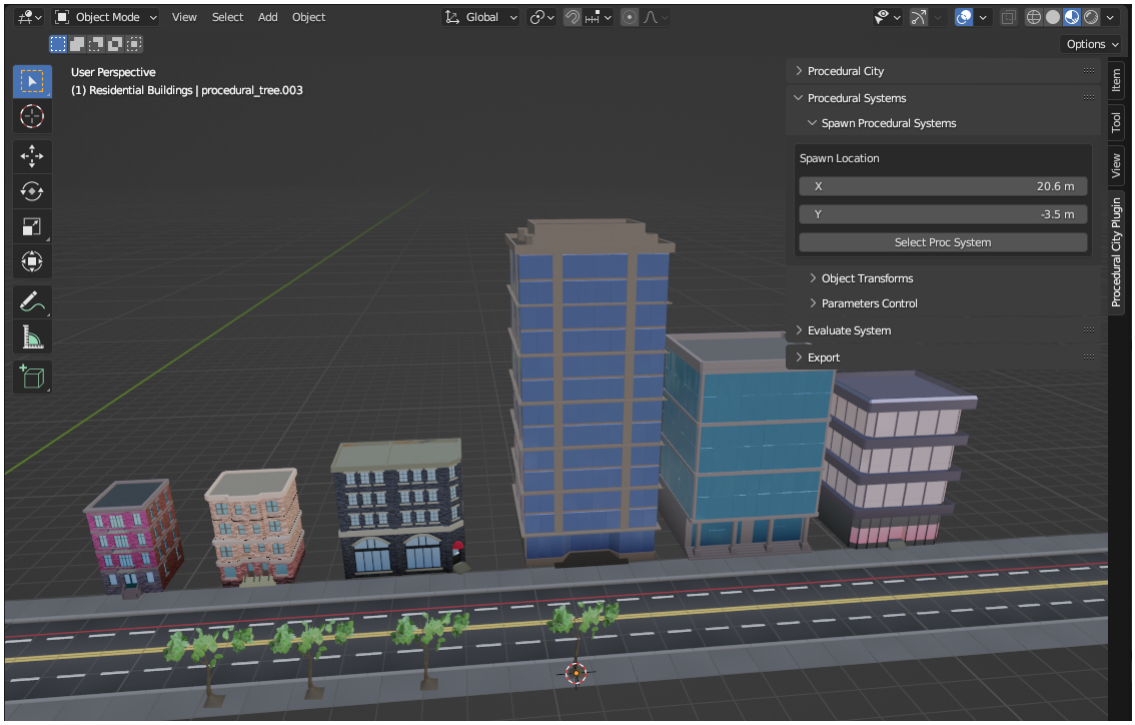

Figure 34: Creation of Multiple Procedural System

Users have the flexibility to repeat the entire process multiple times, allowing them to progressively build a comprehensive cityscape composed of various urban elements, including residential buildings, road networks, and trees. This iterative approach empowers users to gradually shape and expand the city according to their desired specifications.

3. Manipulating Object Transforms

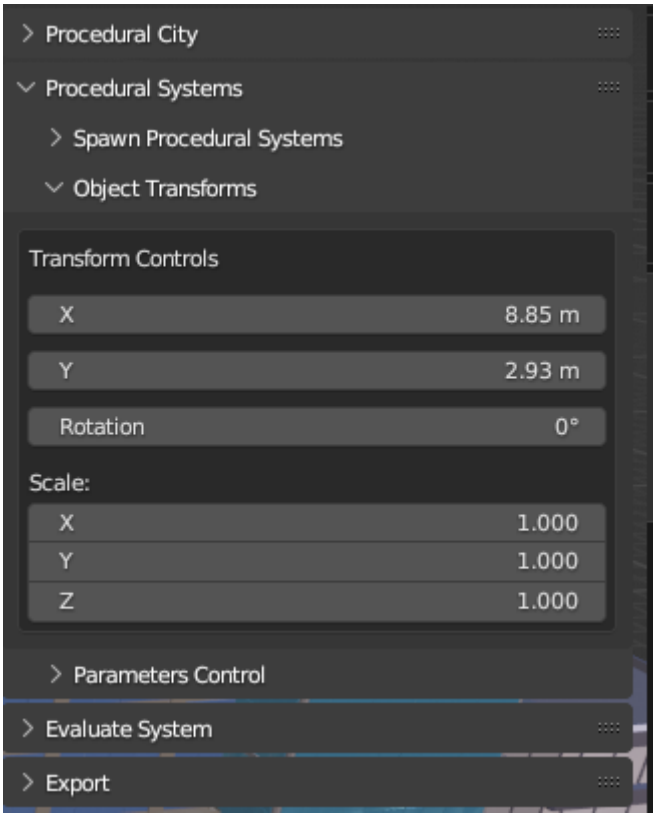

Figure 35: Object Transform Panel

The 'Object Transform' panel serves as a centralized control hub for making precise adjustments to the positioning, orientation, and size of the procedural elements within the scene.

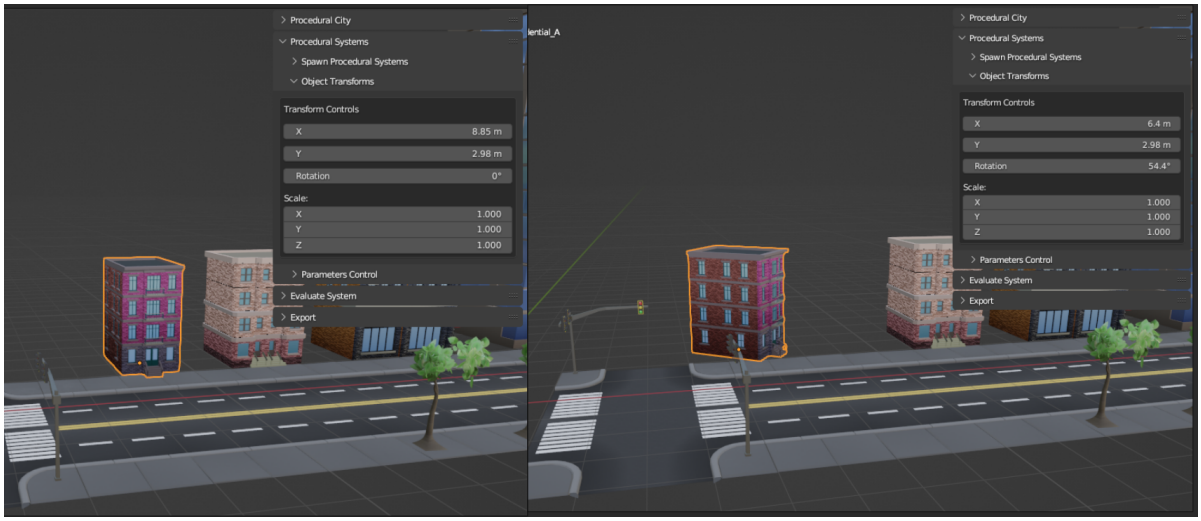

Figure 36: Modification of Object Transforms

Through the modification of control parameters within this panel, users can dynamically and interactively adjust the transform properties of the objects in real-time.

4. Modifying Parameters Control

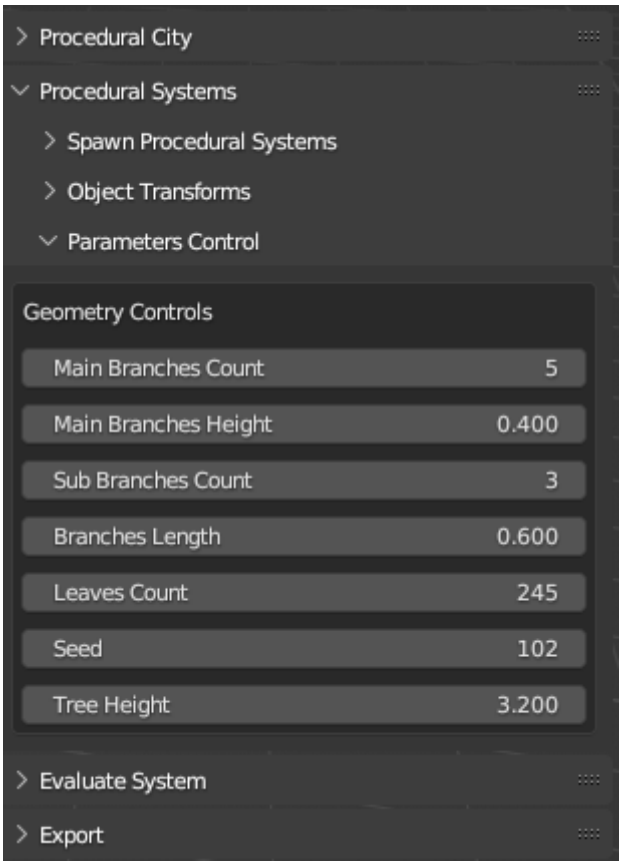

Figure 37: Parameters Control

The 'Parameters Control' panel provides users with direct access to the control parameters associated with the selected procedural system.

Figure 38: Interface of Control Parameters for Trees

This panel serves as an interface for adjusting and fine-tuning the specific parameters that govern the generation and appearance of a particular procedural element.

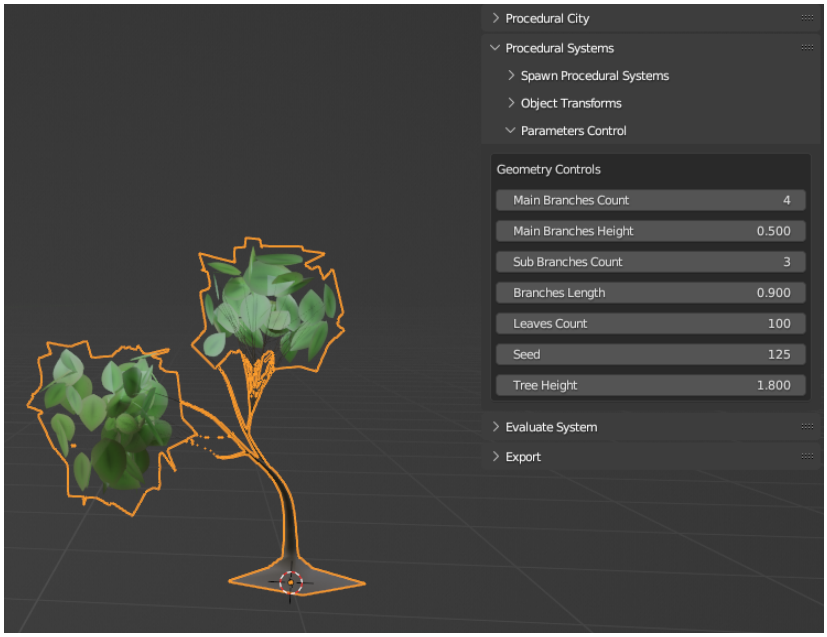

Figure 39: Adjusting Control Parameters

Users can seamlessly interact with the control parameters, observing real-time changes to the selected urban element. This immediate visual feedback facilitates iterative adjustments and customization of the procedural system.

Figure 40: Adjusting Control Parameters of Multiple Objects

Users can repeat the process with different procedural systems, each offering unique control parameters for modifying the selected system. This enables the generation of instant variations, allowing users to explore and create diverse outcomes.

E..3 Evaluate

1. The Evaluate System Panel

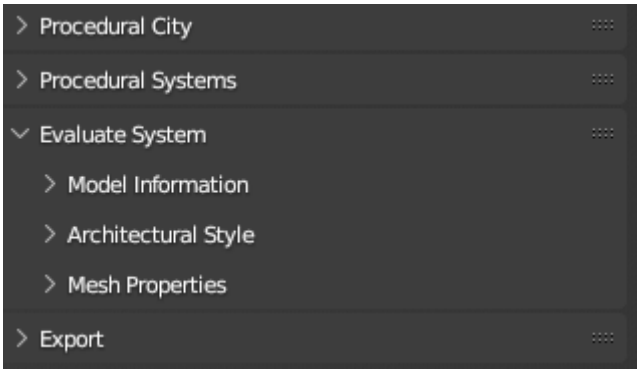

Figure 41: Evaluate System Panel

The 'Evaluate System' panel offers a range of tools for measuring architectural styles and inspecting mesh properties. It consists of three subpanels: 'Model Information', 'Architectural Style', and 'Mesh Properties'. These tools enable comprehensive analysis and evaluation of the procedural systems' architectural characteristics and mesh properties.

2. Examining Model Information

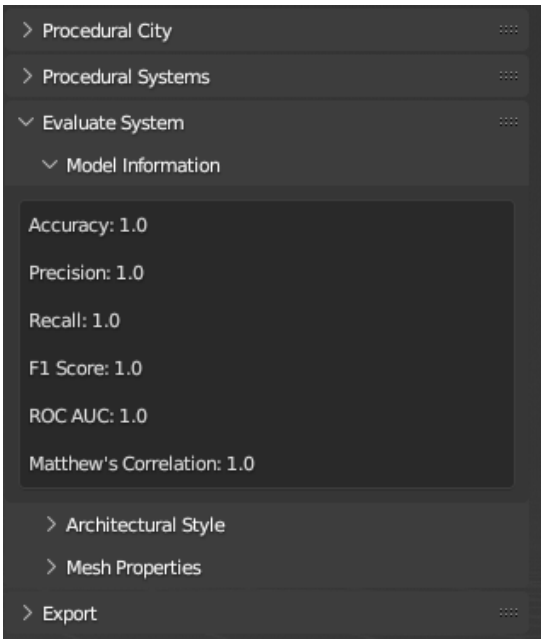

Figure 42: Model Information Panel

The 'Model Information' panel provides performance metrics for the trained machine learning model used to classify architectural styles in the generated

content. It displays key information, including accuracy, precision, recall, and F1 score, which offer insights into the model's effectiveness in accurately categorizing architectural styles.

3. Analyzing Architectural Styles

Figure 43: Architectural Style Panel

The 'Architectural Style' panel performs classification of the generated content on the user's screen, determining whether it belongs to the categories of Contemporary, Spanish Colonial, or Vernacular style. This panel enables users to gain valuable insights into the architectural styles represented in the procedural systems.

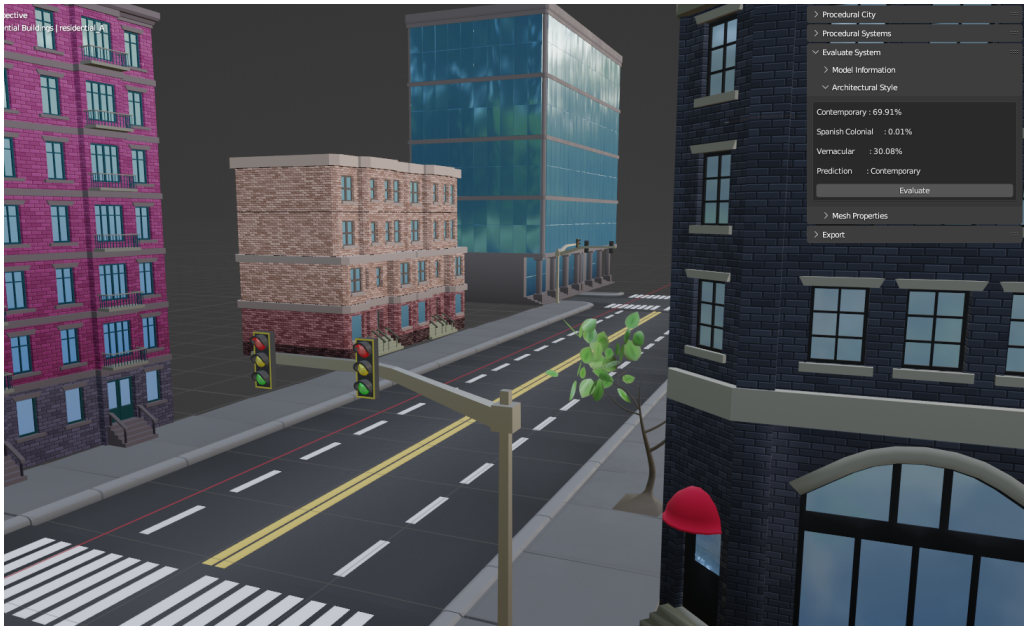

Figure 44: Evaluating Architectural Styles

By clicking the 'Evaluate' button, users can receive assistance in determining the percentage of architectural styles displayed on the camera view. This feature helps minimize architectural dissonance and promotes stylistic coherence within a particular region, supporting the creation of harmonious urban environments.

4. Inspecting Mesh Properties

Figure 45: Mesh Properties Panel

The 'Mesh Properties' panel provides essential information about the generated content, including polygon count, vertices count, edges count, and storage space. It consists of two main sections: 'Global Mesh Properties' and 'Local Mesh Properties'.

Figure 46: Global and Local Mesh Properties

The 'Global Mesh Properties' section monitors the entire scene, while the 'Local Mesh Properties' section focuses on the selected object, which is highlighted in orange. This panel offers valuable insights into the geometric characteristics and storage requirements of the procedural systems.

E..4 Procedural City

1. The Procedural City Panel

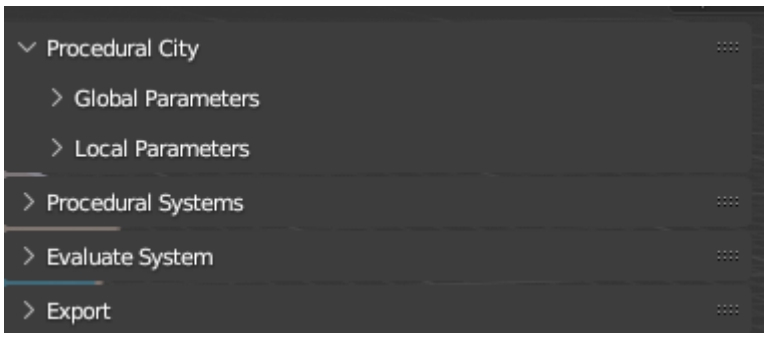

Figure 47: Procedural City Panel

The 'Procedural City' panel serves as the control center for generating variations of the cityscape. It enables users to influence all or specific types of procedural systems within the scene. This panel consists of two subpanels: 'Global Parameters' and 'Local Parameters'. The 'Global Parameters' subpanel allows users to make overarching modifications that affect the entire cityscape, while the 'Local Parameters' subpanel offers the flexibility to make targeted adjustments to individual procedural systems. Together, these subpanels provide users with comprehensive control over the generation of diverse urban environments.

2. Modifying Global Parameters

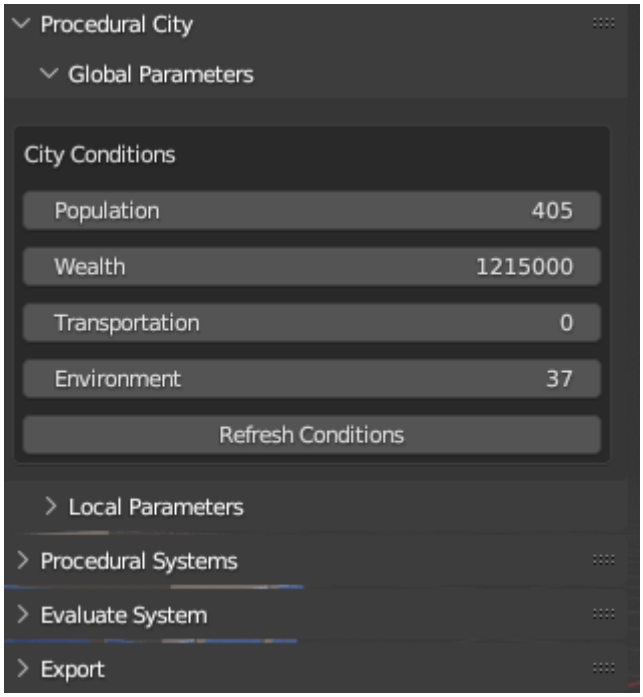

Figure 48: Global Parameters Panel

The 'Global Parameters' subpanel encompasses a range of influential parameters such as population, wealth, transportation, and environment, allowing users to define the overall characteristics and attributes of the generated cityscape. By adjusting these global parameters, users have the ability to shape the demographic, economic, transportation, and environmental as-

pects of the virtual city, resulting in the creation of unique and dynamic urban environments.

To reflect the changes made to the city conditions, users can simply click the 'Refresh Conditions' button.

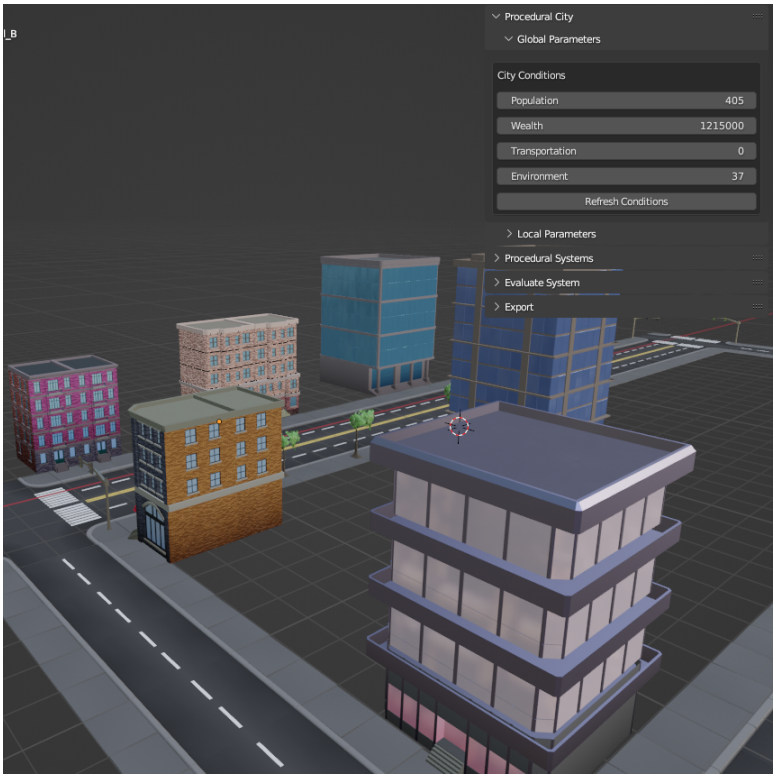

Figure 49: The Population Parameter

The 'Population' parameter plays a significant role in determining the height of the residential buildings within the generated cityscape. By manipulating this parameter, users can effectively control the vertical scale of the buildings.

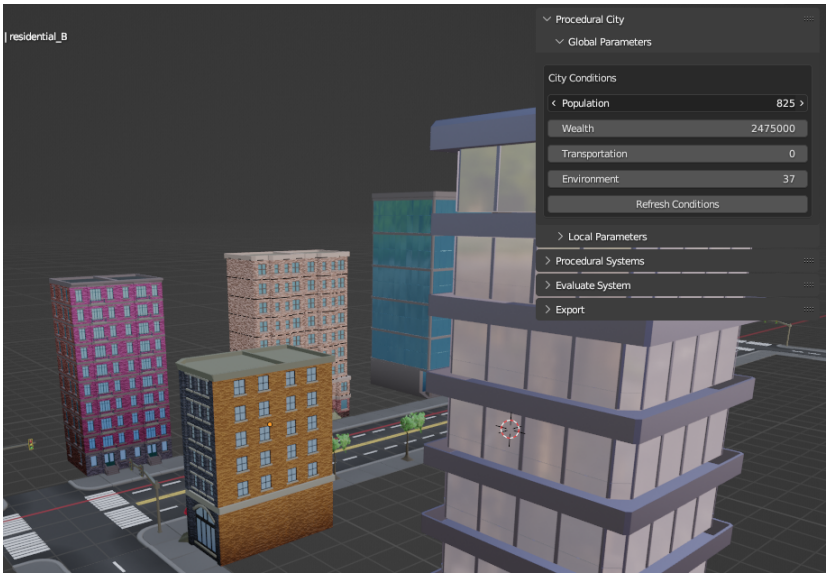

Figure 50: Adjusting Population Parameter

Increasing the population parameter will result in taller residential structures, while decreasing it will lead to shorter buildings. This adjustment applies to a random selection of buildings, allowing for realistic variations in height across the urban environment.

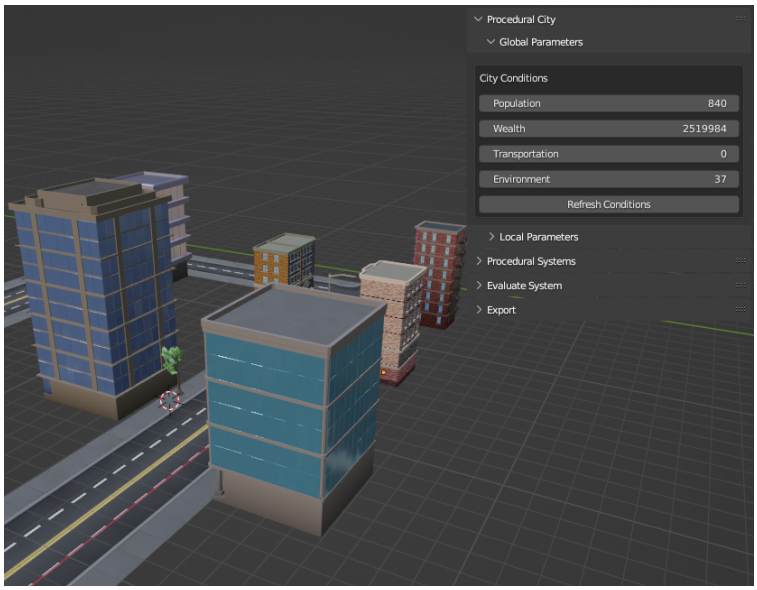

Figure 51: The Wealth Parameter

The 'Wealth' parameter influences both the vertical and horizontal aspects of the residential buildings within the procedural cityscape. By modifying this

parameter, users can manipulate the dimensions of the buildings in terms of height and width.

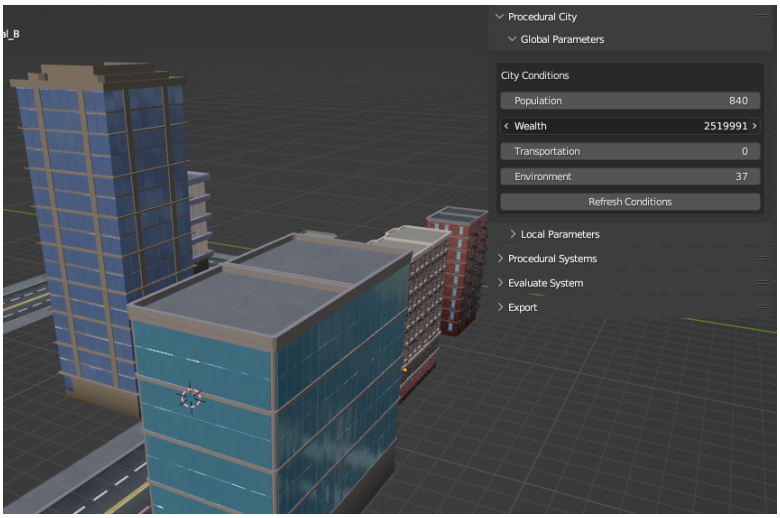

Figure 52: Adjusting Wealth Parameter

Increasing the wealth parameter will result in taller and wider residential structures, while decreasing it will lead to shorter and narrower buildings. This parameter allows for the representation of different socio-economic conditions within the virtual city, creating diverse cityscapes based on the wealth of the simulated population.

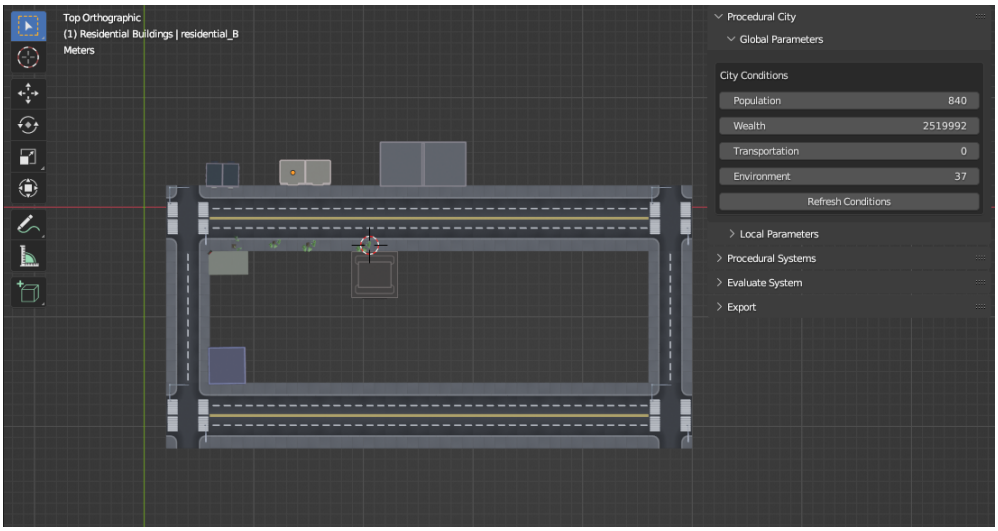

Figure 53: The Transporation Parameter

The 'Transportation' parameter plays a pivotal role in shaping the road

networks within the procedural cityscape. By adjusting this parameter, users have control over the density and arrangement of roads in the virtual environment.

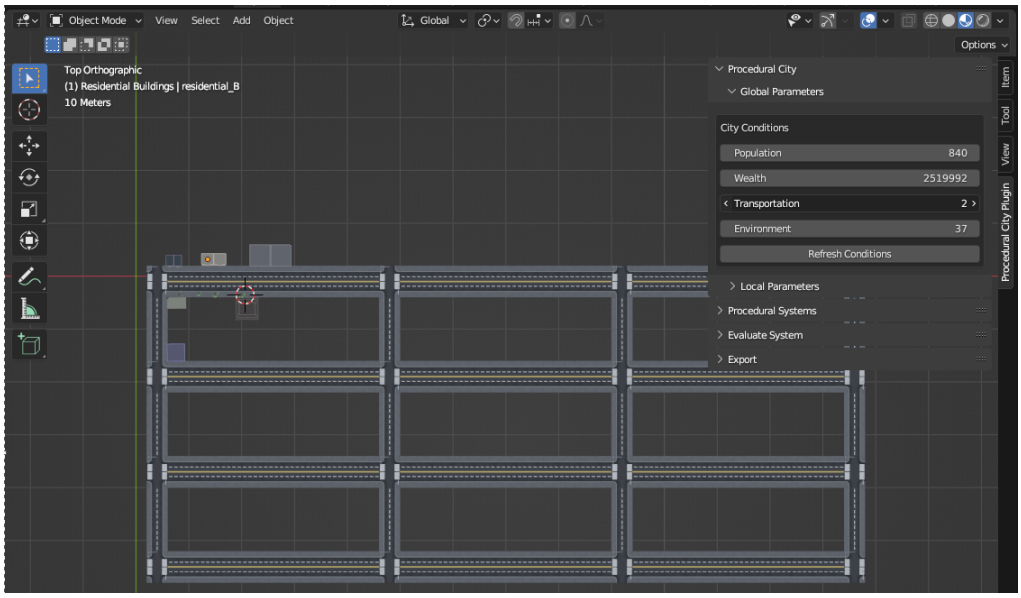

Figure 54: Adjusting Transportation Parameter

Increasing the transportation parameter can result in a higher number of roads and a closer proximity between them, creating a dense and well-connected transportation network. Conversely, decreasing the parameter will lead to fewer roads with larger distances between them, reflecting a more spacious and less congested road system. This parameter enables users to simulate varying degrees of urban development and transportation infrastructure within the generated cityscape.

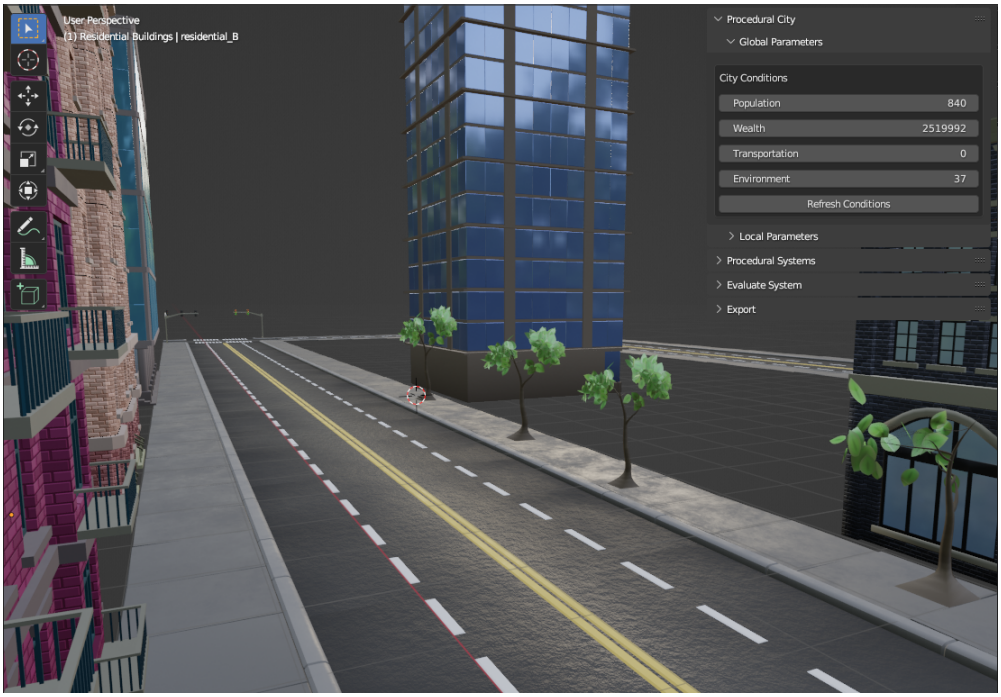

Figure 55: The Environment Parameter

The 'Environment' parameter influences tree properties such as height, number of branches, and leaves count.

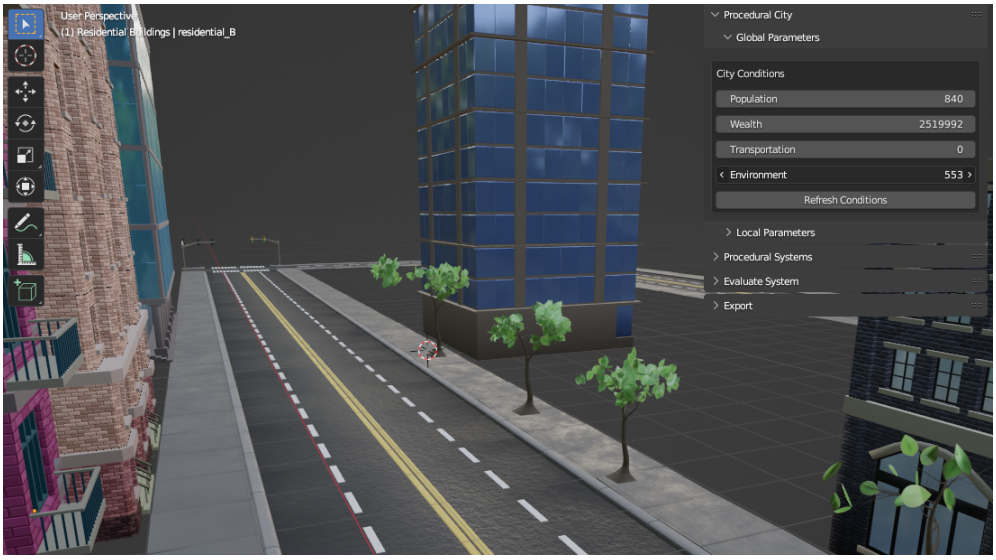

Figure 56: Adjusting Environment Parameter

By adjusting the environment parameter, users can control the characteristics of trees within the procedural cityscape, determining their height, branch complexity, and foliage abundance.

3. Modifying Local Parameters

Figure 57: Local Parameters Panel

The 'Local Parameters' panel allows users to fine-tune the characteristics of specific procedural systems within the scene. It is divided into three main sections: buildings, roads, and trees. Each section focuses on controlling the attributes and properties of the corresponding procedural system type.

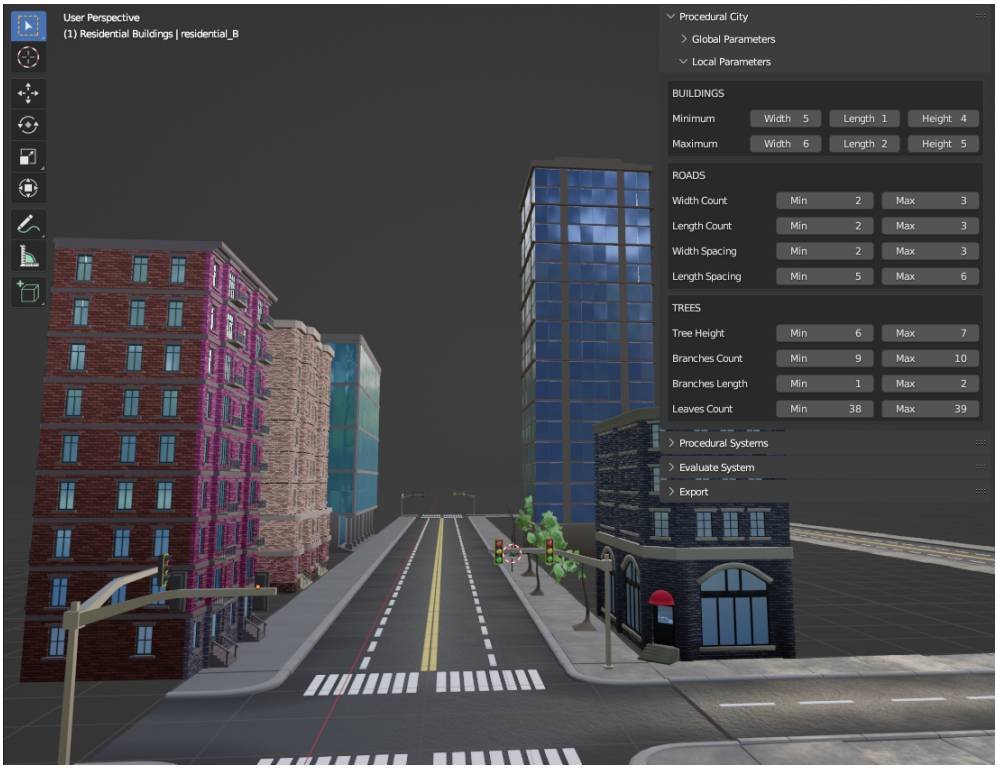

Figure 58: The Buildings Section

Within the buildings section, users can adjust the minimum and maximum width, length, and height of residential buildings.

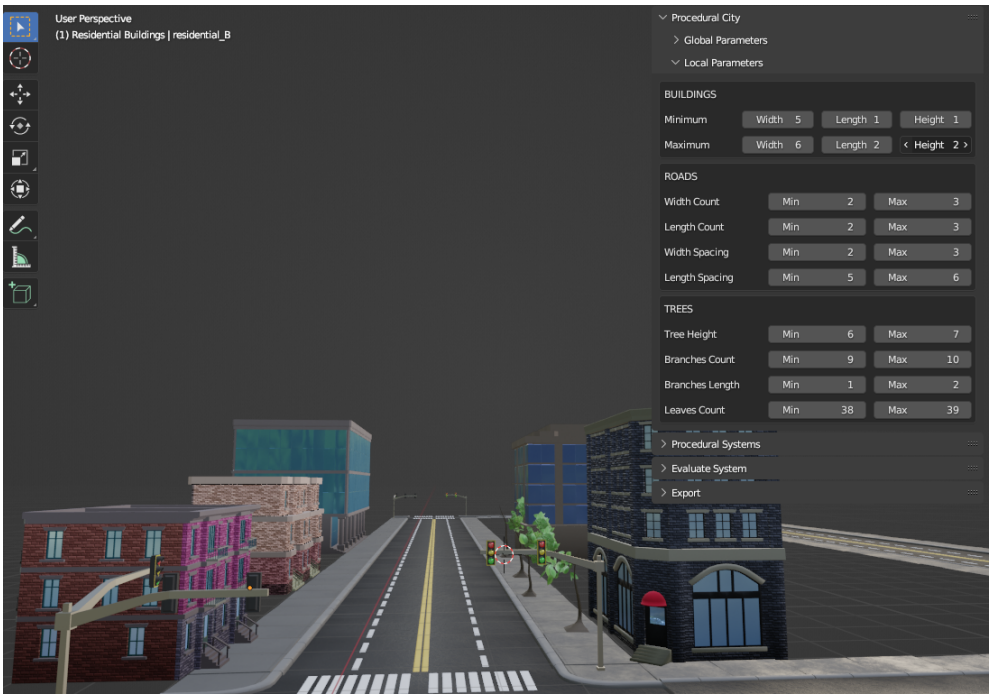

Figure 59: Adjusting Parameters in Buildings Section

Any adjustments made to the minimum and maximum width, length, and height will be applied uniformly to all generated residential buildings.

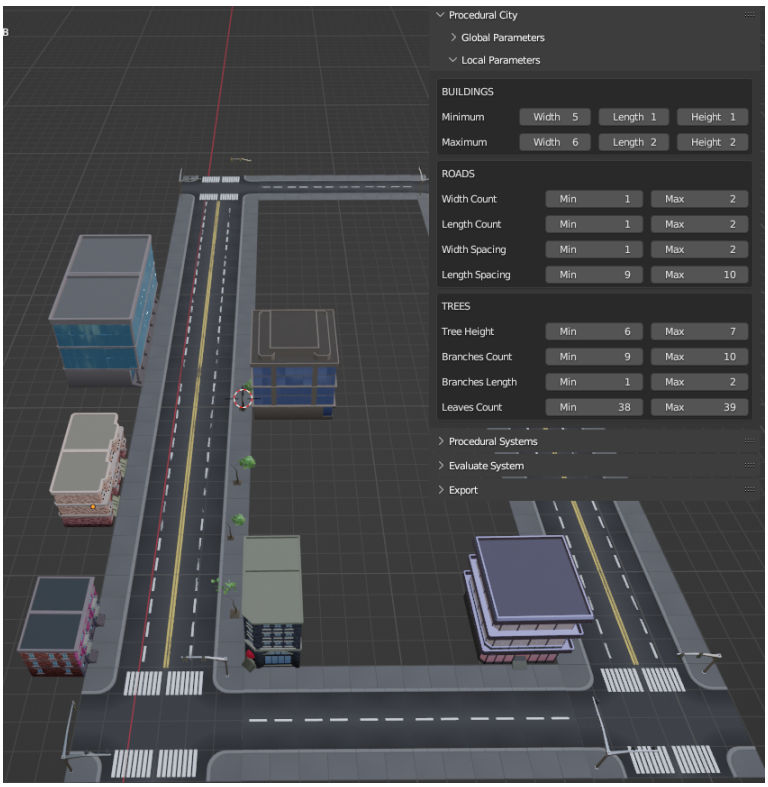

Figure 60: The Roads Section

The 'Roads' section provides users with control over the properties of all roads present in the scene. It includes parameters that determine the minimum and maximum values for attributes such as road length, road width count, and spacings between the roads

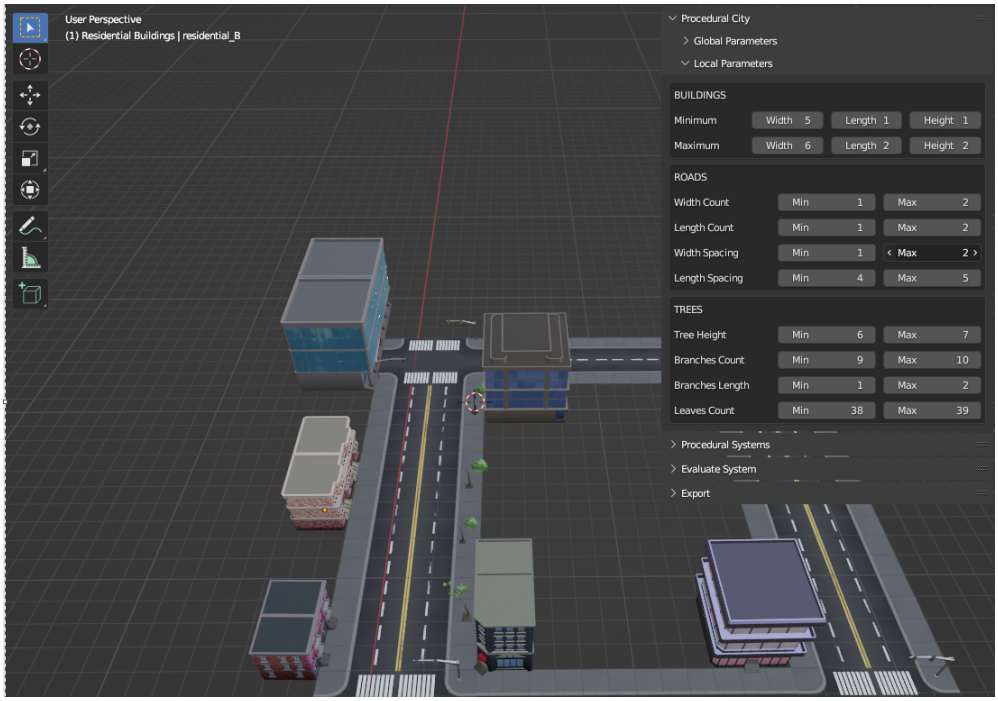

Figure 61: Adjusting Parameters in Roads Section

By adjusting these parameters, users can precisely define the characteristics and dimensions of the road network in their procedural cityscape.

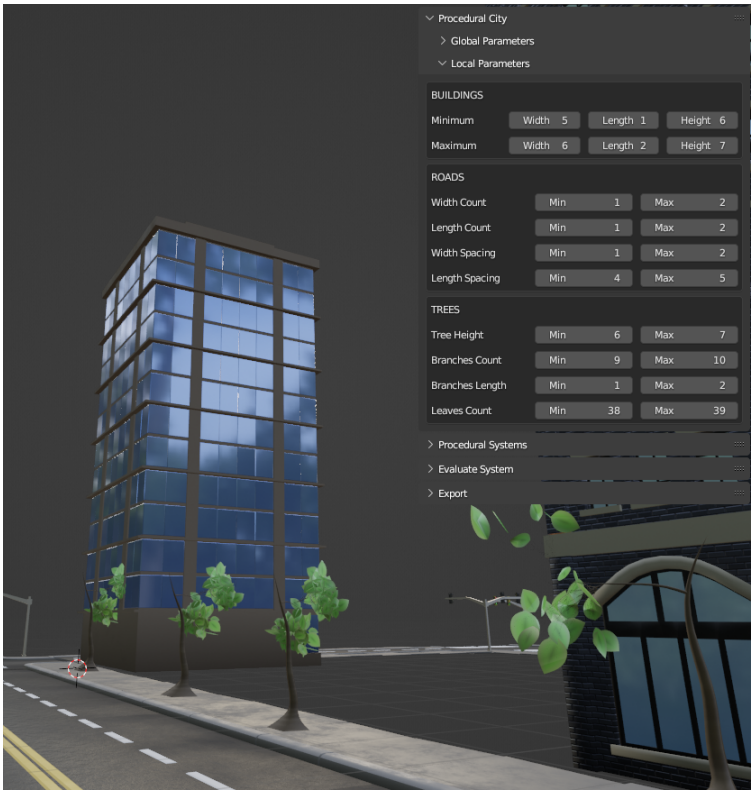

Figure 62: The Trees Section

The 'Trees' section enables users to manipulate the properties of all trees within the scene. It grants control over parameters such as branches count, leaves count, tree height, and branches height.

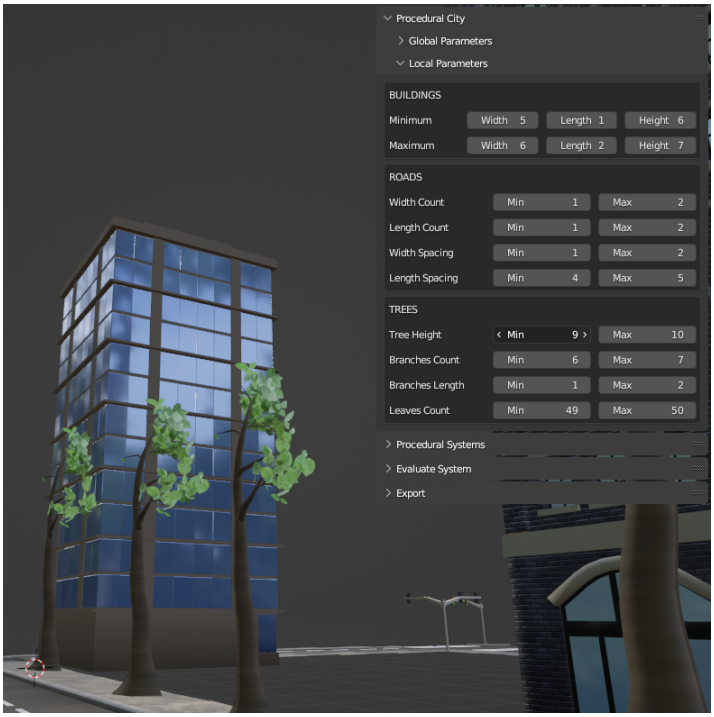

Figure 63: Adjusting Parameters in Trees Section

By adjusting these parameters, users can customize the appearance and characteristics of the trees in their procedural cityscape, determining factors such as tree density, foliage abundance, and overall tree height.

E..5 Export

1. The Export Panel

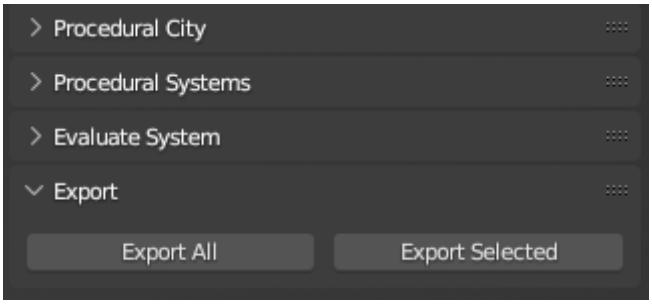

Figure 64: Export Panel

The 'Export' panel facilitates the transfer of procedural systems from Blender to other programs. This panel allows users to select and export either spe-

cific or all procedural systems in the scene. The exported procedural systems are saved in the FBX (Filmbox) format, ensuring compatibility with a wide range of software applications and platforms. By utilizing the Export panel, users can seamlessly integrate the generated procedural systems into their preferred workflows and utilize them in various contexts as needed.

2. Exporting Selected Object

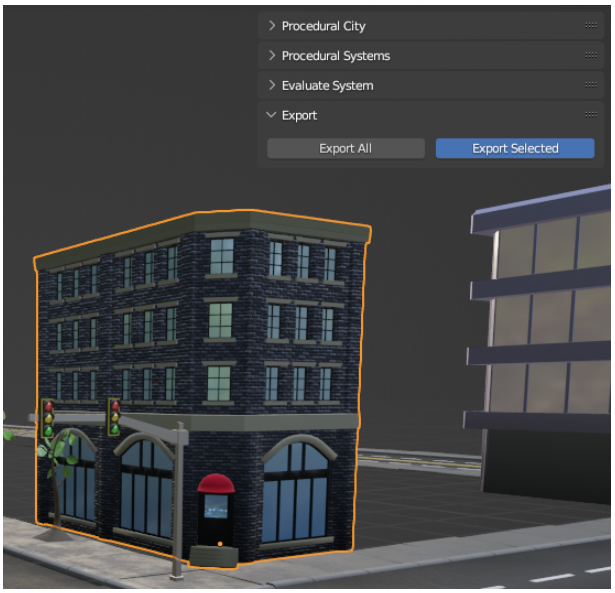

Figure 65: Selecting Procedural Systems for Export

Users have the flexibility to select the desired procedural systems within the scene. Once the desired systems are selected, users can proceed by initiating the export process through the "Export Selected" option.

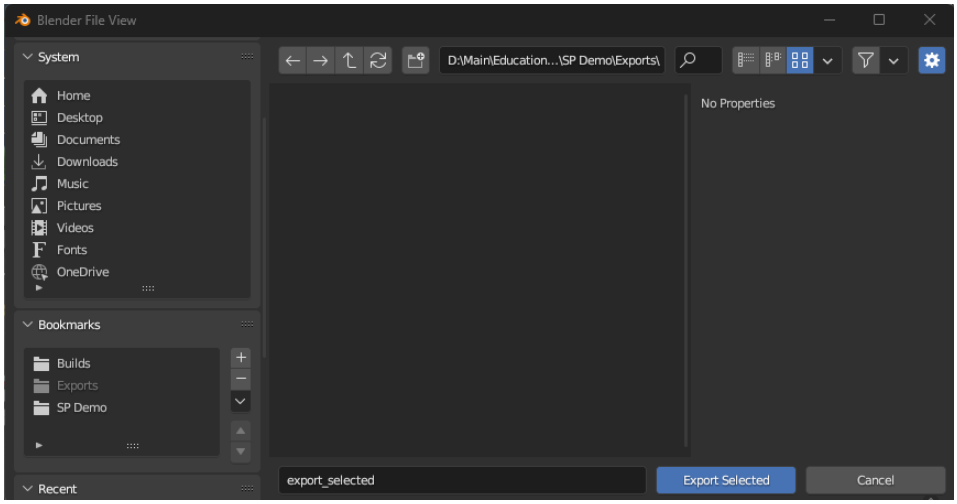

Figure 66: Export Selected Prompt

Upon clicking the "Export Selected" button, a prompt will appear, requesting the user to specify the directory where the exported file will be stored, along with the desired file name.

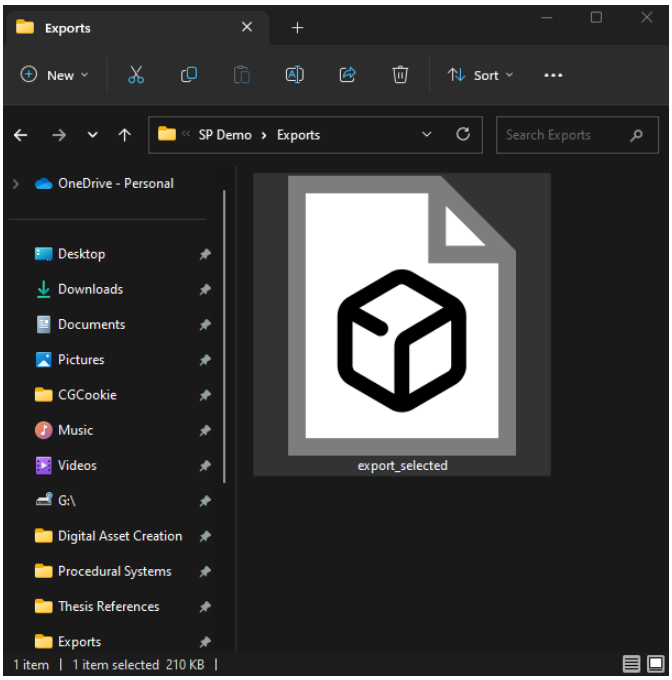

Figure 67: Generated File of Export Selected Process

Once completed, the exported file will be generated and saved in the directory previously specified by the user.

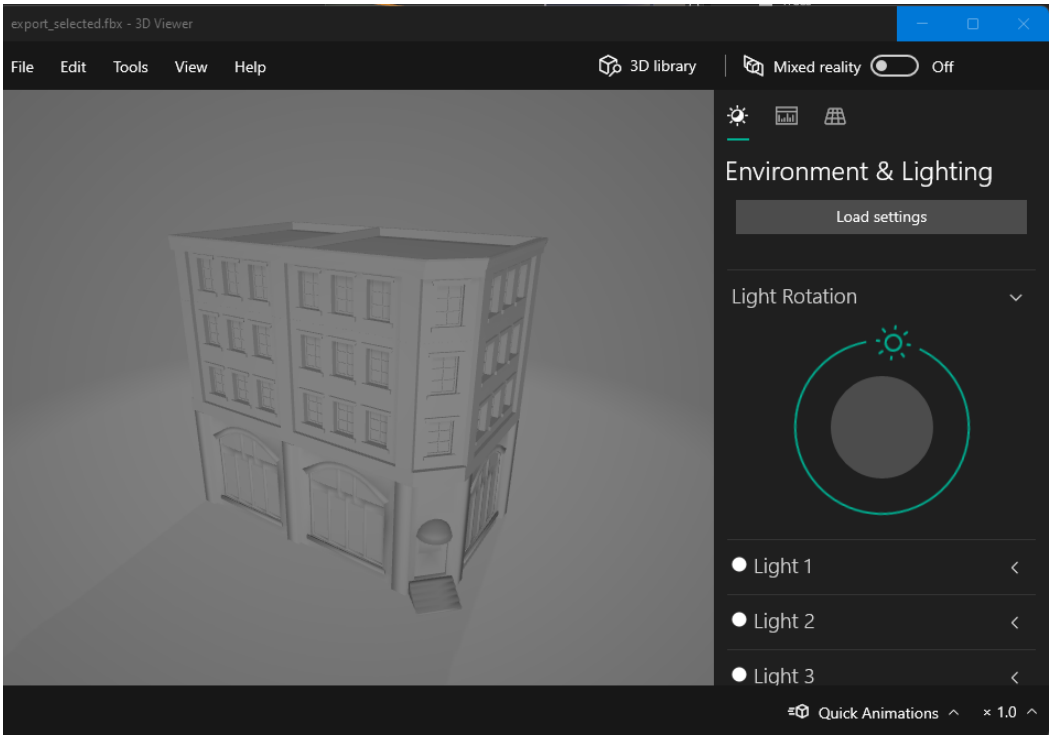

Figure 68: Export Selected View in Microsoft 3D Viewer

The exported file can now be opened in various 3D applications and viewers, including but not limited to Microsoft 3D Viewer. This allows users to utilize the exported procedural systems in their preferred software environments, expanding the possibilities for further editing, visualization, and integration into larger projects.

3. Exporting All Objects

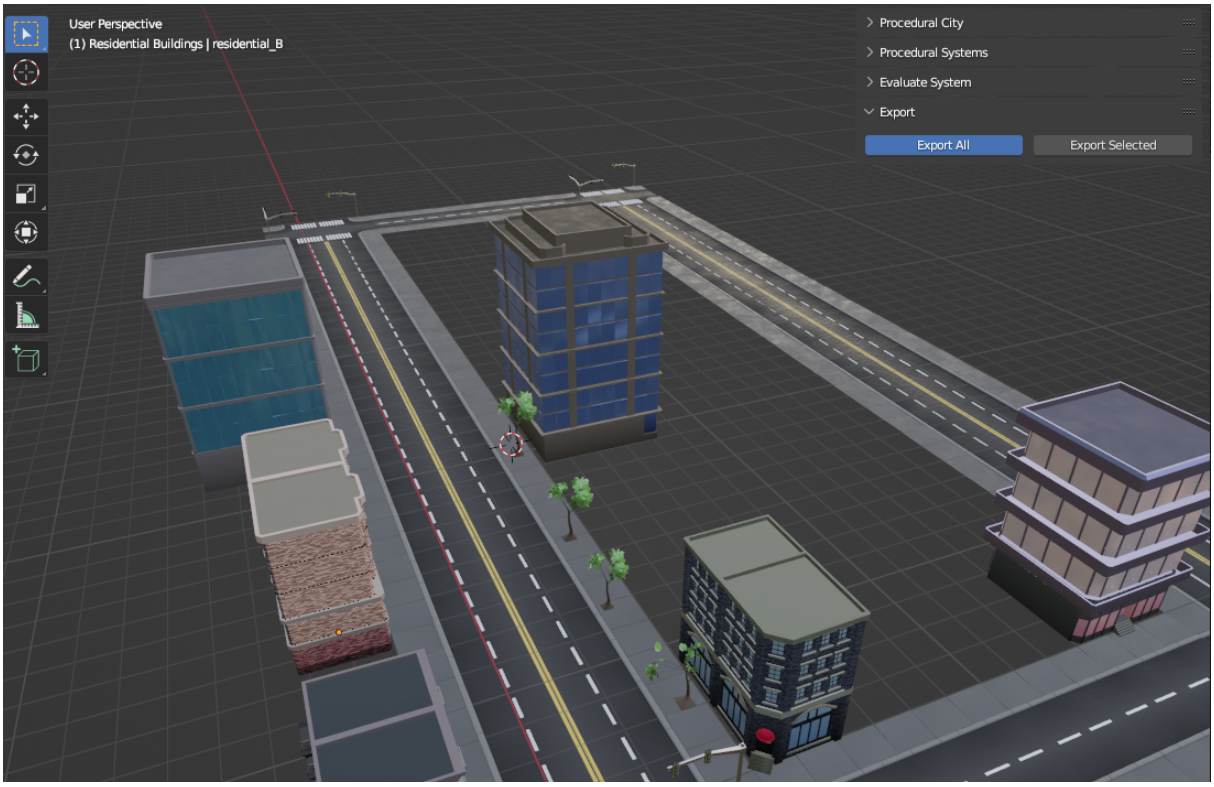

Figure 69: Export All Button

The "Export All" button streamlines the process of exporting all objects in the scene by grouping them into a single object. This simplifies the export workflow and ensures that all objects are exported as a cohesive unit. By clicking the "Export All" button, the user can initiate the export process for the grouped object, which will be saved in the FBX format.

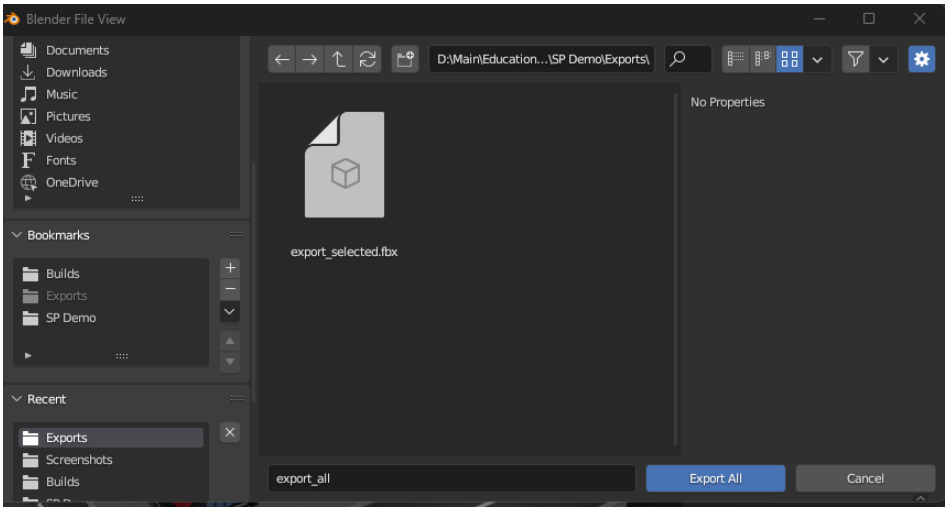

Figure 70: Export All Prompt

During the export process, the user will be prompted to specify the directory where they want the exported file to be saved, as well as provide a desired file name.

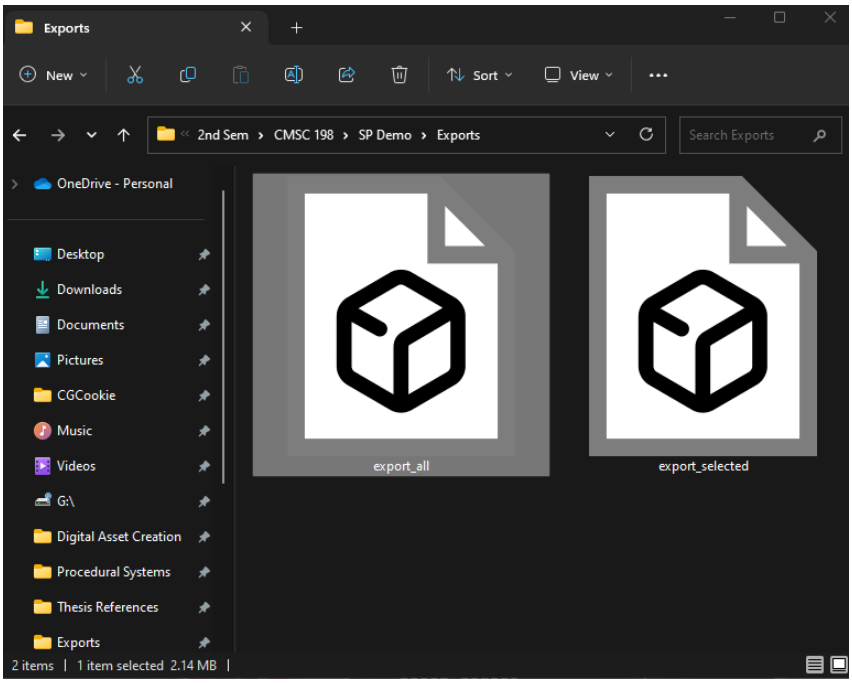

Figure 71: Generated File of Export All Process

After successfully completing the export process, a single FBX file will be generated. This file serves as the final output of the export operation and contains all the selected procedural systems or objects from the scene.

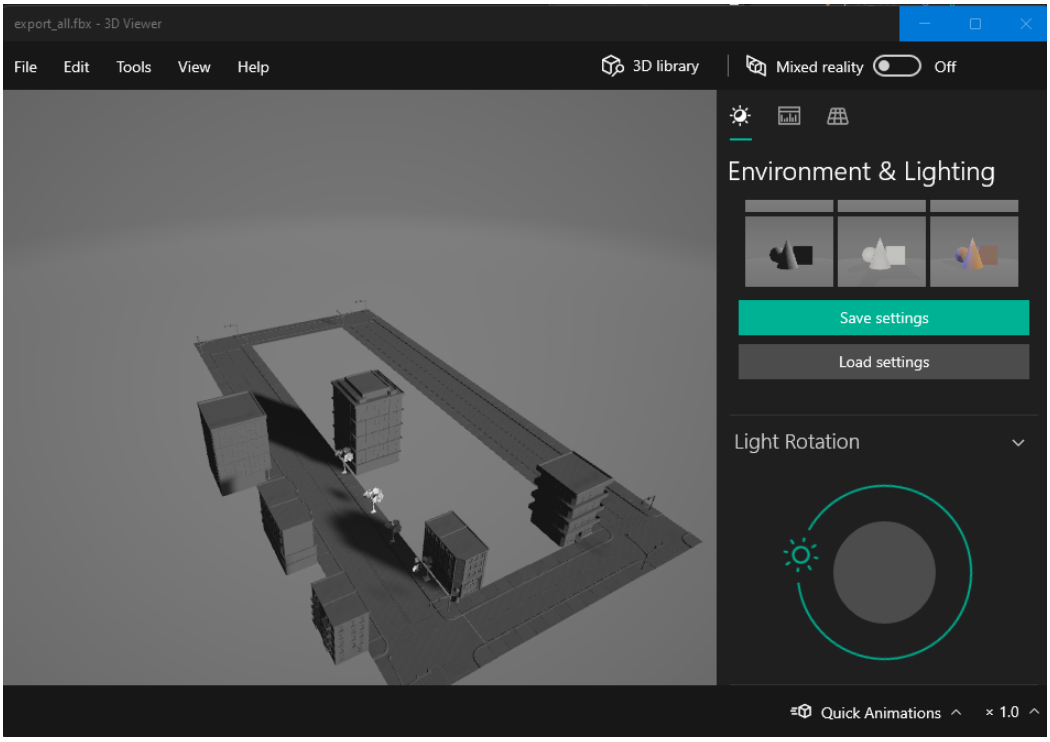

Figure 72: Export All View in Microsoft 3D Viewer

The exported file, saved in the FBX format, can be seamlessly opened and viewed as a single entity by other 3D applications.

VI. Discussions

This section presents a comprehensive discussion on the developed plugin for Blender and its alignment with the study's objectives. It also highlights the resolution of identified problems and the significance of the outcomes in relation to relevant references. Furthermore, the challenges encountered during the system's development and the strategies employed to overcome them are explored. Finally, the main contributions of this work are emphasized, underscoring its advancements in the field of urban modeling and planning.

The study aimed to create procedural systems for representing urban elements in Philippine urban cities, with the overarching goal of providing urban planners with a flexible toolset to manipulate and visualize these elements in a 3D environment. Additionally, to address architectural dissonance as part of sustainability efforts, a tool was created to measure the architectural styles present in the scene. Achieving this objective required the successful completion of several specific goals, which are discussed below.

The study began with a case study conducted on Taguig City, a highly urbanized city in Metro Manila, Philippines. The focus was on urban elements such as residential buildings, road networks, and trees, and a comprehensive collection of images representing these elements was successfully gathered. This allowed for a detailed analysis of their architectural characteristics. Furthermore, common architectural style rules and vocabulary specific to Philippine urban cities and each urban element were identified and selected, providing a solid foundation for the subsequent stages of the study.

Digital assets were then created to represent each urban element. Specifically, six digital assets were developed for residential buildings, three for road networks, and one for trees. These digital assets consist of 3D models enhanced with shaders to provide visual realism.

Procedural systems were developed for each urban element and the city as a whole. This involved translating the identified architectural style rules and vocab-

ulary into geometry nodes, enabling the automated generation of 3D geometry. Additionally, control parameters were created to govern the procedural generation process, allowing users to customize and fine-tune the output. The successful implementation of these systems greatly enhanced the application's dynamic and customizable nature, providing urban planners with powerful tools to generate urban elements that adhere to specific architectural styles.

The study involved training a machine learning model using gathered images, with data preprocessing and training conducted in Teachable Machine. Evaluation of the trained models utilized standard metrics (accuracy, precision, recall, F1 score) and specialized metrics (ROC AUC, correlation coefficient). All datasets performed exceptionally well across these metrics, and the augmented dataset, which incorporated various techniques and had a larger sample size, was chosen for its enhanced robustness and generalizability in classifying architectural styles.

To facilitate the evaluation and optimization of digital assets, a tool was created to measure their performance in terms of geometry and storage space. Specifically, scripts were developed to retrieve polygon count and display storage space, providing valuable insights for urban planners regarding asset complexity and resource requirements.

To integrate the developed application system into the existing workflows of urban planners, a plugin was developed for a 3D application. This plugin offered several key functionalities, including the ability to add procedural systems for urban elements to the scene, modify parameters of existing systems, measure architectural styles, and assess the performance of digital assets based on polygon count, vertex count, edge count, and estimated storage size. Furthermore, the plugin allowed for the seamless export of selected procedural systems or the entire scene for use in other 3D programs, enhancing the versatility and interoperability of the application.

Throughout the development of the application system, several challenges were encountered at different stages of the study. In the case study, a major challenge

was the limited availability of images, primarily restricted to public streets due to the constraints of Google Maps street view. Moreover, the images could only be captured from a specific angle, i.e., the front view, which significantly affected the model's performance in capturing architectural styles from other angles. Additionally, the dataset suffered from class imbalance, further complicating the training process. To address these challenges, data augmentation techniques were employed to enhance the dataset.

In the development of the machine learning component, a notable challenge was the limited customization options available in Teachable Machine. While it provided a user-friendly interface, advanced machine learning customization options such as fine-tuning training settings and accessing specific metrics like averages, Matthews correlation, and ROC-AUC were not readily available. To overcome this limitation, the study incorporated scikit-learn, a comprehensive machine learning library, to compute the desired metrics and effectively evaluate the model's performance.

Regarding the development of the plugin, working with Blender's Python API posed challenges due to limited resources and lack of debugging tools. The absence of a debugger made it difficult to identify and resolve issues efficiently. Furthermore, challenges arose in Blender's API related to UI design, particularly in executing scripts when the value of a slider changed and incorporating thumbnails. Consequently, the plugin does not currently feature thumbnails in the menu for creating a procedural system.

When comparing this work with other studies involving procedural modeling, this study presents a significantly more powerful tool compared to AlFadalat and Al-Azhari [1]. The strength of this system lies in its flexibility, achieved through the implementation of control parameters and transformations that can be applied to the generated procedural models. In contrast to the works of Alomia et al. [6] and Paranjape et al. [11], which focus on generating a large volume of content based on external data such as geographical data, the approach taken in this study

may require more time and effort. However, the developed plugin offers extensive customization and simulation capabilities, providing urban planners with a tool that enables detailed control over the generated content.

Furthermore, this study goes beyond procedural modeling by providing users with a machine learning model to address architectural dissonance. Unlike other studies that focus solely on procedural generation, the integration of a machine learning component allows users to evaluate how well the generated models represent specific architectural styles. This added functionality enhances the utility of the system and supports sustainability efforts by reducing architectural dissonance within urban environments.

This study makes significant contributions in the field of urban modeling and planning through the development of a comprehensive application system. The system, implemented as a plugin for Blender, offers urban planners a powerful toolset with flexible control parameters and transforms, enabling them to manipulate and visualize urban elements in a dynamic 3D environment. The integration of a machine learning model enhances the system's capabilities, allowing for the evaluation of architectural styles and addressing architectural dissonance. By creating digital assets, procedural systems, and a user-friendly plugin within a popular and open-source 3D application, this study provides a practical and accessible solution for urban planning and architectural design in Philippine urban cities. Furthermore, the study addresses specific challenges encountered during development, offering insights and potential solutions for researchers in similar fields. Overall, this work contributes to the advancement of procedural modeling techniques and provides a valuable tool to support sustainable urban development and design decision-making processes.

VII. Conclusions

The study has successfully developed an application system that effectively addresses the objective of representing and manipulating urban elements in Philippine urban cities. Through the integration of machine learning, procedural generation, and interactive 3D visualization, the system provides urban planners with a scientifically grounded and powerful toolset for efficient and accurate urban modeling and planning.

The study conducted a comprehensive case study on Taguig City, leveraging image data of urban elements and identifying architectural style rules and vocabulary specific to Philippine urban cities. This rigorous approach established a solid foundation for the development of digital assets and procedural systems that faithfully capture the essence of urban elements and offer flexibility in their manipulation.

The integration of machine learning techniques further enhanced the application system's capabilities. By training and evaluating a classification model for architectural styles, the system achieved notable performance in terms of accuracy, precision, recall, F1 score, ROC AUC score, and Matthew's correlation coefficient for multiclass classification. The utilization of the best performing model ensures the system's accuracy and classification capabilities for accurate representation and manipulation of architectural styles.

To objectively assess the performance of digital assets, a dedicated tool was developed to measure geometry and storage space. This quantitative analysis provided valuable insights to urban planners, enabling informed decision-making regarding asset complexity and resource requirements during urban planning processes.

The development of a plugin for a 3D application significantly enhanced the usability and practicality of the system. Urban planners can leverage the plugin's functionalities to:

1. **Add Procedural Systems**

   The plugin allows urban planners to incorporate procedural systems into their urban models, facilitating the dynamic generation of various urban elements such as buildings, road networks, and other urban decorations.

2. **Modify Parameters**

   Urban planners can easily adjust and fine-tune parameters within the plugin, enabling them to experiment with different design scenarios and explore the impact of parameter changes on the urban environment.

3. **Measure Architectural Styles**

   The plugin provides a convenient tool for urban planners to measure and analyze architectural styles present in the urban model, allowing for a deeper understanding of the urban composition and facilitating adherence to specific architectural guidelines.

4. **Export Systems for Interoperability**

   The plugin enables urban planners to export the generated procedural systems for use in other 3D programs, promoting interoperability and seamless integration with existing workflows.

By offering these functionalities, the plugin streamlines the urban planning process, empowering urban planners to efficiently create, manipulate, and analyze urban models with accuracy and flexibility.

Throughout the study, various challenges were encountered and successfully addressed through a systematic problem-solving approach and iterative development. As a result, the application system produced is robust, functional, and capable of fulfilling the objectives set forth in the study.

The outcomes of this study make significant contributions to the advancement of the field of urban planning by providing innovative solutions to the challenges encountered during urban planning processes. By empowering urban planners with

a scientifically grounded toolset for urban modeling and planning, this system enables informed decision-making and fosters the creation of sustainable urban environments in Philippine urban cities.

VIII. Recommendations

The developed application system has demonstrated significant potential in facilitating urban modeling and planning processes. However, there are areas for improvement and enhancements that could further enhance its functionality and usability. These recommendations serve as a guide for future related studies and researchers looking to advance the field.

1. **Conduct User Testing**

   Although the study has implemented various capabilities, it is essential to conduct user testing to gather feedback and improve how urban planners interact with the system. User testing will provide valuable insights into usability issues, feature enhancements, and overall user experience, resulting in a more intuitive and user-friendly application system. This iterative feedback loop with end-users will lead to a system that better meets the needs of urban planners and facilitates efficient urban modeling and planning.

2. **Enhance the Machine Learning Model**

   While the developed machine learning model achieved satisfactory results in classifying architectural styles, further improvements can be made to enhance its performance and robustness. Future work could involve exploring advanced algorithms and techniques, such as deep learning architectures, to improve the model's accuracy and generalization capabilities. Additionally, expanding the dataset used for training to include a wider range of architectural styles and variations would contribute to a more comprehensive and accurate classification model.

3. **Expand Lighting Conditions**

   Currently, the machine learning component of the system is limited to daytime lighting conditions. To make the system more adaptable to different scenarios and lighting conditions, it is recommended to explore methods for

incorporating variations such as night-time or different weather conditions. This will provide more realistic and reliable representations of urban elements, enhancing its overall accuracy and usefulness.

4. **Improvement of Global Parameters**

   The simulation of city conditions, such as population dynamics and the effects of global warming, was naively implemented in this study. Future research should conduct in-depth studies on how these factors influence cities and incorporate them into the simulation process. This will enable urban planners to simulate and evaluate different urban scenarios with greater precision.

5. **User Interface and User Experience (UI/UX) Enhancements**

   Improving the user interface and user experience of the application system is crucial to enhance its usability and adoption. Future work should focus on designing an intuitive and visually appealing interface that allows urban planners to easily navigate and manipulate the procedural systems. Incorporating interactive visualizations, informative tooltips, and contextual help features can further enhance the user experience, enabling urban planners to gain better insights and perform more efficient analysis of the generated urban models.

6. **Integration with Geographic Information System (GIS) Tools**

   Integrating popular GIS tools, such as ArcGIS or QGIS, would greatly streamline the process of representing existing cities. In the current study, the construction of the city involved manual addition and positioning of procedural systems. By integrating GIS tools, the application system can automatically generate 3D representations of existing cities, saving significant time and effort. This integration would enable urban planners to quickly and accurately visualize real-world cities, enhancing the efficiency and realism of the urban modeling and planning process.

7. **Performance Optimization**

   As the complexity of urban models and datasets increases, optimizing the performance of the application system becomes essential. Future work should focus on optimizing computational processes, such as procedural generation and machine learning model training, to enhance efficiency and reduce processing times. Techniques like parallel computing and hardware acceleration, such as GPU utilization, can be explored to further improve system performance. Ensuring that the application system operates smoothly and efficiently will enhance productivity and user satisfaction.

These recommendations provide insights for future researchers and related studies, guiding them towards further advancing the functionality, usability, and performance of application systems for urban

IX. Bibliography

[1] M. AlFadalat and W. Al-Azhari, "An integrating contextual approach using architectural procedural modeling and augmented reality in residential buildings: the case of amman city," *Heliyon*, vol. 8, no. 8, p. e10040, 2022.

[2] V. Viro, "Problems in turning concept art into 3d objects: concept art to 3d object pipeline," 2022.

[3] L. D. Frank, N. Iroz-Elardo, K. E. MacLeod, and A. Hong, "Pathways from built environment to health: A conceptual framework linking behavior and exposure-based impacts," *Journal of Transport Health*, vol. 12, pp. 319–335, 2019.

[4] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022.

[5] G. R. C. Cruz, "A review of how philippine colonial experience influenced the country's approaches to conservation of cultural heritage," *Padayon Sining: A Celebration of the Enduring Value of the Humanities*, vol. 13, pp. 1–20, 2019.

[6] G. Alomía, D. Loaiza, C. Zúñiga, X. Luo, and R. Asorey-Cacheda, "Procedural modeling applied to the 3d city model of bogota: a case study," *Virtual Reality & Intelligent Hardware*, vol. 3, no. 5, pp. 423–433, 2021.

[7] F. Roumpani, "Procedural cities as active simulators for planning," *Urban Planning*, vol. 7, no. 2, pp. 321–329, 2022.

[8] C. Scholl and J. de Kraker, "Urban planning by experiment: practices, outcomes, and impacts," *Urban Planning*, vol. 6, no. 1, pp. 156–160, 2021.

[9] H. Xu and T.-H. Wang, "An integrated parametric generation and computational workflow to support sustainable city planning," 2022.

[10] A. Hudson-Smith, "Incoming metaverses: Digital mirrors for urban planning," *Urban Planning*, vol. 7, no. 2, 2022.

[11] 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)*, pp. 162–168, IEEE, 2020.

[12] W. M. Syafuan, R. M. Husin, and M. I. F. Azizi, "3d campus map towards sustainable development and infrastructure management in upnm," in *IOP Conference Series: Earth and Environmental Science*, vol. 1019, p. 012035, IOP Publishing, 2022.

[13] I. M. Badwi, H. M. Ellaithy, and H. E. Youssef, "3d-gis parametric modelling for virtual urban simulation using cityengine," *Annals of GIS*, pp. 1–17, 2022.

[14] N. R. Lambe and A. R. Dongre, "A shape grammar approach to contextual design: a case study of the pol houses of ahmedabad, india," *Environment and Planning B: Urban Analytics and City Science*, vol. 46, no. 5, pp. 845–861, 2019.

[15] F. Liauw, "Reference for contextual design," in *IOP Conference Series: Materials Science and Engineering*, vol. 508, p. 012031, IOP Publishing, 2019.

[16] F. Buonamici, M. Carfagni, R. Furferi, Y. Volpe, and L. Governi, "Generative design: an explorative study," *Computer-Aided Design and Applications*, vol. 18, no. 1, pp. 144–155, 2020.

[17] M. Denk, J. Mayer, H. Völkl, S. Wartzack, *et al.*, "Procedural concept design with computer graphic applications for light-weight structures using blender with subdivision surfaces," in *DS 119: Proceedings of the 33rd Symposium Design for X (DFX2022)*, pp. 1–10, 2022.

[18] N. Sharma, V. Jain, and A. Mishra, "An analysis of convolutional neural networks for image classification," *Procedia Computer Science*, vol. 132, pp. 377–384, 2018. International Conference on Computational Intelligence and Data Science.

[19] K. Dong, C. Zhou, Y. Ruan, and Y. Li, "Mobilenetv2 model for image classification," in *2020 2nd International Conference on Information Technology and Computer Application (ITCA)*, pp. 476–480, IEEE, 2020.

X. Appendix

A. Source Code

```
#include <iostream>

using namespace std;

int main{
        cout << "Hello world!" << endl;
        return 0;
}
```

```
bl_info = {
    "name": "Procedural City Tool",
    "author": "Adrian Neil Santos",
    "version": (1, 0),
    "blender": (2, 80, 0),
    "location": "View3D > Toolshelf > Procedural City Plugin",
    "description": "A tool for creating 3D digital replicas of cities.",
    "warning": "",
    "wiki_url": "",
    "category": "Plugin"
}

import sys
import subprocess
import os
import platform
import bpy


# Append the current directory to the system path
# This allows us to import modules from the subdirectories
sys.path.append(os.path.dirname(__file__))

# Import the submodules
from . import panels
from . import operators
from . import properties




#### Module Installation ####

def isWindows():
    return os.name == 'nt'

def isMacOS():
    return os.name == 'posix' and platform.system() == "Darwin"

def isLinux():
    return os.name == 'posix' and platform.system() == "Linux"

def python_exec():
    if isWindows():
        return os.path.join(sys.prefix, 'bin', 'python.exe')
    elif isMacOS():
        try:
            # 2.92 and older
            path = bpy.app.binary_path_python
        except AttributeError:
            # 2.93 and later
            path = sys.executable
        return os.path.abspath(path)
    elif isLinux():
        return os.path.join(sys.prefix, 'sys.prefix/bin', 'python')
    else:
        print("sorry, still not implemented for ", os.name, " - ", platform.system)


def installModule(packageName):
    try:
        subprocess.call([python_exe, "-c", "import " + packageName])
    except:
        python_exe = python_exec()
        # upgrade pip
        subprocess.call([python_exe, "-m", "ensurepip"])
        subprocess.call([python_exe, "-m", "pip", "install", "--upgrade", "pip"])
        # install required packages
        subprocess.call([python_exe, "-m", "pip", "install", packageName])


def installModules():
    installModule("pandas")
    installModule("tensorflowjs")
    installModule("tensorflow")
    installModule("keras")
    installModule("opencv-python")
```

```
    installModule("Pillow")
    installModule("protobuf<=3.20.0")


def register():
    installModules()
    properties.register()
    operators.register()
    panels.register()


def unregister():
    properties.unregister()
    operators.unregister()
    panels.unregister()


if __name__ == "__main__":
    register()
```

A..1 Operators

```
from . import evaluate_system_operators
from . import proc_city_operators
from . import proc_system_operators
from . import export_operators


def register():
    evaluate_system_operators.register()
    proc_city_operators.register()
    proc_system_operators.register()
    export_operators.register()

def unregister():
    evaluate_system_operators.unregister()
    proc_city_operators.unregister()
    proc_system_operators.unregister()
    export_operators.unregister()


import bpy


import bpy
import numpy as np
import cv2
import tensorflowjs as tfjs
import os
import mathutils



def classify():

    # Set the render engine to Eevee
    bpy.context.scene.render.engine = 'BLENDER_EEVEE'


        # Production directory
    try:

        addon_name = "Source Code"
        addon_dir = os.path.join(bpy.utils.user_resource('SCRIPTS'), "addons", addon_name)

        # Get the path to the blend file
        model_dir = os.path.join(addon_dir, "Model Export")

        # load the model
        model = tfjs.converters.load_keras_model(os.path.join(model_dir, 'model.json'))


    # Test directory
    except:

        # Get the directory of the script
        script_dir = os.path.dirname(os.path.realpath(__file__))

        # Get the path to the blend file
        model_dir = os.path.join(script_dir, "../Source Code", "Model Export")

        # load the model
        model = tfjs.converters.load_keras_model(os.path.join(model_dir, 'model.json'))




    create_compositor_node()

    # moves the camera to scene view
    # TO IMPLEMENT
    move_camera_to_view()

    # render depends on camera
```

```
    bpy.ops.render.render()

    render_result = bpy.data.images['Viewer Node']

    # get viewer pixels
    pixels = render_result.pixels

    # copy buffer to numpy array for faster manipulation
    arr = np.array(pixels[:])
    print(f"img data raw: {arr}")

    # Scale the RGB pixel values to the range 0-255
    min_value = 0.04561729
    max_value = 1.3825562
    scale_factor = (max_value - min_value) * 255


    arr[0::4] = (arr[0::4] * scale_factor).astype(np.uint8)
    arr[1::4] = (arr[1::4] * scale_factor).astype(np.uint8)
    arr[2::4] = (arr[2::4] * scale_factor).astype(np.uint8)
    print(f"img data scaled: {arr}")

    # reshape the array into a 2D image
    img_data = arr.reshape(render_result.size[1], render_result.size[0], 4)

    # Convert to RGB format
    img_data = np.clip(img_data, 0, 255).astype(np.uint8)
    img_data = cv2.cvtColor(img_data, cv2.COLOR_RGBA2BGR)


    # display the image in a window

    # Flip the image vertically
    flipped_img = cv2.flip(img_data, 0)


    # Predict the class of the image
    prediction = predict_class(flipped_img, model)
    class_idx = np.argmax(prediction)

    print(f"prediction is {prediction}")
    i = 0
    for prediction_percent in prediction[0]:
        print(f"Class {i}: {prediction_percent:.2f}%")
        i += 1


    print(f"Prediction is {class_idx}")

    # Save the image to disk
    image_path = os.path.join(model_dir, "render/render_ML.jpg")

    cv2.imwrite(image_path, flipped_img)

    # cv2.imshow('image', flipped_img)
    # cv2.waitKey(0)
    # cv2.destroyAllWindows()

    return prediction



def create_compositor_node():
    # switch on nodes
    bpy.context.scene.use_nodes = True
    tree = bpy.context.scene.node_tree
    links = tree.links

    # clear default nodes
    for n in tree.nodes:
        tree.nodes.remove(n)


    # create input render layer node
    rl = tree.nodes.new('CompositorNodeRLayers')
    rl.location = 185,285

    # create output node
    v = tree.nodes.new('CompositorNodeViewer')
    v.location = 750,210
    v.use_alpha = True

    # Links
    links.new(rl.outputs[0], v.inputs[0])  # link Image output to Viewer input
    links.new(rl.outputs[1], v.inputs[1])  # link alpha



def predict_class(img_data, model):
    # Preprocess the image data
    img_data = cv2.resize(img_data, (224, 224)) # Assuming the model requires 224x224 input size
    img_data = img_data.astype(np.float32) / 255.0 # Normalize pixel values to [0, 1]
    img_data = np.expand_dims(img_data, axis=0) # Add batch dimension

    # Predict the class
    prediction = model.predict(img_data)

    return prediction
```

```
# To implement
def move_camera_to_view():
    # Get the active scene
    scene = bpy.context.scene

    # Get the active camera
    camera = scene.camera

    # Get the view matrix from the current region
    view_matrix = bpy.context.region_data.view_matrix.copy()

    # Set the camera's location and rotation to match the view matrix
    camera.matrix_world = view_matrix.inverted()

    # Update the scene
    bpy.context.view_layer.update()




class EvaluateOperator(bpy.types.Operator):
    """An operator that evaluates a procedural system"""
    bl_idname = "operator.evaluate"
    bl_label = "Evaluate"

    def execute(self, context):
        probs = classify()
        evaluate_props = context.scene.evaluate_properties
        evaluate_props.class1 = "{:.2%}".format(probs[0][0])
        evaluate_props.class2 = "{:.2%}".format(probs[0][1])
        evaluate_props.class3 = "{:.2%}".format(probs[0][2])

        prediction_result = np.argmax(probs)

        labels = ["Contemporary", "Spanish Colonial",
                  "Vernacular"]

        evaluate_props.actual_prediction = labels[prediction_result]

        return {'FINISHED'}

    def draw(self, context):
        # Create a label for the class probabilities
        label = self.layout.label()
        label.text = "Click Evaluate to classify the mesh"

        # Add a button to trigger the classification
        self.layout.operator("operator.evaluate")

        # Create a label to display the class probabilities
        label = self.layout.label()
        label.text = context.scene.my_class_probabilities







def register():
    bpy.utils.register_class(EvaluateOperator)


def unregister():
    bpy.utils.unregister_class(EvaluateOperator)

import bpy


class ExportAllOperators(bpy.types.Operator):
    """Operator to export the entire scene"""
    bl_idname = "operator.export_all"
    bl_label = "Export All"

    filepath: bpy.props.StringProperty(subtype='FILE_PATH')

    @classmethod
    def poll(cls, context):
        return context.scene is not None

    def execute(self, context):
        # Combine the user-specified file path with the file name
        filepath = bpy.path.ensure_ext(self.filepath, ".fbx")
        print(f"Exporting to: {filepath}")

        if filepath:
            print(f"Exporting the scene as FBX to: {filepath}")

            # Set the export file path
            bpy.context.scene.render.filepath = filepath

            # Export the scene as FBX
            bpy.ops.export_scene.fbx(filepath=filepath, use_selection=False)
        else:
            print("Please specify a file path for export.")
```

```
        return {'FINISHED'}

    def invoke(self, context, event):
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}


class ExportSelectedOperators(bpy.types.Operator):
    """Operator to export the selected object"""
    bl_idname = "operator.export_selected"
    bl_label = "Export Selected"

    filepath: bpy.props.StringProperty(subtype='FILE_PATH')

    @classmethod
    def poll(cls, context):
        return context.active_object is not None

    def execute(self, context):
        # Combine the user-specified file path with the file name
        filepath = bpy.path.ensure_ext(self.filepath, ".fbx")
        print(f"Exporting to: {filepath}")

        if filepath:
            print(f"Exporting the selected object as FBX to: {filepath}")

            # Set the export file path
            bpy.context.scene.render.filepath = filepath

            # Export the selected object as FBX
            bpy.ops.export_scene.fbx(filepath=filepath, use_selection=True)
        else:
            print("Please specify a file path for export.")

        return {'FINISHED'}

    def invoke(self, context, event):
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}



def register():
    bpy.utils.register_class(ExportAllOperators)
    bpy.utils.register_class(ExportSelectedOperators)


def unregister():
    bpy.utils.unregister_class(ExportAllOperators)
    bpy.utils.unregister_class(ExportSelectedOperators)

import bpy



def spawn_cube():
    # Create a new cube object
    bpy.ops.mesh.primitive_cube_add()

    # The newly created cube will be the active object
    cube_object = bpy.context.object

    # Modify the cube's properties if needed
    cube_object.location = (0, 0, 0)
    cube_object.scale = (1, 1, 1)

    # Optional: Set the cube as the selected object
    bpy.context.view_layer.objects.active = cube_object
    cube_object.select_set(True)



class ProcCityOperator(bpy.types.Operator):
    """An operator that evaluates a procedural system"""
    bl_idname = "operator.proc_city"
    bl_label = "Evaluate"

    def execute(self, context):
        print("Hello, World!")
        return {'FINISHED'}


class RefreshPropertiesOperator(bpy.types.Operator):
    """An operator that evaluates a procedural system"""
    bl_idname = "operator.refresh_props"
    bl_label = "Refresh Properties"

    def execute(self, context):
        print("Hello, World!")
        refresh_conditions(self, context)
        return {'FINISHED'}



def refresh_conditions(self, context):
    properties = context.scene.conditions_properties
```

```
    target_parameters = ["height", "length", "width", "X_Count", "Y_Count", "X_Spacing", "Y_Spacing"]
    target_collections = ["Residential Buildings"]
    object_list = get_objects(target_collections)
    updated_conditions = recalculate_conditions(object_list, target_parameters)

    properties["population"] = updated_conditions["population"]
    properties["wealth"] = updated_conditions["wealth"]
    properties["transportation"] = updated_conditions["transportation"]



def recalculate_conditions(object_list, target_parameters):
    population = 0
    wealth = 0
    transportation = 0

    for obj in object_list:
        height = 1
        width = 1
        length = 1

        x_count = 1
        y_count = 1
        x_spacing = 1
        y_spacing = 1

        if "GeometryNodes" in obj.modifiers:
            geo_tree = obj.modifiers["GeometryNodes"].node_group

            # Extracting properties
            for node in geo_tree.nodes:
                if any(param in node.name for param in target_parameters):
                    if "height" in node.name:
                        height = node.integer
                    elif "length" in node.name:
                        length = node.integer
                    elif "width" in node.name:
                        width = node.integer
                    elif "X_Count" in node.name:
                        x_count = node.integer
                    elif "Y_Count" in node.name:
                        y_count = node.integer
                    elif "X_Spacing" in node.name:
                        x_spacing = node.integer
                    elif "Y_Spacing" in node.name:
                        y_spacing = node.integer


        population += height*width*length*15
        wealth += height*width*length*45000

        transport_add = (x_count * x_spacing) + (y_count * y_spacing)
        if(transport_add == 2):
            transportation += 0
        else:
            transportation += transport_add



    conditions = {
        "population": population,
        "wealth": wealth,
        "transportation": transportation,
    }

    return conditions



def get_objects(target_collections):
    object_list = []

    for collection_name in target_collections:
        collection = bpy.data.collections.get(collection_name)
        if collection is not None:
            for obj in collection.objects:
                if obj.type == 'MESH':
                    object_list.append(obj)

    return object_list



def get_objects_nodes(object_list, target_parameters):
    property_nodes = []
    for obj in object_list:
        if "GeometryNodes" in obj.modifiers:
            geo_tree = obj.modifiers["GeometryNodes"].node_group

            # Extracting properties
            for node in geo_tree.nodes:
                if any(param in node.name for param in target_parameters):
                    property_nodes.append(node)
```

```
    return property_nodes



def modify_nodes(object_list, target_parameters, diff):
    nodes = get_objects_nodes(object_list, target_parameters)

    # modifying node values
    for node in nodes:
        node.integer += diff



def register():
    bpy.utils.register_class(ProcCityOperator)
    bpy.utils.register_class(RefreshPropertiesOperator)


def unregister():
    bpy.utils.unregister_class(ProcCityOperator)
    bpy.utils.unregister_class(RefreshPropertiesOperator)

import bpy
import os



class SpawnResidentialAOperator(bpy.types.Operator):
    bl_idname = "operator.spawn_residential_a"
    bl_label = "residential-A"

    def execute(self, context):
        # Set the name of the collection to add
        collection_name = "residential-A"
        spawn_collection(collection_name)
        # Add the spawned object to collection
        # get the object
        obj = bpy.context.object
        # provide collection name
        collection = "Residential Buildings"
        add_obj_to_collection(obj, collection)

        return {'FINISHED'}



class SpawnResidentialBOperator(bpy.types.Operator):
    bl_idname = "operator.spawn_residential_b"
    bl_label = "residential-B"

    def execute(self, context):
        # Set the name of the collection to add
        collection_name = "residential-B"
        spawn_collection(collection_name)
        # Add the spawned object to collection
        # get the object
        obj = bpy.context.object
        # provide collection name
        collection = "Residential Buildings"
        add_obj_to_collection(obj, collection)

        return {'FINISHED'}


class SpawnResidentialCornerAOperator(bpy.types.Operator):
    bl_idname = "operator.spawn_corner_a"
    bl_label = "residential_corner_A"

    def execute(self, context):
        # Set the name of the collection to add
        collection_name = "residential_corner_A"
        spawn_collection(collection_name)
        # Add the spawned object to collection
                # get the object
        obj = bpy.context.object
        # provide collection name
        collection = "Residential Buildings"
        add_obj_to_collection(obj, collection)

        return {'FINISHED'}


class SpawnSkyscraperAOperator(bpy.types.Operator):
    bl_idname = "operator.spawn_skyscraper_a"
    bl_label = "skyscraper-A"

    def execute(self, context):
        # Set the name of the collection to add
        collection_name = "skyscraper-A"
        spawn_collection(collection_name)
        # Add the spawned object to collection
                # get the object
        obj = bpy.context.object
        # provide collection name
        collection = "Residential Buildings"
```

```
        add_obj_to_collection(obj, collection)

        return {'FINISHED'}


class SpawnSkyscraperBOperator(bpy.types.Operator):
    bl_idname = "operator.spawn_skyscraper_b"
    bl_label = "skyscraper-B"

    def execute(self, context):
        # Set the name of the collection to add
        collection_name = "skyscraper-B"
        spawn_collection(collection_name)
        # Add the spawned object to collection
                # get the object
        obj = bpy.context.object
        # provide collection name
        collection = "Residential Buildings"
        add_obj_to_collection(obj, collection)

        return {'FINISHED'}



class SpawnSkyscraperCOperator(bpy.types.Operator):
    bl_idname = "operator.spawn_skyscraper_c"
    bl_label = "skyscraper-C"

    def execute(self, context):
        # Set the name of the collection to add
        collection_name = "skyscraper-C"
        spawn_collection(collection_name)
        # Add the spawned object to collection
                # get the object
        obj = bpy.context.object
        # provide collection name
        collection = "Residential Buildings"
        add_obj_to_collection(obj, collection)

        return {'FINISHED'}



class SpawnTreeOperator(bpy.types.Operator):
    bl_idname = "operator.spawn_tree"
    bl_label = "tree"

    def execute(self, context):
        # Set the name of the collection to add
        collection_name = "procedural_tree"
        spawn_collection(collection_name)
        # Add the spawned object to collection
                # get the object
        obj = bpy.context.object
        # provide collection name
        collection = "Trees"
        add_obj_to_collection(obj, collection)

        return {'FINISHED'}


class SpawnRoadOperator(bpy.types.Operator):
    bl_idname = "operator.spawn_road"
    bl_label = "road"

    def execute(self, context):
        # Set the name of the collection to add
        collection_name = "procedural_road"
        spawn_collection(collection_name)
        # Add the spawned object to collection
            # get the object
        obj = bpy.context.object
        # provide collection name
        collection = "Roads"
        add_obj_to_collection(obj, collection)

        return {'FINISHED'}






class ProcSystemMenu(bpy.types.Operator):
    """Menu of Proc System"""
    bl_idname = "operator.proc_system_menu"
    bl_label = "Proc System Menu"

    selection: bpy.props.EnumProperty(
        name="",
        description="Select an option",
        items=[
            ("RES_A", "Residential A", "Description"),
            ("RES_B", "Residential B", "Description"),
            ("RES_Corner_A", "Residential Corner A", "Description"),
            ("SKY_A", "Skyscraper A", "Description"),
            ("SKY_B", "Skyscraper B", "Description"),
```

```
            ("SKY_C", "Skyscraper C", "Description"),
            ("TREE", "Tree", "Description"),
            ("ROAD", "Road", "Description"),
        ],
    )

    def invoke(self, context, event):
        wm = context.window_manager
        return wm.invoke_props_dialog(self)



    def execute(self, context):
        # get the selected option
        selected_option = self.selection

        # do something with the selected option
        if selected_option == "RES_A":
            bpy.ops.operator.spawn_residential_a()
        elif selected_option == "RES_B":
            bpy.ops.operator.spawn_residential_b()
        elif selected_option == "RES_Corner_A":
            bpy.ops.operator.spawn_corner_a()
        elif selected_option == "SKY_A":
            bpy.ops.operator.spawn_skyscraper_a()
        elif selected_option == "SKY_B":
            bpy.ops.operator.spawn_skyscraper_b()
        elif selected_option == "SKY_C":
            bpy.ops.operator.spawn_skyscraper_c()
        elif selected_option == "TREE":
            bpy.ops.operator.spawn_tree()
        elif selected_option == "ROAD":
            bpy.ops.operator.spawn_road()



        return {'FINISHED'}


    def draw(self, context):
        layout = self.layout
        layout.prop(self, "selection")



# UTILIIES
def spawn_collection(collection_name):

    # Production directory
    try:
        blend_name = "assets.blend"

        addon_name = "Source Code"
        addon_dir = os.path.join(bpy.utils.user_resource('SCRIPTS'), "addons", addon_name)

        # Get the path to the blend file
        blend_path = os.path.join(addon_dir, "Blend File", blend_name)

        # Load the blend file
        with bpy.data.libraries.load(blend_path) as (data_from, data_to):
            data_to.collections = [collection_name]



    # Test directory
    except:
        blend_name = "assets.blend"

        # Get the directory of the script
        script_dir = os.path.dirname(os.path.realpath(__file__))

        # Get the path to the blend file
        blend_path = os.path.join(script_dir, "../Source Code", "Blend File", blend_name)

        # Load the blend file
        with bpy.data.libraries.load(blend_path) as (data_from, data_to):
            data_to.collections = [collection_name]



    # Append the collection to the scene
    bpy.ops.object.select_all(action='DESELECT')
    for collection in data_to.collections:
        for obj in collection.objects:
            bpy.context.scene.collection.objects.link(obj)
            obj.select_set(True)
            bpy.context.view_layer.objects.active = obj  # Make the object active
            # set object location to 3D cursor location
            obj.location = bpy.context.scene.cursor.location




def add_obj_to_collection(obj, collection):
    # get the collection to add the object to
    my_collection = bpy.data.collections.get(collection)
```

```
    # make sure the collection exists
    if my_collection is None:
        # create the collection if it does not exist
        my_collection = bpy.data.collections.new(collection)
        bpy.context.scene.collection.children.link(my_collection)

    # remove the object from all collections
    for coll in obj.users_collection:
        coll.objects.unlink(obj)
    # add the object to the desired collection
    my_collection.objects.link(obj)


def spawn_cube():
    # create a cube object
    bpy.ops.mesh.primitive_cube_add()

    # get the object
    obj = bpy.context.object
    # provide collection name
    collection = "Cubes"

    add_obj_to_collection(obj, collection)




def register():
    bpy.utils.register_class(SpawnResidentialAOperator)
    bpy.utils.register_class(SpawnResidentialBOperator)
    bpy.utils.register_class(SpawnResidentialCornerAOperator)
    bpy.utils.register_class(SpawnSkyscraperAOperator)
    bpy.utils.register_class(SpawnSkyscraperBOperator)
    bpy.utils.register_class(SpawnSkyscraperCOperator)


    bpy.utils.register_class(ProcSystemMenu)




    bpy.utils.register_class(SpawnTreeOperator)
    bpy.utils.register_class(SpawnRoadOperator)


def unregister():
    bpy.utils.unregister_class(SpawnResidentialAOperator)
    bpy.utils.unregister_class(SpawnResidentialBOperator)
    bpy.utils.unregister_class(SpawnSkyscraperAOperator)
    bpy.utils.unregister_class(SpawnSkyscraperBOperator)
    bpy.utils.unregister_class(SpawnSkyscraperCOperator)


    bpy.utils.unregister_class(ProcSystemMenu)




    bpy.utils.unregister_class(SpawnTreeOperator)
    bpy.utils.unregister_class(SpawnRoadOperator)
```

A..2 Panels

```
from . import proc_city_panels
from . import proc_system_panels
from . import evaluate_system_panels
from . import export_panels



def register():
    proc_city_panels.register()
    proc_system_panels.register()
    evaluate_system_panels.register()
    export_panels.register()

def unregister():
    proc_city_panels.unregister()
    proc_system_panels.unregister()
    evaluate_system_panels.unregister()
    export_panels.unregister()


import bpy


class EvaluateSystemPanel(bpy.types.Panel):
        bl_label = "Evaluate System"
        bl_idname = "Evaluate_System_Panel"
        bl_space_type = "VIEW_3D"
        bl_region_type = "UI"
        bl_category = "Procedural City Plugin"

        def draw(self, context):
```

```
            pass


class ModelInformationPanel(bpy.types.Panel):
    bl_label = "Model Information"
    bl_idname = "Model_Information_Panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"
    bl_category = "Procedural City Plugin"
    bl_parent_id = 'Evaluate_System_Panel'

    def draw(self, context):
        layout = self.layout
        box = layout.box()
        box.label(text=f"Accuracy: {1.00}")
        box.label(text=f"Precision: {1.00}")
        box.label(text=f"Recall: {1.00}")
        box.label(text=f"F1 Score: {1.00}")
        box.label(text=f"ROC AUC: {1.00}")
        box.label(text=f"Matthew's Correlation: {1.00}")


class ArchitecturalStylePanel(bpy.types.Panel):
    bl_label = "Architectural Style"
    bl_idname = "Architectural_Style_Panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"
    bl_category = "Procedural City Plugin"
    bl_parent_id = 'Evaluate_System_Panel'

    def draw(self, context):
        layout = self.layout

        # Create a box to hold the class labels
        box = layout.box()

        # Add the class labels to the box
        evaluate_props = context.scene.evaluate_properties
        box.label(text=f'Contemporary : {evaluate_props.class1}')
        box.label(text=f'Spanish Colonial     : {evaluate_props.class2}')
        box.label(text=f'Vernacular       : {evaluate_props.class3}')

        box.label(text=f'Prediction       : {evaluate_props.actual_prediction}')

        box.operator("operator.evaluate", text="Evaluate")



class MeshPropertiesPanel(bpy.types.Panel):
    bl_label = "Mesh Properties"
    bl_idname = "Mesh_Properties_Panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"
    bl_category = "Procedural City Plugin"
    bl_parent_id = 'Evaluate_System_Panel'

    def draw(self, context):
        layout = self.layout

        # Global Mesh Properties
        row = layout.row(align=True)
        row.label(text="Global Mesh Properties")
        # Property Group
        box = layout.box()
        global_mesh_properties = get_global_mesh_properties()
        box.label(text=f'Total Objects : {global_mesh_properties["total_objects"]}')
        box.label(text=f'Total Polygon Count : {global_mesh_properties["total_polygon_count"]}')
        box.label(text=f'Total Vertex Count : {global_mesh_properties["total_vertex_count"]}')
        box.label(text=f'Total Edge Count : {global_mesh_properties["total_edge_count"]}')
        box.label(text=f'Total Storage Size : {global_mesh_properties["total_storage_size"]:.2f} kilobytes')

        # Local Mesh Properties
        row = layout.row(align=True)
        row.label(text="Local Mesh Properties")
        # Property Group
        box = layout.box()


        #local_mesh_properties = context.scene.local_mesh_properties
        local_mesh_properties = get_local_mesh_properties()

        box.label(text=f'Polygon Count : {local_mesh_properties["polygon_count"]}')
        box.label(text=f'Vertex Count : {local_mesh_properties["vertex_count"]}')
        box.label(text=f'Edge Count : {local_mesh_properties["edge_count"]}')
        box.label(text=f'Storage Space : {local_mesh_properties["storage_size"]:.2f} kilobytes')




def get_global_mesh_properties():
    total_polygon_count = 0
    total_objects = 0
    total_storage_size = 0
    total_vertex_count = 0
    total_edge_count = 0
```

```
    # Specify the names of the collections to consider
    collection_names = ["Residential Buildings", "Roads", "Trees"]

    # Iterate over the collections
    for collection_name in collection_names:
        collection = bpy.data.collections.get(collection_name)
        if collection is not None:
            # Iterate over the objects in the collection
            for obj in collection.objects:
                if obj.type == 'MESH':
                    total_objects += 1
                    mesh_object = obj
                    output_node = mesh_object.evaluated_get(bpy.context.evaluated_depsgraph_get()).data
                    total_polygon_count += len(output_node.polygons)
                    storage_size_bytes = (len(output_node.edges) * 4) + (len(output_node.vertices) * 12)
                    storage_size_kilobytes = storage_size_bytes / 1024.0
                    total_storage_size += storage_size_kilobytes
                    total_vertex_count += len(output_node.vertices)
                    total_edge_count += len(output_node.edges)




    # Create a dictionary to store the mesh properties
    mesh_properties = {
        'total_polygon_count': total_polygon_count,
        'total_objects': total_objects,
        'total_vertex_count': total_vertex_count,
        'total_edge_count': total_edge_count,
        'total_storage_size': total_storage_size,
    }

    return mesh_properties


def get_local_mesh_properties():
    mesh_object = bpy.context.object
    try:
        # Get the output mesh data from the geometry node tree
        output_node = mesh_object.evaluated_get(bpy.context.evaluated_depsgraph_get()).data
    except:
        output_node = mesh_object.data

    polygon_count = len(output_node.polygons)
    storage_size_bytes = (len(output_node.edges) * 4) + (len(output_node.vertices) * 12)
    storage_size_kilobytes = storage_size_bytes / 1024.0
    vertex_count = len(output_node.vertices)
    edge_count = len(output_node.edges)

    # Create a dictionary to store the mesh properties
    mesh_properties = {
        'polygon_count': polygon_count,
        'storage_size': storage_size_kilobytes,
        'vertex_count': vertex_count,
        'edge_count': edge_count,
    }

    return mesh_properties




def register():
    bpy.utils.register_class(EvaluateSystemPanel)
    bpy.utils.register_class(ModelInformationPanel)
    bpy.utils.register_class(ArchitecturalStylePanel)
    bpy.utils.register_class(MeshPropertiesPanel)


def unregister():
    bpy.utils.unregister_class(EvaluateSystemPanel)
    bpy.utils.unregister_class(ModelInformationPanel)
    bpy.utils.unregister_class(ArchitecturalStylePanel)
    bpy.utils.unregister_class(MeshPropertiesPanel)

import bpy


class ExportPanel(bpy.types.Panel):
        bl_label = "Export"
        bl_idname = "Export_Panel"
        bl_space_type = "VIEW_3D"
        bl_region_type = "UI"
        bl_category = "Procedural City Plugin"

        def draw(self, context):
            layout = self.layout


            row = layout.row()
            row.operator("operator.export_all", text="Export All")
            row.operator("operator.export_selected", text="Export Selected")
```

```
def register():
    bpy.utils.register_class(ExportPanel)


def unregister():
    bpy.utils.unregister_class(ExportPanel)

import bpy

class ProceduralCityPanel(bpy.types.Panel):
        bl_label = "Procedural City"
        bl_idname = "Procedural_City_Panel"
        bl_space_type = "VIEW_3D"
        bl_region_type = "UI"
        bl_category = "Procedural City Plugin"

        def draw(self, context):
            pass




class GlobalParametersPanel(bpy.types.Panel):
    """A custom panel in the 3D Viewport"""
    bl_label = "Global Parameters"
    bl_idname = "Global_Parameters_Panel"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "Procedural City Plugin"
    bl_parent_id = 'Procedural_City_Panel'



    def draw(self, context):
        layout = self.layout

        row=layout.row()


        box = layout.box()
        box.label(text="City Conditions")
        conditions_props = context.scene.conditions_properties

        box.prop(conditions_props, "population", text='Population')
        box.prop(conditions_props, "wealth", text='Wealth')
        box.prop(conditions_props, "transportation", text='Transportation')
        box.prop(conditions_props, "environment", text='Environment')
        box.operator(operator="operator.refresh_props", text="Refresh Conditions")






class LocalParametersPanel(bpy.types.Panel):
    """A custom panel in the 3D Viewport"""
    bl_label = "Local Parameters"
    bl_idname = "Local_Parameters_Panel"
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = "Procedural City Plugin"
    bl_parent_id = 'Procedural_City_Panel'


    def draw(self, context):
        layout = self.layout



        box = layout.box()

        box.label(text="BUILDINGS")
        buildings_props = context.scene.buildings_properties

        row = box.row()
        row.label(text="Minimum")

        row.prop(buildings_props, "min_width", text='Width')
        row.prop(buildings_props, "min_length", text='Length')
        row.prop(buildings_props, "min_height", text='Height')


        row = box.row()
        row.label(text="Maximum")
        row.prop(buildings_props, "max_width", text='Width')
        row.prop(buildings_props, "max_length", text='Length')
        row.prop(buildings_props, "max_height", text='Height')



        box=layout.box()
```

```
        box.label(text="ROADS")
        roads_props = context.scene.roads_properties

        row = box.row()
        row.label(text="Width Count")
        row.prop(roads_props, "min_width_count", text='Min')
        row.prop(roads_props, "max_width_count", text='Max')
        row = box.row()
        row.label(text="Length Count")
        row.prop(roads_props, "min_length_count", text='Min')
        row.prop(roads_props, "max_length_count", text='Max')
        row = box.row()
        row.label(text="Width Spacing")
        row.prop(roads_props, "min_width_spacing", text='Min')
        row.prop(roads_props, "max_width_spacing", text='Max')
        row = box.row()
        row.label(text="Length Spacing")
        row.prop(roads_props, "min_length_spacing", text='Min')
        row.prop(roads_props, "max_length_spacing", text='Max')



        box=layout.box()
        box.label(text="TREES")
        trees_props = context.scene.trees_properties

        row = box.row()
        row.label(text="Tree Height")
        row.prop(trees_props, "min_tree_height", text='Min')
        row.prop(trees_props, "max_tree_height", text='Max')
        row = box.row()
        row.label(text="Branches Count")
        row.prop(trees_props, "min_branches_count", text='Min')
        row.prop(trees_props, "max_branches_count", text='Max')
        row = box.row()
        row.label(text="Branches Length")
        row.prop(trees_props, "min_branches_length", text='Min')
        row.prop(trees_props, "max_branches_length", text='Max')
        row = box.row()
        row.label(text="Leaves Count")
        row.prop(trees_props, "min_leaves_count", text='Min')
        row.prop(trees_props, "max_leaves_count", text='Max')




def register():
    bpy.utils.register_class(ProceduralCityPanel)
    bpy.utils.register_class(GlobalParametersPanel)
    bpy.utils.register_class(LocalParametersPanel)


def unregister():
    bpy.utils.unregister_class(ProceduralCityPanel)
    bpy.utils.unregister_class(GlobalParametersPanel)
    bpy.utils.unregister_class(LocalParametersPanel)

import bpy


class ProceduralSystemsPanel(bpy.types.Panel):
    bl_label = "Procedural Systems"
    bl_idname = "Procedural_Systems_Panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"
    bl_category = "Procedural City Plugin"

    def draw(self, context):
        pass




class SpawnSystemsPanel(bpy.types.Panel):
    bl_label = "Spawn Procedural Systems"
    bl_idname = "Spawn_Systems_Panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"
    bl_category = "Procedural City Plugin"
    bl_parent_id = 'Procedural_Systems_Panel'

    def draw(self, context):
        layout = self.layout
        box = layout.box()
        show_spawn_location(box)
        show_operators(box, context)



class ObjectTransformsPanel(bpy.types.Panel):
    bl_label = "Object Transforms"
    bl_idname = "Object_Transform_Panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"
    bl_category = "Procedural City Plugin"
```

```
    bl_parent_id = 'Procedural_Systems_Panel'

    def draw(self, context):
        layout = self.layout
        box = layout.box()

        # Check if an object is selected
        obj = context.active_object

        if obj is not None and obj.type == 'MESH':
            show_transform_controls(obj, box)
        else:
            box.label(text="No object selected")


class ParametersControlPanel(bpy.types.Panel):
    bl_label = "Parameters Control"
    bl_idname = "Parameters_Control_Panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"
    bl_category = "Procedural City Plugin"
    bl_parent_id = 'Procedural_Systems_Panel'

    def draw(self, context):
        layout = self.layout
        box = layout.box()

        # Check if an object is selected
        obj = context.active_object

        if obj is not None and obj.type == 'MESH':
            show_geometry_controls(obj, box)
        else:
            box.label(text="No object selected")




def show_operators(box, context):
    box.operator("operator.proc_system_menu", text="Select Proc System")



def show_spawn_location(box):
    box.label(text="Spawn Location")
    box.prop(bpy.context.scene.cursor, "location", index=0, text="X")
    box.prop(bpy.context.scene.cursor, "location", index=1, text="Y")


def show_transform_controls(obj, box):

    box.label(text="Transform Controls")
    box.prop(obj, "location", index=0, text="X")
    box.prop(obj, "location", index=1, text="Y")
    box.prop(obj, "rotation_euler", index=2, text="Rotation")
    box.prop(obj, "scale")



def show_geometry_controls(obj, box):
    box.label(text="Geometry Controls")

    if "GeometryNodes" in obj.modifiers:
        # Check if the object has a GeometryNodeTree
        geo_tree = obj.modifiers["GeometryNodes"].node_group

        # Extracting properties
        property_nodes = []
        for node in geo_tree.nodes:
            if "property" in node.name:
                property_nodes.append(node)

        property_nodes.sort(key=lambda node: node.name)

        property_types = ["integer", "boolean", "default_value"]

        for property_node in property_nodes:
            if hasattr(property_node, 'integer'):
                # Access integer property
                box.prop(property_node, 'integer', text=property_node.label)
            elif hasattr(property_node, 'boolean'):
                # Access boolean property
                box.prop(property_node, 'boolean', text=property_node.label)
            elif len(property_node.outputs) > 0:
                try:
                    # Access float property from the first output socket
                    box.prop(property_node.outputs[0], "default_value", text=property_node.label)
                except:
                    pass




def register():
```

```
    bpy.utils.register_class(ProceduralSystemsPanel)
    bpy.utils.register_class(SpawnSystemsPanel)
    bpy.utils.register_class(ObjectTransformsPanel)
    bpy.utils.register_class(ParametersControlPanel)



def unregister():
    bpy.utils.unregister_class(ProceduralSystemsPanel)
    bpy.utils.unregister_class(SpawnSystemsPanel)
    bpy.utils.unregister_class(ObjectTransformsPanel)
    bpy.utils.unregister_class(ParametersControlPanel)
```

A..3 Properties

```
from . import proc_city_properties
from . import evaluate_system_properties
from . import proc_system_properties


def register():
    proc_city_properties.register()
    evaluate_system_properties.register()
    proc_system_properties.register()


def unregister():
    proc_city_properties.unregister()
    evaluate_system_properties.unregister()
    proc_system_properties.unregister()


import bpy


class EvaluateProperties(bpy.types.PropertyGroup):
    class1: bpy.props.StringProperty(options={'HIDDEN'})
    class2: bpy.props.StringProperty(options={'HIDDEN'})
    class3: bpy.props.StringProperty(options={'HIDDEN'})
    actual_prediction: bpy.props.StringProperty(options={'HIDDEN'})






def register():
    bpy.utils.register_class(EvaluateProperties)
    bpy.types.Scene.evaluate_properties = bpy.props.PointerProperty(type=EvaluateProperties)



def unregister():
    bpy.utils.unregister_class(EvaluateProperties)


import bpy
import random



class ConditionsProperties(bpy.types.PropertyGroup):

    def get_population(self):
        return self.get("population", 0)

    def get_wealth(self):
        return self.get("wealth", 0)

    def get_transportation(self):
        return self.get("transportation", 0)

    def get_environment(self):
        return self.get("environment", 0)


    def set_population(self, value):
        old_value = self.get("population", 0)
        self["population"] = value
        diff = value - old_value
        self.test = f"{diff}"

        target_parameters = ["height"]
        target_collections = ["Residential Buildings"]
        object_list = get_objects(target_collections)
        object_list = get_random_objects(object_list, diff)
        modify_nodes(object_list, target_parameters, diff)


    def set_wealth(self, value):
        old_value = self.get("wealth", 0)
        self["wealth"] = value
        diff = value - old_value
        self.test = f"{diff}"
```

```
        target_parameters = ["height", "length"]
        target_collections = ["Residential Buildings"]
        object_list = get_objects(target_collections)
        object_list = get_random_objects(object_list, diff)
        modify_nodes(object_list, target_parameters, diff)


    def set_transportation(self, value):
        old_value = self.get("transportation", 0)
        self["transportation"] = value
        diff = value - old_value
        self.test = f"{diff}"

        target_parameters = ["Count"]
        target_collections = ["Roads"]
        object_list = get_objects(target_collections)
        object_list = get_random_objects(object_list, diff)
        modify_nodes(object_list, target_parameters, diff)



    def set_environment(self, value):
        old_value = self.get("environment", 0)
        self["environment"] = value
        diff = value - old_value
        self.test = f"{diff}"

        target_parameters = ["tree_height", "branch_1_count", "branch_2_count", "leaves_count"]
        target_collections = ["Trees"]
        object_list = get_objects(target_collections)
        object_list = get_random_objects(object_list, diff)
        modify_nodes(object_list, target_parameters, diff)



    population: bpy.props.IntProperty(get=get_population, set=set_population)
    wealth: bpy.props.IntProperty(get=get_wealth, set=set_wealth)
    transportation: bpy.props.IntProperty(get=get_transportation, set=set_transportation)
    environment: bpy.props.IntProperty(get=get_environment, set=set_environment)
    test: bpy.props.StringProperty()



class BuildingsProperties(bpy.types.PropertyGroup):

    def get_min_height(self):
        return self.get("min_height", 0)

    def get_min_length(self):
        return self.get("min_length", 0)

    def get_min_width(self):
        return self.get("min_width", 0)


    def get_max_height(self):
        return self.get("max_height", 0)

    def get_max_length(self):
        return self.get("max_length", 0)

    def get_max_width(self):
        return self.get("max_width", 0)



    def set_min_height(self, value):
        minimum = "min_height"
        maximum = "max_height"
        target_collections = ["Residential Buildings"]
        property_name = "height"
        set_minimum(self, value, minimum, maximum, target_collections, property_name)

    def set_min_length(self, value):
        minimum = "min_length"
        maximum = "max_length"
        target_collections = ["Residential Buildings"]
        property_name = "length"
        set_minimum(self, value, minimum, maximum, target_collections, property_name)

    def set_min_width(self, value):
        minimum = "min_width"
        maximum = "max_width"
        target_collections = ["Residential Buildings"]
        property_name = "width"
        set_minimum(self, value, minimum, maximum, target_collections, property_name)


    def set_max_height(self, value):
        minimum = "min_height"
        maximum = "max_height"
        target_collections = ["Residential Buildings"]
        property_name = "height"
        set_maximum(self, value, minimum, maximum, target_collections, property_name)

    def set_max_length(self, value):
        minimum = "min_length"
        maximum = "max_length"
```

```
        target_collections = ["Residential Buildings"]
        property_name = "length"
        set_maximum(self, value, minimum, maximum, target_collections, property_name)

    def set_max_width(self, value):
        minimum = "min_width"
        maximum = "max_width"
        target_collections = ["Residential Buildings"]
        property_name = "width"
        set_maximum(self, value, minimum, maximum, target_collections, property_name)




    min_height: bpy.props.IntProperty(get=get_min_height,set=set_min_height)
    min_length: bpy.props.IntProperty(get=get_min_length,set=set_min_length)
    min_width: bpy.props.IntProperty(get=get_min_width,set=set_min_width)

    max_height: bpy.props.IntProperty(get=get_max_height, set=set_max_height)
    max_length: bpy.props.IntProperty(get=get_max_length, set=set_max_length)
    max_width: bpy.props.IntProperty(get=get_max_width, set=set_max_width)



class RoadsProperties(bpy.types.PropertyGroup):

    def get_min_width_count(self):
        return self.get("min_width_count", 0)

    def get_min_length_count(self):
        return self.get("min_length_count", 0)

    def get_min_width_spacing(self):
        return self.get("min_width_spacing", 0)

    def get_min_length_spacing(self):
        return self.get("min_length_spacing", 0)


    def get_max_width_count(self):
        return self.get("max_width_count", 0)

    def get_max_length_count(self):
        return self.get("max_length_count", 0)

    def get_max_width_spacing(self):
        return self.get("max_width_spacing", 0)

    def get_max_length_spacing(self):
        return self.get("max_length_spacing", 0)



    def set_min_width_count(self, value):
        minimum = "min_width_count"
        maximum = "max_width_count"
        target_collections = ["Roads"]
        property_name = "X_Count"
        set_minimum(self, value, minimum, maximum, target_collections, property_name)


    def set_min_length_count(self, value):
        minimum = "min_length_count"
        maximum = "max_length_count"
        target_collections = ["Roads"]
        property_name = "Y_Count"
        set_minimum(self, value, minimum, maximum, target_collections, property_name)


    def set_min_width_spacing(self, value):
        minimum = "min_width_spacing"
        maximum = "max_width_spacing"
        target_collections = ["Roads"]
        property_name = "X_Spacing"
        set_minimum(self, value, minimum, maximum, target_collections, property_name)


    def set_min_length_spacing(self, value):
        minimum = "min_length_spacing"
        maximum = "max_length_spacing"
        target_collections = ["Roads"]
        property_name = "Y_Spacing"
        set_minimum(self, value, minimum, maximum, target_collections, property_name)


    def set_max_width_count(self, value):
        minimum = "min_width_count"
        maximum = "max_width_count"
        target_collections = ["Roads"]
        property_name = "X_Count"
        set_maximum(self, value, minimum, maximum, target_collections, property_name)


    def set_max_length_count(self, value):
        minimum = "min_length_count"
        maximum = "max_length_count"
```

```
        target_collections = ["Roads"]
        property_name = "Y_Count"
        set_maximum(self, value, minimum, maximum, target_collections, property_name)


    def set_max_width_spacing(self, value):
        minimum = "min_width_spacing"
        maximum = "max_width_spacing"
        target_collections = ["Roads"]
        property_name = "X_Spacing"
        set_maximum(self, value, minimum, maximum, target_collections, property_name)


    def set_max_length_spacing(self, value):
        minimum = "min_length_spacing"
        maximum = "max_length_spacing"
        target_collections = ["Roads"]
        property_name = "Y_Spacing"
        set_maximum(self, value, minimum, maximum, target_collections, property_name)



    min_width_count: bpy.props.IntProperty(get=get_min_width_count,set=set_min_width_count)
    min_length_count: bpy.props.IntProperty(get=get_min_length_count,set=set_min_length_count)
    min_width_spacing: bpy.props.IntProperty(get=get_min_width_spacing,set=set_min_width_spacing)
    min_length_spacing: bpy.props.IntProperty(get=get_min_length_spacing,set=set_min_length_spacing)


    max_width_count: bpy.props.IntProperty(get=get_max_width_count,set=set_max_width_count)
    max_length_count: bpy.props.IntProperty(get=get_max_length_count,set=set_max_length_count)
    max_width_spacing: bpy.props.IntProperty(get=get_max_width_spacing,set=set_max_width_spacing)
    max_length_spacing: bpy.props.IntProperty(get=get_max_length_spacing,set=set_max_length_spacing)



class TreesProperties(bpy.types.PropertyGroup):

    def get_min_tree_height(self):
        return self.get("min_tree_height", 0)

    def get_min_branches_count(self):
        return self.get("min_branches_count", 0)

    def get_min_branches_length(self):
        return self.get("min_branches_length", 0)

    def get_min_leaves_count(self):
        return self.get("min_leaves_count", 0)


    def get_max_tree_height(self):
        return self.get("max_tree_height", 0)

    def get_max_branches_count(self):
        return self.get("max_branches_count", 0)

    def get_max_branches_length(self):
        return self.get("max_branches_length", 0)

    def get_max_leaves_count(self):
        return self.get("max_leaves_count", 0)




    def set_min_tree_height(self, value):
        minimum = "min_tree_height"
        maximum = "max_tree_height"
        min = 1
        max = 1000
        target_collections = ["Trees"]
        property_name = "tree_height"
        set_minimum(self, value, minimum, maximum, target_collections, property_name, min, max)


    def set_min_branches_count(self, value):
        minimum = "min_branches_count"
        maximum = "max_branches_count"
        min = 1
        max = 1000
        target_collections = ["Trees"]
        property_name = "branch_1_count"
        set_minimum(self, value, minimum, maximum, target_collections, property_name, min, max)
        property_name = "branch_2_count"
        set_minimum(self, value, minimum, maximum, target_collections, property_name, min, max)


    def set_min_branches_length(self, value):
        minimum = "min_branches_length"
        maximum = "max_branches_length"
        min = 1
        max = 1000
        target_collections = ["Trees"]
        property_name = "branches_length"
        set_minimum(self, value, minimum, maximum, target_collections, property_name, min, max)


    def set_min_leaves_count(self, value):
        minimum = "min_leaves_count"
```

```
        maximum = "max_leaves_count"
        min = 0
        max = 1000
        target_collections = ["Trees"]
        property_name = "leaves_count"
        set_minimum(self, value, minimum, maximum, target_collections, property_name, min, max)


    def set_max_tree_height(self, value):
        minimum = "min_tree_height"
        maximum = "max_tree_height"
        min = 1
        max = 1000
        target_collections = ["Trees"]
        property_name = "tree_height"
        set_maximum(self, value, minimum, maximum, target_collections, property_name, min, max)


    def set_max_branches_count(self, value):
        minimum = "min_branches_count"
        maximum = "max_branches_count"
        min = 1
        max = 1000
        target_collections = ["Trees"]
        property_name = "branch_1_count"
        set_maximum(self, value, minimum, maximum, target_collections, property_name, min, max)
        property_name = "branch_2_count"
        set_maximum(self, value, minimum, maximum, target_collections, property_name, min, max)


    def set_max_branches_length(self, value):
        minimum = "min_branches_length"
        maximum = "max_branches_length"
        min = 1
        max = 1000
        target_collections = ["Trees"]
        property_name = "branches_length"
        set_maximum(self, value, minimum, maximum, target_collections, property_name, min, max)


    def set_max_leaves_count(self, value):
        minimum = "min_leaves_count"
        maximum = "max_leaves_count"
        min = 0
        max = 1000
        target_collections = ["Trees"]
        property_name = "leaves_count"
        set_maximum(self, value, minimum, maximum, target_collections, property_name, min, max)




    min_tree_height: bpy.props.IntProperty(get=get_min_tree_height,set=set_min_tree_height)
    min_branches_count: bpy.props.IntProperty(get=get_min_branches_count,set=set_min_branches_count)
    min_branches_length: bpy.props.IntProperty(get=get_min_branches_length,set=set_min_branches_length)
    min_leaves_count: bpy.props.IntProperty(get=get_min_leaves_count,set=set_min_leaves_count)


    max_tree_height: bpy.props.IntProperty(get=get_max_tree_height,set=set_max_tree_height)
    max_branches_count: bpy.props.IntProperty(get=get_max_branches_count,set=set_max_branches_count)
    max_branches_length: bpy.props.IntProperty(get=get_max_branches_length,set=set_max_branches_length)
    max_leaves_count: bpy.props.IntProperty(get=get_max_leaves_count,set=set_max_leaves_count)






# Utility Functions for updates

def scale_object(self, context):
        # Get the active object
        obj = context.active_object


        # Calculate the new dimensions based on the population
        dimension_factor = 1.01 # Scale by 10 for every population unit
        new_dimensions = obj.dimensions * dimension_factor

        # Resize the object to the new dimensions
        bpy.ops.transform.resize(value=new_dimensions)



def get_objects(target_collections):
    object_list = []

    for collection_name in target_collections:
        collection = bpy.data.collections.get(collection_name)
        if collection is not None:
            for obj in collection.objects:
                if obj.type == 'MESH':
```

```
                    object_list.append(obj)

    return object_list


def get_random_objects(object_list, quantity):
    randomized_list = []

    for _ in range(abs(quantity)):
        random_object = random.choice(object_list)
        randomized_list.append(random_object)

    return randomized_list


def get_objects_nodes(object_list, target_parameters):
    property_nodes = []
    for obj in object_list:
        if "GeometryNodes" in obj.modifiers:
            geo_tree = obj.modifiers["GeometryNodes"].node_group

            # Extracting properties
            for node in geo_tree.nodes:
                if any(param in node.name for param in target_parameters):
                    property_nodes.append(node)

    return property_nodes



def modify_nodes(object_list, target_parameters, diff):
    nodes = get_objects_nodes(object_list, target_parameters)

    # modifying node values
    for node in nodes:
        node_value = 0
        if hasattr(node, 'integer'):
            node_value = node.integer
        elif len(node.outputs) > 0:
            try:
                node_value = node.outputs[0].default_value
            except:
                pass

        result = node_value + diff
        # Reached maximum
        if(node_value > 12):
            break
        # Reached minimum
        if(result < 1):
            result = 1
        if(result > 12):
            result = 12 - 1
        node_value = result

        # Assigning values
        if hasattr(node, 'integer'):
            node.integer = node_value
        elif len(node.outputs) > 0:
            try:
                node.outputs[0].default_value = node_value
            except:
                pass



def modify_max_property(object_list, max_height, property_name):
    nodes = get_objects_nodes(object_list, [property_name])

    #modifying node values
    for node in nodes:
        # Exceeds max_height
        node_value = 0
        if hasattr(node, 'integer'):
            node_value = node.integer
        elif len(node.outputs) > 0:
            try:
                node_value = node.outputs[0].default_value
            except:
                pass

        if(node_value > max_height):
            node_value = max_height

        # Assigning values
        if hasattr(node, 'integer'):
            node.integer = node_value
        elif len(node.outputs) > 0:
            try:
                node.outputs[0].default_value = node_value
            except:
                pass



def modify_min_property(object_list, min_height, property_name):
    nodes = get_objects_nodes(object_list, [property_name])
```

```
    #modifying node values
    for node in nodes:
        # Exceeds min_height
        node_value = 0
        if hasattr(node, 'integer'):
            node_value = node.integer
        elif len(node.outputs) > 0:
            try:
                node_value = node.outputs[0].default_value
            except:
                pass

        if(node_value < min_height):
            node_value = min_height

        # Assigning values
        if hasattr(node, 'integer'):
            node.integer = node_value
        elif len(node.outputs) > 0:
            try:
                node.outputs[0].default_value = node_value
            except:
                pass



def set_minimum(self, value, minimum, maximum, target_collections, property_name, min=2, max=12):
    MIN_HEIGHT = min
    MAX_HEIGHT = max
    old_value = self.get(minimum, 0)
    self[minimum] = value

    min_height = self[minimum]
    max_height = self[maximum]

    if(min_height <= MIN_HEIGHT):
        self[minimum] = MIN_HEIGHT

    if(min_height >= max_height):
        if(max_height <= MAX_HEIGHT - 1):
            self[minimum] = max_height
            self[maximum] = max_height + 1

    if(min_height >= MAX_HEIGHT):
        self[minimum] = MAX_HEIGHT - 1

    diff = value - old_value

    object_list = get_objects(target_collections)
    modify_min_property(object_list, min_height, property_name)



def set_maximum(self, value, minimum, maximum, target_collections, property_name,min=2, max=12):
    MIN_HEIGHT = min
    MAX_HEIGHT = max
    old_value = self.get(maximum, 0)
    self[maximum] = value

    min_height = self[minimum]
    max_height = self[maximum]

    if(max_height <= MIN_HEIGHT):
        self[maximum] = MIN_HEIGHT

    if(max_height <= min_height):
        if(min_height >= MIN_HEIGHT):
            self[maximum] = min_height
            self[minimum] = min_height - 1

    if(max_height >= MAX_HEIGHT):
        self[maximum] = MAX_HEIGHT

    diff = value - old_value

    object_list = get_objects(target_collections)
    max_height = self[maximum]
    modify_max_property(object_list, max_height, property_name)




def register():
    bpy.utils.register_class(BuildingsProperties)
    bpy.types.Scene.buildings_properties = bpy.props.PointerProperty(type=BuildingsProperties)
    bpy.utils.register_class(ConditionsProperties)
    bpy.types.Scene.conditions_properties = bpy.props.PointerProperty(type=ConditionsProperties)
    bpy.utils.register_class(TreesProperties)
    bpy.types.Scene.trees_properties = bpy.props.PointerProperty(type=TreesProperties)
    bpy.utils.register_class(RoadsProperties)
    bpy.types.Scene.roads_properties = bpy.props.PointerProperty(type=RoadsProperties)


def unregister():
```

```
bpy.utils.unregister_class(BuildingsProperties)
bpy.utils.unregister_class(ConditionsProperties)
bpy.utils.unregister_class(TreesProperties)
bpy.utils.unregister_class(RoadsProperties)
```

XI. Acknowledgment

I would like to express my deepest gratitude to the **University of the Philippines Manila** for providing invaluable resources, unwavering support, and guidance throughout my journey. The university's esteemed faculty members, outstanding facilities, and enriching academic environment have played a crucial role in the successful completion of my Special Problem and my degree. I am truly grateful for their contributions.

My heartfelt appreciation goes to my SP advisor, **Ma. Sheila A. Magboo**, for her exceptional guidance and invaluable insights throughout this transformative journey. Her expertise, mentorship, and dedication have shaped the direction and execution of my Special Problem, and I am grateful for her belief in my abilities and commitment to my growth.

I also extend my deepest gratitude to the faculty members of the **Department of Physical Sciences and Mathematics (DPSM)** for their knowledge, encouragement, and contributions to my intellectual development. Their insightful feedback, constructive criticisms, and engaging discussions have expanded my understanding of the subject matter and refined my research.

I am immensely thankful to the **Department of Science and Technology (DOST)** for their unwavering support and financial aid as their scholar throughout my college journey. Their assistance has been instrumental in enabling me to pursue my studies and achieve my academic goals. Without their generous support, it would not have been possible for me to continue my education and embark on this beautiful journey. I am truly grateful for the opportunities they have provided me, which have played a crucial role in shaping my intellectual and personal growth. I am forever indebted to them and to the Filipino people for their investment in my education.

I am also immensely grateful for the camaraderie and unwavering support from my blockmates, who have been by my side throughout this memorable experience. A special thanks goes to **Romaine Dara M. Regala** for being my dedicated ac-

countability partner during the entire duration of my Special Problem. Additionally, I would like to express my gratitude to **Julius Allen A. Reyes**, a blockmate whom I have admired since our first year. It is through this admiration that I discovered the fascinating realm of game development and computer graphics, which ultimately became the foundation of my Special Problem. This realization also helped me identify the field that resonates with my interests and strengths the most.

I would also like to express my deepest gratitude to my dear friends, **Kyle Mari Angelo M. Aquino** and **Romwell Joackin O. Santos**. Throughout my college journey, they have been my constant support network, accompanying me through the highs and lows. Their friendship has been instrumental in my personal growth and well-being, and I am truly grateful for their unwavering support, encouragement, and camaraderie. In the most challenging times, they have provided me with strength, motivation, and a shoulder to lean on. Their contribution to my success in this endeavor cannot be overstated.

Furthermore, I want to express my deepest gratitude to my beloved parents, **Emmanuel M. Santos** and **Rosella P. Santos**, as well as my dear brothers, **Aldrin Karl P. Santos** and **Az Kaiser P. Santos**. Their unwavering love, support, encouragement, and understanding have been a constant source of inspiration and motivation throughout my journey. Their belief in my abilities and their enduring patience during challenging times have been the guiding lights that have propelled me towards success. I am truly grateful for their unwavering presence in my life and for being the pillars of strength that I can always rely on.

To all the individuals mentioned above, I am deeply grateful for your unwavering support, guidance, and encouragement throughout my Special Problem and my degree. Your contributions have shaped my academic and personal growth, and I feel honored to have had the privilege of being with such exceptional individuals.

Thank you.