UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

An API For Building Artificial Worlds For Machine Learning Using Blender

Vasco Duarte Ribeiro de Caires Calheiros Cruz

Mestrado em Informática

Trabalho de Projeto orientado por:
Thibault Nicolas Langlois

2022

Acknowledgments

Here I would like to direct a few words as a way of thanking those involved, otherwise, it would have not been possible to finish this endeavour.
To my coordinator, Professor Thibault Nicolas Langlois, I can only begin to express my gratitude as his guidance was all but essential in this long and arduous process, his ability to pinpoint the areas of focus and better judgement were key throughout the development.
To my family and friends as without them I am nothing and a lot of the emotional weight was carried by them. In the difficult times they were there helping me and giving me strength so I could push on and finish this project.

Many thanks to all involved in this endeavour,
I could not have done this without you.

Abstract

Virtual technologies are a facet of advancement that permeates almost all major fields in our world, from physics simulations to videogames and movies, urban design and engineering to rocket science, all make use of this wonderful and powerful tool. One such field is Machine Learning. This project aims to leverage said virtual technologies to help such a field in the shape of an object identification model being developed by Thibault Nicolas Langlois.

This is done by using a tool such as Blender and a language such as Python to create an API that leverages both in order to facilitate the procedural generation of virtual worlds. By generating said worlds and taking snapshots of them with the rendering power of Blender, this project aims to help generate vast amounts of data that can be catalogued and sent to feed said models.

To do this, the API was constructed in a way that allows for modular adaptation to whatever purpose it is to be used for, all while containing an example focused on city generation. This can help guide others intending on using the API, on how to adapt it to their use case.

This proved to be extremely complex and arduous, requiring many hours of work and lines of code written. The scale of the endeavour was vast and required a lot to reach the state it currently presents itself in. It also presented how capable and versatile the technology can be, giving many a tool to the user's disposal, as well as the developer.

This proves the path forward for testing many aspects of products or experiments is indeed virtualization as it has shown to be capable enough in providing the tools necessary to emulate even the most complex scenarios, all while potentially saving numerous amounts of investment and time, depending on the scale of the products taking advantage of it.

Keywords: Blender, Python, Procedural Generation, Scripting, Virtual Worlds

Resumo

Tecnologias virtuais compreendem uma componente de avanço tecnológico que permeia quaste todas as grandes áreas de estudo pelo mundo for a, de simulações físicas a jogos e films, de design e engenharia urbanística a engenharia aeroespacial, todas têm casos de uso para esta magnífica e poderosa ferramenta. A sua influência é de tal ordem vasta que algumas das maiores empresas do mundo aplicam nos dias de hoje enormíssimas quantias de dinheiro e recursos específicamente para este mesmo propósito, tal como é possível observar com o Omniverse da NVIDIA e o Metaverse da Meta. A espectativa é de crescimento acelerado para esta e futuras novas tecnologias a serem desenvolvidas por influência direta de exemplos como estes. Uma tal área de estudo é a Aprendizagem Automática.
Este projeto tem o objetivo de fazer uso de ditas tecnologias virtuais para ajudar a área de Aprendizagem Automática, mais específicamente, um modelo de identificação de objetos que se encontra em desenvolvimento por Thibault Nicolas Langlois, através da criação de uma API (IAP – Interface de Aplicação de Programação) em Python para o Blender. Modelos e aplicações de Aprendizagem Automática deste género possuem uma característica muito específica quanto ao modo de treino do produto, isto é, requerem vastas quantias de dados, informação, corretamente legendada. Isto permite ao modelo reconhecer objetos e legendá-los corretamente, pois já observou enormes números de exemplos semelhantes com a mesma nomenclatura, ou seja, aprender por associação e reconhecimento. O problema que se põe deve-se a esse mesmo fator pois, ou a informação corretamente legendada apresenta custos elevados, ou é simplesmente muito difícil de encontrar, se não mesmo impossível.
É aqui que as tecnologias virtuais e, mais expecíficamente, este projeto podem revolucionar este meio. Através da simulação da vida real, é possível replicar as condições necessárias para capturer informação próxima o suficiente da esperada de modo a treinar o modelo. Nos últimos anos tem-se observado uma evolução drástica na capacidade de simular componentes, tanto em termos de escala, como também de qualidade, definição. De tal modo, torna-se possível simular o mundo real com tamanha precisão e qualidade, que informação obtida de meios virtuais consegue servir o propósito de treinar tais modelos e estes apresentarem resultados positivos quando postos em prática com cenários comerciais. O mundo virtual é de tal modo semelhante ao real que se torna capaz de o substituir.
Isto tem a benece de poder obter informação diretamente de computadores a correrem tais simulações e, como pode ser observado em outras áreas, computadores são excelentes a produzir informação em vastas escalas.
Esse é o foco deste trabalho, produzir informação suficientemente detalhada a escalas antes impossíveis de realizar, tudo num espaço de tempo muitíssimo inferior ao que levaria fazer o mesmo processo manualmente. Também resolve o problema de custo pois, com a informação a ser produzida a largas escalas, o custo da mesma desce, tornando a área de estudo mais acessível para todos.
De modo a conseguir alcançar tais objetivos, duas tecnologias foram essenciais: Blender e Python.
Python é uma linguagem de programação de alto nível que suporta design orientado a objetos e é bastante versátil e poderosa quanto às suas capacidades. Blender é um software de modelação 3D e 2D que permite gerar modelar, texturizar, renderizar, animar, ou aplicar compozições num ambiente virtual. Através da utilização destas duas ferramentas, é possível criar um produto capaz de simular mundos virtuais e retirar imagens renderizadas do mesmo para fins de alimentar modelos de Aprendizagem Automática. Para simplificar a sua utilização, o paradigma de programação orientada a objetos facilita a partilha de informação pelo código, associando a mesma a “objetos”, algo que diminui bastante a quantidade de linhas de código necessárias para executar qualquer ação pois, grande parte do processo fica relegado para as components ou classes base presents na API, com as quais os utilizadores não interagem. Isto serve de modo a facilitar o uso deste projeto sem ter de conhecer o Blender e as suas bibliotecas Python, como por exemplo, a “bpy”.

Para segmentar as bases da API com o objetivo de simplificar o projeto, foram criadas 4 classes principais, das quais todo o resto do projeto vai depender: mesh, curve, camera e light.
Estas servem de base para o projeto, contendo as operações necessárias para efetuar as ações requeridas. Também contêm todas as chamadas a "bpy" pois interação direta com o Blender não é simples e depende de vários aspetos, como context, para não produzirem erros e, de modo a exigir conhecimento profundo do programa a utilizadores, estas são reservadas para a API, tornando a sua utilização mais fácil e simplificada.
Foram também criadas subclasses simples com ações básicas, como por exemplo, a criação de cubos ou cones. Subclasses são classes que surgem, derivam das classes base e, no processo, herdam todas as funções e operações das mesmas. Se a classe mesh contém a operação "translate" ou "mover", então, a classe cube que é subclasse de mesh, também a pode utilizar, o que gera a possibilidade de usar as classes base através de objetos das suas subclasses, extendendo o seu alcance e impacto perante todo o projeto.
Por fim, classes mais complexas foram criadas para o propósito mais específico deste trabalho. De modo a gerar uma cidade virtual, as classes house e road têm a finalidade de gerar os aspetos básicos que se esperam num ambiente urbano. Utilizando classes simples como cube ou bezier e tirando proveito de todas as operações oriundas das classes base, torna-se possível gerar cidades virtuais com um nível de detalhe aceitável. De modo a popular a mesma, operações de importação de objetos feitos em antemão por outros criadores tornou-se essencial, pois a geração procedimental, ou aleatória não foi atingível com o tempo e recursos disponíveis. Outros elementos, como a texturização e iluminação são levados a cabo por operações presents nas classes base como a mesh e a light.
Apesar de todos os sistemas necessários estarem em funcionamento, existem situações ainda a resolver, especialmente em termos de correção de erros ou melhorias em áreas de detalhe ou de comportamento esperado. Problemas com interseções entre objetos ou fraco posicionamento dos mesmos são apectos a melhorar.
Isto para demonstrar como o nível de trabalho e dedicação necessários, até para trabalhos de menor escala como este, são elevadíssimos.
Tal como foi referido, este tipo de trabalho não tem um fim, é sempre possível melhorar certos aspetos, criar novos cenários, inovar. A chave necessária para terminar este projeto foi a capacidade do meu coordenador e minha de definer limites atingíveis para o contexto deste trabalho.
A retirar deste esforço, é a demonstração de que tecnologias como esta são realmente capazes de simular mundos e sistemas reais com grande efficácia e detalhe e que o seu futuro poderá trazer muita inovação em múltiplos setores de desenvolvimento tecnológico, como também uma redução enorme de custos de desenvolvimento e testes. Um exemplo mais concreto pode ser observado em áreas como o desenvolvimento de veículos podem tirar proveito de testar sistemas de condução automática em ambientes virtuais antes de fabricar protótipos para continuar a testagem em ambiente real. Isto é algo que não só disponibiliza muito mais controlo sobre os testes a executar, como também poupa muito investimento em prototipagem e testes, pois requere um processo menor de testagem em ambiente real.

Palavras-chave: Blender, Python, Procedural Generation, Scripting, Virtual Worlds

Index

Abstract i

Resumo ii

Index v

Figure List vi

1. **Introduction** **1**
   - 1.1. Background 2
     - 1.1.1. Blender 2
   - 1.2. Motivation 3
   - 1.3. Objectives 3

2. Related Work **4**

3. Design Choices **7**

4. Implementation **10**
   - 4.1. Base Classes 10
     - 4.1.1. Mesh 10
     - 4.1.2. Curve 12
     - 4.1.3. Camera 14
     - 4.1.4. Light 15
   - 4.2. Simple Subclasses 16
   - 4.3. Complex Subclasses 18
     - 4.3.1. Road Class 18
     - 4.3.2. House Class 21
   - 4.4. Culmination of work 24
   - 4.5. Extra Work 29

5. Future Work **30**

6. Conclusion **31**

7. Bibliography **32**

Figure List

Figure 2.1: Example of the implementation of the Growth Rule 4

Figure 2.2: Example of the implementation of the Population Density 5

Figure 2.3: Example of an image of the result of combining Growth Rule and Population Density 5

Figure 2.4: Example of the node-based workflow used by Scene City 6

Figure 3.1: Example of initial city generation results 7

Figure 4.1: Example of object creation and object-oriented code implementation and use 12

Figure 4.2: Material application function designed to function in an object-oriented aspect 12

Figure 4.3: Example of handle transformation to same curve 13

Figure 4.4: Handle transformation function set to automatic type 13

Figure 4.5: Example of camera object representation in Blender 14

Figure 4.6: Render function callable by camera class object 14

Figure 4.7: Function responsible for creating sky texture and applying it to the world's background 15

Figure 4.8: Example of sky texture value optimization for clear sky scenario 15

Figure 4.9: Two examples of changing Sun-type object positions influencing lighting conditions 16

Figure 4.10: Cube subclass code demonstration 17

Figure 4.11: Array modifier applicating function 18

Figure 4.12: Curve positioning modifier applicating function 18

Figure 4.13: Code representing road generation and texturing 19

Figure 4.14: Sample of untextured road generation 20

Figure 4.15: Sample of textured road generation 20

Figure 4.16: Intersect function present in the mesh class 21

Figure 4.17: Section of the selection process for imports in building generation 22

Figure 4.18: Part of the coded method used to calculate mesh clone number for building generation 22

Figure 4.19: Example of result of building generation 23

Figure 4.20: Building class object creation with randomized 23

Figure 4.21: Blender UI for material and texture application and editing 24

Figure 4.22: File imports in StreetGeneration.py that allow to generate the virtual world 25

Figure 4.23: Cleaning methods for the world scene (left) and the sky texture (right) 25

Figure 4.24: UV unwrapping function used on meshes to alter texture scale 26

Figure 4.25: Example of building placement particle system 27

Figure 4.26: Remesh process and function to be applied to every floor object 28

Figure 4.27: Example of particle system at work for world population with person-like objects 28

Figure 4.28: Example of camera traveling through path created for render purposes 29

Chapter 1

Introduction

Due to advancements in hardware and software, virtual world generation has gotten to a point where the level of detail can fool even humans. This opens the possibility to use virtual worlds to train object identification models that are then used in the real world. The project aims to execute on that possibility. To fully generate a virtual world and do it in a procedural fashion, first, the hard boundaries must be determined, such as size and level of detail. Afterwards, the procedural generation elements factor in, such as position, orientation, size, and type of objects to be generated. There are also the considerations regarding operation cost and performance, optimization.
This project may be viewed as a tool for various situations and other projects like video games, movies, data analysis, city planning and management and architectural endeavours. Instead of mandating a manual hand-made generation of every single individual mesh, as well as their respective positioning and orientation, this approach frees up a lot of time that would have been spent on the large-scale endeavours, allowing for said time to be spent on more detail-driven work or more specific modifications or applications.
The project engages with various elements of randomness. It wasn't done in a specifically different fashion from other projects that already exist and I've researched about. Certain elements are manually defined and delimited, but other come as a result of chance and randomness, as is intended. The overall randomness is present in all other projects, but the specifics of the implementation are what differs.
Some, however, take a different approach to the procedural generation of the city, specifically in the street and area or block generation.
Focusing on procedural generation means most of the work was dedicated to defining the boundaries of the random elements and making said elements work well together. The difficulty depends on the size of the world and on the number of random elements. Cohesion is a big part of the project.
To do this, each element comprising the overall objective was segmented into a module that encompasses exclusively its specific area.
The scope of such a project entails amounts of work not possible for a single student with the time attributed. This required a focusing of the goal, the level of detail and the features and techniques developed. The results of the focusing of the work done can be seen throughout the more basic elements of the project.
The idea is to develop the Python API in such a manner that all is done in a relativistic way, not an absolute one. This is done to accommodate the randomness of the asset generation within Blender. There is also the intention of executing it locally in one machine, "offline", instead of using servers and massive amounts of previously acquired data, in other words, the API generates the data instead of using other's data. This is not to say models made by other people aren't used in this project, but rather the objects are used locally.
This report will provide more context to the project and information regarding related work, as well as provide an insight into the methods used and the results of the project. There will also be a look into possible future work to add to what has already been achieved.

1.1 Background

Touching on the advancements previously mentioned, these have allowed for ever more detailed, realistic and animated scenarios. From video games to movies, all have felt the impact from said advances. One place which has not felt said improvements to the same degree though is research areas such as Machine Learning. Here, the specific type of technique relevant to the project is object recognition in images. The aspect most lacking for said research area is the lack of information that is correctly labelled, as the most widely used techniques rely on vast amounts of said labelled data. This means it is either very expensive or very difficult to use said techniques. Especially when referring to "real world" data as these pose an even bigger issue with identification given their higher complexity and overlapping with other data, something that makes model estimation harder.
For object identification models, practice makes perfect, meaning the more data is fed into model's training process, the more accurate they become. Real-world model training is expensive, and time-consuming so, through the use of virtual object generation, given the current state of software and hardware, it is possible to create virtual objects with levels of detail so close to their real-world counterparts that they can effectively replace them for the model training, meaning said training can be done in a much faster and much cheaper way, effectively giving everyone the ability to train their models instead of relying on expensive ones.

1.1.1 Blender

A great tool for 3D object generation and modification is Blender. This engine has gained a vast number of updates and improvements throughout all its iterations, to a point where it is considered one of the best in the world. Even though most associate this software to video game making and generalized 2D and 3D objects and models, in more recent times, its capabilities have grown and improved in such a way that it has started to be used in the movie industry, a place that has famously demanded so much detail in its products that it has depended in online editing tools and vast time and resources to generate every frame. This means the level of detail of engines such as Blender are now capable of generate is so high that they can be used for the purpose of this project and teach identification models with a good translation to real world scenarios.
The ability to use Python as the language for the API is a big bonus reason for choosing it for the project, seeing as Blender already possesses its own Python library of functions to be called upon to execute actions in it. The library is extensive, the documentation even more so and the problem solving found online with the community is very helpful.
Some examples of uses of Blender for movies, models, and games:

- Future City Pack (model 2019)[1]
- Hippo (model 2021)[2]
- Princess Ira and Amina (model 2021)[4]
- The Ancient Temple (game demo 2021)[5]
- Red Leaves (model 2021)[6]
- Rookies Container - Weekly drill (model 2021)[7]
- Noire (model 2021)[9]
- Sprite Fright (movie 2021)[11]
- Rocket League (game 2015)[12]
- Ubisoft (publisher)[13]

1.2 Motivation

To help advance and prosper the idea of using virtual technologies to simulate the real world in order to add more versability to the ability to test other technologies that would otherwise require a lot of work if done in the more manually intensive ways used today, this project was designed to help the testing of an object identification model developed by Thibault Nicolas Langlois.

This area shows immense potential and is finally seeing major investment by bigger companies, bringing even more eyes to the possibilities it brings. A great example of this can be seen in the latest NVIDIA presentation where their newest graphics cards were unveiled. There, a new virtual space for collaborations for work, simulation and testing was announced: The Omniverse.

In their words, Omniverse is said to function as a virtual hub where many technologies and products will be able to be simulated and tested by anyone, before bringing said products to the real world. This can save vast amounts of money as it removes a big part of investment into R&D made by companies when testing of the products is needed before releasing it for commercial use. This is especially true for scaling scenarios as products made for general use with a wide market can exponentially decrease cost of production for testing if the product can be tested before hand in a virtual environment.

1.3 Objectives

This project operates in a similar mindset as that of NVIDIA's Omniverse, and with the same objective, albeit with a much smaller scale. To help testing the object identification model designed by Thibault, instead of requiring someone to traverse a city taking pictures of objects, buildings, people and more, those objects of interest can be replicated in the virtual world and, by using a virtual camera, the photos can be acquired and sent to train the model, the same way. This saves great amounts of time and effort in the development of the product.

The main focus is on the representation of a virtual city by creating streets, blocks, and buildings, while also being able to receive other objects already made by other to help populate the virtual world and make it more believable. In order to make said city generation feel realistic, complex and diverse, the implementation of procedural generation was decided upon. This makes so every time a city is generated it does so in a different way, making different buildings with changing sizes, colours, and shapes, as an example.

The project is also designed in a way to allow others to use its base functionality to create more than just virtual cities. Basic tools are made available to anyone trying to create virtual objects, for whatever purpose intended, be it the creation of 3D objects for printing, room modelling and design, or videogame world creation. The possibilities are truly never ending.

Chapter 2

Related Work

The best way to understand where the world sits in this particular space is to research and review other approaches. Be it in study cases, products such as movies or games, or even in research spaces.
An example of a range of products that very effectively use and evolve the practice of procedural generation is the video game space and No Man's Sky is a great target to evaluate.
No Man's Sky[8] is a video game that uses procedural generation to generate a galaxy as a playable space where players can enter and leave planets and travel between star systems, having the main objective of reaching the centre of the galaxy. The journey itself represents the game more than the ending/main objective, as the players travel from planet to planet, star system to star system and collect resources, make new developments, both technological and cultural, interact with different species, learn new languages, trade, occupy land and establish bases. The player also has the ability to modify terrain by removing or adding to it. Such mechanics also make use of procedural generation, but work in a much smaller scale and a much smaller pool of possibilities.
The system that supports all of these aspects is the procedural generation of the worlds, the fauna, the flora, the technology, the weather, the context behind the galaxy itself. All of this is run on an in-house proprietary engine, so there is little known about the details of the implementation and how those compare to applications in open engines such as Unity or Blender using C#, C++ or Python languages.
Changing from product-based examples of procedural generation to generalized applications, we reach an open-source project called "Procedural City Generation in Python" by Jonathan Sauder or "josauder"[10] on GitHub. It's an application of Blender and Python that focuses on a few main aspects/methods to define the city generation: Growth Rule, Population Density, Vertex Management, Polygon Extraction/Subdivision and 3D Data Generation.
As is shown below in Figure 2.1, Growth Rule defines how the city grows depending on which colour is attributed to each area: blue - radial growth rule, red - grid growth rule, green - organic growth rule.

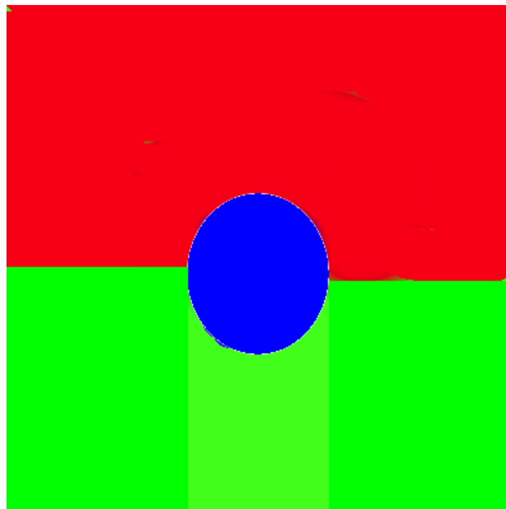

Figure 2.1 – Example of the implementation of the Growth Rule (sourced from: https://josauder.github.io/procedural_city_generation/#visualization-in-blender)

Population Density is determined by yet another image with a pattern defined by a dark side and a light side such as Figure 2.2, attributing a higher probability to generating and connecting edges. This, along with Growth Rule help design the city as seen in Figure 2.3.

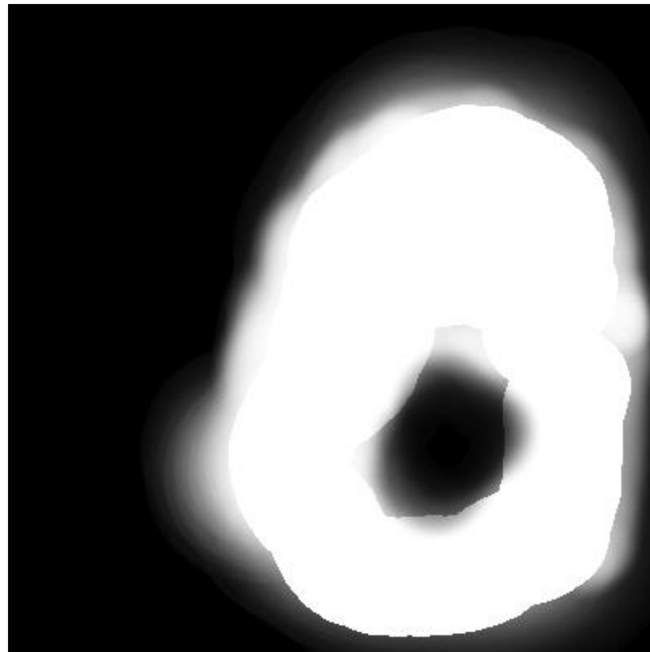

Figure 2.2 – Example of the implementation of the Population Density (sourced from: https://josauder.github.io/procedural_city_generation/#visualization-in-blender)

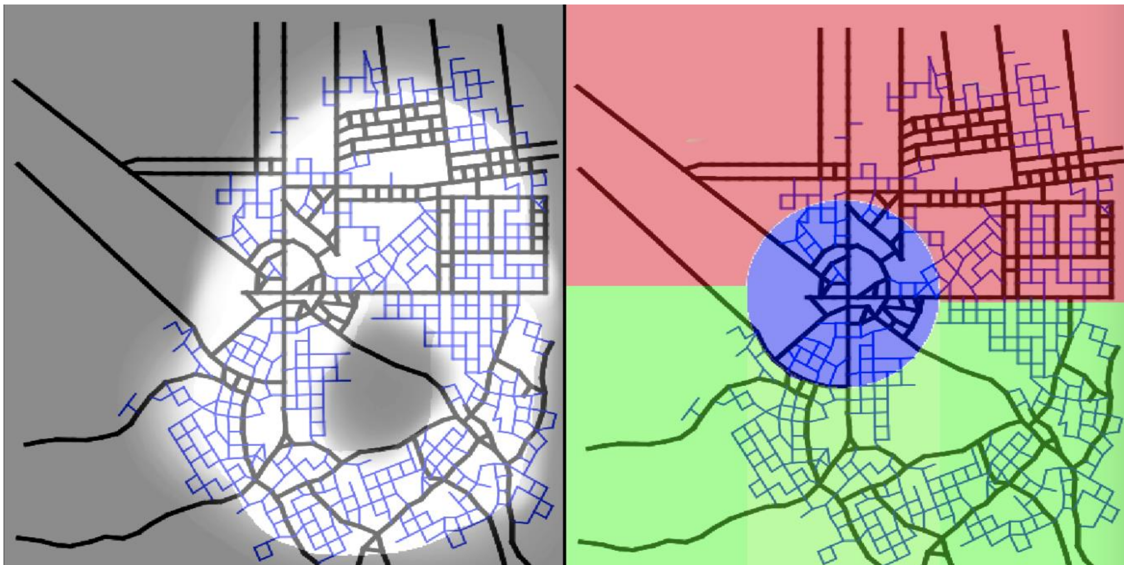

Figure 2.3 – Example of an image of the result of combining Growth Rule and Population Density (sourced from: https://josauder.github.io/procedural_city_generation/#visualization-in-blender)

To generate procedurally in Blender, there are options to help people without the ability to write their own code, such as SceneCity[3], a Blender add-on that adds the tools to procedurally generate cities. The basic organization of SceneCity is done in a node-based workflow, instead of a script-based one, as shown in Figure 2.4.

This can work to simplify the understanding process of how the add-on works and how to change and modify it. SceneCity not only contains the terrain manipulation, but also has the building, road and props generation embedded in it.
Some techniques present in their product include Layouts (Scatter or Grid), Building Generation, and Terrain Modification/Manipulation.
SceneCity also has a Unity add-on, meaning it is even more accessible to people who want such a tool but don't want to have to change engines. It presently costs 97 dollars in a one-time purchase to gain access to the software.

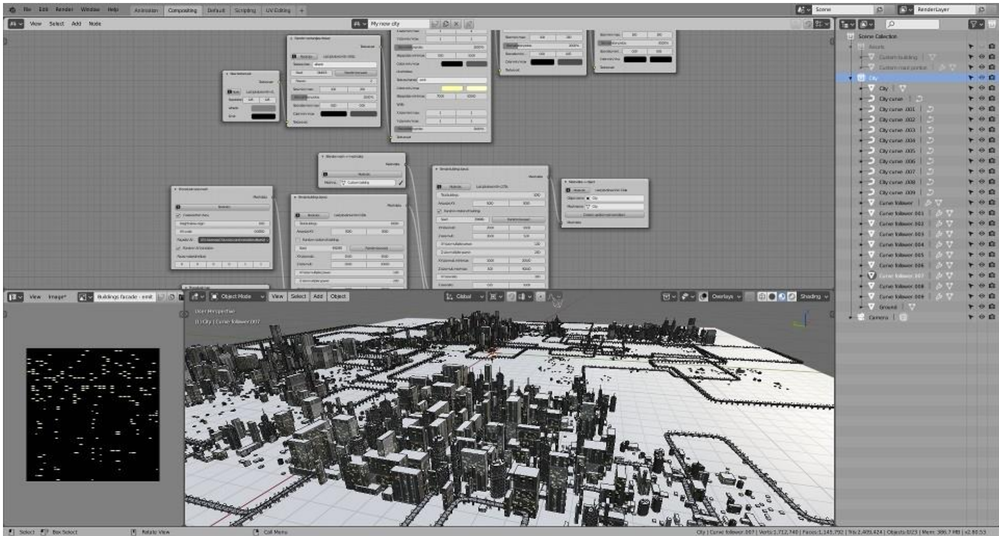

Figure 2.4 – Example of the node-based workflow used by Scene City (sourced from: http://www.cgchan.com)

Chapter 3

Design Choices

Diving into a more meticulous explanation of how the project itself works and how it was designed, we arrive at the methodology.
As mentioned before in the introduction, this project makes use of the Python programming language and the open source 2D and 3D creation tool Blender. Making use of the Blender/Python library, “bpy”, it is possible to engage in said creative endeavours solely through coding/scripting.
As for the scripting itself, the size of the project is the determining factor for the complexity of the script. In this case, since procedural generation is the main aspect being tackled, the complexity of the script is slightly higher, requiring several conditions in its approach in order to maintain the code well segmented, easier to read and change and, most of all, easier to run. This was a big factor that ended up helping decide to write the code as object-oriented, even though the inspiration for the project is heavily influenced by the OpenScad software, which is procedural, not object-oriented.
There were several iterations of the project before arriving at its current state.
Initially, the generation of the world was far more basic and limited, lacking in complexity, variety and, most importantly, detail. Paradoxically, the way to call upon the functions used to generate the world was much more complex and unintuitive than it is now. There were no base functions, no specific objects and classes to segment parts of the city generation code in an effort to clean up the presentation, and no complex functions that adapted complex behaviours to simple function calls.
As is demonstrated in Figure 3.1, the objects created, their positioning in the world, and the variety of objects was sorely lacking, the detail was non-existing and the adaptability and complexity were barebones.

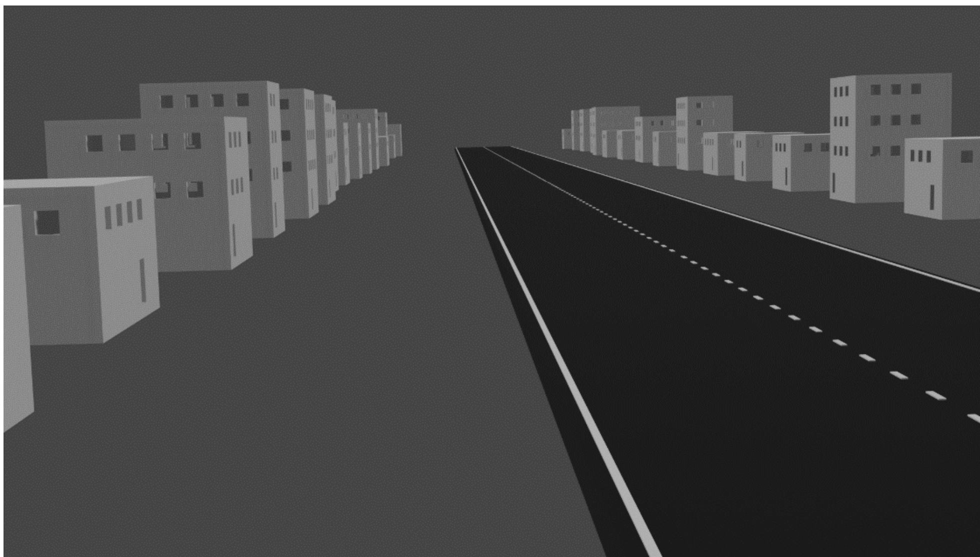

Figure 3.1 – Example of initial city generation results

Buildings were no more than hollow cubes with square shaped window and door holes, streets had no sidewalks and were completely straight, there was no population of objects, such as people, posts, or cars, no lighting, and no textures or colours.
Some of these base functions made it to the final product, but their implementation is much less prominent or noticeable as it was initially.

With the state of these initial iterations, the conclusion was simple: the design of the API was flawed. To rectify this, the design changed to reflect its users needs. Simpler, more flexible, more competent. Separating major components such as building creation or street generation into separate classes, creating base classes that operate throughout the API, such as mesh, curve, or light classes, hiding calls to the Blender Python library “bpy” in said base classes, so users don’t have to interact directly with it as it brings several problems for usability due to complex issues like context or errors, all of these elements helped, not only to simplify the API’s use, but also its functionality and capabilities.
More classes, both basic and complex ones, serving all purposes, from creating cubes, to adding light, to making buildings, to putting it all together to generate a city, all manner of objective can and is organized in a way to easily identify what functions to call, where to call them from and how to do it.
By designing the API to be modular and segmented, it can be easily changed, updated and reworked to better fit whatever scenario is needed.
For the purpose of the project, there is a class made to exclusively produce buildings but, if the user has other intentions, they can create a class that generates tables. To achieve this, they can make use of the basic functions present in the mesh and curve classes to manually generate tables through code. This versatility is what makes the API so powerful as it can be adapted to any circumstance, giving all the tools necessary for users to create what they want.
There is also the possibility of importing works of other people easily found online and using them to create more complex or diverse objects or conglomerates. This was an important decision, not only to give even more tools to users, but also to allow for a good level of detail and complexity to the final project version within the timeframe given, which would not have been possible to reach otherwise.

Here we observe the project's organization, which will be further explained throughout the report:

- Assets
  - BricksPaintedWhite001
  - CityStreetAsphaltGenericCracked002
  - CityStreetRoadAsphaltTwoLaneWorn001
  - CobblestoneArches002
  - ConcreteWall001
  - GroundCobblestone001
  - MetalPanelRectangular001
  - MetalStainlessSteelBrushedElongated005
  - Plaster17
  - RoadCityWorn001
  - Tiles05
  - WoodFlooring044
  - WoodFlooringMahoganyAfricanSandy
  - WoodFlooringMerbauBrickBondNatural001
  - WoodPlanks028
  - WoodPlanksWorn33
  - WoodQuarteredChiffon001
- Basics
  - bezier.py
  - cone.py
  - cube.py
  - curve.py
  - grid.py
  - light.py
  - mesh.py
  - plane.py
- buildings
  - house.py
- camera.py
  - camera.py
- floors
  - floor.py
- imports
  - STL
  - FBX
    - House
      - Door_Red_Wall_Wood_Interior
      - Door_Dark_Glass_Wall
      - Door_Dark_Glass_Wall_Double
      - House_Window_Beige_Wall
      - House_Window_Crossed_Black_Wall
      - House_Two_Window_Crossed_White_Wall_Double
      - House_Window_White_Wall_Double
      - Window_Covered_Blue_Wall_White_Cover_Wood_Interior
      - Window_Floor_Brown_Wall_White_Cover_Wood_Interior

                - Window_Green_Wall_Wood_Interior
                - Windowless_Pink_Wall
                - Windowless_Orange_Wall_Double
                - Windowless_Dark_Glass_Wall
                - Windowless_Dark_Glass_Wall_Double.fbx
            - Street
                - Population
                - Vehicles
    - OBJ
        - Bleuprints
            - House_Blueprint_1_4P5_1
            - House_Blueprint_1_4P5_2
            - House_Blueprint_1_4P5_3
            - House_Blueprint_1_5P5_1
            - House_Blueprint_1_6P5_1
            - House_Blueprint_2_4P5_1
            - House_Blueprint_2_5P5_1
            - House_Blueprint_2_6P5_1
            - Office_Blueprint_30_6P5_1
            - Office_Blueprint_35_6P5_1
- Renders
    - Example_1.png
- roads
    - road.py
    - streetGeneration.py
- SmallStreetAnimation.blend

Chapter 4

Implementation

4.1 Class Bases

As was mentioned before, the entirety of the project stands on the principle of creating a set of classes that function as the basis for every other class to pull from. These are the only main classes in the project and all others are mere subclasses that derive from them and call their functions in order to operate whatever purpose they have been given.
All calls to the Blender Python library of functions "bpy" are done exclusively in these functions in order to simplify the code in all other classes, as well as to make it easier for future users to create their own subclasses without having to worry about directly interacting with Blender.

4.1.1 Mesh

The mesh class is responsible for all meshes used. The cube, plane, grid and cone subclasses are all derived from this class.
As was mentioned before, with the decision of making this class the centre piece for all mesh subclasses, all manner of functions used to alter any aspect of meshes is included in it and can be called upon by said subclasses and their respective objects. These functions are not supposed to be altered or removed by users of the API as they represent basic code that communicates directly with Blender and require advanced knowledge of its inner workings and contexts necessary to execute them.

There are also some more general functions that could be used for objects other than meshes but, in order to give mesh subclasses and their objects the ability to used said functions, these were included in the mesh class. Some of the functions mentioned are present in other classes such as curve or light.

Going over the myriad of functions present in this class would not make sense, therefore, a more general approach will be optimal, focusing mostly on the more impactful and important ones, while also mentioning others less so.

Since the project is written in an object-oriented format, the first component of the mesh class is the initiator function, or "__init__", which is used to initialize the object calling this class. This is achieved by inputting a string for a name. The class data contains another element, its object. This represents the object created in the virtual world, meaning, if a cube is created with the name "House", the class object will have "House" as its name and have the cube associated as its object.

The use of the initializing function will be observed in more practical ways in its subclasses.

Moving onto other especially important functions, the "select" set is the second more encompassing one as it is used in every operation. This set is comprised of functions used to select objects, individually or by group, by name, by class object, by operation mode, or even to deselect. To do so, the "bpy" library is called as it governs these types of operations that directly interact with Blender.

To select every object in the scene, the class is called, in this case as "Mesh" and use the function "*select_all()*". It proceeds execute this "bpy" call: "*bpy.ops.object.select_all(action='SELECT')*"

This has Blender cycle through every object in the scene and select it. To get the selected objects, the function "*get_all_selected()*" operates in the same way as the "*select_all()*" function in the way it is called, returning the objects to be transformed and used.

The "get" set of functions is also extensive, ranging from getting selected objects with "*get_selected()*", object dimensions with "*get_dimensions()*", object location with "*get_location()*", object name with "*getName()*", the scene with "*get_scene()* ", or vertices or edges with "*get_all_vertices()*" and "*get_all_selected_edges()*" respectively. Some of these even have variations for more specific operations, such as getting just a specific dimension instead of all three, which can be achieved with the "*get_dimension_y()*" function.

There is also the inclusion of importing functions. These serve to import meshes already created and saved to a specific folder in the project. To do so, one must first know the file type of the mesh they intend to import and then, choose the function accordingly. An example would be the importing of a mesh saved in a "FBX" file type, typically used when material and texture data is important and required with the import (this is not the case with "STL" file type as these just save geometry data). For this, one needs to call the class "Mesh" and the specific function "fromFBX(filename, directory)". With this, by giving the name of the file and the local directory where it is located, Blender will go to the directory where the "blend" file is located, identify the directory mentioned by the user and search for the file with the given name. If found, it imports it to the current scene with all its data preserved.

Most other functions deemed important usually deal with basic transformations required for most operations:

- Translation
    - *object.translate(dx, dy, dz)*
- Rotation
    - *object.rotateX(degrees)*
    - *object.rotateY(degrees)*
    - *object.rotateZ(degrees)*
- Scale
    - *object.scale(fx, fy, fz)*
- Deletion
    - *object.delete_mesh(global_use)*
    - *Mesh.delete_mesh_by_name(name, global_use)*
- Duplication
    - *object.duplicate_move_origin()*
    - *object.clone(self, newname)*
    - *object.clone0(newname)*
- Application of modifierers like "join" or "difference"
    - *object.diff(other, solver, use_self, hole_tolerant)*
    - *object.diff_keep(other, solver, use_self, hole_tolerant)*
    - *object. join(solver, use_self, hole_tolerant, *others)*
    - *object.join_keep(solver, use_self, hole_tolerant, *others)*
    - *Mesh. join_all(meshes)*
- Material application
    - *object.apply_material(material_name, texture_name, directory)*
- Blender mode toggle
    - *Mesh.toggle_edit_mode()*

We can observe a use case for a function like the "*apply_material*" through a mesh object prior to calling said function in Figures 4.1 and 4.2, demonstrating the object-oriented design choice coming into play:

```
Mesh.select_all()
for mesh in Mesh.get_all_selected():
    if mesh.name == "Sidewalks":
        Sidewalks = Mesh("Sidewalks")
        Sidewalks._object = mesh

Sidewalks.apply_material("Sidewalk_Material", sidewalk_textures[random.randint(0, len(sidewalk_textures)-1)], texture_directories[2])
Sidewalks.uv_unwrap()

Mesh.deselect_all()
```

Figure 4.1 – Example of object creation and object-oriented code implementation and use

```
    def apply_material(self, material_name, texture_name, directory):
        # Turn object to have the material applied to into active object
        bpy.context.view_layer.objects.active = self._object

        self.select()

        material = bpy.data.materials.new(material_name)

        self._object.data.materials.append(material)

        bpy.context.object.active_material.use_nodes = True

        principled_BSDF = material.node_tree.nodes.get('Principled BSDF')

        tex_node = material.node_tree.nodes.new('ShaderNodeTexImage')
        tex_node.image = bpy.data.images.load("//Assets\\" + directory + "\\" + texture_name)

        # Altering the value for the specularity
        bpy.data.materials[material_name].node_tree.nodes["Principled BSDF"].inputs[7].default_value = 0.77

        # Altering the value for the roughness of the material
        bpy.data.materials[material_name].node_tree.nodes["Principled BSDF"].inputs[9].default_value = 0.77

        material.node_tree.links.new(tex_node.outputs[0], principled_BSDF.inputs[0])
```

Figure 4.2 – Material application function designed to function in an object-oriented aspect

This design choice reverberates throughout the entire project and allows for complex applications and operations to be a simple function call from the user's perspective. Instead of running calls to the "bpy" library and having to worry about context and error messages, the user simply needs to call the function to the object in mind and the operations happen in the background, which also keeps the code clean for the user, vastly reducing the amount of code written.

4.1.2 Curve

The curve class, as the name implies, is entirely focused on operations on curve-type objects.
The distinction between curve-type objects and mesh-type ones is mostly derivative from the way the "bpy" library needs to be called for each of them. It differs enough between them that joining both objects into a single class is counterproductive as it increases complexity and generates confusion when objects are called or created. They contain distinct functions, different calls for similar functions and have distinct types of operations possible for each. All of this culminates in enough differences to the point where even in Blender they are catalogued as distinct types, and so, when designing the API, the same choice was made to separate them into two different classes.
This is not to say there are no similar functions that operate in an analogous way, as that would be false. Translation, rotation, scale, importing and some select and get functions operate the same way.

It is when diving to the more specific functions, and some of the base select and get ones, that we find the reasons to separate these classes from one another.
An example on how the curve and mesh classes diverge can be seen with some select functions. In the mesh class, in order to select points from the object geometry, besides requiring a Blender mode toggle just like the same operation in the curve class, the call to the “bpy” library is different. With the mesh class, the call is “*bpy.ops.**mesh**.select_all(action='SELECT')*”, while in the “curve” class, it’s “*bpy.ops.**curve**.select_all(action='SELECT')*”.
Besides minor differences with similar operations, there are also major differences in the types of operations permitted in each object type, which is reflected in their respective classes.
One such difference is the transformation of what are called handles. These represent elements that allow to change a curve’s shape, while keeping the same point coordinates in their respective place. They are exclusive to curve objects. We can see the way they function in Figure 4.3 and the way their called and transformed through code in Figure 4.4.

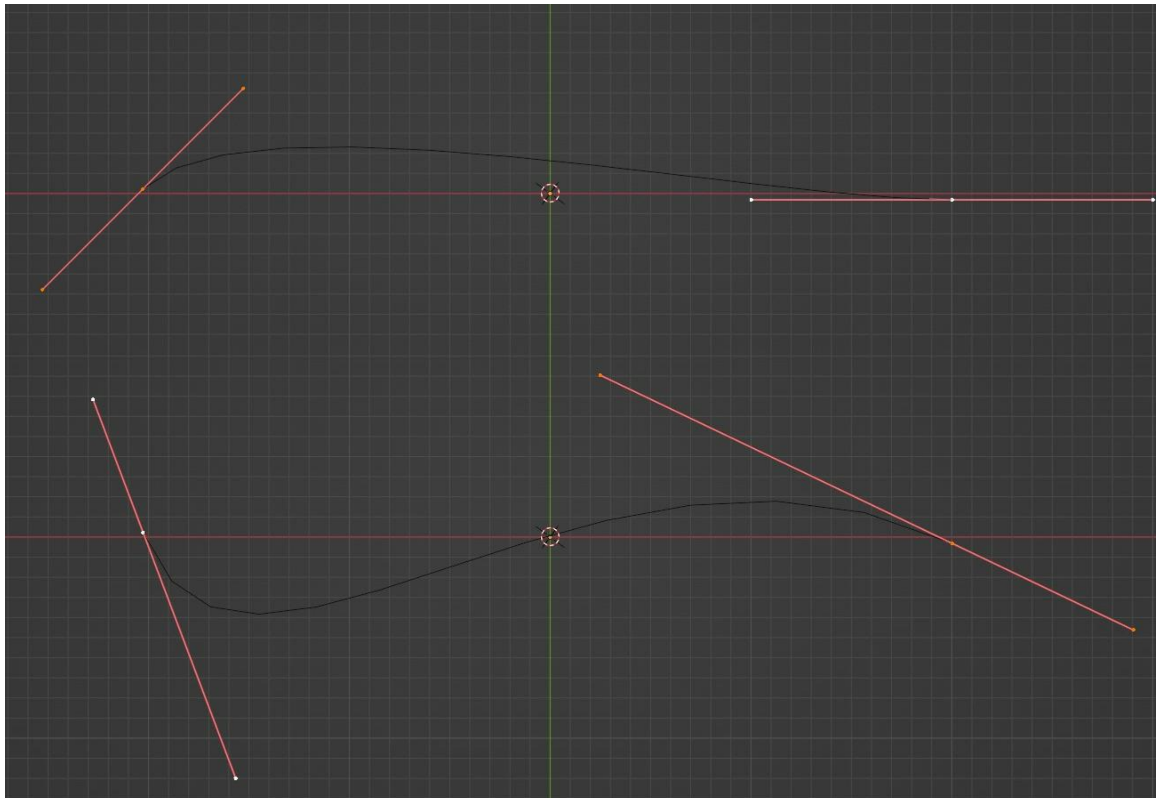

Figure 4.3 – Example of handle transformation to same curve

```
def set_curve_handle_to_automatic(self):
    bpy.data.objects[self._object.name].select_set(True)
    bpy.ops.object.mode_set(mode='EDIT')
    bpy.ops.curve.select_all(action='SELECT')
    bpy.ops.curve.handle_type_set(type='AUTOMATIC')
    bpy.ops.curve.select_all(action='DESELECT')
    bpy.ops.object.editmode_toggle()
    bpy.data.objects[self._object.name].select_set(False)
```

Figure 4.4 – Handle transformation function set to automatic type

Another curve-type object specific operation is point extruding. To be able to add points to a curve, one needs to extrude them with given position coordinates. It is through this method that street generation is possible, as after a curve object is created, the point coordinates for the street need to be inserted into it.

This is achieved by cycling through all the coordinates inputted by the user and applying the extrude function to the curve object for every set of coordinates found.

4.1.3 Camera

As mentioned with both previous classes, the main reason to split camera-type objects into their own class is their specific operations and functions.
With the camera class, we are interacting with the object responsible for capturing, or rendering, the virtual world both in image or video format and exporting or saving it to a folder, to be used for whatever purpose the user entails for it.
To generate a camera object, the class must be called with "Camera", along with the function "*add_camera()*", however, this can only be done after creating the class object as demonstrated by this line, "*Cam = Camera("Camera")*". The variable name and the string given as an input can vary according to user choice, but the call must be done in this fashion. Now the "Cam" object can call functions belonging to the camera class, like so: "*Cam.add_camera()*".
There is also the possibility of generating a camera object with the "*Cam.add_camera_in_specific_location(coordinates)*" function. All to give more tools and options to the user. It also contains some of the basic operations seen in other classes such as select and get, as well as rotation and location.
Another specific operation is render. As the camera is the object responsible for rendering the virtual creations, it contains the function call to the "bpy" library that allows the user to do so. The only input required to use the function is a string for the name of the file to be created with the render, aside from needing the camera object to exist before calling the function as is expected from an object-oriented design. The camera object representation in Blender can be seen in Figure 4.5 and the code to apply the render operation can be seen in Figure 4.6.

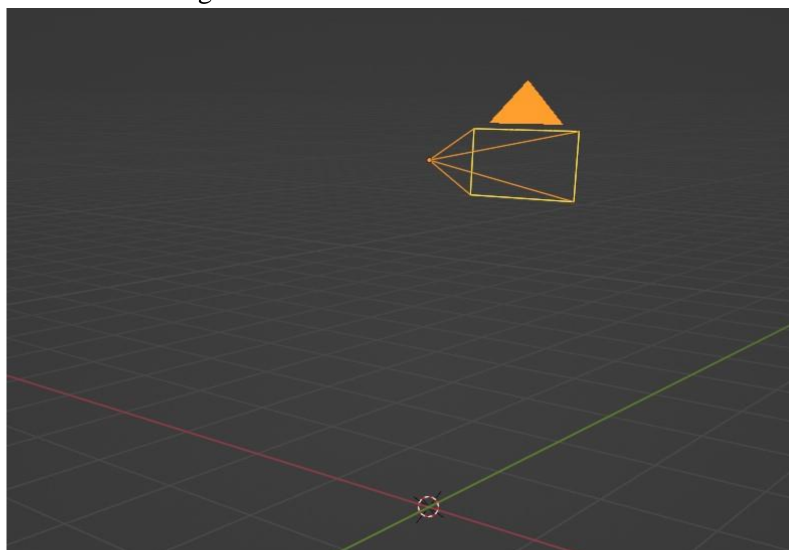

Figure 4.5 – Example of camera object representation in Blender

```
def render(self, filename):
    scene = bpy.context.scene
    scene.render.image_settings.file_format='PNG'
    path = bpy.data.filepath
    path_dir = os.path.dirname(path)
    scene.render.filepath= path_dir + "/Renders/" + filename
    bpy.ops.render.render(write_still=1)
```

Figure 4.6 – Render function callable by camera class object

4.1.4 Light

The final base class is responsible for lighting in the virtual world. This can be Sun-type lighting, focal, or area lighting. The main differences between focal and area lighting are self-explanatory, while Su-type acts as an area light that is "infinitely" far away. The one single purpose all these objects serve is to light the environment in order for renders to be able to capture it. Streetlamps may use area lights to light the floor, or the Sun may be used to light the entire world.

For this project, the most amount of work was put into the sun in an attempt to simulate time of day and the differences it brings to sunlight, in direction, intensity, and colour.

To achieve this, besides using a Sun-type light object, a sky texture was also added to address the difference in colour, intensity, and inclination or direction of the sunlight rays. To apply a sky texture, a tree node needs to be created of the "ShaderNodeTexSky" type and be added to the background of the world. This is achieved simply by calling the "*create_sky_texture()*" that can be seen in Figure 4.7.

```
def create_sky_texture():
    sky_texture = bpy.context.scene.world.node_tree.nodes.new("ShaderNodeTexSky")
    bg = bpy.context.scene.world.node_tree.nodes["Background"]
    bpy.context.scene.world.node_tree.links.new(bg.inputs["Color"], sky_texture.outputs["Color"])
```

Figure 4.7 – Function responsible for creating sky texture and applying it to the world's background

Certain aspects of the texture can be modified, such as the ozone and dust density, size the Sun occupies in the sky, or the intensity, besides the main longitude and latitude values. It is important, however, to only apply said operations after a Sun-type light object has been created as it is necessary to include in the parameters for the operation to function, as can be seen in Figure 4.8, as well as an example of how to tweak the sky texture's values to create specific scenarios, like one of a clear sky. Important to mention that values for the latitude and longitude need to be inputted by the user. This can be done in a random way by making use of the "*random.randint(minimum_value, maximum_value)*" capabilities of the "math" library in Python.

```
def sky_texture_visible_sun_clear_day(latitude, longitude):
    bpy.context.scene.sun_pos_properties.sun_object = bpy.data.objects["Sun"]
    bpy.context.scene.sun_pos_properties.sky_texture = "Sky Texture"
    bpy.data.worlds["World"].node_tree.nodes["Sky Texture"].dust_density = 0.538462
    bpy.data.worlds["World"].node_tree.nodes["Sky Texture"].ozone_density = 2.84615
    bpy.data.worlds["World"].node_tree.nodes["Sky Texture"].sun_size = 0.0391826
    bpy.data.worlds["World"].node_tree.nodes["Background"].inputs[1].default_value = 0.1
    bpy.context.scene.sun_pos_properties.latitude = latitude
    bpy.context.scene.sun_pos_properties.longitude = longitude
```

Figure 4.8 – Example of sky texture value optimization for clear sky scenario

An example of the end result of this process can be observed in Figure 4.9, where the time-of-day influences, not only position, but also inclination and colour of the sunlight on a simple cube mesh as seen by the two different scenarios presented.

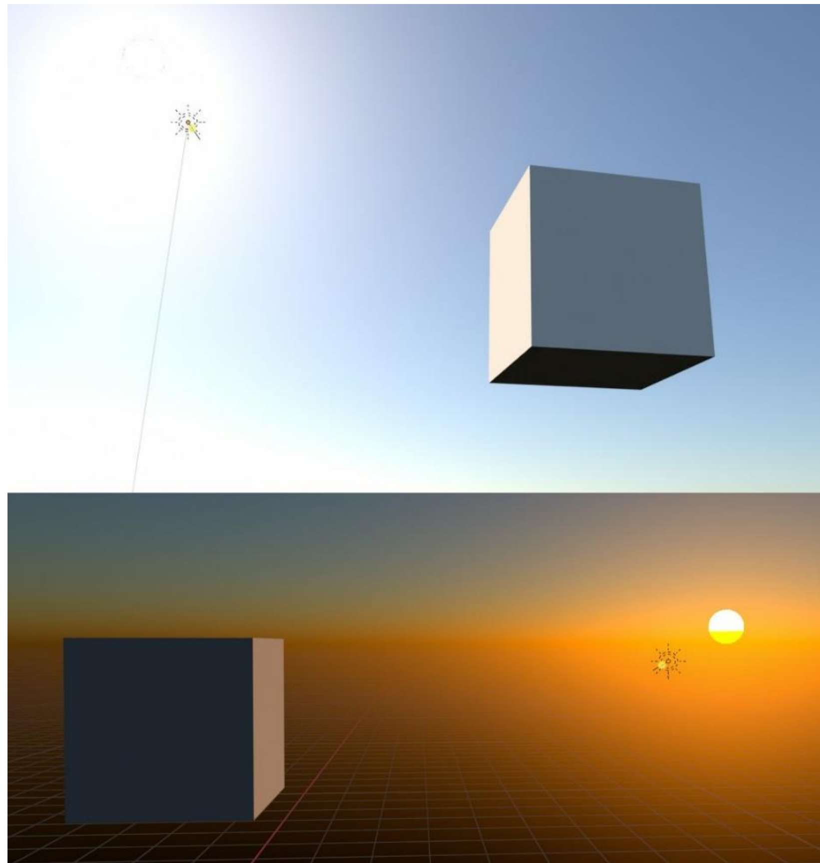

Figure 4.9 – Two examples of changing Sun-type object positions influencing lighting conditions

The way the Sun-type object tracks the centre of the world is by creating a small mesh in the origin coordinates and applying a constraint to the light object that tracks to the position of the origin, done with the function "*track_object*", using the following call to the "bpy" library: "*bpy.context.object.constraints["Track To"].target = bpy.data.objects[object.name]*". This assures the location of the Sun in the sky is relative to the centre of the world.

4.2 Simple Subclasses

The simple subclasses encompass all the direct subclasses to the Mesh or Curve classes, but where the single purpose for their existence is to generate objects.

The subclasses present in this project are the following, separated by class:

- Mesh
    - Cube
    - Cone
    - Cylinder
    - Plane
    - Grid
    - Torus
    - Tube
- Curve
    - Bezier

The way these subclasses are implemented allows for users to generate objects the subclasses are named after by simply creating a class object after the subclass in mind and generating the respective object. Being that these are subclasses to the main base classes, all the functions and operations of the main classes are passed onto these, meaning, a Cube class object can use all transformations, such as translate or rotate, as these have been passed down to it by the Mesh class. This means the actual size of the subclasses in terms of amount of code written is miniscule, as is shown in Figure 4.10.

```
import bpy
from basics.mesh import Mesh


class Cube(Mesh):

    def __init__(self, name, cube_size):
        super().__init__(name)
        self.cube_size = cube_size
        # Generate cube mesh
        bpy.ops.mesh.primitive_cube_add(size=1, scale=self.cube_size)
        # Assign mesh to variable
        cube = bpy.context.selected_objects[0]
        # Name said mesh
        cube.name = self.name
        self._object = bpy.data.objects[self.name]

    def duplicate(self, newname):
        return Cube(newname, self.cube_size)
```

Figure 4.10 – Cube subclass code demonstration

Now, to create a Cube class object and generate a cube mesh, only a single line of code is needed by the user, "*cube = Cube(name, scale)*", where scale is a tuple with three values, one for each axis, name is a string and the name of the variable the Cube class object will be associated with.
The same process is true for every simple subclass, be it a subclass of the Mesh class or the Curve class. As can be seen in the code shown in Figure 4.10, the class object retains the "name" component and the "_object" component continued by the Mesh class when it is initialized.
The possibilities for simple subclasses are much more extensive as Blender has many more simple objects created this way, but for the scope of the project, most were unnecessary or out of reach with the time and resources at hand. Below lies a list with all objects available to receive the same treatment, many from classes not included in the project, such as Text, Volume, Grease Pencil, Meatball, Surface, Armature, Lattice, Empty, Image, Light Probe, Speaker, Force Field, and Collection Instance:

- Mesh – Circle, UV Sphere, Ico Sphere, Monkey (yes, seriously)
- Curve – Circle, Nurbs Curve, Nurbs Circle, Path
- Surface – Nurbs Curve, Nurbs Circle, Nurbs Surface, Nurbs Cylinder, Nurbs Sphere, Nurbs Torus
- Meatball – Ball, Capsule, Plane, Ellipsoid, Cube
- Text
- Volume – Empty, Imported
- Grease Pencil – Blank, Stroke, Monkey
- Armature
- Lattice
- Empty – Plain Axis, Arrows, Single Arrow, Circle, Cube, Sphere, Cone, Image
- Image – Reference, Background
- Light Probe – Reflection Cubemap, Reflection Plane, Irradiance Volume
- Speaker
- Force Field – Force, Wind, Vortex, Magnetic, Harmonic, Charge, Lennard-Jones, Texture, Curve Guide, Boid, Turbulence, Drag, Fluid Flow

4.3 Complex Subclasses

4.3.1 Road Class

This class functions exclusively to build and alter the roads. By doing this, the road generation can be focused and well defined.
Throughout the class, methods, functions, operations belonging to the Mesh class are used in a way to apply all transformations required, all while keeping the code clean and concise. In initial iterations, this class would reach vast volumes of code, crossing more than four hundred and fifty lines of code alone. With the current design applied and several changes made to the generation, it barely crosses the two hundred and fifty lines, all while being much more capable than the previous iterations.
Putting efficiency aside, the main area of focus was to elaborate a straightforward way to generate complex geometry that can look believable. To do so, while this is a subclass of the Bezier subclass, which itself is a subclass of Curve, both mesh and curve objects are needed.
First, the class object needs to be initialized with the given inputs of either a string for a name or a list. For this example, the object will be initialized with a single string. Then, to generate the roads, the "*generate_all(roads)*" function is called by giving the input of a list containing several elements, where each contain a list of coordinates, a name for the street, and a scale value. It cycles through said list, calling upon the function "*generate()*" twice per element, once for the street and another for the sidewalk
In the "*generate()*" function, the first order of business is to generate a Bezier curve and apply "*extrude_points()*" functions to it to place every curve point in the correct inputted coordinates.
Then, the function creates a cube mesh to which it applies the scale values inputted previously, flattening it in a way. To fill the path created by the curve, the cube mesh, two modifiers are applied: array and curve.
The array modifier is used to multiply the cube mesh as many times as needed to fill the distance of the curve object and the curve modifier is used to guide the cube mesh and its copies along the curve object itself. Both methods can be seen in Figures 4.11 and 4.12, respectively.

```
def set_array(self, curve):
    bpy.data.objects[self._object.name].select_set(True)
    bpy.ops.object.modifier_add(type='ARRAY')
    bpy.context.object.modifiers["Array"].fit_type = 'FIT_CURVE'
    bpy.context.object.modifiers["Array"].curve = bpy.data.objects[curve.name]
    bpy.ops.object.modifier_apply(modifier="Array")
    bpy.ops.object.select_all(action='DESELECT')
```

Figure 4.11 – Array modifier applicating function

```
def set_curve_positioning(self, curve):
    bpy.data.objects[self._object.name].select_set(True)
    bpy.ops.object.modifier_add(type='CURVE')
    bpy.context.object.modifiers["Curve"].object = bpy.data.objects[curve.name]
    bpy.context.object.modifiers["Curve"].deform_axis = 'POS_Z'
    bpy.ops.object.modifier_apply(modifier="Curve")
    bpy.ops.object.select_all(action='DESELECT')
```

Figure 4.12 – Curve positioning modifier applicating function

This class contains 4 functions:

- generate()
- generate_road_curves_only()
- generate_all(roads)
- generate_all_curves_only(roads)
- place_vehicles(objects, path, roads, delete_cars_not_placed, specific_x_rotation, specific_y_rotation, specific_z_rotation, specific_x, specific_y, specific_z)

The differences between the main "*generate()*" function and the others are small, coming down to the amount of code included in them from the main function, instead of differentiating themselves with different, innovative code. They represent function over form in every way as they are extremely useful for various scenarios in the project.
As the name implies, "*generate_road_curves_only()*" operates in an identical manner as the base "*generate()*" function, but instead of generating both curve objects and mesh objects, it stops short, producing only the curve objects. This will prove useful when talking about the pathmaking for the camera, as cloning curve objects proved more challenging than expected, so, this was the solution found.
For the "*generate_all()*" functions, these act as simple ways to go through a set of inputs. When there is a need to create several objects of a type, these functions iterate through all the inputs and call the "*generate()*" and "*generate_road_curves_only()*" functions as many times as needed to complete the set of inputs.
As for the randomness or procedural aspect of road generation, this comes per user input, not as an inherent feature or function of the class itself. This means, no part of the generation of the roads is random without the user inputting random elements. In this case, the road is generated by receiving a list with the coordinates, names and scale values to generate them. If the user inputs a randomly generated set of values, the functions will generate said roads in the random fashion. The same happens with the application of the materials to both street and sidewalk. To do this, a list containing all the materials is created and, when calling the objects to have materials applied to them, a "*random.randint(minimum_value, maximum_value)*" function is used to randomly choose a material from the list. With this, every time the roads are generated, they contain a randomly applied material and texture. The culmination of the entire class can be seen in the code in Figure 4.13. 3D results of this class can be seen in Figure 4.14 untextured and in Figure 4.15 with textures.

```
joint_road = Road("Roads")
joint_road.generate_all(roads)

Mesh.deselect_all()

Mesh.select_all()
for mesh in Mesh.get_all_selected():
    if mesh.name == "Streets":
        Streets = Mesh("Streets")
        Streets._object = mesh

Streets.apply_material("Street_Material", road_textures[random.randint(0, len(road_textures)-1)], texture_directories[1])
Streets.uv_unwrap()

Mesh.deselect_all()

Mesh.select_all()
for mesh in Mesh.get_all_selected():
    if mesh.name == "Sidewalks":
        Sidewalks = Mesh("Sidewalks")
        Sidewalks._object = mesh

Sidewalks.apply_material("Sidewalk_Material", sidewalk_textures[random.randint(0, len(sidewalk_textures)-1)], texture_directories[2])
Sidewalks.uv_unwrap()

Mesh.deselect_all()
```

Figure 4.13 – Code representing road generation and texturing

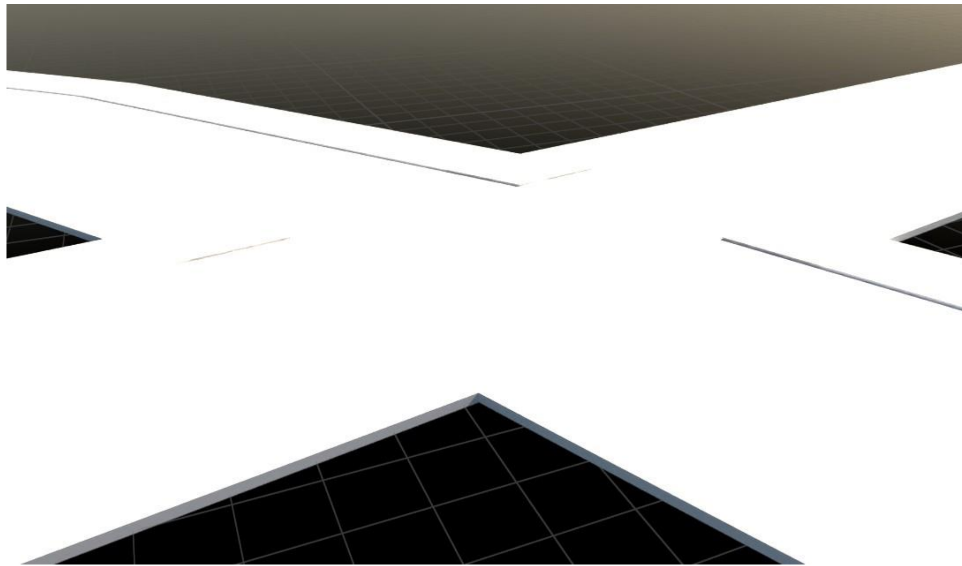

Figure 4.14 – Sample of untextured road generation

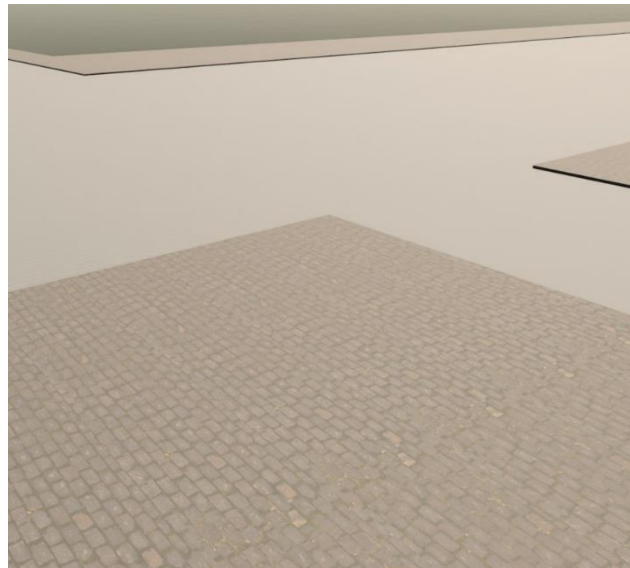

Figure 4.15 –Sample of textured road generation

To join all roads that intersect, a check for intersection is made and if it verified to be true, the road meshes are joined with each other, as are the sidewalk meshes. This finalizes in a single mesh containing all roads and another single mesh containing all sidewalks.
To verify the intersection, there are two ways of applying said check: "*intersect(object)*" and "*bmesh_check_intersect_objects(obj)*".
These functions achieve the same goal, albeit in two distinct ways. Both functions operate with the use of either "BVHTree", or "Bounding Volume Hierarchy Tree" or "BMesh", or "Boundary Mesh", which represent a way of forming bounding volumes around objects, in order to understand where their borders lie. This is useful to detect collisions or to apply ray tracing.
While "*intersect(object)*" uses these boundary detecting operations to evaluate whether or not there is a collision, through vertex coordinate comparison, effectively searching to see if any vertex from a given object exists within the space of another, "*bmesh_check_intersect_objects(obj)*" however, uses ray tracing to calculate volumes and detect if any intersect by evaluating ray paths.

In terms of complexity and processing cost, which leads to longer wait times for the work to finish, "*bmesh_check_intersect_objects(obj)*" is just as expensive as "*intersect(object)*" as both require the cloning of the objects in question in order to operate, which can become very taxing when said objects are very complex and take some time to generate. The only area where "*intersect(object)*" may have some advantage in cost, is scenarios where the amount of ray bounces occurring is high enough to heighten the cost of using the same ray tracing to then detect intersections, but that is not the case with this project. The snippet of code in Figure 4.16 shows the "*intersect(object)*" function in the mesh class.

```
def intersect(self, object):

    mesh_1 = bpy.data.objects[self._object.name]
    m_1 = mesh_1.matrix_world.copy()
    mesh_1_verts = [m_1 @ vertex.co for vertex in mesh_1.data.vertices]
    mesh_1_polys = [polygon.vertices for polygon in mesh_1.data.polygons]

    mesh_2 = bpy.data.objects[object._object.name]
    m_2 = mesh_2.matrix_world.copy()
    mesh_2_verts = [m_2 @ vertex.co for vertex in mesh_2.data.vertices]
    mesh_2_polys = [polygon.vertices for polygon in mesh_2.data.polygons]


    mesh_1_bvh_tree = BVHTree.FromPolygons(mesh_1_verts, mesh_1_polys)
    mesh_2_bvh_tree = BVHTree.FromPolygons(mesh_2_verts, mesh_2_polys)

    return mesh_1_bvh_tree.overlap(mesh_2_bvh_tree)
```

Figure 4.16 – Intersect function present in the mesh class

With vehicle placement, this function focuses on retrieving vehicle objects and placing them in a relatively realistic manner throughout the road mesh, avoiding nonsensical orientations and locations. This makes the use of a few inputs:

- objects (the objects to be placed on the roads)
- path (a path made with a curve object placed along the road objects)
- roads (the road objects themselves)
- delete_cars_not_placed (a Boolean object that determines if leftover objects are deleted)
- specific_x_rotation (a specific value for a rotation on the x-axis)
- specific_y_rotation (a specific value for a rotation on the y-axis)
- specific_z_rotation (a specific value for a rotation on the z-axis)
- specific_x (a specific value for a placement on the x-axis)
- specific_y (a specific value for a placement on the y-axis)
- specific_z (a specific value for a placement on the z-axis)

These inputs help give a good amount of control to the user in regards to the placement of objects on the road meshes. By determining the location of the points in the path provided, it is possible to identify the following and previous point locations relative to the current one and conclude the correct placement of the current object's placement and orientation as shown below in a snippet of code that demonstrates part of the orientation angle of the object being determined:

```
z_angle = math.atan2(y_difference, x_difference)
normal_z_angle = math.atan2(normal_y_difference, normal_x_difference)
```

Figure 4.16 – Intersect function present in the mesh class

4.3.2 House Class

Finishing the complex subclasses, the house class is responsible for generating buildings. This is a subclass of cube, which is a subclass of mesh.

The way this class has been designed focuses more on handmade meshes representing windows, walls, doors and more, that are imported and then grouped together, instead of generating all from scratch. This approach comes mainly due to the lack of time and resources required to find and learn a way to generate such complex meshes through code alone. To circumnavigate this, a set of meshes was created by hand and exported in "FBX" format, with textures already placed, all in order for these to be selected and imported later in the generation of the buildings. The textures were also pre-applied as was just mentioned, due to the amount of complexity in the meshes created and the number of textures and materials each contains. There may be a way to implement the texture application through code alone, but it was not achievable during this time.

To generate a building, one must first create a class object as such: "*Build = House(name, scale, type, width, length, height, color_scheme, meshes, resolution)*".

Afterwards, the "*generate()*" function is called and the process for building generation begins. First order of business is to import the relevant meshes to construct the building with. The guidelines for which meshes to import are part of the inputs delivered by the user, in this case, "*color_scheme*" and "*meshes*". The "*color_scheme*" refers to the type of material to be selected and used, with some examples being "Beige_Wall", or "Blue_Wall", determining the colour scheme to be used. With the "*meshes*", the objective is to limit or denote which types of meshes to use, with examples such as "No_Covered" or "No_Floor", meaning no covered windows or no floor windows to be used in the building generation.

These methods can be expanded to allow for more freedom and variety in determining how to create the buildings. This selection can be seen in Figure 4.17 in a section of the selection process.

```
if self.color_scheme == "Green_Wall":
    for m in Mesh.getImportFiles():
        if "Green_Wall" in m:
            if self.meshes == "Random":
                meshes.append(m)
            elif self.meshes == "No_Covered":
                if "Covered" not in m:
                    meshes.append(m)
            elif self.meshes == "No_Uncovered":
                if "Covered" in m or "Cover" in m or "Door" in m:
                    meshes.append(m)
            elif self.meshes == "No_Floor":
                if "Floor" not in m:
                    meshes.append(m)
            else:
                meshes.append(m)
```

Figure 4.17 – Section of the selection process for imports in building generation

After selecting the appropriate meshes, the following step is to import them with the function "*fromFBX(name, directory)*", applying a change in the object's origin to better suit their organization, all while separating them into their own respective lists for an easier way to find them later in the process.

Before all pieces are put in place, it is necessary to understand how many of each piece are necessary, something that is also given by both the available meshes gathered after the selection process, as well as the dimensions of the building which were inputted by the user, as is shown in Figure 4.18.

```
clone_num = 0
place = 0

for l in floor_windows:
    while mesh_class_object_list[place].name != l.name:
        place += 1
    for go in range(round((self.width*2 + self.length*2)/len(floor_windows))):
                j = mesh_class_object_list[place]
                p = j.clone0(j.name + "_Floor_0_" + str(clone_num+1))
                mesh_class_object_clone_list.append(p)
                clone_num += 1
    place = 0
    clone_num = 0
```

Figure 4.18 – Part of the coded method used to calculate mesh clone number for building generation

With this value calculated and the correct number of meshes cloned from the original imported ones, now begins the process of constructing the building object by carefully cycling through all the available meshes that are allowed in each floor of the building and placing them accordingly. This segmentation by floor was put in place to create some order to the placement of meshes as it would not make much sense to find a door on the fourth floor. The positions are calculated with the use of both the mesh sizes, as well as the building dimensions, in order for all to come into contact in the exact places, avoiding intersections or gaps when all are joined to form a single building object.

There are two meshes generated by the code that do not come from manually made imports, and those are the floor and ceiling meshes. These are basic planes that are placed on the top and bottom of the building to conclude the mesh placements.

In an effort to remove any possible leftover meshes that found no use, there is a final cycle through all the meshes in the scene with the specific names that fit the meshes imported and cloned and, whichever ones are found outside the expected places are deleted.

After all checks are made and all meshes are in the correct place, all are joined with the use of "*join_all(meshes)*" function present in the Mesh class, finalizing the building generation into the objects shown in Figure 4.19.

Figure 4.19 – Example of result of building generation

These building were generated using the exact same lines of code. This was achieved by applying random inputs when creating the class objects. Where the "*color_scheme*", "*mesh*", "*width*", "*length*" and "*height*" inputs are placed when creating the class object, "*random.randint(minimum_value, maximum_value)*" were used to randomly assign values to said inputs, within the accepted ones, meaning, every time the class object is created, it contains different inputs, all while being written in the exact same way. Figure 4.20 shows this implementation in action:

```
Build = House(
                "Building",
                (3, 3, 3),
                "Building",
                random.randint(3, 7),
                random.randint(3, 7),
                random.randint(3, 7),
                house_color_restrictions[random.randint(0, len(house_color_restrictions)-1)],
                house_mesh_restrictions[random.randint(0, len(house_mesh_restrictions)-1)],
                1
               )
```

Figure 4.20 – Building class object creation with randomized

Touching on textures and materials, as these were implemented before hand and no code was used to apply or generate them, a description of these is in order.

The generation of materials is quite simple. After selecting a mesh, on the lateral panel in Blender's UI, there is a section for material and texture work, shown in Figure 4.21. After accessing this area, many options are presented to the user.

Type of surface ("*Principled BSDF*", "Diffuse BSDF", "Backfround", "Glass BSDF", "Subsurface Scattering", "Specular BSDF", and many more), base colour, metallic properties, sheen, specularity, transmission, roughness, emission, alpha, so on and so forth, all these elements allow for the user to create virtually any type of material imaginable.

As for textures, these are applied in the base color section, along with a myriad of other options, as well as many distinct types of textures.

An example would be the creation of the glass textures implemented in the windows of the buildings previously presented. To create these, the surface chosen was "*Principled BSDF*" and the base colour was white. To create the transparent, yet slightly reflective surface, the metallic element received the maximum value, and the specular element received a half value, all while the alpha received only a quarter value as it determines the amount of light the material reflects back.

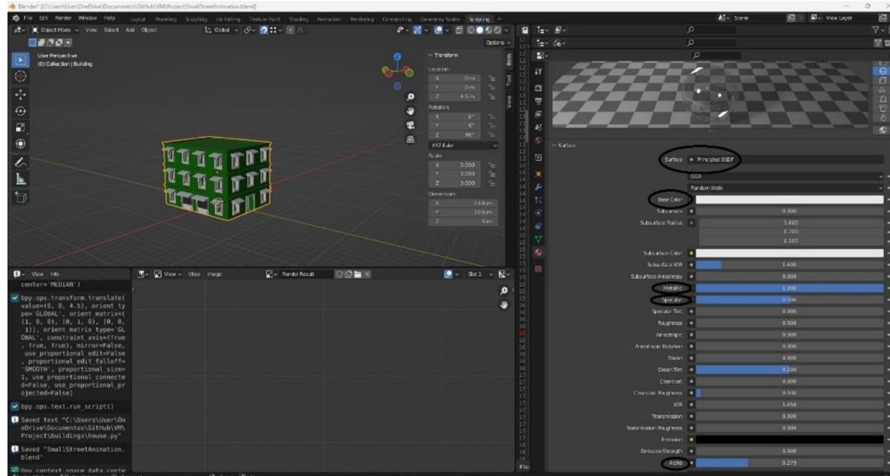

Figure 4.21 – Blender UI for material and texture application and editing

The remaining textures and materials used in the building meshes are more straightforward as it mostly implies as simple change of colour value in the base colour element, with some choices being green, blue, beige, black, white, and red. Patterns are also possible to create but require more finesse and ability with the texture elements of Blender. Striped, dotted, irregular, or blocked are some of the possibilities present here.
Taking everything demonstrated above, another method of building generation was developed in order to give more options and freedom to users in their creative endeavours: blueprints.
By using pre-made blueprints, users can apply more complex building designs whilst still only having to focus most of their time on the meshes that will comprise said buildings. The blueprints themselves are made of curve objects, mostly straight lines, that fill out and define the line work where the meshes will be placed. The individual curve objects will be named according to the following scheme in order for the rest of the code to be able to determine elements such as height, orientation, and type of mesh:

- mesh type (wall, door, pillar)
- floor (1, 2, 3)
- orientation (0D, 90D, 180D, 270D)
- number of object in the same previous three conditions (1, 2, 3)
- type of the building line object (BezierCurve)

The scheme results in object names such as this: Wall_4_180D_3_BezierCurve. This can be interpreted as a wall on the fourth floor rotated 180 degrees and it is the third one in these specific conditions. To ease the amount of work on the creation of the blueprints, a solution to identical floors was added to avoid needing to create a blueprint for every level of a building. To do so, a section of code was added to check the floor in all the elements of the blueprint provided and simply copy and paste the same layout distribution of a floor a number of times that represents de floor difference between a blueprint level and the following one. Simply put, a user can make a blueprint section on floor three and another on floor twenty and the code will apply the layout and specifications from the floor three section all the way to floor twenty, where it will switch to the new one presented at that floor. It will apply this same idea to the last section presented if it does not represent the last floor as is indicated in the file the blueprint is included in, meaning, if the blueprint specifies the building is to have fifty floors and the last section it holds is presented at floor thirty, the function will apply the layout and specifications from that section all the way to floor fifty.
Below there are present a snippet of code for this method and an example of both an office building and a house generated through the blueprint method, seen in Figure 4.22, 4.23 and 4.24, respectively:

```
for floor in range(2, Floors + 2):
    is_there_a_higher_floor = False
    for fl in differing_floors[::-1]:
        if not is_there_a_higher_floor:
            if fl < floor:
                mesh_list = floor_meshes[str(fl)]
                for m in mesh_list:
                    if "Door" in m.name and floor == 2:
                        m_name = m.name.split("_")
                        door_m = Mesh(Doors[0].name)
                        door_m._object = Doors[0]._object
                        Doors[0].rotateZ(int(m_name[2][0 : len(m_name[2]) - 1]))
                        if Doors[0]._object.dimensions.x < 3 or Doors[0]._object.dimensions.z < 3:
                            Doors[0].scale(1.5, 0.15, 1.5)

                        door_m.make_active_object()
                        door_m.select()
                        Mesh.toggle_edit_mode()
                        vert_w_1 = door_m.select_lower_front_vertex()
                        scene.cursor.location = vert_w_1
                        Mesh.toggle_edit_mode()
                        door_m.set_origin_cursor()
                        door_m.deselect_specific()
```

Figure 4.22 – Small segment of code demonstrating the blueprint method in effect

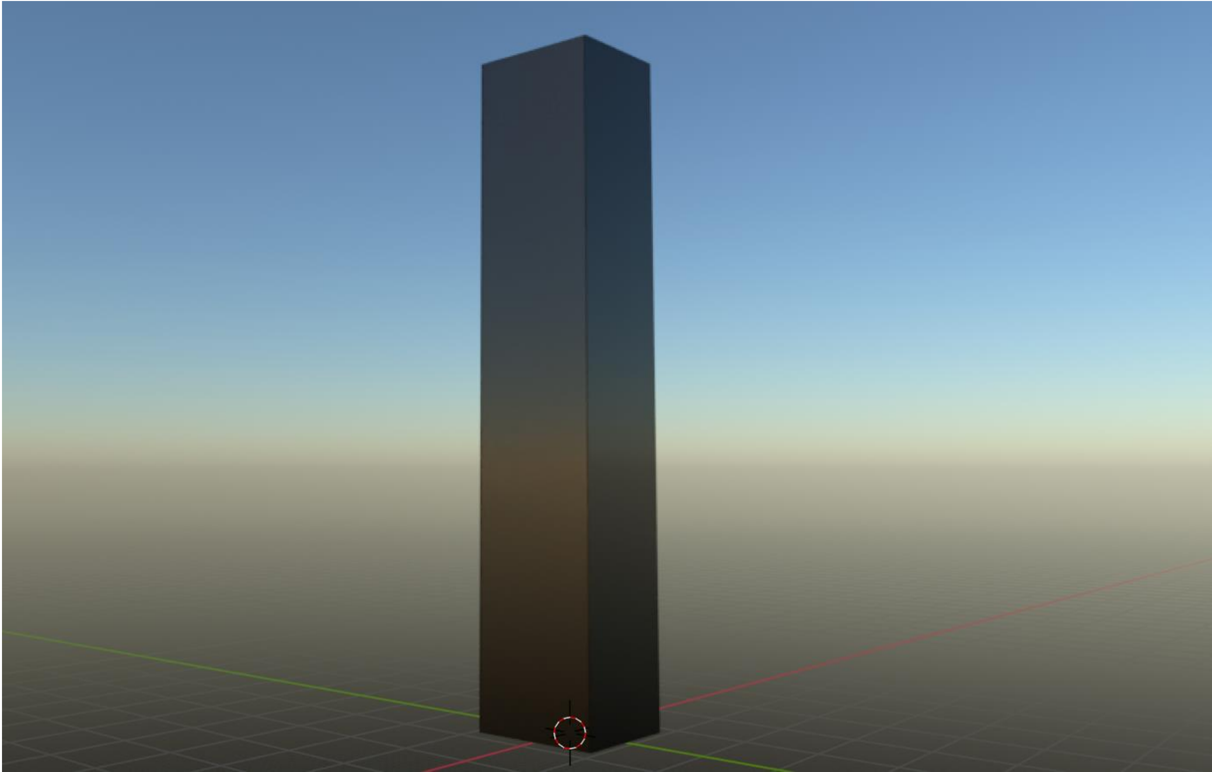

Figure 4.23 – Example of an Office Building

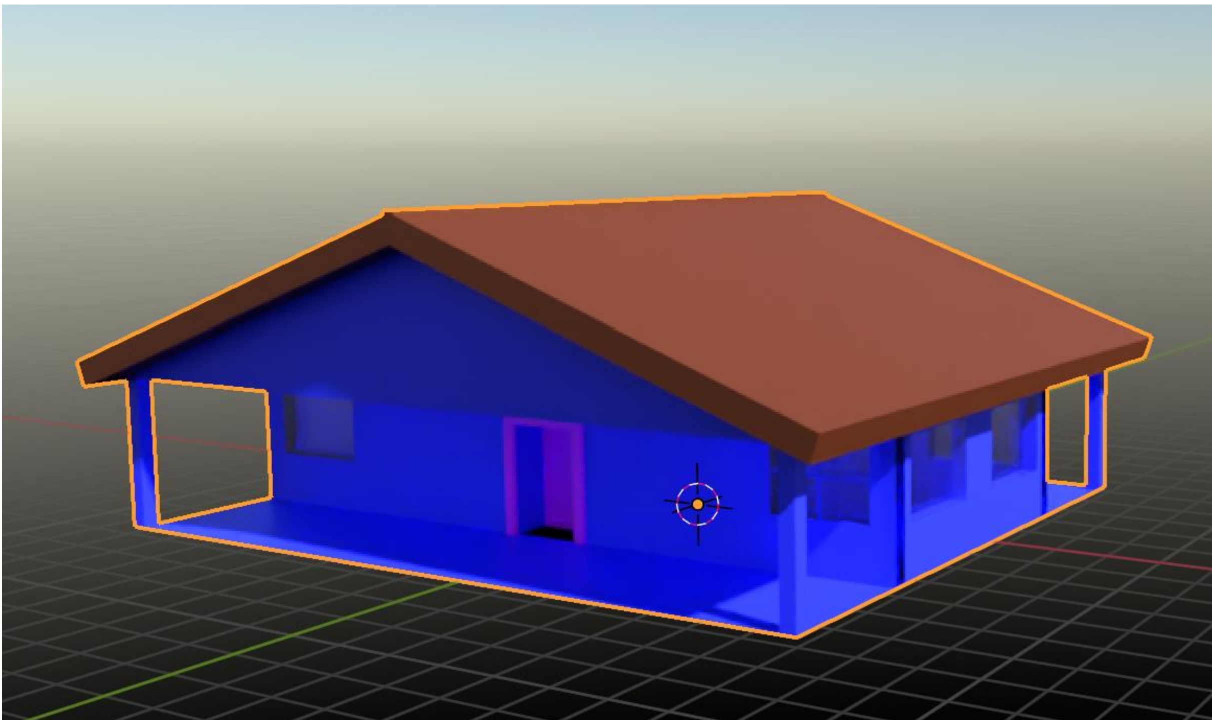

Figure 4.24 – Example of a House Building

4.4 Culmination of Work

With all the main bases covered for both the design choices and the implementation process for the building blocks of this project, it all culminates in the StreetGeneration Python file, where all aspects of the API, classes, subclasses, imports and assets, come together to form the virtual world generation.
As is the case for all files, the first order of business is to import all files needed to execute the operations. This being the culminating work, every file present in the world formation is imported. Figure 4.25 shows this aspect in detail.

```
from Operations import *
from basics.cube import *
from buildings.house import *
from roads.road import *
from basics.cone import *
from basics.mesh import *
from basics.light import *
from camera.camera import *
from basics.curve import *
from basics.plane import *
from basics.grid import *
from floors.floor import *
```

Figure 4.25 – File imports in StreetGeneration.py that allow to generate the virtual world

After acquiring all necessary files, a pre-emptive scene cleaning operation removes all possible leftover objects or unnecessary components. To achieve this, the Mesh class's "*clean_scene()*" function is used. It cycles through all objects, materials, textures, and meshes and removes them all from the scene. This deletes most components that have any meaning or impact to our scene. The only exception to this, in the context of this project, is the sky texture that, for some reason, exists outside the parameters searched when cleaning the scene. To remove this texture, a special function is needed from the Light class, "*clear_sun_settings(node_tree, nodes)*". Both cleaning methods are present in Figure 4.26.

```
def clean_scene():

    # Removes all objects from the scene
    for object in bpy.data.objects:
        bpy.data.objects.remove(object)

    # Removes all materials from the scene
    for material in bpy.data.materials:
        bpy.data.materials.remove(material)

    # Removes all textures from the scene
    for texture in bpy.data.textures:
        bpy.data.textures.remove(texture)

    # Remove all meshes from the scene
    for mesh in bpy.data.meshes:
        bpy.data.meshes.remove(mesh)
```

```
def clear_sun_settings(nd_tree, nds):
    bpy.context.scene.sun_pos_properties.sky_texture = ""
    bpy.context.scene.sun_pos_properties.sun_object = None
    for node in nds:
        if node.name == "Sky Texture":
            nd_tree.nodes.remove(node)
```

Figure 4.26 – Cleaning methods for the world scene (left) and the sky texture (right)

With the scene clean, the last aspect to take care of before beginning the generation process is to create the input lists with all the necessary information. This is done beforehand so that when the objects are created and the functions are called, the procedural effect or randomness element can be applied with the "*random.randint(minimum_value, maximum_value)*" function.

The following lists were created for an example of city generation (some only contain part of their contents as they would not fit here):

- roads = [[(65, 10, 0), (100, 10, 0)], “Avenida_Duque_de_Ávila”, (1, 5, 0.1)]
- road_textures = [“CityStreetAsphaltGenericCracked002_COL_1K.jpg”]
- sidewalk_textures = [“CobblestoneArches002_COL_1K.jpg”]
- wall_textures = [“BricksPaintedWhite001_COL_1K.jpg”]
- floor_textures = [“WoodFlooring044_COL_1K.jpg”]
- population = [“Brunette_Woman_Black_Clothes_Low_Poly.fbx”]
- texture_directories = [“Floor_Textures”, “Road_Textures”, “Sidewalk_Textures”, “Wall_Textures”]
- house_mesh_restriction = [“No_Covered”, “No_Floor”]
- house_color_restrictions = [“Green_Wall”, “Beige_Wall”, “Red_Wall”]

Having finished all preparations, the world generation can begin, starting with the Sun creation and sky texture application. To do so, a cube is created in the world origin to serve as the tracking point for the Sun object, followed by the Sun object creation and the application of the sky texture.

After doing so, the road generation takes place, using the Road class function “*generate_all(roads)*” to create all the road objects in a single line. To segment the objects by specific names, the inner road objects are joined into a single object named “Streets” and the same is done to the outer objects to create the “Sidewalks”. At the same time this is being done, the Mesh class function “*apply_material(material_name, texture_name, directory)*” is used to apply materials and textures to said objects. In order to better fit the textures into the objects, their UV maps’ scale is altered with the “uv_unwrap()” function, present in the Mesh class. This is responsible for changing scale, orientation, direction, among other operations on UV maps, but for this purpose and for this project, not only is it exclusively used to alter scale, it does not take user inputs, instead containing its own pre-written values that were achieved through a manual process of understanding which fit best, and is also being used solely in the “Streets” and “Sidewalks” objects, seen in Figure 4.27. Some work can be done here to allow user expression and control over this aspect of objects.

```
def uv_unwrap(self):
    # Turn object to be remeshed into active object
    bpy.context.view_layer.objects.active = self._object

    self.select()

    uv_map = self._object.data.uv_layers["UVMap"]

    for uv_index in range(len(uv_map.data)):
        if "Street" in self._object.name:
            uv_map.data[uv_index].uv = (0.5 + 550 * (uv_map.data[uv_index].uv[0] - 0.5),
                                        0.5 + 550 * (uv_map.data[uv_index].uv[1] - 0.5))
        else:
            uv_map.data[uv_index].uv = (0.5 + 1.4 * (uv_map.data[uv_index].uv[0] - 0.5),
                                        0.5 + 25 * (uv_map.data[uv_index].uv[1] - 0.5))
```

Figure 4.27 – UV unwrapping function used on meshes to alter texture scale

To create the world floor where buildings will be placed upon, the respective Floor subclass is used in “*floor_for_building_placement = Floor("Floor_For_Building_Placement", highest_value, 1)*” and subsequent “*generate()*”. The highest value is obtained by cycling through the dimension values of both “Streets” and “Sidewalks” objects and using the highest one found, allowing for the world floor generated to fill all inner areas between both road objects.

This massive mesh is then “cut” into pieces perfectly filling the spaces between the road objects with the use of a “difference” modifier.

This modifier takes two objects and removes the volume of one of the objects to the other. In this case, the “Sidewaks” object was used to remove their volume to the floor object, resulting in several smaller objects named “Block”.
Prior to beginning the building generation, to clean possible unwanted results of the world floor generation, a cycle process goes through all “Block” meshes in the world in order to find all objects that are too small, resultant of error margins in the operation, as well as removing all objects that are too big, resultant of the remaining floor mesh residing outside the roads meshes. When found, these objects are removed.
Moving to the building generation, at the time of delivery, the main focus came from the particle system provided by Blender. This is a system that allows the copying of objects and their spreading throughout the surface of other objects, be it hair on a person’s head or grass on a lawn. For this scenario, the objects to replicate and spread out are buildings and to use the function correctly, said buildings are created before applying the particle system.
This is due to each particle system requiring the objects to be created before applying it to another object’s surface. To add all different building objects generated into a single particle system, these must either be placed in a new Collection under the same project as this is the only way to add more than one object to the same particle system, or create multiple particle systems, each with a single object.
To avoid particles collision, an add-on named “Molecular”[14] can be used, as well as another addon named “Blue Noise”[15], in order to prevent said scenario, although it is also possible to achieve the same result by changing the physics of the particle system to “Fluid” and raising the values of stiffness and viscosity to the maximum possible. It needs to be said, however, that neither these nor any other methods from any iteration using a particle system was able to fully remove collisions from taking place at all, something that needs further work to be achieved, as does the avoidance of particles overlapping with “Sidewalks” or “Streets” objects. Figure 4.28 shows the problem of building collision and overlapping in action, as the orange building demonstrates said problems with the green building.

Figure 4.28 – Example of building placement particle system

Briefly mentioning a prior iteration for building generation, this will mostly serve to show the vast improvement to performance achieved in this project.

As was trialled with in one of the prior iterations, to generate buildings, each floor object would be used to determine positions and, when starting the coordinate selection process, three particular aspects were focused on: mesh vertices, distance to closest building, and intersection with either "Sidewalks" or "Streets".This was especially important to achieve coherence in world building, as it makes no sense to place buildings on top of each other or in the middle of streets.
Here is where the first major processing cost appeared. To generate more vertices in a mesh, the "*remesh(value)*" function was used to split the mesh as many times as requested, creating said vertices. The processing cost was not inherent to the function, but to the scale at which it was being employed. The size was the problem for the specific example in place, which is somewhat median or even small in other contexts, that ended up resulting in hundreds of thousands of vertices created. Cycling through all these points and comparing distances to all other already placed buildings, as well as verifying whether the new building placed intersected or not with any of the road objects slowed down performance to the point of having to wait several hours, if not days, for the process to finish. This only worsened with rising levels of detail and resolution, meaning high-resolution textures and materials were difficult to apply as the processing budget was spread out too thin. To apply the "*remesh(value)*" function, the respective mesh to which the process would be applied needed to be turned into the active object and selected first. After doing so, the Blender mode needed to be toggled to "Edit Mode".
Only then, could the function be applied. This process is shown in Figure 4.29, as well as the "*remesh(value)*" function itself.

Figure 4.29 – Remesh process and function to be applied to every floor object

With the generation of all needed vertices in place, the following step was to select all the relevant ones and cycle through them to start the verification process in order to place buildings.
Coming back to the final iteration, as the goal of the project is to correctly simulate a city, it was decided that it was preferable to have the program take longer to execute all requests in building the city, rather than building it quickly with collisions and overlapping issues. There is also the possibility to improve the methods used here with further development and resources.
With that, the process used for the building placement is a "manual" one. This means no add-ons, no plugins, no inherent functions from the Blender libraries, but a way to verify each component of the city needed to place buildings and the application of all transformations necessary. With the aid of the "*remesh*" function and the "*difference*" Boolean modifier, "*remeshed*" plane can be placed as a building placement area which has many vertices to its structure. This allows for the ability to select and use certain vertices as building placement coordinates. By selecting all vertices from the building placement planes that border other meshes, such as "Streets" or "Sidewalks", and all vertices within a set distance from those, we can create a list of vertices that are not to be used for building placement, artificially creating a safety net so that the placed buildings do not overlap with the "Sidewalk" and "Street" meshes.

Then, for all the vertices left, a simple calculation of distances between them will allow for a process of placing buildings without placing them over each other. In order to orient the buildings correctly, the centre coordinates of the plane these currently occupy can be used to determine the building's relative position (front, behind, left, right) and orient it accordingly.
Besides building placement, there is also a process to place several different meshes within the area of both "Streets" and "Sidewalks" objects. The world population process functions utilizes the same particle system mentioned in the building placement section, with one small difference: all objects in this process are imported, not made by the code.
Here, all the objects saved and ready to be imported for population purposes are requested and placed onto the world to be spread out within the margins of the "Sidewalk" meshes. Both with the world population process, as well as the building placement process, the higher number of different objects in place, the better variety and believability can be achieved. The example of the particle system feature shown in Figure 4.30 shows an early discovery of the process and merely represents the way it works.

Figure 4.30 – Example of particle system at work for world population with person-like objects

With this, the virtual world generation is finalized, with its road system, building distribution and population placing.

4.5 Extra Work

While the focus of this project is to generate virtual worlds, one cannot ignore the context behind it. The main goal of this proposition is to help feed an object identification model that my coordinator Thibault Lanlgois is currently developing, so, even if not necessarily part of the world generation focus of the project, the inclusion of a camera for the purpose of capturing renders of said virtual world comes as process almost as integral as the rest.
In order to help identify the objects in view, a bounding box system has been created so that the data fed to the algorithm may be correctly labelled. This required the use of some complex functions and processes that extrapolate information from a two-dimensional view captured by the camera and, by using several sets of information from the world and its objects, such as camera position and orientation for example, and converting said information into a set of three-dimensional vertices through which a curve object will be passed, generating the seemingly two-dimensional rectangle bounding box.

From the position and angle the camera is currently operating from, said bounding box will seem to envelop the object in question in a close fashion to avoid confusion as to which object the bounding box is responsible for. In Figure 4.31 we can observe the process being applied to several objects.

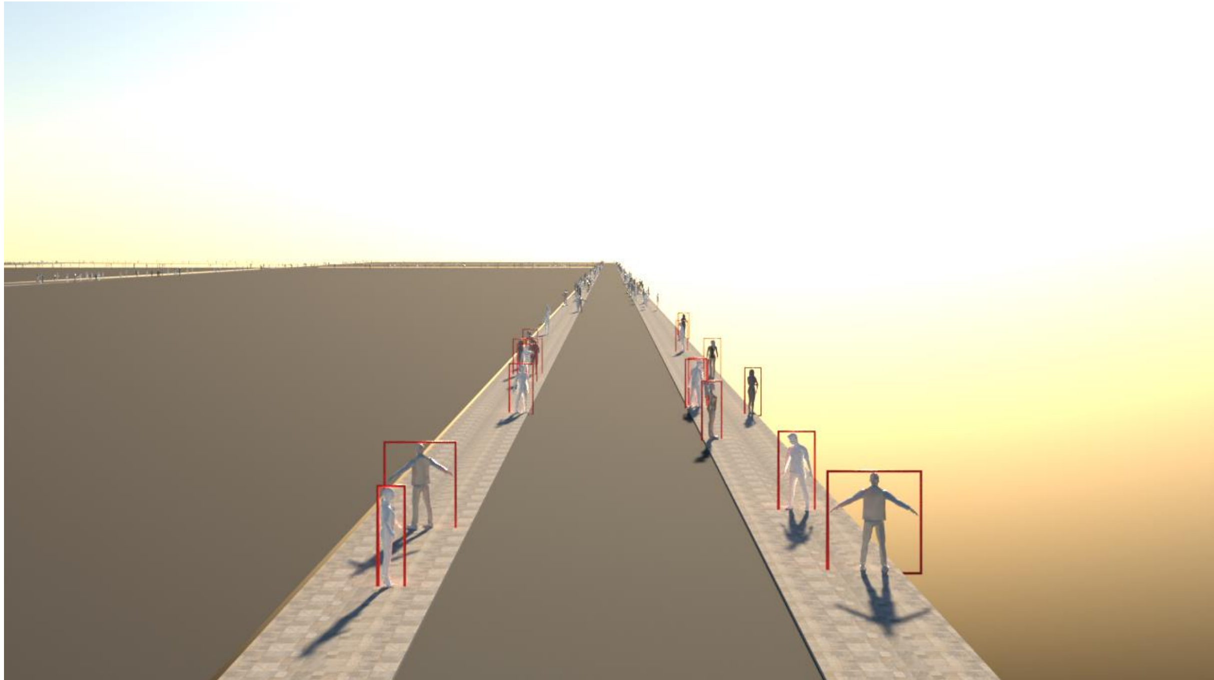

Figure 4.31 – Demonstration of the result of the bounding box process

After concluding the generation of the virtual world, the final process may take place, the camera system for world rendering into image format.
By creating the camera object with the lines "*Cam = Camera("Camera")*" and "*Cam.add_camera()*", the main focus changes to the steps needed to be taken to obtain a path for the camera to follow.
The best way to do this is to replicate the road curve objects, but since no way was found to duplicate or clone curve objects, here is where the "*generate_all_curves_only(roads)*" function of the Road class comes into play. By generating all roads without creating the meshes, the only biproduct of the function is the curves needed to draw a path, all joined into a single curve object as shown in Figure 4.32.
With the camera path concluded, all that remains is to randomly create a set of coordinates within the curve object for the camera object to visit and also generate a random set of orientation angles for the camera object to point towards, executing a render function at all points, with the line "*Cam.render(name)*".

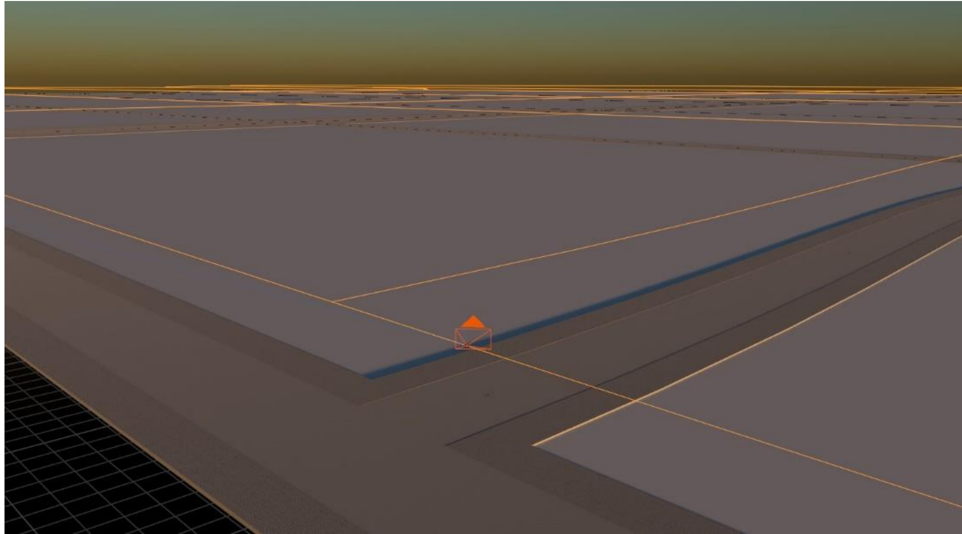

Figure 4.32 – Example of camera traveling through path created for render purposes

Depending on the amount of objects, their resolution, and their detail, the lighting conditions, the number of renders requested, and obviously, the processing capabilities of the computer running the project, the process may take some time.

This is to be expected as renders attempt to capture the virtual world with the highest quality possible, within the parameters in Blender. Lighting usually adds a vast amount of computation necessity as the program must calculate rays and ray bounces throughout the world, which is something very expensive to do. Also, since there is no level of detail aspect implemented in this project, no matter the distance to the camera, the objects in the world will always be rendered with their maximum level of detail.

Chapter 5

Future Work

As is the case with all projects I have ever been a part of, there is always more to do.
In this case specifically, there are four main components that require more work:

- Building variety, both in type and in meshes used
- Population variety
- Building collision, both between particles in the system and with road objects
- Bounding Boxes

These areas all fit within the objectives of the project and need more work to improve and bring more variety, complexity, and believability. The last one specifically helps my coordinator in developing his component that encompasses the overall bigger context of this project.
In terms of building variety, different types of buildings may help diversify the world, with houses and office buildings being some of the more obvious examples. As for the meshes, different types of windows, doors, walls, materials and textures, scales, orientations or even form, as all meshes present in the buildings are standardized into square forms that neatly fit into one another when generating the buildings, meaning the overall building creation is very similar. Some more irregularity or change in shape and volume may help.
Population variety works in a similar way but regarding finding, testing inserting and verifying more models. These can be more meshes of people, of fire hydrants, light posts, mailboxes, garbage cans, animals, and many other objects that may be found in the sidewalks. More vehicles will also help variety in populating the roads.
With building collisions is where the problems heighten. None of the methods found eliminated consistently both collisions between buildings and with sidewalks or streets. This can very easily break immersion when two buildings exist on top of one another, al whilst being placed in the middle of the sidewalk. Some problems arise due to the scale of the buildings, other due to the scale of the floor areas being targeted. The particle systems make it possible to have a vast number of buildings placed within the selected floor object, but also increase vastly the number of collisions between buildings and with road objects, acting as a double-edged sword. There certainly is a solution to this, but it remains illustrious to me.
Finally, bounding boxes. This area fits better in the extra work category but, nevertheless, it represents an important feature that still has a lot of possible improvements yet to be applied. The way this is used in machine learning algorithms that focus in using large amounts of labelled data is by making it much easier to identify the objects presented. A building would have a bounding box surrounding it with the label "Building", making it much easier to train said models. Whilst these have been implemented, this was done in a very rudimentary way and can be further improved on, especially in the readability and differentiation between each bounding box, to avoid confusing the algorithms that use them when two or more are presented very close to each other.
Besides all these main aspects, there is also a lot of work that could be done in general performance improvement, bug fixing, or better organization of the code.

Chapter 6

Conclusion

While the aim of this project can be found in other places, many of which executing it to a far greater extent both in scale and quality, I found none that attempted this in a solely code-based approach, relying usually in nodes or some external programs to achieve the same goal. It could be inferred that the means to which the goal is achieved do not matter as much as achieving said goal does, but I disagree. There is value in changing methods as it allows for other means of procuring the same goal that do not work well with certain methods. For example, the idea of having even more fine control over the interactions with Blender through it's "bpy" library mean that, even if in the current form, this project can be considered lacklustre when compared to other already existing similar products, with more time, resources, and effort, it can reach greater hights solely out of the flexibility it provides.
The scale this type of product can achieve is remarkably large and unfathomably complex, to the point that it was mostly unachievable to me with the current timescale and resources available, therefore, said scale had to be lowered into the realm of possibility. In a way, this project also serves as a proof of concept, delivering the tools and the guidance so that others, or even myself, can add to it and create something even greater.
The complexity of this cannot be underestimated. The amount of work and hours testing features, debugging, evolving, and innovating that took place in this project barely come through in the final product. Not because the final product is weak, but due to the effort necessary to produce it being spent in error correction and small details than, in the end, may not be immediately apparent to those who witness it. It is to be expected, as complexity usually drives the amount of work to rise at a much faster pace than the scale and quality of the products it creates.
This is not to imply the work done here is of pristine quality and no flaws are to be detected within it. There are many problems yet to be solved, minor issues to be dealt with and improvements to be had. All achievable with time and resources but, unfortunately, were not possible to solve before the deadline of this project's delivery. I still find the final version complete enough to be treated for delivery, but one thing I realized when developing it is that there is no such thing as complete in this area, there is always more to be done, more to create, to innovate, to add.
In terms of overall value this area of simulation has to add, just like its scope, there is practically no limit. From movies, to videogames, to urban design and engineering, to machine learning, and many, many more, all have aspects that can benefit greatly from technology such as this and bigger companies are certainly taking notice, as seen with NVIDIA.
This project, in its essence, is a simple virtual world generation tool that uses Blender and Python to help users have an easier way of producing said constructs. It generates streets and sidewalks, world floors, buildings, lighting and a way to export all of this to other places. It applies details such as materials and textures, it generates the different lighting conditions present at distinct times of day. It also allows for randomization or application of procedural elements allowing for even more distinct end products without having to create all by hand and, most of all, it is available for the world to try, use, improve and create, unlike most other similar products found during research on this area. That is, in my humble opinion, a great resolution. Giving others the tools to create and innovate, unshackled by pricing or complexity.

Bibliography

[1] Antonis Bouras. 2019. Future City Pack. https://blendermarket.com/products/future-city-pack

[2] Amed Bueno. 2021. Hippo. https://www.therookies.co/projects/29201?_ga=2.16031798.1080744759.1641231012-1532129142.1641231012

[3] Arnaud Couturier. 2009. SceneCity. http://www.cgchan.com Updated in 2021 for added support for Blender 2.9+ and 3.0 with version 1.9.

[4] Funso Sylvester Dorgu. 2021. princess Ira and Amina. https://www.therookies.co/projects/29502?_ga=2.48669734.1080744759.1641231012-1532129142.1641231012

[5] Juan Guillen. 2021. The Ancient Temple. https://www.therookies.co/projects/22813?_ga=2.43868197.1080744759.1641231012-1532129142.1641231012

[6] Andy Huang. 2021. Red Leaves. https://www.therookies.co/projects/24395?_ga=2.48669734.1080744759.1641231012-1532129142.1641231012

[7] julien Malgorn. 2021. Rookies Container - Weekly drill. https://www.therookies.co/projects/27957?_ga=2.16031798.1080744759.1641231012-1532129142.1641231012

[8] Innes McKendrick. 2017. Continuous World Generation in 'No Man's Sky'. https://www.gdcvault.com/play/1024265/Continuous_World_Generation_in__No_Man_s_Sky_

[9] Abhay Rajeev. 2021. Noire. https://www.therookies.co/projects/29299?_ga=2.48669734.1080744759.1641231012-1532129142.1641231012

[10] Jonathan Sauder. 2016. Procedural City Generation in Python. https://josauder.github.io/procedural_city_generation/#visualization-in-blender

[11] Francesco Siddi. 2021. Sprite Fright Open Movie. https://www.blender.org/press/sprite-fright-open-movie/

[12] Kent Trammell. 2019. How Rocket League Artists Use Blender for Game Production. https://cgcookie.com/articles/how-rocket-league-artists-use-blender-for-game-production

[13] Ubisoft. 2019. Ubisoft Joins Blender Development Fund to Support Open-Source Animation. https://news.ubisoft.com/en-us/article/1Fse1XyXzj76UJ1gKFohbz/ubisoft-joins-blender-development-fund-to-support-open-source-animation?isSso=true&refreshStatus=noLoginData

[14] scorpion81. 2022. Blender-Molecular-Script for 3.1/3.2. https://github.com/scorpion81/Blender-Molecular-Script/releases

[15] BorisTheBrave. 2019. blue-noise-particles. https://github.com/BorisTheBrave/blue-noise-particles/releases/tag/v1.0.1

Annexes

Computer Specifications and Blender Version and Settings

Personal Computer Specifications:

- CPU – AMD Ryzen 5 3600
- GPU – NVIDIA RTX 3080
- RAM – Corsair 16GB DDR4 3600MHz
- Motherboard – Gigabyte X570 Gaming X
- Storage – Samsung 970 EVO 1TB
- Windows 11 Pro Version 22H2 OS build 22623.1325

Blender Specifications:

- Version – 3.1.2
- Render Engine – Cycles (GPU Compute)
- Cycles Rendering Devices – GPU (NVIDIA RTX 3080) with OptiX
- Video Sequencer – 8192 MB Memory Cache Limit w/ Use Disk Cache option selected

Class Diagram

Here, we can observe a class diagram detailing the connections and relations between all classes in this project.