Creating Scenes with LLM Agents

Diogo Pedrosa
Master in Data Science
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

Creating Scenes with LLM Agents

Diogo Pedrosa

Dissertation carried out as part of the
Master in Data Science
Department of Computer Science
2025

Supervisor
Alípio Jorge, Full Professor,
Faculty of Sciences of the University of Porto

External Host Supervisor
Fátima Barros, Deep Learning Engineer,
Bosch Car Multimedia Portugal, S.A.



Acknowledgements

I want to express my gratitude to all those who supported me throughout the development of this thesis.

First, I thank my advisor, Alípio Jorge, for his invaluable guidance, encouragement, and insightful feedback throughout this work. His expertise and vision have been essential at every stage of this journey.

I am also very grateful to Fátima Barros, my supervisor at Bosch, for her support and helpful feedback.

I extend my thanks to all the people I had the opportunity to meet and work with at Bosch, whose collaboration and knowledge were truly enriching. I am also thankful to the broader community at FCUP, where I found a stimulating academic environment and many helpful interactions.

To my friends and family, thank you for your support, understanding, and encouragement. Your presence made this journey lighter and more meaningful.

This work was developed under Project ATLAS—Trusted Autonomous Navigation, supported by Portuguese National Funds (COMPETE2030), with Funding Reference COMPETE2030-FEDER-00646700.

Resumo

A automatização da criação e edição de cenas 3D a partir de texto tem elevada importância em múltiplos domínios. No domínio da condução autónoma, permite criar de forma mais rápida e económica os cenários em 3D necessários para o treino de modelos e validação de diferentes metodologias.

Esta tese aborda o desenvolvimento de um método para criar cenas 3D a partir de texto composto por linguagem natural, com recurso a modelos de linguagem de grande dimensão (LLMs). Estes modelos têm revelado capacidades notáveis na resolução de uma variedade de tarefas, demonstrando um potencial significativo para a geração e edição de cenas 3D a partir de texto.

A *pipeline* desenvolvida cria cenas 3D através do processamento de um prompt inicial que descreve a cena-alvo, o qual é subsequentemente traduzido para um ambiente 3D no Blender, usando uma biblioteca pré-existente de *assets*. O sistema adota uma arquitetura estruturada que utiliza um LLM como núcleo para o planeamento e execução, ao mesmo tempo que recorre a uma combinação de intrumentos disponibilizados e outros desenvolvidos para apoiar processos intermédios necessários.

A avaliação do nosso método demonstra que as componentes centrais da pipeline têm um impacto positivo no resultado final, possibilitando a construção de cenas 3D com elevada fidelidade ao texto fornecido pelo utilizador. Apesar de funcional, o método revela margem para melhoramento, uma vez que exibe erros de composição esporádicos e uma notória sensibilidade à forma como o texto inicial foi formulado.

Palavras-chave: Geração de Cenas 3D, Texto-para-3D, Modelos de Linguagem de Grande Dimensão (LLMs), Processamento de Linguagem Natural (PLN / NLP), Aprendizagem Automática, Inteligência Artificial (IA)

Abstract

A method that can automate the creation and editing of 3D scenes via simple text input is highly valuable across many fields. In the autonomous driving domain, it can enable a faster and more cost-effective means of generating the necessary 3D scenes for training models and testing various methodologies.

This thesis explores the development of a method to create 3D scenes from natural language text input, by leveraging Large Language Models (LLMs). These models have shown remarkable capabilities in solving a variety of tasks, demonstrating significant potential for text-to-3D scene generation and editing.

The developed pipeline creates 3D scenes by processing an initial prompt that describes the target scene, which is then translated into a 3D environment within Blender using a pre-existing library of assets. The system follows a structured pipeline that utilizes an LLM as the core engine for planning and execution, while also leveraging a combination of off-the-shelf and custom-built frameworks to support key intermediate processes.

An evaluation of our method shows that the primary components of the pipeline are beneficial, enabling the construction of 3D scenes that are well-aligned with the input text. While capable, the method also shows room for enhancement, as it exhibits some compositional errors and a notable sensitivity to how the input prompt is formulated.

Keywords: 3D Scene Generation, Text-to-3D, Large Language Models (LLMs), Natural Language Processing (NLP), Machine Learning, Artificial Intelligence (AI)

Table of Contents

List of Figures vii
List of Abbreviations x

1. Introduction 1
   1.1. Motivations 1
      1.1.1. 3D scene generation relevance 1
      1.1.2. Using LLMs for the task 2
   1.2. Objectives 2
   1.3. Approach 3
   1.4. Contributions 4
   1.5. Dissertation structure 5
2. Background 6
   2.1. 3D software 6
      2.1.1. Blender 6
      2.1.2. CARLA 7
      2.1.3. Unity 7
      2.1.4. Unreal Engine 7
   2.2. 3D formats 7
      2.2.1. Explicit representations 8
      2.2.2. Implicit representations 8
      2.2.3. Hybrid representations 9
   2.3. Textual representation of 3D scenes 10
   2.4. Transformers architecture 10
   2.5. LLM 12
      2.5.1. Historical Context 12
      2.5.2. What they are and how they work 13
      2.5.3. Multi modality 14
      2.5.4. Prompt Engineering 14
   2.6. Other Deep generative models 15
   2.7. Other approaches 17
      2.7.1. Rule based approach 17
      2.7.2. Math based approach 17
      2.7.3. Statistical approach 18
      2.7.4. Procedural generation 18

3. Literature 19
3.1. Methodologies 19
3.1.1. Asset selection 19
3.1.2. Reasoning structure 23
3.2. Scene textual structure 33
3.3. Evaluation 35
3.4. Datasets 39
3.5. Prompt engineering 40
4. Methodology and Development 44
4.1. Objective 44
4.1.1. Input description 45
4.1.2. General vs Domain-specific method 45
4.2. 3D Software and Format 46
4.3. Scene Construction Pipeline 46
4.3.1. Scene divider 48
4.3.2. Asset selection pipeline 51
4.3.3. Qualitative formulation 54
4.3.4. Function Library 55
4.3.5. Scene code generation 56
4.3.6. Feedback Loop 59
4.3.6.1. Scene graph processing 61
4.3.6.2. Visual evaluation 63
4.3.6.3. Constraint evaluation 65
4.3.7. Simple floor and weather maker 69
4.4. LLM Prompting 69
4.5. Scene editing pipeline 73
5. Results and Discussion 75
5.1. Experimental setting 75
5.1.1. Assets 75
5.2. Evaluation 76
5.2.1. Quantitative evaluation 77
5.2.1.1. Metrics choice 77
5.2.1.2. Description dataset 79
5.2.1.3. Ablation study 79
5.2.1.4. Method robustness over scene complexity 86
5.2.1.5. Run time analysis 88
5.2.2. Qualitative evaluation 89
5.2.2.1. Successful generations examples 89

5.2.2.2. Failures and limitations examples 92
5.2.2.3. Generation variability 95
5.2.2.4. Sensitivity to input variations 96
6. Conclusion 102
6.1. Research Summary 102
6.1.1. Key Findings and Remarks 102
6.2. Limitations 104
6.3. Future work 104
6.4. Final Considerations 106

Bibliography 106
A. Appendix 129
A.1. Prompts 129
A.2. Function library: All defined functions 153
A.3. Constraint evaluation: All defined constraints and report format 155
A.4. Generated assets 159
A.5. Blender physics simulator 160
A.6. Optimization pipeline 161
A.7. Automatic library construction 165

List of Figures

1.1. Objective Pipeline 4

2.1. Transformers baseline Architecture [18] 11
2.2. Major milestones in the development of large language models [19] 12

3.1. Scenecraft method pipeline [57] 24
3.2. LLM reasoning structure in [60] 25
3.3. Pipeline of LLplace method [65] 26
3.4. Pipeline of SceneMotifCoder method [73] 28
3.5. Example of interaction between agents and the user in WorldCraft method [75] 30
3.6. Pipeline schema from Scenethesis method [79] 32
3.7. Constrain definition in[87] 39
3.8. Example of usage of the ”Think step-by-step” method, also called Zero-shot- CoT. Few-shot- CoT refers to few-shot chain-of-thought, where chain-of-thought is another prompting technique in [92] 41
3.9. CoT example versus general prompting in [74] 42
3.10Least-to-most prompting example in [71] 43

4.1. Scene construction pipeline 47
4.2. Asset selection pipeline 52
4.3. Feedback loop pipeline 60
4.4. Scene edition pipeline 73

5.1. Generated assets examples 76
5.2. Image of a generated scene depicting a cat and a dog in a park bench. 80
5.3. Image of a generated scene with a woman riding a bike. 81
5.4. Distributions of performance metrics for each ablation method. The MOP-IoU metric is shown first, followed by pairs of related metrics for description to image alignment, and constraint satisfaction. The violins show the distribution shape, with internal dashed lines representing the quartiles and a white circle indicating the mean. 84
5.5. Scalability analysis of the full pipeline. Each plot shows the trend of the average metric score as a function of scene complexity (number of objects). 87
5.6. Run time in relation to the number of objects for the full pipeline and the pipeline without the feedback loop 88
5.7. ”The scene has a straight road in the middle, with 4 cars. 2 of the cars are on the right side of the road, while the other 2 are on the left side. On the left side of the road are 3 cones that are protecting a motorbike that had an accident. On the right side, there are 2 small houses close to each other, and a third house slightly more distant. Close to the motorbike, there is also a street lamp.” 90
5.8. ”One cat and a dog face each other, almost fighting on top of a park bench oriented towards the -x axis. Nearby should be a sunflower. Far away, a woman riding a bike looks at the cat and the dog.” 90
5.9. ”A table with four chairs around it, each oriented towards the table. On top of the table, a cat is sleeping. Nearby, two street lamps, far away on different sides of the table, illuminate the scene.” 91

5.10 "A car surrounded by 4 cones. Three people are side by side, not too far away, oriented toward the car. A good distance from the car, there are 3 bikes parked in a straight line, on the side of a tree. Behind the tree is a street lamp with a cat on top." 91
5.11 "Two parallel straight roads, aligned with the y axis. In between the roads should be a row composed of two trees and a bush in the middle. A motor bike passing on top of the right road, on the left lane, and two cars on the left road, on different lanes with opposing orientations." 91
5.12 "A junkyard composed of a bus is the center of the scene, with two cars on top of it, one in front of the other. A pile of three bikes near the bus, and behind the bus, a pile of one sports car on the bottom, one car in the middle, and another sports car on top. Very far from the bus, there is a man looking at the bus with two cones on its side." . 92
5.13 "A straight road oriented to the x axis with 6 vehicles on top. On top of the road, in the right lane oriented to the -x axis, is a line of vehicles with a motorcycle, a bus, and a car. On the left lane of the road, a line of 3 sports cars is oriented towards the x direction. On the left side of the road, two bushes are placed." 93
5.14 "This scene has a very long straight road made up of two road parts (one right in front of the other), forming a continuous block of road. Then, on each side of the road are 6 small houses, 12 in total. They are spaced evenly and facing towards the road. In front of the fourth house on the right, a bus is driving along the road." 93
5.15 "In the center of the scene, a table with four chairs, one at a different side of the table and oriented towards the table, near an umbrella. Not too far away, to the y direction, a complexly similar setup of table, chairs, and umbrella. To the x direction of on of the tables, and oriented to the y axis, a road with two motor bikes on top of it, in the left lane oriented to the y direction, and also on top of the road, now in the right lane two sports cars oriented to the -y direction." 94
5.16 "3 park benches facing each other forming a triangle. In between them, there is a small bush." 94
5.17 "A residential building, with some vehicles parked nearby. More precisely, the vehicles are all parked on the left side of the building, all side by side and aligned with the -x direction. The vehicles are in order: a car, a sports car, a bike, a bus, and three more cars. On the opposite side of where the vehicles are parked, on the side of the building, two trees are placed with a park bench in between." 95
5.18 "A residential building at the center of the scene oriented to the -x direction. On the left of the residential building is a store building, and on the right is a small house (all oriented in the same direction). In front of these buildings is a small park where 4 trees, 3 bushes, and two street lamps. In the center of this park are two park benches near each other, one oriented to the + y axis and the other to the -y axis. Further to the front of the buildings (very far) is a road oriented to the y direction. On top of the road is a car that, on its side, has a crosswalk sign and is waiting for a cat to cross the road." 95
5.19 Output variance for a single prompt. All three scenes were generated from the identical input prompt: "Make a scene of a central road oriented towards the x-axis. On top of the road should be a car in the left lane of the road oriented towards the x-axis and a bus on the right lane of the road oriented towards the -x axis. On the left of the road, a table should be in the middle distance. This table should have 3 chairs around it, each chair on a different side of the table and oriented towards the table. A store building should also be in place, so it is in the +y Direction in relation to the table and oriented towards the table. Finally, on the right side of the road, a row of trees should be present, containing 3 trees. " 96

5.20 Text input: ”A central road is oriented along the +y-axis. A red car is on top of the road, in its right lane, pointing in the same direction as the road. To the left of the road and at some distance, there is a table with a single chair nearby, facing the table.” ... 97
5.21 Two scenes generated by changing the initial prompt, so it introduces changes in the objects used. ... 98
5.22 Two scenes generated by changing the initial prompt, so it introduces changes to the number of objects. ... 99
5.23 Two scenes generated by changing the initial prompt, so it introduces changes to the order of the exposition. ... 100
5.24 Two scenes generated by changing the initial prompt, so it introduces changes to the words used. ... 101

A.1. Example of a before and after application of the Blender physics simulator in a generated 3D scene ... 160
A.2. Example of a before and after application of developed optimization method in a generated 3D scene ... 165
A.3. Automatic library construction pipeline ... 166

List of Abbreviations

2D two dimensions. 9

3D three dimensional. ii, iv, v, ix, 1– 10, 15– 19, 23, 28– 32, 35, 36, 39, 40, 44– 47, 51, 55– 57, 59, 63, 65, 73, 75– 79, 81, 82, 89, 92, 95, 96, 102, 103, 105, 106, 160, 165, 166

API Application Programming Interface. 4, 30, 46, 55, 59, 61, 74, 77, 103, 104, 153

AR Augmented Reality. 7

BBD Bounding Box Distortion. 21, 22

CLIP Contrastive Language–Image Pretraining. 20– 23, 36, 45, 46, 51, 53, 78

CoT Chain of Thought. vii, 41, 42, 57, 70, 71

FFNs Feed-Forward Networks. 11

GANs Generative Adversarial Networks. 15, 16

ID identifier. 49, 50

IoU Intersection Over Union. 37, 77

JSON Javascript Object Notation. 27, 33, 54, 66

LiDAR Light Detection and Ranging. 39

LLM Large Language model. iv, 3, 4, 6, 12, 15, 17, 19– 29, 40– 42, 44, 46– 53, 55– 57, 59– 61, 66, 69, 71, 75, 82, 85, 88, 92, 95, 102– 104, 163, 167, 168

LLMs Large Language models. iii, iv, 1– 4, 6, 10, 12– 16, 19, 22– 29, 36, 37, 39– 44, 46– 48, 56, 69, 77, 102– 104

LoRA Low-Rank Adaptation. 27

NeRFs Neural Radiance Fields. 8, 9

NLMs Neural Language Models. 12

OCR Optical Character Recognition. 20

ReLU Rectified Linear Unit. 11

SLMs Statistical Language Models. 12

ToT Tree-of-Thoughts. 43

VAE Variational Autoencoder. 16

VAEs Variational Autoencoders. 15, 16

VLLM Vision Large Language Model. 44, 51, 53, 60, 61, 74, 75, 78

VLLMs Vision Large Language Models. 47, 53, 63, 104

VR Virtual Reality. 7

1. Introduction

This section presents the main concepts of this dissertation.

First, we describe the motivations behind constructing a methodology that leverages Large Language models (LLMs) to generate three dimensional (3D) scenes just by using natural language. How and why methods like this are relevant in a variety of fields, namely the autonomous driving domain, that is our main focus. Then we will state the objectives and contributions of this dissertation, and finally conclude with the structure and contents of this work.

1.1. Motivations

1.1.1 3D scene generation relevance

Generation of 3D scene compositions is the task of placing and rearranging objects on a scene to get the desired composition. For instance, if we want to produce a composition of a house with a garden, a feasible composition for that objective would be something like a house in the middle with grass around it, maybe with a couple of trees or a garden bench. This is to say that a scene composition generically is a task of rearranging objects on a scene according to a logic that is determined by the context of the scene, the relations between objects, and how these objects usually relate to the space themselves (putting in more clear words this last statement with an example, a tree usually is grounded and is upright). It is straightforward to understand that in today's world, this task is needed in a variety of contexts like the game industry [1], interior design and architecture [2], or the autonomous driving domain [3].

The main challenge is that producing scene compositions is a time-consuming and difficult task when done manually. First, when making a 3D composition, it implies using specialized software depending on the context, like Blender, Unreal Engine, Unity, CARLA [3], etc., that adds a layer of complexity to the process. Second, planning the layout of the scene, object coordinates, and orientations to achieve the desired result takes time and effort, and depending on the context and needs, can be considered a skill that is not learned overnight [4]. Giving the example of the autonomous driving domain, there is a constant need for 3D scenes to train and evaluate the models, that today is far from being fulfilled (there is a lack of datasets for scenes in road context [5] [6]). This lack of scenes is hard to solve by just building and editing scenes in the conventional way, due to

the effort and resources that are involved. Due to these facts, a tool that could automate most of the scene composition-building process would be highly useful.

1.1.2 Using LLMs for the task

As we will see further in the next chapters, this task can be approached by a variety of different methodologies. So we will explain why we chose to use LLMs as the leading tool to solve this task. There are three reasons why this is our choice:

- First, given that the objective is to create a methodology that, with minimal input, can generate 3D scene in the way that the end user wants, we empower the user to convey their intentions to the tool in a natural, low effort way. LLMs are a great tool to achieve that, given that they work with natural language. The end user can just write their intentions for the scene, and with LLMs, those intentions can be understood and translated into a scene (it is noteworthy that this is not the only way of using natural language as user input in this task, but it is the currently most powerful one, as we will see in future chapters)
- The second reason to use these models is because of how powerful and versatile they are, being reasonable to assume that it is possible to use these models to construct a methodology to generate 3D scenes with great performance (and because it has already been tested, as we will see in the next chapters).
- The final reason is simply how relevant this kind of model is at the time of writing. It is not bold to say that LLMs are the models that the community has the most interest in, making it appealing to analyze solutions that involve these models. This is not only from a research standpoint, to grasp the full capability of these models, but also because solutions based on these models will only improve with time, given the constant efforts of the community to develop better iterations of these models.

1.2. Objectives

The main objective of this dissertation is:

- Build a pipeline that leverages LLMs to generate 3D scenes from simple text descriptions. The proposed pipeline should be both capable and efficient, and be able to build 3D scenes in the autonomous driving domain.

The secondary objective is:

- Develop an effective method for evaluating the constructed pipeline.

1.3. Approach

Our approach to accomplish the objectives of this dissertation can be described as follows:

1. Make an overview of the state-of-the-art approaches that leverage LLMs to automatically construct and/or edit 3D scenes. This is essential to understand the current landscape of methodologies so that we can better produce and refine our own. It is also very important to devise how to consistently evaluate the results, given that it is not trivial to evaluate a 3D construction methodology in a setting where no directly usable datasets are available.

2. In our proposed methodology, the user requests are in the form of text inputs. The user should be able to request the construction from the ground up of a new scene. Given this, it is necessary to find a way to successfully get the 3D elements to fulfill a given scene request.

3. After, we want to be able to generate the scene given the text input and the necessary 3D elements obtained in the step before, so the next step is to build a method that, using LLMs, can reason over the text input, and the available information to come up with a scene layout that respects the text input. This will involve some smaller steps like finding an effective way to store the scene information (object positions, orientations, and existing relations in the scene), and how to make the process that takes the input information to actually create and transform scene layouts.

4. With a plan of the scene layout created, we need a way to translate it into a ready-to-use 3D scene in specialized software, and not just a written idea formulated by the Large Language model (LLM). In figure 1.1, there is an overall representation of the objective pipeline.

5. With the pipeline done, validate the methodology and compare it to the state-of-the-art approaches.

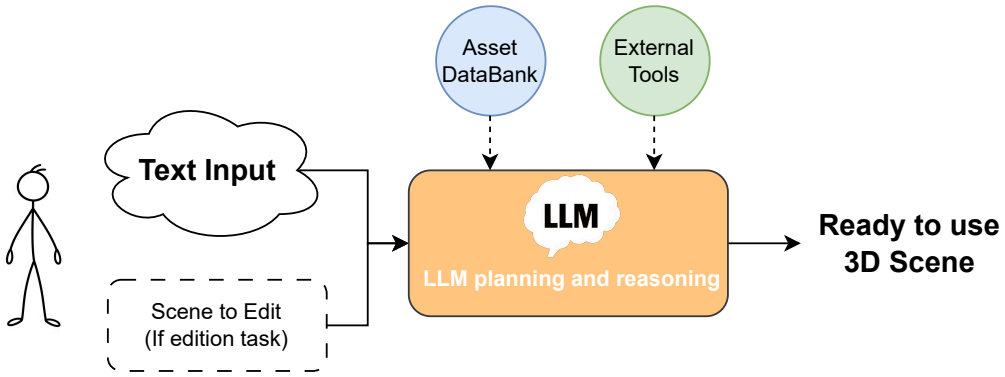


Figure 1.1: Objective Pipeline

1.4. Contributions

Summary of the contributions of this thesis:

- **A novel, modular pipeline for text-to- 3D scene creation:** This work introduces a complete design of a multi-stage pipeline that enables the creation of 3D scenes leveraging LLMs. This pipeline is a combination of a set of distinct tools, methodologies, and other custom pipelines (namely, an asset selection pipeline and the feedback loop pipeline) that intend to translate natural language descriptions into a 3D scene.

- **A Practical Implementation with Reusable Components:** Beyond the conceptual design, this thesis delivers an implementation of the pipeline to create 3D scenes leveraging LLMs. This includes many distinct features, mainly a variety of tailored made LLM prompts designed to accomplish many individual tasks, a custom function library leveraging Blender Application Programming Interface (API) to manipulate objects in Blender, and a constraint evaluation framework that scores the constraints in a given 3D scene.

- **A Comprehensive and Multi-Faceted Evaluation Methodology:** This work presents an evaluation of the developed methodology that enlightens the relevance of some key steps in this kind of method, while proposing and exploring other evaluation methods that are not usually used in literature for this specific topic. Namely, the ImageReward [7], VQAScore [8], SigLip 2 [9] ”score” and the introduction of a new metric, the Mop-IoU score, that measures the overlap on a scene independently of the scene scale.

- **Exploratory Research and Foundational Work for Future Extensions:** This thesis documents several different tools and methods that (while some of them were not rigorously evaluated, or did not improve the generated scenes) can provide valuable knowledge for the development of similar methods and define groundwork for further improvements. For example, a function library for object manipulation in Blender, a constraint evaluation framework that enables the evaluation of individual constraints on a scene, a pipeline that enables the edition of 3D scenes, a pipeline that tries to automatically add new functionality to the function library, and a optimization framework that tries to optimize the state of objects to reach the desired configuration.

1.5. Dissertation structure

The following content is structured into 6 chapters, which can be described as follows:

- **Chapter 2, Background:** The necessary background knowledge for a good understanding and contextualization of the research topic.
- **Chapter 3, Literature:** A detailed review of the available literature on the topic, detailing relevant methods, techniques, and tools important in this field of research.
- **Chapter 4, Methodology and Development:** The description and reasoning behind the developed method.
- **Chapter 5, Results and Discussion:** An exposition of the results achieved with the developed methodology, paired with a comprehensive analysis of these results.
- **Chapter 6, Conclusion:** A summary of the key remarks extracted from this study, an exposition of the main limitations faced during development of the method, and suggestions for future work.

2. Background

In this chapter, we will explain the different concepts, methodologies, and tools that are relevant to automated 3D scene generation and transformation using LLMs. First, we will briefly present and explain some tools that are important when dealing with 3D scenes. Namely, some software that is used to work with 3D scenes, and the different kinds of data formats that the scenes or assets can be encoded in. Then we will make a little introduction to the transformer architecture, given that it is the base of many model-based methodologies, including LLMs. After, we will briefly explain LLMs and relevant topics related. Finally, we will explain different kinds of methodologies, excluding the LLM based approaches, that we will analyze deeply in Chapter 3.

We will start with the model-based ones, referring to the main methods that are used, followed by non-model-based approaches, discussing their nuances and referencing representative examples.

2.1. 3D software

To make a 3D scene, we need an environment that lets us freely add, remove, or make transformations on objects, and most importantly, that enables the visualization of the built scene. This is what we are referring to as a 3D software related to our project. We will briefly mention some of the most popular or useful software for our use case. We will focus more on free-to-use options given their easier accessibility, and that have an inbuilt API, given that we want to integrate everything in a pipeline.

2.1.1 Blender

Blender [10] is a free, open-source 3D creation software used in various fields, from indie creators to professional studios. Its primary use cases include 3D scene creation and editing, modeling, animation, rendering, sculpting, and visual effects. Blender also offers a Python API, making it suited for automation and integration into pipelines. Blender's accessibility and large tool set, easy-to-use Python API, and being very lightweight, make it the most used software to make automated general-purpose 3D scenes methodologies.

2.1.2 CARLA

Because the main objective of this work is to produce scenes in a road context, it is also relevant to refer CARLA [3]. This is an open-source simulator for autonomous driving research, focusing on road context environments and vehicle interaction. While its primary use case is in robotics and AI research, CARLA's underlying 3D scene-building capabilities make it a strong tool for creating realistic and dynamic road context landscapes. CARLA includes a Python API, which facilitates integration with machine learning pipelines and external control systems. For these reasons, this software is especially used in 3D generation methodologies that focus on road context. Although it is not as straightforward and lightweight as Blender.

2.1.3 Unity

Unity [11] is a free-to-use development platform widely employed in game development, Augmented Reality (AR)/ Virtual Reality (VR) applications, and interactive 3D scene creation. Its primary strength lies in its ease of use and flexibility, supported by extensive documentation and tutorials. Unity is well-suited for creating interactive 3D scenes due to its modular approach and the Unity Scripting API, which supports C# and can be extended to other languages.

2.1.4 Unreal Engine

Unreal Engine [12] is a free (with royalties on commercial revenue) real-time 3D engine known for its exceptional graphics and rendering capabilities. It is extensively used in game development, film production, and architectural visualization. Unreal Engine's strengths include its real-time ray tracing, dynamic lighting, and asset library, making it ideal for building high-detail 3D scenes. The Unreal Engine also includes a robust API in C++. It is important to point out that this is by far the heaviest tool described.

2.2. 3D formats

Besides the environment that lets us manipulate the scene at will with the necessary tools, it is important to talk about the actual formats that the individual 3D elements (also known as assets) and scenes can be encoded with. There are multiple different types of formats that have different characteristics, so it is important to make a brief analysis of the most relevant ones, given that we will need to choose a format to work with. We will

separate these different formats into 3 distinct categories: Explicit representation, Implicit representation, and Hybrid representation. [13]

2.2.1 Explicit representations

Explicit representations are overall the most used type of 3D representation. This is because they are easy to use and understand, leveraging basic primitives like points or meshes that give detailed 3D elements paired with an understandable visualization.

Let's first see what **mesh representation** is. This type of representation, in the most basic form, is all about depicting 3D elements as a connection of vertices with edges. The representation can then be further enhanced by including polygons, typically triangles or quadrilaterals. This achieves very good representations of 3D elements, which are easy to manipulate and lightweight to be rendered. This is the most used representation, being compatible with every main 3D software. [13]

Now the **point clouds**. In this representation, 3D elements are depicted as a set of discrete points in 3D space (each point can have different attributes like color). This representation is particularly useful for reconstructing real-world environments or objects, given that it excels in capturing highly accurate spatial data from sensors such as LiDAR or 3D scanners. This is very relevant given that for autonomous driving, the data is almost completely taken with these sensors that map points in space, so it is relevant in the end to be able to have the generated scenes in this format, although it is important to note that this representation demands more computational power. [14]

2.2.2 Implicit representations

Implicit representations define 3D objects and scenes using continuous mathematical functions rather than discrete primitives like vertices, edges, or points. Unlike explicit representations, which primarily focus on object surfaces, implicit methods can describe the entire volume of a 3D object. By leveraging volume rendering techniques, these representations allow for realistic image synthesis.

One of the most prominent implicit representations is **Neural Radiance Fields (NeRFs)**. NeRFs are neural networks that encode a 3D scene as a volumetric function, mapping spatial coordinates and view direction to density and radiance values. This allows them to synthesize highly photorealistic images from arbitrary viewpoints using volume rendering techniques. NeRFs excels at capturing fine details, lighting variations, and complex

textures. However, the computational demands of NeRFs are significant, requiring substantial resources for both training and inference. Despite this, they have revolutionized 3D graphics by achieving state-of-the-art results in rendering complex scenes. [15]

Another key type of implicit representation is **Neural Implicit Surfaces**, which focuses on defining object surfaces as continuous fields. These surfaces are represented as level sets of a function, where the zero level typically defines the object boundary. Neural implicit surfaces are well-suited for tasks like surface reconstruction and shape modeling, as they produce smooth, continuous representations with compact storage requirements. While extracting explicit surfaces like meshes from these representations requires additional processing, their flexibility and ability to represent complex shapes make them a valuable tool in 3D graphics and modeling. [16]

2.2.3 Hybrid representations

Hybrid representations aim to combine the strengths of explicit and implicit methods to achieve a balance between performance and versatility. By using discrete structures for simplicity and efficiency alongside continuous functions for flexibility and detail, these representations offer compelling solutions for many practical applications.

One widely used hybrid representation is the **Voxel Grid**. Voxel grids represent 3D elements as a uniform grid of volumetric cells (voxels), each containing attributes like density or color. While this approach is intuitive and compatible with traditional graphics pipelines, its major drawback is the high memory consumption, especially for large or highly detailed scenes. To address this, sparse voxel representations are often used, allocating memory only to occupied or relevant regions. Voxel grids are particularly useful in applications like medical imaging and robotics, where regular grids simplify computations and processing. [13]

Another notable hybrid approach is **Tri-Plane Representation**, which encodes 3D elements using three orthogonal two dimensions (2D) planes that store spatial information. This method is compact and computationally efficient while maintaining the ability to represent fine details. Tri-plane representations are especially effective in neural rendering tasks, as they offer a structured yet lightweight format suitable for integration with deep learning frameworks. By blending the compactness of 2D grids with 3D spatial coverage, tri-planes have been successfully employed in generating realistic scenes and objects in tasks like video synthesis and generative modeling. [14]

2.3. Textual representation of 3D scenes

We saw how to represent 3D elements and the software that could be used to manipulate these elements. Still, given that we will use LLMs to create and reason over scenes, it is important to have some kind of textual form to represent 3D scenes. And that is when scene graphs come into play.

Scene graphs can be described as a systematic and structured way of describing a scene. There is an example in 3.2 aligned to our context. Fundamentally, the core elements of a scene graph are the following: the objects that are on the scene, the attributes of the objects, and the relations between objects. For the attributes of the objects, we usually have two types of information. The ones that describe the object itself, like sizes, colors, material, etc. And the ones that describe how the objects are in the scene, like coordinates and orientation. The relations, in the most simple form, are expressions that involve two objects and tell us about the relative position and orientation between the two objects. Giving a simple example, ”A bottle on top of a table” is a relation between a bottle and a table. And we gain knowledge of the height coordinate of the bottle being superior to the height of the table, and also that the base of the bottle is in contact with the top of the table. The graph terminology arises from the fact that the mentioned structure of information can be organized as nodes, which represent the objects and the respective object attributes, and edges, which are the relations between objects.[17]

2.4. Transformers architecture

Before presenting a broad explanation of LLMs and other models, it is important to dive into the Transformer architecture for neural networks, given that this architecture forms the backbone of many models, including the most significant LLMs.

The main selling point of the transformer architecture with respect to previous architectures is the high scalability because of an efficient structure to leverage parallel computing and the very capable built-in attention mechanism. We will provide a quick overview of its structure. The overall structure is represented in 2.1

Let us start with the key elements, and then see how they are put together. The self-attention mechanism allows every element of the input to interact with and attend to every other element, enabling the model to capture the global context. Then, what is called Multi-head Attention layers can execute multiple Self-Attention Mechanisms in parallel, allowing the model to focus on multiple key points of the input simultaneously.

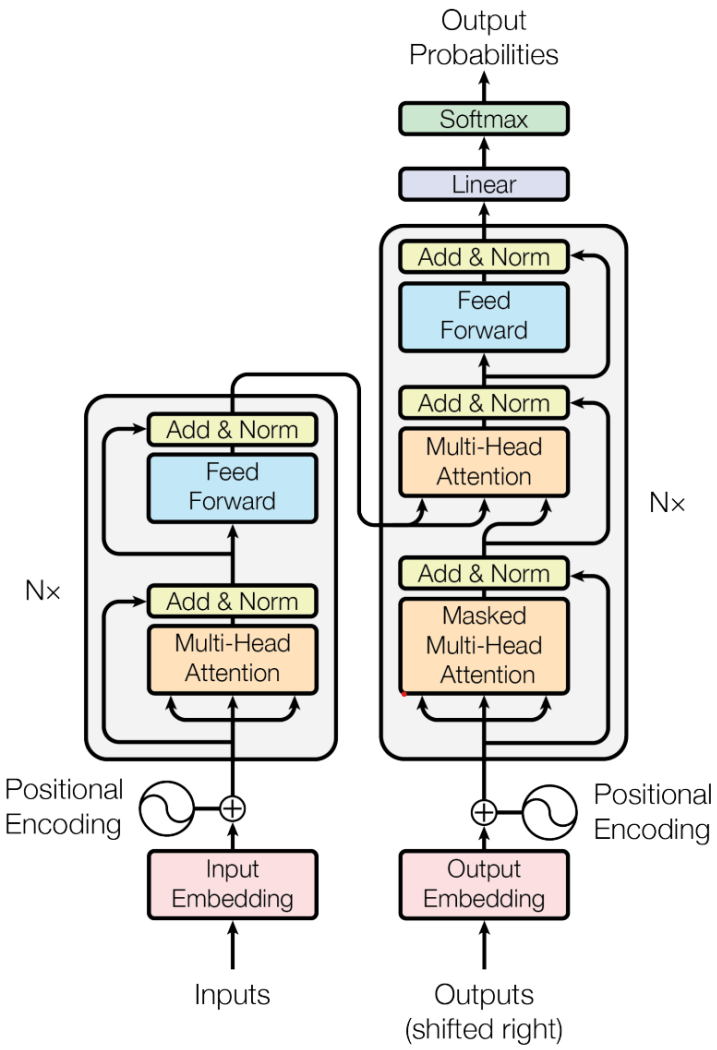


Figure 2.1: Transformers baseline Architecture [18]

Next, we have the Feed-Forward Networks (FFNs), which are applied independently to each position in the sequence after the attention mechanism. These FFNs consist of two linear transformations separated by a Rectified Linear Unit (ReLU) activation, adding non-linearity and depth to the model. Then the positional encoding method enables the architecture to understand the order of the elements of the input.

The Transformer architecture is structured into two main components: the encoder and the decoder. The encoder processes the input sequence through a stack of identical layers, each composed of multi-head self-attention and a feed-forward network. The decoder is similarly structured but includes two types of attention: masked multi-head self-attention to process partial outputs while preventing access to future ones, and encoder-decoder attention to incorporate information from the encoder's output. Positional encoding is added to the input sequence before it is passed to the encoder or decoder.

Finally, the outputs of the encoder and decoder layers are combined to generate the final prediction. The encoder builds a rich, context-aware representation of the input, while the decoder uses these representations to generate outputs step-by-step, attending to both the input sequence and previous outputs [18].

2.5. LLM

2.5.1 Historical Context

LLM does not refer to a single unique architecture capable of handling natural written language. Instead, it is the result of various inventions and breakthroughs made over the decades. In figure 2.2 is a simplified introduction to the history of LLMs.

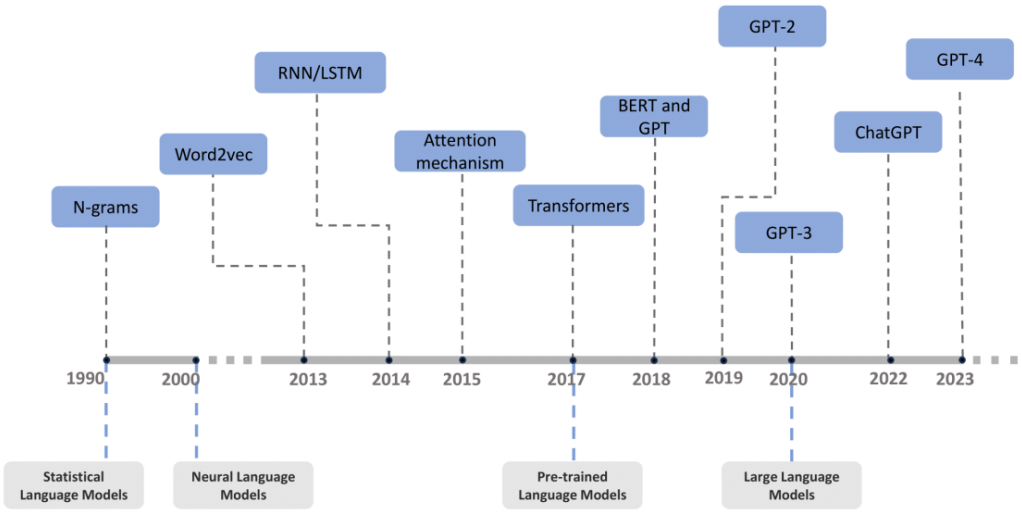


Figure 2.2: Major milestones in the development of large language models [19]

The Statistical Language Models (SLMs) were widely adopted around the 1990s [20] (though their origin is significantly older, dating to the 1940s [21]). These were simple mathematical models that used statistics to model relevant properties of the text, namely the probability of a sentence in a given text using a simple conditional probability formulation. In subsequent years Neural Language Models (NLMs) [22] appeared, and instead of relying on classical statistical methods to model relevant properties of the text and predict probabilities, they used simple neural networks. These models were more capable and scalable, as they avoided the exponential growth in complexity associated with the number of words in the input.

The next big thing was the conversion of words into a vector space, where closer words have similar semantics and vice versa, the Word2Vec [23]. This conversion, paired with

neural networks, led to further improvements in the performance of the models. The introduction of attention mechanisms in neural networks [24] and the introduction of a very capable new neural network architecture with a lot of potential for scalability, the transformers [18] led to the development of the first pre-trained language models like GPT-2 [25] and BERT [26]. These models were trained on very large unlabeled datasets, leading them to grasp fundamental structures and logic behind the text in the data. Then these models could be adapted to a variety of NLP tasks like questioning and answering, text summarization, etc, with less demanding fine-tuning. The performance of these models was already very impressive.

What comes next is really the breakthroughs for LLMs, where they started to demonstrate incredible capabilities, starting to invade mainstream media and build up all the interest and investment that there is nowadays, namely with GPT-3 [27]. There are some differences between the LLMs and the first pre-trained language models, but the biggest difference comes down to the scale of the models, the first having many more parameters and being trained in much larger datasets. From this point, the performance of the models just skyrocketed due to increased interest and investment. At the time of writing, modern iterations include the GPT-o1 [28], Gemini 2.0 [29], Llama 3.1 (405B) [30], Claude 3.5 Sonnet among others.

2.5.2 What they are and how they work

The core structure of the LLMs is the transformer structure. Some LLMs don't use this core structure, but overall, the best LLMs at the time of writing are based on this architecture. Now, there are additional key components and processes of these models that are worth noting.

The first important concept is tokenization. These models do not process text or characters directly. Instead, they work with an encoded representation that is more computationally efficient and easier for the model to process [31]. The tokenizer is responsible for converting raw text into tokens—units of text such as words, subwords, or characters. These methods strike a balance between efficiency and preserving meaning by breaking phrases or words into smaller units when necessary. For example, the word ”unbelievable” might be tokenized into ”un,” ”believ,” and ”able.” The tokenizer is needed before the model to convert the input into a specific format. And after that, to convert the generated output back into human-readable text.

Another critical aspect of LLMs is how they are trained to develop their capabilities. Training typically occurs in two phases:

1. First, these models are pre-trained on vast amounts of diverse, unlabeled data, allowing them to learn general linguistic patterns, syntax, semantics, and world knowledge. This phase gives the models a broad understanding of language and context.
2. Then they are trained even further in a smaller sample of data, to make the alignment with human values, and to optimize them to follow instructions [32].

We saw in a very simplified way what LLMs are, and the core mechanisms behind them, but this is not enough to understand how they work. At a high level, their operation is straightforward: they predict the most probable next token based on a sequence of input tokens. This capability allows them to model relationships, patterns, and structures in language. However, the emergence of higher-level reasoning, creativity, and knowledge within these models is not well understood, remaining an area of active research [33]. In essence, by generalizing from the patterns observed during training, LLMs can produce coherent and contextually appropriate outputs across a wide range of tasks.

2.5.3 Multi modality

We mentioned LLMs as models that deal with text, but there are iterations of these models that can handle a variety of data modalities, making them multi-modal. Some examples include LLMs that can interpret images directly, like GPT-o1. Others can even understand and generate audio, like the o1-preview from OpenAI. For this dissertation, we are especially interested in the multi-modal models that can look at images, or even 3D data [34], given that they can understand and reason over the scenes or assets directly.

2.5.4 Prompt Engineering

To use LLMs, we write a piece of text containing the information and/or instructions, to then give as input to the LLMs. This text is called a prompt.

Now, written text is very nuanced, and it is possible to transmit information in numerous ways. Given this, the process of creating a prompt to transmit an idea to the model can be approached in many different ways. Prompt engineering is the process of optimizing how prompts are crafted to achieve better results. It can even be said that prompt engineering

is a way to program the LLMs with certain objectives in mind. Prompt engineering is not limited to a single prompt. It can be a multi-prompt process to achieve a goal. [35]

Note that this is not a methodology exclusive to LLMs. For example, diffusion models that take text as input for generating outputs can benefit from prompt engineering techniques, albeit differently [36].

Prompt engineering can be relevant to the method we want to develop, as it can allow us to optimize how we make the LLM perform certain tasks in our pipeline to achieve better results. And the pipeline structure can also be designed with these kinds of techniques in mind.

Prompt engineering holds significant promise, but there are important nuances to consider. A technique that yields good results in one context may not necessarily work in others. This can happen for several reasons, such as the technique being effective only for a specific model or family of models, sensitivity to parameter settings (e.g., temperature or random seed), by introducing small changes to the technique, it can make it not effective, or by changing the context, originates worst results. Since the mechanisms behind LLMs reasoning capabilities and knowledge representation are still not fully understood, it is difficult to predict or explain such phenomena. Therefore, it is relevant to examine how these techniques were utilized and validated before applying them in new scenarios [37], [38].

2.6. Other Deep generative models

Our objective is to build 3D scenes using LLMs, but it is relevant to examine other kinds of models that can be used for this purpose.

First, let's see what deep generative models are, given that most of the model-based approaches use these kinds of models. Deep generative models use deep neural networks to learn the underlying probability distributions of datasets, enabling the generation of new, similar data samples. There are a variety of models that fit this notion. Apart from LLMs that can be considered deep generative models, we have, for example, Variational Autoencoders (VAEs) [39], Generative Adversarial Networks (GANs) [40], transformers, or diffusion models [41], among others.

The use of the mentioned models for the intended task is far different than LLMs. LLMs in this context function like a reasoning agent that, given the information available, will rationalize over it to find a solution that fits the request provided. These other models are

usually used to generate the scenes or the objects directly. In other words, the output of this model is data in 3D format. However, generating a scene directly often does not yield optimal results with current models, so most of the time, there is some sort of process to make a guide to help the model generate the scene. The backward can also be made, first the assets are generated, and then reasoning over them to be placed on the scene. This last is quite similar to our desired approach, given that an asset selection is replaced by asset generation, and then the reasoning can be done by LLMs. We will briefly explain some of these different approaches:

VAEs are models that compress data into a latent space (a smaller space that encapsulates the key features of the data) with an encoder and then reconstruct the data using a decoder. If trained on 3D data, it is possible to sample points from the referred latent space to originate new 3D data. Examples include the method in [42], which uses a Variational Autoencoder to encode a scene graph into a latent space, to construct an indoor layout with the help of constraint-based methodologies. Another example is [43] that uses Variational Autoencoders with a diffusion model to generate and augment 3D scenes. This time, the generation is mostly made by the diffusion models, but the denoising is made in the latent space created with a Variational Autoencoder (VAE).

GANs consist of two networks: a generator that creates synthetic data and a discriminator that evaluates whether the data is real or fake. These networks are trained together, with the generator improving to produce outputs that the discriminator cannot distinguish from real data. This adversarial process enables GANs to generate high-quality outputs. As an illustration, this work [44] takes as input semantic-segmented depth images, and the GAN model generates the scenes depicted in the images (although it is just made for indoor scenes).

Transformers, when used for 3D scene generation directly, essentially encode the 3D scenes into a sequence of tokens, to then apply the mechanisms already explained in section 2.4. As an example, this method [45] takes as input the room type and its floor plan, and generates indoor room scenes using transformers.

Diffusion Models start with random noise and gradually refine it into structured data through an iterative denoising process. During training, the model learns how to reverse the process of adding noise to data, modeling how meaningful data emerges from randomness. Maybe the most popular use cases for these models are the generation of

images, like stable diffusion [46], video, like SORa [47] from OpenAI, or audio, like DiffWave for audio synthesis [48]. But turns out they are also very capable in 3D generation. These are some specific iterations of diffusion models used for 3D scene generation. This work [49], includes bounding boxes organized in space with the description of the object as input, and then the diffusion model will generate a whole scene where each bounding box is the object generated with the associated prompt. GraphDreamer [50], on the other hand, has textual input and leverages an LLM to make a scene graph of the scene. And then the diffusion model, with the scene graph, generates the scene. DiffuScene [51] can make and edit indoor scenes, and to support natural language instructions, it just uses a pre-trained Bert encoder to extract the embedding of the instructions a gives it to the diffusion model.

2.7. Other approaches

2.7.1 Rule based approach

Rule-based approaches in the 3D scene generation topic are related to methods that use defined rules to construct the scene layout. For instance, imagine that we are doing a scene that involves roads and cars. One rule could be that if cars and roads are present, then the cars should always be grounded on the road.

For illustrations, these two works [52], [53] use this approach. The first one, in a shared instance, deploys agents, each one related to one object and a preset of rules. Then the agents will act in the shared environment until the rules are satisfied, and in the end, a scene layout is constructed. In the second, a spatial database that has the rules and relations with each kind of object is used. Then, it can detect the instructions and objects from a text input with a non-model-based methodology and make the scene with the rules in the database.

Full rule-based approaches are older and, in general, less capable, but still, elements of this approach are used in more capable methodologies.

2.7.2 Math based approach

Math-based approaches use mathematical formulas to construct some form of object configuration. This can arise in the form of actual formulas to make the placements, or some sort of optimization algorithms to get object placement. In nature, they are not so different than the rule-based methods but allow for a little more generalization.

For illustration, this work [54] uses a Greedy Cost Minimization algorithm to search for a good object configuration for certain objects given a set of aesthetics and functionality aspects.

2.7.3 Statistical approach

Statistical-based methodologies usually employ a way of modeling the overall distribution of the arrangement and relations of objects on a set of already-made scenes, to then apply that learned underlying statistical distribution when constructing new cases.

This specific work [55] uses max entropy to get an overall distribution of how the objects are arranged using a set of pre-made scenes. Then, based on these distributions, it can create new scenes.

2.7.4 Procedural generation

Procedural generation is not a novel approach, but more a widely used term to refer to methodologies that use non-model-based approaches. Most of the time, they can be described as combinations of rule-based, mathematical, and statistical approaches.

Procedural generation is widely used in many contexts nowadays as fast and effective 3D scene generation with minimal input /guidance.

Giving a clear example, this work [56] implemented a procedural generation methodology in Blender, using the inbuilt node system. It defines rules and mathematical formulas to not only generate 3D compositions but also make unique objects.

3. Literature

In this section, we will provide a review of the state-of-the-art methodologies and tools for 3D scene generation using LLMs. We will analyze the state-of-the-art pipelines presented in the literature. Then, we will focus on evaluation methods, which are far from trivial in this context. Lastly, we will examine specific prompt engineering methods that appear to be suitable for our use case.

3.1. Methodologies

This section provides an overview of the pipelines used in different methods found in the literature. To better structure this analysis, we will employ some categorization for the various steps of the pipelines.

- On how the assets required for a given request are obtained
- The reasoning structure that is used leveraging LLMs, that goes from the initial input to the final planned scene.

3.1.1 Asset selection

Asset selection methods are defined as a way to select the most appropriate assets for a given task from a group of available assets. This excludes methods that explicitly generate 3D content, rather than rearranging pre-made assets in a 3D space.

Several methods have been introduced in the literature to address this topic. We will analyze the ones that we found most appropriate.

CLIP based methods: Let's begin with a method used in SceneCraft, introduced in [57]. For initial context, this work has a very similar objective as ours, leveraging LLMs to construct 3D scenes with natural language requests. The method starts by employing an LLM for deriving a list of necessary assets and respective descriptions from the initial text input made by the user. After having the necessary assets and respective descriptions using the LLM, two steps are made for each asset: First, it chooses the top n assets from a database composed of assets that have a description that better fits the wanted one. In this step, there are two things to note, first is not explicitly mentioned how the resemblance of the descriptions to the objective is actually addressed, and this implies

that pre-made descriptions of the assets in the repository are available beforehand. To address the top n selected candidates with the goal of choosing the best one, a model called Contrastive Language–Image Pretraining (CLIP) is used. To explain how this is executed is important to understand how this CLIP model works.

Introduced in [58], CLIP models are trained by predicting which caption goes with which image, to learn an image representation that is encoded in the same space as the text representation. By doing the previous training, these models then can receive both textual information and images, and encode both types of data in a common space in a meaningful way (for example in this shared space, an embedding created from an image of a dog and an embedding of a caption saying ”dog” would be somewhat similar). This functionality of the CLIP models made them effective for a variety of tasks like Optical Character Recognition (OCR), action recognition in videos, geo-localization, and many types of object classification, as mentioned by the authors. Turns out it is also very good at image+caption evaluation. [59] This image+caption evaluation made with CLIP models works in the following way: First, both image and text description are processed through their respective CLIP feature extractors to produce embeddings for each of them. After the cosine similarity between the embeddings obtained from the image and the caption is calculated. This numerical value measures how well the caption and the image fit each other and seem to be aligned with human judgment.

This can be translated to the following formula:

$$\text{CLIP-S}(c, v) = w \cdot \max(\cos(c, v), 0) \tag{3.1}$$

where v is the caption embedding, c the image embedding. In the reference paper [59], they also introduce the weight w, which represents a scaling factor in conjunction with the max clause, just to make the result easier to interpret.

To the asset selection method, for the top n assets, the CLIP-S between the wanted description and the selected assets' representative images is calculated. The asset with a higher score is selected, being the one that should be better aligned with the original user intentions.

Another approach that uses CLIP model is presented in [60]. This method is a little more convoluted than the last one, but the principles are the same. This time, the scene layout was already planned, but at this point, specific assets are not used, instead, the bounding boxes of the wanted objects are predefined by an LLM, alongside a description of the

wanted object and the category (bed, chair, table, etc). Next, from a large database of assets with the embeddings corresponding to the images of renders of the assets already pre-calculated, the embedding of the descriptions is calculated using a CLIP model, and the top 100 assets are retrieved for each object ordered. Then, to optimize the category selection, they apply two methods. First, they re-order the assets by considering a formulation that uses the similarity between the description and the asset, plus the category and the asset defined as:

$$d(v, c_{desc}) + Yd(v, c_{category})$$

where c_{desc} is the description of the wanted asset, $c_{category}$ is the category of the wanted asset, and Y is a factor that, in their use case, they define as 1. After, they use a multi-modal vision LLM to select m objects of the correct category. More in detail, they give the model images of the objects, starting with the top 1, and select it if the LLM judges it as a member of the correct category until m are selected. After this step, there are still two more quality assessment steps to make the selection even more consistent. First, a multi-modal LLM is employed again to judge if an object is composed of multiple objects or not, excluding if it is (giving an example, if we want a TV, and the asset includes a TV in a stand, this asset would be discarded). The other step measures how well the bounding box of the retrieved objects fits the wanted one.

To measure this fit of the bounding boxes, the Bounding Box Distortion (BBD) is used. BBD compares the side lengths of the bounding boxes along the x, y, and z axes as:

$$BBD(B_r, B_p) = \sum_{i \in \{x,y,z\}} \left| \log \frac{B_{r,i}}{B_{p,i}} \right|$$

where B_r is the bounding box of the retrieved object, B_p is the bounding box of the object as specified in the program, $B_{r,i}$ is the length of the bounding box B_r along axis i, where $i \in \{x, y, z\}$ and $B_{p,i}$ is the length of the bounding box B_p along axis i, where $i \in \{x, y, z\}$.

Since the object's orientation might not align with the desired specification, different rotations are applied to the retrieved object. The goal is to find the rotation that minimizes the BBD.

The minimal BBD after applying a rotation is:

$$mBBD = \min_{R \in R} BBD(R \cdot B_r, B_p)$$

where $R \cdot B_r$ refers to the bounding box B_r of the retrieved object after applying rotation R, being $\{(0,0,0),(0,90,0),(90,0,0),(90,0,90)\}$ the set of rotations.

If the minimal BBD ($mBBD$) is lower than a threshold τ, the object is considered a good choice. The threshold is typically set to $\tau = 0.4$.

This method is in the context of using a very large and variate dataset, and that is why the authors made it convoluted, but this is to show that this simple act of choosing the assets can become very complex.

Methods that use CLIP models, in today's standard, are the most common approaches. It is reasonably expected, given the good alignment with human preferences, versatile use, and because it's a model that can be used out of the box while not being computationally expensive. For reference, there are some more methods that use this model [61] [62].

LLM based approaches. We already saw some methods that use LLMs for object retrieval, but it is important to address this matter with more care. First, it is possible to use just normal LLMs for the task. This will resume to compare different object descriptions and labels to see the best fit for the wanted objects (this method is just not very doable when working with a large number of assets). Or just give the LLM the task of specifying the objects needed, to then do a basic search in a database of already labeled assets.[63] [64] There is also the use of Multi-modal LLMs, namely LLMs with vision capability, that can look at images of the assets directly and reach their conclusions. Another example is this work [65] that uses a simple text matching retrieval (without specifying the specific method) over already annotated objects using the GPT-4V.

Bounding box approach. For methods where the bounding boxes of the wanted scene are planned beforehand, it is common to use methods that calculate distances and/or deviations using simple arithmetic formulations to see the asset that best fits a certain bounding box. We already saw the use of BBD, but another commonly used method is the Euclidean distance in relation to the sizes of the boxes.

$$d = \|s_{\text{cand}} - s_i\| \tag{3.2}$$

where d is the Euclidean distance between the bounding box dimensions, s_{cand} represents the bounding box dimensions for the candidate object from the dataset, s_i denotes the bounding box dimensions of the target object in the scene. This is, for example, used in

those two methods [66] [67].

The methods introduced were the most commonly used in related works, but multiple different methods could be employed for this task. There are multiple ways of getting similarity between text [68], other ways of measuring similarity between images and text [69], and there are even methods that can have as input the 3D models directly, like iterations of the CLIP model that can receive 3D data [70], or even LLMs that have as input 3D data [34].

3.1.2 Reasoning structure

The reasoning of the pipelines in our context can usually be divided into two main sections. The qualitative reasoning over the request. This will include the interpretation of the user input, and a qualitative planning of the scene, where the relations over objects can be addressed, the removal or addition of new objects, possible transformations on the present objects, and others (basically everything that can be done to plan the course of action without deciding any specific coordinates and orientations). The other section makes the decisions on the coordinates and orientations. But it is noteworthy that there are many different methods, so this segregation is not explicitly present every time, or at least without some level of overlap.

There is also another aspect that is important to take into account before going into specific methods. The reasoning, especially the qualitative reasoning, most of the time is related to some prompt engineering technique and some form of a textual format that encodes scenes. We will mention shortly some of these elements in this section, but the details will be addressed in the respective following sections.

Referring again to the SceneCraft methodology [57]. It has some interesting features that are worth analyzing.

For reference, the figure 3.1 represents the schema adopted by the method.

After retrieving the assets needed from the initial text input, the first reasoning process is applied. Using an LLM, they first divide the request into multiple sub-scenes, each described by a title, the assets needed for each scene, and a description. This has the objective of dividing the problem into multiple simpler problems that are meant to be processed separately and then assembled at later stages. This is inspired by the prompt

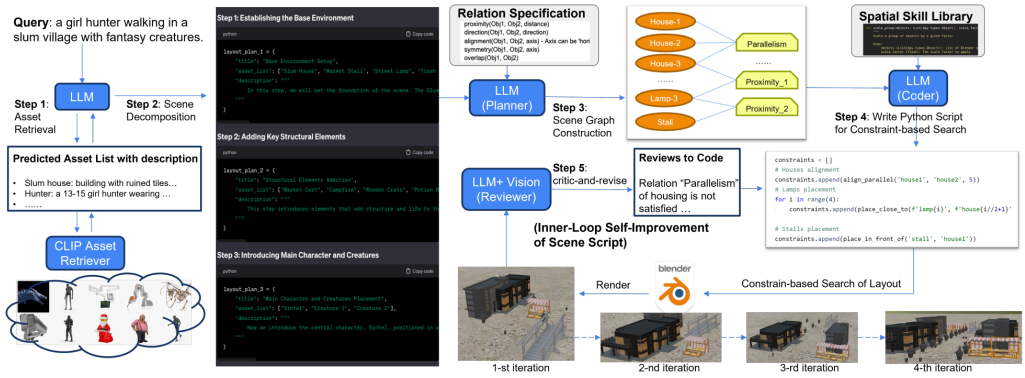


Figure 3.1: Scenecraft method pipeline [57]

engineering technique in [71] and is meant to achieve greater performance with the LLM, by handling simpler problems at the time.

After, also with an LLM, they construct specific spatial and contextual relations between the objects in the descriptions formulated previously. This involves relations like proximity, direction, alignment, symmetry, etc. This is meant to be a middle step, to then construct an iteration of a scene graph that organizes the assets and relations into a structured form. The next step also uses LLMs, this time to construct a Blender script that decides specific placements and orientations to generate an actual scene. It is important to note that at this point, the information that is given to the LLM is composed of the selected assets, the scene graph made, and a skill library that contains useful functions like scoring functions. Having this prior information and set of functions, the burden in the LLMs is decreased, because it does not need to do the whole planning, and can use the scoring functions to optimize specific numeric values without explicitly needing to decide them. But even with this set of tools, it is still difficult to get the objective scene, so they make use of an LLM with vision capabilities. It takes an image of the rendered scene constructed before, and with the prior information, tries to find inconsistencies, and then applies corrections, including the update of the coding library if needed, and this is applied n number of times.

For summarization, these are the key elements of this method:

1. Use LLM to divide the request into sub-scenes.
2. Construct spatial and contextual relations (e.g., proximity, alignment).
3. Generate a bipartite scene graph organizing assets and relations.
4. Use LLM to create a Blender script for placements and orientations using prior information and skill library

5. Improvement loop leveraging LLM with vision capabilities to analyze rendered images, detect mistakes, update skill library, and generate new, improved scenes.

Next, we will briefly mention this method [60]. Although it is limited to indoor scenes, it has some insightful particularities that are worth mentioning.

This method does not select the needed assets at the start. Instead, they are selected in the end, after a layout is made with bounding boxes. The initial steps involving LLMs can be synthesized in figure 3.2. First, the LLM processes the user input text that describes the desired scene. It imputes some missing information that is required for the method (like room size or object density, but this is not relevant in our use case) and then it also makes a list of all the objects in the described textual scene, their characteristics, and how their are in place in relation to other elements. To improve the result for this step, they put examples in the LLM prompt, which is a common prompt engineering technique, the few-shot prompting [27]. In the next step, the previous output is used to generate code with 3 objectives: choosing the most appropriate constructor for each object (basically, there is a pre-made set of functions that builds and labels bounding boxes, to after instantiate them in a scene), to put each object into a category, and also to accomplish the desired object density). Next, there is the final step that uses LLMs, and resumes using a base set of library of functions and the previous outputs to construct function calls that translate the relations present.

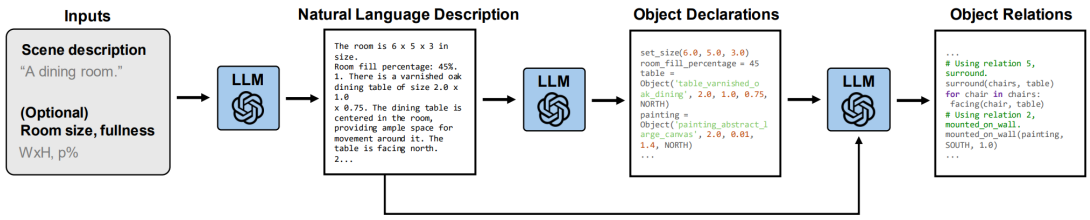


Figure 3.2: LLM reasoning structure in [60]

The next step is meant to define the object's positions and orientations on the final layout. Basically, with the code written before, the functions used can translate the scene construction problem into a constraint satisfaction problem. With this constraint satisfaction problem defined, there is a backbone constructed and designed to find a vector of object positions that satisfies all the constraints. To do this, they defined loss functions for each of the constraints and used gradient descent to find the optimal vector that minimizes the loss.

Here is a quick summary of the key points of this method:

1. Do not pick assets initially, instead, select them after layout generation.
2. LLM extracts object list, characteristics, and spatial relations.
3. LLM generates code to choose appropriate constructors for objects and categorize objects.
4. LLM constructs function calls, translating object relations into a constraint satisfaction problem.
5. Defines loss functions for constraints.
6. Uses gradient descent to optimize object positions and orientations.

These methods use LLMs as the main tool for qualitative reasoning, but for deciding specific quantitative aspects of the scene, the heavy lifting is done by other tools. The next method uses an LLM for deciding the quantitative aspects (coordinates and orientations) and allows the edition of the scene. Although with some trade-offs, as it only allows to generate indoor scenes that seem more simple and limited than the previous method.

This method is called LLplace [65]. For illustration, in figure 3.3 is the representation of the pipeline by the authors.

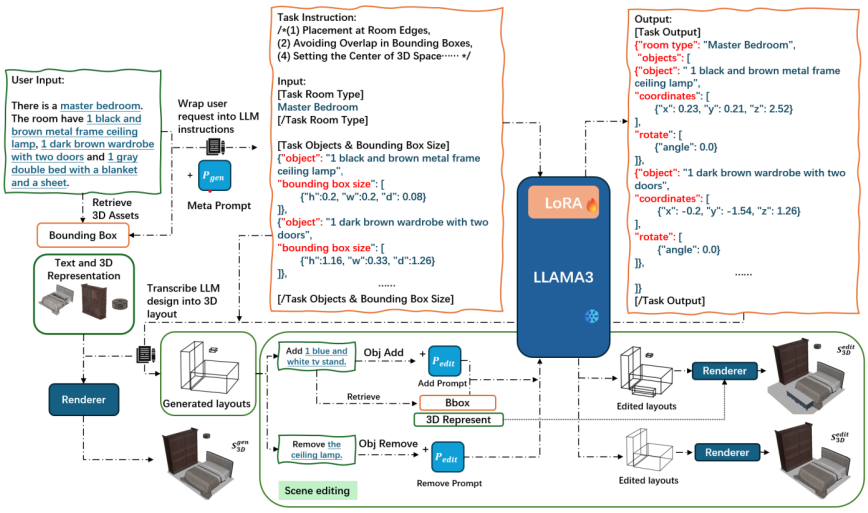


Figure 3.3: Pipeline of LLplace method [65]

The overall interpretation and translation of the user input into a qualitative scene representation is quite simple, given that the textual input for the method follows a defined

structure. So with the initial input text, the needed assets are retrieved, and in this process, the specific bounding boxes of the assets are also available. So with this information and the initial text input from the user, a prompt to the LLM is designed that structures the information in a Javascript Object Notation (JSON) format. This format is used when representing scenes in textual form because allows easy readability while following a well-defined structure, and because is a common practice, this also means that, in the training phase, the LLMs most likely were shown this kind of data in this structure, so it can have a positive impact in performance (this format is used in almost every method described here). Besides the JSON format, this prompt also has some guidelines to follow, like reducing overlap, encouraging alignment, etc. What is interesting is that with this prompt, they make the LLMs decide the coordinates and orientations of the objects, not only for scene generation but also for scene editing. To be able to achieve some degree of consistency in the LLM quantitative generation, what they did was fine-tune the LLM in a dataset that is composed of textual instructions with a variety of different commands and respective results related to indoor scene. To fine-tune an LLM they used a Low-Rank Adaptation (LoRA) [72]

To summarize, these are the main points of this method:

- The user provides an input text following a well-defined format, specifying the room type and desired objects.
- The retrieved assets and user input are structured into a prompt for the LLM, using a JSON format for readability and consistency.
- The LLM determines the positions and orientations of objects, for scene generation and editing.
- The LLM is fine-tuned on a dataset containing diverse textual instructions and corresponding indoor scene layouts.

Other methods fine-tune an LLM. For example, this paper [65] is dedicated to scenes in an industrial setting. The problem with these specific methods is that they are limited to a specific context and formatting.

Another strategy is to make the LLM construct the tools needed. This is partially done in the second-referred method, but SceneMotifCoder [73] does this differently and more explicitly. The method can be used in virtually any context, and the library constructed is

almost made from the ground up by the LLM, instead of already having the tools constructed and being dependent on a specific method to make the scene (in case of the second method, the optimization of loss functions). The downside of this specific iteration is that it takes as input example arrangements paired with text descriptions, to then learn and make similar arrangements. After the learning is done, the input is still a text describing the desired scene.

For reference, there is a scheme in figure 3.4 made by the authors representing the pipeline.

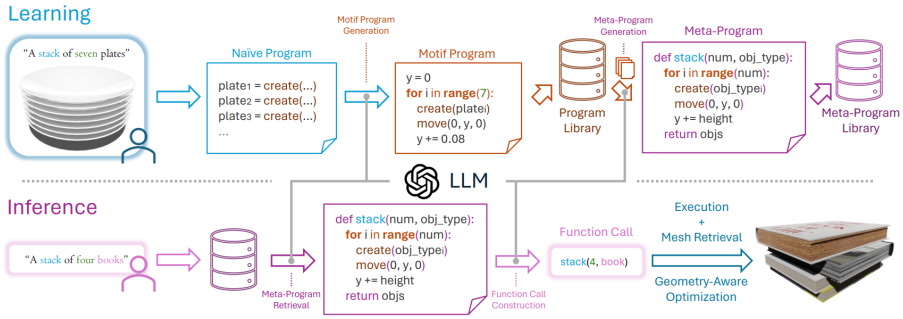


Figure 3.4: Pipeline of SceneMotifCoder method [73]

This method can be described as follows: starting with the 3D layouts paired with descriptions as input (it is important to note that these 3D layouts are similar), the first thing that is done is the creation of a naive program for each input example. This naive program is constructed directly based on the input, and it can reconstruct said input in an exact way, but without implying any kind of generalization or pattern. Then the LLM is used to generate new iterations of the naive programs, but trying to grasp any patterns present, and translate them into the programs. Now these versions of the programs made by the LLMs, which already have some degree of generalization, are given as input to an LLM, now to create a unique program that generalizes the input functions for a broad use case and defines parametrization for possible variations. To better guarantee that the generated functions are functional, some automatic testing mechanisms are used, paired with iterative prompting of the LLM to improve the program until it passes the tests. While doing this, a very well-known prompting technique is used, the chain-of-thought [74]. This was what the authors call the learning phase, which adds to the library general-use functions that can generate similar layouts as the input with customization. The other step, the inference phase, is when a new scene is made. With a text description of the desired scene, the necessary assets are retrieved using analogous methods of what was already

described in other instances, and utilizing the LLMs and the previously generated program, function calls are generated with parameters chosen by the LLM, to make a first iteration of the scene. After this, some mechanisms to optimize the generated layout are used, namely an interesting method based on rays. For this method, rays are emitted from a certain object to the surroundings, to then check for intersections, and small changes are made to the position of the objects according to the information of the rays casting results. This process is iteratively applied until the scene conforms with the objective and is physically plausible.

- The method requires example 3D layouts paired with text descriptions as input to learn and generate similar arrangements.
- For each input example, a naive program is created to reconstruct the layout exactly.
- The LLM analyzes naive programs and attempts to extract patterns.
- The LLM uses the previously generated versions to create a final program that generalizes input functions that allow for parametrization for broader applicability.
- The LLM generates function calls from previously made programs with chosen parameters to create an initial scene iteration.
- Optimization method based on Rays is employed to iteratively make small changes until the scene conforms to the objective and remains physically plausible.

The methods mentioned until now all introduce different ways to tackle the problem. But besides SceneCraft [57] and SceneMotifCoder [73], all the methods are domain-specific. Given that we are interested in a more general approach, let us analyze some more general-purpose methods.

WorldCraft [75] has a unique approach, pipeline-wise. The input of this method is an unstructured text describing the scene in general. But instead of being given to a pipeline with pre-established and ordered steps to construct the asked scene, it is given to an environment where 3 distinct LLM agents with different roles can collaborate by communicating with each other and assigning tasks to achieve the goal, enabling automatic scene creation. It is also noteworthy that, optionally, the user can interact with the LLM agents in a chat-like environment where the LLMs communicate with each other to make further requests on how the scene should be constructed. Figure 3.5 shows an example provided by the authors.

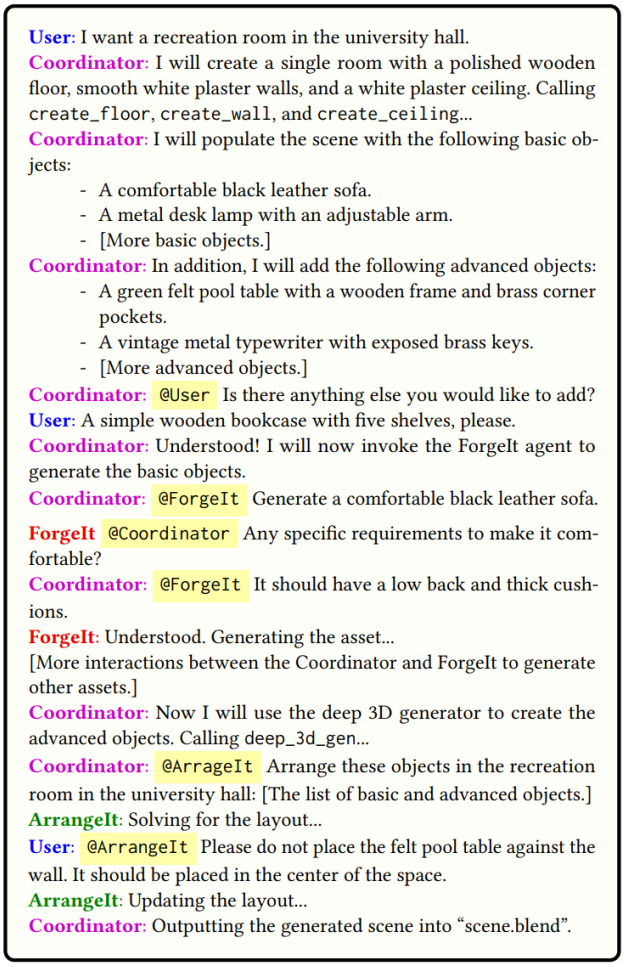


Figure 3.5: Example of interaction between agents and the user in WorldCraft method [75]

To better understand how this method works, we need to see the role of each LLM agent.

The coordinator agent, as the name implies, coordinates the construction of the scene. This means tackling the specific tasks, decomposing them into smaller steps if needed, and solving them by calling and collaborating with other agents. It is also responsible for processing the user request in the middle of the scene construction.

The ForgeIt LLM agent is the one responsible for the creation of 3D assets to be used on the scene. This agent can be given a certain description of an object to be synthesized, and then it uses a procedural generation framework called Infinigen [76] [77] to create the objects. It is remarkable how this framework is capable of constructing a large variety of assets just using procedural generation, ranging from natural elements like plants, animals, etc, to indoor elements like furniture, architecture elements, etc, and how it can also generate procedural terrains and backgrounds. This framework is based on Python in conjunction with the Blender API, enabling a straightforward utilization by the LLM

by simply having access to the framework, and then generating and running scripts that take advantage of it. This agent also includes some particularities to ensure better quality generation, like internal critic models to evaluate the generations, and the continuous and autonomous construction of a manual to utilize the Infinigen framework using LLMs, enabling a more robust use of it.

The last agent, the ArrangeIt, is responsible for the arrangement of the objects in the space. This agent decomposes the scene layout into a set of hierarchical numerical optimization problems and then leverages an optimization framework made by the authors to solve these problems and achieve the desired placements. This hierarchical numerical optimization works as follows: first, a tree-like structure is constructed with a hierarchical decomposition of the layout into subproblems. Each subproblem is translated to be solved by the optimization framework that is based on [78]. This optimization framework basically functions by extracting hierarchical and spatial relationships from pre-made scenes and translating this into a cost function, which is optimized by simulated annealing using a Metropolis-Hastings state search step.

- Takes a scene description as input.
- Three specialized LLM agents collaborate through communication and task delegation to construct the 3D scene.
- A chat-based interface allows optional user interaction, enabling both guided and fully automatic scene creation.
- The Coordinator Agent manages task decomposition, agent collaboration, and user input integration during the process.
- The ForgeIt Agent generates 3D assets using the Infinigen procedural framework, enhanced by internal quality controls and self-generated documentation.
- The ArrangeIt Agent organizes scene layout using hierarchical optimization, based on spatial priors and solved simulated annealing using a Metropolis-Hastings state search step.

To finish the overview of some of the existent methods, Scenethesis [79] receives as input a textual description of the scene in two different formats: a free from text where the scene is just described in generality, or the option to give a more detailed and description of the

scene, object by object. From there, the input enters a pipeline. For reference, there is a schema of the pipeline made by the authors in figure 3.6

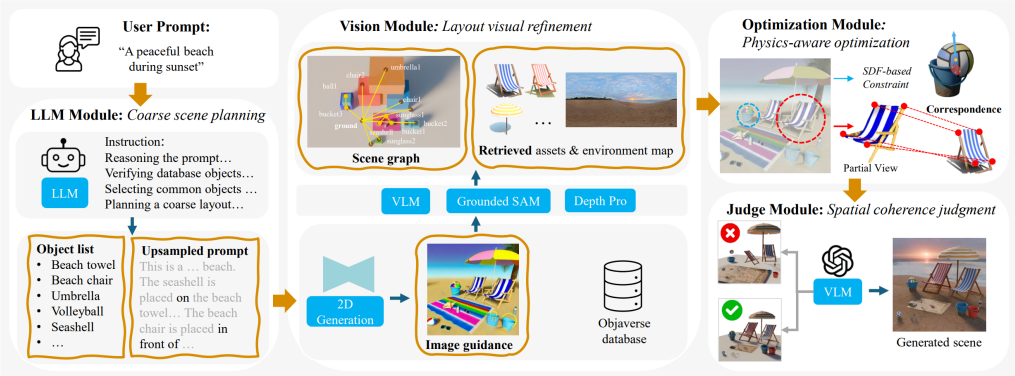


Figure 3.6: Pipeline schema from Scenethesis method [79]

The first step of the pipeline is only applied if the prompt is in a general description format. This step consists of analyzing the database of objects for the available category that makes sense to use in the given context. After selecting the best candidate categories, it generates a more detailed description where the specific objects and relationships are defined. During this process an anchor object is defined, that, as the name suggests, is used as the reference object to define the subsequent spatial relationships in the description.

After having this more detailed description of the scene (note that is still a free form text but instead of being something like ”A scene of a bedroom”, is more like ”There is a bed as the central object of the scene, on the right of the bed is a desk...”), the description is used in what the authors call the layout visual refinement. The first step consists of using the description to create an image of the scene using a text-to-image model. With the image and using GPT-4o (an LLM with vision capabilities), Grounded-SAM (an object detection and segmentation model [80]), and DepthPro (a depth estimator from image model [81]), a scene graph is created. This scene graph is a representation of the image in 3D space, where each object in the image is connected to a bounding box in the 3D space, and the relations between objects are explicitly stated. From this scene graph, a point cloud projection is made to have complete positions and orientation defined. After this, there is the asset retrieval step where the object's assets are retrieved from a subset of Objaverse [82]. There is also the retrieval of a background scenario from a custom-made scenario dataset created by the authors. And after, the assets are placed on the scene.

After having a first iteration for the scene, a variety of optimization steps are used to better align the objects' positions and orientations with the reference image previously created

and to ensure a physically plausible configuration. A pose alignment step that leverages RoMa [83], a dense feature matching model, to adjust the positions, orientations, and scale of the objects to better align with the reference image. And the definition of a set of constraints to ensure the stability of the scene and no overlaps.

After this optimization steps an LLM with vision capabilities (the GPT-4o model) is used to judge the resulting scene based on tree different criteria: object category accuracy (by comparing the generated scene with the reference image), object orientation alignment, by comparing the orientation of the objects in the scene with the reference image and overall spatial coherence (all metrics scaled fro m 0 to 1). If a predefined threshold is reached, it is determined that the scene is complete. If not, the scene is replanned.

- Scenethesis accepts either a general free-form scene description or a detailed object-by-object input
- A text-to-image model generates a reference image from the refined description, which is then used to create a 3D scene graph via GPT-4o, Grounded-SAM [80], and DepthPro [81], enabling point cloud projection with spatial and depth information.
- Scene assets are retrieved from Objaverse and a custom background dataset, then positioned based on the scene graph structure.
- An optimization phase adjusts object pose and alignment using RoMa [83] and enforces physical constraints to ensure scene stability and realism.
- GPT-4o evaluates the generated scene based on object category accuracy, orientation alignment, and spatial coherence (each scored from 0 to 1). If the scores exceed a threshold, the scene is finalized. Otherwise, it is refined in additional iterations.

3.2. Scene textual structure

With the previous information, we already saw how a scene graph is the go-to choice when representing a scene layout in textual format. We already explained what they are, but it is important to introduce some specific examples for reference. So next is a scene graph representation of a simple scene composed of a table with a vase on top, in a JSON format. This representation is inspired by the kind of scene graphs used in the previously mentioned methods:

Nodes

```
{
  "objects": [
    {
      "id": 1,
      "name": "Table",
      "bounding_box": {
        "x": 50,
        "y": 100,
        "width": 200,
        "height": 150
      },
      "coordinates": {
        "x": 150,
        "y": 175
      },
      "orientation": [0, 0, 0],
      "description": "A rectangular wooden table with four legs.",
      "tags": ["furniture", "wood", "rectangular"]
    },
    {
      "id": 2,
      "name": "Vase",
      "bounding_box": {
        "x": 120,
        "y": 50,
        "width": 60,
        "height": 80
      },
      "coordinates": {
        "x": 150,
        "y": 90
      },
      "orientation": [0, 45, 0],
      "description": "A tall ceramic vase with a blue floral pattern.",
      "tags": ["decor", "ceramic", "blue", "floral"]
    }
  ]
```

```
}
```

Edges

```
{
  "edges": [
    {
      "source": 2,
      "target": 1,
      "relation": "on top of"
    }
  ]
}
```

For the nodes, we have the objects, paired with the necessary information. Typically, each entry has a unique ID, so there is no confusion when iterations of the same object are used. Then the actual name of the object name typically paired with some more object descriptors. In this example, we have a description of the object and some related tags that the object is associated with. For quantitative information, usually the bounding box dimensions, coordinates, and orientation are provided. For the relations, the unique IDs of the objects involved are used and are defined by a simple expression or identifier. In the specific relation shown, there is a sense of direction, where the vase (ID 2) is on top of the table (ID 1).

3.3. Evaluation

Evaluating the outputs of a 3D scene generator is not a trivial task, even more so if the input is just an unstructured text input describing the wanted result. It is not trivial because a textual unstructured text input can be interpreted in many ways, and even if we do not consider these many meanings, there is also the fact that a scene that semantically follows a text description can take countless different forms. For example, let us consider the following text input ”*A table on the side of a chair*”. The number of possible configurations that are semantically compatible with this scene is countless. We can put the chair on the right of the table, left, at 1 meter, 10 cm, etc, making it very difficult to make an evaluation that can take into account these nuances correctly. The other point is that in general 3D scenes are pieces of data that are complex by nature, that involve many objects in

a shared environment, and form relations to each other. That being said, the evaluation of 3D scenes is still an ongoing topic that has not reached a very consistent method of evaluation. But there are a lot of useful approaches. Were will mention the most relevant ones that we found.

CLIP model: The most used method to evaluate this kind of method is the use of CLIP models to calculate the CLIP score mentioned in the equation (3.1). This was already introduced, but in short, it can evaluate with alignment with human perception how well a certain image is related to a textual description by giving it a numerical score. This can then be useful, especially in the use case of generating scenes with different methods, while using the same text input, to then compare the alignment with the input of different methods. Usually, the baseline model used for this kind of evaluation is the CLIP model from OpenAI [58], but now multiple new iterations of this model can potentially provide a better evaluation. They can go from fine-tuned versions to specific domains/datasets like in this work [84], versions with different architecture and other key differences, like Siglip 2 [9] that as shown in the paper, outperforms the CLIP model from OpenAI in a variety of task, methods that use a CLIP based model as backbone to do a different task, as is the case of ImageReward [7], that uses Blip [85] as backbone and was trained in textual descriptions with images paired with human ratings, having the capacity to output a human aligned rating directly, and the already mentioned versions that can receive 3D data directly[57].

Multi-modal LLMs: For model-based methods for evaluating, multimodal LLMs seems to be another good option, given that LLMs seems to also align with human opinion when it comes to analyzing 3D content [86]. This evaluation can take numerous amount of forms. For example, LLplace [65] uses GPT-4O to look at the images of the generated scenes and ask it to evaluate them based on Functionality, Activity-based Alignment, and Aesthetics. SceneMotifCoder[73] uses a GPTEval3D [86] to measure text-asset alignment and 3D plausibility. Or VQAScore [8] that uses a VLLM to calculate the likelihood of the answer being ”Yes” from simple yes or no questions about a given image, being these yes or no questions constructed based on the textual description of the image.

It is worth noting that this evaluation using LLMs can be extended to also evaluate not only the 3D scene generated, but also to evaluate intermediate steps, like the contributed scene graph, etc. The problem with evaluating using LLMs is that is really difficult to

understand the full scope of how a certain evaluation is obtained, even with very well-defined instructions, and by the fact that LLMs are not that consistent with their responses.

Human evaluation: Human evaluation is also a commonly used method to address the quality of these methods. It needs considerable human resources and a carefully selected group of people to be effective (to avoid potential biases), but we get an evaluation that, of course, goes with human values. It can take a variety of forms, like asking to compare different layouts of different methods, and decide the better one [87], or judging a layout based on some criteria [61].

Mathematical metrics: There are metrics based on mathematical formulations that can be used to evaluate the scenes generated by these methods. In general, there is no metric that successfully evaluates the entire scene, but each metric can be used to evaluate specific aspects of the scene. I will mention some examples:

Intersection Over Union (IoU) in our context can be used to measure the overlap between bounding boxes.

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|}$$

Where:

- A and B are bounding boxes of different objects.

Fréchet Inception Distance (FID) compares statistics (mean and covariance) of feature representations between reference data and generated data.

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr}\left(\Sigma_r + \Sigma_g - 2\left(\Sigma_r\Sigma_g\right)^{1/2}\right)$$

Where:

- μ_r and Σ_r: Mean and covariance of features of reference data
- μ_g and Σ_g: Mean and covariance for generated data

The Object Overlap Rate (OOR) quantifies the degree of spatial overlap between distinct objects in a generated 3D scene. In the context of evaluating scene rationality, lower overlap is often desired because excessive overlaps can indicate unrealistic object placement.

$$\text{OOR} = \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{\text{Volume}(B_i \cap B_j)}{\text{Volume}(B_i \cup B_j)}$$

Where:

- N: Total number of objects in the scene.
- B_i and B_j: Bounding volumes of objects i and j.
- Volume$(B_i \cap B_j)$: The volume of the intersection between the bounding volumes of objects i and j.
- Volume$(B_i \cup B_j)$: The volume of the union of the bounding boxes.

Collision-Free Score (CF) measures the proportion of objects in a generated 3D scene that do not collide with any other object. A higher CF indicates fewer inter-object collisions and a more realistic scene layout.

$$\text{CF} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1} \left(\bigwedge_{\substack{j=1 \\ j \neq i}}^{N} \{B_i \cap B_j = \emptyset\} \right)$$

Where:

- N is the total number of objects in the scene.
- B_i is the bounding box of object i.
- $B_i \cap B_j = \emptyset$ indicates that object i does not overlap (collide) with object j.
- $\mathbf{1}$ is the indicator function that returns 1 if the condition is true and 0 otherwise.

Constrain evaluation: There are methods to evaluate the relations/constraints that should be present on the scene (like if an object is on the right of another, etc) with direct formulation. For this, for every type of constraint/ relation, there could be a specific method to make the evaluation. This is the overall open idea, making it possible to be defined in many ways. For a specific example, in this method [87], the constraints are defined by the table 3.7. Then, for example, we can just measure the proportion of constraints that are well translated into the scene in relation to all constraints.

Relationship	Rule
left of	$x_i < x_j$ and $IoU(b_i, b_j) < 0.3$
right of	$x_i > x_j$ and $IoU(b_i, b_j) < 0.3$
front of	$y_i < y_j$ and $IoU(b_i, b_j) < 0.3$
behind of	$y_i > y_j$ and $IoU(b_i, b_j) < 0.3$
bigger than	$(w_i l_i h_i - w_j l_j h_j)/w_i l_i h_i > 0.15$
smaller than	$(w_i l_i h_i - w_j l_j h_j)/w_i l_i h_i < -0.15$
taller than	$((z_i + h_i) - (z_j + h_j))/(z_i + h_i) > 0.1$
shorter than	$((z_i + h_i) - (z_j + h_j))/(z_i + h_i) < -0.1$
close by	$Dis(Cor(b_i), Cor(b_j)) < 0.45$
symmetrical to	$Dis(Fli(b_i), b_j) < 0.45$

Figure 3.7: Constrain definition in[87]

Using method output as training data or input for other models: There is also the possibility of using the output scenes for training other methods, or as input, and then evaluating this other method's results. For example, in SceneCraft [57], they try to recreate a scene of a movie using two distinct scene generation methods. Then, using a video generation model, they use the output scenes from the scene generation methods as input to generate two distinct short films. After this, they compare these two generated short films. The best version of the short film should be generated using the scene made with the best method.

3.4. Datasets

Datasets, in our context, can be relevant to train/fine-tune a model, namely the LLMs, but also to test the results of an approach, using the dataset as ground truth.

Our objective is mainly to construct scenes in a road context. Ideally, a dataset that contains a variety of scenes in a road context would be a main candidate. But this is not the whole picture. Given that our goal is a method that can build scenes by rearranging pre-made assets, we also need the assets to work with. So, unless we can find similar assets in a given scene, the dataset has to be composed of separable 3D assets. The big problem about this is that the majority of datasets in road context are obtained by Light Detection and Ranging (LiDAR), or other sensors, so they are typically composed of reconstructed points of view and do not contain the full objects, nor are they separable in an easy way. To add even more constraints, the dataset should be labeled in a way that at least each scene has a natural language description (other textual descriptions would also be useful, like a scene graph). A dataset like this, as far as we know, is not available. Now, there are suboptimal options out there. First, some datasets respect all

the necessities apart from the context. For example, the 3D-GRAND dataset, introduced in [88] is composed of various 3D indoor scenes with individual 3D assets paired with a variety of related textual information. For training purposes, this dataset is not very appropriate given our context, but it can be useful for testing a method if we end up making a general approach adaptable to any domain. Another option is to use a pre-built map of road context scenes, like the available maps in CARLA that are made with individual assets, and of course appropriate to our context, to build a dataset of different scenes in that map, labeled with textual descriptions, possibly generated with the help of a multi-modal LLMs. Although not optimal, with assess to a repository of assets (or paired with a system that can generate assets), then we can use datasets that contain or are possible to extract good-quality images of related road scenes. In this context, we do not have a ground truth scene, but at least have an image of a scene that can already be useful. As an example of this kind of dataset, we have KITTI [89]. We could extract images of road context, label them, and then use external assets to build the scenes.

But instead of trying to focus on domain-specific datasets, it is possible to get datasets whose whole point is to improve the reasoning capabilities of LLMs in our context. Maybe these are not so useful for the evaluation part, but can be significant. For example, Visual Genome [90] is a dataset of images, where there are annotated descriptors and relations between elements of the image, that could help a vision LLM to better understand spatial compositions and relations. Or CLEVR [91], which is composed of questions and answers about images of simple block compositions.

3.5. Prompt engineering

We already introduced what prompt engineering is in the context of LLMs and how these methods can be useful. Now we will look at specific prompt engineering techniques, starting with more basic approaches to more complex focusing on general prompting methods that can improve the planning and reasoning of the model on a given task.

Zero-shot prompting and few-shot prompting: Zero-shot prompting is the most basic form to prompt an LLM to perform a certain task. This resumes to explaining the task with a unique text prompt and expect the LLM, with that text prompt describing the task as input, to be able to perform the task correctly. Few-shot prompting [27] includes not only the explanation of the task but also adds examples of that task being completed. So, in a single text prompt, we put the task description and examples of different iterations of the same task. This is remarkably successful for tasks that do not need a lot of reasoning

abilities [74], like simple question-answering requests, giving the LLMs extra guidance to understand and tackle the problem. This is important because this example-driven approach is not only useful on its own, but can be easily used in combination with other techniques.

"Let's think step-by-step": This technique revolves around adding to the text prompt a simple instruction "Let's think step-by-step" [92]. This method is especially effective when we have a task that needs a sequence of different steps to be accomplished, where some level of reasoning is needed. This is exemplified in 3.8 by the authors.

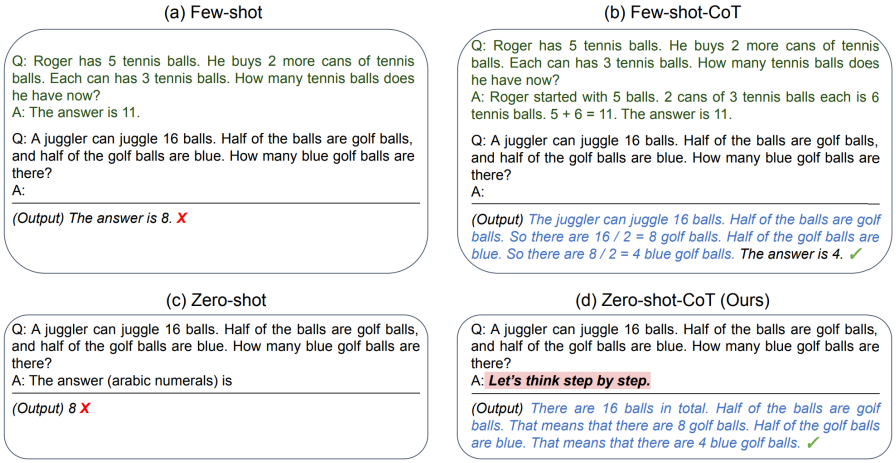


Figure 3.8: Example of usage of the "Think step-by-step" method, also called Zero-shot- Chain of Thought (CoT). Few-shot- CoT refers to few-shot chain-of-thought, where chain-of-thought is another prompting technique in [92]

EmotionPrompt: The method of EmotionPrompt is described as giving emotional stimuli to an LLM. For example, it could be something like describing the task, saying "Complete this task like your life depended on it", where a sense of urgency is introduced, and there is a lot at stake. This can be effective in a variety of different tasks, like introduction following [93].

Role Prompting: Giving a role to an LLM is also a valid prompt technique that can result in performance gains in a variety of different tasks, given its flexibility. This method resumes to include a statement that says a role that the LLM should take while executing the task. For example, we wanted to plan an indoor scene layout, so one viable option could be to add the phrase "You are a professional interior designer." [94].

Self-Ask prompting: Making the LLMs question itself about a given problem and formulating the answers to that sub-question before solving the task at hand is also a simple and shown effective prompting technique.[95].

Chain-of-Though (CoT) As the name implies, the CoT method consists of making the LLMs generate a reasoning process. [74] The idea is to try to mimic how people solve a complex problem. Usually, the problem is broken down into smaller parts, and the reasoning is made in a sequential way until a solution is found. This technique is somewhat similar to the ”Let’s think step-by-step” method introduced earlier. This is because that method can be considered the zero-shot version of this technique. In the original paper is referenced that this method can be done with a few-shot approach, this means that by providing examples where a chain of thought is used to solve a problem, capable enough LLMs can construct a similar ”though” process when solving the problems, an example can be seen in 3.9 . But this is not the only way to execute this method. This can also be made by manually constructing a pipeline that generates this thought process, or going further, it can arise from how a LLM is trained, like with Deepseek-R1 model [96], although this goes beyond prompt engineering.

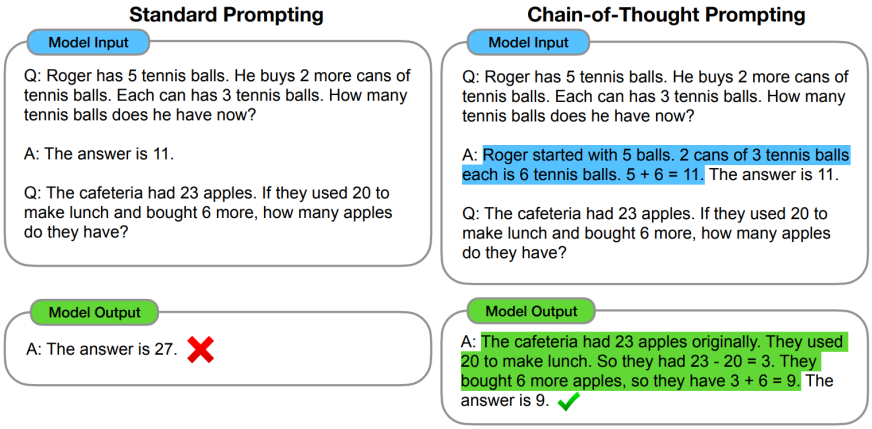


Figure 3.9: CoT example versus general prompting in [74]

Least-to-Most prompting (LtM): This prompting technique involves solving a complex problem by breaking it down into a sequence of simpler subproblems [71]. The key idea is that by solving these smaller problems step by step, the model can incrementally build toward the final solution. This seems similar to Cot, but this process is made more explicit. This approach is structured into two main stages. First is the Decomposition Stage, where the model is prompted with examples demonstrating how to break down problems into manageable subproblems. After these examples, the actual problem to be decomposed is presented. Next is the Subproblem Solving Stage, where the model is prompted to solve each subproblem sequentially. The prompt for this stage consists of examples demonstrating how subproblems are solved, a growing list of previously answered subproblems and their solutions, and the next subproblem to be solved. As the model progresses

through the sequence, it builds on prior solutions until it arrives at the final answer. This method ensures that complex reasoning tasks are solved in a structured manner, helping models avoid confusion by focusing on one small task at a time. This can be visualized in 3.10

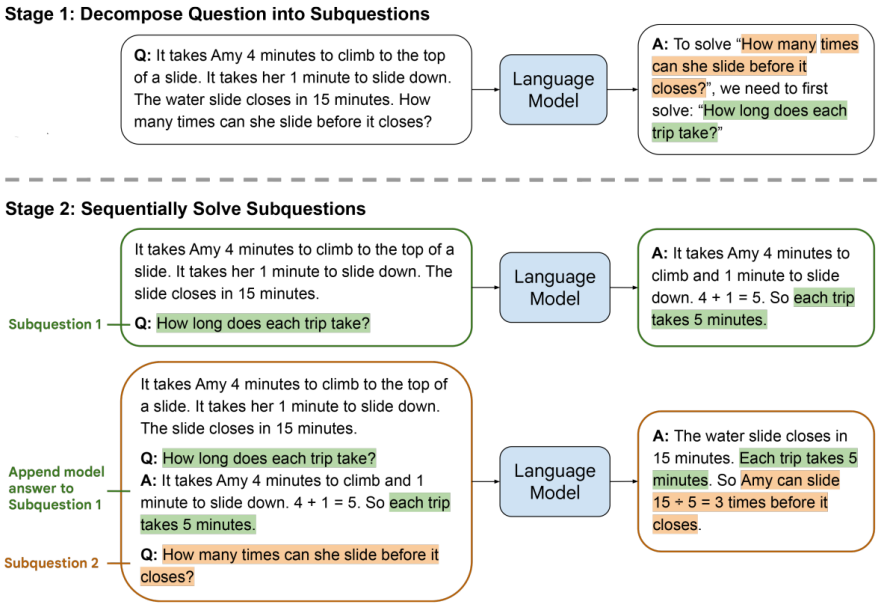


Figure 3.10: Least-to-most prompting example in [71]

Tree-of-Thoughts (ToT) : This method consists of exploring multiple reasoning paths simultaneously, structured in a tree-like format [97]. This approach allows the model to deliberate on various potential solutions before arriving at a final answer. In more detail, using this method, the LLMs is prompted to generate several possible intermediate steps at each decision point. These thoughts branch out, forming a tree structure where each node represents a partial solution or a step in the reasoning process. The model evaluates these branches, considering different paths and backtracking when necessary, to determine the most promising route to the solution. Additionally, depending on the task's complexity, the depth and breadth of the tree can be adjusted, allowing for a balance between computational efficiency and thoroughness in exploration.

There are many more possible methods that can be useful to our approach. These are just some representative examples[98] [99].

4. Methodology and Development

In this chapter, we describe the developed methodology to generate 3D scenes. We start by making some clarifications of the initial objective defined in 1.2, because when translating a broad objective into a specific implementation, some details are left undefined. Namely, we explain our decisions for the textual input of our method, the scenes that are meant to generate, and the specific software used to store and manipulate the 3D scenes, accompanied by the particular format for the 3D assets. Following these details, we explain the built pipeline for constructing 3D scenes leveraging LLMs, which was the main focus of our work. We also provide a rundown of how the prompts for the LLM and Vision Large Language Model (VLLM) were constructed.

We also introduce a simple pipeline for a developed method to edit pre-made scenes using textual input. This could be used as a complement to our scene construction pipeline, by enabling the edition of 3D scenes.

4.1. Objective

We already introduced the overall objective of this work in 1.2, which sums to building a pipeline that leverages LLMs to produce 3D scenes with simple text inputs. But this description ends up being open-ended in some regards when it is applied to a real implementation.

It is mentioned in section 1.2 that we want to make 3D scenes using text as input that describes the scene to build. This is a very broad definition of an input, so we will provide a more precise clarification and definition of the textual input used in our method.

We also mention that we want to develop a method that can construct 3D scenes, with a special interest in autonomous driving-related scenes. This leaves doubt if we construct a domain-specific method, which has tailor-made tools for the autonomous driving context. Or a more open-ended method, which, while not focusing on any domain, is general, supporting a variety of different domains (including the autonomous driving context).

Finally, to make the 3D scenes, we want to use the LLMs. But generically, they cannot do that directly. As explained in 2.1, dedicated software is necessary to actually have a 3D scene in place. And the elements of the 3D scene also have different formats as explained in 2.2.

4.1.1 Input description

For the input of our method, regarding the scene construction task, our objective is to use a text that describes a given scene. This definition of the input is not sufficient. Like documented in Scenethesis [79], it can be a more general description where not all (or even none) objects are mentioned, just referring to the general setting of the scene, like the description ”A scene of a busy street”. Or a more detailed description, that while still being a free-form text, depicts the individual objects present on the scene, and gives information about the relation between them. For example, ”A scene of a straight road, a couple of buildings on each side of the road, ...” (note how it is not a very rigorous qualitative description of the scene, but gives the information of all objects present and an overall idea of the intended configuration).

The more general description is more practical and easier to create for the end user, but from the method standpoint, it involves more internal processing, by defining/choosing the objects that would be present on the scene, and the need for the construction of an overall configuration. From an evaluation standpoint, this description is also harder to judge from a qualitative perspective. The more detailed description already gives the information of the objects present and the overall configuration, turning the processing of the initial input simpler, but comes with the downside of being more extensive and harder to formulate. It also leaves the need for some pre-processing of the description, because it can go beyond the length supported by the models used for evaluation (namely CLIP based models).

We decided to use the more detailed definition of the scene as the input, as it would be easier to assess the method while being accessible enough for the user. And in the potential main use case of this pipeline, that is the generation of specific scenes in the autonomous driving contexts, having a more fine grain control over the generations would probably be more useful.

4.1.2 General vs Domain-specific method

Another decision point is whether we focus on a specific domain, which in our case would be the autonomous driving-related scenes, or we go for a more generalist approach, that could adapt to a specific domain without having tailor-made tools for a given domain.

We choose to build a more generalist approach. Although we have a domain where we have special interest, the literature on domain-specific methods for generating 3D scenes

with LLMs at the time of writing is almost exclusive to indoor scenes. So with fairly limited literature on the topic, we decided to go with a more generalist approach.

4.2. 3D Software and Format

For the 3D Software, which provides an environment to create and manipulate a 3D scene, we used Blender [10]. We choose this specific tool because it is very powerful, and documented to be capable in a text-to- 3D scene generation pipeline, for example, in the SceneCraft [57] method. There is also the facts that is an open source software, while being the most light weight of the considered options and providing all the tools needed to construct a pipeline like ours, namely a Python-compatible API that enables the creation or editing of 3D scenes by generating scripts.

For the format of the 3D assets, this choice is straightforward, as any Blender-compatible format would realistically work, but for the sake of being convenient and lightweight, we used a mesh format. It is noteworthy that this choice is more a matter of what the format of the assets that we have in hand is than a very thoughtful choice. It would be simple to adapt to any Blender-compatible format and mix different ones if necessary.

4.3. Scene Construction Pipeline

The pipeline to construct 3D scenes with a text description is represented in figure 4.1.

In general, the pipeline functions as follows:

With a text description provided by the user, the first pipeline step is to process the text description into smaller sub-scenes if some criteria are met, leveraging LLMs. If there is a division into sub-scenes, then each sub-scene is treated independently and is merged in the end, after all sub-scenes are constructed.

After the scene division step, we enter the asset selection pipeline. In this pipeline, we analyze the description available and first define the individual objects necessary to fulfill a given description using LLMs. Then we search an available repository of 3D assets, and make a link of a given necessary object to an existing asset based on some criteria using a CLIP model and the LLM.

After selecting the necessary assets, there is the qualitative formulation step. In this step, using LLMs, a more structured and well-defined formulation of the scene in the form of a scene graph is created.

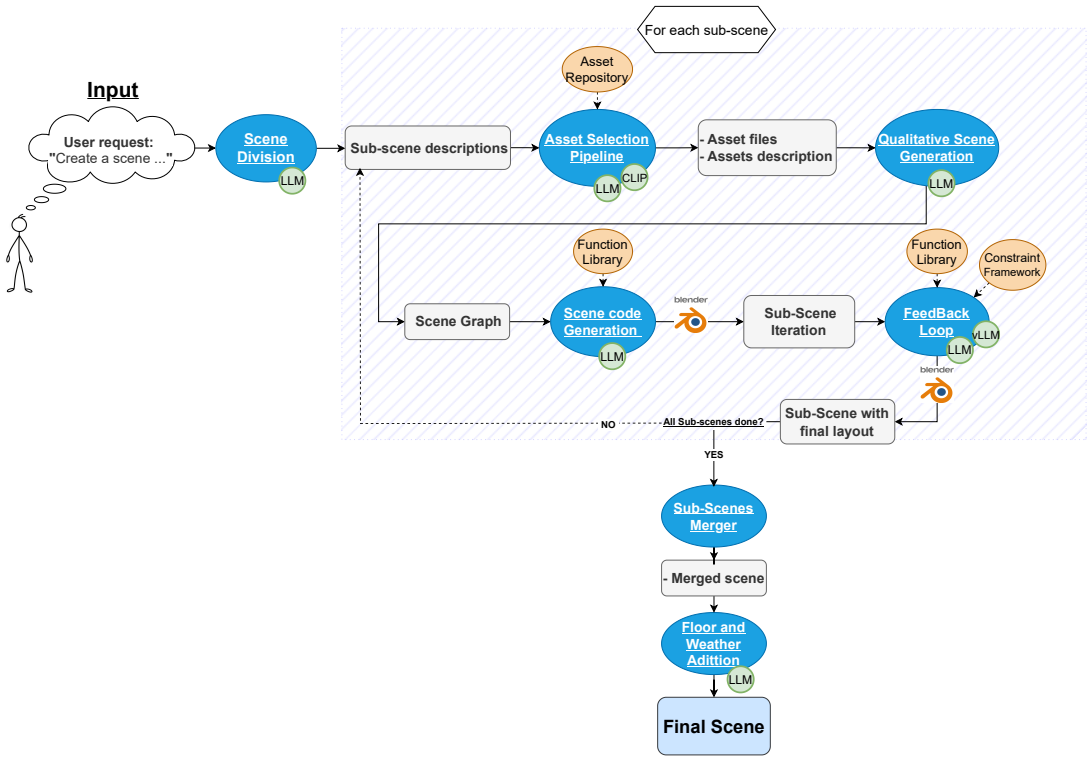


Figure 4.1: Scene construction pipeline

With the scene graph, the first iteration of the scene is built. With access to an external set of functions previously made that have basic functionality related to placing objects on the scene and adjusting their orientation in Blender, first is generated a Python script using a LLM, which then accomplishes the scene depicted in the scene graph in Blender.

After the first iteration of the scene is made, we enter the feedback loop step. In this step, LLMs, Vision Large Language Models (VLLMs), and a custom-made constraint evaluator framework are used to understand the current state of the scene and detect possible discrepancies with the objective scene, to then make the necessary adjustments to better align with the objective scene.

After a defined number of runs of the feedback loop, the different sub-scenes are merged if there is more than one. At last, a post processing step, where a simple ground floor and a sky are added to the scene using an LLM.

This is the overall description of the final pipeline. Some additional steps were developed but were not included in the final version for various reasons, which are documented in the Appendix A. The use of the Blender built-in physics simulator to potentially correct any physical problems with the scene (namely overlaps and floating objects) in the Appendix A.5. An optimization pipeline that transforms a given 3D scene into an optimization

problem, given the objective constraints in the Appendix A.6. And the automatic function library construction, which was meant to add new functionality not present in the function library in the Appendix A.7

With the overall pipeline described. Now we will address the specifics of each step, the tools used, how exactly these steps work, and the assumptions behind them.

4.3.1 Scene divider

The use of an approach that can break down the scene construction into smaller sub-scenes when a complexity threshold is reached is present in different methods in literature, for example, in WorldCraft [75] and SceneCraft [57](although there is always a lack of detail on how the actual implementation is made).

The reason behind the use of this kind of methods comes down to with the increase in complexity and scale of the task, the overall performance of LLMs decreases. This is a statement shown multiple times in literature:[100] [74] [101]. And because there is a limit to how many tokens an LLM can keep in ”memory” and output. This is very detrimental. If not all input can fit in the context length of the model, then the model can lose access to crucial information necessary to solve the problem. If the solution and the steps necessary to achieve it can not fit in the maximum output token length of the model, then it can give incomplete solutions or not achieve a solution at all.

We thought of two main ways to tackle this scene division problem: An incremental one, where the scene should be constructed incrementally. And another, that divides the scene from the start and then merges all the sub-scenes in the end. We decided to go with the second option, although both have their upsides and downsides.

Building a scene incrementally leads to the need to select and store the necessary information to perform each incremental step. There should be a balance between the information stored and the complexity of the incremental step: If the information stored is too much, we are back to the problem of having too much information to process. If the information stored is too little, it is not enough to make the increment step effective. If the incremental step is too big, we fall into the problem of a very complex problem again. If too small, then the execution time could dramatically increase, because we would need a large amount of increments to fulfill the scene.

Dividing the scene from the start has a different subset of challenges. A good criterion to fragment the scene into sub-scenes should be defined to enable an effective and efficient fragmentation. This means that the fragmentation should not create too complex sub-scenes, so we are back to dealing with a very complex problem at once, nor fragment the scene too much, so the execution time remains reasonable. This criterion also should enable a good quality merging of the sub-scenes, so that in the end we have a good quality scene.

We decided to divide the scene at the start of the pipeline for two reasons. The fact that choosing the necessary information for each incremental step is a non-trivial task. And because at the point of making this step of the pipeline, we already had prototypes of some subsequent steps of the pipeline, and their implementation was not fully compatible with an incremental approach.

Now, for the specific details, this step consists of a single LLM prompt that fragments the initial description into smaller descriptions with the following main criteria:

- The number of objects in the scene: A scene description should be fragmented if it has more than 12 objects referred to in the description. There is also a minimum criterion for the sub-scene object number (defined as 4) to not make the decision too fragmented (these numbers were defined from experimentation and with a specific LLM, Gemini 2.0 flash from Google, so they are probably not ideal and tied to our specific setup). To ensure a better quality in this step, we make the LLM identify the elements of each description.
- The presence of groups of objects that could be separated in space (clusters of objects) that allows independent placement and orientation. This is to ensure that the description of sub-scenes is as independent as possible, while minimizing information lost. For example, if we have a scene with a road with a couple of cars on top, and a far away farm with a house, some trees, and animals, the road area and the farm area can be considered independent clusters that could be constructed independently.

To ensure a somewhat meaningful merging of the sub-scenes at the end, alongside this description division, there were some extra features defined for each sub-scene to define the position of the sub-scene in the final scene when merging. First, each scene has a unique identifier (ID) for identification. Then, for each scene, two parameters were

defined. One that refers to another scene ID, and the other that refers to the relative position that the current sub-scene should be relative to the mentioned scene. It can be ”right”, ”left”, ”front”, ”back”, ”back-left”, ”back-right”, ”front-left”, ”front-right” (note that although refereed in this way, is explained to the LLM how these directions refer to fixed axis directions, like for example ”right” means in the -X direction of the referenced scene). There is only one exception to this definition of parameters. While making the sub-scenes, there is always one sub-scene that is defined as the center scene. This sub-scene serves as an anchor to start placing the other sub-scenes in space (being defined in the merge step as being located in the (0,0,0) coordinates).

We show below a practical example where there is a division of a description:

Scene description:

”A road is placed at the center of the scene. On the road, six bikes are positioned, each ridden by a woman. To the right of the road, there is a table with four bottles of water.”

Segmented description, generated with the prompt represented in the listing A.1 from the Appendix A.1

```
[
    {
        "subscene_name": "Main_Road_Section",
        "description": "A road is placed. Six bikes, each ridden by a woman, are positioned on this road.",
        "objects_list": [
            "road",
            "bike",
            "woman"
        ],
        "relative_positioning": {
            "relative_to": null,
            "direction": "center"
        }
    },
    {
        "subscene_name": "Table_Area",
        "description": "A table with four bottles of water is present. This area is situated towards the -X axis.",
```

```
        "objects_list": [
            "table",
            "bottle of water"
        ],
        "relative_positioning": {
            "relative_to": "Main_Road_Section",
            "direction": "right"
        }
    }
]
```

With this strategy, it is possible to merge the sub-scenes given that each sub-scene is defined with a relative position to another sub-scene, which at the end of the pipeline is converted into a translation in relation to the reference scene. This is not an ideal merging strategy by any means, and can lead to a ”blocky” scene and loss of coherence and information (an example is depicted in figure 5.18 of the results in chapter 5), but it is a simple and effective solution, assuming that the criteria mentioned are followed properly by the LLM.

4.3.2 Asset selection pipeline

Another important feature of our scene construction pipeline consists of identifying the objects mentioned in the textual description and then selecting and retrieving the most appropriate assets that align with the identified objects from an asset repository. Our approach is inspired on CLIP based methods, like the methods used in [57] [60] [61] [62]

The asset repository is composed of individual folders, each folder containing a unique 3D asset file and 3 other files that were generated in an offline processing step. An image of that asset was created by rendering the asset standalone in Blender. Then, with this image, an embedding vector was created using the CLIP model CLIP-vit-largepatch14 [58]. The choice behind this specific model is that it is a common baseline model for similar use cases, although another CLIP like model could be used and potentially achieve better results. With the same image, using a VLLM was also created a simple text description of the object.

The asset selection pipeline is represented in figure 4.2.

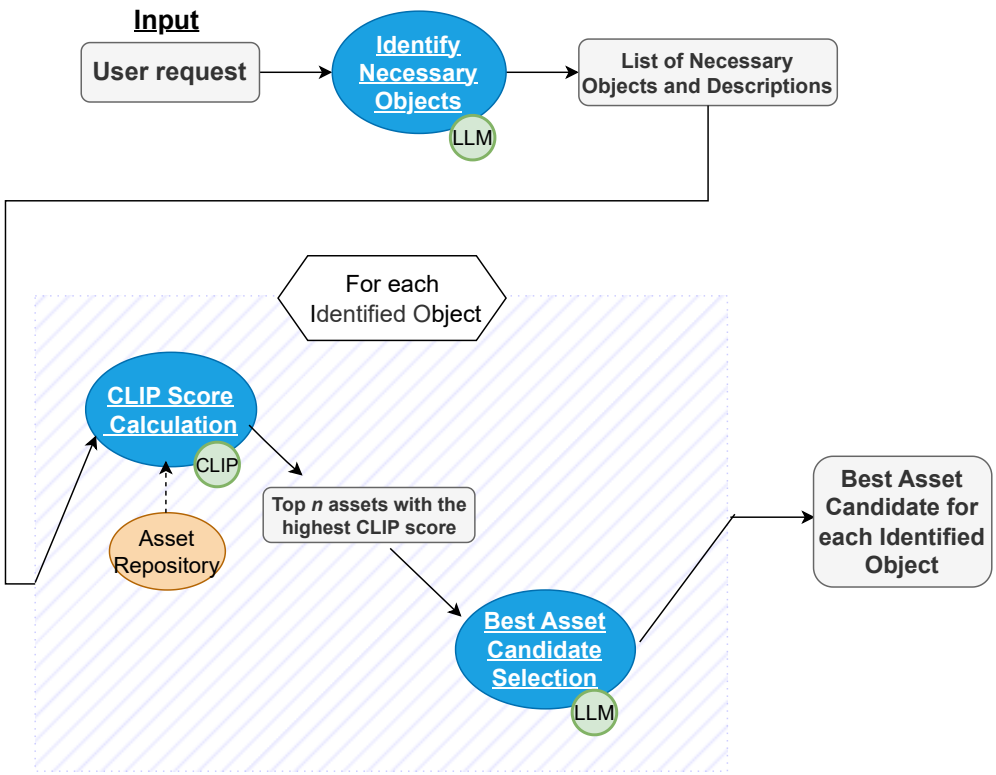


Figure 4.2: Asset selection pipeline

The first step of the asset selection uses an LLM to identify the necessary objects to fulfill a scene description. This extraction involves not only the individual objects (like ”car”, ”house”, etc.) but also possible descriptors of the object, like color and shape. With this information extracted with the LLM, individual descriptions are formulated for each of the objects identified.

For example, for the following input description:

”A scene with a tree and a nearby park bench oriented toward the tree. A man is standing on the bench, also facing the tree.”

The LLM, with the prompt depicted in 4.1, generated the following object identification:

```
"""
Let's break down this scene request to identify the necessary assets.

1.  **Tree:** A tree is explicitly mentioned. We need a 3D model of a tree. The
    request doesn't specify the type of tree, so any realistic-looking tree model
     will suffice.
```

```
2.  **Park Bench:** The scene requires a park bench. Its orientation is important
    , as it should be facing the tree. So we need a 3D model of a park bench.

3.  **Man:** The scene includes a man. The man is located on top of the park
    bench and oriented toward the tree. This means we need a 3D model of a man in
     a pose that allows them to be on top of the bench and looking at the tree.

The assets necessary are: tree, park bench, man
"""
```

That is then translated into the following list of descriptions using the prompt depicted in 4.2:

```
"""
["tree", "park bench", "man"]
"""
```

After having the objects identified and the respective descriptions that match the input, for each object description, the following is done:

- An embedding vector is created for the constructed description using the CLIP model [58], and then the ClipScore [59] is calculated between the embedding of the constructed description and the pre-formulated embeddings of the images of the assets in the repository. From those, the top n (we are using n = 15) assets with greater ClipScore are selected.
- A final selection is made where we ask an LLM to pick a specific asset to represent an object by comparing the generated description with the descriptions made offline of the top n assets present in the repository.

There could be a different variation to this step, namely the use of a VLLM during the pipeline to select the best asset candidate by analyzing the images of the assets, but in general the use of VLLMs is more resource intensive and time consuming, so we choose not to use them because this method is functional and supported by the literature documented in [57] [60] [61] [62].

As an alternative to using a set of already-made assets, or even as an extra tool alongside the use of this asset repository, we could use methods to create the assets. This would

make the method more versatile, especially when the quantity of assets available is small. But it would come at some costs, namely a slower and heavier pipeline, more variability in the results, and another potential step to be evaluated.

4.3.3 Qualitative formulation

This step is all about translating the scene description into a qualitative and structured textual representation of the scene. As evident from the literature, a step like this is mostly always used to ensure a better plan to specifically arrange all the different objects of the scene to accomplish the desired configuration. To achieve this, we use an LLM.

As it is known, the scene graph can be represented in numerous forms. For the sake of efficiency and readability, we decided not to use the general JSON format, as represented in 3.2. Instead, we chose a more compact version with just the essential information. It is important that when defining the scene graph, an anchor object is defined, whose position and orientation do not depend on any other object, to facilitate the definition of the relations, and further, facilitate the construction of the scene.

Following is a clear example for the scene graph constructed using the scene description mentioned in 4.3.2 as input:

Scene graph example, generated using the prompt depicted in the listing A.3 from the Appendix A.1

```
### Object: tree
*   Object ID: tree_1
*   Intrinsic Attributes: None
*   State:
    *   Stance: Upright
    *   Support: Grounded
    *   Independent spatial and orientational state:
        *   Central Position
        *   Orientation: Oriented towards +Z axis.

### Object: park_bench
*   Object ID: park_bench_1
*   Intrinsic Attributes: None
*   State:
    *   Stance: Upright
```

```
    *   Support: Grounded
*   Spatial and orientational relationships:
    *   Spatial relationship: Near to tree_1.
    *   Orientational relationship: Oriented towards tree_1.


### Object: man
*   Object ID: man_1
*   Intrinsic Attributes: None
*   State:
    *   Stance: Upright
    *   Support: On top of park_bench_1
*   Spatial and orientational relationships:
    *   Spatial relationship: On top of park_bench_1, centered on the bench seat.
    *   Orientational relationship: Oriented towards tree_1.
```

Note how there is just a basic description of the objects with a detailed overview of their state and their relations. There is no information about explicit coordinates, orientations, or dimensions. This is intentional, as we want a fully qualitative scene graph. Explicit numerical data are, in general, to be processed with other external resources, as we want to minimize the LLM utilization of specific numerical values, like most of the methods literature mentioned in 3.1.2

4.3.4 Function Library

The function library is not an explicit step of the pipeline, but a tool set to be used by the LLM when generating code to construct and edit 3D scenes.

This tool set enables the construction or editing of 3D scenes with an abstraction to the necessary numerical and compositional considerations to place objects in certain places on the scene, or orient objects in specific directions.

To enable this abstraction, the function library is composed of custom-made functions with high-level parametrization implemented in Python that use the Blender API. These functions perform specific and generic operations with the objects in the Blender environment.

For a better understanding, we will give a specific example. One of the functions in this code library is called ”place_relative_to_object_facing” . This function’s purpose is to

place an object (let's call it object 1) in relation to another object (let's call it object 2), taking into account the orientation of object 2. With the input of just the pointers of object 1 and object 2, a flag to the relative position (left, right, front, etc) and the desired distance (this distance is between the nearest point of the bounding boxes of the two objects) this functions does the necessary calculations and adjusts the coordinates of object 1 to be correctly placed in relation to object 2.

Now, a similar philosophy is used for many more functions, not only for relative placements between objects, but also to define relative orientations, positioning, and orientation relative to the axis, and functions to regulate the state of an object (if it is upright, grounded, etc). Note how these functions were constructed for very generic operations involving mostly up to two objects. They could be further extended to accomplish in a single function call more complex and specific configurations involving more than two objects, or adjusted to be used for specific objects, but to keep the function library as general as possible, we implemented just these operations. It is worth noting that if used and combined correctly, these functions alone should suffice to accomplish most of the scene's typical configurations.

In the Appendix A.2, there are more details on the functions made available in our pipeline.

4.3.5 Scene code generation

This step is where we actually construct the first iteration of the scene.

At this point in the pipeline, we have the scene graph and the objects mentioned in the description, with the respective pointers to the most appropriate assets.

Giving the LLM access to the function library, we can already generate the desired scene by making the LLMs create a Python script that leverages the library functions to build the 3D scene. Although this is a possibility, we introduced an intermediate step.

To generate this script, there are many factors to consider. The objects and their overall configuration, all the functions existent in the library, the adequate functions to achieve a given configuration, the parameters to use in the function that better align with the objective and a necessary execution order (as the reference object of a relation needs to be in its final and correct position and orientation to then be correctly used as reference to adjust other objects position or orientation).

For these reasons, we make the LLM generate an ordered plan step by step, on how the scene should be accomplished. This plan promotes the explicit consideration of all factors mentioned above before generating the actual script, enabling a more systematic and well-planned script construction. This is mainly based on the CoT prompting technique [74], to make the LLM explicitly divide the plan by individual and ordered steps, and to have access to the ”reasoning” behind the LLM scene generation script. Then this plan is transformed by the LLM in the actual script that constructs the 3D scene.

For example, using the scene graph in the example introduced before, we have the following plan:

Scene execution plan example (not complete), generated with the prompt depicted in the listing A.4 from the Appendix A.1

```
[
  {
    "step_id": 1,
    "action": "IMPORT_OBJECT",
    "object_id": "tree_1",
    "description": "Import the 'tree_1' object into the scene.",
    "function_call": {
      "name": "import_asset",
      "parameters": {
        "filepath": "path_to_your_tree_asset.obj",
        "object_name": "tree_1"
      }
    },
    "qualitative_instruction": null
  },
  {
    "step_id": 2,
    "action": "SET_STATE",
    "object_id": "tree_1",
    "description": "Set 'tree_1' to upright stance.",
    "function_call": {
      "name": "set_object_upright",
      "parameters": {
        "object_to_orient": "tree_1"
```

```
      }
    },
    "qualitative_instruction": null
  },
    (...)
]
```

And with this plan and the assets selected, we got the following scene generation script:

Scene generation script example, generated with the prompt depicted in the listing A.5 from the Appendix A.1

```
from code_library.code_library import *
import mathutils

# Assets paths:
tree_asset = '../made_assets/tree/tree.obj'
park_bench_asset = '../made_assets/park_seat/park_seat.obj'
man_asset = '../made_assets/men/men.obj'

# Step 1: Import the 'tree_1' object into the scene.
import_asset(filepath=tree_asset, object_name='tree_1')
# Step 2: Set 'tree_1' to upright stance.
set_object_upright(object_to_orient='tree_1')
# Step 3: Set 'tree_1' to grounded.
set_object_grounded(object_to_ground='tree_1')
# Step 4: Place 'tree_1' at the center of the scene.
place_object_at_coordinates_simple(obj_ref_or_name='tree_1', coords=(0.0, 0.0,
    0.0))
# Step 5: Orient 'tree_1' towards the +Z axis.
align_object_to_axis(object_to_align='tree_1', target_axis='+Z')
# Step 6: Import the 'park_bench_1' object into the scene.
import_asset(filepath=park_bench_asset, object_name='park_bench_1')
# Step 7: Set 'park_bench_1' to upright stance.
set_object_upright(object_to_orient='park_bench_1')
# Step 8: Set 'park_bench_1' to grounded.
set_object_grounded(object_to_ground='park_bench_1')
# Step 9: Place 'park_bench_1' near 'tree_1'.
```

```
place_relative_to_object_facing(obj_to_move_input='park_bench_1',
    reference_obj_input='tree_1', direction='RIGHT', distance=1.0)
# Step 10: Orient 'park_bench_1' towards 'tree_1'.
orient_object_towards(object_to_orient='park_bench_1', target_object='tree_1')
# Step 11: Import the 'man_1' object into the scene.
import_asset(filepath=man_asset, object_name='man_1')
# Step 12: Set 'man_1' to upright stance.
set_object_upright(object_to_orient='man_1')
# Step 13: Place 'man_1' on top of 'park_bench_1', centered on the bench seat.
put_on_top_normalized(top_obj_input='man_1', bottom_obj_input='park_bench_1',
    normalized_x=0.5, normalized_y=0.5)
# Step 14: Orient 'man_1' towards 'tree_1'.
orient_object_towards(object_to_orient='man_1', target_object='tree_1')
```

4.3.6 Feedback Loop

At this moment in the pipeline, we already have a constructed 3D scene in a Blender file. The problem is that even with all the steps and measures used to try to construct a 3D scene that has a good composition and follows the user description, more often than not, there are problems in the 3D scene built. These problems can be very diverse, from objects overlapping, being in the wrong position, wrong orientations, etc.

For this reason, and inspired by the literature [57] [79], we decided to make a step where the current state of the scene is described and reviewed, and changes are applied if any compositional problems that go against the objective are detected.

The feedback loop implemented is represented in figure 4.3.

The first step, the scene graph processing, constructs a scene graph of the current 3D scene. This is different from the previous scene graph construction in the logic formulation step 4.3.3, as this scene graph is constructed with the data extracted from the .blend file containing the 3D scene (the information extracted is about the current assets in the scene and their layout, namely their positions and orientations), instead of based on a textual description, leveraging the Blender API to extract the information, to then formulate the scene graph with an LLM.

After having a scene graph representation of the current scene, we have two scene graphs: the current scene graph, which is the one constructed in this step, and represents

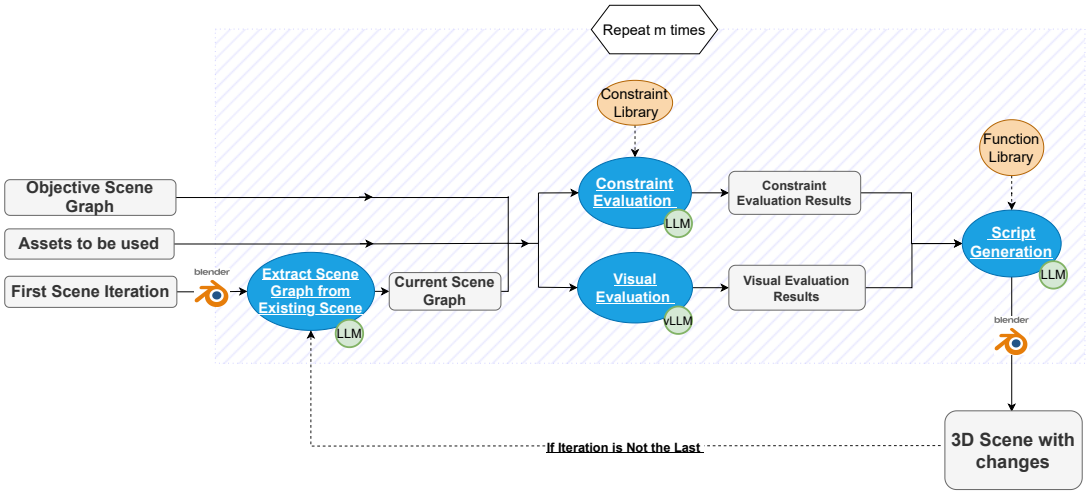


Figure 4.3: Feedback loop pipeline

the current iteration of the scene, and the objective scene graph, which has been built previously in the logic formulation step in section 4.3.3, represents the objective scene to be built. This is important because in the next steps in the feedback loop, we always use the current scene graph as an extra descriptive reference to how the state of the scene is at the moment, and the objective scene graph is used as the objective configuration to achieve.

After this, there are two evaluation steps: the visual evaluation and the constraint evaluation. In the visual evaluation, we use a VLLM to visualize the scene and provide feedback, focusing on what is wrong (basically anything that goes against the objective scene graph). In the constraint evaluation, we use a custom-made framework composed of various functions that aim to give scores on individual spatial, orientation-related, and state-related constraints of the current scene. After these scores are calculated, a summarized report is built and given to an LLM to highlight the key aspects of the scene that could be wrong based on the report and suggest possible solutions (again using the objective scene graph as reference and the current scene graph as extra context of the current state of the scene).

After the visual evaluation step and constraint evaluation step are completed, we are left with two reports on possible errors and areas to improve on the current scene. With these reports, the context given by the current scene graph and the objective scene graph, and access to the code library, an LLM builds a script to make the changes in the scene, with the objective of correcting the mistakes detected.

This feedback loop can be used multiple times. It is an important remark that we did not

implement an automatic detection mechanism to stop the feedback loop, as it is difficult to define a good stopping point. Instead, we just define a predefined m number of times, that is $m = 2$. By experimentation seems to be a sufficient number of iterations to achieve good results, without very long processing times.

We decided to use this to evaluation mechanism as they are by far the most used methods in the literature, including the methods mentioned in 3.1.2

In the following subsections, we will go more into the core process in the feedback loop.

4.3.6.1 Scene graph processing

Having a scene graph representation of the current state of the scene enables the LLM to have a context of the current overall state of the scene. This is relevant, not only because the other forms or assessment of the current state of the scene are not enough to have full context of the scene (in the image evaluation there always missing information as realistically we cant feed the VLLM with all the possible points of view, and for the constraint evaluation as it only gives very localized overview on specific constraints in the scene, lacking a broad context) and because it enables a direct comparison between the current state of the scene and a conceptualized objective scene, because they are both in a scene graph format.

The way the scene graph of the current iteration of the scene is built is similar to the reference scene graph made in 4.3.3, but this time instead of creating it from a textual description, we construct it based on a textual summary of the objects present on the scene, along side their coordinates and orientations extracted from the scene using the Blender API. This, of course, is not perfect as going from a numerical depiction of the scene to a more high-level qualitative representation of the scene is a hard task. It is necessary to understand the relative positioning and orientations between objects and contextualize them into a qualitative description, while also taking into account the types of objects involved. But still, generating this scene graph with LLM gives an insightful textual depiction of the current scene.

This is an example of a formulated scene graph:

Extracted scene graph, generated with the prompt depicted in the listing A.3 from the Appendix A.1

```
 ### Object: tree_1
*   **Object ID:** tree_1
*   **Intrinsic Attributes:** Tree
*   **State:**
    *   **Stance:** Upright
    *   **Support:** Grounded
    *   **Independent spatial and orientational state:**
        *   Central position of the scene: \[0.0, 0.0, 0.0]
        *   Orientation: Aligned with the world axes.
*   **Size:** Approximately 7 units wide, 7 units deep, and 11 units tall.

### Object: park_bench_1
*   **Object ID:** park_bench_1
*   **Intrinsic Attributes:** Park Bench
*   **State:**
    *   **Stance:** Upright
    *   **Support:** Grounded
*   **Spatial and orientational relationships:**
    *   Spatial relationship: Located approximately 5.79 units away from the
    tree_1 along the positive x-axis.
    *   Orientational relationship: Rotated approximately 45 degrees around the y
    -axis in relation to the central position of the scene.
*   **Size:** Roughly 2.4 units wide, 0.6 units tall, and 1.1 units deep,
    relative to its rotated orientation.

### Object: man_1
*   **Object ID:** man_1
*   **Intrinsic Attributes:** Man
*   **State:**
    *   **Stance:** Upright
    *   **Support:** Grounded
*   **Spatial and orientational relationships:**
    *   Spatial relationship: Located very near the park_bench_1.
    *   Spatial relationship: Located slightly behind the park_bench_1 on the X
    axis (Man X: 5.767, Bench X: 5.794)
    *   Spatial relationship: Located slightly above the park_bench_1 (Man Z:
    0.514, Bench Z: 0.005)
```

```
    *   Orientational relationship: Rotated approximately 45 degrees around the y
    -axis in relation to the central position of the scene, very similar to the
    park_bench_1.
*   **Size:** Roughly 0.6 units wide, 0.34 units deep and 1.7 units tall,
    relative to his rotated orientation.
```

4.3.6.2 Visual evaluation

A visual evaluation method using VLLMs is mentioned multiple times in literature, as referred in 3.1.2. The reason behind this choice is clear: Not only because the visual capabilities of many VLLMs to understand visual information is remarkable, but also because it makes sense to give the model used to construct the scenes the ability to have a visual context of the scene, as a visual representation of a 3D scene is probably the most informative format available to access the state of a scene.

The way this evaluation is made is by making a VLLMs construct a review of the scene by giving it 3 different view angles of the 3D scene, front view, back view and top down view, while having context of the current scene graph and the objective scene graph.

Ideally, the use of more angles, which allows to have a better visual depiction of the scene, would be ideal (a good practice would be even to generate random views of the scene). But now we enter again the choice of possibly better quality vs resources needed. With this in mind, we choose the mentioned angles as they usually allow for a fairly good depiction of the 3D scene, without the need for many images. It also helps that we, based on complexity, divide the scene into sub-scenes, as mentioned in 4.3.1, so in this step, the scenes are not very complex.

Below is an example of the generated feedback:

Visual evaluation feedback example, generated with the prompt depicted in the listing A.6 from the Appendix A.1

```
## Scene Analysis Report
**Overall Assessment:** Major Deviations Found

### Issue 1: Man not on the bench
*   **Issue Type:** Relationship Mismatch
*   **Involved Objects:** `man_1`, `park_bench_1`
```

```
*   **Description:** The man is not sitting ON the bench. He appears to be
    floating next to it, with a spatial offset, failing to adhere to the `on top
    of` relationship specified in the objective scene graph.
*   **Scene Graph Reference:** The graph states: "man_1 ... Support: On top of
    park_bench_1" and "Spatial relationship: On top of park_bench_1, centered on
    the bench seat."
*   **Suggested Correction:** Adjust the position of `man_1` so that he is
    properly seated `on top of` `park_bench_1`, ensuring appropriate contact and
    centering on the seat. The X and Z coordinates must be adjusted to satisfy
    this constraint.

### Issue 2: Orientation of the bench.
*   **Issue Type:** Parameter Mismatch
*   **Involved Objects:** `park_bench_1`, `tree_1`
*   **Description:** The bench is not oriented towards the tree. It appears to be
     rotated about 45 degrees relative to the world origin. While the current
    scene graph reflects this 45 degree rotation, the objective scene graph
    specifies that the bench should be oriented towards the tree. These
    statements are contradictory.
*   **Scene Graph Reference:** Objective scene graph: "Orientational relationship
    : Oriented towards tree_1." Current Scene Graph: "Orientational relationship:
     Rotated approximately 45 degrees around the y-axis in relation to the
    central position of the scene".
*   **Suggested Correction:** Adjust the Y rotation of `park_bench_1` so that it
    faces `tree_1`. The Current scene graph has the bench at 45 degrees which
    means we need to add 135 degrees to that (45 + 135 = 180 which is opposite).

### Issue 3: Orientation of the Man.

*   **Issue Type:** Parameter Mismatch
*   **Involved Objects:** `man_1`, `tree_1`
*   **Description:** The man is not oriented towards the tree. His orientation
    matches that of the bench, rotated around the y axis. The objective scene
    graph defines that the man should be facing the tree.
*   **Scene Graph Reference:** Objective scene graph: "Orientational relationship
    : Oriented towards tree_1." Current Scene Graph: "Orientational relationship:
     Rotated approximately 45 degrees around the y-axis in relation to the
    central position of the scene, very similar to the park_bench_1."
*   **Suggested Correction:** Change the Y rotation of the man to face the tree.
```

```
**General Comments:** The scene exhibits significant deviations from the
    objective scene graph, specifically in object placement (man on the bench)
    and object orientation.
```

4.3.6.3 Constraint evaluation

Constraint evaluation is a common practice in the literature. This method complements the other tools of the feedback loop that we have, as instead of giving an overall assessment of the scene allows for a precise evaluation of specific relations and states present in the scene.

This can be done in many different ways. We decided to construct an evaluation framework that could easily reflect the quality of most of the elementary and common relations and states found in 3D scenes while being easily used and accessed by an LLM.

Our constraint evaluation framework is mainly composed of a set of scoring functions.

All these scoring functions follow a general pattern based on an exponential decay. The score S for a given constraint is calculated as follows:

$$S = \begin{cases} 1.0 & \text{if } D_{\text{metric}} \leq T_{\text{perfect}} \\ e^{-k \cdot (D_{\text{metric}} - T_{\text{perfect}})} & \text{if } D_{\text{metric}} > T_{\text{perfect}} \end{cases} \tag{4.1}$$

Or, more compactly, by defining an effective deviation D_{eff}:

$$D_{\text{eff}} = \max(0, D_{\text{metric}} - T_{\text{perfect}}) \tag{4.2}$$

$$S = e^{-k \cdot D_{\text{eff}}} \tag{4.3}$$

Where:

- $S \in [0, 1]$ is the calculated score. A score of 1.0 indicates perfect satisfaction (within tolerance), and values closer to 0.0 indicate increasing dissatisfaction.
- $D_{\text{metric}} \in [0, \infty)$ is the primary non-negative scalar deviation metric calculated for the specific constraint with objective formulations, using the numerical information of the related objects. This metric quantifies how much the current state deviates

from the ideal state for that constraint. This deviation can take a variety of forms, namely:

  - Penetration depth for the overlap constraint.
  - Shortfall distance for the minimum distance constraint.
  - Absolute vertical distance from target for the grounded object constraint.
  - Angular difference in radians for orientation constraints.
  - For multi-component constraints (e.g., the constraint that checks if an object is on top of another), this formula is applied to each component's deviation, and the component scores are then combined (weighted sum) to produce the final constraint score.

- $T_{\text{perfect}} \geq 0$ is the "perfect score tolerance". If the calculated D_{metric} is less than or equal to this tolerance, the constraint is considered perfectly satisfied, and the score is 1.0. This allows for minor, acceptable deviations without penalty, being especially useful for constraints where minor deviations are very penalized.

- $k > 0$ is the "decay rate" parameter. It controls how quickly the score decreases as the D_{metric} exceeds T_{perfect}. A larger k results in a steeper drop in score for a given deviation. This parameter, along side some internal weights for some of the constraints where tuned for each constraint, so that we got a somewhat homogeneous scale between all the constrains, because as we refereed before, the deviations between constrains represent different things, so is important to make a effort so a given score means a similar deviation from the ideal state across the different scoring functions.

The use of this framework in the pipeline resumes to make the LLM first translate the objective scene graph into a pre-established JSON where every constraint is formatted in a specific way. Then this JSON is used as input to the constraint evaluation framework, which gives in return a written summary of the results. This summary contains more than just the score for each of the constraints, it also gives the calculated deviation for each individual constraint, an assessment per constraint type, where some statistics like average, median, etc, and the same for the overall constraints. This summary is given to the LLM alongside the current scene graph and the objective scene graph to make a review of things that should potentially be improved and how.

There is an example of the constraints constructed for the case exemplified in the previous pipeline steps:

Example of constraint definition (not complete), generated with the prompt depicted in the listing A.7 from the Appendix A.1

```
[
    {
        "id": "tree_grounded",
        "type": "ensure_grounded",
        "object": "tree_1"
    },
    {
        "id": "tree_upright",
        "type": "align_local_top_with_world_up",
        "object": "tree_1"
    },
    {
        "id": "bench_grounded",
        "type": "ensure_grounded",
        "object": "park_bench_1"
    },
    (...)
]
```

And the feedback originated from the analysis of the constraint evaluation report:

Constraint evaluation feedback example, generated with the prompt depicted in the listing A.8 from the Appendix A.1

```
Let's analyze the constraint evaluation report and the current scene graph to
    determine which relationships are well-executed and which are not.

**Well-Executed Constraints (Score near 1.0):**

*   **tree_upright (Constraint 2):** Score is 1.0. The tree is perfectly upright.
```

```
*   **park_bench_grounded (Constraint 4):** Score is 1.0. The park bench is
    perfectly grounded.
*   **park_bench_upright (Constraint 5):** Score is 1.0. The park bench is
    perfectly upright.
*   **park_bench_oriented_to_tree (Constraint 7):** Score is 1.0. The park bench
    is oriented towards the tree as expected.
*   **man_upright (Constraint 9):** Score is 1.0. The man is perfectly upright.
*   **man_oriented_to_tree (Constraint 10):** Score is 0.998. The man is oriented
     towards the tree (very good).

**Poorly Executed Constraints (Score near 0.0 - 0.1):**

*   **tree_grounded (Constraint 1):** Score is 0.002.  This is very poor. The
    tree is significantly *not* grounded. The deviation is 0.2170. Looking at the
     scene graph, the tree is reported to be grounded but the score is almost
    zero so there is something wrong with the evaluation, may be there are
    offsets to ground that are not accounted for.
*   **man_on_park_bench (Constraint 8):** Score is 0.300. This is poor. While the
     horizontal alignment seems fine, the vertical positioning is significantly
    off. The man is not properly *on top* of the bench.  The vertical deviation
    is high (0.5930). Looking at the scene graph, it is confirm that the man Z
    (0.514) is slightly above the Bench Z (0.005), and given the scale deviation
    its a good insight of what is happening.

**Constraints with Intermediate Performance:**

*   **tree_faces_z (Constraint 3):** Score is 0.227. The tree is not oriented
    towards the +Z axis as expected. The angle deviation is 90 degrees. The scene
     graph states "Orientation: Aligned with the world axes.", which, if true, is
     a contradiction.  This suggests an error in either the scene graph
    description or the evaluation of this constraint.
*   **park_bench_near_tree (Constraint 6):** Score is 0.468. The park bench is "
    near" the tree, but the distance is further than the defined range. The bench
     is too far away along the +X axis (deviation 0.4858). The scene graph
    confirms the bench is located 5.79 units away from the tree along the x-axis.
     The "too_far" state indicates that the current distance violates the limits
    imposed by the constraint definition.

**Summary and Conclusions:**
```

```
1.  **Upright and Orientation:** The upright constraints for the tree, bench, and
     man are all perfectly satisfied.  The bench and man are also well-oriented
    towards the tree.
2.  **Grounding Issues:** The grounding of the tree is a major failure.
3.  **Man on Bench Problem:** The man's placement *on* the bench is not well
    executed, the man is not correctly vertically positioned *on* the bench.
4.  **Distance Issue:** The park bench is too far from the tree to satisfy the "
    near" constraint.
5.  **Tree orientation**: There is a problem with the way the tree is oriented
    towards the +z axis. The score is low, the deviation is very high and the
    scene graph contradicts it.

In summary, while the orientation and upright stances are generally good, the
    grounding of the tree and the placement of the man on the bench are the
    biggest problems. The distance between the park bench and the tree is also
    outside the desired range.
```

More details about the constraint evaluation framework are in the A.3, where we explicitly show the constraint evaluation report generated with our constraint evaluation framework.

4.3.7 Simple floor and weather maker

After the feedback loop, we designed two final steps that add a simple textured plane and weather to the scene. The aim of this step is just to improve the visual appeal of the scene, although these are simple place holders that could be greatly improved.

For the simple floor, we have a set of textures, and we make the LLM choose the texture that best fits the scene, so that the ground plane is created with that given texture.

For the weather, we have some pre-established weather conditions built with Blender native tools (fairly limited) and use the LLM to identify the best candidate for the specific scene description.

4.4. LLM Prompting

Showing all the prompts used in the pipelines would lead to a very extensive demonstration, as we have many steps involving LLMs. Considering this fact, we will provide an

overview of how the prompts were constructed, while providing some practical examples included in the pipeline.

Defining an optimal prompt for a specific task is generically not possible, there are numerous amount of ways to describe a given task. With this, we constructed prompts while following some good practices and using some effective prompting techniques documented in the literature.

These good practices revolve around just having an objective and structured text while developing the prompt. Having an objective connotation leads to a better definition of the task at hand, leaving the output of the model less prone to misinterpretation while performing the task. This is relevant in our prompts as we want the model to perform specific and often technical tasks. Having a structured prompt does not necessarily translate to a better output by the LLM, but it makes the prompt more readable, not only while designing it to make the process simpler, but also to make it easier for people to understand what the prompt describes.

To formulate the prompts, we mainly use the following prompting techniques:

- Zero-shot CoT [92].
- Role prompting [94] .
- Few-shot prompting [27].
- EmotionPrompt [93].

The reason behind using these specific prompt techniques is the fact that they are documented in literature to have a positive impact, while being simple to apply. It is noteworthy that there are prompts that do not necessarily use any explicit prompting technique. Behind their formulation, there was some trial and error until the outputs of the models seemed satisfactory enough (although not rigorously, because it would be very time-consuming to test every possibility), and these techniques were implemented according to the complexity of the task and/or if the output in previous trials was not satisfactory. Giving some explicit examples:

In the asset identification from the asset selection pipeline 4.3.2, we use a set of two prompts. One to identify the assets, and the other to filter the identification results to a usable format.

The prompt to identify the objects is as follows:

```
"""
Your task is to determine the assets needed to fulfill the user's request to
    build a scene.

This task is highly important, so strive for maximum accuracy. Carefully analyze
    the 'users request to ensure you account for all relevant details, reasoning
    step-by-step to arrive at the best possible answer.

You should reason to better understand the assets needed, while grasping the
    characteristics the assets should have. At the end of your answer, list the
    necessary assets in the following template:
"The assets necessary are: asset\_1 (details of asset\_1 if referenced in the
    request), asset\_2 (details of asset\_2 if referenced in the request), ..."

Note that only objects are to be considered needed assets. Do not consider a
    generic floor or weather as needed assets. But if specific floor elements are
     stated, like a road or rugs, you need to include them.

Here is the user's request: {user_request}
"""
```

Listing 4.1: Prompt for identification of the objects in the input description

Where we use the Zero-shot CoT by asking the LLM to reason "step-by-step" and the EmotionPrompt while referring to the importance of this task, "This task is highly important, so strive for maximum accuracy", while following an clear structure and using objective language, giving details on easy to misunderstand aspects like in the penultimate paragraph.

Following is the prompt to filter the object identification:

```
"""
Your task is to create a list of all needed assets, given the input.
You should focus on the section of the input that is formatted as follows:

"The assets necessary are: asset_1 (details of asset_1 if referenced in the
    request), asset_2 (details of asset_2 if referenced in the request), ..."
```

```
Example:
Input: <reasoning> "The assets necessary are: a car (red), a house, two trees"
Output: ["a red car","a house","two trees"]

Do not omit any details present in the input about the needed assets, and ensure
    that your output matches the format provided in the example.
Each entry should refer only to specific objects and possible visual
    characteristics.
Don't include information about the object's placement, orientation, or any other
     subjective detail that doesn't refer to a specific object.
For multiple instances of the same object, group them into a single statement
    that includes the quantity.
For example, if five similar chairs are needed, there should be only one entry
    referring to those chairs, saying "five chairs".

Assets needed: {assets_needed}
"""
```

Listing 4.2: Prompt to filter the object identification

While being clear by following a structure, it uses a few-shot prompting by giving an explicit example ”Input: <reasoning> ”The assets necessary are: a car (red), a house, two trees”. Output: [”a red car”,” a house” , ”two trees”]”

Another example is shown below, illustrating part of the prompt used for visual evaluation of the scene during the feedback loop at 4.3.6.2:

```
"""
Role:
You are an AI Scene Composition Analyst. Your task is to rigorously evaluate
    rendered images of a 3D scene against its corresponding objective scene graph
     to identify discrepancies and suggest corrections.

Task:
```

```
Analyze the provided images (front, back, top-down views) and compare the
    depicted object arrangement against the relationships and states defined in
    the `objective_scene_graph`. Identify specific issues where the visual
    representation deviates from the graph or exhibits significant visual errors.
     Output your findings and suggested corrections using the **structured
    Markdown format** specified below.

(...)

"""
```

There, we use role prompting to better clarify the task: ”You are an AI Scene Composition Analyst.”

The full prompt, together with the remaining relevant prompts, is depicted in the Appendix A.1.

4.5. Scene editing pipeline

Although not our main objective, we also implemented a pipeline that can do simple edits on a 3D scene. There wasn't much investment in developing this method and testing it. There is also the fact that scene editing is less explored in literature. But it is an potentially useful proof of concept that complements the scene construction pipeline.

In figure 4.4, there is a representation of the pipeline.

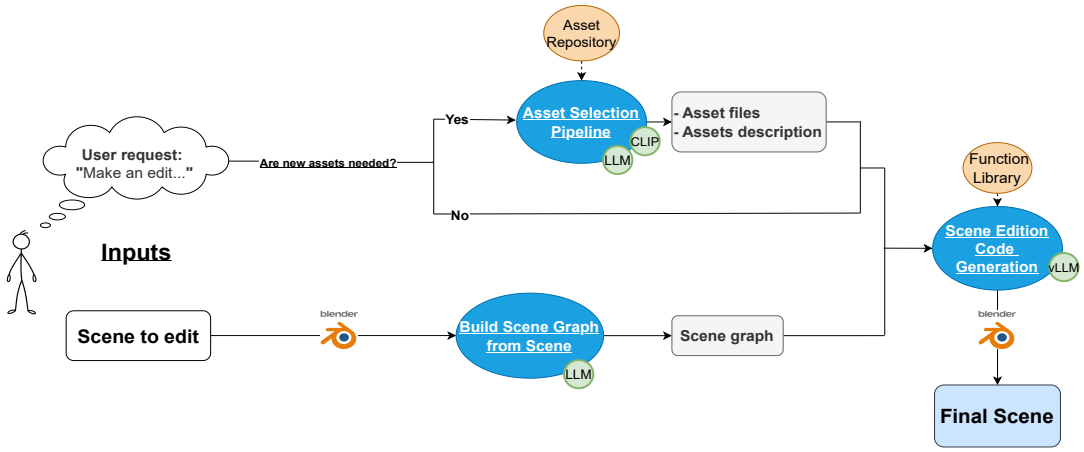


Figure 4.4: Scene edition pipeline

This pipeline takes as input the .blend file containing the 3D scene to edit, and a text description detailing the changes to make in the scene.

First, it is determined from the user's request whether insertion of new assets in the scene is necessary or not. If insertion is determined necessary, then we go to the asset selection pipeline, which is very similar to the one described in 4.3.2, if insertion is not necessary, the asset selection pipeline is skipped.

The scene to edit is also processed by constructing a scene graph with the data extracted using the Blender API. This scene graph construction is very similar to the one used in the feedback loop in the scene construction described in 4.3.6.1

After these independent steps, we go to the construction of the script to edit the scene. For this, we use a VLLM to inspect some images generated from the scene (using the front, back and top views) to gain further context and knowledge of the existing scene while having assess to the user request and the constructed scene graph and make it build the script (with the code library as reference.)

5. Results and Discussion

This chapter first describes the experimental setting used for the evaluation. It then presents the evaluation of the developed method: the pipeline for 3D scene creation from textual input. As part of this evaluation, the experimental results are presented and followed by a comprehensive analysis of their significance.

5.1. Experimental setting

The experiments were conducted on a laptop with an AMD Ryzen 7 7840HS processor, an NVIDIA RTX 4070 mobile GPU (8 GB VRAM), and 16 GB of RAM.

However, not all the resources were offloaded to this machine, as we used the Gemini API from Google to leverage a Gemini model [29] in our pipeline.

The pipeline was designed for modularity, enabling the integration of various models. Initial development involved testing local, open-source models. However, due to significant hardware and technical constraints, the models that could be run locally demonstrated considerable performance limitations.

Consequently, the decision was made to leverage the Gemini API to access more powerful models without being limited by local hardware. The Gemini 2.0 Flash model was selected, as it shows superior performance compared to the viable open-source alternatives, and because of its integrated vision capabilities, which allowed it to function as both the LLM and VLLM in the pipeline.

5.1.1 Assets

The 3D assets required for the evaluation were generated using the InstantMesh method [102]. To align with the project's focus on the autonomous driving context, the assets generated were all contextualized in the outdoor theme.

The choice of InstantMesh was mainly because it is lightweight and easy to set up. We tried other potentially better methods, like Trellis [103] or Hunyuan3D 2.0 [104], but we had trouble running these methodologies.

Some examples of the assets generated are depicted in figure 5.1. The way these assets are generated using the InstantMesh method is by using an image of the object that we

want to represent as an asset (the images were generated using the Gemini API). A full list of the assets can be found in the Appendix A.4.



(a) Generated asset of a tree



(b) Generated asset of a man



(c) Generated asset of a park bench



(d) Generated asset of a car

Figure 5.1: Generated assets examples

5.2. Evaluation

The evaluation of the 3D scene construction focuses on outdoor scenes, as they are the most relevant scenes in the autonomous driving context. The evaluation framework is divided into two distinct components: a quantitative evaluation and a qualitative evaluation.

- **Quantitative evaluation:** We leverage some metrics and methods to study the relevance of different components of the pipeline. It also includes an analysis of the method's scalability, by assessing the method's run time and metric performance on increasingly complex scenes.
- **Qualitative evaluation:** We explicitly show some generated scenes to illustrate the capacities and limitations of the developed method. We also study the generation variation over the same input description, and the method's sensitivity to certain variations in the input text.

A direct comparative analysis with methods from the literature would be relevant, but unfortunately, the methods documented in the literature somewhat similar to our own

(methods that are not domain specific, and that leverage LLMs to generate 3D scenes with textual descriptions), do not have the code open to the community.

In this context is relevant to mention that there is one publicly available method that could be used for a comparative analysis, the BlenderGPT [105], although it is not quite the same as it functions as an assistant within Blender. However, it was excluded from a direct comparison for two primary reasons. First, it lacks any accompanying scientific literature, making a rigorous methodological comparison difficult. Second, its implementation as a Blender extension is coupled with the OpenAI API. Adapting this framework to support the Gemini API used in this project would require a significant effort. With limited time, we couldn't execute the necessary modifications.

5.2.1 Quantitative evaluation

The primary objective of the quantitative evaluation is to systematically assess the methods using a defined set of metrics. To ensure the consistency and comparability of results, a dataset of scene descriptions was created. Each method under evaluation generated scenes from this common dataset, enabling a direct and fair comparison of their outputs.

As we want to assess the relevance of different steps in the pipeline, the evaluation procedure is an ablation study. This study compares the performance metrics of the complete pipeline against several ablated versions, each with a specific component removed.

It is also relevant to assess the scalability of our method, so we make an analysis of the evolution of the run time and the metrics scores with the increase in complexity of the scene to be generated.

5.2.1.1 Metrics choice

The set of metrics used is composed of the baselines that are mostly used in literature, while also experimenting with other metrics that can possibly give a more insightful overview. Following is a list of all metrics used:

- **Mean Overlapping Pairwise IoU (MOP-IoU):** A variation we formulated of the IoU metric (explained in the section 3.3), that can measure the severity of the present overlaps in the scene while not being influenced by the scene's scale. It's defined as follows:

Let $S = \{O_1, O_2, \ldots, O_N\}$ be the set of N objects in the scene, and IoU(O_i, O_j) the IoU between objects O_i and O_j bounding boxes

$$\text{MOP-IoU}(S) = \frac{\sum_{1 \leq i < j \leq N} \text{IoU}(O_i, O_j)}{\left| \{(O_i, O_j) \mid 1 \leq i < j \leq N, \text{IoU}(O_i, O_j) > 0\} \right|} \tag{5.1}$$

Where the numerator is the sum of IoU scores over all unique pairs of objects, and the denominator is the count of pairs with a non-zero IoU. If the denominator is zero (i.e., no objects overlap), the MOP-IoU is defined to be 0. This MOP-IoU can measure the severity of the overlap present on the scene while being somewhat independent of the scene scale (namely number of objects in the scene). A bigger MOP-IoU means the presence of a more severe overlap

- **ClipScore 3.1:** The baseline used to measure how well a given image of a 3D scene matches a given text description. As mentioned in 3.3), it is defined as the cosine similarity between the embeddings of an image and a text description using the baseline CLIP model from OpenAI [58].

- **SigLip 2 [9] "score":**. Defined in a similar way as the ClipScore, but instead of using the baseline CLIP model from OpenAI, leverages a similar model that outperforms CLIP in a variety of tasks, which hopefully can lead to a better evaluation of the text-image alignment.

- **VQAScore [8]:** A Visual Question Answering (VQA) based metric that leverages an VLLM to assess text-image correspondence in a more nuanced manner. It evaluates aspects like object composition, spatial relationships, and aesthetics by generating and answering specific yes or no questions about the image based on the prompt.

- **ImageReward [7]:** Measures the alignment between description and image pairs with a model trained in pairs rated with scores by people, being able to output a score directly, that are aligned with the human judgment.

- **Constraint average score:** Defined by the average score over all the constraints evaluated using our constraint evaluation framework.

- **Constraint precision:** Gives the proportion of constraints that were satisfied (or almost fully satisfied) in relation to all constraints defined using our constraint evaluation framework.

5.2.1.2 Description dataset

The dataset of scene descriptions used to make the quantitative evaluation is composed of 40 scene descriptions handwritten by us, that depict outdoor scenes and are composed exclusively of objects available in a set of pre-generated assets (a full list of the assets is present in the Appendix A.4). Additionally, each description is annotated with the total number of objects it specifies.

5.2.1.3 Ablation study

The ablation study was designed to evaluate the contribution of key pipeline components by testing four distinct configurations:

- **The full pipeline**.
- **The pipeline without the feedback loop**.
- **The pipeline without the qualitative formulation (no scene graph)**.
- **The pipeline without the function library**.

For each configuration, every scene description in the dataset was used to generate a corresponding 3D scene. Afterwards, the full set of evaluation metrics was calculated for each generated scene.

For the text-to-image alignment metrics, three images were rendered from distinct perspectives: frontal, rear, and top-down. The final score for each scene was determined by taking the maximum value from these three views. This procedure serves to minimize the impact of perspective bias present in these models.

The results of the ablation study are summarized in three tables: 5.1, 5.2, 5.3

Table 5.1: Ablation results for scene object overlap. Lower scores indicate less overlap.

Method	**MOP-IoU** (↓)
Full Pipeline	0.1052
No Feedback Loop	0.0728
No Function Library	**0.0666**
No Qualitative Formulation	0.1485

The values of MOP-IoU in table 5.1 show that there is less overlap when the function library is not used. This is a nuanced finding. The code library is made to make precise

placements and orientations of the objects. There is also the fact that the pipeline without a feedback loop has less overlap than when using the full pipeline, pointing out that the feedback loop can also introduce overlap to the scene. The pipeline without qualitative formulation is the one with the biggest overlap, and this is not surprising, as the lack of planning of the scene layout leads to overlaps.

At first glance, these results might suggest that both the function library and the feedback loop are detrimental. However, there are some important details not considered. The presence of overlap is not a direct indicative of scene quality.

First, the MOP-IoU metric is calculated over the objects' bounding boxes. This means that there are some layouts where precise placements that adjust to the objects' mesh can lead to better scenes, while resulting in a bigger measured overlap. For example, in the scene in figure 5.2, the cat and the dog are meant to be on top of the park bench. There, a precise placement was made so that both are placed in the sitting area of the park bench. If this precise placement was not made, the animals could be placed flying over the bench, or on top of the backrest, which would lead to a less plausible scene.

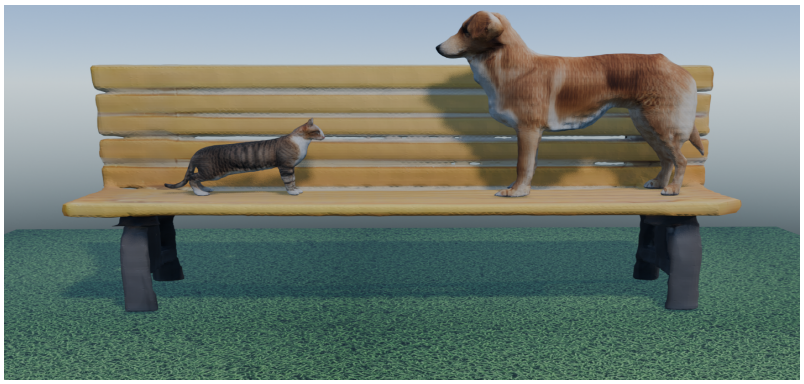


Figure 5.2: Image of a generated scene depicting a cat and a dog in a park bench.

A second factor influencing the MOP-IoU scores is the limited nature of the available asset library. The assets are available only in a single, static pose, which is lacking to fulfill many interactions. Consequently, achieving semantic alignment with a prompt sometimes requires creating intentional overlaps to produce a more plausible scene.

For example, in the scene depicted in figure 5.3, the asset representing a woman is available only in the upright position. To make it look like the woman is riding the bike, the asset is placed so it gives the impression that she is sitting on the bike seat, leading to some overlap while making the scene more aligned with the objective.



Figure 5.3: Image of a generated scene with a woman riding a bike.

Therefore, there is some evidence to believe that the additional overlap generated by the function library and feedback loop often corresponds to more successful and deliberate scene construction. The following results will further support this possibility.

Below is a list of the main takeaways from the analysis of the MoP-IoU average values in the table 5.1:

- The pipeline without qualitative formulation results in the most object overlap, as the lack of planning in the form of a scene graph leads to a more uncoordinated scene layout.
- The additional overlap introduced by the function library and feedback loop can often correspond to more successful scenes, as it can originate from deliberate efforts to make precise placements or to compensate for the static poses of available assets.
- Consequently, MOP-IoU alone is not a direct indicator of scene quality, since semantically correct scenes can require intentional overlap in certain situations, which increases this metric, but can lead to useful findings.

Table 5.2: Ablation results for core performance and image-to-description alignment. Higher scores are generally better.

Method	**ClipScore** (↑)	**VQA Score** (↑)	**Siglip2 Score** (↑)	**ImageReward** (↑)
Full Pipeline	**24.37**	0.7504	0.0960	**-1.417**
No Feedback Loop	23.57	**0.7628**	0.0953	-1.521
No Function Library	24.21	0.7244	**0.0983**	-1.534
No Qualitative Formulation	23.87	0.7339	0.0920	-1.550

Before analyzing the text-to-image alignment scores in table 5.2, it is crucial to acknowledge their limitations. These metrics can be sensitive to factors like asset visual quality, the perspective of the image representation of the 3D scene, and ultimately depend on the

data used for training of the models employed to calculate these metrics (as these metrics are all model-based metrics). If a given scene image approximates more to cases used in training, a bias can appear in the result.

With these considerations in mind, the full pipeline demonstrates the best overall performance in alignment between the description and the generated scene. It achieves the highest scores for ClipScore (the baseline used to evaluate these methods) and ImageReward, while ranking second for the other two metrics.

Addressing the fact that the pipeline without the feedback loop has a higher VQAScore and the pipeline with no function library has a higher Siglip 2 score. One hypothesis is that these methods are more capable in some specific aspects. The feedback loop may introduce some errors to the scene that the VQAScore can capture better, or not using the function library leaves an opportunity for the LLM to perform some operations better, giving that is not restricted to the function library. But knowing all the details about why a given score is produced is very difficult (or even nearly impossible) in these cases. It's also interesting to see that the method with no qualitative formulation seems to be the worst performer, having the worst ImageReward and Siglip 2 score, and being the second worst on the other two. This finding provides evidence that the formulation of a scene graph is beneficial to the alignment between the generated scene and the description.

Below is a list of the main takeaways from the analysis of the text-to-image alignment metrics in the table 5.2:

- The full pipeline demonstrates the best overall performance in aligning the generated scene with the text description, achieving the highest score for ClipScore in ImageReward and the second-highest scores for VQAScore and Siglip 2, showing that all the evaluated components of the pipeline contribute to the alignment between the input description and the created 3D scene.
- The method without qualitative formulation is the worst overall performer, which provides evidence that formulating a scene graph is especially beneficial for the alignment between the input description and the created 3D scene.

The constraint evaluation results, depicted in the table 5.3, indicate that the full pipeline seems to be the best method, achieving the highest scores in both constraint precision and average score. Notably, the performance of the pipeline without the feedback loop

Table 5.3: Ablation results for constraint satisfaction. Higher scores indicate better performance.

Method	**Constraint Avg. Score** (↑)	**Constraint Precision (%)** (↑)
Full Pipeline	**0.7184**	**60.28**
No Feedback Loop	0.7095	59.80
No Function Library	0.6523	49.61
No Qualitative Formulation	0.6608	54.44

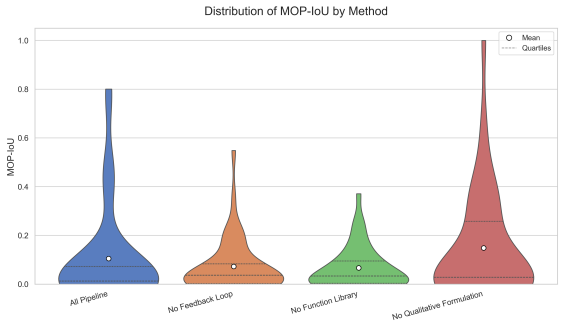
is very close to using the full pipeline in this regard, so the improvements introduced by the feedback loop do not seem to be very significant regarding local constraints. It is also noteworthy that the worst performer is the version that does not use the function library. This is expected, as without the function library, it is more difficult to define precise placements and orientations, resulting in more inaccuracies in this regard.

Below is a list of the main takeaways from the analysis of the constraint satisfaction metrics in the table 5.3:

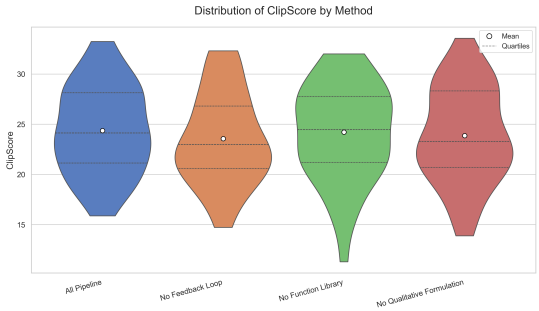
- The full pipeline is the most effective method for satisfying constraints, achieving the highest scores in both precision and average score. This shows that all evaluated components are beneficial for defining precise placements and orientations aligned with the input description.
- The pipeline without the function library is the worst performer, which is an expected outcome as it lacks the tools to define precise placements and orientations, showing that the function library is especially relevant to accomplish precise placements and orientations aligned with the input description.
- The improvements from the feedback loop are not very significant for the satisfaction of the constraints, as the pipeline without it achieves a similar performance to the full pipeline.

The information in the tables 5.1, 5.2 and 5.3 offers valuable initial findings, but it is possible to go further. An examination of the full data distributions allows for a more robust validation of these observations and may provide access to new information hidden in the provided summarized results. Figure 5.4 depicts these distributions using violin plots.

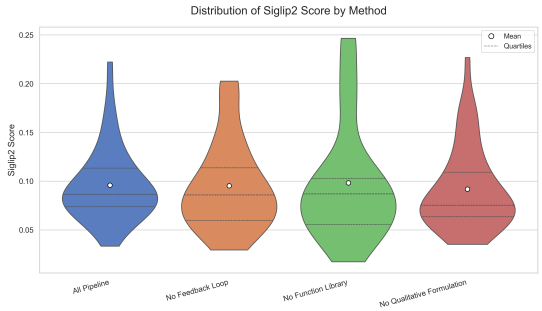
The MOP-IoU violin plot for the complete pipeline leads to a distribution heavily concentrated near zero while featuring a distinct tail of higher-value outliers. This can also be visualized for the cases where the feedback loop and function library are not used with



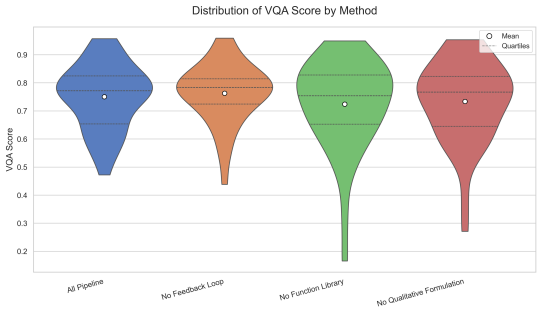
(a) MOP-IoU (Scene Overlap)



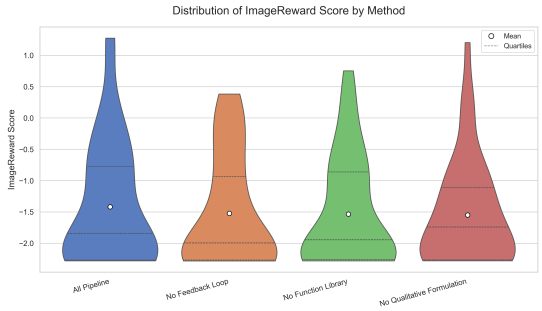
(b) ClipScore



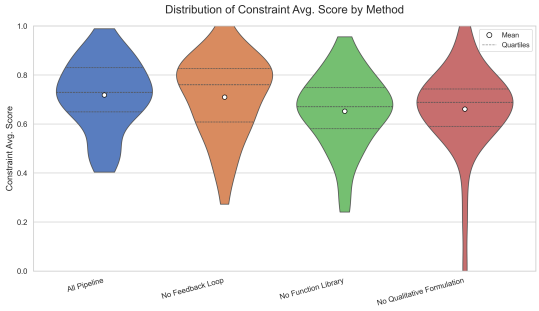
(c) Siglip2 Score



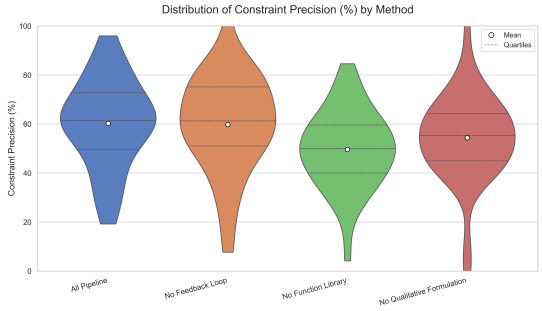
(d) VQA Score



(e) ImageReward Score



(f) Constraint Avg. Score



(g) Constraint Precision (%)

Figure 5.4: Distributions of performance metrics for each ablation method. The MOP-IoU metric is shown first, followed by pairs of related metrics for description to image alignment, and constraint satisfaction. The violins show the distribution shape, with internal dashed lines representing the quartiles and a white circle indicating the mean.

diminished tails of outliers. This reinforces the earlier hypothesis regarding intentional overlap, where overlap is induced on purpose to increase the overall ”quality”. This intentional overlap decreases as the capacity to make precise adjustments in the scene also decreases, shown by the progressively lower tails in the cases where the feedback loop and function library are not used, respectively. In contrast, the pipeline with no qualitative formulation displays a much wider and more uniform distribution, indicating that the presence of overlap is much more systematic, showing how relevant the planning of the scene is to decreasing the overlap.

A key initial observation across the text-to-image alignment plots is the differences in the distributions for each metric, suggesting that each model is indeed sensitive to different aspects of scene quality.

In the case of the full pipeline, it is the method with the fewest outliers with bad scores. This indicates a better consistency in maintaining a certain minimal level of quality of the scene. This variation is slightly bigger in the pipeline without the feedback loop, and is much more accentuated in the configurations without the function library and no qualitative formulation. Namely is possible to see that even if the pipeline without the function library has the best Siglip2 score, it is also the one that generated scenes with the worst performance for that metric. This shows that, especially the function library and the qualitative formulation, really increase the consistency of the generated scenes. The feedback loop also increases the consistency, but it’s not so pronounced.

Regarding the constraint metrics, what stands out is the range of values of the pipeline without the qualitative formulation. This is expected as, without proper planning, when generating the code for the scene, it has fewer guardrails. The pipeline without the function library is notably the worst performer. This was expected, as without the function library, the LLM loses the tools to generate precise layouts. The full pipeline and the pipeline without the feedback loop are similar, although the pipeline without the feedback loop seems to add more variance to the results, which is expected given that it lacks the internal evaluation steps, resulting in a less processed result.

Below is a summary of the analysis of the violin plots in figure 5.4.

- The MOP-IoU distribution reinforces the hypothesis of intentional overlap. The full pipeline shows a tail of deliberate high-overlap outliers. The absence of qualitative

formulation results in widespread, systematic overlap, highlighting the importance of planning.

- The analysis of the text-to-image alignment metrics reveals that the full pipeline is the most consistent, producing the fewest low-scoring outliers. The function library and the qualitative formulation, in particular, are shown to be crucial for increasing the consistency of the generated scenes, as the pipelines without those elements show distributions with high variance.
- For constraint metrics, the pipeline without the feedback loop is similar to the full pipeline but introduces more variance, as it lacks internal evaluation steps. Also, the relevance of the function library is made more clear, as the pipeline without the function library had notably lower values for these metrics.

5.2.1.4 Method robustness over scene complexity

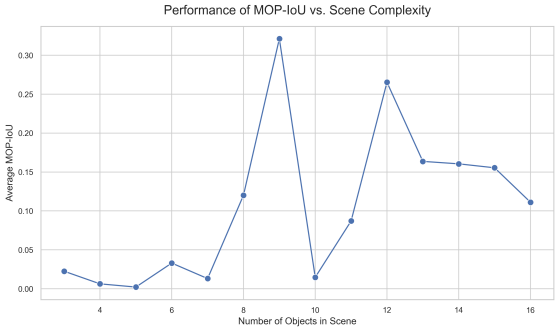
Visualizing the evolution of the metrics as we increase the scene complexity can give us information on whether the method developed is scalable or not.

In figure 5.5 are plots that show the evolution of the metrics with the increase in object count in the scenes.

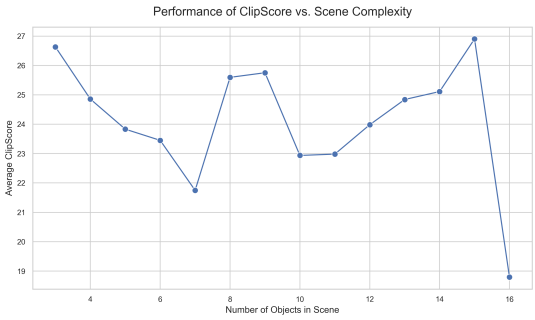
The MOP-IoU plot reveals an increase in overlap with the number of objects, although there is a high variability, showing that it mostly depends on the individual scene setting rather than the number of objects. So there is an indication of a mild increase in overlap with the scale of the scene, but nothing dramatic.

The text-to-image alignment metrics exhibit divergent trends. The ClipScore shows no dependence of the number of objects on the scene, being dependent on the individual scenes. For the Siglip 2 score, there is an unexpected positive trend with the number of objects. There isn't any clear evidence to believe that the scene alignment with the description increases with the object count. This is most likely due to model bias towards giving bigger scores to bigger/more complex scenes.

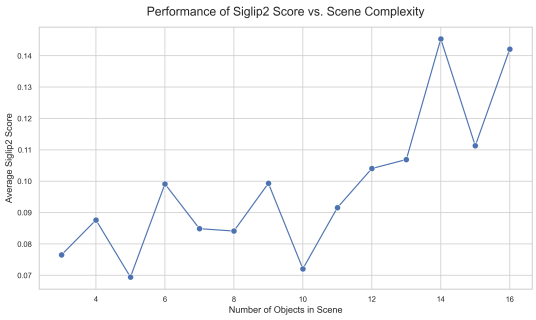
In contrast, both the VQscore and ImageReward score show a decreasing trend, while still exhibiting a strong variance, suggesting to have more of an individual scene dependence. In these cases, there is the suggestion that the quality decreases with the number of objects, although it can also be because of a similar reason as the Siglip 2 score, because the methods have a bias, which in this case penalizes bigger/more simple scenes.



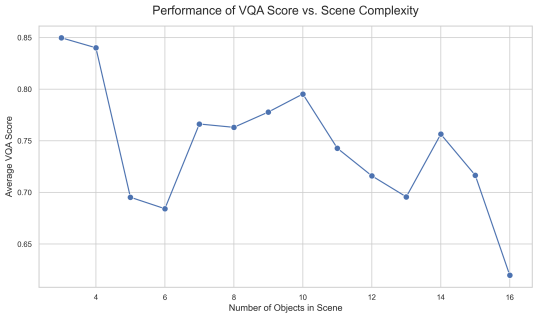
(a) MOP-IoU (Scene Overlap)



(b) ClipScore



(c) Siglip2 Score



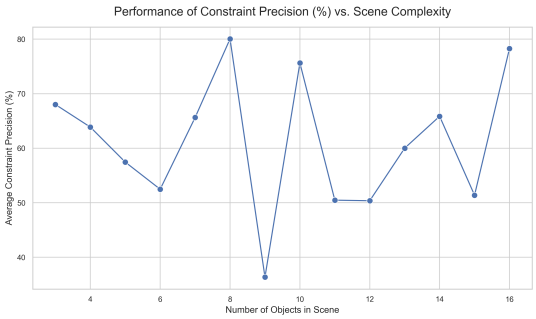
(d) VQA Score



(e) ImageReward Score



(f) Constraint Avg. Score



(g) Constraint Precision (%)

Figure 5.5: Scalability analysis of the full pipeline. Each plot shows the trend of the average metric score as a function of scene complexity (number of objects).

The constraint satisfaction score and precision show no visible correlation with the number of objects. This indicates that the pipeline's ability to adhere to specific, local constraints is robust and does not degrade as the overall scene complexity increases.

In summary, while some metrics suggest a slight degradation in quality with increasing scene complexity, the evidence is not very conclusive and the effects are not dramatic. Within the tested range of 3 to 16 objects across 40 scene descriptions, the results indicate that the method's performance is not very impacted by the scale of the scene, demonstrating reasonable scalability.

5.2.1.5 Run time analysis

In addition to quality metrics, the runtime of the scene generation process was measured. This can be useful to access if we can run our method in a reasonable amount of time. We can also analyze the scalability of the method, given that we have the number of assets of each scene.

In figure 5.6 is the run times in relation to the number of objects in the scene. We show both for the full pipeline and the pipeline without the feedback loop, as it gives some extra insights.

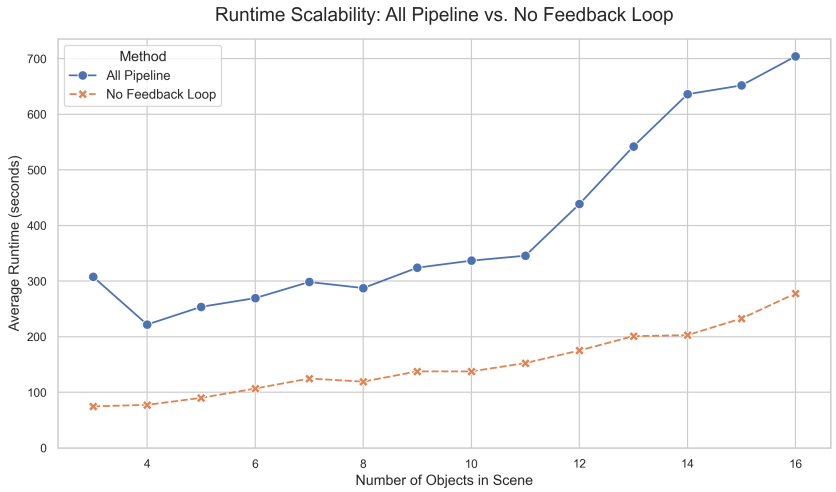


Figure 5.6: Run time in relation to the number of objects for the full pipeline and the pipeline without the feedback loop

There is an increase in run time with the number of objects on the scene. This is expected, as there are more sub-scenes to process, more assets to select, and the LLM needs to process and generate more tokens. For the full pipeline, the times go from around 5 minutes for very basic scenes, up to a little over 10 minutes for scenes with 16 objects.

While these specific times are not really representative, as they depend on the hardware used and the overall response time of the Gemini API, it still shows that this method is able to generate 3D scenes in a reasonable time. For scenes with increased complexity, the run time can start to become long, so depending on the aimed complexity of the scene, we can say that there are some scalability issues, although it is kinda inevitable that a method like this experiences an increase in run time with the increase in scene complexity.

We also represented the run times for the pipeline without the feedback loop to show that this step is responsible for a big percentage of the run times, as the times are cut basically in half with a pipeline not using it. In some scenarios where a decrease in performance is acceptable while valuing the run time, using the pipeline without the feedback loop could be a viable choice.

5.2.2 Qualitative evaluation

This section presents a qualitative evaluation of the pipeline, using curated examples to visually demonstrate its capabilities, limitations, and characteristic behaviors across various scenarios. The analysis is structured as follows:

- **Successful generations examples:** A showcase of well-executed scenes that highlight the method's strengths in producing plausible and semantically coherent outputs.
- **Failures and limitations examples:** An examination of less successful generations, which serves to identify and analyze common failures and limitations of our approach.
- **Generation variability:** An exploration of the random nature of the pipeline by presenting multiple outputs generated from the same input prompt.
- **Sensitivity to input variations**: A brief analysis of the method's response to certain modifications in the input text.

5.2.2.1 Successful generations examples

Figures 5.7 to 5.12 present some well-executed 3D scenes generated from given text descriptions. We selected scenes with very different (and maybe a little strange) settings to show the potential of the method. Note that a good generation does not necessarily

mean a very well-structured layout, but mainly a good alignment with the request in the input text.

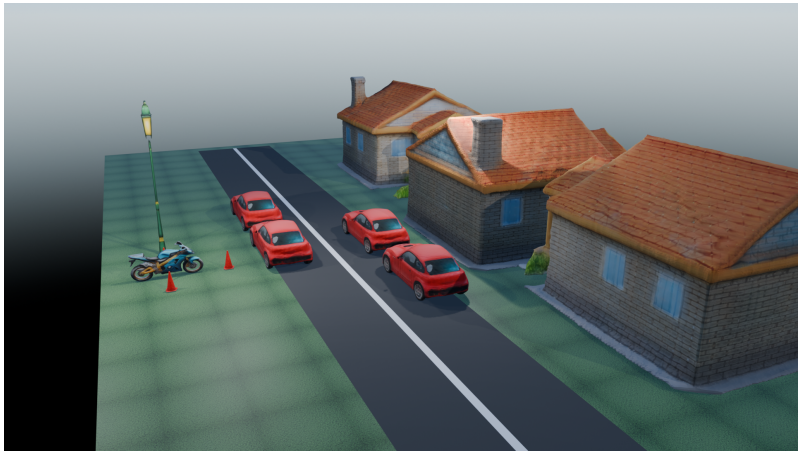


Figure 5.7: ”The scene has a straight road in the middle, with 4 cars. 2 of the cars are on the right side of the road, while the other 2 are on the left side. On the left side of the road are 3 cones that are protecting a motorbike that had an accident. On the right side, there are 2 small houses close to each other, and a third house slightly more distant. Close to the motorbike, there is also a street lamp.”



Figure 5.8: ”One cat and a dog face each other, almost fighting on top of a park bench oriented towards the -x axis. Nearby should be a sunflower. Far away, a woman riding a bike looks at the cat and the dog.”

These examples demonstrate that the pipeline is capable of generating scenes with some level of complexity and quality, successfully adhering to the input descriptions.

For instance, the method can make precise positions. For example, when putting the vehicles on top of the road in the respective lanes in figures 5.7 and 5.11, or placing the cat and dog on top of the bench in figure 5.8.

The pipeline can define correct orientations, for example, in figure 5.9, the chairs are oriented to the table in the middle.

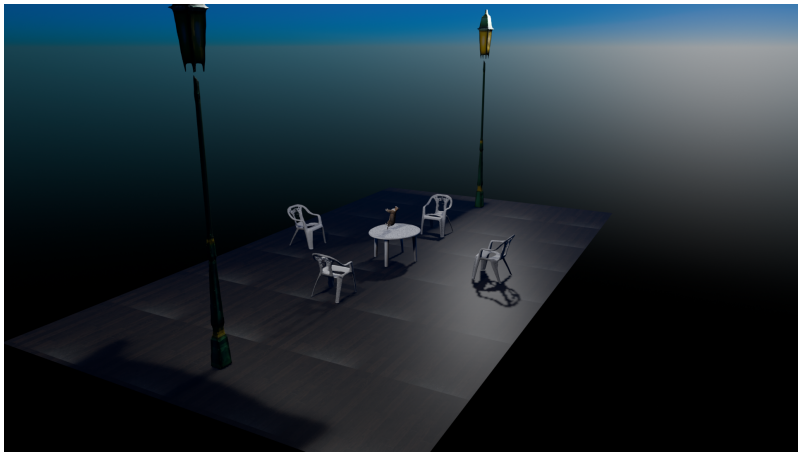


Figure 5.9: ”A table with four chairs around it, each oriented towards the table. On top of the table, a cat is sleeping. Nearby, two street lamps, far away on different sides of the table, illuminate the scene.”

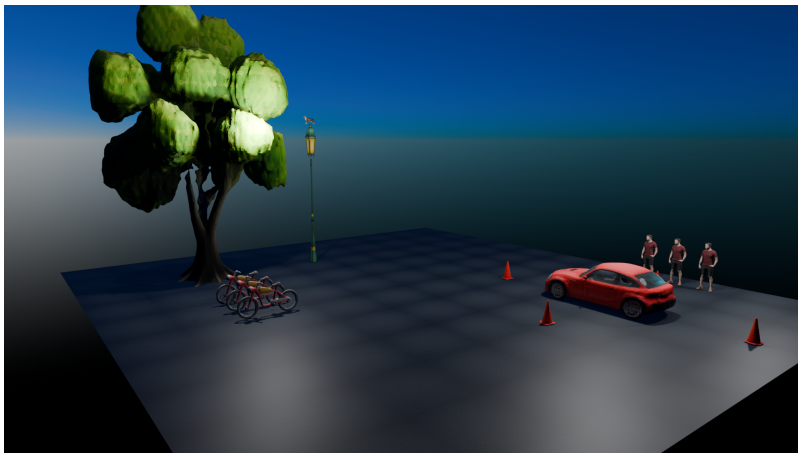


Figure 5.10: ”A car surrounded by 4 cones. Three people are side by side, not too far away, oriented toward the car. A good distance from the car, there are 3 bikes parked in a straight line, on the side of a tree. Behind the tree is a street lamp with a cat on top.”



Figure 5.11: ”Two parallel straight roads, aligned with the y axis. In between the roads should be a row composed of two trees and a bush in the middle. A motor bike passing on top of the right road, on the left lane, and two cars on the left road, on different lanes with opposing orientations.”

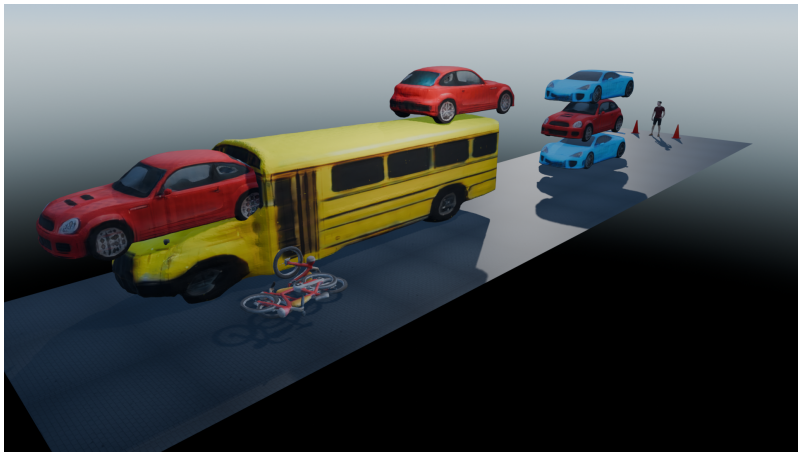


Figure 5.12: ”A junkyard composed of a bus is the center of the scene, with two cars on top of it, one in front of the other. A pile of three bikes near the bus, and behind the bus, a pile of one sports car on the bottom, one car in the middle, and another sports car on top. Very far from the bus, there is a man looking at the bus with two cones on its side.”

The method can extrapolate from the description to define details that are not specifically mentioned. Like in figure 5.7, which precisely puts one cone in front of the motorbike and two on the sides, like they are ”protecting” it.

Finally, the system can work around limitations of the available tools. For example, in our bank of assets isn't an asset of a cat sleeping, so it improvised a sleeping position by manipulating the orientation of the asset in figure 5.9. Or in figure 5.8, when putting the straight up asset of a woman riding a bike, where it correctly places the asset so it seems the woman is on the seat of the bike.

5.2.2.2 Failures and limitations examples

Figures 5.13 to 5.18 showcase several less successful 3D scene generations. An analysis of these examples provides insight into limitations and common failures of the developed pipeline.

The pipeline occasionally fails to execute correct placements of the objects. For example, in figure 5.17, the vehicles parked do not respect the configuration stated in the description, namely, they are not all ”side by side”. These kinds of errors sometimes occur, often as a result of occasional mistakes made by the LLM in steps such as interpreting the input description, planning the scene, and translating the planned scene into Python script using the function library.

The same can be said for the orientations. Not all orientations are correctly defined. For example, in figure 5.15, the chairs are not all correctly oriented towards the respective

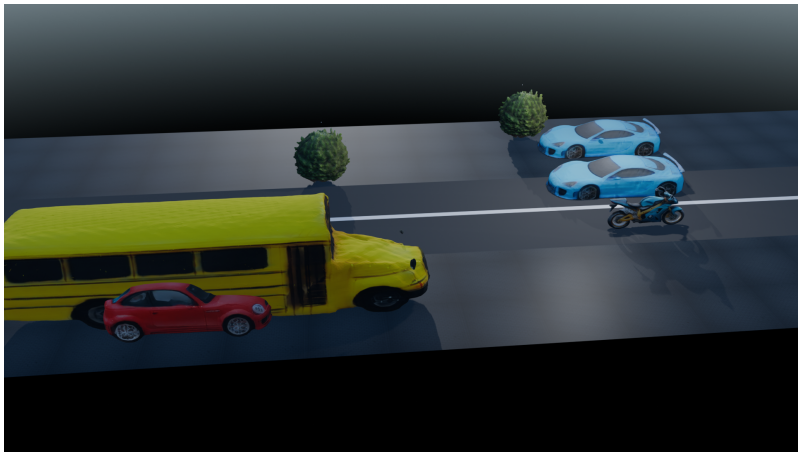


Figure 5.13: ”A straight road oriented to the x axis with 6 vehicles on top. On top of the road, in the right lane oriented to the -x axis, is a line of vehicles with a motorcycle, a bus, and a car. On the left lane of the road, a line of 3 sports cars is oriented towards the x direction. On the left side of the road, two bushes are placed.”



Figure 5.14: ”This scene has a very long straight road made up of two road parts (one right in front of the other), forming a continuous block of road. Then, on each side of the road are 6 small houses, 12 in total. They are spaced evenly and facing towards the road. In front of the fourth house on the right, a bus is driving along the road.”

table. Or in figure 5.14, where the houses on the side of the road have the opposite orientation of what they should have, being oriented away from the road.

There are also instances where the correct number of assets is not selected. For example, in figure 5.13 there are only 2 sports cars (the blue car) instead of 3.

Some unintentional overlap errors are also present in some scenes. For example, in figure 5.13 the red car is partially inside the bus.

The initial scene division process can sometimes result in a fragmented or illogical layout. For example, in 5.18 a clear scene fragmentation is present, that do not respect the input in its entirety.

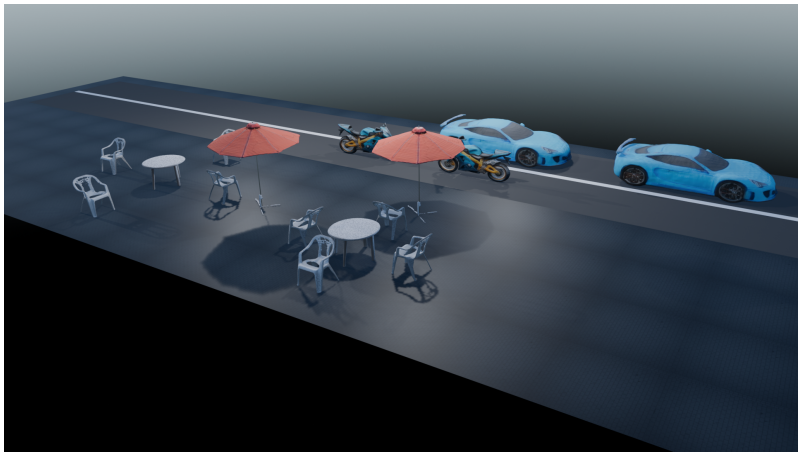


Figure 5.15: ”In the center of the scene, a table with four chairs, one at a different side of the table and oriented towards the table, near an umbrella. Not too far away, to the y direction, a complexly similar setup of table, chairs, and umbrella. To the x direction of on of the tables, and oriented to the y axis, a road with two motor bikes on top of it, in the left lane oriented to the y direction, and also on top of the road, now in the right lane two sports cars oriented to the -y direction.”

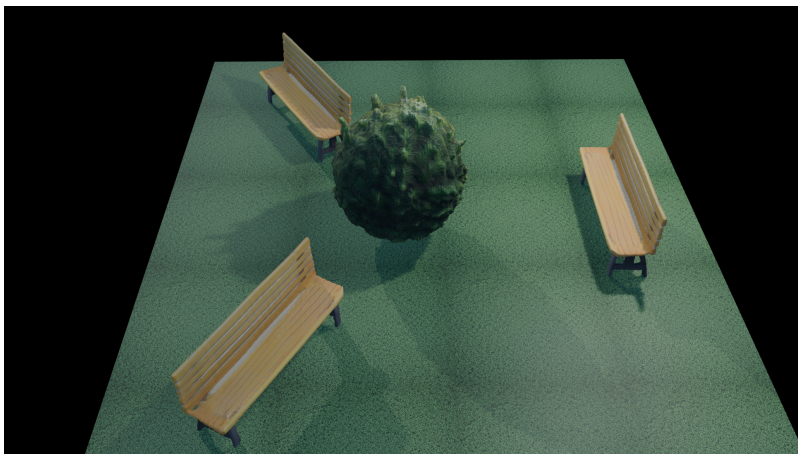


Figure 5.16: ”3 park benches facing each other forming a triangle. In between them, there is a small bush.”

Some generations that explicitly ask for things that are not trivial with the available tools (namely, the code available in the function library) can show some problems. For example, in image 5.16, with the overhead of forming a triangle, because there isn’t any direct way of creating one using the tools in the library, it fails to correctly orient the benches, even in simple scenarios.

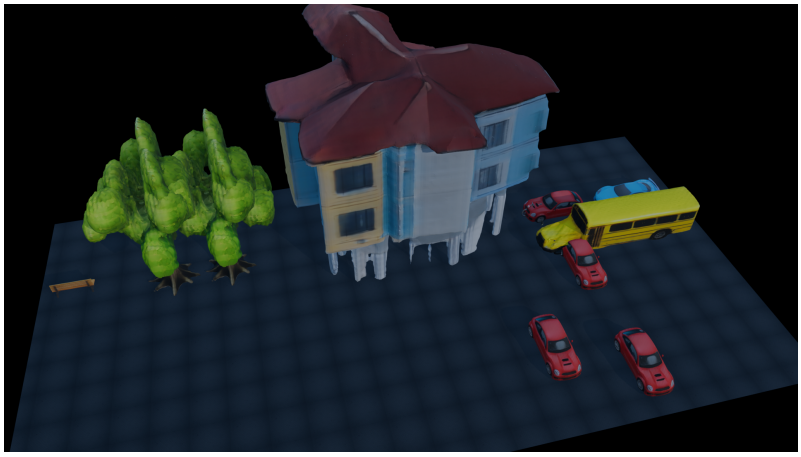


Figure 5.17: ”A residential building, with some vehicles parked nearby. More precisely, the vehicles are all parked on the left side of the building, all side by side and aligned with the -x direction. The vehicles are in order: a car, a sports car, a bike, a bus, and three more cars. On the opposite side of where the vehicles are parked, on the side of the building, two trees are placed with a park bench in between.”



Figure 5.18: ”A residential building at the center of the scene oriented to the -x direction. On the left of the residential building is a store building, and on the right is a small house (all oriented in the same direction). In front of these buildings is a small park where 4 trees, 3 bushes, and two street lamps. In the center of this park are two park benches near each other, one oriented to the + y axis and the other to the -y axis. Further to the front of the buildings (very far) is a road oriented to the y direction. On top of the road is a car that, on its side, has a crosswalk sign and is waiting for a cat to cross the road.”

5.2.2.3 Generation variability

When making 3D scenes, even when using the same description, the method developed can create different outputs. This is a somewhat predictable behavior as we use a generative model to make most of the operations, the LLM. But while it is an expected behavior, it is important to know this phenomenon, as it can have consequences. Namely, in the numeric evaluation, we did a set of 40 scenes, a relatively modest sample size. Consequently, it is possible that some of the measured performance differences between

pipeline configurations are not statistically significant, but are instead artifacts of this inherent generation variance. A more extensive study with a larger sample size would be required to reach more definitive conclusions.

We will explicitly demonstrate this phenomenon by showing three different generations from the same input text.

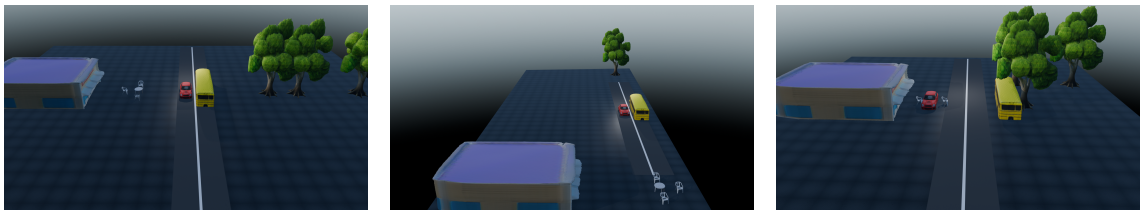


Figure 5.19: Output variance for a single prompt. All three scenes were generated from the identical input prompt: ”Make a scene of a central road oriented towards the x-axis. On top of the road should be a car in the left lane of the road oriented towards the x-axis and a bus on the right lane of the road oriented towards the -x axis. On the left of the road, a table should be in the middle distance. This table should have 3 chairs around it, each chair on a different side of the table and oriented towards the table. A store building should also be in place, so it is in the +y Direction in relation to the table and oriented towards the table. Finally, on the right side of the road, a row of trees should be present, containing 3 trees. ”

In the examples in figure 5.19, we can see clear differences between the generations with the same exact input text, namely in how the objects are positioned, illustrating the variance between distinct generations.

5.2.2.4 Sensitivity to input variations

Lastly, we will make a short study on the pipeline’s sensitivity to specific variations in the input prompt, to get a better overview of the robustness of the method.

It is noteworthy that, as we saw in 5.2.2.3, there can be some degree of variability in generations. This means that sometimes the variations visualized by using a similar prompt can be attributed to this variation, instead of the actual differences in the prompt. Still, we think that this study can provide useful insights.

To accomplish this study, we will use a single reference prompt. In the figure 5.20, there is the original input description and the respective 3D scene generated.

The analysis will examine the pipeline’s response to the following specific modifications of the reference prompt:

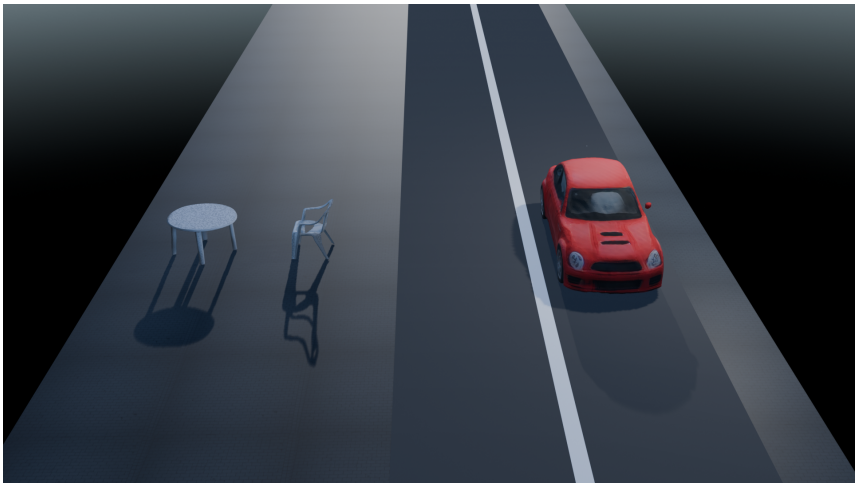


Figure 5.20: Text input: ”A central road is oriented along the +y-axis. A red car is on top of the road, in its right lane, pointing in the same direction as the road. To the left of the road and at some distance, there is a table with a single chair nearby, facing the table.”

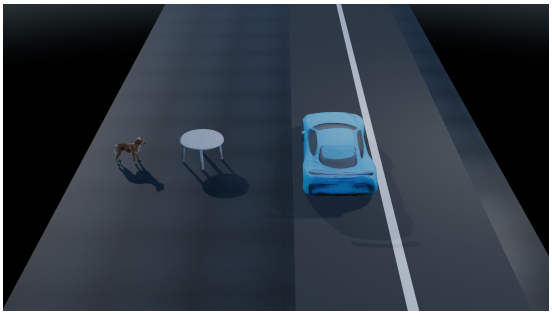
- **Object substitution:** Replacing one or more objects with different ones from the asset library.
- **Object quantity Modification:** Altering the number of a specified object.
- **Clause reordering:** Changing the order of clauses or phrases within the prompt.
- **Synonym replacement:** Substituting keywords with their synonyms or equivalent expressions.

The first of these variations, object substitution, is analyzed below.

In figure 5.21 we can see that it handled the different objects well. However, an important secondary effect is also observable: there are some unrelated changes, like the car is not in the same position as the initial scene. So even introducing small changes in the input description can lead to unexpected and even unrelated changes in the final scene.

The next analysis examines the pipeline’s response to modifications in object quantity. Figure 5.22 depicts two such variations: one where the number of chairs was increased, and another where an additional car was introduced.

In both cases, the pipeline successfully adapted to the new object counts, generating scenes that are consistent with the modified descriptions. This suggests a robust capability to handle changes in object quantity.



(a) ”A central road is oriented along the +y-axis. A blue sports car is on top of the road, in its right lane, pointing in the same direction as the road. To the left of the road and at some distance, there is a table with a dog nearby, facing the table.”



(b) ”A central road is oriented along the +y-axis. A red car is on top of the road, in its right lane, pointing in the same direction as the road. To the left of the road and at some distance, there is a tree with a single chair nearby, facing the table.”

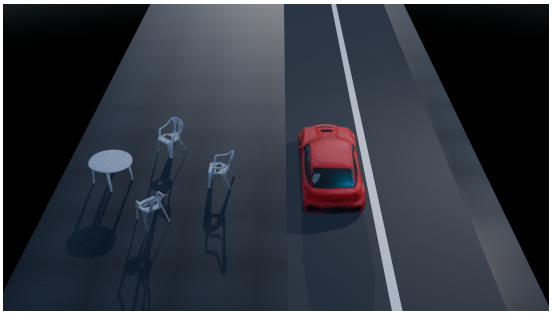
Figure 5.21: Two scenes generated by changing the initial prompt, so it introduces changes in the objects used.

The pipeline’s sensitivity to the ordering of clauses in the prompt is examined in Figure 5.23. The results indicate that the model can be dependent on the sequential structure of the input text.

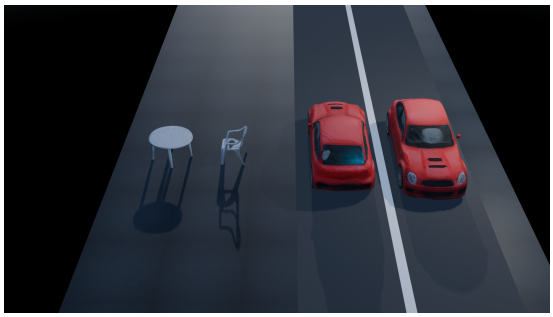
In the first example, there are some changes. Namely, the position of the table and the position of the car relative to the scene generated with the original prompt. But the scene remained coherent and aligned with the input text.

However, in the second image, we can see more notable and detrimental differences. Most notably, the car is no longer placed on top of the road, which goes against the input text. There are some other notable, but not detrimental changes: modification of the floor texture to grass and a different chair position.

This comparison provides evidence that our method is dependent on the order of exposition of the distinct elements in the text input. And while some reorderings are handled



(a) A central road is oriented along the +y-axis. A red car is on top of the road, in its right lane, pointing in the same direction as the road. To the left of the road and at some distance, there is a table with three chairs nearby, facing the table.



(b) A central road is oriented along the +y-axis. On top of the road, there are two red cars. The first car is in the right lane, pointing in the same direction as the road. The second car is in the left lane, pointing in the opposite direction. To the left of the road and at some distance, there is a table with a single chair nearby, facing the table.

Figure 5.22: Two scenes generated by changing the initial prompt, so it introduces changes to the number of objects.

correctly, others can lead to a significant degradation in output quality, including the violation of fundamental layout details present on the input text.

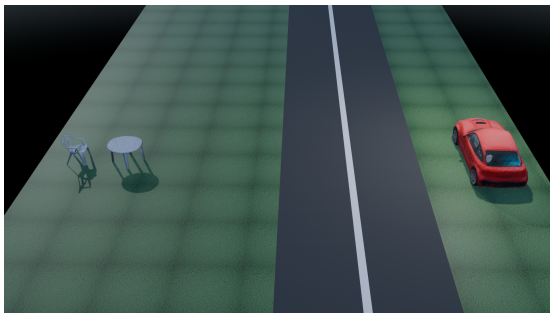
The pipeline's sensitivity to the words used was tested by replacing keywords in the prompt with synonyms or equivalent terms. The results, shown in Figure 5.24, reveal that the system is highly sensitive to such changes, impacting the final output drastically.

This sensitivity manifested in two ways. First, it led to the wrong positioning of the car, by being out of the road. It also led to the use of a different asset for the chair. While this latter change could be considered a valid interpretation, the combination of these effects is notable.

This demonstrates that the pipeline's behavior and performance are dependent on the



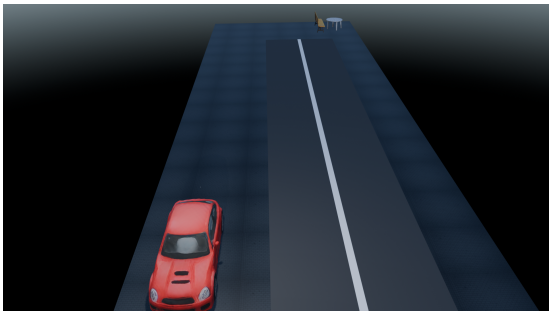
(a) There is a table with a single chair nearby, facing the table. This is to the left of and at some distance from a central road that is oriented along the +y-axis. On top of the road, in its right lane, is a red car pointing in the same direction as the road.



(b) A red car is pointing along the +y-axis. It is on top of a central road, in its right lane. The road is also oriented along the +y-axis. To the left of the road and at some distance, there is a table with a single chair nearby, facing the table.

Figure 5.23: Two scenes generated by changing the initial prompt, so it introduces changes to the order of the exposition.

words and terms used. So when executing the input text, some care is necessary, or it can lead to unexpected or even wrong outputs.



(a) A main street is aligned with the +y-axis. A red vehicle is in the right-hand lane, aimed in the same direction as the street. To the left side of the street and set back a bit, there is a table with one seat next to it, turned towards the table.



(b) A main roadway is positioned along the +y-axis. A red automobile is on the roadway, in its right lane, going in the same direction as the roadway. To the left of the roadway and with some space, there is a table with a single seat nearby, oriented to the table.

Figure 5.24: Two scenes generated by changing the initial prompt, so it introduces changes to the words used.

6. Conclusion

In this work we developed a methodology that is capable of building 3D scenes from a textual input. We adapted some of the ideas found in methods available in the literature and incorporated novel design choices to build our own approach. We also evaluated it to test its robustness and identify possible ways to improve the work. This concluding chapter summarizes the research, enumerates the main lessons learned on the use of LLMs to generate 3D scenes, and provides an overview of the method's capabilities. We also give an overview of limitations faced during the development of our method. After that, we suggest directions for future work that can enhance our methods' capabilities, and some procedures that could give better insights into the method's performance.

6.1. Research Summary

The objective in this dissertation project was to build a method that leverages LLMs to build 3D scenes using a natural language input. We successfully created the methodology to generate 3D scenes with LLMs. Inspired on the available literature and counting on our own contributions, our proposed method is a modular pipeline composed of individual steps that achieve specific goals. These steps leverage not only the LLMs, but also other tools that greatly help execute the task.

We tested our method to produce outdoor scenes, namely in road-related scenarios, as we are particularly interested in that domain, given the motivational context of this thesis. We found that our general-purpose method is capable of generating 3D scenes with some degree of quality and fidelity to the input text.

Although demonstrating relevant capabilities, the method still exhibits some shortcomings and struggles to keep a high level performance in many instances due to either a lack of ability or a lack of consistency. Efforts were made to increase its performance in this regard, but there is still room for improvement.

6.1.1 Key Findings and Remarks

- **Using out-of-the-box LLMs for 3D scene generation is viable, but heavily relies on external tools for efficiency and reliability.** 3D Scene generation leveraging LLMs is viable, but is very dependent on external tools to support the LLM. Not only to get a better performance, but to get a more efficient and reliable method.

- **LLMs are very versatile, excel at high-level scene planning, and can leverage external tools.** The LLMs are relevant in 3D scene generation via text input, as they can deal with text, and are great for general planning of the scene. While lacking the capability to consistently build a scene on their own, they can leverage external tools to accomplish tasks, namely by code generation. And are capable of a very elastic behavior, adapting to different scenarios.

- **While eliminating the need for specific training datasets, LLMs introduce significant computational or financial costs.** The use of LLMs dismisses the need for a dataset to train a model to achieve 3D scene construction, as using off the shelf models is enough, even when adjusting to a specific domain. But this comes at a cost. LLMs are resource-intensive models, so there is a need for heavy computational resources (or monetary ones if using a model via API).

- **Our method generates quality scenes aligned with the text input, but suffers from occasional compositional errors and prompt sensitivity.** Our method seems capable of generating 3D scenes with some level of quality, while aligning with the text input. However, it is not perfect: Sometimes it makes compositional problems. It is sensitive to how the prompt is constructed, namely, to the words and expressions used. It also shows a notable dependence on how good and robust the tools made available to the LLM are to construct the scenes.

- **The ablation study validates that key pipeline components positively impact scene quality and consistency.** The ablation study shows that the core process developed in the pipeline has a positive impact on the generation of the scenes. Namely, the function library, the qualitative formulation, and the feedback loop, all increase the overall performance of the method to build better quality scenes more consistently.

- **The method exhibits high generation variability.** Our method shows a high degree of variability in the generations. This is not necessarily bad, but it can lead to a lot of unpredictability and inconsistency.

- **The method demonstrates reasonable scalability.** The method shows some degree of scalability. It can generate scenes in a reasonable amount of time without a heavy loss in quality for an increasing object count in the input description.

6.2. Limitations

There were two main limitations when performing this thesis:

- **Resource limitation**. LLMs need a lot of computational resources to run. We had a cluster of GPUS, but even in that cluster, we weren't able to run a reasonable LLM to use in the pipeline locally. Consequently, we resorted to the Gemini API. This led to its share of problems, as we were heavily dependent on the availability of the model in the API, and limited rates to make the necessary experimentation, as we were using the free plan.
- **Time constraints with a very open-ended method**. Developing an open-ended method gives a large variety of different options, even for the more specific task. With limited time, it was not possible to explore everything that we would like to. Also, in perfect conditions, the evaluation should address every single step of the pipeline, but that was unrealistic with our time, so we chose to evaluate the most crucial steps.

6.3. Future work

Many things could be done to improve the methods and it's assessment. There are some proposals for future work:

- **Make a more consistent evaluation**. This includes not only complementing the ablation study (for example, testing pipelines just using the constraint evaluation or only the visual evaluation), but also increasing the number of scenes used, and their scope (this would need more assets). Also, be able to test the method against the only available similar method, the BlenderGPt, or even try to do a fair assessment against domain-specific methods. It would also be interesting to explore other evaluation methods that could complement the ones used.
- **Use better quality assets**. This seems significant in two things: to the visual assessment using VLLMs, and when using the image-to-text alignment models (in evaluation and in the asset selection), as both these kinds of models are sensitive to the asset quality.
- **More rigorously texting and definition of the prompts**. We defined the prompts using a combination of techniques documented in the literature and some principles.

It would be good to do more experimentation, or even more rigorous testing, to find more optimal prompting and optimal sequences of prompts to achieve a goal.

- **Expand the asset availability**. We used a very limited set of assets. It would be beneficial to incorporate an asset construction method, and use of bigger set of already existing assets, like Objaverse [82].
- **Enable the use of more general descriptions**. Our method focuses on free text descriptions that mention most of the objects in the scene. It would be useful to also include less informative and broader descriptions.
- **Explore a more reliable scalability solution**. Our scalability solution is naive, enabling only the arrangement of independent scenes in the XY plane. A solution that enables the generation of highly complex scenes with less fragmentation of the output scene would be beneficial.
- **Expand the function library with new functions**. Not only with more offline-made functions, but also an automatic library construction that can generate tailored-made functionality on the fly for a given scene request.
- **Further develop the scene edition pipeline**. The proposed scene edition pipeline could be enhanced, starting by adapting and incorporating the tools already developed for the scene construction pipeline, as it can be a great extension of the construction method.
- **Further explore the constraint optimization methods**. Construct a constraint pipeline that leverages gradient-based methods, and try approaches that do not consider the problem at once. This constraint pipeline should further optimize the positions and orientations of the objects on the scene, to potentially improve the constructed layout and correct any mistakes (or even explore the exclusive use of a constraint framework to construct the layout).
- **Use external tools to construct more complex environments for the scenes**. Namely, the use of pre-made scenarios or leveraging tools like Infinigen [76] for complex background and weather in the scenes.
- **Incorporate the time component**. After being able to construct 3D scenes, the next step would be to also consider the time component. Enabling the creation of live environments where, for example, the autonomous driving models could be tested directly.

6.4. Final Considerations

In conclusion, this thesis demonstrates that the combination of large-scale language models and external tools offers a powerful and flexible paradigm for text-to- 3D scene generation. While challenges in consistency and output quality remain, this work shows that the path towards truly intuitive, language-driven, high-quality 3D scene creation is viable. Our research contributes with a robust, modular pipeline for constructing 3D scenes from text, a methodology for assessing performance using both off-the-shelf and custom metrics, and a thorough analysis of the key components required to advance this field.

References

[1] A. Mcmahan, “Immersion, engagement, and presence: A method for analyzing 3-d video games,” *The Video Game Theory Reader*, pp. 67–86, 01 2003. [Cited on page 1.]

[2] D. Bouchlaghem, H. Shang, J. Whyte, and A. Ganah, “Visualisation in architecture, engineering and construction (aec),” *Automation in Construction*, vol. 14, no. 3, pp. 287–295, 2005, international Conference for Construction Information Technology 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0926580504001025 [Cited on page 1.]

[3] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “Carla: An open urban driving simulator,” 2017. [Online]. Available: https://arxiv.org/abs/1711.03938 [Cited on pages 1 and 7.]

[4] H. Yi, C.-H. P. Huang, S. Tripathi, L. Hering, J. Thies, and M. J. Black, “Mime: Human-aware 3d scene generation,” 2022. [Online]. Available: https://arxiv.org/abs/2212.04360 [Cited on page 1.]

[5] S. Dokania, A. H. A. Hafez, A. Subramanian, M. Chandraker, and C. V. Jawahar, “Idd-3d: Indian driving dataset for 3d unstructured road scenes,” in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, January 2023, pp. 4482–4491. [Cited on page 1.]

[6] B.-K. Ruan, H.-T. Tsui, Y.-H. Li, and H.-H. Shuai, “Traffic scene generation from natural language description for autonomous vehicles with large language model,” 2024. [Online]. Available: https://arxiv.org/abs/2409.09575 [Cited on page 1.]

[7] J. Xu, X. Liu, Y. Wu, Y. Tong, Q. Li, M. Ding, J. Tang, and Y. Dong, “Imagereward: Learning and evaluating human preferences for text-to-image generation,” 2023. [Online]. Available: https://arxiv.org/abs/2304.05977 [Cited on pages 4, 36, and 78.]

[8] Z. Lin, D. Pathak, B. Li, J. Li, X. Xia, G. Neubig, P. Zhang, and D. Ramanan, “Evaluating text-to-visual generation with image-to-text generation,” *arXiv preprint arXiv:2404.01291*, 2024. [Cited on pages 4, 36, and 78.]

[9] M. Tschannen, A. Gritsenko, X. Wang, M. F. Naeem, I. Alabdulmohsin, N. Parthasarathy, T. Evans, L. Beyer, Y. Xia, B. Mustafa, O. Hénaff, J. Harmsen, A. Steiner, and X. Zhai, "Siglip 2: Multilingual vision-language encoders with improved semantic understanding, localization, and dense features," 2025. [Online]. Available: https://arxiv.org/abs/2502.14786 [Cited on pages 4, 36, and 78.]

[10] L. Soni, A. Kaur, and A. Sharma, "A review on different versions and interfaces of blender software," in *2023 7th International Conference on Trends in Electronics and Informatics (ICOEI)*, 2023, pp. 882–887. [Cited on pages 6 and 46.]

[11] A. Hussain, H. Shakeel, F. Hussain, N. Uddin, and T. Ghouri, "Unity game development engine: A technical survey," *University of Sindh Journal of Information and Communication Technology*, vol. 4, 10 2020. [Cited on page 7.]

[12] V. Agrahari and S. Chimalakonda, "What's inside unreal engine? - a curious gaze!" in *Proceedings of the 14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ser. ISEC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3452383.3452404 [Cited on page 7.]

[13] X. Li, Q. Zhang, D. Kang, W. Cheng, Y. Gao, J. Zhang, Z. Liang, J. Liao, Y.-P. Cao, and Y. Shan, "Advances in 3d generation: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2401.17807 [Cited on pages 8 and 9.]

[14] Z. Wang, "3d representation methods: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2410.06475 [Cited on pages 8 and 9.]

[15] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," 2020. [Online]. Available: https://arxiv.org/abs/2003.08934 [Cited on page 9.]

[16] P. Wang, L. Liu, Y. Liu, C. Theobalt, T. Komura, and W. Wang, "Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction," 2023. [Online]. Available: https://arxiv.org/abs/2106.10689 [Cited on page 9.]

[17] G. Zhu, L. Zhang, Y. Jiang, Y. Dang, H. Hou, P. Shen, M. Feng, X. Zhao, Q. Miao, S. A. A. Shah, and M. Bennamoun, "Scene graph generation: A comprehensive survey," 2022. [Online]. Available: https://arxiv.org/abs/2201.00443 [Cited on page 10.]

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023. [Online]. Available: https://arxiv.org/abs/1706.03762 [Cited on pages vii, 11, 12, and 13.]

[19] Z. Wang, Z. Chu, T. V. Doan, S. Ni, M. Yang, and W. Zhang, “History, development, and principles of large language models-an introductory survey,” 2024. [Online]. Available: https://arxiv.org/abs/2402.06853 [Cited on pages vii and 12.]

[20] R. Rosenfeld, “Two decades of statistical language modeling: where do we go from here?” *Proceedings of the IEEE*, vol. 88, no. 8, pp. 1270–1278, 2000. [Cited on page 12.]

[21] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. [Cited on page 12.]

[22] Y. Bengio, R. Ducharme, and P. Vincent, “A neural probabilistic language model,” in *Advances in Neural Information Processing Systems*, T. Leen, T. Dietterich, and V. Tresp, Eds., vol. 13. MIT Press, 2000. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2000/file/728f206c2a01bf572b5940d7d9a8fa4c-Paper.pdf [Cited on page 12.]

[23] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” 2013. [Online]. Available: https://arxiv.org/abs/1310.4546 [Cited on page 12.]

[24] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” 2016. [Online]. Available: https://arxiv.org/abs/1409.0473 [Cited on page 13.]

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:160025533 [Cited on page 13.]

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019. [Online]. Available: https://arxiv.org/abs/1810.04805 [Cited on page 13.]

[27] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter,

C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020. [Online]. Available: https://arxiv.org/abs/2005.14165 [Cited on pages 13, 25, 40, and 70.]

[28] T. Zhong, Z. Liu, Y. Pan, Y. Zhang, Y. Zhou, S. Liang, Z. Wu, Y. Lyu, P. Shu, X. Yu, C. Cao, H. Jiang, H. Chen, Y. Li, J. Chen, H. Hu, Y. Liu, H. Zhao, S. Xu, H. Dai, L. Zhao, R. Zhang, W. Zhao, Z. Yang, J. Chen, P. Wang, W. Ruan, H. Wang, H. Zhao, J. Zhang, Y. Ren, S. Qin, T. Chen, J. Li, A. H. Zidan, A. Jahin, M. Chen, S. Xia, J. Holmes, Y. Zhuang, J. Wang, B. Xu, W. Xia, J. Yu, K. Tang, Y. Yang, B. Sun, T. Yang, G. Lu, X. Wang, L. Chai, H. Li, J. Lu, L. Sun, X. Zhang, B. Ge, X. Hu, L. Zhang, H. Zhou, L. Zhang, S. Zhang, N. Liu, B. Jiang, L. Kong, Z. Xiang, Y. Ren, J. Liu, X. Jiang, Y. Bao, W. Zhang, X. Li, G. Li, W. Liu, D. Shen, A. Sikora, X. Zhai, D. Zhu, and T. Liu, “Evaluation of openai o1: Opportunities and challenges of agi,” 2024. [Online]. Available: https://arxiv.org/abs/2409.18486 [Cited on page 13.]

[29] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican, D. Silver, M. Johnson, I. Antonoglou, J. Schrittwieser, A. Glaese, J. Chen, E. Pitler, T. Lillicrap, A. Lazaridou, O. Firat, J. Molloy, M. Isard, P. R. Barham, T. Hennigan, B. Lee, F. Viola, M. Reynolds, Y. Xu, R. Doherty, E. Collins, C. Meyer, E. Rutherford, E. Moreira, K. Ayoub, M. Goel, J. Krawczyk, C. Du, E. Chi, H.-T. Cheng, E. Ni, P. Shah, P. Kane, B. Chan, M. Faruqui, A. Severyn, H. Lin, Y. Li, Y. Cheng, A. Ittycheriah, M. Mahdieh, M. Chen, P. Sun, D. Tran, S. Bagri, B. Lakshminarayanan, J. Liu, A. Orban, F. Güra, H. Zhou, X. Song, A. Boffy, H. Ganapathy, S. Zheng, H. Choe, Ágoston Weisz, T. Zhu, Y. Lu, S. Gopal, J. Kahn, M. Kula, J. Pitman, R. Shah, E. Taropa, M. A. Merey, M. Baeuml, Z. Chen, L. E. Shafey, Y. Zhang, O. Sercinoglu, G. Tucker, E. Piqueras, M. Krikun, I. Barr, N. Savinov, I. Danihelka, B. Roelofs, A. White, A. Andreassen, T. von Glehn, L. Yagati, M. Kazemi, L. Gonzalez, M. Khalman, J. Sygnowski, A. Frechette, C. Smith, L. Culp, L. Proleev, Y. Luan, X. Chen, J. Lottes, N. Schucher, F. Lebron, A. Rrustemi, N. Clay, P. Crone, T. Kocisky, J. Zhao, B. Perz, D. Yu, H. Howard, A. Bloniarz, J. W. Rae, H. Lu, L. Sifre, M. Maggioni, F. Alcober, D. Garrette, M. Barnes, S. Thakoor, J. Austin, G. Barth-Maron, W. Wong, R. Joshi, R. Chaabouni, D. Fatiha, A. Ahuja, G. S. Tomar, E. Senter, M. Chadwick, I. Kornakov, N. Attaluri, I. Iturrate,

R. Liu, Y. Li, S. Cogan, J. Chen, C. Jia, C. Gu, Q. Zhang, J. Grimstad, A. J. Hartman, X. Garcia, T. S. Pillai, J. Devlin, M. Laskin, D. de Las Casas, D. Valter, C. Tao, L. Blanco, A. P. Badia, D. Reitter, M. Chen, J. Brennan, C. Rivera, S. Brin, S. Iqbal, G. Surita, J. Labanowski, A. Rao, S. Winkler, E. Parisotto, Y. Gu, K. Olszewska, R. Addanki, A. Miech, A. Louis, D. Teplyashin, G. Brown, E. Catt, J. Balaguer, J. Xiang, P. Wang, Z. Ashwood, A. Briukhov, A. Webson, S. Ganapathy, S. Sanghavi, A. Kannan, M.-W. Chang, A. Stjerngren, J. Djolonga, Y. Sun, A. Bapna, M. Aitchison, P. Pejman, H. Michalewski, T. Yu, C. Wang, J. Love, J. Ahn, D. Bloxwich, K. Han, P. Humphreys, T. Sellam, J. Bradbury, V. Godbole, S. Samangooei, B. Damoc, A. Kaskasoli, S. M. R. Arnold, V. Vasudevan, S. Agrawal, J. Riesa, D. Lepikhin, R. Tanburn, S. Srinivasan, H. Lim, S. Hodkinson, P. Shyam, J. Ferret, S. Hand, A. Garg, T. L. Paine, J. Li, Y. Li, M. Giang, A. Neitz, Z. Abbas, S. York, M. Reid, E. Cole, A. Chowdhery, D. Das, D. Rogozińska, V. Nikolaev, P. Sprechmann, Z. Nado, L. Zilka, F. Prost, L. He, M. Monteiro, G. Mishra, C. Welty, J. Newlan, D. Jia, M. Allamanis, C. H. Hu, R. de Liedekerke, J. Gilmer, C. Saroufim, S. Rijhwani, S. Hou, D. Shrivastava, A. Baddepudi, A. Goldin, A. Ozturel, A. Cassirer, Y. Xu, D. Sohn, D. Sachan, R. K. Amplayo, C. Swanson, D. Petrova, S. Narayan, A. Guez, S. Brahma, J. Landon, M. Patel, R. Zhao, K. Villela, L. Wang, W. Jia, M. Rahtz, M. Giménez, L. Yeung, J. Keeling, P. Georgiev, D. Mincu, B. Wu, S. Haykal, R. Saputro, K. Vodrahalli, J. Qin, Z. Cankara, A. Sharma, N. Fernando, W. Hawkins, B. Neyshabur, S. Kim, A. Hutter, P. Agrawal, A. Castro-Ros, G. van den Driessche, T. Wang, F. Yang, S. yiin Chang, P. Komarek, R. McIlroy, M. Lučić, G. Zhang, W. Farhan, M. Sharman, P. Natsev, P. Michel, Y. Bansal, S. Qiao, K. Cao, S. Shakeri, C. Butterfield, J. Chung, P. K. Rubenstein, S. Agrawal, A. Mensch, K. Soparkar, K. Lenc, T. Chung, A. Pope, L. Maggiore, J. Kay, P. Jhakra, S. Wang, J. Maynez, M. Phuong, T. Tobin, A. Tacchetti, M. Trebacz, K. Robinson, Y. Katariya, S. Riedel, P. Bailey, K. Xiao, N. Ghelani, L. Aroyo, A. Slone, N. Houlsby, X. Xiong, Z. Yang, E. Gribovskaya, J. Adler, M. Wirth, L. Lee, M. Li, T. Kagohara, J. Pavagadhi, S. Bridgers, A. Bortsova, S. Ghemawat, Z. Ahmed, T. Liu, R. Powell, V. Bolina, M. Iinuma, P. Zablotskaia, J. Besley, D.-W. Chung, T. Dozat, R. Comanescu, X. Si, J. Greer, G. Su, M. Polacek, R. L. Kaufman, S. Tokumine, H. Hu, E. Buchatskaya, Y. Miao, M. Elhawaty, A. Siddhant, N. Tomasev, J. Xing, C. Greer, H. Miller, S. Ashraf, A. Roy, Z. Zhang, A. Ma, A. Filos, M. Besta, R. Blevins, T. Klimenko,

C.-K. Yeh, S. Changpinyo, J. Mu, O. Chang, M. Pajarskas, C. Muir, V. Cohen, C. L. Lan, K. Haridasan, A. Marathe, S. Hansen, S. Douglas, R. Samuel, M. Wang, S. Austin, C. Lan, J. Jiang, J. Chiu, J. A. Lorenzo, L. L. Sjösund, S. Cevey, Z. Gleicher, T. Avrahami, A. Boral, H. Srinivasan, V. Selo, R. May, K. Aisopos, L. Hussenot, L. B. Soares, K. Baumli, M. B. Chang, A. Recasens, B. Caine, A. Pritzel, F. Pavetic, F. Pardo, A. Gergely, J. Frye, V. Ramasesh, D. Horgan, K. Badola, N. Kassner, S. Roy, E. Dyer, V. C. Campos, A. Tomala, Y. Tang, D. E. Badawy, E. White, B. Mustafa, O. Lang, A. Jindal, S. Vikram, Z. Gong, S. Caelles, R. Hemsley, G. Thornton, F. Feng, W. Stokowiec, C. Zheng, P. Thacker, Çağlar Ünlü, Z. Zhang, M. Saleh, J. Svensson, M. Bileschi, P. Patil, A. Anand, R. Ring, K. Tsihlas, A. Vezer, M. Selvi, T. Shevlane, M. Rodriguez, T. Kwiatkowski, S. Daruki, K. Rong, A. Dafoe, N. FitzGerald, K. Gu-Lemberg, M. Khan, L. A. Hendricks, M. Pellat, V. Feinberg, J. Cobon-Kerr, T. Sainath, M. Rauh, S. H. Hashemi, R. Ives, Y. Hasson, E. Noland, Y. Cao, N. Byrd, L. Hou, Q. Wang, T. Sottiaux, M. Paganini, J.-B. Lespiau, A. Moufarek, S. Hassan, K. Shivakumar, J. van Amersfoort, A. Mandhane, P. Joshi, A. Goyal, M. Tung, A. Brock, H. Sheahan, V. Misra, C. Li, N. Rakićević, M. Dehghani, F. Liu, S. Mittal, J. Oh, S. Noury, E. Sezener, F. Huot, M. Lamm, N. D. Cao, C. Chen, S. Mudgal, R. Stella, K. Brooks, G. Vasudevan, C. Liu, M. Chain, N. Melinkeri, A. Cohen, V. Wang, K. Seymore, S. Zubkov, R. Goel, S. Yue, S. Krishnakumaran, B. Albert, N. Hurley, M. Sano, A. Mohananey, J. Joughin, E. Filonov, T. Kępa, Y. Eldawy, J. Lim, R. Rishi, S. Badiezadegan, T. Bos, J. Chang, S. Jain, S. G. S. Padmanabhan, S. Puttagunta, K. Krishna, L. Baker, N. Kalb, V. Bedapudi, A. Kurzrok, S. Lei, A. Yu, O. Litvin, X. Zhou, Z. Wu, S. Sobell, A. Siciliano, A. Papir, R. Neale, J. Bragagnolo, T. Toor, T. Chen, V. Anklin, F. Wang, R. Feng, M. Gholami, K. Ling, L. Liu, J. Walter, H. Moghaddam, A. Kishore, J. Adamek, T. Mercado, J. Mallinson, S. Wandekar, S. Cagle, E. Ofek, G. Garrido, C. Lombriser, M. Mukha, B. Sun, H. R. Mohammad, J. Matak, Y. Qian, V. Peswani, P. Janus, Q. Yuan, L. Schelin, O. David, A. Garg, Y. He, O. Duzhyi, A. Älgmyr, T. Lottaz, Q. Li, V. Yadav, L. Xu, A. Chinien, R. Shivanna, A. Chuklin, J. Li, C. Spadine, T. Wolfe, K. Mohamed, S. Das, Z. Dai, K. He, D. von Dincklage, S. Upadhyay, A. Maurya, L. Chi, S. Krause, K. Salama, P. G. Rabinovitch, P. K. R. M, A. Selvan, M. Dektiarev, G. Ghiasi, E. Guven, H. Gupta, B. Liu, D. Sharma, I. H. Shtacher, S. Paul, O. Akerlund, F.-X. Aubet, T. Huang, C. Zhu, E. Zhu, E. Teixeira, M. Fritze, F. Bertolini, L.-E.

Marinescu, M. Bölle, D. Paulus, K. Gupta, T. Latkar, M. Chang, J. Sanders, R. Wilson, X. Wu, Y.-X. Tan, L. N. Thiet, T. Doshi, S. Lall, S. Mishra, W. Chen, T. Luong, S. Benjamin, J. Lee, E. Andrejczuk, D. Rabiej, V. Ranjan, K. Styrc, P. Yin, J. Simon, M. R. Harriott, M. Bansal, A. Robsky, G. Bacon, D. Greene, D. Mirylenka, C. Zhou, O. Sarvana, A. Goyal, S. Andermatt, P. Siegler, B. Horn, A. Israel, F. Pongetti, C.-W. L. Chen, M. Selvatici, P. Silva, K. Wang, J. Tolins, K. Guu, R. Yogev, X. Cai, A. Agostini, M. Shah, H. Nguyen, N. □Donnaile, S. Pereira, L. Friso, A. Stambler, A. Kurzrok, C. Kuang, Y. Romanikhin, M. Geller, Z. Yan, K. Jang, C.-C. Lee, W. Fica, E. Malmi, Q. Tan, D. Banica, D. Balle, R. Pham, Y. Huang, D. Avram, H. Shi, J. Singh, C. Hidey, N. Ahuja, P. Saxena, D. Dooley, S. P. Potharaju, E. O'Neill, A. Gokulchandran, R. Foley, K. Zhao, M. Dusenberry, Y. Liu, P. Mehta, R. Kotikalapudi, C. Safranek-Shrader, A. Goodman, J. Kessinger, E. Globen, P. Kolhar, C. Gorgolewski, A. Ibrahim, Y. Song, A. Eichenbaum, T. Brovelli, S. Potluri, P. Lahoti, C. Baetu, A. Ghorbani, C. Chen, A. Crawford, S. Pal, M. Sridhar, P. Gurita, A. Mujika, I. Petrovski, P.-L. Cedoz, C. Li, S. Chen, N. D. Santo, S. Goyal, J. Punjabi, K. Kappaganthu, C. Kwak, P. LV, S. Velury, H. Choudhury, J. Hall, P. Shah, R. Figueira, M. Thomas, M. Lu, T. Zhou, C. Kumar, T. Jurdi, S. Chikkerur, Y. Ma, A. Yu, S. Kwak, V. Ähdel, S. Rajayogam, T. Choma, F. Liu, A. Barua, C. Ji, J. H. Park, V. Hellendoorn, A. Bailey, T. Bilal, H. Zhou, M. Khatir, C. Sutton, W. Rzadkowski, F. Macintosh, K. Shagin, P. Medina, C. Liang, J. Zhou, P. Shah, Y. Bi, A. Dankovics, S. Banga, S. Lehmann, M. Bredesen, Z. Lin, J. E. Hoffmann, J. Lai, R. Chung, K. Yang, N. Balani, A. Bražinskas, A. Sozanschi, M. Hayes, H. F. Alcalde, P. Makarov, W. Chen, A. Stella, L. Snijders, M. Mandl, A. Kärrman, P. Nowak, X. Wu, A. Dyck, K. Vaidyanathan, R. R, J. Mallet, M. Rudominer, E. Johnston, S. Mittal, A. Udathu, J. Christensen, V. Verma, Z. Irving, A. Santucci, G. Elsayed, E. Davoodi, M. Georgiev, I. Tenney, N. Hua, G. Cideron, E. Leurent, M. Alnahlawi, I. Georgescu, N. Wei, I. Zheng, D. Scandinaro, H. Jiang, J. Snoek, M. Sundararajan, X. Wang, Z. Ontiveros, I. Karo, J. Cole, V. Rajashekhar, L. Tumeh, E. Ben-David, R. Jain, J. Uesato, R. Datta, O. Bunyan, S. Wu, J. Zhang, P. Stanczyk, Y. Zhang, D. Steiner, S. Naskar, M. Azzam, M. Johnson, A. Paszke, C.-C. Chiu, J. S. Elias, A. Mohiuddin, F. Muhammad, J. Miao, A. Lee, N. Vieillard, J. Park, J. Zhang, J. Stanway, D. Garmon, A. Karmarkar, Z. Dong, J. Lee, A. Kumar, L. Zhou, J. Evens, W. Isaac, G. Irving, E. Loper, M. Fink,

I. Arkatkar, N. Chen, I. Shafran, I. Petrychenko, Z. Chen, J. Jia, A. Levskaya, Z. Zhu, P. Grabowski, Y. Mao, A. Magni, K. Yao, J. Snaider, N. Casagrande, E. Palmer, P. Suganthan, A. Castaño, I. Giannoumis, W. Kim, M. Rybiński, A. Sreevatsa, J. Prendki, D. Soergel, A. Goedeckemeyer, W. Gierke, M. Jafari, M. Gaba, J. Wiesner, D. G. Wright, Y. Wei, H. Vashisht, Y. Kulizhskaya, J. Hoover, M. Le, L. Li, C. Iwuanyanwu, L. Liu, K. Ramirez, A. Khorlin, A. Cui, T. LIN, M. Wu, R. Aguilar, K. Pallo, A. Chakladar, G. Perng, E. A. Abellan, M. Zhang, I. Dasgupta, N. Kushman, I. Penchev, A. Repina, X. Wu, T. van der Weide, P. Ponnapalli, C. Kaplan, J. Simsa, S. Li, O. Dousse, F. Yang, J. Piper, N. Ie, R. Pasumarthi, N. Lintz, A. Vijayakumar, D. Andor, P. Valenzuela, M. Lui, C. Paduraru, D. Peng, K. Lee, S. Zhang, S. Greene, D. D. Nguyen, P. Kurylowicz, C. Hardin, L. Dixon, L. Janzer, K. Choo, Z. Feng, B. Zhang, A. Singhal, D. Du, D. McKinnon, N. Antropova, T. Bolukbasi, O. Keller, D. Reid, D. Finchelstein, M. A. Raad, R. Crocker, P. Hawkins, R. Dadashi, C. Gaffney, K. Franko, A. Bulanova, R. Leblond, S. Chung, H. Askham, L. C. Cobo, K. Xu, F. Fischer, J. Xu, C. Sorokin, C. Alberti, C.-C. Lin, C. Evans, A. Dimitriev, H. Forbes, D. Banarse, Z. Tung, M. Omernick, C. Bishop, R. Sterneck, R. Jain, J. Xia, E. Amid, F. Piccinno, X. Wang, P. Banzal, D. J. Mankowitz, A. Polozov, V. Krakovna, S. Brown, M. Bateni, D. Duan, V. Firoiu, M. Thotakuri, T. Natan, M. Geist, S. tan Girgin, H. Li, J. Ye, O. Roval, R. Tojo, M. Kwong, J. Lee-Thorp, C. Yew, D. Sinopalnikov, S. Ramos, J. Mellor, A. Sharma, K. Wu, D. Miller, N. Sonnerat, D. Vnukov, R. Greig, J. Beattie, E. Caveness, L. Bai, J. Eisenschlos, A. Korchemniy, T. Tsai, M. Jasarevic, W. Kong, P. Dao, Z. Zheng, F. Liu, F. Yang, R. Zhu, T. H. Teh, J. Sanmiya, E. Gladchenko, N. Trdin, D. Toyama, E. Rosen, S. Tavakkol, L. Xue, C. Elkind, O. Woodman, J. Carpenter, G. Papamakarios, R. Kemp, S. Kafle, T. Grunina, R. Sinha, A. Talbert, D. Wu, D. Owusu-Afriyie, C. Du, C. Thornton, J. Pont-Tuset, P. Narayana, J. Li, S. Fatehi, J. Wieting, O. Ajmeri, B. Uria, Y. Ko, L. Knight, A. Héliou, N. Niu, S. Gu, C. Pang, Y. Li, N. Levine, A. Stolovich, R. Santamaria-Fernandez, S. Goenka, W. Yustalim, R. Strudel, A. Elqursh, C. Deck, H. Lee, Z. Li, K. Levin, R. Hoffmann, D. Holtmann-Rice, O. Bachem, S. Arora, C. Koh, S. H. Yeganeh, S. Põder, M. Tariq, Y. Sun, L. Ionita, M. Seyedhosseini, P. Tafti, Z. Liu, A. Gulati, J. Liu, X. Ye, B. Chrzaszcz, L. Wang, N. Sethi, T. Li, B. Brown, S. Singh, W. Fan, A. Parisi, J. Stanton, V. Koverkathu, C. A. Choquette-Choo, Y. Li, T. Lu, A. Ittycheriah, P. Shroff, M. Varadarajan,

S. Bahargam, R. Willoughby, D. Gaddy, G. Desjardins, M. Cornero, B. Robenek, B. Mittal, B. Albrecht, A. Shenoy, F. Moiseev, H. Jacobsson, A. Ghaffarkhah, M. Rivière, A. Walton, C. Crepy, A. Parrish, Z. Zhou, C. Farabet, C. Radebaugh, P. Srinivasan, C. van der Salm, A. Fidjeland, S. Scellato, E. Latorre-Chimoto, H. Klimczak-Plucińska, D. Bridson, D. de Cesare, T. Hudson, P. Mendolicchio, L. Walker, A. Morris, M. Mauger, A. Guseynov, A. Reid, S. Odoom, L. Loher, V. Cotruta, M. Yenugula, D. Grewe, A. Petrushkina, T. Duerig, A. Sanchez, S. Yadlowsky, A. Shen, A. Globerson, L. Webb, S. Dua, D. Li, S. Bhupatiraju, D. Hurt, H. Qureshi, A. Agarwal, T. Shani, M. Eyal, A. Khare, S. R. Belle, L. Wang, C. Tekur, M. S. Kale, J. Wei, R. Sang, B. Saeta, T. Liechty, Y. Sun, Y. Zhao, S. Lee, P. Nayak, D. Fritz, M. R. Vuyyuru, J. Aslanides, N. Vyas, M. Wicke, X. Ma, E. Eltyshev, N. Martin, H. Cate, J. Manyika, K. Amiri, Y. Kim, X. Xiong, K. Kang, F. Luisier, N. Tripuraneni, D. Madras, M. Guo, A. Waters, O. Wang, J. Ainslie, J. Baldridge, H. Zhang, G. Pruthi, J. Bauer, F. Yang, R. Mansour, J. Gelman, Y. Xu, G. Polovets, J. Liu, H. Cai, W. Chen, X. Sheng, E. Xue, S. Ozair, C. Angermueller, X. Li, A. Sinha, W. Wang, J. Wiesinger, E. Koukoumidis, Y. Tian, A. Iyer, M. Gurumurthy, M. Goldenson, P. Shah, M. Blake, H. Yu, A. Urbanowicz, J. Palomaki, C. Fernando, K. Durden, H. Mehta, N. Momchev, E. Rahimtoroghi, M. Georgaki, A. Raul, S. Ruder, M. Redshaw, J. Lee, D. Zhou, K. Jalan, D. Li, B. Hechtman, P. Schuh, M. Nasr, K. Milan, V. Mikulik, J. Franco, T. Green, N. Nguyen, J. Kelley, A. Mahendru, A. Hu, J. Howland, B. Vargas, J. Hui, K. Bansal, V. Rao, R. Ghiya, E. Wang, K. Ye, J. M. Sarr, M. M. Preston, M. Elish, S. Li, A. Kaku, J. Gupta, I. Pasupat, D.-C. Juan, M. Someswar, T. M., X. Chen, A. Amini, A. Fabrikant, E. Chu, X. Dong, A. Muthal, S. Buthpitiya, S. Jauhari, N. Hua, U. Khandelwal, A. Hitron, J. Ren, L. Rinaldi, S. Drath, A. Dabush, N.-J. Jiang, H. Godhia, U. Sachs, A. Chen, Y. Fan, H. Taitelbaum, H. Noga, Z. Dai, J. Wang, C. Liang, J. Hamer, C.-S. Ferng, C. Elkind, A. Atias, P. Lee, V. Listík, M. Carlen, J. van de Kerkhof, M. Pikus, K. Zaher, P. Müller, S. Zykova, R. Stefanec, V. Gatsko, C. Hirnschall, A. Sethi, X. F. Xu, C. Ahuja, B. Tsai, A. Stefanoiu, B. Feng, K. Dhandhania, M. Katyal, A. Gupta, A. Parulekar, D. Pitta, J. Zhao, V. Bhatia, Y. Bhavnani, O. Alhadlaq, X. Li, P. Danenberg, D. Tu, A. Pine, V. Filippova, A. Ghosh, B. Limonchik, B. Urala, C. K. Lanka, D. Clive, Y. Sun, E. Li, H. Wu, K. Hongtongsak, I. Li, K. Thakkar, K. Omarov, K. Majmundar, M. Alverson, M. Kucharski, M. Patel, M. Jain, M. Zabelin, P. Pelagatti, R. Kohli,

S. Kumar, J. Kim, S. Sankar, V. Shah, L. Ramachandruni, X. Zeng, B. Bariach, L. Weidinger, T. Vu, A. Andreev, A. He, K. Hui, S. Kashem, A. Subramanya, S. Hsiao, D. Hassabis, K. Kavukcuoglu, A. Sadovsky, Q. Le, T. Strohman, Y. Wu, S. Petrov, J. Dean, and O. Vinyals, "Gemini: A family of highly capable multimodal models," 2024. [Online]. Available: https://arxiv.org/abs/2312.11805 [Cited on pages 13 and 75.]

[30] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Prasad, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, K. Lakhotia, L. Rantala-Yeary, L. van der Maaten, L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh, M. Paluri, M. Kardas, M. Tsimpoukelli, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, N. Zhang, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Maheswari, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman,

T. Fowler, T. Sheasha, T. Georgiou, T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez, V. Gonguet, V. Do, V. Vogeti, V. Albiero, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. Wang, X. E. Tan, X. Xia, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Srivastava, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Teo, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Dong, A. Franco, A. Goyal, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Liu, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, C. Gao, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E.-T. Le, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Kokkinos, F. Ozgenel, F. Caggioni, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Inan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, H. Zhan, I. Damlaj, I. Molybog, I. Tufanov, I. Leontiadis, I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Lam, J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Jagadeesh, K. Huang, K. Chawla, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. Liu, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. Mehta, N. P. Laptev, N. Dong,

N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Parthasarathy, R. Li, R. Hogan, R. Battey, R. Wang, R. Howes, R. Rinott, S. Mehta, S. Siby, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Mahajan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Patil, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Deng, S. Cho, S. Virk, S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Koehler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wu, X. Wang, X. Wu, X. Gao, Y. Kleinman, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Zhao, Y. Hao, Y. Qian, Y. Li, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, Z. Zhao, and Z. Ma, “The llama 3 herd of models,” 2024. [Online]. Available: https://arxiv.org/abs/2407.21783 [Cited on page 13.]

[31] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot, and S. Tan, “Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp,” 2021. [Online]. Available: https://arxiv.org/abs/2112.10508 [Cited on page 13.]

[32] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, “A comprehensive overview of large language models,” 2024. [Online]. Available: https://arxiv.org/abs/2307.06435 [Cited on page 14.]

[33] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, “Emergent abilities of large language models,” 2022. [Online]. Available: https://arxiv.org/abs/2206.07682 [Cited on page 14.]

[34] Y. Hong, H. Zhen, P. Chen, S. Zheng, Y. Du, Z. Chen, and C. Gan, “3d-llm: Injecting the 3d world into large language models,” in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 20 482–20 494.

[Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/413885e70482b95dcbeeddc1daf39177-Paper-Conference.pdf [Cited on pages 14 and 23.]

[35] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," 2023. [Online]. Available: https://arxiv.org/abs/2302.11382 [Cited on page 15.]

[36] S. Mahajan, T. Rahman, K. M. Yi, and L. Sigal, "Prompting hard or hardly prompting: Prompt inversion for text-to-image diffusion models," 2023. [Online]. Available: https://arxiv.org/abs/2312.12416 [Cited on page 15.]

[37] S. Lu, H. Schuff, and I. Gurevych, "How are prompts different in terms of sensitivity?" in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for Computational Linguistics, Jun. 2024, pp. 5833–5856. [Online]. Available: https://aclanthology.org/2024.naacl-long.325/ [Cited on page 15.]

[38] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3544548.3581388 [Cited on page 15.]

[39] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, p. 307–392, 2019. [Online]. Available: http://dx.doi.org/10.1561/2200000056 [Cited on page 15.]

[40] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014. [Online]. Available: https://arxiv.org/abs/1406.2661 [Cited on page 15.]

[41] J. Sohl-Dickstein, E. A. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep unsupervised learning using nonequilibrium thermodynamics," 2015. [Online]. Available: https://arxiv.org/abs/1503.03585 [Cited on page 15.]

[42] A. Chattopadhyay, X. Zhang, D. P. Wipf, H. Arora, and R. Vidal, “Learning graph variational autoencoders with constraints and structured priors for conditional indoor 3d scene generation,” in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, January 2023, pp. 785–794. [Cited on page 16.]

[43] Z. Wu, Y. Li, H. Yan, T. Shang, W. Sun, S. Wang, R. Cui, W. Liu, H. Sato, H. Li, and P. Ji, “Blockfusion: Expandable 3d scene generation using latent tri-plane extrapolation,” *ACM Trans. Graph.*, vol. 43, no. 4, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3658188 [Cited on page 16.]

[44] M.-J. Yang, Y.-X. Guo, B. Zhou, and X. Tong, “Indoor scene generation from a collection of semantic-segmented depth images,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2021, pp. 15 203–15 212. [Cited on page 16.]

[45] D. Paschalidou, A. Kar, M. Shugrina, K. Kreis, A. Geiger, and S. Fidler, “Atiss: Autoregressive transformers for indoor scene synthesis,” in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 12 013–12 026. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/64986d86a17424eeac96b08a6d519059-Paper.pdf [Cited on page 16.]

[46] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, “High-resolution image synthesis with latent diffusion models,” 2022. [Online]. Available: https://arxiv.org/abs/2112.10752 [Cited on page 17.]

[47] Y. Liu, K. Zhang, Y. Li, Z. Yan, C. Gao, R. Chen, Z. Yuan, Y. Huang, H. Sun, J. Gao, L. He, and L. Sun, “Sora: A review on background, technology, limitations, and opportunities of large vision models,” 2024. [Online]. Available: https://arxiv.org/abs/2402.17177 [Cited on page 17.]

[48] Z. Kong, W. Ping, J. Huang, K. Zhao, and B. Catanzaro, “Diffwave: A versatile diffusion model for audio synthesis,” 2021. [Online]. Available: https://arxiv.org/abs/2009.09761 [Cited on page 17.]

[49] R. Po and G. Wetzstein, “Compositional 3d scene generation using locally conditioned diffusion,” in *2024 International Conference on 3D Vision (3DV)*, 2024, pp. 651–663. [Cited on page 17.]

[50] G. Gao, W. Liu, A. Chen, A. Geiger, and B. Schölkopf, “Graphdreamer: Compositional 3d scene synthesis from scene graphs,” in *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024, pp. 21 295–21 304. [Cited on page 17.]

[51] J. Tang, Y. Nie, L. Markhasin, A. Dai, J. Thies, and M. Nießner, “Diffuscene: Denoising diffusion models for generative indoor scene synthesis,” in *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024, pp. 20 507–20 518. [Cited on page 17.]

[52] R. Maddegoda and A. S. Karunananda, “Multi agent based approach to assist the design process of 3d game environments,” in *International Conference on Advances in ICT for Emerging Regions (ICTer2012)*, 2012, pp. 36–44. [Cited on page 17.]

[53] S. N. Dessai and R. Dhanaraj, “Creation of 3d scene from raw text,” in *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, 2016, pp. 1466–1469. [Cited on page 17.]

[54] P. Kán and H. Kaufmann, “Automatic furniture arrangement using greedy cost minimization,” in *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, 2018, pp. 491–498. [Cited on page 18.]

[55] M. A. Dema and H. Sari-Sarraf, “3d scene generation by learning from examples,” in *2012 IEEE International Symposium on Multimedia*, 2012, pp. 58–64. [Cited on page 18.]

[56] A. Raistrick, L. Lipson, Z. Ma, L. Mei, M. Wang, Y. Zuo, K. Kayan, H. Wen, B. Han, Y. Wang, A. Newell, H. Law, A. Goyal, K. Yang, and J. Deng, “Infinite photorealistic worlds using procedural generation,” in *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 12 630–12 641. [Cited on page 18.]

[57] Z. Hu, A. Iscen, A. Jain, T. Kipf, Y. Yue, D. A. Ross, C. Schmid, and A. Fathi, “Scenecraft: An llm agent for synthesizing 3d scene as blender code,” 2024.

[Online]. Available: https://arxiv.org/abs/2403.01248 [Cited on pages vii, 19, 23, 24, 29, 36, 39, 46, 48, 51, 53, and 59.]

[58] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," 2021. [Online]. Available: https://arxiv.org/abs/2103.00020 [Cited on pages 20, 36, 51, 53, and 78.]

[59] J. Hessel, A. Holtzman, M. Forbes, R. L. Bras, and Y. Choi, "Clipscore: A reference-free evaluation metric for image captioning," 2022. [Online]. Available: https://arxiv.org/abs/2104.08718 [Cited on pages 20 and 53.]

[60] R. Aguina-Kang, M. Gumin, D. H. Han, S. Morris, S. J. Yoo, A. Ganeshan, R. K. Jones, Q. A. Wei, K. Fu, and D. Ritchie, "Open-universe indoor scene generation using llm program synthesis and uncurated object databases," 2024. [Online]. Available: https://arxiv.org/abs/2403.09675 [Cited on pages vii, 20, 25, 51, and 53.]

[61] M. Zhou, Y. Wang, J. Hou, S. Zhang, Y. Li, C. Luo, J. Peng, and Z. Zhang, "Scenex: Procedural controllable large-scale scene generation," 2024. [Online]. Available: https://arxiv.org/abs/2403.15698 [Cited on pages 22, 37, 51, and 53.]

[62] J. Deng, W. Chai, J. Huang, Z. Zhao, Q. Huang, M. Gao, J. Guo, S. Hao, W. Hu, J.-N. Hwang, X. Li, and G. Wang, "Citycraft: A real crafter for 3d city generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.04983 [Cited on pages 22, 51, and 53.]

[63] X. Xia, D. Zhang, Z. Liao, Z. Hou, T. Sun, J. Li, L. Fu, and Y. Dong, "Scenegenagent: Precise industrial scene generation with coding agent," 2024. [Online]. Available: https://arxiv.org/abs/2410.21909 [Cited on page 22.]

[64] L. R. Kouzelis and O. Spantidi, *Synthesizing Play-Ready VR Scenes with Natural Language Prompts Through GPT API*, 12 2023, pp. 15–26. [Cited on page 22.]

[65] Y. Yang, J. Lu, Z. Zhao, Z. Luo, J. J. Q. Yu, V. Sanchez, and F. Zheng, "Llplace: The 3d indoor scene layout generation and editing via large language model," 2024. [Online]. Available: https://arxiv.org/abs/2406.03866 [Cited on pages vii, 22, 26, 27, and 36.]

[66] B. M. Öcal, M. Tatarchenko, S. Karaoglu, and T. Gevers, "Sceneteller: Language-to-3d scene generation," 2024. [Online]. Available: https://arxiv.org/abs/2407.20727 [Cited on page 23.]

[67] W. Feng, W. Zhu, T.-J. Fu, V. Jampani, A. Akula, X. He, S. Basu, X. E. Wang, and W. Y. Wang, "Layoutgpt: Compositional visual planning and generation with large language models," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 18 225–18 250. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/3a7f9e485845dac27423375c934cb4db-Paper-Conference.pdf [Cited on page 23.]

[68] P. Jiang and X. Cai, "A survey of text-matching techniques," *Information*, vol. 15, no. 6, 2024. [Online]. Available: https://www.mdpi.com/2078-2489/15/6/332 [Cited on page 23.]

[69] R. Guo, J. Wei, L. Sun, B. Yu, G. Chang, D. Liu, S. Zhang, Z. Yao, M. Xu, and L. Bu, "A survey on image-text multimodal models," 2024. [Online]. Available: https://arxiv.org/abs/2309.15857 [Cited on page 23.]

[70] D. Hegde, J. M. J. Valanarasu, and V. M. Patel, "Clip goes 3d: Leveraging prompt tuning for language grounded 3d recognition," 2023. [Online]. Available: https://arxiv.org/abs/2303.11313 [Cited on page 23.]

[71] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi, "Least-to-most prompting enables complex reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2205.10625 [Cited on pages vii, 24, 42, and 43.]

[72] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685 [Cited on page 27.]

[73] H. I. I. Tam, H. I. D. Pun, A. T. Wang, A. X. Chang, and M. Savva, "Scenemotifcoder: Example-driven visual program learning for generating 3d object arrangements," 2024. [Online]. Available: https://arxiv.org/abs/2408.02211 [Cited on pages vii, 27, 28, 29, and 36.]

[74] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models,"

2023. [Online]. Available: https://arxiv.org/abs/2201.11903 [Cited on pages vii, 28, 41, 42, 48, and 57.]

[75] X. Liu, C.-K. Tang, and Y.-W. Tai, "Worldcraft: Photo-realistic 3d world creation and customization via llm agents," 2025. [Online]. Available: https://arxiv.org/abs/2502.15601 [Cited on pages vii, 29, 30, and 48.]

[76] A. Raistrick, L. Mei, K. Kayan, D. Yan, Y. Zuo, B. Han, H. Wen, M. Parakh, S. Alexandropoulos, L. Lipson, Z. Ma, and J. Deng, "Infinigen indoors: Photorealistic indoor scenes using procedural generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.11824 [Cited on pages 30 and 105.]

[77] A. Raistrick, L. Lipson, Z. Ma, L. Mei, M. Wang, Y. Zuo, K. Kayan, H. Wen, B. Han, Y. Wang, A. Newell, H. Law, A. Goyal, K. Yang, and J. Deng, "Infinite photorealistic worlds using procedural generation," 2023. [Online]. Available: https://arxiv.org/abs/2306.09310 [Cited on page 30.]

[78] L.-F. Yu, S.-K. Yeung, C.-K. Tang, D. Terzopoulos, T. F. Chan, and S. J. Osher, "Make it home: automatic optimization of furniture arrangement," *ACM Trans. Graph.*, vol. 30, no. 4, Jul. 2011. [Online]. Available: https://doi.org/10.1145/2010324.1964981 [Cited on page 31.]

[79] L. Ling, C.-H. Lin, T.-Y. Lin, Y. Ding, Y. Zeng, Y. Sheng, Y. Ge, M.-Y. Liu, A. Bera, and Z. Li, "Scenethesis: A language and vision agentic framework for 3d scene generation," 2025. [Online]. Available: https://arxiv.org/abs/2505.02836 [Cited on pages vii, 31, 32, 45, and 59.]

[80] T. Ren, S. Liu, A. Zeng, J. Lin, K. Li, H. Cao, J. Chen, X. Huang, Y. Chen, F. Yan, Z. Zeng, H. Zhang, F. Li, J. Yang, H. Li, Q. Jiang, and L. Zhang, "Grounded sam: Assembling open-world models for diverse visual tasks," 2024. [Online]. Available: https://arxiv.org/abs/2401.14159 [Cited on pages 32 and 33.]

[81] A. Bochkovskii, A. Delaunoy, H. Germain, M. Santos, Y. Zhou, S. R. Richter, and V. Koltun, "Depth pro: Sharp monocular metric depth in less than a second," 2025. [Online]. Available: https://arxiv.org/abs/2410.02073 [Cited on pages 32 and 33.]

[82] M. Deitke, R. Liu, M. Wallingford, H. Ngo, O. Michel, A. Kusupati, A. Fan, C. Laforte, V. Voleti, S. Y. Gadre, E. VanderBilt, A. Kembhavi, C. Vondrick, G. Gkioxari, K. Ehsani, L. Schmidt, and A. Farhadi, "Objaverse-xl: A universe

of 10m+ 3d objects," 2023. [Online]. Available: https://arxiv.org/abs/2307.05663 [Cited on pages 32 and 105.]

[83] J. Edstedt, Q. Sun, G. Bökman, M. Wadenbäck, and M. Felsberg, "Roma: Robust dense feature matching," 2023. [Online]. Available: https://arxiv.org/abs/2305.15404 [Cited on page 33.]

[84] H. Rasheed, M. U. Khattak, M. Maaz, S. Khan, and F. S. Khan, "Fine-tuned clip models are efficient video learners," 2023. [Online]. Available: https://arxiv.org/abs/2212.03640 [Cited on page 36.]

[85] J. Li, D. Li, C. Xiong, and S. Hoi, "Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation," 2022. [Online]. Available: https://arxiv.org/abs/2201.12086 [Cited on page 36.]

[86] T. Wu, G. Yang, Z. Li, K. Zhang, Z. Liu, L. Guibas, D. Lin, and G. Wetzstein, "Gpt-4v(ision) is a human-aligned evaluator for text-to-3d generation," 2024. [Online]. Available: https://arxiv.org/abs/2401.04092 [Cited on page 36.]

[87] Y. Wei, M. R. Min, G. Vosselman, L. E. Li, and M. Y. Yang, "Planner3d: Llm-enhanced graph prior meets 3d indoor scene explicit regularization," 2024. [Online]. Available: https://arxiv.org/abs/2403.12848 [Cited on pages vii, 37, 38, and 39.]

[88] J. Yang, X. Chen, N. Madaan, M. Iyengar, S. Qian, D. F. Fouhey, and J. Chai, "3d-grand: A million-scale dataset for 3d-llms with better grounding and less hallucination," 2024. [Online]. Available: https://arxiv.org/abs/2406.05132 [Cited on page 40.]

[89] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3354–3361. [Cited on page 40.]

[90] R. Krishna, Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, M. S. Bernstein, and F.-F. Li, "Visual genome: Connecting language and vision using crowdsourced dense image annotations," 2016. [Online]. Available: https://arxiv.org/abs/1602.07332 [Cited on page 40.]

[91] J. Johnson, B. Hariharan, L. van der Maaten, L. Fei-Fei, C. L. Zitnick, and R. Girshick, "Clevr: A diagnostic dataset for compositional language and elementary visual reasoning," 2016. [Online]. Available: https://arxiv.org/abs/1612.06890 [Cited on page 40.]

[92] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," 2023. [Online]. Available: https://arxiv.org/abs/2205.11916 [Cited on pages vii, 41, and 70.]

[93] C. Li, J. Wang, Y. Zhang, K. Zhu, W. Hou, J. Lian, F. Luo, Q. Yang, and X. Xie, "Large language models understand and can be enhanced by emotional stimuli," 2023. [Online]. Available: https://arxiv.org/abs/2307.11760 [Cited on pages 41 and 70.]

[94] Z. M. Wang, Z. Peng, H. Que, J. Liu, W. Zhou, Y. Wu, H. Guo, R. Gan, Z. Ni, J. Yang, M. Zhang, Z. Zhang, W. Ouyang, K. Xu, S. W. Huang, J. Fu, and J. Peng, "Rolellm: Benchmarking, eliciting, and enhancing role-playing abilities of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2310.00746 [Cited on pages 41 and 70.]

[95] O. Press, M. Zhang, S. Min, L. Schmidt, N. A. Smith, and M. Lewis, "Measuring and narrowing the compositionality gap in language models," 2023. [Online]. Available: https://arxiv.org/abs/2210.03350 [Cited on page 41.]

[96] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Ding, H. Xin, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Wang, J. Chen, J. Yuan, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, S. Ye, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Zhao, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie,

X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang, “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning,” 2025. [Online]. Available: https://arxiv.org/abs/2501.12948 [Cited on page 42.]

[97] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, “Tree of thoughts: Deliberate problem solving with large language models,” 2023. [Online]. Available: https://arxiv.org/abs/2305.10601 [Cited on page 43.]

[98] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta, H. Han, S. Schulhoff, P. S. Dulepet, S. Vidyadhara, D. Ki, S. Agrawal, C. Pham, G. Kroiz, F. Li, H. Tao, A. Srivastava, H. D. Costa, S. Gupta, M. L. Rogers, I. Goncearenco, G. Sarli, I. Galynker, D. Peskoff, M. Carpuat, J. White, S. Anadkat, A. Hoyle, and P. Resnik, “The prompt report: A systematic survey of prompting techniques,” 2024. [Online]. Available: https://arxiv.org/abs/2406.06608 [Cited on page 43.]

[99] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, “Unleashing the potential of prompt engineering in large language models: a comprehensive review,” 2024. [Online]. Available: https://arxiv.org/abs/2310.14735 [Cited on page 43.]

[100] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, “Lost in the middle: How language models use long contexts,” 2023. [Online]. Available: https://arxiv.org/abs/2307.03172 [Cited on page 48.]

[101] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” 2023. [Online]. Available: https://arxiv.org/abs/2210.03629 [Cited on page 48.]

[102] J. Xu, W. Cheng, Y. Gao, X. Wang, S. Gao, and Y. Shan, “Instantmesh: Efficient 3d mesh generation from a single image with sparse-view large reconstruction

models," 2024. [Online]. Available: https://arxiv.org/abs/2404.07191 [Cited on page 75.]

[103] J. Xiang, Z. Lv, S. Xu, Y. Deng, R. Wang, B. Zhang, D. Chen, X. Tong, and J. Yang, "Structured 3d latents for scalable and versatile 3d generation," 2025. [Online]. Available: https://arxiv.org/abs/2412.01506 [Cited on page 75.]

[104] Z. Zhao, Z. Lai, Q. Lin, Y. Zhao, H. Liu, S. Yang, Y. Feng, M. Yang, S. Zhang, X. Yang, H. Shi, S. Liu, J. Wu, Y. Lian, F. Yang, R. Tang, Z. He, X. Wang, J. Liu, X. Zuo, Z. Chen, B. Lei, H. Weng, J. Xu, Y. Zhu, X. Liu, L. Xu, C. Hu, S. Yang, S. Zhang, Y. Liu, T. Huang, L. Wang, J. Zhang, M. Chen, L. Dong, Y. Jia, Y. Cai, J. Yu, Y. Tang, H. Zhang, Z. Ye, P. He, R. Wu, C. Zhang, Y. Tan, J. Xiao, Y. Tao, J. Zhu, J. Xue, K. Liu, C. Zhao, X. Wu, Z. Hu, L. Qin, J. Peng, Z. Li, M. Chen, X. Zhang, L. Niu, P. Wang, Y. Wang, H. Kuang, Z. Fan, X. Zheng, W. Zhuang, Y. He, T. Liu, Y. Yang, D. Wang, Y. Liu, J. Jiang, J. Huang, and C. Guo, "Hunyuan3d 2.0: Scaling diffusion models for high resolution textured 3d assets generation," 2025. [Online]. Available: https://arxiv.org/abs/2501.12202 [Cited on page 75.]

[105] gd3kr, "Blendergpt," https://github.com/gd3kr/BlenderGPT, 2023. [Cited on page 77.]

[106] M. J. D. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *The Computer Journal*, vol. 7, no. 2, pp. 155–162, 01 1964. [Online]. Available: https://doi.org/10.1093/comjnl/7.2.155 [Cited on page 164.]

[107] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 01 1965. [Online]. Available: https://doi.org/10.1093/comjnl/7.4.308 [Cited on page 164.]

[108] M. J. D. Powell, *A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation*. Dordrecht: Springer Netherlands, 1994, pp. 51–67. [Online]. Available: https://doi.org/10.1007/978-94-015-8330-5_4 [Cited on page 164.]

[109] N. Hansen, "The cma evolution strategy: A tutorial," 2023. [Online]. Available: https://arxiv.org/abs/1604.00772 [Cited on page 164.]

A. Appendix

A.1. Prompts

All the most relevant prompts used in our method that are not explicitly shown in the main analysis are depicted in this Appendix. In note worthy that these prompts use the f-string notation from Python to receive the inputs, so each input is marked as { input name } in the middle of the prompt.

Prompt for Scene Division:

```
"""
You are an expert 3D scene architect assistant. Your task is to analyze user
    requests for 3D scene construction and, when necessary, divide complex scenes
     into smaller, manageable, logically distinct sub-scenes for independent
    construction. You will always output the results in a specific JSON format,
    regardless of whether division occurs.

You will receive a textual description of a 3D scene provided by a user.

Your core task involves these steps:
First, analyze the complexity by carefully reading the input description and
    counting the number of distinct primary objects or assets requested.

Second, decide if division is necessary based on the object count and how the
    scene is clustered.
    - If the object count is {number_obj} or less, the scene does NOT require
    division in any circumstance.
    - If the object count is greater than {number_obj}, then you perform a
    division if it is possible to identify different clusters where it is viable
    to clearly define the relative positions between these clusters. The objects
    within the clusters should be placed and oriented between each other.
Regardless of this decision, you will ALWAYS proceed to the final Output
    Generation step.

Third, perform scene division (ONLY if the decision in the previous step
    determined that division IS required):
Assign every object mentioned in the original prompt to exactly one sub-scene.
```

```
When assigning objects, create sub-scenes that represent distinct spatial areas.
    But try not to define subscenes with a very low number of objects (1-4
    objects). Each subscene also should not have more than {number_obj}.
Keep this ALWAYS in mind: Do not mention any objects or subscenes names that
    belong to another subscene in a given subscene description. If it is
    requested to orient a certain object in a subscene according to another
    subscene you simply have to mention the correct alignment with the cardinal
    axis (-x right, the +x is the left, the -y is the front, and +y is the back).
In a subscene description can only be mentioned the objects assigned to that
    subscene. Nothing more nor less.
The overall goal is to strike a balance: simplify the complex scene into
    manageable parts, but with minimal fragmentation.
Choose one sub-scene to be the anchor or 'center'.

Fourth, generate the output. Prepare the output as a JSON list. This list will
    contain one or more JSON objects, each representing a sub-scene.
    - If scene division WAS performed: The list will contain multiple JSON
    objects, one for each defined sub-scene.
    - If scene division WAS NOT performed: The list will contain EXACTLY ONE JSON
     object representing the entire scene.

For each sub-scene JSON object (whether it's one of many or the single one
    representing the whole scene), include the following keys:
'subscene_name': (String) A concise, unique, and descriptive name (e.g., "
    LivingRoom_MainSeating", "OfficeDeskArea", or simply "Full_Office_Scene" if
    not divided).
'description': (String) A textual description containing ONLY the objects and
    their relationships strictly WITHIN this specific sub-scene.
'objects_list': (List of Strings) An explicit list of the names of the objects
    assigned to this sub-scene (or all objects if not divided).
'relative_positioning': (JSON Object) An object describing the position.
    'relative_to': (String or null) The 'subscene_name' of the sub-scene this one
     is positioned relative to. If this is the designated 'center' sub-scene, OR
    if the scene was NOT divided (i.e., this is the only object in the list),
    this value MUST be null.
```

```
    'direction': (String) The positioning flag. If 'relative_to' is null, this
    value MUST be "center". Otherwise, it MUST be one of the 8 relative direction
     flags ("right", "left", "front", "back", "back-left", "back-right", "front-
    left", "front-right") indicating its position relative to the sub-scene named
     in 'relative_to'. Remember that these relative positions between subscenes
    are fixed axis directions in relation to the subscene in question. So "front"
     corresponds to the -Y axis, "back" to the +Y axis, "left" to the +X axis,
    and "right" to the -X axis. Diagonal directions are combinations of these: "
    front-left" lies in the -Y and +X direction, "front-right" in -Y and -X, "
    back-left" in +Y and +X, and "back-right" in +Y and -X. This is important so
    you don't commit mistakes by putting sub-scenes in the same place.



Constraints & Guidelines:
You MUST include every object mentioned in the original user prompt in exactly
    one of the output sub-scenes.
Try to preserve all relationships and details mentioned in the original prompt,
    unless it makes it impossible to do independent subscenes.
The final output MUST be a valid JSON list `[...]` containing one or more valid
    sub-scene JSON objects `{...}` structured exactly as defined above. Ensure
    correct JSON formatting.
Do not, under any circumstances, include in the description of a given subscene
    any references to other subscenes and their objects. So when there is a
    request to exist some orientation or a relative positioning of objects that
    you decided to be in different subscenes you have to use axis alignment in
    order to have the right orientation, choose the position of the subscene in
    relation to other subscenes that accomplish that relative positioning (again
    without referring to any other subcenes names or objects present in other
    subscenes)
When assigning the subscene position in relation to another, think about it as if
     you were dealing with a 2d grid where each direction of the grid is defined
    by the positional flags. This is important, especially for subscenes where
    the relative positioning is not relative to the center, so that you dont put
    the scene in an already occupied place.
Do not add any text regarding ground planes, walls, etc, in the descriptions of
    the subscene if they are not explicitly mentioned in the user request.

Output structure example:
```

```
[
  {
    "subscene_name": "<Center_subscene_name>",
    "description": "<description>",
    "objects_list": ["<object1>", "<object2>", ...],
    "relative_positioning": {
      "relative_to": null,
      "direction": "center"
    }
  },
  {
    "subscene_name": "<Subscene_name>",
    "description": "<description>",
    "objects_list": ["<object3>", "<object4>", ...],
    "relative_positioning": {
      "relative_to": "<Center_subscene_name>",
      "direction": "<Direction>"
    }
  }
]



End of example



The user input: {initial_input}

"""
```

Listing A.1: Prompt for Scene Division

Prompt to select the best asset out of the top n candidates:

```
"""

Your task is to select the asset that best suits the user's needs from a given
    list of asset descriptions.

Asset descriptions: {all_assets_info}
```

```
User's wanted asset: {asset_desc}

You must provide only the name of the chosen asset in the following format: "[
    asset_name]"

Notes:
Always enclose the asset name in square brackets.
Even if there is no perfect match for the user's request, you must still select
    an asset.



"""
```

Listing A.2: Prompt to select the best asset out of the top n candidates

Prompt to develop the scene graph:

```
"""

Role:
You are an AI Scene Analyst. You are a specialist at converting the available
    information provided about a 3D scene and transforming it into a complete and
     precise description.

Your Task:
Analyze the provided User Request and Pre-existing Scene information. And create
    a textual, logical scene description. This description MUST detail every
    object's state and its spatial/orientational relationships with other objects
     using standardized terminology and a clear structure.

Inputs info
1.  **User Request:** (A free text description of the scene the user wants to be
    accomplished)
2.  **Pre-existing Scene:** (Description of objects already present, "None" if
    empty).

Output Requirements (Logical Scene Description):
-   **Format:** Use clear headings (e.g., `### Object: [Object Name]`, `###
    Relationships`) and bullet points for a better structure.
```

```
-   **Object Identification:** Refer to objects consistently using names from the
     user request or pre-existing scene (e.g., "the red car", "table", "tree_1").
     If multiple instances are requested without specific names (e.g., "add 5
    books"), refer to them systematically (e.g., "book_instance_1", "
    book_instance_2", etc.).
-   **Completeness:** Address every object mentioned or implied in the request
    and pre-existing scene.
-   **Clarity:** Use precise language. Avoid vague terms where a more specific
    description is possible.
-   **Dont omit details:** leave descriptions of relations /objects if necessary


Guidelines for Description Construction:

1.  **Deconstruct Request:** Break down the user request into individual objects,
     their state, their relation with the overall environment, and finally their
    spatial and orientation relationships with other objects
2.  **Integrate Existing Scene:** Combine the request with the pre-existing scene
     information logically. Resolve conflicts by prioritizing the user request
    unless it creates a physical impossibility.
3.  **Identification of the anchor objects:** Each logical description of the
    scene that you have to formulate must have an object whose position and
    orientation must not depend on others. You can think of if as the root of the
     tree, and from this root, is possible to describe other elements of the
    scene by adding new nodes that represent other objests that are related by
    the spatial and orientational relashioships (this new children can subquently
     have other children and so on, but not necessarly an children only haves a
    father node )
4.  **Description of each object:**
For each object an the following should be defined (assithe from some exceptions
    clarified later)
 * An object id: This should be a completely unique name exclusive to this object
 * Intrinsic Attributes: attributes of the object that are stated or inferred
    from the information available.
 * State of the object: this should clarify all the descriptors of the object's
    state on the scene that don't rely on other objects:
 - its stance (obligatory): if the object is in its normal standing position (
    upright) or any other abnormal state (namely lying down)
```

```
- its support (obligatory): if the object is grounded (on the floor), floating,
   or on top of another object (at this stage, the top object is not referred)
- Independent spatial and orientational state: Information about the object's
   position and orientation that is independent of another object (optional only
    if present, but mandatory for the anchor object):
-- Central position of the scene (this is the default position of the anchor
   object)
-- Orientation defined by the axis (ex. orineted towards x, -x, y, -y)
* Spatial and orientational relationships:
- Spatial relationships:
-- These relationships are the ones that define the object's position in
   relation to another object's position and/or orientation.
-- It is mandatory that at least one is defined (useless is the central object
   and further exceptions), and multiple can be defined if not contradictory.
-- For these relations, all should be paired with an indication of the distance
   (touching, very near, near, far, very far, etc.)
-- For the case of an object on top of another, it should not have a distance
   specified, but instead explain where on top the top object is in relation to
   the bottom.
-- If an object is on top of another, it should still be explicitly specified in
    this field even if already specified in the intrinsic attributes.
- Orientation relationships:
- The relationships that define the orientation of an object in relation to
   another object's position and/or orientation.
- This is not a mandatory relation to objects where the orientation is not
   relevant, like purely symmetric objects
- An object should only be the target of one orientation relation because that is
    enough for an undoubtedly definition of its orientation.
- If an object orientation is defined in relation to an axis, that also counts,
   so there shouldn't be any other orientation-related relation towards this
   object.


Exceptions:
- Some objects, instead of having the Spatial and orientational relationships
   part, should be directed to what is called a group relationship.
- These group relationships are relations that include a target to define more
   than one object position and orientation.
```

```
- This group relations should include a term describing the object involved and a
    full description of what it is about.
- Its most common use is to define rows of objects. We can easily defines the
    position and orientation of a variety of objects in relation to a leading
    object (don't make part of the group relation)



Notes:
- You can't put two objects in the same position. This means that an object
    position can be defined in the exact same way as another. For example, if
    multiple objects are on the right of another object. You can define a row
    that is parallel to that object, containing all the objects, or put one
    object on the right, and then put other objects in relation to that one on
    the right, and so on.
- For relative positioning isn't enough to say that an object is near another.
    You should state that it is on the right, left, etc, or relative to an axis
    in relation to another object according to the user request (or add this
    information if lacking in the user request)
- If an object is on top of another object, or is implicit, you should always
    state that the given object is on top of another. Then you should give a more
    detailed description stating more details if present on how the top object
    is positioned (but remember that this is an elaboration of the on top
    relation, dont create any other relative positioning relations with the top
    object as the target)
- If an object is on one lane of a road, that object is on top of the road, then
    you should elaborate on how the top object is positioned on top of the other
- If you make a group relationship, the individual object should still be
    instantiated.



Execute the Task:
Now, analyze the inputs below. Think step-by-step, applying the guidelines.
    Construct the Logical Scene Description textually, ensuring maximum clarity
    and structure for the next stage.

Inputs:
```

```
User Request: {user_request}

Pre-existing Scene: {pre_existing_scene_interpreted}



"""
```

Listing A.3: Pronpt to construct the scene graph

Prompt for the execution plan, previous to the code generation:

```
"""

Your task is to generate an **`executable_plan`**: an ordered sequence of steps
    required to construct the scene described in the input **Scene Graph**. The
    plan translates the objective scene descriptors into imperative actions.

Description of the inputs:
1.  The **Scene Graph** (a structured representation, like JSON, detailing
    objects, their IDs, types, attributes, and relationships).
2.  The **Function Documentation** (available Python functions, their names, and
    expected parameters).

(...)

**Plan Requirements:**

*  **Structure:** The plan must be a **JSON list** of "step" objects. Each step
    object must contain the following fields:

1.  `step_id`: (Integer) A unique, sequential identifier for the step (e.g., 1,
    2, 3...). This defines the primary execution order.
2.  `action`: (String) A concise category representing the core purpose of the
    step, inferred from the Scene Graph element being processed and the chosen
    function. Examples: `IMPORT_OBJECT`, `SET_STATE`, `PLACE_RELATIVE`, `
    ORIENT_ABSOLUTE`, `ORIENT_RELATIVE`, `QUALITATIVE_ACTION`.
3.  `object_id`: (String) The unique identifier (from the input Scene Graph) of
    the primary object this step targets or acts upon.
```

```
4.  `description`: (String) A brief natural language description of what this
    specific step achieves (e.g., "Create the 'table' object", "Execute '
    on_top_of' relationship between 'book' and 'table'", "Set 'lamp' state to '
    upright'", "Orient 'table' using absolute rotation").
5.  `function_call`: (Object or Null) Details of the specific function to execute
     for this step.
    *   If a suitable function exists in the provided documentation:
        *   `name`: (String) The **exact** name of the Python function to call (e
    .g., `put_on_top`, `set_object_state`, `set_orientation_absolute`).
        *   `parameters`: (Object) A dictionary where keys are the **exact**
    argument names expected by the function, and values are the corresponding
    values. These values must be derived *directly* from the input **Scene Graph
    ** (e.g., object IDs like `subject_object_id: "book1"`, attributes like `
    color: "red"`) and spatial reasoning based on relationships.
    *   If no suitable function exists (`action` is `QUALITATIVE_ACTION`): This
    field must be `null`.
6.  `qualitative_instruction`: (String or Null)
    *   If `action` is `QUALITATIVE_ACTION`: A clear description of the required
    action (derived from the Scene Graph) that **cannot** be mapped to an
    existing documented function.
    *   Otherwise: This field must be `null`.



*   Each object (without exception) must have four fundamental facts clearly
    defined in the plan:

0. Importing the object into the scene

1. The object's state in the scene
   This refers to the inherent state of the object, independent of surrounding
    objects. This includes:
   - Whether the object is lying down, upright, or has a specific inclination (
    mandatory)
   - Whether the object is grounded or floating (optional - by default, every
    object is considered grounded,
     but there can be object-to-object relations that override this, such as one
    object being placed on top of another.
```

```
2. The object's relative position to other objects in the scene
   - Spatial relationships between the object and surrounding objects that
    unambiguously describe its position
     in relation to them (there may be multiple such relationships)
   - Exception: some objects may be placed centrally in the scene, independent of
     other objects, serving as a
     reference point around which the rest of the scene is constructed

3. The orientation of the object
   Each object must have one and only one defined orientation, which can be
    specified in one of two ways:
   - Orientation is defined relative to the global axes
   - Orientation is defined relative to the position or orientation of other
    objects

*  Plan Order and Scene Construction

The plan should follow a logical order that enables the construction of the scene
     step by step.

- When constructing a scene, there must be a clear sequence in which actions are
    executed to ensure the scene
  is built successfully. Some steps require objects to already be in their
    correct positions and/or orientations,
  while others do not.
- For example, properties like "lying down," "upright," or "specific inclination
    ," as well as orientation
  defined by the global axes, can be specified independently of the rest of the
    scene, while orientation in relation to other objects or position in relation
     to another object requires that the other object already be in a valid place
    /orientation
- If clearly defined, one object should serve as the center of the scene (placed
    at coordinates (0, 0, 0)),
  with the rest of the scene built sequentially around it. If no such object is
    explicitly defined,
  the plan should identify the best candidate to serve this role.
- For each object, first the are the individual relations, constraints, then the
    positional relations/constraints, and finally the orientation relations/
    constraints
```

```
- Note that you should use common sense (assume one unit is one meter) to define
    reasonable distances for instructions where you place objects relative to
    another (but remember that the distance is between the nearest points of the
    objects, not the center, so don't overdo it)
- When putting objects on top of another, remember that you can personalize the
    place. The CENTER is not the only option, especially when there are multiple
    objects on top of a singular object
- The objects' names should follow strictly the names of the objects in the scene
     graph.
- Ground objects like rugs or roads should be placed upright (upright and
    oriented correctly to what is requested)
- First, import all the necessary asset instances before starting to organize
    them on the scene.
- Respect the scene graph. You should use the functions in the library that
    better translate the scene graph instructions. (For example, if an object is
    stated to be on the left of another, you should not position it in relation
    to an axis, and the opposite is also true.)


**Example Step (Applying an "on_top_of" relationship):**

Assuming the input Scene Graph contains `{{"subject": "book1", "verb": "on_top_of
    ", "object": "table1"}}` and you have a documented function `place_object(
    subject_id, reference_id, relationship_type)`:

{plan_format_example}

Generate the complete, ordered executable_plan as a single JSON list following
    this structure, ensuring all elements from the input Scene Graph are
    translated into executable steps or qualitative instructions.

There are the inputs:
**Scene Graph**: {scene_graph}
**Function Documentation**: {code_library_documentation}


"""
```

Listing A.4: Prompt to generate the execution plan

Prompt to generate the Python code for the first iteration of the sub-scene:

```
"""
Your task is to execute a given plan to construct a Python script that creates a
    3d scene in Blender.

Inputs info
- **Execution plan**: a detailed plan on how to construct the scene. There is the
     overall structure of each entry of the plan:
1.  `step_id`: (Integer) A unique, sequential identifier for the step (e.g., 1,
    2, 3...). This defines the primary execution order.
2.  `action`: (String) A concise category representing the core purpose of the
    step, inferred from the Scene Graph element being processed and the chosen
    function. Examples: `IMPORT_OBJECT`, `SET_STATE`, `PLACE_RELATIVE`, `
    ORIENT_ABSOLUTE`, `ORIENT_RELATIVE`, `QUALITATIVE_ACTION`.
3.  `object_id`: (String) The unique identifier (from the input Scene Graph) of
    the primary object this step targets or acts upon.
4.  `description`: (String) A brief natural language description of what this
    specific step achieves (e.g., "Create the 'table' object", "Execute '
    on_top_of' relationship between 'book' and 'table'", "Set 'lamp' state to '
    upright'", "Orient 'table' using absolute rotation").
5.  `function_call`: (Object or Null) Details of the specific function to execute
     for this step.
    *   If a suitable function exists in the provided documentation:
        *   `name`: (String) The **exact** name of the Python function to call (e
    .g., `put_on_top`, `set_object_state`, `set_orientation_absolute`).
        *   `parameters`: (Object) A dictionary where keys are the **exact**
    argument names expected by the function, and values are the corresponding
    values. These values must be derived *directly* from the input **Scene Graph
    ** (e.g., object IDs like `subject_object_id: "book1"`, attributes like `
    color: "red"`) and spatial reasoning based on relationships.
    *   If no suitable function exists (`action` is `QUALITATIVE_ACTION`): This
    field must be `null`.
6.  `qualitative_instruction`: (String or Null)
    *   If `action` is `QUALITATIVE_ACTION`: A clear description of the required
    action (derived from the Scene Graph) that **cannot** be mapped to an
    existing documented function.
    *   Otherwise: This field must be `null`.
```

```
- **Assets paths**: The list of assets paths to be used to do the script (YOU
    CANT USE ANY OTHER ELEMENTS BESIDES THE ONES PRESENT IN THIS LIST)
- **Library documentation**: The documentation of the library of functions, to
    provide you with better context of the functions referred to in the plan and
    the tools available.

Very important notes:
- You need to use the full path of an asset to import it, like they are provided
    in the  **Assets paths**.
- You should change the assets instances' names to match the names present in the
     plan.
-

Important remarks:
- Always include this line of code: "from code_library.code_library import *"
    This enables the use of the functions from the code library.
- Simplicity: Use library functions where possible to achieve the required state
    simply. Avoid overly complex calculations if a library function achieves the
    init_input goal directly.
- Don't include ''' on your code.
- Do not include any camera settings or related configurations.
- Do not create a main clause.
- Do not forget to import `mathutils`.
- Ensure that the code is functional and runs without errors related to the use
    of functions, packages, or object attributes.
- Be very focused to not commit errors with the tool you will be using to make
    something, and when using the functions, be very careful to use the correct
    arguments
- First, import all the necessary asset instances before starting to organize
    them on the scene.



Inputs:
- **Execution plan**: {execution_plan}

- **Assets paths**: {list_path} IS MANDATORY THAT YOU USE THE EXACT PATHS THAT
    ARE IN THERE, DO NOT INCLUDE PARTIAL PATHS

- **Library documentation**: {code_library_documentation}
```

```
"""
```

Listing A.5: Prompt to generate the Python code for the first iteration of the sub-scene

Prompt to generate the visual feedback in the feedback loop:

```
"""
Role:
You are an AI Scene Composition Analyst. Your task is to rigorously evaluate
    rendered images of a 3D scene against its corresponding objective scene graph
     to identify discrepancies and suggest corrections.

Task:
Analyze the provided images (front, back, top-down views) and compare the
    depicted object arrangement against the relationships and states defined in
    the `objective_scene_graph`. Identify specific issues where the visual
    representation deviates from the graph or exhibits significant visual errors
    (like collisions). Output your findings and suggested corrections using the
    **structured Markdown format** specified below.

Inputs:
1.  **Objective Scene Graph (`objective_scene_graph`):** \n{objective_scene_graph
    }\n (This defines the intended state, relationships, and object IDs).
2.  **Image Views:** Access to images rendered from the front, back, and top-down
     perspectives of the generated scene.

Evaluation Criteria (Focus on these issue types):
1.  **Relationship Mismatch:** Visual relationship contradicts the graph's `
    relation_type`.
2.  **Parameter Mismatch:** Visual relationship inconsistent with implied/
    explicit parameters (distance, alignment).
3.  **State Mismatch:** Visual state (orientation) contradicts the graph's
    defined state.
4.  **Collision/Intersection:** Unnatural object penetration.
5.  **Floating/Instability:** Objects defying gravity without justification in
    the graph.
6.  **Missing/Extra Objects:** Visuals don't match the graph's object list.
```

```
Guidelines:
-   **Prioritize Graph Adherence:** Ensure the visual scene accurately reflects
    the `objective_scene_graph`.
-   **Use All Views:** Synthesize information from all views (Top-down for XY,
    Front/Back for Z/orientation).
-   **Object Identification:** Refer to objects using their `id`s from the scene
    graph.
-   **Focus:** Concentrate on placement, orientation, and interactions. Ignore
    minor lighting/texture issues.
-   **Be Specific:** Clearly describe issues and involved objects.

**Output Format (Structured Markdown):**

Your entire response MUST follow this structure precisely:

```markdown
## Scene Analysis Report

**Overall Assessment:** [Choose ONE: Matches Graph | Minor Deviations Found |
    Major Deviations Found | Errors Found]

---

### Issue 1: [Brief Title Summarizing the Issue]

*   **Issue Type:** [Choose from Evaluation Criteria list: e.g., Relationship
    Mismatch, Collision/Intersection]
*   **Involved Objects:** [`<object_id_1>`, `<object_id_2>`, ...]
*   **Description:** [Clear textual description of the visual discrepancy or
    error found in the images.]
*   **Scene Graph Reference:** [Describe the relevant part of the scene graph, e.
    g., "Graph specifies 'cup' `on_surface_of` 'table.001'".]
*   **Suggested Correction:** [Describe the suggested change, aiming for
    actionable advice. E.g., "Adjust Z position of 'cup' to make contact with '
    table.001' surface.", or "Modify parameters for `adjacent_to` relationship
    between 'chair.001' and 'table.001' to decrease distance."]

---
```
```

```
### Issue 2: [Brief Title Summarizing the Issue]

*   **Issue Type:** [...]
*   **Involved Objects:** [...]
*   **Description:** [...]
*   **Scene Graph Reference:** [...]
*   **Suggested Correction:** [...]

---
*(Repeat the Issue block for each distinct problem found. If no issues, state "No
     issues found." under the Overall Assessment and omit the Issue blocks.)*

**General Comments:** [Optional: Any overall feedback not captured in specific
    issues.]

"""
```

Listing A.6: Prompt to generate the visual feedback

Prompt to generate the constraint list to be evaluated in the feedback loop:

```
"""
Your task is to construct a JSON that contains all the constraints in the 3D
    scene described in the scene graph that will be provided. The aim of this is
    to use this framework to get an evaluation of how well the constraints are
    satisfied.

Explanation of the inputs:
- Score framework documentation:  the documentation explaining the constraints
    available and how to use them.
- Scene graph: a scene graph where the 3d scene is conceptualized.



Technical Guidelines:
- The output should be the asked JSON and nothing more.
- You must use the names of the objects present in the scene graph. Do not come
    up with new names or slightly changed versions of the names in the scene
    graph. The names should be the exact same as the Object Name/object id
```

```
- Respect the constraint formatting. Remember to put all parameters needed for
    each constraint with the correct inputs.
- Dont define any constraints that have the same name as object_A and object_B



Conceptual Guidelines:
- You should reflect the constraints present in the scene graph into the JSON
    file using the documentation guidance.
- You have to make an effort to include all the constraints, but if there is a
    constraint that can't be translated using the documentation is better to
    ignore it than define it wrong.

Some remarks
- Constraints don't work for constructs of groups of objects. They should always
    be applied in relation to the individual objects.

There are the inputs:
Score  framework documentation with some utilization examples:

{score_documentation }

Scene graph:

{scene_graph}

"""
```

Listing A.7: Prompt to generate constraint list

Prompt to generate the constraint feedback in the feedback loop:

```
"""

Your task is to reach conclusions over how well the relations/constraints are
    satisfied based on a report that documents the results on a 3d scene that is
    meant to replicate the reference scene graph.
```

```
This report should be the result of a variety of functions that evaluate how well
     the constraints are respected. The way it works is that 0 means it is not
    respected at all, and 1 is respected.
(Note that the evaluation function is exponential for intermediate values.)

The objective of this analysis is to conclude which ones are well executed and
    the ones are not well executed. For the not well, the results of the
    constraint evaluation can give insights on why they are not well executed.
Just keep in mind that the constraint evaluation isn't perfect. There are cases
    where the sensitivity is too much and others where it is too low.
Just point out as wrong constraints that are very near zero (0.1 -0).



There are the inputs:

Reference Scene graph: {scene_graph}

Evaluation results: {constrain_evaluation_results}

Current scene graph (this is the scene graph representing the current state of
    the 3d scene, this should serve as extra guidance when reaching the
    conclusions over what is good and what is wrong):  {current_scene_graph}

"""
```

Listing A.8: Prompt to generate the constraint feedback

Prompt to generate to apply the scene changes in the feedback loop:

```
"""
Role:
You are an AI Scene Revision Engineer. Your task is to generate a Blender Python
    script that modifies an existing scene to precisely match the user's
    intentions expressed in init_input, using the objective_scene_graph as a
    structured interpretation and incorporating corrections based on provided
    evaluations and constraints to ensure a plausible and visually coherent
    result.

Task:
```

```
Generate a complete, ready-to-use Blender Python script (bpy) that performs the
    necessary transformations, additions, or deletions to make the scene
    described by current_scene_graph align with the user's intent described in
    init_input. Use the objective_scene_graph as a primary reference for
    structure derived from the intent, consult evaluation results (
    constrain_evaluation_results, image_evaluation_results) for refinement and
    physical plausibility, utilize the provided code_library, and adhere strictly
     to the plan you will formulate first.

Inputs:
User Intention (init_input): {init_input} (PRIMARY GOAL: The textual description
    of the desired scene. The final scene MUST satisfy the core meaning and
    elements described here.)
Objective Scene Graph (objective_scene_graph): {objective_scene_graph} (A
    structured representation derived from init_input. Use this as a guide for
    object presence and relationships, but it is secondary to init_input if there
    's a direct conflict in meaning. Note that orientations/coordinates within
    this graph are often defaults from asset import and likely not precise user
    specifications unless explicitly stated as such in the context of init_input
    .)
Current Scene Graph (current_scene_graph): {current_scene_graph} (The starting
    state - structure, objects, relationships.)
Spatial Constraints Evaluation (constrain_evaluation_results): {
    constrain_evaluation_results} (Results from rule-based checks, e.g.,
    collision detection, boundary checks. Use to ensure physical plausibility in
    realizing init_input.)
Visual Evaluation Results (image_evaluation_results): {image_evaluation_results}
    (Structured Markdown report from VLM comparing visuals. Use this to refine
    placement, orientation, and resolve ambiguities in a way that best reflects
    the visual implication of init_input and corrects obvious visual errors
    flagged in Suggested Correction fields.)
Code Library (code_library): {code_library_documentation} (Available Python
    functions for object manipulation.)
Asset Paths (asset_path): {asset_path} (Paths for importing new assets if needed
    .)
Objects in Scene (objects_in_scene): {objects_in_scene} (Canonical list of
    current object instance IDs in Blender. Use these exact IDs when referencing
    objects in your script.)
```

```
Decision Logic & Prioritization (CRUCIAL):

Primary Goal: The script MUST aim to make the scene semantically match the
    init_input. If the current_scene_graph already satisfies the description in
    init_input, minimal or no changes should be made, even if it differs from
    objective_scene_graph.
Interpret init_input: Understand the core objects, their states, and
    relationships mentioned in init_input.
It is very important that you build the script logic so that the scene is
    constructed step by step. This way, you avoid making the mistake of setting
    relative positions and orientations before the reference object is correctly
    placed and oriented. Start by placing one object accurately, and then use
    that correct placement to position the other elements. The same applies to
    orientations.

Use objective_scene_graph as a Guide:

Compare current_scene_graph to objective_scene_graph. Discrepancies indicate
    potential areas needing change if they also contradict init_input.
If objective_scene_graph details align with init_input, use them to plan actions
    (ADD, DELETE, MODIFY relationships).
If objective_scene_graph details contradict init_input, ignore those specific
    graph details and prioritize fulfilling init_input.

Incorporate Feedback for Refinement & Plausibility:
Use constrain_evaluation_results to fix physical impossibilities (collisions,
    floating objects) that prevent a sensible realization of init_input.
    Prioritize fixing these.
Use image_evaluation_results (Suggested Correction) to:
Refine parameters (placement, orientation, distance) to better match the visual
    interpretation of init_input.
Resolve ambiguities in init_input (e.g., init_input says "cup near book", VLM
    suggests "cup to the left of book").
Correct visual errors only if they detract from achieving the core init_input.
Do not try to change the scene lighting.
```

```
Crucially: If visual feedback suggests a change that looks "better" but alters
    the fundamental meaning of init_input, ignore that feedback and stick to
    init_input. If visual feedback confirms the scene already looks good and
    aligns with init_input, this is a strong signal not to make unnecessary
    changes, even if the objective_scene_graph has minor differences.

Conflict Resolution Summary:
init_input > Constraint Violations (Fix violations while preserving intent) >
    Visual Feedback (Use for refinement aligned with intent) > Objective Scene
    Graph (Use as structural guide subordinate to intent).
If the current scene satisfies init_input, avoid changes unless necessary to fix
    constraint violations or critical visual errors flagged by feedback that
    misrepresent the intent.

Phase 1: Planning (Mandatory)

Before generating any Python code, outline a step-by-step plan detailing the
    necessary actions. For each required action (ADD, DELETE, MODIFY):
Action Type: ADD / DELETE / MODIFY / NO_ACTION
Target Object ID(s): The exact ID(s) from objects_in_scene or new IDs to be
    created (e.g., chair.002).
Reason: State why this action is needed, specifically in relation to init_input.
    (e.g., "Object chair missing, required by init_input", "Relationship cup
    on_surface_of table required by init_input, current graph incorrect", "Adjust
     lamp position based on VLM feedback to better realize 'near the sofa' from
    init_input", "Fix collision between book and table reported by constraints",
    "No action needed, current scene already satisfies init_input"). Reference
    specific feedback points if applicable, explaining how they help achieve the
    init_input goal.
(If MODIFY/ADD):
Objective: Describe the target state/relationship derived from init_input (e.g.,
    "Ensure cup is on_surface_of table.001 as per init_input").
Method: Specify the exact code_library function OR bpy command to use (e.g.,
    library.place_on_surface, bpy.ops.object.delete).
Key Arguments: List the precise arguments and their values. Adjust arguments
    based on feedback reconciliation if it enhances the alignment with init_input
    .
(If DELETE): State the object ID to be deleted and why it contradicts init_input
    or is extraneous.
```

```
(If NO_ACTION): Briefly state why the current scene already fulfills init_input.

Phase 2: Script Generation

Based ONLY on the validated plan above, generate the complete Blender Python
    script.
Script Requirements:
Import necessary modules: import bpy, import mathutils.
Import library functions: from code_library.code_library import * (or specific
    imports).
Implement each step from your plan sequentially. If the plan is NO_ACTION,
    generate an empty script or a script with only comments indicating no changes
     are needed.
Use the exact object instance IDs from objects_in_scene.
Handle object selection/referencing carefully.
Use bpy.ops.wm.obj_import(filepath=...) for imports, using asset_path.
Use bpy.ops.object.duplicate() for duplicates.
Use bpy.ops.object.delete() for deletions.
Ensure correct naming (asset_name, asset_name.001, ...).
The script must be functional and directly executable in Blender.
Simplicity: Use library functions where possible to achieve the required state
    simply. Avoid overly complex calculations if a library function achieves the
    init_input goal directly.
Don't rewrite the functions present in the library in the script. just make the
    function calls.
Blender Specific Notes (Reminder):
The default import position is grounded, upright, front facing -Y.
The default import includes +90 X rotation for uprightness; removing it makes
    objects lie flat.
Z-axis rotation primarily controls left/right orientation on the XY plane.

Execute:
Dont rewrite the functions present in the library in the script. Just make the
    function calls.
Just call the functions from the library
Carefully analyze all inputs, prioritizing init_input according to the Decision
    Logic & Prioritization rules.
Formulate the detailed step-by-step Plan, justifying each step based on alignment
     with init_input.
```

```
Translate the Plan precisely into the final Blender Python Script. If the plan
    dictates no changes, the script should reflect that (e.g., be empty or
    contain only comments).

"""
```

Listing A.9: Prompt to generate the Python to apply scene changes in feedback loop

Prompt to select the weather:

```
"""
Your task is to decide the best weather state out of some given options that
    better suit the user request for the creation of a 3D scene.

These are the available options:
"Clear"
"cloudy"
"super_cloudy"
"atmosferic_sand"

There is the user request: {User_input}

Your final answer should be the name of the best candidate weather state
    contained in [] paired with an hour of day (1 - 24):

["Clear", "12"] #always put the weather state first and then the hour,

"""
```

Listing A.10: Prompt to select the weather

Prompt to select the floor:

```
"""
Your task is to decide the best floor texture out of some given options that
    better suit the user's request for the creation of a 3D scene.

These are the available options:
"dark_wood_floor.png"
"light_dirt.png"
```

```
"light_wood_floor.png"
"orange_dirt.png"
"side_walk_floor.png"
"grass.png"
"side_walk_floor_white.png"

There is the user request: {User_input}

Your final answer should be the name of the best candidate texture contained in
    [], like this example:

["dark_wood_floor.png"]

"""
```

Listing A.11: Prompt to select the floor texture

A.2. Function library: All defined functions

The function library is composed of many functions that leverage the Blender API to place and orient the objects in the scene with an abstraction to numerical considerations necessary. These functions were also designed to be general, meaning they are not tailored to a specific domain or assets.

There is a comprehensive list of all the functions included:

- `import_asset(filepath, object_name=None)`: Imports an OBJ asset from a specified file path and can optionally rename it.
- `duplicate_object(object_to_duplicate, new_name=None)`: Creates a full, independent copy of a specified Blender object.
- `remove_object(object_to_remove)`: Deletes a specified object from the scene.
- `set_object_grounded(object_to_ground, ground_level=0.0)`: Moves an object so its lowest point rests on a specified ground level (Z-axis).
- `set_object_upright(object_to_orient, preserve_z_rotation=True)`: Orients an object to be upright, aligning its local +Z axis with the world's +Z axis.

- `set_object_laying_down_directional(...)`: Orients an object to a ”laying down” position, simulating it falling in a specific direction (FORWARD, BACKWARD, etc.).
- `align_object_to_axis(object_to_align, target_axis="+X")`: Rotates an upright object so its front (local -Y) faces a specific world axis.
- `set_object_floating(object_to_float, float_height, ground_level=0.0)`: Places an object so its lowest point is a specific height above a ground level.
- `place_relative_to_object_facing(...)`: Places an object relative to a reference object based on its facing direction (e.g., to the left, right, front, or behind).
- `place_object_relative_to_other_axis(...)`: Places an object relative to another along a specified world axis (+X, -X, +Y, -Y) with a defined distance.
- `orient_object_towards(object_to_orient, target_object)`: Rotates an object to face towards a specified target object.
- `orient_object_away_from_y_front(...)`: Rotates an object to face directly away from a target object.
- `match_object_orientation(object_to_orient_input, reference_obj_input)`: Matches the world orientation of one object to that of a reference object.
- `place_object_at_coordinates_simple(obj_ref_or_name, coords=(0.0, 0.0, 0.0))`: Sets the location of an object’s origin to specific world coordinates, defaulting to the world origin (0,0,0).
- `put_on_top_normalized(...)`: Places one object on top of another, with precise placement controlled by normalized coordinates on the bottom object’s surface.
- `arrange_objects_in_row(...)`: Arranges a list of objects into a row with customizable spacing, orientation, and positioning (either absolute or relative to a reference object).

It is important to note that the functions are made assuming that the assets with (0,0,0) rotation in their axis have their front pointing towards the -y direction (so the front of the objects is their local -y direction). This is a convention defined to implement the functions more straightforwardly (this does not make impossible the use of other assets with different defaults, but another custom asset importer needs to be made or the current one be

changed). There is also the fact that assets with multiple meshes can show unexpected behaviors when using these functions.

A.3. Constraint evaluation: All defined constraints and report format

Similarly to the function library, the constraints defined were meant to be general, while encompassing most of the basic relations between objects and their possible states.

There is a comprehensive list of all the constraints defined:

- `avoid_overlap`: Scores the separation of two objects, returning a high score for no overlap and a decreasing score based on the penetration depth if they collide.
- `min_distance`: Scores how well two objects maintain a minimum distance, returning a high score if the distance is met or exceeded, and a lower score if they are too close.
- `ensure_grounded`: Scores how close an object's lowest point is to a specified ground-level Z-coordinate.
- `ensure_floating`: Scores whether an object's lowest point is within a specified vertical height range, penalizing deviations outside this range.
- `relative_position_local`: Scores the position of one object relative to another's local axes (e.g., 'front', 'back') and a target distance category.
- `relative_position_world`: Scores the position of one object relative to another along a global world axis (e.g., +X, -Y) and a target distance category.
- `on_top_of`: Scores how well one object is vertically stacked on another by evaluating both the vertical gap and the horizontal alignment.
- `inside`: Scores how completely one object is contained within another, penalizing any part of the object that protrudes outside the container.
- `orient_front_to_world_axis`: Scores the alignment of an object's 'front' vector with a specified global cardinal axis.
- `orient_front_to_object_pos`: Scores how well an object's 'front' is oriented towards or away from another object's position.
- `align_local_top_with_world_up`: Scores how "upright" an object is by measuring the alignment of its local 'top' axis with the world's up vector.

There is an example of a report automatically generated using the constraint evaluation. We give not only details about the individual constraints but also by type of constraint and considering all the constraints together.

Generated report example

```
Constraint 1 (ID: tree_grounded, Type: ensure_grounded):
  Involved: Obj: tree_1
  Score (0-1): 0.002
  Primary Scaled Deviation(s): 0.2170
  Scoring Details: ObjMinZ: -0.2170, TargetGroundZ: 0.0000. AbsDeviation: 0.2170
    (Signed: -0.2170). Scored with exp decay (k=30.00) on effective deviation
    0.2070.

Constraint 2 (ID: tree_upright, Type: align_local_top_with_world_up):
  Involved: Obj: tree_1
  Score (0-1): 1.000
  Primary Scaled Deviation(s): 0.0000
  Scoring Details: Obj='tree_1' local_top to WorldUp. AngleDev=0.00deg (Dot
    =1.000). Angle within perfect score tolerance. Score: 1.00

Constraint 3 (ID: tree_faces_z, Type: orient_front_to_world_axis):
  Involved: Obj: tree_1
  Score (0-1): 0.227
  Primary Scaled Deviation(s): 1.5708
  Scoring Details: Obj='tree_1' to WorldAxis='+z'. AngleDev=90.00deg (Dot=0.000).
     Scored with exp decay (k_angle=1.00) on effective angle_dev 85.00deg. Score:
     0.23

Constraint 4 (ID: park_bench_grounded, Type: ensure_grounded):
  Involved: Obj: park_bench_1
  Score (0-1): 1.000
  Primary Scaled Deviation(s): 0.0000
  Scoring Details: ObjMinZ: 0.0000, TargetGroundZ: 0.0000. AbsDeviation: 0.0000 (
    Signed: 0.0000). Deviation within perfect score tolerance (<=0.0100).

Constraint 5 (ID: park_bench_upright, Type: align_local_top_with_world_up):
  Involved: Obj: park_bench_1
```

```
  Score (0-1): 1.000
  Primary Scaled Deviation(s): 0.0000
  Scoring Details: Obj='park_bench_1' local_top to WorldUp. AngleDev=0.00deg (Dot
    =1.000). Angle within perfect score tolerance. Score: 1.00

Constraint 6 (ID: park_bench_near_tree, Type: relative_position_world):
  Involved: A: park_bench_1, B: tree_1
  Score (0-1): 0.468
  Primary Scaled Deviation(s): axial_deviation_from_range:0.4858,
    perpendicular_offset:0.0399
  Scoring Details: WorldAxis='+x'. Axial: TargetRange=[0.80-1.50], ActualDist
    =1.99, Status='too_far', DevFromRange=0.486. Axial score exp(k=5.0*eff_dev
    =0.436). PerpOffset: 0.040. Perp OK. Scores(Axial:0.11, Perp:1.00). Combined
    :0.47

Constraint 7 (ID: park_bench_oriented_to_tree, Type: orient_front_to_object_pos):
  Involved: A: park_bench_1, B: tree_1
  Score (0-1): 1.000
  Primary Scaled Deviation(s): 0.0008
  Scoring Details: ObjA='park_bench_1' towards ObjB='tree_1'. AngleDev=0.05deg (
    Dot=1.000, Dist=5.79). Angle within perfect score tolerance. Score: 1.00

Constraint 8 (ID: man_on_park_bench, Type: on_top_of):
  Involved: A: man_1, B: park_bench_1
  Score (0-1): 0.300
  Primary Scaled Deviation(s): vertical_absolute_deviation:0.5930,
    horizontal_deviation_active_mode:0.0355
  Scoring Details: Vertical: ActualMinZ_A=0.507, TargetZ_A_Bottom=1.100, AbsDev
    =0.593. Vert.score exp(k=50.0*eff_dev=0.588). Horizontal (mode='
    specific_placement' (placement='center'), type='specific_placement_offset'):
    Dev=0.036. Horiz.OK. Scores(Vert:0.00, Horiz:1.00). Combined:0.30

Constraint 9 (ID: man_upright, Type: align_local_top_with_world_up):
  Involved: Obj: man_1
  Score (0-1): 1.000
  Primary Scaled Deviation(s): 0.0000
  Scoring Details: Obj='man_1' local_top to WorldUp. AngleDev=0.00deg (Dot=1.000)
    . Angle within perfect score tolerance. Score: 1.00
```

```
Constraint 10 (ID: man_oriented_to_tree, Type: orient_front_to_object_pos):
  Involved: A: man_1, B: tree_1
  Score (0-1): 0.998
  Primary Scaled Deviation(s): 0.0889
  Scoring Details: ObjA='man_1' towards ObjB='tree_1'. AngleDev=5.09deg (Dot
    =0.996, Dist=5.79). Scored with exp decay (k_angle=1.00) on effective
    angle_dev 0.09deg. Score: 1.00

--- Evaluation Summary ---
Total constraints defined: 10
Constraints with processing errors: 0
Successfully evaluated constraints with scores: 10

--- Scores by Constraint Type ---
  Type: align_local_top_with_world_up (Instances: 3)
    Avg Score: 1.000, Median Score: 1.000
    Min Score: 1.000, Max Score: 1.000
  Type: ensure_grounded (Instances: 2)
    Avg Score: 0.501, Median Score: 0.501
    Min Score: 0.002, Max Score: 1.000
  Type: on_top_of (Instances: 1)
    Avg Score: 0.300, Median Score: 0.300
    Min Score: 0.300, Max Score: 0.300
  Type: orient_front_to_object_pos (Instances: 2)
    Avg Score: 0.999, Median Score: 0.999
    Min Score: 0.998, Max Score: 1.000
  Type: orient_front_to_world_axis (Instances: 1)
    Avg Score: 0.227, Median Score: 0.227
    Min Score: 0.227, Max Score: 0.227
  Type: relative_position_world (Instances: 1)
    Avg Score: 0.468, Median Score: 0.468
    Min Score: 0.468, Max Score: 0.468

--- Overall Scores ---
  Total Scored Instances: 10
  Overall Avg Score: 0.700
  Overall Median Score: 0.999
  Overall Min/Max Score: 0.002 / 1.000
```

A.4. Generated assets

There is a comprehensive list of the assets generated:

- Man
- Woman
- Truck with cargo
- Truck with no cargo
- Tree
- Traffic light
- Sunflower
- Street lamp
- Store umbrella
- Store
- Stop sign
- Small house
- Road (made with Blender tools)
- Residential building
- Red cone
- Red car
- Plastic table
- Plastic chair
- Crosswalk sign
- Park bench
- Motor bike
- Dog

- Danger Sign
- Cat
- Bush
- Bus
- Blue sports car
- Bike

A.5. Blender physics simulator

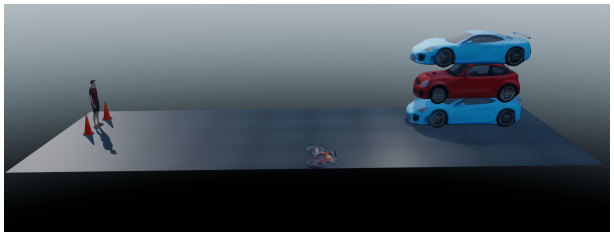
With the presence of physical implausibilities in scenes generated with our method (namely, objects floating or overlapping), it would be worthwhile to find a way to correct those.

Given that we are working with Blender, we tried the built-in physics simulator.

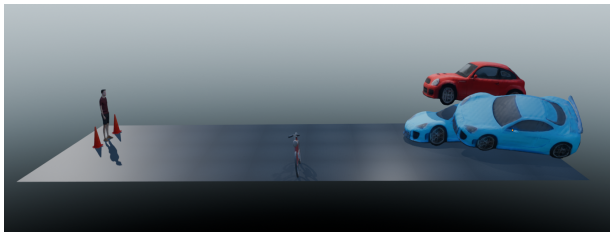
This physics simulator was effective at correcting errors related to physical implausibilities, but clearly did not make the scene layout better. Basically, the simulations were unstable, leading to large disruptions of the scene layout, namely when overlaps were present.

One thing to consider is that there were a vast number of hyperparameters. We tried a variety of different values and settings, but did not reach a satisfactory result.

There is an illustrative example of a scene before running the physics simulator and after, where we can see the problem.



(a) 3D scene before physics simulation



(b) 3D scene after physics simulation

Figure A.1: Example of a before and after application of the Blender physics simulator in a generated 3D scene

In this example, the pile of cars got destroyed, and the bike ended up in a more straight position, when it should have remained in a lying down position.

A.6. Optimization pipeline

Given that our pipeline still generated scenes with visible errors, not only in physical-related, but also layout-related errors, we decided to try an optimization framework to correct these errors. It's also a widespread tactic in literature to use optimization methods to help or even construct the scenes' layouts from the ground up.

First, the input to the layout optimization.

Given that the target changes are the positions and orientations of the objects on the scene, we decided to use the position of the objects in the scene and their respective orientations:

- To represent the position of an object, we used a single reference point (with (x,y,z) coordinates) located in the middle of the base of the object's bounding box. The fact of using a single point is to decrease the quantity of values to optimize while having something that is representative of the object's position in the scene. Now the choice of this specific point position, being the other realistic alternative the geometric center, is that this point is more representative of the object placement, given that does not introduce an z value dependent of the height of the assets (this when the asset is upright, that is typically the case).
- To represent the orientations, we choose to use the rotation of the asset in relation to the z-axis. This choice as two reasons, because in the vast majority of cases, the objects used are in an upright position, meaning the rotation in relation to the x and y axis are irrelevant in that case, and to simplify the problem, given that if we did not make this choice, each object, instead of having 4 representative numbers to optimize would need 6 (or even more if we choose a more robust orientation representation like Quaternions)

The input of our optimization framework is a vector that contains four values for each asset on the scene. This can be formalized as follows:

$$\mathbf{State_vector} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n], \quad \text{where } \mathbf{a}_i = (x_i, y_i, z_i, \theta_i)$$

with n being the number of assets in the scene, and θ_i the rotation around the z-axis for asset i.

With the input defined, we will introduce the actual penalization function to optimize.

We decided to construct a straightforward penalization function that translates how well the relational and state constraints are respected or not.

Similar to the constraint evaluation framework (this framework was the basis of the constraint evaluation framework) we have a set of functions that, taking as input some hight level parameters to better define the constraint and the representative coordinates and orientations of the involved objects (the values present in the state vector), they can access how well this constraints are accomplished on the scene by calculating a quantity. This time, we do not use a score, but simply a weighted distance to the hypothetical optimal state where the constraint is fully respected.

There is a rundown of the available constraints.

- `"avoid_overlap"`: Calculates a penalty based on the intersection volume of two objects, driving them apart.
- `"min_distance"`: Penalizes the state if the distance between two objects falls below a specified minimum. The penalty increases the further the objects are within this minimum distance.
- `"ensure_grounded"`: Calculates a penalty based on the vertical distance between an object's lowest point and a target ground level.
- `"stay_above_ground"`: Adds a penalty if an object's lowest point is below a minimum height, proportional to the depth of the violation.
- `"ensure_floating"`: Penalizes an object if its lowest point is outside a specified vertical (min/max) height range.
- `"relative_position_local"`: Calculates a penalty based on an object's distance from a target position defined relative to another object's local axes (e.g., ”front”, ”back”).
- `"relative_position_world"`: Calculates a penalty based on an object's distance from a target position defined along a global world axis (X/Y) relative to another object.

- `"on_top_of"`: Penalizes vertical and optional horizontal misalignment from an ideal ”on top” placement of one object on another.
- `"bellow_of"`: Penalizes misalignment when placing one object directly below another, with the cost increasing with the distance from the ideal position.
- `"inside"`: Calculates a penalty based on the extent to which one object's bounding box is not contained within another's.
- `"orient_front_to_world"`: Penalizes angular deviation between an object's ”front” vector and a target world axis.
- `"orient_front_to_target"`: Penalizes angular deviation between an object's ”front” vector and the direction towards (or away from) a target object.
- `"align_front_with_front"`: Calculates a penalty based on how far the ”front” vectors of two objects are from being perfectly parallel or anti-parallel.
- `"align_local_top_with_world_up"`: Penalizes the angular deviation between an object's local ”up” vector and the world's global +Z axis, encouraging the object to stay upright.

Similar to the constraint evaluation framework, we use the LLM to extract the constraints from the scene graph, and adapt them to a structured input format, to then give as input to the framework. Those constraints are translated to be individually calculated using the constraint penalization functions.

Finally, this calculated penalization for each defined constraint is used to define the penalization function to be optimized.

It is defined as:

$$\mathcal{L}(\mathbf{x}) = \sum_{i=1}^{m} w_i \cdot \mathcal{C}_i(\mathbf{x}, \mathbf{p}_i)$$

where:

- $\mathbf{x}$ is the full scene state vector (positions and orientations of all objects),
- m is the number of active constraints,
- $\mathcal{C}_i$ is the penalization function for constraint i,

- $\mathbf{p}_i$ contains the high-level parameters for constraint i (e.g., threshold distances, directions, target objects),
- w_i is the weight assigned to constraint i to control its influence on the total loss.

To the optimization algorithm, we tried a set of non-gradient-based ones that take as input the initial state, and start from that state to search for better configurations that lead to a lower penalization function.

- **Powell** [106]: A derivative-free algorithm that works by performing a series of one-dimensional line searches along a set of directions. After each main step, it intelligently updates these directions to more efficiently explore the search space.
- **Nelder-Mead** [107]: A popular derivative-free method that uses a geometric shape called a simplex (e.g., a triangle in 2D) to explore the solution space. The simplex flips, expands, and shrinks as it approximates towards a minimum.
- **COBYLA** (Constrained Optimization by Linear Approximation) [108]: A derivative-free algorithm specifically designed to handle inequality constraints directly, without needing to convert them into penalties in the objective function. It builds a linear approximation of the problem at each step to decide where to move next.
- **CMA-ES** (Covariance Matrix Adaptation Evolution Strategy) [109]: A evolutionary algorithm. It works with a population of candidate solutions sampled from a probability distribution. It intelligently adapts the shape and orientation of this distribution based on the success of previous candidates.

We tried these different optimization algorithms in our optimization framework, paired with different combinations of weights for different constraints, and hyperparameters related to each optimization algorithm. Unfortunately, we did not reach a setup that improved the results, so we did not include it in the final pipeline.

There is an example of before and after using this optimization framework in figure A.2

In this case, the road went below the ground level, not being visible, and one of the cars moved to the side of where the road should be, instead of being on top of the road.

There are many possible reasons for this framework not being effective:



(a) 3D scene before optimization procedure



(b) 3D scene after optimization procedure

Figure A.2: Example of a before and after application of developed optimization method in a generated 3D scene

- The constraints are not well defined (probably not the leading cause, as for very simple scenarios with one or two objects to test the behavior, we got the expected behavior).
- The combination of weights and optimization algorithm parameters was not well defined for the problem (less likely, as we tested with a lot of different combinations)
- The specific optimization algorithms used. This is probably one of the main factors, as we used only non-gradient-based algorithms, and in the literature, most of the methods used gradient-based ones. We made this choice because it was more straightforward to implement a non-gradient-based framework, namely because the gradient-based approaches need the gradient.
- The fact that we just merged everything into a unique penalization function. This is also probably one of the main reasons, as it makes the problem very complex. Fragmenting the optimization to certain categories, or even doing it to singular constraints, could improve the results.

A.7. Automatic library construction

We constructed a set of functions in the function library to perform object manipulation in the scene.

Using just a set of predefined functions, even if they are for a general use case, and can enable a lot of configurations if used properly, is still limiting. These predefined functions can't realistically reflect every single necessary object manipulation to construct 3D scenes. There is also the fact that even if these functions can be used for most of the configurations, because they are very general and designed for simple operations, finding the correct combinations of these functions that enable a desired result can be very challenging for more complex scenes.

It would be helpful to have a method to add new functions to the function library, if certain functionality is needed to construct a given scene, but it is lacking or difficult to achieve with the current tools.

There is a representation of the automatic library construction pipeline implemented in figure A.3.

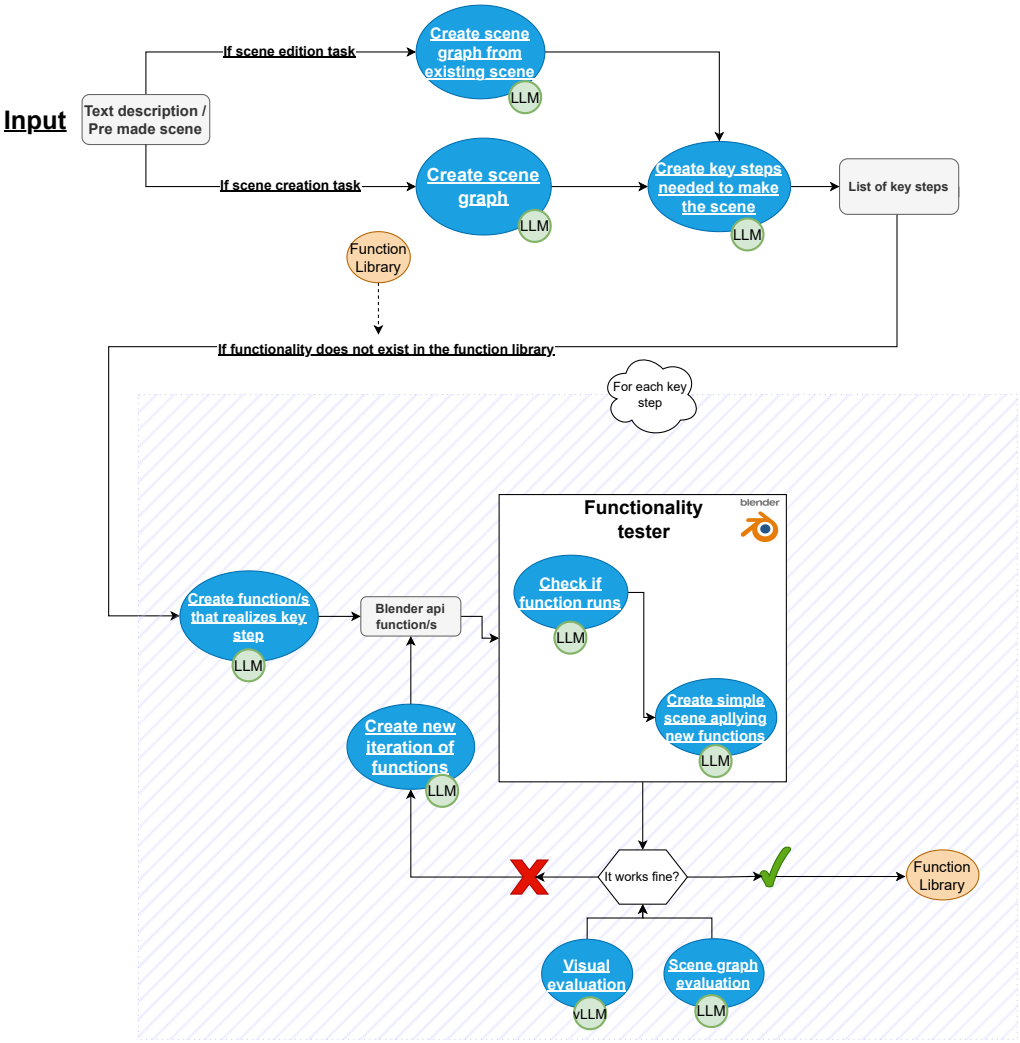


Figure A.3: Automatic library construction pipeline

This pipeline could be used in the context of scene construction or editing. If it were scene editing, the scene graph would be constructed using the scene to use alongside the edition request. If it were scene construction, we would use the description of the scene to construct the scene graph. These steps are executed using a LLM.

After the generation of the scene graph, a list of key steps necessary to accomplish the scene construction, or the scene edition, is defined using an LLM.

After a verification is done on the functionalities available in the function library. This verification is to verify if it has the tools necessary to accomplish each key step. This is made by making an LLM understand what is necessary to do in order to accomplish the key step. Then, having access to the code library, it checks whether that step can be easily accomplished with the functionalities present or not. This way, the key steps that can be accomplished are filtered, and we remain with a list with only the key steps that supposedly can not be accomplished with the function library or are difficult to execute using it (if all the key steps are filtered, then the pipeline ends there).

Then, for each key step remaining, first, an LLM creates a function that can achieve that step. After, this function is the target of the functionality tester, where we have two main mechanisms to test if the function works properly: Run the function in the Blender environment to test if any errors occur. If an error occurs, a new iteration of the functionality is done and goes back to the functionality tester. If no error happens, we use an LLM to construct a simple scene where the functionality is applied by making a visual evaluation and a scene graph evaluation. The scene graph evaluation is made by first extracting the scene graph from the scene, and then an LLM checks if the functionality was well applied. If it is concluded that the functionality is not well applied, the function is remade and goes to the functionality tester again. If it is concluded that the functionality did its intended job, then it is added to the function library.

The main reasons not to include this pipeline in the final method were

- The fact that the pipeline takes a very long time to run.
- The resulting functions were not very useful, and often the new functionality made was already in a similar form in the function library.

The run time being very long is not necessarily a very detrimental problem, as this could be used just in an offline setting. But if we wanted a decent run time, the most likely option

should be to remove the step where the functionality is tested in a simple scene, which is the most time-consuming step.

The other problem is probably because of a combination of a sub optimal prompting of the LLM, the lack of context of the current state of the library in the function formulation steps, where we should gave it the function library as reference of what was already made, and the fact that building general purpose functions for specific functionality allot of the times is not ideal. In some cases, giving the option to make functions to adjust to very specific cases, namely by giving context of the assets to be used, should be beneficial if we also want to make more specific functions.

It would be interesting to explore these changes, as we did not had the time to implement and test them.