香港中文大學
The Chinese University of Hong Kong

Final Year Project

Structure-Based XR Scene Synthesis

Author:

LAM Yiu Fung Anson

(SID: 1155175097)

Supervisor:

Prof. Michael R. LYU

LYU2406

Department of Computer Science and Engineering

Faculty of Engineering

November 27, 2024

Acknowledgement

I would like to express my sincere gratitude to my supervisor, **Professor Michael R. Lyu**, and my advisor, **Ms. Shuqing Li** (one of Prof. Lyu's PhD students), for their invaluable guidance, support, and patience throughout the course of this project. Their expertise and insights greatly contributed to the completion of this project. Without them, I would not have been able to finish this report.

Teaser

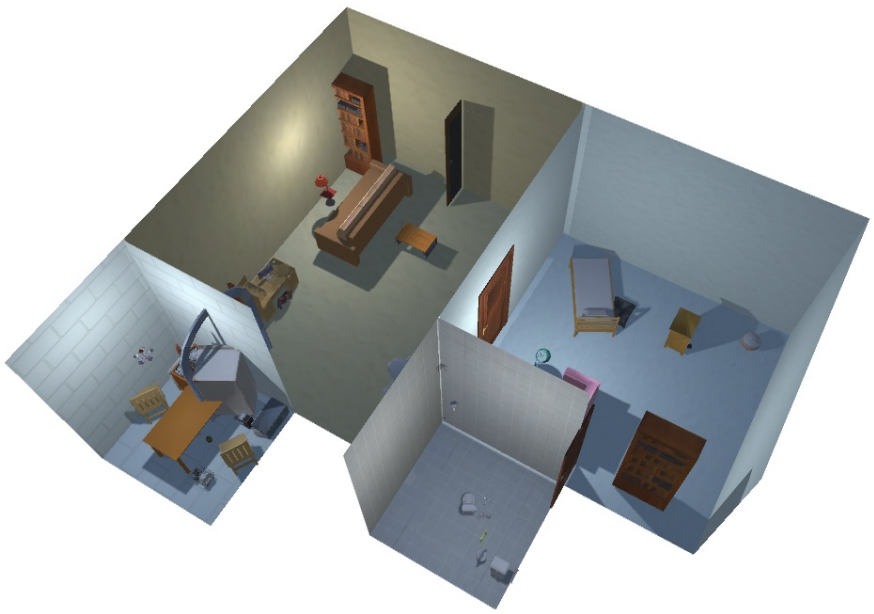


Figure 1: A 2D rendering of a virtual scene generated using our DSL-based pipeline from the prompt: "a 1b1b apartment of a researcher who has a cat."

Abstract

Extended Reality (XR), encompassing Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR), has emerged as a transformative domain with applications spanning entertainment, education, and commerce. A critical challenge in developing and testing XR applications is the creation of diverse and realistic virtual scenes, essential for ensuring robust performance across varied environments. This paper presents a novel Domain-Specific Language (DSL) designed to describe and procedurally generate 3D scenes tailored for XR applications. Our DSL offers expressive constructs to model scene geometry, materials, objects, spatial and temporal constraints, and probabilistic parameters, enabling diverse and physically plausible scene generation. By addressing the limitations of traditional scene graphs, our approach provides human-readable, flexible, and low-level control over scene specifications, ensuring both explainability and adaptability. Extensive experiments validate the efficiency of our DSL and framework in generating stereoscopic 3D XR scenes.

Contents

Figure 2: An illustration of people having a meeting in the Metaverse (from [39]).

1 Introduction

1.1 Background

Whether you like it or not, the term "Extended Reality" (XR), whose fundamental concept surprisingly dates back to as early as the 1800s [61, 8, 9], has been ubiquitous since the past few decades. Some people may however claim that they have never heard of the term, and may wonder: the reality may not be appreciated by everyone, but it is something everyone must deal with, so why on earth do we want to extend it which could potentially make our lives even more complicated? It turns out, XR does not mean adding a few more dimensions to our one and only reality, but is instead an umbrella term for immersive technologies that the general public should be more familiar with: Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR) [44, 9].

Among the three types of immersive technology, VR ought to be the easiest to understand. In short, you immerse yourself in a totally digital world (and hence the name "virtual"). That virtual world is 100% simulated, and is completely isolated from the real world (i.e., whatever you do in the virtual world does not affect the real world, and vice versa) [113, 44]. Thanks to the extraordinary capabilities of computer simulation, anybody can perform countless actions in virtual world that literally no one (yet) can perform in reality. For instance, you can slash rapidly approaching blocks using two glowing sabers[1], or swing through the sky of a huge city using spider webs[2]. More recently, the notion of Metaverse, which is like an online universe in which users represented by avatars can meet one another in virtual spaces decoupled from the real world (see Figure 2) [98, 39], is becoming more and more well-known, and has caught the eyes of numerous big companies [49].

On the contrary, AR and MR are not totally distinctive. Both of them aim to combine the real world and the virtual world, and they allow users to "place" virtual objects into the virtual world [8, 44]. However, in MR, things happening in the physical world can affect the virtual world (but currently not vice versa), meaning users can interact with virtual objects in a way almost identical to how they would normally interact with real objects. This is something AR cannot do [44]. In other words, AR simply overlays digital content onto the device displaying the combined world [44, 113],

[1]Beat Saber: https://beatsaber.com

[2]Spider-Man: Far From Home Virtual Reality: https://www.meta.com/experiences/pcvr/spider-man-far-from-home-virtual-reality/2190323657748572/

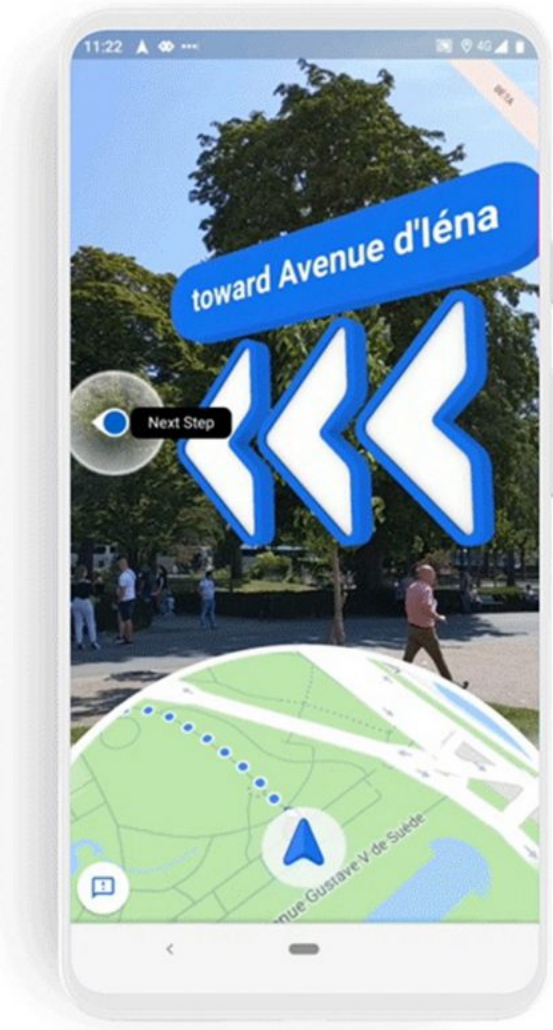


Figure 3: A runtime example of Google Maps' Live View, in which a virtual arrow points to the next (intermediate) destination (from [41]).

yielding a lower interactability compared to MR.

Typical examples of AR include Pokémon GO[3], Google Maps' Live View (see Figure 3), and IKEA Place (see Figure 4). As for MR, one of the most popular headsets is Meta Quest[4] (indeed, Apple Vision Pro[5] is also quite popular, but it seems that Apple does not want their product to be described as MR [3]). By the way, just for fun, if you are a fan of the

[3]Pokémon GO: https://pokemongolive.com/?hl=en
[4]Meta Quest: https://www.meta.com/quest/
[5]Apple Vision Pro: https://www.apple.com/apple-vision-pro/

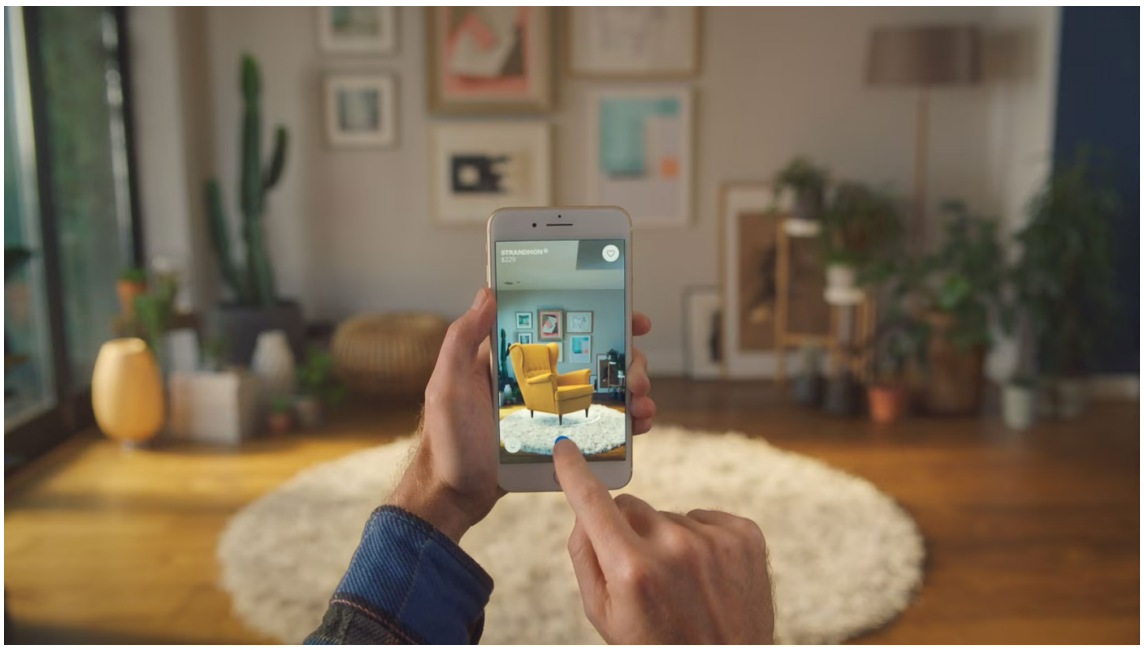


Figure 4: A runtime example of IKEA Place, in which a virtual chair is overlaid onto a real rug (from [38]).

Figure 5: In one of the scenes in the movie *Iron Man 2* (produced by Marvel Studios), Tony Stark (the male character on the right) creates and throws a virtual ball while talking to the female character.

Marvel Cinematic Universe, you may recall that in one of the scenes in *Iron Man 2*, Tony Stark (the male protagonist) throws a virtual ball while talking to another person in the physical world (see Figure 5). This is indeed an example that demonstrates the interactability of MR (though under the settings of the movie, all virtual objects are projected holographically, and so no headset is used).

1.2 Purpose of Automatic Virtual Scene Generation

Up to this point, you may already have a clear picture of what VR, AR, and MR are. Naturally, the next question would be: why do we need automatic virtual scene generation?

For VR, one may argue that for users to completely immerse themselves in a virtual world, some sort of virtual scene is undeniably required, and a virtual scene will not suddenly appear from nowhere unless someone actually creates (i.e., generates) it, which in turn desires automation to speed up the generation process. This is true. However, the use of fully simulated environment is also the exact reason why we should not use automatic scene generation. The virtual scene is not just viewed by developers, but is in fact targeted at end users. Undoubtedly, the quality of the final virtual scene would determine a large proportion of the users' impressions and ratings towards the entire software. Under this circumstance, human-oriented scene generation is the only way to ensure product quality, unless one day there exists an artificial general intelligence (AGI) that is capable to learn, reason, design, and reflect like a real human being.

What about AR and MR? Their functionalities are based on the physical world, and they are mainly focused on merging virtual objects (not virtual scenes) to the real world. So we do not even need a scene, right?

Not really.

The actual problem lies in software testing. The importance of software

testing has been studied and emphasized by (metaphorically) countless pieces of academic work (including but not limited to [72, 2, 120, 55, 45, 119, 50]). Given that about 1 billion people around the globe have experienced mobile AR (MAR) applications in 2023 [112], and the worldwide revenue in the AR and VR market is forecasted to be about US$62.0 billion in 2029 [111], it is crystal clear that ensuring and maintaining the quality of XR software is of utmost importance to the world economy. In this sense, it is completely reasonable to argue that XR software should also be thoroughly tested. Nonetheless, this may not be as straightforward as testing non-XR software.

Take MAR application as an example. Traditionally, when a group of developers want to test their MAR application, they have to (in a much simplified manner) (i) design and construct at least one testing environment in the physical world that fits their requirements (and the needs of target audience), (ii) find multiple testers to explore and test all functions in the application within the environment, and finally (iii) manually analyze the video recordings and other collected data to determine if the application complies with some pre-defined guidelines [69, 118, 76, 7, 90]. By simply imagining the above-mentioned pipeline, you will most likely immediately come to the conclusion that it is highly labor-intensive and time-consuming. Wouldn't it be much better if all three stages (or at least any one of them) can be automatically performed? This is exactly what this project aims at achieving.

The first half of the whole project (which is the main topic of this re-

port) is focused on automatically designing and constructing diverse virtual scenes (based on any desired scenario), in which some kind of automated virtual agents can travel around, that can be used to substitute the physical environment when testing XR (primarily AR) applications. The second half will be about automatically analyzing and evaluating runtime recordings of XR applications that are tested in virtual scenes generated by our method.

1.3 Motivation

Section 1.2 discusses, from the perspective of XR testing, one of the reasons why someone may want to work on the topic of automatic virtual scene generation. Now, let's discuss why **we** decided to contribute to this topic.

You may already be able to guess by now that we are not the first one to propose new method on automatic virtual scene generation (Section 2.1.1). Yet, they have some general drawbacks that render them unsuitable for XR-oriented virtual scene generation. The biggest issue is the dominent use of scene graph (Section 2.1.1.2) as the underlying scene-representing structure. To put it simply, each node and each edge in a scene graph can only represent one predefined object class and one predefined relation respectively. The heavy reliance on predefined sets greatly restrict the diversity and preciseness of generated scenes. In the context of XR testing, developers more often than not want to test their target XR application under as many diverse environments as possible, so as to assure users that they can normally use the application no matter where they are and how complex the environment is (of course under the assumption that the

environment is reasonably lit). Besides, since a scene graph is usually realized in the form of a matrix (which typically stores high-dimensional non-human-understandable values), if there are some minor physical details that one would like to modify after the scene is generated, what should that person do? It is arguably unwise (and in fact uneconomical) to generate everything from scratch again. In other words, scene graph fails to provide users with low-level controllability of the virtual scene, which is a key to allowing developers to fine-tune small details in the scene in the hope of fulfilling their requirements.

All in all, scene graph is not the best option for representing an XR-oriented virtual scene. But we still need something to represent a generated scene, right? Otherwise, everything would be locked inside a blackbox, making such method non-transparent and hence not practically trust-worthy. Though, we do not simply want any "something". We want a structure that is:

1. **human-understandable** (which boosts explainability and controllability);
2. **unambiguous** (i.e., all possible configurations in the scene can be clearly formulated);
3. **flexible** (i.e., there exists more than one solution for diversity, if applicable); and
4. **universal** (i.e., all physically possible environments, or at least the common ones, can be modeled).

This is where our proposed domain-specific language (DSL) (Section 3.1) comes into play.

1.4 Summary of the Proposed Method

We propose a novel Domain-Specific Language (DSL) for 3D scene description and generation, designed to address the challenges of creating diverse, realistic, and physically plausible virtual environments for XR applications. The DSL provides expressive constructs for defining scene geometry, materials, objects, spatial and temporal constraints, and probabilistic parameters. Unlike traditional scene graphs, our approach emphasizes human-readability, flexibility, and low-level control, enabling precise customization and procedural generation of scenes.

In short, our main **contributions** include:

1. The design and implementation of a human-readable, expressive DSL for XR-oriented 3D scene description and generation.
2. A procedural generation engine that supports diverse and physically consistent scenes through probabilistic and constraint-based modeling.

2 Related Work

2.1 3D Scene Generation

While the topic of 3D scene generation seems to be emerging recently partially because of the rise in popularity of robot training in virtual environment [105, 74] and XR applications, 3D scene generation in fact has a very long history, with some of the earliest works [106, 117, 71, 14] even attempting to combine scene generation with natural language processing (NLP) (of course they did not have access to any large language model (LLM) back then). We emphasize that there exist hundreds if not thousands of academic papers regarding this broad topic, making it basically impractical (and honestly unnecessary) to list them all in this report. So, we try our best to include and briefly discuss relevant works from only the past 10 years.

2.1.1 Playable Scene

At first glance, you may think that the word "playable" implies some kind of game. But it is not. In the physical world all of us are living in, when we see an object, we can (most likely) pick it up, move it somewhere else, and put it down. This configurability in real time is what makes a virtual scene (in the context of this report) playable. In other words, runtime interactability can be achieved.

2.1.1.1 Probabilistic Generation

This group of works (including but not limited to [13, 15, 102, 51, 32, 73, 146]) contribute to an earlier (but still modern) phase of virtual scene generation. The fundamental concept is to generalize the distribution of objects (and perhaps their relations and positions) in all training scenes using some probabilistic model. Then, we can utilize the knowledge learned for random sampling during inference. For example, SceneSeer [15] parses a text prompt using some fixed grammar, and computes the most probable scene template under that prompt. Because of the use of a probabilistic model, the number of classes of 3D objects these methods can handle is limited, which in turn harms the diversity of generated scenes. For XR testing, the diversity of testing scenes is very crucial since only then can developers be assured that their product can function properly under a broad spectrum of scenarios.

2.1.1.2 Deep Generation

The word "deep" refers to the use of deep learning strategies. This group of works is by far (to the best of our knowledge) the most prevalent in the field of interactable 3D scene generation. Generally speaking, they are all trying to learn and represent a 3D scene using some underlying structure (e.g., matrix or scene graph [52, 60]). The major difference is how they learn — architectures such as Convolutional Neural Network [123, 97, 122, 133], Encoder-Decoder [65, 24, 131, 17, 35, 129, 126], Generative Adversarial

Network [5, 66], Transformer [124, 88, 82, 125, 70, 148, 137], and Diffusion [68, 150, 142, 115, 143, 134] have been extensively utilized and studied (other methods include [147, 151]). Once the underlying structure is established (by learning from a lot of manually crafted scenes, with one of the most popular datasets being 3D-FRONT [31]), it can be conditioned on different types of input for downstream tasks. For instance, ATISS [88] can be conditioned on floor layout for scene generation, scene completion, and object suggestion; InstructScene [68] can be conditioned on text for scene generation, scene stylization, scene re-arrangement, and scene completion; EchoScene [142] can be conditioned on scene graph for scene generation.

Scene Graph A scene graph, just like any other graphs, consists of nodes and edges. Each node encodes an object in the scene, while each edge encodes the relation between the two connected objects. Even though it sounds like an intuitive and naturally suitable structure for virtual scene generation, there are a few problems that make it a suboptimal structure for scene generation when it comes to XR testing. First of all, a typical scene graph can only encode one discrete number $c \in \{1, ..., K_c\}$ for each node as its object class, where K_c is the predefined number of object classes in the training set (hence making the method scene-specific). This means that the number of object classes you can possibly get is bounded, severely affecting the diversity of all virtual scenes that could be generated. And if you want to add a new object class, you will have to re-train (or at least do some fine-tuning) to your existing model. This is not something desired in XR testing, since we want to be able to handle any type of scene with

any class of object using one single pipeline, making it a truly practical method. The same **categorical** problem applies to edges, as each edge can only encode one discrete number $e \in \{1, ..., K_e\}$ for its relation class, where K_e is the predefined number of relation classes. Such restriction on edges can potentially damage the expressiveness of the scene graph due to the fact that (1) there may exist some relations that cannot be accurately described by one of K_e relation classes, or (2) even if there exists a perfect match, the relation cannot be precisely controlled. The use of such edge also gives rise to one more problem: the assumption that there can only be at most 1 relation between 2 objects. This assumption eliminates the possibility of using multiple relations to comprehensively and unambiguously describe how two objects spatially influence each other, which in turn makes it more challenging to finely adjust a generated scene (for XR testing) upon user request. With that said though, we do believe that having an underlying structure is critical to ensuring that our method is interpretable and controllable [130]. Hence, one of our contributions is a new **domain-specific language (DSL)** (Section 3.1) that can be used to substitute scene graph, reducing ambiguity when describing a virtual scene while maintaining explainability and controllability.

2.1.1.3 View-based Generation

This group of works (including but not limited to [46, 83, 132, 16, 21]), while could also be utilizing deep learning methods, is more focused on generating a portion of a scene (from an angle). They typically take an RGB image as

input, and try to figure out the best way to put several objects visible in the image into a scene such that when rendered from a certain angle, the scene looks exactly like the original image. For example, Total3DUnderstanding [83] detects 2D bounding boxes and 3D objects from an RGB image, and estimates tha layout of the scene captured by the image. On one hand, this kind of method does make it easier for people to more accurately re-create their surroundings. But on the other hand, it also lacks the capability to generate scenes that people cannot usually see or have access to. As for XR testing, since we want to create a variety of virtual scenes without the need to construct a real one, it would be somehow illogical to seek for a physical environment in the first place for view-based generation.

2.1.1.4 LLM-based Generation

As Generative Pre-Trained Transformer (GPT) [93, 94] quickly gains its enormous popularity, lots of research have been conducted to show that its potential is beyond our imagination. Thanks to its unparalleled capability in understanding natural human languages, researchers from different fields are giving their best effort to integrate GPT-based Large Language Models (LLMs) in their projects. Naturally, researchers working on automatic virtual scene generation have also done so. This group of works (including but not limited to [28, 136, 34, 33, 1, 11, 85, 135]) aims at exploiting the vast knowledge "learned" by an LLM (particularly knowledge regarding scene design) and prompting it to lead (or at least guide) the entire generation process.

As an example, one of our competitors, Holodeck [136], first utilizes GPT-4 [86] to (1) generate a floor plan (with coordinates and materials), (2) generate connections (i.e., doors) and window positions, (3) generate the attributes of target objects, and (4) generate the constraints that all objects need to obey. Then, it employs a depth-first search solver to solve the constraints to output each object's optimal position and rotation, producing the final scene (out of many possibilities). While their results look promising, the concept of scene graph is still being applied during constraint generation and solving, which again is suboptimal in our case. In this project, we implement a similar module-by-module pipeline, but we also inject our DSL into the prompts.

2.1.1.5 Procedural Generation

This group of works (including but not limited to [23, 95]) does not (primarily) use Artificial Intelligence for scene generation. Instead, they achieve automated creation of virtual environments based on a set of predefined (mathematical) rules or algorithms. While they can indeed generate a set of diverse and visually pleasing virtual scenes in a relatively short amount of time, the existence of rules implies some sort of limitation, and limitation reduces flexibility (and in some sense, creativity). In XR testing, we sometimes want a high level of flexibility to a point where edge cases can be produced in order to evaluate the stability and adaptability of the target XR application. For instance, ProcTHOR [23] (which has already been integrated into AI2-THOR [59]) employs specialized algorithms to

first cut indoor boundaries (i.e., create rooms), and then iteratively place (and rotate) retrieved 3D models with the goal of making the final scene physically possible and semantically meaningful.

2.1.2 Non-Playable Scene

Just like how a coin has two sides, apart from playable scene generation, there is of course non-playable scene generation. This line of methods can undeniably generate high-quality and plausible contents, but the generated scenes are either (1) a single unified mesh (imagine every single object in front of you fuses with whatever it is touching, and the resulting objects in turn fuse with other objects, ultimately forming one unseparable mesh), or (2) not an environment in which you can freely move around (i.e., no mesh is generated). While these properties are suboptimal for XR testing because it is desirable to be able to actually interact with our scenes (i.e., moving an object using a virtual hand in real time) such that the target application can be tested in a more physically accurate and realistic environment, we still believe that this line of works has its own values and thus is worth sharing.

2.1.2.1 Depth-based Paranoma-like Generation

From a high-level and general perspective (which may not be 100% correct, but it makes it easier for one to grasp the basic concepts), this group of works (including but not limited to [84, 114, 107, 43, 110, 144, 58, 116, 26, 18, 104, 77, 87, 25, 149, 75, 138, 108, 64]) is trying to (1) fix a camera in some 3D position pointing at some direction, (2) estimate or generate the depth

map of the portion of the scene visible from the camera, (3) generate the RGB values of what the camera should see based on the previously obtained depth map, and finally (4) fix the camera at another position (pointing at another direction) and repeat the whole process. They mainly differ in (1) how they find a suitable position and direction to initialize and move the camera, (2) what method they use to obtain a depth map, (3) how they ensure that depth maps from different camera settings can actually align with one another such that the final mesh can be as watertight as possible, and (4) what method they use to generate RGB values. Their generated scenes are described as "paranoma-like" because they usually position the camera near the center of the scene and make it orbit the center while constantly rotating to point outward, forming something you would usually see in Google Maps' Street View. Because these methods essentially generate 2D images iteratively, there is no separation among physical structures, producing one unifed mesh at the end.

For example, PerspectiveNet [84] takes a sparse set of reference RGB-D room views as input, and uses a denoising autoencoder to generate the RGB-D values of the remaining scene. Text2Room [43] takes a text prompt as input, and utilizes Stable Diffusion [100] for RGB generation and Iron Depth [4] for depth generation to produce the final textured 3D mesh. FastScene [75] also takes a text prompt as input, and utilizes Diffusion360 [27] for panorama generation and EGformer [141] for depth estimation to produce an intermediate point cloud.

2.1.2.2 View-dependent Volume Rendering

Unlike the above depth-based paranoma-like generation methods, this group of works (including but not limited to [20, 91, 6, 145, 140]) begins with some basic and simple 3D representation (such as a 3D bounding box). Then, some method is employed to generate the required scene on some relatively complex yet informative 3D representation. Finally, some Computer Graphics technique such as ray casting is used to render the scene from a certain angle. For instance, Set-the-Scene [20] takes a text prompt and some 3D bounding boxes as inputs, and generates the final scene in neural radiance fields (NeRF) [79]. DreamScape [140] also takes a text prompt as input, converts it to some useful 3D information using LLM, and generates the final scene using 3D Gaussian Splatting [54].

2.1.2.3 Full Volume Denoising

To the best of our knowledge, this group of works (including but not limited to [53, 127, 62, 78]) only appeared very recently. Instead of applying diffusion-based models [42, 109] to 2D image generation [100] or object-level 3D generation [128], they directly apply the notion of diffusion (and denoising) to large-scale scene-level 3D generation. After determining the underlying 3D representation, they iteratively denoise a certain volume of 3D space to generate a detailed yet unified mesh. For instance, DiffInDScene [53] and LT3SD [78] use truncated (un)signed distance function as the 3D representation, while BlockFusion [127] and SemCity [62] use tri-plane

[89, 12]. Thanks to the design of block-by-block diffusion, BlockFusion and LT3SD can in fact (theoretically) generate an infinitely large scene.

2.1.2.4 Motion-simulating Generation

Compared to the above 3 types of methods, this group of works (including but not limited to [10, 30, 139, 63, 67]) is quite different in the sense that they are more like video generation. Imagine you are sitting on an office chair. Take a look the view in front of you, remember it, and then close your eyes. Now, if someone pulls your chair from behind, how would you expect the view in front of you changes? This is the fundamental concept of these papers. Given at least one input image (and perhaps a text prompt), they inject camera movements to the input image(s) to perform 4D (time dimension in addition to 3D) exploration. While, to a certain extent, what they produce can be considered as a scene, since there is not really any object that can be isolated from the scene, and you cannot maneuver in the scene, they are suboptimal to XR testing.

2.2 Domain-Specific Language (DSL)

Domain-Specific Languages (DSLs) have garnered significant attention as tools for enhancing productivity, maintainability, and expressiveness in software engineering. Research has demonstrated their utility across various domains, including programming languages, software engineering, and security, by offering tailored abstractions and mechanisms to address specific challenges.

Negm et al. [81] propose a semantic-based approach for developing DSLs, leveraging ontologies for domain representation and reasoning. This methodology emphasizes a systematic integration of domain knowledge, improving the development and usability of DSLs for business and technical applications. Similarly, Cisternini et al. [19] introduce GeoDSL, designed for Internet measurements, enabling non-expert users to manage and analyze measurement data effectively. This research underscores the role of DSLs in lowering barriers for domain-specific tasks.

In the security domain, Kim et al. [57] present PoE, a DSL tailored for exploit development. By standardizing exploit construction, PoE improves reusability and collaboration among security researchers. Zhu et al. [152] explore architectural concerns with C-Saw, an embedded DSL for reconfigurable, distributed software systems. Their work highlights the benefits of separating architectural specifications from application logic, improving clarity and reuse.

The health and space exploration domains also benefit from DSL innovations. Rojco et al. [99] propose PHM4HHP, a DSL for integrating heterogeneous models and data in health support systems, particularly for crewed space missions. Forbrig et al. [29] develop a DSL for humanoid robot behavior, incorporating sensor data to enhance task modeling and execution.

In the field of machine learning, Giner-Miguelez et al. [36] introduce a DSL for describing machine learning datasets. This DSL addresses dataset

structure, provenance, and social concerns, fostering standardization and quality improvement. Similarly, Vuković et al. [121] focus on fluent APIs, using a DSL to automate graphical representation and code generation, streamlining API development.

For IoT applications, Salman et al. [101] discuss the challenges and opportunities of DSLs in simplifying the integration of IoT capabilities. In a related context, Kharisma et al. [56] propose a DSML for REST-compliant services, enhancing abstraction and usability for service design.

Several studies investigate DSLs for specific computational problems. Nagashima [80] designs a DSL for encoding induction heuristics in theorem proving, and Gleißner [37] demonstrates how DSLs can optimize numerical solutions for ordinary differential equations. Furthermore, Gupta et al. [40] explore systematic methodologies for industrial DSL development, emphasizing ease of use and efficient modeling.

These works collectively highlight the diverse applications and advantages of DSLs in addressing domain-specific challenges. They underline the importance of abstraction, customization, and tool support in advancing domain-specific software engineering practices.

3 Methodology

This section discusses our proposed DSL ScenethesisLang (Section 3.1) and our proposed method, Scenethesis, for scene generation (Section 3.2). From a high-level point of view, our DSL is carefully designed to unambiguously describe any scene, and Scenethesis incorporates this language to assist the second half of the entire generation pipeline.

3.1 Domain-Specific Language for Structured Scene Synthesis (ScenethesisLang)

The proposed Domain-Specific Language (DSL) is designed to describe and generate diverse, realistic, and physically plausible 3D scenes for XR applications, with a focus on expressiveness, flexibility, and usability. This subsection details the language's syntax, semantics, and operational principles, highlighting how it facilitates the modeling and procedural generation of 3D environments.

3.1.1 Language Design Goals

The design of the DSL was guided by the following objectives:

- **Expressiveness:** The DSL must capture diverse aspects of a 3D scene, including geometry, materials, objects, and constraints.
- **Human-Readability:** Developers should find the DSL intuitive, enabling seamless writing, understanding, and modification of scene specifications.

- **Generative Capability:** The DSL must support probabilistic parameters for procedural generation and the creation of diverse testing scenarios.
- **Physical Plausibility:** The DSL should enforce constraints to ensure that generated scenes are realistic and adhere to the principles of physics.

3.1.2 Core Abstractions

The DSL models 3D scenes using a hierarchy of abstractions, enabling a modular and systematic specification of environments. These abstractions are described below.

3.1.2.1 Scenes

A *scene* represents the overall 3D environment and is defined by its type (*indoor*, *outdoor*, or *mixed*), metadata description, and components such as regions, connections, constraints, probabilistic parameters, and temporal requirements. The formal definition is:

$scene \in$ *Scenes* ::= **SceneType:** $scene\_type$;

SceneDescription: $scene\_description$;

Regions: $regions$;

Connections: $connections$;

Constraints: $constraints$;

ProbabilisticParameters: $probabilistic\_parameters$;

TemporalRequirements: $temporal\_requirements$

Scene type (*scene_type*) specifies the overall nature of the environment (e.g., indoor or outdoor), while **scene description** (*scene_description*) captures descriptive metadata.

3.1.2.2 Regions

A *region* represents a spatial subdivision within a scene, such as a room or outdoor area. Each region is characterized by its geometry (*shape*), materials, construction details, contained objects, visibility conditions, and constraints:

$$region \in \textit{Region} ::= id \leftarrow \textbf{region(}description, shape, materials, construction, objects, visibility, constraints\textbf{)}$$

The **shape** of a region can be specified using geometric primitives such as cuboids, ellipsoids, and meshes, or their combinations using union ($\cup$),

intersection ($\cap$), or subtraction ($-$) operations. For example:

$$shape \in \textit{Shape} ::= cuboid \mid ellipsoid \mid mesh \mid shape \cup shape \mid shape \cap shape \mid shape - shape$$

3.1.2.3 Materials

Materials define the physical and visual properties of a region's surfaces. These include color, texture, reflectivity, and opacity:

$$floor\_material \in \textit{Material} ::= \textbf{material(}color, texture, reflectivity, opacity\textbf{)}$$

Separate materials can be defined for floors and non-floor surfaces, allowing detailed customization of a region's appearance.

3.1.2.4 Objects

Objects are the entities within a region and are defined by their category, physical dimensions, position, orientation, and visibility:

$$object \in \textit{Object} ::= id \leftarrow \textbf{object(}category, description, dimensions, position, rotation, bounding\_box, probabilistic\_properties, visibility\textbf{)}$$

The **position** of an object can be specified absolutely or relative to another object. Probabilistic parameters enable dynamic placement during gener-

ation, such as using a uniform or Gaussian distribution to define position offsets.

3.1.2.5 Connections

Connections represent spatial relationships between regions, such as doorways, windows, or open passages. Each connection specifies the connected regions, type (e.g., single door, window), and optional descriptive metadata:

$$connection \in \textit{Connection} ::= \textbf{connection(}region1, region2, type, description\textbf{)}$$

3.1.2.6 Constraints

Constraints enforce physical plausibility and domain-specific requirements. These include spatial conditions (e.g., alignment, containment), visibility conditions (e.g., line-of-sight), and temporal conditions (e.g., event sequencing):

$$constraint \in \textit{Constraint} ::= spatial\_condition \mid probabilistic\_condition \mid temporal\_condition$$

For example, a spatial condition might ensure that an object is always inside a specific region, while a probabilistic condition could control the likelihood of a certain configuration.

3.1.3 Probabilistic Scene Generation

A distinguishing feature of the DSL is its support for probabilistic parameters. These parameters allow developers to specify ranges or distributions for attributes such as object positions, dimensions, and material properties. During generation, the DSL engine samples these distributions to produce diverse scenes.

$probabilistic\_condition \in$ *ProbabilisticCondition* ::=

probability(p): condition |

distributionBased(object, param: distribution)

For instance, the position of a chair within a room can be defined using a Gaussian distribution centered at the room's midpoint, allowing slight variations between generated scenes.

3.1.4 Temporal Requirements

Temporal requirements are crucial for dynamic scenes where interactions occur over time. The DSL provides constructs to define temporal logic conditions, such as requiring a specific object configuration to exist during a time interval or ensuring sequential constraints between actions:

$temporal\_requirements \in$ *TemporalRequirements* ::=

always(condition) | **eventually(condition)** | **until(condition1, condition2)**

$$
\begin{aligned}
scene \in \textit{Scenes} ::= & \textbf{SceneType: } scene\_type; \\
& \textbf{SceneDescription: } scene\_description; \\
& \textbf{Regions: } regions; \\
& \textbf{Connections: } connections; \\
& \textbf{Constraints: } constraints; \\
& \textbf{ProbabilisticParameters: } probabilistic\_parameters; \\
& \textbf{TemporalRequirements: } temporal\_requirements
\end{aligned}
$$

$$
\begin{aligned}
scene\_type \in \textit{SceneType} &::= \textbf{indoor} \mid \textbf{outdoor} \mid \textbf{mixed} \\
scene\_description \in \textit{Description} &::= text
\end{aligned}
$$

$$
\begin{aligned}
regions \in \textit{Regions} &::= region \mid region;\ regions \\
region \in \textit{Region} &::= id \leftarrow \textbf{region(}description, shape, materials, \\
&\qquad construction, objects, visibility, constraints\textbf{)} \\
shape \in \textit{Shape} &::= cuboid \mid ellipsoid \mid mesh \mid \\
&\qquad shape \cup shape \mid shape \cap shape \mid shape - shape \\
cuboid \in \textit{Cuboid} &::= \textbf{cuboid(}corner.min, corner.max\textbf{)} \\
ellipsoid \in \textit{Ellipsoid} &::= \textbf{ellipsoid(}center, a, b, c\textbf{)} \\
mesh \in \textit{Mesh} &::= \textbf{mesh(}vertices, faces\textbf{)} \\
corner \in \textit{Corner} &::= \textbf{min} \mid \textbf{max} \\
materials \in \textit{Materials} &::= floor\_material;\ non\_floor\_material \\
floor\_material \in \textit{Material} &::= \textbf{material(}color, texture, reflectivity, opacity\textbf{)} \\
non\_floor\_material \in \textit{Material} &::= \textbf{material(}color, texture, reflectivity, opacity\textbf{)}
\end{aligned}
$$

$$
\begin{aligned}
construction \in \textit{Construction} &::= vertices, faces \\
vertices \in \textit{Vertices} &::= \textbf{[}(x, y, z), ...\textbf{]} \mid \textbf{[}(x, y), ...\textbf{]} \\
faces \in \textit{Faces} &::= \textbf{[}(index1, index2, index3), ...\textbf{]}
\end{aligned}
$$

Figure 6: ScenethesisLang for 3D Scene Description and Generation (more in Figures 7 and 8).

3.2 Scenethesis: Structure-Driven Scene Synthesis Framework

As shown in Figure 9, Scenethesis employs a systematic pipeline to fully automatically generate a playable scene from a single user prompt, invoking a series of interdependent modules: (1) Scene Analysis Module

$$
\begin{aligned}
\textit{objects} \in \textit{Objects} &::= \textit{object} \mid \textit{object};\ \textit{objects} \\
\textit{object} \in \textit{Object} &::= \textit{id} \leftarrow \textbf{object(}\textit{category}, \textit{description}, \textit{dimensions}, \\
&\quad \textit{position}, \textit{rotation}, \textit{bounding\_box}, \\
&\quad \textit{probabilistic\_properties}, \textit{visibility}\textbf{)} \\
\textit{category} \in \textit{Categories} &::= \textbf{text} \\
\textit{description} \in \textit{ObjectDescription} &::= \textit{text} \\
\textit{dimensions} \in \textit{Dimensions} &::= \textbf{(length, width, height)} \\
\textit{position} \in \textit{Position} &::= \textbf{(x, y, z)} \mid \textbf{offset(position, vector)} \mid \\
&\quad \textbf{relativeTo(reference)} \\
\textit{rotation} \in \textit{Rotation} &::= \textbf{(roll, pitch, yaw)} \mid \textbf{relativeTo(reference)} \\
\textit{bounding\_box} \in \textit{BoundingBox} &::= \textbf{bounding\_box(}\textit{min}, \textit{max}\textbf{)} \\
\textit{visibility} \in \textit{Visibility} &::= \textbf{rayTrace(}\textit{density}, \textit{occlusion}\textbf{)} \mid \\
&\quad \textbf{regionVisibility(}\textit{region}, \textit{conditions}\textbf{)}
\end{aligned}
$$

$$
\begin{aligned}
\textit{connections} \in \textit{Connections} &::= \textit{connection} \mid \textit{connection};\ \textit{connections} \\
\textit{connection} \in \textit{Connection} &::= \textbf{connection(}\textit{region1}, \textit{region2}, \textit{type}, \\
&\quad \textit{description}\textbf{)} \\
\textit{type} \in \textit{ConnectionTypes} &::= \textbf{open} \mid \textbf{single\_doorframe} \mid \textbf{double\_doorframe} \mid \\
&\quad \textbf{single\_door} \mid \textbf{double\_door} \mid \textbf{window} \\
\textit{description} \in \textit{ConnectionDescription} &::= \textit{text} \\
\textit{constraints} \in \textit{Constraints} &::= \textit{constraint} \mid \textit{constraint};\ \textit{constraints} \\
\textit{constraint} \in \textit{Constraint} &::= \textit{spatial\_condition} \mid \textit{probabilistic\_condition} \mid \\
&\quad \textit{temporal\_condition} \mid \textit{object\_relation} \mid \\
&\quad \textit{visibility\_condition} \mid \textit{physical\_constraint} \mid \\
&\quad \textit{user\_defined}\textbf{(}\textit{logic}, \textit{priority}\textbf{)}
\end{aligned}
$$

$$
\begin{aligned}
\textit{spatial\_condition} \in \textit{SpatialCondition} &::= \textbf{inside(region)} \mid \textbf{outside(region)} \mid \\
&\quad \textbf{above(object, height)} \mid \\
&\quad \textbf{below(object, height)} \mid \\
&\quad \textbf{nearby(object, distance)} \mid \\
&\quad \textbf{alignedWith(object, axis)} \mid \\
&\quad \textbf{tangentTo(surface)} \mid \\
&\quad \textbf{distanceBetween(object1, object2) == d} \\
\textit{probabilistic\_condition} \in \textit{ProbabilisticCondition} &::= \textbf{probability(p):} \\
&\quad \textbf{condition} \mid \\
&\quad \textbf{distributionBased(object,} \\
&\qquad \textbf{param: distribution)} \mid \\
&\quad \textbf{resample(object, param)}
\end{aligned}
$$

Figure 7: Continuation of Figure 6.

$$
\begin{aligned}
\textit{temporal\_condition} \in \textit{TemporalCondition} ::= & \textbf{always(condition)} \mid \\
& \textbf{eventually(condition)} \mid \\
& \textbf{until(condition1, condition2)} \mid \\
& \textbf{next(condition)} \mid \\
& \textbf{during(interval, condition)} \\
\textit{visibility\_condition} \in \textit{VisibilityCondition} ::= & \textbf{canSee(observer, target)} \mid \\
& \textbf{occludes(object1, object2)} \mid \\
& \textbf{visibleInRegion(observer, region)} \mid \\
& \textbf{rayTraceValid(observer, target, density)} \\
\textit{physical\_constraint} \in \textit{PhysicalConstraint} ::= & \textbf{noCollision(object1, object2)} \mid \\
& \textbf{collisionFreeRegion(region)} \mid \\
& \textbf{stablePosition(object)} \mid \\
& \textbf{gravityAligned(object)} \\
\textit{user\_defined} \in \textit{UserDefinedConstraint} ::= & \textbf{customLogic(logicExpression, priority)} \\
\textit{temporal\_requirements} \in \textit{TemporalRequirements} ::= & \textbf{always(condition)} \mid \\
& \textbf{eventually(condition)} \mid \\
& \textbf{until(condition1, condition2)} \mid \\
& \textbf{next(condition)}
\end{aligned}
$$

$$
\begin{aligned}
\textit{object\_relation} \in \textit{ObjectRelation} ::= & \textbf{relation(object1, object2, relation\_type)} \\
\textit{relation\_type} \in \textit{RelationTypes} ::= & \textbf{above} \mid \textbf{below} \mid \textbf{inside} \mid \textbf{outside} \mid \textbf{nearby} \mid \textbf{aligned} \mid \\
& \textbf{occludes} \mid \textbf{intersects}
\end{aligned}
$$

$$
\begin{aligned}
\textit{lights} \in \textit{Lights} ::= & \textit{light} \mid \textit{light};\ \textit{lights} \\
\textit{light} \in \textit{Light} ::= & \textit{id} \leftarrow \textbf{light}(\textit{category}, \textit{description}, \textit{intensity}, \textit{position}, \textit{visibility}) \\
\textit{intensity} \in \textit{Intensity} ::= & \textbf{float} \mid \textbf{distribution}
\end{aligned}
$$

Figure 8: Continuation of Figure 7.

(Section 3.2.1), (2) Scene Conceptualization Module (Section 3.2.2), (3) Region Construction Module (Section 3.2.3), and (4) Object Placement Module (Section 3.2.4). In the following sections, we introduce them one by one. Please be reminded that (1) all prompt templates in this report for invoking an LLM are written as a multiline string in Python[6], (2) they all require the original user prompt, and (3) they all ask the LLM to output in

[6]Particularly, the `{...}` is for string formatting with the `str.format()` method, while the `{{...}}` is for an actual pair of curly braces.

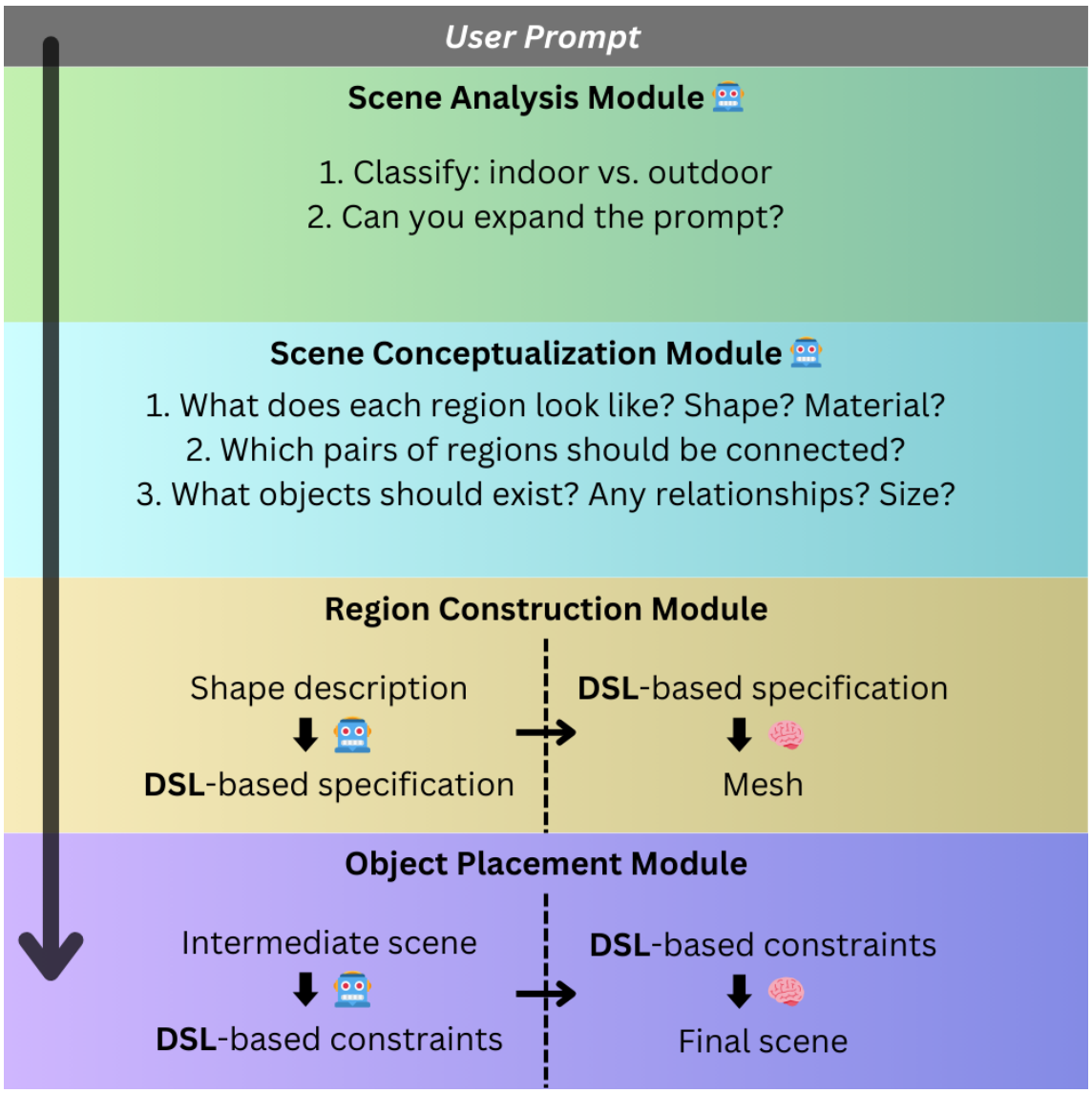


Figure 9: With the help of LLM, Scenethesis employs a module-by-module pipeline to generate a virtual scene from a single user prompt.

JSON format. One last thing is that we are using the left-handed coordinate system (whose length unit is meter) throughout this report, i.e., the positive x-axis points rightward, the positive y-axis points upward, and the positive z-axis points forward.

3.2.1 Scene Analysis Module

Illustrated in the first row of Figure 9, this module takes advantage of LLM to enhance the overall understanding of the desired scene. As an

old Chinese saying goes, “know yourself and know your enemy.” It is of utmost importance for an individual to possess as much information about a particular issue as possible before beginning to tackle it. If this is the case for human being, then it would be unfair to assume that an LLM can perform similarly well (when compared to a human) with less information. In other words, we have to provide the LLM with as much information as we can. Nonetheless, it would be too arrogant to assume that any user using this system is going to input every piece of detail through the prompt. To resolve this, we ask the LLM to analyze and expand the original user prompt (while strictly following any requirements stated in the user prompt) such that it can have a clearer picture of what the final scene should look like before proceeding to the scene generation procedure. This ensures that it has some kind of instructions to follow and will not simply output whatever it wants, increasing the predictability of the final scene.

3.2.1.1 Scene Type

This submodule is responsible for classifying the final scene into indoor or outdoor based on the original user prompt. This allows the LLM to be more aware of the scenario or space that it is dealing with. The output from this submodule is also used in later prompts to determine the suitable content in the prompts.

Prompt template:

```
## Task description

You are given a user prompt: {prompt}
```

```
Classify the scene into "indoor" or "outdoor".

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**:

{{"scene_type": "indoor/outdoor"}}
```

3.2.1.2 Scene Description

This submodule is responsible for adding details to the user prompt, effectively producing a new longer prompt that is more semantically comprehensive and meaningful. The LLM is required to elaborate more on (but not limited to) (1) the scene's purpose of existence (i.e., why would one build such scene), (2) example objects that are likely to exist (i.e., when people are talking about such scene, what objects pop up in their mind), (3) expected human activities (i.e., what can people do in the scene), and (4) atmosphere (i.e., how would one feel when looking at the scene). By doing so, we are able to get our hands on a more complete high-level description of the final scene. We expect all these pieces of information to guide the generation process in the next module, such as increasing the likelihood of generating objects that actually match the scene. One thing to note is that the LLM is required to strictly follow all requirements stated in the original user prompt. By enforcing this, we can better limit the creativity of the LLM such that it only outputs what the user wants and never outputs what the user does not want, making the upcoming scene generation procedure more aligned with the original user prompt. Of course, creativity is still

encouraged in places that are not explicitly mentioned in the text prompt, up to the point of not significantly changing the overall structure or meaning of the scene.

Prompt template (requires output from Section 3.2.1.1):

```
## Task description

You are given a user prompt describing an {scene_type}
scene: {prompt}
Rephrase the user prompt comprehensively to produce a
meaningful description, covering topics including (but
**not** limited to):
1. purpose of existence (i.e., why would one build such
scene);
2. example objects that are likely to exist (i.e., when
people are talking about such scene, what objects pop up
in their mind);
3. expected human activities (i.e., what can people do
in the scene); and
4. atmosphere (i.e., how would one feel when looking at
the scene).

## Guidance

- You **must** strictly follow **all** requirements in
the prompt.

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
<FILL_IN> represents the description you need to
generate:

{{"scene_description": "<FILL_IN>"}}
```

3.2.2 Scene Conceptualization Module

Illustrated in the second row of Figure 9, this module exploits the generation capability of LLM to create a semantic draft of the desired scene. It is "semantic" in the sense that only text is outputted. What about "draft"? It means that the anticipated scene, after this module, is only described in a high-level manner. For instance, instead of outputting every single vertex of a region, we use proper sentences to describe how that region should look like when being viewed from outside (perhaps using some common shapes). In other words, this module generates a rough blueprint of the final scene that cannot be immediately and directly realized, but is thorough and imaginable enough for us to construct the scene inside our mind (hence the word "conceptualization"). You may consider this module as a bridge that connects Section 3.2.1 and later modules.

3.2.2.1 Region Design

This submodule is responsible for generating the details of each region. For indoor scenes, a region is equivalent to a room, which is something very natural and straightforward since every indoor region is nothing but an enclosed space. But for outdoor scenes, we use the word "area" to represent a region, which could be counterintuitive because the concept of area is usually used in 2D context. So, we specifically define what an area is when we are using the prompt template designed for outdoor scenes:

> "An outdoor area is defined as the bottom shape (from bird's eye

view) of a prism with infinite height."

We use this definition because we want to remind the LLM that it only has to deal with objects placed on the ground (because there is naturally no wall nor ceiling in an outdoor scene, which is also why "infinite height" is added), but it is still working in a 3D space (hence the word "prism").

Putting the special definition aside, the LLM has to generate, for each region (no matter it is indoor or outdoor), (1) a unique name for the region, (2) a detailed description summarizing (i) its purpose of existence, (ii) example objects that are likely to exist inside it, (iii) expected human activities inside it, and (iv) its atmosphere, and (3) a detailed textual description illustrating the shape of the region. As a human being, you may already be able to foresee what the scene would look like (or even what objects would probably be placed inside it) by simply analyzing (2) and (3). We believe the LLM can also benefit from this kind of imagination and analysis in later modules. Note that for (3), unlike Holodeck [136], we argue that the chance of seeing a perfect cuboid as a room is much less than that of seeing an irregular shape (such as the English letter "L" when being observed from above), and so we do not explicitly require the LLM to output a cuboid (or rectangle). Furthermore, we emphasize in the prompt that extra regions not requested by the user must not be generated, reducing the odds of the LLM generating more than one region for the sake of making the scene more realistic while in fact the user, for instance, only wants one single region.

For indoor scenes, the LLM has to generate a little bit more. Opposite

to the natural assumption that there is no wall nor ceiling in an outdoor scene, it is rational to assume that there is always a wall and ceiling in an indoor scene. Notice how a typical room uses different colors and materials for its floor, wall, and ceiling? If we want our final scene to be vivid, the appearance of these boundaries must also be considered. In view of this, the LLM is simultaneously asked to generate descriptions of the materials of these types of structures. Using these descriptions, we can then find from a material database $\mathcal{D}_m = \{m_1, \ldots, m_{n_m}\}$ the most suitable candidate (in the form of an image), where n_m is the number of available materials.

In particular, given a material description q_m, we use

$$m^* = \underset{m \in \mathcal{D}_m}{\text{argmax}}\, \text{CLIP}\left(m, q_m\right) \tag{1}$$

to find the most suitable candidate material m^*, where

$$\text{CLIP}\left(m, q_m\right) = \hat{\text{CLIP}}_{\text{img}}\left(m\right) \cdot \hat{\text{CLIP}}_{\text{txt}}\left(q_m\right) \tag{2}$$

in which $\text{CLIP}_{\text{img}}\left(\cdot\right)$ and $\text{CLIP}_{\text{txt}}\left(\cdot\right)$ are the image encoder and text encoder respectively from CLIP [92] that return high-dimensional feature embeddings, $\hat{\mathbf{v}}$ denotes a vector $\mathbf{v}$ after normalization, and $\mathbf{a} \cdot \mathbf{b}$ denotes the dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$. Using simple English, we are finding the material in the database that has the highest visual similarity with the material description. In reality, since $\mathcal{D}_m$ is (assumed to be) fixed, we can precompute the embeddings $\hat{\text{CLIP}}_{\text{img}}\left(m\right)\ \forall m \in \mathcal{D}_m$, and save them somewhere on the system in order to reduce generation time.

Though, the terms "wall" and "ceiling" should actually be collectively called "non-floor", due to the fact that not every enclosed structure consists of distinguishable wall and ceiling (e.g., a dome-shaped observatory or an Egyptian pyramid). So at the end of the day, two material-related descriptions (floor and non-floor) will be generated for each indoor region. The floor will use the floor description, while anything else will use the non-floor description.

Prompt template (for indoor; requires outputs from Sections 3.2.1.1 and 3.2.1.2):

```
## Task description

You are given a user prompt describing an indoor scene:
_{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
Now, generate **all** rooms the final scene has.
For each room, generate
1. its name (which must be unique);
2. its overall description, which should be as
comprehensive and detailed as possible, covering topics
including (but **not** limited to)
  (i) purpose of existence (i.e., why would one need
  such room);
  (ii) example objects that are likely to exist (i.e.,
  when people are talking about such room, what objects
  pop up in their mind);
  (iii) expected human activities (i.e., what can people
  do in the room); and
  (iv) atmosphere (i.e., how would one feel when staying
  inside the room);
3. a description of its three-dimensional shape;
4. a description of its floor material (including color
and texture); and
```

```
5. a description of its wall (i.e., non-floor) material
(including color and texture).

## Guidance

- Do **not** generate rooms **not** requested by the
user.
- Each room **must** have a flat floor.
- When describing the shape of each room, **only**
mention the overall shape when viewed from outside. You
**must not** mention anything like atmosphere or objects
inside the room.
- Do **not** output the (estimated) dimension of each
room.

## Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<FILL_IN> represents the required information you need
to generate:

{{
  "regions": {{
    "<FILL_IN_ROOM_NAME_1>": {{
      "overall_description": "<FILL_IN>",
      "shape_description": "<FILL_IN>",
      "floor_material": "<FILL_IN>",
      "non_floor_material": "<FILL_IN>"
    }},
    "<FILL_IN_ROOM_NAME_2>": {{
      "overall_description": "<FILL_IN>",
      "shape_description": "<FILL_IN>",
```

```
      "floor_material": "<FILL_IN>",
      "non_floor_material": "<FILL_IN>"
    }},
    ...
  }}
}}
```

Prompt template (for outdoor; requires outputs from Sections 3.2.1.1 and 3.2.1.2):

```
## Task description

You are given a user prompt describing an outdoor scene:
_{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
Now, generate **all** areas the final scene has.
For each area, generate
1. its name (which must be unique);
2. its overall description, which should be as
comprehensive and detailed as possible, covering topics
including (but not limited to):
  (i) purpose of existence (i.e., why would one need
  such area);
  (ii) example objects that are likely to exist (i.e.,
  when people are talking about such area, what objects
  pop up in their mind);
  (iii) expected human activities (i.e., what can people
  do in the area); and
  (iv) atmosphere (i.e., how would one feel when staying
  inside the area);
3. a description of its three-dimensional shape;

## Guidance

- {area_def}
- Do **not** generate areas **not** requested by the
user.
```

```
- Do **not** output the (estimated) dimension of each
area.

## Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<FILL_IN> represents the required information you need
to generate:

{{
  "regions": {{
    "<FILL_IN_AREA_NAME_1>": {{
      "overall_description": "<FILL_IN>",
      "shape_description": "<FILL_IN>"
    }},
    "<FILL_IN_AREA_NAME_2>": {{
      "overall_description": "<FILL_IN>"
      "shape_description": "<FILL_IN>"
    }},
    ...
  }}
}}
```

3.2.2.2 Region Connection

This submodule is responsible for establishing connections among regions. The regions generated in Section 3.2.2.1 are, metaphorically, randomly floating on the sea — their positions in the world space are not fixed. Even if we explicitly require them to form a single connected graph, there could

be way too many combinations. Naturally, we want to first fix which pairs of regions are connected (i.e., objects and/or light are able to travel between the two regions), effectively shrinking the solution space. Besides, this is more intuitive because we want to make it clear from which regions you can access a particular region.

For indoor scenes, we know from common sense that if we want to move from one room to another, we have to pass through some obstructing non-floor boundaries between the two rooms. How did people from a thousand years ago solve this problem? They used holes and doors (which by the way are something assumed to be oriented in a vertical manner, implying the presence of a wall). So, for each connection, the LLM has to generate the type of connection, which is one of "open" (i.e., absence of a shared boundary between two rooms), "doorframe" (1 m or 2 m wide), "door" (1 m or 2 m wide), and "window". For other attributes you see in the below prompt template, please refer to Section 3.2.2.3. Though, we have genuine concern that the LLM may find it confusing to use a window as a type of connection. Therefore, we specifically ask the LLM to use anything but a window for a walkable connection, and make sure all rooms together form a single connected graph with non-window connections. One last thing is that in order to enhance the realisticness of the final scene, we want to ensure that one can freely enter and exit the scene (i.e., the scene itself is not a locked/secret environment that is completely isolated and inaccessible). To this end, we additionally define a special region "outside" which describes the space outside the indoor structure, and request the LLM to generate at

least one connection that links one of the rooms to “outside”.

Prompt template (for indoor; requires outputs from Sections 3.2.1.1, 3.2.1.2 and 3.2.2.1):

```
## Task description

You are given a user prompt describing an indoor scene:
_{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
Based on the above description, you have generated
several rooms with the following descriptions:
_{{room_descriptions}}_
Now, generate all pairs of rooms that should be
connected (i.e., objects and/or light are able to travel
between the two rooms).
For each connection, generate
1. its name (which **must** be unique);
2. its type;
3. the description of the object that will be used to
realize that connection, which should be as
comprehensive and detailed as possible, covering topics
including (but **not** limited to) its shape and
appearance (e.g., color, texture, material); and
4. 3 values (x, y, z) describing the object's dimension.

## Guidance

- We are using the left-handed coordinate system, i.e.,
the positive x-axis points rightward, the positive
y-axis points upward, and the positive z-axis points
forward.
- The length unit in this coordinate system is
**meter**.
- At least one of the rooms **must** be connected to
"outside" (a special region describing the space outside
the final building), using a door.
- Two "outside" regions **must** not simultaneously
exist in the same connection.
```

```
- The type of connection **must** be one of {{"open",
"single_doorframe", "double_doorframe", "single_door",
"double_door", "window"}}. "open" means the mere absence
of a shared boundary between two regions;
"single_doorframe" means a 1 m wide empty rectangle on a
vertical wall between the two regions;
"double_doorframe" means a 2 m wide empty rectangle on a
vertical wall between the two regions; "single_door"
means a 1 m wide door on a vertical wall between the two
regions; "double_door" means a 2 m wide door on a
vertical wall between the two regions; "window" means a
light-passing window on a vertical wall between the two
regions.
- Make sure that **all** rooms (plus the special region
"outside") together form a **single** connected graph
with the generated **non-window** connections.
- If you think that a human being should be able to walk
back and forth between two rooms, you **must** use
anything **other than** "window".
- Do **not** generate zero for any dimension.

## Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

_{{room_descriptions}}_: {region_descriptions}

## Output format

Room name must be either "outside" or one of these:
{region_names}
You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<FILL_IN> represents the required information you need
to generate:

{{
```

```
  "<FILL_IN_CONNECTION_NAME_1>": {{
    "region1": "outside",
    "region2": "<FILL_IN_ROOM_NAME_1>",
    "type": "<FILL_IN>",
    "description": "<FILL_IN>",
    "dim_x": <FILL_IN>,
    "dim_y": <FILL_IN>,
    "dim_z": <FILL_IN>
  }},
  "<FILL_IN_CONNECTION_NAME_2>": {{
    "region1": "outside",
    "region2": "<FILL_IN_ROOM_NAME_2>",
    "type": "<FILL_IN>",
    "description": "<FILL_IN>",
    "dim_x": <FILL_IN>,
    "dim_y": <FILL_IN>,
  }},
  ...
  "<FILL_IN_CONNECTION_NAME_3>": {{
    "region1": "<FILL_IN_ROOM_NAME_3>",
    "region2": "<FILL_IN_ROOM_NAME_4>",
    "type": "<FILL_IN>",
    "description": "<FILL_IN>",
    "dim_x": <FILL_IN>,
    "dim_y": <FILL_IN>,
    "dim_z": <FILL_IN>
  }},
  "<FILL_IN_CONNECTION_NAME_4>": {{
    "region1": "<FILL_IN_ROOM_NAME_5>",
    "region2": "<FILL_IN_ROOM_NAME_6>",
    "type": "<FILL_IN>",
    "description": "<FILL_IN>",
    "dim_x": <FILL_IN>,
    "dim_y": <FILL_IN>,
    "dim_z": <FILL_IN>
  }},
  ...
}}
```

Prompt template (for outdoor; requires outputs from Sections 3.2.1.1, 3.2.1.2 and 3.2.2.1):

```
## Task description

You are given a user prompt describing an outdoor scene:
_{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
Based on the above description, you have generated
several areas with the following descriptions:
_{{area_descriptions}}_
Now, generate all pairs of areas that should be
connected (i.e., one is able to walk back and forth
between the two areas).
For each connection, generate its name (which **must**
be unique).

## Guidance

- {area_def}
- Make sure that **all** areas together form a
**single** connected graph with the generated
connections.

## Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

_{{area_descriptions}}_: {region_descriptions}

## Output format

Area name must be one of these: {region_names}
You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<FILL_IN> represents the required information you need
to generate:
```

```
{{
  "<FILL_IN_CONNECTION_NAME_1>": {{
    "region1": "<FILL_IN_ROOM_NAME_1>",
    "region2": "<FILL_IN_ROOM_NAME_2>"
  }},
  "<FILL_IN_CONNECTION_NAME_2>": {{
    "region1": "<FILL_IN_ROOM_NAME_3>",
    "region2": "<FILL_IN_ROOM_NAME_4>"
  }},
  ...
}}
```

3.2.2.3 Object Selection

This submodule is responsible for picking the target objects in the final scene. Since one region (no matter it is from an indoor or outdoor scene) can already house a considerable amount of objects, asking the LLM to generate objects in all regions at once could make the LLM pay less attention to each region, which may in turn reduce the number of generated objects in each region. To guarantee that the LLM can properly focus on each region, we have to deal with each region one by one. However, our preliminary experiments reveal that if we prompt the LLM to generate all objects in a region together with their corresponding attributes at once, the number of generated objects could still be below satisfactory. We think this happens because each object has 5 attributes, and thus the LLM shifts its focus on generating attributes rather than the actual existence of objects. To tackle this problem, we use 3 separate prompts sequentially to select objects in each region. Consequently, if there are n_r regions, where n_r is the number

of generated regions in Section 3.2.2.1, then the total number of LLM invocations in this submodule is $3n_r$.

Object names. The LLM is prompted to generate the (unique) names of all objects, including dependent (or small) objects, i.e., objects (such as a coffee cup) that cannot exist without relying on another object (such as a table). Because in Section 3.2.2.2, we have already obtained objects used in connections, it would be redundant to generate connection-related objects here. So, we explicitly ask the LLM not to generate doors nor windows. We also do not want the LLM to generate stairs, because currently we do not support multi-level scenes (i.e., scenes in which one region is positioned above another region). On the other hand, there is something we explicitly ask the LLM to generate. No matter it is indoor or outdoor scene, people cannot see without light. Therefore, we prompt the LLM to generate at least one light-emitting object in each region.

Note that there are two special placeholders `{region_type}` and `{area_def}` in the prompt template. If the scene type is classified as outdoor (Section 3.2.1.1), then `{region_type}` will be "room" and `{area_def}` will be nothing. Otherwise, `{region_type}` will be "area" and `{area_def}` will the definition of an outdoor area (Section 3.2.2.1). By doing so, we can inject necessary information to the LLM.

Prompt template (requires output from Sections 3.2.1.1, 3.2.1.2 and 3.2.2.1):

```
## Task description
```

```
You are given a user prompt describing an {scene_type}
scene: _{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
One of your generated {region_type}s named
"{region_name}" is described by the following:
_{{region_description}}_
Now, generate the names (which must be unique) of
**all** 3D objects this particular {region_type} has.

## Guidance

{area_def}- Do **not** generate doors, windows, nor
stairs.
- You **must** simultaneously consider and return
**all** dependent objects (a.k.a. small objects, i.e.,
objects that cannot exist without relying on another
object; e.g., coffee cup on a table).
- Each {region_type} **must** have at least one
light-emitting object.

## Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

_{{region_description}}_: {region_description}

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<FILL_IN> represents the required information you need
to generate:

{{
  "objects": [
    "<FILL_IN_OBJECT_NAME_1>",
```

```
    "<FILL_IN_OBJECT_NAME_2>",
    ...
  ],
  "lights": [
    "<FILL_IN_LIGHT_NAME_1>",
    "<FILL_IN_LIGHT_NAME_2>",
    ...
  ]
}}
```

Object relationships. Based on the generated object names, together with the previously generated region description Section 3.2.2.1, the LLM is prompted to infer the relationships among all objects. More specifically, the LLM is asked to use one or more paragraphs to clearly describe how one object relates to other objects as well as how they are positioned, with the goal of making whoever is reading those paragraphs able to easily picture the configuration in target region. The relationships generated here will be of use later.

Prompt template (requires output from Sections 3.2.1.1, 3.2.1.2 and 3.2.2.1, and object names):

```
## Task description

You are given a user prompt describing an {scene_type}
scene: _{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
One of your generated {region_type}s named
"{region_name}" is described by the following:
_{{region_description}}_
In this {region_type}, you have generated the following
objects: _{{objects}}_
Now, using one paragraph, describe clearly **all**
relationships among the given objects.
```

```
## Guidance

{area_def}- In your paragraphs, **all** objects **must**
be mentioned.
- One should be able to picture how the objects are
positioned after reading your paragraphs.

## Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

_{{region_description}}_: {region_description}

_{{objects}}_: {object_names}

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
<FILL_IN> represents the required information you need
to generate:

{{
  "object_relationships": "<FILL_IN>"
}}
```

Object attributes. The LLM is prompted to generate, for each proposed object, (1) its category (our preliminary experiments reveal that the LLM is likely to output vague categories like furniture and component, which is something we can tackle using prompt engineering by blacklisting nouns like “furniture” and “fixture”), (2) a detailed description covering topics including but not limited to (i) purpose of existence, (ii) shape, (iii) appearance

(e.g., color), and (iv) expected actions that can be done with it, and (3) its dimension. Semantic information from (1) and (2) can be used to retrieve the most suitable object from an object database $\mathcal{D}_o = \{(o_1, d_1), \ldots, (o_{n_o}, d_{n_o})\}$, where o_i is a 3D mesh, d_i is a description of the corresponding object, and n_o is the number of available objects.

In particular, given an object query q_o, we use

$$(o^*, d^*) = \underset{(o,d)\in\mathcal{D}_o}{\text{argmax}} \frac{\alpha \cdot \max\limits_{r\in\mathcal{R}} \text{CLIP}\,(r, q_o) + \beta \cdot \text{SBERT}\,(d, q_o)}{\alpha + \beta} \tag{3}$$

to find the most suitable candidate material o^*, where

$$\text{SBERT}\,(d, q_o) = \hat{\text{SBERT}}_{\text{txt}}\,(d) \cdot \hat{\text{SBERT}}_{\text{txt}}\,(q_o) \tag{4}$$

in which $\text{SBERT}_{\text{txt}}\,(\cdot)$ is the text encoder from Sentence Transformers [96] that returns high-dimensional feature embeddings, $\mathcal{R}$ is the set of 2D renderings of o from different angles, and α and β are hyperparameters. Basically, the most suitable object is the one that has the highest weighted average of visual and textual similarities with the object query. We use the maximum CLIP score among 2D renderings instead of the average because some renderings could be looking at the back of an object, which may not be able to fully represent the object. So, we are only interested in the rendering that can best represent the object, in the sense that the object has the potential to be this similar to the q_o. Furthermore, given that the embedding models remain unchanged, we can deduce that the same q_o always gives the same o^*. Therefore, we explicitly state in the prompt that if two objects

are supposed to be identical, their descriptions must also be identical. In reality, since $\mathcal{D}_o$ is (assumed to be) fixed, we can precompute the embeddings $\hat{\text{CLIP}}_{\text{img}}(r) \ \forall r \in \mathcal{D}_o$ and $\hat{\text{SBERT}}_{\text{txt}}(d) \ \forall d \in \mathcal{D}_o$, and save them somewhere on the system in order to reduce generation time.

As for (3), unlike Holodeck [136], we do not use it for object retrieval. It is because to ensure our method can adapt to any database, we cannot assume that each object in the database is correctly scaled. So, we instead consider the dimension from (3) an anticipated space that should be occupied by the object, and rescale (fit or stretch) the candidate object such that it can be put inside the 3D bounding box.

Prompt template (requires output from Sections 3.2.1.1, 3.2.1.2 and 3.2.2.1, object names, and object relationships):

```
## Task description

You are given a user prompt describing an {scene_type}
scene: _{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
One of your generated {region_type}s named
"{region_name}" is described by the following:
_{{region_description}}_
In this {region_type}, you have generated the following
objects: _{{objects}}_
These objects have the following relationships:
_{{relationships}}_
Now, for each object, generate
1. its category (which can be repeated; must be
semantically meaningful --- do **not** generate vague
categories like "furniture", "fixture", or "component");
2. its description, which should be as comprehensive and
detailed as possible, covering topics including (but
**not** limited to):
```

```
  (i) purpose of existence;
  (ii) shape;
  (iii) appearance (e.g., color, texture, material); and
  (iv) expected actions that can done with it.
3. 3 values (x, y, z) describing its dimension.

## Guidance

{area_def}- We are using the left-handed coordinate
system, i.e., the positive x-axis points rightward, the
positive y-axis points upward, and the positive z-axis
points forward.
- The length unit in this coordinate system is
**meter**.
- If you expect several objects to be identical, their
categories and descriptions **must** also be identical.
- Do **not** generate zero for any dimension.

## Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

_{{region_description}}_: {region_description}

_{{objects}}_: {object_names}

_{{relationships}}_: {relationships}

## Output format

You **must** include **all** provided object names
as-is.
You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<USE_GIVEN> represents a thing that you **must**
directly copy from the given information.
```

```
<FILL_IN> represents the required information you need
to generate:

{{
  "objects": {{
    "<USE_GIVEN_OBJECT_NAME_1>": {{
      "category": "<FILL_IN>",
      "description": "<FILL_IN>",
      "dim_x": <FILL_IN>,
      "dim_y": <FILL_IN>,
      "dim_z": <FILL_IN>
    }},
    ...
  }},
  "lights": {{
    "<USE_GIVEN_LIGHT_NAME_1>": {{
      "category": "<FILL_IN>",
      "description": "<FILL_IN>",
      "dim_x": <FILL_IN>,
      "dim_y": <FILL_IN>,
      "dim_z": <FILL_IN>
    }},
    ...
  }}
}}
```

3.2.3 Region Construction Module

Illustrated in the third row of Figure 9, this module constructs the actual boundaries of each region. If Section 3.2.2.1 is a designer, then this module is an architect, digesting and utilizing information from the designer for construction. This architect first comprehends what the designer is picturing about and expresses those ambiguous thoughts using his own unambiguous instructions. Then, solely based on these instructions, the architect constructs the region.

3.2.3.1 Shape Generation

This submodule is responsible for translating the high-level textual illustration of each region from Section 3.2.2.1 into something (relatively) low-level. To a certain extent, this submodule is quite similar to a compiler in programming, which translates high-level language into assembly code (something low-level, but not the lowest). Here is also where our DSL is for the first time directly used in the whole pipeline. In particular, the LLM is prompted, with selected snippets of the DSL, to generate a sequence of DSL-based statements, which are in fact definitions of common shapes and/or arbitrary mesh (vertices and faces). These statements are supposed to represent the final structure of the target region accurately. For instance, an L-shaped room could be represented by the union of two touching cuboids. We believe that using DSL as a type of intermediate output can reduce the workload of later submodules, because they no longer have to deal with high-level semantic information, but can instead directly work on lower-level processed information that is unambiguous.

To facilitate the generation procedure, some extra requirements are added into the prompt. First of all, the LLM must construct regions that contain a flat ground on the XZ-plane. In this way, the LLM knows that it should put everything else above the ground. Next, each region is required to touch at least one other region, making sure that the regions form a single connected graph with no physical gap in between. On a similar issue, the LLM must also ensure that no region overlaps with any other regions. This is a very

basic reqirement for maintaining the physical plausibility of the scene.

Prompt template (requires outputs from Sections 3.2.1.1, 3.2.1.2, 3.2.2.1 and 3.2.2.2; please replace $\in$ with "∈"):

```
## Task description

You are given a user prompt describing an {scene_type}
scene: _{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
Based on the above description, you have generated
several {region_type}s with the following shape
descriptions: _{{shape_descriptions}}_
Now, using the below domain-specific language (DSL),
express the shape of each {region_type} line by line
using syntaxes from the DSL.

## Guidance

{area_def}- We are using the left-handed coordinate
system, i.e., the positive x-axis points rightward, the
positive y-axis points upward, and the positive z-axis
points forward.
- The length unit in this coordinate system is
**meter**.
- You **must** generate all shapes in **world
coordinates**.
- Keep the sequence of DSL statements of each
{region_type} **precise** and **concise**. Do **not**
output redundant DSL statements.
- The **last** statement in each DSL sequence will be
final shape of the corresponding {region_type}.
- If the scene is classified as an indoor scene, each
vertex should contain 3 values (x, y, z); otherwise,
each vertex should contain only 2 values (x, y).
- Each {region_type} **must** contain a flat ground on
the xz-plane.
- Each {region_type} **must** be sized just right,
**not** too large nor too small.
```

````
- Each {region_type} **must** be **touching** at least
one other {region_type} **from outside**.
- Each {region_type} **must not overlap** with any other
{region_type}s.

## Domain-Specific Language (DSL)

```
shape $\in$ Shape ::= cuboid
cuboid $\in$ Cuboid ::= cuboid(corner.min, corner.max)
corner $\in$ Corner ::= min | max
```

where:
- `center`, `corner.min`, and `corner.max` are vertices.

### Examples

```
shape1 = cuboid((0, 0, 0), (3, 3, 6))
```

## Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

_{{shape_descriptions}}_: {shape_descriptions}

## Output format

You **must** include all provided region names as-is.
You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<USE_GIVEN> represents a thing that you **must**
directly copy from the given information.
<FILL_IN> represents the required information you need
to generate:
````

```
{{
  "<USE_GIVEN_REGION_NAME_1>": [
    "<FILL_IN_SHAPE_DSL_1>",
    "<FILL_IN_SHAPE_DSL_2>",
    ...
  ],
  ...
}}
```

3.2.3.2 Mesh Generation

This submodule is responsible for parsing the DSL-based statements from Section 3.2.3.1 into the basis of any 3D model: its mesh (we use $\mathcal{P}_s$ to denote this parser). Given how low-level the output of $\mathcal{P}_s$ is, if we again use a compiler to analogize Section 3.2.3.1, then it would reasonable to use an assembler to analogize $\mathcal{P}_s$. A mesh is made up of a set of vertices $\mathcal{V}$ and a set of faces. The vertices are just (2D or 3D) points in the coordinate system you are working with. The faces, on the other hand, are slightly more complicated.

Every face consists of 3 distinct indices, each corresponding to one (and only one) vertex in $\mathcal{V}$, ultimately forming a triangle. This may sound trivial, but the order of the 3 indices of each face is also something we must pay attention to. Basically, it controls the direction of the surface normal of that face. For example, if the order `(0, 1, 2)` results in a surface normal that points towards you, then `(0, 2, 1)` results in a surface normal that points away from you. Why does it matter? In most 3D modelling software, we can see a surface only if its normal forms an angle less than 90° with us (this

is known as backface culling). In other words, the order of indices of each face determines the visibility of that face.

After $\mathcal{P}_s$ finishes its job, we essentially possess the actual 3D model of each region.

In this report, for simplicity, we use an LLM as $\mathcal{P}_s$. Specifically, the LLM is first prompted to generate a minimal set of vertices necessary for outlining the outermost surface of each region. For example, if the region has the shape of a cuboid, then the LLM should output 8 coordinates. We incorporate the adjective "minimal" because we want to reduce the chance of the LLM outputting redundant vertices that are useless for constructing the region.

Prompt template (requires outputs from Sections 3.2.1.1, 3.2.1.2, 3.2.2.1 and 3.2.3.1; please replace `$\in$` with "∈"):

```
## Task description

You are given a user prompt describing an {scene_type}
scene: _{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
Using the below domain-specific language (DSL), your
generated {region_type}s' shapes are described by the
following DSL statements: _{{dsl}}_
Now, for each {region_type}, generate a **minimal** set
of vertices necessary for constructing the outermost
surface.

## Guidance
```

````
{area_def}- We are using the left-handed coordinate
system, i.e., the positive x-axis points rightward, the
positive y-axis points upward, and the positive z-axis
points forward.
- The length unit in this coordinate system is
**meter**.
- You **must** generate all vertices in **world
coordinates**.
- The **last** statement in each DSL sequence is final
shape of the corresponding {region_type}.
- If the scene is classified as an indoor scene, each
vertex should contain 3 values (x, y, z); otherwise,
each vertex should contain only 2 values (x, y).
- Do **not** output **redundant** vertices (i.e.,
vertices without which the overall shape will not
change).
- Each {region_type} **must** contain a flat ground on
the xz-plane.
- Each {region_type} **must** be sized just right,
**not** too large nor too small.
- Each {region_type} **must** be **touching** at least
one other {region_type} **from outside**.
- Each {region_type} **must not overlap** with any other
{region_type}s.

## Domain-Specific Language (DSL)

```
shape $\in$ Shape ::= cuboid
cuboid $\in$ Cuboid ::= cuboid(corner.min, corner.max)
corner $\in$ Corner ::= min | max
```

where:
- `center`, `corner.min`, and `corner.max` are vertices.

## Known information

_{{user_prompt}}_: {prompt}
````

```
_{{scene_description}}_: {scene_description}

_{{dsl}}_: {dsl}

## Output format

You **must** include **all** provided {region_type}
names as-is.
You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
<USE_GIVEN> represents a thing that you **must**
directly copy from the given information.
<FILL_IN> represents the required information you need
to generate:

{{
  "<USE_GIVEN_REGION_NAME_1>": [
    [<FILL_IN>, ...],
    ...
  ],
  ...
}}
```

Equipped with these vertices, the LLM then generates a minimal set of faces necessary for constructing a triangular mesh of the region. We again use "minimal" because we do not want the LLM to output erroneous faces that are placed inside the region. Instead, we want the LLM to only focus on the outermost structure of the region.

Prompt template (requires outputs from Sections 3.2.1.1, 3.2.1.2, 3.2.2.1 and 3.2.3.1, and mesh vertices; please replace $\in$ with "∈"):

```
## Task description

You are given a user prompt describing an {scene_type}
scene: _{{user_prompt}}_
```

```
You have generated a description for the scene:
_{{scene_description}}_
Using the below domain-specific language (DSL), your
generated {region_type}s' shapes are described by the
following DSL statements: _{{dsl}}_
For each {region_type}, you have further generated a
**minimal** set of vertices necessary for constructing
the outermost surface: _{{vertices}}_
Now, for each {region_type}, generate a **minimal** set
of faces necessary for constructing a **triangular**
mesh with the vertices provided.

## Guidance

{area_def}- We are using the left-handed coordinate
system, i.e., the positive x-axis points rightward, the
positive y-axis points upward, and the positive z-axis
points forward.
- The length unit in this coordinate system is
**meter**.
- The **last** statement in each DSL sequence is final
shape of the corresponding {region_type}.
- Each face **must** contain exactly 3 indices of
vertices.
- For each mesh, the maximum index used in "faces"
**must** be **less than** the number of vertices.
- Keep the mesh of each {region_type} **precise** and
**concise**.
- Each mesh **must** be **watertight** (i.e., each edge
is shared by exactly two faces).
- Each mesh **must not** contain vertices or faces
underneath the outermost surface.
- Each {region_type} **must** contain a flat ground on
the xz-plane.
- Each {region_type} **must not** collide with any other
{region_type}s.

## Domain-Specific Language (DSL)

```
```
```

```
shape $\in$ Shape ::= cuboid
cuboid $\in$ Cuboid ::= cuboid(corner.min, corner.max)
corner $\in$ Corner ::= min | max
```

where:
- `center`, `corner.min`, and `corner.max` are vertices.

Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

{{dsl}}: {dsl}

{{vertices}}: {vertices}

Output format

You **must** include **all** provided {region_type} names as-is.
You should respond in **JSON ONLY**.
DO NOT output other contents, **not even comments**.
<USE_GIVEN> represents a thing that you **must** directly copy from the given information.
<FILL_IN> represents the required information you need to generate:

```
{{
  "<USE_GIVEN_REGION_NAME_1>": [
    [<FILL_IN>, <FILL_IN>, <FILL_IN>],
    ...
  ],
  ...
}}
```

Because our preliminary experiments reveal that LLMs perform relatively poor when it comes to generating shapes other than cuboid (e.g.,

ellipsoid or raw mesh) as well as dealing with binary operations (union, intersection, subtraction) between two regions, we intentionally omit the relevant DSL syntaxes in the above 3 prompt templates. However, we emphasize that this is only a problem with (current) LLMs. If we implement a non-LLM-based parser (which is in fact one of our future objectives), we will be able to generate all sorts of shapes.

3.2.4 Object Placement Module

Illustrated in the fourth row of Figure 9, this final module strategically places selected objects from Section 3.2.2.3 to their optimal position in the scene.

3.2.4.1 Constraint Generation

This submodule is responsible for unambiguously determining the relationships between each region and all objects inside that region. By injecting selected snippets of our DSL, together with the textual relationships from Section 3.2.2.3, into the prompt, we ask the LLM to generate all DSL-based constraints that are required to make the final scene as realistic and physically plausible as possible. With more constraints, we can more easily shrink the solution space and predict the outcome. Nonetheless, preliminary experiments reveal that simply mentioning "realistic and physically plausible" is not sufficient. Therefore, we explicitly state that (1) each object's y-coordinate must be at least half of its height (so that it does not penetrate the ground), (2) all objects must be positioned inside their corresponding

region, and (3) no object should overlap with its corresponding region's boundaries and with other objects.

Prompt template (requires outputs from all previous submodules; please replace $\in$ with "∈"):

```
## Task description

You are given a user prompt describing an {scene_type}
scene: _{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
You have also generated a high-level world view of the
final scene: _{{scene}}_
Now, using the below domain-specific language (DSL),
generate
1. **all** constraints on **all** objects in each region
that are mandatory or desired to make the scene as
**realistic** and **physically plausible** as possible;
and
2. **all other** constraints that are related to the
**overall** scene (e.g., connections).

## Guidance

{area_def}- We are using the left-handed coordinate
system, i.e., the positive x-axis points rightward, the
positive y-axis points upward, and the positive z-axis
points forward.
- The length unit in this coordinate system is
**meter**.
- Each object's `pos_y` must be **at least** half of its
`dim_y`.
- **All** objects **must** be positioned **inside**
their corresponding region.
- Make sure **no** object overlaps with its
corresponding region's boundaries and with other
objects.

## Domain-Specific Language (DSL)
```

```
constraints $\in$ Constraints ::= constraint |
constraint; constraints
constraint $\in$ Constraint ::= spatial_condition |
probabilistic_condition | temporal_condition |
object_relation | visibility_condition |
physical_constraint | user_defined(logic, priority)
spatial_condition $\in$ SpatialCondition ::=
inside(region) | outside(region) | above(object, height)
| below(object, height) | nearby(object, distance) |
alignedWith(object, axis) | tangentTo(surface) |
distanceBetween(object1, object2) == d
probabilistic_condition $\in$ ProbabilisticCondition ::=
probability(p): condition | distributionBased(object,
param: distribution) | resample(object, param)
temporal_condition $\in$ TemporalCondition ::=
always(condition) | eventually(condition) |
until(condition1, condition2) | next(condition) |
during(interval, condition)
object_relation $\in$ ObjectRelation ::=
relation(object1, object2, relation_type)
relation_type $\in$ RelationTypes ::= above | below |
inside | outside | nearby | aligned | occludes |
intersects
visibility_condition $\in$ VisibilityCondition ::=
canSee(observer, target) | occludes(object1, object2) |
visibleInRegion(observer, region) |
rayTraceValid(observer, target, density)
physical_constraint $\in$ PhysicalConstraint ::=
noCollision(object1, object2) |
collisionFreeRegion(region) | stablePosition(object) |
gravityAligned(object)
user_defined $\in$ UserDefinedConstraint ::=
customLogic(logicExpression, priority)
position $\in$ Position ::= (x, y, z) | offset(position,
vector) | relativeTo(reference)
rotation $\in$ Rotation ::= (roll, pitch, yaw) |
relativeTo(reference)
lights $\in$ Lights ::= light | light; lights
```

```
light $\in$ Light ::= id ← light(category, description,
intensity, position, visibility)
intensity $\in$ Intensity ::= float | distribution
```

Known information

_{{user_prompt}}_: {prompt}

_{{scene_description}}_: {scene_description}

{{scene}}: {scene}

Output format

You **must** include all provided region names as-is.
You should respond in **JSON ONLY**.
DO NOT output other contents, **not even comments**.
DO NOT miss any fields.
<USE_GIVEN> represents a thing that you **must**
directly copy from the given information.
<FILL_IN> represents the required information you need
to generate:

```
{{
  "regions": {{
    "<USE_GIVEN_REGION_NAME_1>": [
      "<FILL_IN_CONSTRAINT_DSL_1>",
      "<FILL_IN_CONSTRAINT_DSL_2>",
      ...
    ],
    ...
  }},
  "overall": [
    "<FILL_IN_CONSTRAINT_DSL_3>",
    "<FILL_IN_CONSTRAINT_DSL_4>",
    ...
  ]
}}
```

3.2.4.2 Constraint Satisfaction

This submodule is responsible for solving the constraints generated in Section 3.2.4.1. In other words, this module acts as a constraint solver (we use $\mathcal{S}_c$ to denote it), computing all unspecified positions and orientations.

In this report, for simplicity, we use an LLM as $\mathcal{S}_c$. Specifically, the LLM is prompted to generate the position and rotation of each object (including objects used for connecting two regions), as well as the positions of point lights. Again, we need lights because realistically speaking, there is no vision if there is no light. We use point light only because it is the easiest to implement while being sufficient to test our DSL.

Prompt template (requires outputs from all previous submodules; please replace `$\in$` with "$\in$"):

```
## Task description

You are given a user prompt describing an {scene_type}
scene: _{{user_prompt}}_
You have generated a description for the scene:
_{{scene_description}}_
You have also generated a high-level world view of the
final scene: _{{scene}}_
Now, based on the given constraints driven by the below
domain-specific language (DSL), generate a corresponding
scene by computing the optimal solution.
Particularly, for each object, generate
1. 3 values (x, y, z) describing its position in **world
coordinates**; and
2. 3 values (pitch, yaw, roll) describing its rotation
in Euler Angles (degree, not radian).
```

````
Next, in each anticipated region, generate 3 values (x,
y, z) describing the positions (in **world
coordinates**) of **all** point lights, whcih should
match the light-emitting part of the corresponding
light-emitting object.
Furthermore, for each connection object, generate
1. 3 values (x, y, z) describing its position in **world
coordinates**; and
2. 3 values (pitch, yaw, roll) describing its rotation
in Euler Angles (degree, not radian).

## Guidance

{area_def}- We are using the left-handed coordinate
system, i.e., the positive x-axis points rightward, the
positive y-axis points upward, and the positive z-axis
points forward.
- The length unit in this coordinate system is
**meter**.
- Each object's `pos_y` must be **at least** half of its
`dim_y`.
- **All** objects are positioned **inside** their
corresponding region.
- Make sure **no** object overlaps with its
corresponding region's boundaries and with other
objects.

## Domain-Specific Language (DSL)

```
constraints $\in$ Constraints ::= constraint |
constraint; constraints
constraint $\in$ Constraint ::= spatial_condition |
probabilistic_condition | temporal_condition |
object_relation | visibility_condition |
physical_constraint | user_defined(logic, priority)
````
````

```
spatial_condition $\in$ SpatialCondition ::=
inside(region) | outside(region) | above(object, height)
| below(object, height) | nearby(object, distance) |
alignedWith(object, axis) | tangentTo(surface) |
distanceBetween(object1, object2) == d
probabilistic_condition $\in$ ProbabilisticCondition ::=
probability(p): condition | distributionBased(object,
param: distribution) | resample(object, param)
temporal_condition $\in$ TemporalCondition ::=
always(condition) | eventually(condition) |
until(condition1, condition2) | next(condition) |
during(interval, condition)
object_relation $\in$ ObjectRelation ::=
relation(object1, object2, relation_type)
relation_type $\in$ RelationTypes ::= above | below |
inside | outside | nearby | aligned | occludes |
intersects
visibility_condition $\in$ VisibilityCondition ::=
canSee(observer, target) | occludes(object1, object2) |
visibleInRegion(observer, region) |
rayTraceValid(observer, target, density)
physical_constraint $\in$ PhysicalConstraint ::=
noCollision(object1, object2) |
collisionFreeRegion(region) | stablePosition(object) |
gravityAligned(object)
user_defined $\in$ UserDefinedConstraint ::=
customLogic(logicExpression, priority)
position $\in$ Position ::= (x, y, z) | offset(position,
vector) | relativeTo(reference)
rotation $\in$ Rotation ::= (roll, pitch, yaw) |
relativeTo(reference)
lights $\in$ Lights ::= light | light; lights
light $\in$ Light ::= id ← light(category, description,
intensity, position, visibility)
intensity $\in$ Intensity ::= float | distribution
```

Known information

_{{user_prompt}}_: {prompt}

```
_{{scene_description}}_: {scene_description}

_{{scene}}_: {scene}

## Output format

You **must** include all provided region, object, and
connection names as-is.
You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<USE_GIVEN> represents a thing that you **must**
directly copy from the given information.
<FILL_IN> represents the required information you need
to generate:

{{
  "regions": {{
    "<USE_GIVEN_REGION_NAME_1>": {{
      "objects": {{
        "<USE_GIVEN_OBJECT_NAME_1>": {{
          "pos_x": <FILL_IN>,
          "pos_y": <FILL_IN>,
          "pos_z": <FILL_IN>,
          "rot_x": <FILL_IN>,
          "rot_y": <FILL_IN>,
          "rot_z": <FILL_IN>
        }},
        ...
      }},
      "point_lights": [
        {{
          "pos_x": <FILL_IN>,
          "pos_y": <FILL_IN>,
          "pos_z": <FILL_IN>
        }},
        ...
      ]
    }},
```

```
    ...
  }},
  "connections": {{
    "<USE_GIVEN_CONNECTION_NAME_1>": {{
      "pos_x": <FILL_IN>,
      "pos_y": <FILL_IN>,
      "pos_z": <FILL_IN>,
      "rot_x": <FILL_IN>,
      "rot_y": <FILL_IN>,
      "rot_z": <FILL_IN>
    }},
    ...
  }}
}}
```

Up to this point, the entire scene, which is saved as a JSON file, is generated. We can then upload the JSON file to some 3D modeling software to build a playable scene.

4 Experiments

4.1 Implementation Details

Both $\mathcal{D}_o$ and $\mathcal{D}_m$ are from Holodeck [136], which extracts around 50K objects from Objaverse [22] and annotates them using OpenAI's `gpt-4-1106-preview`. Unless otherwise specified, we use OpenAI's `gpt-4o-2024-08-06` as our LLM throughout our pipeline. We also by default set the temperature parameter to 0 when we are invoking an LLM. For Equation (2), we employ OpenCLIP's implementation [48] of the ViT-L/14 variant trained on the LAION-2B dataset [103]. For Equation (4), we use the `all-mpnet-base-v2` model [47] from Sentence Transformers [96]. We use Blender[7] for generating region meshes and handling 3D objects in $\mathcal{D}_o$. We use Unity Editor[8] to put the region meshes and selected objects together to compose a playable scene (which is automatically converted into an executable). All experiments are conducted on an Apple MacBook Pro with M1 Pro CPU and 16 GB of system memory. The time required for the whole pipeline (from scene type classification to constraint solving) ranges from 45 seconds to 14 minutes, with a mean of 2.5 minutes. This of course depends on the number of regions and the number of objects in each region as these affect how much time the LLM takes to respond.

[7] Version 4.3
[8] Version 6000.0.24f1

4.1.1 System Prompt

All conversations with LLM (except the one in Section 4.1.3) use the following system prompt. We utilize role playing to make the LLM strictly follow instructions.

System prompt:

```
You are a professional indoor/outdoor virtual scene
designer.
Your job is to, given a text prompt from a client,
convert that prompt into a 3D scene.
You **must** strictly follow the requirements of your
clients.
You **must** also adhere to any guidance provided.
Make sure that the scenes you generate are **realistic**
and **physically plausible**.
You need to respond in JSON format.
```

4.1.2 Baseline

To evaluate the effectiveness of our pipeline in making generated scenes more realistic, we set up a baseline that invokes GPT-4o only once to obtain a minimal JSON file.

Prompt template:

```
## Task description

You are given a user prompt: {prompt}
Please generate a corresponding scene.

## Guidance
```

```
- We are using the left-handed coordinate system, i.e.,
the positive x-axis points rightward, the positive
y-axis points upward, and the positive z-axis points
forward.

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<FILL_IN> represents the description you need to
generate:

{{
  "scene_type": "indoor/outdoor",
  "regions": {{
    "<FILL_IN_REGION_NAME_1>": {{
      "floor_material": "<FILL_IN>",
      "non_floor_material": "<FILL_IN>",
      "mesh": {{
        "vertices": [
          [<FILL_IN>, ...],
          ...
        ],
        "faces": [
          [<FILL_IN>, <FILL_IN>, <FILL_IN>],
          ...
        ]
      }},
      "objects": {{
        "<FILL_IN_OBJECT_NAME_1>": {{
          "category": "<FILL_IN>",
          "dim_x": <FILL_IN>,
          "dim_y": <FILL_IN>,
          "dim_z": <FILL_IN>,
          "pos_x": <FILL_IN>,
          "pos_y": <FILL_IN>,
          "pos_z": <FILL_IN>,
          "rot_x": <FILL_IN>,
          "rot_y": <FILL_IN>,
```

```
          "rot_z": <FILL_IN>
        }},
        ...
      }},
      "point_lights": [
        {{
          "pos_x": <FILL_IN>,
          "pos_y": <FILL_IN>,
          "pos_z": <FILL_IN>
        }},
        ...
      ]
    }},
    ...
  }},
  "connections": [
    {{
      "type": "<FILL_IN>",
      "dim_x": <FILL_IN>,
      "dim_y": <FILL_IN>,
      "dim_z": <FILL_IN>,
      "pos_x": <FILL_IN>,
      "pos_y": <FILL_IN>,
      "pos_z": <FILL_IN>,
      "rot_x": <FILL_IN>,
      "rot_y": <FILL_IN>,
      "rot_z": <FILL_IN>
    }},
    ...
  ]
}}
```

4.1.3 Testing Prompts

To test the robustness of our method using multiple prompts, we employ GPT-4o to effciently generate a large set of n diverse prompts that cover a variety of scenarios.

Prompt template:

```
## Task description

You are using a system that can, based on a given text
prompt, automatically generate a corresponding
high-quality 3D scene.
Now, you want to test how the system handles
**{scene_type}** scenes.
Please generate a **diverse** set of {n} prompts.

## Guidance

- **All** scenes must contain **only** {scene_type}
objects.
- **Half** of the scenes **must** contain two or more
regions (rooms/areas).
- **All** scenes must contain only **one** level (i.e.,
no region is positioned above another region).
- Make sure that **no** two prompts are semantically
similar to each other.

## Examples

- Japanese-style living room
- An arcade room with a pool table
- A sculpture museum with diverse statues
- A 1b1b apartment of a researcher who has a cat
- Three professors' office connected to a long hallway,
the professor in office 1 is a fan of Star Wars

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
<FILL_IN> represents the prompt you need to generate:

{{
  "1": "<FILL_IN>",
  "2": "<FILL_IN>",
  ...
```

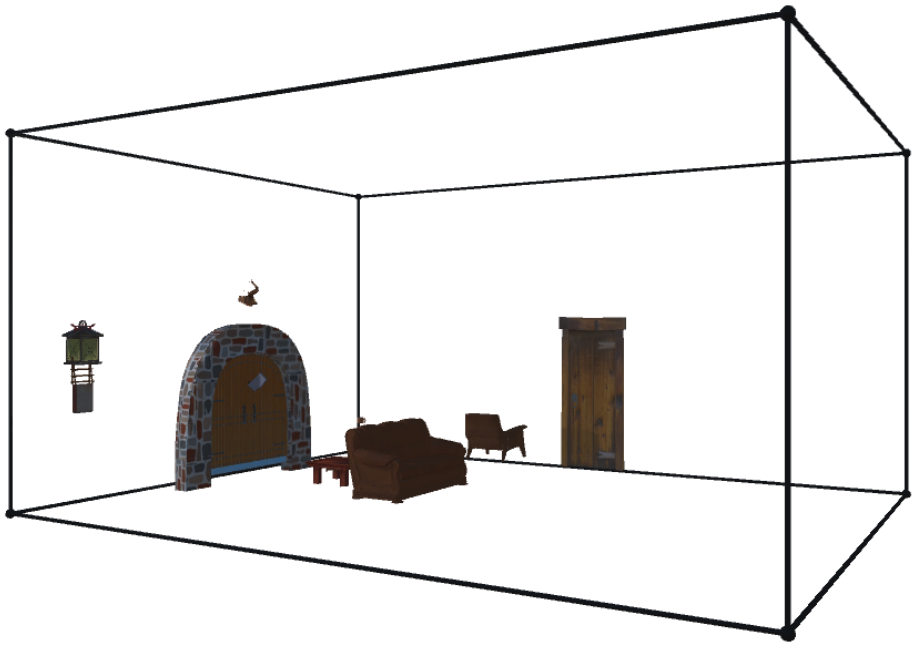


Figure 10: An example of wireframe.

```
  "{n}": "<FILL_IN>"
}}
```

Note, however, that since the object database $\mathcal{D}_o$ used in this report contains only indoor objects, we only use the above prompt template to generate indoor-oriented prompts.

4.1.4 Evaluation Metrics

4.1.4.1 Quantitative

Following [68] and [136], we measure the visual similarities between 2D renderings of a generated scene and relevant text queries using CLIP [92]:

$$\text{Score}_{\text{CLIP}}(r, q) = (\text{CLIP}(r, q) + 1) \times 50 \tag{5}$$

where r is a 2D rendering and q is a text query. We do not directly use Equation (2) because dot product between two normalized vectors results in a number between -1 and 1, and we want to rescale it such that it lies between 0 and 100 (for graphing purposes). For the 2D renderings (which consist of 120 camera angles with pitch $\in \{0, 30, 45, 60, 90\}$ and yaw $\in \{0, 15, 30, \ldots, 330, 345\}$), rather than showing the actual textured scene boundaries (i.e., floor and wall), we instead show the wireframe of the boundaries (see Figure 10). We choose this approach because we do not want walls of one room obstructing objects in other rooms. For the text queries, we first consider the sentence "an image of a vibrant indoor scene" (denote as $\mathcal{C}_1$) and the original user prompt (denote as $\mathcal{C}_2$). If the baseline method (Section 4.1.2) is not being used, then we also consider the scene description generated by LLM in Section 3.2.1.2 (denote as $\mathcal{C}_3$).

Speaking of scene description, since our pipeline depends on the contents in the description, we also measure the textual similarity between the

Figure 11: An example of bird's-eye view.

generated scene description and the original user prompt using SBERT [96]:

$$\text{Score}_{\text{SBERT}}\left(d, p\right) = \left(\text{SBERT}\left(d, p\right) + 1\right) \times 50 \tag{6}$$

where d is the scene description and p is the user prompt.

4.1.4.2 Qualitative

To mimic human evaluation, we give GPT-4o a bird's-eye view (see Figure 11) of a generated scene and ask it to rate the generated scene from 1 to

10 (with 10 being the best) by considering 4 questions:

1. Does the generated scene contain every region mentioned in the prompt?
2. Does the generated scene contain every object mentioned in the prompt?
3. Is the generated scene physically plausible (e.g., are there any objects colliding with region boundaries or other objects)?
4. Is the generated scene visually pleasing?

We do not send renderings from other angles because at those angles, some objects could be obstructed by other objects and region boundaries, negatively affecting the LLM's decision-making. Also, sending multiple images considerably increases evaluation time.

Prompt template:

```
## Task description

You are a harsh tester.
Your job is to test the performance of a system that,
given a text prompt, generates a 3D scene based on the
prompt.
Now, you are given an image (bird's-eye view) of a 3D
scene generated based on the following prompt: {prompt}
Please measure the extent of correlation between the
prompt and the corresponding generated scene.
In particular, you first elaborate in detail about
whether the image match the prompt.
Then, rate the generated scene (from 1 to 10
inclusively, the higher the better) by considering the
following questions:
1. Does the generated scene contain every region
mentioned in the prompt?
```

```
2. Does the generated scene contain every object
mentioned in the prompt?
3. Is the generated scene physically plausible (e.g.,
are there any objects colliding with region boundaries
or other objects)?
4. Is the generated scene visually pleasing?

## Output format

You should respond in **JSON ONLY**.
**DO NOT** output other contents, **not even comments**.
**DO NOT** miss any fields.
<FILL_IN> represents the required information you need
to generate:

{{
  "elaboration": "<FILL_IN>",
  "region_score": <FILL_IN_1_TO_10>,
  "object_score": <FILL_IN_1_TO_10>,
  "physical_score": <FILL_IN_1_TO_10>,
  "visual_score": <FILL_IN_1_TO_10>
}}
```

4.2 Results

4.2.1 Comparing with Baseline

We generate a set of 50 prompts and use them to test both our DSL-based pipeline and the baseline. For each method, every prompt is ran 3 times, generating a total of 300 scenes.

We show quantitative results comparing $\mathcal{C}_1$ and $\mathcal{C}_2$ of our DSL-based pipeline and the baseline in Figure 12. Our pipeline has means of 57.298 and 58.316 respectively, while the baseline has means of 56.130 and 57.019 respectively. While the differences are not that significant, we should bear in

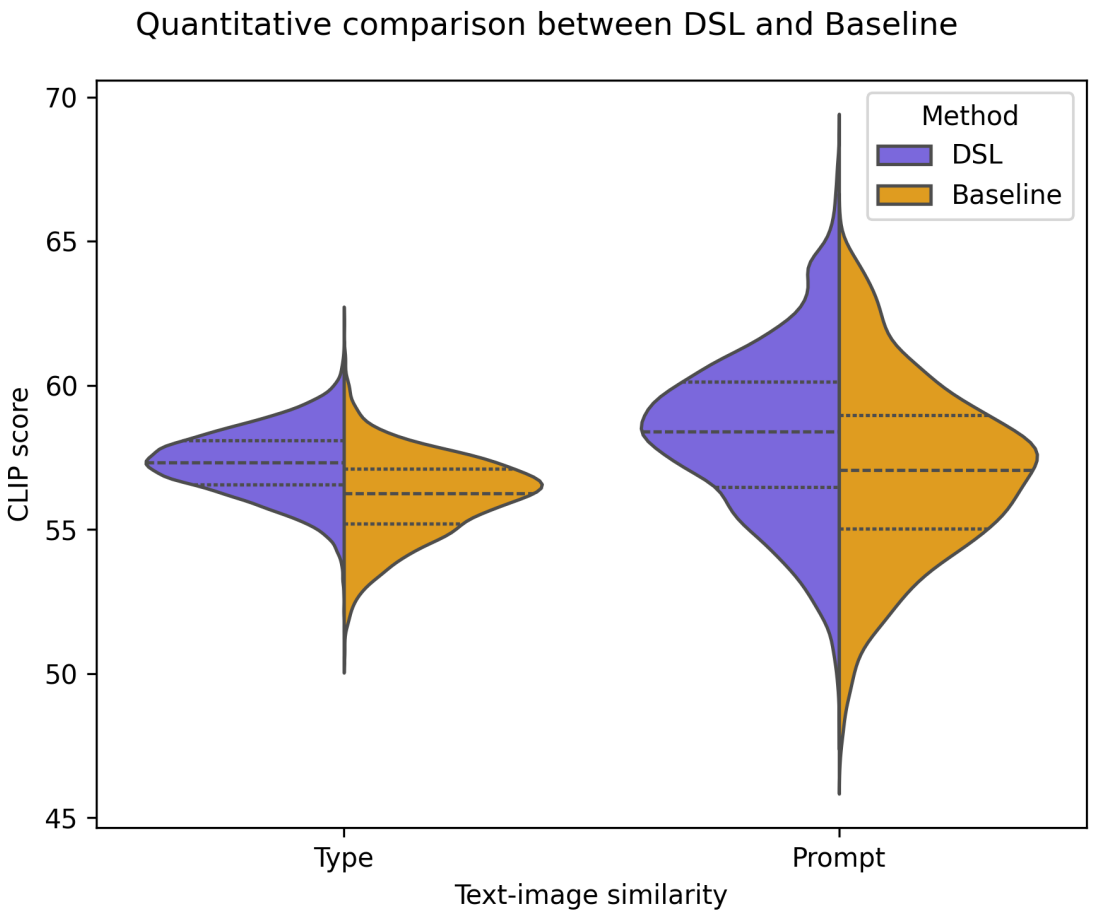


Figure 12: CLIP score comparison between our DSL-based pipeline and the baseline. "Type" refers to $\mathcal{C}_1$ and "Prompt" refers to $\mathcal{C}_2$.

	Average # of regions	Average # of objects
DSL	**1.36**	**20.973**
Baseline	1.35	3.63

Table 1: Comparison between the number of regions and number of objects generated by our DSL-based pipeline and the baseline.

mind that the CLIP models are trained with real-life images, many of which have a main subject, rather than compositions of objects in a virtual scene. In other words, the CLIP models may not be highly sensitive to renderings of virtual scenes. Given the insensitiveness, the small differences could mean a lot.

We also compare the number of regions and number of objects generated by the two methods in Table 1. While both of them on average generate very

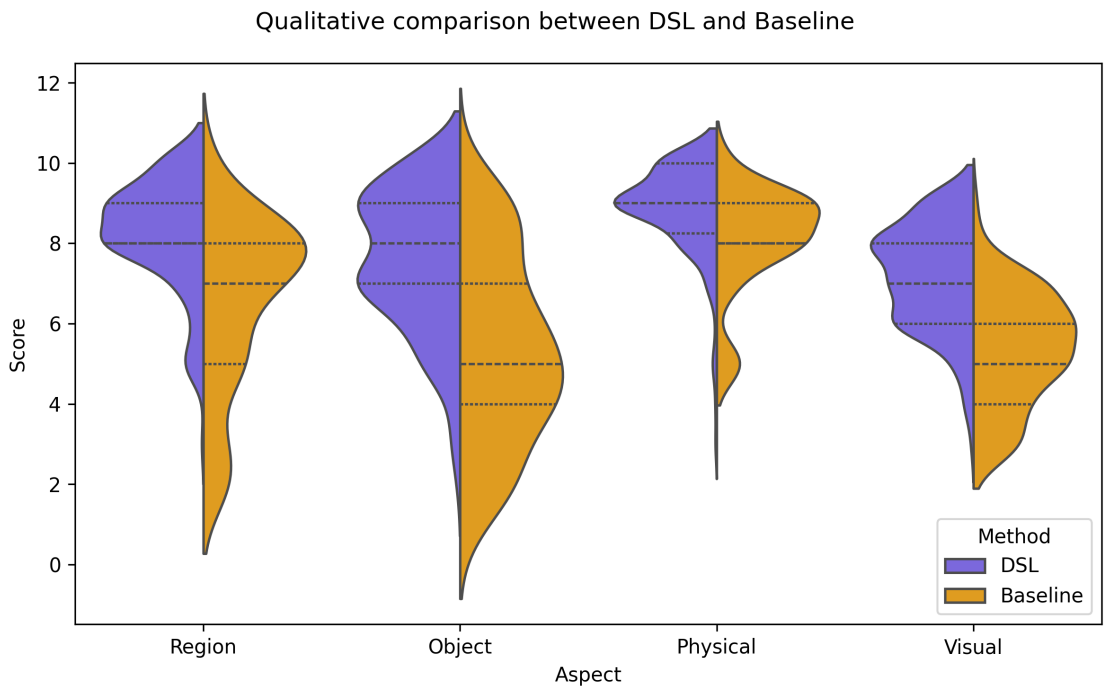


Figure 13: Qualitative comparison between our DSL-based pipeline and the baseline.

similar number of regions, it is obvious that our DSL-based pipeline can generate far more number of objects, demonstrating our proposed method's capability of generating more vibrant scenes. This would be very useful if the user does not want to input a lengthy prompt but still expects to see many objects.

We further show qualitative results from our LLM evaluator in Figure 13. While they receive similar score regarding matching regions (same the above quantitative results, which is because the LLM is not allowed to generate whatever regions it desires), their performance in matching objects and producing visually pleasing scenes differs a lot. This again demonstrates that our DSL-based approach can better follow the requirements stated (explicitly or implicitly) in user prompts while still producing scenes with higher quality. As of why they have similar physical score (about whether the generated scene is physically plausible), we believe it is because the

LLM cannot determine whether an object has penetrated the floor from bird's-eye view.

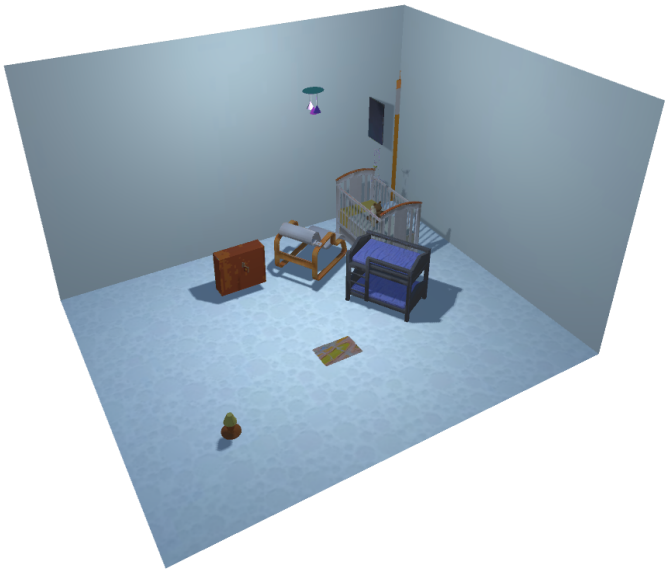
Figures 14 and 15 show two side-by-side comparisons between the two approaches.

4.2.2 Comparing different Temperatures

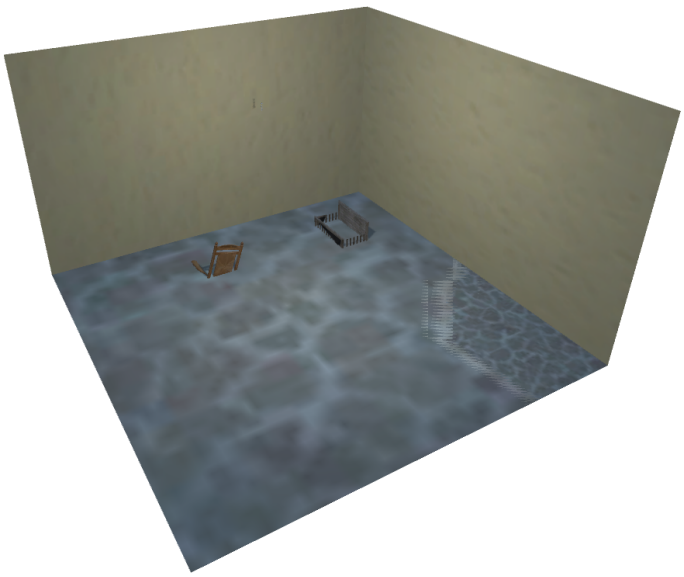
We generate a set of 20 prompts and use them to test how the performance of our DSL-based framework changes as we vary the temperature parameter of GPT-4o. We pick 0, 0.2, and 0.5 as our subjects. For each temperature, every prompt is ran 2 times, generating a total of 120 scenes.

We show quantitative results comparing temperature 0 with 0.2 and 0.5 in Figure 16 and Figure 17 respectively. Apparently, our pipeline remains relatively stable in different temperature settings. Nonetheless, as we increase the temperature parameter, a slight performance gain can be observed (particularly in Figure 17). This suggests that if we do not require the LLM to be highly deterministic, it maybe able to unleash its "creativity" to generate a more relevant scene.

We also show qualitative results comparing temperature 0 with 0.2 and 0.5 in Figure 18 and Figure 19 respectively. Surprisingly, they have identical medians and interquartile ranges. This suggests that the LLM evaluator does not really care about the subtle differences brought by different temperatures. This is also very human-like, as one would tend to give two things identical rating if they are very similar to each other.

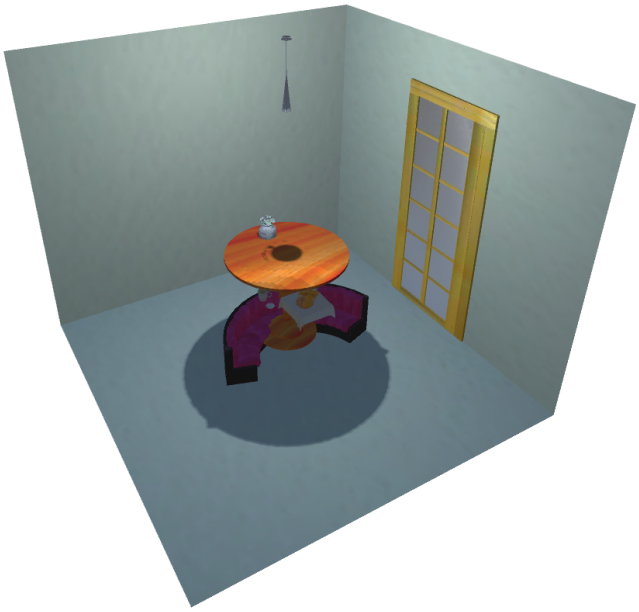


(a) DSL

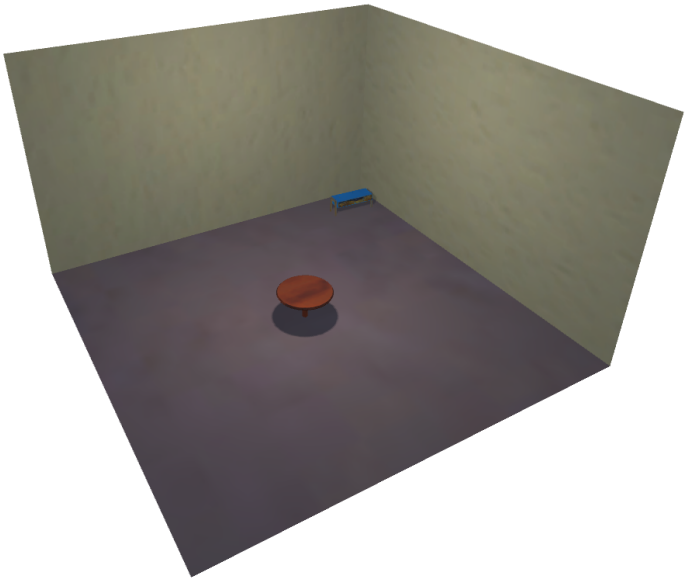


(b) Baseline

Figure 14: Side-by-side comparison of the scene generated from the prompt: “A colorful nursery with a crib, rocking chair, and playful decor.”



(a) DSL



(b) Baseline

Figure 15: Side-by-side comparison of the scene generated from the prompt: “A quaint breakfast nook with a round table and cushioned bench seating.”

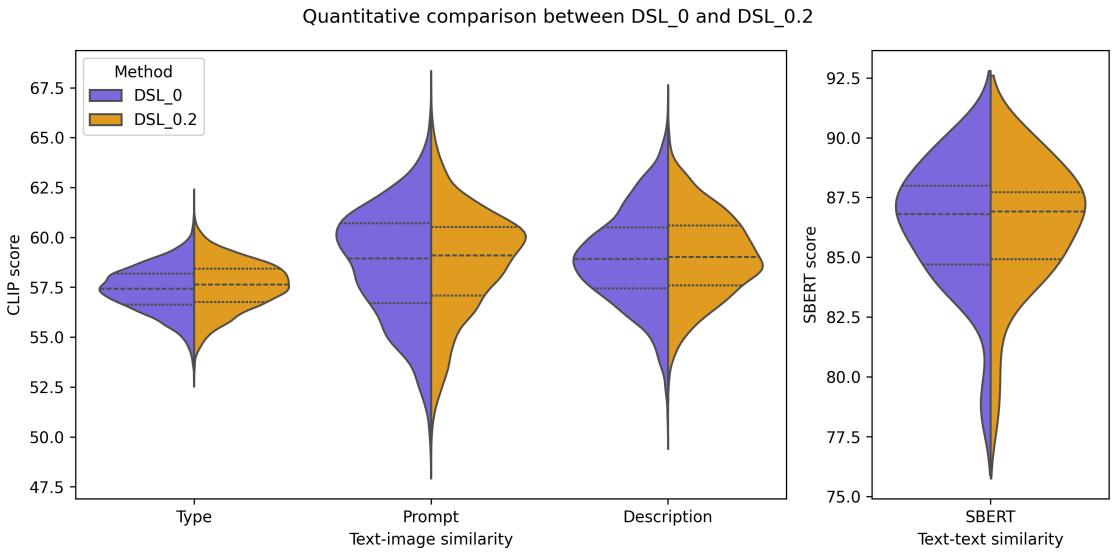


Figure 16: Quantitative comparison between temperatures 0 and 0.2.

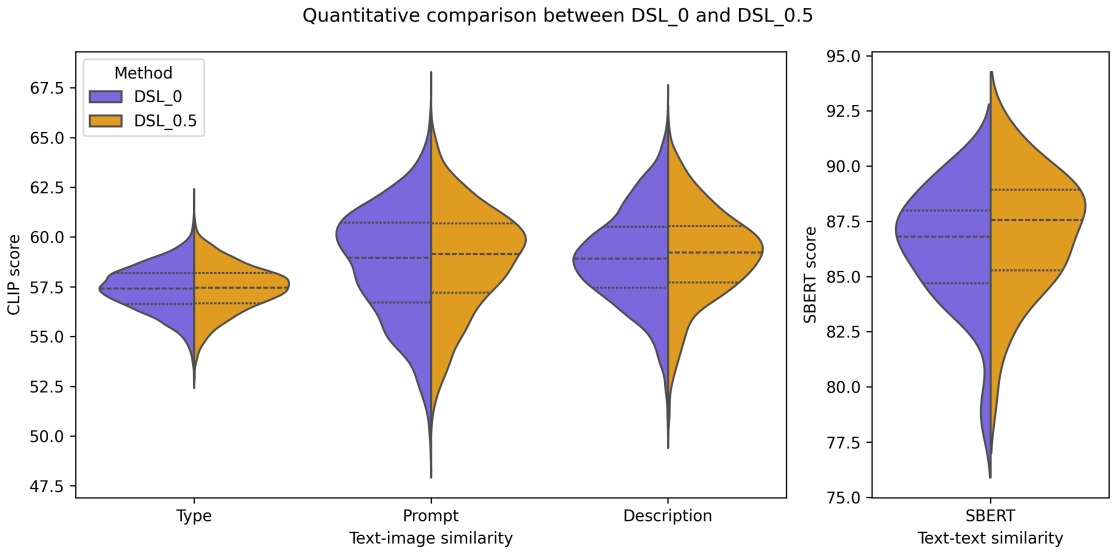


Figure 17: Quantitative comparison between temperatures 0 and 0.5.

4.2.3 Comparing different LLMs

Using the same set of prompts and the same procedure in Section 4.2.2, we test how another LLM from OpenAI, `gpt-4o-mini-2024-07-18`, performs.

We show quantitative results comparing `gpt-4o-2024-08-06` and `gpt-4o-mini-2024-07-18` in Figure 20. Unsurprisingly, `gpt-4o-mini` per-

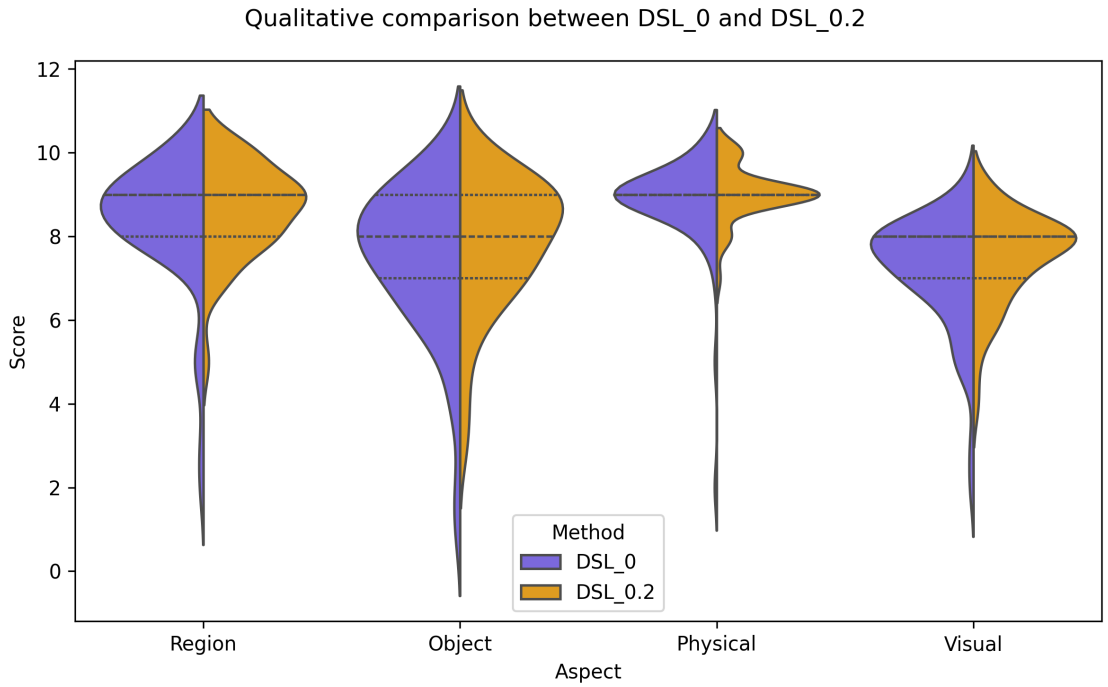


Figure 18: Qualitative comparison between temperatures 0 and 0.2.

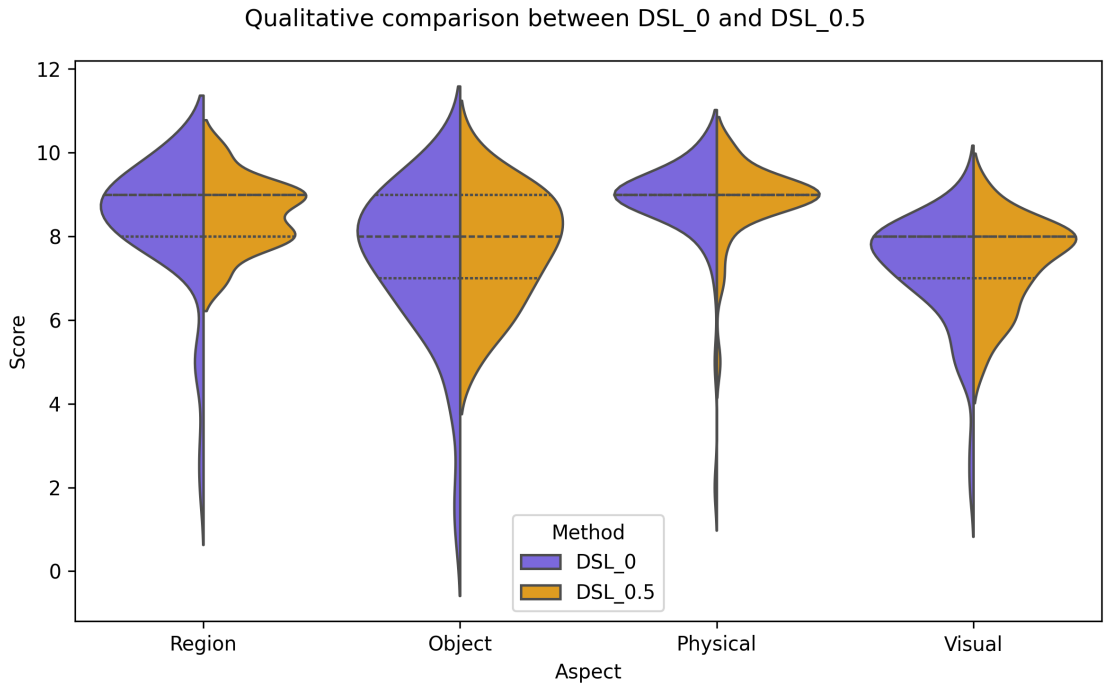


Figure 19: Qualitative comparison between temperatures 0 and 0.5.

forms significantly worse than `gpt-4o`, even though `gpt-4o-mini` has a slightly higher SBERT score. This aligns with our expectation, as it is known `gpt-4o-mini` has lower processing and generative capabilities. This is also confirmed by the qualitative results in Figure 21.

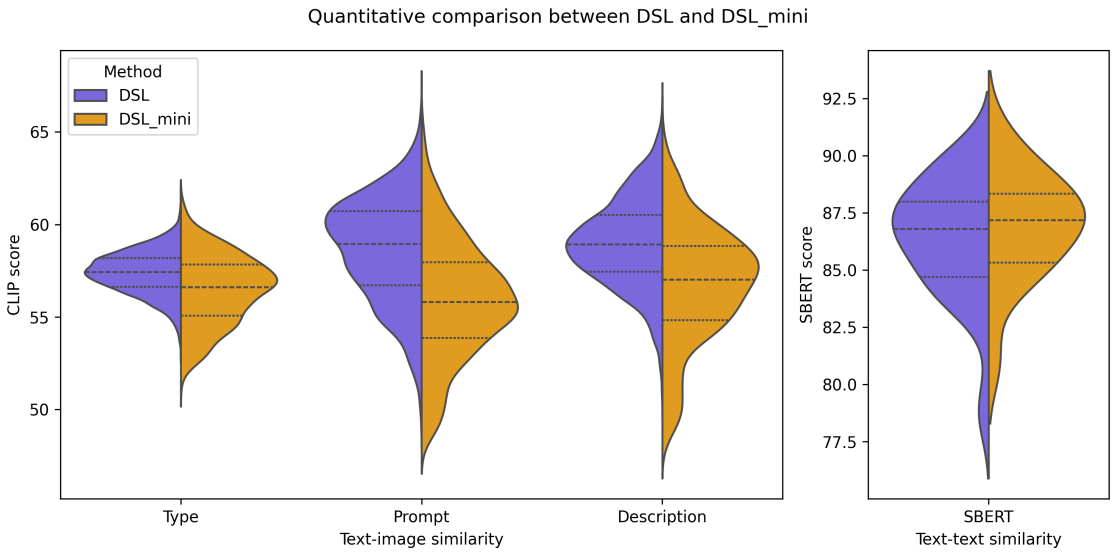


Figure 20: Quantitative comparison between `gpt-4o-2024-08-06` and `gpt-4o-mini-2024-07-18`. "DSL" refers to the use of `gpt-4o-2024-08-06` while "DSL_mini" refers to `gpt-4o-mini-2024-07-18`.

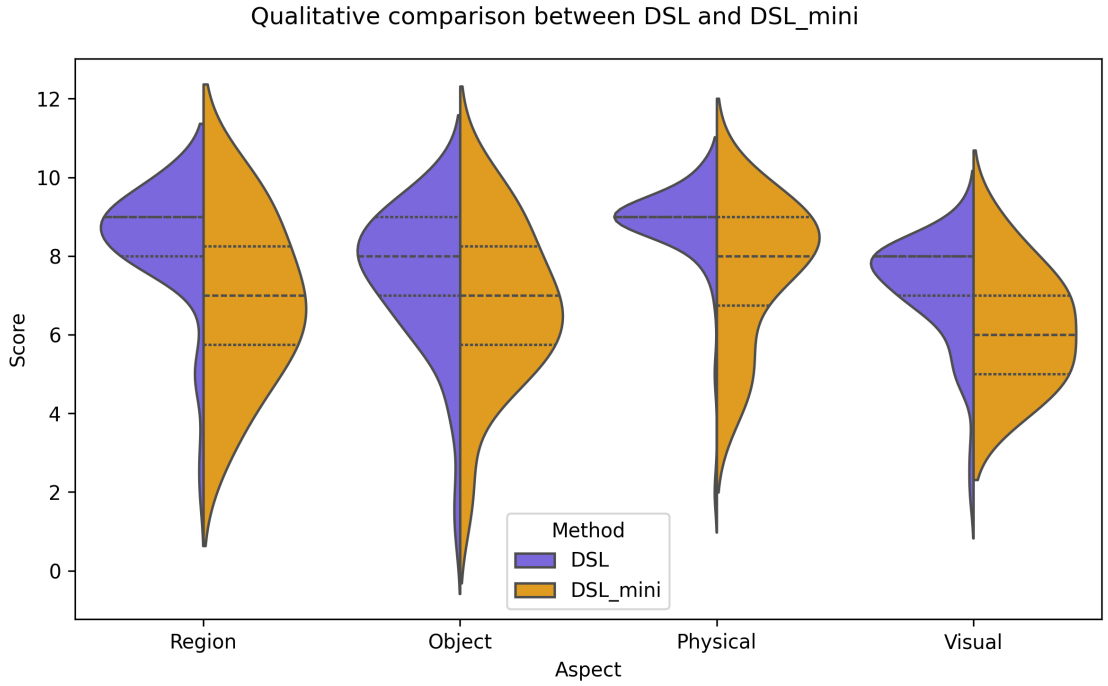


Figure 21: Qualitative comparison between `gpt-4o-2024-08-06` and `gpt-4o-mini-2024-07-18`. "DSL" refers to the use of `gpt-4o-2024-08-06` while "DSL_mini" refers to `gpt-4o-mini-2024-07-18`.

We actually also tested `claude-3-5-sonnet-20241022` from Anthropic. However, due to some technical problems (probably because our prompts are quite long), only 5 scenes were successfully generated, outputting a total of 600 2D renderings. Though, since the results are already here, we think

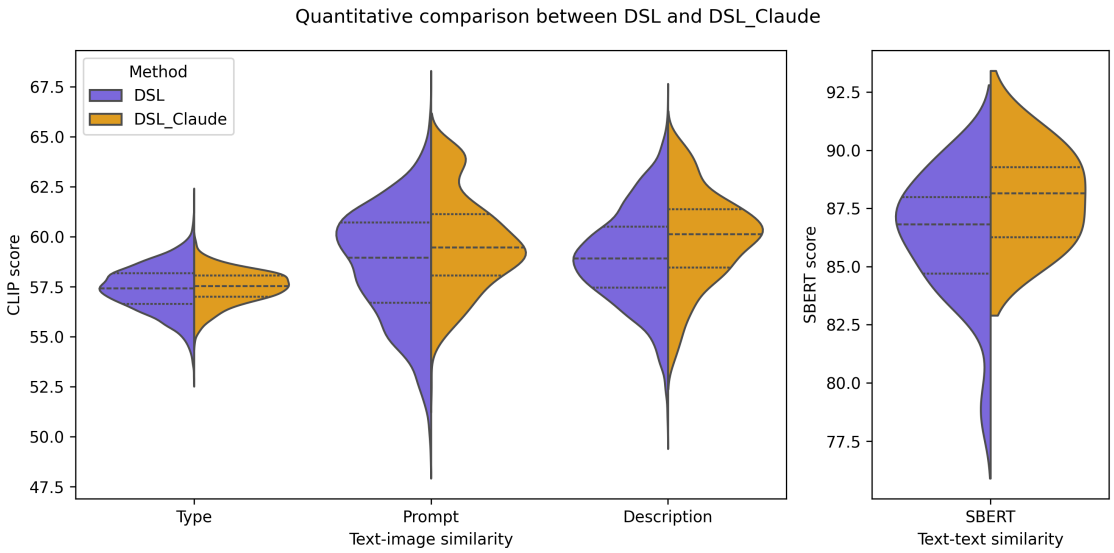


Figure 22: Quantitative comparison between `gpt-4o-2024-08-06` and `claude-3-5-sonnet-20241022`. "DSL" refers to the use of `gpt-4o-2024-08-06` while "DSL_Claude" refers to `claude-3-5-sonnet-20241022`.

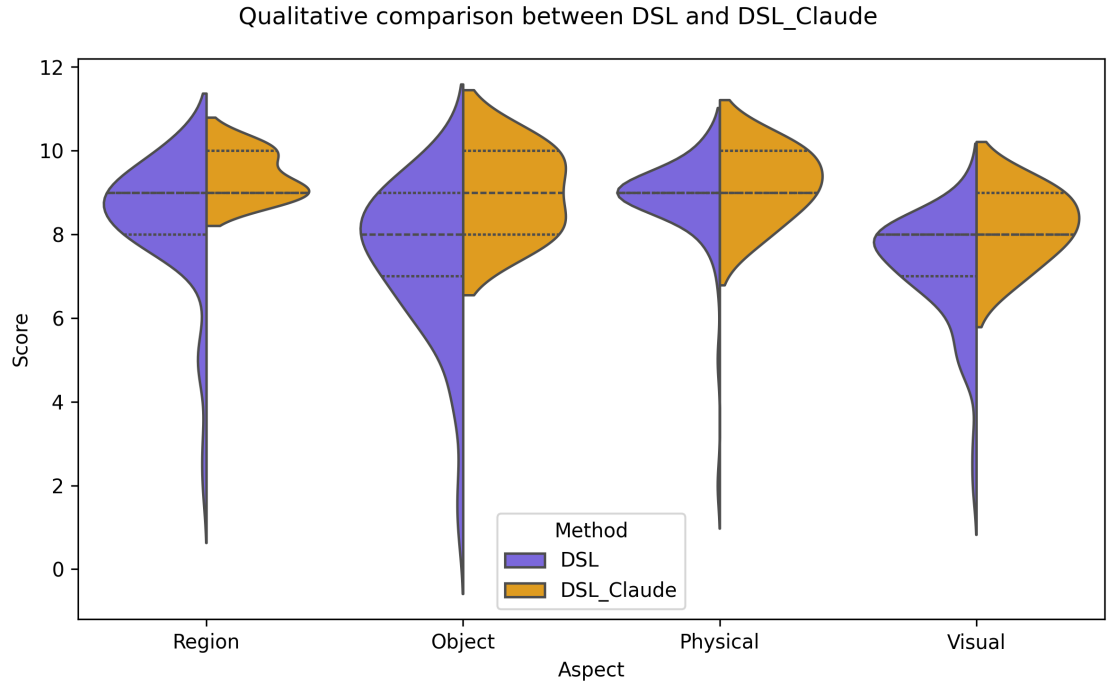


Figure 23: Qualitative comparison between `gpt-4o-2024-08-06` and `claude-3-5-sonnet-20241022`. "DSL" refers to the use of `gpt-4o-2024-08-06` while "DSL_Claude" refers to `claude-3-5-sonnet-20241022`.

it is still worth it to put them in this report. While the qualitative results in Figure 23 does not tell a lot (apart from the fact that textttclaude-3-5-sonnet-20241022 has a slightly higher object-matching score), the quantitative results in Figure 22 are quite interesting, as they are generally higher than

the gpt-4o-2024-08-06 counterparts. Particularly, the SBERT score of claude-3-5-sonnet-20241022 is significantly higher. This could be the evidence that if an LLM can write a better scene description, then the corresponding generated scene would also be better. The higher SBERT score could also be a sign that claude-3-5-sonnet-20241022 has a superior writing capability when compared to gpt-4o-2024-08-06.

5 Conclusion and Future Work

We propose Scenethesis, a novel method that can systematically and automatically generate a playable virtual scene from a single user prompt. By utilizing a modular pipeline, which includes Scene Analysis, Scene Conceptualization, Region Construction, and Object Placement, the tool ensures that each aspect of the scene is meticulously crafted to align with user specifications. Using our novel ScenethesisLang, Scenethesis can generate instruction-guided virtual scenes unambiguously. Experiments show that our method can improve the realisticness when compared to a baseline.

Ultimately, Scenethesis stands as a new tool for scene synthesis, blending advanced AI capabilities with a user-centric design philosophy. As future developments aim to refine the underlying technologies, the potential for even more dynamic and responsive scene generation remains promising, opening new directions for creative applications in virtual environments.

In the future, we will try exploring different shape parser and constraint solver, so as to not solely rely on the processing and generative capabilities of LLM. This is something current LLMs still fail to do, particularly constraint solving, which is a very complex task. We believe that if we can implement a robust constraint solver, our scenes can be much more realistic and physically plausible.

Another thing we can do is to use not just object database. Perhaps we can find some object generation models that can generate high-quality

meshes. Then, when we are quering for an object, if the weighted CLIP and SBERT score is lower than a certain threshold, then we can use the object generation model. In this way, we are assured that if a certain desired object does not exist in the database, we still have a way to get that object.

Lastly, of course, is our journey to fully automatic XR testing. Scenethesis is just the first step in XR testing, i.e., constructing an interactable scene. With a diverse set of virtual environments in hand, we will then be able to run different, for example, mobile AR applications, inside those environments. As long as we have a way to retrieve depth information and maneuver freely in the scene, we can test all sorts of functionalities of the target application. Afterwards, we can, for instance, use a Visual Language Model to analyze the screen recordings, detecting any abnormal events. At that moment, fully automatic XR testing will be achieved.

References

[1] Rio Aguina-Kang, Maxim Gumin, Do Heon Han, Stewart Morris, Seung Jean Yoo, Aditya Ganeshan, R Kenny Jones, Qiuhong Anna Wei, Kailiang Fu, and Daniel Ritchie. Open-universe indoor scene generation using llm program synthesis and uncurated object databases. *arXiv preprint arXiv:2403.09675*, 2024. 20

[2] S. S. Riaz Ahamed. Studying the feasibility and importance of software testing: An analysis. *arXiv preprint arXiv:1001.4193*, 2010. 12

[3] Apple. Submit your apps to the app store for apple vision pro. https://developer.apple.com/visionos/submit/ [Online; accessed 31-October-2024]. 9

[4] Gwangbin Bae, Ignas Budvytis, and Roberto Cipolla. Irondepth: Iterative refinement of single-view depth using surface normal and its uncertainty. In *British Machine Vision Conference (BMVC)*, 2022. 23

[5] Sherwin Bahmani, Jeong Joon Park, Despoina Paschalidou, Xingguang Yan, Gordon Wetzstein, Leonidas Guibas, and Andrea Tagliasacchi. Cc3d: Layout-conditioned generation of compositional 3d scenes. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7171–7181, 2023. 18

[6] Haotian Bai, Yuanhuiyi Lyu, Lutao Jiang, Sijia Li, Haonan Lu, Xiaodong Lin, and Lin Wang. Componerf: Text-guided multi-object compositional nerf with editable 3d scene layout. *arXiv preprint arXiv:2303.13843*, 2023. 24

[7] Sana Behnam. Guidelines for testing mobile augmented-reality apps, 2022. https://www.nngroup.com/articles/testing-ar-apps/ [Online; accessed 31-October-2024]. 12

[8] Jacob Biba. Extended reality: What it is and everything you need to know, 2022. https://digitaltwininsider.com/2022/11/07/everything-you-need-to-know-about-extended-reality-in-2022/ [Online; accessed 31-October-2024]. 7, 8

[9] Jacob Biba and Brennan Whitfield. The fascinating history and evolution of extended reality (xr), 2023. https://builtin.com/hardware/extended-reality [Online; accessed 31-October-2024]. 7

[10] Shengqu Cai, Eric Ryan Chan, Songyou Peng, Mohamad Shahbazi, Anton Obukhov, Luc Van Gool, and Gordon Wetzstein. Diffdreamer: Towards consistent unsupervised single-view scene extrapolation with conditional diffusion models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2139–2150, 2023. 25

[11] Ata Çelen, Guo Han, Konrad Schindler, Luc Van Gool, Iro Armeni, Anton Obukhov, and Xi Wang. I-design: Personalized llm interior designer. *arXiv preprint arXiv:2404.02838*, 2024. 20

[12] Eric R Chan, Connor Z Lin, Matthew A Chan, Koki Nagano, Boxiao Pan, Shalini De Mello, Orazio Gallo, Leonidas J Guibas, Jonathan Tremblay, Sameh Khamis, et al. Efficient geometry-aware 3d generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16123–16133, 2022. 25

[13] Angel Chang, Will Monroe, Manolis Savva, Christopher Potts, and Christopher D. Manning. Text to 3D scene generation with rich lexical grounding. In Chengqing Zong and Michael Strube, editors, *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 53–62, Beijing, China, July 2015. Association for Computational Linguistics. 17

[14] Angel Chang, Manolis Savva, and Christopher D. Manning. Learning spatial knowledge for text to 3D scene generation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2028–2038, Doha, Qatar, October 2014. Association for Computational Linguistics. 16

[15] Angel X Chang, Mihail Eric, Manolis Savva, and Christopher D Manning. Sceneseer: 3d scene design with natural language. *arXiv preprint arXiv:1703.00050*, 2017. 17

[16] Jit Chatterjee and Maria Torres Vega. 3d-scene-former: 3d scene generation from a single rgb image using transformers. *The Visual Computer*, pages 1–15, 07 2024. 19

[17] Aditya Chattopadhyay, Xi Zhang, David Paul Wipf, Himanshu Arora, and René Vidal. Learning graph variational autoencoders with con-

straints and structured priors for conditional indoor 3d scene generation. In *IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 785–794, 2023. 17

[18] Jaeyoung Chung, Suyoung Lee, Hyeongjin Nam, Jaerin Lee, and Kyoung Mu Lee. Luciddreamer: Domain-free generation of 3d gaussian splatting scenes. *arXiv preprint arXiv:2311.13384*, 2023. 22

[19] Giovanni Cisternini et al. A domain specific language for internet measurements. 2024. 26

[20] Dana Cohen-Bar, Elad Richardson, Gal Metzer, Raja Giryes, and Daniel Cohen-Or. Set-the-scene: Global-local training for generating controllable nerf scenes. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2920–2929, 2023. 24

[21] Tianyuan Dai, Josiah Wong, Yunfan Jiang, Chen Wang, Cem Gokmen, Ruohan Zhang, Jiajun Wu, and Li Fei-Fei. Acdc: Automated creation of digital cousins for robust policy learning. *arXiv preprint arXiv:2410.07408*, 2024. 19

[22] Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. Objaverse: A universe of annotated 3d objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13142–13153, 2023. 80

[23] Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Jordi Salvador, Kiana Ehsani, Winson Han, Eric Kolve, Ali Farhadi, Aniruddha Kembhavi, and Roozbeh Mottaghi. ProcTHOR: Large-Scale Embodied AI Using Procedural Generation. In *NeurIPS*, 2022. 21

[24] Helisa Dhamo, Fabian Manhardt, Nassir Navab, and Federico Tombari. Graph-to-3d: End-to-end generation and manipulation of 3d scenes using scene graphs. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 16352–16361, 2021. 17

[25] Paul Engstler, Andrea Vedaldi, Iro Laina, and Christian Rupprecht. Invisible stitch: Generating smooth 3d scenes with depth inpainting. *arXiv preprint arXiv:2404.19758*, 2024. 22

[26] Chuan Fang, Yuan Dong, Kunming Luo, Xiaotao Hu, Rakesh Shrestha, and Ping Tan. Ctrl-room: Controllable text-to-3d room meshes generation with layout constraints. *arXiv preprint arXiv:2310.03602*, 2023. 22

[27] Mengyang Feng, Jinlin Liu, Miaomiao Cui, and Xuansong Xie. Diffusion360: Seamless 360 degree panoramic image generation based on diffusion models. *arXiv preprint arXiv:2311.13141*, 2023. 23

[28] Weixi Feng, Wanrong Zhu, Tsu-jui Fu, Varun Jampani, Arjun Akula, Xuehai He, Sugato Basu, Xin Eric Wang, and William Yang Wang. Layoutgpt: Compositional visual planning and generation with large language models. *Advances in Neural Information Processing Systems*, 36, 2024. 20

[29] Peter Forbrig et al. A domain-specific language for prototyping the behavior of a humanoid robot that allows the inclusion of sensor data. 2023. 26

[30] Rafail Fridman, Amit Abecasis, Yoni Kasten, and Tali Dekel. Scenescape: Text-driven consistent scene generation. *Advances in Neural Information Processing Systems*, 36, 2024. 25

[31] Huan Fu, Bowen Cai, Lin Gao, Ling-Xiao Zhang, Jiaming Wang, Cao Li, Qixun Zeng, Chengyue Sun, Rongfei Jia, Binqiang Zhao, et al. 3d-front: 3d furnished rooms with layouts and semantics. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10933–10942, 2021. 18

[32] Qiang Fu, Xiaowu Chen, Xiaotian Wang, Sijia Wen, Bin Zhou, and Hongbo Fu. Adaptive synthesis of indoor scenes via activity-associated object relation graphs. *ACM Trans. Graph.*, 36(6), November 2017. 17

[33] Rao Fu, Zehao Wen, Zichen Liu, and Srinath Sridhar. Anyhome: Open-vocabulary generation of structured and textured 3d homes. In *European Conference on Computer Vision*, pages 52–70. Springer, 2025. 20

[34] Gege Gao, Weiyang Liu, Anpei Chen, Andreas Geiger, and Bernhard Schölkopf. Graphdreamer: Compositional 3d scene synthesis from scene graphs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21295–21304, 2024. 20

[35] Lin Gao, Jia-Mu Sun, Kaichun Mo, Yu-Kun Lai, Leonidas J Guibas, and Jie Yang. Scenehgn: Hierarchical graph networks for 3d indoor scene generation with fine-grained geometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(7):8902–8919, 2023. 17

[36] Joan Giner-Miguelez et al. A domain-specific language for describing machine learning datasets. *Journal of Computer Languages*, 2022. 26

[37] Benedikt Gleißner. Domänenspezifische sprachen zur umsetzung numerischer lösungsverfahren für gewöhnliche differentialgleichungssysteme. 2019. 27

[38] IKEA Global. Ikea place app launched to help people virtually place furniture at home, 2017. https://www.ikea.com/global/en/newsroom/innovation/ikea-launches-ikea-place-a-new-app-that-allows-people-to-virtually-place-furniture-in-their-home-170912/ [Online; accessed 31-October-2024]. 10

[39] Darien Graham-Smith. What is the metaverse?, 2023. https://www.techfinitive.com/explainers/what-is-the-metaverse/ [Online; accessed 31-October-2024]. 7, 8

[40] Rohit Gupta et al. Towards a systematic engineering of industrial domain-specific language. *arXiv: Software Engineering*, 2021. 27

[41] Google Maps Help. Introducing live view, the new augmented reality feature in google maps, 2019. https://support.google.com/maps/thread/11554255/introducing-live-view-the-new-augmented-reality-feature-in-google-maps?hl=en [Online; accessed 31-October-2024]. 9

[42] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020. 24

[43] Lukas Höllein, Ang Cao, Andrew Owens, Justin Johnson, and Matthias Nießner. Text2room: Extracting textured 3d meshes from 2d text-to-image models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7909–7920, 2023. 22, 23

[44] Jon Holmes. What is xr, and how is it radically transforming industries?, 2024. https://www.autodesk.com/design-make/articles/what-is-xr [Online; accessed 31-October-2024]. 7, 8

[45] Joseph R. Horgan and Aditya P. Mathur. Software testing and reliability. *The Handbook of Software Reliability Engineering*, pages 531–565, 1996. 12

[46] Siyuan Huang, Siyuan Qi, Yixin Zhu, Yinxue Xiao, Yuanlu Xu, and Song-Chun Zhu. Holistic 3d scene parsing and reconstruction from a single rgb image. In *Proceedings of the European conference on computer vision (ECCV)*, pages 187–203, 2018. 19

[47] HuggingFace. sentence-transformers/all-mpnet-base-v2, 2021. https://huggingface.co/sentence-transformers/all-mpnet-base-v2 [Online; accessed 26-November-2024]. 80

[48] Gabriel Ilharco, Mitchell Wortsman, Ross Wightman, Cade Gordon, Nicholas Carlini, Rohan Taori, Achal Dave, Vaishaal Shankar, Hongseok Namkoong, John Miller, Hannaneh Hajishirzi, Ali Farhadi, and Ludwig Schmidt. Openclip, July 2021. https://github.com/mlfoundations/open_clip [Online; accessed 26-November-2024]. 80

[49] Amber Jackson. Top 10: Metaverse companies, 2023. https://technologymagazine.com/top10/top-10-metaverse-companies [Online; accessed 31-October-2024]. 8

[50] Xinyang Jia. The role and importance of software testing in software quality management. *Journal of Industry and Engineering Management*, 1(4):39–44, 2023. 12

[51] Chenfanfu Jiang, Siyuan Qi, Yixin Zhu, Siyuan Huang, Jenny Lin, Lap-Fai Yu, Demetri Terzopoulos, and Song-Chun Zhu. Configurable 3d scene synthesis and 2d image rendering with per-pixel ground truth using stochastic grammars. *International Journal of Computer Vision*, 126:920–941, 2018. 17

[52] Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David Shamma, Michael Bernstein, and Li Fei-Fei. Image retrieval using scene graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. 17

[53] Xiaoliang Ju, Zhaoyang Huang, Yijin Li, Guofeng Zhang, Yu Qiao, and Hongsheng Li. Diffindscene: Diffusion-based high-quality 3d

indoor scene generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4526–4535, 2024. 24

[54] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023. 24

[55] Mohd Ehmer Khan and Farmeena Khan. Importance of software testing in software development life cycle. *International Journal of Computer Science Issues (IJCSI)*, 11(2):120, 2014. 12

[56] Agi Putra Kharisma et al. Towards text-based domain-specific modeling language for representational state transfer compliant services. 2020. 27

[57] Jung Hyun Kim et al. Poe: A domain-specific language for exploitation. 2024. 26

[58] Jing Yu Koh, Harsh Agrawal, Dhruv Batra, Richard Tucker, Austin Waters, Honglak Lee, Yinfei Yang, Jason Baldridge, and Peter Anderson. Simple and effective synthesis of indoor 3d scenes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 1169–1178, 2023. 22

[59] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*, 2017. 21

[60] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A Shamma, et al. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International journal of computer vision*, 123:32–73, 2017. 17

[61] David Layzelle. The fascinating history and evolution of extended reality (xr), 2021. https://unitydevelopers.co.uk/the-fascinating-history-and-evolution-of-extended-reality-xrabcabcabcabcabc/ [Online; accessed 31-October-2024]. 7

[62] Jumin Lee, Sebin Lee, Changho Jo, Woobin Im, Juhyeong Seon, and Sung-Eui Yoon. Semcity: Semantic scene generation with triplane diffusion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 28337–28347, 2024. 24

[63] Yao-Chih Lee, Yi-Ting Chen, Andrew Wang, Ting-Hsuan Liao, Brandon Y Feng, and Jia-Bin Huang. Vividdream: Generating 3d scene with ambient dynamics. *arXiv preprint arXiv:2405.20334*, 2024. 25

[64] Haoran Li, Haolin Shi, Wenli Zhang, Wenjun Wu, Yong Liao, Lin Wang, Lik-hang Lee, and Pengyuan Zhou. Dreamscene: 3d gaussian-based text-to-3d scene generation via formation pattern sampling. *arXiv preprint arXiv:2404.03575*, 2024. 22

[65] Manyi Li, Akshay Gadi Patil, Kai Xu, Siddhartha Chaudhuri, Owais Khan, Ariel Shamir, Changhe Tu, Baoquan Chen, Daniel Cohen-Or, and Hao Zhang. Grains: Generative recursive autoencoders for indoor scenes. *ACM Transactions on Graphics (TOG)*, 38(2):1–16, 2019. 17

[66] Shuai Li and Hongjun Li. Deep generative modeling based on vae-gan for 3d indoor scene synthesis. *International Journal of Computer Games Technology*, 2023(1):3368647, 2023. 18

[67] Xinyang Li, Zhangyu Lai, Linning Xu, Yansong Qu, Liujuan Cao, Shengchuan Zhang, Bo Dai, and Rongrong Ji. Director3d: Real-world camera trajectory and 3d scene generation from text. *arXiv preprint arXiv:2406.17601*, 2024. 25

[68] Chenguo Lin and Yadong Mu. Instructscene: Instruction-driven 3d indoor scene synthesis with semantic graph prior. In *International Conference on Learning Representations (ICLR)*, 2024. 18, 86

[69] LinkedIn. How can you test augmented reality apps on different devices? https://www.linkedin.com/advice/0/how-can-you-test-augmented-reality-apps-different [Online; accessed 31-October-2024]. 12

[70] Jingyu Liu, Wenhan Xiong, Ian Jones, Yixin Nie, Anchit Gupta, and Barlas Oğuz. Clip-layout: Style-consistent indoor scene synthesis with semantic furniture embedding. *arXiv preprint arXiv:2303.03565*, 2023. 18

[71] Jiajie Lu, Canlin Li, Chao Yin, and Lizhuang Ma. A new framework for automatic 3d scene construction from text description. In *2010 IEEE International Conference on Progress in Informatics and Computing*, volume 2, pages 964–968, 2010. 16

[72] Michael R. Lyu, editor. *Handbook of software reliability engineering*. McGraw-Hill, Inc., USA, 1996. 12

[73] Rui Ma, Akshay Gadi Patil, Matthew Fisher, Manyi Li, Sören Pirk, Binh-Son Hua, Sai-Kit Yeung, Xin Tong, Leonidas Guibas, and Hao Zhang. Language-driven synthesis of 3d scenes from scene databases. *ACM Trans. Graph.*, 37(6), December 2018. 17

[74] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv: Arxiv-2310.12931*, 2023. 16

[75] Yikun Ma, Dandan Zhan, and Zhi Jin. Fastscene: Text-driven fast 3d indoor scene generation via panoramic gaussian splatting. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 1173–1181, 2024. 22, 23

[76] QA Madness. What is augmented reality and how to test it? https://www.qamadness.com/knowledge-base/what-is-augmented-reality-and-how-to-test-it/ [Online; accessed 31-October-2024]. 12

[77] Weijia Mao, Yan-Pei Cao, Jia-Wei Liu, Zhongcong Xu, and Mike Zheng Shou. Showroom3d: Text to high-quality 3d room generation using 3d priors. *arXiv preprint arXiv:2312.13324*, 2023. 22

[78] Quan Meng, Lei Li, Matthias Nießner, and Angela Dai. Lt3sd: Latent trees for 3d scene diffusion, 2024. 24

[79] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020. 24

[80] Yutaka Nagashima. Domain-specific language to encode induction heuristics. *arXiv: Logic in Computer Science*, 2019. 27

[81] Eman Negm et al. A semantic-based approach for domain specific language development. *International Journal of Power Electronics and Drive Systems*, 2024. 26

[82] Yinyu Nie, Angela Dai, Xiaoguang Han, and Matthias Nießner. Learning 3d scene priors with 2d supervision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 792–802, 2023. 18

[83] Yinyu Nie, Xiaoguang Han, Shihui Guo, Yujian Zheng, Jian Chang, and Jian Jun Zhang. Total3dunderstanding: Joint layout, object pose and mesh reconstruction for indoor scenes from a single image. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 55–64, 2020. 19, 20

[84] David Novotny, Ben Graham, and Jeremy Reizenstein. Perspectivenet: A scene-consistent image generator for new view synthesis in real indoor environments. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. 22, 23

[85] Başak Melis Öcal, Maxim Tatarchenko, Sezer Karaoglu, and Theo Gevers. Sceneteller: Language-to-3d scene generation. *arXiv preprint arXiv:2407.20727*, 2024. 20

[86] OpenAI. Gpt-4 technical report, 2024. 21

[87] Hao Ouyang, Kathryn Heal, Stephen Lombardi, and Tiancheng Sun. Text2immersion: Generative immersive scene with 3d gaussians. *arXiv preprint arXiv:2312.09242*, 2023. 22

[88] Despoina Paschalidou, Amlan Kar, Maria Shugrina, Karsten Kreis, Andreas Geiger, and Sanja Fidler. Atiss: Autoregressive transformers for indoor scene synthesis. *Advances in Neural Information Processing Systems*, 34:12013–12026, 2021. 18

[89] Songyou Peng, Michael Niemeyer, Lars Mescheder, Marc Pollefeys, and Andreas Geiger. Convolutional occupancy networks. In *European Conference on Computer Vision (ECCV)*, 2020. 25

[90] PixelQA. Testing augmented reality (ar) applications: A comprehensive guide, 2024. https://www.pixelqa.com/blog/post/testing-augmented-reality-applications [Online; accessed 31-October-2024]. 12

[91] Ryan Po and Gordon Wetzstein. Compositional 3d scene generation using locally conditioned diffusion. In *2024 International Conference on 3D Vision (3DV)*, pages 651–663. IEEE, 2024. 24

[92] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell,

Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021. 43, 86

[93] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018. 20

[94] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019. 20

[95] Alexander Raistrick, Lahav Lipson, Zeyu Ma, Lingjie Mei, Mingzhe Wang, Yiming Zuo, Karhan Kayan, Hongyu Wen, Beining Han, Yihan Wang, Alejandro Newell, Hei Law, Ankit Goyal, Kaiyu Yang, and Jia Deng. Infinite photorealistic worlds using procedural generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12630–12641, 2023. 21

[96] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. 58, 80, 87

[97] Daniel Ritchie, Kai Wang, and Yu-an Lin. Fast and flexible indoor scene synthesis via deep convolutional generative models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6182–6190, 2019. 17

[98] Georg David Ritterbusch and Malte Rolf Teichmann. Defining the metaverse: A systematic literature review. *IEEE Access*, 11:12368–12377, 2023. 8

[99] Alex Rojco et al. Phm4hhp domain specific language. 2024. 26

[100] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022. 23, 24

[101] Aymen J. Salman et al. Domain-specific languages for iot: Challenges and opportunities. 2021. 27

[102] Manolis Savva, Angel X Chang, and Maneesh Agrawala. Scenesuggest: Context-driven 3d scene design. *arXiv preprint arXiv:1703.00061*, 2017. 17

[103] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, et al. Laion-5b: An open large-scale dataset for training next generation image-text models. *Advances in Neural Information Processing Systems*, 35:25278–25294, 2022. 80

[104] Jonas Schult, Sam Tsai, Lukas Höllein, Bichen Wu, Jialiang Wang, Chih-Yao Ma, Kunpeng Li, Xiaofang Wang, Felix Wimbauer, Zijian He, et al. Controlroom3d: Room generation using semantic proxy rooms. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6201–6210, 2024. 22

[105] Arnd Sett and Karl Vollmann. Computer based robot training in a virtual environment. In *2002 IEEE International Conference on Industrial Technology, 2002. IEEE ICIT '02.*, volume 2, pages 1185–1189 vol.2, 2002. 16

[106] Lee M. Seversky and Lijun Yin. Real-time automatic 3d scene generation from natural language voice and text descriptions. In *ACM Multimedia*, 2006. 16

[107] Zifan Shi, Yujun Shen, Jiapeng Zhu, Dit-Yan Yeung, and Qifeng Chen. 3d-aware indoor scene synthesis with depth priors. In *European Conference on Computer Vision*, pages 406–422. Springer, 2022. 22

[108] Jaidev Shriram, Alex Trevithick, Lingjie Liu, and Ravi Ramamoorthi. Realmdreamer: Text-driven 3d scene generation with inpainting and depth diffusion. *arXiv preprint arXiv:2404.07199*, 2024. 22

[109] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *arXiv preprint arXiv:2010.02502*, 2020. 24

[110] Liangchen Song, Liangliang Cao, Hongyu Xu, Kai Kang, Feng Tang, Junsong Yuan, and Yang Zhao. Roomdreamer: Text-driven 3d indoor scene synthesis with coherent geometry and texture. *arXiv preprint arXiv:2305.11337*, 2023. 22

[111] Statista. Ar & vr - worldwide, 2023. https://www.statista.com/outlook/amo/ar-vr/worldwide [Online; accessed 14-November-2024]. 12

[112] Statista. Mobile augmented reality (ar) users worldwide from 2023 to 2028, 2024. https://www.statista.com/statistics/1098630/global-mobile-augmented-reality-ar-users/ [Online; accessed 14-November-2024]. 12

[113] Strivr. What is virtual reality? vr usage examples at work. https://www.strivr.com/blog/what-is-vr [Online; accessed 31-October-2024]. 8

[114] Jiaming Sun, Yiming Xie, Linghao Chen, Xiaowei Zhou, and Hujun Bao. Neuralrecon: Real-time coherent 3d reconstruction from monocular video. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 15598–15607, 2021. 22

[115] Jiapeng Tang, Yinyu Nie, Lev Markhasin, Angela Dai, Justus Thies, and Matthias Nießner. Diffuscene: Denoising diffusion models for gerative indoor scene synthesis. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*, 2024. 18

[116] Luming Tang, Menglin Jia, Qianqian Wang, Cheng Perng Phoo, and Bharath Hariharan. Emergent correspondence from image diffusion. *Advances in Neural Information Processing Systems*, 36:1363–1389, 2023. 22

[117] Dan Tappan. Knowledge-based spatial reasoning for scene generation from text descriptions. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, page 1888–1889. AAAI Press, 2008. 16

[118] Alexander Treschev and Oksana Pekach. Augmented reality applications testing 101: A practical guide, 2014. https://mobidev.biz/blog/how-test-augmented-reality-applications [Online; accessed 31-October-2024]. 12

[119] Maneela Tuteja, Gaurav Dubey, et al. A research study on importance of testing and quality assurance in software development life cycle (sdlc) models. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(3):251–257, 2012. 12

[120] Azeem Uddin and Abhineet Anand. Importance of software testing in the process of software development. *International Journal for Scientfic Research and Development*, 12(6), 2019. 12

[121] Milica Vuković et al. Domain-specific language for modeling fluent api. 2023. 27

[122] Kai Wang, Yu-An Lin, Ben Weissmann, Manolis Savva, Angel X. Chang, and Daniel Ritchie. Planit: planning and instantiating indoor scenes with relation graph and spatial prior networks. *ACM Trans. Graph.*, 38(4), July 2019. 17

[123] Kai Wang, Manolis Savva, Angel X Chang, and Daniel Ritchie. Deep convolutional priors for indoor scene synthesis. *ACM Transactions on Graphics (TOG)*, 37(4):70, 2018. 17

[124] Xinpeng Wang, Chandan Yeshwanth, and Matthias Nießner. Sceneformer: Indoor scene generation with transformers. In *2021 International Conference on 3D Vision (3DV)*, pages 106–115. IEEE, 2021. 18

[125] Qiuhong Anna Wei, Sijie Ding, Jeong Joon Park, Rahul Sajnani, Adrien Poulenard, Srinath Sridhar, and Leonidas Guibas. Lego-net: Learning regular rearrangements of objects in rooms. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 19037–19047, 2023. 18

[126] Yao Wei, Martin Renqiang Min, George Vosselman, Li Erran Li, and Michael Ying Yang. Planner3d: Llm-enhanced graph prior meets 3d indoor scene explicit regularization. *arXiv preprint arXiv:2403.12848*, 2024. 17

[127] Zhennan Wu, Yang Li, Han Yan, Taizhang Shang, Weixuan Sun, Senbo Wang, Ruikai Cui, Weizhe Liu, Hiroyuki Sato, Hongdong Li, et al. Blockfusion: Expandable 3d scene generation using latent tri-plane extrapolation. *ACM Transactions on Graphics (TOG)*, 43(4):1–17, 2024. 24

[128] Jiale Xu, Xintao Wang, Weihao Cheng, Yan-Pei Cao, Ying Shan, Xiaohu Qie, and Shenghua Gao. Dream3d: Zero-shot text-to-3d synthesis using 3d shape prior and text-to-image diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20908–20918, 2023. 24

[129] Rui Xu, Le Hui, Yuehui Han, Jianjun Qian, and Jin Xie. Scene graph masked variational autoencoders for 3d scene generation. In *Proceedings of the 31st ACM International Conference on Multimedia*, MM '23, page 5725–5733, New York, NY, USA, 2023. Association for Computing Machinery. 17

[130] Roman V. Yampolskiy. *AI: Unexplainable, Unpredictable, Uncontrollable*. Chapman and Hall/CRC, 1st edition, 2024. 19

[131] Haitao Yang, Zaiwei Zhang, Siming Yan, Haibin Huang, Chongyang Ma, Yi Zheng, Chandrajit Bajaj, and Qixing Huang. Scene synthesis via uncertainty-driven attribute synchronization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5630–5640, 2021. 17

[132] Ming-Jia Yang, Yu-Xiao Guo, Bin Zhou, and Xin Tong. Indoor scene generation from a collection of semantic-segmented depth images. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 15203–15212, 2021. 19

[133] Xinyan Yang, Fei Hu, Long Ye, Zhiming Chang, and Jiyin Li. A system of configurable 3d indoor scene synthesis via semantic relation learning. *Displays*, 74:102168, 2022. 17

[134] Yandan Yang, Baoxiong Jia, Peiyuan Zhi, and Siyuan Huang. Physcene: Physically interactable 3d scene synthesis for embodied ai. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16262–16272, 2024. 18

[135] Yixuan Yang, Junru Lu, Zixiang Zhao, Zhen Luo, James JQ Yu, Victor Sanchez, and Feng Zheng. Llplace: The 3d indoor scene layout generation and editing via large language model. *arXiv preprint arXiv:2406.03866*, 2024. 20

[136] Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, et al. Holodeck: Language guided generation of 3d embodied ai environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16227–16237, 2024. 20, 21, 42, 59, 80, 86

[137] Zhaoda Ye, Yang Liu, and Yuxin Peng. Maan: Memory-augmented auto-regressive network for text-driven 3d indoor scene generation. *IEEE Transactions on Multimedia*, pages 1–14, 2024. 18

[138] Hong-Xing Yu, Haoyi Duan, Charles Herrmann, William T Freeman, and Jiajun Wu. Wonderworld: Interactive 3d scene generation from a single image. *arXiv preprint arXiv:2406.09394*, 2024. 22

[139] Hong-Xing Yu, Haoyi Duan, Junhwa Hur, Kyle Sargent, Michael Rubinstein, William T Freeman, Forrester Cole, Deqing Sun, Noah

Snavely, Jiajun Wu, et al. Wonderjourney: Going from anywhere to everywhere. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6658–6667, 2024. 25

[140] Xuening Yuan, Hongyu Yang, Yueming Zhao, and Di Huang. Dreamscape: 3d scene creation via gaussian splatting joint correlation modeling. *arXiv preprint arXiv:2404.09227*, 2024. 24

[141] Ilwi Yun, Chanyong Shin, Hyunku Lee, Hyuk-Jae Lee, and Chae Eun Rhee. Egformer: Equirectangular geometry-biased transformer for 360 depth estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6101–6112, 2023. 23

[142] Guangyao Zhai, Evin Pınar Örnek, Dave Zhenyu Chen, Ruotong Liao, Yan Di, Nassir Navab, Federico Tombari, and Benjamin Busam. Echoscene: Indoor scene generation via information echo over scene graph diffusion. In *European Conference on Computer Vision*, pages 167–184. Springer, 2025. 18

[143] Guangyao Zhai, Evin Pınar Örnek, Shun-Cheng Wu, Yan Di, Federico Tombari, Nassir Navab, and Benjamin Busam. Commonscenes: Generating commonsense 3d indoor scenes with scene graphs. *Advances in Neural Information Processing Systems*, 36, 2024. 18

[144] Jingbo Zhang, Xiaoyu Li, Ziyu Wan, Can Wang, and Jing Liao. Text2nerf: Text-driven 3d scene generation with neural radiance fields. *IEEE Transactions on Visualization and Computer Graphics*, 2024. 22

[145] Qihang Zhang, Chaoyang Wang, Aliaksandr Siarohin, Peiye Zhuang, Yinghao Xu, Ceyuan Yang, Dahua Lin, Bolei Zhou, Sergey Tulyakov, and Hsin-Ying Lee. Scenewiz3d: Towards text-guided 3d scene composition. *arXiv preprint arXiv:2312.08885*, 2023. 24

[146] Song-Hai Zhang, Shao-Kui Zhang, Wei-Yu Xie, Cheng-Yang Luo, Yong-Liang Yang, and Hongbo Fu. Fast 3d indoor scene synthesis by learning spatial relation priors of objects. *IEEE Transactions on Visualization and Computer Graphics*, 28(9):3082–3092, 2022. 17

[147] Zaiwei Zhang, Zhenpei Yang, Chongyang Ma, Linjie Luo, Alexander Huth, Etienne Vouga, and Qixing Huang. Deep generative modeling for scene synthesis via hybrid representations. *ACM Transactions on Graphics (TOG)*, 39(2):1–21, 2020. 18

[148] Yiqun Zhao, Zibo Zhao, Jing Li, Sixun Dong, and Shenghua Gao. Roomdesigner: Encoding anchor-latents for style-consistent and shape-compatible indoor scene generation. In *2024 International Conference on 3D Vision (3DV)*, pages 1413–1423. IEEE, 2024. 18

[149] Shijie Zhou, Zhiwen Fan, Dejia Xu, Haoran Chang, Pradyumna Chari, Tejas Bharadwaj, Suya You, Zhangyang Wang, and Achuta Kadambi. Dreamscene360: Unconstrained text-to-3d scene generation with panoramic gaussian splatting. In *European Conference on Computer Vision*, pages 324–342. Springer, 2025. 22

[150] Xiaoyu Zhou, Xingjian Ran, Yajiao Xiong, Jinlin He, Zhiwei Lin, Yongtao Wang, Deqing Sun, and Ming-Hsuan Yang. Gala3d: Towards text-to-3d complex scene generation via layout-guided generative gaussian splatting. *arXiv preprint arXiv:2402.07207*, 2024. 18

[151] Yang Zhou, Zachary While, and Evangelos Kalogerakis. Scenegraphnet: Neural message passing for 3d indoor scene augmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7384–7392, 2019. 18

[152] Henry Zhu et al. A domain-specific language for reconfigurable, distributed software. *International Journal of Networking and Computing*, 2024. 26