# The Scene Language: Representing Scenes with Programs, Words, and Embeddings: Report

Ayush Bodade
ayushbodade1@gmail.com

November 2024

**Abstract**

This report investigates the work: "The Scene Language: Representing Scenes with Programs, Words, and Embeddings" highlighting the limitations, key optimizations and potential extensions.

## Contents

# Introduction

**Problem Statement:** Accurate representation of visual scenes has remained a core challenge in computer vision, particularly for high-fidelity and editable generation tasks. Current representations face several limitations:

- **Scene Graphs:** Limited by coarse granularity and inability to encode fine details.
- **Generative Latent Spaces:** Lack interpretability and compositionality.
- **Hybrid Models:** Struggle to balance scalability and fidelity.

**Objective:** The Scene Language addresses these challenges by unifying:

- Structural hierarchy via **programs**.
- Semantic abstraction using **natural language**.
- Instance-level visual attributes with **neural embeddings**.

## Contributions:

1. Introduction of a unified, multimodal representation.
2. A training-free method for scene inference using pre-trained language models (LMs).
3. Compatibility with a variety of rendering pipelines for versatile visual output.
4. Demonstration of applications in 3D and 4D scene generation, editing, and reconstruction.

# Methodology

**Scene Representation:** The Scene Language describes a scene $s$ as $\Phi(s) = (W, P, Z)$, where:

- **Words ($W$):** A collection of semantic labels representing entity categories (e.g., "chair," "tree").
- **Programs ($P$):** A set of entity functions $f_w$, parameterized by embeddings and used to describe hierarchical scene structure. Each program encodes:
  - Relationships between entities (e.g., alignment or repetition).
  - Spatial transformations (e.g., translations, rotations).
- **Embeddings ($Z$):** Neural representations capturing attributes like geometry, texture, and color for each instance.

**Domain-Specific Language (DSL):** The representation is implemented as a DSL, enabling:

- **Macro Definitions:** For transformations (`translate, scale, rotate`) and compositions (`union, transform`).
- **Hierarchical Modeling:** Programs describe scenes as computation graphs, with nodes representing entities and edges denoting spatial or semantic relations.
- **Interoperability:** The DSL is compatible with various rendering engines.

**Training-Free Inference:** The Scene Language leverages pre-trained language models (e.g., Claude 3.5) for generating $W$ and $P$. Two pipelines are implemented:

- **Text-Conditioned Generation:**
  - **Input:** Descriptions such as "A chessboard at game start."
  - **Output:** Programs specifying semantic classes, spatial layout, and relationships.
- **Image-Conditioned Generation:**
  - **Input:** Annotated images.
  - **Process:** Segmentation (via GroundingSAM) and embedding extraction (via Textual Inversion).

**Rendering Pipeline:** Scenes are rendered by:

1. **Program Interpretation:** Evaluating programs to generate an object hierarchy.
2. **Graphics Rendering:** Translating Scene Language outputs into visual images using renderers such as:
   - Neural renderers (e.g., Gaussian splatting).
   - Asset-based renderers (e.g., Minecraft).
   - Diffusion-based models (e.g., MIGC).

# Limitations

## Problems:

**Problem 1: Computational Overhead:** **Issue:** Recursive program execution introduces significant computational complexity, particularly for large scenes with numerous entities.
**Impact:** Latency in real-time or interactive environments.

**Optimization:** Parallelized Program Execution
**Idea:** Evaluate independent sub-graphs of the scene hierarchy simultaneously.
**Execution:**

- Use parallel processing frameworks like CUDA or OpenMP.
- Divide scene graphs into disjoint subsets that can be processed independently.

**Example:** Render chessboard squares (8x8 grid) in parallel instead of sequentially evaluating transformations.
**Benefit:** Reduces latency, enabling real-time rendering for complex scenes.

**Optimization:** Lazy Evaluation
**Idea:** Generate only the entities currently visible or interacted with in the scene.
**Execution:**

- Add a visibility condition $V(h)$, where entities $h$ are generated if $V(h) = \text{True}$.
- Use camera frustum culling techniques to determine visibility dynamically.

**Benefit:** Saves computational resources for high-density environments (e.g., forests, cities).

**Problem 2: Granularity Limitations:** **Issue:** Neural embeddings $Z$ lack fine-grained details for high-frequency textures and intricate geometry.
**Impact:** Limits photorealism in generated scenes.
**Optimization:** Multi-Scale Embeddings
**Idea:** Extend embeddings to represent coarse-to-fine features hierarchically.
**Execution:**

- Create a pyramid of embeddings $Z = \{z_{\text{coarse}}, z_{\text{mid}}, z_{\text{fine}}\}$.
- Use $z_{\text{coarse}}$ for macro geometry and $z_{\text{fine}}$ for details like textures or small deformations.

**Benefit:** Achieves photorealism without sacrificing interpretability.

**Optimization:** Procedural Detail Injection
**Idea:** Dynamically enhance embeddings during rendering.
**Execution:**

- Use procedural generation algorithms (e.g., Perlin noise, fractals) to synthesize high-frequency details at render time.
- Embed procedural parameters into $Z$ to allow customization.

**Benefit:** Generates intricate, realistic details (e.g., cloth wrinkles, tree bark).

**Problem 3: Scalability:** **Issue:** Large, complex scenes increase program size exponentially.
**Impact:** Difficulty in debugging, storage, and reuse.

**Optimization:** Function Abstraction and Reuse
**Idea:** Modularize repetitive components into reusable sub-programs.
**Execution:**

- Abstract patterns (e.g., grids, circles) into high-level functions.
- Example: Use a generate_grid$(rows, cols)$ function for any grid-based scene.

**Benefit:** Reduces code duplication, simplifies debugging, and improves scalability.

**Optimization:** Program Caching
**Idea:** Cache intermediate results of frequently reused sub-programs.
**Execution:**

- Store pre-computed entities (e.g., "a tree model") and transformations in a scene cache.
- Retrieve cached results during subsequent program evaluations.

**Benefit:** Minimizes redundant computation, enhancing efficiency.

## Problem 4: Realism and Physics Integration:

**Issue:** Lack of physical attributes in embeddings limits realism in simulations.
**Impact:** Unrealistic scenes for robotics, gaming, or training environments.
**Optimization:** Embedding Physical Properties
**Idea:** Extend embeddings $Z$ to encode physical attributes like mass, elasticity, and material type.
**Execution:**

- Update embeddings as $Z = \{z_{\text{visual}}, z_{\text{physical}}\}$, where:
  - $z_{\text{visual}}$: Geometry, texture.
  - $z_{\text{physical}}$: Material properties for physics engines.
- Example: Use friction values in $z_{\text{physical}}$ for collision simulations.

**Benefit:** Supports dynamic simulations like falling objects or responsive surfaces.

**Optimization:** Coupling with Physics Engines
**Idea:** Integrate Scene Language with physics engines like PyBullet or NVIDIA PhysX.
**Execution:**

- Add physical simulation modules that interpret $z_{\text{physical}}$.
- Simulate gravity, collision, and deformation in real-time.

**Benefit:** Enables realistic and interactive scenes for robotics and training applications.

## Problem 5: Modalities:

**Issue:** Limited to visual representations; excludes sound and tactile properties.

**Impact:** Misses opportunities in AR/VR and multimodal content creation.
**Optimization:** Multimodal Embeddings
**Idea:** Expand embeddings $Z$ to include audio and tactile properties.
**Execution:**

- Encode audio features (e.g., pitch, duration) as $z_{\text{audio}}$.
- Encode tactile features (e.g., hardness, texture) as $z_{\text{tactile}}$.

**Example:** A "glass object" embedding includes $z_{\text{audio}}$ for shattering sound and $z_{\text{tactile}}$ for smoothness.
**Benefit:** Broadens use in immersive technologies like AR/VR.

# Applications and Future Directions

## Integration of Physics:

- **Objective:** Combine Scene Language with physical simulation engines.
- **Outcome:** Enables realistic and interactive scenes for applications across industries by incorporating real-world physics.

## Real-Time Optimization:

- **Objective:** Optimize neural renderers for faster rendering.
- **Outcome:** Enhances efficiency for interactive applications such as gaming, virtual reality (VR), and augmented reality (AR).

## Multimodal Scene Fusion:

- **Objective:** Extend representations to include audio and tactile elements.
- **Outcome:** Facilitates multi-sensory experiences, enhancing realism and immersion in virtual environments.

## Collaborative Environments:

- **Objective:** Enable multi-agent scene generation in shared virtual spaces.
- **Outcome:** Supports real-time collaboration in virtual spaces by multiple users or agents.

# Domain-Specific Applications

**Robotics:** Simulation for navigation and manipulation. Generate structured environments to train robots for tasks such as object picking or assembly. Ex: Scenes with specific physics properties

to simulate real-world tasks with additional modalities like audio and tactile elements.

**Scene Language on the Edge:** Build dynamic, multimodal environments with realistic interactions on Edge devices. For this the engine has to be lightweight if it has to run on the edge, perhaps how to effectively project it on a smaller dimension couuld be explored.

# Extensions to Scene Language

## Hybrid Scene Representation:

- **Concept:** Combine Scene Language with Neural Radiance Fields (NeRF).
- **Implementation:** Use high-level programs to define scene structures and NeRF for photorealistic rendering.
- **Outcome:** Achieves a balance between interpretable scene programming and high-quality rendering.

## Dynamic 4D Multimodal Scenes:

- **Concept:** Extend Scene Language for time-varying transformations and multimodal outputs.
- **Implementation:**
  - Use temporal embeddings for animations.
  - Integrate sound and tactile feedback in scene generation.
- **Example:** A city scene with moving cars, ambient sound, and tactile textures.

## Procedural Universe Generation:

- **Concept:** Scale Scene Language to generate large-scale ecosystems or galaxies.
- **Implementation:**
  - Define high-level patterns for star clusters or planetary systems.
  - Encode planetary details like climate, terrain, and ecosystems.
- **Outcome:** Creates infinite, procedurally generated universes for gaming, simulations, or scientific modeling.