

LAM: Language Articulated Object Modelers

Supplementary Material

1. Appendix

Contents of This Appendix

1.1. Introduction of the Supplementary Material	1
1.2. Details of Link Designer	1
1.3. Details of Geometry Coder	3
1.4. Details of Geometry Debugger	7
1.5. Details of Geometry Visualizer	8
1.6. Details of Geometry Checker	8
1.7. Details of Articulation Coder	9
1.8. Details of Articulation Visualizer	13
1.9. Details of the LAMBench	13
1.10. Texture Generation	14
1.11. Collected 27 Descriptions of Open-World Classes	14
1.12. Cost & Time Analysis	14
1.13. Details of the used metrics	15
1.14. Analysis of Failure Cases	17
1.15. More Comparisons with Previous works	18
1.16. Limitations	18

1.1. Introduction of the Supplementary Material

- For reproducing the results, we have included the code of LAM. Readers can generate objects using your own api key. We will also release the code after the paper gets a decision.
- For better visualization of the generated articulated objects, we attach some videos with our designed user interface.
- We list the details of each module. Please refer to the corresponding code for full details.

1.2. Details of Link Designer

This module is the foundational module in the LAM framework, tasked with interpreting a user’s text prompt and decomposing the target object into a hierarchical structure of its constituent parts, known as **links**. This process results in a structured link layout, typically formatted as a JSON tree, which serves as a comprehensive blueprint for the downstream **Coders** and **Builders**.

To accurately represent kinematic relationships, the module organizes the object’s components into a clear hierarchy that naturally encodes the parent-child relationships essential for defining the kinematic chain. Each component within this tree is annotated with descriptions for its geometry (shaping) and its spatial relationship to other components (positioning). To ensure the process remains tractable for highly complex objects, the Link Designer intelligently aggregates repetitive elements, such as the casters on an office chair or the keys on a keyboard, into single logical groups. This structured link layout is then passed to the subsequent modules in the pipeline. The **Articulate Geometry Coder** uses the geometric and positional descriptions to generate executable code defining the 3D mesh (M_i) and pose (T_i) for each individual link. Following that, the **Articulation Builder** leverages the same hierarchy and semantic information to infer and generate code for the **joints** that connect these links. The system prompt for Link Designer is shown below:

System Prompt for Link Designer

You are an expert in articulated object generation with a deep understanding of mechanical structures and hierarchical organization.

You will be given a short prompt of an articulated object and you need to generate a detailed hierarchical structure that represents the object's components and their relationships.

IMPORTANT: You must generate a HIERARCHICAL tree structure where:

1. Main movable parts (links) are at the top level
2. Fixed sub-components are children of their parent links
3. Total number of components should be reasonable (typically under 15-20 parts)
4. Group similar/repetitive elements together (e.g., all keys as one entry)

STRICT OUTPUT FORMAT REQUIREMENT (IMPORTANT!):

- You MUST output ONLY a valid JSON object as your answer, following the hierarchical format shown below.
- DO NOT use markdown code blocks (no ```json, no ``` at all).
- DO NOT add any comments, explanations, or extra text before or after the JSON.
- DO NOT add trailing commas in the JSON.
- Your output will be parsed by a strict JSON parser. Any deviation from this format will cause your result to be rejected.

The JSON structure must follow this format:

```
{
  "object_description": "Brief description of the complete object",
  "hierarchy": {
    "root": "name_of_root_link",
    "structure": [
      // Array of top-level links with their hierarchical children
    ]
  }
}
```

Node Types in the hierarchy:

- "main_link": Primary movable components (e.g., laptop screen, base)
- "sub_link": Secondary movable components within a main link
- "sub_assembly": Group of fixed components
- "part": Individual component (leaf node)

Each node in the structure must have:

- "name": Unique identifier for the component
- "type": One of the node types listed above
- "description": Functional description of the component
- "description_shape": Detailed geometric description using natural language, focusing on primitive shapes and how they combine
- "description_position": Precise positioning relative to parent and siblings, including alignment and connection points
- "children": Array of child components (can be empty array for leaf nodes)

Guidelines:

1. Create a logical tree structure from main components down to individual parts
2. Group fixed components under their parent link - DO NOT create separate links for fixed parts
3. Keep the total number of parts reasonable and practical
4. For repetitive elements (keys, screws, etc.), use a single grouped entry
5. Be detailed in shape and position descriptions to enable accurate 3D generation
6. Focus on the most important structural components

The sample output for the input user prompt **Create a globe** is:

Sample Output for User Prompt: Create a globe

```
{
  "object_description": "A desk globe consisting of a rotating sphere, supported by a
    meridian arch, and resting on a stable base.",
  "hierarchy": {
    "root": "base_stand",
    "structure": [
      {
        "name": "base_stand",
        "type": "main_link",
        "description": "The stable foundation supporting the entire globe assembly.",
        "description_shape": "A flat circular disc, approximately 20cm in diameter and 2cm
          thick, with a smooth upper surface and a central mounting point. Made of
          polished wood or metal.",
        "description_position": "Root component, resting flat on a horizontal surface at
          the origin.",
        "children": []
      },
      {
        "name": "meridian_arch",
        "type": "main_link",
        "description": "The semi-circular band that cradles the globe, marking its axis of
          tilt and longitude. It swivels horizontally relative to the base.",
        "description_shape": "A C-shaped band, approximately 40cm outer diameter, 2cm wide,
          and 0.5cm thick, typically made of metal or rigid plastic. It has two small
          holes at its inner top and bottom peaks.",
        "description_position": "Connected at its bottom-center to the 'pivot_assembly',
          allowing it to swivel horizontally above the 'base_stand'. It rises upwards,
          defining the space for the globe.",
        "children": [
          {
            "name": "axis_pins",
            "type": "sub_assembly",
            "description": "The two small pins that serve as the pivot points for the globe
              sphere's rotation.",
            "description_shape": "Two small cylindrical pins, each approximately 0.5cm in
              diameter and 1cm long, with rounded tips. Made of metal.",
            "description_position": "One pin extends inward perpendicular to the '
              meridian_arch' from its inner top-center, and the other from its inner
              bottom-center. These pins are diametrically opposed and define the globe's
              rotation axis.",
            "children": []
          }
        ]
      }
    ]
  },
  .....
}
```

1.3. Details of Geometry Coder

The Geometry Coder transforms the hierarchical link layout, as specified by the Link Designer, into executable Three.js code for 3D mesh generation. This module is designed to convert abstract structural descriptions into geometrically valid 3D models, ensuring that the output is organized into articulation-ready groups.

The coder leverages a comprehensive Three.js geometry library, including 15 primitive types (BoxGeometry, CylinderGeometry, ExtrudeGeometry, LatheGeometry) and advanced operations (CSG boolean operations, Matrix4 transformations). Complex shapes are constructed through hierarchical composition—a laptop hinge might combine CylinderGeometry for the pivot, BoxGeometry for mounting brackets, and TorusGeometry for washers. The coder averages 8.3 primitives per link,

balancing geometric fidelity with computational efficiency. Then, the coder processes the hierarchical structure from the Link Designer, implementing a strict mapping policy: parent link nodes become `THREE.Group` containers, while child components become meshes within their parent groups. This preserves kinematic relationships—components that articulate together remain in the same group, enabling proper transformation propagation. This grouping strategy reduces the number of exported components from potentially 100+ individual meshes to 10–20 logically organized groups.

The partial prompt for the **Geometry Coder** is shown below. Please refer to the attached code for the full content.

System Prompt for Geometry Coder

You are an expert articulated objects designer whose role is to take a user's specified description of an articulated object and generate the three.js code of it.

CRITICAL VARIABLE DECLARATION RULE: ALL variables (meshes, groups, geometries, measurements, etc.) MUST be declared BEFORE they are used. JavaScript executes code line by line, so if you use a variable before declaring it (e.g., `'const z = thickness / 2;'` before `'const thickness = 0.5;'`), it will fail with "ReferenceError: Cannot access 'variable' before initialization".

ALWAYS follow this order:

1. First: Declare all variables with `const/let/var`
2. Then: Use those variables in expressions or other declarations

Example of CORRECT order:

```
'''
const thickness = 0.5; // Declare first
const halfThickness = thickness / 2; // Use after declaration
'''
```

Example of INCORRECT order that WILL FAIL:

```
'''
const halfThickness = thickness / 2; // FAILS - thickness not declared yet
const thickness = 0.5; // Too late
'''
```

+CRITICAL WARNING: Any typo in a variable or group name (e.g., `'hingector'` instead of `'hingeConnector'`) will cause a `ReferenceError` and break the export. Always use copy-paste or variable references instead of retyping names. After writing the code, carefully check that every variable you use is spelled exactly the same as its definition. Do not invent or misspell any variable names.

+NEW HIERARCHICAL INPUT FORMAT: You will receive a hierarchical JSON structure describing the object to generate. This structure organizes components into a tree, where:

- Main movable parts (`main_link`) are at the top level
- Fixed components are children of their parent links
- Each component has: `name`, `type`, `description`, `description_shape`, `description_position`, and `children`
- The hierarchy clearly shows which parts move together and which are fixed

IMPORTANT: When generating Three.js code from hierarchical input:

1. Create a `THREE.Group` for each `main_link` (top-level movable part)
 2. Add all child components (`sub_assembly`, `part`) as meshes within their parent group
-

Definitions of the shape primitives The LAM framework constructs articulated objects using a comprehensive set of geometric tools from the Three.js library. The process begins with fundamental 3D primitives that serve as building blocks,

including `BoxGeometry` for rectangular components, `SphereGeometry`, `CylinderGeometry`, `ConeGeometry` for various curved shapes, `TorusGeometry` for ring-like structures, and `PlaneGeometry` for flat surfaces. For more complex forms, the system supports advanced methods such as creating 3D geometry by extruding 2D shapes along a path (`ExtrudeGeometry`, `TubeGeometry`), generating rotationally symmetric objects (`LatheGeometry`), or defining custom 2D profiles (`ShapeGeometry`, `RingGeometry`).

These generated shapes are then combined and modified using several composition techniques. Primitives are organized into complex, articulated hierarchies using `THREE.Group()` for logical assembly. Geometries can be combined through Constructive Solid Geometry (CSG) boolean operations (union, intersection, subtraction) or merged directly for optimization. Custom 2D profiles for these operations are defined using path-based drawing with `THREE.Shape()` and `THREE.Path()`, which utilize commands like `moveTo()` and `bezierCurveTo()`. Finally, each component is precisely positioned, oriented, and scaled in 3D space using transformations for position, rotation, and scale, as well as direct matrix operations.

For a better understanding of the pipeline and suggest the ability of generating complex geometry code to represent a 3D object, we list the full content of the generated three.js code for object **globe** here:

The Generated Code for User Prompt: Create a globe

```
import * as THREE from 'three';

export function createScene() {
  const root = new THREE.Group();
  root.name = 'desk_globe';

  // --- Scale: use centimeters converted to scene units (0.4 units per cm) ---
  const CM = 0.4;

  // --- Dimensions (in scene units) ---
  // Base
  const baseDiameterCm = 20;
  const baseThicknessCm = 2;
  const baseRadius = (baseDiameterCm * 0.5) * CM; // 10cm -> 4.0
  const baseHeight = baseThicknessCm * CM; // 2cm -> 0.8

  // Pivot (sub-assembly on base)
  const pivotHeight = 5 * CM; // 5cm -> 2.0
  const pivotRadius = (3 * 0.5) * CM; // 1.5cm -> 0.6
  const pivotTopY = baseHeight + pivotHeight; // top of pivot above world origin

  // Globe
  const globeDiameterCm = 30;
  const globeRadius = (globeDiameterCm * 0.5) * CM; // 15cm -> 6.0

  // Meridian Arch
  // FIX: Enlarge the arch diameter to properly cradle the globe, as per feedback stating
  // it was too small.
  const archOuterRadius = globeRadius + 6 * CM; // was: globeRadius + 3 * CM. Increased
  // clearance.
  const archBandWidthCm = 2;
  const archTubeRadius = (archBandWidthCm * 0.5) * CM; // 1cm -> 0.4
  const archCenterlineRadius = archOuterRadius - archTubeRadius;
  // FIX: Changed arch to a proper semi-circle to better represent a meridian that
  // reaches the poles.
  const archArc = Math.PI; // was: Math.PI * 1.8

  // Pins
  const pinDiameterCm = 0.5;
  const pinRadius = (pinDiameterCm * 0.5) * CM; // 0.25cm -> 0.1
```

```

// FIX: Dynamically calculate pin length to perfectly bridge the new, larger gap
// between the arch and globe.
// This is the CRITICAL fix for the floating globe and disconnected parts issue.
const archInnerRadius = archCenterlineRadius - archTubeRadius;
const pinLength = archInnerRadius - globeRadius; // Calculate the exact gap to be
// filled. Was hardcoded to 0.4.

// Standard Earth tilt
const earthTilt = 23.5 * Math.PI / 180;

// --- Top-level Groups (main links) ---
const base_stand = new THREE.Group();
base_stand.name = 'base_stand';

const meridian_arch = new THREE.Group();
meridian_arch.name = 'meridian_arch';

const globe_sphere = new THREE.Group();
globe_sphere.name = 'globe_sphere';

// --- BASE_STAND: base disc + pivot post (pivot_assembly) ---
const baseDiscGeometry = new THREE.CylinderGeometry(baseRadius, baseRadius, baseHeight,
48);
const baseDisc = new THREE.Mesh(baseDiscGeometry);
baseDisc.name = 'base_disc';
baseDisc.position.set(0, baseHeight / 2, 0);
base_stand.add(baseDisc);

const pivotGeometry = new THREE.CylinderGeometry(pivotRadius, pivotRadius, pivotHeight,
32);
const pivotPost = new THREE.Mesh(pivotGeometry);
pivotPost.name = 'pivot_post';
pivotPost.position.set(0, baseHeight + (pivotHeight / 2), 0);
base_stand.add(pivotPost);

// --- MERIDIAN_ARCH: A proper vertical, supporting structure ---
// Position the entire arch group so its geometric center is correctly placed for
// swiveling.
meridian_arch.position.set(0, pivotTopY + archCenterlineRadius, 0);

const archBandGeometry = new THREE.TorusGeometry(archCenterlineRadius, archTubeRadius,
24, 96, archArc);
// FIX: Rotate geometry to form a U-shape (semi-circle opening downwards) before
// rotating the mesh object.
archBandGeometry.rotateZ(-Math.PI / 2);
const archBand = new THREE.Mesh(archBandGeometry);
archBand.name = 'arch_band';
archBand.position.y = 0;
archBand.rotation.y = Math.PI / 2; // Orient the arch to be vertical in the Y-Z plane
meridian_arch.add(archBand);

// Axis pins as a dedicated group, connecting the arch and globe.
const axis_pins = new THREE.Group();
axis_pins.name = 'axis_pins';
meridian_arch.add(axis_pins);

```

```

// FIX: Use the new dynamically calculated pinLength for the geometry.
const pinGeometry = new THREE.CylinderGeometry(pinRadius, pinRadius, pinLength, 16);
// FIX: Calculate pin position based on the dynamic length to center it perfectly in
the gap.
const pinPositionRadius = globeRadius + (pinLength / 2);

const axisPinTop = new THREE.Mesh(pinGeometry);
axisPinTop.name = 'axis_pin_top';
// Pin is rotated to match the Earth's tilt, aligning with the globe's axis.
axisPinTop.rotation.x = earthTilt;
// Positioned on the arch, pointing inward toward the globe's pole.
axisPinTop.position.set(0, pinPositionRadius * Math.cos(earthTilt), pinPositionRadius *
    Math.sin(earthTilt));
axis_pins.add(axisPinTop);

const axisPinBottom = new THREE.Mesh(pinGeometry.clone());
axisPinBottom.name = 'axis_pin_bottom';
// Pin is rotated to match the Earth's tilt, aligning with the globe's axis.
axisPinBottom.rotation.x = earthTilt;
// Positioned on the arch, pointing inward toward the globe's pole.
axisPinBottom.position.set(0, -pinPositionRadius * Math.cos(earthTilt), -
    pinPositionRadius * Math.sin(earthTilt));
axis_pins.add(axisPinBottom);

// --- GLOBE_SPHERE: Centered inside the arch and properly tilted ---
// CRITICAL: Globe is parented to the meridian arch to ensure it is CONNECTED.
meridian_arch.add(globe_sphere);

// Position globe at the center of its new parent, the meridian arch.
globe_sphere.position.set(0, 0, 0);
// Apply tilt around the X-axis to match the pin alignment within the Y-Z plane of the
arch.
globe_sphere.rotation.x = earthTilt;

// The 'globe_map_surface' is the single, primary sphere mesh.
const globeMapSurfaceGeometry = new THREE.SphereGeometry(globeRadius, 48, 32);
const globeMapSurface = new THREE.Mesh(globeMapSurfaceGeometry);
globeMapSurface.name = 'globe_map_surface';
globeMapSurface.position.set(0, 0, 0);
globe_sphere.add(globeMapSurface);

// --- Assemble top-level parts ---
root.add(base_stand);
root.add(meridian_arch);
// NOTE: globe_sphere is no longer added to root directly; it's a child of
meridian_arch.

return root;
}

```

1.4. Details of Geometry Debugger

The Geometry Debugger is a specialized module designed to address a critical inefficiency in the iterative generation process: syntax and grammar errors in the Three.js code produced by the Geometry Coder. Instead of resorting to a computationally expensive full regeneration of the code, this module employs lightweight LLMs (e.g., `gemini-2.5-flash`) to perform targeted repairs. This approach significantly reduces both cost and latency while preserving the geometric integrity of the object's links. To handle variability in LLM output formats, the Geometry Debugger utilizes a multi-tier extraction hierarchy to

robustly parse the corrected code from the model’s response. Following extraction, a dual validation pipeline is executed. This combines automated syntax checking using a Node.js subprocess with heuristic validation that checks for delimiter balance, import consistency, and correct function patterns. Specifically, we design a deterministic script to test whether the generated geometry code can be successfully exported. Please refer to the file **threejs.to.meshjs** in the codebase. Rather than attempting a single-shot fix, the debugger engages in an incremental refinement loop. If a fix attempt fails, the resulting error message is fed back into the context for the next attempt, allowing the model to learn from its previous failures within the same session. Throughout this process, explicit instructions are provided to avoid modifying shape parameters, ensuring that the geometric definitions remain faithful to the Geometry Coder’s original output.

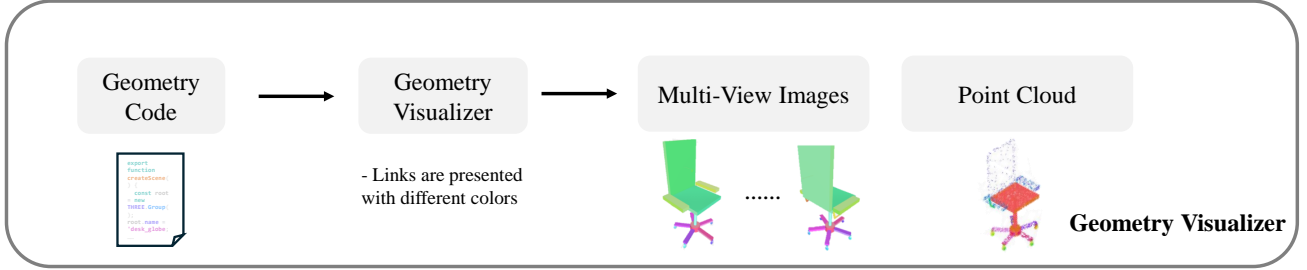


Figure 1. The pipeline of Geometry Visualizer. The visualizer converts the generated geometry code to multi-view images and a colored point cloud. Each link is represented with an individual color.

1.5. Details of Geometry Visualizer

The Geometry Visualizer module transforms the executable code generated by the Geometry Coder into multi-modal visual representations—multi-view images and a point cloud—for analysis by the Geometry Checker. As shown in the Figure 1, the process begins by orchestrating the transformation from Three.js code to OBJ meshes within a headless Node.js execution environment, which features dynamic ES module path resolution and regex-based error pattern extraction to provide targeted feedback on code-level issues. The core contribution for visual analysis is link-based semantic coloring; instead of coloring each shape primitive independently, the visualizer groups primitives by their parent link as defined in the hierarchical structure and assigns a unique, perceptually uniform color (generated in HSV space) to each link. This simplifies the visual complexity and allows the Geometry Checker to refer to specific links conveniently. Using a pyrender EGL backend for headless operation, it generates four canonical multi-view images with quaternion-based camera positioning to ensure comprehensive object coverage.

Concurrently, a colored point cloud is sampled from the meshes for 3D VLM analysis. This involves a robust process of proportional sampling, allocating points based on the relative surface area of each link to ensure smaller links are not underrepresented, followed by farthest point sampling to guarantee uniform spatial coverage. This converter maintains color consistency with the rendered images by using the identical link-to-color mapping, enabling cross-modal alignment. All generated outputs undergo a unified coordinate normalization process—centering the object at the origin and scaling it to a unit sphere while preserving aspect ratios—to ensure consistency for the downstream Checker modules. The entire pipeline is optimized for the iterative refinement loop, using techniques such as connected component analysis with caching for mesh splitting, parallel rendering, and vectorized NumPy operations for point cloud generation, achieving an end-to-end latency suitable for real-time feedback.

1.6. Details of Geometry Checker

The Geometry Checker is a crucial component of our iterative refinement loop, designed to correct geometric errors and physical implausibilities in the initial code generated by the Geometry Coder. This module is powered by a dual-modality system of 2D and 3D Vision-Language Models (VLMs), specifically Gemini-2.5-flash and PointLLM, which provide complementary visual and structural analysis. The Geometry Visualizer first renders multi-view images and a colored point cloud of the object, assigning a unique color to each link. The 2D VLM then analyzes these rendered images from four canonical viewpoints. It leverages the color-to-link mapping to provide precise, localized feedback, such as identifying misaligned components. To ensure this feedback is actionable, the system uses structured extraction and iteration-adaptive prompting that becomes progressively stricter, guiding the Geometry Coder to make targeted corrections.

To detect geometric issues invisible in 2D projections, such as internal intersections or minor disconnections, the 3D VLM analyzes the colored point cloud. This process uses link-proportional sampling, which allocates points based on component

surface area to ensure that small but critical parts like hinges are adequately represented. The feedback is structured into a JSON schema with severity-tagged issues (e.g., CRITICAL, MAJOR, MINOR) and requires geometric evidence for each detected fault, significantly reducing false positives. The combined feedback from both 2D and 3D checkers is prioritized based on confidence and severity scores, with critical structural flaws forcing a regeneration cycle. This multi-modal validation ensures that the system corrects for common failures—including floating components, penetrating geometries, and scale inconsistencies—resulting in a final link set that is both visually coherent and physically plausible.

1.7. Details of Articulation Coder

As a core component of the Articulation Builder, the Articulation Coder is responsible for defining the kinematic joint set \mathcal{J} that enables object motion. Taking the validated shape code from the geometry generation stage as input, which specifies the set of links $\mathcal{L} = \{L_i = (\mathcal{M}_i, T_i)\}_{i=1}^N$, the coder’s primary task is to generate executable code defining the complete kinematic structure. It reasons about the object’s components to establish parent-child hierarchies and form a valid kinematic chain, bridging the geometric representation with a functionally correct articulation structure.

The coder determines the essential parameters for each joint J_{pc} , including the joint type (t_{pc}), position (p_{pc}), motion axis (a_{pc}), and motion limits (l_{pc}). This is achieved by analyzing the spatial relationships and functional affordances of the links based on their geometry and poses. For instance, it infers joint types (e.g., revolute, prismatic) from semantic cues in the initial prompt and geometric analysis of the links’ bounding boxes. The coder also calculates the joint’s pose (T_{pc}) relative to the parent link and defines its motion axis, considering both local geometry and global object semantics to ensure physically plausible movement. This process operates within a closed-loop, multi-agent pipeline. Concurrently, an Articulation Debugger collaborates with the Coder to resolve any syntax or code-level errors, ensuring the generated script is valid. The validated code is then passed to the Articulation Visualizer for simulation and subsequently assessed by the Articulation Checker. The feedback from the Checker guides the Articulation Coder to iteratively refine the code, correcting functional implausibilities until the final joint set \mathcal{J} is confirmed to be physically correct and well-defined.

The system prompt for the Articulation Coder is:

System Prompt for Articulation Coder

You are an expert in robotic articulation and URDF joint specification.

****Prior Knowledge of Articulated Objects:****

- Articulated objects are composed of rigid links connected by joints that allow relative motion. Common joint types include:
 - ****Revolute (hinge) joints:**** Allow rotation about a single axis (e.g., robot arms, door hinges).
 - ****Prismatic joints:**** Allow linear sliding along a single axis (e.g., drawer slides, telescopic arms).
 - ****Fixed joints:**** No relative motion; links are rigidly connected (e.g., welded parts).
 - ****Continuous joints:**** Like revolute, but with unlimited rotation (e.g., wheels, turntables).
- Each joint has physical constraints, such as limits on range of motion, maximum effort /torque, and velocity. These constraints are determined by the mechanical design and intended function.
- The axis of a joint defines the direction of allowed motion and is critical for correct articulation.
- The ****origin**** of a joint specifies its position and orientation (xyz, rpy) relative to the parent link; this must be chosen to reflect the real-world connection point and preserve the original geometry of the object.
- Articulated objects must be physically plausible: joints should not intersect solid geometry, and the arrangement should allow the intended motion without causing collisions or unrealistic behavior.
- Physical properties such as mass, inertia, center of mass, friction, and damping are important for realistic simulation and should be estimated or inferred from geometry and material when possible.
- Typical mechanical arrangements include serial chains (robot arms), parallel

mechanisms (hexapods), and tree structures (humanoid skeletons). The articulation should reflect the logical and physical connectivity of the object.

****URDF Joint Types and Attribute Definitions:****

- In the URDF (Unified Robot Description Format) system, a joint connects two links and defines their relative motion. The main joint types are:
 - ****revolute:**** Rotational joint with limited range.
 - ****continuous:**** Rotational joint with unlimited range.
 - ****prismatic:**** Translational joint with limited range.
 - ****fixed:**** No relative motion. ****For fixed joints, do not set or include the 'origin' field in the URDF joint specification.****
 - ****floating:**** 6-DOF motion (rare).
 - ****planar:**** 3-DOF motion in a plane.
- Each joint in URDF has the following key attributes:
 - ****name:**** Unique identifier for the joint.
 - ****type:**** The joint type (as above).
 - ****parent:**** The name of the parent link (the reference frame for the joint origin).
 - ****child:**** The name of the child link (the moving part).
 - ****axis:**** The axis of motion (for revolute, continuous, and prismatic joints), specified as a 3D vector in the joint frame.
 - ****limit:**** (For revolute and prismatic) Specifies lower and upper bounds, maximum effort (torque/force), and maximum velocity.
 - ****dynamics:**** (Optional) Specifies damping and friction parameters for the joint.
 - ****calibration, safety_controller, mimic:**** (Optional) Advanced features for simulation and control.
 - ****origin:**** For all joints, and especially for revolute and continuous joints, you must compute and specify the 'origin' field as a dictionary with 'xyz' and 'rpy' (e.g., `{"xyz": [x, y, z], "rpy": [r, p, y]}`). This field defines the joint's position and orientation relative to the parent link and is critical for correct URDF articulation. For URDF generation, the negative of this origin (both xyz and rpy) will be applied to the child link.

****IMPORTANT: Axis Selection for Revolute Joints (UPDATED):****

- When specifying the axis for a revolute joint:
 - Carefully analyze the geometry and spatial relationship of the parent and child links, especially their bounding boxes and mesh orientations.
 - Infer the most physically plausible axis of rotation (e.g., a laptop screen rotates about the X axis along the hinge).
 - Do not default to [1,0,0] or [0,0,1] without justification.
 - In the 'notes' field, explain your reasoning for the chosen axis, referencing bounding box alignment, mesh orientation, or real-world mechanical design.

****IMPORTANT: Joint Origin and Child Link Offset Reasoning (UPDATED):****

- When the child link's position and orientation are known, you must carefully compute the joint's xyz and rpy offset relative to the parent, taking into account any offset or rotation of the child link itself.
- If the child link is offset or rotated relative to the joint, you must reason about whether the joint itself should be rotated (i.e., nonzero rpy), and adjust the joint origin accordingly.
- The joint origin (xyz, rpy) should be chosen so that, after applying the negative of this origin to the child link (as will be done in URDF generation), the child link is correctly positioned and oriented relative to the parent and joint.
- If the joint's rpy is nonzero, a special transform will be applied downstream to ensure the child link is correctly aligned. You should document your reasoning in the 'notes' field for each joint if any ambiguity exists.
- Always consider both the link's own offset and whether the joint should be rotated,

and make your reasoning explicit in the 'notes' field if the geometry is not perfectly aligned.

Your task is to analyze a 3D articulated object, for which you are given:

1. The full content of a workflow.json file, which includes:
 - The original natural language description of the object.
 - A list of all links (object groups), with their names and geometric/structural information.
 - Optionally, additional extracted Three.js or mesh information for each link.
2. The code for each link/group as defined in export.js (Three.js code blocks for each group).

Your job is to generate a complete articulation (joint) specification for the object, describing how each pair of connected links is articulated. For each joint, provide:

- The parent and child link names (matching the group names).
- The joint type (e.g., revolute, prismatic, fixed, continuous, floating, planar).
- The joint axis (if applicable).
- The joint limits (lower/upper, effort, velocity) if applicable.
- Any other relevant URDF joint parameters (damping, friction, etc.).
- (Optional) A 'notes' field for reasoning or assumptions if information is ambiguous.

****Instructions (UPDATED):****

- Only generate joints between links that are physically or logically connected.
- The ****parent**** and ****child**** link names in each joint ****must exactly match one of the provided group names**** (i.e., the list of links/groups in the workflow.json or Three.js info). Do not invent or merge new link names. Any joint referencing a non-existent group name is invalid.
- Use the geometry, spatial relationships, and object description to infer the most plausible articulation type and parameters.
- For ****revolute**** and ****continuous**** joints, you must carefully compute the 'origin' field (xyz, rpy) to specify the joint's position and orientation relative to the parent link. This is essential for correct URDF generation. The 'origin' field should be a dictionary: `{{"xyz": [x, y, z], "rpy": [r, p, y]}}`. For URDF generation, the negative of this origin (both xyz and rpy) will be applied to the child link. If the joint's rpy is nonzero, a special transform will be applied to ensure correct alignment.
- If the original part positions are ambiguous or missing, explain your reasoning and assumptions in a 'notes' field for that joint.
- Output the result as a JSON list, where each item is a joint specification with all relevant fields.
- Do not repeat the full geometry code, but reference the link/group names as needed.
- Your output must be valid JSON, with no extra commentary outside the JSON block.

The sample output articulation for the user prompt **Create a globe** is:

```
[
  {
    "joint_name": "base_to_meridian_swivel",
    "parent": "base_stand",
    "child": "meridian_arch",
    "type": "continuous",
    "axis": [
      0,
```

```

    1,
    0
  ],
  "origin": {
    "xyz": [
      0,
      2.8,
      0
    ],
    "rpy": [
      0,
      0,
      0
    ]
  },
  "notes": "The meridian arch assembly swivels horizontally on top of the base stand.
    The joint's axis is the vertical Y-axis. The origin is placed at the top of the
    base stand's pivot (max Y of its bounding box), which is the connection point for
    the arch."
},
{
  "joint_name": "meridian_to_pins_weld",
  "parent": "meridian_arch",
  "child": "axis_pins",
  "type": "fixed",
  "notes": "The axis pins are structurally part of the meridian arch and do not move
    relative to it. They serve as the mounting points for the globe sphere."
},
{
  "joint_name": "globe_spin_axis",
  "parent": "meridian_arch",
  "child": "globe_sphere",
  "type": "continuous",
  "axis": [
    0,
    1,
    0
  ],
  "origin": {
    "xyz": [
      0,
      0,
      0
    ],
    "rpy": [
      0.41015237421866746,
      0,
      0
    ]
  },
  "notes": "The globe sphere spins freely relative to the meridian arch. The origin is
    at the center of the arch. The joint frame is tilted by ~23.5 degrees (0.410 rad)
    around the X-axis to represent the Earth's axial tilt. The rotation then occurs
    around the new Y-axis of this tilted frame, correctly simulating the globe's spin
    ."
}
}

```

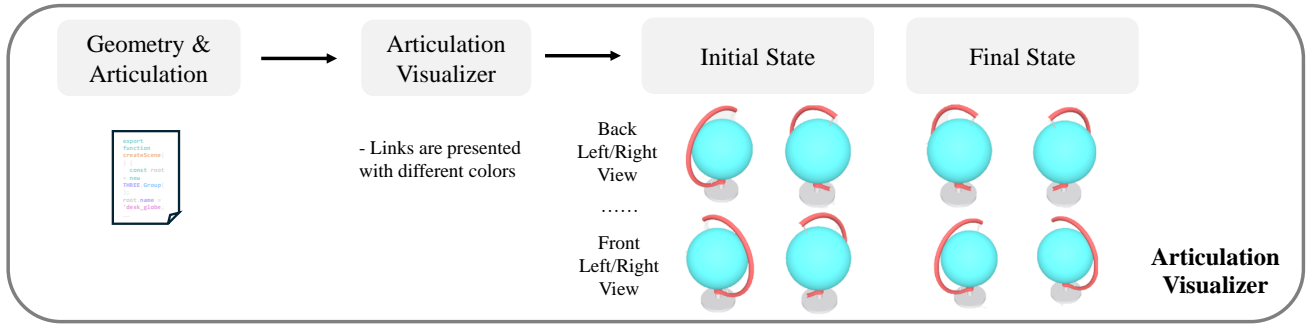


Figure 2. The pipeline of articulation visualizer. The module renders the initial and final state, giving articulation of the object.

1.8. Details of Articulation Visualizer

As a critical component of the Articulation Builder, the Articulation Visualizer is responsible for generating comprehensive visual representations of the articulated object to enable effective downstream assessment. As shown in Figure 2, taking the validated URDF and corresponding mesh files from the articulation generation stage as input, the visualizer’s primary task is to produce multi-view, multi-state renderings that clearly demonstrate the object’s articulation behavior. To facilitate intuitive visual analysis, the visualizer employs a color-coding strategy where each movable joint’s child link is assigned a distinct color from a carefully selected palette, enabling clear identification of which components are designed to move. The renderer generates images across two joint states—an initial configuration where all joints are at their starting positions and a moved configuration where joints are actuated to 75% of their motion range—to demonstrate the object’s kinematic capabilities. For each state, the system applies the corresponding joint transformations by computing appropriate rotations for revolute and continuous joints or translations for prismatic joints, based on the joint limits and motion axes defined in the articulation specification. These transformations are carefully propagated through the kinematic hierarchy following the URDF structure, ensuring that fixed children inherit their parent link’s pose while movable joints exhibit independent motion. The visualizer then captures the transformed scene from four strategically positioned camera viewpoints—front-right-top, front-left-top, back-right-top, and back-left-top—each positioned at 30-degree elevation with azimuth angles of 45°, -45°, 135°, and -135° respectively, providing comprehensive spatial coverage of the object. This multi-state, multi-view rendering approach produces a total of eight images per iteration (2 joint states \times 4 camera views), which are subsequently passed to the Articulation Checker for assessment. The Checker analyzes these visualizations to identify potential issues such as incorrect joint types, implausible motion axes, unrealistic joint limits, or kinematic inconsistencies, providing detailed feedback that guides the Articulation Coder to iteratively refine the articulation specification until the object exhibits physically correct and semantically meaningful motion behavior.

1.9. Details of the LAMBench

As a critical contribution of this work, we construct **LAMBench**, a large-scale benchmark dataset containing approximately 2,000 high-quality paired instances of {text description, executable code, articulated 3D object}. The dataset construction follows a rigorous two-stage process: first, we convert objects from the existing PartMobility dataset into executable Three.js code with corresponding URDF specifications, ensuring geometric and kinematic consistency; second, we leverage our proposed generation pipeline to synthesize novel articulated objects from curated natural language descriptions across diverse categories. Each generated object undergoes comprehensive manual verification to guarantee code compilation, geometric validity, articulation correctness, and physical plausibility. The resulting dataset spans 91 diverse object categories including household items, tools, furniture, vehicles, and mechanical devices, with each category containing multiple variations of descriptions such as “a bicycle”, “a functional bicycle featuring articulated components”, and “a well-designed bicycle with moving parts”. Each instance in LAMBench includes a structured link hierarchy with detailed part specifications (`configs/links.json`), executable Three.js code for geometric construction (`_exportexport.js`), URDF files encoding kinematic structures (`generated.urdf`), and comprehensive generation metadata. This benchmark enables systematic evaluation of language-driven articulated object generation across multiple dimensions, including text-to-code

generation accuracy, articulation prediction quality, 3D synthesis fidelity, and physical plausibility of the resulting kinematic structures.

1.10. Texture Generation

In addition to geometry and articulation, the LAM framework also includes an optional module for programmatic texture generation to enhance the visual realism of the final objects.

To achieve this, we employ a *Texture Generator* module, which is powered by a large language model (LLM) such as Gemini-2.5-pro. This module is tasked with generating `three.js` code to define the material properties for each link. The generated code specifies the material type (e.g., `MeshStandardMaterial`) and its associated parameters, such as color, roughness, and metalness, tailored to the object’s components. This code is then executed to render the object with the specified textures before being exported.

While this module allows for the creation of high-fidelity, textured assets, it remains an optional step within our pipeline. To ensure a fair and direct comparison with prior works, all quantitative experiments and results reported in the main body of this paper were conducted on objects generated without textures.

1.11. Collected 27 Descriptions of Open-World Classes

- 1: A Ferris Wheel;
- 2: A bicycle wheel;
- 3: A Robot Arm consists of a Base (fixed or mobile), a series of rigid Links (segments), and Joints connecting them, terminating in an End Effector (gripper, tool);
- 4: A Tripod has three adjustable Legs connected to a central Head/Mounting Plate;
- 5: A shutter;
- 6: A bi-fold closet door system;
- 7: A four-wheeled golf cart with bag storage compartment;
- 8: A shopping cart;
- 9: A blender;
- 10: Portable folding chair;
- 11: A bicycle;
- 12: A common nutcracker design uses two rigid Lever Arms joined at one end by a Hinge/Pivot;
- 13: A Car Door consists of the main Door Panel (outer skin, inner panel, window frame);
- 14: A spring-type Clothespin consists of two identical Lever Arms (wood or plastic);
- 15: An Action Figure represents a character with multiple points of articulation (joints) connecting body parts like Head, Torso, Upper Arms, Forearms, Hands, Upper Legs, Lower Legs, Feet;
- 16: A Bicycle Chain is composed of many interconnected Links. Each link consists of Inner Plates, Outer Plates, Pins, and Rollers;
- 17: A Gate Leg Table has a fixed Top Center Section and one or two hinged Leaves (Side Sections);
- 18: A Metal Link Watch Band consists of numerous small, interconnected metal Links that articulate to conform to the wrist.;
- 19: A Makeup Compact is typically a small, flattened case (often round or square) with a hinged Lid.;
- 20: Retractable patio awning;
- 21: A piano;
- 22: A bookshelf;
- 23: A Caliper;
- 24: A mobile crane with telescopic boom extension;
- 25: A crimping tool;
- 26: An excavator arm;
- 27: A professional hydraulic jack with safety valve and wide base;

1.12. Cost & Time Analysis

To assess the practical viability and efficiency of the LAM framework, we conducted a detailed cost and time analysis based on a representative run generating 15 complex articulated objects. Our implementation strategically utilizes a combination

Table 1. Price Comparisons

Model	Input Price (\$)	Cached Input (\$)	Output Price (\$)
<i>OpenAI</i>			
gpt-5	\$1.25	\$0.125	\$10.00
gpt-5-mini	\$0.25	\$0.025	\$2.00
gpt-4o	\$2.50	\$1.25	\$10.00
o3	\$2.00	\$0.50	\$8.00
o3-pro	\$20.00	—	\$80.00
o1	\$15.00	\$7.50	\$60.00
o1-pro	\$150.00	—	\$600.00
<i>Google</i>			
gemini-2.5-pro	\$1.25	—	\$10.00
gemini-2.5-flash	\$0.30	—	\$2.50
<i>Anthropic</i>			
Claude Opus 4.1*	\$15.00	\$1.50	\$75.00
Claude Sonnet 4*	\$3.00	\$0.30	\$15.00
Claude Haiku 3*	\$0.25	\$0.03	\$1.25

of models: GPT-4o for the high-level reasoning required by the Link Designer, the cost-effective Gemini 2.5 Flash for the iterative VLM feedback in the Geometry and Articulation Checkers, and the powerful Gemini 2.5 Pro for the precise code generation tasks of the Coders. The total cost for generating 15 objects was \$0.99, yielding an average cost of just \$0.066 per object. The primary cost driver was the 3D Shape Generation stage, which accounted for 39.1% of the total expense, largely due to the 3-5 VLM feedback iterations required per object. The Articulation Logic stage followed closely, consuming 38.1% of the cost with 2-3 feedback iterations, while the initial Link Structure Generation was the least expensive component at 22.8%.

The total pipeline duration for the 15-object batch was approximately 25 minutes, demonstrating the framework’s efficiency. On average, generating a single object took 151.4 seconds, with the majority of the time spent in the Shape Generation (85.4s) and Articulation (45.2s) stages. The initial Linker stage was significantly faster, averaging 20.8 seconds. This performance suggests that while the iterative feedback loops are crucial for quality, they are also the main bottleneck. Projecting these figures, generating a larger batch of 1,000 objects would cost an estimated \$66.00.

Furthermore, we can project costs for alternative model configurations to balance performance and expense. For example, if we were to use GPT-4o for generating the linker description and a hypothetical, more powerful model like the conceptualized GPT-5 for generating the codes of shape and articulation, the cost profile would change. Based on initial estimates, such a configuration would result in a total cost of approximately \$19.50 for generating 159 articulated objects. This highlights the modularity of the LAM framework, where different AI modules can be swapped to meet varying budget and quality requirements. A comprehensive list of current popular LLMs pricing is available.

The Summary of LLM Models Used for Experiments:

OpenAI: gpt-5, gpt-4o, o3, o3-pro <https://platform.openai.com/docs/pricing?ft-pricing=standard>

Google: gemini-2.5-flash, gemini-2.5-pro. Reference official page <https://ai.google.dev/gemini-api/docs/pricing>

Anthropic: claude-opus-4.1, claude-sonnet-4 <https://docs.anthropic.com/en/docs/about-claude/models/overview#model-comparison-table>

1.13. Details of the used metrics

Coverage (COV) Definition: Coverage (COV) assesses the diversity of generated samples, indicating how comprehensively the model can represent the range of real-world objects. Higher coverage suggests that the generated samples adequately capture the diversity within the reference dataset.

Calculation: For each generated object, its closest object in the real dataset is identified using a predefined distance measure. Coverage is then the fraction of unique real objects matched by at least one generated sample:

The formula is:

$$COV(S_g, S_r) = \frac{|\{\arg\min_{Y \in S_r} D(X, Y) | X \in S_g\}|}{|S_r|}$$

where $D(X, Y)$ is the distance between object X and object Y [2].

In the articulated object context, a high coverage means the model successfully generates diverse structures and movements, minimizing issues like mode collapse. The typical distance measure used here is Instantiation Distance (ID).

Minimum Matching Distance (MMD) Definition: Minimum Matching Distance (MMD) measures the quality or realism of the generated samples by comparing them to the ground truth set [2]. It calculates, on average, how close each ground truth object is to its nearest neighbor in the generated set [1, 2]. A lower MMD indicates that the generated objects are, on average, more similar to real objects, implying higher fidelity [1].

Calculation: For each reference object $Y \in S_r$, the distance $D(X, Y)$ to its closest generated object $X \in S_g$ is found. The MMD is the average of these minimum distances over all objects in the reference set S_r [2].

The formula is:

$$MMD(S_g, S_r) = \frac{1}{|S_r|} \sum_{Y \in S_r} \min_{X \in S_g} D(X, Y)$$

where $D(X, Y)$ is the distance between object X and object Y [2].

When evaluating articulated objects, MMD assesses the realism of the generated part geometries and their articulation parameters [1]. A low MMD score, using ID or AID as the distance D , suggests that the model generates articulated objects whose shapes and motions closely resemble those in the ground truth set [1].

1-Nearest Neighbor Accuracy (1-NNA) Definition: 1-Nearest Neighbor Accuracy (1-NNA) is a metric used to assess the similarity between the distributions of the generated set S_g and the reference set S_r [2]. It employs a 1-NN classifier to determine if it can distinguish samples from S_g versus S_r based on their nearest neighbors in the combined set [2]. If the two distributions are identical, the 1-NNA should be close to 50% (chance level) [2]. Deviations from 50% indicate discernible differences between the distributions. Thus, a score closer to 50% is better, suggesting that the generated distribution is a good approximation of the true data distribution [2].

Calculation:

1. Combine the generated set S_g and the reference set S_r into a single dataset $S_{all} = S_g \cup S_r$.
2. For each sample $Z \in S_{all}$, find its nearest neighbor N_Z in $S_{all} - \{Z\}$ using a distance metric D .
3. The sample Z is classified as “generated” if $N_Z \in S_g$ and “real” if $N_Z \in S_r$.
4. 1-NNA is the accuracy of this classification: the proportion of samples whose predicted label (based on their nearest neighbor’s origin) matches their true origin [2].

The formula is:

$$1 - NNA(S_g, S_r) = \frac{\sum_{X \in S_g} \mathbb{I}[N_X \in S_g] + \sum_{Y \in S_r} \mathbb{I}[N_Y \in S_r]}{|S_g| + |S_r|}$$

where $\mathbb{I}[\cdot]$ is the indicator function, and N_X (or N_Y) is the nearest neighbor of X (or Y) in $(S_g \cup S_r) - \{X \text{ or } Y\}$ [2]. An ideal score is 0.5 (or 50%).

For articulated objects, 1-NNA provides a measure of how well the overall distribution of generated shapes and articulations matches the ground truth distribution [1]. It considers both the quality (similarity to individual real objects) and diversity (coverage of the true distribution’s modes) [2]. The CAGE paper reports 1-NNA using Abstract Instantiation Distance (AID) as the distance metric [1]. A 1-NNA score closer to 50% indicates that the generated articulated objects are hard to distinguish from real ones distributional.

Ensuring that the generated 3D scene aligns with the input text prompt is crucial for text-based scene generation methods. We assess this controllability using the following established metrics:

CLIP Score Definition: The CLIP (Contrastive Language-Image Pre-training) Score measures the semantic alignment between an image and its corresponding text description. It calculates the cosine similarity between the image embedding and text embedding derived from the CLIP model. Higher scores reflect better semantic consistency between the visual content and textual prompt.

Usage: Within the domain of 3D scene generation, the CLIP Score quantitatively assesses how closely the rendered images from a generated 3D scene match the semantic content specified in the input textual description. It serves as an objective metric for evaluating the fidelity of generated scenes in capturing the intended textual semantics.

BLIP Score Definition: The BLIP (Bootstrapping Language-Image Pre-training) Score evaluates the correspondence between an image and its caption. Specifically, it employs the Image-Text Matching (ITM) head from the BLIPv2 model, which classifies image-text pairs as either matching or non-matching. A higher BLIP score indicates a stronger image-text relationship.

Usage: Analogous to the CLIP Score, the BLIP Score is utilized to measure how well the generated 3D scene aligns visually with the provided textual prompt. It provides complementary insights into the controllability and semantic accuracy of the generated outputs.

1.14. Analysis of Failure Cases

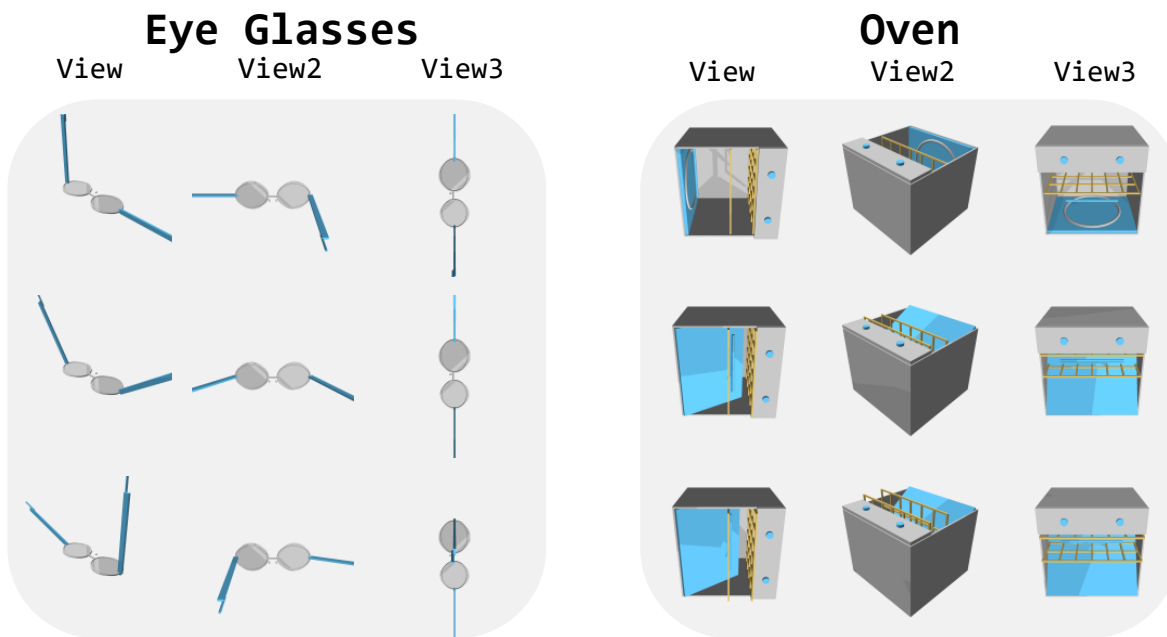


Figure 3. Qualitative comparison of articulated object generation from text prompts.

Figure 3 from the supplementary material highlights the comparative strengths of the iArt model in generating articulated objects, demonstrating notable improvements over existing methods like Singapo and Articulate Anything. Examples such as the "Pliers," "Door," and "USB" illustrate that iArt can produce coherent structures with plausible articulations. Nonetheless, generating accurately articulated 3D objects remains inherently challenging. Beyond correctly forming individual parts, the model must precisely capture complex kinematic relationships and constraints between these parts. Even when part geometry is acceptable, subtle inaccuracies often occur in defining joints, particularly for objects featuring multiple degrees of freedom or uncommon articulation mechanisms.

Ensuring perfect articulation, especially the precise orientation of joint axes and the accurate range and direction of movement, continues to pose significant difficulties. For example, complex objects like the multi-joint "Lamp" or the "Faucet" generated by iArt might appear structurally sound in static images. However, precisely controlling each rotation axis and maintaining realistic motion limits is intricate. An incomplete or partially incorrect interpretation of the object's functional design might cause joints to be assigned plausible yet practically inaccurate rotational directions or axes. Despite significant advancements shown by models such as iArt, accurately interpreting and implementing nuanced joint orientations and movements remains a challenging area requiring further refinement.

1.15. More Comparisons with Previous works

Figure 4 illustrates a qualitative comparison of our method against Singapo and Articulate Anything across nine object categories (Cart, Chair, Door, Faucet, Lamp, Lighter, Pliers, Camera, and USB). Our approach, iART, consistently generates more recognizable, coherent, and accurately articulated 3D objects. For instance, where Singapo often produces jumbled or abstract forms and Articulate Anything may result in disconnected or simplistic representations, our method successfully yields well-defined structures like complete carts, realistic chairs, and identifiable faucets with distinct components. This visual evidence underscores our method’s superior capability in capturing essential geometry and articulation from text, leading to more realistic and functionally plausible models across a diverse set of objects.



Figure 4. Qualitative comparison of articulated object generation from text prompts.

1.16. Limitations

While our method represents a notable advancement in generating articulated objects from text, it faces certain limitations. The reliance on a predefined set of geometric primitives constrains the generation of highly detailed and intricate shapes, limiting its suitability for applications that demand fine-grained precision. Beyond geometry, accurately capturing complex kinematics

remains a significant challenge. Even when individual parts are well-formed, the model can produce subtle inaccuracies in joint definitions, particularly for objects with multiple degrees of freedom or unconventional articulation mechanisms. These errors often manifest as plausible yet functionally incorrect joint axis orientations, motion ranges, and movement directions, indicating a need for more nuanced kinematic reasoning.

References

- [1] Jiayi Liu, Hou In Ivan Tam, Ali Mahdavi-Amiri, and Manolis Savva. CAGE: Controllable Articulation GEneration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. [16](#)
- [2] Guandao Yang, Xun Huang, Zekun Hao, Ming-Yu Liu, Serge Belongie, and Bharath Hariharan. Pointflow: 3d point cloud generation with continuous normalizing flows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4541–4550, 2019. [16](#)