BlenderGym: Benchmarking Foundational Model Systems for Graphics Editing

Supplementary Material

Appendix Overview

In Appendix S1, we extend the verifier scaling experiments to broader tasks. In Appendix S2, we analyze generator failure examples. In Appendix S3, we show a verifier's decision process for a task instance and offer a cross-model verification comparison on the same set of instances. In Appendix S4, we provide a calibrated interpretation of evaluation metrics and analyze their limitations. In Appendix S5, we show the reasoning behind the camera-view selection. Finally, in Appendix S6, we provide all the prompts used by the generator and verifier.

S1. Verifier Scaling

To consolidate our findings on strategic compute allocation between verification and generation, we (1) plot the N-CLIP score and Chamfer distance for blend shape (Fig. 7) and (2) extend our experiments to 15 lighting task instances (Fig. 8).

Our results demonstrate that with increased compute, VLM systems with higher verification ratio **consistently** outperform those with lower verification ratio. However, the size of this performance gap varies across tasks. As shown in Fig. 7 and Fig. 8, the performance gap between higher and lower verification ratios is smaller for lighting than for blend shape tasks. Our interpretation is that this gap is **positively related** to the difficulty of verification for the task –Lighting involves assessing more prominent factors like light intensity and color and, therefore, is easier for verification. In contrast, the blend shape manipulation task requires detecting more subtle and continuous changes, posing a significantly greater challenge.

We directly use the summation of the generation and verification queries as total queries since they incur a similar cost.

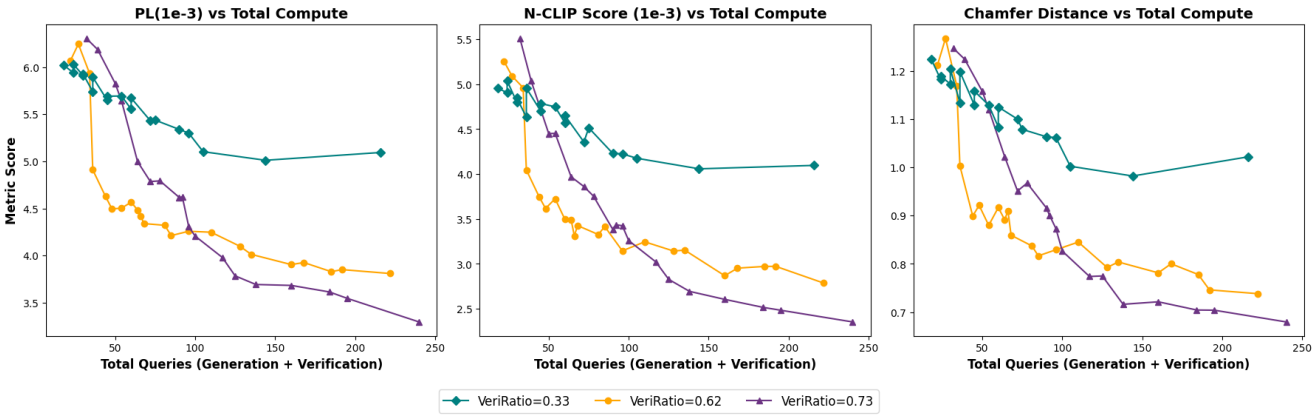

Figure 7. Impact of compute allocation in all three metrics on **blend shape manipulation** task.

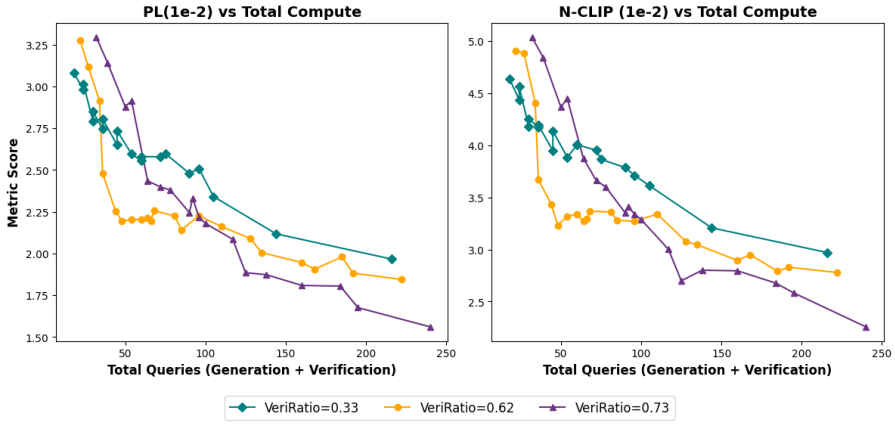

Figure 8. Impact of compute allocation in both PL and N-CLIP metrics on **lighting adjustment** task.

S2. Generator Failure Cases

Despite their capabilities, VLM generators exhibit the following common failures, as shown in Fig. 9, Fig. 10, and Fig. 11:

Failure to capture subtle visual differences. This issue arises mainly because the VLM generator often hallucinates non-existent visual differences between images rather than identifying actual discrepancies. This is a well-known limitation of VLMs and remains challenging to address. To mitigate this, we employ chain-of-thought (CoT) prompting, instructing the VLM to begin by analyzing visual differences and ignoring the code script. Details on our CoT implementation can be found in Appendix S6. However, the generator still occasionally disregards the CoT prompt, prematurely suggesting code changes instead of reasoning step-by-step, disrupting the intended stable reasoning process.

Failure to produce executable Blender Python scripts. Tasks like procedural material and geometry editing present significant challenges in generating executable code that reflects intended changes. These failures often stem from syntax errors, incompatibility with the Blender-Python API, or the inability to effectively incorporate the visual differences identified by the VLM, ultimately resulting in incorrect modifications.

S3. Verifier Failure Cases

We provide a complete verification process of a 3x4 tree generated by GPT4o, shown in Fig. 12, to contextualize the verification process. We also offer cross-model verifier comparisons on identical task instances in Fig. 13 and Fig. 14. Despite the prompt guidance in Fig. 21, only Claude 3.5 Sonnet consistently produces a complete reasoning process for its decisions, potentially enhancing its verification capability and contributing to its status as the most human-aligned VLM verifier.

S4. Calibration of Evaluation Metrics

We calibrate photometric loss (PL), negative-CLIP (N-CLIP), and Chamfer distance (CD) using the examples in Fig. 15 and Fig 3 of the main paper. PL and N-CLIP values are on the scale of 10^{-3} for blend shape and placement tasks and 10^{-2} for geometry, material, and lighting tasks. CD stays at its original scale. We notice that small metric differences can correspond to significant visual changes in the scene.

While these metrics generally align with human perception, they have two key limitations:

Failure in capturing physical plausibility For instance, Qwen2VL-7B leaves the soccer ball unnaturally stuck on the basket, violating physical laws and common sense. In contrast, MiniCPM-V2.6 places the ball outside the basket. MiniCPM-V2.6's edit, despite being suboptimal, adheres to physical laws and should be considered superior to Qwen's edit, a distinction not captured by the metric scores.

Task-dependent disproportionate scale. Lighting and procedural material tasks, due to their large-scale color changes, have higher values for N-CLIP and PL compared to blend shape and placement tasks. Procedural geometry editing also yields larger metric values since the object-of-interest often dominates the scene, making small changes more impactful. Conversely, placement and blend shape tasks typically involve object or feature adjustments of a smaller scale, leaving a significant proportion of the scene unchanged, leading to comparatively smaller metric values. Despite allowing cross-model comparison on a specific task, the disproportionate scales of metrics hinder direct cross-task comparison of a specific VLM system.

S5. Camera Viewpoint Selection

We define VLM-input and evaluation-only as two sets of views, with the former propagated to the VLM system and the latter reserved exclusively for evaluation. Both sets contribute to the evaluation metrics. A comprehensive view (defined below) is first selected and assigned to the VLM input set. Additional views capturing key object details are chosen, with one added to the VLM-input set and the remaining designated as evaluation-only. Importantly, all objects-of-interest are guaranteed to appear in at least one VLM-input view, ensuring the system has access to all critical visual information. Examples of this process are illustrated in Fig. 16 and Fig. 17.

We define a comprehensive view as a camera angle that provides a high-level perspective, typically from an elevated angle, encompassing most objects in the scene. It must clearly convey spatial relationships and object locations, particularly for objects-of-interest. While challenging to formalize in words, comprehensiveness is visually exemplified in Fig. 16 and Fig. 17, where the images in the first column are all comprehensive views.

Figure 9. Examples of generator failure for blend shape manipulation. We present the most visually obvious difference observed by the VLM, the code change proposed, and a failure analysis.

Figure 10. Examples of generator failure for lighting and procedural geometry. We present the most visually obvious difference observed by the VLM, the code change proposed, and a failure analysis.

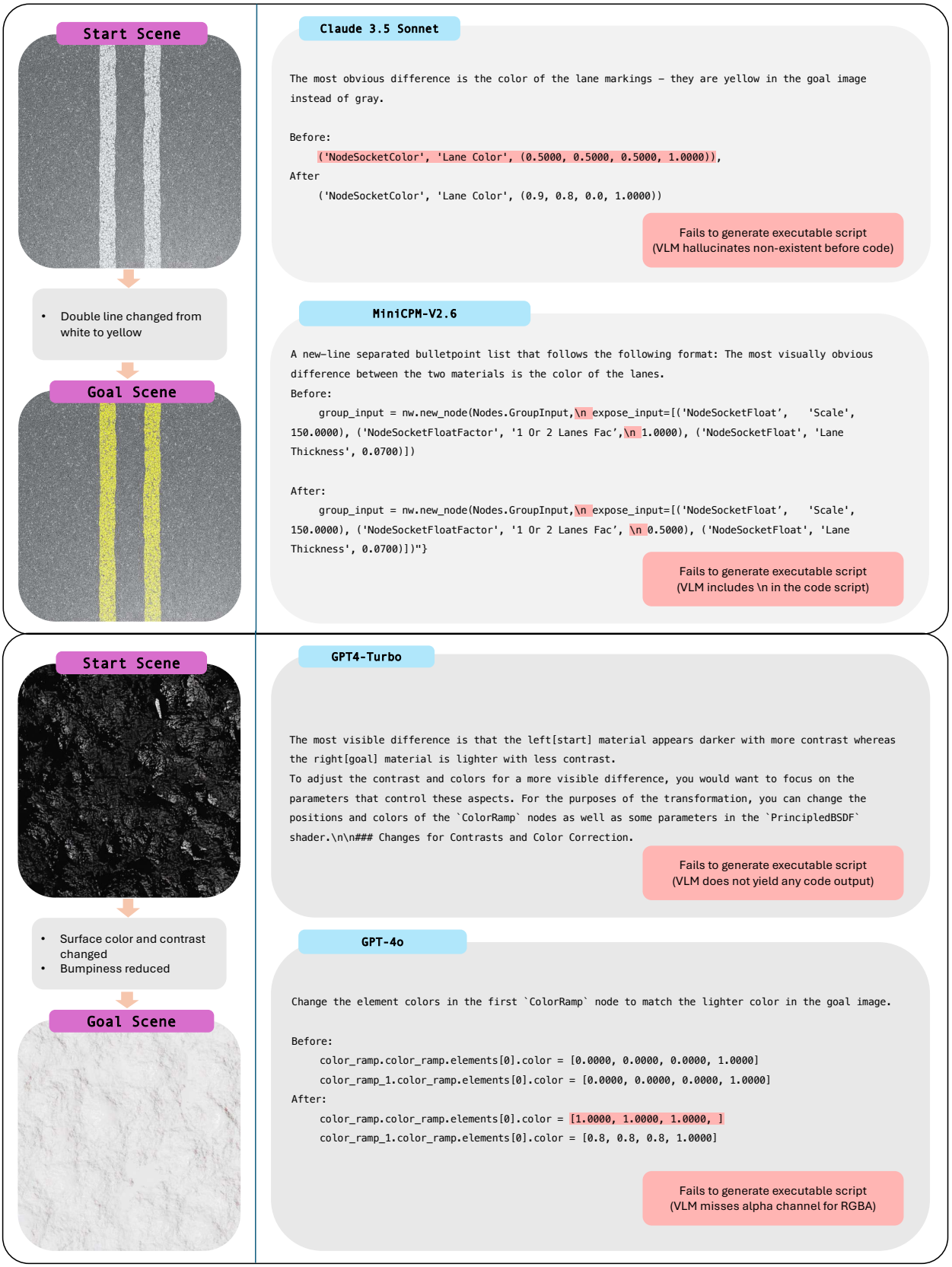

Figure 11. Examples of generator failure for procedural material editing. We present the most visually obvious difference observed by the VLM, the code change proposed, and a failure analysis.

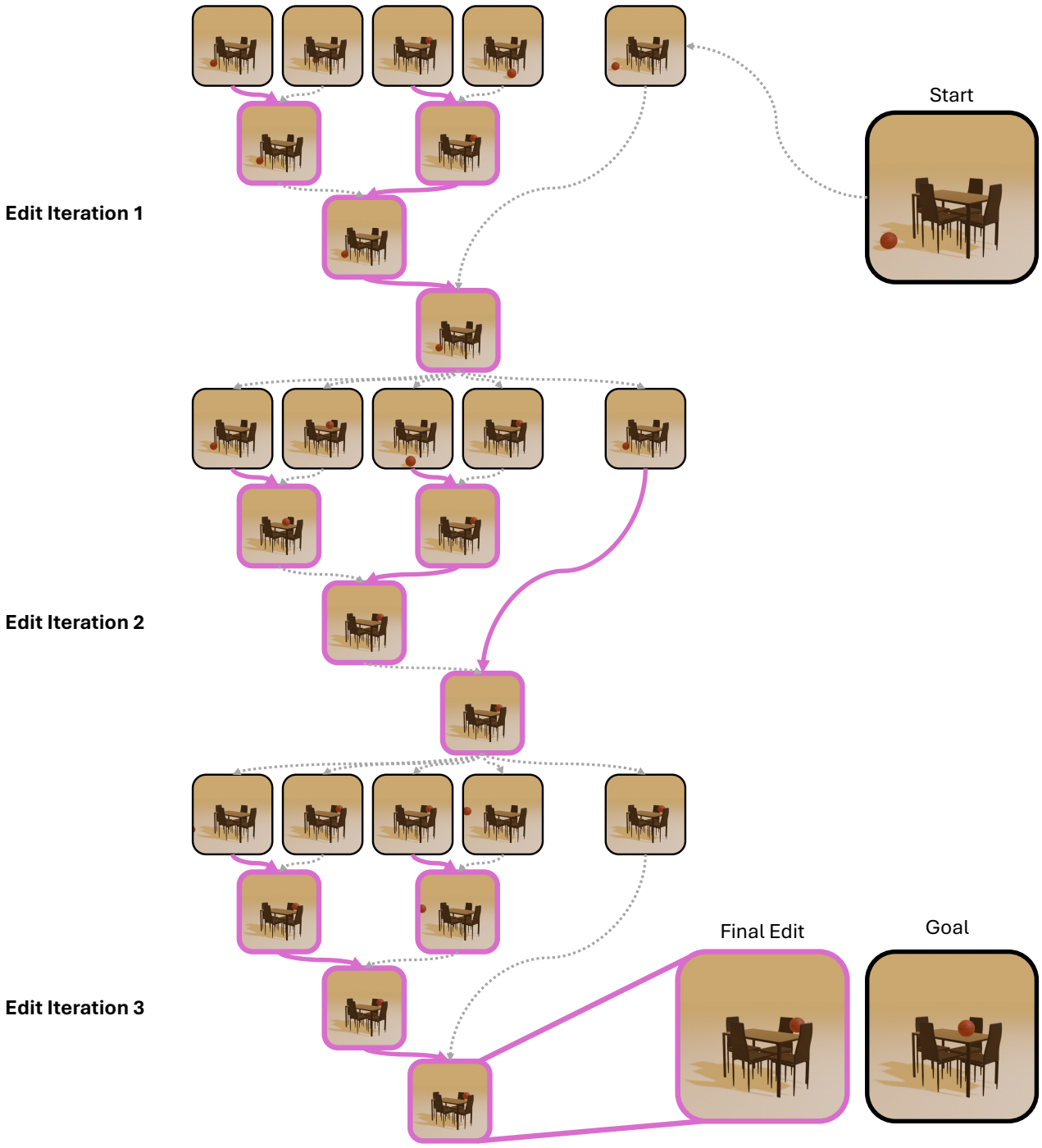

Figure 12. A complete verification process of a 3x4 tree generated by GPT4o on one task instance. We observe that a more human-aligned candidate is generated in edit iteration 2 but is not selected by the verifier.

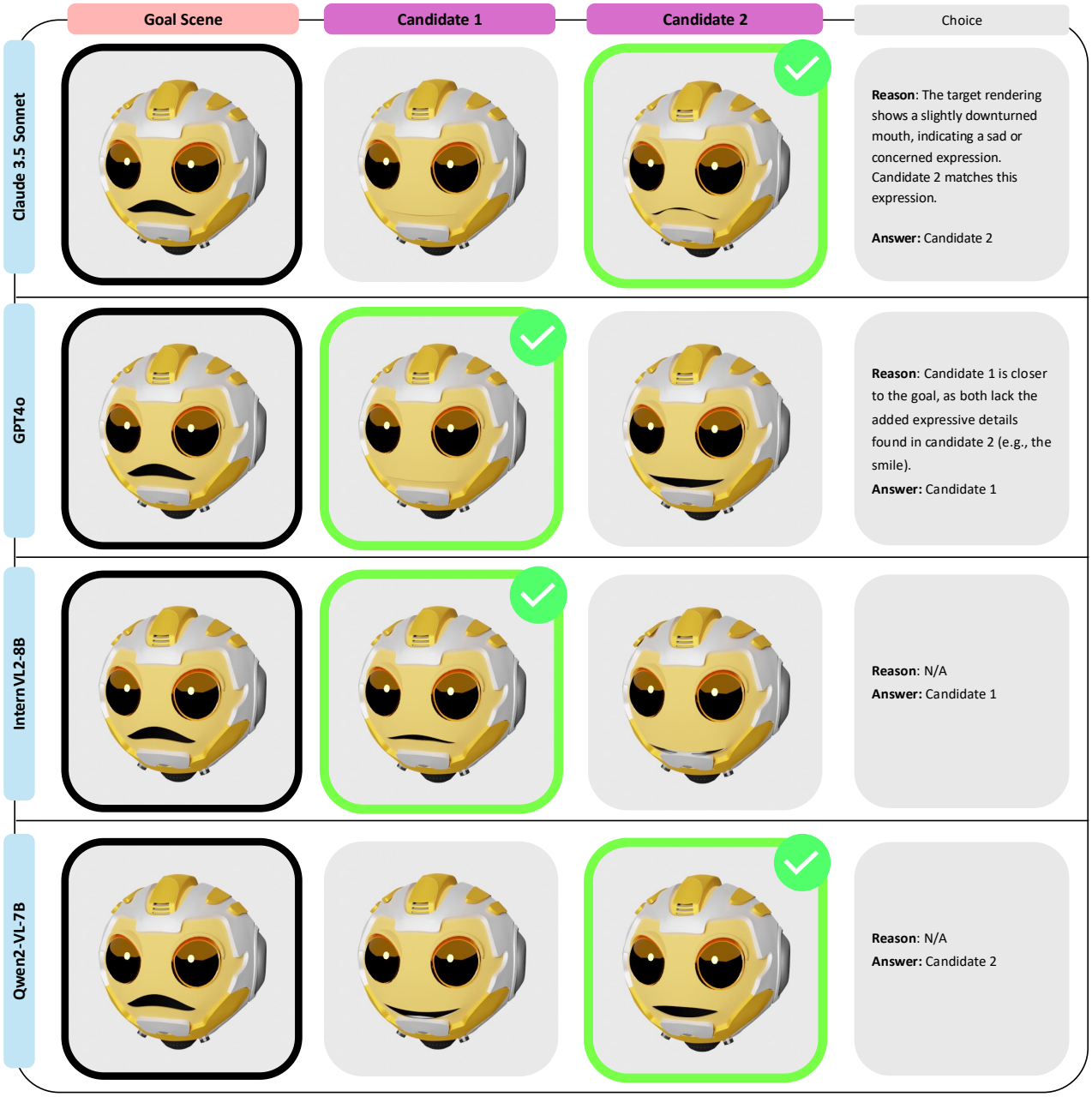

Figure 13. Examples of verifier decisions for a blend shape instance. N/A indicates that no reasoning is provided by the verifier. The candidates differ across models since they are rendered from edits generated by the model itself.

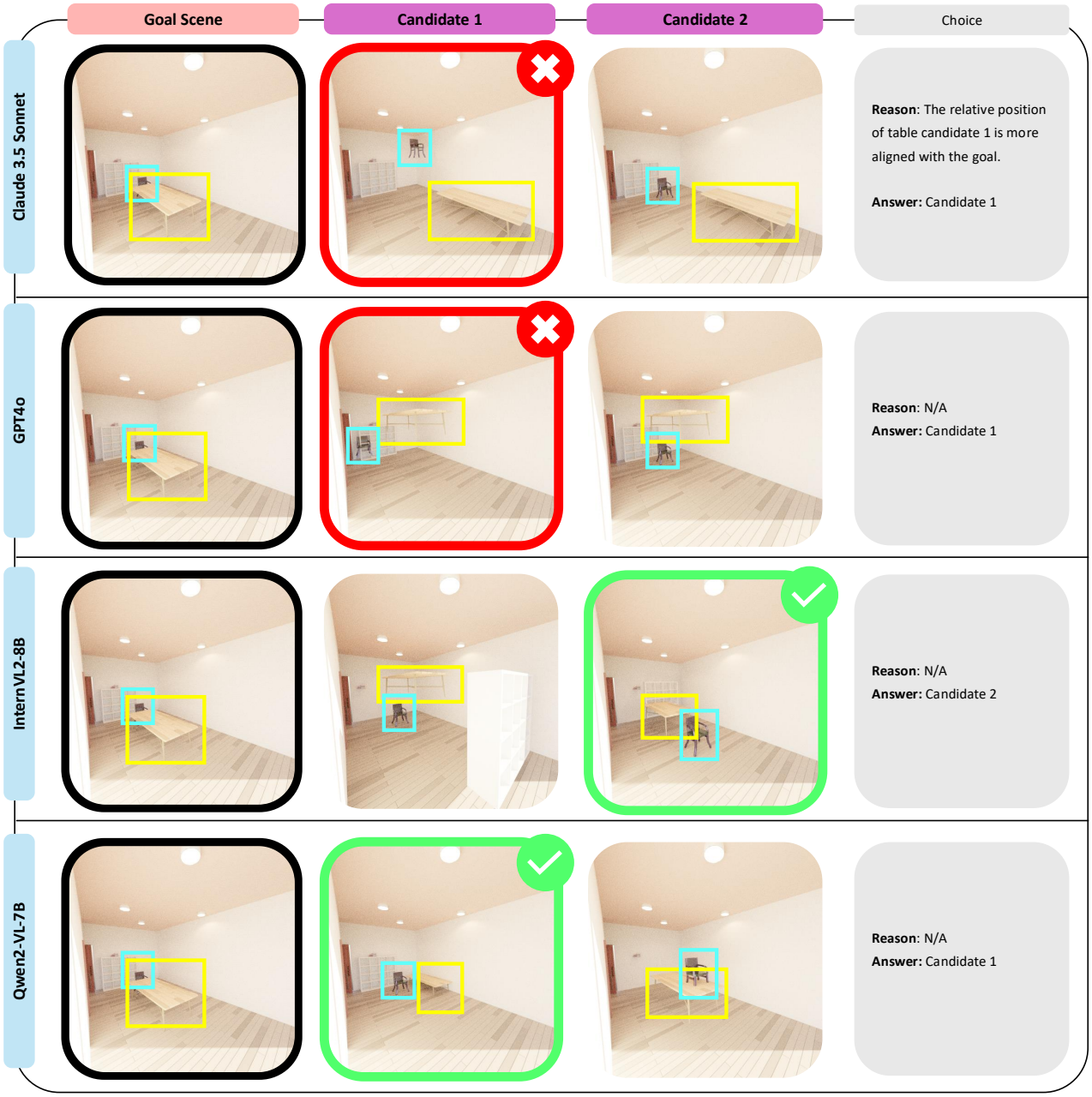

Figure 14. Examples of verifier decisions for an object placement instance. N/A indicates that no reasoning is provided by the verifier. The candidates differ across models since they are rendered from edits generated by the model itself.

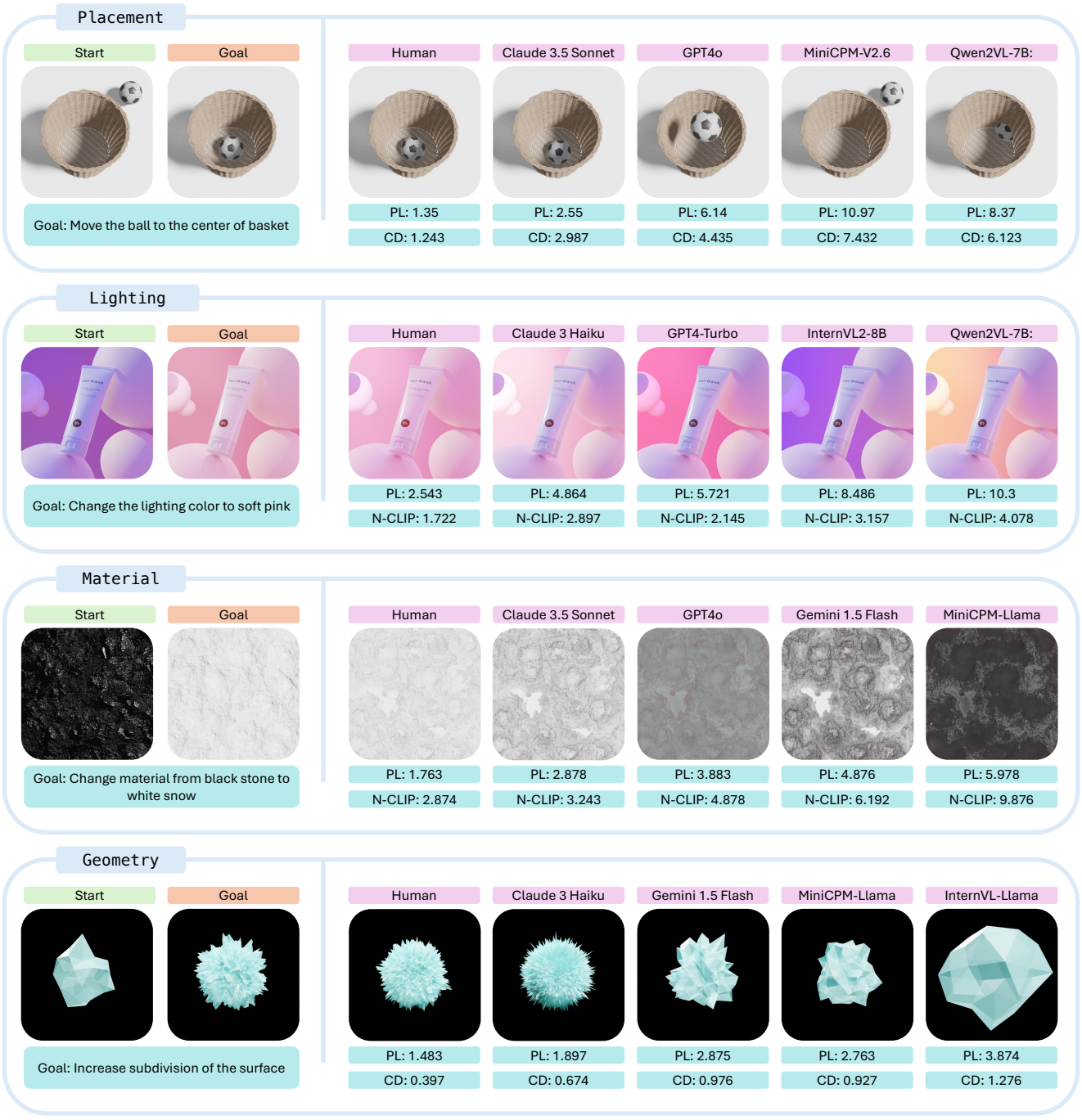

Figure 15. Calibration of metric values with render images of VLM system output edits. We present start scene, goal scene, human user edit, and VLM system edits side by side with their corresponding metric values.

Figure 16. Examples of VLM-input views and evaluation-only views. Images on the first column are all rendered from comprehensive views.

Figure 17. Examples of VLM-input views and evaluation-only views. Images on the first column are all rendered from comprehensive views.

S6. Prompts for VLM System

In our generator-verifier VLM system implementation, three VLM agents are involved: generator, code editor, and verifier. Here we include the prompt template we use for the three agents.

S6.1. Brainstormer

Brainstormer compares the start and goal render images, interprets the Python script of the start scene, and generates instructions for the required modifications on the script. It operates **alternatively** in two distinct modes: *tune* and *leap*, following from BlenderAlchemy. The *tune* mode adjusts parameter values within the existing code, while the *leap* mode proposes structural changes to the code, such as introducing new nodes for procedural editing. We set the return format to be a start-separated list of at most five instruction pieces. The prompt for both modes is given in Fig. 18 and Fig. 19.

S6.2. Code Editor

The code editor iterates through the brainstormer's output list of instruction pieces and integrates each of them into the code script of start scene. It generates a list of Python code differences, each including "CodeBefore," the original code segment from the input script, and "CodeAfter," the corresponding proposed modification to be applied. We use some helper function subsequently substitute CodeBefore with CodeAfter.

S6.3. Verifier

The verifier concatenates the render images of two proposal edits horizontally, compares them with the goal render, and selects one that is more similar to the goal edit. It returns a 'left" or "right" choice over the concatenated image, indicating the choice among the two candidates.

The following Blender code was used to produce a procedural 3D model:

```**[Python script of the START scene]** ```

The final code creates a procedural 3D model and produces the rendering on the left below(The image is concatenated by camera renders from different angles):

The desired procedural 3D model is shown in the image on the right(The image is concatenated by camera renders from different angles). Please describe the difference between the two 3D models, and edit the code above to reflect this desired change.

[A concatenated image of START(on the left) and GOAL(on the right)]

Describe, in a bullet-point list (using * as the bullet points), the biggest visual difference, which lines you would change (quote them in python code blocks) and how you would change them. Every item of the list should reference only ONE or A FEW lines of code and how it should be changed. Make AT MOST 5 such changes, no more than 5. Return in the format below:

raw: A new-line separated bullet point list that follows the following format:

Example:
* first item
* second item
...etc

Figure 18. Prompt for brainstormer in *leap* mode. This prompt is for procedural geometry editing task, but the ones for other tasks follow a similar structure with a few words changed.
```

The following Blender code was used to produce a procedural 3D model:

```**[Python script of the START scene]** ```

This creates a procedural 3D model and produces the rendering on the left below (the image is concatenated by camera renders from different angles):

The desired 3D model is shown in the image on the right (the image is concatenated by camera renders from different angles).

[A concatenated image of START(on the left) and GOAL(on the right)]

Answer the following questions:
1) What is the SINGLE most visually obvious difference between the two models in the two renderings in the image above (both images are concatenated by camera renders from different angles)?
2) Look at the code. Which fields/variables which are set to numerical values are most likely responsible for the obvious visual difference in your answer to question 1?
3) Replace the assignments of such fields/variables accordingly!

Describe, in a bullet-point list (using * as the bullet points), the biggest visual difference, which lines you would change (quote them in python code blocks) and how you would change them. Every item of the list should reference only ONE or A FEW lines of code and how it should be changed. Make AT MOST 5 such changes, no more than 5. Return in the format below:

gpt_raw: A new-line separated bulletpoint list that follows the following format:

Example:
* first item
* second item
...etc

Figure 19. Prompt for brainstormer in *tune* mode. This prompt is for procedural geometry editing task, but the ones for other tasks follow a similar structure with a few words changed.
```

Consider the following code of a procedural 3D model in Blender:

```**[Python script of the START scene]** ```

You'd like to do the following:

[Instruction piece from brainstormer]

Convert this into a concrete code difference indicated by "Before:" and "After:" labels, followed by code blocks that indicate which line should be changed and to what. Do not copy-paste the whole original code.

Example:

Before:
```
python
a = 1
```
After:
```
python
a = 2
```

Figure 20. Prompt for code editor. It receives instruction from brainstormer and incorporates it to the code script of start scene.

Here is the goal model rendering (the image is concatenated by camera renders from different angles):

[goal_model_image]

Below, I show two different models (the images are concatenated by camera renders from different angles). Which one is visually more similar to the goal model rendering?

[A concatenated image of Candidate 1 (on the left) and Candidate 2 (on the right)]

Return your answer in the following format:

raw: A block of text that contains a single word in a text block, indicated by ```. The word should be either "right" or "left". Example:

You've asks me to choose the image (left or right) that best aligns with the goal render.
Though the sample on the left is more realistic, the sample on the right is better aligned with the goal render.
```
right
```

Figure 21. Prompt for verifier.
```