TurtleAI: Benchmarking Multimodal Models for Visual Programming in Turtle Graphics

Chao Wen
MPI-SWS
chaowen@mpi-sws.org

Jacqueline Staub
University of Trier
staub@uni-trier.de

Adish Singla
MPI-SWS
adishs@mpi-sws.org

Abstract

Vision-language models (VLMs) have been explored for visual programming, where they generate code to solve visual tasks. However, most prior work focuses on visual programming for productivity; it remains unclear how well current VLMs perform on education-oriented visual programming and what factors limit their performance. To bridge this gap, we introduce TurtleAI, a benchmark containing 823 tasks curated based on real-world visual programming tasks in the Turtle Graphics domain. Solving these tasks requires models to perceive geometric patterns, reason about spatial relationships, and synthesize Python code that faithfully reproduces geometric patterns. We evaluate 20+ VLMs, including GPT-5, GPT-4o, and Qwen2-VL-72B, and find that they struggle significantly, with most achieving success rates below 30%. To address these limitations, we propose a data generation technique that requires only a small set of seed samples. Fine-tuning Qwen2-VL-72B on the resulting synthetic data yields an improvement of about 20% on real-world tasks. Our failure analysis reveals that GPT-4o struggles with spatial reasoning and precise visual replication, whereas fine-tuning primarily improves the alignment between visual reasoning and code implementation.

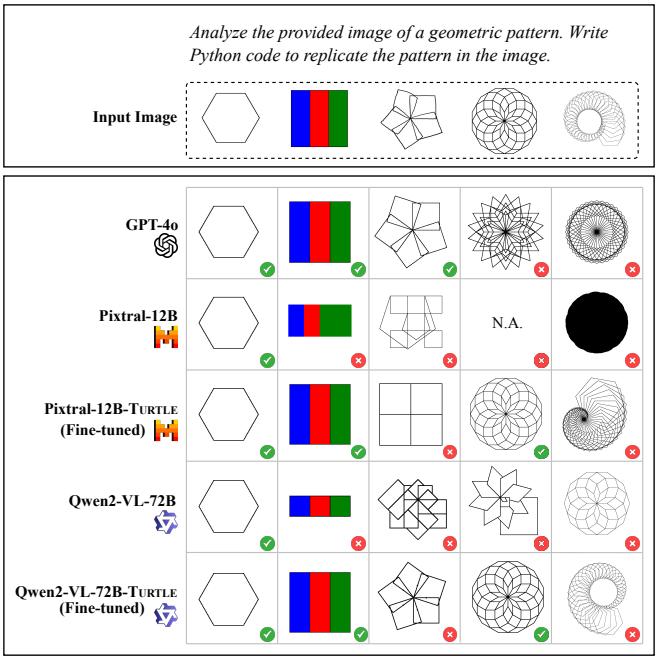
1 Introduction

Vision-language models (VLMs) have shown strong performance on visual understanding tasks (Ramesh et al., 2021; Radford et al., 2021; Yue et al., 2024; Lu et al., 2024). Beyond visual understanding, recent work has explored *visual programming*, in which VLMs are given visual inputs and prompted to generate executable code that solves visually grounded tasks, such as solving visual reasoning problems (Gupta and Kembhavi, 2023; Li et al., 2024c; Wen et al., 2025b) and generating visual artifacts (e.g., UIs, SVGs) (Si et al., 2025; Yang et al., 2025; Wu et al., 2025; Rodriguez et al., 2025; Zou et al., 2024; Wen et al., 2025a).

However, most existing work on visual programming is motivated by improving *productivity* through code generation in applied domains, such as software engineering, data visualization, and computer graphics (Gui et al., 2025; Si et al., 2025; Wu et al., 2025; Yang et al., 2025; Rodriguez et al., 2025; Zou et al., 2024). Beyond productivity, visual programming is also a central paradigm in education-oriented settings, particularly in K-12 programming education for developing students' computational thinking (Grover and Pea, 2013), as exemplified by platforms such as Scratch (Maloney et al., 2010), Code.org (Code.org, 2013), and Blockly (Blockly, 2022). Despite its widespread use in education and the potential of generative models to support it, education-oriented visual programming remains largely underexplored. It remains unclear how well current VLMs perform on educational visual programming tasks and which factors most limit their performance.

To bridge this gap, we introduce TurtleAI, a benchmark for assessing VLMs' visual programming capabilities on education-oriented Turtle Graphics tasks (Python, 2024). The benchmark comprises 823 visual programming tasks curated based on the visual programming platform XLogoOnline (XLogoOnline, 2024), which is used by tens of thousands of students each year (Staub, 2021). In TurtleAI, each task requires a VLM to generate Python code that reproduces a target image. Figure 1 shows representative target images along with corresponding model outputs. Solving these tasks requires recognizing visual patterns, reasoning about spatial relationships (e.g., layout, scale, position, and angles), and translating this analysis into executable Python code that accurately reproduces the target image.

We evaluate 20+ state-of-the-art VLMs on TurtleAI and find that they struggle: even leading models such as GPT-5 (OpenAI, 2025a), GPT-4o (OpenAI, 2024a), and Qwen2-VL-72B (Wang



(a) VLMs' outputs for replicating different input images.

```python
def draw(t):
    """
    Draws a flower-like pattern composed of 12 dodecagons.
    Each dodecagon is rotated by 30 degrees relative to
    the previous one, creating a star-like structure.
    """

    def draw_polygon(side_length=50, num_sides=12):
        """
        Draws a single regular polygon with the
        given side length and number of sides.
        """
        angle = 360 / num_sides
        for _ in range(num_sides):
            t.forward(side_length)
            t.left(angle)

    def draw_flower(num_petals, side_length,
                    num_sides, angle):
        """
        Draws a flower with the given number of
        petals, side length, number of sides, and angle.
        """
        for _ in range(num_petals):
            draw_polygon(side_length, num_sides)
            t.left(angle)

    # Draws 12 dodecagons, each rotated by 30 degrees
    draw_flower(num_petals=12, side_length=50,
                num_sides=12, angle=30)
```

(b) Solution code for replicating the image.

Figure 1: Outputs of VLMs on visual-to-code generation tasks and an example solution code. (a) shows the input images and the visual outputs produced by executing each VLM's generated Python code, with success (✓) or failure (✗) shown for each output. (b) shows an example solution code for replicating the image.

et al., 2024) achieve success rates below 30% on real-world tasks. To improve performance, we propose a data generation technique that synthesizes training data from a small set of seed samples; fine-tuning on this synthetic data yields roughly a 20% improvement on real-world tasks. Our failure analysis reveals that GPT-4o most often fails due to limitations in spatial reasoning and faithful visual replication, whereas Qwen2-VL-72B more frequently fails to align its code implementation with its visual reasoning; fine-tuning primarily reduces these alignment errors rather than improving visual understanding or spatial reasoning.

Our contributions are threefold. First, we introduce TurtleAI, a multimodal benchmark for evaluating VLMs in the Turtle Graphics domain, with datasets curated based on real-world visual programming tasks. Second, we propose TurtleAI-Datagen, a data generation technique that creates large-scale synthetic training data, and show that fine-tuning VLMs on this data improves performance on TurtleAI. Third, we conduct extensive experiments and analyses on TurtleAI, providing insights into VLMs' capabilities and limitations in educational visual programming settings. We publicly release our benchmark for further research.[1]

[1] https://github.com/machine-teaching-group/acl2026-turtleai

2 Related Work

Visual programming benchmarks. Visual programming involves generating executable code to solve visually grounded tasks. Prior work explores visual programming across various productivity-oriented domains such as UI generation (Si et al., 2025; Gui et al., 2025), data visualization (Wu et al., 2025; Yang et al., 2025), and graphics synthesis (Zou et al., 2024; Rodriguez et al., 2025). Recent work has also explored visual programming in educational contexts. However, these benchmarks typically focus on discrete, grid-based navigation or multiple-choice reasoning, often utilizing domain-specific languages with restricted syntax (Wen et al., 2025b; Padurean and Singla, 2024; Singla, 2023). In contrast, our benchmark targets the synthesis of complex geometric patterns in continuous space using Python, which necessitates a more expressive and complex code space.

Program synthesis for inverse graphics. Our tasks can be viewed as a form of inverse graphics, where the goal is to generate code that reconstructs a given visual input. Prior work has studied inverse graphics in domains such as SVGs (Rodriguez et al., 2025; Zou et al., 2024), scientific figures in LaTeX (Belouadi et al., 2024), and charts (Wu et al., 2025; Yang et al., 2025). Existing work has also explored Turtle Graphics (Ellis et al., 2021; Li and

Ellis, 2024; Rismanchian et al., 2025). The most relevant benchmark is TurtleBench (Rismanchian et al., 2025). Our work differs in three key ways: (i) Datasets: Our datasets are curated based on real-world visual programming tasks actively used by students, whereas TurtleBench relies on manually created tasks; (ii) Scope: Our benchmark spans six task categories, three difficulty levels, and three datasets, whereas TurtleBench is built around a single dataset with two task types and simpler black-and-white geometric patterns; and (iii) Scale: We evaluate 20+ VLMs (including fine-tuning) on 823 tasks, while they evaluated 3 VLMs on 260 tasks.

3 Background and Synthesis Objective

In this section, we provide background on Turtle Graphics and XLogoOnline, and introduce the program synthesis objective.

Background on Turtle Graphics. Turtle Graphics is a programmable method for creating vector graphics and has been used in K-12 programming education to teach programming concepts and computational thinking (CodeHS, 2025; XLogoOnline, 2024; Turtle Academy, 2025). In Turtle Graphics, one can create vector graphics using a relative cursor (the "turtle") on a Cartesian plane (Python, 2024). Basic commands like "forward", "turn left", and "pen down" control the turtle's movement to draw lines and shapes. These commands can be combined with programming constructs such as loops, conditionals, and functions to generate visually appealing geometric patterns.

Background on XLogoOnline platform. XLogoOnline (XLogoOnline, 2024) is a Logo-based visual programming platform (Pea, 1987) used by tens of thousands of students each year (Staub, 2021). It offers four levels (Mini, Midi, Maxi, Mega); the Midi (grades 3-4) and Maxi (grades 5-6) levels provide Turtle Graphics tasks, where students learn programming concepts by replicating visual patterns through writing code. We curate our benchmark primarily based on these tasks (e.g., the hexagon pattern in Figure 1).

Task specification. We define a visual programming task in Turtle Graphics as $\texttt{T} := (\texttt{img}, \texttt{ins})$, a tuple consisting of a target image `img` and a text-based instruction `ins`. The target image specifies the desired visual output, while the instruction specifies the requirements for generating the code that will replicate the pattern shown in the target image.

Code specification. The code space for Turtle Graphics tasks is defined using the Python programming language. A *solution code* for a task `T` is Python code `C` that, after being executed, can accurately replicate the target image `img` and satisfy the requirements specified by the instruction `ins`. For consistent evaluation, the task's instruction requires the solution code to be synthesized as a function `draw(t)`, which takes a turtle object `t` as input. For instance, Figure 1b shows a solution code that generates the image  .

Program synthesis objective. The synthesis objective is to develop a synthesizer function, $\mathcal{M} : \texttt{T} \rightarrow \texttt{C}$, which generates a solution code `C` for a given task `T` in Turtle Graphics. To evaluate $\mathcal{M}$ on a task `T`, we first use $\mathcal{M}$ to synthesize a code $\hat{\texttt{C}}$ that contains a Python function `draw(t)`. To evaluate the correctness of this Python function, one straightforward way is to compare the images generated by $\hat{\texttt{C}}$ and `C` pixel-by-pixel. However, this pixel-wise comparison fails to account for differences in the size, position, or line width of patterns being drawn in images (Marbach et al., 2022). In the next section, as part of our benchmark TurtleAI, we will address this by introducing an evaluation framework.

4 The TurtleAI Benchmark

TurtleAI is a visual programming benchmark for Turtle Graphics. Figure 2 illustrates the components of TurtleAI, which consists of (i) TurtleAI-DS, a collection of evaluation datasets; (ii) TurtleAI-Eval, an evaluation framework for assessing the correctness of synthesized code; and (iii) TurtleAI-Datagen, a data generation technique for synthesizing high-quality data. We describe each component in the following sections.

4.1 Evaluation Datasets TurtleAI-DS

TurtleAI includes TurtleAI-DS, a collection of 823 evaluation tasks organized into three distinct datasets. Figure 3 shows example tasks and the distribution of these datasets.

TurtleAI-DSReal (Size 102). This dataset contains real-world tasks curated from the Midi and Maxi levels of XLogoOnline (XLogoOnline, 2024). The tasks were originally designed by experts to teach programming concepts to students in grades 3-6. We manually wrote Python reference solutions for each task and verified their correctness.

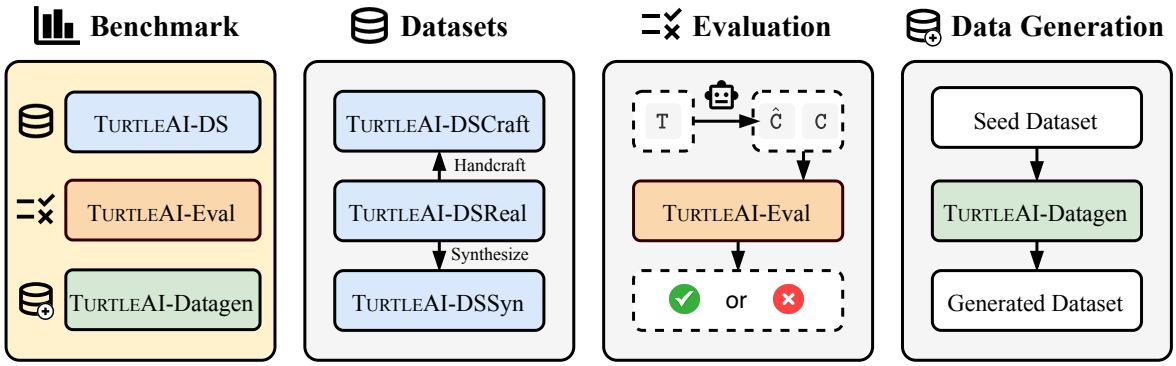
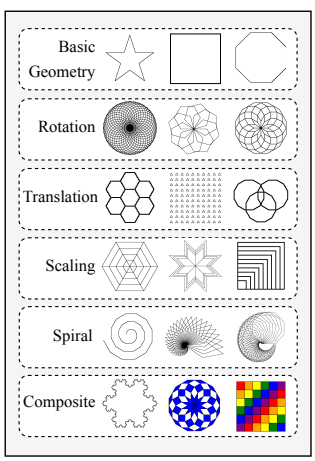
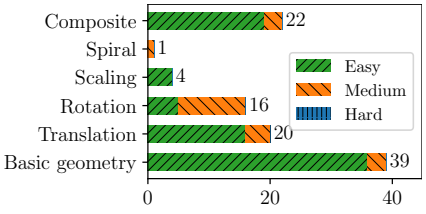


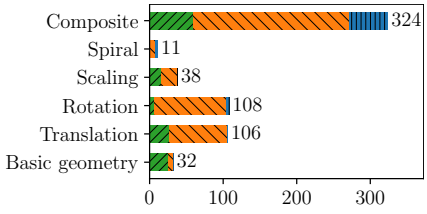
Figure 2: Overview of the TurtleAI benchmark. It comprises three key components: (i) a collection of datasets TurtleAI-DS, (ii) an evaluation framework TurtleAI-Eval for assessing the correctness of generated code, and (iii) a data generation technique TurtleAI-Datagen for generating synthetic datasets.



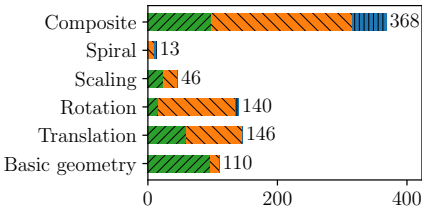
(a) Examples for each category



(b) DSReal/DSCraft



(c) DSSyn



(d) TurtleAI-DS

Dataset	Easy	Medium	Hard	Total
DSReal	80	22	0	102
DSCraft	80	22	0	102
DSSyn	135	425	59	619
DS	295	469	59	823

(e) Difficulty distributions

Figure 3: Dataset composition and statistics. Tasks are categorized into six task categories and three difficulty levels. (a) shows representative examples for each category. (b-d) show the distributions of these categories in DSReal, DSSyn, and TurtleAI-DS, respectively. Note that DSCraft has the same distribution as DSReal, as it is a hand-drawn rendering of the same set of tasks. (e) shows the difficulty distribution across different datasets. The detailed labeling process for task categories and difficulty levels is provided in Appendix C.

TurtleAI-DSCraft (Size 102). This dataset contains hand-crafted tasks created by manually hand-drawing each image from TurtleAI-DSReal, while reusing the same solution code. It is designed to evaluate model robustness to hand-drawn inputs that may include imperfections.

TurtleAI-DSSyn (Size 619). This dataset contains synthetic tasks generated using TurtleAI-DSReal as a seed dataset for our data generation technique (see Section 4.3), followed by manual selection of high-quality tasks. See Appendix C for more details about the dataset generation process.

4.2 Evaluation Framework TurtleAI-Eval

In our benchmark, we evaluate whether the synthesized code $\hat{C}$ produces a drawing that is *semantically* equivalent to the ground-truth code `C`. One natural way is to compare the images generated by $\hat{C}$ and `C` pixel-by-pixel. However, pixel-wise comparison is sensitive to non-semantic factors (e.g., translation, scale, and line width); for example, the same square drawn at $(0, 0)$ and $(1, 1)$ would be judged different despite being semantically identical. To address this, we introduce TurtleAI-Eval, an evaluation framework that canonicalizes drawings before comparison to achieve invariance to translation, scale, and line width. Specifically, TurtleAI-Eval executes `C` and $\hat{C}$ with a custom Turtle emulator that records drawing states (e.g., vertices, strokes, fills, colors). It then normalizes coordinates to a fixed range, centers the drawing at the origin, and standardizes line width to 1, pro-

ducing canonical renderings $\texttt{img}$ and $\hat{\texttt{img}}$.[2] Finally, we compare $\texttt{img}$ and $\hat{\texttt{img}}$ using two methods:

- *Symbolic comparison*: we compare $\texttt{img}$ and $\hat{\texttt{img}}$ pixel-wise. A prediction is marked as *success* if the fraction of matching pixels exceeds a predefined threshold; otherwise, it is *fail*.
- *Embedding-based comparison*: We embed $\texttt{img}$ and $\hat{\texttt{img}}$ using a pre-trained image encoder (e.g., ResNet18 (He et al., 2016)) and compute a normalized similarity score in $[0, 1]$. A prediction is *success* if the similarity exceeds a threshold; otherwise, it is *fail*.

Thresholds are selected based on manual evaluations, i.e., by choosing the values that maximize agreement with human judgments. This yields 99.1% accuracy for symbolic comparison and 98.1% for embedding-based comparison against human judgments (see Appendix E).

4.3 Data Generation Technique TurtleAI-Datagen

As mentioned in Section 1, existing VLMs struggle on TurtleAI. To make these models practically useful in educational settings, it is important to improve their performance. One natural way is to fine-tune VLMs for Turtle Graphics; however, fine-tuning is limited by the scarcity of training data in this domain. To address this, we propose TurtleAI-Datagen, a technique that leverages large models to synthesize high-quality training data for Turtle Graphics. The key idea is to evolve a large dataset from a small seed through iterative stages of code mutation and elite selection. We describe the two stages below (see Figure 4).

Stage 1: Code mutation. This stage aims to generate a larger set of codes from a given small seed dataset $\mathcal{D}_t = \{(\texttt{img}_i, \texttt{C}_i)\}$ (Xu et al., 2024; Wen et al., 2024). This can be done by pre-defining instructions for large language models (LLMs) to mutate the codes (Xu et al., 2024). For example, one can use the instruction "add a loop to the code" to guide the LLM to mutate an input code $\texttt{C}_{\text{in}}$:

$$\texttt{C}_{\text{out}} = \text{LLM}(\texttt{C}_{\text{in}}, \text{instruction} = \text{"add a loop to the code"}).$$

However, a fixed set of pre-defined instructions may limit the diversity of mutated codes by capturing only specified mutation patterns. To address this, we use an LLM to infer mutation patterns. Specifically, the LLM is given a pair of reference codes $(\texttt{C}_{\texttt{ref}_1}, \texttt{C}_{\texttt{ref}_2})$ and prompted to infer the high-level mutation pattern $m(\texttt{C}_{\texttt{ref}_1}, \texttt{C}_{\texttt{ref}_2})$. It then applies this pattern to another input code $\texttt{C}_{\text{in}}$, producing a new mutated output code:

$$\texttt{C}_{\text{out}} = \text{LLM}(\texttt{C}_{\text{in}}, \text{instruction} = m(\texttt{C}_{\texttt{ref}_1}, \texttt{C}_{\texttt{ref}_2})).$$

We apply this process to extend the seed dataset: for each $\texttt{C}_{\text{in}} \in \mathcal{D}_t$, we randomly sample p reference code pairs from $\mathcal{D}_t$ to generate mutated codes. These codes are then executed to obtain images, producing a larger dataset of image-code pairs.

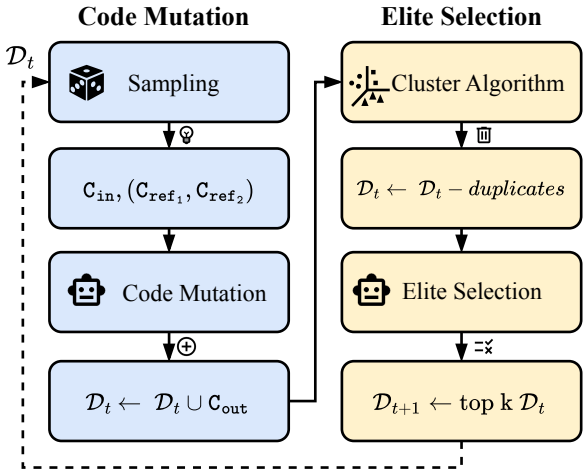


Figure 4: Overview of the data generation technique TurtleAI-Datagen. It consists of two stages: (i) code mutation, which creates diverse code variants; and (ii) elite selection, which deduplicates candidates and retains high-quality samples.

Stage 2: Elite selection. After obtaining a larger dataset of image-code pairs, we first use a clustering algorithm to remove duplicate image-code pairs. Then we use a VLM to score image quality based on predefined rubrics. Finally, we select the top $k\%$ image-code pairs, which serve as the seed dataset for the next iteration $t + 1$. After n iterations, this produces a large-scale dataset $\mathcal{D}_{t+n}$.

Generating training dataset TurtleAI-Train. We construct the training dataset TurtleAI-Train using TurtleAI-Datagen in three steps. First, we start from a manually curated seed set of only 10 image-code pairs. Second, we run TurtleAI-Datagen for five iterations to expand the seed set. Third, we enrich each resulting $(\texttt{img}, \texttt{C})$ pair in the expanded dataset with Chain-of-Thought (CoT) labels (Wei et al., 2022; Zelikman et al., 2022) by prompting a VLM to describe the image,

[2] For simplicity, we use $\texttt{img}$ to denote the canonical rendering when the context is clear.

explain the solution code step by step, and add documentation. We find that adding CoT labels improves fine-tuning and generalization performance (see Appendix D.3). Finally, we obtain the final training dataset containing $738,126$ samples. More details are provided in Appendix C.2.

5 Experimental Evaluation

In this section, we evaluate VLMs on TURTLEAI. We first describe the experimental setup in Section 5.1 and present the main results in Section 5.2. Then we provide fine-grained performance analyses (Section 5.3), failure analysis (Section 5.4), and code length analysis (Section 5.5).

5.1 Experimental Setup

Evaluation procedure and metrics. We use the datasets described in Section 4.1 for evaluation. Each evaluation dataset consists of $(\texttt{T}, \texttt{C})$ pairs. For each pair, we provide the task image `img` and a fixed prompt instructing the model to generate the code snippet $\hat{\texttt{C}}$ in the desired format. The model may also output explanations, but we only extract $\hat{\texttt{C}}$. Then we evaluate $\hat{\texttt{C}}$ against the ground-truth `C` using our evaluation framework (see Section 4.2). The *success rate* is the number of successful predictions divided by the total number of samples. We report both symbolic- and embedding-based success rates. For the main experiments, success rates are based on greedy decoding; additional Pass@K results from random sampling are provided in Appendix D.6.

Models evaluated. We compare various VLMs: (i) *Reasoning VLMs*, including GPT-family models: GPT-5 (OpenAI, 2025a), o3, and o4-mini (OpenAI, 2025b); (ii) *Non-reasoning base VLMs*, covering model families from GPT (OpenAI, 2024a), Qwen (Wang et al., 2024), Molmo (Deitke et al., 2024), Llava (Li et al., 2024a), Pixtral (Agrawal et al., 2024), and InternVL (Zhu et al., 2025); and (iii) *Fine-tuned VLMs*, trained on our TURTLEAI-Train dataset with 738k samples, denoted with the TURTLE suffix. Full model details and fine-tuning details are in Appendix F.2.

5.2 Main Results

Current VLMs struggle, even on real-world tasks designed for students in grades 3-6. As shown in Table 1, all evaluated models struggle on our benchmark. On the real-world dataset DSReal containing tasks designed for students in grades 3-6, even the top-performing model, o3, achieves only a 40.2% symbolic success rate, suggesting current VLMs are still far from ready for classroom deployment. Performance is even lower on the full dataset TURTLEAI-DS: among base models, o4-mini achieves the highest symbolic success rate at 15.9%, while the strongest open-source base model, Qwen2.5-VL, reaches only 6.80%. Overall, these results show that TURTLEAI is challenging for all existing VLMs.

Fine-tuning improves performance, but yields limited gains on hand-drawn tasks. Fine-tuning models (e.g., Pixtral-12B-TURTLE, Qwen2-VL-7B-TURTLE, and Qwen2-VL-72B-TURTLE) consistently improves performance on TURTLEAI-DS, increasing symbolic success rates by over 10% over their base counterparts. This suggests that data generated by TURTLEAI-Datagen is effective for improving model performance. However, performance improvements mainly come from DSReal and DSSyn, while performance on DSCraft remains low, indicating that fine-tuning primarily improves performance on clean and synthetic inputs but does not fully address robustness to out-of-distribution hand-drawn tasks. Additional out-of-distribution results are provided in Appendix D.3.

5.3 Analysis of Model Performance Across Different Dimensions

To enable a more fine-grained analysis of current VLMs, we examine their performance across three dimensions: task categories, difficulty levels, and dataset types. The results are shown in Figure 5.

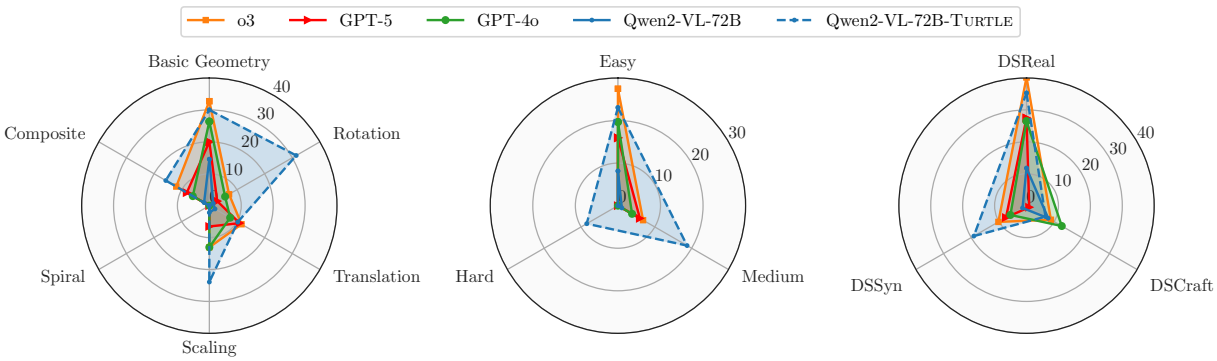
Across task categories (see Figure 5a), models perform best on Basic Geometry, while Spiral is consistently the most challenging category for all models, as it requires long-horizon, sequential control with tightly coupled steps. Besides, Composite tasks are not uniformly harder than single-transformation tasks (i.e., Rotation, Translation, and Scaling). Overall, these results suggest that models struggle more with tasks requiring high precision and long procedural sequences than with tasks that combine transformations.

Across difficulty levels (see Figure 5b), all models exhibit a clear monotonic trend, with success rates highest on easy tasks and decreasing on medium and hard tasks, suggesting that the difficulty levels in our datasets capture meaningful differences in task complexity.

Across datasets (see Figure 5c), most models

	Size	TurtleAI-DSReal		TurtleAI-DSCraft		TurtleAI-DSSyn		TurtleAI-DS	
		Sym. (%)	Emb. (%)	Sym. (%)	Emb. (%)	Sym. (%)	Emb. (%)	Sym. (%)	Emb. (%)
Reasoning:									
o3	-	**40.20**	**44.12**	8.82	9.80	10.18	9.21	13.73	13.61
o4-mini	-	36.27	38.24	**28.43**	**29.41**	**10.50**	**10.50**	15.92	16.28
GPT-5 (medium)	-	27.45	29.41	0.98	0.98	7.43	6.79	9.11	8.87
Non-reasoning (> 30B):									
GPT-4o	-	26.47	28.43	12.75	13.73	5.82	5.17	9.23	9.11
GPT-4V	-	15.69	17.65	8.82	10.78	3.23	2.75	5.47	5.59
Pixtral-Large	124B	10.78	11.76	13.73	15.69	4.68	3.88	6.56	6.32
Llava-OneVision	72B	4.90	3.92	6.86	5.88	0.97	0.97	2.19	1.94
InternVL3	78B	12.75	13.73	7.84	9.80	3.23	2.58	4.98	4.86
Molmo	72B	3.92	4.90	4.90	4.90	1.62	1.45	2.31	2.31
NVLM-1.0-D	72B	0.00	0.00	0.00	0.00	0.16	0.16	0.12	0.12
Qwen2.5-VL	72B	15.69	18.63	**17.65**	**17.65**	3.55	3.55	6.80	7.17
Qwen2-VL	72B	11.76	14.71	7.84	8.82	1.45	1.62	3.52	4.13
Qwen2-VL-Turtle	72B	**35.29**	**39.22**	6.86	6.86	**19.06**	**17.12**	**19.56**	**18.59**
Non-reasoning (≤ 30B):									
Qwen3-VL	30B	18.63	16.67	**13.73**	**14.71**	2.91	2.26	6.20	5.59
Pixtral	12B	9.80	9.80	2.94	2.94	0.97	0.97	2.31	2.31
Pixtral-Turtle	12B	27.45	29.41	9.80	9.80	**13.41**	**12.12**	**14.70**	**13.97**
Llava-OneVision	7B	3.92	3.92	2.94	2.94	0.32	0.48	1.09	1.22
GLM-4V	9B	0.00	0.00	0.00	0.98	0.00	0.00	0.00	0.12
Molmo	7B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Qwen2-VL	7B	0.98	0.98	0.00	0.00	0.00	0.00	0.12	0.12
Qwen2-VL-Turtle	7B	**28.43**	**30.39**	6.86	8.82	11.95	11.15	13.37	13.24

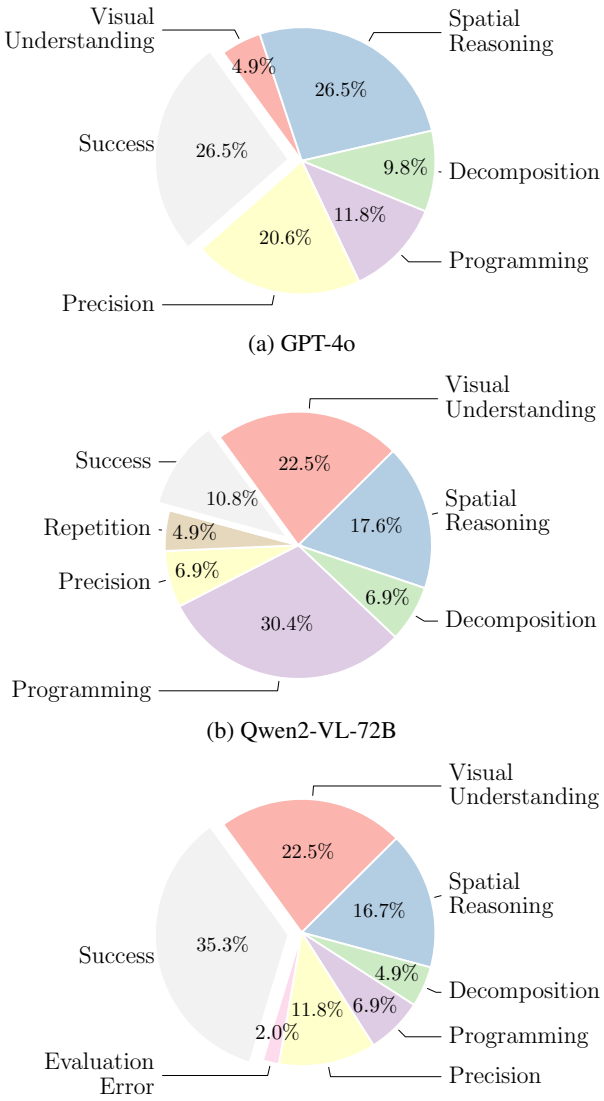
Table 1: Performance comparison of VLMs on different datasets. We evaluate VLMs using both symbolic comparison (Sym.) and embedding-based comparison (Emb.), with results shown as success rates (%). Fine-tuned models are denoted by the suffix Turtle. The best performance within each group is shown in **bold**, and the second-best is underlined. See Appendix D.6 for Pass@K results.



(a) Task Categories (b) Difficulty levels (c) Datasets

Figure 5: Symbolic success rates (%) of representative VLMs across task categories, difficulty levels, and datasets. (a) shows performance by task category, where Basic Geometry is the easiest and Spiral is consistently the most challenging. (b) shows performance by difficulty level, decreasing consistently from easy to medium, and to hard. (c) shows performance by dataset, where most models drop relative to DSReal on both DSCraft and DSSyn.

perform slightly worse on DSCraft than on DSReal. Since DSCraft is constructed by hand-drawing the images from the same tasks as DSReal, this gap indicates that hand-drawn input variations introduce additional challenges. Performance also drops on DSSyn across most models, consistent with its harder difficulty distribution relative to DSReal (see Figure 3). Furthermore, while fine-tuning improves performance on both DSReal and DSSyn, it yields limited gains on DSCraft. This suggests that fine-tuning on clean synthetic data still faces challenges on out-of-distribution tasks, offering limited improvement in robustness to hand-drawn inputs. We provide additional analysis of out-of-distribution generalization for fine-tuned models in Appendix D.3.



(a) GPT-4o

(b) Qwen2-VL-72B

(c) Qwen2-VL-72B-TURTLE

Figure 6: Distribution of failure types on DSReal dataset for three representative VLMs.

5.4 Failure Analysis

To examine the limitations of VLMs, we conduct a systematic failure analysis on three representative models: GPT-4o, Qwen2-VL-72B, and Qwen2-VL-72B-TURTLE. We examine these models using DSReal, since it consists of real-world tasks collected from visual programming platforms and therefore best reflects real-world challenges. We manually review generated images, code, and available explanations to identify root causes of errors, attributing each case to the failure type that contributes most.[3] The distribution of failure types is shown in Figure 6, with definitions and examples in Appendix C.3.

[3]For failure analysis, we apply CoT prompting to Qwen2-VL-72B to elicit image descriptions and reasoning, yielding a 10.78% success rate on DSReal, close to non-CoT's 11.76%.

Models struggle with spatial reasoning. We find that all models struggle with spatial reasoning, which is the ability to reason about the spatial relationships among different patterns in the image, such as relative positions, distances, angles, and sizes of patterns. This might be due to the scarcity of training data that captures spatial relationships when training VLMs.

Models struggle with precise visual details. We find that despite correct visual reasoning, models still face difficulties in achieving visual precision during replication. For instance, both GPT-4o and Qwen2-VL-72B can often replicate the intended image from a high-level perspective but fail to achieve low-level visual accuracy, such as ignoring tiny details like angles and relative positions.

Models often miss crucial details in images. Visual understanding errors remain common between Qwen2-VL-72B and Qwen2-VL-72B-TURTLE. During review, we found that they often overlook small but crucial details and describe images using approximate common patterns. For instance, if an image shows a square with a unique cut-off, the models might just describe and draw a regular square, ignoring the specific cut-off.

Fine-tuning improves code-reasoning alignment. By comparing Qwen2-VL-72B and Qwen2-VL-72B-TURTLE, we observe that fine-tuning increases the success rate from 10.8% to 35.3%, mainly by reducing programming errors (from 30.4% to 6.9%). Visual understanding and spatial reasoning errors remain largely unchanged, suggesting that fine-tuning primarily improves the alignment between code generation and visual reasoning, rather than visual understanding or spatial reasoning itself.

5.5 Code Length Analysis

Beyond code correctness, code quality also matters for education-oriented visual programming. In Turtle Graphics tasks, one aspect of code quality is compactness, as unnecessarily long solutions often indicate redundant operations or missed structural patterns such as loops. To capture this aspect, we use the *length ratio* as a proxy for code compactness. For each task, length ratio is computed as the number of lines in the model-generated code divided by the number of lines in the corresponding reference solution, after removing comments and blank lines. Although this metric does not directly

Model	Length Ratio (Success-Only)	Length Ratio (Failure-Only)	Length Ratio (All)
GPT-4o	1.25 ± 0.08	1.25 ± 0.05	1.25 ± 0.04
Qwen2-VL-72B	1.52 ± 0.16	4.01 ± 1.27	3.74 ± 1.14
Qwen2-VL-72B-TURTLE	1.02 ± 0.05	1.03 ± 0.06	1.03 ± 0.04

Table 2: Length ratios on DSReal dataset for three models. We report length ratio for three cases: *Success-Only* (computed over successful codes), *Failure-Only* (computed over failed codes), and *All* (computed over all codes). Length ratios close to 1.0 indicate generated code lengths similar to the reference solution code, while larger ratios suggest more verbose or potentially redundant codes. Length ratio values are reported as mean $\pm$ standard error.

capture all aspects of code quality, it provides a lightweight and automated measure of code compactness, a relevant aspect of code quality in Turtle Graphics tasks.

We analyze length ratio on DSReal for three representative models: GPT-4o, Qwen2-VL-72B, and Qwen2-VL-72B-TURTLE. As shown in Table 2, GPT-4o maintains a relatively stable ratio across successful and failed cases (ratio = 1.25). Qwen2-VL-72B exhibits a moderate length ratio on successful cases (ratio = 1.52), but its outputs become substantially longer in failure cases (ratio = 4.01), indicating a substantial increase in code verbosity upon failure. In contrast, the fine-tuned Qwen2-VL-72B-TURTLE maintains a length ratio close to 1.0 across all cases, suggesting that fine-tuning is associated not only with improved correctness, but also with more consistent and compact generation behavior, particularly in failure cases.

6 Concluding Discussions

In this paper, we introduced TURTLEAI, a benchmark of 823 tasks for evaluating vision-language models on education-oriented visual programming in Turtle Graphics. Our evaluation showed that current VLMs struggled substantially, even on real-world tasks designed for students in grades 3-6, highlighting a clear gap between current model capabilities and the reliability required for classroom deployment. To improve performance, we proposed TURTLEAI-Datagen, a data generation technique that synthesizes high-quality image-code pairs from a small seed set; fine-tuning Qwen2-VL-72B on the synthesized data improved performance on real-world tasks by over 20% compared to the base model. Our analysis showed that GPT-4o most often failed due to limitations in spatial reasoning and faithful visual replication, while fine-tuning primarily reduced programming errors by aligning code implementation with visual reasoning rather than improving visual understanding or spatial reasoning.

Limitations

We discuss some limitations of our work and propose ideas for addressing them in the future. First, our analysis of code quality focuses on code compactness as measured by length ratio, which does not capture all aspects of code quality. Future work could explore additional metrics for code quality, such as code readability and pedagogical alignment. Second, our evaluation framework includes a normalization step that makes drawing comparison invariant to size, translation, and line width, which may discard meaningful geometric variations that are essential in inverse graphics tasks. Future work could explore evaluation methods that preserve these geometric properties. Third, we did not provide a systematic ablation or comparison with other data synthesis techniques. Future work could conduct controlled ablations and method comparisons to better understand the contribution of each stage in TURTLEAI-Datagen and how different synthesis choices affect fine-tuning performance. Finally, although fine-tuning with our data generation technique improves overall performance, fine-tuned models still struggle on hand-drawn inputs (e.g., DSCraft). This may be because our synthesized training data focuses on clean renderings and lacks sufficient variation that matches the noise and distortions in human drawings. Future work could improve robustness by augmenting training with hand-drawn style perturbations, such as random noise injection and transformations that simulate common hand-drawn artifacts.

Acknowledgments

Funded/Co-funded by the European Union (ERC, TOPS, 101039090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

Pravesh Agrawal, Szymon Antoniak, Emma Bou Hanna, Baptiste Bout, Devendra Chaplot, Jessica Chudnovsky, Diogo Costa, Baudouin De Monicault, Saurabh Garg, Theophile Gervet, Soham Ghosh, Amélie Héliou, Paul Jacob, Albert Q. Jiang, Kartik Khandelwal, Timothée Lacroix, Guillaume Lample, Diego Las Casas, Thibaut Lavril, and 23 others. 2024. Pixtral 12B. *CoRR*, abs/2410.07073.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, and 45 others. 2025a. Qwen3-VL Technical Report. *CoRR*, abs/2511.21631.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Ming-Hsuan Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, and 8 others. 2025b. Qwen2.5-VL Technical Report. *CoRR*, abs/2502.13923.

Jonas Belouadi, Simone Paolo Ponzetto, and Steffen Eger. 2024. DeTikZify: Synthesizing Graphics Programs for Scientific Figures and Sketches with TikZ. In *NeurIPS*.

Blockly. 2022. Games for Tomorrow's Programmers. https://blockly.games/.

Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, Bin Li, Ping Luo, Tong Lu, Yu Qiao, and Jifeng Dai. 2023. InternVL: Scaling up Vision Foundation Models and Aligning for Generic Visual-Linguistic Tasks. *CoRR*, abs/2312.14238.

CodeHS. 2025. Turtle Graphics with Tracy the Turtle. https://codehs.com/hourofcode/tracy. Accessed: 2025-09-01.

Code.org. 2013. Code.org: Learn Computer Science. https://code.org/.

Wenliang Dai, Nayeon Lee, Boxin Wang, Zhuoling Yang, Zihan Liu, Jon Barker, Tuomas Rintamaki, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2024. NVLM: Open Frontier-Class Multimodal LLMs. *CoRR*, abs/2409.11402.

Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, Jiasen Lu, Taira Anderson, Erin Bransom, Kiana Ehsani, Huong Ngo, Yen-Sung Chen, Ajay Patel, Mark Yatskar, Chris Callison-Burch, and 32 others. 2024. Molmo and PixMo: Open Weights and Open Data for State-of-the-Art Multimodal Models. *CoRR*, abs/2409.17146.

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI*.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*.

Shuchi Grover and Roy Pea. 2013. Computational thinking in K–12: A review of the state of the field. *Educational researcher*.

Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, and Xiangliang Zhang. 2025. UICopilot: Automating UI Synthesis via Hierarchical Code Generation from Webpage Designs. In *WWW*.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual Programming: Compositional visual reasoning without training. In *CVPR*.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SIGOPS*.

Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Yanwei Li, Ziwei Liu, and Chunyuan Li. 2024a. LLaVA-OneVision: Easy Visual Task Transfer. *CoRR*, abs/2408.03326.

Hongyu Li, Liang Ding, Meng Fang, and Dacheng Tao. 2024b. Revisiting Catastrophic Forgetting in Large Language Model Tuning. In *EMNLP (Findings)*.

Kaixin Li, Yuchen Tian, Qisheng Hu, Ziyang Luo, Zhiyong Huang, and Jing Ma. 2024c. MMCode: Benchmarking Multimodal Large Language Models for Code Generation with Visually Rich Programming Problems. In *EMNLP (Findings)*.

Wen-Ding Li and Kevin Ellis. 2024. Is Programming by Example Solved by LLMs? In *NeurIPS*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *NeurIPS*.

Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. 2024. MathVista: Evaluating Mathematical Reasoning of Foundation Models in Visual Contexts. In *ICLR*.

John H. Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.*

Jeremy Marbach, Alexandra Maximova, and Jacqueline Staub. 2022. A Tool to Create and Conduct Custom Assessments in Turtle Graphics. In *ISSEP*.

OpenAI. 2024a. GPT-4o. https://openai.com/index/hello-gpt-4o/.

OpenAI. 2024b. GPT-4v. https://openai.com/index/gpt-4v-system-card/.

OpenAI. 2025a. GPT-5. https://openai.com/gpt-5/.

OpenAI. 2025b. o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/.

Victor-Alexandru Padurean and Adish Singla. 2024. Benchmarking Generative Models on Computational Thinking Tests in Elementary Visual Programming. In *NeurIPS Track on Datasets and Benchmarks*.

Roy D Pea. 1987. Logo programming and problem solving.

Python. 2024. Python Turtle Graphics. https://docs.python.org/3/library/turtle.html.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *ICML*.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-Shot Text-to-Image Generation. In *ICML*.

Sina Rismanchian, Yasaman Razeghi, Sameer Singh, and Shayan Doroudi. 2025. TurtleBench: A Visual Programming Benchmark in Turtle Geometry. In *NAACL*.

Juan A. Rodriguez, Abhay Puri, Shubham Agarwal, Issam H. Laradji, Sai Rajeswar, David Vázquez, Christopher Pal, and Marco Pedersoli. 2025. StarVector: Generating Scalable Vector Graphics Code from Images and Text. In *AAAI*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code Llama: Open Foundation Models for Code. *CoRR*, abs/2308.12950.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2025. Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering. In *NAACL*.

Adish Singla. 2023. Evaluating ChatGPT and GPT-4 for Visual Programming. In *ICER - Volume 2*.

Jacqueline Staub. 2021. Logo Environments in the Focus of Time. *Bulletin of EATCS*.

Turtle Academy. 2025. Turtle Academy. https://turtleacademy.com/. Accessed: 2025-09-01.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. 2024. Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. *CoRR*, abs/2409.12191.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.

Chao Wen, Ahana Ghosh, Jacqueline Staub, and Adish Singla. 2024. Task Synthesis for Elementary Visual Programming in XLogoOnline Environment. In *AIED Track on Late Breaking Results*.

Chao Wen, Tung Phung, Pronita Mehrotra, Sumit Gulwani, Roger E. Beaty, Tomohiro Nagashima, and Adish Singla. 2025a. Exploration vs. Fixation: Scaffolding Divergent and Convergent Thinking for Human-AI Co-Creation with Generative Models. *CoRR*, abs/2512.18388.

Chao Wen, Jacqueline Staub, and Adish Singla. 2025b. Program Synthesis Benchmark for Visual Programming in XLogoOnline Environment. In *ACL*.

Chengyue Wu, Zhixuan Liang, Yixiao Ge, Qiushan Guo, Zeyu Lu, Jiahao Wang, Ying Shan, and Ping Luo. 2025. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. In *NAACL (Findings)*.

XLogoOnline. 2024. XLogoOnline Platform. https://xlogo.inf.ethz.ch/.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. WizardLM: Empowering Large Pre-Trained Language Models to Follow Complex Instructions. In *ICLR*.

Cheng Yang, Chufan Shi, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, Gongye Liu, Xiaomei Nie, Deng Cai, and Yujiu Yang. 2025. ChartMimic: Evaluating LMM's Cross-Modal Reasoning Capability via Chart-to-Code Generation. In *ICLR*.

Xiang Yue, Yuansheng Ni, Tianyu Zheng, Kai Zhang, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, and 3 others. 2024. MMMU: A Massive Multi-Discipline Multimodal Understanding and Reasoning Benchmark for Expert AGI. In *CVPR*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. STaR: Bootstrapping Reasoning With Reasoning. In *NeurIPS*.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2024a. AgentTuning: Enabling Generalized Agent Abilities for LLMs. In *ACL (Findings)*.

Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng, Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, and 36 others. 2024b. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools. *CoRR*, abs/2406.12793.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *ACL*.

Jinguo Zhu, Weiyun Wang, Zhe Chen, Zhaoyang Liu, Shenglong Ye, Lixin Gu, Hao Tian, Yuchen Duan, Weijie Su, Jie Shao, Zhangwei Gao, Erfei Cui, Xuehui Wang, Yue Cao, Yangzhou Liu, Xingguang Wei, Hongjie Zhang, Haomin Wang, Weiye Xu, and 32 others. 2025. InternVL3: Exploring Advanced Training and Test-Time Recipes for Open-Source Multimodal Models. *CoRR*, abs/2504.10479.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 2 others. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *ICLR*.

Bocheng Zou, Mu Cai, Jianrui Zhang, and Yong Jae Lee. 2024. VGBench: Evaluating Large Language Models on Vector Graphics Understanding and Generation. In *EMNLP*.

A Table of Contents

In this section, we briefly summarize the contents of the paper's appendices.

- Appendix B provides details about broader impacts and declaration of LLM usage.
- Appendix C provides more details about the dataset generation and labeling process.
- Appendix D provides additional experiments and analysis.
- Appendix E provides more details about the reliability of the evaluation framework.
- Appendix F provides more details about the implementation of the benchmark and experiments.
- Appendix G provides a case study of model failures.
- Appendix H provides more details about the prompts used for fine-tuning and evaluation.

B Broader Impacts and Declaration of LLM Usage

B.1 Broader Impacts

This work introduces a benchmark for evaluating the performance of existing vision-language models (VLMs) in solving visual programming tasks in the Turtle Graphics domain. In addition, we propose a novel data generation technique, TurtleAI-Datagen, for generating large-scale synthetic data to train VLMs.

Our benchmark and data generation technique have several positive broader impacts: (i) our work can facilitate programming education, especially in K-12 settings where Turtle Graphics is commonly used; (ii) our work can help track and improve VLMs' ability to understand and generate Turtle Graphics code, making it easier for both beginners and experts to create vector graphics and complex geometric art; and (iii) our data generation technique can advance synthetic data generation in fields where real data is scarce or difficult to collect, potentially helping researchers and developers build better models in a variety of domains beyond Turtle Graphics.

However, it is essential to acknowledge the potential risks associated with synthetic data generation. For instance, our framework could be misused to generate images with political or sensitive content. We emphasize the need for careful oversight and ethical considerations in the application of our framework to ensure that it is used responsibly and for the benefit of society.

B.2 Declaration of LLM Usage

We declare that large language models (LLMs) were used only to assist with writing and formatting the manuscript. LLMs were not involved in the design of experiments, analysis of results, or any other important aspects in this paper.

C Additional Details About the Dataset Generation and Labeling Process

In this section, we provide detailed information and generation processes for the datasets used in this paper. We then describe our labeling process for task categories, difficulty levels, and failure types.

C.1 Dataset License

The TurtleAI-DSReal dataset is collected from the visual programming platform XLogoOnline and is licensed under CC BY-NC 4.0.[4].

The datasets TurtleAI-Train and TurtleAI-DSSyn were generated with vision-language models. The models used and their licenses are listed below:

- *Qwen2-VL-72B-Instruct* uses Qwen License Agreement.[5]
- *Pixtral-Large* uses Mistral Research License (MRL) for research/educational use.[6]
- *Llama 3.1-70B-Instruct* uses Llama 3.1 Community License Agreement.[7]

C.2 Dataset Generation Process

In this subsection, we provide more details about the generation process for different datasets used in this paper, including the evaluation datasets TurtleAI-DSReal, TurtleAI-DSCraft, and TurtleAI-DSSyn, and the training dataset TurtleAI-Train. Table 3 gives a summary of the datasets used in this paper.

[4] https://xlogo.inf.ethz.ch/terms.html

[5] https://huggingface.co/Qwen/Qwen2-VL-72B-Instruct/blob/main/LICENSE

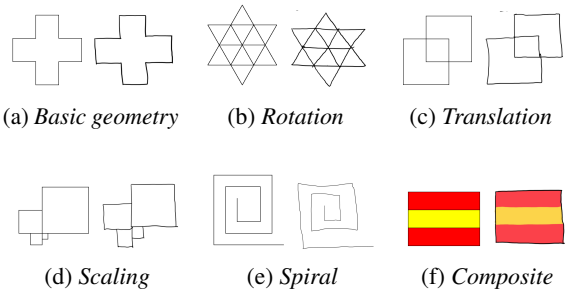
[6] https://mistral.ai/news/pixtral-large

[7] https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/LICENSE

Dataset	# Samples	Purpose	Seed Dataset	Seed Size
TurtleAI-DS	823	Evaluation	-	-
TurtleAI-DSReal	102	Evaluation	-	-
TurtleAI-DSCraft	102	Evaluation	TurtleAI-DSReal	102
TurtleAI-DSSyn	619	Evaluation	TurtleAI-DSReal	102
TurtleAI-Train	738,126	Train & Validation	Manually curated	10

Table 3: A summary of the datasets used in this paper.

TurtleAI-DSReal. This dataset is curated from the visual programming platform XLogoOnline. The tasks in this platform are carefully designed by domain experts and have been used by tens of thousands of students for learning programming every year (Staub, 2021). The involvement of domain experts ensures that this dataset includes a diverse range of high-quality tasks, reflecting real-world learning scenarios.



(a) *Basic geometry* (b) *Rotation* (c) *Translation*

(d) *Scaling* (e) *Spiral* (f) *Composite*

Figure 7: Examples of the reference images and the corresponding hand-drawn images in the dataset TurtleAI-DSCraft. The reference images are shown on the left, and the corresponding hand-drawn images are shown on the right. One example is shown for each task category.

TurtleAI-DSCraft. This dataset is generated by manually drawing the task images from TurtleAI-DSReal using a drawing tool. Specifically, we use each task image in TurtleAI-DSReal as the reference image and ask a human without any prior knowledge of Turtle Graphics or professional drawing skills to manually draw the task image using a digital drawing tool (i.e., an iPad). Finally, we replace each task image in TurtleAI-DSReal with the corresponding hand-drawn image, resulting in the dataset TurtleAI-DSCraft. Figure 7 shows some examples of the reference images and the corresponding hand-drawn images in the dataset TurtleAI-DSCraft. This dataset can be used to evaluate the generalization capabilities of the model to real-world drawing tasks.

TurtleAI-DSSyn. The dataset TurtleAI-DSSyn is generated by our proposed data synthesis framework TurtleAI-Datagen. Specifically, we use TurtleAI-DSReal (Size 102) as the seed dataset for TurtleAI-Datagen. We run our TurtleAI-Datagen for 3 iterations; in each iteration, we keep the top 30% of the generated samples for the next iteration of TurtleAI-Datagen, resulting in a synthetic dataset with $8,214$ image-code pairs. To further ensure the quality, we manually select from this dataset using the rubrics defined in the elite selection stage and make a binary decision on whether to keep an image-code pair. When making the decision, we adopt the rubrics used in the elite selection stage of TurtleAI-Datagen, including (i) geometric structure and symmetry, (ii) visual appeal, clarity, and simplicity, (iii) structural coherence, (iv) alignment and positioning, (v) educational value and solvability, and (vi) color usage and necessity. When evaluating the quality of each sample, we make a binary decision (i.e., "good" or "bad") for each dimension, and only keep the sample if all dimensions are evaluated as "good". After this process, we obtain the final dataset TurtleAI-DSSyn with 619 high-quality samples. Note that we do not apply the CoT labeling for generating TurtleAI-DSSyn since this dataset is not used for training. The implementation details of TurtleAI-Datagen are provided in Appendix F.1.

TurtleAI-DS. The dataset TurtleAI-DS is a union of TurtleAI-DSReal, TurtleAI-DSCraft, and TurtleAI-DSSyn datasets, including a total of $102 + 102 + 619 = 823$ samples.

TurtleAI-Train. The dataset TurtleAI-Train is a large-scale training dataset containing $738,126$ samples. This dataset is generated using a seed dataset with only 10 seed examples. These seed examples are provided in Figure 8a and are

selected based on the following three principles:

- Minimal manual effort: The set should be as small as possible to reduce manual effort.
- Simplicity: The pairs should be easy to design and understand.
- Conceptual diversity: The set should cover a broad range of geometric transformation concepts.

By following these principles, we arrived at a set of 10 pairs that is both minimal and simple, while remaining diverse. These pairs capture a range of geometric transformation types observed in our domain, including:

- Adding or removing edges (e.g., transforming a triangle into a square or vice versa)
- Rotating shapes (e.g., turning a square into a diamond)
- Translating shapes (e.g., placing two squares side by side)
- Scaling (e.g., comparing a large square with a smaller one)
- Changing edge color (e.g., a square with red edges)
- Changing fill color (e.g., a red-filled square)
- Combining shapes (e.g., combining a square and a triangle)

After preparing the seed examples, we generate the training dataset using the same settings as generating TURTLEAI-DSSyn, except that (i) we use a different seed dataset with only 10 manually designed examples; (ii) we iterate 5 times with TURTLEAI-Datagen; (iii) we use $k = 70\%$ for the elite selection stage; and (iv) we apply the CoT labeling after iterating 5 times. Examples of generated samples and the generated CoT label are shown in Figure 8.

C.3 Labeling Process

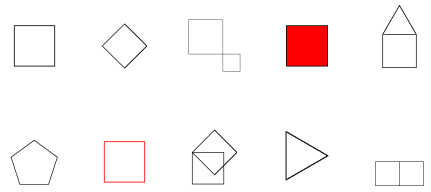
We describe the dataset labeling process for the task categories and difficulty levels in the evaluation datasets.

Labeling process of task categories. We identify 6 different geometric categories of task images based on the visual patterns and transformations in the evaluation datasets. The definitions of these categories are as follows:

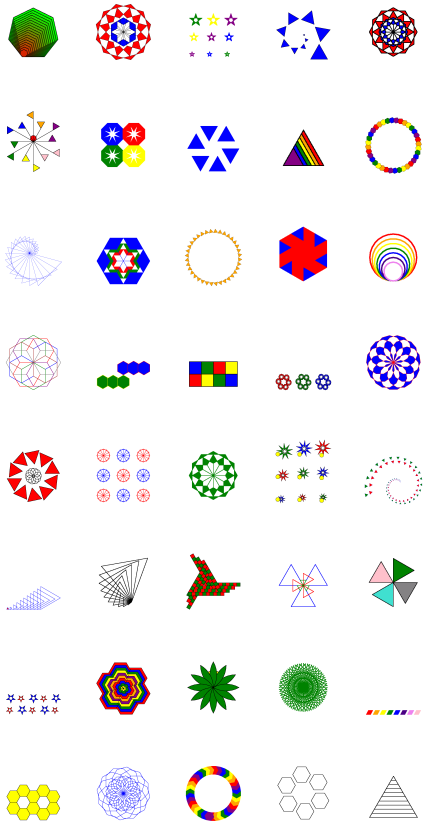
- *Basic geometry*: simple and basic shapes like squares, circles, triangles, and lines without complex patterns or arrangements. Tasks in this category require understanding of basic geometry.
- *Rotation*: patterns formed by rotating basic geometric shapes around a central point to create symmetrical designs, such as a spirograph. These tasks require reasoning about rotation angles and the numbers of repetitions.
- *Translation*: patterns formed by translating a basic pattern to different positions, forming tiling or grid structures. Tasks in this category require reasoning about translation distance and the numbers of basic patterns.
- *Scaling*: patterns formed by scaling basic geometric shapes, creating nested or expanding structures. Tasks in this category require reasoning about scaling factors and the numbers of repetitions.
- *Spiral*: sequential shapes arranged in spiraling paths, creating dynamic patterns with radial symmetry, like an Archimedean spiral. Tasks in this category require reasoning about the spiral pattern, the numbers of repetitions, the degrees of rotation, and scaling factors.
- *Composite*: complex arrangements combining different transformations (scaling, rotation, spiral, and translation) with varied shapes or colors.

Given above definitions, we manually label each task-code pair in the dataset TURTLEAI-DS into one of the 6 categories. Note that an image may involve multiple transformations, we categorize each image based on its predominant geometric characteristic.

Labeling process of difficulty levels. We assigned a difficulty level for each task-code pair in the dataset TURTLEAI-DS. The assigned difficulty level is based on the complexity of the visual patterns, and difficulty of drawing the image using Turtle Graphics. We define 3 difficulty levels as follows:



(a) All the 10 task images in the seed dataset.

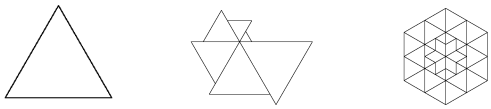


(b) Generated images in TurtleAI-Train.

```
## Image Description

The image displays a vibrant, symmetrical spiral pattern composed of nested
heptagons (seven-sided polygons). The heptagons decrease in size as they spiral
inward, creating a dynamic and visually appealing effect. The colors transition
smoothly from green on the outer edges to red at the center, enhancing the depth
and dimensionality of the spiral. The overall structure is highly symmetrical,
with each heptagon rotated slightly relative to the previous one, contributing to
the spiral effect.

## Analysis & Solution Code

To create the pattern shown in the image using Turtle Graphics,
we need to follow these steps:

1. Initialize the Turtle: Set up the turtle graphics environment and configure the
drawing speed.
2. Define the Heptagon: Create a function to draw a heptagon with a specified size.
3. Draw the Spiral: Use a loop to draw a series of heptagons, decreasing their
size and adjusting their position and rotation with each iteration to form a
spiral.
4. Color Gradient: Apply a color gradient to the heptagons, transitioning from
green to red as the size decreases.

Here is the code with comprehensive docstrings and comments to
create the pattern:

'''python
def draw(t):
    """
    Draws a spiral pattern of nested heptagons with a color
    gradient from green to red.
    """
    t.speed(0) # Set the drawing speed to the maximum
    angle = 10 # Initial angle for the spiral rotation
    size = 100 # Initial size for the heptagons

    def draw_heptagon(size):
        """
        Draws a heptagon with the given size.
        """
        for _ in range(7):
            t.forward(size) # Draw each side of the heptagon
            t.left(360 / 7) # Turn to the next side

    # Draw the spiral pattern of heptagons
    while size > 10:
        # Set the fill color with a gradient from green to red
        t.fillcolor((1 - size / 100, size / 100, 0))
        t.begin_fill() # Start filling the shape
        draw_heptagon(size) # Draw the heptagon
        t.end_fill() # End filling the shape
        t.right(angle) # Rotate right to create the spiral effect
        size -= 5 # Decrease the size of the heptagon for the next iteration
        t.left(10) # Adjust the rotation slightly to maintain the spiral pattern
'''
```

(c) Generated CoT label for the first image in Figure 8b.

Figure 8: Examples of images in the seed dataset and the TurtleAI-Train dataset. The seed dataset includes 10 task images and their corresponding solution codes. The TurtleAI-Train dataset includes 738k images generated from the seed dataset using our data generation technique TurtleAI-Datagen. TurtleAI-Datagen can generate diverse and high-quality images by evolving from a small seed dataset.



(a) *Easy* (b) *Medium* (c) *Hard*

Figure 9: Examples showing images of different difficulty levels from the TURTLEAI-DS dataset.

- *Easy*: basic patterns consisting of single geometric shapes like squares, circles, or lines without complex combinations. Tasks in this category require entry-level visual understanding of geometry, basic math reasoning about transformation parameters (e.g., length, angles), and fundamental programming skills to implement simple geometric shapes.
- *Medium*: patterns involving combinations of basic shapes with one or more transformations. Tasks in this category require the ability to decompose patterns into simpler components, understand relationships between different transformations, and programming skills to implement multiple transformation steps.
- *Hard*: sophisticated patterns featuring multiple complex transformations. Tasks in this category require high-level visual understanding, reasoning about spatial and temporal relationships between components, mathematical reasoning about transformation parameters (e.g., length, angles, scaling factors), and programming skills to convert complex reasoning into executable programs.

Given above definitions, we manually label each image in the dataset TURTLEAI-DS into one of the 3 difficulty levels.

Labeling process of failure types. We identify 7 different failure types based on the failure cases of different models on the TURTLEAI-DSReal dataset. The definitions of these failure types are as follows:

- *Visual understanding*: This involves misinterpreting the image's overall design, layout, or composition, leading to code generation that does not match the target image. For instance, the image shows a square, but the model describes it as a circle and generates the code for a circle.
- *Decomposition*: This involves errors in decomposing the image into feasible drawing steps or errors in decomposing a complex pattern into its constituent parts (shapes, patterns, elements). For example, when a complex pattern results from repeated rotations of a simple shape. The model may fail to decompose it into the base shape and its transformations, instead treating the entire pattern as an indivisible unit. This leads to either an attempt to generate code that represents the entire complex pattern directly or a failure to plan a feasible sequence of steps to generate the pattern.
- *Spatial reasoning*: This involves errors in understanding relative positions, distances, angles, and sizes of patterns within the image. For instance, the model misinterprets the relative positioning of two shapes, placing one above the other when they are actually side-by-side.
- *Programming*: This involves the model having correct visual understanding and reasoning but the code implementation is not consistent with the visual reasoning results or the code implementation contains syntax or logical errors. For instance, the model understands correctly that the image shows a square, but implements the code for a circle instead.
- *Visual precision*: This involves the model having correct visual understanding and reasoning, but failing to achieve very precise details during the code implementation. For instance, the model generates code that captures the overall structure but deviates in specifics, such as lines being slightly too long, angles that are a few degrees off.
- *Repetition*: This involves unnecessary or incorrect repetition of code blocks. For instance, the model keeps generating the same redundant code repeatedly without stopping.
- *Evaluation error*: This is due to the evaluation framework's incorrect evaluation results that are not consistent with the manual evaluation results. For instance, the symbolic comparison incorrectly identifies the generated image as a *success*, but it's actually a *fail* from the manual evaluation.

D Additional Experiments and Analysis

In this section, we provide additional experiments and analysis of our synthetic data generation tech-

nique TURTLEAI-Datagen and the model performance on TURTLEAI.

D.1 Analysis of the Size of the Dataset Generated by TURTLEAI-Datagen

We analyze the exponential growth rate of datasets generated by the TURTLEAI-Datagen framework. Specifically, we derive a mathematical formulation that describes how the dataset size expands iteratively, starting from an initial seed dataset and growing with each iteration. Formally, assume that we have an initial seed dataset $\mathcal{D}_0$ containing $|\mathcal{D}_0|$ samples. Starting from $\mathcal{D}_0$, the framework generates a dataset $\mathcal{D}_t$ after t iterations, where the size $|\mathcal{D}_t|$ can be expressed as:

$$\begin{aligned} |\mathcal{D}_t| &= |\mathcal{D}_{t-1}| \cdot p \cdot k \cdot (1 - d_{t-1}) \\ &= |\mathcal{D}_0| \cdot (pk)^t \cdot \prod_{i=1}^{t} (1 - d_{i-1}), \end{aligned} \tag{1}$$

where $p \geq 1$ is the number of pairs of code sampled from the dataset in each iteration as the reference-guided code-to-code mutation example (e.g., $p = 4$, 8, 16), $d_i \in [0, 1]$ is the duplicate rate in the i-th iteration, and $k \in [0, 1]$ is the top percentage of samples selected from the dataset in the elite selection stage. If we assume that the duplicate rate is the same for all iterations (i.e., $d_i = d$), then the size of the dataset $|\mathcal{D}_t|$ can be simplified as:

$$|\mathcal{D}_t| = |\mathcal{D}_0| \cdot \big(pk(1 - d)\big)^t, \tag{2}$$

where the size of the dataset $|\mathcal{D}_t|$ grows exponentially with the number of iterations t with a growth rate of $pk(1 - d)$. In our implementation, we set $p = 16$ and $k = 0.7$ for generating TURTLEAI-Train, resulting in a growth rate of around 9.7 at each iteration.

D.2 Scaling of Fine-tuning Performance with Dataset Size

We study how fine-tuning performance scales with the size of datasets generated by TURTLEAI-Datagen across different iterations. To this end, we fine-tune Pixtral-12B on datasets from each iteration and evaluate the resulting models, as shown in Figure 10. The dataset grows exponentially with the number of iterations, at a rate of roughly 9.7, meaning each iteration produces a dataset approximately 9.7 times larger than the previous one. Performance improvements depend on the dataset: for TURTLEAI-DSReal, fine-tuning scales linearly

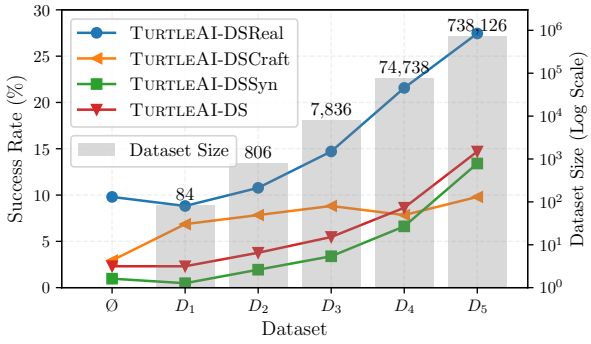


Figure 10: Performance of fine-tuned Pixtral-12B-TURTLE using datasets generated by TURTLEAI-Datagen across different iterations.

with dataset size, while for TURTLEAI-DSSyn, the improvement is nearly exponential. In contrast, for the out-of-distribution dataset TURTLEAI-DSCraft, performance saturates after the first iteration and remains stable in subsequent iterations. These results suggest that exponentially larger datasets generated by TURTLEAI-Datagen can improve fine-tuning performance, although out-of-distribution datasets like TURTLEAI-DSCraft do not benefit from the increased data.

D.3 Analysis of Out-of-distribution Performance of Fine-tuned Models

We analyze fine-tuned models on both in-domain and out-of-domain out-of-distribution (OOD) tasks, providing insights into their generalization capabilities. Results are shown in Table 4.

We examine whether fine-tuned models can solve in-domain out-of-distribution tasks by comparing them with their base models on DSReal and the hand-drawn OOD dataset DSCraft. DSCraft is created by hand-drawing images from DSReal tasks and thus serves as an in-domain OOD dataset. As shown in Table 4a, fine-tuning consistently improves performance on DSReal, but offers little benefit on DSCraft for moderately performing models (Qwen2-VL-72B) and noticeable gains for weaker ones (Qwen2-VL-7B, Pixtral-12B). We hypothesize that the CoT labeling in the data generation process may enhance OOD performance, as it generates image descriptions that help ignore irrelevant variations in hand-drawn images. To test this, we ablate the CoT labeling step in the data generation and fine-tune Pixtral-12B to obtain Pixtral-12B-TURTLE (w/o CoT). As shown in Table 4a, this model fails on DSCraft despite outperforming Pixtral-12B on DSReal, showing that CoT labeling

	DSReal	DSCraft
Qwen2-VL-72B	11.76	**7.84**
Qwen2-VL-72B-TURTLE	**35.29**	6.86
Qwen2-VL-7B	0.98	0.00
Qwen2-VL-7B-TURTLE	**28.43**	**6.86**
Pixtral-12B	9.80	2.94
Pixtral-12B-TURTLE (w/ CoT)	**27.45**	**9.80**
Pixtral-12B-TURTLE (w/o CoT)	22.55	0.00

(a) Success rates (%) on in-domain datasets.

	HumanEval+	MBPP+
Qwen2-VL-72B	**85.4**	**77.5**
Qwen2-VL-72B-TURTLE	67.7	73.5
Qwen2-VL-7B	**70.7**	**55.3**
Qwen2-VL-7B-TURTLE	28.0	33.1

(b) Success rates (%) on out-of-domain datasets.

Table 4: (a) Success rates (%) on in-domain datasets; DSCraft contains hand-drawn OOD tasks from the same domain. (b) Pass@1 success rates (%) on out-of-domain program synthesis benchmarks; fine-tuned models are not tuned for these tasks.

is useful for OOD generalization.

We investigate whether fine-tuning affects performance on out-of-domain tasks. To this end, we test fine-tuned Qwen2-VL models against their base versions on out-of-domain benchmarks, including HumanEval+ and MBPP+ (Liu et al., 2023). Pass@1 success rates are shown in Table 4b. The results show a clear drop in performance after fine-tuning, consistent with prior work showing that fine-tuning may lead to some degree of forgetting (Zeng et al., 2024a; Li et al., 2024b). This forgetting issue is more pronounced in the 7B model, suggesting that smaller models are more prone to overfitting and losing general capabilities. The 72B model retains better generality, which might be due to the higher capacity in retaining general knowledge.

D.4 Influence of the CoT Prompting on Model Performance

We investigate the influence of the CoT prompting on base models' performance. We experiment with various open-source VLMs, with and without CoT prompting. For CoT prompting, we require the model to generate the solution code in the following step-by-step manner: (i) describe the image in detail, (ii) analyze the image and propose steps to create the pattern, and (iii) generate the solution code with comprehensive docstrings and comments. For non-CoT prompting, we only require the model to generate the code, without the above steps explicitly mentioned. The comparison results are shown in Table 5. The results indicate that the effectiveness of CoT prompting varies across different models and datasets, and there is no clear indication that CoT prompting can improve performance in our domain. For instance, Qwen2-VL-7B shows improved performance with CoT prompting on both datasets, whereas InternVL2-76B performs better without CoT prompting on both datasets. This inconsistency may stem from the reasoning-intensive nature of our tasks, where each type of task demands different reasoning steps, making it challenging to devise a consistent CoT prompting strategy applicable to all tasks. Furthermore, models trained on different datasets may develop distinct reasoning preferences, causing the same CoT strategy to enhance performance in some models while potentially confusing others, resulting in inconsistent performance of CoT prompting in our domain.

D.5 Influence of LoRA Rank and Vision Tower for Fine-tuning Performance

We investigate the influence of LoRA rank and vision tower fine-tuning on model performance. To do this, we conduct fine-tuning experiments on Pixtral-12B with LoRA ranks of 64, 128, and 256 using the 738k TURTLEAI-Train dataset (without CoT labeling), training each configuration for 1 epoch. For each LoRA rank, we set the LoRA alpha parameter to twice the rank value. Additionally, we examine the impact of freezing versus unfreezing the vision tower during fine-tuning. By unfreezing the vision tower, we enable parameter tuning of the visual encoder component of the VLM, allowing the model to adapt its visual representations during fine-tuning. The results are shown in Table 6. We find that unfreezing the vision tower can enhance performance. Specifically, in our experiments with LoRA rank 64, unfreezing the vision tower increases the success rate from 10.78% to 17.65% on TURTLEAI-DSReal and from 8.38% to 9.96% on TURTLEAI-DS. Additionally, the choice of LoRA rank also affects the performance, with rank 128 achieving the best results in our case.

	TurtleAI-DSReal		TurtleAI-DS	
	CoT	Non-CoT	CoT	Non-CoT
Pixtral-Large	**11.76**	10.78	4.74	**6.56**
Qwen2-VL-72B	10.78	**11.76**	**4.13**	3.52
Llava-OneVision-72B	**8.82**	4.90	**2.19**	**2.19**
InternVL2-76B	8.82	**11.76**	2.79	**3.28**
Molmo-72B	**9.80**	3.92	**2.92**	2.31
Pixtral-12B	1.96	**9.80**	1.09	**2.31**
Llava-OneVision-7B	**3.92**	**3.92**	0.97	**1.09**
Qwen2-VL-7B	**4.90**	0.98	**1.09**	0.12
Molmo-7B	0.00	0.00	**0.12**	0.00
InternVL2-8B	**2.94**	0.00	**0.73**	0.12

Table 5: Symbolic success rates of different base VLMs on TurtleAI-DSReal and TurtleAI-DS with and without CoT prompting. The best performance is highlighted in **bold** for each model.

	Fine-tuning Parameters		Success Rate (%)	
	Vision Tower	LoRA rank	TurtleAI-DSReal	TurtleAI-DS
Pixtral-12B-Turtle	Freeze	64	10.78	8.38
Pixtral-12B-Turtle	Unfreeze	64	17.65	9.96
Pixtral-12B-Turtle	Unfreeze	128	**22.55**	**12.67**
Pixtral-12B-Turtle	Unfreeze	256	15.69	10.81

Table 6: Influence of LoRA rank and vision tower on the performance of fine-tuning. We experiment with LoRA ranks of 64, 128, and 256, freezing and unfreezing the vision tower to fine-tune Pixtral-12B model using the 738k TurtleAI-Train dataset (without CoT labeling), with each setting trained for 1 epoch. Unfreezing the vision tower and using LoRA rank 128 yields the best performance.

D.6 Performance of VLMs Using Pass@K Metrics

In the main paper, we report the evaluation results of different VLMs on our benchmark using a greedy decoding strategy (i.e., `temperature=0`). To provide a more comprehensive evaluation, we also experiment with a random sampling strategy by randomly sampling N samples from the model and then calculating the Pass@K results. Following previous works (Rozière et al., 2023; Zhuo et al., 2025), we compute Pass@K results with random sampling by generating $N = 5$ samples with `top_p=0.95` and `temperature = 0.8`. Then we calculate Pass@1, Pass@3, and Pass@5, respectively. Although generating many more samples ($N \geq K$) is recommended to reduce bias, we adopt the lower bound due to limited computational resources. The results are provided in Table 7.

E Reliability of the Evaluation Framework

To assess the reliability of our evaluation framework, we perform a manual evaluation and compare it with the accuracies of both symbolic and embedding-based comparisons. To do this, we first perform a manual evaluation of all generated images in the TurtleAI-DS dataset from GPT-4o. This involves comparing the ground-truth image with the corresponding image produced by executing the generated code from GPT-4o, and manually verifying whether each generated image is visually identical to the ground-truth image. This manual evaluation involves a total of 823 image-code pairs. After this manual evaluation, we compare our results with both symbolic and embedding-based comparisons to evaluate the accuracy of our evaluation framework.

Accuracy of the symbolic comparison. After manual evaluation, we use our manual evaluation results as ground truth and calculate the precision, recall, F1 score, and accuracy for the results of

		Greedy Decoding	Random Sampling		
	Size	Pass@1	Pass@1	Pass@3	Pass@5
InternVL2	76B	3.28	2.72	4.79	5.83
Llava-OneVision	72B	2.19	2.09	4.02	5.35
Qwen2-VL	72B	3.52	3.18	5.53	6.93
Qwen2-VL-TURTLE	72B	19.56	16.79	25.65	29.77
InternVL2	8B	0.12	0.19	0.51	0.73
Llava-OneVision	7B	1.09	0.83	1.65	2.19
Qwen2-VL	7B	0.12	0.10	0.29	0.49
Qwen2-VL-TURTLE	7B	13.37	11.96	19.77	23.69
Pixtral	12B	2.31	1.48	3.23	4.25
Pixtral-TURTLE	12B	14.70	10.28	17.36	20.66

Table 7: Symbolic success rates (%) of VLMs on the dataset TURTLEAI-DS with greedy decoding and random sampling. For greedy decoding, we report Pass@1 using `temperature` $= 0$. For random sampling, we use `temperature` $= 0.8$ and `top_p` $= 0.95$, where for each Pass@K metric, we generate $N = 5$ samples.

	Positive (Symbolic)		Negative (Symbolic)
Positive (Manual)	74		5
Negative (Manual)	2		742
$Precision = 0.974$	$Recall = 0.937$	$F1 = 0.955$	$Accuracy = 0.991$

(a) Confusion matrix for the symbolic comparison.

	Positive (Embedding)		Negative (Embedding)
Positive (Manual)	69		10
Negative (Manual)	6		738
$Precision = 0.873$	$Recall = 0.920$	$F1 = 0.896$	$Accuracy = 0.981$

(b) Confusion matrix for the embedding-based comparison.

Table 8: Confusion matrices illustrating the accuracy of our evaluation framework by comparing the results of the symbolic and embedding-based comparisons against the manual comparison. The evaluation is conducted by manually annotating GPT-4o's results on the TURTLEAI-DS dataset. (a) shows the confusion matrix for the symbolic comparison, which demonstrates high accuracy with an F1 score of 0.955 when compared against manual evaluation. (b) shows the confusion matrix for the embedding-based comparison, achieving an F1 score of 0.896 at a threshold value of 0.95.

the symbolic comparison. The results are shown in Table 8a. Our symbolic comparison achieves a precision of 0.974, recall of 0.937, F1 score of 0.955, and accuracy of 0.991, showing that the symbolic comparison can correctly identify almost all of the generated images compared against the manual evaluation.

Accuracy of the embedding-based comparison. The embedding-based comparison first calculates a similarity score and then determines *success* or *fail* by comparing the similarity score against a threshold value. To determine the optimal threshold value, we plot how different threshold values affect the precision, recall, and F1 score, and select the threshold value that maximizes the F1 score. Figure 11 shows the precision, recall, and F1 score at different threshold values. We find that using a threshold of 0.95 achieves the highest F1 score of 0.896. Therefore, we use a threshold of 0.95 for the embedding-based comparison, i.e., if the embedding score is greater than 0.95, we consider the generated image as a *success*. Table 8b shows detailed statistics of the precision, recall, F1 score, and accuracy for the embedding-based comparison with a threshold of 0.95.

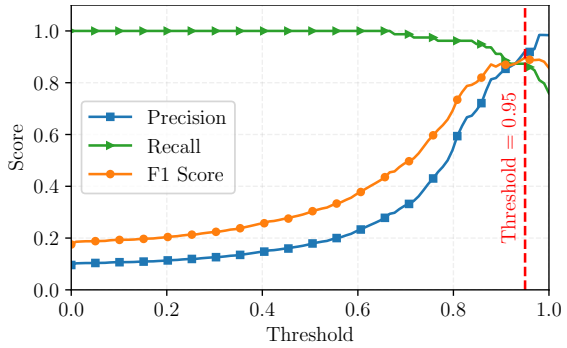


Figure 11: The relationship between precision, recall, and F1 score at different thresholds used in the embedding-based comparison. The best F1 score is achieved at a threshold of 0.95, with F1 score of 0.896.

F Implementation Details

In this section, we detail the implementation of our dataset generation framework TURTLEAI-Datagen, the model fine-tuning process, the evaluation process, and the evaluation framework TURTLEAI-Eval.

F.1 Implementation Details of TURTLEAI-Datagen

We describe the implementation details of the dataset generation framework TURTLEAI-Datagen.

(a) Images for $\texttt{C}_{\text{ref}_1}$ and $\texttt{C}_{\text{ref}_2}$

(b) Images for $\texttt{C}_{\text{in}}$ and $\texttt{C}_{\text{out}}$

```
- Introduce a loop: Add a loop to create multiple instances of a
shape with changing parameters.
- Parameterize shape attributes: Parameterize attributes such as
size, angle, or color to create a dynamic and changing pattern.
- Focus on the pattern: Remove or modify existing features to focus
on the new pattern created by the loop.
```

(c) Inferred mutation pattern $m(\texttt{C}_{\text{ref}_1}, \texttt{C}_{\text{ref}_2})$

Figure 12: An illustrative example for the reference-guided code mutation. (a) shows the corresponding images for a pair of sampled reference codes $(\texttt{C}_{\text{ref}_1}, \texttt{C}_{\text{ref}_2})$. (b) shows the corresponding images for $\texttt{C}_{\text{in}}$ and the corresponding mutated code $\texttt{C}_{\text{out}}$ by the LLM's inferred mutation pattern in (c).

Stage 1: Code mutation. We use the Llama3.1-70B-Instruct model for code mutation. The model is queried with `temperature` $= 0.5$ and `top_p` $= 1$. We use a higher temperature and `top_p` values to encourage the model to generate more diverse and creative code. During code mutation, we randomly sample 16 pairs of $(\texttt{C}_{\text{ref}_1}, \texttt{C}_{\text{ref}_2})$ from the seed dataset for each input code $\texttt{C}_{\text{in}}$. This results in 16 possible mutated codes for $\texttt{C}_{\text{in}}$ after applying the mutation for each pair of $(\texttt{C}_{\text{ref}_1}, \texttt{C}_{\text{ref}_2})$. An illustrative example of the code mutation process is provided in Figure 12.

Stage 2: Elite selection. The elite selection stage consists of two steps: deduplication and selection of elite samples. Given a dataset consisting of image-code pairs, we first perform deduplication to remove duplicate images. Specifically, we use a pre-trained ResNet18 (He et al., 2016) image encoder to obtain the embedding for each image. We then use the DBSCAN clustering algorithm (Ester et al., 1996) to cluster these image embeddings, resulting in a set of clusters, where each cluster consists of similar images. For each cluster, we only preserve one sample and remove the rest. We use the DBSCAN clustering algorithm with parameters $\epsilon = 0.2$, `min_samples` $= 2$, and the `euclidean` distance. After deduplication, we select the elite samples from the deduplicated dataset. To do this, we use Qwen2-VL-72B as the model for selecting elite samples. We use the top $k = 30\%$ for generating the TURTLEAI-DSSyn dataset and $k = 70\%$ for generating the TURTLEAI-Train dataset.

After the above two stages, we use Pixtral-Large as the model for CoT labeling. The model is queried with `temperature` $= 0.1$ and `top_p` $= 0.001$. Note that this stage is only used to generate the TURTLEAI-Train dataset for fine-tuning.

For querying the above models in different stages, we consistently use the vLLM inference engine to speed up inference. During inference, we use 8 $\times$ H100 GPUs, with `tensor_parallel_size` set to 8, and `max_num_seqs` set to 64. For every 100k samples generated during the elite selection (using Qwen2VL-72B-Instruct) or code mutation stage (using Llama3.1-70B-Instruct), the process takes approximately 8 hours. For CoT labeling (using Pixtral-Large), it takes around 13 hours to process every 100k samples.

F.2 Implementation Details of Fine-tuning

We conduct fine-tuning experiments on three models: Qwen2-VL-7B (Wang et al., 2024),

Qwen2-VL-72B (Wang et al., 2024), and Pixtral-12B (Agrawal et al., 2024). In our fine-tuning experiments, we use LoRA (Hu et al., 2022) for parameter-efficient fine-tuning. To determine the best LoRA rank and scaling factor, we experimented with LoRA ranks of 64, 128, and 256, using a scaling factor of two times the LoRA rank in each case. We found that a rank of 128 provides the best performance. Consequently, we use a LoRA rank of 128 and a scaling factor of 256 for all fine-tuning experiments. We also experimented with freezing and unfreezing the vision tower during fine-tuning and found that unfreezing the vision tower provides better performance. Therefore, we unfreeze the vision tower during fine-tuning for all fine-tuning experiments.

During fine-tuning, we use a learning rate schedule that combines a 10% warmup phase where the learning rate linearly increases to $1e-4$, followed by cosine annealing, which gradually reduces the learning rate to 0, ensuring stable training and smooth convergence (Zheng et al., 2024). We also reserve 1% of the 738k training dataset for validation. For our fine-tuning experiments on Qwen2-VL-7B and Pixtral-12B models, we observed that the validation loss increases after the first epoch, so we stopped fine-tuning after one epoch and report their performance at epoch 1 accordingly. Conversely, the Qwen2-VL-72B model's performance continued to improve after the first epoch, so we fine-tuned it for two epochs in total.

All fine-tuning experiments are conducted on an internal cluster using 8 × H100 GPUs, with each epoch taking approximately 15 hours for Qwen2-VL-7B, 112 hours for Qwen2-VL-72B, and 22 hours for Pixtral-12B.

F.3 Implementation Details of Evaluation

Inference details of VLMs. For open-source VLMs, we download their pre-trained weights and perform inference locally. We use the vLLM (Kwon et al., 2023) engine for VLM inference to obtain the outputs of the evaluated open-source VLMs. During inference, we set the `temperature` to 0 and use different numbers of GPUs for different models depending on their parameter sizes: (i) 1 × A100 GPU for models with parameter sizes less than 7B, (ii) 2 × A100 GPUs for models with parameter sizes between 7B and 70B, and (iii) 8 × A100 GPUs for models with parameter sizes larger than 70B. We set the vLLM parameter `tensor_parallel_size` to 1, 2, and 8 for the three cases, respectively, and set `max_num_seqs` to `tensor_parallel_size` × 8. We use the OpenAI API to evaluate proprietary models from OpenAI. For reasoning models, we set `reasoning_effort` to `medium` and `max_completion_tokens` to 8192.

Details of the evaluation procedure. For each task in our evaluation datasets, we provide the task image along with a fixed prompt template (see Figure 15) to guide the VLMs in generating Turtle Graphics Python code. The model's output often includes an explanation along with the predicted code. We extract only the code snippet and disregard the rest. If multiple code snippets are present, we handle them differently based on the comparison method:

- *Symbolic comparison*: we evaluate all the code snippets and consider the result a *success* if any code snippet is successful.
- *Embedding-based comparison*: we evaluate only the longest code snippet. This is because embedding-based comparison involves batch processing when extracting image embeddings, making it inefficient to consider multiple code snippets.

F.4 Implementation Details of the Evaluation Framework

We describe the implementation details of our evaluation framework. Given a task image `img`, the predicted code snippet $\hat{\texttt{C}}$, and the solution code `C`, our evaluation framework works as follows to evaluate the correctness of the predicted code $\hat{\texttt{C}}$. First, we execute both the solution code `C` and the predicted code $\hat{\texttt{C}}$ using a customized Turtle Graphics emulator. This emulator inherits the built-in Turtle Graphics module and enables us to record all the drawing states, including the coordinates and the colors when drawing a line, filling a polygon, etc. Second, we transform the recorded drawing states for `C` and $\hat{\texttt{C}}$ into the same space to ensure the invariance to size, position, and line width of the drawing. Specifically, we perform the following three steps:

- *Normalizing length of lines*: We rescale all recorded coordinates such that the maximum dimension of the entire pattern's bounding box is set to 300. This ensures that the drawings

Model	Version
o3	o3-2025-04-16 (`reasoning_effort=medium`) (OpenAI, 2025b)
o4-mini	o4-mini-2025-04-16 (`reasoning_effort=medium`) (OpenAI, 2025b)
GPT-5 (medium)	gpt-5-2025-08-07 (`reasoning_effort=medium`) (OpenAI, 2025a)
GPT-4o	gpt-4o-2024-11-20 (OpenAI, 2024a)
GPT-4V	gpt-4-turbo-2024-04-09 (OpenAI, 2024b)
Qwen3-VL-30B	Qwen/Qwen3-VL-30B-A3B-Instruct (Bai et al., 2025a)
Qwen2.5-VL-72B	Qwen/Qwen2.5-VL-72B-Instruct (Bai et al., 2025b)
Qwen2-VL-72B	Qwen/Qwen2-VL-72B-Instruct (Wang et al., 2024)
Qwen2-VL-7B	Qwen/Qwen2-VL-7B-Instruct (Wang et al., 2024)
Qwen2-VL-72B-TURTLE	Qwen2-VL-72B-Instruct (fine-tuned on TURTLEAI-Train)
Qwen2-VL-7B-TURTLE	Qwen2-VL-7B-Instruct (fine-tuned on TURTLEAI-Train)
Molmo-72B	allenai/Molmo-72B-0924 (Deitke et al., 2024)
Molmo-7B	allenai/Molmo-7B-D-0924 (Deitke et al., 2024)
Llava-OneVision-72B	llava-hf/llava-onevision-qwen2-72b-ov-chat-hf (Li et al., 2024a)
Llava-OneVision-7B	llava-hf/llava-onevision-qwen2-7b-ov-chat-hf (Li et al., 2024a)
NVLM-1.0-D	nvidia/NVLM-D-72B (Dai et al., 2024)
Pixtral-Large	mistralai/Pixtral-Large-Instruct-2411 (Agrawal et al., 2024)
Pixtral-12B	mistral-community/pixtral-12b (Agrawal et al., 2024)
Pixtral-12B-TURTLE	Pixtral-12B (fine-tuned on TURTLEAI-Train)
InternVL3-76B	OpenGVLab/InternVL3-78B (Zhu et al., 2025)
InternVL2-76B	OpenGVLab/InternVL2-Llama3-76B (Chen et al., 2023)
InternVL2-8B	OpenGVLab/InternVL2-8B (Chen et al., 2023)
GLM-4V-9B	THUDM/glm-4v-9b (Zeng et al., 2024b)

Table 9: Evaluated models and their versions.

are uniformly scaled regardless of their original size, making our comparison invariant to the size of the drawing.

- *Centering around the origin*: We translate all recorded lines so that they are centered around the origin. This involves calculating the center of the bounding box and shifting all coordinates accordingly. This ensures that the comparison is invariant to the position of the drawing.
- *Standardizing pen size*: We standardize the pen size of all lines to 1. This ensures that drawing line width does not affect the comparison of drawings, making our comparison invariant to the line width.

Third, we render these normalized drawing states into images in the sequence as they are recorded, resulting in standardized images $\texttt{img}$ and $\hat{\texttt{img}}$ for $\texttt{C}$ and $\hat{\texttt{C}}$, respectively. Finally, our evaluation framework provides two comparison methods to compare these two images, which are described in detail as follows.

Symbolic comparison. This compares the standardized images $\texttt{img}$ and $\hat{\texttt{img}}$ pixel-by-pixel. The high-level idea is to first count non-white pixels in both images and calculate the percentage of differing pixels among them. If this percentage is below a predefined value, the comparison result is *success*; otherwise, the comparison result is *fail*. More specifically, assume the images $\texttt{img}$ and $\hat{\texttt{img}}$ are of dimensions $H \times W$ and $\texttt{img}_{i,j}$ and $\hat{\texttt{img}}_{i,j}$ are the pixels at position (i, j), respectively. We define a candidate set of pixels $\mathcal{P}$ that are considered for symbolic comparison:

$$\mathcal{P} = \{(i,j) \mid \texttt{img}_{i,j} \neq \text{white} \vee \hat{\texttt{img}}_{i,j} \neq \text{white}, \\ \forall i \in \{1,\dots,H\},\, j \in \{1,\dots,W\}\}. \tag{3}$$

The pixel-wise difference between $\texttt{img}$ and $\hat{\texttt{img}}$ is computed as:

$$\text{pixel\_diff}(\texttt{img}, \hat{\texttt{img}}) = \frac{\sum_{(i,j)\in\mathcal{P}} \mathbb{I}(\texttt{img}_{i,j} \neq \hat{\texttt{img}}_{i,j})}{|\mathcal{P}|}, \tag{4}$$

where $\mathbb{I}(\texttt{img}_{i,j} \neq \hat{\texttt{img}}_{i,j})$ is the indicator function that returns 1 if $\texttt{img}_{i,j} \neq \hat{\texttt{img}}_{i,j}$ and 0 otherwise.

We establish a threshold for pixel-wise differences to determine whether the image $\hat{\texttt{img}}$ is a *success* to match img or not. If $\text{pixel\_diff}(\texttt{img}, \hat{\texttt{img}}) < 1 - \text{threshold}$, the image $\hat{\texttt{img}}$ is considered a *success* to match img; otherwise, it is considered a *fail*. For our symbolic evaluation, we use a threshold $= 0.95$ for drawings with fill colors and 0.92 for those without fill colors. The higher threshold for filled drawings (i.e., using `begin_fill()` and `end_fill()` in the code) is due to the typically larger candidate set $\mathcal{P}$ for these drawings. The pixel-wise similarity in filled areas can overshadow differences in non-filled areas, making them harder to detect. Thus, we set a stricter threshold for filled drawings.

Embedding-based comparison. This method compares the standardized images img and $\hat{\texttt{img}}$ within the embedding space. This is achieved by extracting image embeddings from both img and $\hat{\texttt{img}}$ using a pre-trained image encoder model and then calculating a similarity score between these embeddings using a distance metric. During implementation, when comparing two standardized images img and $\hat{\texttt{img}}$, we first resize them to 256x256 pixels, apply a center crop to 224x224 pixels, convert them to tensors, and normalize them using standard ImageNet statistics to ensure consistency and accuracy. Then we extract 512-dimensional feature vectors from these images using the ResNet18 model pre-trained on ImageNet (He et al., 2016).[8] The similarity score between these embeddings is computed using the Euclidean distance, normalized to the range $[0, 1]$, where a higher score indicates higher similarity in the embedding space. Images are processed in batches of 128 for efficient computation. Any pairs where image processing fails (e.g., empty images or images that are too large) are assigned a similarity score of 0. Finally, we select the optimal threshold $= 0.95$ because it achieves the best F1 score (see Figure 11). If the similarity score exceeds 0.95, the image $\hat{\texttt{img}}$ is considered a *success* in matching img; otherwise, it is considered a *fail*.

G Case Study of Failures

We provide a case study of different models' outputs on tasks in the TURTLEAI-DS dataset. We select five representative VLMs: GPT-4o, Pixtral-12B, Qwen2-VL-72B, Pixtral-12B-TURTLE, and Qwen2-VL-72B-TURTLE. To systematically analyze the failure cases, we enumerate all possible failure cases across different types of models and provide examples for each type.

Specifically, we categorize these 5 models into 3 types: proprietary model (GPT-4o), open-source base models (i.e., Pixtral-12B, Qwen2-VL-72B), and fine-tuned models (Pixtral-12B-TURTLE, Qwen2-VL-72B-TURTLE). Then we categorize the failure cases into 8 different possibilities and show examples for each possibility. These possibilities are summarized in Table 10. Figure 13 and Figure 14 show example model responses.

H Prompts

In this section, we provide the prompts used in TURTLEAI as follows:

- Figure 15 shows the prompt used for guiding VLMs in synthesizing code from a given image input.
- Figure 16 shows the prompt for the reference-guided code generation stage in our data synthesis framework TURTLEAI-Datagen.
- Figure 17 provides the prompt used for the elite selection stage in TURTLEAI-Datagen for scoring the quality of the generated geometric image.
- Figure 18 provides the prompt for the CoT labeling stage in TURTLEAI-Datagen.

[8] We use ResNet-18 primarily because it is a widely adopted, well-performing, and lightweight model (with only 11.7 million parameters) that offers a reasonable trade-off between performance and speed.

	GPT-4o	Pixtral-12B Qwen2-VL-72B	Pixtral-12B-TURTLE Qwen2-VL-72B-TURTLE	# Cases	Percentage
Table 11	✓	✓	✓	6	0.73%
N.A.	✗	✓	✓	0	0.00%
Table 12	✓	✗	✓	17	2.06%
Table 13	✓	✓	✗	3	0.36%
Table 14	✓	✗	✗	23	2.79%
N.A.	✗	✓	✗	0	0.00%
Table 15	✗	✗	✓	54	6.56%
Table 16	✗	✗	✗	591	71.81%

Table 10: Summary of different possible failure cases across different types of models on our benchmark tasks. We identify eight possible failure cases for the GPT-4o, open-source base models (Pixtral-12B and Qwen2-VL-72B), and fine-tuned models (Pixtral-12B-TURTLE and Qwen2-VL-72B-TURTLE). For each possible failure case, we provide the number of occurrences and the corresponding percentage, with references to detailed examples in corresponding figures. Success and failure are indicated by ✓ and ✗, respectively.

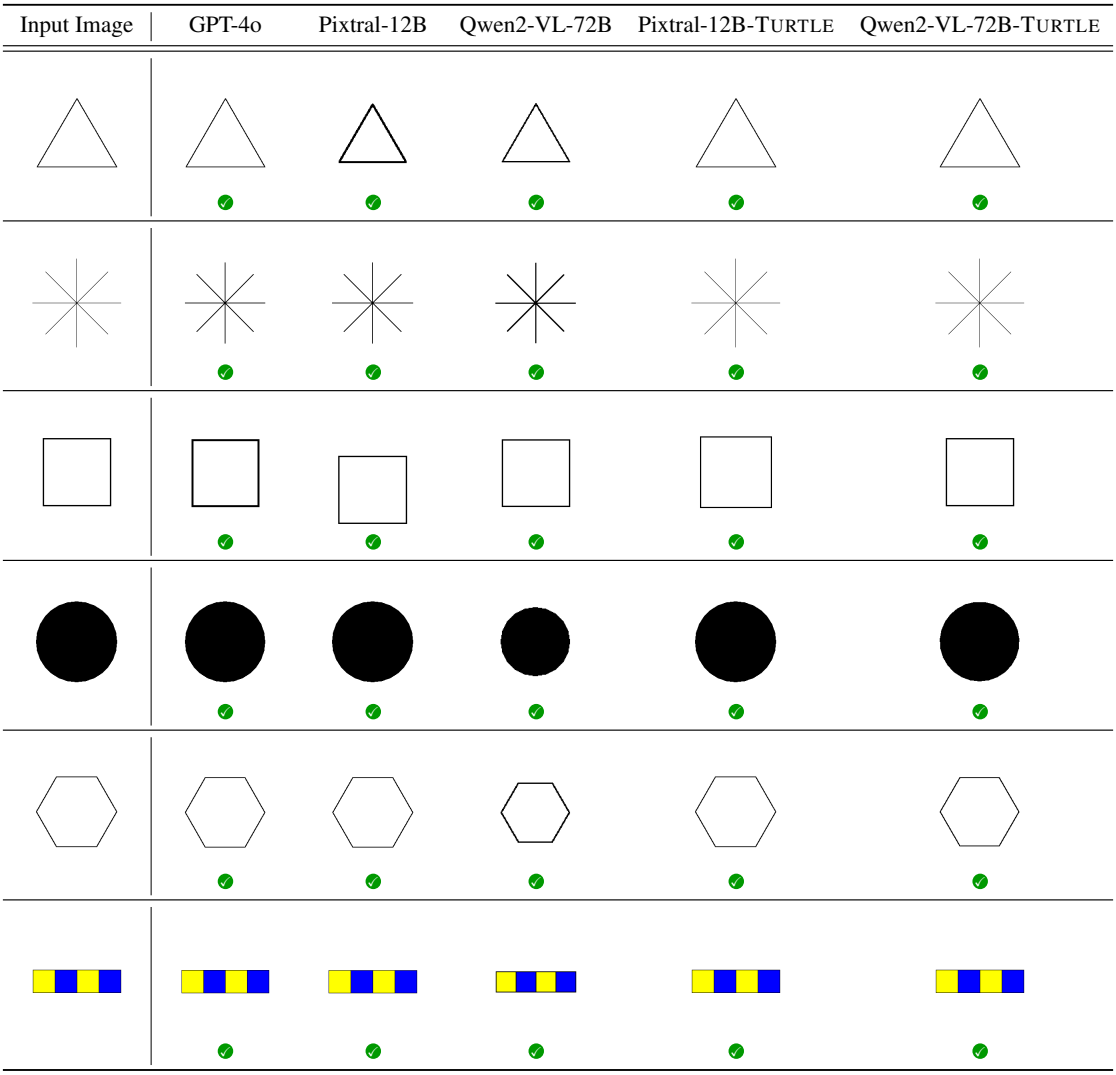


Table 11: Tasks that are successfully solved by base models (i.e., GPT-4o, Pixtral-12B, Qwen2-VL-72B) and our fine-tuned models (i.e., Pixtral-12B-TURTLE, Qwen2-VL-72B-TURTLE) in the TURTLEAI-DS dataset with 823 tasks. A total of 6 tasks (0.73%) match this criteria. Each row shows a ground truth image (leftmost) followed by the corresponding images generated by executing the each model's generated Python code. Success (✓) and failure (✗) are determined by our evaluation framework using symbolic comparison.

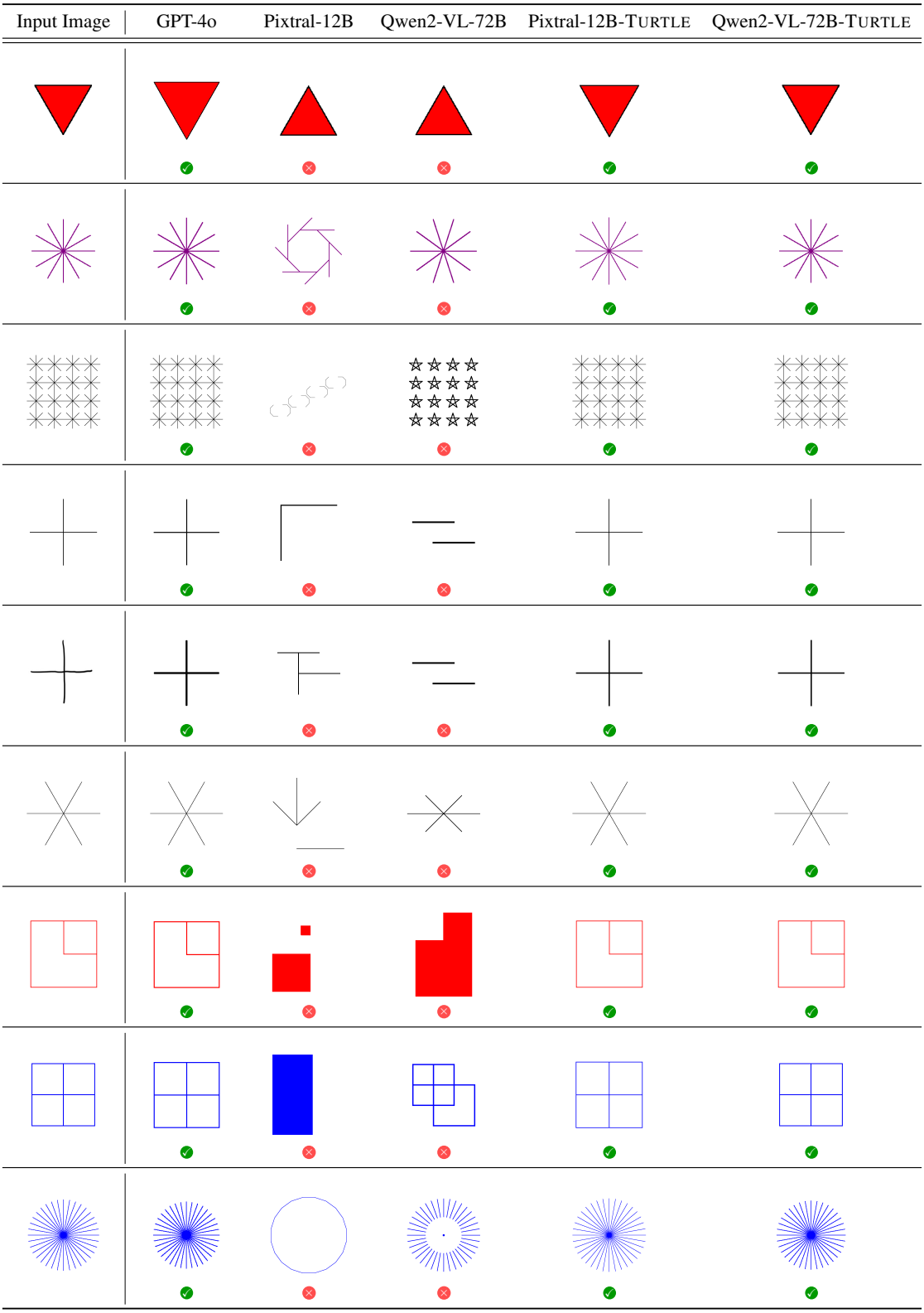


Table 12: Example tasks that are successfully solved by GPT-4o and our fine-tuned models (i.e., Pixtral-12B-TURTLE, Qwen2-VL-72B-TURTLE), but not solved by the base models (Pixtral-12B, Qwen2-VL-72B) in the TURTLEAI-DS dataset. A total of 17 tasks (2.06%) match this criteria. Each row shows a ground truth image (leftmost) followed by the corresponding images generated by executing the each model's generated Python code. Success (✓) and failure (✗) are determined by our evaluation framework using symbolic comparison.

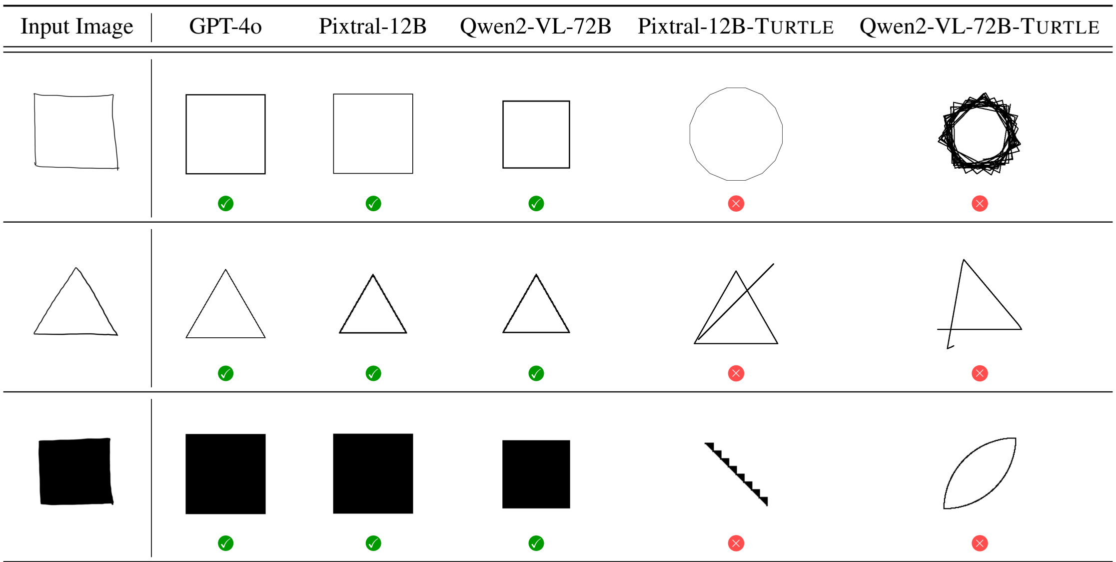


Table 13: Tasks that are successfully solved by base models (i.e., GPT-4o, Pixtral-12B, Qwen2-VL-72B), but not solved by our fine-tuned models (i.e., Pixtral-12B-TURTLE, Qwen2-VL-72B-TURTLE) in the TURTLEAI-DS dataset. A total of 3 tasks (0.36%) match this criteria. Each row shows a ground truth image (leftmost) followed by the corresponding images generated by executing the each model's generated Python code. Success (✓) and failure (⊗) are determined by our evaluation framework using symbolic comparison.

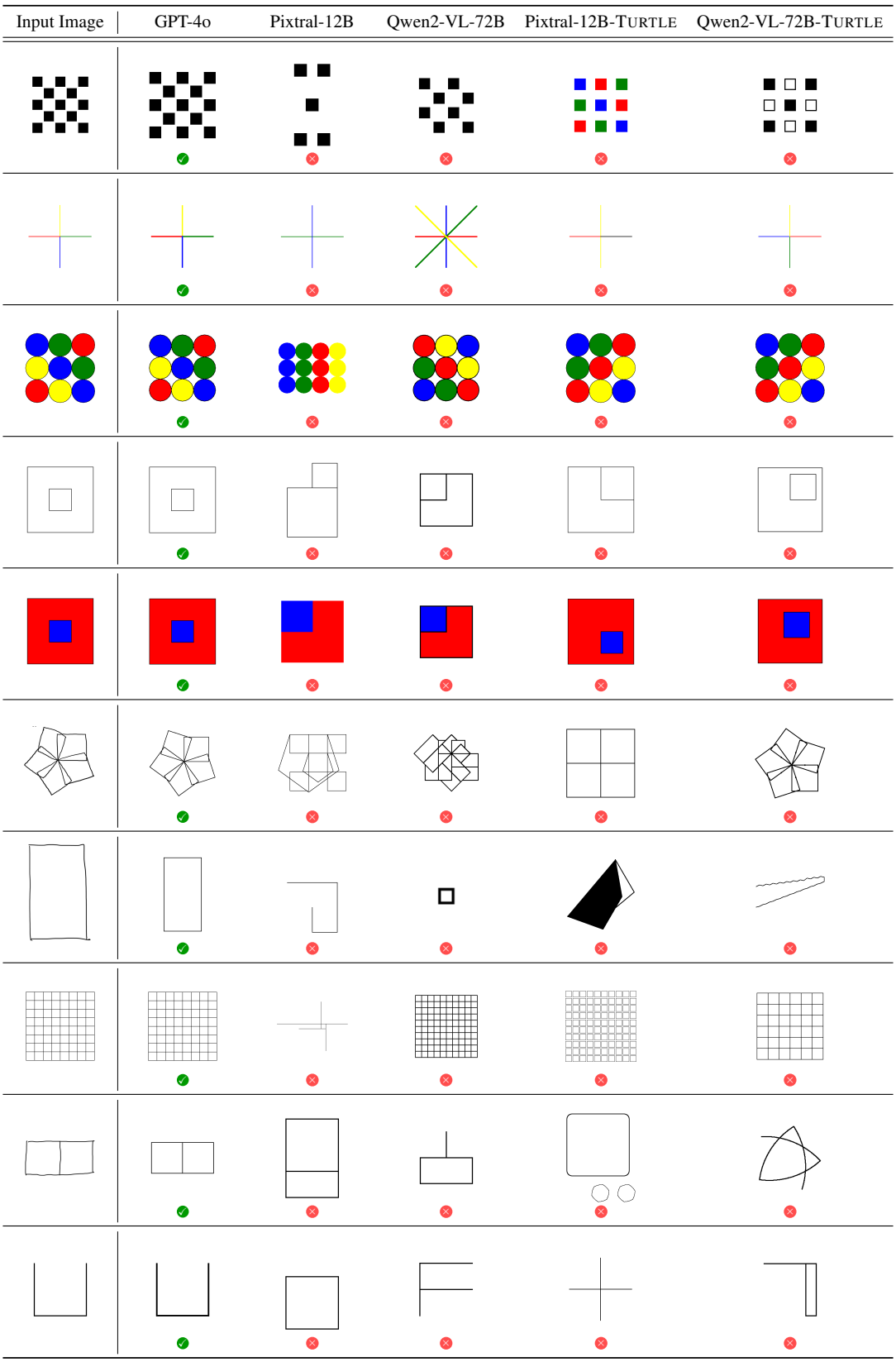


Table 14: Example tasks that are only successfully solved by GPT-4o and not by Pixtral-12B, Qwen2-VL-72B, Pixtral-12B-TURTLE, or Qwen2-VL-72B-TURTLE models in the TURTLEAI-DS dataset. A total of 23 tasks (2.79%) match this criterion. Each row shows a ground truth image (leftmost) followed by the corresponding images generated by executing the each model's generated Python code. Success (✓) and failure (✗) are determined by our evaluation framework using symbolic comparison.

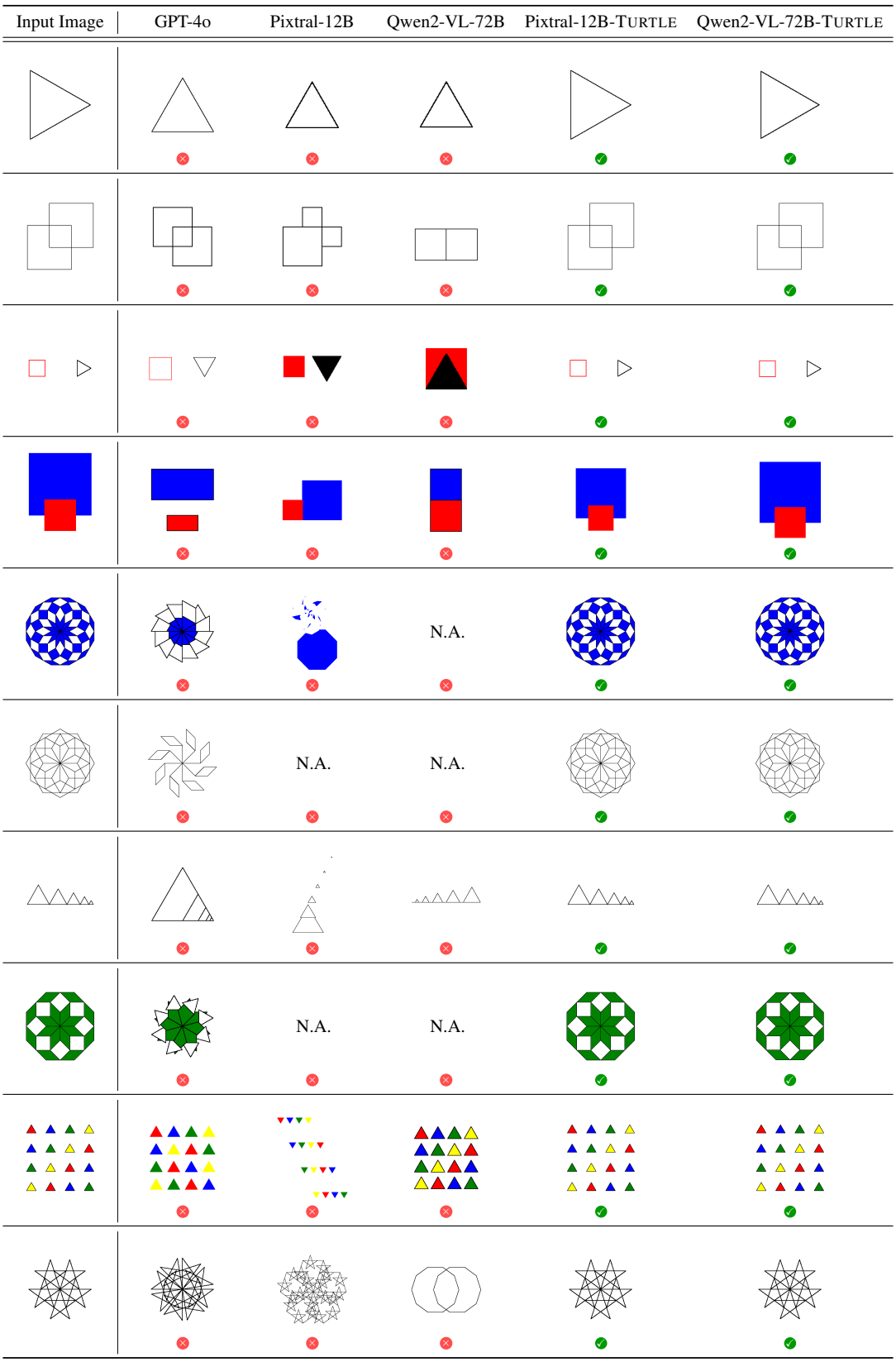


Table 15: Example tasks that are only successfully solved by our fine-tuned models (i.e., Pixtral-12B-TURTLE, Qwen2-VL-72B-TURTLE) and not by Pixtral-12B, Qwen2-VL-72B, or GPT-4o models in the TURTLEAI-DS dataset. A total of 54 tasks (6.56%) match this criterion. Each row shows a ground truth image (leftmost) followed by the corresponding images generated by executing the each model's generated Python code. Success (✓) and failure (✗) are determined by our evaluation framework using symbolic comparison.

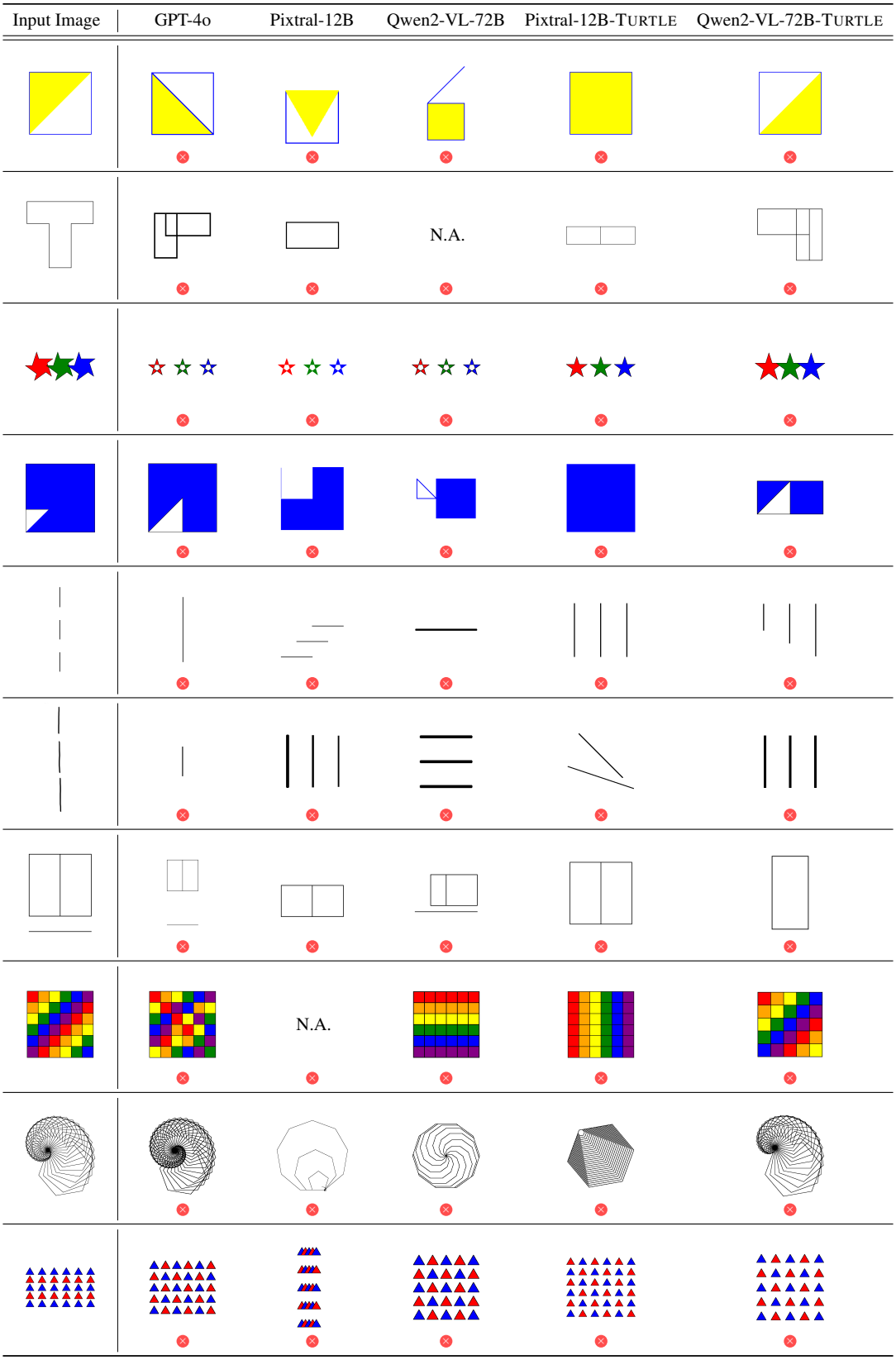
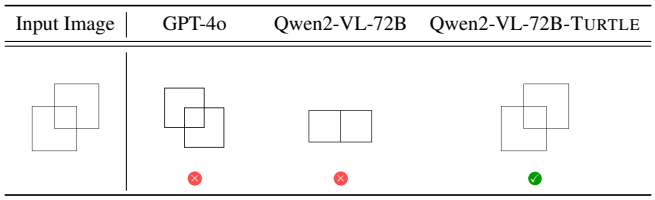


Table 16: Tasks that are not solved by any of the models (i.e., GPT-4o, Pixtral-12B, Qwen2-VL-72B, Pixtral-12B-TURTLE, Qwen2-VL-72B-TURTLE) in the TURTLEAI-DS dataset. A total of 591 tasks (71.81%) match this criteria. Each row shows a ground truth image (leftmost) followed by the corresponding images generated by executing the each model's generated Python code. Success (✓) and failure (✗) are determined by our evaluation framework using symbolic comparison.



(a) Input images and the generated images by different models.

````
Here is the Python code to replicate the given
geometric pattern using the Turtle Graphics
library. The pattern consists of two
overlapping squares.

```python
def draw(t):
    """Draw two overlapping squares."""
    def draw_square(t, size):
        """Helper function to draw a square of
        a given size."""
        for _ in range(4):
            t.forward(size)
            t.right(90)

    # Set up the turtle
    t.penup()
    t.goto(-50, 50) # Move to the starting
    position for the first square
    t.pendown()

    # Draw the first square
    draw_square(t, 100)

    # Move to the starting position for the
    second square
    t.penup()
    t.goto(0, 0)
    t.pendown()
    # Draw the second square
    draw_square(t, 100)
```

### Explanation:
1. **`draw_square` Function**:
    - A helper function to draw a square of a
    given size.
    - The turtle moves forward by the
    specified size and turns 90 degrees to the
    right four times to complete the square.

2. **Positioning**:
    - The turtle starts at `(-50, 50)` to draw
    the first square.
    - After completing the first square, the
    turtle moves to `(0, 0)` to draw the
    second square, ensuring the overlap.

3. **Overlapping**:
    - The two squares overlap at their
    bottom-left and top-right corners, as
    shown in the image.

### Usage:
To execute the code, use the following:
```python
import turtle
# Create a turtle object
t = turtle.Turtle()
# Call the draw function
draw(t)
# Keep the window open
turtle.done()
```
````

(b) GPT-4o

````
```python
import turtle

def draw(t):
    """Draw the specified geometric pattern."""
    # Set the turtle's speed
    t.speed(0)

    # Define the side length of the squares
    side_length = 100

    # Draw the first square
    t.penup()
    t.goto(-side_length / 2, -side_length / 2)
    t.pendown()
    for _ in range(4):
        t.forward(side_length)
        t.right(90)

    # Draw the second square
    t.penup()
    t.goto(side_length / 2, -side_length / 2)
    t.pendown()
    for _ in range(4):
        t.forward(side_length)
        t.right(90)

# Create a turtle object
t = turtle.Turtle()
draw(t)

# Keep the window open
turtle.done()
```

This code defines a function `draw(t)` that uses a
turtle object `t` to draw the specified geometric
pattern. The pattern consists of two overlapping
squares, each with a side length of 100 units. The
turtle starts at the center of the screen and moves
to the starting position of each square before
drawing it. The `turtle.done()` function is used to
keep the window open after the drawing is complete.
````

(c) Qwen2-VL-72B

````
## Image Description

The image displays two overlapping squares.
The squares are of equal size and are
positioned such that one square is shifted
diagonally relative to the other. The
overlapping area forms a smaller square in the
center, while the non-overlapping parts of
each square extend outward, creating a
visually symmetrical pattern. The lines are
black, and the background is white.

## Analysis & Solution Code

To create the pattern shown in the image using
Turtle Graphics, we need to draw two squares
of the same size. The first square is drawn
starting from the initial position. After
completing the first square, the turtle is
moved diagonally to a new position to draw the
second square. This diagonal movement ensures
that the two squares overlap, creating the
desired pattern.

Here is the code with comprehensive docstrings
and comments to create the pattern:

```python
def draw(t):
    """
    Draws two overlapping squares of the same
    size (200 units) with a diagonal offset.

    Args:
        t: Turtle graphics object
    """
    def draw_square(size):
        """
        Draws a square with the given side
        length.

        Args:
            size: The length of each side of
            the square.
        """
        for _ in range(4):
            t.forward(size) # Move the turtle
            forward by the side length
            t.left(90) # Turn the turtle left
            by 90 degrees to form a corner

    # Draw the first square
    draw_square(200)
    # Move the turtle diagonally to the
    starting position of the second square
    t.penup() # Lift the pen to move without
    drawing
    t.goto(100, 100) # Move to the new
    starting position
    t.pendown() # Put the pen down to start
    drawing again
    # Draw the second square
    draw_square(200)
```
````

(d) Qwen2-VL-72B-TURTLE

Figure 13: Example outputs generated by GPT-4o, Qwen2-VL-72B, and Qwen2-VL-72B-TURTLE. (a) shows the input image and the generated images by executing the code generated by models. (b-d) show the outputs generated by different models.

Input Image	GPT-4o	Qwen2-VL-72B	Qwen2-VL-72B-TURTLE
		N.A.	

(a) Input images and the generated images by different models.

````
To replicate the geometric pattern shown in the image using Python's Turtle Graphics
library, we need to break the pattern into smaller components. The pattern consists
of a circular arrangement of blue and white diamond shapes, forming a star-like
structure.

Here is the Python code to recreate the pattern:

```python
def draw(t):
    import turtle
    import math

    # Helper function to draw a diamond shape
    def draw_diamond(t, size, color):
        t.fillcolor(color)
        t.begin_fill()
        for _ in range(2):
            t.forward(size)
            t.left(60)
            t.forward(size)
            t.left(120)
        t.end_fill()

    # Helper function to draw a single segment of the pattern
    def draw_segment(t, size):
        # Draw the blue diamond
        draw_diamond(t, size, "blue")
        # Position for the white diamond
        t.forward(size)
        t.left(60)
        # Draw the white diamond
        draw_diamond(t, size, "white")
        # Return to the center
        t.right(60)
        t.backward(size)

    # Main function to draw the full pattern
    def draw_pattern(t, size, segments):
        for _ in range(segments):
            draw_segment(t, size)
            t.right(360 / segments)

    # Set up the turtle
    t.speed(0)
    t.penup()
    t.goto(0, 0)
    t.pendown()

    # Draw the pattern
    draw_pattern(t, 50, 12) # Adjust size and number of segments as needed

    # Hide the turtle
    t.hideturtle()
```

### Explanation:

<some text is omitted here to save space>

### Usage:

<some text is omitted here to save space>

This code will produce the desired geometric pattern. Adjust the `size` and
`segments` parameters in the `draw_pattern` function to modify the pattern's
appearance.
````

(b) GPT-4o

````
## Image Description

The image displays a symmetrical, flower-like pattern
composed of geometric shapes. The primary shape is a
dodecagon (a 12-sided polygon) repeated in a circular
arrangement. Each dodecagon is filled with a blue color, and
the spaces between the dodecagons are white, creating a
checkerboard-like effect. The pattern exhibits rotational
symmetry, with each dodecagon rotated slightly relative to
its neighbors, forming a visually appealing star-like
structure at the center. The overall design is intricate yet
balanced, with the blue and white sections contrasting
sharply against each other.

## Analysis & Solution Code

To create the pattern shown in the image using Turtle
Graphics, we need to follow these steps:

1. **Set Up the Turtle**: Initialize the turtle graphics and
set the fill color to blue.
2. **Define the Dodecagon**: Create a function to draw a
single dodecagon. A dodecagon has 12 sides, and each internal
angle is 150 degrees (calculated as \( \frac{180 \times
(12-2)}{12} \)).
3. **Draw the Pattern**: Use a loop to draw 12 dodecagons,
each rotated by 30 degrees (360 degrees / 12) relative to the
previous one. This rotation ensures the symmetrical
arrangement of the dodecagons.
4. **Fill the Shape**: Use the `begin_fill` and `end_fill`
methods to fill the dodecagons with the specified color.

Here is the code with comprehensive docstrings and comments
to create the pattern:

```python
def draw(t):
    """
    Draws a symmetrical, flower-like pattern composed of 12
    blue-filled dodecagons.
    Each dodecagon is rotated by 30 degrees relative to the
    previous one, creating a
    visually appealing star-like structure at the center.

    Args:
        t: Turtle graphics object
    """
    t.fillcolor('blue') # Set the fill color to blue

    def draw_dodecagon():
        """
        Draws a single dodecagon with 12 sides.
        Each side is 50 units long, and the internal angle is
        150 degrees.
        """
        for _ in range(12):
            t.forward(50) # Move the turtle forward by 50 units
            t.left(30) # Turn the turtle left by 30 degrees

    t.begin_fill() # Start filling the shape
    for _ in range(12):
        draw_dodecagon() # Draw a dodecagon
        t.left(30) # Rotate the turtle by 30 degrees for the
        next dodecagon
    t.end_fill() # End filling the shape
```
````

(c) Qwen2-VL-72B-TURTLE

Figure 14: Example outputs generated by GPT-4o and Qwen2-VL-72B-TURTLE. (a) shows the input image and the generated images by executing the code generated by models. (b) and (c) show the outputs generated by different models. Qwen2-VL-72B's output is not shown since it generates repetitive text.

Prompt for Generating Code from Image

<image> You are a Turtle Graphics programmer tasked with creating Python code to replicate a specified geometric pattern using the Turtle Graphics library.

Task Overview:

Analyze the provided image of a geometric pattern. Carefully break down the pattern into individual shapes, colors, angles, and layout components. Using this information, write Python code within a function called `draw(t)`, where `t` is a Turtle object. Assume:
- The turtle starts at the center of the screen at coordinates `(0, 0)`.
- The turtle initially faces east (to the right).

The goal is for your `draw(t)` function to accurately recreate the pattern shown in the image, including its positioning, angles, colors, and details.

Requirements:

1. **Code Structure**:
    - Place all code inside the `draw(t)` function.
    - The function takes a turtle object `t` as input.
    - Format your code using triple backticks with the 'python' language specifier, i.e., ```python```.

    Example format:
    ```python
    def draw(t):
        # Your code here
    ```
2. **Color Accuracy**:
    - Match colors in the image exactly, both for fills and outlines.
3. **Pattern Precision**:
    - Reproduce the pattern as accurately as possible, maintaining symmetry, shapes, and angles.
4. **Self-Contained**:
    - Do not include code outside the `draw(t)` function.
    - All necessary imports, variables, and helper functions should be inside `draw(t)`.

Execution Context:

Your `draw(t)` function will be called in the following manner:

```python
import turtle

def draw(t):
    # Describe the drawing steps here
    pass

t = turtle.Turtle()
draw(t)
```

Example Outputs:

- **Example 1 - Drawing a Rectangle**:
```python
def draw(t):
    """Draw a rectangle."""
    def draw_rectangle(t):
        # Draw a rectangle with side length 10
        for _ in range(4):
            t.forward(10)
            t.right(90)
    draw_rectangle(t)
```

- **Example 2 - Drawing a circle**:
```python
def draw(t):
    """Draw a circle."""
    import math

    def draw_circle(t, radius):
        circumference = 2 * math.pi * radius
        step_length = circumference / 360
        step_angle = 1

        for _ in range(360):
            t.forward(step_length)
            t.left(step_angle)

    draw_circle(t, 50)
```

Note: The examples are simplified. Your final code may require nested loops or additional logic to fully replicate complex patterns.

Now, write the code for the `draw(t)` function to recreate the pattern shown in the image as closely as possible in terms of shape, color, and structure.

Figure 15: Prompt template for code synthesis from visual input.

Prompt for the Reference-guided Code Mutation Stage of TurtleAI-Datagen

```
You are a turtle graphics programmer tasked with analyzing and applying code adaptations in Python using the Turtle
Graphics library. You are given two reference codes that perform a certain drawing task. Your mission is to:

1. Identify how the adaptation is done from the first code to the second code.
2. Summarize the adaptation in a high-level way, so it can be applied to any other code.
3. Apply the core idea of the adaptation to a new piece of code provided.

### Key Requirements for Code Adaptation:

1. Syntactic Correctness:
   - The adapted code must be syntactically correct and free of errors.

2. Structural and Logical Consistency:
   - Maintain the structural integrity and logical flow of the original code.
   - Ensure that no unintended behavior is introduced by the adaptation.

3. Geometric Structure & Symmetry (if applicable):
   - Ensure that all drawings consist of clear geometric shapes with symmetry and geometric accuracy.

4. Visual Clarity & Simplicity:
   - The output should be visually clear and simple.
   - Avoid overly complex designs that may confuse or clutter the output.

5. Function and Code Requirements:
   - Define the function `draw(t)` that contains all the drawing code.
   - Use appropriate Turtle Graphics library commands within the `draw(t)` function.
   - Only provide the `draw(t)` function. Do not include import statements or other code outside of the `draw()` function.

6. Different Output:
- The adapted code must generate a different drawing compared to the original new code.
- The drawing must be a different shape or have a distinct pattern to clearly show the adaptation's impact.

### Your Task:

Reference Code 1:
```python
{reference_code_1}
```

Reference Code 2:
```python
{reference_code_2}
```

New Code to Adapt:
```python
{code_to_adapt}
```

Now, follow these steps:

1. Analyze the Adaptation:
   - Examine how Reference Code 1 is adapted into Reference Code 2.
   - Summarize the adaptation in a high-level way that can be applied to other codes.

2. Apply the Adaptation:
   - Apply the core idea of the adaptation to the New Code to Adapt.
   - Provide the Adapted Code that reflects this adaptation.
   - Ensure the adapted code is syntactically correct and that the resulting drawing after execution meets all the
   specified requirements (geometric structure, symmetry, visual clarity, simplicity, etc.).

Adapted Code:

Provide your adapted code here. Ensure it meets all the specified requirements, especially that it must generate a
different drawing compared to the original new code. Use the following Python code block format:

```python
def draw(t):
   # Your adapted code here
```
```

Figure 16: Prompt template for reference-guided code generation.

Prompt for the Elite Selection Stage of TurtleAI-Datagen

<image> You are an evaluator responsible for automatically assessing the quality of a turtle graphics programming task using the following rubrics. Each rubric evaluates different aspects of the task, including its clarity, difficulty, alignment with programming concepts, and creativity. Please assign a score from 0 to 10 for each rubric and provide an explanation for your scoring. Each rubric has equal weight, and the rubrics are as follows:

Rubrics Breakdown:

1. **Geometric Structure & Symmetry**
- Score Breakdown (0–10):
  - 9–10: Perfect geometric accuracy and symmetry – all elements are precisely aligned and balanced.
  - 6–8: Mostly symmetric with minor imperfections – slight deviations that do not detract from overall symmetry.
  - 3–5: Some geometric or symmetry issues – noticeable asymmetries or inaccuracies in shape.
  - 0–2: Significant asymmetry and inaccuracies – major deviations from expected geometric forms.
2. **Visual Appeal, Clarity & Simplicity**
- Score Breakdown (0–10):
  - 9–10: Clear, simple design with purposeful aesthetics – easily understood and visually harmonious. No unnecessary complexity.
  - 6–8: Generally clear design with good balance, but has minor complexity or visual elements that could be simplified.
  - 3–5: Either overly complex, lacks visual harmony, or has clarity issues – may have unnecessary elements or confusing design choices.
  - 0–2: Significant issues with clarity or complexity – cluttered, difficult to interpret, or contains many unnecessary elements.
3. **Structural Coherence**
- Score Breakdown (0–10):
  - 9–10: Strong structural integrity – design is cohesive, whether through repeated patterns, basic geometric shapes, or a purposeful unique design.
  - 6–8: Good structure with minor imperfections – mostly coherent with slight inconsistencies.
  - 3–5: Basic structure present but with noticeable flaws – some elements may seem out of place or poorly integrated.
  - 0–2: Weak or unclear structure – lacks a clear organizational pattern or design logic.
4. **Alignment & Positioning**
- Score Breakdown (0–10):
  - 9–10: Excellent alignment and positioning – all elements are precisely placed and aligned.
  - 6–8: Good alignment with minor issues – generally well-positioned with slight misalignments.
  - 3–5: Some misalignment – noticeable but not critical positioning errors.
  - 0–2: Noticeable misalignment – significant positioning errors that affect the overall design.
5. **Educational Value & Solvability**
- Score Breakdown (0–10):
  - 9–10: Excellent educational value – pattern complexity is appropriate for learning, clear objectives, and perfectly balanced difficulty that students can reasonably solve.
  - 6–8: Strong educational value – complexity is manageable for students with some guidance, mostly clear and appropriately challenging without being overwhelming.
  - 3–5: Moderate educational value – either too simple to be educational or too complex for students to reasonably solve. Would require significant modifications to be classroom-ready.
  - 0–2: Poor educational value – not suitable for classroom use due to excessive complexity, confusing structure, or contains sensitive/inappropriate imagery (e.g., Swastika, Confederate flag, etc.).
6. **Color Usage & Necessity**
- Score Breakdown (0–10):
  - 9–10: Excellent use of minimal colors – either black & white only, or uses very few colors (<5) with clear purpose that enhances understanding.
  - 6–8: Acceptable color usage – slightly more colors than necessary but not distracting. Could be simplified without losing meaning.
  - 3–5: Problematic color usage – too many colors or colors used without clear purpose. Would be clearer with fewer colors.
  - 0–2: Poor color usage – excessive number of colors, random color choices, or colors that make the pattern harder to understand.

Final Evaluation Instructions:

Once you have evaluated each category and assigned scores, sum up all the individual rubric scores. Since each rubric has equal weight, no additional multiplication is needed. The **final score** is simply the sum of all rubric scores. Summarize the individual scores and explanations, then provide the final score (out of 60).

Expected JSON Output:

Please format the final evaluation as a JSON object using the following short keys:
- **geometry**: Geometric Structure & Symmetry
- **visual**: Visual Appeal & Clarity
- **structure**: Structural Coherence
- **alignment**: Alignment & Positioning
- **education**: Educational Value & Solvability
- **color**: Color Usage & Necessity
- **final_score**: Final score out of 60

Example JSON Output:
```json
{
  "geometry": {"score": "<score>", "explanation": "<explanation>"},
  "visual": {"score": "<score>", "explanation": "<explanation>"},
  "structure": {"score": "<score>", "explanation": "<explanation>"},
  "alignment": {"score": "<score>", "explanation": "<explanation>"},
  "education": {"score": "<score>", "explanation": "<explanation>"},
  "color": {"score": "<score>", "explanation": "<explanation>"},
  "final_score": "<final_score>"
}
```
Now, evaluate the turtle graphics task based on the provided image. This image was created using turtle graphics. Please assess its quality using the rubrics outlined above, and provide the final evaluation in the JSON format shown in the example.

Figure 17: Prompt template for the elite selection stage in TurtleAI-Datagen.

Prompt for the CoT Labeling

<image> Your task is to optimize a provided Python code snippet that uses Turtle Graphics, ensuring it is minimal, cleanly documented, and fully aligned with the generated image output. You will be provided with:

1. A Python code snippet using Turtle Graphics.
2. The actual image output generated by this code.

Your Responsibilities:

1. **Describe the Image**
   - Provide a detailed description of the visual pattern in the image **without referencing the code**, focusing on geometric shapes, symmetry, colors, and overall structure.

2. **Optimize the Code**
   - Identify and remove redundant code segments that do not contribute to the visual output, and simplify the logic to enhance readability, ensuring the **final output remains visually identical**.
   - After optimizing the code, provide a detailed step-by-step explanation of how the code generates the image, linking visual features to the corresponding steps.

3. **Add Documentation and Comments**
   - Add a descriptive docstring for the provided code snippet, explaining its purpose, parameters, and any outputs. Include clear and concise inline comments to make the code understandable.

Formatting Instructions:
   - **Markdown-Only Response**: Format your entire response in markdown, enclosed in a single markdown block.
   - **Output Focus**: Provide only the optimized `draw(t)` function within the markdown block, excluding any setup or unrelated code.

Provided Code Snippet

Below is the code snippet that generates the image you are analyzing:

```python
{code}
```

Please provide your response in the following markdown format:

````
```markdown
## Image Description

The image displays...

[Provide a detailed description of the visual pattern]

## Analysis & Solution Code

To create the pattern shown in the image using Turtle Graphics, we need to...
[Explain how to create this pattern using Turtle Graphics, describing the logical steps needed to reproduce the image]

Here is the code with comprehensive docstrings and comments to create the pattern:
```python
def draw(t):
    """
    [Function description]

    Args:
        t: Turtle graphics object
    """
    # Your simplified code with comments
```
```
````

Important: Write your response as if you are only looking at the image, without referencing any provided code (e.g., do not mention `modified code`, `optimized code`, or `provided code` in your response inside the markdown block).

Figure 18: Prompt template for the CoT labeling for generating the training dataset.