Diffusion On Syntax Trees For Program Synthesis

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large language models generate code one token at a time. Their autoregressive generation process lacks the feedback of observing the program's output. Training LLMs to suggest edits directly can be challenging due to the scarcity of rich edit data. To address these problems, we propose neural diffusion models that operate on syntax trees of any context-free grammar. Similar to image diffusion models, our method also inverts "noise" applied to syntax trees. Rather than generating code sequentially, we iteratively edit it while preserving syntactic validity, which makes it easy to combine this neural model with search. We apply our approach to inverse graphics tasks, where our model learns to convert images into programs that produce those images. Combined with search, our model is able to write graphics programs, see the execution result, and debug them to meet the required specifications. We additionally show how our system can write graphics programs for hand-drawn sketches. Video results can be found at `https://td-anon.github.io`.

1 Introduction

Large language models (LLMs) have made remarkable progress in code generation, but their autoregressive nature presents a fundamental challenge: they generate code token by token, without access to the program's runtime output from the previously generated tokens. This makes it difficult to correct errors, as the model lacks the feedback loop of seeing the program's output and adjusting accordingly. While LLMs can be trained to suggest edits to existing code [6, 42, 17], acquiring sufficient training data for this task is difficult.

In this paper, we introduce a new approach to program synthesis using *neural diffusion* models that operate directly on syntax trees. Diffusion models have previously been used to great success in image generation [14, 22, 31]. By leveraging diffusion, we let the model learn to iteratively refine programs while ensuring syntactic validity. Crucially, our approach allows the model to observe the program's output at each step, effectively enabling a debugging process.

In the spirit of systems like AlphaZero [29], the iterative nature of diffusion naturally lends itself to search-based program synthesis. By training a value model alongside our diffusion model, we can guide the denoising process toward programs that are likely to achieve the desired output. This allows us to efficiently explore the program space, making more informed decisions at each step of the generation process.

We implement our approach for inverse graphics tasks, where we posit domain-specific languages for drawing images. Inverse graphics tasks are naturally suitable for our approach since small changes in the code produce semantically meaningful changes in the rendered image. For example, a misplaced shape on the image can be easily seen and fixed in program space.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

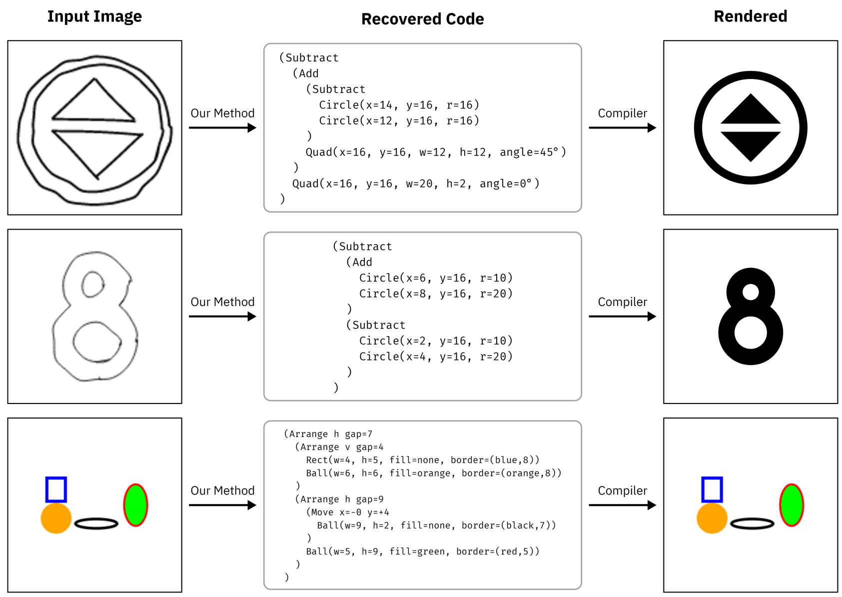


Figure 1: Examples of programs recovered by our system. The top row shows a hand-drawn sketch of an icon (left), the recovered program (middle), and the compilation of the recovered program (right). The top two rows are for the constructive solid geometry language (`CSG2D-Sketch`). The last row is an example output from our `TinySVG` environment that learns to invert hierarchical programs of shapes and colors. Video examples can be found at https://td-anon.github.io.

Our main contributions for this work are (a) a novel approach to program synthesis using diffusion on syntax trees and (b) an implementation of our approach for inverse graphics tasks that significantly outperforms previous methods.

2 Background & Related Work

Neural program synthesis Neural program synthesis is a prominent area of research, in which neural networks generate programs from input-output examples. Early work, such as Parisotto et al. [23], demonstrated the feasibility of this approach. While modern language models can be directly applied to program synthesis, combining neural networks with search strategies often yields better results and guarantees. In this paradigm, the neural network guides the search process by providing proposal distributions or scoring candidate programs. Examples of such hybrid methods include Balog et al. [2], Ellis et al. [12], and Devlin et al. [9]. A key difference from our work is that these methods construct programs incrementally, exploring a vast space of partial programs. Our approach, in contrast, focuses on *editing* programs, allowing us to both grow programs from scratch and make corrections based on the program execution.

Neural diffusion Neural diffusion models, a class of generative models, have demonstrated impressive results for modeling high-dimensional data, such as images [14, 22, 31]. A neural diffusion model takes samples from the data distribution (e.g. real-world images), incrementally corrupts the data by adding noise, and trains a neural network to incrementally remove the noise. To generate new samples, we can start with random noise and iteratively apply the neural network to denoise the input.

Diffusion for discrete data Recent work extends diffusion to discrete and structured data like graphs [35], with applications in areas such as molecule design [15, 27, 8]. Notably, Lou et al. [20] proposed a discrete diffusion model using a novel score-matching objective for language modeling. Another promising line of work for generative modeling on structured data is generative flow networks (GFlowNets) [3], where neural models construct structured data one atom at a time.

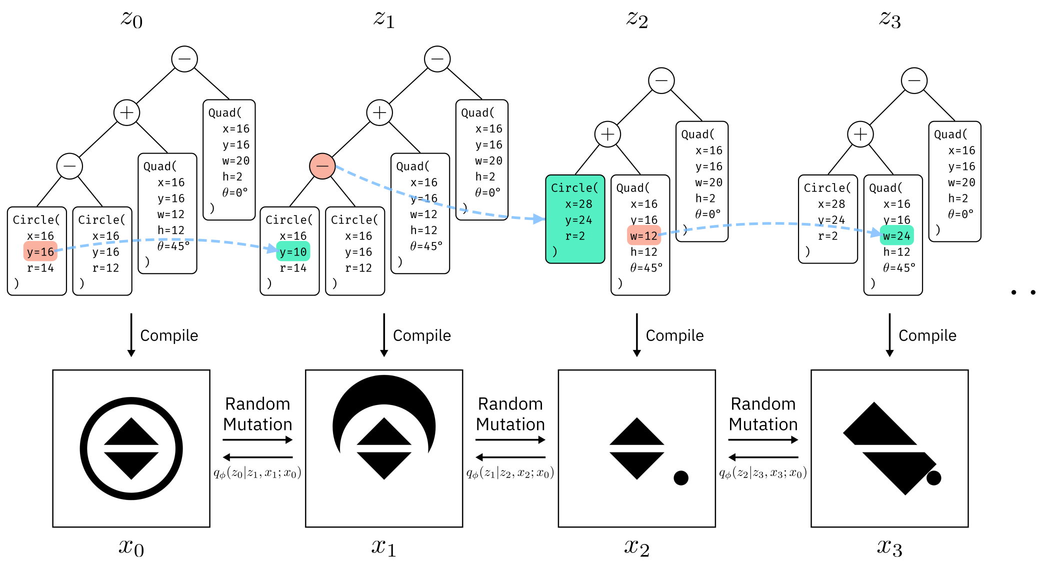


Figure 2: An overview of our method. Analogously to adding noise in image diffusion, we randomly make small mutations to the syntax trees of programs. We then train a conditional neural model to invert these small mutations. In the above example, we operate in a domain-specific language (DSL) for creating 2D graphics using a constructive solid geometry language. The leftmost panel (z_0) shows the target image (bottom) alongside its program as a syntax tree (top). The y value of the circle gets mutated from 16 to 10 in the second panel, making the black circle "jump" a little higher. Between z_1 and z_2, we see that we can mutate the `Subtract` ($-$) node to a `Circle` node, effectively deleting it.

Diffusion for code generation Singh et al. [30] use a diffusion model for code generation. However, their approach is to first embed text into a continuous latent space, train a *continuous* diffusion model on that space, and then unembed at the end. This means that intermediate stages of the latent representation are not trained to correspond to actual code. The embedding tokens latch to the nearest embeddings during the last few steps.

Direct code editing using neural models has also been explored. Chakraborty et al. [6] use a graph neural network for code editing, trained on a dataset of real-world code patches. Similarly, Zhang et al. [42] train a language model to edit code by modifying or inserting `[MASK]` tokens or deleting existing tokens. They further fine-tune their model on real-world comments and patches. Unlike these methods, our approach avoids the need for extensive code edit datasets and inherently guarantees syntactic validity through our pretraining task.

Program synthesis for inverse graphics We are inspired by previous work by Sharma et al. [28], Ellis et al. [10, 11], which also uses the `CSG2D` language. Sharma et al. [28] propose a convolutional encoder and a recurrent model to go from images to programs. Ellis et al. [11] propose a method to provide a neural model with the intermediate program execution output in a read–eval–print loop (REPL). Unlike our method, the ability to execute partial graphics programs is a key requirement for their work. Our system operates on complete programs and does not require a custom partial compiler. As mentioned in their work, their policies are also brittle. Once the policy proposes an object, it cannot undo that proposal. Hence, these systems require a large number of particles in a Sequential Monte-Carlo (SMC) sampler to make the system less brittle to mistakes.

3 Method

The main idea behind our method is to develop a form of denoising diffusion models analogous to image diffusion models for syntax trees.

Consider the example task from Ellis et al. [11] of generating a constructive solid geometry (`CSG2D`) program from an image. In `CSG2D`, we can combine simple primitives like circles and quadrilaterals using boolean operations like addition and subtraction to create more complex shapes, with the context-free grammar (CFG),

$$\texttt{S} \rightarrow \texttt{S} + \texttt{S} \mid \texttt{S} - \texttt{S} \mid \texttt{Circle}^{r}_{x,y} \mid \texttt{Quad}^{w,h}_{x,y,\theta}.$$

In Figure 2, z_0 is our *target program*, and x_0 is the rendered version of z_0. Our task is to invert x_0 to recover z_0. Our noising process randomly mutates `y=16` to `y=10`. It then mutates the whole $\ominus$ sub-tree with two shapes with a new sub-tree with just one shape. Conditioned on the image x_0, and starting at z_3, x_3, we would like to train a neural network to reverse this noising process to get to z_0.

In the following sections, we will first describe how "noise" is added to syntax trees. Then, we will detail how we train a neural network to reverse this noise. Finally, we will describe how we use this neural network for search.

3.1 Sampling Small Mutations

Let z_t be a program at time t. Let $p_{\mathcal{N}}(z_{t+1}|z_t)$ be the distribution over randomly mutating program z_t to get z_{t+1}. We want $p_{\mathcal{N}}$ mutations to be: (1) small and (2) produce syntactically valid z_{t+1}'s.

To this end, we turn to the rich computer security literature on grammar-based fuzzing [41, 13, 32, 36]. To ensure the mutations are small, we first define a function $\sigma(z)$ that gives us the "size" of program z. For all our experiments, we define a set of terminals in our CFG to be *primitives*. As an example, the primitives in our `CSG2D` language are $\{\texttt{Quad}, \texttt{Circle}\}$. In that language, we use $\sigma(z) = \sigma_{\text{primitive}}(z)$, which counts the number of primitives. Other generic options for $\sigma(z)$ could be the depth, number of nodes, etc.

We then follow Luke [21] and Zeller et al. [41] to randomly sample programs from our CFG under exact constraints, $\sigma_{\min} < \sigma(z) \leq \sigma_{\max}$. We call this function $\texttt{ConstrainedSample}(\sigma_{\min}, \sigma_{\max})$. Setting a small value for $\sigma_{\max}$ allows us to sample *small* programs randomly. We set $\sigma_{\max} = \sigma_{\text{small}}$ when generating small mutations.

To mutate a given program z, we first generate a set of candidate nodes in its tree under some σ_{small},

$$\mathcal{C} = \{n \in \texttt{SyntaxTree}(z) \mid \sigma(n) \leq \sigma_{\text{small}}\}.$$

Then, we uniformly sample a mutation node from this set,

$$m \sim \text{Uniform}[\mathcal{C}].$$

Since we have access to the full syntax tree and the CFG, we know which production rule produced m, and can thus ensure syntactically valid mutations. For example, if m were a number, we know to replace it with a number. If m were a general subexpression, we know we can replace it with any general subexpression. Therefore, we sample m', which is m's replacement as,

$$m' \sim \texttt{ConstrainedSample}(\texttt{ProductionRule}(m), \sigma_{\text{small}}).$$

3.2 Policy

3.2.1 Forward Process

We cast the program synthesis problem as an inference problem. Let $p(x|z)$ be our observation model, where x can be any kind of observation. For example, we will later use images x produced by our program, but x could also be an execution trace, a version of the program compiled to bytecode, or simply a syntactic property. Our task is to invert this observation model, i.e. produce a program z given some observation x.

We first take some program z_0, either from a dataset, $\mathcal{D} = \{z^0, z^1, \ldots\}$, or in our case, a randomly sampled program from our CFG. We sample z_0's such that $\sigma(z_0) \leq \sigma_{\max}$. We then add noise to z_0 for $s \sim \text{Uniform}[1, s_{\max}]$, steps, where $s_{\max}$ is a hyper-parameter, using,

$$z_{t+1} \sim p_{\mathcal{N}}(z_{t+1}|z_t).$$

We then train a conditional neural network that models the distribution,

$$q_\phi(z_{t-1}|z_t, x_t; x_0),$$

where ϕ are the parameters of the neural network, z_t is the current program, x_t is the current output of the program, and x_0 is the target output we are solving for.

3.2.2 Reverse Mutation Paths

Since we have access to the ground-truth mutations, we can generate targets to train a neural network by simply reversing the sampled trajectory through the forward process Markov-Chain, $z_0 \to z_1 \to \ldots$. At first glance, this may seem a reasonable choice. However, training to simply invert the last mutation can potentially create a much noisier signal for the neural network.

Consider the case where, within a much larger syntax tree, a color was mutated as,

$$\texttt{Red} \to \texttt{Blue} \to \texttt{Green}.$$

The color in our target image, x_0, is `Red`, while the color in our mutated image, x_2, is `Green`. If we naively teach the model to invert the above Markov chain, we are training the network to turn the `Green` to a `Blue`, even though we could have directly trained the network to go from `Green` to a `Red`.

Therefore, to create a better training signal, we compute an *edit path* between the target tree and the mutated tree. We use a tree edit path algorithm loosely based on the tree edit distance introduced by Pawlik and Augsten [25, 24]. The general tree edit distance problem allows for the insertion, deletion, and replacement of any node. Unlike them, our trees can only be edited under an action space that only permits *small* mutations. For two trees, z_A and z_B, we linearly compare the syntax structure. For changes that are already $\leq \sigma_{\text{small}}$, we add that to our mutation list. For changes that are $> \sigma_{\text{small}}$, we find the first mutation that reduces the distance between the two trees. Therefore, for any two programs, z_A and z_B, we can compute the first step of the mutation path in $O(|z_A| + |z_B|)$ time.

3.3 Value Network & Search

We additionally train a value network, $v_\phi(x_A, x_B)$, which takes as input two rendered images, x_A and x_B, and predicts the edit distance between the underlying programs that generated those images. Since we have already computed edit paths between trees during training, we have direct access to the ground-truth program edit distance for any pair of rendered images, allowing us to train this value network in a supervised manner.

Using our policy, $q_\phi(z_{t-1}|z_t, x_t; x_0)$, and our value, $v_\phi(x_{t_A}, x_{t_B})$, we can perform beam-search for a given target image, x_0, and a randomly initialized program z_t. At each iteration, we maintain a collection of nodes in our search tree with the most promising values and only expand those nodes.

3.4 Architecture

Figure 3 shows an overview of our neural architecture. We use a vision-language model described by Tsimpoukelli et al. [33] as our denoising model, $q_\phi(z_{t-1}|z_t, x_t; x_0)$. We use an off-the-shelf implementation [38] of NF-ResNet-26 as our image encoder, which is a normalizer-free convolutional architecture proposed by Brock et al. [4] to avoid test time instabilities with Batch-Norm [40]. We implement a custom tokenizer, using the terminals of our CFG as tokens. The rest of the edit model is a small decoder-only transformer [34, 26].

We add two additional types of tokens: an `<EDIT>` token, which serves as a start-of-sentence token for the model; and `<POS x>` tokens, which allow the model to reference positions within its context. Given a current image, a target image, and a current tokenized program, we train this transformer model to predict the edit position and the replacement text autoregressively. While making predictions, the decoding is constrained under the grammar. We mask out the prediction logits to only include edit positions that represent nodes in the syntax tree, and only produce replacements that are syntactically valid for the selected edit position.

We set $\sigma_{\text{small}} = 2$, which means the network is only allowed to produce edits with fewer than two primitives. For training data, we sample an infinite stream of random expressions from the CFG. We choose a random number of noise steps, $s \in [1, 5]$, to produce a mutated expression. For some percentage of the examples, ρ, we instead sample a completely random new expression as our mutated expression. We trained for 3 days for the environments we tested on a single Nvidia A6000 GPU.

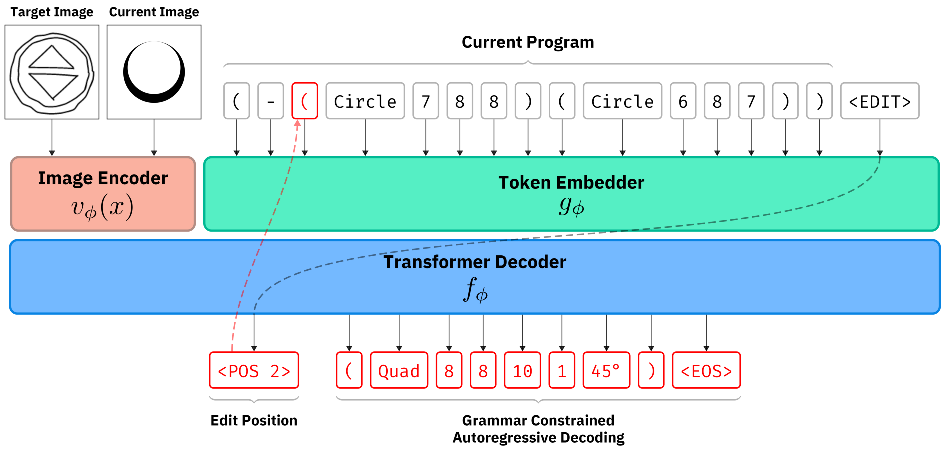


Figure 3: We train $q_\phi(z_{t-1}|z_t, x_t; x_0)$ as a decoder only vision-language transformer following Tsimpoukelli et al. [33]. We use an NF-ResNet as the image encoder, which is a normalizer-free convolutional architecture proposed by Brock et al. [4]. The image encoder encodes the current image, x_t, and the target images, x_0. The current program is tokenized according to the vocabulary in our context-free grammar. The decoder first predicts an *edit* location in the current program, and then tokens that replace what the edit location should be replaced by. We constrain the autoregressive decoding by our context-free grammar by masking only the valid token logits.

4 Experiments

4.1 Environments

We conduct experiments on four domain-specific graphics languages, with complete grammar specifications provided in Appendix B.

`CSG2D` A 2D constructive solid geometry language where primitive shapes are added and subtracted to create more complex forms, as explored in our baseline methods [11, 28]. We also create `CSG2D-Sketch`, which has an added observation model that simulates hand-drawn sketches using the algorithm from Wood et al. [39].

`TinySVG` A language featuring primitive shapes with color, along with `Arrange` commands for horizontal and vertical alignment, and `Move` commands for shape offsetting. Figure 1 portrays an example program. Unlike the compositional nature of `CSG2D`, `TinySVG` is hierarchical: sub-expressions can be combined into compound objects for high-level manipulation. We also create, `Rainbow`, a simplified version of `TinySVG` without `Move` commands for ablation studies due to its reduced computational demands.

We implemented these languages using the Lark [19] and Iceberg [16] Python libraries, with our tree-diffusion implementation designed to be generic and adaptable to any context-free grammar and observation model.

4.2 Baselines

We use two prior works, Ellis et al. [11] and CSGNet [28] as baseline methods.

CSGNet Sharma et al. [28] employed a convolutional and recurrent neural network to generate program statements from an input image. For a fair comparison, we re-implemented CSGNet using the same vision-language transformer architecture as our method, representing the modern autoregressive approach to code generation. We use rejection sampling, repeatedly generating programs until a match is found.

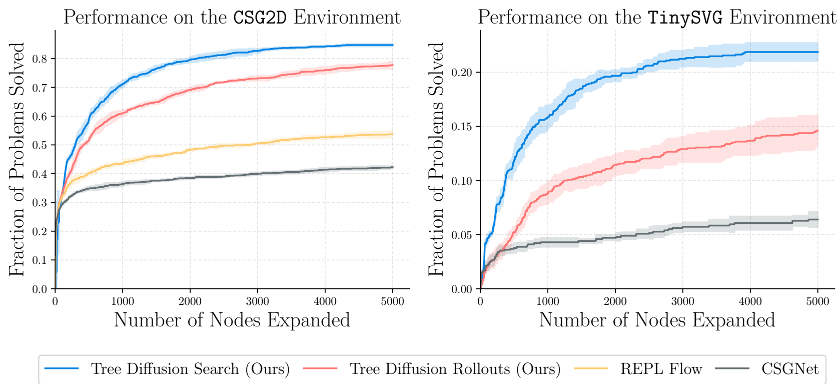


Figure 4: Performance of our approach in comparison to baseline methods in `CSG2D` and `TinySVG` languages. We give the methods $n = 256$ images from the test set and measure the number of nodes expanded to find a solution. The auto-regressive baseline was queried with rejection sampling. Our policy outperforms previous methods, and our policy combined with search helps boost performance further. Error bars show standard deviation across 5 random seeds.

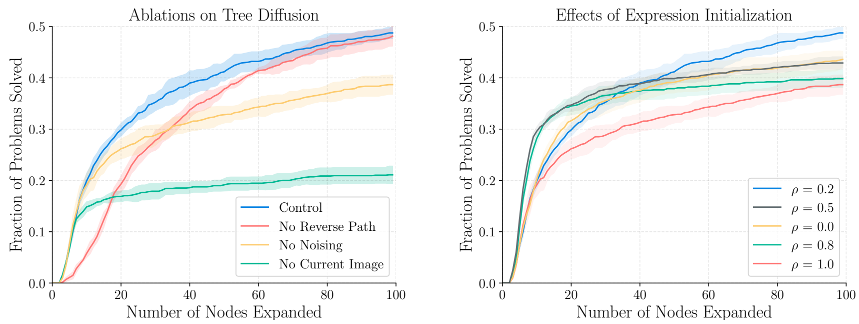
REPL Flow Ellis et al. [11] proposed a method to build programs one primitive at a time until all primitives have been placed. They also give a policy network access to a REPL, i.e., the ability to execute code and see outputs. Notably, this *current image* is rendered from the current *partial program*. As such, we require a custom *partial compiler*. This is straightforward for `CSG2D` since it is a compositional language. We simply render the shapes placed so far. For `TinySVG`, it is not immediately obvious how this partial compiler should be written. This is because the rendering happens bottom-up. Primitives get arranged, and those arrangements get arranged again (see Figure 1). Therefore, we only use this baseline method with `CSG2D`. Due to its similarities with Generative Flow Networks [3], we refer to our modified method as "REPL Flow".

Test tasks For `TinySVG` we used a held-out test set of randomly generated expressions and their images. For the `CSG2D` task, we noticed that all methods were at ceiling performance on an in-distribution held-out test set. In Ellis et al. [11], the authors created a harder test set with more objects. However, simply adding more objects in an environment like `CSG2D` resulted in simpler final scenes, since sampling a large object that subtracts a large part of the scene becomes more likely. Instead, to generate a hard test set, we filtered for images at the $95th$ percentile or more on incompressibility with the LZ4 [7, 37] compression algorithm.

Evaluation In `CSG2D`, we accepted a predicted program as matching the specification if it achieved an intersection-over-union (IoU) of 0.99 or more. In `TinySVG`, we accepted an image if 99% of the pixels were within $0.005 \approx \frac{1}{256}$.

All methods were trained with supervised learning and were not fine-tuned with reinforcement learning. All methods used the grammar-based constrained decoding method described in Section 3.4, which ensured syntactically correct outputs. While testing, we measured performance based on the number of compilations needed for a method to complete the task.

Figure 4 shows the performance of our method compared to the baseline methods. In both the `CSG2D` and `TinySVG` environments, our tree diffusion policy rollouts significantly outperform the policies of previous methods. Our policy combined with beam search further improves performance, solving problems with fewer calls to the renderer than all other methods. Figure 6 shows successful qualitative examples of our system alongside outputs of baseline methods. We note that our system can fix smaller issues that other methods miss. Figure 7 shows some examples of recovered programs from sketches in the `CSG2D-Sketch` language, showing how the observation model does not necessarily need to be a deterministic rendering; it can also consist of stochastic hand-drawn images.



(a) (b)

Figure 5: Effects of changing several design decisions of our system. We train smaller models on the `Rainbow` environment. We give the model $n = 256$ test problems to solve. In (a), for `No Reverse Path`, we train the model without computing an explicit reverse path, only using the last step of the noising process as targets. For `No Current Image`, we train a model that does not get to see the compiled output image of the program it is editing. For `No Noising`, instead of using our noising process, we generate two random expressions and use the path between them as targets. In (b) we examine the effect of training mixture between forward diffusion ($\rho = 0.0$) and pure random initialization ($\rho = 1.0$) further. Error bars show standard deviation across 5 random seeds.

4.3 Ablations

To understand the impact of our design decisions, we performed ablation studies on the simplified `Rainbow` environment using a smaller transformer model.

First, we examined the effect of removing the current image (no REPL) from the policy network's input. As shown in Figure 5(a), this drastically hindered performance, confirming the importance of a REPL-like interface observed by Ellis et al. [11].

Next, we investigated the necessity of our reverse mutation path algorithm. While training on the last mutation step alone provides a valid path, it introduces noise by potentially targeting suboptimal intermediate states. Figure 5(a) demonstrates that utilizing the reverse mutation path significantly improves performance, particularly in finding solutions with fewer steps. However, both methods eventually reach similar performance levels, suggesting that a noisy path, while less efficient, can still lead to a solution.

Finally, we explored whether the incremental noise process is crucial, given our tree edit path algorithm. Couldn't we directly sample two random expressions, calculate the path, and train the network to imitate it? We varied the training data composition between pure forward diffusion ($\rho = 0.0$) and pure random initialization ($\rho = 1.0$) as shown in Figure 5(b). We found that a small proportion ($\rho = 0.2$) of pure random initializations combined with forward diffusion yielded the best results. This suggests that forward diffusion provides a richer training distribution around target points, while random initialization teaches the model to navigate the program space more broadly. The emphasis on fine-grained edits from forward diffusion proves beneficial for achieving exact pixel matches in our evaluations.

5 Conclusion

In this work, we proposed a neural diffusion model that operates on syntax trees for program synthesis. We implemented our approach for inverse graphics tasks, where our task is to find programs that would render a given image. Unlike previous work, our model can construct programs, view their output, and in turn edit these programs, allowing it to fix its mistakes in a feedback loop. We quantitatively showed how our approach outperforms our baselines at these inverse graphics tasks. We further studied the effects of key design decisions via ablation experiments.

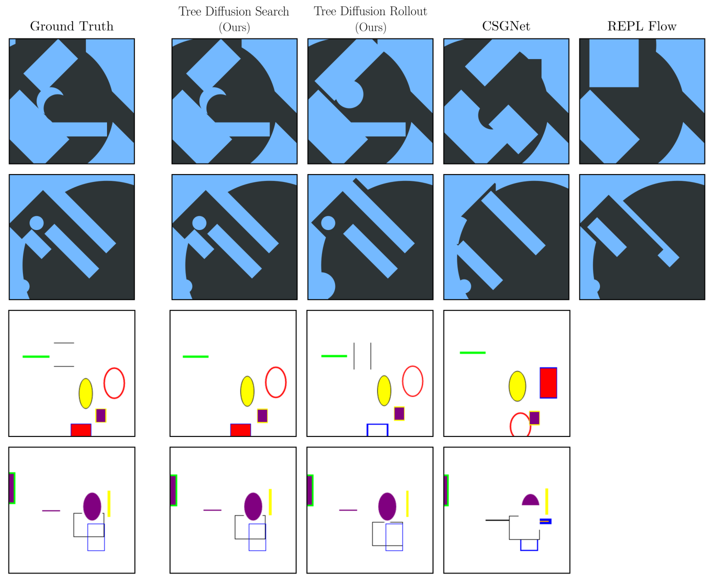


Figure 6: Qualitative examples of our method and baselines on two inverse graphics languages, `CSG2D` (top two rows) and `TinySVG` (bottom two rows). The leftmost column shows the ground-truth rendered programs from our test set. The next columns show rendered programs from various methods. Our methods are able to finely adjust and match the ground-truth programs more closely.

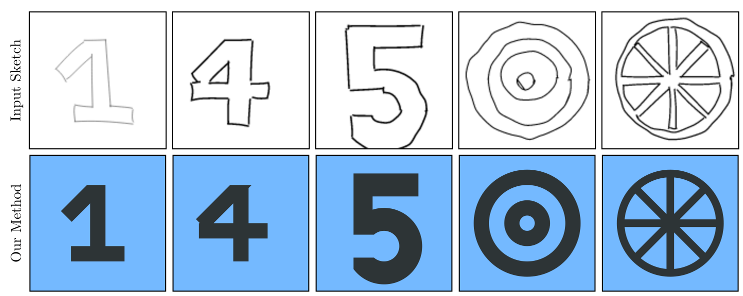


Figure 7: Examples of programs recovered for input sketches in the `CSG2D-Sketch` language. The input sketches are from our observation model that simulates hand-drawn sketches (top-row). The output programs rendered (bottom row) are able to match the input sketches by adding and subtracting basic shapes. Video results for these sketches can be found at https://td-anon.github.io/.

Limitations There are several significant limitations to this work. First, we operate on expressions with no variable binding, loops, strings, continuous parameters, etc. While we think our approach can be extended to support these, it needs more work and careful design. Current large-language models can write complicated programs in many domains, while we focus on a very narrow task. Additionally, the task of inverse graphics might just be particularly suited for inverse graphics where small mutations make informative changes in the image output.

Future Work In the future, we hope to be able to leverage large-scale internet data on programs to train our system, making small mutations to their syntax tree and learning to invert them. We would also like to study this approach in domains other than inverse graphics. Additionally, we would like to extend this approach to work with both the discrete syntax structure and continuous floating-point constants.

Impact Given the narrow scope of the implementation, we don't think there is a direct societal impact, other than to inform future research direction in machine-assisted programming. We hope future directions of this work, specifically in inverse graphics, help artists, engineering CAD modelers, and programmers with a tool to convert ideas to precise programs for downstream use quickly.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024. doi: 10.1145/3620665.3640366. URL https://pytorch.org/assets/pytorch2-2.pdf.

[2] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[3] Yoshua Bengio, Salem Lahlou, Tristan Deleu, Edward J Hu, Mo Tiwari, and Emmanuel Bengio. Gflownet foundations. *Journal of Machine Learning Research*, 24(210):1–55, 2023.

[4] Andy Brock, Soham De, Samuel L Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. In *International Conference on Machine Learning*, pages 1059–1071. PMLR, 2021.

[5] Edwin Catmull and Raphael Rom. A class of local interpolating splines. In *Computer aided geometric design*, pages 317–326. Elsevier, 1974.

[6] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399, 2020.

[7] Yann Collet et al. Lz4: Extremely fast compression algorithm. *code. google. com*, 2013.

[8] Gabriele Corso, Hannes Stärk, Bowen Jing, Regina Barzilay, and Tommi Jaakkola. Diffdock: Diffusion steps, twists, and turns for molecular docking. *arXiv preprint arXiv:2210.01776*, 2022.

[9] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 06–11 Aug 2017.

[10] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

[11] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

[12] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pages 835–850, 2021.

[13] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

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

[15] Emiel Hoogeboom, Vıctor Garcia Satorras, Clément Vignac, and Max Welling. Equivariant diffusion for molecule generation in 3d. In *International conference on machine learning*, pages 8867–8887. PMLR, 2022.

[16] IceBerg Contributors. IceBerg – Compositional Graphics Diagramming, July 2023. URL https://github.com/revalo/iceberg.

[17] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Lark Contributors. Lark - a parsing toolkit for Python, August 2014. URL https://github.com/lark-parser/lark.

[20] Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion language modeling by estimating the ratios of the data distribution. *arXiv preprint arXiv:2310.16834*, 2023.

[21] Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, 2000.

[22] Alex Nichol, Prafulla Dhariwal, Aditya Ramesh, Pranav Shyam, Pamela Mishkin, Bob McGrew, Ilya Sutskever, and Mark Chen. Glide: Towards photorealistic image generation and editing with text-guided diffusion models. *arXiv preprint arXiv:2112.10741*, 2021.

[23] Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis, 2016.

[24] Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)*, 40(1):1–40, 2015.

[25] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[26] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[27] Arne Schneuing, Yuanqi Du, Charles Harris, Arian Jamasb, Ilia Igashov, Weitao Du, Tom Blundell, Pietro Lió, Carla Gomes, Max Welling, et al. Structure-based drug design with equivariant diffusion models. *arXiv preprint arXiv:2210.13695*, 2022.

[28] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5515–5523, 2018.

[29] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[30] Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. Codefusion: A pre-trained diffusion model for code generation, 2023.

[31] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

[32] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*, pages 244–256, 2021.

[33] Maria Tsimpoukelli, Jacob L Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. Multimodal few-shot learning with frozen language models. *Advances in Neural Information Processing Systems*, 34: 200–212, 2021.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[35] Clement Vignac, Igor Krawczuk, Antoine Siraudin, Bohan Wang, Volkan Cevher, and Pascal Frossard. Digress: Discrete denoising diffusion for graph generation. *arXiv preprint arXiv:2209.14734*, 2022.

[36] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[37] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(06):8–19, 1984.

[38] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

[39] Jo Wood, Petra Isenberg, Tobias Isenberg, Jason Dykes, Nadia Boukhelifa, and Aidan Slingsby. Sketchy rendering for information visualization. *IEEE transactions on visualization and computer graphics*, 18 (12):2749–2758, 2012.

[40] David Xing Wu, Chulhee Yun, and Suvrit Sra. On the training instability of shuffling sgd with batch normalization. In *International Conference on Machine Learning*, pages 37787–37845. PMLR, 2023.

[41] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Efficient grammar fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. URL https://www.fuzzingbook.org/html/GrammarFuzzer.html. Retrieved 2023-11-11 18:18:06+01:00.

[42] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

Appendix

A Mutation Algorithm

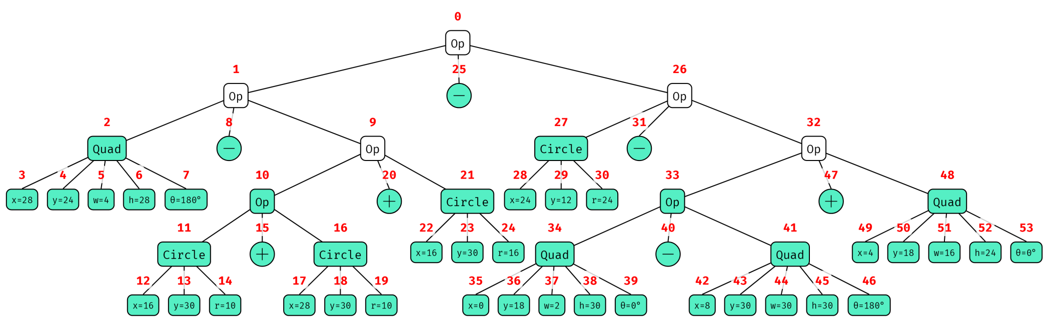
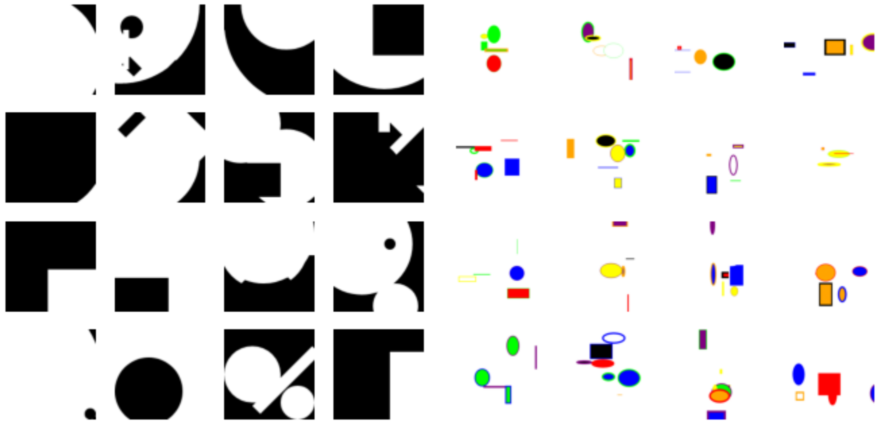


Figure 8: An example expression from CSG2D represented as a tree to help illustrate the mutation algorithm. The green nodes are candidate nodes with primitives count $\sigma(z) \leq 2$. Our mutation algorithm only mutates these nodes.

Here we provide additional details on how we sample small mutations for tree diffusion. We will first repeat the algorithm mentioned in Section 3 in more detail.

Our goal is to take some syntax tree and apply a small random mutation. The only type of mutation we consider is a replacement mutation. We first collect a set of *candidate* nodes that we are allowed to replace. If we select a node too high up in the tree, we end up replacing a very large part of the tree. To make sure we only change a small part of the tree we only select nodes with $\leq \sigma_{\text{small}}$ primitives. In Figure 8, if we set $\sigma_{\text{small}} = 2$, we get all the **green** nodes. We sample a node, m, uniformly from this **green** set. We know the production rule for m from the CFG. For instance, if we selected node 15, the only replacements allowed are $+$ or $-$. If we selected node 46, we can only replace it with an angle. If we selected node 11, we can replace it with any subexpression. When we sample a replacement, we ensure that the replacement is $\leq \sigma_{\text{small}}$, and that it is different than m. Here we show 4 random mutation steps on a small expression,

```
(+ (+ (+ (Circle A D 4) (Quad F E 4 6 K)) (Quad 3 E C 2 M)) (Circle C 2 1))
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ --> (Circle 0 8 A)
(+ (+ (Circle 0 8 A) (Quad 3 E C 2 M)) (Circle C 2 1))
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ --> (Quad 1 0 A 3 H)
(+ (Quad 1 0 A 3 H) (Circle C 2 1))
                            ^ --> 4
(+ (Quad 1 0 A 3 H) (Circle 4 2 1))
                            ^ --> 8
(+ (Quad 1 0 A 3 H) (Circle 8 2 1))
```



(a) (b)

Figure 9: Examples of images drawn with the (a) `CSG2D` and (b) `TinySVG` languages.

During our experiments we realized that this style of random mutations biases expression to get longer on average, since there are many more leaves than parents of leaves. This made the network better at going from very long expressions to target expressions, but not very good at editing shorter expressions into longer ones. This also made our model's context window run out frequently when expressions got too long. To make the mutation length effects more uniform, we add a slight modification to the algorithm mentioned above and in Section 3.

For each of the candidate nodes, we find the set of production rules for the candidates. We then select a random production rule, r, and then select a node from the candidates with the production rule r, as follows,

$$
\begin{aligned}
C &= \{n \in \text{SyntaxTree}(z) \mid \sigma(n) \leq \sigma_{\text{small}}\} \\
R &= \{\text{ProductionRule}(n) \mid n \in C\} \\
r &\sim \text{Uniform}[R] \\
M &= \{n \in C \mid \text{ProductionRule}(n) = r\} \\
m &\sim \text{Uniform}[M]
\end{aligned}
$$

For `CSG2D`, this approach empirically biased our method to make expressions *shorter* 30.8%, equal 49.2%, and longer 20.0% of the times ($n = 10,000$).

B Context-Free Grammars

Here we provide the exact context-free grammars of the languages used in this work.

B.1 CSG2D

```
s: binop | circle | quad
binop: (op s s)
op: + | -

number: [0 to 15]
angle: [0 to 315]

// (Circle radius x y)
circle: (Circle r=number x=number y=number)

// (Quad x y w h angle)
// quad: (Quad x=number y=number
```

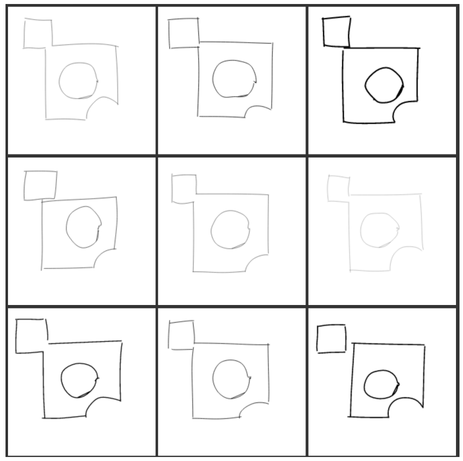


Figure 10: Examples of the same scene being called multiple times by our sketch observation model.

```
                w=number h=number
                theta=angle)
```

B.2 TinySVG

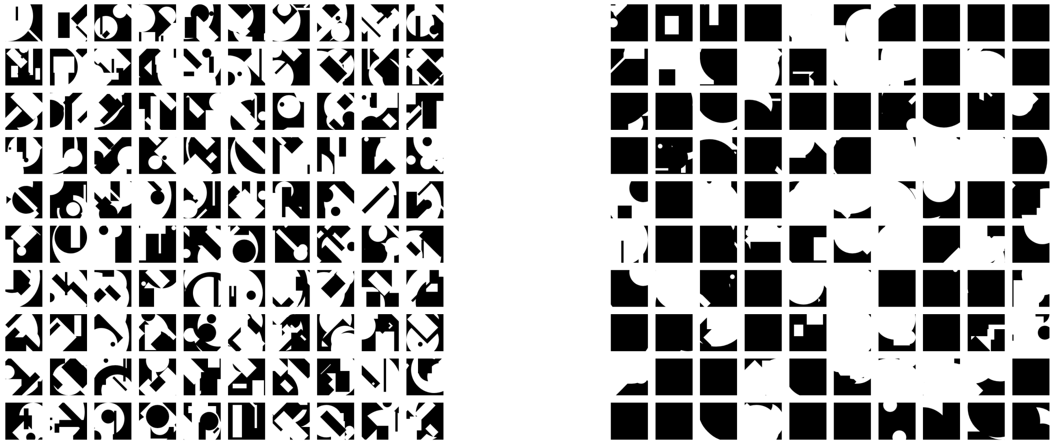
```
s: arrange | rect | ellipse | move
direction: v | h
color: red | green | blue | yellow | purple | orange | black | white | none
number: [0 - 9]
sign: + | -

rect: (Rectangle w=number h=number fill=color stroke=color border=number)

ellipse: (Ellipse w=number h=number fill=color stroke=color border=number)

// Arrange direction left right gap
arrange: (Arrange direction s s gap=number)

move: (Move s dx=sign number dy=sign number)
```

C Sketch Simulation

As mentioned in the main text, we implement the `CSG2D-Sketch` environment, which is the same as `CSG2D` with a hand-drawn sketch observation model. We do this to primarily show how this sort of a generative model can possibly be applied to a real-world task, and that observations do not need to be deterministic. Our sketch algorithm can be found in our codebase, and is based off the approach described in Wood et al. [39].

Our compiler uses Iceberg [16] and Google's 2D Skia library to perform boolean operations on primitive paths. The resulting path consists of line and cubic bézier commands. We post-process these commands to generate sketches. For each command, we first add Gaussian noise to all points stated in those commands. For each line, we randomly pick a point near the 50% and 75% of the line, add Gaussian noise, and fit a Catmull-Rom spline [5]. For all curves, we sample random points at uniform intervals and fit Catmull-Rom splines. We have a special condition for circles, where we ensure that the start and end points are randomized to create the effect of the pen lifting off. Additionally we randomize the stroke thickness.

Figure 10 shows the same program rendered multiple times using our randomized sketch simulator.



(a) (b)

Figure 11: Examples of thresholding scene images using the LZ4 compression algorithm. The left represents our test set, the right represents our training distribution.

D Complexity Filtering

As mentioned in Section 4, while testing our method alongside baseline methods, we reached ceiling performance for all our methods. Ellis et al. [11] got around this by creating a "hard" test case by sampling more objects. For us, when we increased the number of objects to increase complexity, we saw that it increased the probability that a large object would be sampled and subtract from the whole scene, resulting in simpler scenes. This is shown by Figure 11(b), which is our training distribution. Even though we sample a large number of objects, the scenes don't look visually interesting. When we studied the implementation details of Ellis et al. [11], we noticed that during random generation of expressions, they ensured that each shape did not change more that 60% or less than 10% of the pixels in the scene. Instead of modifying our tree sampling method, we instead chose to rejection sample based on the compressibility of the final rendered image.

E Tree Path Algorithm

Algorithm 1 shows the high-level pseudocode for how we find the first step of mutations to transform tree A into tree B. We linearly walk down both trees until we find a node that is different. If the target node is *small*, i.e., its $\sigma(z) \leq \sigma_{\text{small}}$, then we can simply mutate the source to the target. If the target node is larger, we sample a random small expression with the correct production rule, and compute the path from this small expression to the target. This gives us the *first step* to convert the source node to the target node. Repeatedly using Algorithm 1 gives us the full path to convert one expression to another. We note that this path is not necessarily the optimal path, but a valid path that is less noisy than the path we would get by simply chasing the last random mutation.

Figure 12 conceptually shows why computing this tree path might be necessary. The circle represents the space of programs. Consider a starting program z_0. Each of the black arrows represents a random mutation that *kicks* the program to a slightly different program, so $z_0 \to z_1$, then $z_2 \to z_3 \ldots$. If we provide the neural network the supervised target to go from z_5 to z_4, we are teaching the network to take an inefficient path to z_0. The green path is the direct path from $z_5 \to z_0$.

F Implementation Details

We implement our architecture in PyTorch [1]. For our image encoder we use the NF-ResNet26 [4] implementation from the open-sourced library by Wightman [38]. Images are of size $128 \times 128 \times 1$ for `CSG2D` and $128 \times 128 \times 3$ for `TinySVG`. We pass the current and target images as a stack of image planes into the image encoder. Additionally, we provide the absolute difference between current and target image as additional planes.

Algorithm 1 TreeDiff: Find the first set of mutations to turn one tree to another.

Require: `treeA`: source tree, `treeB`: target tree, `max_primitives`: maximum primitives
Ensure: List of mutations to transform `treeA` into `treeB`

```
if NodeEq(treeA, treeB) then
  mutations ← []
  for each (childA, childB) in zip(treeA.children, treeB.children) do
    mutations ← mutations + TreeDiff(childA, childB, max_primitives)
  end for
  return mutations
else
  if treeA.primitive_count ≤ max_primitives and treeB.primitive_count ≤ max_primitives then
    return [Mutation(treeA.start_pos, treeA.end_pos, treeB.expression)]
  else
    new_expression ← GenerateNewExpression(treeA.production_rule, max_primitives)
    tightening_diffs ← TreeDiff(new_expression, treeB, max_primitives)
    new_expression ← ApplyAllMutations(new_expression, tightening_diffs)
    return [Mutation(treeA.start_pos, treeA.end_pos, new_expression)]
  end if
end if
```

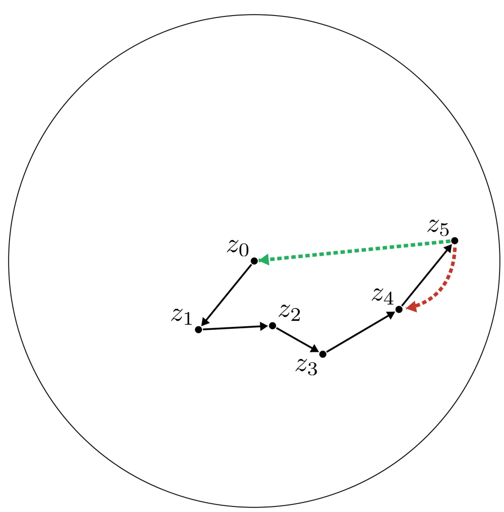


Figure 12: A conceptual illustration of why we need tree path-finding. The red path represents the naive target for the neural network. The green path represents the path-finding algorithm's target.

Our decoder-only transformer [34, 26] uses 8 layers, 16 heads, with an embedding size of 256. We use batch size 32 and optimize with Adam [18] with a learning rate of 3×10^{-4}. The image embeddings are of the same size as the transformer embeddings. We use 4 prefix tokens for the image embeddings. We used a maximum context size of 512 tokens. For both environments, we sampled expressions with at most 8 primitives. Our method and all baseline methods used this architecture. We did not do any hyperparameter sweeps or tuning.

For the autoregressive (CSGNet) baseline, we trained the model to output ground-truth programs from target images, and provided a blank current image. For tree diffusion methods, we initialized the search and rollouts using the output of the autoregressive model, which counted as a single node expansion. For our re-implementation of Ellis et al. [11], we flattened the `CSG2D` tree into shapes being added from left to right. We then randomly sampled a position in this shape array, compiled the output up until the sampled position, and trained the model to output the next shape using constrained grammar decoding.

This is a departure from the pointer network architecture in their work. We think that the lack of prior shaping, departure from a graphics specific pointer network, and not using reinforcement learning to fine-tune leads to a performance difference between their results and our re-implementation. We note that our method does not require any of these additional features, and thus the comparison is fairer. For tree diffusion search, we used a beam size of 64, with a maximum node expansion budget of 5000 nodes.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Figure 4 directly shows the performance of our method over baselines. Further, we discuss the narrow scope that we tested our more general approach in Section 5.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations and scope are discussed in Section 5.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: There are no theoretical results in this work.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: In addition to submitting a clean implementation of our work alongside model weights, our paper has enough details to reproduce the results. All code and weights will also be release publicly.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: At the time of submission we provide anonymous code and weights for our method and our baselines. We will also open source all code and weights.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: Main text has most of the high-level ideas. Specific values and implementation details are in Appendix F. We also release code with our work.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: All quantitative experimental figures have error bars computed by running the experiment multiple times with different random seeds and computing the standard deviation of the results.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 3.4 mentions these details.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: No human subjects or external datasets were used. Additionally, we don't think the scope of the work can cause any direct harm.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Section 5 discusses potential impacts.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.

- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We don't think our work is high-risk for misuse.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: There are no datasets curated or external assets used. We cite all open-source Python libraries used, alongside their URLs.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We provide a well-documented `README.md` file for the code and model weights released.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: There were no human subjects used.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: There were no human subjects used.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.