We thank our reviewers for taking the time to critique and improve our paper.

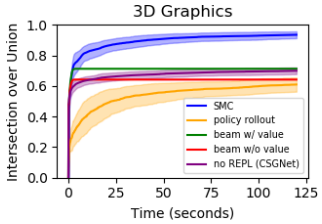
Reviewer 1 (R1) suggests comparing with related 3D graphics synthesis work, particularly Tian et al. 2019. Although our submission cites Tian et al., we agree that this work deserves a more extended discussion. Like us, they introduce an approach to inferring programmatic 3D models of objects. A key similarity between our works is that we both condition on intermediate renders, a feature also shared with other works in the computer vision literature such as Eslami et al. 2016 ("attend-infer-repeat"); Richtie et al. 2016; Ellis et al. 2018; and Ganin et al. 2018 ("SPIRAL"). Our revision will discuss these similarities. However, we disagree that our approach is a "small variation" of this work, as R1 claims. In particular, Tian et al. takes a sophisticated approach that is nonetheless *specialized* to graphics programs, while we propose a *general* framework for program synthesis, allowing us to also solve very different problems such as FlashFill-style text manipulation. Specifically, the success of Tian et al. hinges on exploiting the differentiability of the rendering process to train a program generating policy and generalize it beyond the training distribution via fine-tuning by gradient descent. In contrast, we assume a generic REPL that is not differentiable, and generalize beyond the training distribution by leveraging search (SMC) at test time. Furthermore, the goals and problem statement of our 3D work differ enough from those of Tian et al. to preclude a head-to-head experimental comparison. Our goal is generic geometric shape reconstruction from low-level primitives via algebraic manipulations (i.e. union/difference). In Tian et al., the goal is to decompose everyday objects into sub-parts and symmetries; all components are 'unioned' together, and they assume specialized semantic primitives such as 'Leg' and 'ChairBeam'.

Instead, we feel the most appropriate related work to experimentally compare with is CSGNet (Sharma et al. 2018) for graphics programs and RobustFill (Devlin et al. 2017) for text editing, because they solve the same kinds of inverse CSG and text editing problems we do. We compared with our reimplementation of CSGNet (Figure 5 of our submission) and with a reimplementation of RobustFill (Figure 7 of our submission), in both cases finding that a REPL with learned stochastic search outperforms these alternatives. Our text editing test cases are especially challenging for RobustFill: we originally evaluated on existing text editing benchmarks from the literature (Nye et al. 2019), but found that our REPL approach was at ceiling (98% solved) for these benchmarks (line 204 of submission & Fig. 10 of appendix), and so needed to design an even more challenging benchmark suite (Fig. 7-8 & Appendix Fig. 11). For these reasons, we believe that our current baseline results adequately answer the questions our paper poses, namely whether a REPL is useful to a code-writing agent, and how to integrate the REPL at test time with a symbolic search procedure.

R1 points out a spot of possible confusion: during training, we use a 0/1 reward function without partial credit, but during testing, we use graded metrics (intersection-over-union for inverse CAD) to evaluate the performance of baselines. We are able to successfully use a 0/1 reward during training because we bootstrap our policy with imitation learning.

Reviewer 2 (R2) suggests trying more sophisticated RL training procedures. For instance, one could use Actor-Critic methods to train π and v, *both* for our full model, *and* for our baselines and ablations; or, one could use SMC *at train time*, like how AlphaGoZero uses MCTS both at train and test. We opted for vanilla REINFORCE as the simplest algorithm to answer the question of how to employ a REPL with learned search, and we will discuss these next steps in a revision. Learning the policy π without imitation would be difficult due to the large action space (> 1.3 million different actions for 3D CAD, line 183 of paper), which results in very sparse reward signals even with search.

R2 asks several questions about our SMC sampler that will be clarified in the revision. SMC without value guided proposals would be equivalent to drawing samples from the policy; unlike MCMC methods, SMC does not suffer from "burn-in" or "mixing" issues, but may need a large population of particles (we repeatedly double the particle count until timeout); the SMC seed slightly affects the time to recover the results in Figure 6 (see graph to right: error bars over 5 random seeds and correspond to stddev), but typically leaves the final rerendering unaffected.



Reviewer 3 (R3) asks good questions about the order of syntax tree generation, particularly regarding the string editing domain. We chose bottom-up generation because it allows execution of partially written programs, and most closely mirrors how people write code in a REPL. For string editing, we kept the semantics of the RobustFill DSL the same but modified the syntax (Appendix A.2) to ensure that intermediate executions are available during bottom-up generation. To compare to other forms of program generation, a recent piece of related work, (Nye et al. 2019) considers *top-down* generation of programs, in which partial programs cannot be executed. As noted above, our performance on the dataset of Nye et al. far exceeds the performance of their top-down system (98% solved vs. 77% solved), which we believe helps show the value of a REPL-based approach that can execute code on-the-fly as it is written.

R3 points out that our value estimation loss is nonstandard. This is because our expected future reward coincides directly with the probability of the agent generating a correct program (discussed in lines 116-119 of paper). Thus, it is natural to employ a cross-entropy loss because we are predicting a probability: unlike expected reward in e.g. Atari games such as Pong, our expected reward has a probabilistic interpretation.