IMDIM CAD Modeling/Paramesh — Project Overview, Pretraining & Pivot

A concise overview of our project story: origin, model design, pretraining, failures, fixes, pivot, and future directions.

Original Project Vision

- Task: Convert **images + parameters → parametric 3D CAD models**
- Original plan: **Brute-force program synthesis**
  - Language: **OpenSCAD**, later **MicroCAD / µCAD**
  - Enumerate/search tokens + parameters
  - Compare rendered mesh vs target mesh/image
- Pros: deterministic, interpretable, exact CAD
- Cons: combinatorial explosion, too slow to scale, brittle

Switch to Reinforcement Learning

- CAD models are built **sequentially**
- Each step changes geometry → new feedback signal
- RL fits this pattern:
  - Output token + parameters
  - Environment returns geometric error
  - Reward = improvement
- Hope: learn useful multi-step construction policies not feasible via brute force

Error Embedding (Why)

- The model needs awareness of how well the **current partial shape** matches ground-truth geometry.
- Token history alone isn't enough; parameters need geometric context.
- Reinforcement-learning intuition: "state" should include error-signal.

Error Embedding (What goes in)

- Error Embedding Design:
  - Compute chamfer-like distance
  - Convert numeric error into a compact vector used as conditioning.
- Inputs currently included:
  - chamfer distance (scalar or small vector summary)
  - optionally comparing center/extent differences
  - rotation/angular differences (coarse bins)
  - sign (add/sub) isn’t included here—handled separately in params

Model Architecture

- **PointNetEncoder**
  - Encodes ground-truth point cloud (**gt_embed**)
  - Encodes current partial model cloud (**cur_embed**)
- **Error embedding (8-dim)**
- **CADTransformer** *(6 layers, d_model=256, 8 heads, FFN=1024)*
  - Sequence model over token history
  - Conditions on gt_embed + cur_embed + error features

Model Architecture p2

- **Heads**
  - **TokenHead** *(256→256→V)* → next primitive type
  - **ParamHead** *((256+256)→256→256→10)* → next 10-D parameter vector
    - [cx,cy,cz, p0,p1,p2, rx,ry,rz, sign]

Why Pretraining?

Goal: give the RL policy a useful initialization

- Difficult RL reward landscape → cold-start fails
- Pretraining teaches:
  - Which primitive to place (box/sphere/cylinder/etc.)
  - Rough parameters (position/scale/rotation/positive|negative)
- Uses **supervised learning** on generated datasets of CAD sequences

How Pretraining Works

- Teacher-forcing
- Inputs:
  - History tokens + params
  - Current partial point cloud
  - Ground-truth full point cloud
- Training target: Next token & parameters
- Losses:
  - **token_loss** = CrossEntropy
  - **param_loss** = MSE (with dataset-level normalization)

Mesh Generation — Legacy

- **OpenSCAD**
  - Old and slow for large-scale generation
  - No real introspection / programmatic hooks
  - Codebase is crusty and hard to extend
- Mesh pipeline (then):
  - parameters → codegen → write to disk → export STL → STL renderer
  - Works, but **insanely slow** for iterative search

Mesh Generation — Datasets

- Existing datasets:
  - **ABC** (Onshape / online CAD)
    - Rich, but the underlying “grammar” is far too complex for our DSL
  - **Exported CAD models**
    - Don’t come with the clean parametric programs we need
  - **3D scans**
    - No access to **our** specific inputs (CAD programs + parameters)
- Practical option:
  - **Generate our own models randomly**

Mesh Generation — CAD Tooling Choices

- Other parametric CAD systems:
  - Mostly **proprietary formats**
  - Often **GUI-only**, little to no scriptable backend
- Need:
  - A small, scriptable, open language that we can call in a loop
  - Clean bridge from **parameters** → **geometry** without a human in the loop

Mesh Generation — Rust

- Rust-based CAD kernels:
  - **Fornjot**:
    - Full modeling kernel
    - Would require us to generate valid Rust code per sample
  - **MicroCAD (µCAD)**, based on Manifold:
    - Very fast, simple DSL, new and still evolving
- Pipeline (now):
  - parameters → **MicroCAD codegen** → export mesh → viewer
    - **Blazingly fast** compared to the STL-on-disk loop

Mesh Generation — MicroCAD Limitations

- MicroCAD is **not** designed for our exact use case:
  - No first-class support for massive batched program synthesis / RL loops
  - Limited introspection on intermediate geometry states
- Result:
  - We spent a **lot** of time hacking on the CAD toolchain itself
  - However the mesh generation was fast

Dataset Generation

- Initially just generated single random shapes in a constrained space
- Once we were able to somewhat recreate single shapes we would then go back and upgrade the dataset
- The dataset plan
    - Never got that far

What Went Wrong (Observed)

- Parameter loss dominated by numerical issues:
  - Near-zero ground-truth values caused huge squared error when predicted slightly off.
- No normalization
- Transformer not robust to variable history length, padding, masking

Diagnostic Evidence

- Per-batch debug output showed:
  - Extremely small numeric scales → exploding error
  - Param loss >> token loss (1e9 vs small CE)
- Model “memorized” tiny toy examples but generalization poor
- Prediction qualitatively close in 3D space sometimes, but token wrong
- Under the hood: still fundamentally unstable

Fixes Attempted

- Dataset-level parameter normalization
  - Script computes per-dimension scales, clamps zero-variance dims
- Adjusted training:
  - Rebalanced token vs param loss
- Richer error embeddings
- More expressive transformer hyperparameters

But: did not fix exposure bias or teacher-forcing mismatch.

Still Likely Wrong

- ParamHead receives perfect tokens during pretraining → unrealistic at inference
- Autoregressive rollout not simulated in training
- Padding/masking required for proper batching
- Param ranges huge; some dimensions rarely used
- Transformers need causal masking + scheduled sampling to be reliable

Pivot Back to Brute Force

- RL pretraining too unstable for current time budget
- Iteration speed / visual feedback on brute force is much better
- We returned to:
  - MicroCAD code generation
  - Heuristic search on token sequences
  - Parameter search per-primitive
  - Basic demonstration of incorporating user feedback live
- Heuristics need to be better, currently hard to beat mk1 eyeball
- Incorporate this strategy into CEGIS, sketch-based, etc.

If We Had More Time (RL)

- Debug pretraining and continue to RL
- Proper causal transformer training
- Scheduled sampling to reduce exposure bias:
  - ParamHead always trained with ground-truth tokens, but inference uses predicted tokens
- Masking & padding
- Vectorized ParamHead
- RL fine-tuning: PPO/A2C, Shaped rewards (intermediate + final)
- Curriculum learning: start short programs → longer

If We Had More Time (Brute Force)

- Smarter parameter search:
  - Bayesian optimization
  - Local continuous refinements
- Policy-guided beam search: use pretrained policy only as heuristic
- Multi-stage: coarse → fine
- Hybrid renderer or differentiable surrogate to speed evaluation

Summary

- Original brute-force concept → too slow and inaccurate
- Switched to RL + pretraining
- Built a substantial CAD policy model (PointNet + Transformer)
- Identified major stability & mismatch issues
- Pivoted back to brute-force but incorporate user input
- Next steps: hybrid approach + robust RL training if time permits