Enabling AI in Computer Aided Design through Representations

Benjamin Tod Jones

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2024

Reading Committee:

Adriana Schulz, Chair

Brian Curless

Steven Tanimoto

Vladimir Kim

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Enabling AI in Computer Aided Design through Representations

Benjamin Tod Jones

Chair of the Supervisory Committee:
Adriana Schulz
Paul G. Allen School of Computer Science & Engineering

Most man-made objects started their life as a computer model in a Computer Aided Design (CAD) system, creating a vast trove of data about real objects and a tantalizing target for training artificial intelligence (AI) models. But can, and should, we create AI systems for CAD, and if so, how? This thesis explores this question through the lens of design loops: the iterative process through which people solve problems. It finds that modern CAD systems are designed around a symbiosis of two design representations — programmatic modeling and parametric geometry — that allow CAD systems to integrate into design workflows.

Based on this, I propose to structure AI integration into CAD around these representational choices. I present a representation learning strategy for the parametric boundary representation (B-Rep), the symbolic geometry format common to most modern CAD systems and demonstrate its use in several design tasks. I then explore how generative AI can be used to modify and generate procedural CAD programs. Ultimately, I conclude that AI can and should be integrated into CAD through careful consideration of, and integration with, the design representations that have become inseparable from the discipline.

Contents

Acknowledgments

There are many people who have my gratitude in the process of making this dissertation. First and foremost, I need to thank my family and friends for their unwavering support and understanding many disappearances of this "cryptid" around deadlines.

Within the Allen school I have been fortunate to have several great mentors. Thank you, Brian Curless, for introducing me to graphics in my first days of grad school, even though it took me four years to finally make the switch! And to Steve Tanimoto for helping me find my way when I was lost after leaving my first lab. You gave a name to my interests in design-space exploration, and opened the door to a new (to-me) field of research. To my advisor, Adriana Schulz: I am so very grateful for your tireless efforts introducing me to, and helping me integrate with the design and graphics communities. And thank you for sticking it out through umpteen nail-biting deadlines.

Speaking of collaborators, the work presented here is the effort of many wonderful people, without whom this would not have been possible. Thanks to Ilya Baran for sharing his deep knowledge of CAD systems across numerous projects and for helping to collect the AutoMate dataset. Thanks to Vova Kim and Maaz Ahmad for sharing their expertise in machine learning and programming languages, and for hosting me at Adobe research. My fellow students and mentees have also been integral co-authors on these works. Thank you to Milin Kodnongbua, James Noeckel, Dalton Hildreth, Felix Hahnlien, Michael Hu, and Duowen Chen for the many hours of

brainstorming, coding, writing, and testing, and for the many late nights and early mornings.

Finally, I would like to thank my reading committee, and to paraphrase Blaise Pascal: *I would have written a shorter thesis, but I did not have the time.* Thank you so very much for your support through these many years, I hope you enjoy the fruits of all our labors.

Chapter 1

Introduction

Artificial Intelligence (AI) is advancing at an explosive pace. Large scale foundational models are swiftly devouring all the text, image, audio, and video data that the internet has to offer, and appear hungry for more. But what kind of data is left to be found? Computer Aided Design (CAD) models of manufactured objects present an intriguing possibility. Nearly every object manufactured in the past forty years was, at some point, a model in a CAD system, and CAD data formats have been an ISO standard for three decades.

If such a rich source of data about real world objects exists, a better question might be why machine learning (ML) is not common in CAD already? The short answer is that CAD data has several peculiarities that require customized solutions to integrate with AI technologies.

A secondary, but more important question is *how* to integrate AI into CAD. This is both a question of capability and of strategy; what the right way is to make AI useful in the context of CAD, and what technologies need to be developed to enable this.

To answer these questions, I focus on the "D" of CAD — *design*, and trace how considerations of the design process shaped the structure of existing computer systems for CAD. This analysis finds that modern CAD systems earn their place in design workflows via a careful choice of *representa-*

tions: **symbolic geometry** that is defined by **procedural** specifications. This key observation motivates the development of representation learning techniques for symbolic geometry, and integration of generative AI with procedural specifications.

1.1 Organization

This thesis is organized into four parts. The first is a background that explores what it means to "design" in the context of manufactured objects, and how that influences system building, culminating in an overview of how modern CAD systems are built.

The second part explores the use of machine learning in symbolic geometry, presenting a graph-based representation learning strategy for CAD geometry, two case studies illustrating its effectiveness in engineering workflows, and culminating in a geometric self-supervision loss that paves the way for training a foundation model for CAD geometry.

The third part focuses on existing foundational models, and how they can be integrated with the procedural modeling side of CAD. It contains two systems, one that finds procedural variations using image diffusion models, and another that generates CAD procedures using a large language model (LLM).

The last section describes a vision for the future of AI in CAD and design in general, pulling together and drawing on the lessons learned in building these systems.

Chapter 2

Background

2.1 Design

While colloquially "CAD" has become a verb that is used to mean 3D modeling in a program like SolidWorks [sol], starting with this narrow definition would rob us of valuable insight around why those programs are built the way they are. Instead, *Computer Aided Design* should first be considered in its most general sense: using a computer to help with design. Viewed through this lens, we are forced to answer the questions "what is design?" and "how can computers be helpful in that?"

What is design?

Simply put, design is the process through which people solve problems. That is an unhelpfully broad statement, but luckily a thousand and one people have attempted to analyze and explain how design happens. Exploring this literature is far beyond the scope of this dissertation, so I will instead present a simple, but powerful model of the design that synthesizes the most commonly agreed-upon points.

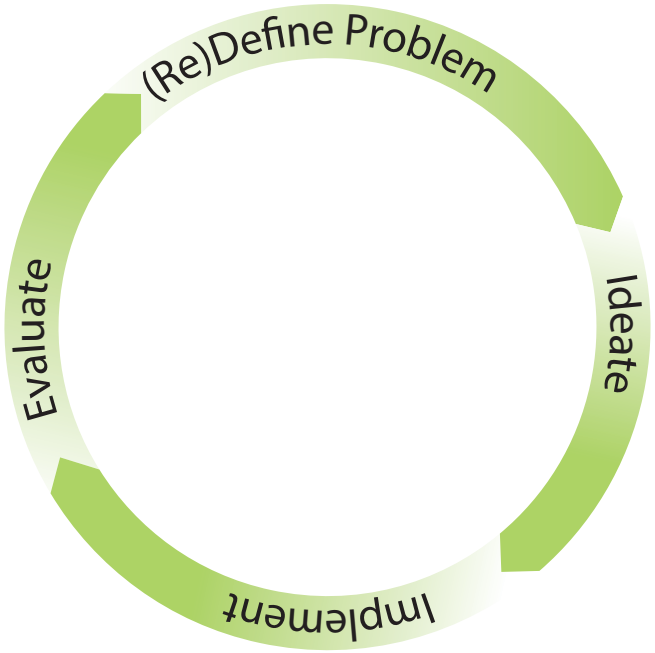


Figure 2.1: The design process. Design is an iterative process beginning with formulating the problem to be solved, moving on to ideation of possible solution designs, followed by implementing designs which are then evaluated. The insights gleaned from this process are used to refine the design question and brainstorming in subsequent iterations.

The first, and most important, of these is that *design is an iterative process.* It is rare that the best answer to a design question can be directly obtained; in fact, it is usually the case that the question itself is wrong from the start! Design proceeds in a four phase loop, illustrated in Figure 2.1.

The stages of the design process are:

1. (Re)Define the Problem: Formulate the question to be answered or problem to be overcome. This is where the goal(s) to be achieved are considered, as well as boundaries to respect. In the language of optimization, this is where *objectives* and *constraints* are defined. Deciding how a potential solution will be judged is important for the final stage, *Evaluation.*

2. Ideation: Brainstorm designs for possible solutions, and pick one or more to prototype.

3. Implementation: Construct prototype(s) for the design(s) chosen during ideation. A prototype does not need to be a physical object or a complete realization of the idea, it just needs to be sufficiently realized in order to evaluate it by the criteria chosen during the Definition stage.

4. Evaluate: Test the implemented prototypes against the criteria chosen in Problem Definition. Evaluations can be objective, like cost or weight, or subjective, like comfort or aesthetic considerations.

In reality, design rarely proceed exactly according to this loop. A designer will skip back and forth between steps, going back to ideation if a design proves too difficult to implement, or skipping going back to Problem Definition if they gain an insight worth reframing the problem before evaluating their current solutions. Real design processes will also contain innumerable sub design problems; design epicycles that are solved in the process of solving a larger problem. In a sense CAD modeling is one such epicycle; the subproblem of designing the physical form of a solution.

Regardless of irregularities in process, there are a few important observations to make about this framework. The first is that I was careful to define Ideation as brainstorming *designs* rather than solutions directly, and to state that implementations need not be complete realizations. Especially in the early stages of design, it is rare to fully specify one's ideas. Brainstorming can be as abstract as making freehand sketches or describing an idea in words. And an implementation may be a rendering or a 3D model; perhaps a designer brainstorms the shape of a handle as a sketch, then implements that by molding their sketch in clay to evaluate it subjectively by how it feels in the hand. Notice that the process of implementation is changing the format of the design, and the design is getting more *concrete*. This is the second observation, that within a design loop, ideas move from less to more concrete. This is the overall structure of design; ideas are refined

from vague to specific via iterations of broadening ideation then concretizing implementation. Design ideas changing format is such an integral part of the design process that it is important to give them a name: **design representations**.

What is A Design?

I use the noun design to refer to a *representation* of a solution to a problem; that is a description of a solution in any medium, and to any level of detail. A representation is considered more or less concrete (higher or lower level) by how many possible unique solutions it describes. Taking an example from computer science; a listing of C++ code can represent several possible machine code programs, depending on which compiler and compile flags are used. The term representation can be used to refer either to a specific design, or to the format in which that design is expressed. The usage will normally be clear from context, but where necessary I will use *representation schema* to refer to the format, and *representation instance* for specific designs. Because implementing an ideated representation instance makes it more concrete, in any design loop iteration there will usually be at least two representation schemas in use; a less concrete *specification* schema, and a more concrete *implementation* one.

Computers in Design

The design process described so far can happen with or without computer assistance, but it was important to set up the distinction between specification and implementation representations before bringing in machines. In computer design tools, these usually map to the user interface, and an internal machine format. For example, in a vector design tool the user works with a high-level representation of circles, lines, and other primitives in screen space, whereas the program itself is manipulating a list of coefficients for spline curve equations. Working in lower-level representations is difficult for people, and the constraints they impose can even reduce creativity!

[IPR10, SC07, BTL09] Computer tools for manufacturable design are no exception to this rule of dual representations, and have been honed into a common pattern, the modern parametric CAD system.

Computer Aided Design

Modern CAD Parametric modeling systems are the de-facto standard in modern Computer Aided Design (CAD) for manufacturing, to the point that "CAD modeler" or "CAD system" has become synonymous with parametric modelers. Though there are many and varied CAD systems, they all owe their success to a common choice of *design representation*; a symbiotic combination of *symbolic geometry* and *procedural definition*. The form of a design is specified from a procedural CAD program, usually edited in a GUI application. CAD programs generate symbolic geometry, usually in the form of a parametric boundary representation (B-Rep). This symbolic geometry is, in turn, visualized back to the designer and used *by reference* to specify the next step of procedural generation. Together, these representation choices allow parametric designs to be *precise*, *accurate*, and *editable*, three properties necessary for the iterative process of mechanical design.

CAD Programs The primary, user-facing representation schema in modern CAD systems are procedural CAD programs. These consist of *construction sequences* of geometric operations in two or three dimensions. Two-dimensional operations are called *sketching*, and involve specifying 2D primitives such as lines, circles, spline curves, etc., as well as specifying and solving for *geometric constraints* (coincident, parallel, etc.) between these primitives. Importantly, sketches can be *dimensioned*, given precise measurements either by definition or by constraint. This makes CAD programs a perfectly *accurate* way of specifying a shape, a requirement for manufacturability.

Two-dimensional sketches are lifted into three dimensions by 3D constructive operations, such as extrusion, revolution, sweeping, or lofting. Other 3D operations can modify existing 3D

geometry, such as chamfering or filleting edges, or applying shape Boolean operations between construction 3D shapes. CAD systems also allow for functional objects to be modeled as *assemblies* of rigidly transformed shapes by specifying further geometric constraints between constructed geometry.

Throughout the modeling process, a designer will jump between each of these stages of construction, 2D sketching, 3D construction, and assembling, as necessary. A core advantage of modern CAD that sets it apart from other 3D modelling systems is that the construction sequence is explicitly modeled and exposed to the designer, allow them to go back and edit previous steps and allow those changes to propagate throughout the model, providing *editability* of the model. This is a step above control over the shape, editability means that the designer can quickly make modifications enabling faster iterations over designs. Editability is further enhanced by exposing the program structure since most systems allow the user to define variables or parameters that drive the CAD program, making some edits as simple as changing the value of an input variable! This is an incredibly powerful tool, but as we will see in Chapter 7, its potential is difficult to realize.

Each construction step builds procedurally on top of the geometric output of the previous step, and the instructions for that procedure rely on being able to specify where in that previous output to modify. For example, creating a surface of revolution requires specifying a sketch curve to revolve and an axis to revolve it around. The ability to make these specifications is called *referencing* and is one of the primary advantages to the kind of symbolic geometry preferred in modern CAD systems; the parametric boundary representation (B-Rep).

Parametric Boundary Representations (B-reps) CAD programs generate, and generate off-of, a form of symbolic geometry called a parametric boundary representation (B-Rep). B-Reps represent shapes as networks of bounded, 2D, parametric surface patches called *faces*.

A face consists of a surface, which is a parametrically defined 2D manifold (e.g. a function $S(u, v) : \mathbb{R}^2 \to \mathbb{R}^3$, commonly planes, cylinders, etc.), plus the *boundaries* of that manifold which are defined by reference to lower dimensional parametric geometry.

Boundaries of faces are defined by referencing *edges*, another B-Rep entity that packages a 1D parametric *curve* ($C(t) : \mathbb{R} \to \mathbb{R}^3$, e.g. line, arc, etc.) along with further references to the *vertices* that bound the edge.

Faces, edges, and vertices are called the *topological entities* of a B-Rep, and form a linked, hierarchical structure where lower dimensional topologies bound higher dimensional ones. This is required because the *geometric entities*, surfaces, curves, and points, associated with each topological entity are generally unbounded.

These three kinds of topological entities form the core of the B-Rep hierarchy, but it is common to add intermediate organizing levels between and above them which do not have explicitly attached geometry. Edges can be grouped into closed *loops* that bound faces, which can be classified as bounding their face on the inside or outside. Similarly to loops, closed *shells* of faces are grouped together, and used to enclose and define volumetric *solids*, sometimes also known as *bodies*.

The reason for defining these extra levels of topological structure is that they provide semantically meaningful decompositions of the shape into entities that can be *referenced*. A full planar or cylindrical face of a B-Rep can be referred to directly as a single entity, in contrast to a representation like a triangle mesh where it would be broken arbitrarily into many triangles. Intermediate topology can reference negative space like holes in a surface (loops), or voids in a solid (shells). B-Rep references are one of the primary types for arguments in CAD programs.

The ability to reference makes B-Reps a perfect companion for procedural definitions, and the choice to define geometry symbolically makes them ideal for manufacturing. This is because symbolic geometry can be specified with exact *accuracy*, inherited from the program definitions, and it

can be evaluated to arbitrary *precision*, which is necessary for achieving manufacturing tolerances. Together, the pairing of CAD programs with B-Rep geometry creates a way of constructing models that are precise, accurate, and editable, and thus appropriate both for manufacturing and for the design process itself. If AI is to break into the CAD world, the representations it uses also need to possess these crucial properties.

2.2 CAD and AI in Manufacturing Design

Current CAD system are primarily used in implementation and evaluation phases of design. The left half of Figure 2.2 highlights some of the most common applications of modern CAD. Once a human designer has an idea they wish to evaluate, they created a CAD program with 3D and assembly modeling, that ultimately creates parametric boundary representations. Having a CAD model opens up many avenues of evaluation. The model can be rendered for subjective aesthetic evaluation or experiencing the design in VR. Physically simulations can evaluate functional considerations, or the design can be used in Computer Aided Manufacturing (CAM) to drive automated or manual fabrication of a physical prototype to test functionality concretely.

The next two parts of this dissertation explore how AI can be used to add capabilities to this design loop. The right side of Figure 2.2 places these projects in the design loop and highlights the AI techniques they apply. The next four chapters work with B-Rep representations, developing new techniques for representation learning that are tested in three contexts: learning to assemble (AutoMate), learning correspondences between B-Reps to assist in collaboration (Matching), and pre-training for few-shot learning, that has a wide range of applications from on-the-fly modeling automation to predictive evaluation (Geometric Self-Supervision).

The following two chapters begin an exploration of how emerging generative AI technologies can be applied to CAD. These focus on the early phases of the design loop. Foundational Image

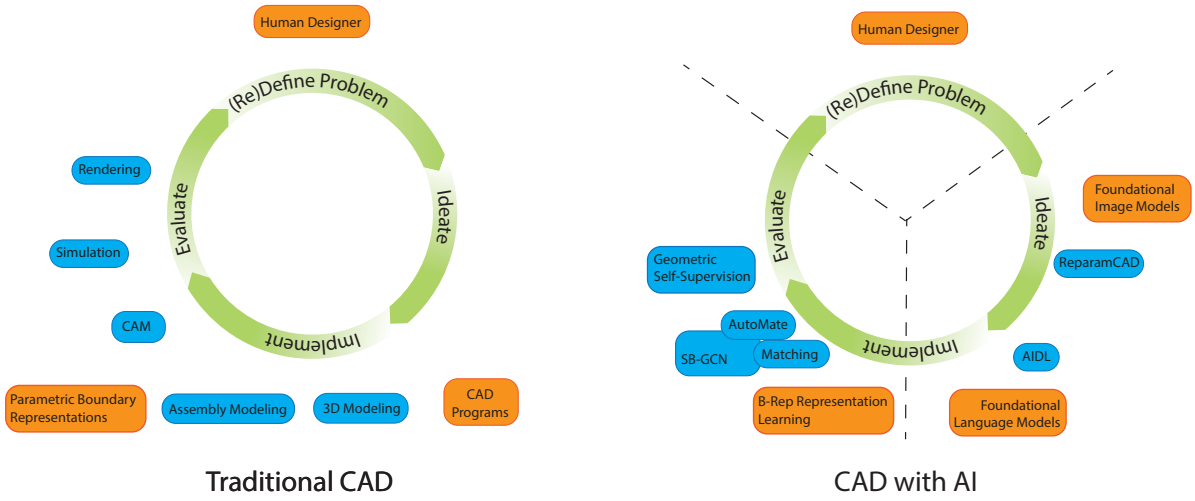


Figure 2.2: The design loop with CAD as it is, and with the projects that comprise this thesis.

Models are used to improve ideation by discovering semantic parameterizations of existing CAD models (ReparamCAD), and I end by exploring the design of a new CAD DSL designed to work well with general-purpose large language models as a co-pilot (AI Design Language: AIDL). These explorations hint at a future of AI augmented CAD that keeps a human designer in control of the most important step of design, discovering and refining the correct problem to be solving.

Chapter 3

SB-GCN: B-rep Representation Learning

3.1 Introduction

Established commercial CAD systems are built around parametric B-Reps. Concurrent work on learned models for B-RepS [WPL+20, LWJ+21] operate on a simplified *homogeneous* B-Rep structure and only learn representations for faces. However, parametric CAD operations can reference any kind of B-Rep topology, requiring a *heterogeneous* B-Rep representation. We propose *Structured B-Rep Graph Convolution Networks* (SB-GCN), a structured graph representation and message passing network for B-Reps that is the first heterogeneous learned B-Rep representation in that it learns embeddings for multiple classes of topological entity.

3.2 Related Work

Learning with CAD Data Recently, interest in data-driven approaches on parametric CAD data has increased with the creation of several CAD datasets. Koch et. al released the ABC dataset [KMJ+19], a collection of CAD models sourced from Onshape, to spur development of learning

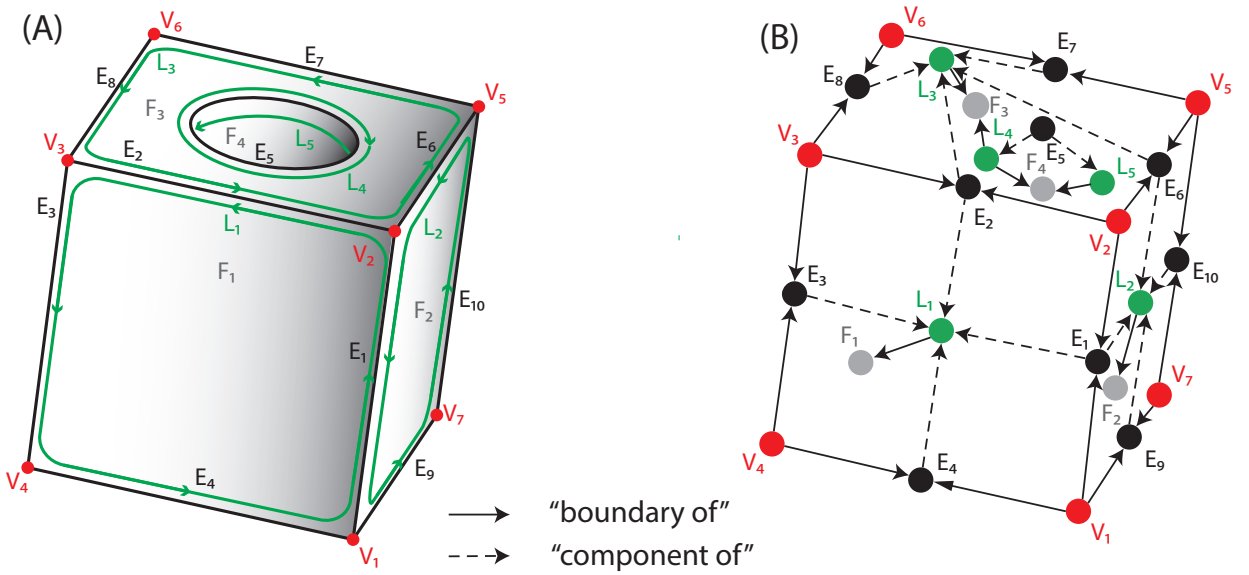


Figure 3.1: Our data model. Left: A simple part modeled as a B-Rep, with the topological entities (faces, loops, edges, and vertices) labeled. Right: Our graph representation of a B-Rep; a subset of the graph for the part on the left, limited to visible entities, is shown. The topological entities (graph nodes) are connected via boundary or component relations (graph edges). This graph has consistent structure; graph edges only go vertex-to-edge, edge-to-loop, or loop-to-face.

on CAD data. Sketchgraphs [SOZA20] is a dataset that collects 2D sketch information from Onshape. That work also presents an application in learning the constraint graph underlying these sketches. Recently, [WPL+20] compiled a dataset of sketch-extrude modeled parts for learning 3D construction sequences, and [CRH+20] proposed a method of generating CAD datasets with segmentation annotations, and a message passing network for segmentation. Concurrent to our work, [LWJ+21] and [JSL+21] propose message passing networks to learn B-Rep face embeddings. Similar to ours, these works use message passing graph neural networks to model the CAD data, but they only learn to embed one type of topology; sketches in the 2d case and faces in the 3d works. Integrating with general CAD system tasks requires representations for faces, loops, edges, and vertices, since all 4 classes of topological entity can be used as references. This means our graph representations are inherently heterogeneous, in contrast to the homogeneous graph representations used in prior works. These datasets and applications are centered around the part modeling part of the CAD workflow.

Graph Network Representation Learning Generic graph-based learning techniques have been applied to geometry, part assemblies, social networks, and other types of data. In this work we leverage these techniques to process CAD data by applying them to a graph structure that takes advantage of the structure of B-Reps. We refer the reader to two recent surveys for a more in-depth overview of the field, [WPC+20], [ZCZ20] and only reference the most relevant works here. Most modern works follow the Graph Convolutional Approach (GCN) of Kipf and Welling [KW17], which generalizes the Euclidean convolution to graph domains via a two-step message passing process: features of neighboring nodes are collected as computed messages, then these messages are aggregated locally using an order-independent aggregation function. Many techniques from CNNs have analogs in GCNs. Li et. al. [LMTG19a] showed that residual connections are crucial for training deep GCNs; we incorporate these into our network design

since GCN depth controls how large of a local neighborhood our learned representation for each topological feature can incorporate. A novel feature of graph domains that is pertinent to CAD data is domain heterogeneity. While basic GCNs handle structural heterogeneity of the domain, there is no natural extension of CNNs or RNNs that captures discrete typings of nodes and edges, which may have heterogeneous labels. [ZXK+18] introduced meta-paths, typed edges connecting all paths with the same node types, and proposed learning a separate message function for each type of meta-path. Learning heterogeneous messages has the disadvantage of greatly expanding model complexity. Our model incorporates both heterogeneous messages and meta-edges, but we leverage the consistent connectivity structure of B-Reps to avoid the increase in model complexity usually associated with typed messages by splitting these into a series of homogeneous message passing layers.

3.3 Methods

Graph Representation Learning for B-Reps B-Reps encode both the geometry and the topology of a shape as a collection of topological entities with associated parametric geometry. Faces, edges, and vertices are topological entities associated with parametric geometry functions of surfaces (2D), curves (1D), and points (0D). While the *parameters* (radius, normal, half-angle; see A.1 for a full list) of these functions are defined for each topological entity, the domain on which to evaluate each function is not. This information is encoded by the *topological* structure of the B-Rep, defining relationships between topological entities. Lower dimensional entities are related to higher dimensional neighbors by a *boundary of* relationship, and their geometry implicitly defines the domain bounds of these neighbors. A fourth type of topological entity, loops, consist of closed paths of edges. Edges are related to loops by a *component of* relation, and are themselves a boundary of a face. Loops have one parameter to specify if they are an inner or outer boundary

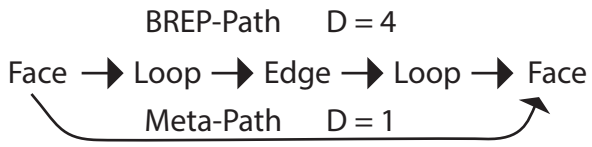


Figure 3.2: The B-Rep graph distance between neighboring faces is 4; with meta-paths, this is reduced to 1, shrinking the graph diameter by about 4 times.

of a surface. Figure 3.1 (A) shows a simple B-Rep with faces bounded by loops composed of edges bounded by vertices.

The *boundary of* and *component of* relations define a directed graph in which the topological entities are *nodes*, pictured in Figure 3.1 (B). This graph is *heterogeneous* in both its node types and relation types.[1] Recent and concurrent works [WPL+20, LWJ+21] have used a simplified, *homogeneous* version of this graph with only face nodes to build message passing networks for learning B-Rep representations. These representations are insufficient for learning how to mate B-Reps however, because B-Rep mates can be defined relative to topological entities of any type. Our representation learning must therefore handle the full, heterogeneous B-Rep graph.

Introducing additional nodes and relations presents additional challenges to representation learning. Having multiple types of bounds increases the path length between topological faces; for example, geometrically neighboring face nodes are at graph distance four. We would like our learned embeddings to capture boundary information and local structure, so we want a wide receptive field for our convolutions. In graph convolutional networks, the receptive field is a function of the number of convolutional layers, but as [LMTG19a] point out, state-of-the-art GCNs are typically only 3-4 layers deep as deeper graph networks are difficult to train. This would barely reach the neighboring face! (See Figure 3.2.)

We adopt two strategies to overcome this. First, we use residual connections in our network,

[1]We use *node* and *relation* to refer to *graph* vertices and edges throughout to avoid confusion with topological entity types of the same names.

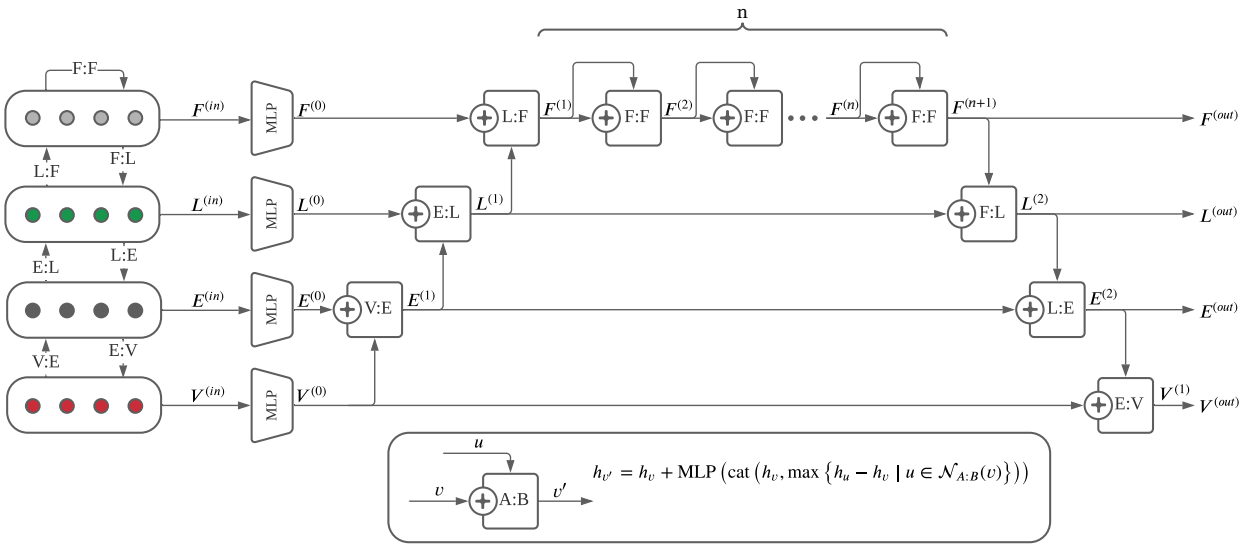


Figure 3.3: The SB-GCN architecture. B-Rep graph nodes are structured into 4 tiers by their topological entity type, with boundaries below the type they bound. Boundary information is passed up the tiers from vertices through to faces via residual MRGCN convolutions. Additional undirected MRGCN convolutions between faces quickly widen the network's receptive field, then the aggregated neighborhood information is propagated back down the tiers from faces through to vertices via more convolutions. Separating the convolutions into layers allows each convolutional layer to be homogeneous while still learning separate message passing functions for each type of graph relation.

which [LMTG19a] showed help stabilize the training of deep GCNs. We also add undirected meta-paths [ZXK+18] of (face←loop←edge→loop→face) relations to connect geometrically adjacent faces and reduce the graph diameter as shown in Figure 3.2. These match the undirected edges of [WPL+20].

Learning a different message passing function for each type of relation adds significant time and memory complexity to the model, and so limit the number of layers we can add and the size of our representation. The complexity of a B-Rep neural network is important to control because B-Reps range wildly in complexity, so a large and complex part can easily exhaust available GPU memory. B-Reps have a very regular connectivity structure that we use to partition the graph into a *Structured B-Rep Graph* that we can use to get the advantages of a wide receptive field,

heterogeneous graph network for less than the cost of an equivalent homogeneous network.

Since each type of node only has one type of relation, to exactly one other type of node, we can separate our B-Rep graphs into four *tiers* by node type, as pictured on the left side of Figure 3.3. We break our heterogeneous message passing into separate homogeneous message passing layers between adjacent tiers. We first pass messages up through the layers from vertices to faces to give each topological entity its boundary information. Then we perform several layers of message passing over just the face-face meta-paths to widen the receptive field, collecting neighborhood information at each face. Finally, we pass this neighborhood information backwards *down* the tiers from faces to vertices, so in the end, all topological entities have incorporated information about their boundaries and their local neighborhood.

Splitting the heterogeneous convolution into several smaller homogeneous ones saves a significant amount of memory (one sixth the number of parameters) without losing any information on the upward or downward passes. Messages passed from, for example, a loop to a face, can only contain incrementally more information until the information from the vertices at the bottom of the structure reaches the loop anyway, so it is more efficient to incrementally accrue that information in a single upward pass. We call our tiered graph network a *Structured B-Rep Graph Convolutional Network* (SB-GCN).

SB-GCN Formally, we define a Structured B-Rep Graph G as a collection of node sets, $N = F \cup L \cup E \cup V$, where F, L, E, and V are the face, loop, edge, and vertex nodes respectively. These nodes are then connected by directed, bipartite relation sets $V:E$, $E:L$, and $L:F$ and their transposes $F:L$, $L:E$, and $E:V$. For, say, $V:E$ the set contains relations r_{uv} that connects the nodes $u \in V$ to $v \in E$. There is also a set of undirected relations $F:F$ which are the meta-path relations between geometrically adjacent faces, computed by vertex-elimination on non-Face nodes in the graph. SB-GCN takes G and (possibly heterogeneous) input feature vectors on the nodes of G as

$N^{(in)} = V^{(in)} \cup E^{(in)} \cup L^{(in)} \cup F^{(in)}$ and produces output feature matrices $V^{(out)}$, $E^{(out)}$, $L^{(out)}$, and $F^{(out)}$ via a series of structured graph convolutions. Figure 3.3 shows the network architecture.

First a shared MLP transforms all input feature vectors into a common feature space prior to convolution:

$$N^{(0)} = MLP^{(in)}\left(N^{(in)}\right). \tag{3.1}$$

Next a cascade of residual graph convolutions are computed. We adapt the Max-Relative GCN (MRGCN) that [LMTG19a] found effective for training deep GCNs with residual connections:

$$\begin{aligned} \Sigma_v^{(l)} &= \max\left(\left\{h_u^{(l)} - h_v^{(l)} \mid u \in \mathcal{N}_{\tau(u):\tau(v)}(v)\right\}\right) \\ h_v^{(l+1)} &= h_v^{(l)} + \text{MLP}^{(l)}\left(\text{cat}\left(h_v^{(l)}, \Sigma_v^{(l)}\right)\right). \end{aligned} \tag{3.2}$$

This is computed for all $h_v^{(l)} \in N^{(l)}$ where $h_v^{(l)}$ is the feature of the node $v \in N^{(l)}$. $\tau(v)$ is an operator for $v \in N^{(l)}$ which returns which of F, L, E, or V that v is in, that is the topological type of the node v, and so $\mathcal{N}_{\tau(u):\tau(v)}(v)$ is the neighborhood of v using the relation set $\tau(u):\tau(v)$. $\text{MLP}^{(l)}$ is then the lth layer's MLP of a linear layer, batch normalization, and ReLU. The important difference in our convolution is that the adjacency of relations is determined depending on the layer, denoted by the subscript of the neighborhood operator. For the first three layers, relations are in the order of the B-Rep hierarchy: Vertex to Edge, Edge to Loop, and Loop to Face. Then, the Face-to-Face meta-relations are used for the inner k layers. Finally, for the last three layers, the relations are reversed from the original B-Rep relations: Face to Loop, Loop to Edge, and Edge to Vertex.

Input Features We use both the analytic geometry of each B-Rep topological element and a few computed geometric summary features as the per node input feature vectors for SB-GCN. The analytic features are a one-hot encoding of the parametric geometry function associated with each

node, plus a vector of each function's parameters. We exclude parameters for functions without a fixed parameter list size (such as B-Splines and NURBS surfaces) as well as geometry defined in reference to other geometry (like spun surfaces or intersection curves). See Appendix A for a full list parametric functions and their parameters. These more complex topological elements are never referenced for MCFs, so excluding their parameters is not too detrimental. We also exclude the origin parameter because we found that in practice it is frequently far away from the realized geometry once boundaries are taken into account and acted like noise.

To compensate for this lost localization feature, and give useful summarizing information about the more complex surfaces we exclude parameters for, we compute 4 additional geometric summaries. To localize each node in space, we compute its center of mass and an axis-aligned bounding box. We know that the size of each topological entity is a critical feature to determine which fit together, so we explicitly compute surface areas and arc lengths and add those features. Finally, we compute the moment-of-inertia tensor to summarize the geometry of more complex topological entities. We tested adding PointNet derived features to each entity, but found these to be unhelpful given the very low sample count per topological entity, and so do not include them.

Prior to extracting analytic and geometric features, we normalize the part geometry by translating both parts to the origins of their respective coordinate systems, and applying a consistent scale factor to both, so the largest dimension of either part has unit length.

Implementation We use Parasolid [Sie] to parse B-Rep files, which we convert to graphs as described above. Our GCN is implemented in Pytorch Geometric [FL19b].

3.4 Conclusion

SB-GCN was the first heterogeneous graph network for B-Reps capable of learning embeddings for many types of topological entities. It is designed to represent the core building block of all B-Rep-based CAD software: topological entities. Because of this, SB-GCN is applicable to a wide range of CAD tasks. The following two chapters will present case studies to prove this. Chapter 4 describes AutoMate, a recommender system that augments a commercial CAD system with suggestions for how parts connect into functional objects. This was the use case that originally motivated the development of SB-GCN. Chapter 5 tackles referencing on a more general level by learning to find corresponding references between similar parts.

Chapter 4

AutoMate: Learning to Assemble

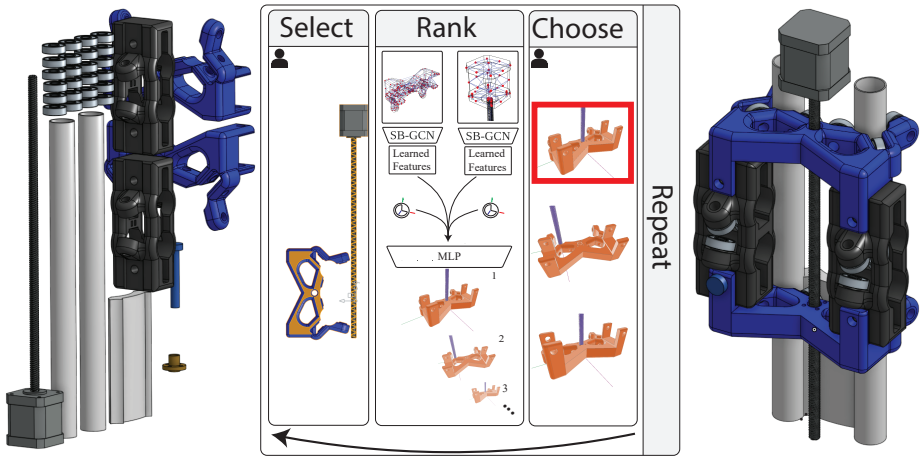


Figure 4.1: Our tool assists a designer in assembling a collection of CAD parts (left) into a functional assembly (right). First user selects a pair of parts to mate, and indicates roughly where the parts should be mated by where they click on each part. SB-GCN then creates embeddings for each topological entity of the selected parts' B-Reps, which are used as input to a classifier that scores potential mates defined in reference to these topological entities. Our system presents a list of suggestions that the user can pick from. This cycle is repeated until a working mechanical assembly is realized.

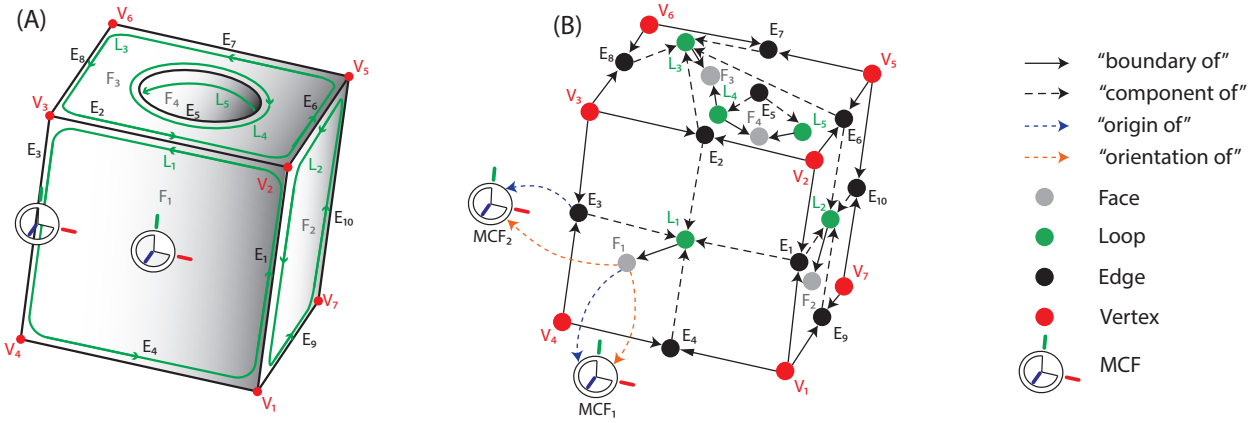


Figure 4.2: Our data model. Left: A simple part modeled as a B-Rep, with the topological entities (faces, loops, edges, and vertices) labeled. Also shown are two mating coordinate frames, located at the centroid of F_1 and the center of E_3. Right: Our graph representation of a B-Rep; a subset of the graph for the part on the left, limited to visible entities, is shown. The topological entities (graph nodes) are connected via boundary or component relations (graph edges). This graph has consistent structure; graph edges only go vertex-to-edge, edge-to-loop, or loop-to-face. Mating Coordinate Frames are defined by reference to topological entities for their location and orientation.

4.1 Introduction

The first case study that demonstrates the effectiveness of SB-GCN for CAD modeling tasks is AutoMate; a modeling assistant that helps designers to model the functionality of objects. Manufactured objects are designed in CAD systems as individual mechanical parts that are assembled into functional objects. In CAD, creating an assembly from parts is a time-consuming manual process of *mating* the parts, which requires both careful positioning of parts in reference to one another, and specification of how these parts can move relative to one another. In practice, users spend roughly one third of their time in modern CAD tools in assembly work [Ons20]. In this work we develop novel machine learning techniques for predicting how to mate parts. We further integrate these predictive models into an existing commercial CAD product [PTC], making it easier to create mates.

While many prior techniques use machine learning in assembly-based 3D modeling tools [SSK+17, ZXC+18, YCC+20a, HZF+20, LAK+18, WWL+19, MGY+19b, MGY+20], these techniques are not suited for a CAD workflow. Established commercial CAD systems are built around boundary representations (B-Reps, Figure 4.2), a data structure that represents the structure of a shape as *topological entities* of different spatial dimensions (2D faces, 1D edges and loops of edges, and 0D vertices), and explicitly captures the *analytic* geometry of each as a parametric equation, as well as the relations between topological entities. B-Rep based CAD systems model assemblies as a collection of pairwise constraints between parts called *mates*, where the constraints are defined relative to the topological entities. Using an analytic shape representation is crucial for the high modeling accuracy required for fabrication and integration with computer aided manufacturing (CAM). Modeling as a system of constraints enables capturing complex part interactions of mechanical motion, and allows the model to adapt globally to local changes. Defining these constraints from the topological entities allows mates to have analytic precision, as well as adapt to some parametric changes in geometry. In contrast to these CAD native assemblies, many existing assembly modeling tools use segmented meshes or point clouds to represent geometry, and predict world-coordinate, often static positions to construct assemblies, making them ill-suited for integration into the CAD design process.

While some work uses pairwise constraints [HZF+20, LAK+18, MGY+19b], to the best of our knowledge no existing assembly modeling systems create topologically defined mates, nor do any operate on B-Rep CAD data. Two key challenges impede the development of such a system; the lack of B-Rep learning models suitable for mating, and a lack of data for training such a system. As discussed previously, concurrent work on learned models for B-Reps [WPL+20, LWJ+21] operate on a simplified *homogeneous* B-Rep structure and only learn representations for faces. B-Reps can be mated relative to any class of topological entity, so the mating problem requires a *heterogeneous* B-Rep representation. Compiled B-Rep datasets [KMJ+19, WPL+20, LWJ+21] are focused on B-Rep

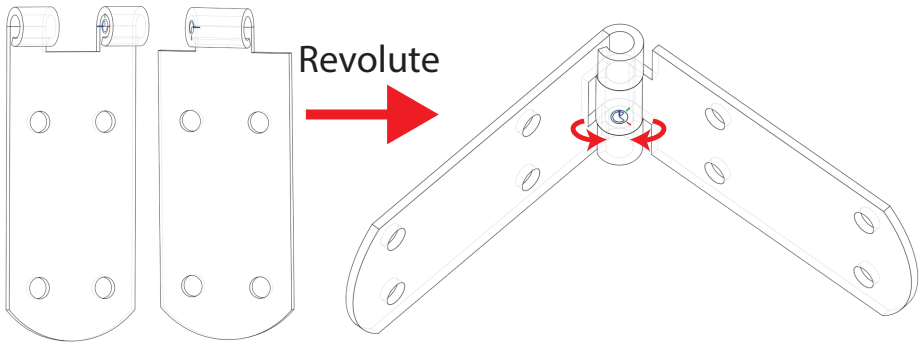


Figure 4.3: In a mate, the references to topological entities specify the location constraints, and the mate type (revolute) determines the degrees of freedom.

modeling and do not contain any mate data, and assembly modeling datasets [MZC+19] do not have B-Rep data or mates, so no suitable dataset exists for training such a model.

The need to represent multiple kinds of topological entities was the initial driver behind the development of SB-GCN. In order to train a useful system, we scraped a large collection of raw B-Rep assemblies, and cleaned it by deduplicating mates and assemblies and converting to a canonical format, to create the first B-Rep assembly and mating dataset, which we are making publicly available. We apply SB-GCN to B-Rep mating by modeling mates as translational and rotational constraints between *mating coordinate frames* (MCFs) – coordinate frames on B-Reps that inherit their origin and alignment by reference to separate topological entities on each B-Rep (see Figure 4.3).

We demonstrate that our approach is directly applicable to existing commercial CAD systems by implementing a mate recommendation system as an extension to the Onshape CAD system. Our mate model maps directly onto 180,102 of the mates in our dataset, which we use to train a ranker for potential mates and a classifier for mate degrees of freedom. Our model suggests a verifiably correct mate location 72.2% of the time on this data, significantly outperforming traditional discrete geometry approaches (46.3%).

4.2 Related Work

Assembly-based Modeling Assembly-based, or compositional, modeling tools create 3d models by aggregating parts from 3d shape collections. Modeling by example [FKS+04] opened this field with two main subproblems; how to choose which parts to combine and how to combine them. The first problem is commonly solved by text search [FKS+04], similarity search [FKS+04, CK10], or consistent segmentation [CKGK11]. In our work, we assume that retrieval is solved and focus solely on the second problem of mating parts.

Traditionally, aligning parts has been performed analytically; using iterative closest points [FKS+04, SBSCO06], surface registration [CK10], or energy optimization [GvK18]. Newer data-driven methods learn to align shapes. In contrast to our work, most approaches [YCC+20a, HZF+20, SSK+17, JBX+20a, WZX+20, MWYG20, YML+20] predict or generate global offsets for each part rather than pairwise constraints. Assemblies defined by global positions cannot handle replacement of parts, parametric variation of parts, or modeling degrees of freedom. Those that do model assemblies by pairwise positioning [ZXC+18, LAK+18, WWL+19, MGY+19b, MGY+20] predict or generate explicit offsets and rotations, forfeiting the precision and adaptability to parametric changes that our system acquires by defining these relative to topological features.

Another limitation of these methods is that they work with non-parametric surface representations (meshes, pointclouds, voxel grids) or bounding boxes, and encode the geometry using a geometric learning model such as PointNet [QSKG17], PointNet++ [QYSG17], PCPNet [GKOM18], or DGCNN [WSL+19]; we refer the reader to Bronstein et. al [BBL+17] for a survey geometric learning methods. Unlike these techniques, our method uses B-Reps as input which provides additional CAD context and enables it to interface with CAD systems.

Our work overcomes limitations of existing assembly modeling methods in the CAD context. Existing methods are trained over categories of shapes, limiting their usefulness for general design

problems. Since we learn local representations on the underlying part structure, we can avoid dependence on global part context. Additionally, none of these systems model the degrees of freedom of connections. Fab-by-example [SSL+14] is the closest existing work to our system in capabilities; its pairwise connections are parametrically defined to adapt to parametric variation, it models the degrees of freedom of connections, and is not limited to a single category. However, it relies on a database of hand-annotated parts and so does not generalize to arbitrary CAD data.

Learning with CAD Data The previous chapter described how existing data-driven approaches to parametric CAD data have used a simplified model that ignores most kinds of referenceable topology in favor of only faces. Our mating task requires us to have representations for faces, loops, edges, and vertices, since all 4 classes of topological entity can be used to mate parts. This means our graph representations are inherently heterogeneous, in contrast to the homogeneous graph representations used in prior works. Existing datasets and applications are centered around the part modeling part of the CAD workflow [KMJ+19, SOZA20, WPL+20, CRH+20, LWJ+21, JSL+21]. An assembly database, PartNet [MZC+19], does exist, and was used to train several of the assembly modeling systems mentioned earlier, but it is based around mesh geometry, not CAD models. Ours is the first dataset and application that captures assembly information - the relationships between parts.

4.3 Overview

Modeling Mates Our predictions are generated by scoring and ranking a collection of potential mates. We model a mate as a pair of *mating coordinate frames* (MCFs), a coordinate frame defined for each mated part in reference to their topological entities (such as faces or edges), and a constraint on their relative position and orientation. For example, the two halves of a hinge in

Figure 4.3 are mated by MCFs defined in reference to cylindrical faces. The pair of MCFs, one on each part, define the *mate location*, assembling the parts by aligning the frames. The constraint between MCFs is specified as one of eight *mate types* (Figure 4.4), which specify which axes the frames can be translated or rotated about relative to one another. For example, Figure 4.3 shows how a hinge is modeled by a *revolute* mate that disallows translation of the parts, but allow rotation around their common z-axis, letting the hinge swing open and shut.

Each MCF is defined as a tuple of two topological entities from the associated part. One referenced topology defines the origin of the MCF. Some topological entities have more than one location to anchor an MCF to (such as the top, bottom, or middle of a cylinder), so this reference has an associated *origin type* to specify where on the entity to place the origin.[1] The other referenced topological entity determines the frame's orientation by specifying its z-axis, for example, normal to a referenced planar face. The other coordinate axes are derived from the frame each part was modeled in. See Appendix A.1 for specifics.

This model is simplified from the raw dataset. We discarded a small number of mates that were defined with additional rotational and translational offsets to their MCFs. We also ignore optional per degree-of-freedom range limits, as these are also rare. The raw data we collected also allowed for the MCFs in a mate to be reoriented by one of 8 orthogonal rotations, and for one of the same 8 rotations to be applied to the mate as a whole, giving 512 possible combinations. We canonicalized these down to 8 unique variations for our dataset, and for the models in this paper we limit ourselves to the most common *canonical orientation*.

Interactive Recommendation System Our method extends a CAD system to provide autocomplete-like suggestions for mates. Automatically suggesting mates is a balance between automation and control from the user. The problem of deciding how to mate B-Reps is inherently ambiguous; it

[1] A complete list of origin types is given in Table B.1.

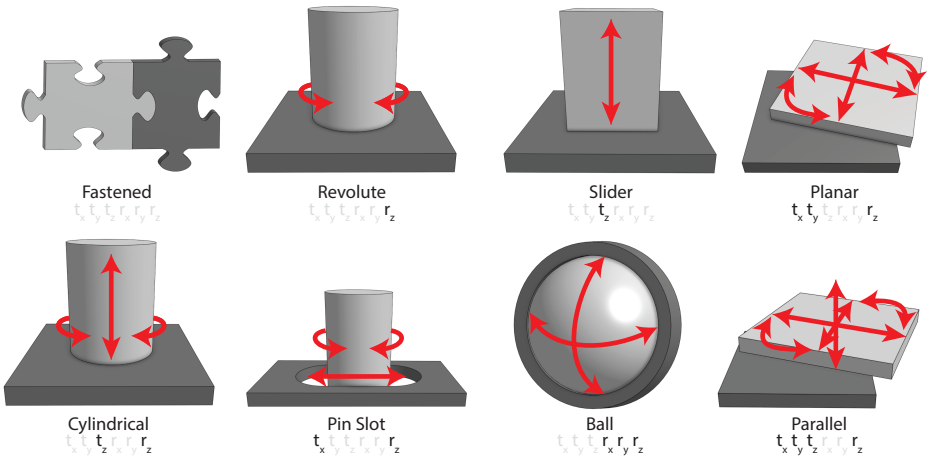


Figure 4.4: The 8 mate types. Red arrows illustrate the available degrees of freedom for each mate type.

depends on the user's design intent, so there is no ground truth. We decided to structure our tool as a recommendation system. In our system, the user clicks on two parts to select them for mating. We use the face that the user clicked as an indication of design intent; the origin of each MCF will be in the neighborhood of the selected faces. We accomplish this by using the selected face as the orientation reference for each MCF, and restrict our search to MCFs with origin on that face or one of its boundaries. This both captures the user's intent and reduces the cost of the search, and is well-supported by our dataset; almost all the origin references in our dataset are within the boundary of the corresponding orientation topological element. Given this input; two B-Reps and face selections on each, our system scores and ranks all pairs of MCFs neighboring the selection, and reports the top 6 to the user. Given a particular MCF pair, our system can also predict the most likely mate types. Figure 4.5 shows our interface for the simple case of two building blocks. The user has selected the top and bottom face of each block to be mated within the system's normal assembly UI. Whenever two faces are selected, our system presents 6 mate location options that the user can choose between with a hotkey, or ignore to continue assembly manually without the autocomplete impeding their workflow. If the user selects a mate suggestion, a *fastened* mate is

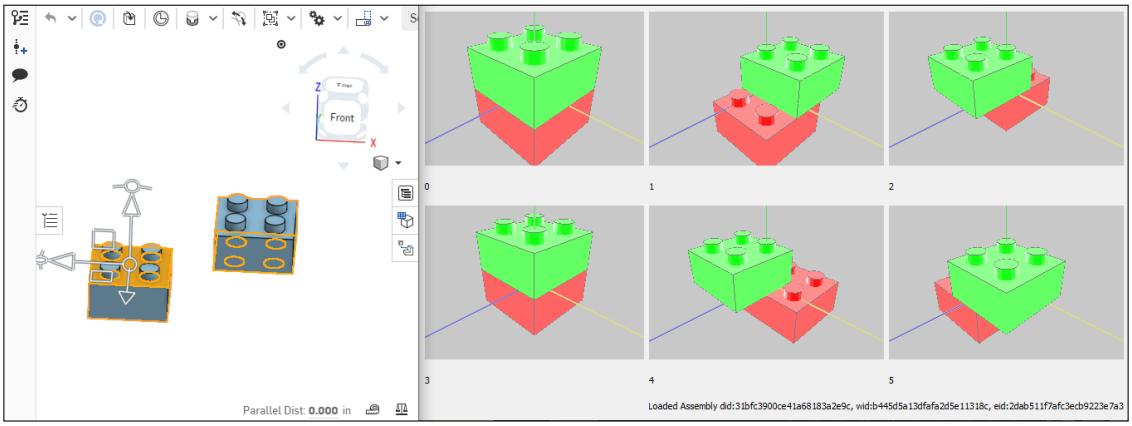


Figure 4.5: Our Onshape Extension. Left: Onshape assembly with two parts selected. Right: Top 6 predicted mate locations. The first and fourth suggestions look similar because differently defined mates can resolve to the same location; the first option is centered on middle of the blocks while the fourth is centered on the front peg.

inserted between the parts, and the user can then select a mate type. Our system also ranks the mate type likelihood conditioned on the chosen mate location.

Predicting Mates We score and classify mates using SB-GCN, our graph neural network for B-Reps. This message passage network is applied to each input B-Rep to produce representations for each topological entity that encode the local neighborhood, as well as a global descriptor for each part. Each MCF is encoded by the representations of each referenced entity, the origin type, and the global part descriptor, then we train a classifier to score each MCF (the *mate location task*), or classify its mate type (the *mate type task*). We train this model over all compatible mates from our dataset. Separating the B-Rep representation learning from the mate classification model is important because while each CAD system has its own constraint model for mates, most use B-Reps as their underlying shape representation and construct their mate by reference to topological entities. By explicitly learning to represent each type of topological entity, our system can be adapted to integrate into other CAD systems.

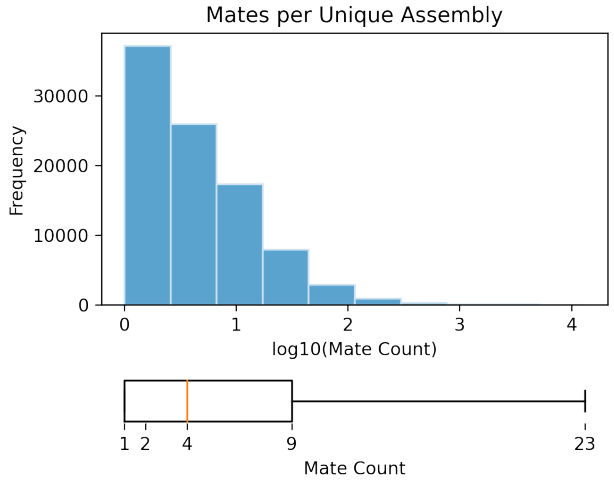


Figure 4.6: Number of mates per assembly in the dataset. Top: histogram of log assembly size. Bottom: Box-plot of assembly sizes with 10th, 25th 50th, 75th, and 90th percentiles labeled.

4.4 Dataset

We scraped all publicly available documents from Onshape using their API. We collected 255,213 assemblies with at least one mate, totaling 3,989,164 mates. However, a significant number of these are complete or partial duplicates created by reuse of common hierarchical sub-assemblies, and by copying and modification of assemblies in this public repository.

To clean this dataset, we first expanded and then flattened all sub-assembly references, and filtered down to only top level assemblies. We also removed all incomplete mates, and pairs of parts with multiple mates between them. To identify identical parts and assemblies, we created a B-Rep fingerprint from the number of each type of topological entity and the moment of inertia tensor and center of mass of the represented solid. We used this to deduplicate re-used parts, and to identify duplicate mates by MCFs, mate types, and mated parts. Assemblies were deduplicated by their mates. The dataset contains 92,529 unique top level assemblies. These unique assemblies range in size from 1 to 13183 mates, with an average of 12, and a 90th percentile of 23. Figure 4.6 shows the distribution of mates per assembly.

Table 4.1: Mate type frequencies.

Mate Type	Frequency
FASTENED	62.1%
REVOLUTE	12.7%
PLANAR	11.8%
SLIDER	5.3%
CYLINDRICAL	5.1%
PARALLEL	1.8%
BALL	0.6%
PIN SLOT	0.5%

At the mate level, we have 541,635 unique mates. Table 4.1 gives the frequency of each mate type among unique mates. Our dataset contains a total of 376,362 unique B-Rep models used in mates.

4.5 Methods

Predicting Mates with SB-GCN We apply SB-GCN to solve two mating tasks: *location prediction*, which scores and ranks pairs of MCFs adjacent to selected faces on two parts, and *mate type prediction*, which classifies the mate type for a mate with a specific pair of MCFs. We use the same classification network with a different output head for both tasks, shown in Figure 4.7. Our classifier uses SB-GCN in a Siamese configuration over both input B-Reps to learn an embedding for each topological entity. A global part feature is computed for each part with a mean-pool of the these embeddings. We then construct a feature vector for each MCF in the mate by concatenating the embeddings of the MCF's orientation and origin topological entities, a one-hot encoding of the origin type, and the global part feature. These MCF features are concatenated and fed to an output MLP for either location prediction (one output) or mate type classification (8 outputs).

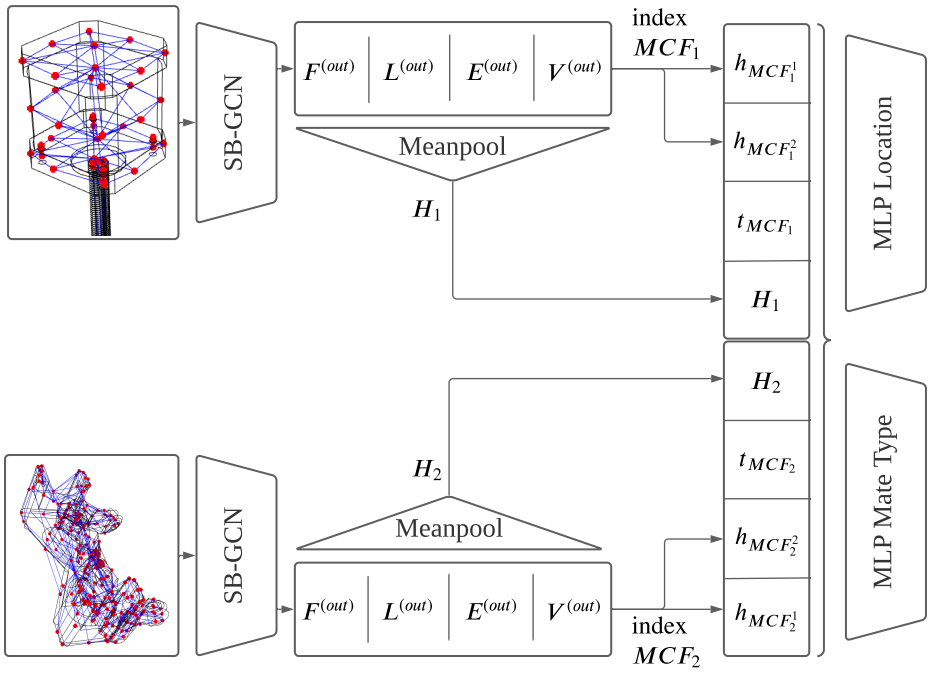


Figure 4.7: Our classification network. SB-GCN is used to learn embeddings for every topological entity of each part. The embeddings of the topological entities defining each MCF's origin and orientation (written as the topological entities MCF_i^1 and MCF_i^2 respectively, and thus with embeddings $h_{MCF_i^1}$ and $h_{MCF_i^2}$) are concatenated with a one-hot encoding of the MCF's *origin type*, t_{MCF_i}, and a global part feature, H_i, computed via a mean-pool, to construct MCF features. Each parts' MCF features are concatenated and then classified or scored with an output MLP, depending on the task.

Dataset Our training data is derived from an earlier version of the dataset described in Section 4.4, which has similar statistics, but fewer overall mates. We also applied further filters; we removed mates with overly simple or overly complex geometry (measured by face count) to weed out basic primitives and remove large part outliers that could overwhelm GPU memory.

We also augmented our dataset by finding alternative ways to construct equivalent MCFs. Figure 4.8 illustrates several ways of constructing the same MCF for a simple part using two different orientation topology selections, as well as other MCFs that can be constructed from those face selections. We augment our dataset by adding mates using all equivalent MCFs as positive

examples for mate location, and include MCFs that use the same orientation topology but are not equivalent to construct negative examples. These are grouped by selected topology on each part to create selection examples we train and test on for location. For mate type, we only use the augmented positive examples. Since our UI only supports selecting faces, we remove selection examples that aren't based on a face selection. We apply one final filter to remove MCFs which are not canonically oriented, as described in Section 4.3, and selection examples with over 10,000 potential mates to limit per-batch memory requirements. After augmentation and filtering we have 267,385 selection examples to train over.

Training All models were optimized using the Adam optimizer with a learning rate of 0.001 and betas of (0.9, 0.999). For mate location, we computed an augmented true positive set as described above, labeled as positive examples, and for each compatible potential user selection, all other compatible mates labeled as negative examples. We trained our output classifier with a binary cross-entropy loss, weighted batch-wise. For mate type, we similarly computed an augmented set of true positive mates, labeled with the ground truth mate type (since we are conditioning this model on the mate location), and then trained with cross-entropy loss. Model selection was performed using the average of the hit ratio at k: over $k \in [1, 10]$ for location and over $k \in [1, 8]$ for mate type.

Implementation We use Parasolid [Sie] to parse B-Rep files, which we convert to graphs as described above. Our GCN is implemented in Pytorch Geometric [FL19b]. We implemented our Onshape integration as a browser extension that captures the user's inputs to Onshape, and uses the Onshape API to download B-Reps and assembly definitions and to create mates. For SB-GCN we used $n = 6$ inner layers, and 64 neurons for every linear layer. 64 neurons were used again for the final predictor MLPs' linear layer.

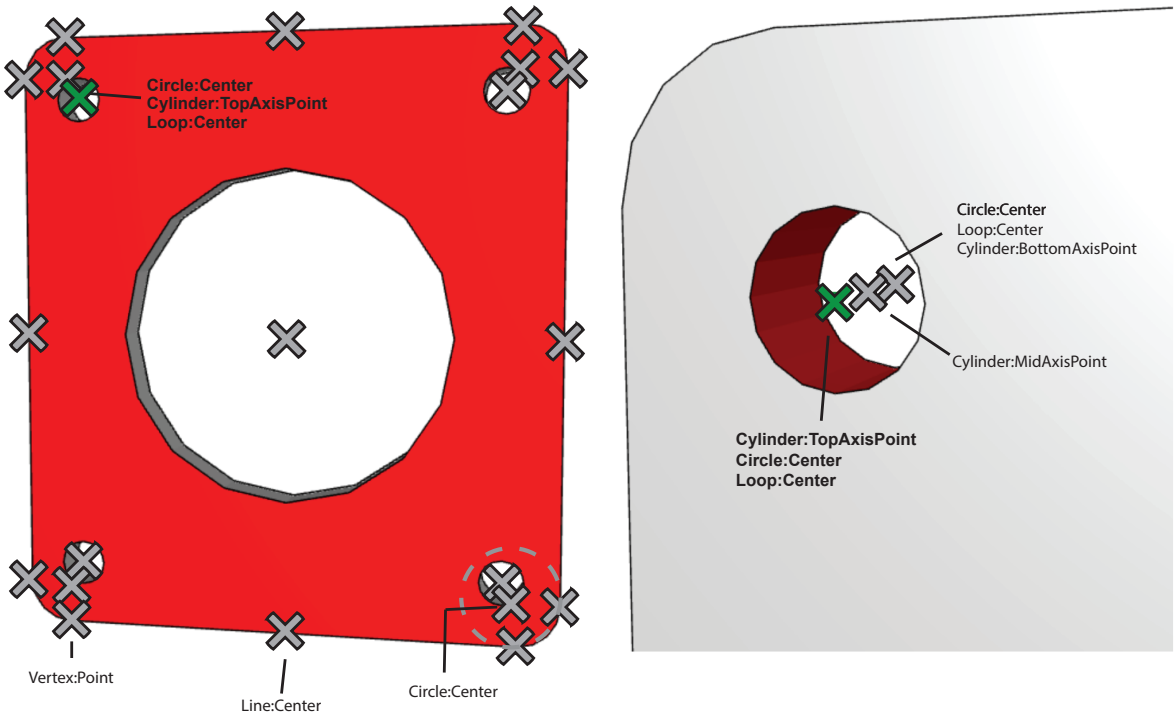


Figure 4.8: MCF data augmentation. The red face is the selected orientation topology. Green Xs are MCFs equivalent to the ground truth, and grey Xs are other MCFs that can be constructed from those selections, which are used to construct negative location examples. Selected MCFs have been annotated with their location topology and origin types. Note that there are multiple locations aliasing the ground truth MCFs for both selections.

4.6 Results

Tasks and Metrics We evaluated our system's performance in two tasks; predicting the exact mate location conditioned on the user's part and face selections, and predicting the type of a mate conditioned on its ground truth location. Our system is structured as a recommendation system, so as our primary evaluation metric we use hit ratio at k; the likelihood of presenting a correct solution to the user at various depths (k) of recommendation list. For the mate location problem in particular, our system as-implemented presents 6 suggestions to the user, so we use this threshold to define our overall system accuracy. We also define the NDCG* as the average inverse log rank of the first correct suggestion to compare methods with a single number. This is a lower-bound approximation to the Normalized Discounted Cumulative Gain commonly used to evaluate retrieval systems, differing in that we don't award additional gain for correct predictions past the first.

4.6.1 Baselines

We evaluate our model against two types of baselines; pointcloud-based deep learning techniques that don't use CAD data, and simple ad-hoc heuristics.

Point Cloud Baselines We chose to use point clouds as our comparison with discrete geometric representations because they are common in recent assembly modeling work [SSK+17, YCC+20a, HZF+20, LMS+20, MWYG20, YML+20, MGY+19b]. We evaluate against three popular methods; PointNet [QSKG17], Pointnet++ [QYSG17], and dynamic graph cnn (DGCNN) [WSL+19]. We structure these predictions following the Siamese network design of ComplementMe [SSK+17], using a different output head for each task. Mate type learns an eight class classifier similar to our network, whereas mate location adopts the strategy of existing assembly modeling works

and uses regression to predict an offset for each part's MCF; potential mates are then ranked by their MCFs' distance from these predicted offset, effectively snapping to the nearest potential mates. In order to give these algorithms the benefit of the user's face selection, we pre-rotate both parts into their final orientation and center and align the user's selection at the origin for the mate location task; for the mate type task, we transform the pair of parts into their final correct positions, re-centered with the mate at the origin aligned with the first part's mating coordinate frame. We expect that these methods will not do as well as ours because they do not have access to precise analytic geometry of the B-Reps, and because they allocate their representation power across the geometry, whereas our method learns representations local to the topological entities on each part that will be interfacing. These models are also much larger than ours; they range from 834k parameters for PointNet to 1.5M for PointNet++, compared to only 126k for our model.

Ad-Hoc Baselines Our ad hoc baselines are task specific. For mate location we have three; *Random*, *Origin Type*, and *Snap to Selection*. *Random* is simply a random ranking of potential mates to get an expected performance floor. *Origin Type* ranks potential mates based on the pair of origin types for their MCFs, in order of the frequency we see such pairs in our dataset. *Snap to Selection* ranks mates by the distance to the center of mass of the user's selected faces; it is intended as a lower bound of the regress-and-snap approach without any regression. For mate type we have one ad-hoc baseline: *Label Distribution*. This always ranks mate type according to their frequency in our test set, ignoring the input.

4.6.2 Data Ambiguity

Location Ambiguities Mate placement is a subjective task that depend on the user's design intent and the context they are working within. Our dataset only contains the mates that were needed to create the objects that the parts were used for, but parts can be used in other ways

not seen in our dataset. In our quantitative metrics we evaluate strictly against the mates which appear in our dataset, so we asked a CAD expert to evaluate some of our system's prediction and indicate which suggestions were plausible uses of those parts. For all examples shown in the paper (Figures 4.11 and 4.16) we have indicated with a green checkmark all mates that the CAD expert selected.

Mate Type Ambiguities The mate type that a design uses is highly dependent on the designer's context. For example, our dataset over-represents *fastened* mates, but visual inspection finds gears marked as fastened rather than revolute; the user who created that assembly was not concerned with modeling the gear motion. To quantify the level of ambiguity on our dataset, we recruited seven expert CAD users to label 96 mates (12 of each mate type as labeled in our dataset) with what mate type they thought it should be. Overall, there was a majority consensus among the CAD users for 70 out of the 96 mates, which suggests we can expect to an upper bound of roughly 73% for top suggestion accuracy. Figure 4.9 shows an example of a mate that our experts gave three different labels to, and the dataset gave a fourth. We further quantified agreement using pairwise Cohen's κ, a 0-1 measure of agreement above random chance pictured in Figure 4.10. We measured agreement among the CAD experts, between the CAD experts and our dataset, and between our dataset and our model predictions. This analysis showed that while our CAD experts agree more with each other than with the dataset labels, our model's predictions agree with the dataset to the same degree as the median CAD expert, indicating that it achieves human-level performance.

4.6.3 Mate Location Prediction

Figure 4.11 shows a selection of location predictions from our system across a variety of complexities from a dozen possibilities to several thousand. Our model has learned to align key features

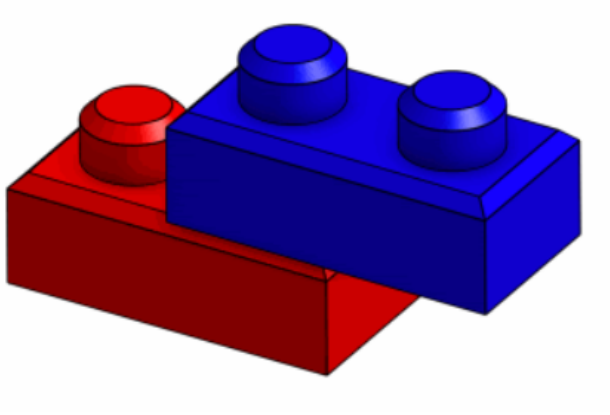


Figure 4.9: Example of an ambiguous mate. The survey participants were divided between cylindrical (blocks can be rotated around the peg and slid off the peg), slider (blocks can be slid off the peg), and fastened (blocks should stay in place). In the dataset, this is classified as a parallel mate.

like holes and edges of similar sizes. In example (H) we see the effect of not knowing the mate type in advance; the boltholes and slots have a similar radius and our model suggests mates that could be either *revolute* (in the boltholes) or *pin slots* (in the slots). Examples (E) and (I) show that our model learns to align features but does not learn to avoid part overlap.

Baseline Comparisons Figure 4.12 illustrates the performance of our model in predicting mate location, as compared to several baselines. Our model outperforms all baselines, and achieves a 72.2% accuracy at our UI threshold of 6 predictions. The inference type baseline, our simplest model using CAD data, lags behind our system by roughly 15%, but still in turn outperforms all geometric baselines. While these geometric baselines do learn something; they perform better than a random selection baseline, they do not significantly outperform Snap to Selection, a model with no learning.

Scaling We evaluate how well our method scales with the number of potential mates to rank. We binned prediction results by the number of potential mates (without the merging present in

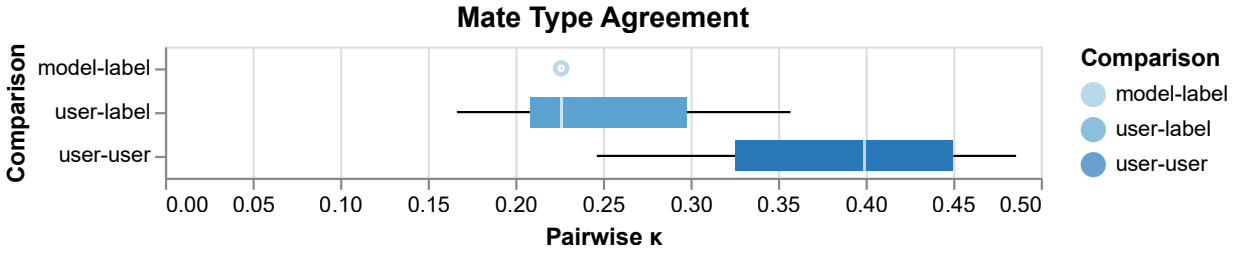


Figure 4.10: Mate type agreement, measured by pairwise Cohen's κ, between surveyed CAD users, our model, and the dataset labels. Our model agrees with the labels to the same degree as the median expert, indicating that we achieve human performance. The experts agree within themselves to a greater degree than they agree with the dataset labels, indicating that there are additional contextual clues for mate type not captured by just looking at a pair of parts.

Figures 4.11 and 4.16, so these numbers are higher than appears in those figures), then computed quantile statistics of the rank of the first correct prediction, which we plot in Figure 4.13. The median rank grows much less quickly than the number of possible mates, and in fact stays below our interface cutoff of six suggestions for six of the eight buckets, showing that our predictor can scale well to more complex problems.

Failure Cases Figure 4.16 illustrates some failures of our model. The far right column shows the first correct prediction and its rank. Examples (A)-(C) illustrate cases where additional context beyond the parts would be needed to correctly infer the mate location. In each of these cases, a third part interacts with the mated part. Our model did, however, pick alternatives that the expert CAD user validated as useful for some use case – just not the one seen in our dataset. Example (D) shows a common problem when parts have multiple locations that need to be aligned, our model does not always align all of them; our chosen locations align one, two, or three holes, but miss the one location that would align all four. In example (E), the walls as modeled are actually hovering above the floor, and so are not topologically close enough to be captured in MCF representations. Both of these examples suggest we should incorporate more extrinsic geometric features and

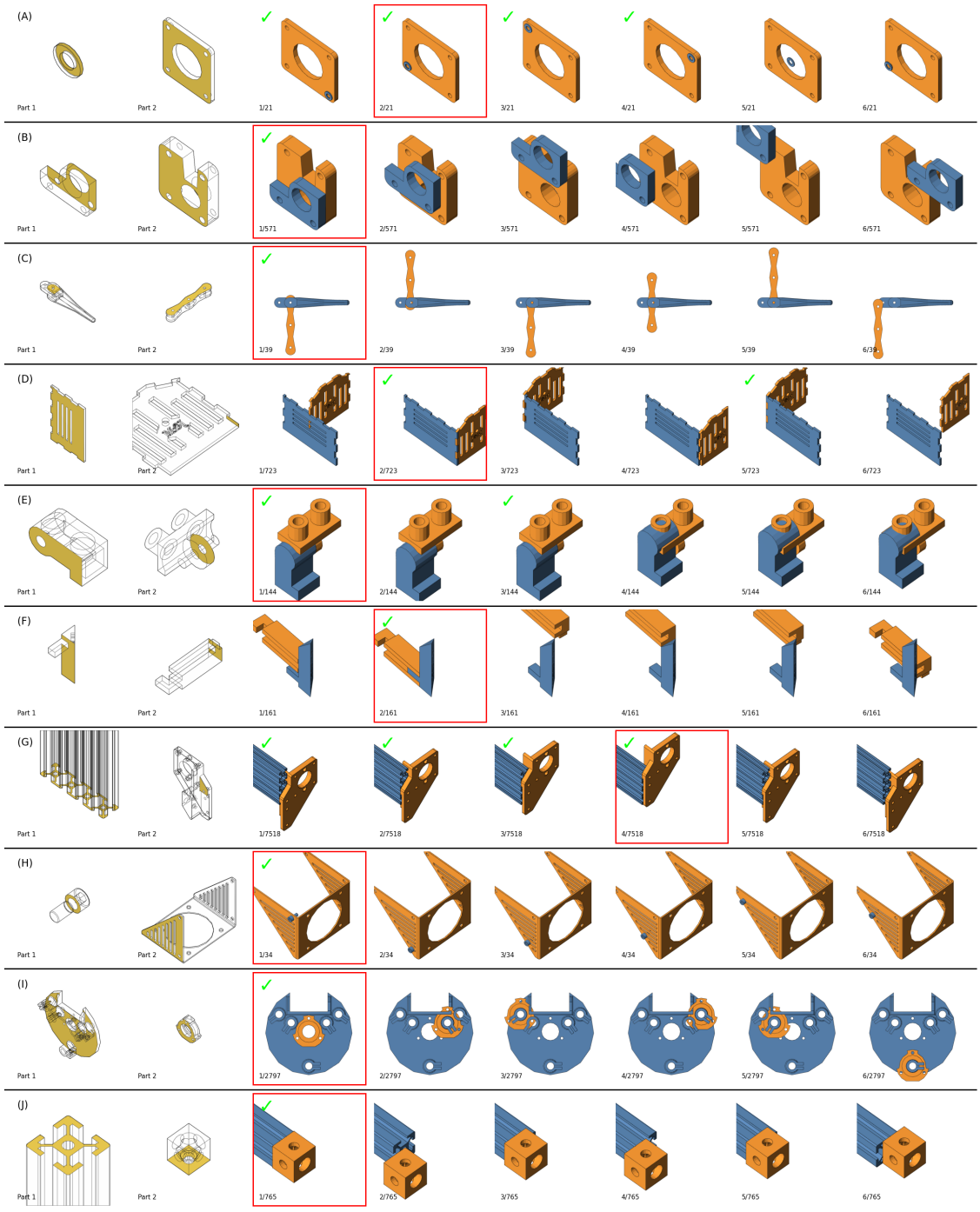


Figure 4.11: Location predictions. Mates labeled as correct in our dataset are boxed, and mates that a CAD expert judged as good prediction have a checkmark. Prediction rank is indicated below. For these examples, we have excluded suggested mate locations that are equivalent to an earlier mate given the mate type to better reflect user preferences, which affects both rank and number of possibilities compared to our quantitative results.

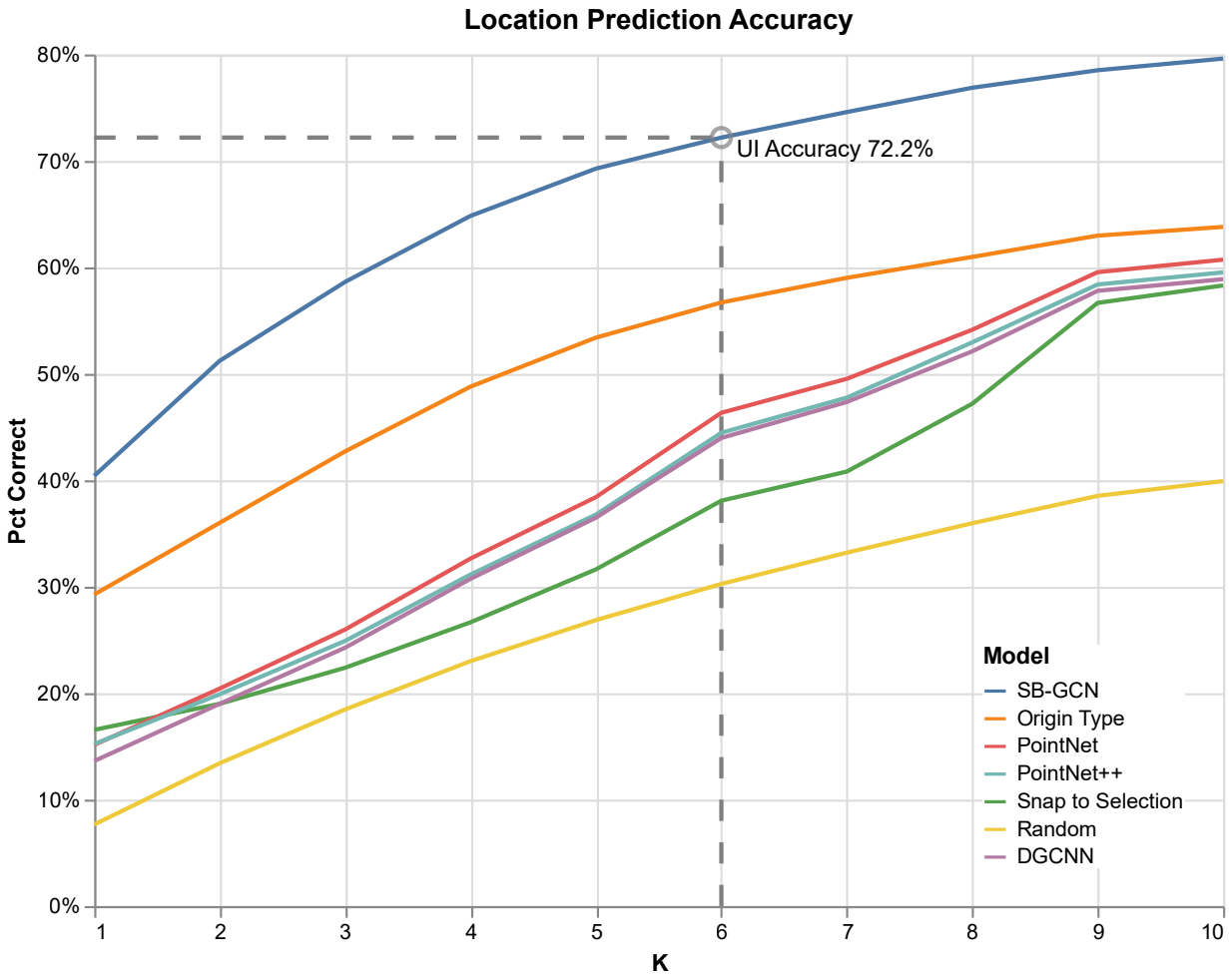


Figure 4.12: Our model compared against baselines on mate location prediction task.

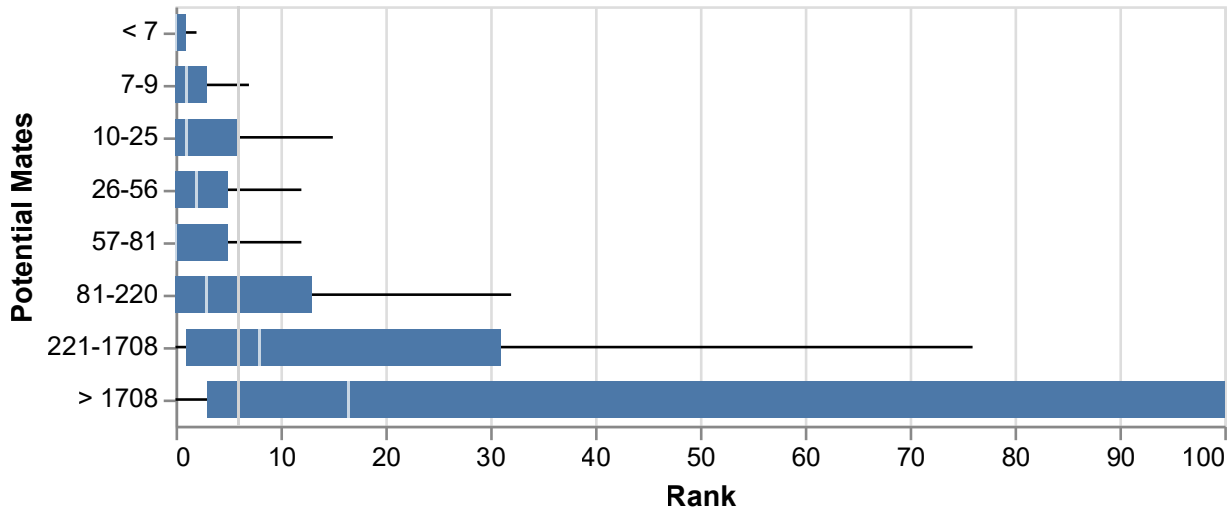


Figure 4.13: Rank of the first correct mate suggestion as the number of potential mates grows. Ranges selected to have roughly even mate counts (3500), except the final bin which has only 646. While variance does grow with the number of potential mates, the median correct prediction rank grows much more slowly, and only exceeds our UI cutoff of 6 selections (light grey line) above 220 possibilities. Outliers have been excluded from whiskers, and the whisker for the final bar has been truncated.

connectivity to account for mismatch between topological and geometric distances. In the final example, the horizontal board has no marks on its end, so our model attempts to match it with the midpoint and a similarly sized edge of the vertical one. The correct mate aligns the end of the horizontal board with a pre-drilled nail hole in the vertical board, and since it's a nail, there is no hole modeled in the end cap to suggest alignment.

4.6.4 Mate Type Prediction

Figure 4.14 illustrates performance against baselines for the mate type task. Our model outperforms all the baselines, though the gap is not as significant as in the location task due to the tighter bounds between the label distribution baseline (presumed lower bound) and our ambiguity-based estimate of 73% first suggestion accuracy upper bound. Having access to definitive analytic information, for example that the MCFs are centered on a cylinder and a circle indicating likely rotational degrees of freedom, gives our model an edge over the pointcloud based methods that must infer this from samples. The 73% upper bound estimate is very rough due to a small sample size, so our model's 70% top suggestion accuracy and alignment with the median human expert (discussed above, see Figure 4.10) leads us to believe our model is approaching the limits of accuracy given the dataset ambiguity.

4.6.5 Ablations and Design Decisions

Analytic versus Discrete Geometry Since we want to integrate with existing B-Rep based CAD software, and need the precision of analytic geometry to interface with CAM systems, any mate definitions we produce will need to be defined in reference to B-Reps. However, there is still a question of whether to use the B-Rep data directly for prediction, or to convert to a discrete geometric representation, like a point cloud or mesh, that is more commonly used in assembly

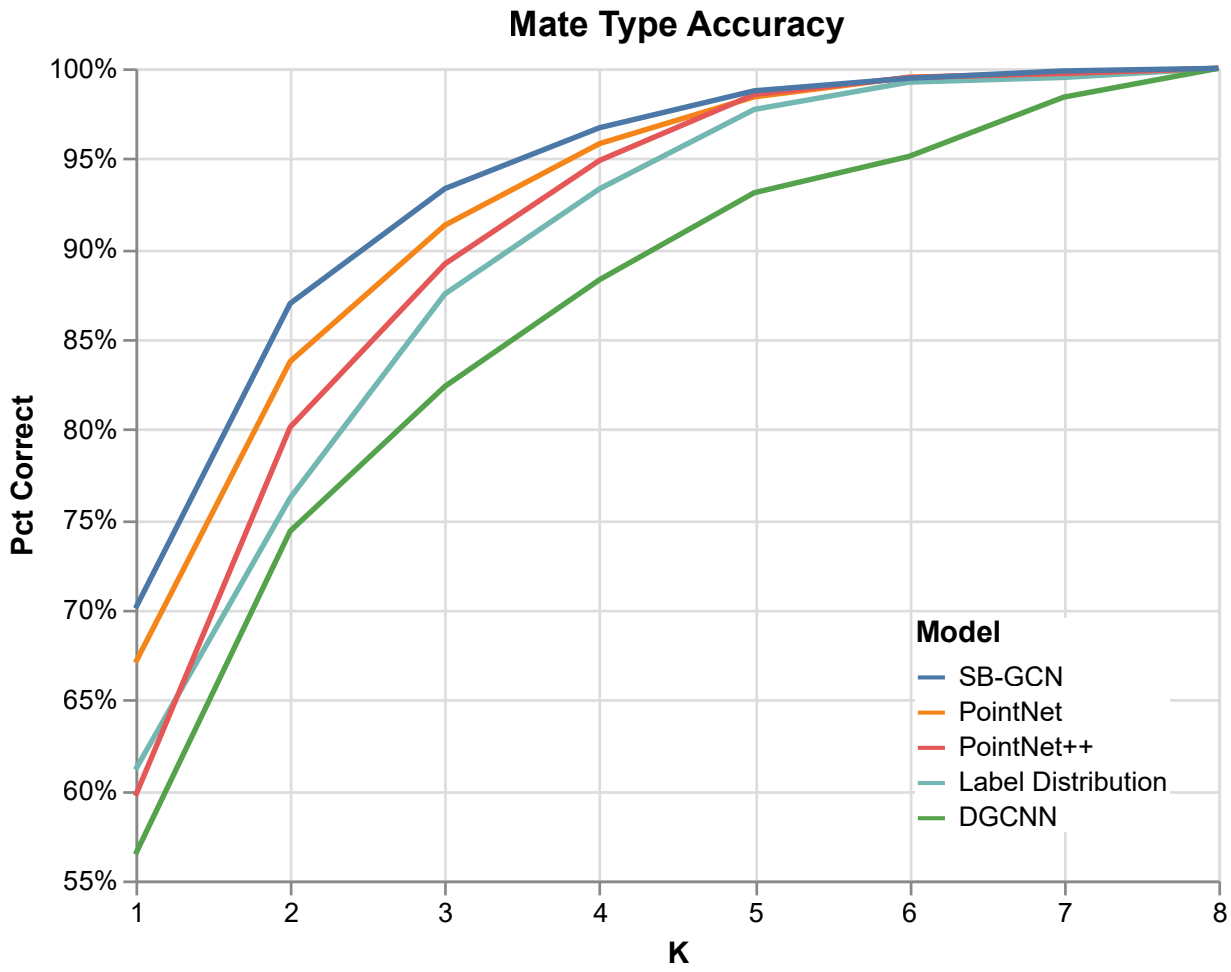


Figure 4.14: Our model compared against baselines on mate type prediction task.

modeling. In this alternative, we would predict relative offsets for each part, then project to the nearest B-Rep mate. Our baseline comparisons show that this approach does not work well for pointcloud based models that have previously been used in assembly modeling. To test the viability of the regress-and-snap strategy in general, we tried removing the learning component by testing against a noisy oracle that always predicts the true offsets, plus a controllable noise vector scaled to be a fraction of the standard deviation of offsets off all mates under consideration. Figure 4.15 shows that regress-and-snap is a losing strategy: accuracy drops at even very small amounts of output noise.

Feature and Network Selection To determine which features to give our network, we incrementally added layers of features; just parametric functions, parametric function parameters, topological entity size, and bounding box plus center of mass and moment of inertia tensor. We also experimented with several convolutional architectures. The simplest was our *plain* network, which does not use the structural B-Rep data at all, instead learning embedding features with a shared MLP across topological entities. Our most direct analog to [WPL+20] is our *homogeneous* network. Similar to Willis et. al., we treat the B-Rep as an undirected graph and apply homogeneous message passing. However, since we need embeddings of all entity types, not just faces, we include all classes of entities in our graph. We also could not use grid-sampled face points as in Willis et. al. since non-face entities cannot be grid-sampled. We tried adapting other sampling strategies to topological entities of lower dimensionality, but this strictly decreased performance and so was dropped. Since our B-Rep graphs are naturally heterogeneous, we also tried a heterogeneous variant of this network that learned a different message passing function for each type of edge (vertex-edge, edge-loop, loop-face, face-loop, loop-edge, and edge-vertex). This approach is more costly as it has six times the number of parameters. Finally, we tried the B-Rep structured network described in Section 4.5, which learns separate message passing functions

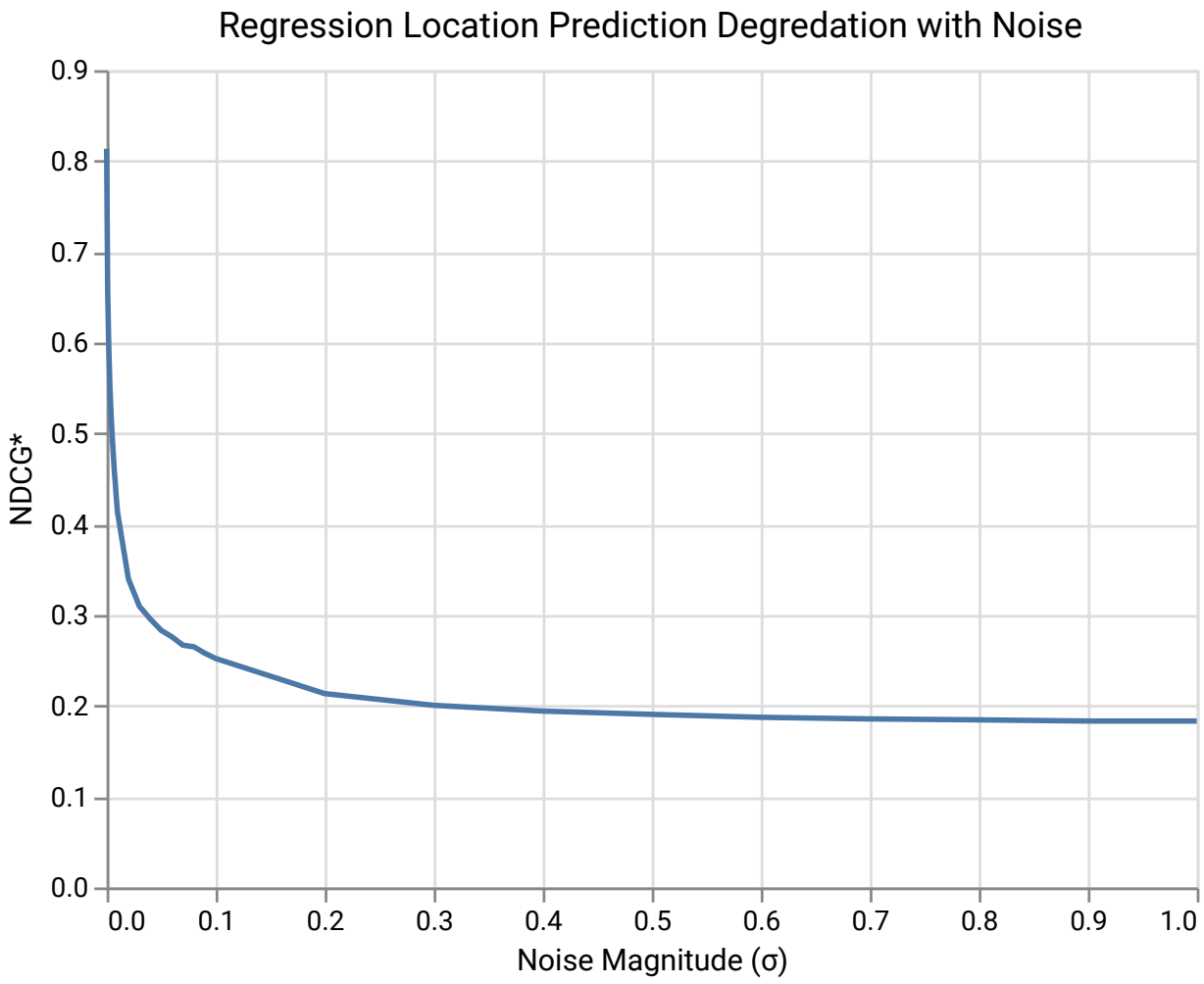


Figure 4.15: Degradation of regression-based location accuracy in the presence of noise. The sharp accuracy drop-off in the presence of small prediction noise shows that regress-and-snap prediction cannot perform well for predicting mate coordinate frames.

Table 4.2: Validation NDCG* for mate type and location problem for experiments with or without a GCN, and with all input features or just the parametric function type.

	Mate Type		Location	
	Fn Type	All	Fn Type	All
No GCN	.832	.846	.470	.561
SB-GCN	.846	.850	.547	.576

for each edge type while having the same number of parameters as the homogeneous network, and fewer overall message passes than either homogeneous or heterogeneous. We found that these two axes of exploration, features and network convolution, both improve results for the location task independently, and that each can compensate for the other. Table 4.2 summarizes these differences. We hypothesize that the additional parameter and computed features capture some of the information that the network otherwise needs neighborhood data to infer. We choose to use our structured model with all features since it performs as well as other architectures that are more costly in computation and parameter count.

4.7 Limitations

The inevitable existence of false negatives in our dataset means that our quantitative results underrepresent the quality of our predictions, as evidenced by the expert annotations in Figure 4.16. While no *in the wild* dataset can ever capture all the ways that a user may wish to use a given part pair, data augmentation can be used to reduce false negatives. In this work we augmented our dataset at the MCF level by computing alternative constructions for the same coordinate frames. Future work could extend this in three ways. First, as shown in Figure 4.17 (a), merging mates with different MCF locations that combine to the same relative positions if the mate type is given, gives us a more accurate picture of result quality (as done in Figures 4.11 and 4.16). While this cannot be done as a post process if mate type is not known at inference (as reported in Figure 4.12),

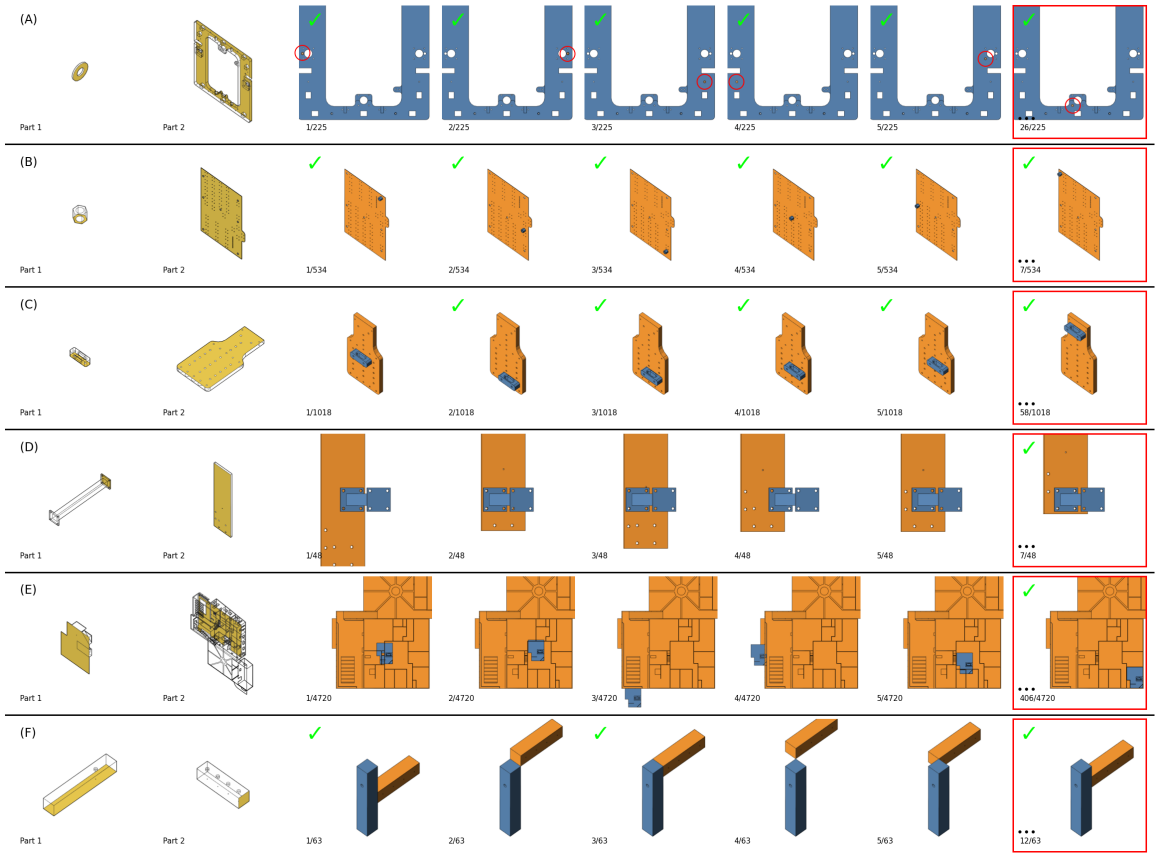


Figure 4.16: Various failure cases. (A)-(C) highlight cases where our model produces plausible results, but does not have the greater context to understand the user's intent. (D) and (E) are cases where a better understanding of extrinsic geometry would help: in (D) to understand multiple-hole alignment, and in (E) to capture geometrically close features (the walls), which are topologically distant from the pertinent MFC reference (the floor). (F) shows a case where the material context would help; these are square lumber, and the original creator intended to align one piece with a pre-drilled nail hole in the other.

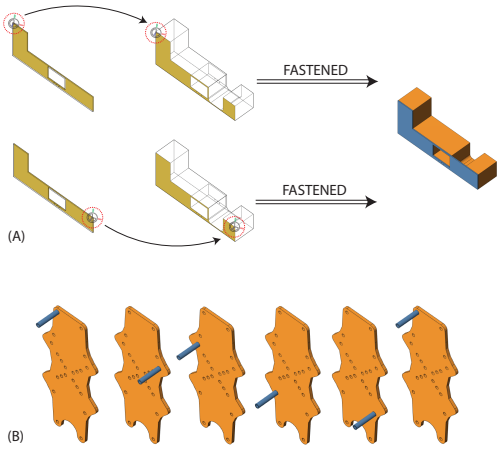


Figure 4.17: Two types of false negatives present in our training data. (a) The same relative degrees of freedom far a part pair can be achieved by different combinations of MCFs, dependent on the mate type; we currently only capture those with equivalent MCFs. (b) Identical parts pairs appear in multiple configurations in our dataset, but since we train and validate our model *per-mate*, these alternatives appear as false negatives. If we can match topological entities across equivalent B-Reps, we could unify these examples.

additional work could be done to augment the database finding all pairs with all equivalent MCFs. Second, as shown in Figure 4.17 (b), our work does not capture the alternative uses of pairs of parts that exist in our database because we separate our training and validation examples by mate pair rather than part pair. Unifying these and separating our examples by part pair would require extending our fingerprint for identifying equivalent B-Reps to a full B-Rep matching including a mapping between B-Rep topological entities. Finally, there are symmetries that we could exploit to augment our data with additional plausible mates; in Figure 4.8 this would result in the center MCFs of each hole being marked as ground truth since that part has 4-fold rotational symmetry.

Several additional features are required to make our system viable in a CAD workflow. Additional user interactions to select non-face topologies would remove the restriction of only selecting face orientations. Some mate suggestions are clearly non-viable due to intersections or displacements, these could be heuristically pruned to improve suggestion quality. We would also train an additional categorical predictor for mate location to allow for non-canonical part

orientations, which together with non-face selection would allow our system to work with the vast majority of mates.

We would also like to include some discrete geometry features into our model. While point density around sharp features presents a challenge for PointNet features, other models such as MeshCNN [HHF+19] or the curve and surface convolution features of UV-Net [JSL+21]. Further, incorporating the greater assembly context into our predictions is an exciting challenge that we leave to future work (see Figure 4.16 (A)-(C)).

Our system is built to work with the OpenCascade and Parasolid modeling kernels and with MCF based mates, but CAD systems use a variety of modeling kernels and mating schemes. Adapting our approach to other modeling kernels is straightforward; the primitives for which we support parametric features and MCF references are common to every B-Rep modeling kernel, and other entity types are only featurized by a categorical variable and geometric summary information. Adapting our data set to other modelers may incur data loss since different primitives may be used to generate the same geometry, leading to different potential MCFs. In addition to both Parasolid format and a pre-featurized serialization to avoid dependence on a modeling kernel, we have released our dataset and code using the neutral STEP format common to all B-Rep-based CAD systems using the open source OpenCascade modeling kernel to allow future work to build off ours.

Finally, this work lays the machine learning foundation for automatic mating interfaces and leaves open exciting avenues for expanding the capabilities of these systems. We limited our mate location search in this work to the neighborhood of the user's selection both to incorporate the user's design intent, and to limit the number of mates under consideration. If we expand the number of MCFs under consideration in each part, the number of possibilities grows quadratically, and will exhaust available memory at training time. This could be somewhat alleviated by random negative sampling, but even in the one-face neighborhood we had to cap each example to at most

10k MCF pairs to limit memory consumption, and some face selections can result in millions of pairs. Future work can address this limitation by making the dependency linear, choosing a constant number of MCFs on one selected part as the mate location intent, then considering all MCFs of the other part. Another direction is to extend the system to include part selection—given an MCF of a single part, suggest other parts that could mate at that location. A system combining part retrieval with placement has the potential to drastically speed up the CAD mating workflow. A follow-up paper from Autodesk research proposed an alternative method to more complete automation of mating. In [WJC+21], a simplified model of mating is instead used that restricts mates to a single pair of topological entities (face or edge), then uses a real-valued offset to regress the final part positions. Using this model they were able to train a model that can predict mates for a full assembly given the B-Reps within that assembly. They released a small but highly-curated dataset of assemblies that use these kinds of mates that complements the much larger but much noisier AutoMate dataset.

4.8 Conclusion

The success of AutoMate demonstrates the effectiveness of SB-GCN in integrating with and augmenting existing, industrial strength CAD tools. This was ultimately achieved by building off-of, and for, representations used in existing design workflows. In doing so, we were able to craft a system that both works better (compared to non-CAD representations), and is directly applicable to real engineering use cases. The next chapter is a case study that further demonstrates the effectiveness of B-Rep learning and SB-GCN for building engineering-ready solutions.

Chapter 5

B-rep Matching: Learning to Share

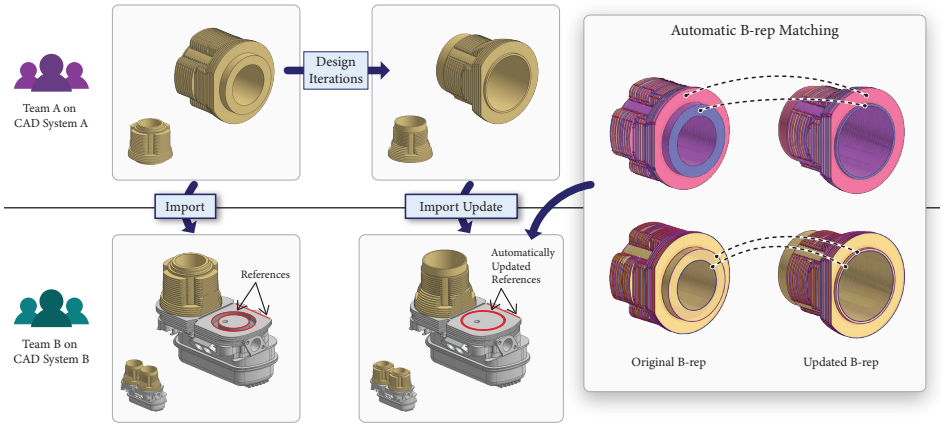


Figure 5.1: Example of collaborative workflow. Team B is modeling an engine in System B and importing a cylinder that is designed by team A in a different CAD system. The engine's cylinder base geometry is referenced off of the imported cylinder to ensure proper fit. When team A updates the cylinder design, team B imports the new B-rep model and our method enables references to be automatically re-assigned. This enables the CAD model in System B to be automatically updated to match the edits made by user A. In this example, the size and positions of cutouts are updated to align with the new model, and the overall length of the base plate changes, becoming smaller to match the new cylinder dimensions.

5.1 Introduction

This chapter is a second case study that demonstrates the effectiveness of using SB-GCN-learned design representations to streamline the implementation phase of CAD design. It shows how a data-driven approach can improve upon the editablity of B-Rep models by making referencing more robust and flexible. A weakness of using CAD software for the design process is that taking advantage of their most powerful features can require locking a workflow down to a single system, which can be impractical in many real-world situations. There are many such systems on the market as well as applications that interoperate with them, including Computer Aided Manufacturing (CAM) systems, visualization systems, and simulations/analysis systems. Large projects typically require collaboration across several such systems. For example, a team of engineers, B, may design an engine in one CAD system using a cylinder that is designed by team A in another system. When designing the engine, team B would import the cylinder's model into their CAD system and use its features as *references* (see Figure 5.1). A fundamental challenge with such collaborations is preserving such *references* after models are updated—e.g, if team B modifies the piston and sends the updated model to team A.

Persistent *referencing* is the ability to programmatically refer to parts of a CAD model, and it is a fundamental construct of modern CAD systems—for example, the chamfer operation references the specific edge where it should be applied. CAD references enable robust design iteration— for instance, if the CAD program parameters are changed, altering the shape and position of that edge, the CAD system still "knows" which edge needs to be chamfered. However, models that are shared across systems cannot generally make use of these referencing schemes, as common CAD exchange formats lack the data structures CAD systems use internally to track entities across edits. This creates a challenge for collaboration, since users would need to manually specify all references every time they re-import a model that has been updated in a different system.

In this work, we address this need by proposing a novel approach to persistent referencing that is agnostic to the CAD system: geometric matching. We observe that what makes CAD referencing schemes system-specific — inhibiting collaboration — is that they are unique, proprietary, and rely on the program history: the sequence of CAD operations that construct the shape. The reason why programmatic tracking approaches have been favored over purely geometric models is, of course, the great challenge of robust geometric matching under the wide range of topological variations common in CAD design iterations. Since program-based historical information is available *within* a single CAD system, it can be leveraged to construct heuristic-driven tracking. In this paper, we ask: *can we learn geometric matching from programmatic tracking schemes?*

Our method takes as input two CAD models: the original $\mathcal{B}_o$ and the updated version $\mathcal{B}_u$. These models are expressed in the standard geometric representation used across all CAD systems: the Boundary-Representation (B-rep). B-rep represents CAD models with infinite resolution, as a topological graph of *entities* (faces, edges, and vertices) each of which has an associated parametric geometry (surfaces, curves, and points, respectively). Our method automatically computes matches between entities of $\mathcal{B}_o$ and $\mathcal{B}_u$. By transferring references across the matched entities, this algorithm allows for seamless collaboration across CAD systems. An example use case is illustrated in Figure 5.1. In the example, the cylinder is updated and its B-rep is re-imported into the CAD system where the engine is being designed. Our method then automatically finds correspondences between the new geometry and the previous import allowing references to be directly transferred. As a result, the cylinder valve is automatically updated when the CAD program is executed with the updated references, changing overall dimensions and position of cutouts to match the imported geometry.

Our matching algorithm is based on two observations. First, we observe that matches depend not only on the geometric features of B-rep entities but also on the topological similarities—e.g. a model may have many edges that are geometrically similar (straight curves) but they may be easy

to distinguish based on the neighboring faces. Second, we observe that CAD model updates tend to affect entities in a non-uniform manner, as some regions of the model may drastically change while others stay intact. As a result, some regions of the model are "easier" to match, while others are "harder".

Based on these two observations our key insight is to use an iterative approach that can leverage the "easier" matches to find "harder" ones from neighborhood information. At each iteration, our algorithm will take a partially matched B-rep pair and suggest the most likely best match to add to the partial matching. Since deterministic algorithms for scoring such matches are challenging to design and lack robustness, we propose to use machine learning for match selection. We bootstrap this algorithm by initializing the partial match with a geometric matching algorithm that searches for entities that are unchanged between the B-reps. This step can be done robustly with a conservative geometric algorithm.

Our proposed framework has three technical contributions. First, we generate a synthetic dataset of B-rep pairs for training, and we release this collection for future research. Second, we develop an inference algorithm for scoring partial matches that can be used at any matching stage (number of partial matches). Finally, we develop an end-to-end algorithm for matching B-reps that combines geometric bootstrapping with iterative inference. We evaluate our approach on synthetically generated ground truth, compare it to different baselines, and further evaluate on a smaller expert-generated dataset to validate that our approach is applicable to real CAD workflows.

5.2 Background and Related Work

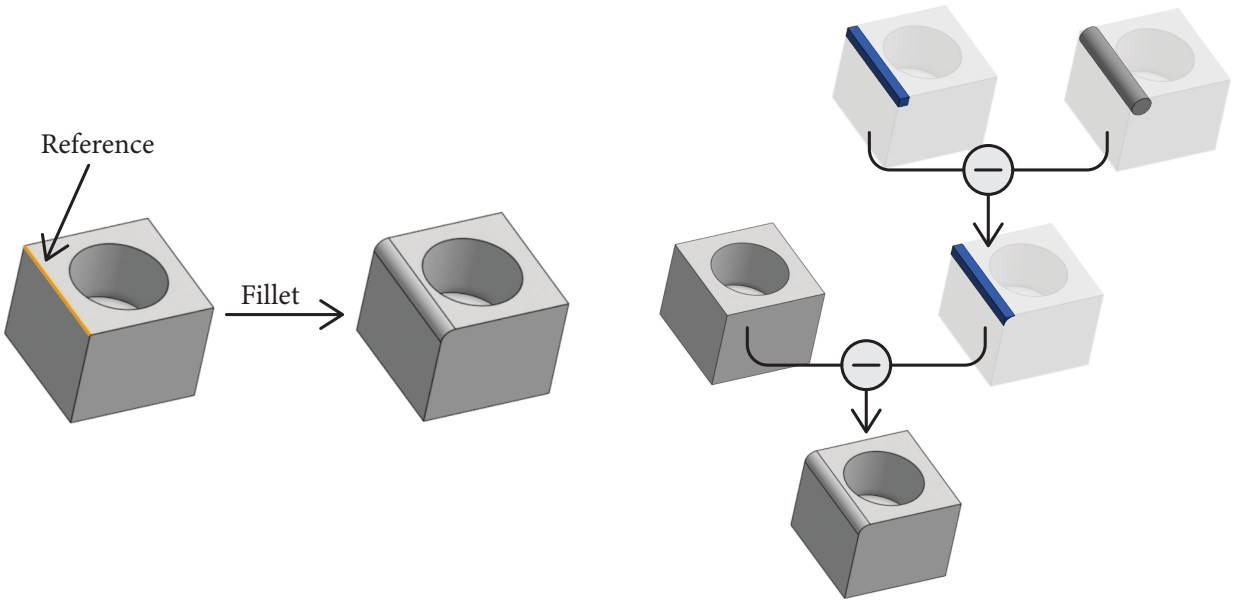
We review the state of the art of referencing and B-rep matching in commercial CAD systems, then discuss related research on shape matching and B-rep learning.

5.2.1 CAD Referencing

CAD models are constructed by tens to thousands of CAD operations (called features), most of which use references to intermediate geometry (see Figure 5.2). Persistent referencing is therefore prevalent in the CAD pipeline and has been the topic of decades of active research both in academia and in industry. Typical approaches fall within two categories. *State-based* referencing schemes allow users to provide custom geometric logic and are common in procedural interfaces (such as CadQuery, Grasshopper, and Houdini). Interactive CAD systems (such as Onshape, SolidWorks, and Fusion360), on the other hand, automatically generate referencing code from user interactions. Such tracking algorithms typically incorporate historical feature information in addition to geometric cues—e.g., a face created by an extrude operation is labeled as the result of that extrude [FH18, CMHK12, BB00, BNB05]. It is important to note persistent referencing is an inherently ambiguous problem and that typical tracking schemes use heuristics to resolve them, leading to system fragility [DCC+16, Yar13]. That said, by leveraging information from the program history in addition to pure geometric information these algorithms perform significantly better than any heuristic that can be developed without this additional contextual data. The goal of this paper is to bridge the gap between schemes based on programmatic tracking, and geometric matching. Instead of proposing novel heuristics, our approach is to learn from data.

5.2.2 B-rep Matching in Commercial CAD Systems

Since enabling collaboration across CAD systems is a fundamental part of the CAD pipeline, some CAD systems enable some form of matching across systems. A common approach is to use the referencing intelligence of the exporting software system, either via an API connection (which requires both systems installed on the same computer) or by reverse-engineering the proprietary CAD-specific file format. For example, CREO's Unite Technology includes special-



(a) Modern CAD (b) Constructive Solid Geometry (CSG)

Figure 5.2: We compare the task of rounding an edge in a modern CAD system (Onshape) and in Constructive Solid Geometry (CSG). While CSG would require a user to manually specify appropriate parameters (translations, scaling) in addition to the sequence of boolean operations, a modern CAD system would allow the user to simply call the *fillet* operation with a *reference* to the edge. This means that the user can change the radius of the fillet by changing only one parameter. Further, and importantly, if the rest of the model is updated (e.g. the cube is resized or the hole is removed), CAD systems can identify the edge and apply the fillet.

purpose methods to enable collaboration with CATIA, Siemens NX, SolidWorks, and Autodesk Inventor. Though these techniques are proprietary, some are outlined in published patents [VGLM13, SR07]. Notably, these approaches rely on specialized data formats and are therefore not general to all CAD systems.

To enable collaboration across any CAD system, some effort has been made to automatically compute correspondences directly on B-reps. Many systems (e.g., SolidWorks) can match topological entities that are not modified during the update. A more general method is described in [VGLM13], but it requires planar faces, cannot handle large modifications of the model, and only produces a rigid motion. Finally, [KS18] propose a user-assisted matching tool that uses heuristics to suggest new matches based on user input. To the best of our knowledge, ours is the first fully automatic approach that can automatically match entities without relying on hand-crafted rules and has been shown to perform well through extensive evaluations.

5.2.3 Shape Correspondence and Retrieval

Computing correspondences between two shapes is a well-studied topic in computer graphics. A thorough review of this literature is beyond the scope of this work, so we refer the reader to recent surveys [Sah20, DYDZ22]. Typical methods compute a deformation field that aligns a source surface with a target surface. Representations for such fields include sampled points, patches, and implicit or parametric functions. Conversely, the problem we address is to find a set of pairwise matches between discrete sets of B-rep entities (faces, edges, and vertices). The ablations in Chapter 4 showed that simply transferring correspondences (in that case nearest mating pairs) from more general representations such as point clouds perform significantly worse than methods that learning directly on the B-rep entities. For this reason, our approach works directly with CAD B-reps.

There is also a body of work on CAD assembly retrieval, which use geometric or topological information to retrieve assemblies, or partial assemblies, similar to a query CAD model [LPMG19]. A myriad of geometric features (curvature, dihedral angle, and other surface information), along with shape descriptors (shape distribution, 2D projections, angle distribution, spherical harmonics), have been used to find similar CAD models in large databases. In particular, [THM+13] performs partial retrieval of CAD models represented as B-reps, and identifies useful geometric and topological descriptors that facilitate finding matching regions of CAD models, such as surface parameters and convexity of adjacent faces. However, we are concerned with matches between individual discrete elements of a B-rep, including separate faces, edges, and vertices, rather than whole regions or parts. Moreover, the existing CAD model matching literature is predominantly focused on query-based part retrieval, with a resulting preference for false positives over false negatives. In our setting, the opposite is true; it is more important not to falsely label matches for the purpose of CAD references.

Data-Driven CAD Applications Several methods have been developed to assist users in various aspects of computer-aided design. For example, AutoMate [JHC+21a] and JoinABLe [WJC+21] learn to assemble CAD models from known parts. [shi22] use graph neural networks to automatically predict correct materials for parts in assemblies, although they do not deal with CAD geometry representations directly, instead using part metadata and rendered images as part-level features.

Another direction that some recent works on CAD learning have taken is to treat CAD B-reps or programs as sequential input, and utilize transformer encoders/decoders to embed or generate entire CAD models. Generative models using transformers have been developed for CAD shapes represented as programs [WXZ21a, XWL+22] and B-reps [JLD+22, GLP+22a], demonstrating the ability to produce plausible examples of CAD models, albeit with limited scale and complexity.

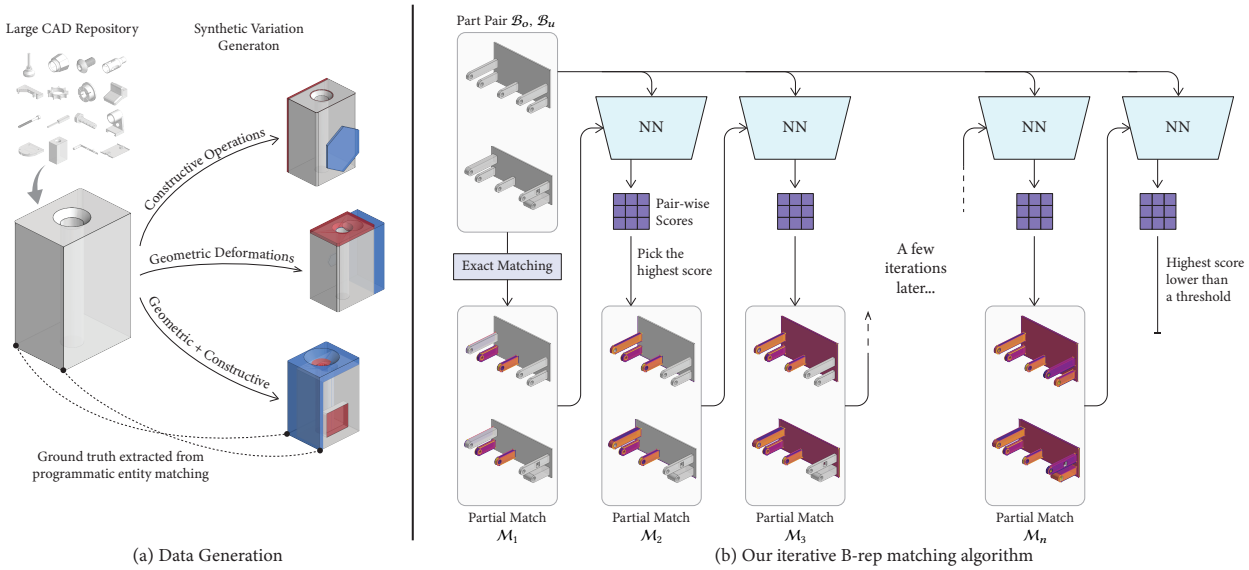


Figure 5.3: (a) Overview of our methods for data generation of B-rep matching. We draw human-created models from a large public CAD repository, and programmatically apply constructive operations and geometric deformations to create part variations. Because these edits are applied programmatically within the CAD system, we can use its programmatic entity tracking to extract ground-truth matches. (b) We use this data to train a neural network that scores potential entity matches given the geometry and topology of two B-reps and a partial matching. Our matching algorithm initializes the partial matching with exact geometric matches, then iteratively selects the next most likely match then re-evaluates matching likelihoods up to a user specified threshold.

Other data-driven CAD applications include segmentation [CRH+20], classification [Bha19], and reverse engineering of editable models [XPC+21a, LWJ+22a, UyCS+22]. To the best of our knowledge, we are the first to address the issue of finding correspondences between an original and an updated version of a B-rep.

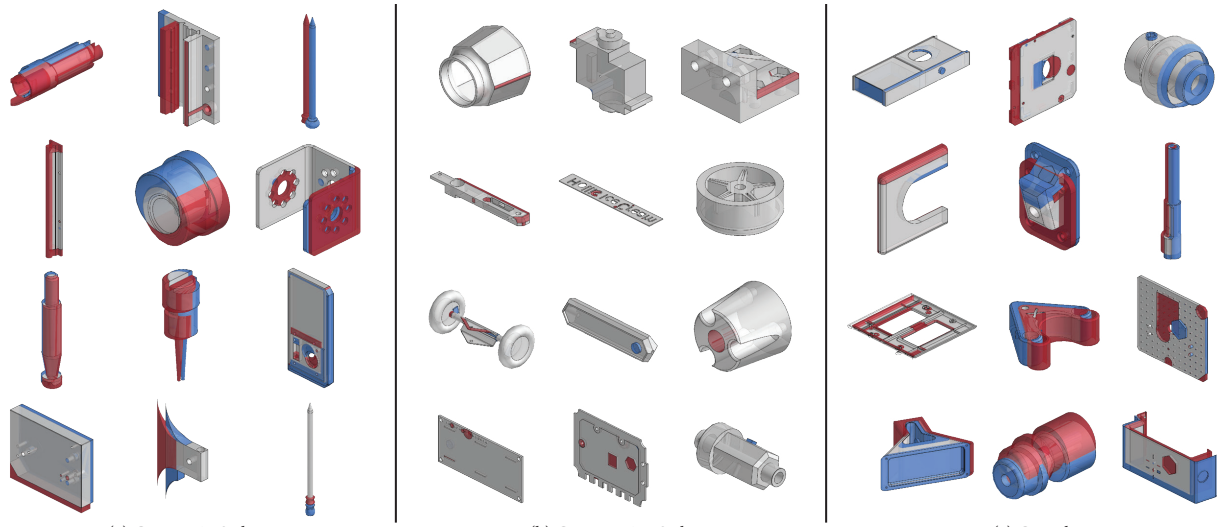
5.3 Overview

The goal of our work is to allow references to be propagated from different versions of CAD models once they are exported into the common sharing formats used when collaborating across CAD systems. This format, the B-rep, represents the geometry as a graph of entities (including

faces, edges, and vertices). Graph nodes are associated with parametric equations that define their associated geometry (surfaces, curves, and points, respectively). Graph edges denote boundary relationships; vertices bound edges, and closed *loops* of edges bound faces. Since loops aggregate other geometry, they do not have an associated parametric definition, but instead a label denoting if they are an inner or outer boundary of the face they bound. Our task is to search for pairs of matched primary entities — faces, edges, and vertices — across the two B-reps. An overview of our system is depicted in Figure 5.3.

Maintaining references across multiple CAD environments is challenging since imported models contain only geometric information, lacking the CAD program history normally used to track entities that change due to edits. One of our key observations is that while history-based tracking cannot be replicated in our context, it can instead be *learned*. We propose to use this logic to generate a large dataset of matches with ground truth labels. By analyzing a smaller set of expert-designed pairs of models and their variations, we identify the key variations that happen across versions: geometric deformations and constructive operations. We develop algorithms for automatically generating these kinds of variations within a CAD system and use its tracking mechanism to generate matching labels.

Our method for learning to match over this dataset is designed around the insight that B-rep entity correspondence is informed not only by the geometry of an entity but also by its neighborhood information. This indicates that when some matches are known, they can be leveraged to find novel matches. Our method, therefore, starts by first taking advantage of geometric information to find and match geometrically equivalent B-rep entities—i.e., entities that were not altered between versions. We then use this matching to bootstrap an iterative process that takes advantage of known matches to consecutively tackle increasingly difficult matches. We use a graph neural network to embed the geometric and topological information of entities in both B-reps, and combine this with a partial match to score all potential next matches, from



(a) Geometric Only (b) Constructive Only (c) Complete

Figure 5.4: (a-c) Examples of geometric deformations, constructive operations, and our complete synthetic dataset, respectively. Red and blue depicts the exclusive regions of original and updated B-reps, respectively, and grey depicts shared regions between the two.

which we greedily choose the next most likely match. We then iteratively re-score the remaining matches using the updated match prior, and continue until a likelihood cutoff is reached.

5.4 Data Generation

To train and validate our data-driven matching algorithm, we collect models using Onshape's public data repository and scripting capabilities, and either generate or find variations representative of typical CAD revision updates.

Expert Collection To validate our method on real-world use cases, we collected examples of updated imports—instances where a B-rep was imported first into Onshape from a different source and the content was then updated to a new version—from Onshape's public repository. While this scenario matches directly to our target application, such examples are challenging to find because doing so requires a partially manual search. This workflow is also more common in Onshape

enterprise users (companies that work on large collaborative projects) and those models are not made public.

However, since Onshape has a built-in version control system, examples of CAD revisions are easily accessible for every model in their collection by looking backward in the history of a model. Because only certain points in the revision history correspond to versions that are ready to be exported (for example, finished versions versus edits-in-progress), we manually searched the collection with the help of a CAD expert to identify version pairs representative of typical CAD workflows.

Using these two methods, we compiled a collection of 25 models: 5 from re-imports and 20 from version control revisions. Although this dataset lacks ground truth labels, and is not large enough to train over, it is a representative sample of our target application that can be used to validate our algorithm. We further take inspiration from the variations observed in these examples in constructing our synthetic training dataset.

Synthetic Variation Collection We built a collection of synthetic variations of human-designed CAD models by scraping 2,400 public models of varying complexity from Onshape's public repository, then automatically applying variations similar to those we observed in the expert validation set. We develop our variation algorithm within the scripting environment of a CAD system. This allows us to make use of internal mechanisms for entity tracking based on program history to create ground truth matches.

By analyzing our expert collection we observe that variations fall within two categories which we term constructive operations and geometric deformations. We developed custom operations that automatically generate variations of each type.

Constructive operations add or remove material from the model—for example, making a hole for a screw, filleting (rounding) an edge, or adding more detail by sketching and extruding some

new geometry. While these operations can modify the existing geometry (e.g., a face becomes shorter as its adjacent edge is filleted), these modifications tend to be minor. There is a much greater impact on the model topology: new entities are created, some are removed, and neighborhood information is fundamentally changed. Our custom constructive operations select faces and extrude polygonal sketches or circles to add or remove material. They can also fillet or chamfer edges. To achieve diverse variations that resemble the operations in the expert collection, we use a biased randomization to select the entities on which to apply these operations as well as the ranges for parameters of the operations that depend on model dimensions.

Geometric deformations, on the other hand, directly alter the geometric properties of existing elements. These typically correspond to parametric variations such as changing the length of an extrusion or modifying a sketch. While the topology can locally change under these variations, it can also be preserved, while the shape and position of entities can vary greatly. Since finding the right CAD parameters to tune without breaking the models is often challenging if they are not exposed by the designer, we instead resort to direct editing functionality that consists of moving around groups of entities. We use several heuristics to group parts of the model in direct editing move operations, then run validations to discard changes that result in undesirable changes (e.g., models being divided into two parts). We again use biased randomization when selecting which faces to move and how to move them.

Our complete data set is generated by first introducing geometric deformations and then constructive operations over the resulting variations. We generate 4-18 variations selected from a random distribution biased by the size of the model. We also run ablations of our method over a data set of exclusively geometric deformations and one with exclusively constructive operations. Figure 5.4 shows examples of all three sets. We have released our code on Onshape's public library for reproducibility.

5.5 B-rep Matching Algorithm

At its core, our method is an iterative greedy matching of two B-rep graphs, $\mathcal{B}_o$ and $\mathcal{B}_u$, to obtain an element-wise matching $\mathcal{M}$ between their topological entities. The basic algorithm is defined by Equation 5.1: at each iteration we add the highest probability matching pair from unmatched candidate pairs C of similar topological type (faces with faces, edges with edges, etc.)

$$\mathcal{M}_{n+1} = \mathcal{M}_n \cup \max_{(i,j)\in C} p_\theta(i, j \mid \mathcal{B}_o, \mathcal{B}_u, \mathcal{M}_n) \tag{5.1}$$

Crucially, this probability p is conditioned not only on the topology and geometry and the B-reps, but also on the previous partial matching $\mathcal{M}_n$. Because typical CAD edits leave some portion of the model unchanged, we are able to use exact geometric matching to effectively bootstrap this method. For the other entities, we propose to learn p_θ from data, the synthesis of which is described in Section 5.4. We continue this iteration until $\max_{(i,j)} p$ is below a user-specified threshold; this allows us to control the precision-recall trade-off at run-time.

5.5.1 Learning Matchings from Data

The scoring function p at the core of our iterative method needs to be able to pick the next most probable match given any particular partial matching $\mathcal{M}_n$. Given a dataset $D = \{(\mathcal{B}_o^0, \mathcal{B}_u^0, \mathcal{M}^0), \ldots\}$ of variation pairs and ground truth matchings, we want to find p_θ that minimizes the error over all partial matchings:

$$\min_\theta \sum_{(\mathcal{B}_o,\mathcal{B}_u,\mathcal{M})\in D} \sum_{\overline{\mathcal{M}}\subset\mathcal{M}} \frac{1}{|\overline{C}|} \sum_{(i,j)\in\overline{C}} \mathcal{L}\left(p_\theta(i, j \mid \mathcal{B}_o, \mathcal{B}_u, \overline{\mathcal{M}}), \mathcal{M}(i, j)\right) \tag{5.2}$$

where $\overline{\mathcal{M}}$ is one potential partial matching, $\overline{C} = \mathcal{B}_o \otimes \mathcal{B}_u \setminus \overline{\mathcal{M}}$ is the set of candidate unmatched pairs relative to $\overline{\mathcal{M}}$, and $\mathcal{M}(i, j)$ denotes whether $(i, j) \in \mathcal{M}$.[1]

For our application, match precision is more important than recall. When a match used as a reference to a CAD operation is missed, the CAD program will throw an error on that operation and label the error as a missed reference. If an entity is incorrectly matched, the operation may go through, causing downstream errors that are more difficult to debug. Therefore, we employ a weighted binary cross-entropy loss that gives more weight to negative match examples:

$$\mathcal{L}(\hat{p}, p) = -\left[p \cdot \log(\hat{p}) + w \cdot (1 - p) \cdot \log(1 - \hat{p})\right], \tag{5.3}$$

where w is the weighting constant. Higher w means fewer errors but with the cost of fewer predicted matches. We choose $w = 2$ in our experiments.

In practice, training over every possible partial matching is infeasible, so we estimate θ as

$$\underset{\theta}{\operatorname{argmin}} \sum_{k=0}^{N} \sum_{(\mathcal{B}_o, \mathcal{B}_u, \mathcal{M}) \in D} \frac{1}{|\widetilde{C}|} \sum_{(i,j) \in \widetilde{C}_k} \mathcal{L}\left(p_\theta(i, j \mid \mathcal{B}_o, \mathcal{B}_u, \widetilde{\mathcal{M}}_k), \mathcal{M}(i, j)\right) \tag{5.4}$$

Here $\widetilde{\mathcal{M}}_k$ is a partial match consisting of 50% of the ground truth match $\mathcal{M}$ chosen randomly in the kth training epoch. We train for $N = 1000$ epochs and select θ by minimum loss on a held out validation set with random but fixed partial matches.

5.5.2 Network Architecture

Our match scoring function $p_\theta(i, j \mid \mathcal{B}_o, \mathcal{B}_u, \mathcal{M})$ is conditioned on the geometric and topological structure of the two B-reps, $\mathcal{B}_o$ and $\mathcal{B}_u$, as well as any prior information we have about correspondences between them $\mathcal{M}$. We encode the geometric and topological information of B-rep entities

[1] $\otimes$ denotes a Cartesian product of graph nodes limited to pairs of like topology (faces $\times$ faces, etc.).

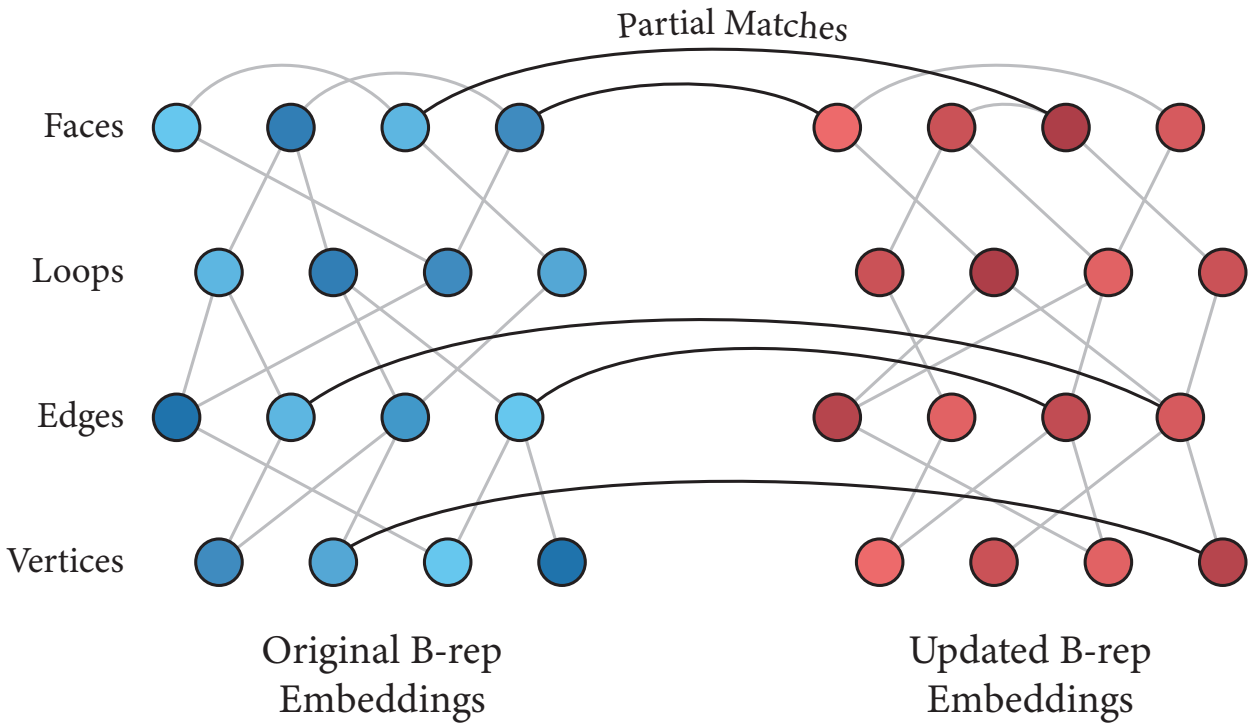


Figure 5.5: Partial match conditioning for the GAT. Partial matches are represented as cross-graph edges in unification of the B-rep graphs $\mathcal{B}_o$ and $\mathcal{B}_u$.

using a hierarchical graph convolutional network, Structured B-rep GCN (SB-GCN) [JHC+21a]. This takes as input both the parametric definitions of geometry, numerically computed statistics about each entity (such as bounding box, surface area, and center of mass), and the topological B-rep graph, and outputs an embedding vector for each B-rep entity. In our experiments, we use a 6-layer SB-GCN with a 64-dimensional embedding space.

To further condition on prior matching information, we use these embeddings as node features in a second graph convolutional network that performs message passing both within and between the parts. To achieve this, we construct a joint-part graph, encoding both the internal topology of both B-reps, as well as matches between entities of the same type as cross-part edges (see Figure 5.5). Node embeddings are initialized as SB-GCN embedding vectors. Edges are undirected, and carry a one-hot encoded type, specifying them as vertex-edge, edge-loop, loop-face, face-

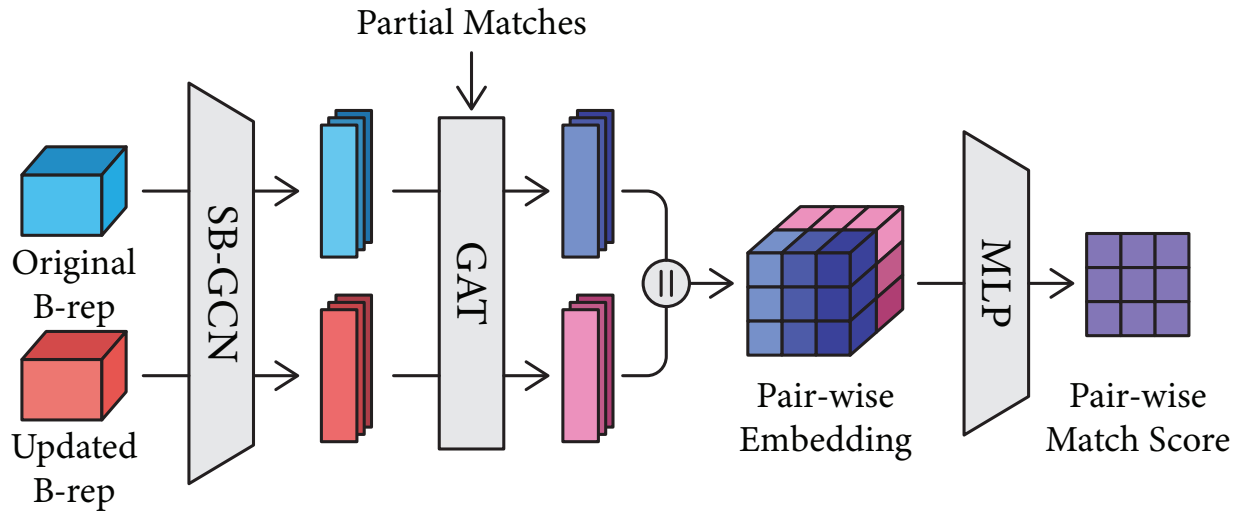


Figure 5.6: Scoring network overview. Given the original and updated B-reps, the SB-GCN computes the embeddings for each topological entity. The GAT takes partial matches and do message passing on the embeddings across two B-reps. The two sets of embeddings are pair-wise concatenated and passed to an MLP to get pair-wise match scores.

face, or prior-match. We compute our final node embeddings using a Graph Attention Network (GAT) with additive edge messages [VCC+18]. In our experiments we use a 4-layer GAT with 8 attention heads per layer and a 64-dimensional embedding space. Embeddings of candidate pairs are embedded as input to a 2-layer MLP to produce matching score logits. The full scoring architecture is illustrated in Figure 5.6.

5.5.3 Bootstrapping Matches

Our scoring model is conditional on matching priors. These have two sources, exact geometric matches, and predictions from previous iterations of our model.

As previously discussed, unchanged entities are common in CAD design updates since iterations tend to affect only sections of the model. We can use a B-rep kernel (e.g. Parasolid) to check if two entities are coincident. To speed up processing, rather than checking every pair, we only check the models whose centroids match, which can be checked for efficiently by hashing the

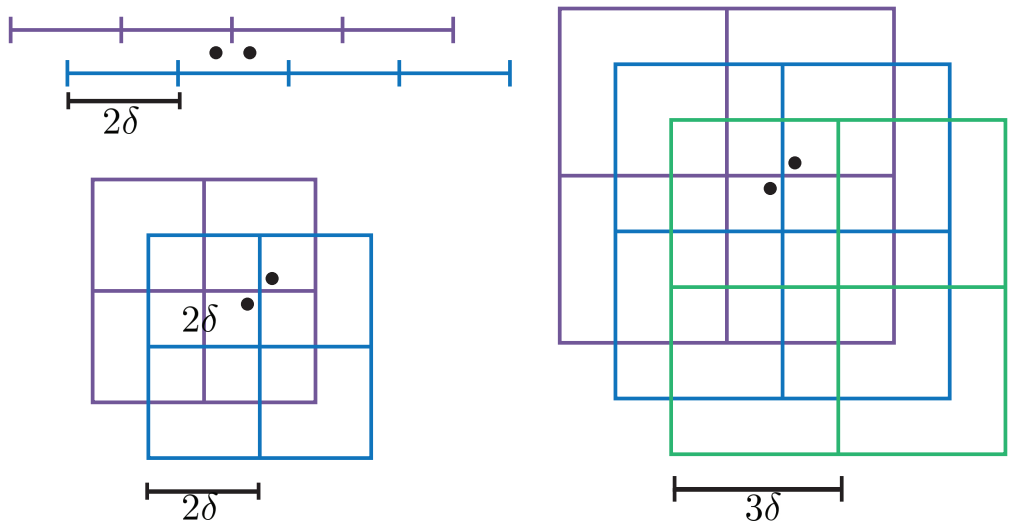


Figure 5.7: We illustrate our hashing approach using shifted grids. The top left corner illustrates the grids in one dimension for finding solutions that are close up to tolerance δ. In this case we have two grids where each cell has length 2δ and the cells are shifted by δ. Notice that if two points are close within tolerance δ, there will *always* exist one cell either on the top or the bottom grid that contains both points. To expand this to higher dimensions vectors, we must create $D+1$ grids, of edge length $(D+1)\delta$, where D is the number of dimensions. We illustrate an example of shifted grids for two dimensions (right) and illustrates why simply using two shifted grids does not generalize for higher dimensions (bottom left). While this method deterministically ensures that pairs of points within tolerance are found, it may further return points that are not within tolerance. Specifically, it may return points that are at most $(D+1)\sqrt{D}\delta$ apart (the dimensions of the diagonal of the grid cell.) In our solution we further check those points and filter them out.

3D vector representing the centroid of each topological entity in the old body. To account for tolerances, we propose a 3D nearest neighbor structure using shifted grids, which is efficient in low dimensions and deterministic (see Figure 5.7).

5.6 Results

Our method was trained using our synthetic data. The training took 12 hours on average using a single NVIDIA RTX 2080 Ti. The trained network was evaluated against the generated ground truth, and we further evaluate our approach over the expert data set to validate that our method,

trained on synthetic data, generalizes well to real-world data.

5.6.1 Synthetic Data Evaluation

We constructed a synthetic variation dataset using the geometric deformations and constructive operations described in Section 5.4. We collected 2,400 CAD models and generated 3 variations of each. After filtering parts that caused import errors and scale outliers over 100 times the size of the median part, our dataset contains 2,266 original models and 6,257 variations. We split this dataset by original model 80-10-10% for training, validation, to ensure both models remain unseen between training and testing. We report the results of our approach and compare them to deterministic baselines and ablations. We refer readers to the Table C.2 in the appendix for numerical comparisons between methods.

Results Across Different Entity Types Our method correctly matches 91.5%, 92.1%, and 94.9% of faces, edges, and vertices, respectively. We note that not all entities have ground truth matches because entities can be added or deleted over design iterations. Our goal is to match as many entities as possible while avoiding errors. We therefore evaluate our results by categorizing the entities in $\mathcal{B}_u$ into five groups:

- *True Positive.* Entities from $\mathcal{B}_u$ that had a ground truth match in $\mathcal{B}_o$ and were matched correctly by our algorithm.
- *True Negative.* Entities from $\mathcal{B}_u$ that did not have a ground truth match in $\mathcal{B}_o$ and were labeled as unmatched by our algorithm.
- *Missed.* Entities from $\mathcal{B}_u$ that had a ground truth match in $\mathcal{B}_o$ but were labeled as unmatched by our algorithm

- *Incorrect.* Entities from $\mathcal{B}_u$ that had a ground truth match in $\mathcal{B}_u$ but were matched by our algorithm to a different entity in $\mathcal{B}_o$.

- *False Positive.* Entities from $\mathcal{B}_u$ that had no ground truth match but were matched by our algorithm.

Table 5.1 reports the percentage of total entities in $\mathcal{B}_u$ that fall within each category. On average, the exact matching finds 15.7%, 23.8%, and 35.4% of the total matches for faces, edges, and vertices. Our neural network finds an additional 72.6%, 63.9%, and 55.2% of the total matches, while maintaining a 2.8%, 1.9%, and 1.1% error rate.

In Figure 5.8, we report the breakdown over different thresholds used as the stopping criteria for the iterative matching algorithm. We stress that for our application, entities that are matched but should not have been matched (both incorrect or false positives, in shades of red) create greater issues for users than missed matches (in beige) as they are more difficult to debug.

As seen in the third column, which is the result using the complete synthetic data, our method is effective at trading off errors and the number of correct matches as this threshold varies, and we have manually selected a threshold of 0.7 based on this result (reported in Table 5.1). We note that while other thresholds would result in even fewer errors, the 1-3% error rate already makes these occurrences very infrequent. Further, not all entities are used for references, so when composed with the likelihood of it being used, the chance of these errors causing issues for the user will be even lower. Finally, we note that zero error is not a realistic goal because of potential ambiguities in the ground truth data and potential one-to-many matches we have not accounted for.

The first two columns of Figure 5.8 show our model when trained and evaluated on synthetic data generated by two simplifications of our variation algorithm: one using exclusively geometric deformations and the other using exclusively constructive operations. We compare these results with the complete synthetic data. We observe that constructive operations lead to a much larger

number of true negative, which is to be expected as those operations create new entities that do not have a previous match. They are also significantly easier to match since geometric changes are small. Geometric deformations, on the other hand, create more challenges for matching. Results over the complete dataset have a similar number of true negatives as constructive operations because it incorporates those changes, and also similar to geometric deformations, it has to trade off missing to incorrect labels because of the challenges created by geometric changes.

Table 5.1: Our results on complete synthetic data reported at threshold 0.7.

		Faces	Edges	Vertices
Exact	True Positive	11.0%	14.0%	17.1%
Ours	True Positive	62.0%	51.6%	43.8%
	True Negative	29.5%	40.5%	51.1%
	Missed	5.6%	6.0%	4.0%
	Incorrect	2.0%	1.2%	0.5%
	False Pos	0.8%	0.7%	0.6%
	Correct Label	91.5%	92.1%	94.9%
	Incorrect Label	2.8%	1.9%	1.1%

Deterministic Baselines We consider three deterministic baselines which incrementally increase the number of potential matches by trading-off potential errors: coincidence matching, overlap matching, and adjacency propagation. The coincidence matching baseline enables us to evaluate the lift that our network has over exclusively applying the coincidence matching algorithm described in Section 5.5.3.

There are many cases where the coincidence matching algorithm will fail to match entities with minimal edits to the geometry. These are very common under constructive operations: for example, a face whose edge has been filleted or where a hole has been applied will only reduce in size by a small amount, but the underlying parametric geometry will remain unchanged. This observation inspires our second baseline, which matches both entities that are coincident, and

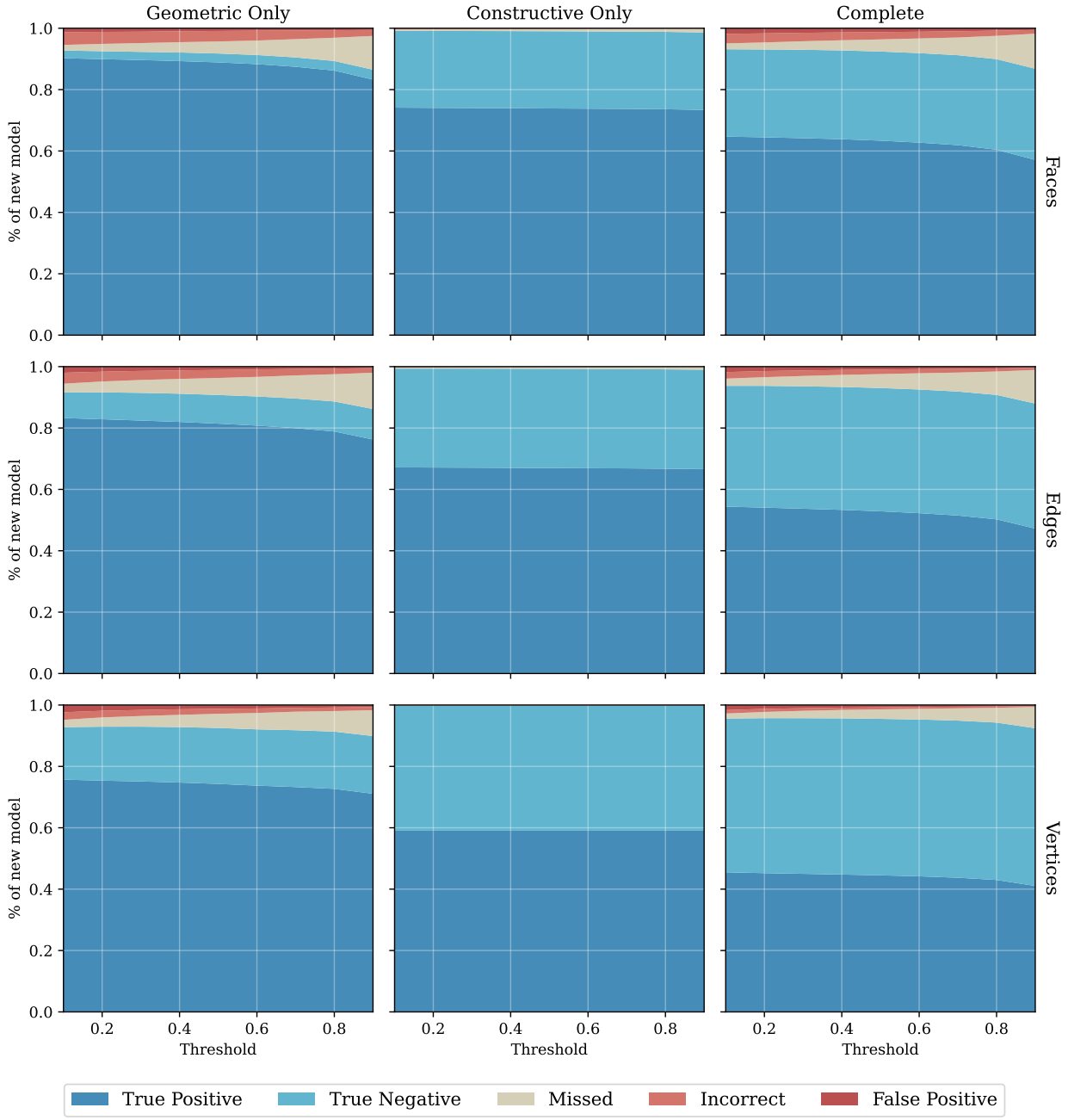


Figure 5.8: Metrics breakdown at different thresholds on synthetic data

ones that have a significant overlap. Our overlap matching function looks for pairs of edges and faces that were not matched by coincidence matching and whose associated geometry (curves

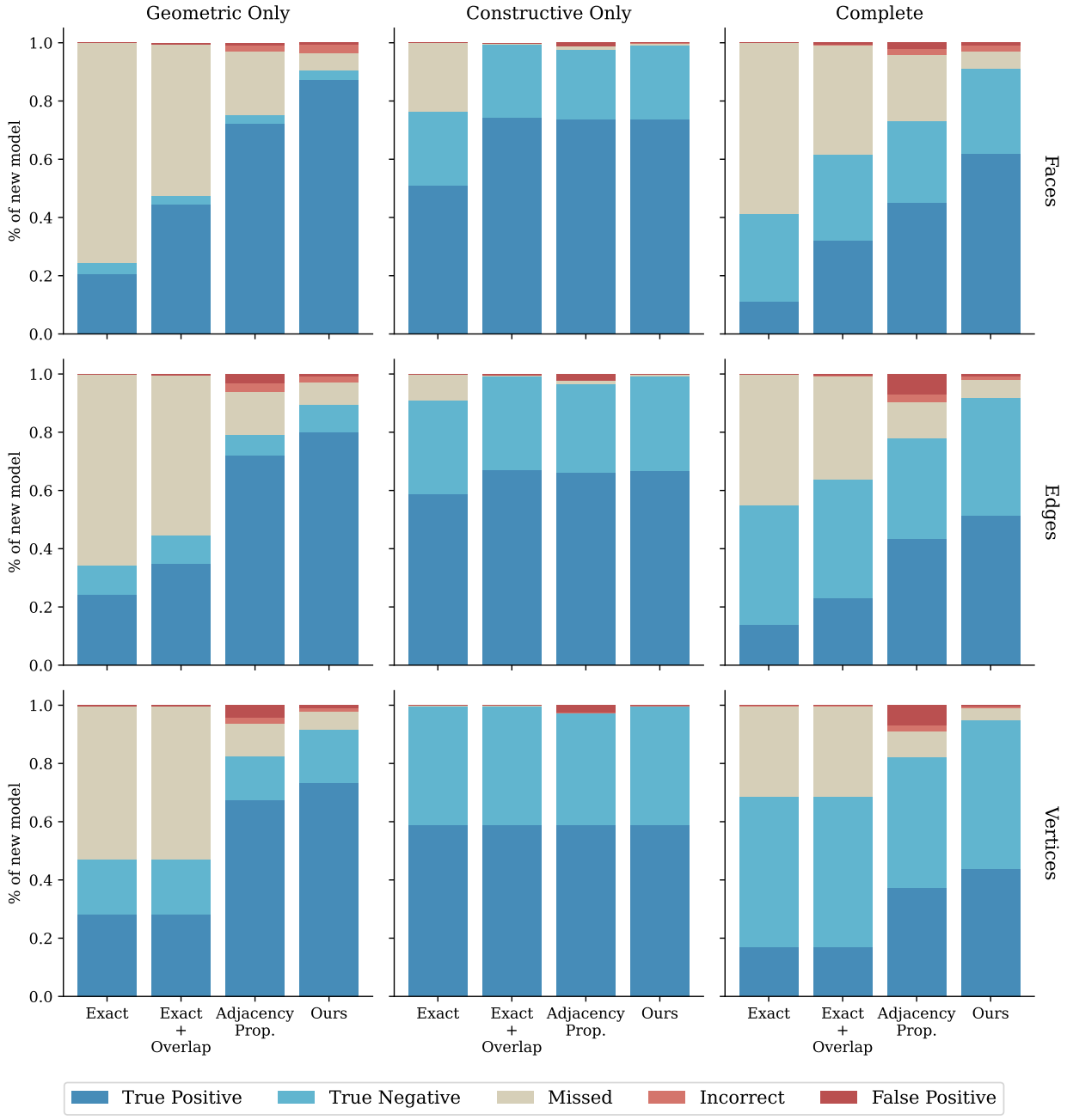


Figure 5.9: Comparisons with deterministic baselines on synthetic data

and surfaces) are identical. For each pair, we then compute their overlap, and if the overlap is a large enough fraction (80% in our implementation) of the smaller of the two entities, the pair is

considered matched. This computation is accelerated by hashing common geometric types and using the shifted grids algorithm described in Figure 5.7.

Finally, we consider an iterative solution that matches over deterministic signatures of adjacency. This method works by first matching by coincidence and overlap, then iteratively matching entities that have a matching adjacency signature. To create an adjacency signature for a given entity, we traverse its neighborhood graph in counterclockwise order and store the match index of each entity in the neighborhood (using a special value of -1 if that entity is unmatched). We use lexicographical sorting to avoid the ambiguity of which entity to start with, treating loops individually and aggregating them in lexicographic order. Adjacency signatures are computed for entities in $\mathcal{B}_u$ and put in a priority queue based on the ratio of matched to unmatched entities (so that the function matches based on more matching adjacency first). We go over this priority queue creating matches if (a) there is a unique entity if $\mathcal{B}_o$ with the same adjacency signature, or if these can be disambiguated by the geometric heuristics (like preferring to maintain curve/surface types or matching coaxial cylinders). The adjacency signatures are updated every time a match is found and the algorithm terminates when no more matches can be found. This final baseline describes our best attempt at addressing this problem in a deterministic manner. This algorithm has been integrated into the Onshape software and is currently executed every time an Onshape user re-imports a model. See more details in the published patent [BS20].

We compare our solution with these three deterministic methods in Figure 5.9, showing results over different entities and testing sets, and in Figure 5.10 for visualizations. Our method significantly outperforms all baselines. In the complete synthetic data set, when comparing our method with exact matching, we see a lift of 50.2, 37.1, and 26.3 percentage points of correctly labeled faces, edges, and vertices (both shares of blue).

We also observe a large lift over overlap matches: 29.8 and 28.3 percentage points of correctly labeled faces and edges, respectively. We note that such lift is significantly smaller if we do not

consider geometric deformations, as overlap matches will handle the majority of constructive operations (See the second column of Fig. 5.9).

Finally, when compared to adjacency propagation, our method not only improves the number of correct labels but is also significantly more robust. For example, in Fig. 5.10b, our method was able to correctly match most of the gear teeth while the adjacency propagation method failed to distinguish between them because they are topologically the same. Another common pattern is that the adjacency propagation method will match one the filleted edges (See Fig. 5.10a and c). Our method learns not to match these one-to-many cases.

Contrastive Learning Baseline In addition to these deterministic baselines, we compare against a contrastive learning representation (CLR)-based method that computes matches by greedy matching of per-element embeddings produced by SBGCN, up to a minimum cosine similarity threshold. We use an N-tuplet contrastive loss to encourage matching elements to have similar embeddings. With this method, we find that there exists no similarity threshold that provides a good trade-off between the number of missed matches and the amount of error. Compared to this baseline, our method has a lift of 48.5, 35.4, and 25.8 percentage points of correctly labeled faces, edges, and vertices. Despite attempting many variations of this approach with different loss functions and geometric features presented to the encoder, such as UV-Net embeddings of faces and edges, we were ultimately unable to improve on the quality of results using this method.

Ablations We compare the performance of our method against the adjacency propagation baseline, the presence or absence of various architectural choices, and the CLR baseline, and we show dominance of our final method using Pareto plots in Figure 5.11. This plot reports the possible trade-offs between correct (true positive and true negative) and incorrect (incorrect

match and false positive) labels. Our method significantly outperforms the adjacency propagation baseline (represented by the pink dot in the plots), especially in terms of the amount of false matches found.

Other design choices we test are preconditioning on partially known and incrementally inferred matches, weighted binary cross-entropy loss, and graph message passing layers along overlap matches (adding them to the match condition GAT as an additional edge type). We note that the iterative approach and preconditioning on partial matches are the most important components we evaluated, as removing them leads to a significant worsening of results across all combinations of topology types and datasets (see the red and orange curves). Both of these curves represent the performance of the single-shot algorithm where all scores are predicted at once, one with preconditioning on exact partial matches and one without. We also show that removing the weighted cross entropy loss that weights incorrect matches more heavily (green) harms performance, especially increasing the incorrect matches over the other methods compared. Finally, we evaluate the effect of additional message passing layers between entities matched by overlap (purple), but this did not consistently improve results to warrant the increased complexity, so it was excluded from our final method (blue).

We also evaluate the sensitivity from initialization by initializing our method with different percentages of exact matches. Initializing with only 30% of the exact matches reduces performance by about 1 percentage points (cyan). However, removing initialization completely reduces performance by 8.1, 7.2, and 5.7 percentage points for faces, edges, and vertices, respectively (yellow).

5.6.2 Expert Data Evaluation

We evaluate our method using the expert collected dataset described in Section 5.4. Because this dataset does not contain ground truth labels, we showed our method's predictions to a CAD expert and asked them to annotate matches they perceived as missing, incorrect, or extraneous through the use of an interactive GUI. The interface presented them with paired 3D visualizations with our predicted matches highlighted on a 3D model of both the original part and the variation and asked them whether the corresponding matches (or lack thereof) contained any errors. When a match was found, they might report that either no match should have been found, or the match found was incorrect. In addition, we filtered the expert collection to 16 examples which had a reasonable number of entities to manually annotate.

Statistics over this dataset are reported in Table 5.2 validating whether our method trained on synthetic data generalizes to real-word examples created by CAD designers. We illustrate some of these matches in Figure 5.12 which show a large number of accurate matches for substantial geometric and topological variations.

Examples of errors found by the expert are shown in Figure 5.13. Missed matches (a) are the most common type of error, owing to the difficulty of tracking the correspondence of topological entities that have changed geometrically (which can be an ambiguous task). In rarer cases, structural changes can lead to topological entities neighboring the change being mismatched (Figure 5.13 (b)). Finally, the rarest type of error (occurring in less than 1% of entities) is for the model to predict a match where none should exist. For instance, when parts of the model drastically change, such as the "T" shape turning into a "B" in Figure 5.13 (c), our model may find some spurious matches along the letter profiles, despite the different-shaped letters not corresponding in any meaningful way.

Table 5.2: Our results on the expert collection validated with user study. All numbers are percentages of $\mathcal{B}_u$ entities.

		Faces	Edges	Vertices
Exact	True Positive	32.1%	40.9%	45.3%
Ours	True Positive	74.2%	65.1%	61.0%
	True Negative	15.3%	22.0%	27.8%
	Missed	8.2%	11.1%	9.9%
	Incorrect	1.8%	1.7%	0.5%
	False Positive	0.5%	0.0%	0.9%
	Correct Label	89.5%	87.1%	88.8%
	Incorrect Label	2.3%	1.7%	1.4%

5.6.3 Limitations

A fundamental limitation of our method is that correspondences are often ambiguous and users will disagree about how matches propagate with changes. While we have used Onshape's programmatic tracking scheme as ground truth, that algorithm has its own limitations as it is driven by heuristics and may not match users' expectations in every circumstance. We argue, however, that while such ambiguities are common and will be present in a large portion of models, they typically account for a small portion of the entities within these models. This is what ensures the high performance of our method: typical models have hundreds of entities and only a very small portion of them will have ambiguous matches.

Our method can readily address these kinds of ambiguities by including a user in-the-loop. Since our method is iterative and driven by previously known matches, it lends itself well to this type of workflow. For example, future directions could explore additional message passes across user-specified matches for resolving ambiguities, as prototyped in Figure 5.14. Efforts to address ambiguities should also consider one-to-many and many-to-one matches. CAD referencing schemes that have access to the program history can output one-to-many or many-to-one correspondences. Since finding such maps is challenging with no contextual information and can lead

to errors, we have chosen a conservative approach that only maps one entity to another, randomly selecting a match as ground truth if the historical referencing scheme returns a one-to-many match.

5.7 Conclusion

This case study demonstrates how augmenting B-Reps with learned representations can improve the performance of existing CAD systems and automate non-creative busy-work in the CAD design process. More fundamentally, allowing smoother operation between incompatible CAD systems reduces friction and lock-in, which is crucial for creative decision-making [IPR10, SC07, BTL09]. Beyond just demonstrating that SB-GCN is a powerful and useful representation learning architecture, it shows that a core feature of symbolic CAD representations, referencing, is robustly learnable across levels of the topological hierarchy. Taken together, the SB-GCN case studies show that when the data is available, SB-GCN based models improve CAD modeling workflows.

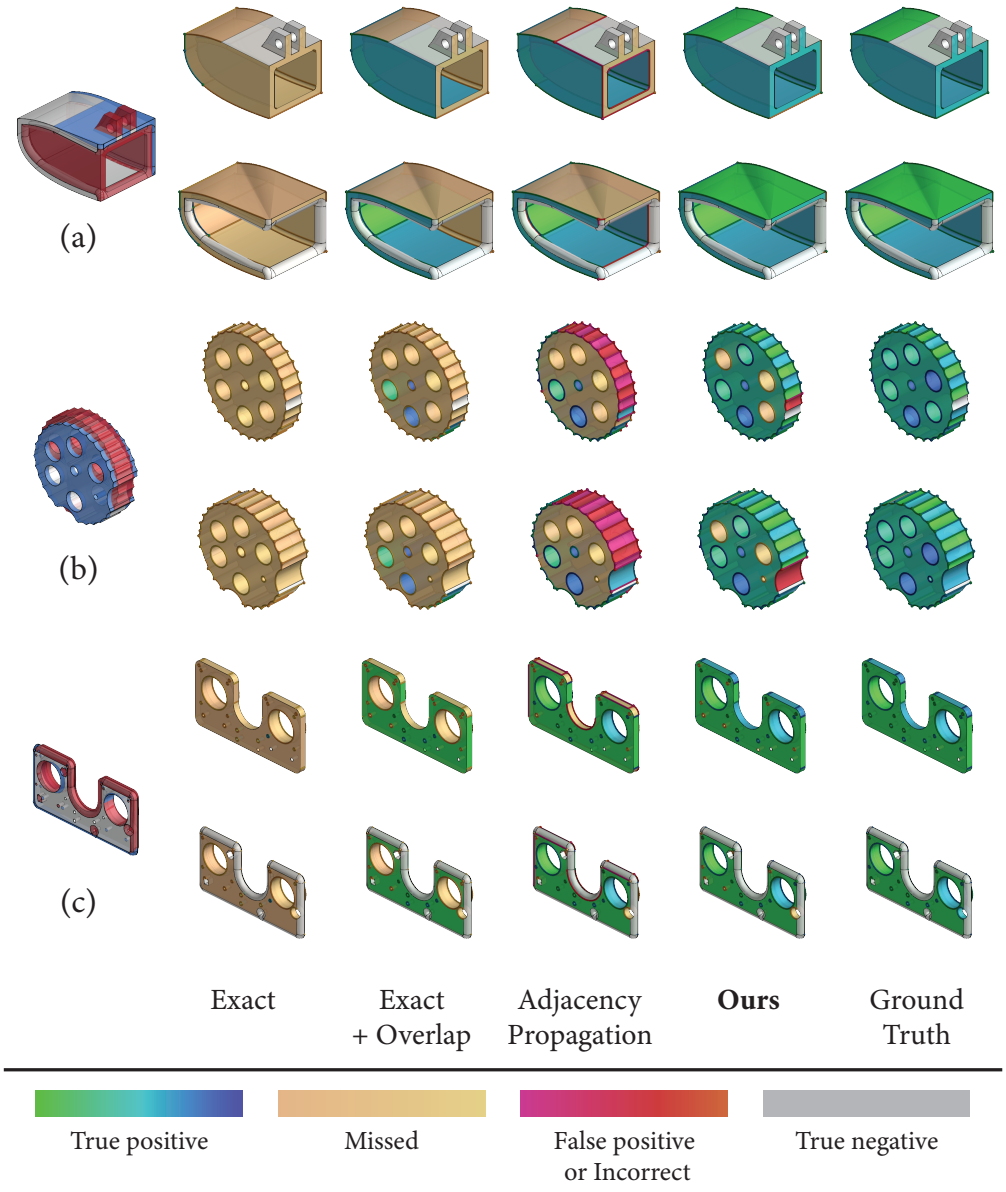


Figure 5.10: Gallery of B-reps pairs matched by different deterministic baselines, as compared with our method. The first column shows the difference between original and updated B-reps. The remaining columns show matching performance by different algorithms. For each pair, the top is the original and the bottom row is the updated B-rep. Entities (top and bottom) that were matched are rendered with the same color. True positives are in shades of blue-green; incorrect labels (incorrect and true negatives) in shades of pink-red; missed matches in shades of yellow; and true negatives in grey.

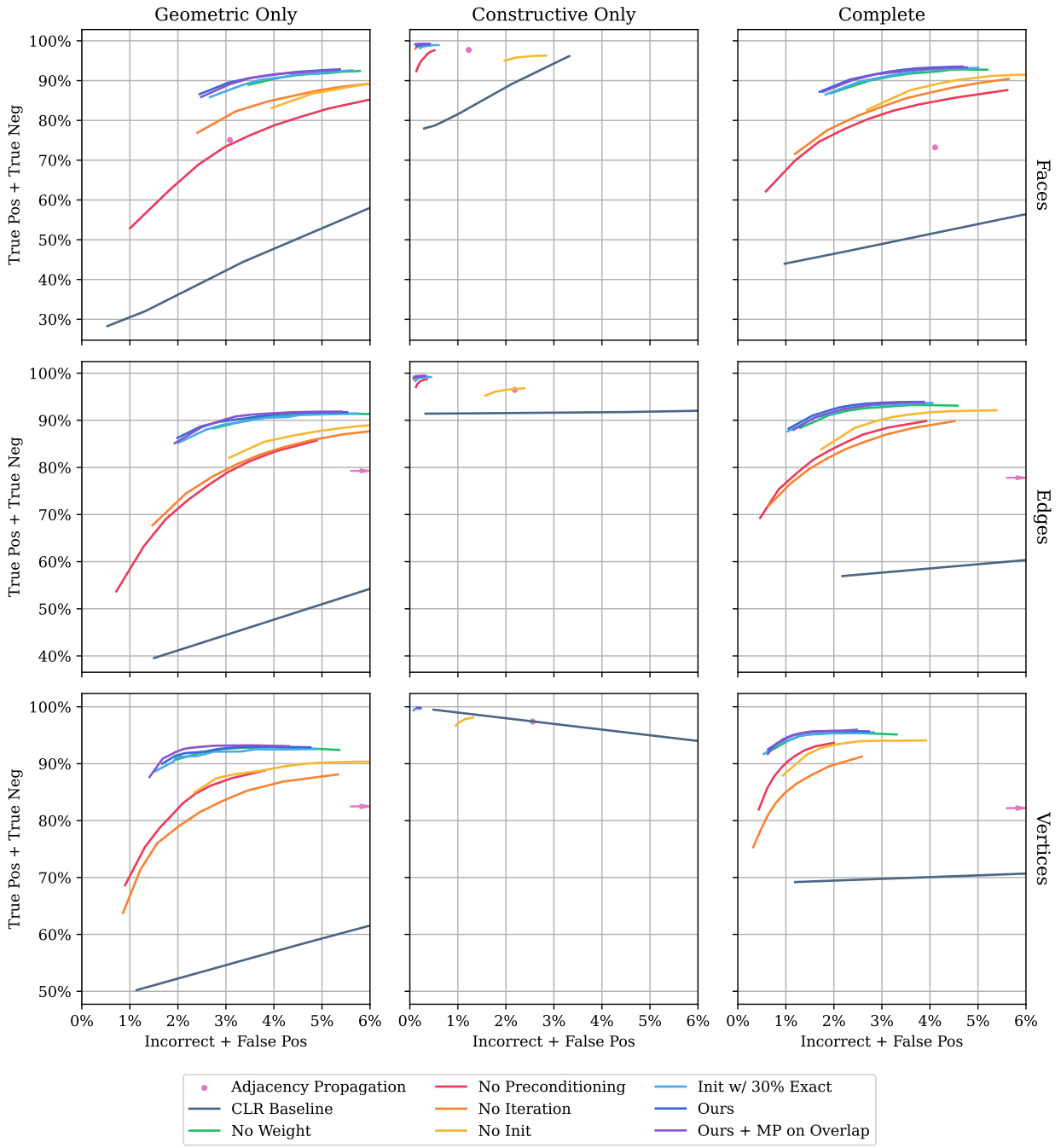


Figure 5.11: Comparisons between variations of our method. The plot shows trade-offs between correct labels (true positive and true negative) and incorrect labels (incorrect and false positive) (upper-left is better) The adjacency propagation baseline is shown as a single point for reference.

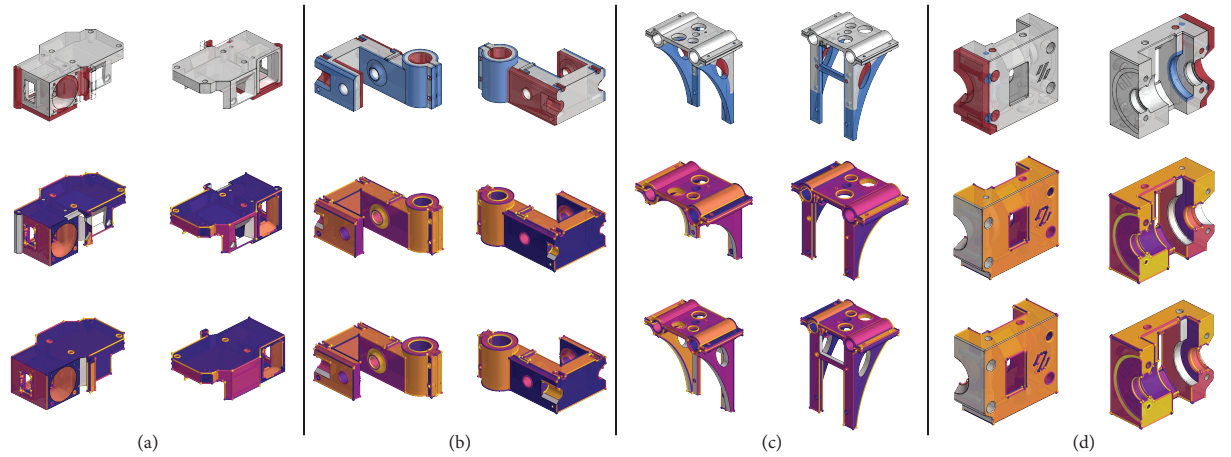


Figure 5.12: Examples of models from our expert data set where matches are shown from two views (columns). The top row shows the difference between original and updated B-reps. The second and third rows show the original and updated B-reps, respectively, with faces color-coded based on matches found by our algorithm (no ground truth is reported).

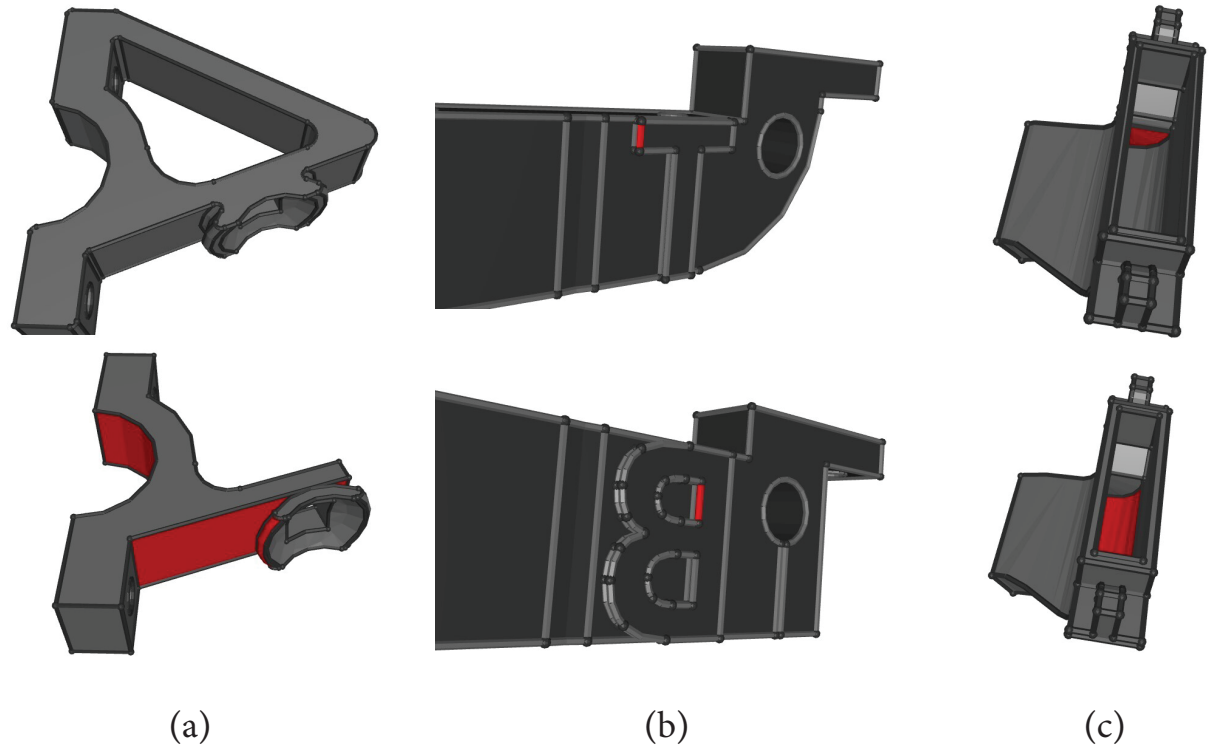


Figure 5.13: Examples of failure cases on the expert dataset. Some cases where this happens are large geometric changes (a), total replacement of sections of a part (b), and topological changes that confuse neighboring matches (c).

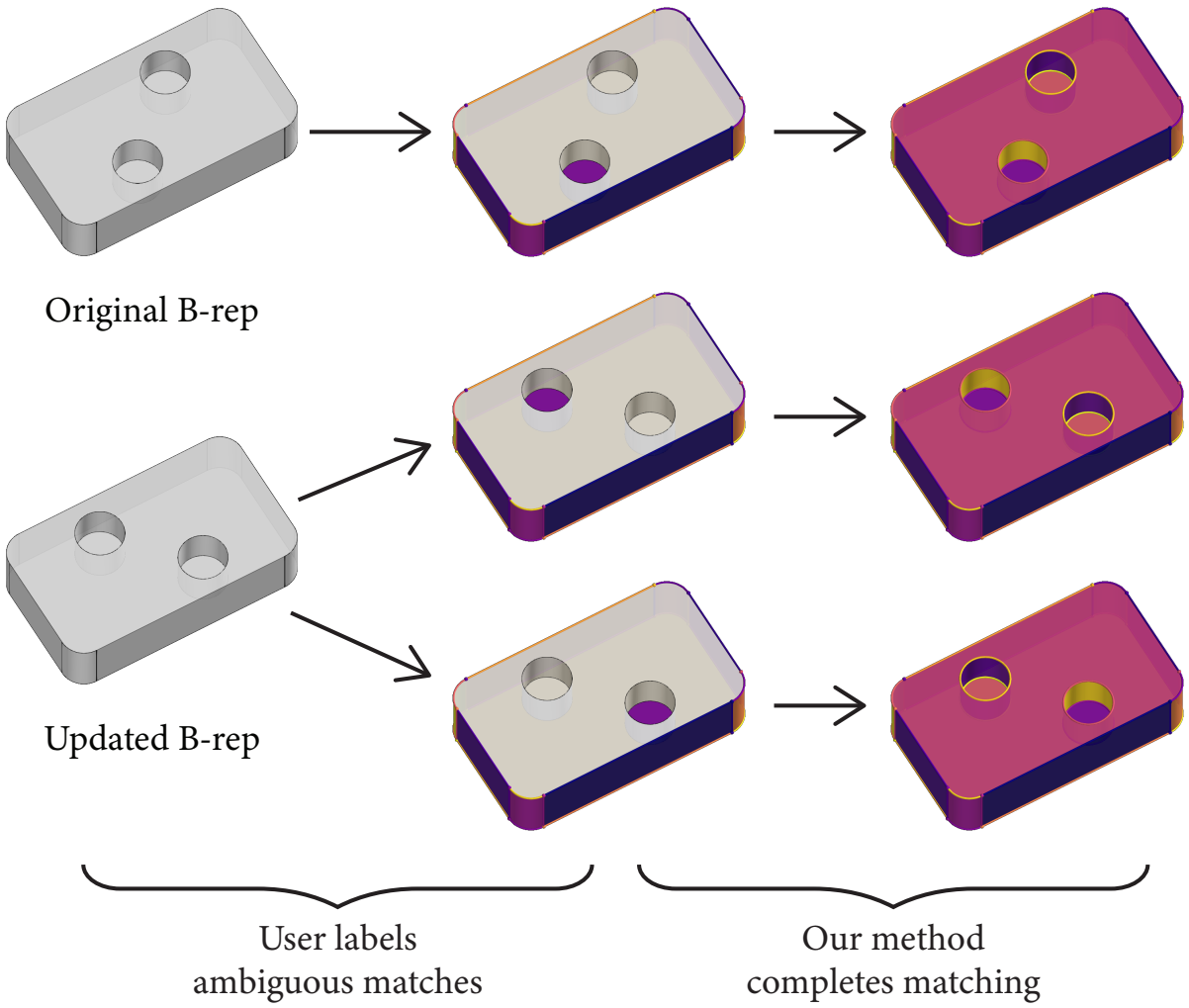


Figure 5.14: An example of user guidance that can be further explored in future work. Here the updated B-rep includes a 90-degree rotation of the two circular cutouts making the match ambiguous. With only *one* user-specified match (face in purple), our algorithm is able to infer the rest of the matched entities.

Chapter 6

Self-Supervised Representation Learning

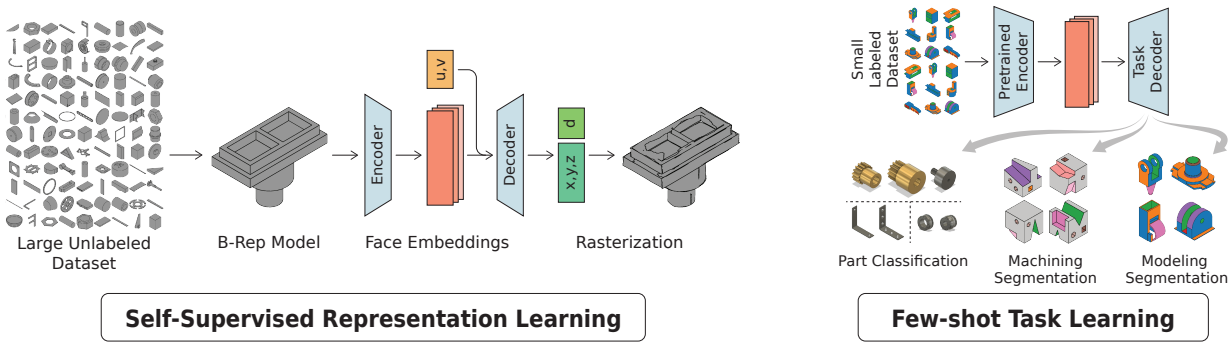


Figure 6.1: Overview of our technique. We train a geometric self-supervision task of a large, unlabeled dataset of CAD Boundary Representations (B-Reps) to learn geometrically relevant representations for each B-Rep face. These pre-trained representations are then used to train few-shot segmentation and classification learning tasks on labeled B-Rep datasets.

6.1 Introduction

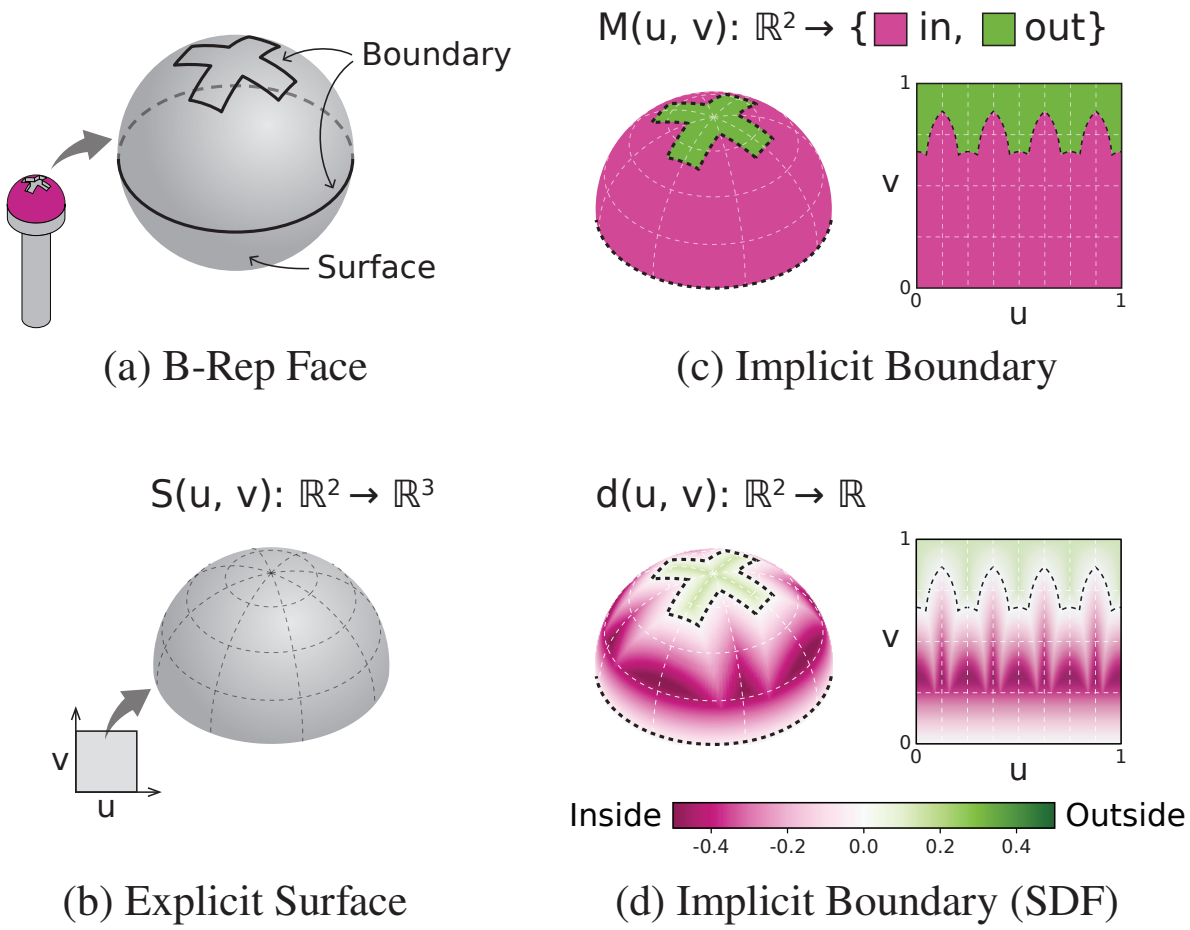
Both SB-GCN case studies relied on the existence of a sufficiently large and well-labeled dataset to train the model over. In the case of AutoMate, the Onshape public dataset provided a very large in-the-wild collection of mated assemblies from which to derive labels. In B-Rep Matching, we

were able to generate a good quality synthetic dataset for training our system. Unfortunately, this situation is not the norm in the CAD world. Manually labeling B-Rep data is time-consuming and expensive, and its specialized format requires CAD expertise, making it impractical for large collections, and making any such collections valuable assets that engineering firms are unlikely to share. As we have seen in the previous chapters, however, there is a copious amount of raw B-Rep geometry available to learn from. In this work we ask: how can we leverage large databases of *unlabeled* CAD geometry for analysis and modeling tasks that typically require labels for learning?

Our work is driven by a simple, yet fundamental observation: the CAD data format was not developed to enable easy visualizing or straightforward geometric interpretation: it is a format designed to be compact, have infinite resolution, and allow easy editing. Indeed, CAD interfaces consistently run sophisticated algorithms to convert the CAD representation into geometric formats for rendering. Driven by this observation, *our key insight is to leverage large collections of unlabeled CAD data to learn to geometrically interpret the CAD data format.* We then leverage the networks trained over the geometric interpretation task in supervised learning tasks where only small labeled collections are available. In other words, we use geometry as a model of *self-supervision* and apply it to *few-shot-learning*.

Specifically, we learn to rasterize local CAD geometry using an encoder-decoder structure. Recall from Chapter 2 that the parametric geometry associated with each node is *unbounded*, and bounds are computed from the topological relationships: curves bounding surfaces and points bounding curves. As shown in Figure 6.2, the geometry of a B-Rep face is computed by *clipping* the surface primitive to construct a surface patch, where the clipping mask is constructed from adjacent edges.

Thus, B-reps are constructed piecewise by *explicitly* defined surfaces with *implicitly* defined boundaries. This observation drives our proposed learning architecture, which reconstructs faces by jointly decoding the explicit surface parameterization as well as the implicit surface boundary.



(a) B-Rep Face (c) Implicit Boundary

(b) Explicit Surface (d) Implicit Boundary (SDF)

Figure 6.2: A B-Rep face (a) is a surface patch cut from a geometric primitive surface (b). The adjacent edges define a clipping mask (c), which we learn an SDF (d).

Our proposed encoder uses message passing on the topological graph to capture the boundary information to encode B-Rep faces. To handle graph heterogeneity (nodes comprised of faces, edges, and, vertices), we use a hierarchical message passing architecture inspired by the Structured B-Rep GCN [JHC+21a]. Our decoder uses the learned embeddings as latent codes for two per-face neural function evaluators: one mapping from $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ that encodes the face's parametric surface (Figure 6.2 (b)), and one mapping $\mathbb{R}^2 \rightarrow \mathbb{R}$ that encodes the face's boundary as a signed distance field (SDF) *within* the parametric surface (Figure 6.2 (c,d)).

We apply our proposed model of B-Rep self-supervision to learn specialized B-Rep tasks from very small sets of labeled data—10s to 100s of examples vs 10k to 100k. To do this, we use the embeddings learned on self-supervision as input features to supervised tasks. We evaluate our approach on three tasks and datasets from prior work [CRH+20, LWJ+21, Bha19] and validate our findings across varying training set sizes. We show that our model consistently outperforms prior supervised approaches, significantly improving performance on smaller training sets. By using less data, our approach also proves substantially faster to train, making possible applications that depend on training speed. We believe that our differentiable CAD rasterizer paves the way to many exciting future applications, and show one possibility by prototyping a reverse engineering example.

6.2 Related Work

Learning from CAD Collections. Recent interest in learning from CAD data has grown due to advances in machine learning (ML) and the release of large CAD datasets. In addition to our own work on CAD assemblies suggestions in Chapter 4, researchers have released collections of B-reps and program structure [KMJ+19, WPL+20] and CAD sketches [SOZA20]. Prior work leveraged such collections for diverse applications, including segmentation [CRH+20], classification [Bha19], assembly suggestions [WJC+21, JHC+21a], and generative design of CAD sketches [SOZA20, GBL+21a, PBG+21b, SZRA21], B-Reps [JLD+22, GLP+22a], and CAD programs [WXZ21b, WPL+20].

However, a fundamental gap separates the capabilities shown in past work and real-world applications that can transform CAD design, such as auto-complete modeling interfaces or reconstruction of complex geometries: the lack of task-specific labels in large datasets. For example, most prior work leveraged the Onshape Public dataset[KMJ+19], which generally contains designs

created by novice CAD users and so does not capture the design process of CAD experts. The Fusion 360 dataset [WPL+20] is significantly smaller than Onshape's; other public resources, such as GrabCAD [Gra], contain designs from multiple CAD systems that are mostly unlabeled. In this work, we advance the development of new CAD learning applications by proposing a unique direction for leveraging unlabeled data in the supervised learning of CAD geometry.

Neural Shape Representations. Neural shape generation is an active research area with diverse representations used by prior techniques. Some methods operate over a fixed discretization of the domain into points [ADMG17, FSG17], voxel grids [BLRW16, LYF17], or vertex coordinates of a mesh template [TGLX18]. Due to irregularity of geometric data, functional representation is often used to learn to represent shapes as continuous functions, such as surface atlases [GFK+18, YFST18] or signed distance fields defined over a volume [PFS+19, MON+19, CZ19a]. In this work we chose to use functional representation of the output shape as a neural occupancy and a 3D mapping function over UV domains. Functional representation is well-suited for heterogeneous geometry with varied levels of detail since it does not require a fixed sampling rate to be chosen. Furthermore, our representation, which combines implicit fields and atlas-like embeddings, directly produces a surface and can be easily supervised with the ground truth B-Rep data.

Few-Shot Learning. Performance of strongly supervised methods is commonly hampered by the lack of training data. Few-shot learning techniques often rely on learning rich features in a self-supervised fashion and then using a few examples to adapt these features to a new task [WYKN20]. One common self-supervision strategy is to withhold some data from the original input and train a network to predict it. For example, one can remove color from images and train a network to colorize [LMS16, ZIE16], remove part of an image and train a network to complete it [PKD+16], randomly perturb orientation and predict the upright position [GSK18], or randomly

shuffle patches and predict their true ordering [DGE15, NF16]. Another commonly used tool is auto-encoders, which encode input to a lower-dimensional space and then attempt to reconstruct it with a decoder [KMRW14, KW13]. Our approach is closest to that of auto-encoders, except our input and output are in two different representations: CAD B-Reps and surface rasterizations. This lets us learn features related to the actual 3D geometry.

6.3 Geometric Self-Supervision

Our goal is to learn relevant features of CAD B-Reps that can be used on many different modeling and analysis tasks. We use an encoder-decoder architecture on B-Rep surfaces to learn a latent space of relevant features. Based on the insight that the learned features should include a geometric understanding of CAD shapes, we train our encoder-decoder to learn to rasterize CAD models. We further choose to learn this embedding at the face level as opposed to learning a feature per part. This is driven by our application domain, where tasks typically require an understanding of local topological information. Figure 6.1 shows our approach at a high-level.

Decoder. As noted previously, the format we selected for decoder output is driven by the observation that B-Reps are compositions of explicitly represented surfaces with implicitly represented boundaries. For example, the geometry associated with faces are unbounded parametric surfaces $S : \mathbb{R}^2 \rightarrow \mathbb{R}^3$, and the boundary of such surfaces is captured by the geometry of the neighboring edges. Edges, in turn, are defined by unbounded parametric curves bounded by neighboring vertices. Vertices are represented as points. In addition to domain parameters (u,v), parametric geometry has other, fixed parameters (like radius or cone angle), which we call *shape parameters*. A bounded surface is called a *clipped surface*, and the bounding function, which is defined *implicitly* by the bounding topology, is its *clipping mask*, illustrated in Figure 6.2 (c).

For self-supervision, we must choose an output geometric representation for surface patches with boundary. Bounded patches lack a natural parameterization and so are not suitable to learn directly as parametric functions. They are also difficult to represent as neural implicits [PSW+22]. However, the clipping function itself defines a closed region *within* the parameterization of the supporting surface, which is a function that lends itself well to implicit representation in 2D. We therefore choose to learn an implicit function of the clipping region as a 2D signed distance function. Since this representation relies on an explicit surface parameterization, we also learn the supporting surface parameterization. Crucially, this does not require parameterizing a boundary.

We choose to use a conditional neural field as our decoder since this representation can capture both explicit and implicit geometry. Specifically, the explicit parametric surface is an $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ function, mapping (u, v) coordinates of a face's supporting surface to the 3D position of that point (x, y, z). The clipping mask encoded as an SDF over the parametric domain is a function $\mathbb{R}^2 \rightarrow \mathbb{R}$, which maps the same (u, v) coordinates to a value (d) measuring the signed distance to the boundary. We combine these two function to learn a single field that maps $\mathbb{R}^2 \rightarrow \mathbb{R}^4$. We parameterize this field as a 4-layer, fully connected ReLU network, where the input coordinates and conditioning vector are concatenated to the output at every hidden layer (similar to DeepSDF [PFS+19], see Appendix D for a full definition). The conditioning vector is the output of our face encoder described below. We call the evaluation of this field "rasterization" because raster sampling the field and filtering by d yields a 3D surface rasterization.

To normalize the field input range and constrain the uv-space, we must sample while rasterizing. We reparameterize the uv-space of each surface prior to training so that the clipping mask fits within the unit square ($[0, 1]^2$). In this way, our encoder-decoder is learning the explicit surface, the implicit boundary, and the support range of the implicit boundary mask.

Encoder. Our encoder design is driven by the same observation, i.e., that B-Reps are compositions of explicitly represented surfaces with implicitly represented boundaries. For the encoder, this means that we must capture adjacent topological entities. We propose to encode this using message passing on the topological graph.

As Figure 6.3 shows, the B-Rep topological graph has a hierarchical structure; high-dimensional topological elements are adjacent only to the immediate lower dimensional entities: faces are adjacent to their bounding edges, which are adjacent to their end-point vertices. Driven by this observation, we use as our face-encoder a hierarchical message passing network inspired by SB-GCN.

Our encoder is structured as two graph message passing layers, first aggregating vertex information for the edges they bound, and then this bounded edge information for the faces. Crucially, we need to (1) support both node and adjacency level features in our message passing — since whether a vertex is the start or end of an edge, and whether an edge is to the left or right of a face interior is critical to correctly reproducing that edge — and (2) support highly variable node degree since faces vary widely in the number of bounding faces. To achieve these goals we use multi-headed additive graph attention [VCC+17] with graph-adjacency features.

Existing B-Rep representation learning networks [JHC+21a, JSL+21, LWJ+21] use evaluated geometry information as input features in addition to pure parametric geometry, such as surface bounding boxes and areas. These features are computed by a CAD kernel, i.e., modeling software that understands how to construct and evaluate CAD geometry. Since we want to force our network to learn the evaluation function of a CAD kernel as well as be fully differentiable back to the shape parameters, we only use the parametric geometry definitions as input features. Please see Appendix D for the full details of our encoder architecture and input encodings.

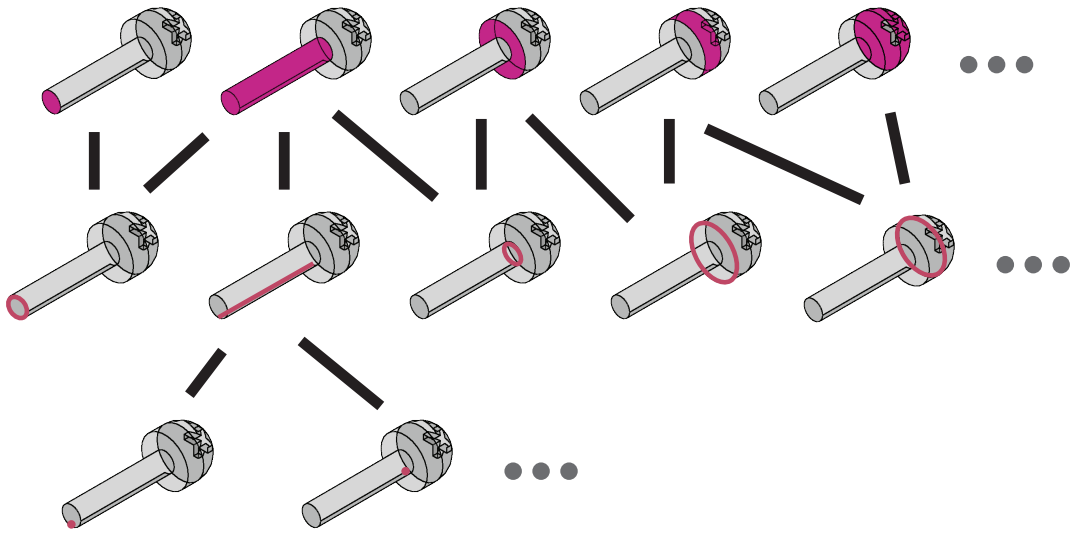


Figure 6.3: A boundary representation consists of parametric surface patches that are bounded by parametric edges, which in turn are bounded by vertices. The relationship between these entities forms a hierarchical graph.

Training. We trained our encoder-decoder to minimize L^2 losses at randomly sampled uv-points on each face's supporting surface. We randomly sampled in the re-parameterized uv-square $(u, v) \in [-0.1, 1.1]^2$ to ensure positive and negative SDF samples were collected, biased to preferentially sample near clipping boundaries (see Appendix D for details).

We trained our geometric self-supervision over 23266 shapes from the training set of the Fusion 360 Gallery Segmentation dataset [WPL+20]. Optimization was performed using the Adam optimizer [KB14] with a learning rate of .0005. For our experiments, we used an embedding size of 64 for all message passing layers, 2 attention heads for the vertex-to-edge layer, 16 attention heads for the edge-to-face layer, and a hidden size of 1024 for all self-supervision decoder layers.

6.4 Few-Shot Learning

In typical task-specific applications, it is challenging to find or construct large collections of labeled CAD data. Therefore, we propose leveraging our rich latent space to enable supervised learning over very small collections. In addition to enabling training on scant data, our approach also ensures that we do not need to use computationally expensive CAD kernel functionality to generate features at inference time.

We propose two few-shot learning setups. The first consists of B-Rep segmentation tasks, which assign a task-specific label to each face of a B-Rep, e.g., the type of machining technique that can be used to construct that surface. The second consists of B-Rep classification tasks, which assign a label to an entire B-Rep, e.g., the mechanical function of that part (gear, bracket, etc.). We frame both setups as multi-class classification problems. See our Appendix D for full network specifications.

B-Rep Segmentation Network. Since our encodings are learned at the face level, they can be used directly as input for face-level predictions. Classification of a face often depends on its context within a part; we therefore use a small message passing network to capture this context, i.e., a 2-layer Residual MR-GCN [LPBM20]. We use pre-computed face embeddings from our geometric self-supervised learning as node features and face-face adjacency as edges. We construct this graph by removing vertex nodes from the B-Rep graph and contracting edge nodes, preserving multiple edges (if any) between faces. Output predictions for each face are then made with a fully connected network with two hidden layers.

B-Rep Classification Network. While we do not have latent codes for entire B-Rep shapes, we take a similar approach to previous B-Rep learning architectures and max-pool learned features for each face. To do this, we project each face's embedding into a new vector for pooling using a fully

connected layer, followed by a second projection of the pooled part features into the prediction output space.

6.5 Results

We validate the application of our proposed approach by applying our method to three few-shot learning tasks. We further evaluate our method by analyzing rasterization results.

6.5.1 Construction-Based Segmentation

The first task we apply our method to is segmentation of B-Rep geometry by the modeling operation used to construct each face (extrusion, revolution, chamfer, etc.). This requires the model to understand how geometry is constructed in CAD software. For this task we use 27450 parts from the Fusion 360 Gallery dataset, a collection of user-constructed parts annotated with one of 8 face construction operations on each face [LWJ+21].

We first pre-trained our geometric self-supervision network over this dataset without labels, then trained our face-level prediction network using the face embeddings from the self-supervision. Figure 6.4 shows some classification results. Our method achieves 65% accuracy after seeing just 10 training examples, 79% after 100, and is 96% accurate when given all 23266 examples in the training set.

We compare our method to three network architectures for B-Rep learning: SB-GCN, BRepNet [LWJ+21], and UV-Net [JSL+21]. Compared to our architecture, SB-GCN uses an intermediate *loop* layer to aggregate closed paths of edges, and uses both the parametric geometry definitions and kernel computed features (bounding boxes and mass properties) as input features. BRepNet defines a convolution operator relative to co-edges in a B-Rep structure and uses parametric face and edge types as well as concavity and edge length features. UV-Net is a message passing

network between adjacent faces; it uses grid-sampled points of faces and edges as its features. For UV-Net, we train an end-to-end version, as well as a self-supervised version using the part augmentation supervision and contrastive loss described in [JLD+22] (UV-Net-SS). For UV-Net-SS, we use the same task-specific network as our method to fairly compare the self-supervision. We trained and tested each network on random subsets of the training data that ranged from 10 to 23266 examples. We repeated this 10 times for each training set size (all three methods were given the same training sets).

Figure 6.5 plots the average accuracy for each training size across these runs with a bootstrapped 95% confidence interval. Our method outperforms the baselines for all dataset sizes, doing so by a significant margin in the few-shot regime. It significantly outperforms UV-Net with self-supervision, indicating that our geometric self-supervision is much more effective than a contrastive loss setup. It also has a smaller confidence interval, indicating that pre-training on our rasterization task makes classifications more robust to choice of training example.[1] In addition to these quantitative results, Figure 6.6 compares classifications at the 100 example level. Our method starts generalizing with 10s to 100s of examples, whereas the baselines fail to generalize and usually learn the most frequent label at this data scale.

We further note that training our method is significantly faster to train than the baselines (see Figure 6.7). This raises the possibility of segmentation tasks trained on-the-fly to enable predictors to be trained and deployed during the modeling process.

6.5.2 Manufacturing-Driven Segmentation

The second segmentation problem we considered is face segmentation, which classifies how each face is manufactured. We evaluated this on the MFCAD dataset [CRH+20], a synthetic dataset of

[1] BRepNet's very small confidence interval at small training set sizes is an artifact of it predominantly or entirely predicting one class.

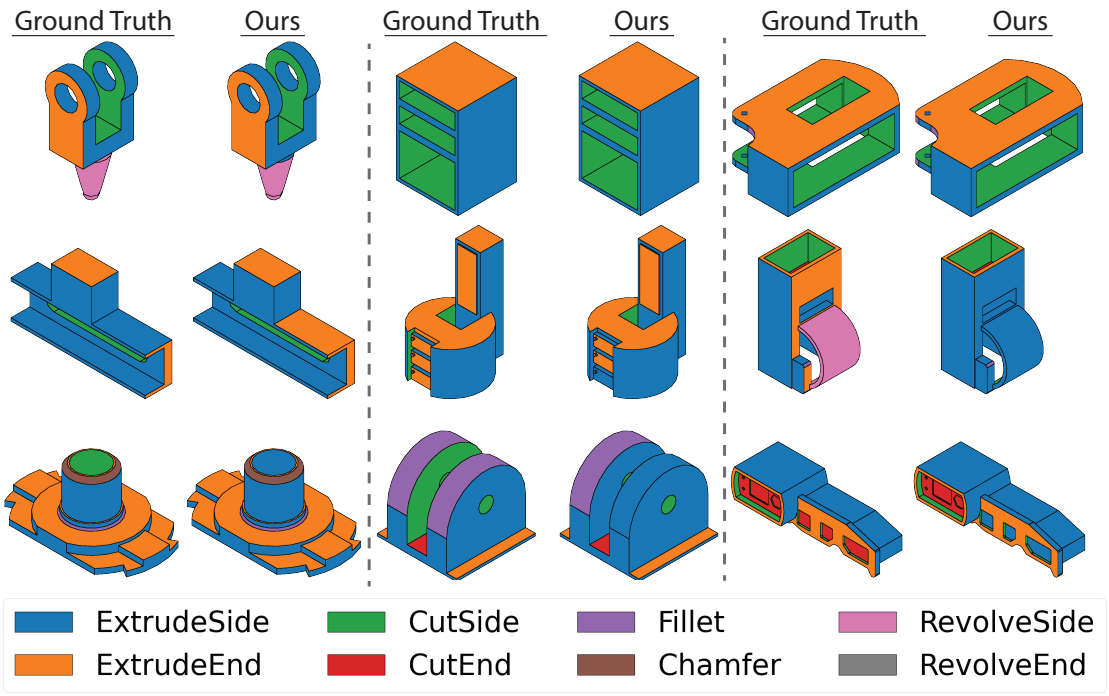


Figure 6.4: Segmentation example results for the Fusion 360 gallery task. The CAD operation that created each face is indicated by color. (Left) Our model's prediction. (Right) The ground truth.

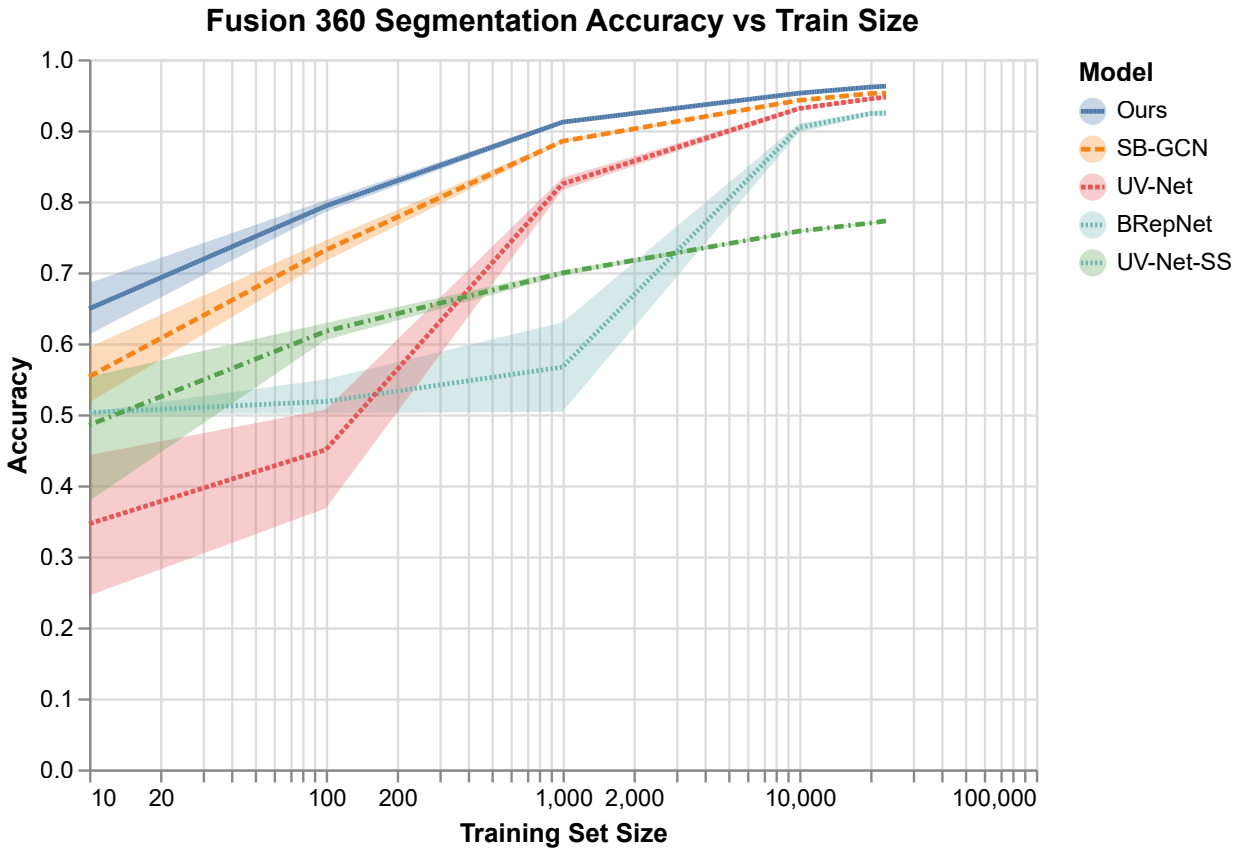


Figure 6.5: Comparison of our method versus SB-GCN, UV-Net, BRepNet, and Self-Supervised UV-Net (UV-Net-SS) for construction-based segmentation on the Fusion 360 Gallery Segmentation dataset. The mean accuracy across 10 training runs is plotted with a bootstrapped 95% confidence interval.

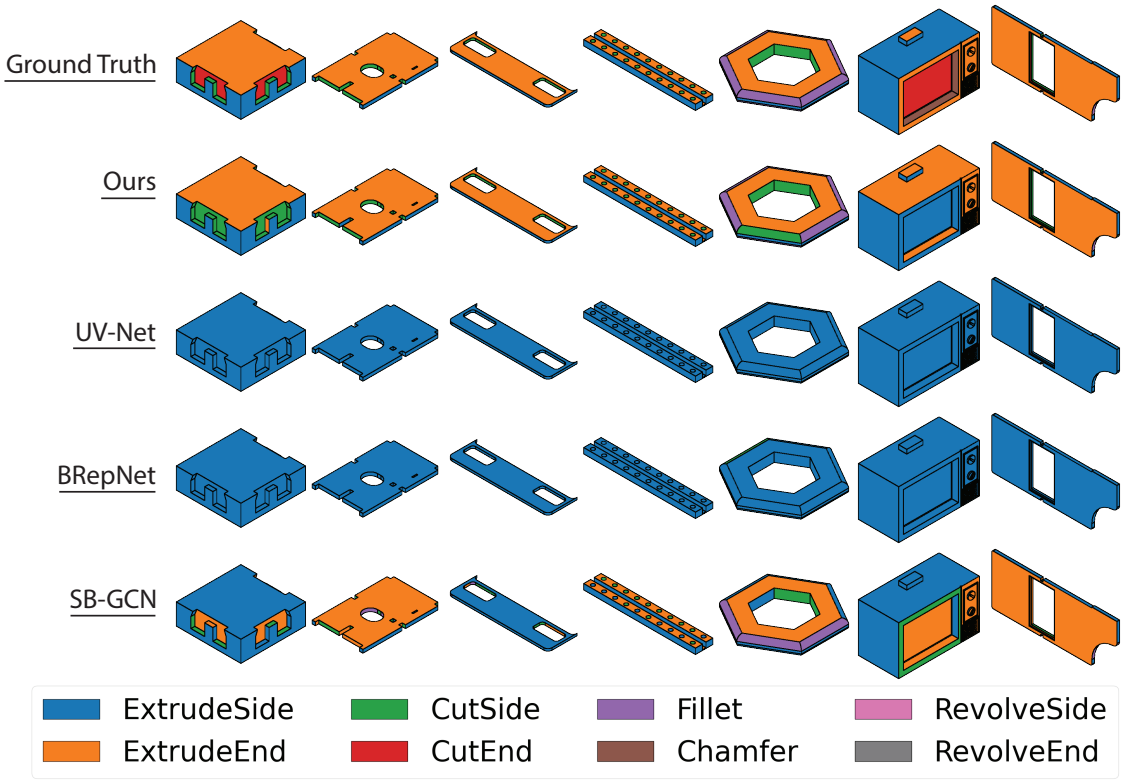


Figure 6.6: Few shot segmentation results from our method versus baseline models on the Fusion 360 Gallery task, trained over just 100 models. Face color indicates the modeling operation used to create a face. Our model performs significantly better in the low data regime, where it can differentiate operations. However, the baselines largely guess the most common operation, which is ExtrudeSide.

Figure 6.7: Training times on the Fusion 360 segmentation task. All models were trained on an NVIDIA RTX 2080 Ti, and reported time is the average time per epoch at each training set size.

15488 CAD models where each face is labeled with one of 16 types of machining feature (chamfer, through hole, blind hole, etc.) The parts in this dataset are generated by applying machining operations to square stock and so consist entirely of planar faces and straight line edges. Compared to the Fusion 360 Gallery dataset, this task has twice as many classes to choose from and relies more on neighborhood and boundary information since all faces are planar. It also shows the ability of our approach to generalize pre-trained features to out-of-distribution data since we use face codes pre-trained on Fusion 360 Gallery models (in this and all examples), which unlike MFCAD are all human-modeled.

We compare this task to the same four baselines, illustrated qualitatively in Figure 6.8 and quantitatively in Figure 6.9. We removed curvature features from BRepNet since they are universally zero on this dataset, and BRepNet requires all input features to have non-zero standard deviation. As before, our method outperforms the baselines at few-shot learning and achieves comparable accuracy at large data sizes (Ours, UV-Net, and BRepNet are essentially perfect given sufficient training data). UV-Net performs slightly better with more samples (1000); we hypothesize this is partially due to a domain gap between human-modeled parts of the Fusion 360 Gallery dataset we used in our self-supervised learning stage, and the synthetic parts of MFCAD, which might enable competing techniques to learn how to use features specific to the synthesis process to their advantage. The stability of our prediction accuracy as measured by confidence interval significantly exceeds both baselines.

6.5.3 Part Classification

In addition to segmentation, we also applied our method to part classification. For this, we used the FabWave [Bha19] dataset, a hand-labeled subset of GrabCAD [Gra] that is categorized into classes of mechanical parts (gears, brackets, washers, etc.). Many parts within a category are

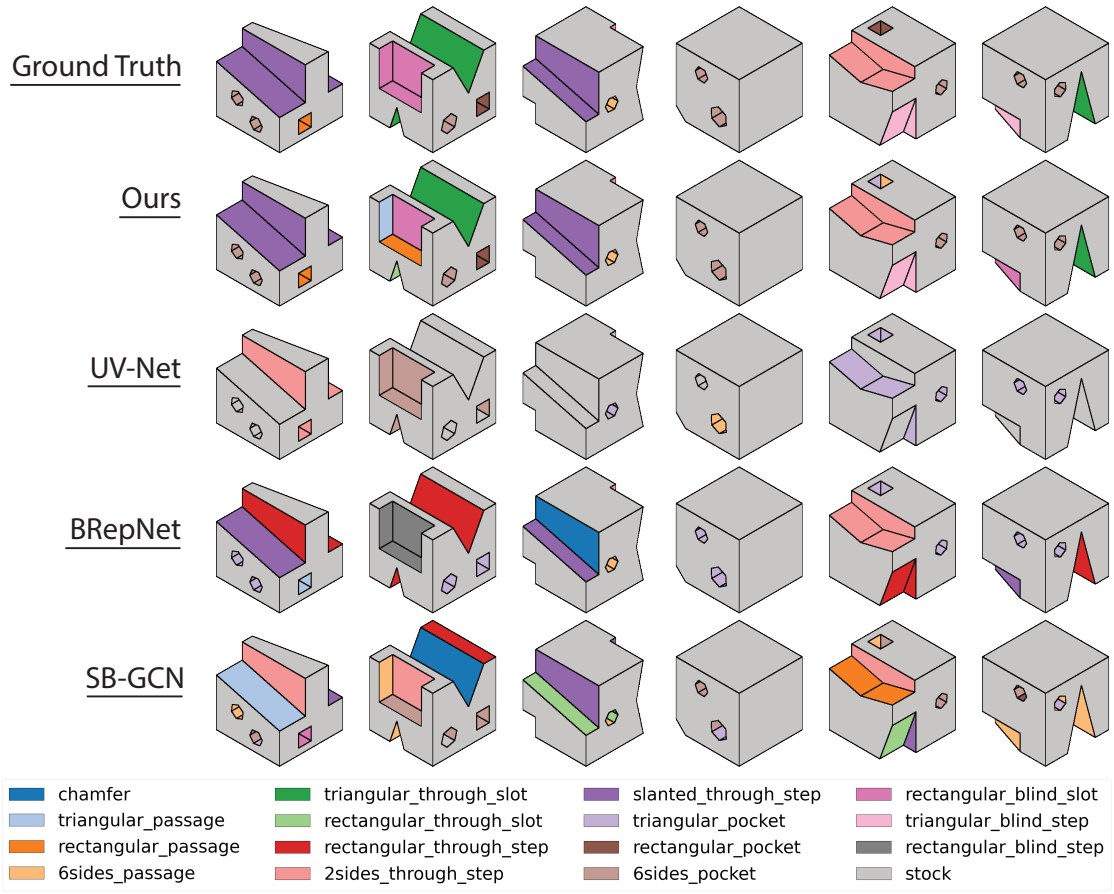


Figure 6.8: MFCAD few-shot segmentation comparison at 100 train samples. Top row shows ground truth labeling, followed by our, UV-Net, and BRepNet predictions.

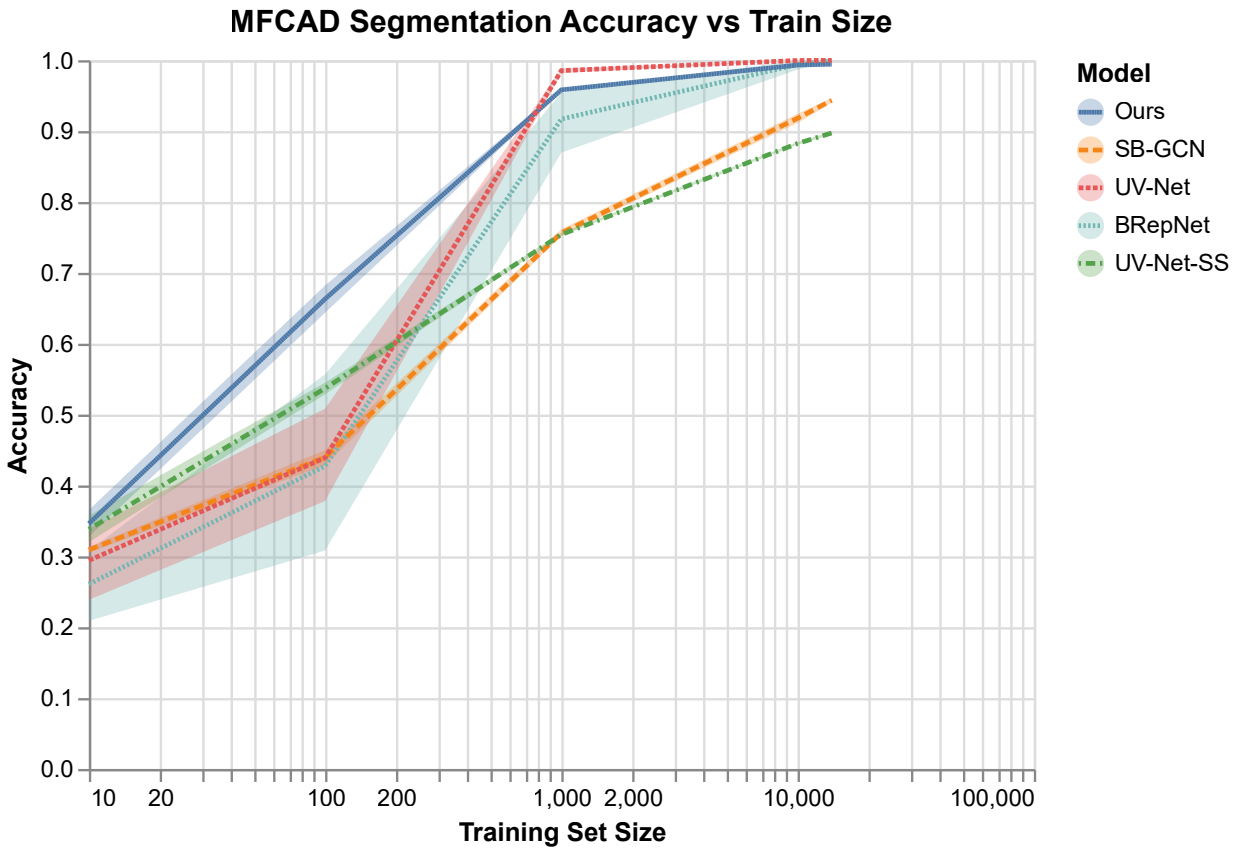


Figure 6.9: Comparison of our method to SB-GCN, UV-Net, BRepNet, and Self-Supervised UV-Net (UV-Net-SS) for manufacturing-based segmentation on the MFCAD dataset. The mean accuracy across 10 training runs is plotted with a bootstrapped 95% confidence interval.

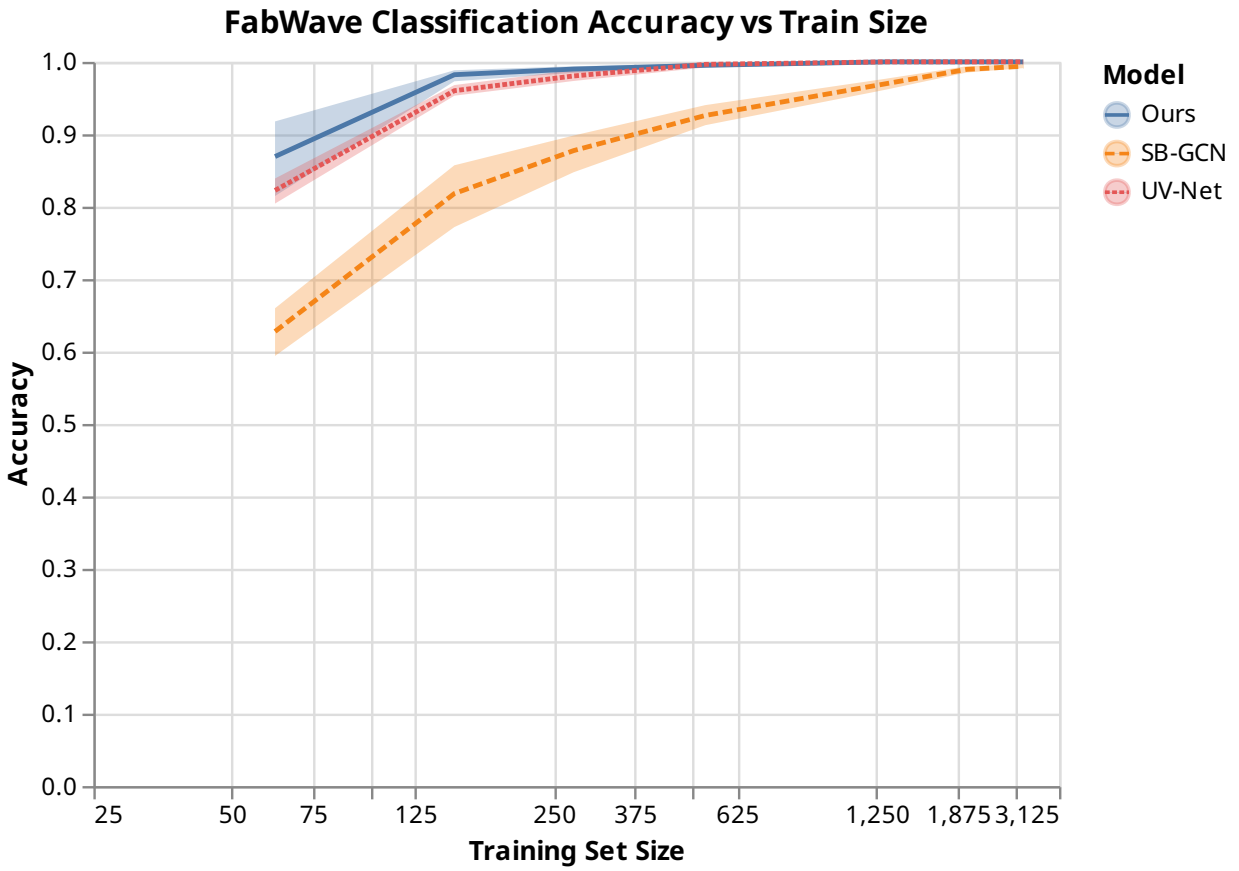


Figure 6.10: Comparison of our method to UV-Net for part classification on the FabWave dataset. The mean accuracy across 10 training runs is plotted with a bootstrapped 95% confidence interval.

parametric variations of each other. This task is important for understanding the function of mechanical parts in assemblies.

As a baseline comparison, we compare only against SB-GCN and UV-Net since BRepNet does not support classification. Again using face codes pre-trained on Fusion 360 Gallery B-Reps, we train our B-Rep Classification Network over the 26 classes that have at least 3 examples compatible with our network (for train, test, and validation); we use stratified sampling to create training subsets that contain at least one part from each category in each split. Since UV-Net cannot be run on B-Reps that lack edges, we restricted our test set only to models with edges; we also removed 2

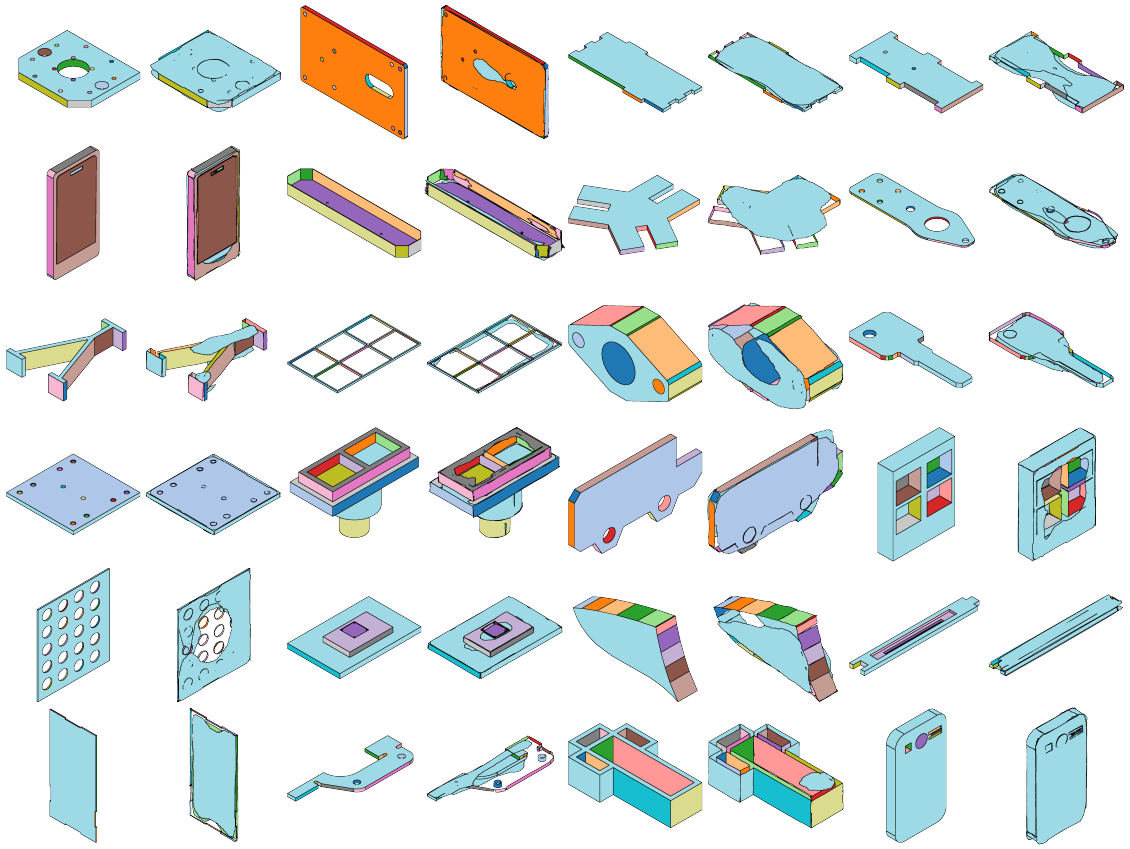


Figure 6.11: Shape reconstructions (right) from face embeddings on the Fusion 360 segmentation test set compared to ground truth (left). Each B-Rep face is given a unique color, which is consistent between ground truth and reconstruction.

classes that UV-Net could not distinguish since they differ only in part orientation (our technique can differentiate these classes, so keeping them in the evaluation would both improve our accuracy and decrease UV-Net's). Figure 6.10 shows the results. Our method slightly outperforms UV-Net across training sizes and significantly outperforms SB-GCN, and even achieves 100% accuracy at higher training set sizes.

6.5.4 Rasterization Evaluation

Figure 6.11 shows a collection of part renderings to illustrate qualitatively our reconstruction and classification results. It shows a gallery of rasterization results on unseen test examples from the Fusion 360 Gallery segmentation dataset. Reconstructions were created by sampling a 100x100 (u,v) grid in the range [-0.1,1.1] for each face to create a mesh for the supporting surface, then removing mesh vertices outside the predicted clipping plane SDF. Supporting surface reconstructions are highly accurate, as are the clipping masks for typical shapes with a single boundary. Complex interior and exterior boundaries of clipping planes are sometimes predicted inaccurately.

6.5.5 Ablations

Self-Supervision Ablations. In addition to the network described in Section 6.3, we also tried using a truncated SB-GCN (only its upwards pass) as the encoder network, using the same shape parameter input features. We evaluated both explicit surface and SDF accuracy; our encoder outperforms SB-GCN by 27% in the former metric and 31% in the latter.

Segmentation Ablations. We tried three types of face-level prediction networks using our self-supervised face embeddings: directly classifying faces from the embedding using a linear support vector machine (SVM), using a multi-layer perceptron (MLP), or using the message passing scheme described in Section 6.4 (MP) to test if neighborhood context is necessary. Table 6.1 shows the results of these experiments. All methods performed similar with little training data, but with more data we observed up to 36% improvement from SVM to MLP, and up to additional 10% from MLP to MP. At very low training sizes, using an SVM did outperform other methods, indicating that for certain very-low data tasks it may be a better way to apply geometrically self-supervised features. We chose to use message passing in our comparisons because it performed the best

Task / Model	Training Set Size / Accuracy					
Fusion 360 Seg.	10	100	1000	10000	20000	23266
SVM	**0.66**	0.79	0.85	0.87	0.87	0.87
MLP	0.65	**0.80**	0.90	0.94	0.94	0.94
MP (Ours)	0.65	0.79	**0.91**	**0.95**	**0.96**	**0.96**
MFCAD	10	100	1000	10000	13940	–
SVM	**0.40**	0.51	0.56	0.57	0.57	
MLP	0.36	0.60	0.86	0.93	0.93	
MP (Ours)	0.35	**0.66**	**0.96**	**0.99**	**0.99**	

Table 6.1: Segmentation ablations. Reported face segmentation accuracy shows the mean of 10 runs at each dataset size with the train set subset at different random seeds (each model sees the same 10 random subsets). Models were selected by best validation loss on a random 20% validation split. Bold indicates the best accuracy at each train size for each task.

across a range of training set sizes.

Classification Ablations. We also tested adding message passing layers prior to pooling for the classification network, but found that this additional complexity did not yield any improvement.

Further details about our ablations experiments and full result tables are in Appendix D.

6.5.6 Limitations

Our work has three main limitations. The first is that we currently only support geometry with a fixed number of shape parameters (e.g. no construction geometry or B-splines), allowing us to support 77% of the Fusion 360 Segmentation dataset. The limitation is due to the choice to use fixed-size vectors as our input feature encoding for simplicity. It could be alleviated by using a sequence or tree encoder to compute fixed-size embeddings for the generic functional geometric expressions of each B-Rep topology.

The second limitation is that we self-supervise only on local information. This means that we rely on additional message passing layers in our task-specific decoders to gather neighborhood features. Finally, we create encodings only for faces, which limits the kinds of tasks we can learn.

Adding a second decoder and loss term to rasterize edges could extend this work to edge-based tasks. We did not add this complexity since there are currently no edge-specific tasks in the literature to compare against.

6.6 Conclusion

This self-supervised representation has two key benefits over end-to-end training with SB-GCN; it enables learning from small collections of labeled data, and it is independent of the CAD kernel and thus fully differentiable back to the input parameters.

Differentiability This makes our network, in essence, a *differentiable renderer* for B-Reps. There are two very exciting implications of this. The first is that direct gradient-based optimization of B-Reps shape-parameters is, in principle, possible. Appendix D demonstrates this with a simple shape-fitting prototype. The second is that differentiably rendering a B-Rep model is one half of the way to a differentiable B-Rep geometry kernel, the other half being a method for differentiable applying CAD operations (extrude, revolve, etc.). Chapter 7 will present a system that uses a differentiable CAD renderer to improve the explorability of CAD models, but it uses a non-B-Rep geometry format (CSG) to work around the lack of feasible B-Rep based kernels.

Few-Shot Learning Opportunities The benchmark datasets we demonstrated our system against are primarily targeted at implementation tasks. Construction-Based Segmentation is a step towards B-Rep reverse engineering, which will enable existing B-Rep models to be used as a implementation starting-points even when their CAD program is unavailable. Manufacturing-Driven Segmentation assists in implementing a design through to a manufacturing plan or physical prototype. And Part Classification, as well as an unsupervised part retrieval prototype in Appendix D

assist in assembly modeling by finding mix-and-match parts.

These implementation tasks have significant datasets available, but because design ultimately aims to produce novel objects, novel tasks should be expected to arise. On the implementation side, autocompletion of references similar to Excel's FlashFill [CGL+23] is a compelling application. AutoMate demonstrated the power of autocompletion for the repetitive task of mating, and CAD modeling is replete with similarly tedious repetition, such as applying fillets and chamfers, or offsetting surfaces while exploring tolerances.

Few-shot learning also has the potential to turbocharge the evaluation part of CAD design. Design objectives are both unique to the problem at hand, and also evolve as the designer learns more through iteration, so it is inevitable that many evaluation metrics will have a paucity of available examples. Enabling few-shot learning over B-Reps will allow AI design assistants to learn evaluation metrics from examples.

From B-Reps to CAD Programs Over the previous four chapters I have developed and refined a learned geometry representation that complements the symbolic B-Rep geometry common to most modern CAD systems. Due to the structure of the CAD design loop, this enables improvement to the implementation and evaluation stages of design, but is not as impactful to ideation and exploration. The following chapters will explore B-Rep's complementary representation, the CAD programs that the designer works with directly, and that are ultimately used to define B-Rep geometry.

Chapter 7

ReparamCAD

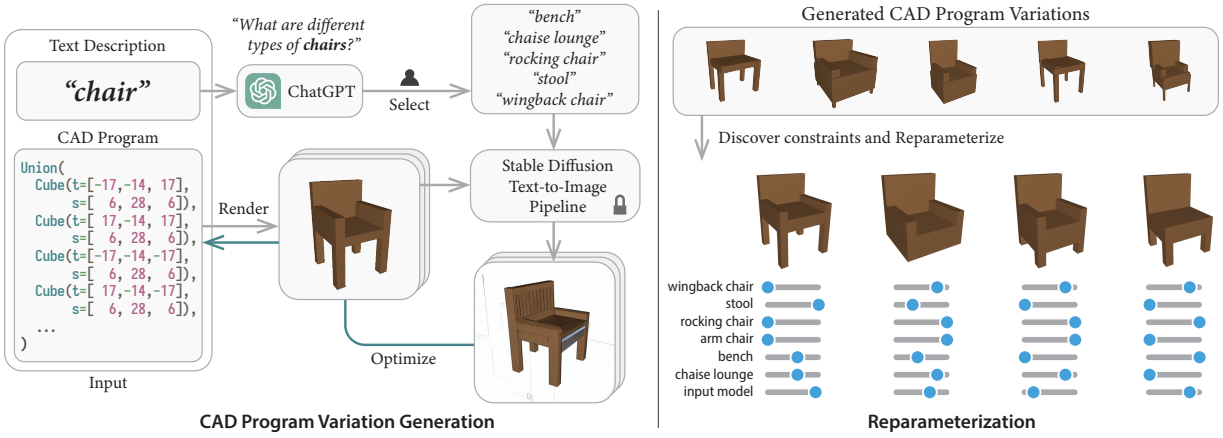


Figure 7.1: ReparamCAD takes a parametric 3D model and a corresponding text description as input and generates a re-parameterized version of the model as output. It uses ChatGPT and user guidance to generate text prompts describing variations of the input model and uses stable diffusion to optimize the CAD parameters towards these prompts. The resulting variations are used by the constraint discovery system to identify common constraints across all designs.

7.1 Introduction

A core promise of parametric CAD design is that it will enable rapid iterations through the design loop by making model editing as simple as manipulating CAD program parameters. Unfortunately, practical parametric CAD falls short of this lofty ambition. The core reason for this is that parametric CAD design has a chicken-and-the-egg problem. CAD programs do not automatically contain parameters for any variable a designer may wish to manipulate. Rather, the designer must carefully consider which parameters they want to have control over, and construct the CAD program around the goal of exposing these parameters. But the design process requires cycles of prototyping in order to know which parameters are of interest in the first place! The best that can be done in practice is attempt to map the existing parameters of a program onto the manipulation intent. For instance, if we model the back of a chair as an extrusion of a base 2D rectangle, we can manipulate the height of the chair by varying the extrusion distance. In practice, modifying CAD parameters directly can be challenging due to a lack of user understanding regarding which parameters to modify to achieve the desired variation. In addition, the absence of constraints can lead to undesirable shape changes that violate user intent.

To address the gap in CAD manipulation, this paper aims to automatically construct a reparametrization of CAD models, introducing what we refer to as *manipulation* parameters. The parameters generated by sequences of constructive CAD operations will be referred to as *constructive* parameters to differentiate them from the proposed abstraction. We observe that CAD programs are typically overparameterized, as multiple constructive operations may be required to create structurally related geometries. However, constraints across constructive parameters are frequently absent or under specified during the CAD modeling sequence, and feasible ranges for constructive parameters are not exposed. Consequently, modifying a single CAD constructive parameter can lead to shapes that lack the essential structure, such as a chair with its legs

disconnected from its base. Thus, to achieve meaningful design variations, users often must simultaneously and consistently modify multiple constructive parameters [Yar13]. Essentially, the space of shape variations defined by constructive parameters predominantly consists of irrelevant outcomes, rendering the extraction of meaningful design variations from this space an arduous and time-consuming process. This work explores the possibility of automatically identifying a constrained subspace within a CAD program that reflects meaningful shape variations. We frame this as a reparametrization problem from *constructive* to *manipulation* parameters.

On one hand, we anticipate that program analysis will shed light on the constraints to consider when constructing this subspace. On the other, this problem is inherently ambiguous: meaningful design variations are ultimately derived from *what* the designer is trying to model, rather than *how* they are modeling it. Any synthetic analysis would inevitably fail to infer semantic meaning. Our key insight is to first develop an understanding of how we may want to manipulate the shape, and subsequently conducting an analysis based on that understanding, to derive a constrained shape space. We note that establishing this understanding is now possible because of novel pre-trained large foundational models [RBL+22] that have learned the space of reasonable shapes. Building upon this insight, our neurosymbolic approach combines AI-driven induction for discovering shape variations with symbolic-driven deductive reasoning for identifying shape constraints. Most notably, our approach operates in a 'zero-shot' manner, eliminating the reliance on large categorical shape datasets. As such, our method is applicable beyond the sparse shape categories covered by existing datasets.

Our novel system ReparamCAD, illustrated in Figure 7.1, takes a parametric model expressed in a simplified CAD language, along with a concise text description of the model. It generates a re-parameterized version of the input CAD model, accompanied by an intuitive slider-based interface allowing users to vary the manipulation parameters introduced in the revised CAD program, and empowering them to easily explore meaningful design variations.

The re-parameterization process begins by generating text prompts describing variations of the model using a large language model and user guidance. Next, we apply our novel method to automatically adjust the parameters of the input model, aligning it with each text prompt by comparing the rendered images of the model with images from a pre-trained stable diffusion text-to-image model. The design variations generated by matching the input model to various text prompts are then fed into our constraint discovery system. This identifies geometric constraints that are common across all variations, accounting for noise. The discovered constraints are used to construct a subspace of the original CAD parameter space, automatically imposing semantically meaningful constraints. Finally, we project the generated variations into this subspace and use them as the basis for a new parameterization of the constrained space along semantic lines.

We demonstrate the efficacy of our approach in generating variations for five distinct models of varying complexity. We conduct a comparative analysis between our neurosymbolic approach and purely symbolic or purely neural-driven methods to underscore the advantages inherent in our approach; the former creates uninteresting variations and the latter can produce incoherent geometry, but our approach produces interesting variations while adhering to semantically meaningful constraints. Furthermore, we conduct a user study and show that our approach discovers similar constraints to what our participants have specified.

7.2 Related Work

Our work builds upon a rich body of work on CAD manipulation as well as structure-preserving shape manipulation. This includes algorithms that operate on input shape collections of a specific class, as well as methods for manipulating individual models. Furthermore, our work builds on text-driven shape generation algorithms.

7.2.1 Parametric CAD Manipulation

Parametric CAD systems represent designs as programs that expose constructive parameters. Manipulating CAD models solely by adjusting these parameters can be challenging due to the need for coordinated changes across multiple parameters to achieve specific design goals [Yar13]. This has encouraged efforts that diverge from the parametric modeling paradigm, proposing interfaces that allow *direct manipulation* instead (e.g. SpaceClaim, KeyCreator, and Rhino). Nevertheless, most CAD systems prioritize preserving program information as it enables the preservation and control of global structures (e.g., Solidworks, Onshape, Catia, Creo and NX).

Recent efforts have focused on exploring hybrid techniques that aim to bridge the gap between the parametric programming paradigm and direct manipulation approaches. This includes commercial systems like Siemens' Synchronous Technology and IronCAD, which aim to facilitate direct manipulation by utilizing complex algorithms to maintain synchronization with the program representation. Efforts within the computer graphics community have made advancements in enabling program updates based on user interactions [CSQ+21, MB21], optimizing program parameters to align with user manipulation. The fundamental challenge with these approaches is that they rely on hand-crafted heuristics to resolve the inherent ambiguities in the system. There are often multiple viable constraints that can be imposed over the program parameters to achieve the edits that adhere to users' manipulation. To eliminate the need for hand-crafted heuristics, we propose leveraging semantic understanding extracted from large pretrained foundational models. By utilizing these models, we can uncover a meaningful set of potential variations for a CAD model and derive constraints directly from those examples.

7.2.2 Enabling Manipulation from Large Shape Collections

A considerable body of research explores methods for understanding the meaningful space of variations within categorical shape collections. These approaches either create an abstraction for a collection of shapes belonging to a specific category or enable manipulation of an image based on a collection of models within that category.

Early approaches combine statistical models, with label-driven shape decomposition [CKGF13, FAVK+14, OLGM11]. Additionally, labels have been used to learn semantic abstractions from shape collections [YCHK15].

More recently, neural networks have been used to learn embeddings that enable design exploration. These approaches do not require segmented labels, but the learned embeddings are not easy for a user to explore [CZ19b, ZLWT22]. To enable user control, several approaches have integrated learning with structural, compositional, and symbolic abstractions. This includes efforts focused on learning abstractions [JHHR22, TSG+17], fitting shapes to categorical abstractions [PLH+22, WSS+20], enabling structure manipulation through handles or mixing and matching [MGY+19a, HAESB20, YCC+20b, JHTG20, HPG+22, LSMS21], and text-driven variations [AHS+22].

Closest to our approach are methods that infer higher-level abstractions that preserve program structure [JCG+21, JGMR23]. Library learning techniques, based on machine learning [EWN+21] or anti-unification [CKN+23], can extract common structure from a corpus of CAD programs into reusable functions that expose more semantically meaningful parameters.

The fundamental limitation of these approaches is that they require a large dataset of models belonging to a specific class, from which meaningful space of variations can be inferred. Rather than being confined to specific classes of objects that are covered by existing datasets, we leverage the much broader understanding embedded in foundational models to infer meaningful variations

from a single input model.

7.2.3 Structured Manipulation from Single Input Shapes

Past research has also devised methods for structure-preserving shape manipulation when only a single input shape is available. Earlier methods used hand-crafted heuristics with numerical optimization to enable shape deformation (see [MWZ+14, SB09] for a more complete overview). While some geometric and physics-inspired heuristics are well suited for organic shapes [IMH05, SCOL+04], heuristics based on geometric-semantic constraints such as symmetry, coplanarity, and replicable patterns have been shown to work well on man-made shapes [BWSK12, GSMCO09b]. These heuristics have been used in two types of editing systems: 1) *variational methods*, where optimization is used to compute a deformation that adheres to the user manipulation [SSP07]; and 2) *direct methods* where the computation is done in advance to generate a set of exposed controls [JBPS11]. Such controls can be in the form of parameter sliders, cages, skeletons, or compositions thereof. Essentially, direct methods generate a type of reparametrization, as we describe in this work.

More recent approaches apply learning approaches for manipulating shapes. However, while approaches that assume categorical data sets can use learning to replace heuristics [SJA+20], as discussed above, efforts that take a single shape as input are more restricted. Most successful efforts focus on a constrained task of matching the input model to a target shape or image [WCMN19, WZL+18]. Some efforts have been made on learning abstractions that are category independent, such as learning to fit cages [YAK+20] and inferring 3D shape programs from a target image or 3D geometry [DIP+18a, NWP+18, YCL+22a, JBX+20b] (see [RGJ+23] for a complete overview). Similarly, domain-specific compilers have been developed that strive to reduce the number of parameters in the model by re-writing CAD programs concisely using looping constructs

[NWA+20]. Such methods still focus on lower-level abstractions, essentially producing the types of programs we take as input.

7.2.4 Text-conditioned 3D generation

Several studies have explored text-conditioned 3D generative models trained directly on text-3D pairs. Most of these approaches rely on learning 3D latent representations and establishing associations between text and 3D embeddings [CCS+19, LWQF22, SCL+22, zVW+22, MCST23, FZC+22, SFL+23]. However, scaling these methods to accommodate diverse text prompts is difficult due to the lack of large-scale 3D datasets.

A growing body of research focuses on text-conditioned 3D generation, using pretrained text-to-image models like CLIP [RKH+21], as well as diffusion-based models [NDR+22, RBL+22, SCS+22]. Differentiable rendering techniques are also used to optimize 3D representations such as meshes [KXBP22] and NeRFs [JMB+21, PJBM22, LGT+23]. These methods tend to struggle to generate coherent 3D objects due to lack of strong 3D priors. Recent approaches attempt to solve this problem by generating a synthetic dataset of image-3D pairs. These methods use learning to generate the initial coarse 3D objects from images, which then serve as a starting point for further refinement through fine-grained 3D shape optimization [XWC+23, NJD+22, SJK+23].

To the best of our knowledge, our work is the first to tackle text-conditioned 3D synthesis within the domain of CAD programs. While we also leverage differentiable rendering and diffusion-based models, our focus lies on the problem of distilling the inherent constraints on CAD parameters from a large pretrained model.

7.3 Methods

Given a well-formed CAD model represented as a simplified constructive solid geometry (CSG) and a categorical description of the model ("chair"), our system synthesizes a new CAD program with fewer parameters capable of reproducing the input space and expressing meaningful semantic variations. We first use a large language model to generate text description of variations of the given object ("bench", "stool", etc.). We then optimize the parameters of the input CAD program to fit these variation prompts using diffusion-generated images as a guidance to a differentiable renderer. From these instances of model variations, we infer constraints and reparameterize to a CAD program that exposes meaningful manipulation parameters to the users.

7.3.1 Simplified CAD Language

Our simplified CAD language is built upon Constructive Solid Geometry (CSG), which forms the basis of popular software like OpenSCAD, and is a supported mode of operation in most commercial CAD systems. While modern CAD systems predominantly employ B-rep history-based languages, the essential boolean operations of CSG persist. To simplify the implementation of a differentiable renderer, we have opted to include only union operators in our language, excluding intersections and subtractions. Despite this restriction, our language still allows for a wide range of CAD designs, showcasing the capabilities of our method. Specifically, we have implemented operators for three primitives (cubes, cylinders, cylinders with changing top radius) and two transformation operators (translation and scale) that can be applied to any primitives.

The constructive parameters of the language include transformation parameters and primitive-dependent parameters (e.g., the top radius of a cylinder). It is worth mentioning that numerous CAD systems allow users to expose high-level variables and define constructive parameters as functions of these variables. However, in our approach, we do not assume the presence of such

variables in the input. Instead, we focus on learning high-level abstractions directly from the constructive parameters themselves, highlighting the effectiveness of our method in uncovering meaningful constraints.

7.3.2 Variation Prompts Generation

We leverage a pre-trained LLM (ChatGPT) to generate text prompts that describe variations of the given model by using the following formulaic query: "What are different types of [object]?". This generally outputs an itemized list of prompts which we can extract. A user can then select a subset of these prompts or add additional prompts that the user cares about.

7.3.3 Text-Conditioned Variation Generation

In this section, our goal is to generate variations of the initial CAD model to serve as examples to discover constraints and re-parameterize the CAD program in a semantically meaningful way. For each text prompt, we follow a general framework to generate 3D shapes from a pre-trained text-image model where it uses a differentiable pipeline to bridge a 3D representation to an image.

A fundamental challenge for our domain is that the space of possible variation of a CAD program is highly constrained compared to meshes or neural radiance fields (NeRFs). Therefore, to optimize a CAD program's parameters from text-driven image guidance, we constantly need to project the target variations back to the feasible space. We notice that losses used in prior work such as ClipMesh [KXBP22] and DreamFusion [PJBM22] create artifacts in our application, essentially disconnecting CAD primitives (see discussion in Section 7.4.5). We attribute these errors to the challenges of finding the proper projection back to the CAD domain. To overcome this challenge, we propose to use an image-space loss because the transformations between CAD parameters, meshes, and images are more straightforward, facilitating the projection process. We

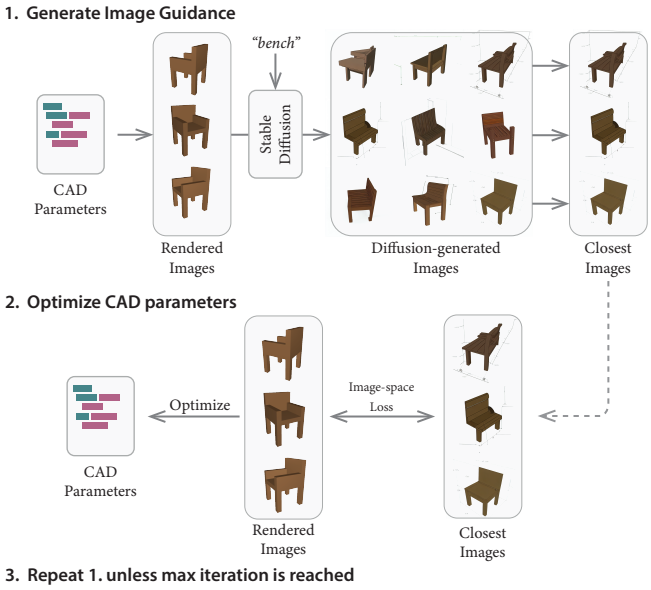


Figure 7.2: Text-conditioned design variation. We iteratively generates images using stable diffusion and refine our model to match the sampled image.

consistently observe that this approach significantly improves the overall results.

Our generation algorithm (Figure 7.2) iteratively generates images using stable diffusion [RBL+22] conditioned on the input prompt and the current rendering of the CAD model. After each sample, it projects back to the CAD parameters by fitting our input model to the sampled image using gradient descent on pixel loss from a differentiable renderer [LHK+20].

In generating the image guidance, we randomly sample 5 camera angles from which we render each image and run the image-to-image diffusion. Similarly to DreamFusion [PJBM22], we also append a viewpoint description to the prompt, e.g., "chair, (front|side|rear) view" as a proxy to condition camera angles.

In the gradient descent steps, we use an SGD optimizer with a learning rate of 0.05 for 30

iterations where we minimize the following loss:

$$\mathcal{L}(\mathbf{x}) := \sum_a \|\text{sharpen}(R_a(\mathbf{x})) - \text{sharpen}(I_a)\|^2 + \lambda\|\mathbf{x} - \mathbf{x}_0\|^2$$

where $\mathbf{x}$ is the CAD parameters, a is a camera angle, $R_a(\cdot)$ is the renderer, I_a is the image from the diffusion process, $\text{sharpen}(I) := I - 0.2 \cdot \text{blur}(I)$, and $\lambda = 0.001$ is the regularization term towards the original parameters $\mathbf{x}_0$. We repeat the process of diffusion and gradient descent 400 times.

A final fundamental challenge we encounter is that diffusion models can produce images with diverse styles and conflicting geometries across different views. This can lead to degenerate results, such as a table with thin metal legs disappearing because when the legs appear at different positions in the generated images, the optimizer does not know which legs to follow and so makes it invisible instead (also see Section 7.4.5). To overcome this, we employ a selection process where we run diffusion three times to generate a set of images and then choose the one that aligns best with the initial rendered image to help improve geometric consistency across views. This selection is performed at each step to provide effective image guidance. We have observed that this approach consistently produces favorable outcomes.

7.3.4 Constraint Discovery

Our generative model produces a collection of examples in the design space we wish to explore. We want to use these examples to provide structure and constraints on that design space to aid in exploration by discovering constraints on the CAD model parameters that are common to the discovered variations. While there are many existing methods for constraint discovery [FMd23, BCFO04, FB18], most of these methods require noise-free examples, which we do not have, except for on our input model. For this reason, we propose algorithms for selecting among possible constraints under the presence of noise.

Discovering Geometric-Semantic Constraints

Because we have only a single model with clean geometry, our initial input, we propose to discover common geometric relationships — coplanarity, coaxiality, keypoint coincidence, and dimensionaly equality — present within the initial model, represented as conjunctions of linear constraints. The subspace these induce is too specific to the input model, so we would like to find a subset of these constraints that is common (in approximation to handle noise) to all discovered variations.

When dealing with constraints, it is crucial to consider their potential interactions. Ideally, we would rank all possible combinations of constraints; however, this combinatorial problem is computationally intractable, so we propose a greedy strategy instead. Using a distortion metric (described below), we score every individual constraint and initialize our constraint set with the best one. We iteratively add to this set by scoring each of the remaining unused constraints unioned with the constraints already chosen. The result of this process is an ordered set of constraints from first to last picked. Plotting the distortion over number of constraints, we observe that there is usually a point where the distortion increases drastically, and we have observed that this jump in the graph typically correlates to visual artifacts in the shape. We use this to compute the cutoff of which constraints to impose. We can find the jump in the graph using a linear change point detection algorithm [KFE12] on the derivative of this curve (computed as a central difference).

To score a candidate constraint set, we would ideally measure the distance in pixel space to the diffusion images from the final stage of generation. This requires solving an optimization problem to find the minimizer of that distance over the CAD parameter space, which is too slow to be tractable with the greedy strategy described above. We also observe that pixel space differences can be unreliable for very small variations arising from small constraint sets because capturing them in an image is highly dependent on viewing angle. To overcome both of these issues, we

propose to instead use a volumetric score, intersection over union (IoU), when initially sorting constraints, and only use pixel space loss when computing the final cutoff. Because we do not have a way to differentiably compute shape booleans, and to gain computational efficiency by avoiding gradient descent entirely, we developed a linear cuboid approximation to IoU which, in conjunction with our linear constraint sets, allows us to minimize the IoU with a single linear least squares step (see Appendix E for details). In cases where the IoU simplification does not generalize (cones with variable half-angle) we fall back to image loss for the full pipeline and avoid the additional check for constraint combinations due to its computational complexity.

Discovering Discrete Variations

Not all examples of the same kind of object have the same parts; for example some cameras have flash bulbs while some do not, and chairs can be armless. We discover these discrete parameters by looking for primitives effectively missing from variations. We iteratively remove one primitive from each variation and compute the pixel loss described above. If this loss is below a threshold (10^{-4} in our experiments), we mark that primitive as optional for that variation. We then group parts that are always optional together across variations (e.g. chair legs are added or removed as a set) and include these as binary variables on top of our continuous reparameterization.

7.3.5 Re-Parameterization

Our generation and constraint discovery algorithms find a set of linear constraints which restrict the construction parameters to semantically appropriate values, as well as a set of semantically labeled variations that obey these constraints. The constraints give us a lower dimensional subspace that maintains object coherence. Using Gauss-Jordan elimination we can construct a basis for this subspace that retains parameter identity for free variables under the constraints.

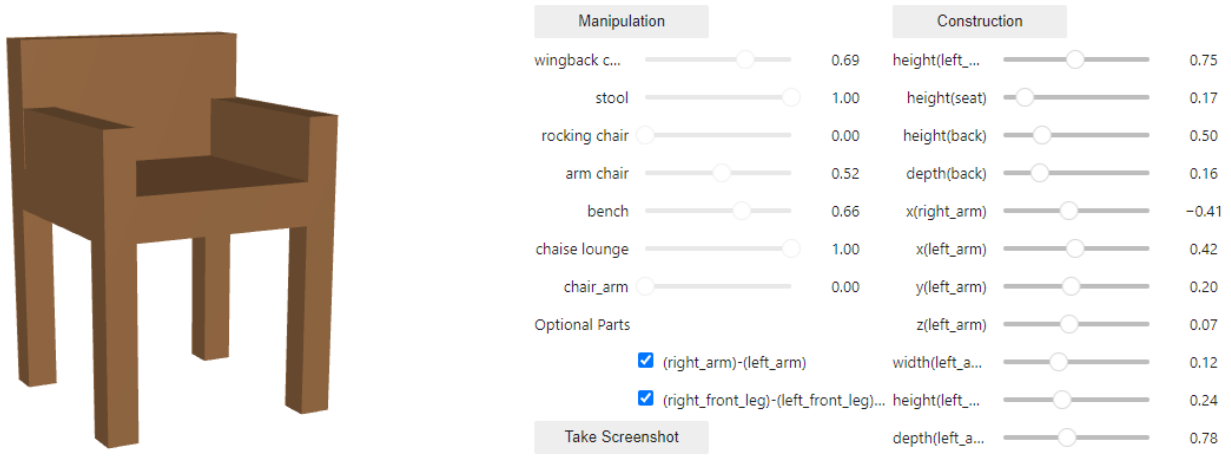


Figure 7.3: ReparamCAD's user interface. Users are presented with two complimentary views of the constrained reparameterization space. The sliders on the left interpolate between discovered variations as a weighted average, while those on the right control the free variables after reparameterization. Checkboxes allow toggling of discrete sets of removable parts.

We construct a second parameterization of this space centered at the initial model's parameters within the subspace that allows these shapes to be mixed and interpolated.

Our user interface (Figure 7.3) allows for exploration in both *constructive* parameters and *manipulation* parameters, but because we only found linear equality constraints, we do not have bounds on this space. We use examples as lower bounds on the feasible region by taking a normalized sum of manipulation parameters, restricting semantic variations to be interpolations between variations, and optionally restricting constructive parameter exploration to the extreme values of the discovered variations. In this bounded subspace, the user can freely and safely explore. Additionally, we add the discrete optional degrees of freedom found in constraint discovery.

7.4 Results

We evaluate ReparamCAD in two key aspects: its ability to generate compelling and valid design variations guided by text prompts, and its ability to infer semantic and geometric constraints that define the family of shapes both qualitatively and quantitatively.

We show the results of running our pipeline on five initial models: chair, table, car, camera, and bottle in Figure 7.7. For each model, we handpicked five prompts from a list generated by ChatGPT and manually added a "bench" prompt for the chair example. For the differentiable renderer, we use Nvdiffrast [LHK+20]. We use stable diffusion model v1.4 from [RBL+22], and we use the mesh boolean algorithm from [CPAL22] to compute the IoU. All experiments were conducted on a machine with a 40-core CPU and an NVIDIA A40 GPU. On average, generating a CAD variation takes 6.5 hours per prompt; and discovering constraints takes from 4 to 35 minutes per model depending on the complexity.

7.4.1 Generating CAD Variations

In Figure 7.5, we demonstrate our method's ability to generate variations of the input model that are coherent with the text prompt with varying geometry and complexity. For example, we can observe that the bench is wider and has no arms while the chaise lounge has arms and is bulkier, and that the SUV has a tall back section while the pickup truck has a low back section. The method is also able to discover part of the program that can be removed, for example, the arms of the chair for "stool" and the camera grip for "action camera" disappear by shrinking and blending into the rest of the shape.

The method's inclination of staying close to the input model and within the parameter space leads to some intriguing results. For example, the "rocking chair" prompt produces a model resembling a nursing glider, which belongs to the same category and can be represented using

the input program. Similarly, the generated "dining table" exhibits a surprising square shape and middle shelf. While these may not conform to the conventional idea of a dining table, such dinning tables exist in reality, making them more likely to be chosen due to their proximity to the input model (see Fig. 7.8 (d)).

Overall, our method performs well in maintaining the overall structure of the shapes while allowing unique changes that define each of the variations. We notice, however, that due to stochasticity of the diffusion process, these generated models contain imperfections on primitive alignment (see corners of the chairs or table where the legs do not perfectly line up with the seat). These imperfections must be captured and cleaned up by our symbolic approach.

7.4.2 Inferring Constraints and Reparameterization

Running our constraint discovery algorithm on the input models resulted in a dimensionality reduction from 19-48 parameters to 9-15, summarized in Table 7.1. We find semantically meaningful constraints; for example, car wheels are co-axial, chair arms stay at the sides of the seat, and table and chair legs are all the same height. Our algorithm also rejects overly specific constraints such as the chair's seat being square and the same thickness as the arms and legs, which would overly constrain the design space as seen in Figure 7.8 (c). The variety and quality of generated models in Figure 7.7 show that our constraints strike a balance between restriction and expression. They also have the effect of removing noise present in the generated variations, which propagates to the interpolated results as shown in Figure 7.8 (a). In addition, we also support extrapolation by direct control of free construction variables. As Figure 7.4 shows, extrapolation without constraints often produce incoherent geometry, but ReparamCAD's constraints can prevent this.

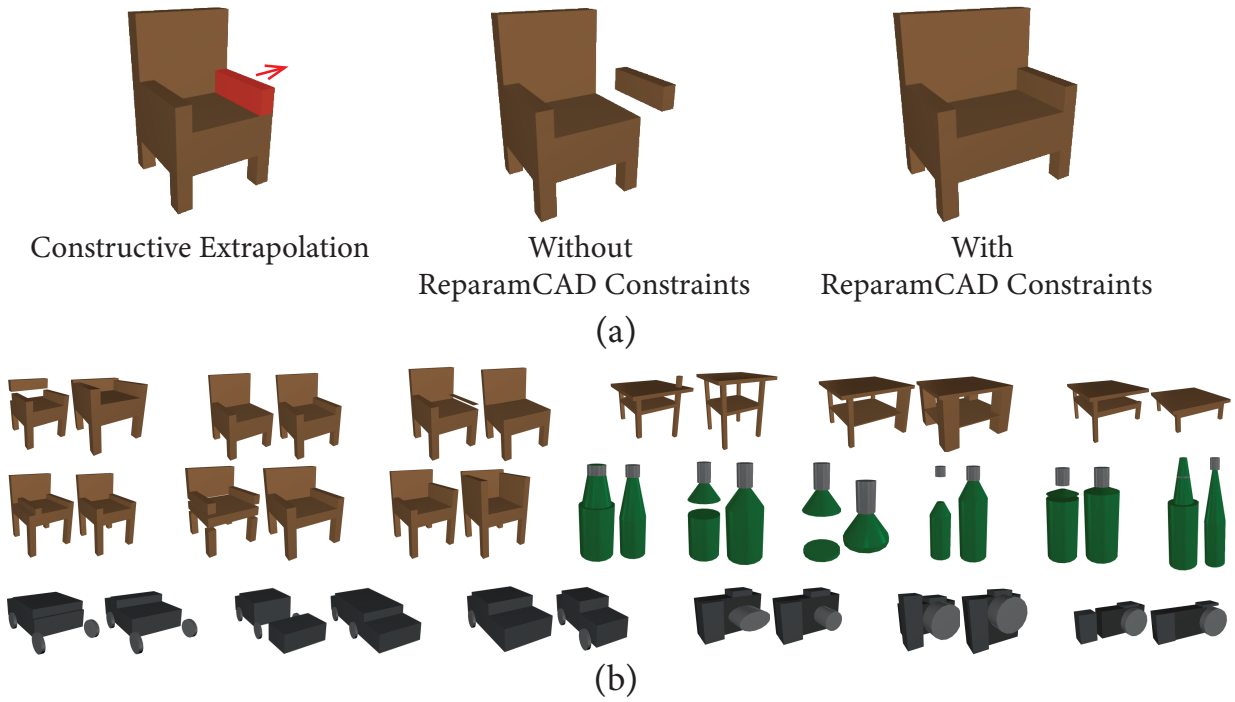


Figure 7.4: (a) Applying an extrapolative edit to a model can lead to broken geometry, but the constraints we discover prevent this and result in meaningful global edits from local changes. (b) A selection of random extrapolative edits without ReparamCAD (left) and with (right). ReparamCAD maintains geometric coherence.

7.4.3 Quantitative Evaluation and Robustness

We asked four CAD users with 2-14 years experience (mean 6.25) to select constraints for each base model from the set our system considers, given the variation names. We evaluated precision and recall of our system, computed by subspace inclusion because the constraints are non-orthogonal. Macro-averaged across users, models, repeated 4 times to test robustness, our method achieves 76.7% precision and 88% recall. To quantify robustness, we computed the standard deviation in average precision and recall between runs of our system for each model: 2.4 pp for precision and 6.8 pp for recall. Finally, to quantify the effect of regularization, we compared precision and recall across models and users for a run with and without regularization using a related samples t-test and found no significant difference for either ($p = .38, .29$).

7.4.4 Ablations of our Neurosymbolic Approach

We formalize the need for our neurosymbolic approach by comparing it to pure neural and symbolic techniques. A purely neural approach only considers the generated model without any constraints. As seen in Figure 7.8 (a), this results in many artifacts.

On the other hand, if we apply all the geometric-semantic constraints inferred from the original model without using the generated images as input, we would end up with a more constrained space, as shown in Figure 7.8 (c); most of the chairs are simply scaling variations that can appear overly boxy or skewed. We note that the variations we show in Figure 7.8 (c) are still leveraging the bounds that we have discovered with our technique. In practice just imposing constraint does not define a bounded space for direct manipulation, and infinitely large boxy results can be generated—indeed these methods are used in companion with variational techniques for interaction [GSMCO09a].

7.4.5 Ablations of our Text-Conditioned Variation Generation

Figure 7.8 (b) shows optimized CAD models using different loss functions: (1) CLIP similarity loss; (2) distillation scores, where the gradient is the difference between the added noise and predicted noise in latent space; (3) L2 difference in the image latent by the autoencoder; and (4) our image-space loss. For CLIP, we believe the unfavorable results are due to how the gradient is propagating back to the low-dimensionality of the CAD parameters. For stable diffusion, we observe that a full generation process is necessary to get an effective image guidance as opposed to using the difference between a single denoise step as seen with the distillation score. Using the full diffusion process, whether using the image similarity or its latent, better preserves the overall structure and connectivity and converges to a desirable shape variation. We decide to use the image difference to avoid having to backpropagate to the image encoder.

In Figure 7.6, we show the optimization process of methods using a different number of diffusion-generated images per camera pose. Since stable diffusion generates slight different images every time, thin parts like the legs of the table is more susceptible to disappearance because they are more likely to not overlap and end up confusing the differentiable renderer. We observe that using only the generated image that is the closest to the rendering reduces this effect and helps maintain thin structures.

7.5 Conclusion

ReparamCAD demonstrates two things. First, it suggests that letting AI re-write CAD programs to have new parameterizations is a way out of the paradox of parametric CAD. The second is that it forges a path for one of the dominant forms of generative AI, image diffusion models, to be interfaced with CAD models via a differentiable renderer. We used a simplified geometry

Table 7.1: Number of parameters, number of constraints in the base model, number of inferred constraints, number of reparameterized dimensions for each model.

Model	Parameters	Base Constraints	Common Constraints	Constraint Dimensionality
Bottle	19	20	8	9
Camera	24	21	9	15
Chair	48	172	37	11
Table	36	90	22	14
Car	42	94	26	13

representation, CSG over unions, in order to sidestep the technical problems of differentiable CAD rendering and explore potential applications directly, but one could imagine a future where similar techniques are used with more mainstream, B-Rep based CAD modeling, potentially building off the differentiable rasterization we developed for geometric self-supervision in B-Reps (see Appendix D for an example of gradient-based optimization of B-Reps.) ReparamCAD provides some automation over the ideation stage of CAD design, as well as implementation as long as the model topology does not need to change. It still requires a designer to manually create an input model, however, and is limited to variations that would be possible, if difficult, in the initial parameterization. The next chapter will take a complementary approach to the early stages of the design loop; generating CAD models de novo from textual prompts using large language models.

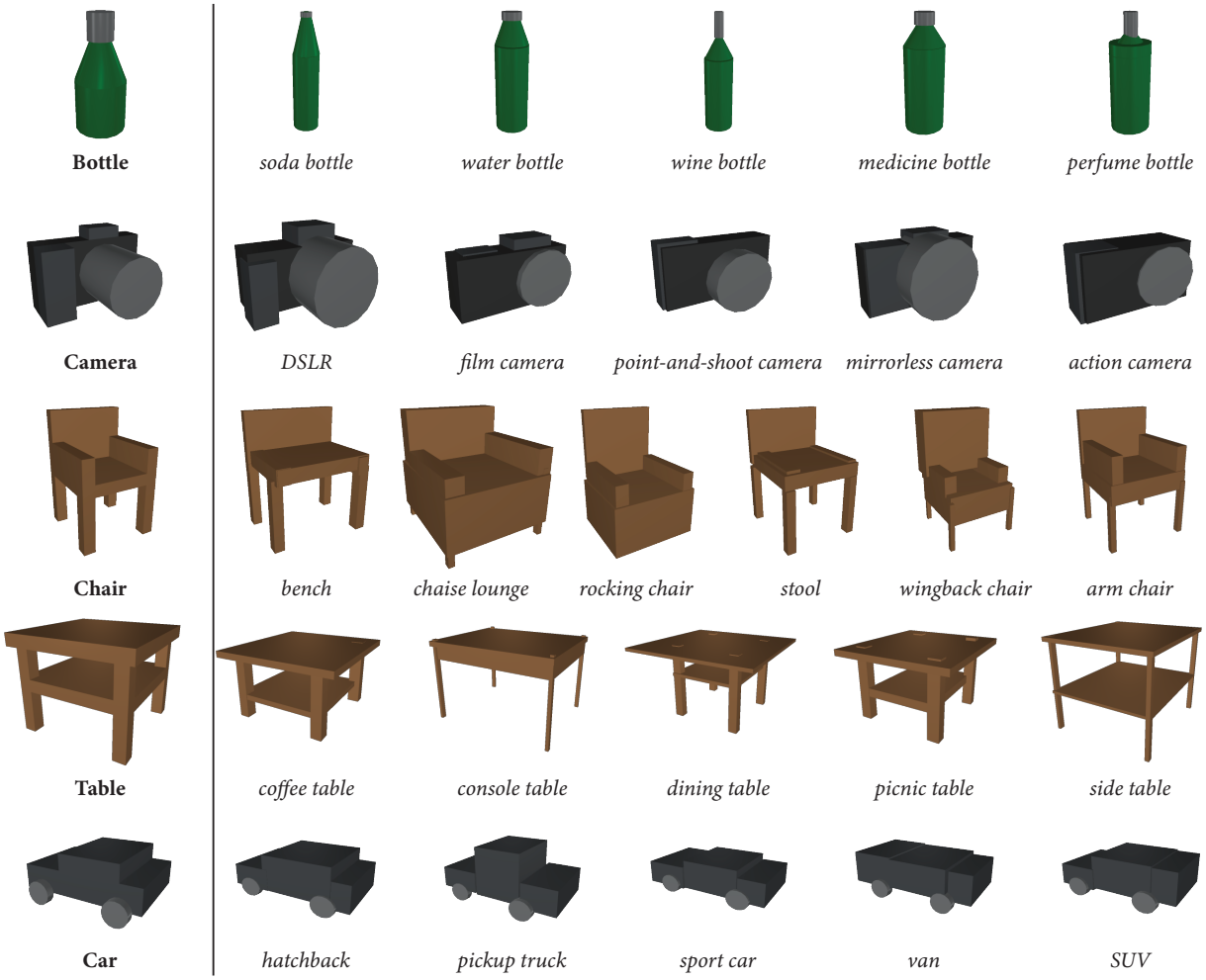


Figure 7.5: A gallery of text-condition CAD parameter optimization of different input models (first column) towards varying prompts.

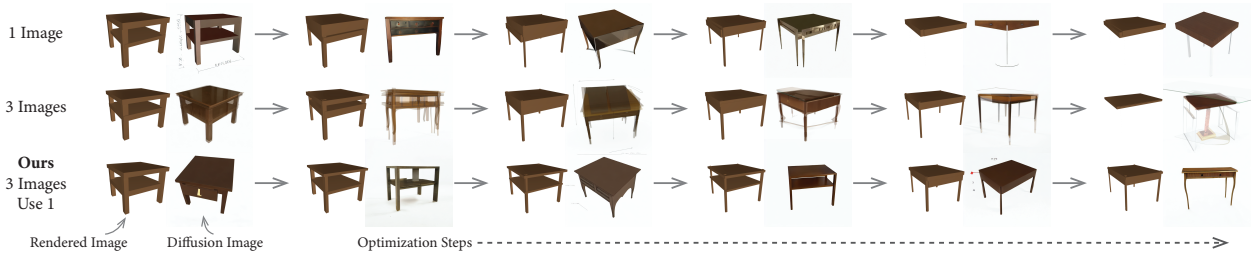


Figure 7.6: Rendered and diffusion-generated images at different iterations during optimization using (top) one diffusion-generated image per camera pose; (middle) three images; and (bottom) three images and select one image that is closest to the rendering. Note that the camera angles for the diffusion-generated images are random.

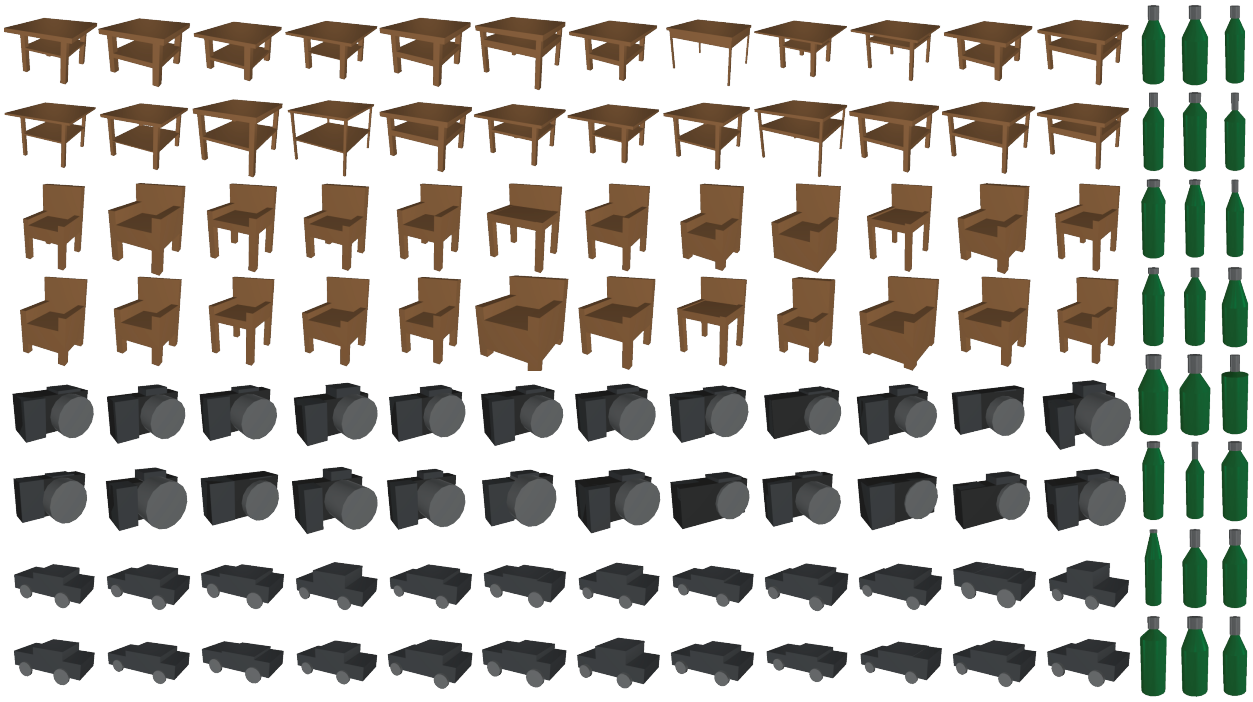
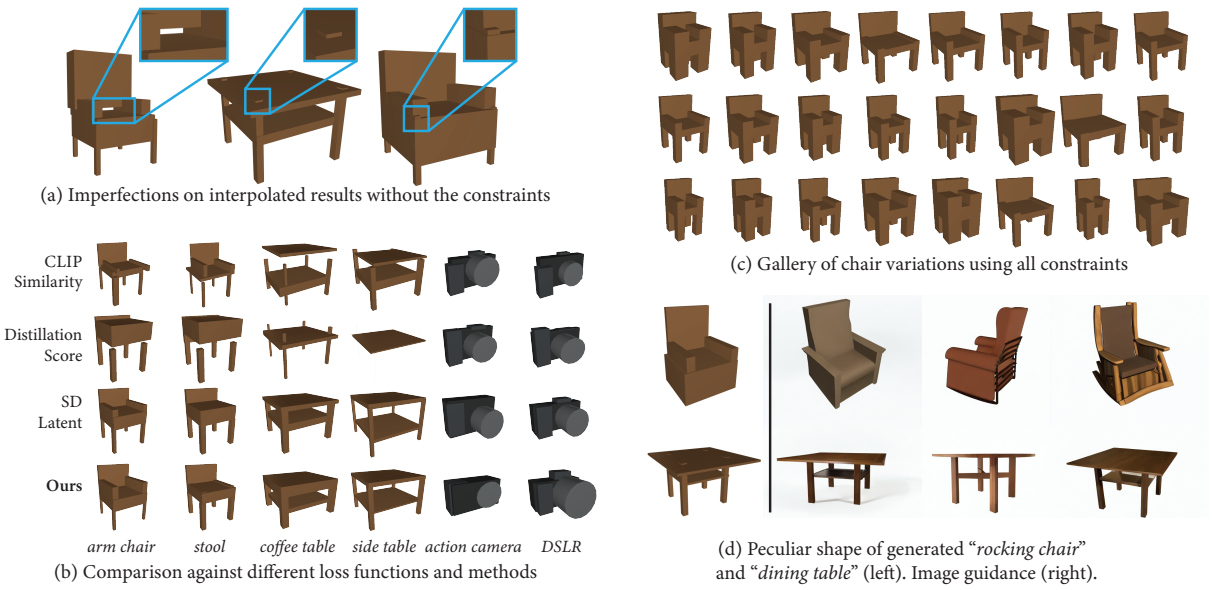


Figure 7.7: A gallery of design variations using the constrained inferred from our pipeline.



(a) Imperfections on interpolated results without the constraints

(b) Comparison against different loss functions and methods

(c) Gallery of chair variations using all constraints

(d) Peculiar shape of generated "*rocking chair*" and "*dining table*" (left). Image guidance (right).

Figure 7.8: (a) Imperfections on interpolated results without the constraints. (b) Generation results using different loss functions: CLIP similarity, distillation score, L2 difference of the image latent of the rendered and diffusion-generated images, and L2 difference in the images (c) A gallery of chairs when all constraints of the base model are imposed. (d) Peculiar shape of generated "rocking chair" and "dining table" (left) and the image guidance (right).

Chapter 8

AI Design Language

8.1 Introduction

The previous chapter explored how foundational text-image models can be applied to improve the ideation stage of design, but it began with an assumption that a CAD model had already been built. This chapter looks at how we can kick-start the CAD modeling phase of design by enlisting a large language model (LLM) as a CAD co-pilot. Coding assistance has been one of the hallmark successes of LLMs in the recent AI boom, so applying this technology to CAD programs seems like a natural fit. However, as I explained in Section 2.1, traditional CAD languages are a representation designed around GUI-based editing in the context of continuous visual geometric feedback. This raises a crucial question: How can we reimagine the traditional CAD DSL principles, which have been designed for a constant visual feedback loop, to craft innovative languages for design in an age where code is generated with support from AIs?

In this work, we address this question and propose a new DSL for CAD modeling with LLMs, which we call AIDL: AI Design Language. Through experiments with different existing models and prior work that analyzes their observed behavior, we identify four key ***design goals*** for our

DSL. Namely, we propose a *solver-aided approach* that enables LLMs to concentrate on high-level reasoning that they excel at while offloading finer computational tasks that demand precision to external solvers. For CAD, this means that the DSL should enable implicitly referencing previously constructed geometry (***dependencies***) and specifying relationships between parts that can then be solved by the solver (***constraints***). Further, we aim to create *semantically meaningful abstractions* that leverage the LLM's proficiency in understanding and manipulating natural language (***semantics***). Finally, we advocate for a *hierarchical design approach*, which allows for encapsulating reasoning within different model parts (***hierarchy***).

Our analysis of existing CAD DSLs reveals that none achieve all four design goals, and supporting all goals simultaneously presents challenges due to conflicting requirements. For example, the ability to unambiguously reference all intermediate parts of the geometry (***dependencies***) is a known challenge in CAD. While recent work proposes a language that supports unambiguous referencing, it requires semantic complexity (***semantics***). Additionally, while constraints are widely used in specific aspects of CAD design, such as assembly modeling (***constraints***), supporting them in a complex model with hierarchically defined constraints (***hierarchy***) is computationally challenging. Our key insight is that we can address these challenges by both limiting and expanding different language constructs from prior CAD DSLs. While we limit the use of references to *constructed* geometry, without loosing geometric expressiveness, we expand the use of constraints to hierarchical groups of geometry, so called *structures*. We support these novel language constructs with a back-end recursive solver. We evaluate AIDL as a DSL for AI assisted modeling, showing that a commercial LLM (GPT 4o) can generate AIDL programs that are easily human editable and provide near-complete initial models from text-prompt input.

8.2 Related Work

8.2.1 CAD Generation

There have been several attempts in recent years to generate CAD models. Some attempt to directly generate symbolic geometry[WJL+21, GLP+22b, JLD+23, NGEB20, XLJ+24], while others use procedural representations [WXZ21a, ERSLT18, GBL+21b, RZC+22, LGZY23, XWL+22, LWJ+22b, PBG+21a, SZRA22, WPL+21]. Some procedural approaches employ symbolic techniques either alone or in concert with learned heuristics [NWP+18, XPC+21b, DIP+18b]. A fundamental challenge with these tools is the ability to control the generation. While many methods can be conditioned on an input allowing for reverse engineering applications [LWJ+22b, GLP+22b], the few methods that directly focus on generation give limited control over their output [JLD+23, WXZ21b, XLJ+24, SZRA22]. The highest degree of control is afforded by those that take sketches as input, such as Free2CAD [LPBM22] but these are effectively reverse engineering an existing geometric design rather than enabling high level guidance. The goal of AIDL is to enable control without direct geometric supervision, which is why we propose to use an LLM to create CAD models through code, and therefore focus on DSL design rather than the design or training of a generative model. Importantly all this prior work uses CAD DSLs that have limitations when it comes to LLM needs as will be discussed and surveyed in details in Section 8.3.

8.2.2 Symbolic and Neurosymbolic CAD

While many generative and reconstructive models treat the procedural form of CAD models as linear output and rely on large-data statistics to take care of syntactic and semantic concerns, some works employ symbolic techniques directly, either alone or in concert with learned heuristics. ReIncarnate [NWP+18], Zone Graphs [XPC+21b], and InverseCSG [DIP+18b] treat reconstruction

as a synthesis problem given inferred initial geometry, but use different synthesis techniques; oracle guided heuristics, neurally guided search, and constraint guided synthesis. Program synthesis can also be used to improve the robustness of existing CAD programs. [MZ21] synthesizes symbolic constraints on input parameters that exclude configurations which break the model. Some of these constraints are discoverable through static program analysis, but their work highlights the necessity of access to a CAD kernel to understand the semantics of a CAD program, necessitating *dynamic* analysis as well. [MPZ20] improves robustness by replacing specific references with programs synthesized with a small program size heuristic, using the hypothesis that smaller programs will generalize better, thus capturing user intent. With AIDL, we want to incorporate semantic understanding beyond that of existing CAD programs, and therefore design our system around *general purpose* language models rather than CAD specific models.

Code Generation with LLMs Code generation has been one of the headline applications of the recent LLM explosion. There are several coding-specific LLMs that have been specifically trained or fine-tuned on code repositories and coding specific tasks [LAZ+23, LLA+24, GDC+23]. Generating solutions to complex prompts is difficult. Some works explore prompt engineering and multiagent strategies for pre-planning or coordinating a divide-and-conquer strategy for complexity [DJJL23, BSK+23, SDS+23], while more recently models have been fine-tuned on synthetic, complex examples generated by LLMs [XSZ+23, LXZ+23]. LLMs can perform worse generating code in DSL than general purpose languages since they will have fewer or no examples of these in their training data. For context-free languages, grammar prompting [WWW+23] can constrain output to valid DSL expressions. This approach is not feasible for more complex languages like AIDL, however there are prompting strategies that can increase the likelihood of valid program structures [JNS23]. Another recent work proposes an LLM specific general purpose language, BOSQUE [Mar23], that shares a similar the philosophy with AIDL of designing the

language around an LLM's strengths and weaknesses. In particular, BOSQUE's embrace of pre- and post-conditions mirrors AIDL's use of constraints and strong validation, but does not go so far as to employ a solver to enforce constraints.

8.3 Domain Analysis and Design Goals

We ground the design of our DSL on a survey of existing CAD languages and an analysis of LLMs.

8.3.1 CAD DSLs

We group CAD DSLs into three broad categories:

Constructive Solid Geometry (CSG) In CSG, users can specify 2D and 3D parametric primitives, such as rectangles or spheres, directly in global coordinates. Using boolean operations, such as union or intersection, users then combine these primitives in a hierarchical tree structure to achieve complex designs. While some CSG languages, such as OpenSCAD, allow the use of variables or expressions for primitive parameters, they do not support specifying relationships or dependencies between different parts of the geometry. This absence of dependencies simplifies the abstraction, making CSG widely used in inverse design and reconstruction tasks [DIP+18b, NWA+20, YCL+22b, MB21]. However, this limitation also makes modeling more challenging, which is why CSG is not commonly used in most commercial CAD tools.

Query-based CAD Most commercial CAD tools use query-based languages, such as FeatureScript [Ons24], which employ a sequence of operators to create and modify models (e.g., extrude, fillet, chamfer). These operators reference intermediate geometry—e.g., a chamfer operator takes a reference to an edge. This referencing creates implicit dependencies, simplifying modeling and

enabling easy editing as operations propagate when intermediate geometry is updated. However, a challenge arises when edits lead to topological changes, making reference resolution ambiguous. For example, if an edge gets split or disappears, where should the chamfer be applied? To address this, these languages do not reference geometry explicitly. Instead, geometric references are specified *implicitly* via a language construct called *queries*. These queries are resolved during runtime by a solver [Cad24, Ons24], which typically uses heuristics to resolve ambiguities. This makes automating design challenging, and generative tools that use CAD operators restrict themselves to sequences where references are not needed, such as sketch and extrude [WXZ21b, WPL+21, LWJ+22b]. While recent work allows for the unambiguous direct specification of references [CBS23], mastering this language is complex and demands significant expertise.

Constraint-based CAD As the name implies, constraint-based CAD DSLs natively enable users to create geometric constraints between geometric primitives. This frees designers from specifying parameters consistently, allowing for free-form design while ensuring that relationships between parts are preserved. This approach is used in content creation languages like Shape-Assembly [JBX+20b], GeoCode [PLH+22], and SketchGen [PBG+21b]. In typical commercial CAD tools, constraint-based abstractions are used in sketches—2D drawings that get extruded to form 3D geometry—and during assembly modeling, but not during solid modeling which uses queries. These languages do not provide operations to modify primitives or to create intermediate geometry, and therefore they reference geometry directly. Designs specified in these languages are non-hierarchical, all constraints are being solved simultaneously.

8.3.2 LLM Analysis and Design Goals

We review the strengths and weaknesses of LLMs and formulate design goals that our DSL should support.

structure	=	⟨ *frame*, *sketch*, [*ref<structure>*], [*constraint*] ⟩
frame	=	⟨ *type* ∈ { Assembly, Solid, Hole, Drawing }, *orientation* ∈ { Top, Front, Side }, … ⟩
sketch	=	⟨ [*ref<geometry>*], [*ref<parameter>*] ⟩
parameter	=	⟨ val ∈ $\mathbb{R}$, mutable ∈ $\mathbb{B}$ ⟩
ref<τ>	=	⟨ name ∈ String, ptr ∈ τ ⟩
geometry	=	Point | Line | Arc | Circle | ⟨ [*ref<geometry>*], [*ref<parameter>*] ⟩
primitives	::=	make_point | make_line | make_arc | make_circle | make_rectangle | …
constraint	::=	*logical_expr* | *structural_constraint (ref<τ>, ref<τ>)*
	|	*unary_geometric_constraint (ref<τ>)* | *binary_geometric_constraint (ref<τ>, ref<τ>)*
structural_constraint	::=	above | center_inside | left_of | taller | ...
unary_geometric_constraint	::=	horizontal | diameter | fixed | ...
binary_geometric_constraint	::=	coincident | tangent | equal | symmetric |...
logical_expr	::=	*arith_expr* = *arith_expr* | *arith_expr* ≤ *arith_expr* | *arith_expr* ≥ *arith_expr*
	|	*logical_expr* ∧ *logical_expr*
arith_expr	::=	c ∈ $\mathbb{R}$ | *parameter* | *u_op arith_expr* | *arith_expr b_op arith_expr*
u_op	::=	− | sin | cos | arcsin | arccos | sqrt | abs | norm | square
b_op	::=	− | + | × | ÷ | min | max

Figure 8.1: Language types and operations. τ represents the union type (structure|parameter|geometry). $[\theta]$ is the notation used to represent an array or list of θ.

Direct vs. indirect computation In their seminal paper, [BCE+23a] observe that GPT-4 is unable to solve mathematical equations directly, but that it can correctly make use of external math libraries. Similarly, [MFW+23] show that with a simple CSG DSL, geometric primitives often intersect each other and parts can be disconnected. Using a query-based CAD DSL featuring sketch primitives, GPT-4 more successfully constructs connected objects. These results suggest that LLMs perform better in tandem with external tools, like a math library or a constraint solver. For CAD, we want to give the LLM the means to express design intent not through direct computation, but by specifying constraints.

In modern CAD tools, users can specify geometric relationships with two constructs: implicit dependencies through references and explicit constraints, each with its trade-offs. Dependencies on geometric entities create implicit constraints that are easy to reason about locally. However, the effect of a long chain of references is challenging to interpret, a problem that arises for example if the desired constraint is non-local [MFW+23]. In GUI CAD tools, users do not have to interpret long chains of references, as the current state of the CAD model is rendered interactively. On the other hand, explicit constraints, such as in CAD sketches or assemblies, allow users to

express design intent without indirection. While this is easier to reason about even for non-local constraints, these constraint systems are often under-or over-constrained, and finding just the right amount of constraints is challenging.

To achieve the best of both worlds, we aim to support both **implicit constraints through geometric dependencies (*dependencies*)** and **specification of geometric relationships via constraints (*constraints*)**.

Named variables and semantic cues LLMs are designed to manipulate words, i.e., terms with semantic meaning. In their experiments, [MFW+23] reparametrize CSG programs with and without informing the LLM about the modeled object. Their results suggest that better reparametrizations are obtained by providing additional semantic knowledge. In computational design DSLs, users can employ semantic variables or named constraints, but they are not required to.

Intuitively named operators (*semantics*) Our CAD DSL should use natural language terms for design operations, references, and constraints. Our language should also expose geometric entities easily, without much semantic indirection.

Design complexity and modularity [BCE+23a] investigate the programming capability of GPT-4 and note that it can produce "syntactically invalid or semantically incorrect code, especially for longer or more complex programs". [MFW+23] experiment with complex designs, and note that they can miss components and that components can be wrongly placed. One of the proposed strategies for dealing with increased complexity is to introduce "subassemblies". However, a complex design can still be challenging to generate if the entire task is tackled all at once. In both programming [BCE+23b], and in design [MFW+23], up-front planning and iteration lead to better results.

Language	***dependencies***	***constraints***	***semantics***	***hierarchy***
CSG			✓	✓
Constraint-based		✓	✓	
Query-based	✓			
AIDL (Ours)	✓	✓	✓	✓

Table 8.1: We review how well the three major CAD DSL groups align with our design goals. We can see that neither of the already existing paradigms complies with all of the desiderata.

Hierarchical design and planning (*hierarchy*) In our CAD DSL, hierarchical design should be a first class construct. Our hierarchy construct should support planning and iteration in code generation.

8.3.3 Summary

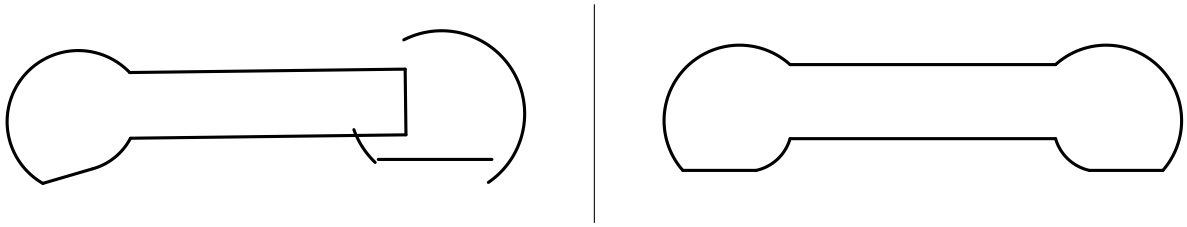
Here, we summarize which design goals are supported by different CAD DSLs, see Table 8.1.

CSG DSLs employ boolean operations which are intuitively named and LLMs can readily manipulate them (***semantics***). Designs in these languages result in a tree structure which allows for a hierarchical approach. Furthermore, a node in this tree can later on be edited or replaced by another CSG design, enabling partial evaluation (***hierarchy***). However, CSG DSLs do not enable users to specify constraints, neither implicitly via references (***dependencies***) nor explicitly via constraints (***constraints***).

Query-based DSLs allow implicit constraints through dependencies (***dependencies***), but do not support explicit constraints (***constraints***). Queries can also lead to quickly increasing semantic complexity (***semantics***). Lastly, queries create a chain of dependencies between distant parts of the program, making this category less modular than CSG DSLs (***hierarchy***).

Constraint-based CAD DSLs feature constraints (***constraints***) between geometric entities, which are often intuitively named—e.g., “coincident” or “symmetric” (***semantics***). While they use

references to geometry to define constraints, they do not create dependencies (***dependencies***). They further lack hierarchy, since solving constraints is performed in a global pass over a flat design. The design can change significantly after a local edit, making it unsuitable for partial evaluation (***hierarchy***).



```
# 1) Create structure and geometric primitives
handset = Solid()
handset.base = Rectangle( Point(0, 0), 5, 1)
handset.left_round = Arc(Point(-3, 0), handset.base.top_left, Point(-4, -1.5))
handset.left_line = Line(handset.left_round.end, Point(-3, -1.5))
handset.right_round = Arc(Point(3, 0), Point(4, -1.5), Point(2.5, 1))
handset.right_line = Line(Point(2, -1), Point(4, -1))
... # more primitives
# 2) Add constraints
handset.AddConstraint(Coincident(handset.right_round.end, handset.base.
    ↪ top_right))
handset.AddConstraint(Horizontal(handset.left_line))
handset.AddConstraint(Equation(handset.left_line.length == handset.right_line.
    ↪ length))
handset.AddConstraint(HorizontallySymmetric(handset.left_round.center, handset
    ↪ .right_round.center))
handset.AddConstraint(Equal(handset.left_fillet, handset.right_fillet))
handset.AddConstraint(Diameter(handset.left_fillet, 1.5))
... # more constraints
# 3) Solve system
solved = handset.Solve()
```

Figure 8.2: Design of a phone handset in AIDL. (Top left) Design before constraints applied. (Top right) Design after constraints applied. (Bottom) AIDL code for handset design.

8.4 AIDL - A Language for AI Design

In this section, we present AIDL, a CAD DSL for LLM-generated designs.

8.4.1 Key design decisions

The goal of AIDL is to fulfill the four design goals from Sec. 8.3. Combining all of these goals in a single CAD DSL requires addressing two key challenges.

The first is creating dependencies on intermediate geometry (***dependencies***) without increasing the semantic complexity of operators (***semantics***). As explained in Sec. 8.3, DSLs that reference intermediate geometry require queries, which in turn increase semantic complexity.

Our solution arises from the observation that all geometric primitives in CAD are created either through constructive operations that instantiate primitives or through boolean operations (e.g., when two edges intersect, a new vertex is generated). While this is evident for CSG DSLs we note that query-based CAD DSLs are not more expressive than CSG DSLs since all CAD operators (e.g. chamfering) can be expressed as a combination of a constructive and a boolean operation [CBS23]. Reference challenges emerge from boolean operations, as changes in parameters can lead to a varying number of generated primitives.

While we still want the geometric expressiveness enabled by boolean operations, we want to reference geometry without queries. To overcome this problem, we decide to restrict our DSL to only use references for geometry created before boolean operations. In our DSL, boolean operations are applied to *structures*, which is an intermediate type to create tree-structured hierarchies, see Fig. 8.1. The result of these booleans cannot be referenced, just as with CSG DSLs, however, we can reference *constructed* geometry and structures themselves. While this is a language limitation, it does not impact geometric expressiveness since hierarchical booleans are still allowed. Further, our analysis, see Sec. 8.3.2, shows that LLMs are struggling to reason about queries with long

chains, motivating our choice to disable them by design.

Second, using constraints (***constraints***) to specify the relationship between elements within hierarchical designs (***hierarchy***) is computationally challenging. Hierarchical designs encourage growing complexity and an increasing number of constraints, driving down solver performance. Query-based languages deal with this complexity by solving constraints in intermediate, *flat* designs, e.g. constraints between sketch elements in a CAD sketch are first solved before the user can extrude the sketch. Solving constraints from all CAD operations simultaneously is computationally too expensive for these systems. To tackle this challenge, we introduce (1) two types of constraints, one between geometry and one between *structures*, and (2) a custom recursive solver to hierarchically solve constraints in a design. This strategy allows us to explicitly define the hierarchy of constraints and to practically solve it, without providing intermediate feedback to the LLM.

8.4.2 AIDL by example

Next, we showcase AIDL by example and show how the different language constructs fulfill our design goals. First, we will illustrate the basic constructs of AIDL with the phone handset example in Fig. 8.2. An AIDL program starts by defining the high-level logic of a design. These high-level building blocks are called structures, and they are of different types, such as `Solid` and `Hole`, and they can be empty, a list of primitives, a list of substructures or any combination of these, see Fig. 8.1.

In the handset example, we first define an empty structure, L.2, which we populate with primitives, such as rectangles, lines and arcs, L.3-L.8. Next, we add unary and binary geometric constraints, e.g. Horizontal and Coincident, between these primitives, L.10-L.16. Finally, we solve the constraint system to optimize for the final parameters of each geometric primitive, L.18.

```
# 1) Create hierarchy
phone = Solid()
phone.handset = Solid()
phone.base = Solid()
base_solid = Solid()
base_hole = Hole()
phone.dial = Solid()
# 2) Specify geometry of structures
phone.handset = ... # program from Fig. 8.2
base_solid = ... #
base_hole = ... #
phone.dial = ... #
# 3) Compositional constraints between structures
phone.AddConstraint(HorizentallyAligned(
    phone.base, phone.dial))
phone.AddConstraint(Above(phone.handset, phone.base, 3*mm))
# more constraints ...
# 4) Boolean composition
phone.base.base_solid = base_solid
phone.base.base_hole = base_hole
# 5) Solve system
solved = phone.Solve()
```

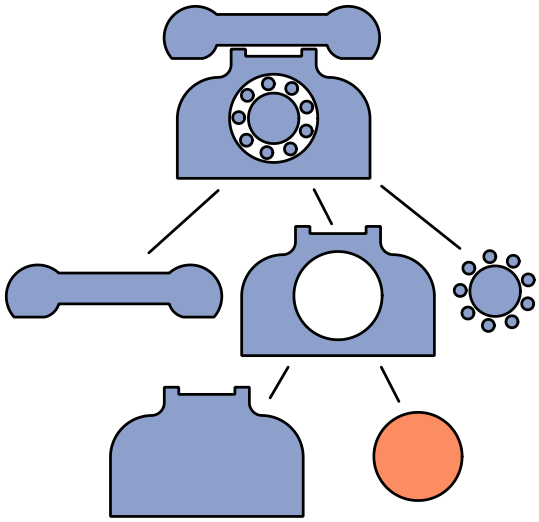


Figure 8.3: Design of a phone assembly. (Left) AIDL code for hierarchical design. (Right) Hierarchical tree structure of design. Solids are marked in **Purple** and the Hole is marked in **Orange**.

References In AIDL, references are pointers to geometry, parameters or structures. They have various usages.

First, instead of specifying coordinates directly such as in L.3, we can use references to reuse already defined geometry. For example, in L.4, we define an Arc, which in the AIDL API is defined via `Arc(center, start, end)`. The `left_round` arc starts at the upper left corner of the `base` rectangle via the reference `handset.base.top_left`. This strategy lowers the risk of erroneously recomputing coordinates of the upper left point. Second, this reference ensures that `base` and

`left_round` stay attached during the constraint solving process. Indeed, by sharing a common point, we *implicitly* define a coincidence constraint between them.

Geometric primitives can also be referenced within constraint calls. In L.10, we *explicitly* define a coincidence constraint between the upper right corner of `base` and the end point of the arc `right_round`. The arc `right_round` has been defined with explicit coordinates in L.6, which, without further constraints, is not necessarily connected to the rest of the shape, see Fig. 8.2 (top left).

Lastly, as can be seen in Fig. 8.1, references can also point to parameters of geometric primitives. This allows for more control and more expressiveness when defining geometry and constraints. Consider L.12, where we used equation constraints to express a symmetric design intent on the two lines `left_line` and `right_line`. L.12 declares that both lines should have the same `length`, which is a parameter of the `Line` primitive. Parameters are referenceable on the same level as geometry and structures, making them first-class constructs in our language.

Constraints Constraints express design intent, i.e., the way that geometry should behave under change. As we have already seen, in AIDL, constraints can be implied by sharing a reference, see L.4, or by explicitly adding them to the design via `AddConstraint` calls. Constraint operations have a certain constraint type and they take as input references. Depending on the constraint type, either equality or inequality constraints will be enforced on the geometric parameters specified by the input references. For example, in L.14, the `Equal` constraint type enforces the `diameter` of the two arcs `left_fillet` and `right_fillet` to be the same.

Using references and constraints, we can explicitly state the design intent, which will be realized by an external solver, L.18, (***dependencies***), (***constraints***).

Synonymous operators References and constraints in a DSL are useful if they are easy to use. For human users, learning a new DSL can be challenging if its API is long and redundant. Concise APIs are usually preferred. However, designing a DSL for LLMs introduces a different criterion, which is that the LLM might write a function call which is not part of the API, but which is semantically equivalent. For example, consider the two constraint calls: (1) `AddConstraint(Perpendicular( line_1, line_2))` and (2) `AddConstraint(Orthogonal( line_1, line_2))`.

Intuitively, both `Perpendicular` and `Orthogonal` should enforce the same angle between the two lines, i.e., they are synonyms. However, to reduce redundancy, most APIs will choose only one of them. In AIDL, we expose both constraint types, to account for syntactical weaknesses of LLMs and to take advantage of their semantic versatility (***semantics***).

More generally, we opt for a robust API vocabulary, allowing for different ways of constructing primitives, e.g. `Triangle(center, base, height)` vs. `Triangle(pt_a, pt_b, pt_c)` and synonymous combinations of compositional constraints, e.g. `Underneath` is equivalent to `Below` plus `HorizontallyInside`.

Note that even though we have synonymous references in AIDL, they are all being compiled to unique identifiers. During the interpretation of the program, we include only referenced entities in the model.

Hierarchical designs Next, we illustrate the use of hierarchical designs with a complete phone design, see Fig. 8.3.

Hierarchical designs start by specifying the high-level structure. The phone is an assembly made out of three different structures, of which the `base` is the result of two additional structures, L.2-L.7. Next, we specify the geometry and the internal constraints for each structure, L.9-L.12. An example for this is the phone handset from Fig. 8.2. Then, we specify constraints between

structures, L.14-L.17. For example, the constraint operation `CenterInside( phone.base, phone.dial)` places the dial in the middle of the base. Constraints between structures act on the respective bounding boxes, for more details, see Sec.8.4.3. Then, we specify boolean operations between structures. Here, the phone base is a result of a solid shape from which we cut out a hole to place the dial, L.19-L.20. Finally, we solve the entire, hierarchical design in L.21.

The structure of this example program closely matches our code generation strategy described in Sec. 8.5. Here, we follow with a few remarks about the language constructs which enable this strategy.

In our top-down approach, we encourage planning ahead by first building up a tree structure, see Fig. 8.3 (right) with empty nodes. In AIDL, at any point in time we have a valid design since structures can be empty, see Fig.8.1. In other words, we allow for partial evaluation, which we realize by instantiating empty structures with virtual bounding boxes.

In AIDL, the operation of a boolean operation cannot be referenced, since the parameter-dependent topological outcome requires queries, see Sec. 8.3. To implement this, boolean operations are implied by using different structure types. In L.5-L.6, we define two structures as `Solid` and as `Hole` and in L.19-L.20, we assign them to the same parent, which realizes the desired subtraction operation seen in Fig. 8.3 (right).

8.4.3 Compilation

After a model is created in the AIDL DSL, its validity is checked and it is compiled into an AIDL model, which is a tree structure in which every node is also a valid AIDL model. The AIDL model defines a system of geometry, parameters, and constraints in a hierarchy. The constraint system is then solved recursively bottom-up using an iterated Newton's method solver to find parameter values that satisfy all constraints while minimizing the size of constraint subproblems. Finally, the

geometry is aggregated up the tree from leaves-to-root, combined via boolean operations to form the final geometry.

Validation and Compilation To compile an AIDL model, we flatten Geometry nodes such that just primitive geometry (Points, Lines, Circles, and Arcs), Parameters, and Constraints remain, and are attached directly to a Structure. We ensure every subtree can be solved independently by validating that Geometry only references other Geometry on the same Structure, and Constraints only reference Geometry and Parameters within their own subtree.

Many numeric expressions in AIDL can only be defined once the model structure is finalized; these are called *deferred* expressions, and exist as Expression-valued functions of the final model hierarchy that get evaluated at the end of compilation. Two kinds of expressions get deferred in AIDL; ambiguous geometric constraints, and bounding boxes. Some geometric constraints, like fixed "Angle" have multiple common interpretations (clockwise or counter-clockwise); AIDL resolves this to whichever is closest to being satisfied by the initialization values. Bounding boxes are deferred because both local geometry and structural hierarchy are not fixed until compilation time. To support validating deferred constraints involving yet empty Structures, we create virtual bounding boxes.

Constraint Solving Constraints in AIDL are solved using an iterated Newton's method solver. AIDL expressions support "min" and "max", which are used to express bounding boxes. The piece-wise nature of these functions can hinder the convergence of Newton's method. To promote convergence, we remove these discontinuities by re-writing expressions involving "min" and "max" on the active branch, using the model parameter initialization before the Newton solve. We then check if the original constraint equations are satisfied, and iterate this process until convergence or failure.

In addition to "min" and "max," AIDL models can also contain inequality constraints. These occur when bounding boxes are constructed for empty structures to ensure that the boxes do not invert, e.g. $height \geq 0$. To support these constraints, we borrow the idea of slack variables from linear programming, and rewrite constraints of the form $A \leq b$ as $A - b + s = 0 \wedge s - |s| = 0$. The second equality is equivalent to $s \geq 0$, ensuring that the slack variable s is positive. Using inequality constraints in CAD is usually unhelpful because they negate the accuracy benefits of CAD models, and tend to lead to very unpredictable solutions. For this reason, we do not expose the inequality capabilities of AIDL to the programmer. In our system, inequality constraints only appear at intermediate stages of programming when they are used to *validate* the feasibility of a constraint system before concrete geometry is specified.

Model Solving Because AIDL can express compositional constraints between structures, the constraint system can involve the entire tree and grow in complexity, making it difficult to solve and its solution unintuitive. By structuring AIDL models into trees of small, independent constraint problems, which is being reflected by our validation criteria, we ameliorate both issues. Except for constraints with references into substructures, we can recursively solve the AIDL model node-by-node.

If constraints refer to substructures, it may not be possible to solve the constraints locally, only modifying their defining structure. We iteratively deepen the parameters and constraints considered, one tree-level at a time, whenever a subproblem cannot be solved. This is a greedy approach to finding a minimal solvable subproblem at each stage of the recursive solve. To further reduce changes to solved substructures, we attempt the iterative deepening in several stages; first only adding 2D structure translational degrees of freedom on deeper attempts, then 3D, and finally allowing the local parameters and geometry on subnodes to be modified on the final pass. This preferentially preserves the local geometry of already solved nodes, helping to preserve solutions

near the initial parameter values.

Boolean Post-Processing The final stage of evaluating an AIDL model is to recursively collect and combine geometry. AIDL supports boolean union and subtraction through its Solid and Hole Structures, operations that are performed on surfaces and volumes, but its highest dimensional geometry primitives are curves. To allow booleans, curve endpoints in Solid and Hole Structures are matched to find closed loops that bound planar regions. This includes geometry from attached child nodes, post boolean. At each node, a boolean union is first carried out over faces discovered from local and Solid descendant edges, and then boolean subtraction performed with faces from Hole children. This recursion continues until an Assembly Structure or the root is reached, at which point face geometry is propagated without further boolean operations. Drawing geometry is interpreted purely as edges and so propagated directly to the root Structure.

Implementation We forked our constraint solver from SolvesSpace [Wes22], adding support for arbitrary constraint expressions, inequality constraints, extra arithmetic operators, dynamic activate of parameters and constraints, and iterative Newton solving. Boolean operations are implemented with the OpenCascade Modeling Kernel [OCC21].

8.5 Proposed Front End

AIDL enables LLM generated text-to-CAD through a front end generation pipeline. This pipeline takes advantage of 3 properties of AIDL: Structure subtrees are independently executable, runtime feedback references semantically named objects, and incomplete models are valid and executable. The pipeline, illustrated in Figure 8.4, has 4 generation stages that follow a common **validate-until-correct** pattern. First the LLM is prompted to describe **semantically** (in English) what it

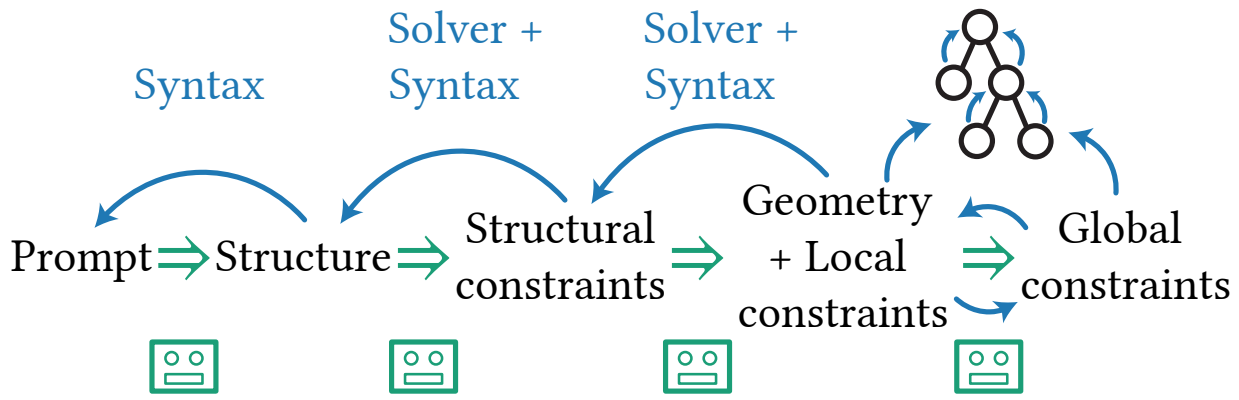


Figure 8.4: LLM Generation Pipeline. An initial text prompt is given to the LLM along with instructions to generate a hierarchical structure. Guided by Solver feedback, the LLM then iterates to add compositional constraints between these structures until a feasible set is found. From these we generate a template AIDL program, and direct the solver to add to it, in a bottom-up fashion, to add first geometry and local constraints, followed by global constraints. Solver feedback is used to iterate until correctness at each of these stages as well.

will generate. Next it is prompted to do so **syntactically**, as code or structured data (json). The front end adds this to the AIDL program under construction, then attempts to execute it, returning any feedback directly to the LLM in case of failure. The syntactic generation is repeated until the program runs without errors, taking advantage of incomplete executability to give feedback on partial generations.

The four stages fit into two halves. The first half generates the Structural hierarchy and constraints. It does so monolithically, and uses structured json as its generation syntax, relying on the front end to generate the final AIDL code. Letting the front end control the code representation up to the structure level gives us enough understanding and control to re-order program statements when in the second half, which is useful for controlling the context and editing scope when the LLM is editing code directly. Defining Structure hierarchies and and constraints is very formulaic in AIDL, so we do not lose much expressiveness by forcing the LLM to use a template.

Geometry has a much richer space of expression, so we allow the LLM to directly write AIDL

code in the second half, in which Geometry and Geometric constraints are defined. This half is proceeds in a bottom-up, pipelined fashion, so its two stages are run back-to-back on each hierarchy node in a postorder traversal of the tree generated in the first half. The remainder of this section describes the four stages.

Structure Generation The LLM is given an overview of the entire generation process, plus the semantic meaning and validity rules of Structures in AIDL. It produces a tree of Structures with semantic names, Structure types, and relative orientations, which is validated to conform to the AIDL model validity rules.

Structure Constraint Generation The LLM is given the names and definitions of all the compositional structure constraints. It generates a list of constraints with types, constrained structures, and parameters. The front end generates AIDL constraints for each of these on last common ancestor of the constrained Structures. Successful execution validates that the structural constraints are feasible together.

Local Geometry and Constraint Generation The LLM is given a generated template program with a hole to fill; create the geometry and local constraints for a given Structure. Because this is completely local to one Structure, we can reorder the template program so that the hole is at the end, requiring only appending code instead of modifying it. Executing the program validates constraint feasibility, and feedback-loop modifications are restricted to operating only on the appended code.

Global Constraint Generation In the final stage, the LLM is again given a template program and generates AIDL code that constrains Geometry across an entire subtree. The template program is still re-ordered for ease-of-editing, as in the prior stage, but because cross-structure constraints

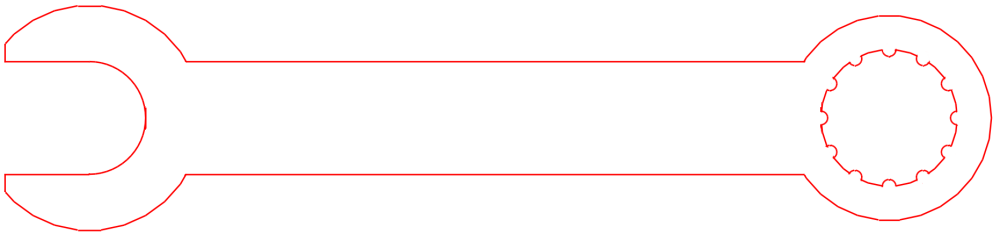


Figure 8.5: The AIDL model given as an example for the single shot generation experiments.

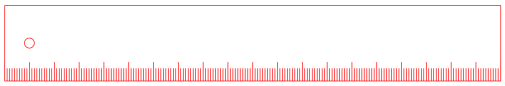
now exist, the code generating any subtree that has been modified in this stage can only be moved as block. Performing this stage bottom-up maximizes the leeway to reorder template code throughout generation, as well as assures that any solve failures are the result of constraint added at the current node in the recursion.

8.6 Preliminary Results

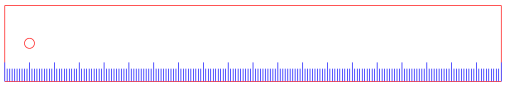
A robust generation frontend for AIDL is under active development, but preliminary experiments with single shot generation show promise for AIDL-backed LLMs as a CAD co-pilot. In these experiments, GPT 4o was provided with a description of the AIDL language along with one complete AIDL program example, the wrench model shown in Figure 8.5, then asked to generate either a ruler or a protractor. We used the generated AIDL code as a starting point to realize the models, which only required minor edits, shown in Figures 8.6 and 8.7.

For the ruler, GPT 4o failed to use a Drawing for the scale marks, which would result in them being fully cut out, while in the protractor example, the angle marks were all assigned to the same variable instead of a list. Correcting each of these mistakes required modifying only a few lines of code.

As a baseline, GPT 4o was also asked to generate SVG code for the same models, using red for cuts, blue for engraving, and making reasonable assumptions for any measurements or parameters.

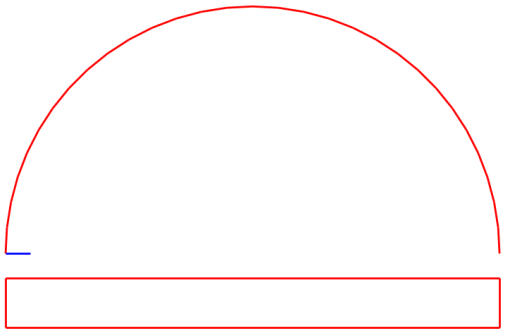


```
...
# Create the centimeter and millimeter markings
ruler.markings = []
for i in range(int(length/mm) + 1):
  if i % 10 == 0:
    # Long marking for each centimeter
    ruler.markings.append(Line((i*mm - length/2, -width/2), (i*mm
        ↪ - length/2, -width/4)))
  else:
    # Shorter marking for each millimeter
    ruler.markings.append(Line((i*mm - length/2, -width/2), (i*mm
        ↪ - length/2, -width/3)))
...
```



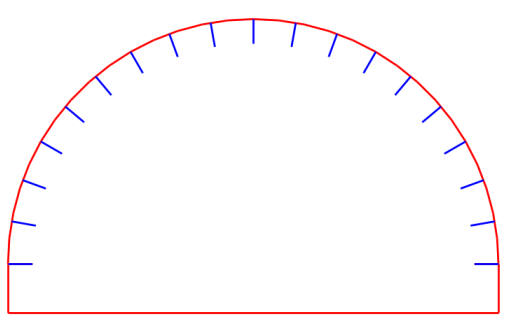
```
...
# Create the centimeter and millimeter markings
ruler.markings = Drawing()
ruler.markings.marks = []
for i in range(int(length/mm) + 1):
  if i % 10 == 0:
    # Long marking for each centimeter
    ruler.markings.marks.append(Line((i*mm - length/2, -width/2),
        ↪ (i*mm - length/2, -width/4)))
  else:
    # Shorter marking for each millimeter
    ruler.markings.marks.append(Line((i*mm - length/2, -width/2),
        ↪ (i*mm - length/2, -width/3)))
...
```

Figure 8.6: GPT 4o generated AIDL code for a ruler (left), and the user corrected version (right). The code diff below shows that only a minor edit was required to fix the model; creating a Drawing Structure for the scale to sit under so that it would be engraved instead of cut.



```
...
base = Solid(shift=(0, -base_height))
...
# Add degree markings
markings = Drawing()

for angle in range(0, 181, 10):
    start_point = ...
    end_point = ...
    markings.lines = Line(start_point, end_point)
...
```



```
...
base = Solid(shift=(0, -base_height / 2))
...
# Add degree markings
markings = Drawing()
markings.lines = []
for angle in range(0, 181, 10):
    start_point = ...
    end_point = ...
    markings.lines.append(Line(start_point, end_point))
...
```

Figure 8.7: GPT 4o generated AIDL code for a protractor (left), and the user corrected version (right). In this instance there were two corrections; adding a factor of one-half to the base's initial position (GPT appears to have erroneously assumed the bottom-origin instead of center-origin convention for rectangles), and creating a list for the markings to be appended to rather than repeatedly reassigning a single reference.

A comparison between the SVG and AIDL results is shown in Figure 8.8. GPT's attempt at an SVG ruler initially appears on-par with the raw AIDL generation; each constructs a ruler with centimeter and millimeter marks, and each makes one mistake: with AIDL, GPT fails to engrave the scale lines, and with SVG, GPT only marks millimeters for the first centimeter of the scale. The AIDL code is a significantly more useful starting point, however, because the mistake requires minimal editing to correct. In the SVG example, each mark is an explicitly placed element with hard-coded endpoints, requiring 190 additional lines of code to complete the model. The protractor example highlights GPT's inability to perform geometric reasoning; it generated (and correctly commented) an SVG line for each angular mark, but completely failed to produce appropriate endpoint coordinates. In contrast, GPT was able to write a formula for generating the proper angular marks in the AIDL code, only failing to properly connect those marks to the model Structure.

8.7 Discussion

All the previous chapters of this dissertation have asked variations of the question "how can we make AI tools work with existing CAD representations, and what can that enable," but AIDL is the first attempt at crafting a new, AI aware CAD representation that makes affordances for LLMs without compromising the core requirements' precision, accuracy, and editability. The final chapter of this dissertation will continue this line of thought to imagine the future of CAD.

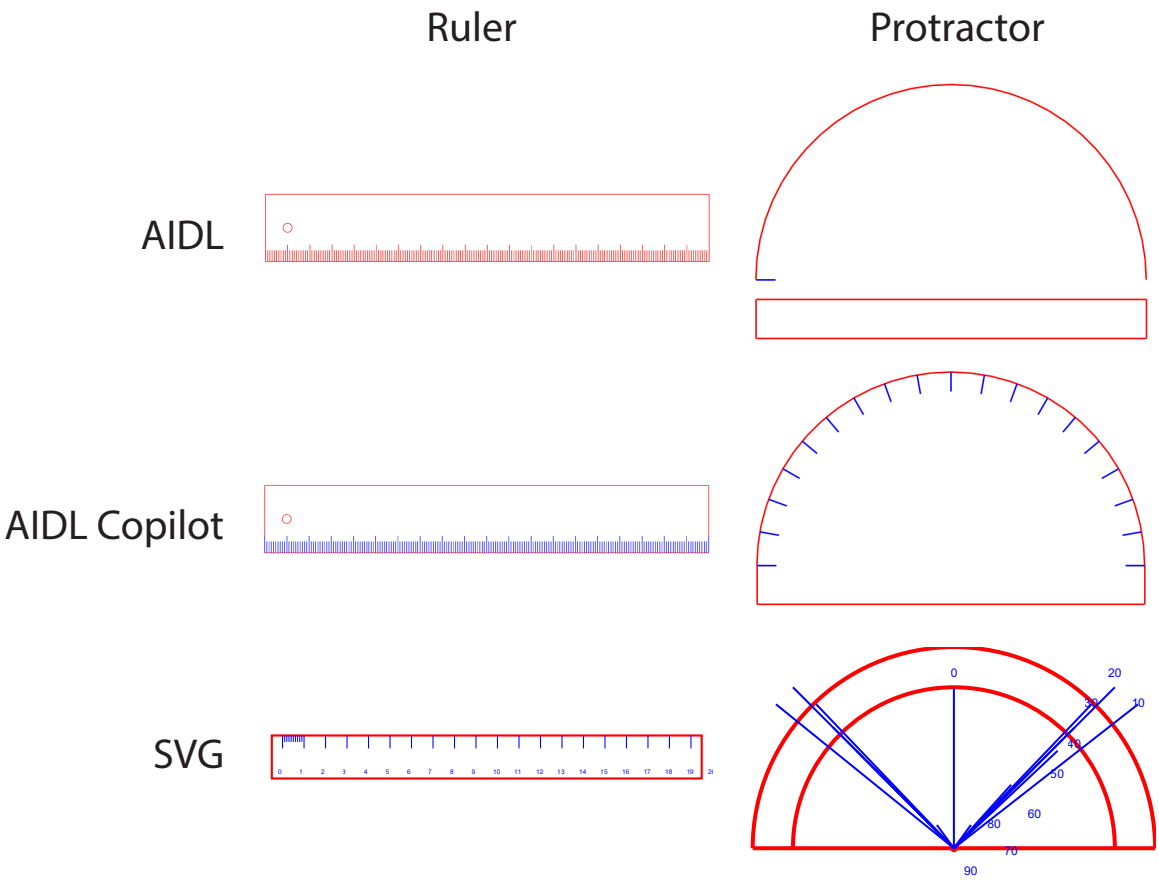


Figure 8.8: Comparing using GPT 4o as a CAD co-pilot with AIDL generation to directly generating SVG output with GPT 4o. The AIDL approach provides a more beneficial initial design. For the ruler it successfully places all millimeter lines whereas SVG generation stops after 10.

Chapter 9

The Future of AI in CAD

This thesis has explored the application of AI to Computer Aided Design from the perspective of the design loop and the representations used within it. In this chapter, I consider how to connect and expand the ideas of this dissertation, to construct the near future of CAD, then look even further ahead to the role of AI in design as a whole.

9.1 The Near Future of CAD

There are many smaller details or improvements that could be made to the systems and ideas described in this thesis. These include extending self-supervised pre-training to all levels of topology, gradient-based B-Rep optimization, or extending the AIDL front-end to 3D. Some of these have since been tackled by other groups, like automating mating of full assemblies simultaneously, while others are waiting for breakthroughs in related areas of research, like differentiable B-Rep kernels so ReParam CAD can be applied to more mainstream representations. Discussion of these avenues can be found in the published versions of these works [JHC+21b, JNK+23, JHK+23, KJA+23].

The most tantalizing low-hanging fruit emerging from this dissertation is instead to break the silo between the two main representations and kinds of models used throughout the thesis; representation learning of B-Reps, and generative modeling on CAD programs. I will propose using our pre-trained B-Rep embeddings as an encoding for adding CAD to multi-modal generative models, and enabling the evolution of existing CAD languages into neurosymbolic CAD languages that will bring automatic and predictive capabilities to CAD modeling.

9.1.1 Neurosymbolic CAD Languages

The systems described in this dissertation dealt with either the geometric representation of CAD objects or the programmatic, but not both. We saw in AIDL that existing generative models have difficulty understanding geometry in general, let alone CAD-specific geometry. Generative models are increasingly being trained to work with multiple data modalities. The sheer number of raw CAD models and programs available makes them ripe to add to this cohort, but there are several challenges that currently stand in the way, some of which this dissertation solves, and some that need fresh ideas.

Multimodal models rely on two things to integrate non-text modalities; robust pre-trained vectorized or tokenized representations of the new modality, and paired examples that tie them to others. SB-GCN and Self-Supervised B-Rep learning provide vector embeddings for B-Reps, and large databases of full construction sequences and edit histories for CAD models are freely available. So, these requirements are satisfied.

Another set of roadblocks to multimodal CAD AI are the opacity of queries and references in CAD programs due to black-box kernels, and the Balkanization of the programmatic side of the CAD ecosystem; while every major CAD system uses B-Reps internally, they all have their own flavor CAD programs, referencing, and querying system that can depend on their choice of opaque

CAD kernel. I believe that there is one solution to both problems; creating a neurosymbolic CAD language that can transpile to and from existing CAD systems, then training multimodal models on the transpiled programs.

Building Neurosymbolic CAD This is possible because while CAD systems vary in their implementation specifics, they all support the same basic set of geometric operations that are implemented by their underlying kernel, just with their own particular "syntax." The most unique parts of each system are their methods of querying for and specifying references. Understanding how CAD program references associate with B-Rep geometry requires knowledge of system and kernel internals, making them ill-suited as a general construct. Their purpose and output are always the same, however; for a particular B-Rep state, given a description, retrieve the topological entities that are used in an operation. We showed in B-Rep matching that it is possible to associate references between variants of a B-Rep, essentially learning to answer the query "which reference is equivalent to this element in the other B-Rep." What I propose to do is directly learn to reference in the context of an operation and a B-Rep. By training a key-query system that can distinguish between topological entities given a query-key and a B-Rep to query over, we can replace system-specific references with a vectorized or tokenized key that is system agnostic. Transpilation could be achieved by optimizing over keys to go from specific system to neurosymbolic CAD, and by evaluating a query to go from neurosymbolic CAD to an existing CAD system.

The final piece of this puzzle is how to train the model. CAD programs have a wonderful property that they can produce valid B-Rep output after operation. This is necessary for them to work in graphical editing environments, because the designer interacts with a visible depiction of the B-Rep. Simply pairing complete CAD programs with their final B-Rep output both wastes tons of potential data about intermediate representations, but also forces a generative model to reason about the evolving internal geometric state and how to reference that state across a potentially

very long program context.

I propose to take advantage of the intermediate B-Rep outputs by representing neurosymbolic CAD programs as code sequences with encoded B-Rep output following every statement. This gives the model intermediate representations to learn how to reason about the evolving geometric state, training it to be a neural program interpreter. It also opens the door to online feedback and correction at generation time since the generated intermediate states can be checked and corrected by evaluating the program mid-generation.

9.1.2 Applications of Neurosymbolic CAD Languages

Copilot for Existing CAD Systems Creating a neurosymbolic CAD language and training a multimodal model on it would unite the early and late stages of the design loop under a single AI model, allowing two-way feedback between design iterations. It would also enable a host of practical applications. There is massive potential for direct integration with existing, widely used CAD systems. Such a system could provide autocompletion, suggestions, or even generation to models in-progress, or facilitate high-level editing directives such as text-to-edit, or generative refactoring of CAD models to, for example, create more flexible parameterizations or expose or isolate a design parameter that was not considered in the initial implementation.

Reverse Engineering A second intriguing application is B-Rep reverse engineering. While many CAD programs exist, the vast majority of models have been divorced from their original programmatic specification, and exist only as B-Rep geometry. These are still useful for modeling because they can be referenced to interface with existing parts, and have forward-edits applied to them, but they cannot be modified because they have no construction history. However, because my proposed neurosymbolic CAD system contains both code and intermediate geometry, a generative model could be trained on reversed sequences, effectively learning to "unmake" a

model to create a hypothesized construction sequence. A powerful observation is that such a model does not need to succeed in completely deconstructing an object, nor in reproducing the true construction sequence. It only needs to produce a sequence that correctly reproduces the output.

9.1.3 Research Challenges

Although this thesis and prior work in data collection provide the construction sequences and representation learning strategy critical to realize neurosymbolic CAD, several technical and research hurdles remain.

A Complete Pre-Trained Embedding In order to build the proposed language and models, the geometric self-supervision method from Chapter 6 would need to be extended to its logical conclusion of training a model that works across all levels of B-Rep topology. While this is straightforward for vertices and edges, an appropriate self-supervision loss for loops, shells, solids, and compounds (heterogeneous collections of B-Reps of various dimensions) is an open question for research. It would also be necessary to design a new input geometry encoder to enable the encoding of variably sized geometric primitives like splines, offset curves, etc.

Data Collection and Generation The same reference opacity that motivates developing neurosymbolic references also makes collecting an appropriate set of training data time-consuming and expensive. Writing a transpiler for geometric CAD operations is tedious but straightforward, but the only way to translate the references is to re-execute the construction sequence in its original CAD system and extract the references and B-Reps from its intermediate steps. This is problematic because the vast majority of available CAD sequences are in Onshape, which can only be accessed via a web API, so massive parallelization is infeasible without their direct support. A

similar project was carried out by DeepCAD [WXZ21a] several years ago, but this dataset focuses on a narrow subset of CAD (sketch and extrude sequences) and did not retain the necessary intermediate detail.

Robust Referencing While using exact topological entity embeddings as neurosymbolic query-keys will work if they are exactly correct, the failure of contrastively trained embeddings in B-Rep matching shows that such a solution is likely to be incredibly brittle. I suspect that using geometric self-supervision pre-training will somewhat improve on these results, but new reference learning strategies are likely necessary. For B-Rep matching, including partial matches as priors was the key to success. It may be necessary to include additional priors, such as the programmatic context of the reference. It may also be necessary to collect empirical data about user intent if CAD system heuristics are insufficient to learn robust referencing.

Semantic Understanding Since available CAD are GUI edited, they generally lack the embedding comments that help LLMs reason about general purpose programming languages. This will make interacting with the model's general knowledge difficult. Recent work like CADTalk [YXP+23] uses generative models to synthetically comment CAD programs, but more progress is required to give meaningful semantic labels to the pieces and substructures of mechanical CAD models, which are typically components of a larger system rather than common objects. Solutions could include labeling parts in the context of semantically understandable assemblies, or collecting data from parts catalogs, which are densely labeled and include B-Rep geometry (but not CAD programs)

B-Rep Autoencoding If the reverse engineering scenario fails to completely disassemble a model, its partial program is only usable if a real B-Rep, not just a representative embedding, can be generated to start from. This would require a way to decode B-Reps from the generated

prior-state representation. Recent work has shown that diffusion models can be used to generate B-Reps with valid geometry and topology [XLJ+24]. This approach is a good starting point for building a geometric B-Rep auto-encoder, in fact this could be the missing loss function necessary to pre-train on B-Rep solids. Such an approach is likely to be noisy, with compounding errors making it unlikely that the generated sequence exactly matches the original input B-Rep. However, assuming that a differentiable B-Rep kernel is successfully developed that is efficient over at least single operations, it could be used in conjunction with the direct B-Rep geometry optimization described in Chapter 6 to incrementally fix the prior-step predictions and operation parameters at each stage so that the generated program and input B-Rep are consistent.

9.2 The Future of AI in Design

The previous section on neurosymbolic CAD proposed an engineering and research effort that would allow AI to reach its full potential in the parts of the design loop that existing CAD systems are already useful for: implementation, and a limited scope of ideation and evaluation. But most of the work of a designer currently lives outside of computers, it is in sketches in a notebook, or thoughts shared in conversation with a collaborator. Traditional CAD modeling formats are far too concrete to support the flexible thinking needed for true creative freedom, and so CAD tools are only picked up late into the process of design. Because current systems and models cannot understand the various high-level ways that ideas are explored and shared, they are unable to participate in the full loop, let alone the long iterative process of design, and are instead simple tools used and then discarded along the way. I envision a future in which CAD AIs can model the *process* of design rather than just its end products. Such systems will be able to assist throughout the design process and will blur the line between tool and collaborator.

There are two technological advances needed to realize this future. The first is to create

generative models that can work seamlessly across design representations at all levels of specificity. Such a multi-level generative AI would be capable of modeling the design thinking of a person in much the same way as a human collaborator, enabling the freedom to work creatively while still enjoying the benefits of computation support. The second technological hurdle is developing models that can concretize imprecisely specified objectives and constraints, and associate these with design representations.

With these two pieces future CAD systems could zoom out in scale to encompass the entire design process from initial inception to final solution realization, and assist in the problem definition phase of design loops while still leaving human designers firmly in control.

9.2.1 Multi-Level Design AI

Generative Models of the Design Process Before delving into multi-level models, I first want to highlight why I think generative models are a natural fit for modeling the process of design. As a design process progresses, and the designer becomes more and more clear of what problem is being solved and how to solve it, the space of possible designs under consideration shrinks. This can be visualized as an internal probability distribution over the space of possible solutions, which gets more and more narrow the more **design priors** they gain through iterative exploration. This is incredibly similar to how a generative design model works. A generative model samples from the distribution $P(S|C)$, where S is a solution, and C is a *context*, which for design problems are design priors. This correspondence between the mental state of the designer and the distribution sampled by the model closely mirrors the similar, but not quite matching mental states of two human designers collaborating, where the context C becomes the shared experiences and communications between those two people. A design AI built around these principles would be similar to a collaborator in that a designer would "communicate" design intent

by modifying the model context.

Multi-Level Models of the Design Process This view of design misses a crucial piece of modeling the mental state of a designer; designers think in representations, not solutions. Especially at the early stages of design, high level representations (effectively partial specifications) allow for vast swathes of design space to be considered with a single instance of an abstract idea. This has two implications for a generative model of the design process: (1) the contextual information communicated to the model is unlikely to be from the same representation schema as the solution space, and (2) it will sometimes be helpful or necessary for the model to sample from different representation schema; e.g. sometimes it is helpful for the model to generate a B-Rep, but other times a sketch is all the designer wants. For a generative model to be truly useful across the design loop, it needs to be able to seamlessly take input and sample from a variety of representation modalities and various level of concreteness, and do so in a way that is consistent: regardless of which representation schema "view" the model is taking of the design process, it represents the same underlying distribution over possible solutions.

Multi-level models could allow generation from, and to a variety of design schema. Since the contextual priors have encoded the process of design, it would be much easier to understand design intent and purpose of specific model features as compared to current CAD systems that only model the end product. Encoding this intent directly will also make designs more editable because that design intent will persist and inform future edits, removing the burden of preserving previously specified constraints from the designer.

9.2.2 AI in Problem Definition

A multi-level generative model could assist a designer across many iterations and levels of design, but there is still a whole quadrant of the loop in which they cannot help: problem definition.

Thus far I have justified this by arguing that design is for solving human problems, so humans should be deciding what the criteria are, but the fact of the matter is that people are rather bad at knowing what they want, and even worse at expressing it — especially in a concrete form amenable to computation. This is where a design AI could help. An AI could interpret vague objective specifications into computable quantities (generation of evaluation code from a text description, for instance). They could automatically score otherwise subjective metrics, like sentiment analysis applied design evaluation. Or they could infer constraints that people find too obvious to encode, but which are crucial to automated exploration.

An AI system capable of modeling the considered solution space through multiple levels of hierarchy, and the constraints and objectives of a designer, will be able to assist across all stages and iterations of the design process without inhibiting creativity the way current software tools do. This is achieved by designing the AI to adapt to human interaction modes, rather than forcing the designer to box their thoughts into the computer's preferred forms of input. Such a system will act more like a collaborator than a tool, and so I call it *Collaborative AI*.

9.2.3 Benefits of Collaborative AI

The generative capabilities of Collaborative AI would allow for extremely quick ideation by helping a designer explore the variations implied by a partial specification. It would also enable stronger evaluation earlier in the design process; such a model could sample more concrete (and thus evaluable) variations within the space of a high-level design and provide quick feedback about computable objectives. Since a Collaborative AI models both the problem definition and the solution space, it could explain how design decisions are derived from design goals. This explainability could further enhance editability and hence iteration velocity by allowing for safer edits: because the Collaborative Model knows which features contribute to objectives, they can

preferentially protect those features while implementing a potentially conflicting edit.

9.3 Ethical Considerations

Whenever a new technology is developed, it is incumbent upon the technologist to consider what unintended consequences could arise from the capabilities their invention enables. Advancements in AI have an unfortunate history of spreading quickly before such considerations have been made, leaving society to scramble for mitigations after-the-fact. With the current crop of generative models, the primary negative externality has been a glut of disinformation and loss of public trust due to the ease of generating false or misleading media that appears genuine. With generative CAD, that problem could extend from the proliferation of dangerous ideas to the proliferation of dangerous objects. From within the system, we could attempt to mitigate this risk by leaning on the explainability afforded by modeling objectives. If our system understands the functional reason an object design has certain features, it could allow us to build features similar to the self-censoring mechanisms of generative text and image models.

9.4 Conclusion

This dissertation started by asking if AI can fit into Computer Aided Design, and if so, how. The success of the systems in the past six chapters answer an emphatic yes; AI can be helpful in actual design processes. They validate my hypothesis that building off the existing primary CAD representations, procedural specification and symbolic geometry, is key to the successful integration of AI into CAD. They also point to a future of design where AI is deeply embedded throughout the design loop, envisioned in Figure 9.1. AI systems of the future will be trained to understand all the myriad design representations, from high to low level, that people employ when

working a design problem, and using this understanding will become indispensable collaborators, helping people more facilely express and realize their design criteria and ideas.

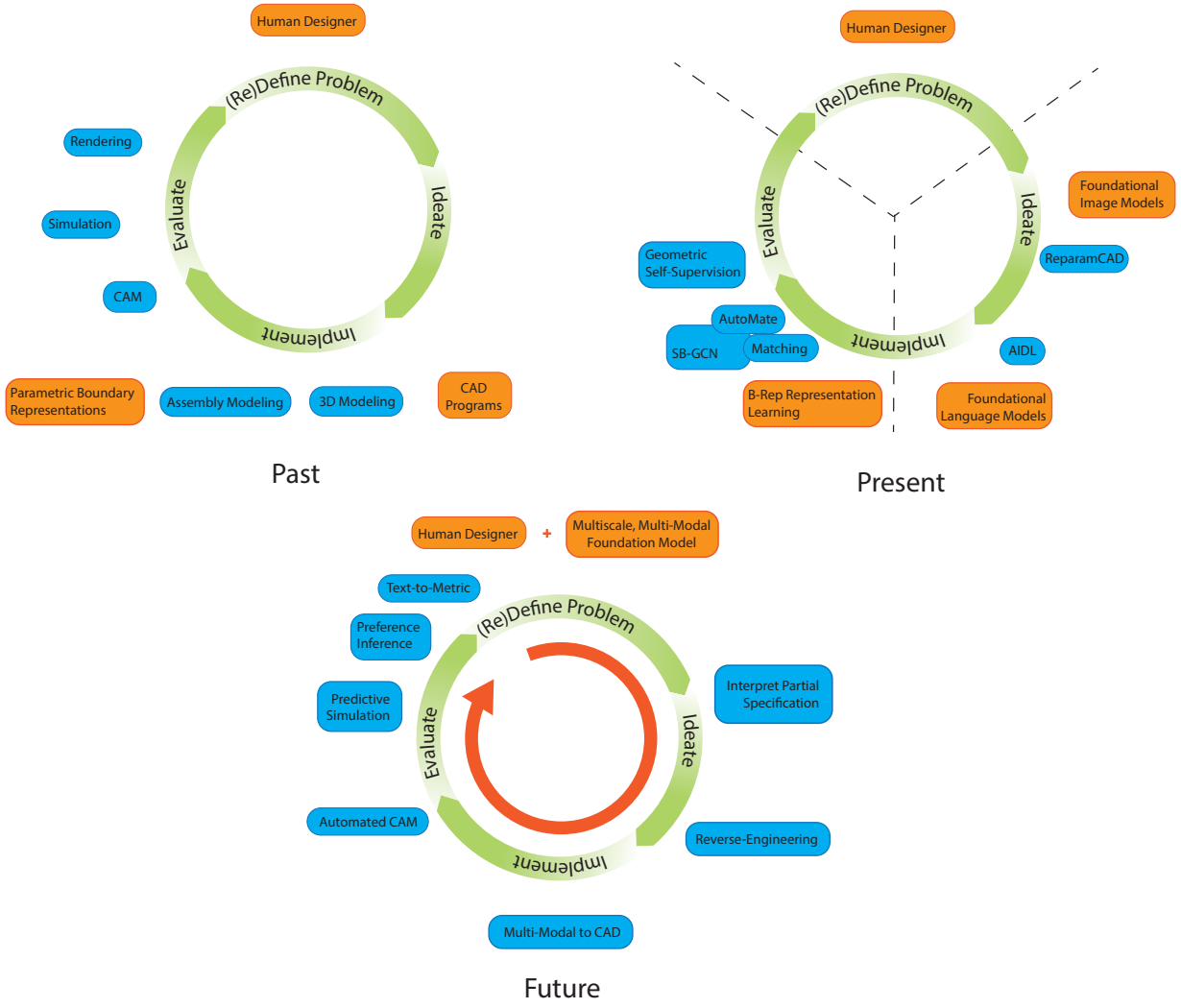


Figure 9.1: The past, present, and future of CAD. Past: current common tasks and use cases for existing CAD systems. These predominantly cluster in the implementation and evaluation phases. Present: the capabilities enabled by the projects in this thesis. B-Rep representation learning opens up the capability of AI assisting in modeling and evaluation tasks, while generative foundational models can interface with CAD programs to aid in ideation and the transition from idea to implementation. These techniques are currently separated however, leaving a divide between the low and high specificity representations in the design loop, and still giving no assistance in problem definition, and cannot carry context throughout the full design process. Future: a vision of how multi-modal, multi-level AI models will be able to understand all forms of representation used in CAD design, and thus be able to share and maintain context throughout the entire design process. These models will also be able to assist in problem definition by interpreting imprecisely expressed goals from designers, and by inferring unspoken designer preferences from examples.

Bibliography

[ADMG17] Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas J Guibas. Learning representations and generative models for 3d point clouds. *arXiv preprint arXiv:1707.02392*, 2017.

[AHS+22] Panos Achlioptas, Ian Huang, Minhyuk Sung, Sergey Tulyakov, and Leonidas Guibas. Changeit3d: Language-assisted 3d shape edits and deformations, 2022.

[BB00] Rafael Bidarra and Willem F Bronsvoort. Semantic feature modelling. *Computer-Aided Design*, 32(3):201–225, 2000.

[BBL+17] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, jul 2017.

[BCE+23a] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[BCE+23b] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of Artificial General Intelligence: Early experiments with GPT-4, mar 2023. arXiv:2303.12712 [cs].

[BCFO04] Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 123–137, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Bha19] *Development of a Pilot Manufacturing Cyberinfrastructure With an Information Rich Mechanical CAD 3D Model Repository*, volume Volume 1: Additive Manufacturing; Manufacturing Equipment and Systems; Bio and Sustainable Manufacturing of *International Manufacturing Science and Engineering Conference*, 06 2019. V001T02A035.

[BLRW16] André Brock, Theodore Lim, James M. Ritchie, and Nick Weston. Generative and discriminative voxel modeling with convolutional neural networks. *CoRR*, abs/1608.04236, 2016.

[BNB05] Rafael Bidarra, Paulos J Nyirenda, and Willem F Bronsvoort. A feature-based solution to the persistent naming problem. *Computer-Aided Design and Applications*, 2(1-4):517–526, 2005.

[BS20] Ilya Baran and Adriana Schulz. B-rep matching for maintaining associativity across cad interoperation, 2020. US Patent 11,288,411.

[BSK+23] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, C. VageeshD, Arun Shankar Iyer, Suresh Parthasarathy, S. Rajamani, B. Ashok, and Shashank P. Shet. CodePlan: Repository-level Coding using LLMs and Planning. September 2023.

[BTL09] Andrea Bunt, Michael Terry, and Edward Lank. Friend or Foe?: Examining CAS Use in Mathematics Research. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 229–238, New York, NY, USA, 2009. ACM.

[BWSK12] Martin Bokeloh, Michael Wand, Hans-Peter Seidel, and Vladlen Koltun. An algebraic model for parameterized shape editing. *ACM Transactions on Graphics (TOG)*, 31(4):1–10, 2012.

[Cad24] CadQuery. Cadquery. https://github.com/CadQuery/cadquery, 2024.

[CBS23] Dan Cascaval, Rastislav Bodik, and Adriana Schulz. A lineage-based referencing dsl for computer-aided design. *Proceedings of the ACM on Programming Languages*, 7(PLDI):76–99, 2023.

[CCS+19] Kevin Chen, Christopher B. Choy, Manolis Savva, Angel X. Chang, Thomas Funkhouser, and Silvio Savarese. Text2shape: Generating shapes from natural language by learning joint embeddings. In C. V. Jawahar, Hongdong Li, Greg Mori, and Konrad Schindler, editors, *Computer Vision – ACCV 2018*, pages 100–116, Cham, 2019. Springer International Publishing.

[CGL+23] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. Flashfill++: Scaling programming by example by cutting to the chase. In *Principles of Programming Languages*. ACM SIGPLAN, ACM, January 2023.

[CK10] Siddhartha Chaudhuri and Vladlen Koltun. Data-driven suggestions for creativity support in 3D modeling. *ACM Transactions on Graphics*, 29(6):183:1–183:10, dec 2010.

[CKGF13] Siddhartha Chaudhuri, Evangelos Kalogerakis, Stephen Giguere, and Thomas Funkhouser. Attribit: Content creation with semantic attributes. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, page 193–202, New York, NY, USA, 2013. Association for Computing Machinery.

[CKGK11] Siddhartha Chaudhuri, Evangelos Kalogerakis, Leonidas Guibas, and Vladlen Koltun. ACM New York, NY, USA, 2011.

[CKN+23] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[CMHK12] Sang-Uk Cheon, Duhwan Mun, Soonhung Han, and Byung Chul Kim. Name matching method using topology merging and splitting history for exchange of feature-based cad models. *Journal of mechanical science and technology*, 26(10):3201–3212, 2012.

[CPAL22] Gianmarco Cherchi, Fabio Pellacini, Marco Attene, and Marco Livesu. Interactive and robust mesh booleans. *ACM Transactions on Graphics (SIGGRAPH Asia 2022)*, 41(6), 2022.

[CRH+20] Weijuan Cao, Trevor Robinson, Yang Hua, Flavien Boussuge, Andrew R. Colligan, and Wanbin Pan. Graph representation of 3d cad models for machining feature recognition with deep learning. volume Volume 11A: 46th Design Automation Conference (DAC) of *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 08 2020.

[CSQ+21] Dan Cascaval, Mira Shalah, Phillip Quinn, Rastislav Bodík, Maneesh Agrawala, and Adriana Schulz. Differentiable 3d CAD programs for bidirectional editing. *CoRR*, abs/2110.01182, 2021.

[CZ19a] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. *IEEE Computer Vision and Pattern Recognition (CVPR)*, 2019.

[CZ19b] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5939–5948, 2019.

[DCC+16] Jorge Dorribo Camba, Manuel Contero, et al. Parametric cad modeling: An analysis of strategies for design reusability. 2016.

[DGE15] Carl Doersch, Abhinav Gupta, and Alexei A Efros. Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1422–1430, 2015.

[DIP+18a] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Transactions on Graphics*, 37(6), dec 2018.

[DIP+18b] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. InverseCSG: Automatic Conversion of 3D Models to CSG Trees. *ACM Trans. Graph.*, 37(6):213:1–213:16, dec 2018.

[DJJL23] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration Code Generation via ChatGPT. 2023. Publisher: arXiv Version Number: 2.

[DYDZ22] Bailin Deng, Yuxin Yao, Roberto M Dyke, and Juyong Zhang. A survey of non-rigid 3d registration. In *Computer Graphics Forum*, volume 41, pages 559–589. Wiley Online Library, 2022.

[ERSLT18] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[EWN+21] 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*, PLDI 2021, page 835–850, New York, NY, USA, 2021. Association for Computing Machinery.

[FAVK+14] Noa Fish, Melinos Averkiou, Oliver Van Kaick, Olga Sorkine-Hornung, Daniel Cohen-Or, and Niloy J Mitra. Meta-representation of shape families. *ACM Transactions on Graphics (TOG)*, 33(4):1–11, 2014.

[FB18] Grigory Fedyukovich and Rastislav Bodík. Accelerating syntax-guided invariant synthesis. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 251–269, Cham, 2018. Springer International Publishing.

[FH18] Shahjadi Hisan Farjana and Soonhung Han. Mechanisms of persistent identification of topological entities in cad systems: A review. *Alexandria engineering journal*, 57(4):2837–2849, 2018.

[FKS+04] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 652–663, New York, NY, USA, aug 2004. Association for Computing Machinery.

[FL19a] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[FL19b] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. *arXiv:1903.02428 [cs, stat]*, apr 2019. arXiv: 1903.02428.

[FMd23] Adejuyigbe O. Fajemisin, Donato Maragno, and Dick den Hertog. Optimization with constraint learning: A framework and survey. *European Journal of Operational Research*, 2023.

[FSG17] Haoqiang Fan, Hao Su, and Leonidas Guibas. A point set generation network for 3d object reconstruction from a single image. *CVPR*, 2017.

[FZC+22] Rao Fu, Xiao Zhan, Yiwen Chen, Daniel Ritchie, and Srinath Sridhar. Shapecrafter: A recursive text-conditioned 3d shape generation model. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[GBL+21a] Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, and Stefano Saliceti. Computer-aided design as language. *Advances in Neural Information Processing Systems*, 34, 2021.

[GBL+21b] Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, and Stefano Saliceti. Computer-Aided Design as Language. In *Advances in Neural Information Processing Systems*, volume 34, pages 5885–5897. Curran Associates, Inc., 2021.

[GDC+23] Wenhan Xiong Grattafiori, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[GFK+18] Thibault Groueix, Matthew Fisher, Vladimir G Kim, Bryan C Russell, and Mathieu Aubry. Atlasnet: A papier-mache approach to learning 3d surface generation. *arXiv preprint arXiv:1802.05384*, 2018.

[GKOM18] Paul Guerrero, Yanir Kleiman, Maks Ovsjanikov, and Niloy J. Mitra. PCPNET: Learning Local Shape Properties from Raw Point Clouds. *Computer Graphics Forum*, 37(2):75–85, may 2018. arXiv: 1710.04954.

[GLP+22a] Hao-Xiang Guo, Shilin Liu, Hao Pan, Liu Yang, Xin Tong, and Baining Guo. Complexgen: Cad reconstruction by b-rep chain complex generation. *ACM Transactions on Graphics (TOG)*, 39(4):106:1–106:14, 2022.

[GLP+22b] Haoxiang Guo, Shilin Liu, Hao Pan, Yang Liu, Xin Tong, and Baining Guo. ComplexGen: CAD reconstruction by B-rep chain complex generation. *ACM Transactions on Graphics*, 41(4):129:1–129:18, jul 2022.

[Gra] Free CAD designs, files & 3D models | the GrabCAD community library. `https://grabcad.com/library`.

[GSK18] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. Unsupervised representation learning by predicting image rotations. In *ICLR 2018*, 2018.

[GSMCO09a] Ran Gal, Olga Sorkine, Niloy Mitra, and Daniel Cohen-Or. iWires: An analyze-and-edit approach to shape manipulation. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH)*, 28(3):33:1–33:10, 2009.

[GSMCO09b] Ran Gal, Olga Sorkine, Niloy J Mitra, and Daniel Cohen-Or. iwires: An analyze-and-edit approach to shape manipulation. In *ACM SIGGRAPH 2009 papers*, pages 1–10. 2009.

[GvK18] Diego Gonzalez and Oliver van Kaick. 3D synthesis of man-made objects based on fine-grained parts. *Computers & Graphics*, 74:150–160, aug 2018.

[HAESB20] Zekun Hao, Hadar Averbuch-Elor, Noah Snavely, and Serge Belongie. Dualsdf: Semantic shape manipulation using a two-level representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7631–7641, 2020.

[HHF+19] Rana Hanocka, Amir Hertz, Noa Fish, Raja Giryes, Shachar Fleishman, and Daniel Cohen-Or. Meshcnn: A Network with an Edge. *ACM Transactions on Graphics*, 38(4):90:1–90:12, jul 2019.

[HPG+22] Amir Hertz, Or Perel, Raja Giryes, Olga Sorkine-Hornung, and Daniel Cohen-Or. Spaghetti: Editing implicit shapes through part aware generation. *ACM Trans. Graph.*, 41(4), jul 2022.

[HZF+20] Jilei Huang, Guanqi Zhan, Qingnan Fan, Kaichun Mo, Lin Shao, Baoquan Chen, Leonidas J. Guibas, and Hao Dong. Generative 3D Part Assembly via Dynamic Graph Learning. *Advances in Neural Information Processing Systems*, 33, 2020.

[IMH05] Takeo Igarashi, Tomer Moscovich, and John F Hughes. As-rigid-as-possible shape manipulation. *ACM transactions on Graphics (TOG)*, 24(3):1134–1141, 2005.

[IPR10] Rahinah Ibrahim and Farzad Pour Rahimian. Comparison of CAD and manual sketching tools for teaching architectural design. *Automation in Construction*, 19(8):978–987, dec 2010.

[JBPS11] Alec Jacobson, Ilya Baran, Jovan Popovic, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.*, 30(4):78, 2011.

[JBX+20a] R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. ShapeAssembly: learning to generate programs for 3D shape structure synthesis. *ACM Transactions on Graphics*, 39(6):234:1–234:20, nov 2020.

[JBX+20b] R Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Transactions on Graphics (TOG)*, 39(6):1–20, 2020.

[JCG+21] R Kenny Jones, David Charatan, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. Shapemod: macro operation discovery for 3d shape programs. *ACM Transactions on Graphics (TOG)*, 40(4):1–16, 2021.

[JGMR23] R Kenny Jones, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. Shapecoder: Discovering abstractions for visual programs from unstructured primitives. *arXiv preprint arXiv:2305.05661*, 2023.

[JHC+21a] Benjamin Jones, Dalton Hildreth, Duowen Chen, Ilya Baran, Vladimir G. Kim, and Adriana Schulz. Automate: A dataset and learning approach for automatic mating of cad assemblies. *ACM Transactions on Graphics*, 40(6), dec 2021.

[JHC+21b] Benjamin Jones, Dalton Hildreth, Duowen Chen, Ilya Baran, Vladimir G. Kim, and Adriana Schulz. AutoMate: a dataset and learning approach for automatic mating of CAD assemblies. *ACM Transactions on Graphics*, 40(6):227:1–227:18, dec 2021.

[JHHR22] R Kenny Jones, Aalia Habib, Rana Hanocka, and Daniel Ritchie. The neurally-guided shape parser: Grammar-based labeling of 3d shape regions with approximate inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11614–11623, 2022.

[JHK+23] Benjamin T Jones, Michael Hu, Milin Kodnongbua, Vladimir G Kim, and Adriana Schulz. Self-supervised representation learning for cad. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21327–21336, 2023.

[JHTG20] Chiyu Jiang, Jingwei Huang, Andrea Tagliasacchi, and Leonidas J Guibas. Shapeflow: Learnable deformation flows among 3d shapes. *Advances in Neural Information Processing Systems*, 33:9745–9757, 2020.

[JLD+22] Pradeep Kumar Jayaraman, Joseph G Lambourne, Nishkrit Desai, Karl DD Willis, Aditya Sanghi, and Nigel JW Morris. Solidgen: An autoregressive model for direct b-rep synthesis. *arXiv preprint arXiv:2203.13944*, 2022.

[JLD+23] Pradeep Kumar Jayaraman, Joseph George Lambourne, Nishkrit Desai, Karl Willis, Aditya Sanghi, and Nigel J. W. Morris. SolidGen: An Autoregressive Model for Direct B-rep Synthesis. *Transactions on Machine Learning Research*, feb 2023.

[JMB+21] Ajay Jain, Ben Mildenhall, Jonathan T. Barron, P. Abbeel, and Ben Poole. Zero-shot text-guided object generation with dream fields. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 857–866, 2021.

[JNK+23] Benjamin Jones, James Noeckel, Milin Kodnongbua, Ilya Baran, and Adriana Schulz. B-rep matching for collaborating across cad systems. *ACM Transactions on Graphics*, 42(6):227:1–227:18, aug 2023.

[JNS23] Rijul Jain, Wode Ni, and Joshua Sunshine. Generating domain-specific programs for diagram authoring with large language models. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH 2023, page 70–71, New York, NY, USA, 2023. Association for Computing Machinery.

[JSL+21] Pradeep Kumar Jayaraman, Aditya Sanghi, Joseph G Lambourne, Karl DD Willis, Thomas Davies, Hooman Shayani, and Nigel Morris. Uv-net: Learning from boundary representations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11703–11712, 2021.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KFE12] R. Killick, P. Fearnhead, and I. A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598, 2012.

[KJA+23] Milin Kodnongbua, Benjamin Jones, Maaz Bin Safeer Ahmad, Vladimir Kim, and Adriana Schulz. Reparamcad: Zero-shot cad re-parameterization for interactive manipulation. In *SIGGRAPH Asia 2023 Conference Papers*, SA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[KMJ+19] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. ABC: A Big CAD Model Dataset for Geometric Deep Learning. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9593–9603, Long Beach, CA, USA, jun 2019. IEEE.

[KMRW14] Durk P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Advances in neural information processing systems*, pages 3581–3589, 2014.

[KS18] Robert Kirkwood and James A Sherwood. Sustained cad/cae integration: integrating with successive versions of step or iges files. *Engineering with Computers*, 34(1):1–13, 2018.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[KW17] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*, feb 2017. arXiv: 1609.02907.

[KXBP22] Nasir Mohammad Khalid, Tianhao Xie, Eugene Belilovsky, and Tiberiu Popa. CLIP-mesh: Generating textured meshes from text using pretrained image-text models. In *SIGGRAPH Asia 2022 Conference Papers*. ACM, nov 2022.

[LAK+18] Hubert Lin, Melinos Averkiou, Evangelos Kalogerakis, Balazs Kovacs, Siddhant Ranade, Vladimir G. Kim, Siddhartha Chaudhuri, and Kavita Bala. Learning Material-Aware Local Descriptors for 3D Shapes. *2018 International Conference on 3D Vision (3DV)*, pages 150–159, sep 2018. arXiv: 1810.08729.

[LAZ+23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: may the source be with you!, December 2023. arXiv:2305.06161 [cs].

[LGT+23] Chen-Hsuan Lin, Jun Gao, Luming Tang, Towaki Takikawa, Xiaohui Zeng, Xun Huang, Karsten Kreis, Sanja Fidler, Ming-Yu Liu, and Tsung-Yi Lin. Magic3d: High-resolution text-to-3d content creation, 2023.

[LGZY23] Pu Li, Jianwei Guo, Xiaopeng Zhang, and Dong-ming Yan. SECAD-Net: Self-Supervised CAD Reconstruction by Learning Sketch-Extrude Operations. 2023. Publisher: arXiv Version Number: 1.

[LHK+20] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. Modular primitives for high-performance differentiable rendering. *ACM Transactions on Graphics*, 39(6), 2020.

[LLA+24] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The Next Generation, February 2024. arXiv:2402.19173 [cs].

[LMS16] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Learning representations for automatic colorization. In *European Conference on Computer Vision*, pages 577–593. Springer, 2016.

[LMS+20] Yichen Li, Kaichun Mo, Lin Shao, Minhyuk Sung, and Leonidas Guibas. Learning 3d part assembly from a single image. In *European Conference on Computer Vision*, pages 664–682. Springer, 2020.

[LMTG19a] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. DeepGCNs: Can GCNs Go As Deep As CNNs? In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9266–9275, Seoul, Korea (South), oct 2019. IEEE.

[LMTG19b] Guohao Li, Matthias Müller, Ali Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In *The IEEE International Conference on Computer Vision (ICCV)*, 2019.

[LPBM20] Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J Mitra. Sketch2cad: Sequential cad modeling by sketching in context. *ACM Transactions on Graphics (TOG)*, 39(6):1–14, 2020.

[LPBM22] Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J. Mitra. Free2CAD: parsing freehand drawings into CAD commands. *ACM Transactions on Graphics*, 41(4):1–16, jul 2022.

[LPMG19] Katia Lupinetti, Jean-Philippe Pernot, Marina Monti, and Franca Giannini. Content-based CAD assembly model retrieval: Survey and future challenges. *Computer-Aided Design*, 113(C):62–81, aug 2019.

[LSMS21] Minghua Liu, Minhyuk Sung, Radomir Mech, and Hao Su. Deepmetahandles: Learning deformation meta-handles of 3d meshes with biharmonic coordinates. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12–21, 2021.

[LWJ+21] Joseph G. Lambourne, Karl D. D. Willis, Pradeep Kumar Jayaraman, Aditya Sanghi, Peter Meltzer, and Hooman Shayani. BRepNet: A topological message passing system for solid models. *arXiv:2104.00706 [cs]*, apr 2021. arXiv: 2104.00706.

[LWJ+22a] Joseph George Lambourne, Karl Willis, Pradeep Kumar Jayaraman, Longfei Zhang, Aditya Sanghi, and Kamal Rahimi Malekshan. Reconstructing editable prismatic cad from rounded voxel models. In *SIGGRAPH Asia 2022 Conference Papers*, SA '22, New York, NY, USA, 2022. Association for Computing Machinery.

[LWJ+22b] Joseph George Lambourne, Karl Willis, Pradeep Kumar Jayaraman, Longfei Zhang, Aditya Sanghi, and Kamal Rahimi Malekshan. Reconstructing editable prismatic CAD from rounded voxel models. In *SIGGRAPH Asia 2022 Conference Papers*, SA '22, pages 1–9, New York, NY, USA, nov 2022. Association for Computing Machinery.

[LWQF22] Zhengzhe Liu, Yi Wang, Xiaojuan Qi, and Chi-Wing Fu. Towards implicit text-guided 3d shape generation. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 17875–17885, 2022.

[LXZ+23] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering Code Large Language Models with Evol-Instruct, June 2023. arXiv:2306.08568 [cs].

[LYF17] Jerry Liu, Fisher Yu, and Thomas Funkhouser. Interactive 3d modeling with a generative adversarial network. *International Conference on 3D Vision (3DV)*, 2017.

[Mar23] Mark Marron. Toward programming languages for reasoning: Humans, symbolic systems, and ai agents. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2023, page 136–152, New York, NY, USA, 2023. Association for Computing Machinery.

[MB21] Elie Michel and Tamy Boubekeur. Dag amendment for inverse control of parametric shapes. *ACM Transactions on Graphics (TOG)*, 40(4):1–14, 2021.

[MCST23] Paritosh Mittal, Yen-Chi Cheng, Maneesh Singh, and Shubham Tulsiani. Autosdf: Shape priors for 3d completion, reconstruction and generation, 2023.

[MFW+23] Liane Makatura, Michael Foshey, Bohan Wang, Felix HähnLein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Elaine Owens, Peter Yichen Chen, et al. How can large language models help humans in design and manufacturing? *arXiv preprint arXiv:2307.14377*, 2023.

[MGY+19a] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy Mitra, and Leonidas Guibas. Structurenet: Hierarchical graph networks for 3d shape generation. *ACM Transactions on Graphics (TOG), Siggraph Asia 2019*, 38(6):Article 242, 2019.

[MGY+19b] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy J. Mitra, and Leonidas J. Guibas. Structurenet: Hierarchical Graph Networks for 3d Shape Generation. *ACM Transactions on Graphics*, 38(6):242:1–242:19, nov 2019.

[MGY+20] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy J. Mitra, and Leonidas J. Guibas. StructEdit: Learning Structural Shape Variations. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8856–8865, Seattle, WA, USA, jun 2020. IEEE.

[MON+19] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4460–4470, 2019.

[MPZ20] Aman Mathur, Marcus Pirron, and Damien Zufferey. Interactive Programming for Parametric CAD. *Computer Graphics Forum*, 39(6):408–425, sep 2020.

[MWYG20] Kaichun Mo, He Wang, Xinchen Yan, and Leonidas Guibas. PT2PC: Learning to Generate 3D Point Cloud Shapes from Part Tree Conditions. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, volume 12351, pages 683–701. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

[MWZ+14] Niloy J Mitra, Michael Wand, Hao Zhang, Daniel Cohen-Or, Vladimir Kim, and Qi-Xing Huang. Structure-aware shape processing. In *ACM SIGGRAPH 2014 Courses*, pages 1–21. 2014.

[MZ21] Aman Mathur and Damien Zufferey. Constraint Synthesis for Parametric CAD. In *Pacific Graphics Short Papers*, volume Posters, page 6 pages. The Eurographics Association, 2021. Artwork Size: 6 pages Version Number: 075-080.

[MZC+19] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. PartNet: A Large-Scale Benchmark for Fine-Grained and Hierarchical Part-Level 3D Object Understanding. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 909–918, Long Beach, CA, USA, jun 2019. IEEE.

[NDR+22] Alexander Quinn 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. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 16784–16804. PMLR, 17–23 Jul 2022.

[NF16] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In *European Conference on Computer Vision*, pages 69–84. Springer, 2016.

[NGEB20] Charlie Nash, Yaroslav Ganin, S. M. Ali Eslami, and Peter Battaglia. PolyGen: An Autoregressive Generative Model of 3D Meshes. In *Proceedings of the 37th International Conference on Machine Learning*, pages 7220–7229. PMLR, nov 2020. ISSN: 2640-3498.

[NJD+22] Alex Nichol, Heewoo Jun, Prafulla Dhariwal, Pamela Mishkin, and Mark Chen. Point-e: A system for generating 3d point clouds from complex prompts, 2022.

[NWA+20] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–44, 2020.

[NWP+18] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–31, jul 2018.

[OCC21] OCCT3D. OpenCascade, feb 2021.

[OLGM11] Maks Ovsjanikov, Wilmot Li, Leonidas Guibas, and Niloy J Mitra. Exploration of continuous variability in collections of 3d shapes. *ACM Transactions on Graphics (TOG)*, 30(4):1–10, 2011.

[Ons20] UX Team Onshape. Personal Communication, 2020.

[Ons24] Onshape. Featurescript. https://cad.onshape.com/FsDoc/, 2024.

[ope] Opencascade technology (occt) kernel. `https://www.opencascade.com/open-cascade-technology/`. Accessed: 2022-10-11.

[par] Siemens. parasolid cad kernel. `https://www.plm.automation.siemens.com/global/en/products/plm-components/parasolid.html`. Accessed: 2022-05-19.

[PBG+21a] W. Para, S. Bhat, Paul Guerrero, T. Kelly, N. Mitra, L. Guibas, and Peter Wonka. SketchGen: Generating Constrained CAD Sketches. jun 2021.

[PBG+21b] Wamiq Para, Shariq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas J Guibas, and Peter Wonka. Sketchgen: Generating constrained cad sketches. *Advances in Neural Information Processing Systems*, 34:5077–5088, 2021.

[PFS+19] Jeong Joon Park, Peter Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. *CVPR*, 2019.

[PJBM22] Ben Poole, Ajay Jain, Jonathan T. Barron, and Ben Mildenhall. Dreamfusion: Text-to-3d using 2d diffusion. *arXiv*, 2022.

[PKD+16] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A Efros. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2536–2544, 2016.

[PLH+22] Ofek Pearl, Itai Lang, Yuhua Hu, Raymond A Yeh, and Rana Hanocka. Geocode: Interpretable shape programs. *arXiv preprint arXiv:2212.11715*, 2022.

[PSW+22] David Palmer, Dmitriy Smirnov, Stephanie Wang, Albert Chern, and Justin Solomon. DeepCurrents: Learning implicit representations of shapes with boundaries. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[PTC] PTC Inc. Onshape.

[QSKG17] Charles R. Qi, Hao Su, Mo Kaichun, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 77–85, Honolulu, HI, jul 2017. IEEE.

[QYSG17] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. *arXiv:1706.02413 [cs]*, jun 2017. arXiv: 1706.02413.

[RBL+22] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10674–10685, 2022.

[RGJ+23] Daniel Ritchie, Paul Guerrero, R. Kenny Jones, Niloy J. Mitra, Adriana Schulz, Karl D. D. Willis, and Jiajun Wu. Neurosymbolic Models for Computer Graphics. *Computer Graphics Forum*, 2023.

[RKH+21] 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. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 18–24 Jul 2021.

[RZC+22] Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, and Junzhe Zhang. ExtrudeNet: Unsupervised Inverse Sketch-and-Extrude for Shape Parsing, sep 2022. arXiv:2209.15632 [cs].

[Sah20] Yusuf Sahillioğlu. Recent advances in shape correspondence. *The Visual Computer*, 36(8):1705–1721, 2020.

[SB09] Olga Sorkine and Mario Botsch. Interactive shape modeling and deformation. In *Eurographics (Tutorials)*, pages 11–37, 2009.

[SBSCO06] Andrei Sharf, Marina Blumenkrants, Ariel Shamir, and Daniel Cohen-Or. SnapPaste: an interactive technique for easy mesh composition. *The Visual Computer*, 22(9):835–844, sep 2006.

[SC07] Catherine Stones and Tom Cassidy. Comparing synthesis strategies of novice graphic designers using digital and traditional design tools. *Design Studies*, 28(1):59–72, jan 2007.

[SCL+22] Aditya Sanghi, Hang Chu, Joseph G. Lambourne, Ye Wang, Chin-Yi Cheng, Marco Fumero, and Kamal Rahimi Malekshan. Clip-forge: Towards zero-shot text-to-shape generation. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 18582–18592, 2022.

[SCOL+04] Olga Sorkine, Daniel Cohen-Or, Yaron Lipman, Marc Alexa, Christian Rössl, and H-P Seidel. Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 175–184, 2004.

[SCS+22] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily L Denton, Kamyar Ghasemipour, Raphael Gontijo Lopes, Burcu Karagol Ayan, Tim Salimans, Jonathan Ho, David J Fleet, and Mohammad Norouzi. Photorealistic text-to-image diffusion models with deep language understanding. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 36479–36494. Curran Associates, Inc., 2022.

[SDS+23] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. Generalized Planning in PDDL Domains with Pretrained Large Language Models. 2023. Publisher: arXiv Version Number: 1.

[SFL+23] Aditya Sanghi, Rao Fu, Vivian Liu, Karl Willis, Hooman Shayani, Amir Hosein Khasahmadi, Srinath Sridhar, and Daniel Ritchie. Clip-sculptor: Zero-shot generation of high-fidelity and diverse shapes from natural language, 2023.

[shi22] *Material Prediction for Design Automation Using Graph Representation Learning*, volume Volume 3A: 48th Design Automation Conference (DAC) of *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 08 2022. V03AT03A001.

[Sie] Siemens AG. Parasolid.

[SJA+20] Minhyuk Sung, Zhenyu Jiang, Panos Achlioptas, Niloy J. Mitra, and Leonidas J. Guibas. Deformsyncnet: Deformation transfer via synchronized shape deformation spaces. *ACM Trans. Graph.*, 39(6), nov 2020.

[SJK+23] Junyoung Seo, Wooseok Jang, Min-Seop Kwak, Jaehoon Ko, Hyeonsu Kim, Junho Kim, Jin-Hwa Kim, Jiyoung Lee, and Seungryong Kim. Let 2d diffusion model know 3d-consistency for robust text-to-3d generation, 2023.

[sol] Solidworks. `https://www.solidworks.com/`. Accessed: 2022-10-11.

[SOZA20] Ari Seff, Yaniv Ovadia, Wenda Zhou, and Ryan P. Adams. SketchGraphs: A Large-Scale Dataset for Modeling Relational Geometry in Computer-Aided Design. *arXiv:2007.08506 [cs, stat]*, jul 2020. arXiv: 2007.08506.

[SR07] Steven Spitz and Ari Rappoport. Boundary representation per feature methods and systems, Oct 2007. US Patent 7,277,835.

[SSK+17] Minhyuk Sung, Hao Su, Vladimir G. Kim, Siddhartha Chaudhuri, and Leonidas Guibas. ComplementMe: Weakly-Supervised Component Suggestions for 3D Modeling. *ACM Transactions on Graphics*, 36(6):1–12, nov 2017. arXiv: 1708.01841.

[SSL+14] Adriana Schulz, Ariel Shamir, David IW Levin, Pitchaya Sitthi-Amorn, and Wojciech Matusik. Design and fabrication by example. *ACM Transactions on Graphics (TOG)*, 33(4):1–11, 2014.

[SSP07] Robert W Sumner, Johannes Schmid, and Mark Pauly. Embedded deformation for shape manipulation. In *ACM siggraph 2007 papers*, pages 80–es. 2007.

[SZRA21] Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P Adams. Vitruvion: A generative model of parametric cad sketches. *arXiv preprint arXiv:2109.14124*, 2021.

[SZRA22] Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P. Adams. Vitruvion: A Generative Model of Parametric CAD Sketches. Technical Report arXiv:2109.14124, arXiv, apr 2022. arXiv:2109.14124 [cs] type: article.

[TGLX18] Qingyang Tan, Lin Gao, Yu-Kun Lai, and Shihong Xia. Variational autoencoders for deforming 3d mesh models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[THM+13] Songqiao Tao, Zhengdong Huang, Lujie Ma, Shunsheng Guo, Shuting Wang, and Youbai Xie. Partial retrieval of cad models based on local surface region decomposition. *Comput. Aided Des.*, 45(11):1239–1252, nov 2013.

[TSG+17] Shubham Tulsiani, Hao Su, Leonidas J Guibas, Alexei A Efros, and Jitendra Malik. Learning shape abstractions by assembling volumetric primitives. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2635–2643, 2017.

[UyCS+22] Mikaela Angelina Uy, Yen yu Chang, Minhyuk Sung, Purvi Goel, Joseph Lambourne, Tolga Birdal, and Leonidas Guibas. Point2cyl: Reverse engineering 3d objects from point clouds to extrusion cylinders. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[VCC+17] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[VCC+18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[VGLM13] Jan H Vandenbrande, Thomas A Grandine, Miriam Lucian, and John Monahan. Methods and apparatus for automated part positioning based on geometrical comparisons, Nov 2013. US Patent 8,576,224.

[WCMN19] Weiyue Wang, Duygu Ceylan, Radomir Mech, and Ulrich Neumann. 3dn: 3d deformation network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1038–1046, 2019.

[Wes22] Jonathan Westhues. SolveSpace, nov 2022.

[WJC+21] Karl DD Willis, Pradeep Kumar Jayaraman, Hang Chu, Yunsheng Tian, Yifei Li, Daniele Grandi, Aditya Sanghi, Linh Tran, Joseph G Lambourne, Armando Solar-Lezama, et al. Joinable: Learning bottom-up assembly of parametric cad joints. *arXiv preprint arXiv:2111.12772*, 2021.

[WJL+21] Karl D. D. Willis, Pradeep Kumar Jayaraman, Joseph G. Lambourne, Hang Chu, and Yewen Pu. Engineering Sketch Generation for Computer-Aided Design. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 2105–2114, Nashville, TN, USA, jun 2021. IEEE.

[WPC+20] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2020. Conference Name: IEEE Transactions on Neural Networks and Learning Systems.

[WPL+20] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 Gallery: A Dataset and Environment for Programmatic CAD Reconstruction. *arXiv:2010.02392 [cs]*, oct 2020. arXiv: 2010.02392.

[WPL+21] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 gallery: a dataset and environment for programmatic CAD construction from human design sequences. *ACM Transactions on Graphics*, 40(4):54:1–54:24, jul 2021.

[WSL+19] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic Graph CNN for Learning on Point Clouds. *ACM Transactions on Graphics*, 38(5):146:1–146:12, oct 2019.

[WSS+20] Fangyin Wei, Elena Sizikova, Avneesh Sud, Szymon Rusinkiewicz, and Thomas Funkhouser. Learning to infer semantic parameters for 3d shape editing. In *2020 International Conference on 3D Vision (3DV)*, pages 434–442. IEEE, 2020.

[WWL+19] Zhijie Wu, Xiang Wang, Di Lin, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. SAGNet: structure-aware generative network for 3D-shape modeling. *ACM Transactions on Graphics*, 38(4):91:1–91:14, jul 2019.

[WWW+23] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. Grammar Prompting for Domain-Specific Language Generation with Large Language Models, November 2023. arXiv:2305.19234 [cs].

[WXZ21a] Rundi Wu, Chang Xiao, and Changxi Zheng. DeepCAD: A Deep Generative Network for Computer-Aided Design Models. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6752–6762, Montreal, QC, Canada, oct 2021. IEEE.

[WXZ21b] Rundi Wu, Chang Xiao, and Changxi Zheng. Deepcad: A deep generative network for computer-aided design models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6772–6782, 2021.

[WYKN20] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Comput. Surv.*, 53(3), jun 2020.

[WZL+18] Nanyang Wang, Yinda Zhang, Zhuwen Li, Yanwei Fu, Wei Liu, and Yu-Gang Jiang. Pixel2mesh: Generating 3d mesh models from single rgb images. In *ECCV*, 2018.

[WZX+20] Rundi Wu, Yixin Zhuang, Kai Xu, Hao Zhang, and Baoquan Chen. PQ-NET: A Generative Part Seq2Seq Network for 3D Shapes. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 826–835, Seattle, WA, USA, jun 2020. IEEE.

[XLJ+24] Xiang Xu, Joseph G Lambourne, Pradeep Kumar Jayaraman, Zhengqing Wang, Karl DD Willis, and Yasutaka Furukawa. Brepgen: A b-rep generative diffusion model with structured latent geometry. *arXiv preprint arXiv:2401.15563*, 2024.

[XPC+21a] Xianghao Xu, Wenzhe Peng, Chin-Yi Cheng, Karl D. D. Willis, and Daniel Ritchie. Inferring cad modeling sequences using zone graphs. In *CVPR*, 2021.

[XPC+21b] Xianghao Xu, Wenzhe Peng, Chin-Yi Cheng, Karl D.D. Willis, and Daniel Ritchie. Inferring CAD Modeling Sequences Using Zone Graphs. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6058–6066, Nashville, TN, USA, jun 2021. IEEE.

[XSZ+23] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. WizardLM: Empowering Large Language Models to Follow Complex Instructions, June 2023. arXiv:2304.12244 [cs].

[XWC+23] Jiale Xu, Xintao Wang, Weihao Cheng, Yan-Pei Cao, Ying Shan, Xiaohu Qie, and Shenghua Gao. Dream3d: Zero-shot text-to-3d synthesis using 3d shape prior and text-to-image diffusion models, 2023.

[XWL+22] Xiang Xu, Karl D. D. Willis, Joseph G. Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. SkexGen: Autoregressive Generation of CAD Construction Sequences with Disentangled Codebooks. In *Proceedings of the 39th International Conference on Machine Learning*, pages 24698–24724. PMLR, jun 2022. ISSN: 2640-3498.

[YAK+20] Wang Yifan, Noam Aigerman, Vladimir G Kim, Siddhartha Chaudhuri, and Olga Sorkine-Hornung. Neural cages for detail-preserving 3d deformations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 75–83, 2020.

[Yar13] Evan Yares. The failed promise of parametric cad part 1: From the beginning. https://www.3dcadworld.com/the-failed-promise-of-parametric-cad/, 6 2013. (Accessed on 09/06/2019).

[YCC+20a] Kangxue Yin, Zhiqin Chen, Siddhartha Chaudhuri, Matthew Fisher, Vladimir G. Kim, and Hao Zhang. COALESCE: Component Assembly by Learning to Synthesize Connections. *arXiv:2008.01936 [cs]*, nov 2020. arXiv: 2008.01936.

[YCC+20b] Kangxue Yin, Zhiqin Chen, Siddhartha Chaudhuri, Matthew Fisher, Vladimir G. Kim, and Hao Zhang. COALESCE: component assembly by learning to synthesize connections. *CoRR*, abs/2008.01936, 2020.

[YCHK15] Mehmet Ersin Yumer, Siddhartha Chaudhuri, Jessica K. Hodgins, and Levent Burak Kara. Semantic shape editing using deformation handles. *ACM Transactions on Graphics*, 34(4):86:1–86:12, jul 2015.

[YCL+22a] Fenggen Yu, Zhiqin Chen, Manyi Li, Aditya Sanghi, Hooman Shayani, Ali Mahdavi-Amiri, and Hao Zhang. Capri-net: Learning compact cad shapes with adaptive primitive assembly. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11768–11778, June 2022.

[YCL+22b] Fenggen Yu, Zhiqin Chen, Manyi Li, Aditya Sanghi, Hooman Shayani, Ali Mahdavi-Amiri, and Hao Zhang. Capri-net: Learning compact cad shapes with adaptive primitive assembly. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11768–11778, 2022.

[YFST18] Yaoqing Yang, Chen Feng, Yiru Shen, and Dong Tian. Foldingnet: Point cloud auto-encoder via deep grid deformation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 3, 2018.

[YML+20] Jie Yang, Kaichun Mo, Yu-Kun Lai, Leonidas J. Guibas, and Lin Gao. DSM-Net: Disentangled Structured Mesh Net for Controllable Generation of Fine Geometry. *arXiv:2008.05440 [cs]*, aug 2020. arXiv: 2008.05440.

[YXP+23] Haocheng Yuan, Jing Xu, Hao Pan, Adrien Bousseau, Niloy Mitra, and Changjian Li. Cadtalk: An algorithm and benchmark for semantic commenting of cad programs. *arXiv preprint arXiv:2311.16703*, 2023.

[YYL20a] Jiaxuan You, Rex Ying, and Jure Leskovec. Design space for graph neural networks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[YYL20b] Jiaxuan You, Zhitao Ying, and Jure Leskovec. Design space for graph neural networks. *Advances in Neural Information Processing Systems*, 33:17009–17021, 2020.

[ZCZ20] Z. Zhang, P. Cui, and W. Zhu. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020. Conference Name: IEEE Transactions on Knowledge and Data Engineering.

[ZIE16] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful image colorization. In *European conference on computer vision*, pages 649–666. Springer, 2016.

[ZLWT22] X Zheng, Yang Liu, P Wang, and Xin Tong. Sdf-stylegan: Implicit sdf-based stylegan for 3d shape generation. In *Computer Graphics Forum*, volume 41, pages 52–63. Wiley Online Library, 2022.

[zVW+22] xiaohui zeng, Arash Vahdat, Francis Williams, Zan Gojcic, Or Litany, Sanja Fidler, and Karsten Kreis. Lion: Latent point diffusion models for 3d shape generation. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 10021–10039. Curran Associates, Inc., 2022.

[ZXC+18] Chenyang Zhu, Kai Xu, Siddhartha Chaudhuri, Renjiao Yi, and Hao Zhang. SCORES: Shape Composition with Recursive Substructure Priors. *ACM Trans. Graph.*, 37(6):211:1–211:14, dec 2018.

[ZXK+18] Yizhou Zhang, Yun Xiong, Xiangnan Kong, Shanshan Li, Jinhong Mi, and Yangyong Zhu. Deep Collective Classification in Heterogeneous Information Networks. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*, pages 399–408, Lyon, France, 2018. ACM Press.

Appendix A

SB-GCN

A.1 Data Model

Node Types Table A.1 summarizes the parametric functions in our system, along with their parameters. We exclude the origin parameter because these do not need to correspond to the topological entity location once boundaries are taken into account, and in practice are often far from surfaces, resulting in noise.

Table A.1: Parametric functions and their parameters.

Function	Topology Type	Parameters	Description
plane	face	~~origin~~, normal	A plane.
cylinder	face	~~origin~~, axis, radius	A cylinder.
cone	face	~~origin~~, axis, half-angle	A cone.
torus	face	~~origin~~, axis, major-radius, minor-radius	A torus.
bsurf	face		A NURBS surface.
swept	face		Surface swept along a curve.
spun	face		Surface of rotation about an axis.
blend	face		Surface blending two other surfaces.
outer loop	loop		The outer trimming curve of a face.
inner loop	loop		An inner trimming curve of a face.
line	edge	~~origin~~, direction	A line.
circle	edge	~~origin~~, normal, radius	A circle.
ellipse	edge	~~origin~~, normal, major axis direction, major axis, minor axis.	An ellipse.
bcurve	edge		A B-spline curve.
icurve	edge		The intersection of two surfaces.
spcurve	edge		Curve embedding in a surface.
tcurve	edge		Trimmed curve.
pline	edge		A polyline.
point	vertex	~~origin~~	A point.

Appendix B

AutoMate

Origin Types Each MFC has an *origin type* that specifies how the origin is computed from its reference topological entity. Table B.1 lists all the available origin types, along with the types of geometry they are applicable. Geometry without supported origin types are not able to be referenced to define MFCs. Note that all of these computations except *center* and *point* rely on boundary information, and so are not computable from the parametric geometry a topological entity in isolation, highlighting the need to integrate boundary information to properly understand the mating coordinate frames. Also note that the origin type is highly correlated with the geometry type of the referenced topological entity, which is why it is a useful proxy for our Origin Type baseline.

Table B.1: Supported origin types for mating coordinate frames.

Origin Type	Supported Functions	Description
center	circle, ellipse, loop	The center of a curve or inner loop.
centroid	plane	The centroid of a planar face.
mid point	line	The midpoint of a line segment.
point	point, cone	The location of a point or vertex of a cone.
top axis point	cylinder, torus, cone, spun	The most positive point of a surface of revolution projected onto its axis.
mid axis point	cylinder, torus, cone, spun	The mid-point of a surface of revolution projected onto its axis.
bottom axis point	cylinder, torus, cone, spun	The most negative point of a surface of revolution projected onto its axis.

Computing MCF Orientation The orientation of an MCF is defined by reference to a supported topological entity. Table B.2 lists the geometric functions from which orientations can be determined, and describes how they are computed.

The orientation reference of an MCF only determines the frame's z-axis. The y-axis is computed as the cross product of this z-axis with the x-axis of the coordinate from that the part was modeled in, or the cross product with its y-axis if that x-axis is parallel to the chosen z direction.

B.1 Ablations

We tried sixteen combinations of graph architecture and feature set across the mate location and mate type problem. Figure B.1 shows all of these results, compared along either axis. The mate type problem does not show much difference between methods, but for mate location there is a clear trend that having a GCN is better than not, that more features are better, and that adding features can partly make up for not using a GCN.

Table B.2: Z-axis computation for supported topological entities.

Topological Entity Function(s)	z-Axis
plane	normal
cylinder, cone, torus, spun	axis parameter
inner loop	plane normal if inner loop of a plane
line	along edge in positive parameterization direction
circle, ellipse	plane normal if adjacent to a plane, otherwise axis parameter
point	z-axis of CS part was modeled in.

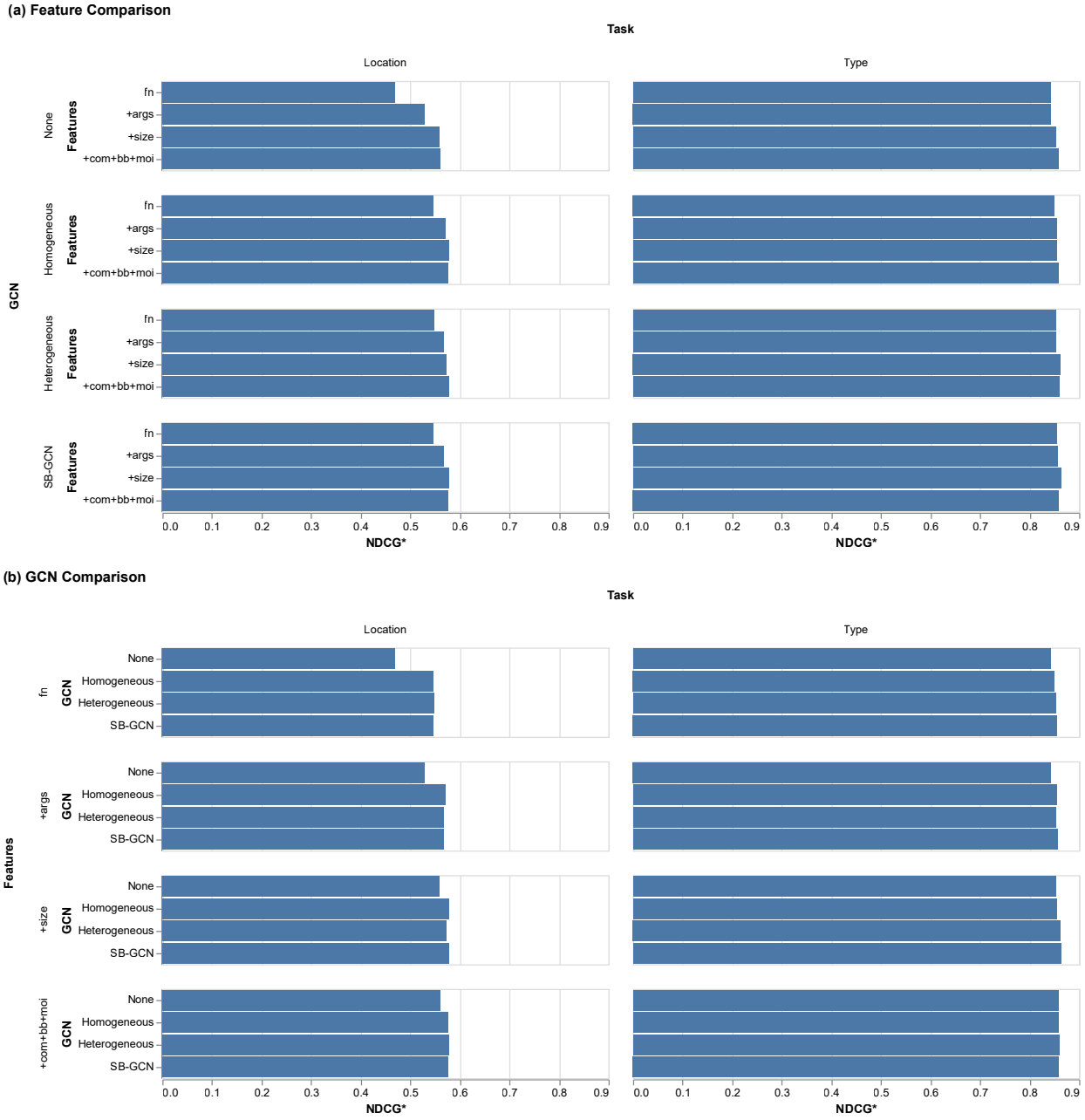


Figure B.1: Validation NDCG* for GCN type and feature set experiments trained against 70k subset. NDCG* is a modified NDCG score that stops counting at the first correct mate. For mate type this is equivalent to NDCG since there is only one correct mate type.

Appendix C

B-rep Matching

C.1 Code and Data

Our code and data is available at `https://github.com/deGravity/brepmatching`.

C.2 Architectural Details

C.2.1 Input Encodings

We initially featurize B-reps with fixed-length vectors for each face, loop, edge, and vertex. These are encoded with the following properties;

Faces:

- Parametric Geometry
- Orientation
- Surface Area
- Circumference
- Axis-Aligned Bounding Box
- Non-Axis-Aligned bounding Box
- Center of Gravity
- Moment of Inertia Tensor

Loops:

- Loop Type

- Length
- Non-Axis-Aligned Bounding Box
- Center of Gravity
- Moment of Inertia Tensor

Edges:

- Parametric Geometry
- Orientation
- Parameterization Range
- Length
- Start Position
- End Position
- Mid-Point Position
- Bounding Box
- Non-Axis-Aligned Bounding Box
- Center of Gravity
- Moment of Inertia Tensor

Vertices:

- Position

Orientation is a boolean value that indicates if the normal (in the case of faces) or tangent (in the case of edges) is parallel or anti-parallel to that of the associated Parametric Geometry. Parametric Geometry is represented as a one-hot encoded function type (one of PLANE, CYLINDER, CONE, SPHERE, TORUS, SPUN, BSURF, OFFSET, SWEPT, BLENDSF, or OTHER for surfaces, LINE, CIRCLE, ELLIPSE, BCURVE, ICURVE, SPCURVE, CPCURVE, PLINE, or OTHER for curves), and a zero-padded array of floating-point arguments for these geometric functions. Functions that can take variable numbers of arguments (like B-geometry) or which depend of construction geometry (like swept surfaces or intersection curves) have empty parameter sets (all zeros). Parameter sets are zero-padded to have length 11, the maximum number of parameters of any of the fixed-encoding-size parametric geometry.

Loop Type is also a one-hot encoded property that represents how a loop is related to the face it bounds. It is one of INNER, OUTER, WIRE, VERTEX, WINDING, INNER_SINGULARITY, LIKELY_OUTER, or LIKELY_INNER. The type of a loop is heuristically determined by the Parasolid [par] geometry kernel.

Table C.1: Initial sample sizes.

Face Count	Original Models Sampled
10-20	200
20-40	500
40-60	1000
60-100	500
100-200	200

C.2.2 Part Featurization Architecture

As described in the main text of the paper, the initial step of our prediction network is to process these input features with SB-GCN [JHC+21a] to create features embedding for each topological entity in a shared embedding space. This encoder works by applying an individual MLP to input features of each type, and then performing a series of hierarchical residual graph message passes from vertices up to faces to aggregate boundary information for higher-dimensional topology. A wider receptive field is acheived by passing messages between adjacent faces, followed by a downward pass from faces back down to vertices to share this wider neighborhood information with the lower-dimensional bounding topology. The full details of this architecture are shown in Figure C.3

C.2.3 Implementation

we implemented our neural network usingn PyTorch Geometric [FL19a]. We used the open source implementation of SB-GCN provided by the authors [JHC+21a] for initial part featurization, and PyTorch Geometric's implimentation of GeneralConv (based on [YYL20b]) to implement the graph attention network described in the main paper.

C.3 Dataset Details

C.3.1 Synthetic Sampling and Edits

Our synthetic data consists of 2400 original B-rep models sampled from the AutoMate Dataset. We randomly sampled models at various levels of complexity as measured by face count, listed in Table C.1. We applied 3 sets of randomly CAD operations to each original model to construct up-to 3 *updated* models for each original (generated CAD operations would sometimes be invalid). After filtering out invalid edits, we collected 6263 pairs of original and updated B-rep models.

Each set of CAD operations consists of up-to 9 geometric and 9 topological edits. Models with fewer than 40 topological edits received between 2 and 5 of each kind of edit, while more complex models received between 6 and 9 of each.

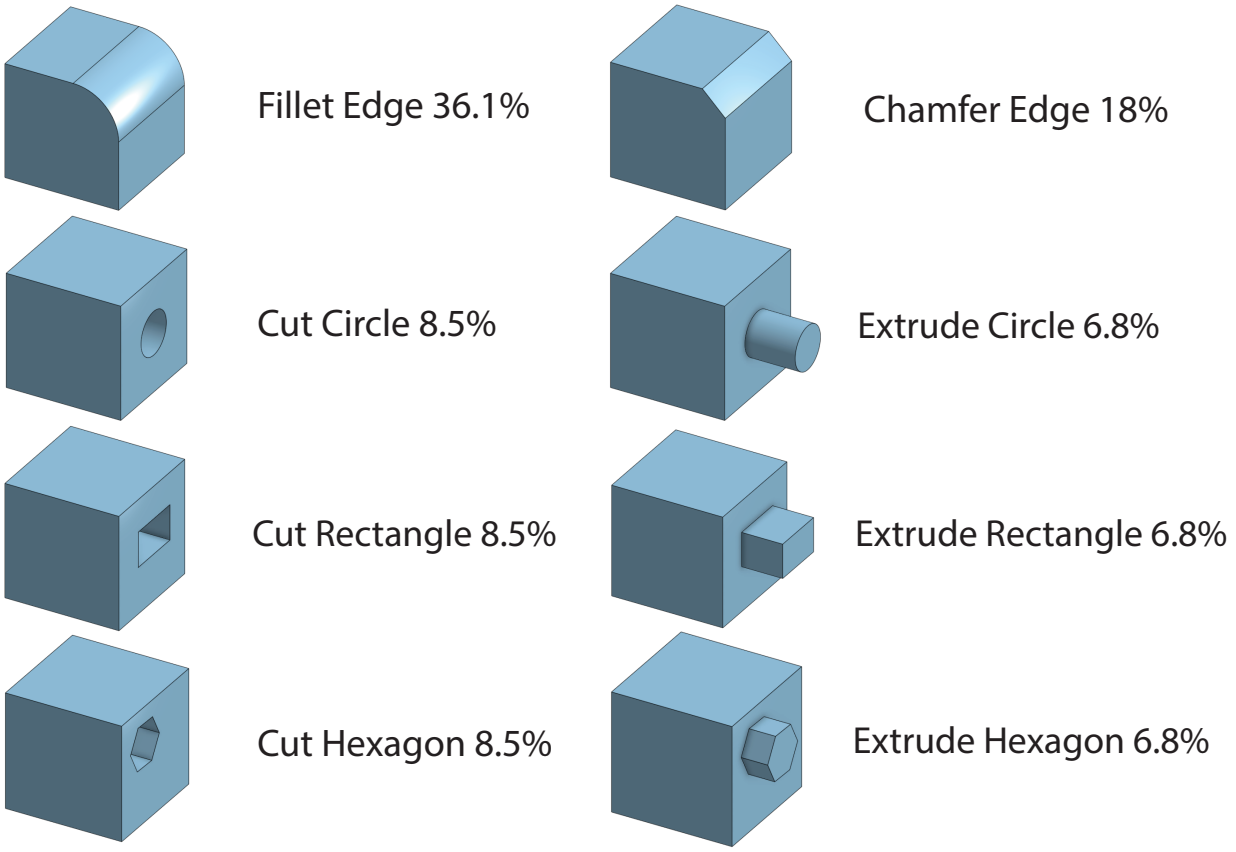


Figure C.1: Types of constructive edits and their frequency

Geometric edits of moving a group of faces (up-to 3 edits per part) or offsetting a face. Topological edits were sampled from the set of 8 options illustrated in Figure C.1 along with their relative frequencies.

C.3.2 Dataset Statistics

Figure C.2 shows the breadth of complexity of both models and edits across our synthetic training dataset and expert validation dataset. Model complexity, as measured by the total number of toplogical entities, is predominantly between 0 and 500 for the original models in both datasets, though the expert data contains models of much higher complexity as well. For the synthetic data, this shifts slightly higher for the updated models because our edit operations can only add complexity.

We measure local model complexity by measuring the valence of each B-rep face in a face-adjacency-graph. By this metric, both datasets have very similar distributions, indicating that while our training data has smaller models overall, the local geometric complexity is on-par with

our expert validated data. We also see the same shift towards more local complexity after updates, suggesting that only using complexity adding operations in our synthetic variation generation is a valid choice.

Finally, we measure the size of our edits by the chamfer distance between the original and updated models. Distances were computed and summed over clouds of 4000 points each, sampled uniformly with respect to surface area. Prior to sampling part pairs were normalized by a transform that centers the Original model on the origin and scales it uniformly to fill a unit cube. We report the logarithm of these distances because they vary over several orders of magnitude. Our synthetic edits average larger than those of the expert dataset. This could mean that our edits are exaggerated in size from what a typical edit would be, or that the larger models from the expert dataset have less of their surface area (proportionally) modified in an update.

In summary, the local complexity of our training data is similar to that of the expert validation data, but the models we use are smaller overall. We also train over more drastic edits than the expert data. Despite these differences, our validation shows that our model does generalize well to expert data, and does not appear to be overfitting to our generated edits.

C.4 Ablations

In Table C.2 we report exact numbers for our ablations at a comparable false-positive rate to that we chose for our final proposed algorithm. In addition to the ablations in the main paper, we report further experiments regarding our training method and sensitivity to initialization.

Compared to our training method of using random 50% of exact matches, training with random percentages of exact matches performs virtually the same (see row Train w/ Random %).

We have also tried adding 5% (of $\mathcal{B}_u$ entities) random matches to the initialization and found that our method retains 67.0%, 71.1%, and 77.5% of correct label rates for the complete dataset (also see row Init w/ Exact + 5% Random). We emphasize that dependence on partial matches is an important and fundamental aspect of our work. Our insights are that certain matches are easier than others and partial matches are strong signals in identifying additional matches. Additionally, building on partial matches is an asset when incorporating user input, as shown in Figure 14 in the main paper.

The third section of the table reports performance using different percentages of exact matches for the initialization. As noted in the main paper, initializing with only 30% reduces the performance by around 1 percentage points.

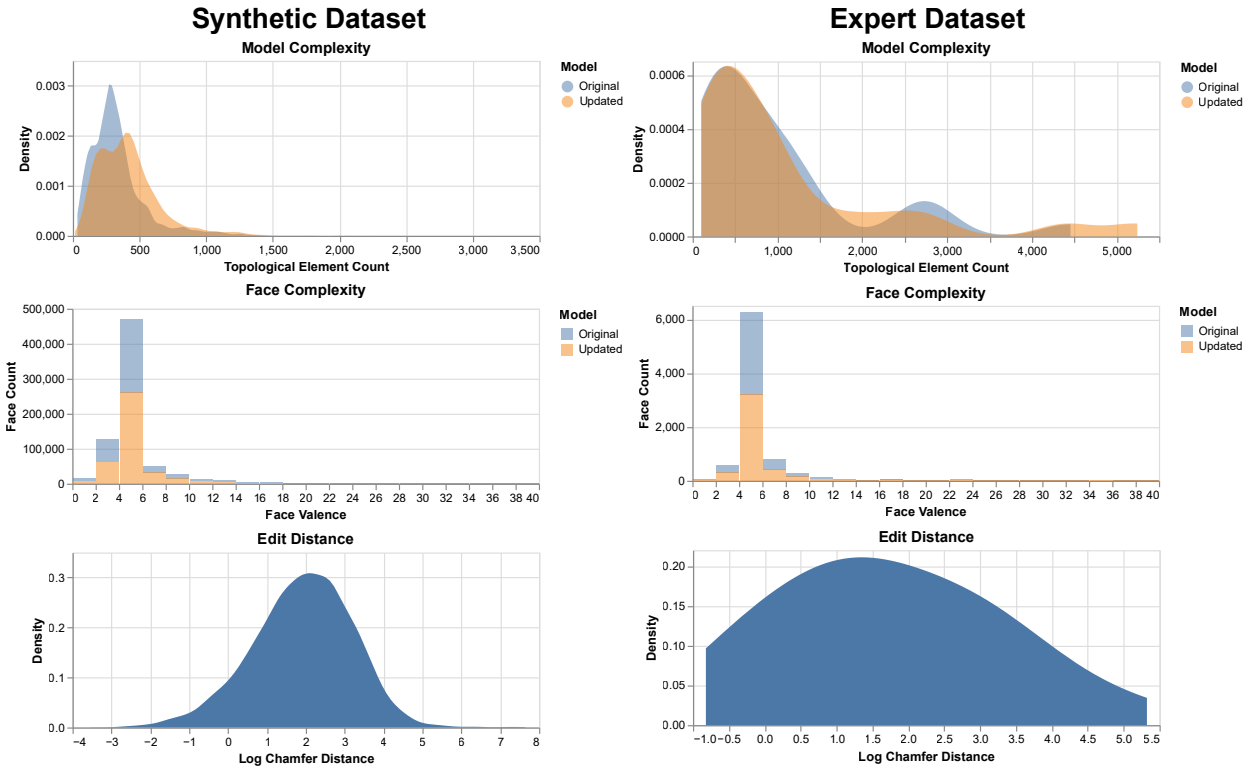


Figure C.2: Dataset metrics; left column synthetic dataset, right column expert dataset. Top: density plot of model complexity in original and updated models. Middle: histogram of face complexities. Synthetic updated models have both more faces, and more complex faces due to constructive edits. Bottom: Density plot distance between original and updated models, measured as the log of the bi-directional chamfer distance between models (4000 points per model, sampled uniformly with respect to surface area.) The expert validation set has a broader distribution of model complexity, but very similar local complexity as measured by face valence. The expert updated models are also more similar to the original than the synthetic models; this could be due to models with more topological elements having a lower proportion of those elements modified in a revision.

Table C.2: Correct label rates of different baselines and variations of our algorithm. Algorithms with threshold are set so that incorrect label rates are comparable to ours (last row). All numbers are percentages of $\mathcal{B}_u$ entities.

	Geometric Only			Topological Only			**Complete**		
	F	E	V	F	E	V	F	E	V
Exact	24.3%	34.4%	47.0%	76.4%	91.1%	99.8%	41.3%	55.0%	68.6%
Exact + Overlap	47.5%	44.6%	47.0%	99.4%	99.1%	99.8%	61.7%	63.8%	68.6%
Adjacency Propagation	75.1%	79.3%	82.5%	97.7%	96.4%	97.4%	73.2%	77.8%	82.2%
CLR Baseline	45.1%	44.0%	52.6%	77.8%	91.4%	99.8%	48.5%	56.7%	69.1%
No Weight	89.0%	88.6%	91.1%	98.9%	99.2%	99.8%	90.2%	91.1%	94.3%
No Preconditioning	76.2%	77.7%	83.3%	94.4%	96.4%	99.8%	80.8%	83.5%	90.9%
No Iteration	83.3%	78.8%	79.7%	98.6%	98.9%	99.8%	82.6%	82.1%	85.7%
Ours + MP on Overlap	90.5%	89.8%	92.6%	99.3%	99.3%	99.8%	91.5%	91.5%	94.8%
Train w/ Random %	90.7%	89.3%	91.9%	98.9%	99.0%	99.8%	91.8%	92.0%	94.7%
Init w/ Exact + 5% Random	65.1%	65.0%	70.0%	82.8%	89.6%	95.3%	67.0%	71.1%	77.5%
Init w/ 0% Exact	73.7%	77.0%	79.4%	32.7%	37.2%	51.1%	83.4%	84.9%	89.2%
Init w/ 10% Exact	87.5%	87.3%	89.6%	60.9%	62.8%	99.5%	89.1%	90.3%	93.7%
Init w/ 20% Exact	88.7%	88.2%	90.7%	79.2%	83.4%	99.7%	90.0%	90.9%	94.0%
Init w/ 30% Exact	89.4%	88.8%	91.2%	98.2%	98.4%	99.7%	90.4%	91.3%	94.2%
Init w/ 40% Exact	90.0%	89.2%	91.4%	98.4%	98.5%	99.7%	90.7%	91.6%	94.6%
Init w/ 50% Exact	90.1%	89.3%	91.6%	98.6%	98.9%	99.8%	91.2%	91.9%	94.5%
Init w/ 60% Exact	90.2%	89.4%	91.6%	98.7%	99.1%	99.8%	91.1%	91.8%	94.7%
Init w/ 70% Exact	90.5%	89.6%	91.7%	98.7%	99.2%	99.8%	91.4%	92.1%	94.9%
Init w/ 80% Exact	90.6%	89.7%	91.8%	98.8%	99.1%	99.8%	91.4%	92.0%	94.9%
Init w/ 90% Exact	90.6%	89.7%	91.9%	98.9%	99.2%	99.8%	91.5%	92.1%	95.0%
Ours	90.6%	89.7%	91.8%	98.9%	99.2%	99.8%	91.5%	92.1%	94.9%
Ours Incorrect Label	3.5%	2.9%	2.1%	0.2%	0.1%	0.2%	2.8%	1.9%	1.1%

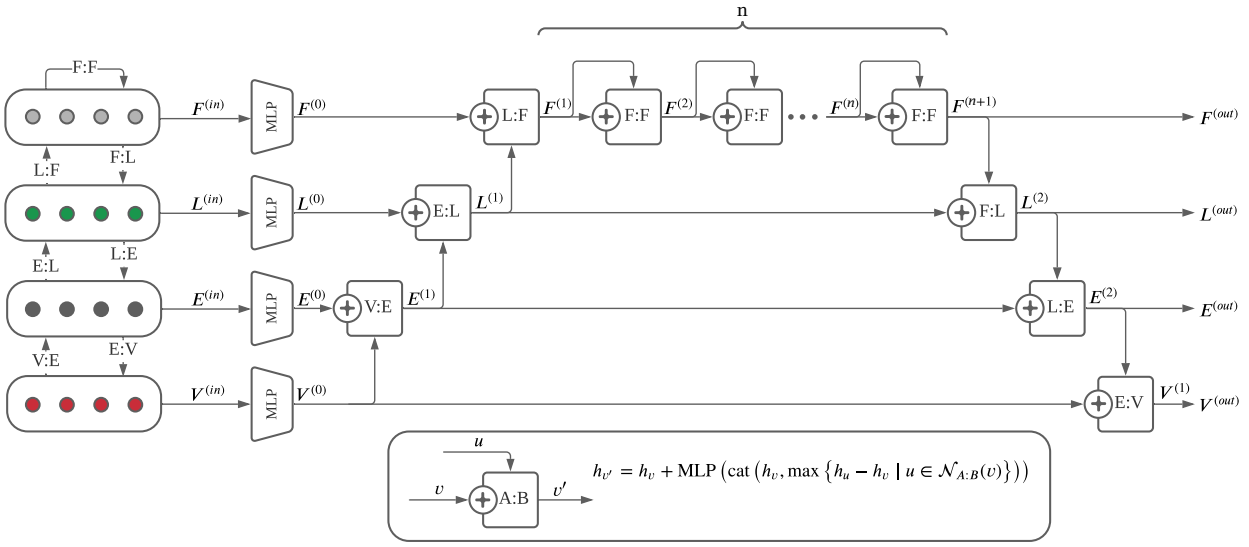


Figure C.3: SB-GCN network architecture. Image from [JHC+21a], used with permission of the authors.

Appendix D

Self-Supervised Representation Learning

D.1 Representation Learning Architecture

Input Encodings. Each topological entity is initially encoded as a vector encoding its parametric geometry and that geometry's relationship to the topological entity. The three dimensions of entity; face, edge, and vertex, each have their own unique encoding of their own size, illustrated in Figure D.1.

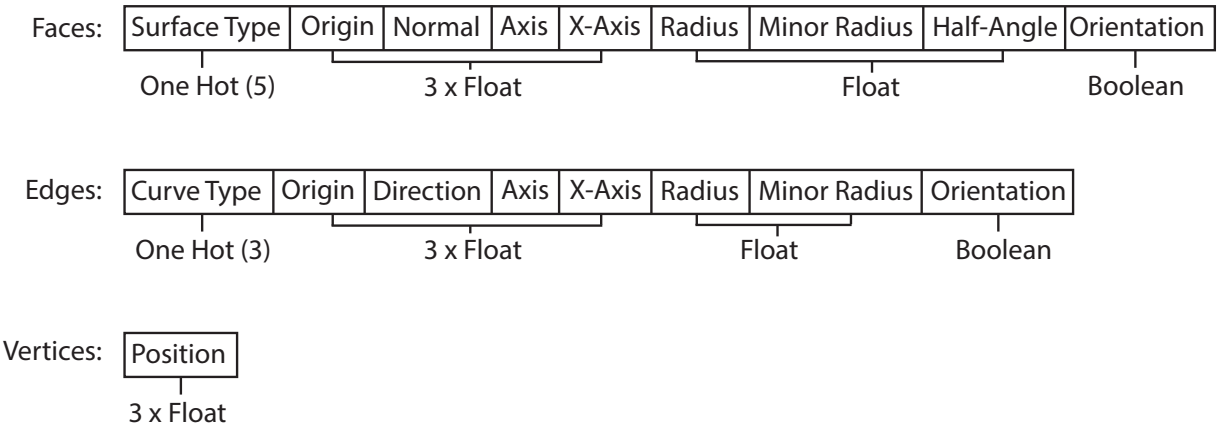


Figure D.1: Input encodings for the 3 different topology types. Not all surface and curve types use all the available parameters; unused parameters are encoded as zeros.

We want a common, fixed size for each level of topology input vector, so we limit our input to B-Reps that have geometry with a fixed number of parameters: planes, cylinders, cones, spheres, and tori for surfaces, and lines, circles, and ellipses for edges. We validated this choice by filtering the parts in the Fusion 360 Segmentation and Automate datasets [LWJ+21, JHC+21a] and found that 77% and 72%, respectively, contained only these primitives.

Surfaces types use the following parameters:

- **Plane**: Origin, Normal, X-Axis
- **Cylinder**: Origin, Axis, X-Axis, Radius
- **Cone**: Origin, Axis, X-Axis, Radius, Half-Angle
- **Sphere**: Origin, Axis, X-Axis, Radius
- **Torus**: Origin, Axis, X-Axis, Radius, Minor Radius

Orientation is common to all faces, and is a boolean value indicating if the surface normal is parallel or antiparallel to the face normal.

Curve types use the following parameters:

- **Line**: Origin, Direction, X-Axis
- **Circle**: Origin, Axis, X-Axis, Radius
- **Ellipse**: Origin, Axis, X-Axis, Radius, Minor Radius

Orientation is common to all edges, and is a boolean value indicating if the curve parameterization is the same as, or reversed from, the edge's. This is important because curve orientation is necessary to determine the inside, versus outside, of bounded faces.

We re-parameterize plane, cylinder, and line geometries so that the origin coordinate is as close to the coordinate system origin as possible. This is done because CAD programs often choose origins far from the actual geometric position. Prior work has dealt with this by omitting these parameters [JHC+21a].

Face, edge, and vertex input encodings are collected into 3 input matrices; $F^{(0)}$, $E^{(0)}$, and $V^{(0)}$ for use in an adjacency list representation of the B-Rep graph. Two adjacency lists are used to represent the multi-partite graph; $\overrightarrow{VE}$ mapping vertices to the edges they bound, and $\overrightarrow{EF}$, mapping edges to faces they bound. Edge graph adjacency has an associated boolean *orientation* feature, $O(e)$, $e \in \overrightarrow{VE} \cup \overrightarrow{EF}$. For $\overrightarrow{VE}$ this indicates if a vertex is a start or end point of the associated edge, and for $\overrightarrow{EF}$ it represents if the inside of the associated face is on the left or right of the edge, relative to the edge's parameterization direction.

Encoder. We structure our encoder as a hierarchical message passing network, inspired by the upwards pass of SB-GCN, but using graph attention message passing with edge features (adapted from [YYL20a] and implemented in PyG [FL19a]), allowing us to incorporate the adjacency features, as well as omit the input MLPs used by SB-GCN. The full encoder architecture is shown on the left half of Figure D.2.

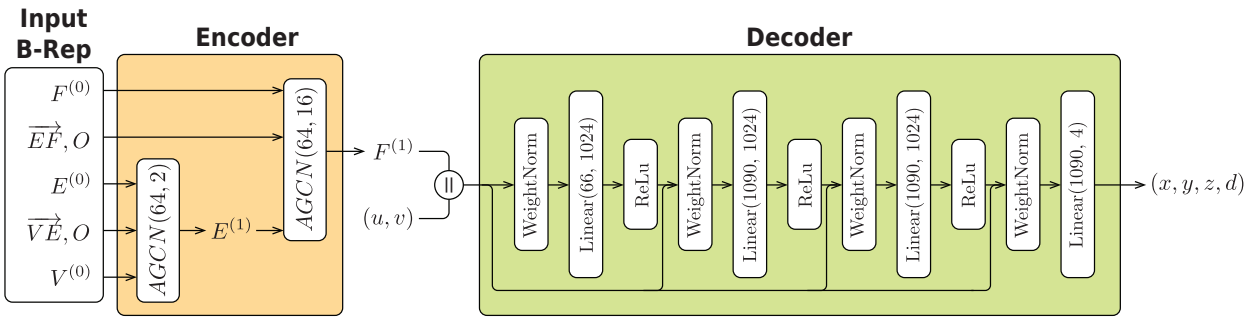


Figure D.2: Our geometric self-supervision encoder and decoder architecture. || denotes concatenation, and $AGCN(S, H)$ is a graph message passing layer with embedding size S and H-headed attention. We use $H = 2$ attention heads for $\overrightarrow{VE}$ since edges have at most two vertices, and $H = 16$ for $\overrightarrow{EF}$ because we observed greater performance with many attention heads, and 16 was an empirically determined balance between model size and accuracy.

Decoder. Our decoder is modeling a function that maps (u, v) coordinates to (x, y, z) positions plus a clipping mask SDF d conditioned on a face code $f \in F^{(1)}$. We parameterize this as a 4-layer, fully connected ReLu network similar to that of DeepSDF [PFS+19], as shown on the right side of Figure D.2.

D.2 Few-Shot Learning Architectures

Segmentation Our segmentation architecture takes as input the learned face embeddings $F^{(1)}$ from our representation learning, and the B-Rep face-to-face adjacencies $\overrightarrow{FF}$, and consists of two Residual MR-GCN graph message passing layers [LMTG19b], followed by a 2 layer LeakyReLu MLP, pictured in Figure D.3.

Classification Our classification architecture takes as input the learned faces embeddings $F^{(1)}$ and consists of two linear layers separated by a max-pool and LeakyReLu, pictured in Figure D.4.

D.3 Biased SDF Sampling

To facilitate training of the neural implicit part of our geometric self supervision, we want to preferentially sample each face's parameter space near clipping mask boundaries, and to approximate the distance to boundaries. Neither of these operations are supported natively by CAD kernels (OpenCascade in our case [ope]). They do, however, support efficiently querying if a point in a surface parameterization is inside or outside the clipping function. Therefore, we approximate the SDF by sampling many uv-points (5000) and matching each inside and outside

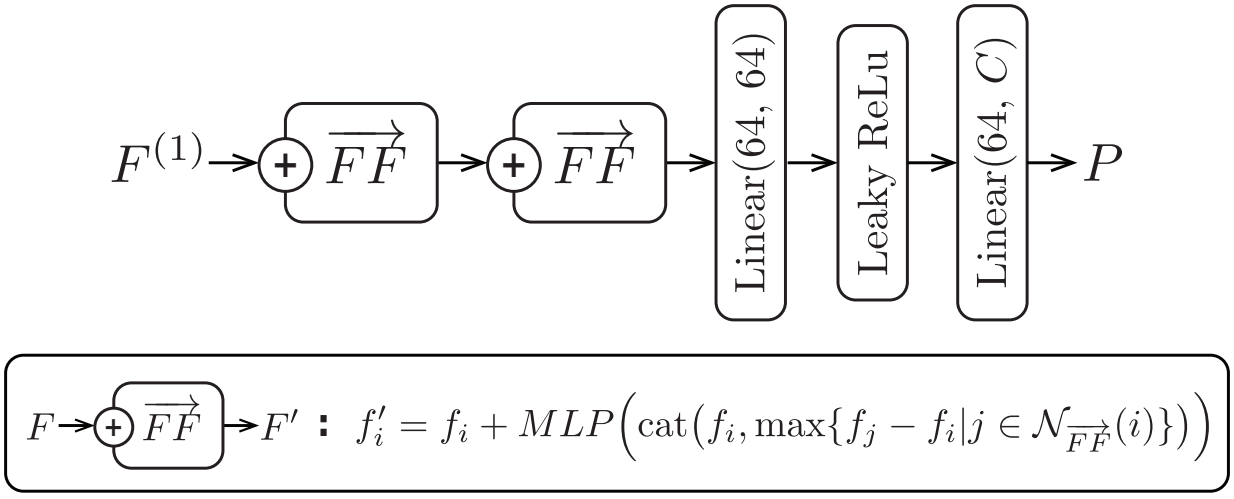


Figure D.3: Few-Shot segmentation architecture. C denotes the number of classes, and P the output logits. In the inset, f_i and f_j denote rows of the feature matrix F, and $\mathcal{N}_{\overrightarrow{FF}}(i)$ denotes the neighbors of node i in adjacency list $\overrightarrow{FF}$.

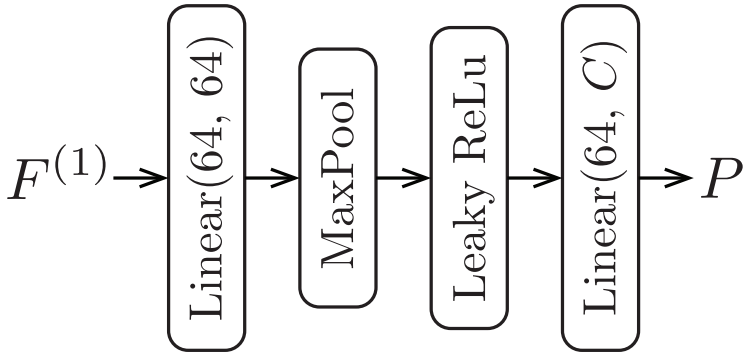


Figure D.4: Few-Shot classification architecture. C denotes the number of classes, and P the output logits.

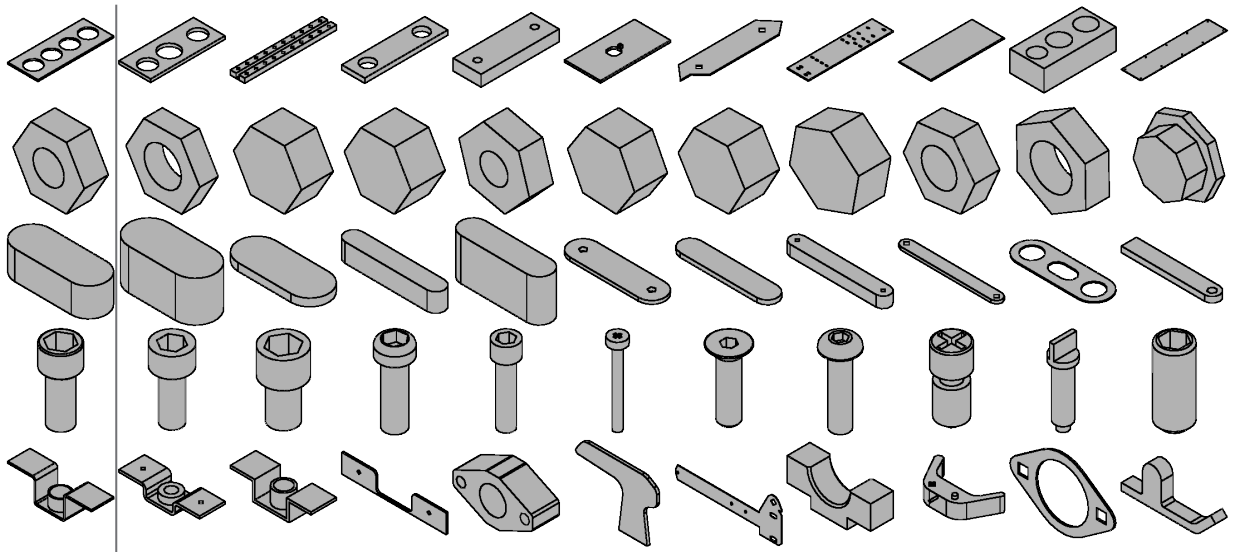


Figure D.5: Unsupervised part retrieval on Fusion 360 Segmentation Test set. Query on left, 10 nearest neighbors, sorted from closest to furthest on right.

point to its nearest neighbor in the opposite set using a KD-Tree to accelerate these queries. We keep $N = 500$ of these points on each face for training. To bias our sample towards the 0-level set, as is common when training implicit SDFs [PFS+19], we sort by $|d|$, task 40% of our sample to be the points nearest to the boundary, and randomly sample the rest.

D.4 Unsupervised Part Retrieval

Although our embeddings are computed per face rather than per-part, we have found that the face embeddings are still useful for part-level retrieval. We max-pooled face embeddings across each part to form a part embedding and used these to find nearest neighbors with the Fusion 360 Segmentation Test set. Figure D.5 shows the 10 nearest neighbors for a diverse collection of part queries, showing qualitatively that face-level embeddings can be used to search for similar parts.

D.5 Ablations

Self-Supervision Ablations. In addition to the network described in Section 3 of the main paper, we also tried using a truncated SB-GCN (only its upwards pass) as the encoder network, using the same shape parameter input features. We evaluate both explicit surface and SDF accuracy in Table D.1 and see that our architecture significantly outperforms SB-GCN for both measures.

Model	XYZ Error	SDF Error
Ours	**0.0256**	**0.0147**
SB-GCN	0.035	0.0214

Table D.1: Self-Supervision ablations. XYZ Error is the average pointwise distance between predicted and sampled surface position. SDF Error is the average absolute difference between predicted and actual SDF value.

Task / Model		Training Set Size					
Fusion 360 Segmentation	Accuracy@:	10	100	1000	10000	20000	23266
Self-Supervision + SVM		**0.66**	0.79	0.85	0.87	0.87	0.87
Self-Supervision + MLP		0.65	**0.80**	0.90	0.94	0.94	0.94
Self-Supervision + MP		0.65	0.79	**0.91**	**0.95**	**0.96**	**0.96**
MFCAD	Accuracy@:	10	100	1000	10000	13940	--
Self-Supervision + SVM		**0.40**	0.51	0.56	0.57	0.57	
Self-Supervision + MLP		0.36	0.60	0.86	0.93	0.93	
Self-Supervision + MP		0.35	**0.66**	**0.96**	**0.99**	**0.99**	

Table D.2: Segmentation ablations. Reported face classification accuracy shows the mean of 10 runs at each dataset size with the train set subset at different random seeds (each model sees the same 10 random subsets). Models were selected by best validation loss on a random 20% validation split, except for the SVM models. Bold indicates the best accuracy at each train size for each task.

Segmentation Ablations. We tried three types of face-level prediction network using our self-supervised face embeddings to determine which was best. The first two try to directly classify faces from the embedding, one using a linear support vector machine (SVM) and the other using a multi-layer perceptron. These test linear and non-linear decision boundaries based on face-codes alone. The third is the message passing scheme described in Section 4, which tests if neighborhood context is necessary.

To evaluate how well each method worked across labeled dataset sizes, we trained each model repeatedly on all supported tasks at a variety of training set sizes using a scheme similar to our baseline comparisons. Table D.2 records the average face segmentation accuracy across dataset sizes for both segmentation tasks. In most cases a non-linear decision boundary is more accurate than a linear one, and neighborhood information improves accuracy leading us to choose the message passing network as our method. For the Fusion 360 Segmentation task the simpler architectures performed very slightly better than the message passing network at low training set sizes, and for MFCAD SVM performed significantly better. We hypothesize that this is due to a combination of SVMs known ability to generalize well with few examples.

Task / Model		Training Set Fraction						
FabWave	Accuracy@:	1%	5%	10%	20%	50%	75%	100%
Self-Supervision + Pooling		.895	**.989**	.994	**.997**	**1.00**	**1.00**	**1.00**
Self-Supervision + MP + Pooling		**0.899**	**0.989**	**0.994**	0.997	**1.00**	**1.00**	**1.00**

Table D.3: Classification ablations. Reported accuracy shows mean of 10 runs, similar to Table D.2. Adding message passing prior to pooling does not confer an advantage, so we do not use it in our reported results.

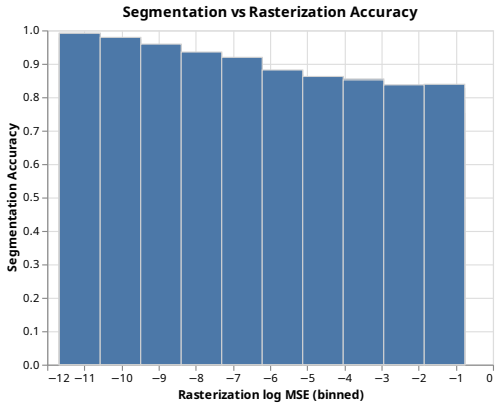


Figure D.6: Segmentation accuracy compared to rasterization accuracy on the Fusion 360 segmentation task. The X-axis is the log of the per-face MSE of our rasterization, binned into 10 groups; the Y-axis is the fraction of faces segmented accurately within each bin. Data is aggregated across all segmentation training set sizes and seeds (the trend is similar across all sizes, with lower accuracies at lower training set sizes).

Classification Ablations. We also tested adding message passing layers prior to pooling for the classification network. As Table D.3 shows, this additional complexity did not yield any improvement.

D.6 Effect of Rasterization Accuracy

Segmentation accuracy is correlated with the accuracy of our model's rasterization. Figure D.6 shows how the likelihood of correctly classifying a face decreases with the reconstruction accuracy of that face. While this figure aggregates data from all training set sizes, the trend is the same across each of them, merely with different slopes. This suggests that downstream task performance could be substantially improved by improving the rasterization learning accuracy, allowing us to learn better representations of complex faces.

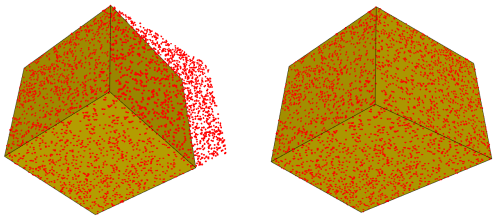


Figure D.7: Shape matching with differentiable rasterization. The input cube (left) was optimized using SGD over its B-Rep shape parameters (right) to match a target point cloud (red).

D.7 Additional Examples

Here we present additional results on randomly sampled parts in rasterization and segmentation. Figure D.8 shows reconstruction results on random parts from the Fusion 360 Gallery Segmentation test set. Figure D.9 shows random part comparisons against baselines for construction based segmentation on the same dataset. Figure D.10 shows additional manufacturing based segmentation results on random parts sampled from the MFCAD test set.

D.8 Gradient Based Optimization

Operating our self-supervision network as a rasterizer creates, in effect, a differentiable CAD renderer, we can use it for gradient-based optimization of B-Rep *shape parameters*. To demonstrate the potential of such an application, we prototyped a shape matching application, where an input B-Rep is optimized via stochastic gradient descent to match a target point cloud. Figure D.7 shows the results of using this optimization to angle the face of a cube. We find that this technique struggles on more complex shapes, which may overcome by improved rasterization performance.

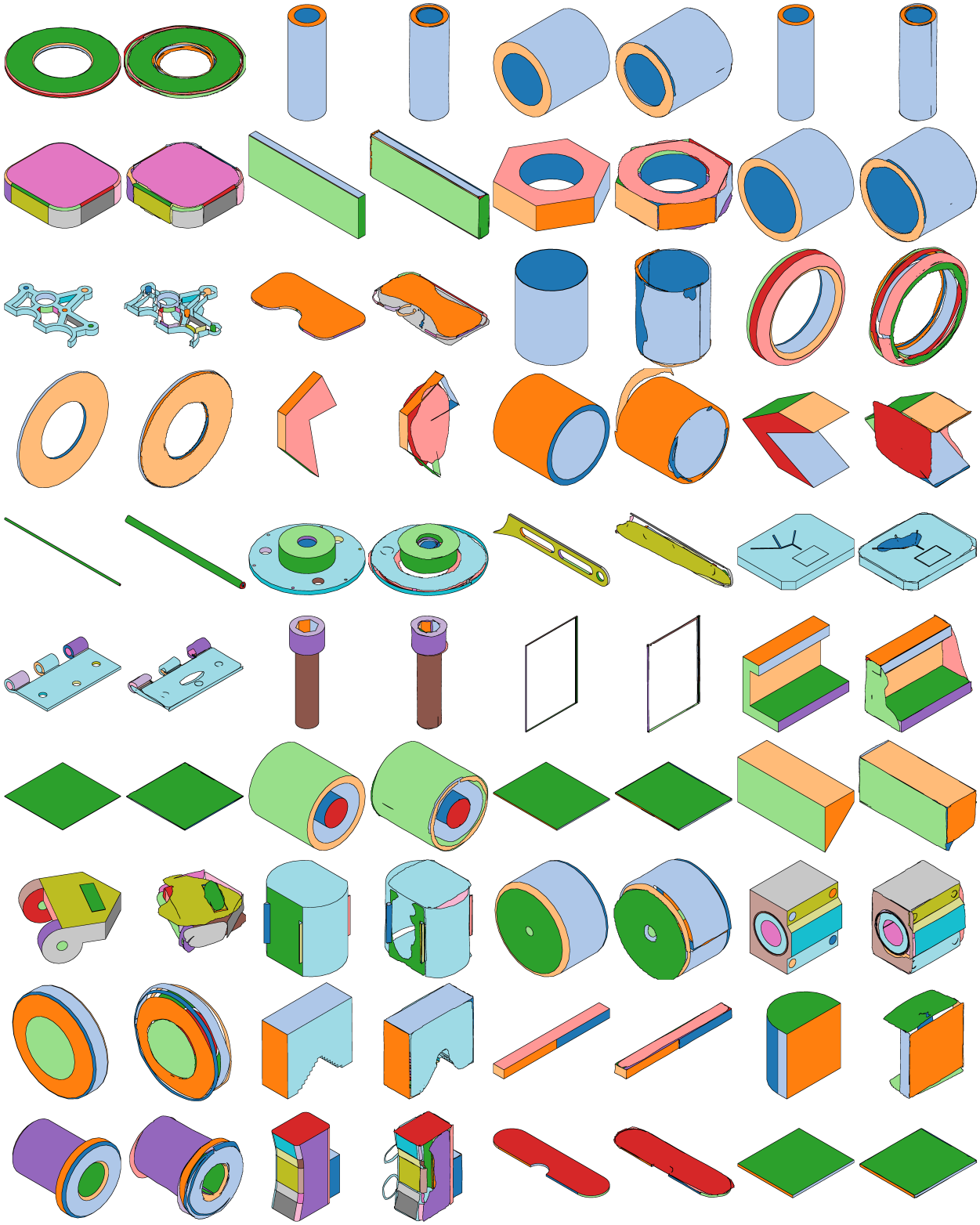


Figure D.8: Additional reconstruction examples sampled randomly from the Fusion 360 Gallery Segmentation test set.

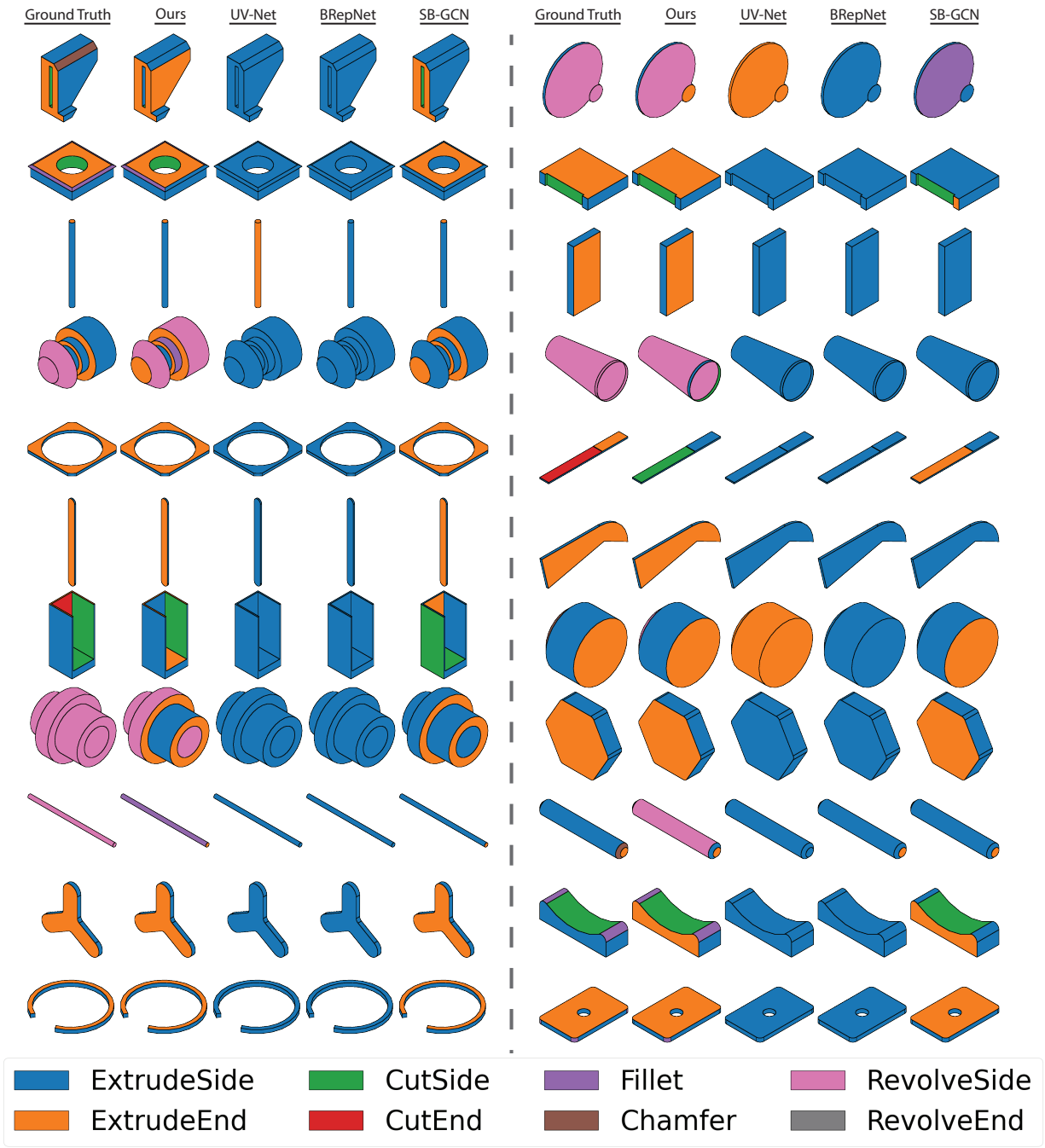


Figure D.9: Additional construction based segmentation examples sampled randomly from the Fusion 360 Gallery Segmentation test set.

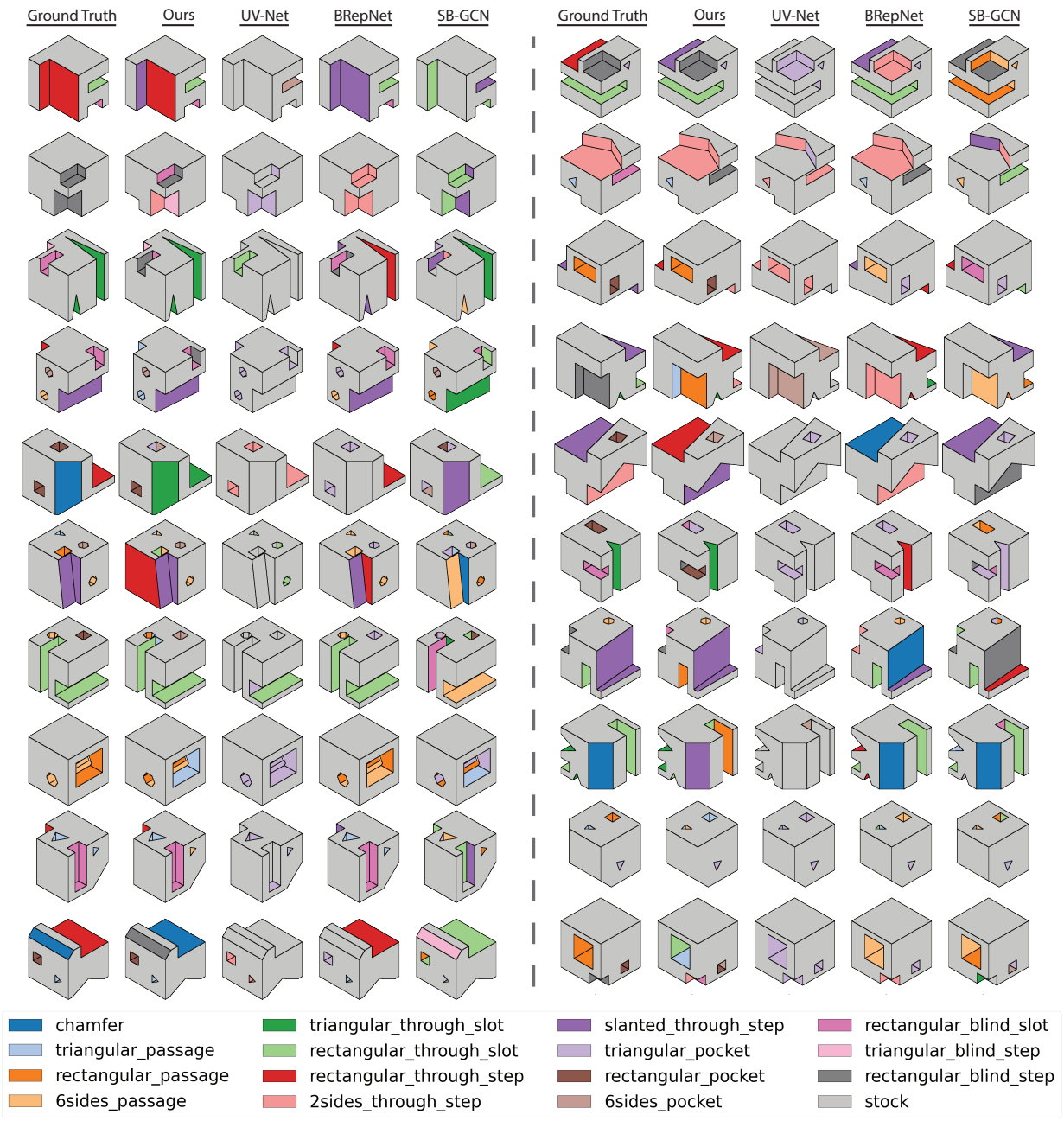


Figure D.10: Additional manufacturing based segmentation examples sampled randomly from the MFCAD test set.

Appendix E

ReparamCAD

E.1 Implementation Details

E.1.1 Parameters for our CSG

We implemented three primitives: cubes, cylinders (for each 3 axes), and cylinders with changing top radius. Each primitive has an associated translation (t_x, t_y, t_z) and scale (s_x, s_y, s_z) parameters. To convert a primitive to a mesh, we start from a base mesh of that primitive in a origin-centered, unit cube from which we apply the scale and translation transformation. For cylinder with changing top radius, we expose another parameter r_{top} which controls the top radius of the base cylinder (before scaling and translation).

E.1.2 Algorithms for Imposing Constraint

As described in the main paper, we want to project parameters to a constrained space that will maximize the IoU between the constrained and unconstrained 3D shape. Since our simplified CSG programs allow overlapping primitives, we would require a differentiable mesh boolean algorithm which is an open research problems. We propose two algorithms. The first one makes a simplifying assumption that primitives cannot change in shape (a cylinder cannot change its top radius). It is based on linear transformation and is fast to compute. This assumption does not hold for the bottle which have a changing top radius due to it not being linear in the way we parameterize it, so we propose another algorithm that optimizes image-space loss via a differentiable renderer.

Algorithm 1

Given a CAD parameters $\mathbf{x_0} \in \mathbb{R}^d$ and a constraint matrix $C \in \mathbb{R}^{m \times d}$, where m is the number of constraints and d is the dimensionality of the input CAD model, we will describe a way to project x to the constraint manifold that will approximately maximizes the IoU score. Since we assume

the base geometry of a primitive cannot change, we can use its canonical bounding cube as a proxy.

We observe that to maximize IoU, we need faces to stay at the same location as possible and we want to move faces that would cause minimal volume changes if necessary. We formulate the cost of shifting a face from its unconstrained position in its normal direction as the squares of the volume change generated by the shift. For cubes, the amount of volume change would be equal to the area of the face multiply by the amount of shifting. For cylinder, we additionally multiply that by the volume ratio (i.e. $\pi 0.5^2 = \pi/4$).

Let $Q \in \mathbb{R}^{6P \times d}$, where P is the number of primitives in the model, be a mapping from parameters to reduced coordinates of each face (a face on an xy-plane is represented by its z coordinate, etc.); and let $\mathbf{a} \in \mathbb{R}^{6P}$ be the volume change associated with each face. We want to minimize $\|AQ\mathbf{x} - AQ\mathbf{x}_0\|^2$ where $A := diag(\mathbf{a})$.

To minimize this on a constrained manifold, we first find the null space $N \in \mathbb{R}^{d \times d'}$ of the constraints matrix C where d' is the nullity of C. We can now formulate the optimization as:

$$y^* := \underset{y}{\operatorname{argmin}} \|AQNy - AQx\|^2$$

where $y \in \mathbb{R}^{d'}$ is the reduced coordinates. Our projected parameters is $x^* := Ny^*$.

Algorithm 2

We optimize the parameters based on the image difference loss. We first uniformly sample the camera poses and have that fixed for all subsequent runs. We optimize:

$$y* := \underset{y}{\operatorname{argmin}} \sum_{\alpha} \|R_\alpha(Ny) - I_\alpha\|^2$$

where α is a camera pose, $R_\alpha(\cdot)$ is the differentiable renderer, and I_α is the image we optimize towards. This can be images from diffusion or the initial renderings (i.e., $R_\alpha(\mathbf{x}_0)$).

E.2 ChatGPT Transcript

Below are the output from ChatGPT by asking "What are different types of [object]?". Selected prompts are highligted in bold.

E.2.1 Chair

There are numerous types of chairs available, each designed with specific purposes, functions, and styles in mind. Here are some common types of chairs:

- **Armchair**: A large, comfortable chair with armrests and often upholstered for relaxation and lounging.
- Dining chair: A chair designed for use at a dining table, typically with a straight back and no armrests.
- Office chair: A chair specifically designed for use in an office or workspace, typically featuring adjustable height, backrest, and armrests.
- Accent chair: A decorative chair that adds visual interest to a room. It may have unique patterns, colors, or shapes to stand out.
- Recliner chair: A comfortable chair with a reclining backrest and often a footrest, allowing the user to recline and relax in various positions.
- **Rocking chair**: A chair mounted on curved legs or rockers, allowing the user to sway back and forth.
- Lounge chair: A long, low chair designed for relaxation, often found in living rooms, bedrooms, or outdoor spaces.
- Club chair: A plush, upholstered chair with a deep seat and high backrest, commonly associated with a traditional or vintage style.
- Folding chair: A portable chair that can be folded for easy storage or transportation, often used for outdoor events or additional seating.
- Bar **stool**: A tall chair with no backrest or armrests, typically used at bars, kitchen counters, or high tables.
- **Chaise lounge**: A long chair with an extended seat to support the legs, allowing the user to recline or lie down.
- **Wingback chair**: A chair with "wings" on the backrest, originally designed to shield the user from drafts, often associated with a classic or traditional style.
- Bean bag chair: A chair filled with small foam or beads, providing a casual and relaxed seating option.
- Director's chair: A folding chair with a wooden frame and canvas seat and back, commonly used by directors and artists.
- Egg chair: A distinctive chair with a rounded shape that envelops the user, providing a cozy and private seating experience.

These are just a few examples, and there are many more unique chair designs available, catering to different needs, preferences, and interior styles.

E.2.2 Table

Tables can come in various forms, designed for different purposes and contexts. Here are some commonly recognized types of tables:

- **Dining Table**: Used for meals and typically found in dining rooms or kitchens. They come in various shapes such as rectangular, round, or square.
- **Coffee Table**: Usually placed in living rooms, it is a low table used for serving beverages, holding magazines, and displaying decorative items.
- **Side Table**: Also known as an end table, it is smaller in size and often placed beside sofas or beds to hold lamps, books, or other items.
- **Console Table**: Typically long and narrow, it is placed against a wall and used for displaying decorative objects, photographs, or as a hallway table.
- Writing Desk: A table designed specifically for writing or working on a laptop. It usually has a flat surface and may include drawers for storage.
- Study Table: Similar to a writing desk, it provides a dedicated space for studying, reading, or doing homework. It often includes storage compartments or shelves.
- Workbench: A sturdy table used in workshops or garages for woodworking, metalworking, or other manual tasks. It usually has a solid top and may include clamps or vices.
- Folding Table: A portable table that can be folded for easy storage or transport. They are often used for events, picnics, or as temporary work surfaces.
- **Picnic Table**: Specifically designed for outdoor use, these tables often have attached benches or seating and are commonly found in parks or outdoor recreational areas.
- Drafting Table: Used by architects, designers, and artists, it has an adjustable top that can be angled for drawing or drafting. It usually has a smooth surface and may include storage for art supplies.
- Pool Table: A specialized table used for playing billiards or pool. It has a flat, cloth-covered surface with pockets, and it is typically larger than other tables.
- Boardroom Table: A large table used in meeting rooms or boardrooms to accommodate a group of people during discussions or presentations. It is often long and rectangular or oval-shaped.

These are just a few examples of the different types of tables available. There are many more variations and styles based on specific functional needs and aesthetic preferences.

E.2.3 Car

There are many different types of cars, each designed for a specific purpose or use case. Here are some of the most common types of cars:

- Sedans: These are four-door cars that typically have a comfortable ride and good fuel economy. They are often used as family cars or for commuting.
- **SUVs**: Sport Utility Vehicles (SUVs) are larger than sedans and often have four-wheel drive, making them suitable for off-road driving. They are popular with families and people who need to transport a lot of cargo.
- Coupes: Coupes are two-door cars with a sporty design. They are typically smaller than sedans and are popular with people who value style and performance.
- **Hatchbacks**: Hatchbacks have a rear door that opens upwards to provide access to the cargo area. They are smaller than SUVs and often have good fuel economy.
- Convertibles: Convertibles have a retractable roof that can be opened or closed. They are popular with people who enjoy the freedom of driving with the wind in their hair.
- **Sports cars**: Sports cars are designed for performance and speed. They typically have a two-seat design and are often expensive.
- Electric cars: Electric cars use electricity to power their motors, rather than gasoline. They are becoming increasingly popular as people seek to reduce their carbon footprint.
- Hybrid cars: Hybrid cars use a combination of gasoline and electricity to power their motors. They are popular with people who want to reduce their fuel consumption and emissions.
- **Pick-up trucks**: Pick-up trucks have an open cargo area at the back and are popular with people who need to transport heavy or bulky items.
- **Vans**: Vans are designed for carrying passengers or cargo. They are often used as family cars, taxis, or delivery vehicles.

E.2.4 Camera

There are various types of cameras available, each designed for specific purposes and catering to different photography and videography needs. Here are some common types of cameras:

- **DSLR** (Digital Single-Lens Reflex) Cameras: DSLRs are versatile cameras that use a mirror and prism system to reflect light from the lens into an optical viewfinder. They offer interchangeable lenses and manual controls, making them popular among professionals and enthusiasts.

- **Mirrorless Cameras**: Mirrorless cameras are similar to DSLRs but lack the mirror and optical viewfinder. Instead, they use electronic viewfinders or the camera's LCD screen to preview the image. Mirrorless cameras are compact, lightweight, and offer excellent image quality.
- **Point-and-Shoot Cameras**: Point-and-shoot cameras, also known as compact cameras, are small, lightweight, and designed for casual photography. They typically have fixed lenses, automatic settings, and are easy to use. Point-and-shoot cameras are suitable for everyday photography and travel.
- Medium Format Cameras: Medium format cameras use larger image sensors compared to DSLRs and mirrorless cameras, resulting in higher image resolution and better dynamic range. They are commonly used in commercial and studio photography where image quality and detail are paramount.
- **Action Cameras**: Action cameras are compact, rugged cameras designed for capturing adventures and high-action activities. They are often mountable and can withstand harsh environments, making them popular for sports, underwater photography, and other outdoor activities.
- **Film Cameras**: Film cameras use photographic film to capture images, which must be developed before viewing. They come in various formats, such as 35mm, medium format, and large format. Although digital cameras have become more prevalent, film cameras are still appreciated by enthusiasts and professionals for their unique aesthetics.
- Instant Cameras: Instant cameras, also known as Polaroid cameras, produce physical prints immediately after taking a photo. They have gained popularity for their nostalgic appeal and the ability to share physical prints instantly.
- Smartphone Cameras: With the widespread use of smartphones, built-in cameras have become incredibly popular. They are convenient, always accessible, and offer impressive image quality, thanks to continuous advancements in smartphone camera technology.
- 360-Degree Cameras: 360-degree cameras capture a complete spherical view of the surroundings. They are useful for virtual reality (VR) content creation, immersive experiences, and capturing panoramic photos and videos.
- Surveillance Cameras: Surveillance cameras, commonly used for security purposes, monitor and record specific areas. They come in various designs, including CCTV (Closed-Circuit Television) cameras, IP cameras, and wireless cameras.
- These are just a few examples of the different types of cameras available. Each camera type has its own unique features and capabilities, catering to different photography styles and requirements.

E.2.5 Bottle

There are numerous types of bottles available, designed for various purposes and materials. Here are some common types of bottles:

- **Water Bottles**: These bottles are typically made of plastic, stainless steel, or glass and are used for carrying and drinking water.
- **Soda Bottles**: These are usually made of plastic and are used for carbonated beverages such as soda or soft drinks.
- Sports Bottles: Sports bottles are designed for athletes and fitness enthusiasts. They often have a sipper or straw for convenient drinking during physical activities.
- Baby Bottles: These bottles are specifically designed for feeding infants. They are typically made of plastic or glass and come with nipples or teats.
- **Perfume Bottles**: Perfume bottles are designed to hold and dispense perfumes or fragrances. They often have intricate designs and are made of glass or crystal.
- **Wine Bottles**: Wine bottles are designed to store and serve wine. They come in various shapes and sizes, typically made of glass.
- Beer Bottles: Beer bottles are used for storing and serving beer. They are commonly made of glass and come in different sizes and shapes, such as the traditional long-neck bottle or the stubby bottle.
- **Medicine Bottles**: These bottles are used for storing and dispensing medications. They are usually made of plastic and may have child-proof caps.
- Glass Bottles: Glass bottles are versatile and can be used for various purposes, including storing beverages, sauces, oils, and other liquids.
- Plastic Bottles: Plastic bottles are widely used for packaging beverages, personal care products, cleaning agents, and more. They come in different shapes and sizes, including single-use disposable bottles and reusable options.
- Stainless Steel Bottles: Stainless steel bottles are durable and often used as an alternative to plastic or glass bottles. They are popular for carrying water, beverages, or hot liquids.
- Oil Bottles: These bottles are designed for storing and pouring cooking oils. They are typically made of glass or plastic and may feature a pour spout or a dispenser cap
- These are just a few examples of the many types of bottles available. The specific type of bottle you choose depends on its intended purpose, material preference, and personal needs.