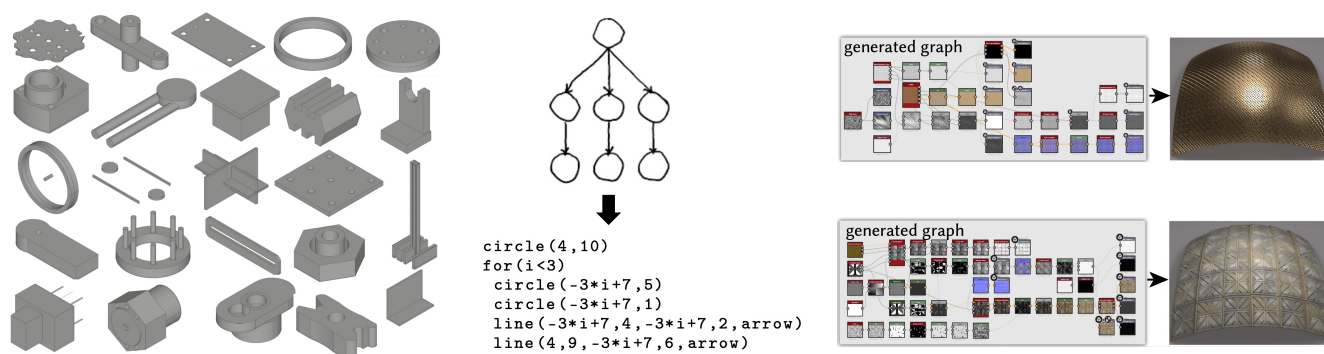


# Neurosymbolic Models for Computer Graphics

Daniel Ritchie<sup>1</sup> Paul Guerrero<sup>2</sup> R. Kenny Jones<sup>1</sup> Niloy J. Mitra<sup>2,3</sup> Adriana Schulz<sup>4</sup> Karl D. D. Willis<sup>5</sup> Jiajun Wu<sup>6</sup>

<sup>1</sup>Brown University <sup>2</sup>Adobe Research <sup>3</sup>University College London <sup>4</sup>University of Washington <sup>5</sup>Autodesk Research <sup>6</sup>Stanford University



**Figure 1:** Neurosymbolic models produce visual data via a combination of symbolic programs and machine learning. From left to right: outputs of CAD programs written by a neural network [XWL\*22]; inferring a 2D drawing program that reproduces an input hand-drawn diagram [ERSLT18]; procedural material programs (i.e. node graphs) generated by a neural network [GHS\*22].

## Abstract

Procedural models (i.e. symbolic programs that output visual data) are a historically-popular method for representing graphics content: vegetation, buildings, textures, etc. They offer many advantages: interpretable design parameters, stochastic variations, high-quality outputs, compact representation, and more. But they also have some limitations, such as the difficulty of authoring a procedural model from scratch. More recently, AI-based methods, and especially neural networks, have become popular for creating graphic content. These techniques allow users to directly specify desired properties of the artifact they want to create (via examples, constraints, or objectives), while a search, optimization, or learning algorithm takes care of the details. However, this ease of use comes at a cost, as it's often hard to interpret or manipulate these representations. In this state-of-the-art report, we summarize research on **neurosymbolic** models in computer graphics: methods that combine the strengths of both AI and symbolic programs to represent, generate, and manipulate visual data. We survey recent work applying these techniques to represent 2D shapes, 3D shapes, and materials & textures. Along the way, we situate each prior work in a unified design space for neurosymbolic models, which helps reveal underexplored areas and opportunities for future research.

## CCS Concepts

• **Computing methodologies** → Shape modeling; Reflectance modeling; Texturing; Neural networks; Computer vision; • **Software and its engineering** → Domain specific languages; Programming by example;

## 1. Introduction

Throughout the history of computer graphics, progress in the field has often been driven by advancements in *representations* of visual data. New representations of visual data have enabled new creative capabilities, or have facilitated existing capabilities more efficiently. For example, texture maps as a representation of surface de-

tail have enabled virtual objects to efficiently mimic the appearance of real-world objects [Cat74]. Level-of-detail systems for polygonal meshes have enabled the rendering of densely-populated virtual scenes at interactive rates [Cla76]. Various types of splines have enabled higher-fidelity modeling of smooth, curved surfaces [dB78]. Implicit surface representations have enabled beautiful scenery

to be encoded in compact algebraic expressions [Ini] and have proven useful for reconstructing 3D geometry from image observations [NZIS13, MST\*20].

One broad class of representation that has been historically popular throughout computer graphics is *procedural models*: symbolic programs that output some visual datum when executed. Procedural models are in some sense as old as computer graphics itself, dating back to the constraint programs used to produce engineering sketches in Ivan Sutherland’s SketchPad system [Edw63]. Nowadays, procedural approaches are widely used for modeling certain classes of 3D shapes, including trees and other vegetation, buildings, and whole cityscapes [IDV, Esr, Sid]. They are also used for modeling object surface appearance, i.e. materials and textures [Adoa], and even the behaviors of virtual characters in a crowd [Mas]. Procedural models have achieved such longevity and popularity because they have many desirable properties. They offer interpretable parameters that can be manipulated to change attributes of the visual data they generate (e.g. the height of a virtual building). Furthermore, randomization of these parameters allows a single procedural model to generate a wide variety of different visual outputs—this is useful for rapidly exploring their design space or for populating large virtual worlds with non-repetitive content. Procedural models do have some limitations, though. First and foremost, creating a procedural model is challenging: doing so typically requires both programming expertise and artistic/design acumen, a combination that only certain practitioners possess. Also, the types of variations achievable by a single procedural model are usually limited to parametric variations; more complex structural variations usually require non-trivial modifications to the structure of the procedural model itself (e.g. changing a model of sports cars to produce trucks).

Recently, machine learning models have become popular for producing visual data, with deep neural networks being especially prevalent. Use of such models has rapidly become widespread for applications in image synthesis, processing, and manipulation [IZZE17, ZPIE17, GEB15, KLA21, RDN\*22, SCS\*22, RBL\*22], 3D shape modeling [WZX\*16, LXC\*17, MGY\*19, JBX\*20, CZ19, LLHF21, HLHF22, ZVW\*22], material and texture modeling [FAW19, DAD\*18, HDMR21], 2D drawing and sketching [HE17, RBCP20, VPB\*22], and more. In principle, these models are easy to create: just provide examples/training data, and a learning algorithm takes care of the rest. What’s more, they are quite general: the same model architecture (and sometimes even the same trained model) can represent a huge variety of different kinds of visual data (e.g. the space of all human faces). They are not without their own limitations, however. The representations these models learn are usually opaque and uninterpretable, making them hard to edit or manipulate (though some researchers have made progress in this space [BZS\*20, BLW\*20]). Additionally, as machine learning methods produce statistical *approximations* of the true function implied by their training data, such models may generate outputs that exhibit artifacts: failing to generalize beyond their training set, producing e.g. blurry images, blobby geometry, etc.

As summarized in Table 1, procedural/symbolic models and learned/neural models have complementary strengths and weaknesses. One might naturally ask: can one get the best of both worlds

Property	Procedural/Symbolic	Learned/Neural
Ease of authoring	Hard	Easy
Interpretability	High	Low
Output artifacts	No	Yes
Output variability	Medium	High

**Table 1:** *Procedural/symbolic models and learned/neural models have complementary strengths and weaknesses for generating and manipulating visual data. This report discusses research that combines them to get the best of both worlds.*

by somehow combining these two types of representations? In this report, we summarize the state-of-the-art for research that focuses on such **neurosymbolic** models which generate visual data using symbolic programs augmented with AI/ML techniques. We first define a design space for such neurosymbolic models, taxonomizing the ways in which neural and symbolic representations can be hybridized to represent visual data (Section 3). We then survey recent work which applies these representations to several different computer graphics domains: 2D shapes, 3D shapes, and materials/textures (Sections 5–7; see Figure 1 for examples). In the process, we situate each prior work within our design space. Finally, we conclude with a look at open problems and future research opportunities in the field, including pointing out potentially fruitful regions of the neurosymbolic design space which have yet to be explored (Section 8).

## 2. Background & Scope

The subject matter of this report may be of interest to many potential readers; we have written it with new graduate students in visual computing (e.g. computer graphics, computer vision) in mind. We assume the reader has a solid background in the fundamentals of computer graphics (including its mathematical prerequisites, e.g. linear algebra). We also assume some familiarity with basic machine learning concepts (e.g. training vs. test datasets, overfitting) and basic neural network concepts (network weights, stochastic gradient descent, etc.) Some familiarity with basic programming language concepts is also helpful (e.g. different programming paradigms, abstract syntax trees).

To keep this report clear and concise, it is intentionally limited in scope. Specifically, we focus on the task of generating visual data using symbolic programs augmented with AI/ML techniques. In this report, when we say that a neurosymbolic model “uses symbolic programs”, we mean that it explicitly constructs an intermediate symbolic representation of the visual data it generates: the visual data is represented by discrete symbols composed into a structure using one or more symbolic operators (i.e. functions with well-defined symbolic implementations). By contrast, visual data may be represented with non-symbolic functions (e.g. a neural network which generates the data) or data structures (e.g. a raster image or a signed distance field).

Research on neurosymbolic modeling is interdisciplinary, drawing on ideas from deep learning, procedural modeling, and program synthesis. Each of these areas would itself merit a full report; in this

section, we discuss which material from each is relevant to our discussion, and which is out of scope.

## 2.1. Deep Learning

Deep learning and neural networks have seen widespread adoption across the field of computer graphics. Neural networks are often used as learned function approximators: given a source domain  $X$  and a target domain  $Y$ , a neural network can find a mapping from  $X$  to  $Y$  using gradient-based optimization techniques [LBH15]. In deep learning, complex neural architectures are developed by composing many simple neural layers and non-linear operations together. These approaches have found success across many computer graphics tasks that can be framed as mapping problems (e.g. feature detection, denoising, rendering, animation, etc.) [MKG\*18]. The questions of how deep learning should be applied, and when it will be successful, are complex and dependent on many problem-specific factors outside the scope of this report.

Neural approaches have proven useful for reconstruction tasks. With enough data and compute, learning-based methods are able to produce visual outputs that correspond to partially observed or under-specified inputs. Typically, such systems learn to directly predict visual outputs in an end-to-end, differentiable fashion (e.g. methods for image-based 3D reconstruction [HLB19]). However, we instead focus on approaches that create (either explicitly or implicitly), a non-trivial symbolic intermediary representation capable of producing visual data. This symbolic representation must be more than a trivial combination of primitives, and as such, we consider neural methods that directly predict low-level representations entities (e.g. triangle meshes [NGEB20]) or even complex geometric representations (e.g. hierarchies of bounding boxes or boundary representations [MGY\*19, SLM\*20, GLP\*22]) to be outside the scope of this report. Relatedly, neural approaches have also been investigated that learn how to represent visual data in a structure-aware fashion [CRW\*20]. While neurosymbolic representations are always structured, not all structure-aware representations fall under our definition of a “symbolic program”. As such, our framework does not capture some approaches that represent visual data as sequences of discrete codes [YLM\*22], or those that simply combine primitives or [TSG\*17] learned parts [PKG\*21]. In later sections, we discuss methods for visual reconstruction tasks that use learning methods to produce symbolic representations that, when executed, generate visual outputs that reconstruct the input [TLS\*19, ENP\*19, ERS\*18, JWR22].

Beyond reconstruction, neural approaches have also proven useful as generative models of computer graphics content. Deep generative models learn to represent the probability distribution over an input domain  $X$  (e.g. a collection of visual data). This probability distribution can then be sampled to synthesize novel instances from  $X$  (e.g. new visual data). There are a multitude of deep generative modeling paradigms [BTLLW22, CHIS22], all with different pros and cons: generative adversarial networks (GANs), variational autoencoders (VAEs), normalizing flows, auto-regressive models, and diffusion models. These deep generative models have been applied across many visual domains such as natural images [KLA21], materials [GSH\*20], sketches [VPB\*22], scenes [WSC18], voxels [WZX\*16], meshes [NGEB20], implicit shapes [CZ19], and

character motion [LZCvdP20]. As in the reconstruction context, many of these approaches have been designed to produce visual outputs directly, without any intermediate symbolic representation, and thus are outside the scope of what our survey covers. Generative neurosymbolic models will often use one of the aforementioned learning paradigms, but will instead train this network to synthesize novel symbolic representations that can be executed to produce new visual outputs.

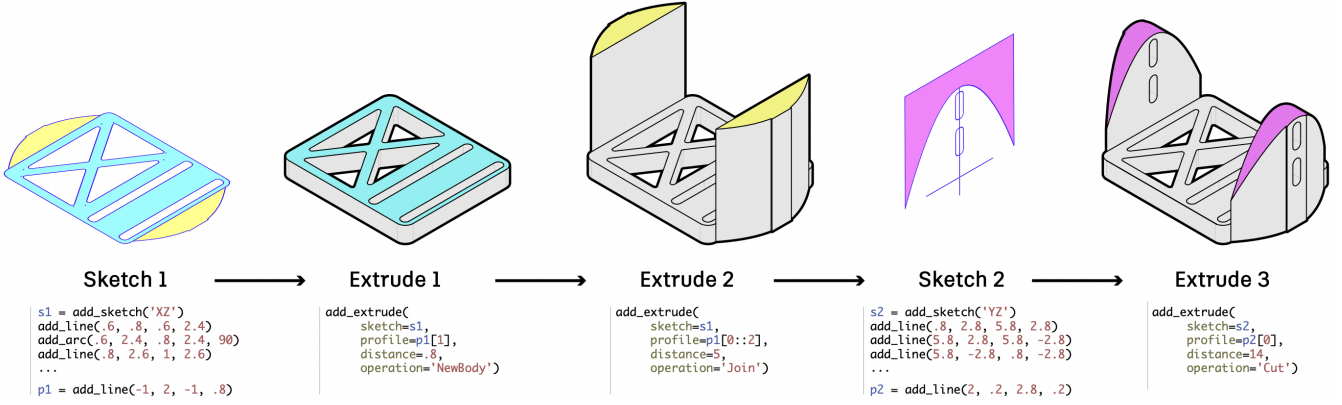
Finally, there has been growing interest centered around approaches that *represent* visual data neurally [XTS\*22]. In some applications, a neural representation will be specialized to a particular datum. For instance, in Neural Radiance Fields (NeRFs), a scene is represented by a neural network that learns a mapping from spatial location and viewing direction input pairs to volume density and emitted radiance output pairs [MST\*20]. Neural representations can also generalize across a distribution of data. For instance, a neural network can learn to map a point and a latent code to an occupancy output prediction. This network can then train on a large collection of shapes and compress each complex geometry into a small latent code [DNJ20]. Most approaches that represent visual data neurally fall outside the scope of our report, as they lack any form of symbolic representation. However, as we discuss in Section 3.2, some neurosymbolic models augment their symbolic languages with learned neural primitives.

## 2.2. Procedural Modeling

Procedural modeling, or the use of symbolic programs to generate visual data, is as old as computer graphics itself. Procedural approaches have been proposed for data at virtually every stage of the graphics pipeline, but they have seen the most widespread use in 3D geometry and texture modeling. As we discuss applications of neurosymbolic models to these domains in Sections 6 and 7, respectively, we review some relevant background material here. A full treatment of the history of these fields is outside the scope of this report.

Note that throughout this report, we use “procedural model” to refer to a program in *any* symbolic programming paradigm which generates visual data. In the programming languages literature, the term “procedural programming” is sometimes used to denote a programming paradigm in which a program is interpreted as a sequence of instructions to be executed; this is contrasted with other programming paradigms, such as functional and declarative programming. In this report, we refer to programs which provide sequences of instructions as “imperative programs,” to avoid overloading the word “procedural.”

**Procedural geometry:** The earliest application of procedural geometric modeling is to computer-aided design (CAD), dating back to Ivan Sutherland’s SketchPad system [Edw63]. A CAD model consists of a set of parameterized operations (possibly including constraints); these parameters can be adjusted and the modeling program re-executed to produce updated geometry. *Constructive solid geometry* (CSG) is one such programming model for 3D geometry generation: sets of parametric primitives which are combined via Boolean set operations (intersection, union, difference) to create more complex shapes [Gha08]. By construction, CSG



**Figure 2:** Procedural modeling as applied to CAD: using feature-based modeling to produce solid geometry as a composition of operations on 2D sketches extruded into boundary representations [WPL\*21].

programs produce watertight surface geometry that bounds a solid object, making it well-suited for modeling objects for manufacturing. *Boundary representations* (or B-reps) are an alternative to CSG for representing solid geometry [Str06]. A B-rep consists of a set of connected surface patches (of any manifold shape) which collectively form the boundary of a solid region. One can perform Boolean operations on B-reps (as with CSG), but they also support additional useful modeling operations, such as filleting or chamfering edges. They are also often authored by lifting 2D engineering sketches into 3D via extrusion, revolution, or other operations (Figure 2). Taken together, this paradigm of solid modeling is typically referred to as *feature-based modeling* [Hof89]; it is now the dominant form of solid modeling adopted by industry-standard CAD software [Das, Auta]. We survey neurosymbolic CAD modeling work in Section 6. Some prior work seeks to infer B-reps from “raw” input geometry such as point clouds [GLP\*22]; such inference systems do not consider the higher-level CAD programs which could produce such B-reps and thus fall outside the scope of this report.

*Context-free grammars* are another popular paradigm for procedural geometry generation. *L-systems* are a particularly prominent example: context-free string rewriting systems whose output strings are interpreted to produce geometry [PHHM96]. Their recursive nature makes them well-suited for modeling naturally-occurring fractal structures such as trees and other types of vegetation [PL96, PJM94]. Other popular instantiations of context-free grammars include *shape grammars*: rewriting systems that operate directly on geometry, typically by subdividing regions of space [SG71]. They are widely used for regular, repeating structures, particularly those occurring in architecture: buildings [MWH\*06], facades [MVG13], and cities [Kel21]. The neurosymbolic modeling techniques we discuss later in this report are applicable to such grammars, because they apply to programs in any language (of which context-free grammars are a subset).

Since trees are common use case for procedural models: there exists prior work on inferring tree models from unstructured inputs (e.g. images or point clouds), much of which uses non-procedural

tree representations [FBA22,LGB\*21,LKK\*21]. Such work is outside the scope of this report.

**Procedural textures:** Procedural techniques have also been popular for creating textures, as they can create intricate details with infinite spatial resolution. The foundation of most procedural texturing systems is a library of primitive patterns, especially *noise* functions such as Perlin noise [Per85] and Worley noise [Wor96]. More complex textures can then be created by systematically composing these texture primitives via various mixing and blending functions. This paradigm, first elucidated in the Shade Trees paper [Coo84], has become widespread in commercial modeling and rendering software, such as Maya’s Hypershade [Autb] and Substance Designer [Adoa]. We survey neurosymbolic modeling work applied to material and texture authoring in Section 7.

### 2.3. Program Synthesis

Program synthesis is an important area of research in programming languages that investigates methods for automatically generating programs that satisfy some high-level specifications. While the dream of automating code generation can be traced back to the birth of computer science, and the first paper that proposed a synthesis algorithm dates from 1957 [BBB\*57], it was not until the last couple of decades that the field saw great advances in what is called *inductive synthesis* [SLTB\*06, ABJ\*13]. In inductive synthesis, users give a (potentially partial) specification to describe the desired *intent*, and search methods are used to explore the space of possible programs generating one that satisfies the specifications. This involves (i) developing methods for specifying user intent, (ii) defining a search space by restricting an existent language or designing novel domain-specific languages, and (iii) developing search algorithms to efficiently explore this space.

The field of program synthesis has advanced by proposing different interfaces for specifying intent, ranging from input-output examples, demonstrations, natural languages, partial programs, and assertions. Search algorithms include enumeration, constraint solving, probabilistic search, and combinations thereof. The first



commercial application of program synthesis was FlashFill in Excel 2013, which derives small programs from data manipulation examples [Gul11]. Today, program synthesis is used in many applications, and frameworks such as Sketch [SLTB\*06] and Rosette [TB13] make it easy to develop synthesis tools for new languages. We refer the reader to [GPS\*17] for an overview of traditional synthesis techniques; we will discuss search algorithms relevant to neural symbolic reasoning in further detail in Section 3.3.

In recent years, program synthesis techniques have also been applied in novel and exciting ways to solve problems in computer graphics. These applications are inspired by procedural representations of shapes which transform modeling into a code generation task. Recent work includes reverse-engineering CAD programs from 3D shapes [DIP\*18, NWP\*18] and shape program manipulation [HLC19].

More recently, work in program rewriting has been used to generate 3D CAD code that is more compact and easy to manipulate [NWA\*20]. They leverage a data structure called E-graphs [NO80]. E-graphs are popular for code optimization, which requires searching over a large number of programs that are syntactically different but semantically equivalent. The key insight is that programs are typically viewed as treelike structures containing smaller sub-programs, many of which are shared across the different semantically equivalent variations. The E-graph data structure is capable of representing many equivalent programs efficiently by sharing sub-programs whenever possible, and further supports operations for extracting programs that have minimal cost. Please refer to [WNW\*21] for a more comprehensive overview. While such program optimization methods live at the boundary of program synthesis and compilers, they are becoming increasingly popular within computer graphics, for example, to optimize design and fabrication plans for carpentry [ZWZ\*21, WZN\*19].

Despite these advances, there are still fundamental challenges in expanding the reasoning capabilities of synthesis techniques to complex domains. This is because (i) the search space grows exponentially with the size of the synthesized code and (ii) because inferring intent from natural forms of human interaction is challenging [GSLT\*18]. Inspired by deep learning's successes over search and inference tasks, the programming languages community has looked for ways to apply these ideas to enable automatic code generation; for instance, synthesizing complex code fragments from natural language specifications [LCC\*22]. We refer the reader to a recent survey on neurosymbolic programming [CEP\*21] for more examples outside the domain of computer graphics.

We also note that some research has explored using neurosymbolic methods for computer graphics applications, but does not *represent* visual data with a neurosymbolic model. For example, recent work uses neurosymbolic representations and search algorithms to discover mosaics algorithms that trade-off performance and output quality [MGA\*22]. This line of work, however, is concerned with finding *transformative* programs (that take input data to output data), instead of *generative* programs (that generate some visual datum) and therefore falls outside the scope of our report.

### 3. A Design Space for Neurosymbolic Models

Thus far, we have described neurosymbolic models in broad terms: symbolic programs augmented with machine learning. There are many possible realizations of this core idea; in this section, we attempt to organize these different instantiations into a formal *design space*. For our purposes, a design space is a set of *design axes* along which the design of a system may vary, coupled with a set of *design choices* for each axis. The design space we present in this section was created to span a large set of neurosymbolic models presented in prior research, which we will discuss later in this report. Given its ability to fit such a broad range of work under its umbrella, we believe that this design space should remain a useful intellectual framework for the field of neurosymbolic modeling as it progresses. In Section 8, we highlight some new, unexplored points in this design space that may merit further investigation.



Figure 3 gives a diagrammatic overview of our design space, organized left to right according to the pipeline of stages followed by typical neurosymbolic models. A neurosymbolic model takes as input some specification of user intent (Section 3.1) and produces programs (Section 3.3) which can then be executed (Section 3.4) to produce visual data which satisfies the user intent. This output visual data might then be further refined by a non-symbolic (e.g. neural) postprocessing step (Section 3.5). To perform this task, the model must also take as input a specification of the language in which its output program should be expressed (Section 3.2); some neurosymbolic models will make changes to this language as part of their operation. Various stages of the model contain neural components which much be learned in some way (Section 3.6).

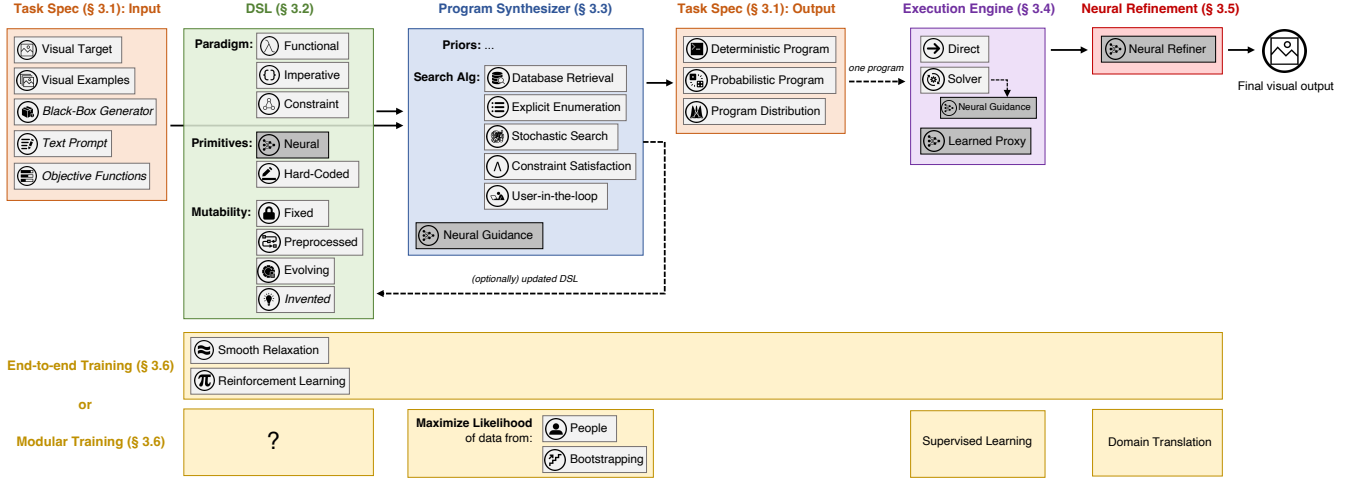
Figure 3 uses specific colors to refer to different neurosymbolic model stages, and it uses specific icons to refer to different design choices. These colors and icons will be used throughout this report to identify where in the design space different prior works fall. Finally, we note that some of the design choices identified here are speculative, in that they are possible design choices but have not yet been executed by any prior work of which we are aware. We identify these design choices with an *italic* font.

#### 3.1. Task Specification

The task that a neurosymbolic model should perform can be specified by its given inputs and the outputs it should produce. In general, a neurosymbolic model will take some user input about the kind of visual data the user would like the model to generate; it will then provide as output one or more programs, which can be executed to generate visual data which satisfies these inputs. The model is also typically given a formal specification for the language in which these output programs should be expressed (see Section 3.2).

There are many ways for a user to specify their intent to a neurosymbolic model. Potential user inputs include (but are not limited to):

-  **Visual Target:** A single visual datum (e.g. an image or a 3D shape) to be reconstructed.
-  **Visual Examples:** a collection of visual data whose distribution should be emulated.



**Figure 3:** The design space of neurosymbolic models. A neurosymbolic model takes as input a specification of user intent and a domain-specific language (DSL), and it synthesizes programs that output visual data that satisfy the user intent. The visual output may then be further refined in a non-symbolic, neural postprocess. Several approaches have been proposed for learning the various neural components of such a model. Note: the icons and color scheme introduced in this figure are used throughout the report to identify where in this design space different prior works fall.

- **Black-Box Generator:** an existing visual data generator whose behavior should be emulated (e.g. a pretrained deep generative model of 3D shapes [LLHF21, WZX\*16, HLHF22]). We are not aware of any existing work which implements this particular design choice.
- **Text Prompt:** a natural-language description of the type of visual data to generate. We are not aware of any existing work which implements this particular design choice.
- **Objective Functions:** functions which map a program’s visual output to a desirability score (e.g. the stability of a generated 3D shape under gravity [MBBO22]). We are not aware of any existing work which implements this particular design choice.

As output, the neurosymbolic model could produce:

- **Deterministic Program:** a single program that returns the same visual output each time it is executed. Such a program could include continuous parameters that can be varied to produce a range of possible visual outputs. Many CAD programs fall into this category [WPL\*21, WXZ21].
- **Probabilistic Program:** a single program that makes random choices, producing a different visual output each time it is executed (e.g. the type and position of wings on a spaceship [RJT18]). As these random choices can affect both continuous and structural elements, these programs can capture a wider variety of visual inputs compared with Deterministic Programs.
- **Program Distribution:** a sampler for a distribution of different programs. Sampled programs could have a range of different structures, allowing this representation to capture an even wider range of visual outputs than either of the two previous types. In the works we will survey in this report, such samplers are usually represented generative neural networks that output programs.

Additionally, the model may produce a new language in which its output programs are expressed (if no such language was provided

as input, or if the input language was modified by the neurosymbolic model).


### 3.2. Domain-Specific Language (DSL)

The heart of a neurosymbolic model is the programs it produces; in order to specify a program, one needs a language in which those programs are to be expressed. Although it is in principle possible to use general-purpose programming languages such as C++ or Python, the standard practice is instead to use a *domain-specific language*, or DSL: a language whose data types and operators are specialized for a particular domain of interest (e.g. generating 3D shapes) [Fow10]. Such languages allow useful programs in that domain to be expressed more concisely, which makes them both easier to use and modify as well as easier for the neurosymbolic model to produce (as the space of possible output programs is smaller).





For the purposes of neurosymbolic modeling, there are several important design axes to consider when specifying a DSL:

**Programming paradigm:** A DSL can adopt any of the many programming language paradigms. The work we survey in this report includes examples of **Functional** languages (e.g. node-based dataflow programs representing procedural materials [GHS\*22]), **Imperative** programs (e.g. scalable vector graphics, SVG, files describe sequence of drawing commands [CDAT20, RGLM21]), and **Constraint** languages (e.g. CAD engineering sketch languages which specify constrained relationships between the geometric attributes of different sketch elements [SZRA22, GBL\*21, PBG\*21]).

**Primitive types:** One of the ways that a symbolic program can be augmented with machine learning is to have learned **Neural Primitives** within the language—either primitive data types or

primitive functions. For example, a language for 3D shapes could include a library of 3D parts, each of which is represented by a learned neural field [DKD\*22]). If a DSL does not have such primitives, we say that it uses  **Hard-coded Primitives** only.

**Mutability:** The language given as input (if any) to a neurosymbolic model isn't necessarily fixed; the model could choose to modify the language, if this allows the model to better explain the visual data it is trying to produce. We classify a language's mutability into one of the following categories:


-  **Fixed:** the language given as input remains unchanged.
-  **Preprocessed:** the language given as input is modified by the neurosymbolic model in a preprocessing step before the model learns to produce programs (e.g. by finding common patterns in some example programs and factoring them out into new procedures [JCG\*21]).
-  **Evolving:** the language given as input is continually modified by the neurosymbolic model as it learns to produce programs (e.g. by discovering increasingly complex subroutines which make it easier to produce more complex programs [EWN\*21]).
-  **Invented:** no language is given as input; the neurosymbolic model invents the entire language itself. The process of “inventing” a language would involve proposing a set of primitive data types and functions which combine those data types to parsimoniously represent visual data from a particular domain. We are not aware of any existing work which implements this particular design choice. We briefly discuss potential future work ideas along this direction in Section 8.





### 3.3. Program Synthesizer

Given a task specification and a domain-specific language (DSL), the neurosymbolic model must produce generative programs that satisfy the specification in that DSL. This problem is fundamentally a search program: searching through the space of programs expressible in the DSL to find satisfying ones. This is a challenging search problem, and there are many possible strategies for solving it. We can organize these strategies along the following design axes:


**Program priors:** In any nontrivially complex DSL, it is likely there will exist multiple programs that satisfy the input specification equally well. To disambiguate between them, we must ask ourselves the question: what kind of programs would we prefer *a priori*? To the extent that this question has been considered by prior work (in both neurosymbolic models for graphics and in the field of program synthesis more generally), the answer has been to prefer shorter/simpler programs (i.e. following Occam's razor). This design axis could admit different/additional priors, however; we will come back to this point in Section 8.

**Search algorithm:** The heart of a program synthesizer is the algorithm it uses to search through the (typically vast) space of possible programs. Prior work has used methods of the following types:

-  **Database Retrieval:** the simplest form of program “synthesis”: rather than generate a new program, retrieve one from a database of existing programs. Some methods will then modify the retrieved program, e.g. by tweaking its parameters to better satisfy user goals.

-  **Explicit Enumeration:** deterministic exploration of possible programs in the language. Enumeration can proceed either in *top-down* or *bottom-up* fashion, i.e. whether the search starts from the root or the leaves of the program's abstract syntax tree. This approach is typically only viable for finding short programs in relatively simple languages; additional techniques are often required to make the search tractable in more complex settings. The most common such approach is to restrict the search space using some heuristic(s). Greedy search is the most extreme form of restriction. Variants of *beam search* are also widely used [Red76]. In top-down search, the *branch and bound* strategy can be applied [Cla03]. A more recent class of techniques involves compressing the search space by compactly representing large classes of equivalent sub-programs; these representations include version spaces [Mit77] and e-graphs [TSTL09].
-  **Stochastic Search:** for languages with very large program spaces where enumeration is not tractable, a randomized form of search can be a viable alternative. These algorithms start with some initial program(s) (which could be random programs from the language or initialized using some heuristic) and then iteratively make random changes to the program(s) in the hopes of improving them. Some of these algorithms maintain a single candidate program: these include Markov Chain Monte Carlo (MCMC) [AFDJ03] and simulated annealing [vLA87]. Other algorithms maintain a set, or *population*, of programs; the random changes the algorithm uses may involve reasoning about multiple programs at once. Examples of such algorithms include Sequential Monte Carlo (SMC) [DDFG01] and genetic programming [Koz92].
-  **Constraint Satisfaction:** in some cases, it is possible to convert the task specification into a set of constraints and treat the problem as one of constraint satisfaction. One common approach is to convert the spec into a Boolean satisfiability problem. If this can be done without intractable blowup in the size of the problem, then efficient SAT solvers can be applied [VWM15]. If such a conversion is not possible, it may be possible to convert the spec into a first-order logic satisfiability problem, which permits the use of SMT solvers [BT18].
-  **User-in-the-loop:** rather than synthesize a program automatically, involve a person to help the synthesizer make decisions about what different parts of the output program should look like (e.g. sparse user scribbles to guide the decomposition of a texture image into multiple procedural components [HHD\*22]).

Finally, it is worth discussing the search algorithm's termination conditions. The enumeration-based and constraint-based algorithms will terminate once they have explored all possible programs; in practice, reaching this termination condition can require an intractable amount of computing time. Furthermore, the stochastic search algorithms have no natural termination condition: they can, in principle, be run forever and may continue to produce better programs. With the above in mind, most program synthesizers treat their search algorithms as *anytime algorithms*, terminating and returning the best program(s) found so far once some fixed compute budget is exceeded.

-  **Neural Guidance:** Many of the search algorithms discussed above have steps that can benefit from some form of heuristic guid-

ance: beam search needs to rank the set of possible next search states to explore; branch-and-bound needs to bound the possible score of part of the search tree before deciding whether to expand it; stochastic search methods can use non-uniform distributions for their random changes. In prior work thus far, this is the most common place for machine learning to come into play: *learning* a search guidance heuristic. This heuristic often takes the form of “suggestions for what to add to the program next.” As such, autoregressive language models are the most widely-used neural guidance architectures, particularly transformers [BMR\*20]. Some past work also uses pointer networks [VFJ15], so that the neural guide can suggest repetitions of prior parts of the program (e.g. variable re-use). Various forms of graph convolution networks [GSR\*17, BHB\*18] have also been applied, though these have largely fallen out of favor since the advent of transformers and other attention-based models. For systems that “synthesize” programs by retrieving them from a database, a learned deep feature space (where similarity search is performed) acts as a form of neural guidance. Section 3.6 discusses how to train these guidance networks.

We note that some models in prior work perform program synthesis via a single forward pass of such a guide network, i.e. there “is no search” being performed. In our categorization of prior work, we label such models as either using (⊖) **Explicit Enumeration** with neural guidance if the network takes the argmax over discrete program choices (i.e. beam search with a beam width of one) or (⊗) **Stochastic Search** with neural guidance if it randomly samples them (i.e. Sequential Monte Carlo with a single particle).

### 3.4. Execution Engine

When the neurosymbolic model has produced a program, its job is not yet finished—the program must then be *executed* to produce a visual output. The process of executing a program can take different forms:

- (→) **Direct**: the program is linearized and its operations executed in sequential order. This type of execution is most common; it corresponds to the execution model used by most (non-parallel) general-purpose programming languages (whether interpreted or compiled).
- (⊖) **Solver**: in some languages, executing the program requires solving a search, optimization, or constraint-satisfaction problem. We think it worthwhile to assign such execution engines a distinct point in the neurosymbolic model design space for two reasons. First, using them can make the neural components of the model harder to learn, if learning requires computing gradients w.r.t. the execution engine (see Section 3.6). Second, just as search during program *synthesis* presents opportunities for neural guidance, so too does search during program *execution*.
- (⊗) **Learned Proxy**: instead of a hardcoded execution engine (either direct or solver-based), one may instead opt to learn a mapping between program text and program output. As with any learned function, such an executor can only be approximately correct; however, it can have advantages for learning the other neural components of a neurosymbolic model (see Section 3.6).

### 3.5. (⊗) Optional Neural Postprocessing

There is one final place in a neurosymbolic model where learning can come into play: applying learned postprocessing to the visual data output by a synthesized program. Such a step can help close the “reality gap” between procedurally-generated data and data which is acquired from the world via sensors (as the former is almost always “cleaner” than the latter).

### 3.6. Learning Algorithm

At this point, we have completed our tour through the pipeline of a neurosymbolic model: from input specification to final visual outputs. However, there is still one critical design axis to discuss; one that touches multiple parts of the neurosymbolic model: how are its learnable components trained?

**End-to-end (Unsupervised) Training:** The most obvious approach is to train all the neural components of the model at once by backpropagating the final task loss function through the entire model. This is desirable because it is conceptually simple and does not require any ground-truth supervision for intermediate steps in the pipeline—most notably, no pairs of (input space, ground-truth program) need be provided. As part of this process, loss gradients must be backpropagated through every choice the synthesizer guide network makes in constructing the program. If those choices represent continuous values in the program, then this backpropagation is well-defined. However, programs also typically exhibit complex structure dictated by discrete choices (e.g. branching decisions, types of primitives to use). Gradients through such discrete choices are not well-defined. There are at least two approaches to work around this conundrum:

- (⊖) **Smooth Relaxation**: defining a continuous relaxation of discrete structural program choices, so that a single program can smoothly blend between different discrete structures. This strategy also requires the design of a program executor which can produce a correspondingly (and semantically meaningful) smoothed output in the program’s output domain. Designing such relaxations (and their executors) is challenging, time-consuming, and domain-specific: the creation of a new relaxation typically warrants a research publication [KZK20, RZC\*21]. Some methods use end-to-end differentiable learning to train only the continuous parts of their model; for simplicity, we also group such methods under this category.
- (⊗) **Reinforcement Learning**: specifically, policy gradient RL with a score function gradient estimator (e.g. REINFORCE [Wi92] or PPO [SWD\*17]). These algorithms stochastically estimate gradients and do not require any part of the model to be differentiable (aside from the neural networks themselves). While unbiased, these gradient estimates can have extremely high variance, resulting in training that converges slowly, to a poor local optimum, or not at all.

It is worth noting that using a solver-based executor in end-to-end training requires running an iterative solver loop within the outer iterative optimization loop of neurosymbolic model training, which will be slow, not to mention likely not (easily) differentiable. Thus, it is usually best to use a learned proxy executor instead, whenever



the approximation error from doing so is acceptable. On a related note: in the end-to-end learning paradigm, a learned proxy executor may be incompatible with having neural primitives in the program (as gradients may never propagate through the actual executor in order to train the neural primitives).

**Modular Training:** The alternative to end-to-end training is to train different stages of the neurosymbolic model separately. This often, but not always, necessitates supervision in the form of ground-truth outputs for the stage of the model being trained. For some stages, this supervision is easier to come by than others (e.g. a “ground-truth program” for a given input spec may be difficult to obtain). In the remaining paragraphs in this section, we describe how modular training can be applied to different stages of the neurosymbolic modeling pipeline from Figure 3:

*Modular training for a program synthesizer guidance network:* If one can produce paired (task spec, program) data, then this network can be trained to maximize the likelihood of that data. There are several ways such data could be obtained/constructed:

- **👤 People:** in the best-case scenario, one has access to a curated set of programs written by people to satisfy a particular task specification. Such data is rare, but examples do exist; see Section 4.
- **🧑‍🔬 Synthetic Programs:** in the absence of “ground-truth” programs created by people, one can instead write a procedure to create a variety of synthetic programs along with task specs that they satisfy. Such a procedure could be as simple as sampling random derivations from the DSL grammar, or it could make use of domain knowledge. Models trained on such data typically do not generalize well to actual task specs of interest, so they are usually only used to provide an initialization for another learning method.
- **🔄 Bootstrapping:** starting from a model pretrained on synthetic data, some algorithms can improve the model by iteratively training on variations of the model’s own predictions [JWR22].

Alternatively, the **🌀 Smooth Relaxation** or **🎯 Reinforcement Learning** approaches can be used here (typically also initialized with maximum likelihood pretraining on synthetic data).

*Modular training for a learned proxy execution engine or a solver guidance network:* Both types of networks can be trained using supervised learning on (program, program output) pairs. As with training a synthesizer guidance network, such data can be obtained by creating and executing synthetic programs; however, networks trained on such data may not generalize to “real” data presented to the model at inference time.

*Modular training for a neural refinement network:* Refining the output of a procedural modeling program can be phrased as a type of *domain translation* problem: transforming the output of the program from the “looks procedurally-generated” appearance domain to the “looks realistic” appearance domain. Since one typically does not have access to paired (procedural, real) data, unpaired domain translation approaches are useful here [ZPIE17, SWB21].

*Modular training for neural language primitives:* We are unaware of any existing work which learns neural language primitives using a modular training paradigm. If one could obtain (input,

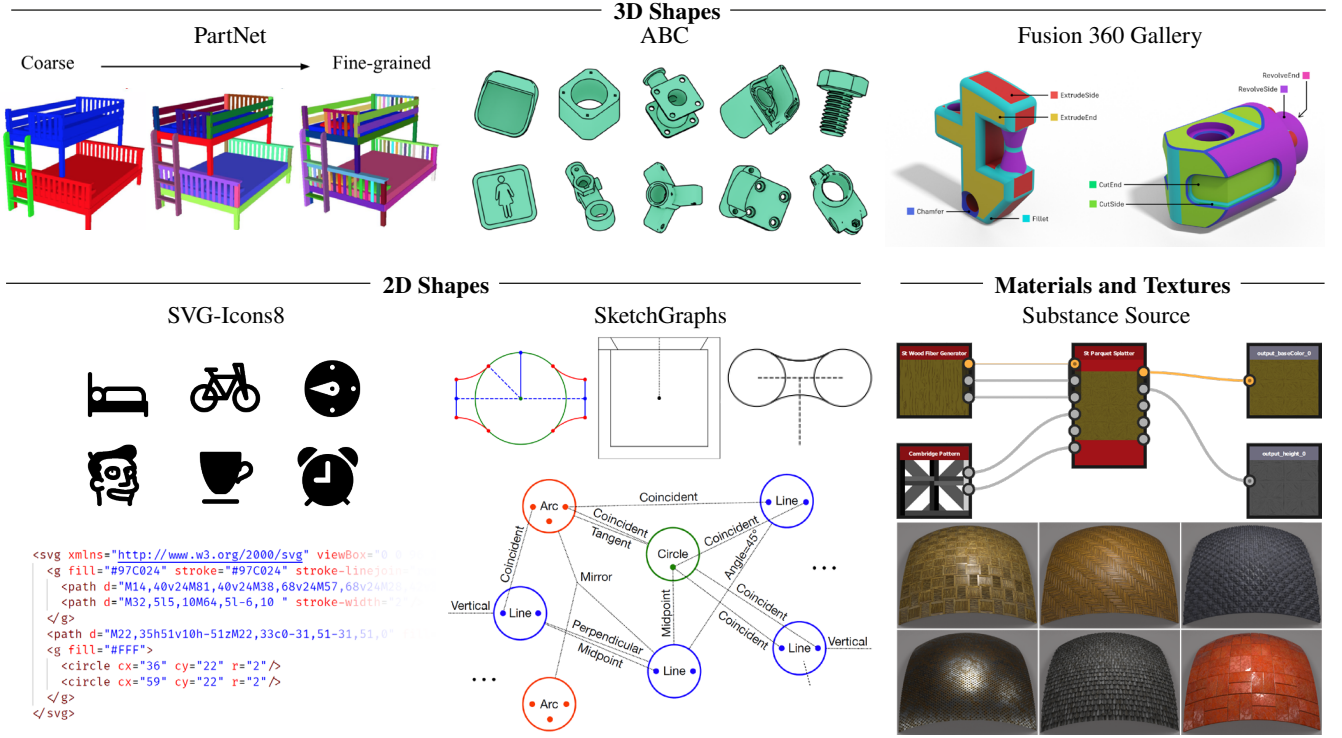
output) example pairs for the part of the program’s execution that a neural primitive should perform, then such a primitive could be trained with supervised learning. It is less clear how to learn neural primitives in the absence of such data (i.e. when only the *entire* program’s desired output is available as a training signal) within a modular training regime; this remains an open problem for future work.

#### 4. Datasets for Neurosymbolic Modeling

Several neurosymbolic methods that we discuss in this paper require some form of supervision. For some of the methods, a traditional dataset such as a set of 3D shapes or 2D images is sufficient. Other methods, however, require more structured data as supervision that is more amenable to neurosymbolic modeling. Here, we give an overview of datasets that provide a more symbolic representation of the data, for example in the form of a program or a high-level structure. Examples of several of these datasets are shown in Figure 4.

**2D Shapes:** Several datasets provide higher-level descriptions of 2D shapes, such as their composition from parametric primitives, or the set of strokes that were used to create a sketch. CAD programs are a possible source for this type of data. They typically provide an interface for authoring 2D engineering sketches. These sketches are represented by parametric primitives, such as circles, lines, and arcs, and spatial relationships between primitives, such as adjacency or tangency. Two datasets provide 2D sketches from CAD programs. *SketchGraphs* [SOZA20] provides 15 million 2D sketches from OnShape [PTC]; however, it is known to contain a significant amount of duplication in the data samples. The *CAD as Language* dataset [GBL\*21] contains roughly 4.5 million sketches, also scraped from OnShape, and tries to address some of these issues. Wong et al. [WMG\*22] introduce two procedurally generated datasets of 2D drawings. The first contains 1k drawings from categories such as furniture, vehicles, and gadgets, each consisting of simple primitives like lines and curves. The second dataset contains 1k sketches of simple 2D buildings such as towers, bridges, and houses composed of blue and red blocks. Several datasets provide high-level information about strokes in sketches, in the form of vectorized strokes and the order in which they were drawn. The *Omniglot* dataset [LST15, LST19] contains 1.6k handwritten characters from 50 different alphabets, each drawn by 20 different people. The *Quick, Draw!* [HE17, Qui] dataset contains 50 million drawings across 345 categories. Scalable Vector Graphics (SVG) is a popular format for vector graphic content and can be seen as a simple programming language to generate parametric shapes. Several datasets provide 2D shapes in this format. DeepSVG [CDAT20] introduced the *SVG-Icons8* dataset, which consists of 100k icons obtained from the Icons8 [Ico] website. The icons depict various categories of objects, such as beds, bicycles, and cups. *SVG-Fonts* [LHES19] provides 14 million vectorized characters in different fonts.

**3D Shapes:** Three datasets provide higher-level structural information for 3D shapes, in addition to meshes. *PartNet* [MZC\*19] is a dataset that provides 26k synthetic 3D shapes across several categories of manufactured objects typically found in interior scenes,



**Figure 4:** Examples of datasets that provide structured data that is more amenable to neurosymbolic models. We show examples of datasets for 2D shapes, 3D shapes, and materials. All datasets provide some form of structural information, such as part decompositions, construction sequences, or geometric relationships between parts.

such as chairs, tables, and lamps. These shapes originate from the ShapeNet [CFG\*15] dataset, and PartNet additionally provides a hierarchical decomposition of each shape into its constituent parts. For example, a chair could be first decomposed into backrest, seat, and base parts, which could be further decomposed into smaller constituent parts. This decomposition is consistent across all shapes of a category. The ABC dataset [KMJ\*19] contains a set of one million mechanical parts and assemblies that were created with the On-Shape [PTC] CAD modeling software. In addition to the meshes, the ABC dataset provides the parametric boundary curves and surfaces that were used to author each shape. The Fusion 360 Gallery dataset [WPL\*21, WJC\*22] is based on 20k designs created with the Autodesk Fusion 360 CAD modelling software [Auta]. In addition to meshes, several different types of high-level information are provided, including a hierarchical decomposition into sub-assemblies, joints between parts, holes, contact surfaces, ‘sketch and extrude’ construction sequences of some designs, and a segmentation of each shape into the modeling operations used to create each part of the surface.

**Materials and Textures:** Procedural materials used in practice are typically defined as node graphs, which are visual representations of functional programs. Several recent neurosymbolic methods have experimented with synthesizing these node graphs. Three main data sources are available, only two of which are public. Substance Source [Ado21b] is a dataset of roughly 7k node graphs

that were generated by professional material artists. It was used as the training set by MatFormer [GHS\*22], but is not publicly available. A public alternative is the Substance 3D Community Assets [Ado21a], which is a website containing publicly available node graphs created by users of Substance 3D Designer [Adob]. However, the graphs on this website have not been assembled into a dataset yet. Nvidia vMaterials [Nvi22] is a dataset of over 800 realistic materials that also provide node graphs.

## 5. Application: 2D Shapes

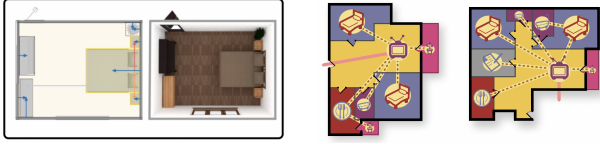
From freeform sketches of ideas, to detailed technical drawings, 2D shapes are a building block of human communication. Augmenting peoples’ ability to communicate and express concepts is at the heart of research in neurosymbolic 2D shape creation. 2D shapes can also play a foundational role in the creation of 3D shapes, as discussed in Section 6. In this section, we review the application areas of layout generation, engineering sketch generation, vector graphics generation, and inverse 2D graphics. Table 2 situations each prior work we discuss within our design space.

### 5.1. Layout Generation

The problem of layout generation arises in a number of domains, including graphic design layout, floor plan synthesis, and furniture layout. Common across these domains is the spatial and topological arrangement of layout primitives (potentially to meet a set of

Method	Task Spec		DSL			Synthesizer		Execution	Refinement	Learning	
	Input	Output	Paradigm	Primitives	Mutability	Search	Guidance			End-to-end	Modular
Para et al. [PGK*21]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
PlanIT [WLW*19]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
CurveGen [WJL*21]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
SketchGen [PBG*21]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
CAD as Language [GBL*21]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
Vitruvion [SZRA22]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
SkexGen [XWL*22]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
SVG-VAE [LHES19]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
DeepSVG [CDAT20]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
Im2Vec [RGLM21]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
DeepVecFont [WL21]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
Ellis et al. [ERSLT18]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
Guo et al. [GJB*20]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
Ellis et al. [ENP*19]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
SPiRAL [GKB*18]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
DreamCoder [EWN*21]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
Yang et al. [YP22a]	Visual Target	Visual Examples	Visual Examples	Hard-coded Primitives	Deterministic Program	Preprocessed	Preprocessed	Program Distribution			Functional
⊗ Neural	⊗ Visual Target	⊗ Visual Examples	⊗ Visual Examples	⊗ Hard-coded Primitives	⊗ Deterministic Program	⊗ Preprocessed	⊗ Preprocessed	⊗ Program Distribution			⊗ Functional
⊙ Imperative	⊙ Constraint	⊙ Hard-coded Primitives	⊙ Hard-coded Primitives	⊙ Fixed	⊙ User-in-the-loop	⊙ Probabilistic Program	⊙ Preprocessed	⊙ Evolving			⊙ Database Retrieval
⊕ Explicit Enumeration	⊕ Stochastic Search	⊕ Constraint Satisfaction	⊕ Constraint Satisfaction	⊕ Synthetic Programs	⊕ Synthetic Programs	⊕ Direct	⊕ Direct	⊕ Solver			⊕ Learned Proxy
⊖ Smooth Relaxation	⊖ Reinforcement Learning	⊖ Programs from People	⊖ Programs from People	⊖ Synthetic Programs	⊖ Synthetic Programs	⊖ Bootstrapping	⊖ Bootstrapping				

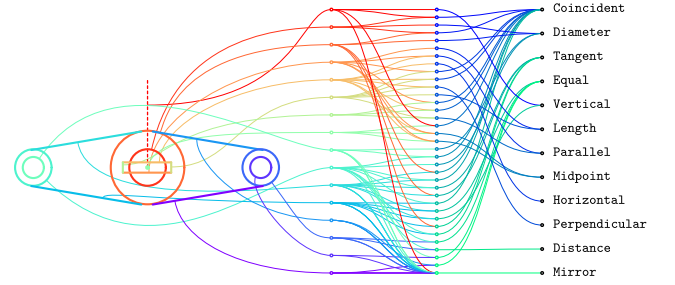
**Table 2:** A summary of the work on neurosymbolic 2D shape modeling discussed in Section 5, where each approach is situated in our design space.



**Figure 5:** Neurosymbolic layout generation via learning to synthesize layout constraint programs. Left: A graph neural network synthesizes a constraint graph specifying furniture spatial relationships; a neurally-guided search then finds object placements that satisfy those constraints [WLW\*19]. Right: A transformer + pointer network architecture synthesizes a floor plan layout constraint program, which is then solved to produce a floor plan [PGK\*21].

constraints). In floor plan synthesis, for example, an adjacency constraint may dictate that the master bedroom reside next to an ensuite bathroom, forming a spatial and topological relationship between the two rooms. A significant amount of research related to layout generation exists; here, we cover only select works that fall under the neurosymbolic modeling umbrella.

Floor plan synthesis involves the generation of realistic floor plans to meet a series of architectural constraints. Para et al. [PGK\*21] introduce a neurosymbolic approach for this task. First, a transformer network is used to generate constraints on the parameters of layout elements (e.g. room type and width/height ranges); second, relationship constraints (e.g. door or wall edges) are generated between elements using pointer networks. These constraints together make up a constraint program which is then executed (i.e. solved) to generate a spatial layout consistent with the constraints (Figure 5 right). The goal of furniture layout, or more generally scene generation, is to position objects (e.g. furniture) in a scene (e.g. an empty bedroom with a given shape) in a realistic manner. PlanIT [WLW\*19] divides the problem into a plan-

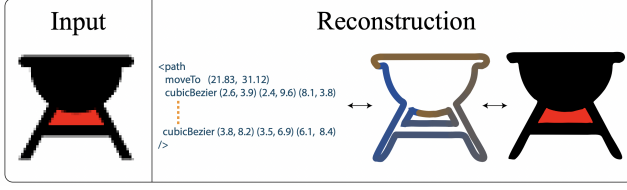


**Figure 6:** Engineering sketches form the 2D basis of parametric CAD. Geometric primitives, such as circles and lines (left), are represented as nodes in the first column. Sketch constraints are applied to the geometric primitives and represented as nodes in the second column. The third column lists the constraint types in order of frequency. The order of nodes (top to bottom) follows the generation order of the learned model from Ganin et al. [GBL\*21].

ning phase and an instantiation phase. In the planning phase, a deep graph convolutional generative model is used to synthesize relation graphs which form a constraint program: nodes indicate objects to be placed in the scenes, and edges indicate spatial relationship constraints between them. In the synthesis phase, a backtracking search is used to find placements for the objects implied by the graph nodes. This search is guided by a convolutional network that operates on top-down views of the partial scene (Figure 5 left).

## 5.2. Engineering Sketch Generation

Software for the creation of 2D engineering sketches dates back to the very first CAD system [Edw63]. Engineering sketches form the 2D basis of parametric CAD, the foremost 3D modeling paradigm used to design manufactured objects from automobile parts, to electronic devices, to furniture. Engineering sketches (as shown

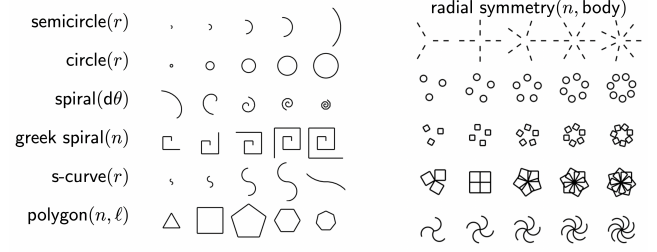


**Figure 7:** The *Im2Vec* system learns a neural network that can reconstruct, synthesize, and interpolate between vector graphics programs [RGLM21]. It uses an end-to-end training setup with a differentiable vector graphics rasterizer, avoiding the need for ground-truth SVG programs as training data.

in Figure 6) consist of composite curves made up of 2D geometric primitives (e.g. lines, arcs, circles), topological information about how these primitives connect together, and constraints defined using topology (e.g. coincidence, tangency, symmetry). Essentially, an engineering sketch is a program: given a change to its parameters, the program can be re-executed to produce new 2D geometry (which satisfies any constraints). The ability to generate high-quality engineering sketches is an enabling technology for the automatic generation of parametric CAD files suitable for manufacturing. The availability of large-scale engineering sketch datasets (see Section 4) has enabled learning-based approaches to engineering sketch generation. A number of concurrent works [WJL\*21, PBC\*21, GBL\*21, SZRA22] approach the task of engineering sketch generation by treating the sketch as a sequential language that can be modeled using Transformers, either with or without constraints. Providing control over the generation of engineering sketches is another outstanding challenge, as designers need ways to influence the generated shape. One approach is to condition the network on user-provided images [GBL\*21] or hand-drawn sketches [SZRA22]. Another approach, introduced by Skex-Gen [XWL\*22], is the use of codebooks [RVdOV19] to separate control of sketch geometry and topology into learned codes that can be selected to guide the generation of topologically or geometrically similar shapes. Yang et al. [YP22b] learn modular ‘concepts’ from engineering sketch graphs to capture repetitive design patterns and aid with image-conditioned generation and auto-completion tasks. Transformer-based generation approaches have several limitations. Spatial resolution is currently limited and addressed by using quantized vertex positions on a 8 bit grid or smaller. Higher spatial resolution allows more accurate sketches to be generated but vastly increases the size of the prediction space. Another limitation, due to the well known issue of modeling long sequences with Transformers, is generating complex sketches with higher numbers of curves.

### 5.3. Vector Graphics Generation

Adjacent to engineering sketches is the domain of vector graphics, which also makes use of 2D geometric primitives, such as lines and Bézier curves, to represent scalable graphical artwork. An SVG file is essentially a simple program, which uses compositions of parametrized functions (e.g. `moveTo`, `lineTo`, `cubicBezier`) to produce geometry. SVG-VAE [LHES19] was one of the first deep



**Figure 8:** Example learned library routines using *Dream-Coder* [EWN\*21] on a LOGO graphics task. The learned routines include both parametric routines for drawing families of curves (left) as well as higher-order functions that take entire sub-programs as input (right).

learning-based approaches to generate vector graphics using an LSTM-based VAE trained on font characters. DeepSVG [CDAT20] demonstrated both font character and icon generation using a non-autoregressive Transformer-based architecture. Although promising, both works produce results lacking the regularities found in human-designed vector graphics, such as symmetric, concentric, or tangent curves. To address this limitation, recent work has leveraged alternate representations to sequences of vector graphic primitives. DeepVecFont [WL21] uses a hybrid raster/vector representation where the raster graphic is used as a supervision signal to improve the vector program output. This approach combines the benefits of learning human-designed curve topology from vector supervision (i.e. how curves are connected together) and curve geometry from raster supervision (i.e. the location of the curves). As vector graphics do not conform well to a grid structure, like engineering sketches do, higher spatial resolution is required. The geometric loss provided by raster supervision appears to be critical at this time to get good visual results.

### 5.4. Inverse 2D Graphics

In addition to the generation of graphics, the inverse problem of recovering a 2D shape program from approximate input, such as an image or hand-drawn sketch, has been studied in recent literature. Ellis et al. [ERSLT18] introduce an approach to convert simple hand drawings into a program representation that captures the regularities commonly found in human design, such as symmetry, repetition, and structural reuse. This approach uses neurally-guided search to extract primitive graphical elements from the input sketch, followed by constraint-based program synthesis to find a program that generates those primitives. Guo et al. [GJB\*20] show how neural guidance can be used to learn an L-system representation of pixel images with branching structures, as found in trees and other natural patterns. Ellis et al. [ENP\*19] use reinforcement learning to synthesize simple 2D CAD programs that reproduce an input shape; similarly, SPIRAL [GKB\*18] uses RL and adversarial learning to synthesize simple painting programs that reproduce an input image. Im2Vec [RGLM21], shown in Figure 7, uses a differentiable rasterization pipeline and reconstruction loss between the input image and the rasterized generated vector graphic. Dream-Coder [EWN\*21], shown in Figure 8, can infer a LOGO graphics



program for a target shape. It interleaves learning to infer programs via neurally guided search with learning new abstractions (i.e. sub-routines) for its drawing language, which then makes it easier to infer programs for more complex targets. DreamCoder’s discovered abstractions capture complex concepts, producing primitive shapes such as semi-circles and polygons, as well as regularities such as radial symmetry (Figure 8).

## 6. Application: 3D Shapes

The demand for 3D models has never been higher. Applications from within entertainment & gaming systems, to augmented & virtual reality, and even those in vision & robotics, all desire access to high-quality 3D objects. Stakeholders in these areas are often not content with unstructured assets: their applications need shapes that are interactive & editable, yet cover a wide range of outputs while maintaining high fidelity. Neurosymbolic methods for 3D shapes have been explored with these criteria in mind. Table 3 situates each prior work we discuss within our design space.

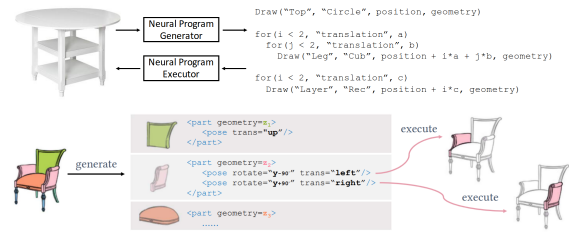
### 6.1. Inferring 3D Shape Programs

Many methods have investigated how to infer the underlying structure of a 3D object. For instance, if one is able to find a program that is a good representation of an input shape, then the program structure can be used to analyze or manipulate the underlying 3D object. This is a sub-problem of program synthesis, where the input specification is a visual representation of a 3D object, which the inferred program’s output must geometrically match. This problem is also sometimes referred to as visual program induction.

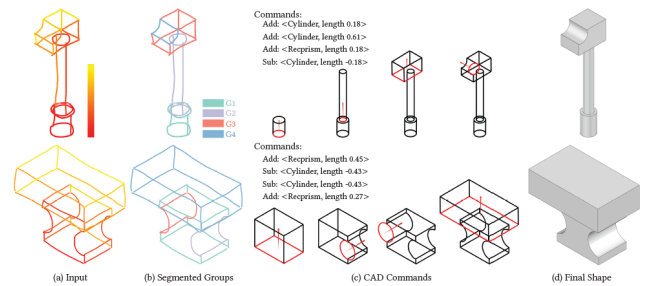
Within this area, a general approach is to linearize programs into sequences of tokens. Then a program inference network can be trained to autoregressively generate program tokens, conditioned on visual data. In Fusion 360 Gallery [WPL\*21], a model learns to represent 3D shapes as sequences of sketch and extrude CAD operations by training in a supervised fashion on human-written programs. While supervised learning is the preferred strategy when program datasets are available, many domain+language combinations lack such information. When a program dataset is not available, approaches have investigated how to use reinforcement learning to reconstruct 3D shapes as sequences of commands.

CSGNet [SGL\*18] uses beam search guided by an autoregressive inference network to find CSG programs that achieve a low-reconstruction error with respect to the input. The parameters of these programs can be further improved with a heuristic refinement step. Ellis et al. [ENP\*19] employ a Sequential Monte Carlo search guided by a policy network that synthesizes code and a value network that learns to assess the prospects of partial programs. They equip this search with access to a REPL (read-eval-print-loop), framing program search as a Markov Decision Process (MDP) where each network reasons over a *state* represented by the executed output of partially constructed programs. While policy gradient reinforcement learning offers a domain-agnostic solution that does not depend on a dataset of ground-truth programs, it is notoriously unstable, due to high variance gradients, which can hurt its convergence speed and performance.

To deal with more complex DSLs for 3D shapes,



**Figure 9:** (Top) Shape2Prog [TLS\*19] learns how to infer visual programs that capture the structure of input shapes. The program generator is trained in an end-to-end fashion with a learned proxy executor. (Bottom) ProGRIP [DKD\*22] improves the reconstruction fidelity of this approach by replacing hard-coded primitives with learned neural implicits.



**Figure 10:** Free2CAD [LPBM22] parses freehand drawings into visual programs. The program is built up through an iterative procedure that alternates between (i) grouping related sketch strokes and (ii) searching for CAD operations that correspond to the segmented group.

PLAD [JWR22] introduced a self-supervised learning method that fine-tunes an inference model for a target domain of interest. This bootstrapping approach maintains domain-generalizability, while avoiding the pitfalls of policy-gradient RL by training with maximum likelihood updates on approximately correct (shape, program) pairs. Alternatively, some methods have explored learning a network that acts as a differentiable relaxation of a program executor: allowing the system to train in an end-to-end fashion with respect to a geometric loss. Shape2Prog [TLS\*19] uses a network that learns from synthetic data to act as a differentiable proxy for their execution engine (Figure 9, Top). This approach has been recently extended in ProGRIP [DKD\*22] (Figure 9, Bottom). ProGRIP replaces the hard-coded primitives used by Shape2Prog with learned neural primitives, which also removes the need to pretrain any proxy network with hand-crafted synthetic data.

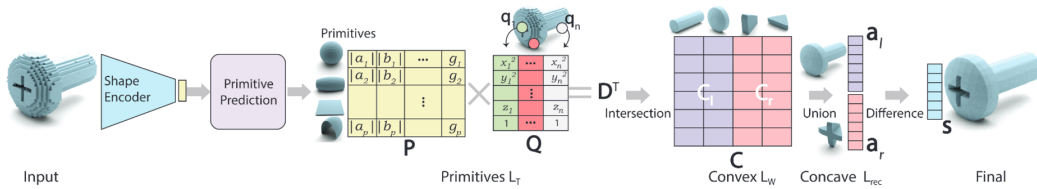
Some prior work has developed shape program inference approaches specially tailored for their particular domain. When inferring a 3D CAD model conditioned on an input sketch, a network’s noisy predictions can be regularized by mapping them to symbolic CAD operations with a fitting procedure. In the Sketch2CAD framework [LPBM20], a network is trained to predict CAD operations that correspond with segmented sketch strokes.

Method	Task Spec		DSL			Synthesizer		Execution	Refinement	Learning	
	Input	Output	Paradigm	Primitives	Mutability	Search	Guidance			End-to-end	Modular
Fusion 360 Gallery [WPL*21]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
CSGNet [SGL*18]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
Ellis et al. [ENP*19]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
PLAD [JWR22]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
Shape2Prog [TLS*19]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
ProGRIP [DKD*22]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
Sketch2CAD [LPBM20]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
Free2CAD [LPBM22]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
Zone Graphs [XPC*21]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
Lambourne et al. [LWJ*22]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
Point2Cyl [UyCS*22]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
CSG-Stump [RZC*21]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
CAPRI-Net [YCL*22]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
ExtrudeNet [RZC*22]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
UCSGNet [KZK20]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
DeepCAD [WX21]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
SkeXGen [XWL*22]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗	⊗
ShapeAssembly [JBX*20]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗
ShapeMOD [JCG*21]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		⊗
Ritchie et al. [RJT18]	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗		*

⊗ Neural	⊗ Visual Target	⊗ Visual Examples	⊗ Deterministic Program	⊗ Probabilistic Program	⊗ Program Distribution	⊗ Functional
⊗ Imperative	⊗ Constraint	⊗ Hard-coded Primitives	⊗ Fixed	⊗ Preprocessed	⊗ Evolving	⊗ Database Retrieval
⊗ Explicit Enumeration	⊗ Stochastic Search	⊗ Constraint Satisfaction	⊗ User-in-the-loop	⊗ Direct	⊗ Solver	⊗ Learned Proxy
⊗ Smooth Relaxation	⊗ Reinforcement Learning	⊗ Programs from People	⊗ Synthetic Programs	⊗ Bootstrapping		

**Table 3:** A summary of the work on neurosymbolic 3D shape modeling discussed in Section 6, where each approach is situated in our design space. \* Ritchie et al. [RJT18] use supervised training for their neural language primitives.



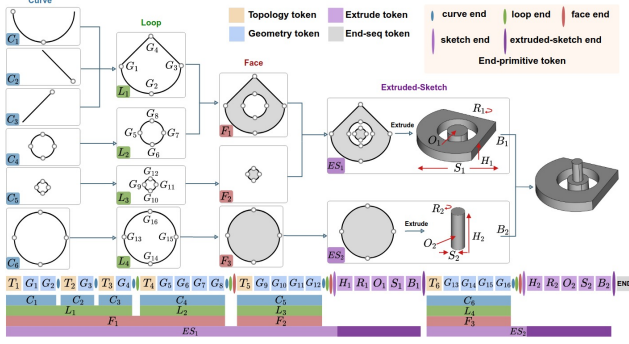
**Figure 11:** CAPRI-Net [YCL\*22] infers a CSG program that reconstructs an input shape. Its neural architecture acts as a CSG execution engine, which limits the types of program structures that it can discover but allows the system to train in an end-to-end fashion.

Free2CAD [LPBM22] generalizes this system by additionally learning how to segment a complete sketch into groups that can be mapped to CAD operations (Figure 10). As the search space of methods that aim to reverse-engineer CAD models can be expansive, some approaches have investigated heuristics that make the problem more tractable. Zone Graphs [XPC\*21] converts a boundary representation of a 3D Shape into a partition of zones. While this partitioning permits the use of enumerative search strategies, further guiding this search with a neural module results in improved programs. These domain-specific methods perform well for their respective applications, but they rely on heuristics and design principles that do not readily generalize to other domains.

A central difficulty of neurosymbolic models is that gradients cannot flow through symbolic elements, complicating end-to-end learning. A workaround for this issue is to train a system where symbolic DSL components are replaced with differentiable proxies. Then at inference time, these proxies can be replaced by DSL expressions with similar outputs. In Lambourne et al. [LWJ\*22], a network learns to reconstruct a target shape with a collection of differentiable extrusions learned from human-written programs. At inference time, these extrusions are replaced with observed profiles from an input collection, resulting in a complete sequence of CAD

operations that reconstruct the input. In Point2Cyl [UyCS\*22], a network learns to represent a shape by combining differentiable extrusion proxies with boolean operations. These proxies can then be converted into extrusion cylinders through differentiable, closed-form formulations in order to produce an editable CAD model. This scheme of relaxation with replacement allows for end-to-end learning with high-fidelity outputs but often requires clever insights for both architecture construction and the replacement scheme that are domain-specific.

Pushing this trend further, some methods learn how to infer 3D shape programs in a purely end-to-end fashion, by designing neural architectures that act as a smooth relaxation of a DSL executor. Some languages are more amenable to this approach than others. For instance in CSG, primitives are easily differentiable, and *hard* boolean set operations can be replaced with *soft* versions. The neural architecture used often places restrictive constraints on the *structures* of programs that can be found with these approaches. We depict the architecture of one such method, CAPRI-Net [YCL\*22] in Figure 11. In their formulation primitives are converted into convexes through intersection, *left* and *right* shapes are created by unioning convexes, and the final output is the difference between the *left* shape and *right* shape. In CSG-Stump [RZC\*21], the out-



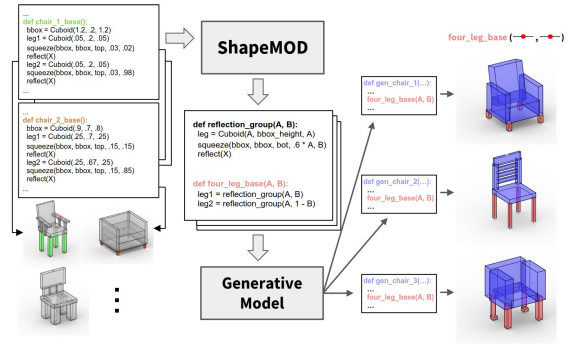
**Figure 12:** SkexGen [XWL\*22] represents 3D shapes as sequences of sketch-and-extrude operations, where the tokens of different operator types are produced by disentangled codebooks. A Transformer learns over a distribution of shapes encoded in this representation, and is able to synthesize novel 3D geometry through auto-regressive sampling.

put CSG program is a union over intersections of either primitives or complements of primitives. ExtrudeNet [RZC\*22] extends upon the CAPRI-Net architecture by replacing quadric primitives with extruded 2D sketches, while maintaining end-to-end differentiability. UCSGNet [KZK20] explores a higher diversity of program structures, by evaluating many boolean operations in parallel, but this comes at the cost of more complex programs with worse reconstruction performance. For domains that permit baking program execution behavior into a neural architecture, such methods typically achieve the best geometric reconstruction performance. The limitation of these approaches is that each architecture is inherently tied to a specific domain, and even within a domain, the class of programs producible by the neural network is typically constrained.

## 6.2. 3D Shape Generation

A number of techniques have also explored how neurosymbolic models can be used to generate novel 3D shape instances. When a dataset of ground-truth (i.e. human-written) shape programs exists, deep generative modeling techniques can be directly employed. DeepCAD [WXZ21], trains a transformer-based autoencoder to auto-regressively generate sequences of sketch and extrude commands conditioned on the input vector. Novel shapes are then synthesized by prompting the decoder with new conditioning vectors output by a latent GAN. SkexGen [XWL\*22] improves upon this paradigm by designing an architecture where sub-networks are specialized for particular types of CAD operations (Figure 12). Specifically, a "Sketch" branch reasons about 2D sketches (tokens related to their topology and geometry), while an "Extrude" branch predicts how sketches should be lifted into 3D (tokens related to an extrusion direction).

When human-written programs are not available, an alternative approach is to heuristically *parse* programs from dataset of 3D shapes with structured annotations. ShapeAssembly [JBX\*20], introduced a new DSL designed for specifying the part structure of



**Figure 13:** ShapeMOD [JCG\*21] takes a collection of 3D shape programs as input and makes them more compact by automatically discovering common macros which can be re-used across the collection. Programs rewritten with these macros can benefit downstream applications such as generative modeling.

manufactured 3D objects. A hierarchical sequence VAE learns to write new ShapeAssembly programs after training over a dataset of programs parsed from PartNet shapes. One issue with parsing programs directly from shape repositories is that the resulting programs might be overly complex. The ShapeMOD algorithm [JCG\*21], addresses this issue by automatically discovering macro operations that abstract out common structural and parametric patterns over a collection of shape programs (Figure 13). Deep generative models can then benefit from training over program distributions that are rewritten to use macro operations.

Related to the problem of learning a generative model over a program dataset, Ritchie et al. [RJT18] developed a procedure for inferring probabilistic programs that explain a small set of example shapes. As inferring probabilistic programs is very difficult, the method makes a number of simplifying assumptions: it requires that the input shapes have a consistent hierarchical organization of parts and it produces programs that are similar to context-free grammars. Under these assumptions, the method demonstrates the power of this representation: the inferred probabilistic programs capture both hierarchical structure and continuous relationships of parts present in modular inputs, where part transform distributions are parameterized by a learned network. This allows new objects in the style of the exemplars to be synthesized by sampling from the distributions captured by the probabilistic program.

## 7. Application: Materials & Textures

Procedural workflows have gained increased popularity in the material and texture design community over the last few years, driven by modern authoring tools [Adob, Sid, Ble]. Materials are a good fit for procedural workflows, due to the presence of repeated structures and self-similarity, typically with some stochastic variations between the repetitions. Figure 4 bottom-right, shows some examples of procedurally generated materials. Procedural workflows offer several benefits to material artists, such as (i) a non-destructive workflow, where any operation can be changed at any point in the authoring process, (ii) the ability to quickly create variations



Method	Task Spec		DSL			Synthesizer		Execution	Refinement	Learning	
	Input	Output	Paradigm	Primitives	Mutability	Search	Guidance			End-to-end	Modular
Liu et al. [LGD*18]	⊗	⊗	⋈	⋈	⬇	⊞	⊗	→	⊗	⬇	⬇
Hu et al. [HDR19]	⊗	⊗	⋈	⋈	⬇	⊞	⊗	→			⬇
Tchapmi et al. [TRT*22]	⊗	⊗	⋈	⋈	⬇	⊞	⊗	→			⬇
MATch [SLH*20]	⊗	⊗	⋈	⋈	⬇	⊞	⊗	→			⬇
Differentiable Proxies [HGH*22]	⊗	⊗	⋈	⋈	⬇	⊞	⊗	→			⬇
Hu et al. [HHD*22]	⊗	⊗	⋈	⋈	⬇	⊞	⊗	→	⊗	⬇	⬇
MatFormer [GHS*22]	⊗	⊗	⋈	⋈	⬇	⊞	⊗	→			⬇
⊗ Neural	⊗ Visual Target	⊗ Visual Examples	⊗ Deterministic Program	⊗ Probabilistic Program	⊗ Program Distribution	⋈ Functional					
⬇ Imperative	⬇ Constraint	⬇ Hard-coded Primitives	⬇ Fixed	⬇ Preprocessed	⬇ Evolving	⊞ Database Retrieval					
⊞ Explicit Enumeration	⊞ Stochastic Search	⊞ Constraint Satisfaction	⊞ User-in-the-loop	⊞ Direct	⊞ Solver	⊗ Learned Proxy					
⊗ Smooth Relaxation	⊗ Reinforcement Learning	⊗ Programs from People	⊗ Synthetic Programs	⊗ Bootstrapping							

**Table 4:** A summary of the work on neurosymbolic material & texture modeling discussed in Section 7, where each approach is situated in our design space.

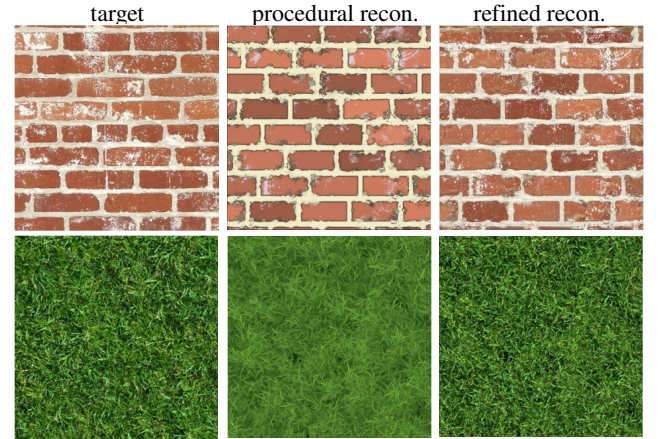
of a material by adjusting parameters of the procedure, (iii) automatic tileability, since the provided procedural operations typically ensure tileability, (iv) resolution-independence, and (v) the ability to easily scale the material up or down by adjusting the number/frequency of repetitions. Procedural materials offer an increasingly fertile field for research into neurosymbolic models, as the interest in procedural materials increases and more data becomes available.

Early approaches for procedural materials and textures [Pea85, Per85, WK91, Wor96, LP00, GLLD12] and some more recent approaches [LDHM16, GAD\*20] provide a black-box function to the user and expose parameters that can be modified to control the generated result. This black-box may, for example, implement a parametric noise function, a reaction-diffusion approach [WK91], a physically-inspired simulation [LDHM16], or point process basis functions [GAD\*20].

Procedural materials that are used in practice [Adob, Sid, Ble] also provide control over the content of the black-box using a DSL. To create a material, the program is executed, combining and transforming a set of initial noises and patterns through several image filtering operations. This DSL is exposed to the user as a *node graph* (Figure 4 bottom-right). Nodes correspond to image filtering/processing functions, or to generators for the initial noises/patterns. Each node has a set of inputs and outputs. Inputs typically consist of images or scalar/vector-valued parameters, and outputs are one or multiple processed images. Edges control the data flow between the inputs and outputs of different nodes—in essence, the node graph specifies a functional program.

Neurosymbolic approaches in this domain aim to synthesize such node graph programs: either inferring a program for a given input image, or sampling novel programs whose characteristics match a dataset of examples. They differ in the approach used to synthesize such programs. Typically these methods have separate approaches to handle the generation of the program structure (node types and edges) and the generation of the program parameters (node parameters). Table 4 situates each prior work we discuss within our design space.

**Program and parameter retrieval:** An early method [LGD\*18] creates both program structure and parameters for a given visual target image by retrieving them from a pre-defined dataset of pa-

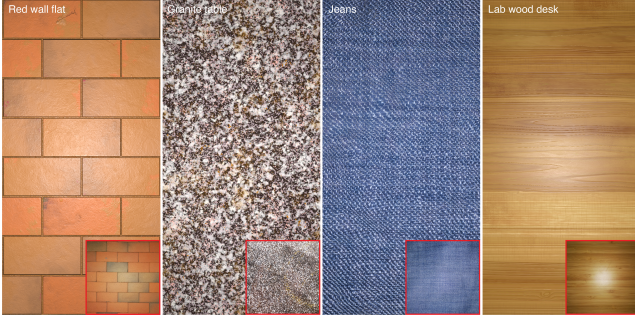


**Figure 14:** Hu et al. [HDR19] synthesize a procedural material matching a given target image by retrieving a procedural program. Parameters for the program that reproduces the target image are estimated using a trained network. Additionally, Hu et al. use a style transfer network to refine the output of the program to further improve its match with the target.

rameterized programs. A perceptual metric based on psychophysical experiments is introduced as a distance metric for both program and parameter retrieval. Generation of programs through retrieval from a large dataset is limited to the programs available in the dataset and has, therefore, limited coverage over the space of possible visual targets.

**Program retrieval, parameter synthesis:** Two approaches [HDR19, TRT\*22] improve upon this work by using a neural network to estimate parameters given a retrieved program structure and a visual target, rather than retrieving parameters (Figure 14). The earlier approach [HDR19] requires training a parameter estimator for each program, while the more recent approach [TRT\*22] improves scalability by training a single estimator for multiple programs. MATch [SLH\*20] goes a different route by directly optimizing the parameters of a retrieved program to match the program’s output to the given visual target, rather than using a network to amortize the optimization. This improves





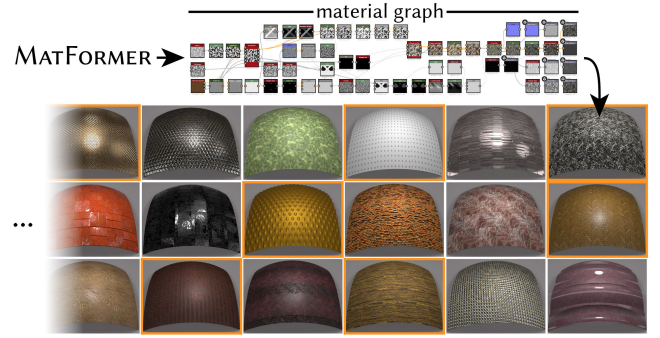
**Figure 15:** MATch [SLH\*20] synthesizes a procedural material that matches a given target image (lower right corner). A differentiable procedural program is retrieved from a large dataset of material programs, and the parameters of this program are optimized to match the target using gradient descent.

accuracy over amortized optimization. The authors introduce differentiable versions of most operators in the program to enable optimization with gradient descent. A few examples of optimized materials are shown in Figure 15. Since not all operators can be made differentiable, only a subset of the program parameters can be optimized. Differentiable Proxies [HGH\*22] tackles the problem of non-differentiable operators by training small neural networks to approximate these operators. The networks act as proxies that are differentiable and have parameters that can be optimized with gradient descent.

**Program and parameter synthesis:** Recently, methods have started to synthesize the program structure in addition to the parameters, rather than retrieving it from a dataset. Hu et al. [HHD\*22] propose to synthesize programs that reconstruct given target images. The programs have a special constrained structure, where procedural masks based on point process basis functions [GAD\*20] are first used to define a tree of sub-regions in a material that contains increasingly uniform texture, and noise generators are then applied to the leaves to generate a texture. The procedural masks are found using interactive segmentation of a visual target, and noise generators are fit to the content of a region based on local spectra. The required user interaction for segmentation and the reduced generality of the programs due to their constrained structure are limitations of the method. MatFormer [GHS\*22] proposes an unconditional generator for more general material programs. It is based on three Transformer models that are trained to synthesize nodes, edges, and node parameters of a node graph, respectively. Since Transformers work on linear sequences, various strategies to linearize a node graph are proposed. Figure 16 shows an example of a generated node graph and several examples of materials that were created with generated node graphs. However, MatFormer does not support conditional generation, for example, to reproduce a given target image.

## 8. Conclusion and Future Work

In this report, we reviewed *neurosymbolic* models: techniques that combine the best features of procedural models and machine learn-



**Figure 16:** MatFormer [GHS\*22] is an unconditional generator for procedural materials represented as node graphs. It creates a node graph with three Transformers that are trained to generate nodes, node parameters, and edges, respectively.

ing to generate visual data for computer graphics applications. We defined a formal design space for such models, providing a framework for organizing different possible instantiations of neurosymbolic models. We then surveyed recent work on neurosymbolic modeling in 2D shape modeling, 3D shape modeling, and material & texture modeling, placing each prior work into our design space.

In addition to organizing past work, our design space provides another benefit: identifying areas of the space which are sparsely populated by prior work (or empty), as such areas may warrant further exploration. With this in mind, the following are some open problems and opportunities for future work in neurosymbolic modeling, in the context of Computer Graphics.

**New application domains:** Perhaps the most immediate opportunity for future work is to apply some of the techniques described in this report to modeling data in other graphics domains. Any type of visual data is fair game, but it is especially worth looking at domains where programmatic representations have already proven useful. One such example is shader programming, a domain for which large repositories of data exist [Ini]. Character animation & behavior modeling could also warrant investigation, as some crowd simulation software already uses hand-authored animation procedures [Mas]. This report covered some recent work in fabrication-aware 3D shape modeling; other fabrication-aware design domains are worth exploring. For example, sewing patterns for clothing exhibit recurring structural patterns which a neurosymbolic model could learn to predict as programs [BGK\*13]. Finally, there also exist opportunities in non-visual domains which are relevant to computer graphics. For instance, games can use procedural representations of music to dynamically alter their soundtrack in response to player actions [Sys]; neurosymbolic modeling techniques might bring new capabilities to this application or could produce such representations more automatically.

**More complex programs:** Learning to produce generative programs with complex structure is a challenging problem; as a consequence, much of the early work in neurosymbolic modeling, which we have covered in this report, uses simple languages (e.g.

CSG for 3D shape modeling). To move beyond the research lab and be useful for real-world design applications, neurosymbolic models must be able to produce complex programs in rich, fully-featured domain-specific languages: for example, full CAD languages such as OpenSCAD [Ope], or shading languages such as GLSL. The MatFormer model, which produces programs in the form of detailed Substance material graphs, is a step in this direction [GHS\*22]. However, this model was trained on a large dataset of human-authored programs, which brings us to our next open problem:

**Learning without direct supervision:** The most straightforward way for a neurosymbolic model to learn to produce programs is to train on programs authored by **people** (👤). However, such data is rarely available at scale for most languages of interest. Learning without such supervision, especially as the target language becomes more complex, remains a challenge. Our design space identifies three learning approaches for this setting: (🧘) **Smooth Relaxation**, (🎮) **Reinforcement Learning**, and (🔁) **Bootstrapping**. Each of these approaches warrants further investigation: can one design more general-purpose smooth relaxations for new languages? Are there new techniques from the RL literature which work well for synthesizing generative programs? Can bootstrapping approaches such as PLAD [JWR22] be extended to handle more complex languages, possibly involving hierarchical structure?

**Discovering new languages:** Thus far, work on neurosymbolic modeling has targeted existing domain-specific languages, potentially modifying them by adding new abstractions. Might it be possible for entirely new languages to be **invented** (🔧), to best fit the domain of data being modeled? Neurosymbolic models make it possible to consider this question: one could imagine learning a set of **neural primitives** (🧠) which represent the visual “atoms” of the data domain being modeled (e.g. objects, parts) and then learning a set of primitive functions which combine these atoms to create more complex visual structures. The language discovered could adapt to the particular data distribution on which it is trained, lending a new definition to the abbreviation “DSL”: *distribution-specific languages*.

**Capturing user intent:** The usefulness of a neurosymbolic model is largely determined by how well its *task specification* captures a user’s intent. On the input side of this specification, supporting more input types can help make neurosymbolic models more useful for a wider population of users. While defining our design space (Section 3.1), we have already mentioned several new types of input that may warrant future exploration. For instance: we are not aware of any prior neurosymbolic models which take a **text prompt** (🗨️) as input, though this seems like an appealing way to specify user intent (if the recent success of text-to-image generators is any indicator [RDN\*22, SCS\*22, RBL\*22]). Such natural language input could specify desired attributes of the visual data to be generated, of the programs which will generate that data, or both (including connections between the two, e.g. that a particular program construct should be used to generate a particular visual feature).

On the output side of the task specification, neurosymbolic models must produce programs that are usable by people. How do we

assess how usable a program is, and how can we encourage neurosymbolic models to produce such programs? As mentioned in Section 3.3, the only prior on program structure that most previous work considers is a preference for shorter programs. This is justified with appeals to information theory or Occam’s razor, but not to usability. In fact, excessively short programs can become *unusable* (see obfuscated code contests or business card raytracers [San]). Furthermore, different users may prefer different program structures. Are there better priors than program length for the usability of a program? What are effective mechanisms for users to specify their preferences about program structure?

**Human interpretation and interaction:** While program representations have the key advantage of exposing an interpretable structure, reasoning about programs and manipulating them is often quite challenging and requires domain and coding expertise. By combining program representations with learning, we can advance symbolic reasoning in three fundamental directions: interpretation, manipulation, and composition.

*Code interpretation* refers to the ability to analyze code: exposing its structure, capabilities, and potential bugs. The programming language community has made significant advances in this space by using theorem provers or SMT solvers to detect and expose bugs [IK16]. Such methods, however, are still limited in scale and domain. How can we leverage advances in machine learning to help detect code errors and expose them? Can such systems go a step beyond detection and also propose solutions?

For code that generates a visual output, *manipulation* is extremely important to enable customization and iterative design. While recent work describes techniques to optimize code based on the direct manipulation of the visual output [HLC19, CSQ\*22, GKG\*22], these methods are limited in the types of variations they enable, only allowing the program parameters to change, but not its structure. Furthermore, existing techniques fundamentally struggle to infer the user intent, since direct visual manipulation is a *partial* (and therefore ambiguous) specification. How can neurosymbolic models address these challenges to enable a wider class of variations that are not limited to parameter changes but also require program rewrites? Can we leverage existing datasets or extract information from user interactions to learn to disambiguate the partial specifications?

Typical programs that generate visual outputs are designed with hierarchical and compositional structures. This design decision is not only critical for human-understandable editing and interaction, it also allows analysis methods to reason over different semantic parts of a visual model. Creating new visual content by mixing and matching existing components is a fundamental design strategy in computer graphics [FKS\*04]. However, code *composition* is challenging, as it requires that certain properties hold at the interface of program components. Generating code that can be easily decomposed and re-used is a tedious, error-prone task. Further, understanding how to constrain or manipulate parts to enable seamless compositions is challenging. Can neurosymbolic reasoning enable design through composition with symbolic representations?

## 9. Author Bios

**Daniel Ritchie** is the Eliot Horowitz Assistant Professor of Computer Science at Brown University. He received his PhD from Stanford University. His research sits at the intersection of computer graphics and artificial intelligence, where he is particularly interested in data-driven methods for designing, synthesizing, and manipulating visual content. In the area of neurosymbolic modeling, he has worked on applying probabilistic programming to procedural modeling, on learning procedural models from a small number of examples, and on deep learning models for generating/inferring procedural representations of 3D shapes. His most recent work focuses on learning procedural representations in the absence of ground-truth programs.

**Paul Guerrero** is a research scientist at Adobe. He received his PhD from the Institute for Computer Graphics and Algorithms, Vienna University of Technology, and at the Visual Computing Center in KAUST. He is working on the analysis of irregular and compositional structures, such as graphs, meshes, or vector graphics, by combining methods from machine learning, optimization, and computational geometry [1, 2, 3, 4, 5, 6].

**R. Kenny Jones** is a PhD student at Brown University where he is supported by a Brown University Presidential Fellowship and advised by Daniel Ritchie. His research explores how machine learning and artificial intelligence techniques can be leveraged to better understand and represent visual data. Relevant to this report, his recent publications have investigated using neurosymbolic models for 3D shape generation, automatic macro discovery, and inferring visual programs.

**Niloy J. Mitra** leads the Smart Geometry Processing group in the Department of Computer Science at University College London and the Adobe Research London Lab. He received his PhD from Stanford University under the guidance of Leonidas Guibas. His current research focuses on developing machine learning frameworks towards generative models for high-quality geometric and appearance content for CG applications. He received the 2019 Eurographics Outstanding Technical Contributions Award, the 2015 British Computer Society Roger Needham Award, and the 2013 ACM SIGGRAPH Significant New Researcher Award. He was elected as a fellow of Eurographics in 2021 and was the SIGGRAPH Technical Papers Chair in 2022.

**Adriana Schulz** is an Assistant Professor at the Paul G. Allen School of Computer Science and Engineering at the University of Washington. She received her PhD from the Massachusetts Institute of Technology. Her research is in the area of computational design and fabrication. Relevant to this report, her recent work uses programmable abstractions to represent and optimize both 3D designs and their fabrication plans. She also uses machine learning and program synthesis techniques to support the design and manipulation of CAD models.

**Karl D.D. Willis** is a Senior Research Manager at Autodesk Research focused on data-driven design software for manufacturing. He holds a PhD in Computational Design from Carnegie Mellon University and has presented his research internationally at

conferences such as ACM SIGGRAPH, IEEE/CVF CVPR, and ICML. His work at Autodesk has won numerous awards including Fast Company Innovation By Design Honoree and Core77 Design Awards Research and Strategy Honoree.

**Jiajun Wu** is an Assistant Professor of Computer Science at Stanford University. He received his PhD from Massachusetts Institute of Technology. His research is in the area of computer vision, artificial intelligence, graphics, and robotics. In the area of neurosymbolic modeling, he has worked on building and learning structured representations for visual data of various modalities (images, shapes, videos) by integrating domain knowledge with data-driven methods.

## Acknowledgments

Daniel Ritchie was supported by NSF awards #1907547 and #1941808. He is also an advisor to Geopipe and owns equity in the company. Geopipe is a start-up that is developing 3D technology to build immersive virtual copies of the real world with applications in various fields, including games and architecture. Jiajun Wu was supported by NSF awards #2120095 and #2211258, Autodesk, IBM, and Salesforce. Adriana Schulz was supported by NSF awards #2219864 and #2017927, Adobe, Intel, Meta, and Amazon.

## References

- [ABJ\*13] ALUR R., BODIK R., JUNIWAŁ G., MARTIN M. M., RAGHOTHAMAN M., SESHIA S. A., SINGH R., SOLAR-LEZAMA A., TORLAK E., UDUPA A.: Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)* (2013). 4
- [Adoa] ADOBE: Substance Designer. <https://www.adobe.com/products/substance3d-designer.html>. Accessed: 2022-09-26. 2, 4
- [Adob] ADOBE INC.: Substance 3D. URL: <https://www.substance3d.com/>. 10, 15, 16
- [Ado21a] ADOBE: Substance 3D community assets, 2021. <https://substance3d.adobe.com/community-assets>. 10
- [Ado21b] ADOBE: Substance source, 2021. <https://substance3d.adobe.com/assets>. 10
- [AFDJ03] ANDRIEU C., FREITAS N., DOUCET A., JORDAN M.: An introduction to mcmc for machine learning. *Machine Learning* 50 (2003), 5–43. 7
- [Auta] AUTODESK: Fusion 360. <https://www.autodesk.com/products/fusion-360/>. Accessed: 2022-10-16. 4, 10
- [Auth] AUTODESK MAYA WIKI: Hypershade. <https://autodeskmaya.fandom.com/wiki/Hypershade>. Accessed: 2022-10-16. 4
- [BBB\*57] BACKUS J. W., BEEBER R. J., BEST S., GOLDBERG R., HAIBT L. M., HERRICK H. L., NELSON R. A., SAYRE D., SHERIDAN P. B., STERN H., ET AL.: The FORTRAN automatic coding system. In *Western Joint Computer Conference: Techniques for Reliability* (1957), pp. 188–198. 4
- [BGK\*13] BERTHOUSOZ F., GARG A., KAUFMAN D. M., GRINSPUN E., AGRAWALA M.: Parsing sewing patterns into 3D garments. *ACM Transactions on Graphics (TOG)* 32, 4 (2013). 17
- [BHB\*18] BATTAGLIA P. W., HAMRICK J. B., BAPST V., SANCHEZ-GONZALEZ A., ZAMBALDI V., MALINOWSKI M., TACCHETTI A., RAPOSO D., SANTORO A., FAULKNER R., ET AL.: Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018). 8



- [Ble] BLENDER ONLINE COMMUNITY: Blender - a 3D modelling and rendering package. URL: <http://www.blender.org>. 15, 16
- [BLW\*20] BAU D., LIU S., WANG T., ZHU J.-Y., TORRALBA A.: Rewriting a deep generative model. In *European Conference on Computer Vision (ECCV)* (2020). 2
- [BMR\*20] BROWN T., MANN B., RYDER N., SUBBIAH M., KAPLAN J. D., DHARIWAL P., NEELAKANTAN A., SHYAM P., SASTRY G., ASKELL A., AGARWAL S., HERBERT-VOSS A., KRUEGER G., HENIGHAN T., CHILD R., RAMESH A., ZIEGLER D., WU J., WINTER C., HESSE C., CHEN M., SIGLER E., LITWIN M., GRAY S., CHESSE B., CLARK J., BERNER C., MCCANDLISH S., RADFORD A., SUTSKEVER I., AMODEI D.: Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)* (2020), vol. 33, pp. 1877–1901. 8
- [BT18] BARRETT C., TINELLI C.: Satisfiability modulo theories. In *Handbook of model checking*. Springer, 2018, pp. 305–343. 7
- [BTLLW22] BOND-TAYLOR S., LEACH A., LONG Y., WILLCOCKS C. G.: Deep generative modelling: A comparative review of VAEs, GANs, normalizing flows, energy-based and autoregressive models. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 44, 11 (2022), 7327–7347. 3
- [BZS\*20] BAU D., ZHU J.-Y., STROBELT H., LAPEDRIZA A., ZHOU B., TORRALBA A.: Understanding the role of individual units in a deep neural network. *Proceedings of the National Academy of Sciences* (2020). 2
- [Cat74] CATMULL E. E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, The University of Utah, 1974. 1
- [CDAT20] CARLIER A., DANELLJAN M., ALAHI A., TIMOFTE R.: DeepSVG: A hierarchical generative network for vector graphics animation. In *Advances in Neural Information Processing Systems (NeurIPS)* (2020), vol. 33, pp. 16351–16361. 6, 9, 11, 12
- [CEP\*21] CHAUDHURI S., ELLIS K., POLOZOV O., SINGH R., SOLAR-LEZAMA A., YUE Y., ET AL.: Neurosymbolic programming. *Foundations and Trends® in Programming Languages* 7, 3 (2021), 158–243. 5
- [CFG\*15] CHANG A. X., FUNKHOUSER T., GUIBAS L., HANRAHAN P., HUANG Q., LI Z., SAVARESE S., SAVVA M., SONG S., SU H., XIAO J., YI L., YU F.: ShapeNet: An information-rich 3D model repository. *arXiv preprint arXiv:1512.03012* (2015). 10
- [CHIS22] CROITORU F.-A., HONDURU V., IONESCU R. T., SHAH M.: Diffusion models in vision: A survey. *arXiv preprint arXiv:2209.04747* (2022). 3
- [Cla76] CLARK J. H.: Hierarchical geometric models for visible surface algorithms. *Communications of the ACM* 19, 10 (1976), 547–554. 1
- [Cla03] CLAUSEN J.: *Branch and Bound Algorithms-Principles and Examples*. Tech. rep., University of Copenhagen, 2003. 7
- [Coo84] COOK R. L.: Shade trees. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (1984), p. 223–231. 4
- [CRW\*20] CHAUDHURI S., RITCHIE D., WU J., XU K., ZHANG H.: Learning generative models of 3D structures. *Computer Graphics Forum (CGF)* 39, 2 (2020), 643–666. 3
- [CSQ\*22] CASCAVAL D., SHALAH M., QUINN P., BODIK R., AGRAWALA M., SCHULZ A.: Differentiable 3D CAD programs for bidirectional editing. *Computer Graphics Forum (CGF)* 41, 2 (2022), 309–323. 18
- [CZ19] CHEN Z., ZHANG H.: Learning implicit fields for generative shape modeling. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019). 2, 3
- [DAD\*18] DESCHAINTRE V., AITTALA M., DURAND F., DRETTAKIS G., BOUSSEAU A.: Single-image SVBRDF capture with a rendering-aware deep network. *ACM Transactions on Graphics (TOG)* 37, 128 (2018), 15. 2
- [Das] DASSAULT SYSTEMES: SOLIDWORKS. <https://www.solidworks.com/>. Accessed: 2022-10-16. 4
- [dB78] DE BOOR C.: A practical guide to splines. In *Applied Mathematical Sciences* (1978). 1
- [DDFG01] DOUCET A., DE FREITAS N., GORDON N. (Eds.): *Sequential Monte Carlo Methods in Practice*. Springer, 2001. 7
- [DIP\*18] DU T., INALA J. P., PU Y., SPIELBERG A., SCHULZ A., RUS D., SOLAR-LEZAMA A., MATUSIK W.: InverseCSG: Automatic conversion of 3D models to CSG trees. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–16. 5
- [DKD\*22] DENG B., KULAL S., DONG Z., DENG C., TIAN Y., WU J.: Unsupervised learning of shape programs with repeatable implicit parts. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022). 7, 13, 14
- [DNJ20] DAVIES T., NOWROUZEZAHRAI D., JACOBSON A.: On the effectiveness of weight-encoded neural implicit 3D shapes. *arXiv preprint arXiv:2009.09808* (2020). 3
- [Edw63] EDWARD S. I.: *SketchPad: A man-machine graphical communication system*. PhD thesis, Massachusetts Institute of Technology, 1963. 2, 3, 11
- [ENP\*19] ELLIS K., NYE M., PU Y., SOSA F., TENENBAUM J. B., SOLAR-LEZAMA A.: Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems (NeurIPS)* (2019). 3, 11, 12, 13, 14
- [ERSLT18] ELLIS K., RITCHIE D., SOLAR-LEZAMA A., TENENBAUM J.: Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems (NeurIPS)* (2018), vol. 31. 1, 3, 11, 12
- [Esr] ESRI: ArcGIS CityEngine. <https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview>. Accessed: 2022-09-26. 2
- [EWN\*21] ELLIS K., WONG C., NYE M., SABLÉ-MEYER M., MORALES L., HEWITT L., CARY L., SOLAR-LEZAMA A., TENENBAUM J. B.: DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)* (2021), pp. 835–850. 7, 11, 12
- [FAW19] FRÜHSTÜCK A., ALHASHIM I., WONKA P.: TileGAN: Synthesis of large-scale non-homogeneous textures. *ACM Transactions on Graphics (TOG)* 38, 4 (2019). 2
- [FBA22] FIROZE A., BENES B., ALIAGA D.: Urban tree generator: spatio-temporal and generative deep learning for urban tree localization and modeling. *The Visual Computer* 38 (06 2022), 1–13. doi: 10.1007/s00371-022-02526-x. 4
- [FKS\*04] FUNKHOUSER T., KAZHDAN M., SHILANE P., MIN P., KIEFER W., TAL A., RUSINKIEWICZ S., DOBKIN D.: Modeling by example. *ACM Transactions on Graphics (TOG)* 23, 3 (2004), 652–663. 18
- [Fow10] FOWLER M.: *Domain-specific languages*. Pearson Education, 2010. 6
- [GAD\*20] GUEHL P., ALLEGRE R., DISCHLER J.-M., BENES B., GALIN E.: Semi-procedural textures using point process texture basis functions. *Computer Graphics Forum (CGF)* 39, 4 (2020), 159–171. 16, 17
- [GBL\*21] GANIN Y., BARTUNOV S., LI Y., KELLER E., SALICETI S.: Computer-aided design as language. In *Advances in Neural Information Processing Systems (NeurIPS)* (2021). 6, 9, 11, 12
- [GEB15] GATYS L. A., ECKER A. S., BETHGE M.: A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576* (2015). 2
- [Gha08] GHALI S.: *Constructive solid geometry*. Springer, 2008, pp. 277–283. 3



- [GHS\*22] GUERRERO P., HASAN M., SUNKAVALLI K., MECH R., BOUBEKEUR T., MITRA N.: MatFormer: A generative model for procedural materials. *ACM Transactions on Graphics (TOG)* 41, 4 (2022). 1, 6, 10, 16, 17, 18
- [GJB\*20] GUO J., JIANG H., BENES B., DEUSSEN O., ZHANG X., LISCHINSKI D., HUANG H.: Inverse procedural modeling of branching structures by inferring l-systems. *ACM Transactions on Graphics (TOG)* 39, 5 (2020), 1–13. 11, 12
- [GKB\*18] GANIN Y., KULKARNI T., BABUSCHKIN I., ESLAMI S. A., VINIYALS O.: Synthesizing programs for images using reinforced adversarial learning. In *International Conference on Machine Learning (ICML)* (2018), PMLR, pp. 1666–1675. 11, 12
- [GKG\*22] GAILLARD M., KRS V., GORI G., MÈCH R., BENES B.: Automatic differentiable procedural modeling. *Computer Graphics Forum* 41, 2 (2022), 289–307. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14475>, arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14475>, doi:<https://doi.org/10.1111/cgf.14475>. 18
- [GLLD12] GALERNE B., LAGAE A., LEFEBVRE S., DRETTAKIS G.: Gabor noise by example. *ACM Transactions on Graphics (TOG)* 31, 4 (2012). 16
- [GLP\*22] GUO H., LIU S., PAN H., LIU Y., TONG X., GUO B.: Complexgen: Cad reconstruction by b-rep chain complex generation. *ACM Trans. Graph. (SIGGRAPH)* 41, 4 (July 2022). URL: <https://doi.org/10.1145/3528223.3530078>, doi:10.1145/3528223.3530078. 3, 4
- [GPS\*17] GULWANI S., POLOZOV O., SINGH R., ET AL.: Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. 5
- [GSH\*20] GUO Y., SMITH C., HAŠAN M., SUNKAVALLI K., ZHAO S.: MaterialGAN: Reflectance capture using a generative SVBRDF model. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 254:1–254:13. 3
- [GSLT\*18] GOTTSCHLICH J., SOLAR-LEZAMA A., TATBUL N., CARBIN M., RINARD M., BARZILAY R., AMARASINGHE S., TENENBAUM J. B., MATTSON T.: The three pillars of machine programming. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)* (2018), pp. 69–80. 5
- [GSR\*17] GILMER J., SCHOENHOLZ S. S., RILEY P. F., VINIYALS O., DAHL G. E.: Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML)* (2017), p. 1263–1272. 8
- [Gul11] GULWANI S.: Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)* (2011). 5
- [HDMR21] HENZLER P., DESCHAIANTRE V., MITRA N. J., RITSCHER T.: Generative modelling of BRDF textures from flash images. *ACM Transactions on Graphics (TOG)* 40, 6 (2021). 2
- [HDR19] HU Y., DORSEY J., RUSHMEIER H.: A novel framework for inverse procedural texture modeling. *ACM Transactions on Graphics (TOG)* 38, 6 (2019). 16
- [HE17] HA D., ECK D.: A neural representation of sketch drawings. *arXiv preprint arXiv:1704.03477* (2017). 2, 9
- [HGH\*22] HU Y., GUERRERO P., HASAN M., RUSHMEIER H., DESCHAIANTRE V.: Node graph optimization using differentiable proxies. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (2022). 16, 17
- [HHD\*22] HU Y., HE C., DESCHAIANTRE V., DORSEY J., RUSHMEIER H.: An inverse procedural modeling pipeline for SVBRDF maps. *ACM Transactions on Graphics (TOG)* 41, 2 (2022). 7, 16, 17
- [HLB19] HAN X.-F., LAGA H., BENNAMOUN M.: Image-based 3D object reconstruction: State-of-the-art and trends in the deep learning era. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 43, 5 (2019), 1578–1604. 3
- [HLC19] HEMPEL B., LUBIN J., CHUGH R.: Sketch-n-Sketch: Output-directed programming for SVG. In *ACM Symposium on User Interface Software and Technology (UIST)* (2019), pp. 281–292. 5, 18
- [HLHF22] HUI K.-H., LI R., HU J., FU C.-W.: Neural wavelet-domain diffusion for 3D shape generation. In *Annual Conference on Computer Graphics and Interactive Techniques Asia (SIGGRAPH Asia)* (2022), pp. 1–9. 2, 6
- [Hof89] HOFFMANN C. M.: *Geometric and solid modeling*. CUMIN-CAD, 1989. 4
- [Ico] Icons8. <https://icons8.com/>. Accessed: 2022-20-20. 9
- [IDV] IDV, INC.: SpeedTree – 3D vegetation modeling and middleware. <https://store.speedtree.com/>. Accessed: 2022-09-26. 2
- [IK16] ISHII Y., KUTSUNA T.: Effective fault localization using dynamic slicing and an smt solver. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2016), pp. 180–188. 18
- [Ini] INIGO QUILEZ AND POL JEREMIAS: Shadertoy. <https://www.shadertoy.com/>. Accessed: 2022-10-16. 2, 17
- [IZZE17] ISOLA P., ZHU J.-Y., ZHOU T., EFROS A. A.: Image-to-image translation with conditional adversarial networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2017). 2
- [JBX\*20] JONES R. K., BARTON T., XU X., WANG K., JIANG E., GUERRERO P., MITRA N. J., RITCHIE D.: ShapeAssembly: Learning to generate programs for 3D shape structure synthesis. *ACM Transactions on Graphics (TOG)* 39, 6 (2020). 2, 14, 15
- [JCG\*21] JONES R. K., CHARATAN D., GUERRERO P., MITRA N. J., RITCHIE D.: ShapeMOD: Macro operation discovery for 3D shape programs. *ACM Transactions on Graphics (TOG)* 40, 4 (2021). 7, 14, 15
- [JWR22] JONES R. K., WALKER H., RITCHIE D.: PLAD: Learning to infer shape programs with pseudo-labels and approximate distributions. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022). 3, 9, 13, 14, 18
- [Kel21] KELLY T.: CityEngine: An introduction to rule-based modeling. In *Urban Informatics*. Springer, 2021, pp. 637–662. 4
- [KLA21] KARRAS T., LAINE S., AILA T.: A style-based generator architecture for generative adversarial networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 43, 12 (2021), 4217–4228. 2, 3
- [KMJ\*19] KOCH S., MATVEEV A., JIANG Z., WILLIAMS F., ARTEMOV A., BURNAEV E., ALEXA M., ZORIN D., PANOZZO D.: ABC: A big CAD model dataset for geometric deep learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019). 10
- [Koz92] KOZA J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. 7
- [KZK20] KANIA K., ZIEBA M., KAJDANOWICZ T.: UCSG-NET - unsupervised discovering of constructive solid geometry tree. In *Advances in Neural Information Processing Systems (NeurIPS)* (2020), vol. 33, pp. 8776–8786. 8, 14, 15
- [LBH15] LECUN Y., BENGIO Y., HINTON G.: Deep learning. *Nature* 521, 7553 (2015), 436–444. 3
- [LCC\*22] LI Y., CHOI D., CHUNG J., KUSHMAN N., SCHRITTWIESER J., LEBLOND R., ECCLES T., KEELING J., GIMENO F., LAGO A. D., ET AL.: Competition-level code generation with AlphaCode. *arXiv preprint arXiv:2203.07814* (2022). 5
- [LDHM16] LIU A. J., DONG Z., HAŠAN M., MARSCHNER S.: Simulating the structure and texture of solid wood. *ACM Transactions on Graphics (TOG)* 35, 6 (2016). 16
- [LGB\*21] LIU Y., GUO J., BENES B., DEUSSEN O., ZHANG X., HUANG H.: Treepartnet: Neural decomposition of point clouds for 3d tree reconstruction. *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA)* 40, 6 (2021), 232:1–232:16. 4

- [LGD\*18] LIU J., GAN Y., DONG J., QI L., SUN X., JIAN M., WANG L., YU H.: Perception-driven procedural texture generation from examples. *Neurocomputing* 291 (2018), 21–34. 16
- [LHES19] LOPES R. G., HA D., ECK D., SHLENS J.: A learned representation for scalable vector graphics. In *IEEE/CVF International Conference on Computer Vision (ICCV)* (2019). 9, 11, 12
- [LKK\*21] LI B., KALUŻNY J., KLEIN J., MICHELS D. L., PAŁUBICKI W., BENES B., PIRK S.: Learning to reconstruct botanical trees from single images. *ACM Trans. Graph.* 40, 6 (dec 2021). URL: <https://doi.org/10.1145/3478513.3480525>. doi:10.1145/3478513.3480525. 4
- [LLHF21] LI R., LI X., HUI K.-H., FU C.-W.: SP-GAN: Sphere-guided 3D shape generation and manipulation. *ACM Transactions on Graphics (TOG)* 40, 4 (2021). 2, 6
- [LP00] LEFEBVRE L., POULIN P.: Analysis and synthesis of structural textures. In *Graphics Interface* (2000), vol. 2000, pp. 77–86. 16
- [LPBM20] LI C., PAN H., BOUSSEAU A., MITRA N. J.: Sketch2CAD: Sequential CAD modeling by sketching in context. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 164:1–164:14. 13, 14
- [LPBM22] LI C., PAN H., BOUSSEAU A., MITRA N. J.: Free2CAD: Parsing freehand drawings into CAD commands. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 93:1–93:16. 13, 14
- [LST15] LAKE B. M., SALAKHUTDINOV R., TENENBAUM J. B.: Human-level concept learning through probabilistic program induction. *Science* 350, 6266 (2015), 1332–1338. 9
- [LST19] LAKE B. M., SALAKHUTDINOV R., TENENBAUM J. B.: The Omniglot challenge: a 3-year progress report. *Current Opinion in Behavioral Sciences* 29 (2019), 97–104. 9
- [LWJ\*22] LAMBOURNE J. G., WILLIS K. D. D., JAYARAMAN P. K., ZHANG L., SANGHI A., MALEKSHAN K. R.: Reconstructing editable prismatic CAD from rounded voxel models. *arXiv preprint arXiv:2209.01161* (2022). 14
- [LXC\*17] LI J., XU K., CHAUDHURI S., YUMER E., ZHANG H., GUIBAS L.: GRASS: Generative recursive autoencoders for shape structures. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–14. 2
- [LZCvdP20] LING H. Y., ZINNO F., CHENG G., VAN DE PANNE M.: Character controllers using motion VAEs. *ACM Transactions on Graphics (TOG)* 39, 4 (2020). 3
- [Mas] MASSIVE SOFTWARE: Massive Software. <https://www.massivesoftware.com/>. Accessed: 2022-09-26. 2, 17
- [MBBO22] MEZGHANNI M., BODRITO T., BOULKENAFED M., OVSIANIKOV M.: Physical simulation layer for accurate 3d modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2022), pp. 13514–13523. 6
- [MGA\*22] MA K., GHARBI M., ADAMS A., KAMIL S., LI T.-M., BARNES C., RAGAN-KELLEY J.: Searching for fast demosaicking algorithms. *ACM Transactions on Graphics (TOG)* (2022). 5
- [MGY\*19] MO K., GUERRERO P., YI L., SU H., WONKA P., MITRA N., GUIBAS L.: StructureNet: Hierarchical graph networks for 3D shape generation. *ACM Transactions on Graphics (TOG)* 38, 6 (2019). 2, 3
- [Mit77] MITCHELL T. M.: Version spaces: A candidate elimination approach to rule learning. In *International Joint Conference on Artificial Intelligence (IJCAI)* (1977). 7
- [MKG\*18] MITRA N. J., KOKKINOS I., GUERRERO P., THUEREY N., RITSCHER T.: CreativeAI: Deep learning for graphics. In *SIGGRAPH Asia 2018 Courses* (2018). 3
- [MST\*20] MILDENHALL B., SRINIVASAN P. P., TANCIK M., BARRON J. T., RAMAMOORTHY R., NG R.: NeRF: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision (ECCV)* (2020). 2, 3
- [MVG13] MARTINOVIC A., VAN GOOL L.: Bayesian grammar learning for inverse procedural modeling. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2013), pp. 201–208. 4
- [MWH\*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (2006), pp. 614–623. 4
- [MZC\*19] MO K., ZHU S., CHANG A. X., YI L., TRIPATHI S., GUIBAS L. J., SU H.: PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3D object understanding. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019). 9
- [NGEB20] NASH C., GANIN Y., ESLAMI S. M. A., BATTAGLIA P. W.: PolyGen: An autoregressive generative model of 3D meshes. In *International Conference on Machine Learning (ICML)* (2020). 3
- [NO80] NELSON G., OPPEN D. C.: Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)* 27, 2 (1980), 356–364. 5
- [Nvi22] NVIDIA: VMaterials, 2022. <https://developer.nvidia.com/vmaterials>. 10
- [NWA\*20] NANDI C., WILLSEY M., ANDERSON A., WILCOX J. R., DARULOVA E., GROSSMAN D., TATLOCK Z.: Synthesizing structured CAD models with equality saturation and inverse transformations. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)* (2020), pp. 31–44. 5
- [NWP\*18] NANDI C., WILCOX J. R., PANCHEKHA P., BLAU T., GROSSMAN D., TATLOCK Z.: Functional programming for compiling and decompiling computer-aided design. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)* (2018). 5
- [NZIS13] NIESSNER M., ZOLLHÖFER M., IZADI S., STAMMINGER M.: Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (TOG)* (2013). 2
- [Ope] OPENS CAD: OpenSCAD - the programmers solid 3D modeler. <https://openscad.org/>. Accessed: 2022-10-21. 18
- [PBG\*21] PARA W. R., BHAT S. F., GUERRERO P., KELLY T., MITRA N., GUIBAS L., WONKA P.: SketchGen: Generating constrained CAD sketches. In *Advances in Neural Information Processing Systems (NeurIPS)* (2021). 6, 11, 12
- [Pea85] PEACHEY D. R.: Solid texturing of complex surfaces. *ACM Transactions on Graphics (TOG)* 19, 3 (1985), 279–286. 16
- [Per85] PERLIN K.: An image synthesizer. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (1985). 4, 16
- [PGK\*21] PARA W., GUERRERO P., KELLY T., GUIBAS L., WONKA P.: Generative layout modeling using constraint graphs. In *IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), pp. 6670–6680. 11
- [PHHM96] PRUSINKIEWICZ P., HAMMEL M., HANAN J., MĚCH R.: L-systems: From the theory to visual models of plants. In *CSIRO Symposium on Computational Challenges in Life Sciences* (1996). 4
- [PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic topiary. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (1994), pp. 351–358. 4
- [PKG\*21] PASCHALIDOU D., KATHAROPOULOS A., GEIGER A., FIDLER S.: Neural parts: Learning expressive 3D shape abstractions with invertible neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021). 3
- [PL96] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg, 1996. 4
- [PTC] PTC INC.: OnShape. URL: <https://www.onshape.com/>. 9, 10
- [Qui] The Quick, Draw! Dataset. <https://quickdraw.withgoogle.com/data>. Accessed: 2022-20-20. 9
- [RBCP20] RIBEIRO L. S. F., BUI T., COLLOMOSSE J., PONTI M.: Sketchformer: Transformer-based representation for sketched structure. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020). 2

- [RBL\*22] ROMBACH R., BLATTMANN A., LORENZ D., ESSER P., OMMER B.: High-resolution image synthesis with latent diffusion models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022). 2, 18
- [RDN\*22] RAMESH A., DHARIWAL P., NICHOL A., CHU C., CHEN M.: Hierarchical text-conditional image generation with CLIP latents. *arXiv preprint arXiv:2204.06125* (2022). 2, 18
- [Red76] REDDY D.: Speech understanding systems: summary of results of the five-year research effort. *Computer Science, Carnegie-Mellon University, Pittsburgh, PA* (1976). 7
- [RGLM21] REDDY P., GHARBI M., LUKAC M., MITRA N. J.: Im2Vec: Synthesizing vector graphics without vector supervision. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021), pp. 7342–7351. 6, 11, 12
- [RJT18] RITCHIE D., JOBALIA S., THOMAS A.: Example-based authoring of procedural modeling programs with structural and continuous variability. *Computer Graphics Forum (CGF)* 37, 2 (2018), 401–413. 6, 14, 15
- [RVdOV19] RAZAVI A., VAN DEN OORD A., VINYALS O.: Generating diverse high-fidelity images with VQ-VAE-2. In *Advances in Neural Information Processing Systems (NeurIPS)* (2019), vol. 32. 12
- [RZC\*21] REN D., ZHENG J., CAI J., LI J., JIANG H., CAI Z., ZHANG J., PAN L., ZHANG M., ZHAO H., ET AL.: CSG-Stump: A learning friendly CSG-like representation for interpretable shape parsing. In *IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), pp. 12478–12487. 8, 14
- [RZC\*22] REN D., ZHENG J., CAI J., LI J., ZHANG J.: ExtrudeNet: Unsupervised inverse sketch-and-extrude for shape parsing. In *European Conference on Computer Vision (ECCV)* (2022). 14, 15
- [San] SANGALARD F.: Decyphering the business card raytracer. [https://fabiensanglard.net/rayTracing\\_back\\_of\\_business\\_card/](https://fabiensanglard.net/rayTracing_back_of_business_card/). Accessed: 2022-10-19. 18
- [SCS\*22] SAHARIA C., CHAN W., SAXENA S., LI L., WHANG J., DENTON E., GHASEMPOUR S. K. S., AYAN B. K., MAHDAVI S. S., LOPES R. G., ET AL.: Photorealistic text-to-image diffusion models with deep language understanding. *arXiv preprint arXiv:2205.11487* (2022). 2, 18
- [SG71] STINY G., GIPS J.: Shape grammars and the generative specification of painting and sculpture. *Information Processing* (1971). 4
- [SGL\*18] SHARMA G., GOYAL R., LIU D., KALOGERAKIS E., MAJI S.: CSGNet: Neural shape parser for constructive solid geometry. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2018). 13, 14
- [Sid] SIDEFX: Houdini. <https://www.sidefx.com/products/houdini/>. Accessed: 2022-09-26. 2, 15, 16
- [SLH\*20] SHI L., LI B., HASAN M., SUNKAVALLI K., BOUBEKEUR T., MECH R., MATUSIK W.: MATch: Differentiable material graphs for procedural material capture. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–15. 16, 17
- [SLM\*20] SHARMA G., LIU D., MAJI S., KALOGERAKIS E., CHAUDHURI S., MECH R.: Parsenet: A parametric surface fitting network for 3d point clouds. In *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part VII* (2020), Vedaldi A., Bischof H., Brox T., Frahm J., (Eds.), vol. 12352 of *Lecture Notes in Computer Science*, Springer, pp. 261–276. URL: [https://doi.org/10.1007/978-3-030-58571-6\\_16](https://doi.org/10.1007/978-3-030-58571-6_16), doi:10.1007/978-3-030-58571-6\_16. 3
- [SLTB\*06] SOLAR-LEZAMA A., TANCAU L., BODIK R., SESHIA S., SARASWAT V.: Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), pp. 404–415. 4, 5
- [SOZA20] SEFF A., OVADIA Y., ZHOU W., ADAMS R. P.: Sketch-Graphs: A large-scale dataset for modeling relational geometry in computer-aided design. In *International Conference on Machine Learning Workshops (ICML Workshop)* (2020). 9
- [Str06] STROUD I.: *Boundary representation modelling techniques*. Springer Science & Business Media, 2006. 4
- [SWB21] SASAKI H., WILLCOCKS C. G., BRECKON T. P.: UNIT-DDPM: Unpaired image translation with denoising diffusion probabilistic models. *arXiv preprint arXiv:2104.05358* (2021). 9
- [SWD\*17] SCHULMAN J., WOLSKI F., DHARIWAL P., RADFORD A., KLIMOV O.: Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017). 8
- [Sys] SYSTEMS I. M.: Intelligent music systems. <https://www.intelligentmusicsystems.com/>. Accessed: 2022-10-19. 17
- [SZRA22] SEFF A., ZHOU W., RICHARDSON N., ADAMS R. P.: Vitruvion: A generative model of parametric CAD sketches. In *International Conference on Learning Representations (ICLR)* (2022). 6, 11, 12
- [TB13] TORLAK E., BODIK R.: Growing solver-aided languages with rosette. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)* (2013), pp. 135–152. 5
- [TLS\*19] TIAN Y., LUO A., SUN X., ELLIS K., FREEMAN W. T., TENENBAUM J. B., WU J.: Learning to infer and execute 3D shape programs. In *International Conference on Learning Representations (ICLR)* (2019). 3, 13, 14
- [TRT\*22] TCHAPMI L. P., RAY T., TCHAPMI M., SHEN B., MARTIN-MARTIN R., SAVARESE S.: Generating procedural 3D materials from images using neural networks. In *International Conference on Image, Video and Signal Processing (IVSP)* (2022), p. 32–40. 16
- [TSG\*17] TULSIANI S., SU H., GUIBAS L. J., EFROS A. A., MALIK J.: Learning shape abstractions by assembling volumetric primitives. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2017). 3
- [TSTL09] TATE R., STEPP M., TATLOCK Z., LERNER S.: Equality saturation: a new approach to optimization. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)* (2009), pp. 264–276. 7
- [UyCS\*22] UY M. A., YU CHANG Y., SUNG M., GOEL P., LAMBOURNE J., BIRDAL T., GUIBAS L.: Point2Cyl: Reverse engineering 3D objects from point clouds to extrusion cylinders. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022). 14
- [VFJ15] VINYALS O., FORTUNATO M., JAITLEY N.: Pointer networks. In *Advances in Neural Information Processing Systems (NeurIPS)* (2015), vol. 28. 8
- [vLA87] VAN LAARHOVEN P. J. M., AARTS E. H. L.: Simulated annealing: Theory and applications. In *Mathematics and Its Applications* (1987). 7
- [VPB\*22] VINKER Y., PAJOUHESHGAR E., BO J. Y., BACHMANN R. C., BERMANO A. H., COHEN-OR D., ZAMIR A., SHAMIR A.: CLIPasso: Semantically-aware object sketching. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (2022). 2, 3
- [VWM15] VIZEL Y., WEISSENBACHER G., MALIK S.: Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE* 103 (2015), 2021–2035. 7
- [Wil92] WILLIAMS R. J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8 (1992). 8
- [WJC\*22] WILLIS K. D., JAYARAMAN P. K., CHU H., TIAN Y., LI Y., GRANDI D., SANGHI A., TRAN L., LAMBOURNE J. G., SOLAR-LEZAMA A., MATUSIK W.: JoinABLE: Learning bottom-up assembly of parametric CAD joints. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022), pp. 15849–15860. 10
- [WJL\*21] WILLIS K. D. D., JAYARAMAN P. K., LAMBOURNE J. G., CHU H., PU Y.: Engineering sketch generation for computer-aided design. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshop)* (2021). 11, 12



- [WK91] WITKIN A., KASS M.: Reaction-diffusion textures. *ACM Transactions on Graphics (TOG)* 25, 4 (1991), 299–308. [16](#)
- [WL21] WANG Y., LIAN Z.: DeepVecFont: Synthesizing high-quality vector fonts via dual-modality learning. *ACM Transactions on Graphics (TOG)* 40, 6 (2021). [11](#), [12](#)
- [WLW\*19] WANG K., LIN Y.-A., WEISSMANN B., SAVVA M., CHANG A. X., RITCHIE D.: PlanIT: Planning and instantiating indoor scenes with relation graph and spatial prior networks. *ACM Transactions on Graphics (TOG)* 38, 4 (2019). [11](#)
- [WMG\*22] WONG C., MCCARTHY W. P., GRAND G., FRIEDMAN Y., TENENBAUM J. B., ANDREAS J., HAWKINS R. D., FAN J. E.: Identifying concept libraries from language about object structure. In *Annual Meeting of the Cognitive Science Society (CogSci)* (2022). [9](#)
- [WNW\*21] WILLSEY M., NANDI C., WANG Y. R., FLATT O., TATLOCK Z., PANCHEKHA P.: egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (Jan. 2021). URL: <https://doi.org/10.1145/3434304>, doi:10.1145/3434304. [5](#)
- [Wor96] WORLEY S.: A cellular texture basis function. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (1996), p. 291–294. [4](#), [16](#)
- [WPL\*21] WILLIS K. D. D., PU Y., LUO J., CHU H., DU T., LAMBOURNE J. G., SOLAR-LEZAMA A., MATUSIK W.: Fusion 360 Gallery: A dataset and environment for programmatic CAD construction from human design sequences. *ACM Transactions on Graphics (TOG)* 40, 4 (2021). [4](#), [6](#), [10](#), [13](#), [14](#)
- [WSCR18] WANG K., SAVVA M., CHANG A. X., RITCHIE D.: Deep convolutional priors for indoor scene synthesis. *ACM Transactions on Graphics (TOG)* 37, 4 (2018). [3](#)
- [WXZ21] WU R., XIAO C., ZHENG C.: DeepCAD: A deep generative network for computer-aided design models. In *IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), pp. 6772–6782. [6](#), [14](#), [15](#)
- [WZN\*19] WU C., ZHAO H., NANDI C., LIPTON J. I., TATLOCK Z., SCHULZ A.: Carpentry compiler. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–14. [5](#)
- [WZX\*16] WU J., ZHANG C., XUE T., FREEMAN W. T., TENENBAUM J. B.: Learning a probabilistic latent space of object shapes via 3D generative-adversarial modeling. In *Advances in Neural Information Processing Systems (NeurIPS)* (2016), pp. 82–90. [2](#), [3](#), [6](#)
- [XPC\*21] XU X., PENG W., CHENG C.-Y., WILLIS K. D. D., RITCHIE D.: Inferring CAD modeling sequences using zone graphs. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021). [14](#)
- [XTS\*22] XIE Y., TAKIKAWA T., SAITO S., LITANY O., YAN S., KHAN N., TOMBARI F., TOMPKIN J., SITZMANN V., SRIDHAR S.: Neural fields in visual computing and beyond. *Computer Graphics Forum (CGF)* (2022). [3](#)
- [XWL\*22] XU X., WILLIS K. D., LAMBOURNE J. G., CHENG C.-Y., JAYARAMAN P. K., FURUKAWA Y.: SkexGen: Autoregressive generation of CAD construction sequences with disentangled codebooks. In *International Conference on Machine Learning (ICML)* (2022). [1](#), [11](#), [12](#), [14](#), [15](#)
- [YCL\*22] YU F., CHEN Z., LI M., SANGHI A., SHAYANI H., MAHDAVI-AMIRI A., ZHANG H.: CAPRI-Net: Learning compact CAD shapes with adaptive primitive assembly. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022), pp. 11768–11778. [14](#)
- [YLM\*22] YAN X., LIN L., MITRA N. J., LISCHINSKI D., COHEN-OR D., HUANG H.: ShapeFormer: Transformer-based shape completion via sparse representation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022), pp. 6239–6249. [3](#)
- [YP22a] YANG Y., PAN H.: Discovering design concepts for cad sketches. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022). [11](#)
- [YP22b] YANG Y., PAN H.: Discovering design concepts for cad sketches. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022). [12](#)
- [ZPIE17] ZHU J.-Y., PARK T., ISOLA P., EFROS A. A.: Unpaired image-to-image translation using cycle-consistent adversarial networks. In *IEEE/CVF International Conference on Computer Vision (ICCV)* (2017). [2](#), [9](#)
- [ZVW\*22] ZENG X., VAHDAT A., WILLIAMS F., GOJCIC Z., LITANY O., FIDLER S., KREIS K.: LION: Latent point diffusion models for 3D shape generation. In *Advances in Neural Information Processing Systems (NeurIPS)* (2022). [2](#)
- [ZWZ\*21] ZHAO H., WILLSEY M., ZHU A., NANDI C., TATLOCK Z., SOLOMON J., SCHULZ A.: Co-optimization of design and fabrication plans for carpentry. *arXiv preprint arXiv:2107.12265* (2021). [5](#)