PolyGen: An Autoregressive Generative Model of 3D Meshes

Charlie Nash [1] **Yaroslav Ganin** [1] **S. M. Ali Eslami** [1] **Peter W. Battaglia** [1]

Abstract

Polygon meshes are an efficient representation of 3D geometry, and are of central importance in computer graphics, robotics and games development. Existing learning-based approaches have avoided the challenges of working with 3D meshes, instead using alternative object representations that are more compatible with neural architectures and training approaches. We present an approach which models the mesh directly, predicting mesh vertices and faces sequentially using a Transformer-based architecture. Our model can condition on a range of inputs, including object classes, voxels, and images, and because the model is probabilistic it can produce samples that capture uncertainty in ambiguous scenarios. We show that the model is capable of producing high-quality, usable meshes, and establish log-likelihood benchmarks for the mesh-modelling task. We also evaluate the conditional models on surface reconstruction metrics against alternative methods, and demonstrate competitive performance despite not training directly on this task.

1. Introduction

Polygon meshes are an efficient representation of 3D geometry, and are widely used in computer graphics to represent virtual objects and scenes. Automatic mesh generation enables more rapid creation of the 3D objects that populate virtual worlds in games, film, and virtual reality. In addition, meshes are a useful output in computer vision and robotics, enabling planning and interaction in 3D space.

Existing approaches to 3D object synthesis rely on the recombination and deformation of template models (Kalogerakis et al., 2012; Chaudhuri et al., 2011), or a parametric shape family (Smelik et al., 2014). Meshes are challenging for deep learning architectures to work with because of their unordered elements and discrete face structures. Instead, recent deep-learning approaches have gener-
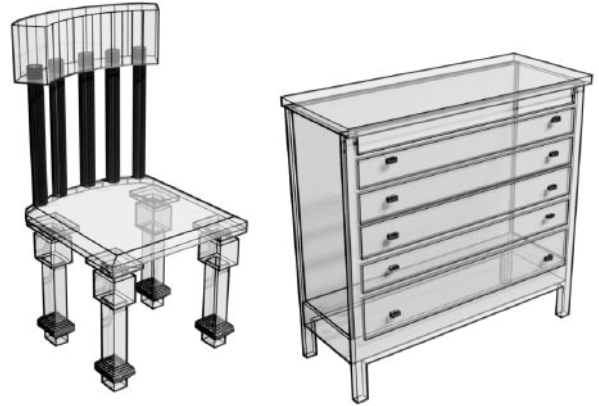
[1]DeepMind, London, United Kingdom. Correspondence to: Charlie Nash <charlienash@google.com>.

Figure 1. Class conditional n-gon meshes generated by PolyGen.

ated 3D objects using alternative representations of object shape—voxels (Choy et al., 2016), pointclouds, occupancy functions (Mescheder et al., 2019), and surfaces (Groueix et al., 2018)—however mesh reconstruction is left as a post-processing step and can yield results of varying quality. This contrasts with the human approach to mesh creation, where the mesh itself is the central object, and is created directly with 3D modelling software. Human created meshes are compact, and reuse geometric primitives to efficiently represent real-world objects. Neural autoregressive models have demonstrated a remarkable capacity to model complex, high-dimensional data including images (van den Oord et al., 2016c), text (Radford et al., 2019) and raw audio waveforms (van den Oord et al., 2016a). Inspired by these methods we present PolyGen, a neural generative model of meshes, that autoregressively estimates a joint distribution over mesh vertices and faces.

PolyGen consists of two parts: A vertex model, that unconditionally models mesh vertices, and a face model, that models the mesh faces conditioned on input vertices. Both components make use of the Transformer architecture (Vaswani et al., 2017), which is effective at capturing the long-range dependencies present in mesh data. The vertex model uses a masked Transformer decoder to express a distribution over the vertex sequences. For the face model we combine Transformers with pointer networks (Vinyals et al., 2015) to express a distribution over variable length vertex sequences.
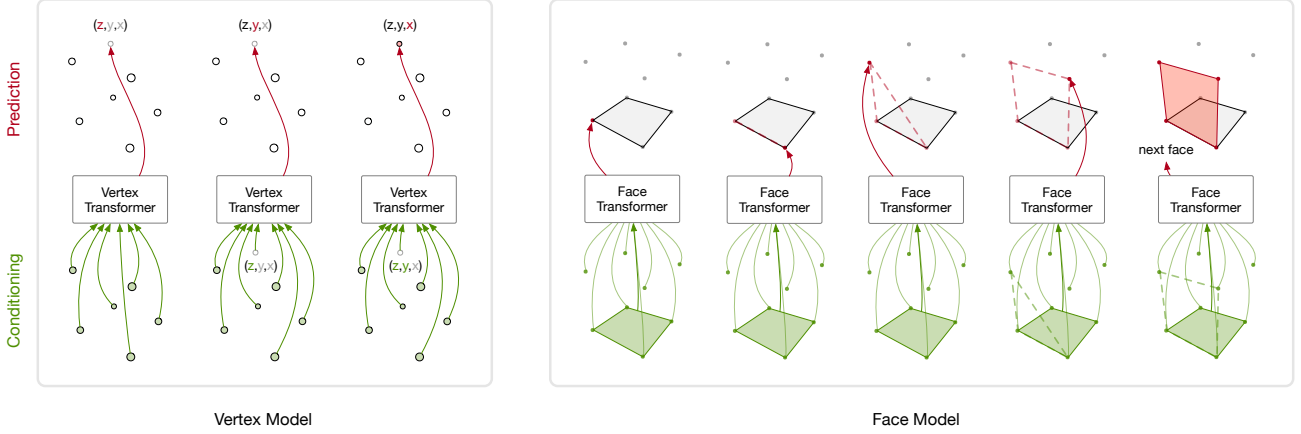
Figure 2. PolyGen first generates mesh vertices (left), and then generates mesh faces conditioned on those vertices (right). Vertices are generated sequentially from lowest to highest on the vertical axis. To generate the next vertex the current sequence of vertex coordinates is passed as context to a vertex Transformer, which outputs a predictive distribution for the next vertex coordinate. The face model takes as input a collection of vertices, and the current sequence of face indices, and outputs a distribution over vertex indices.

We evaluate the modelling capacity of PolyGen using log-likelihood and predictive accuracy as metrics, and compare statistics of generated samples to real data. We demonstrate conditional mesh generation with object class, images and voxels as input and compare to existing mesh generation methods. Overall, we find that our model is capable of creating diverse and realistic geometry that is directly usable in graphics applications.

2. PolyGen

Our goal is to estimate a distribution over meshes $\mathcal{M}$ from which we can generate new examples. A mesh is a collection of 3D vertices $\mathcal{V}$, and polygon faces F, that define the shape of a 3D object. We split the modelling task into two parts: i) Generating mesh vertices $\mathcal{V}$, and ii) generating mesh faces $\mathcal{F}$ given vertices. Using the chain rule we have:

$$p(\mathcal{M}) = p(\mathcal{V}, \mathcal{F}) \qquad (1)$$
$$= p(\mathcal{F}|\mathcal{V})p(\mathcal{V}) \qquad (2)$$

We use separate vertex and face models, both of which are autoregressive; factoring the joint distribution over vertices and faces into a product of conditional distributions. To generate a mesh we first sample the vertex model, and then pass the resulting vertices as input to the face model, from which we sample faces (see Figure 2). In addition, we optionally condition both the vertex and face models on a context $\mathbf{h}$, such as the mesh class identity, an input image, or a voxelized shape.

2.1. n-gon Meshes

3D meshes typically consist of collections of triangles, but many meshes can be more compactly represented using
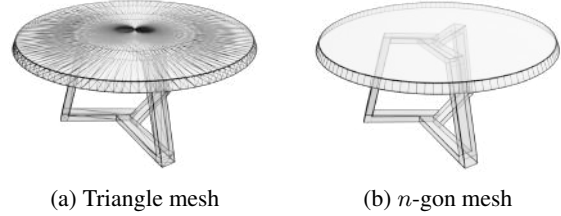

(a) Triangle mesh   (b) n-gon mesh

Figure 3. Triangle meshes consist entirely of triangles. n-gon meshes efficiently represent shapes using variable size polygons.

polygons of variable sizes. Meshes with variable length polygons are called n-gon meshes:

$$\mathcal{F}_{\text{tri}} = \left\{ \left(f_1^{(i)}, f_2^{(i)}, f_3^{(i)} \right) \right\}_i \qquad (3)$$
$$\mathcal{F}_{n\text{-gon}} = \left\{ \left(f_1^{(i)}, f_2^{(i)}, \dots, f_{N_i}^{(i)} \right) \right\}_i \qquad (4)$$

where N_i is the number of faces in the i-th polygon and can vary for different faces. This means that large flat surfaces can be represented with a single polygon e.g. the top of the circular table in Figure 3. In this work we opt to represent meshes using n-gons rather than triangles. This has two main advantages: The first is that it reduces the size of meshes, as flat surfaces can be specified with a reduced number of faces. Secondly, large polygons can be triangulated in many ways, and these triangulations can be inconsistent across examples. By modelling n-gons we factor out this triangulation variability.

A caveat to this approach is that n-gons do not uniquely define a 3D surface when n is greater than 3, unless the vertices it references are planar. When rendering non-planar n-gons, polygons are first triangulated by e.g. projecting vertices to a plane (Held, 2001), which can cause artifacts

if the polygon is highly non-planar. In practice we find that most of the n-gons produced by our model are either planar, or close to planar, such that this is a minor issue. Triangle meshes are a subset of n-gon meshes, and PolyGen can therefore be used to model them if required.

2.2. Vertex Model

The goal of the vertex model is to express a distribution over sequences of vertices. We order the vertices from lowest to highest by z-coordinate, where z represents the vertical axis. If there are vertices with the same z-value, we order by y and then by x value. After re-ordering, we obtain a flattened sequence by concatenating tuples of $(z_i, y_i, x_i)_i$ coordinates. Meshes have variable numbers of vertices, so we use a stopping token s to indicate the end of the vertex sequence. We denote the flattened vertex sequence $\mathcal{V}^{\text{seq}}$ and its elements as $v_n, n = 1, \ldots, N_V$. We decompose the joint distribution over $\mathcal{V}^{\text{seq}}$ as the product of a series of conditional vertex distributions:

$$p(\mathcal{V}^{\text{seq}}; \theta) = \prod_{n=1}^{N_V} p(v_n | v_{<n}; \theta) \tag{5}$$

We model this distribution using an autoregressive network that outputs at each step the parameters of a predictive distribution for the next vertex coordinate. This predictive distribution is defined over the vertex coordinate values as well as over the stopping token s. The model is trained to maximize the log-probability of the observed data with respect to the model parameters θ.

Architecture. The basis of the vertex model architecture is a Transformer decoder (Vaswani et al., 2017), a simple and expressive model that has demonstrated significant modeling capacity in a range of domains (Child et al., 2019; Huang et al., 2019; Parmar et al., 2018). Mesh vertices have strong non-local dependencies, with object symmetries and repeating parts, and the Transformer's ability to aggregate information from any part of the input enables it to capture these dependencies. We use the improved Transformer variant with layer normalization inside the residual path, as in (Child et al., 2019; Parisotto et al., 2019). See Figure 12 in the appendix for an illustration of the vertex model and appendix C for a full description of the Transformer blocks.

Vertices as discrete variables. We apply 8-bit uniform quantization to the mesh vertices. This reduces the size of meshes as nearby vertices that fall into the same bin are merged. We model the quantized vertex values using a Categorical distribution, and output at each step the logits of the distribution. This approach has been used to model discretized continuous signals in PixelCNN (van den Oord et al., 2016c), and WaveNet (van den Oord et al., 2016a), and has the benefit of being able to express distributions without
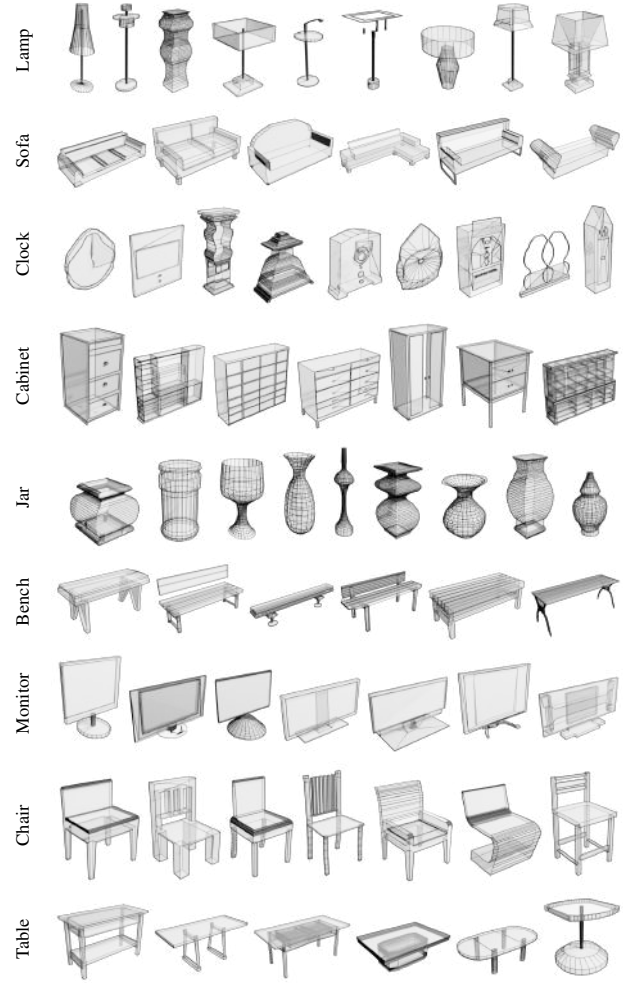

Figure 4. Class conditional samples generated by PolyGen using nucleus sampling and top-$p = 0.9$.

shape limitations. Mesh vertices have strong symmetries and complex dependencies, so the ability to express arbitrary distributions is important. We find 8-bit quantization to be a good trade-off between mesh fidelity, and mesh size. However, it should be noted that 14-bits or higher is typical for lossy mesh compression, and in future work it would be desirable to extend our methods to higher resolution meshes.

Embeddings. We found the approach of using learned position and value embedding methods proposed in (Child et al., 2019) to work well. We use three embeddings for each input token: A coordinate embedding, that indicates whether the input token is an x, y, or z coordinate, a position embedding, that indicates which vertex in the sequence the token belongs to, and a value embedding, which expresses a token's quantized coordinate value. We use learned discrete embeddings in each case.
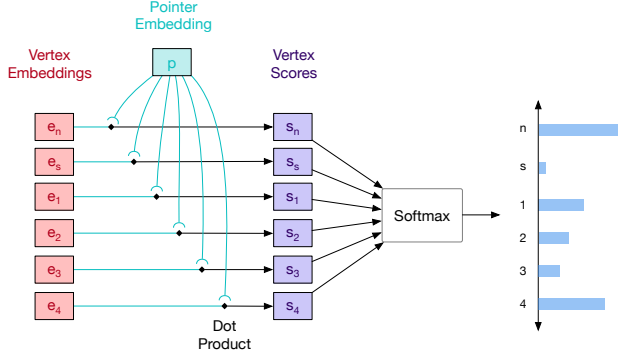
Figure 5. The mesh pointer network produces a distribution over variable length vertex sequences by comparing an output pointer embedding to vertex embeddings. In this example the number of vertices under consideration $N_V = 4$ and therefore the distribution is over 6 elements.

Improving efficiency. One of the downsides of using Transformers for modelling sequential data is that they incur significant computational costs due to the quadratic nature of the attention operation. This presents issues when it comes to scaling our models to larger meshes. To address this, we explored several modifications of the model inspired by (Salimans et al., 2017). All of them relieve the computational burden by chunking the sequence into triplets of vertex coordinates and processing each of them at once. The first variant uses a mixture of discretized logistics to model whole 3D vertices. The second replaces the mixture with a MADE-based decoder (Germain et al., 2015). Finally, we present variants that use a Transformer decoder but rely on different vertex embedding schemes. These modifications are described in more detail in appendix E.

2.3. Face Model

The face model expresses a distribution over a sequence of mesh faces conditioned on the mesh vertices. We order the faces by their lowest vertex index, then by their next lowest vertex and so on, where the vertices have been ordered from lowest to highest as described in Section 2.2. Within a face we cyclically permute the face indices so that the lowest index is first. As with the vertex sequences, we concatenate the faces $(f_1^{(i)}, f_2^{(i)}, \ldots, f_{N_i}^{(i)})_i$ to form a flattened sequence, with a final stopping token. We write $\mathcal{F}^{\text{seq}}$ for this flattened sequence, with elements $f_n, n = 1, \ldots, N_F$.

$$p(\mathcal{F}^{\text{seq}}|\mathcal{V};\theta) = \prod_{n=1}^{N_F} p(f_n|f_{<n}, \mathcal{V};\theta) \qquad (6)$$

As with the vertex model, we output a distribution over the values of f at each step, and train by maximizing the log-likelihood of θ over the training set. The distribution is a categorical defined over $\{1, \ldots, N_V + 2\}$ where N_V is

the number of input vertices, and we include two additional values for the end-face n and stopping s tokens.

Mesh pointer networks. The target distribution $p(f_n|f_{<n}, \mathcal{V};\theta)$ is defined over the indices of an input set of vertices, which poses the challenge that the size of this set varies across examples. Pointer networks (Vinyals et al., 2015) propose an elegant solution to this issue; Firstly the input set is embedded using an encoder, and then at each step an autoregressive network outputs a pointer vector that is compared to the input embeddings via a dot-product. The resulting scores are then normalized using a softmax to form a valid distribution over the input set.

In our case we obtain contextual embeddings $\mathbf{e}_v$ of the input vertices using a Transformer encoder E. This has the advantage of bi-directional information aggregation compared to the LSTM used by the original pointer networks. We jointly embed new-face and stopping tokens with the vertices, to obtain a total of $N_V + 2$ input embeddings. A Transformer decoder D operates on the sequence of faces and outputs pointers $\mathbf{p}_k$ at each step. The target distribution can be obtained as

$$\{\mathbf{e}_v\}_{v=1}^{N_V} = E(\mathcal{V};\theta) \qquad (7)$$
$$\mathbf{p}_n = D(f_{<n}, \mathcal{V};\theta) \qquad (8)$$
$$p(f_n = k \mid f_{<n}, \mathcal{V};\theta) = \text{softmax}_k(\mathbf{p}_n^T \mathbf{e}_k) \qquad (9)$$

See Figure 5 for an illustration of the pointer mechanism and Figure 13 in the appendix for an illustration of the whole face model. The decoder D is a masked Transformer decoder that operates on sequences of embedded face tokens. It conditions on the input vertices in two ways, via dynamic face embeddings as explained in the next section, and optionally through cross-attention into the sequence of vertex embeddings.

Embeddings. As with the vertex model we use learned position and value embeddings. We decompose a token's position into the index of the face it belongs to, as well as the location of a token within a face, using separate learned embeddings for both. For value embeddings we follow the approach of pointer networks and simply embed the vertex indices by indexing into the contextual vertex embeddings outputted by the vertex encoder.

2.4. Masking Invalid Predictions

For both the vertex and face model only certain predictions are valid at each step. For instance, the z-coordinates must increase monotonically, and the stopping token can only be placed after an x coordinate. Similarly mesh faces can not have duplicate indices, and every vertex-index must be referenced by at least one face. When evaluating the model we mask the predicted logits to ensure that the model can only make valid predictions. This has a non-negative
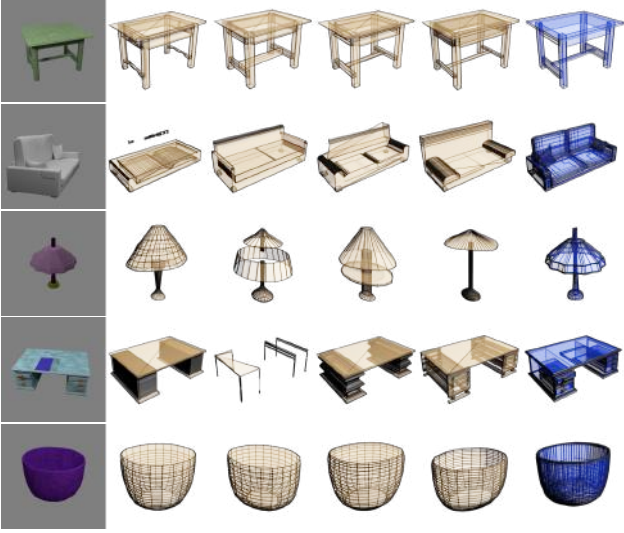
Figure 6. Image conditional samples (yellow) generated using nucleus sampling with top-p=0.9 and ground truth meshes (blue).

Table 1. Modelling performance of unconditional models trained on ShapeNet and baseline methods. Negative log-likelihood is reported in bits per vertex, averaged across test examples. Accuracy refers to the classification accuracy of next step predictions: discrete vertex coordinates for the vertex model, or vertex indices for face models. *Draco is evaluated on triangulated meshes rather than n-gon meshes.

Model	Bits per vertex		Accuracy	
	Vertices	Faces	Vertices	Faces
Uniform	24.08	39.73	0.004	0.002
Valid predictions	21.41	25.79	0.009	0.038
Draco* (Google)	Total: 27.68		-	-
PolyGen	2.46	1.79	0.851	0.900
- valid predictions	2.47	1.82	0.851	0.900
- discr. embed. (V)	2.56	-	0.844	-
- data augmentation	3.39	2.52	0.803	0.868
+ cross attention (F)	-	1.87	-	0.899

Table 2. Comparison of vertex model variants. The first two columns correspond to the test negative log-likelihood and predictive accuracy (see Table 1). The last column shows training speed in steps per second.

Model	Bits per vertex	Accuracy	Steps per sec
Mixture	3.01	-	7.19
MADE decoder	2.65	0.844	7.02
Tr. decoder	2.50	0.851	4.07
+ Tr. embed.	2.48	0.851	4.60
Base model	2.46	0.851	2.98

effect on the model's log-likelihood scores, as it reassigns probability mass in the invalid region to values in the valid region (Table 1). Surprisingly, we found that masking during training to worsen performance to a small degree, so we always train without masking. For a complete description of the masks used, see appendix F.

2.5. Conditional Mesh Generation

We can guide the generation of mesh vertices and faces by conditioning on a context. For instance, we can output vertices consistent with a given object class, or infer the mesh associated with an input image. It is straightforward to extend the vertex and face models to condition on a context **h**. We incorporate context in two ways, depending on the domain of the input. For global features like class identity, we project learned class embeddings to a vector that is added to the intermediate Transformer representations following the self-attention layer in each block. For high dimensional inputs like images, or voxels, we jointly train a domain-appropriate encoder that outputs a sequence of context embeddings. The Transformer decoder then performs cross-attention into the embedding sequence, as in the original machine translation Transformer model.

For image inputs we use an encoder consisting of a series of downsampling residual blocks. We use pre-activation residual blocks (He et al., 2016), and downsample three times using convolutions with stride 2, taking input images of size $[256, 256, 3]$ to feature maps of size $[16, 16, E]$ where E is the embedding dimensionality of the model. For voxel inputs we use a similar encoder but with 3D convolutions that takes inputs of shape $[28, 28, 28, 1]$ to spatial embeddings of shape $[7, 7, 7, E]$. For both input types we add coordinate

embeddings to the feature maps before flattening the spatial dimensions. For more architecture details see appendix C.

3. Experiments

Our primary evaluation metric is log-likelihood, which we find to correlate well with sample quality. We also report summary statistics for generated meshes, and compare our model to existing approaches using chamfer-distance in the image and voxel conditioned settings.

3.1. Training Details

We train all our models on the ShapeNet Core V2 dataset (Chang et al., 2015), which we subdivide into 92.5% training, 2.5% validation and 5% testing splits. The training set is augmented as described in Section 3.2. In order to reduce the memory requirements of long sequences we filter out meshes with more than 800 vertices, or more than 2800 face indices after pre-processing. We train the vertex and face

models for $1e6$ and $5e5$ weight updates respectively, using four V100 GPUs per training run for a total batch size of 16. We use the Adam optimizer with a gradient clipping norm of 1.0, and perform cosine annealing from a maximum learning rate of $3e-4$, with a linear warm up period of 5000 steps. We use a dropout rate of 0.2 for all models.

3.2. Data Augmentation and Rendering

In general we observed significant overfitting due to the relatively small size of the ShapeNet dataset, which is exacerbated by the need to filter out large meshes. In order to reduce this effect, we augmented the input meshes by scaling the vertices independently on each axis, using a random piecewise-linear warp for each axis, and by varying the decimation angle used to create n-gon meshes. For each input mesh we create 50 augmented versions which are then quantized (Section 2.2) for use during training. We found that augmentation was necessary to obtain good performance (Table 1). For full details of the augmentations and parameter settings see appendix A.

Rendering. In order to train image-conditional models we create renders of the processed ShapeNet meshes using Blender (Blender Online Community). For each augmented mesh, and each validation and test-set mesh, we create renders at 256×256 resolution, using randomly chosen lighting, camera and mesh material settings. For more details see appendix B.

3.3. Unconditional Modelling Performance

We compare unconditional models trained under varying conditions. As evaluation metrics we report the negative log-likelihood obtained by the models, reported in bits per vertex, as well as the accuracy of next step predictions. For vertex models this is the accuracy of next vertex coordinate predictions, and for face models this is the accuracy of the next vertex index predictions. In particular we compare the effect of masking invalid predictions (Section 2.4), of using discrete rather than continuous coordinate embeddings in the vertex model (Section 2.2), of using data augmentation (Section 3.2), and finally of using cross-attention in the face model. Unless otherwise specified we use embeddings of size 256, fully connected layers of size 1024, and 18 and 12 Transformer blocks for the vertex and face models respectively. As there are no existing methods that directly model mesh vertices and faces, we report the scores obtained by models that allocate uniform probability to the whole data domain, as well as models that are uniform over the region of valid predictions. We additionally report the compression rate obtained by Draco (Google), a mesh compression library. For details of the Draco compression settings see appendix G.
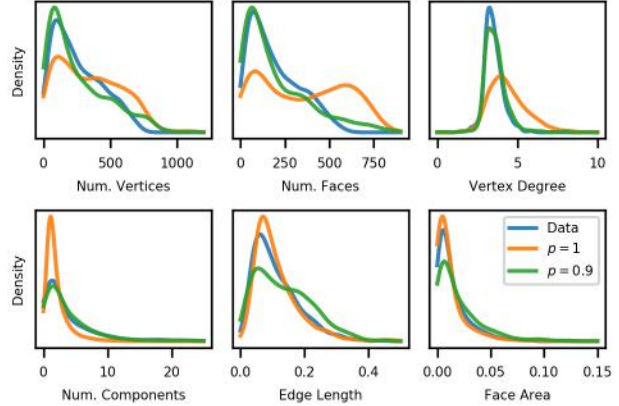
Table 1 shows the results obtained by the various models.

Figure 7. Distribution of mesh statistics for unconditional samples from our model and the ShapeNet test set. We compare samples generated using with nucleus sampling and top-$p = 0.9$, to true model samples ($p = 1$).

We find that our models achieve significantly better modelling performance than the uniform and Draco baselines, which illustrates the gains achievable by a learned predictive model. We find that restricting the models predictions to the range of valid values results in a minor improvement in modelling performance, which indicates that the model is effective at assigning low probability to the invalid regions. Using discrete rather than continuous embeddings for vertex coordinates provides a significant improvement, improving bits-per-vertex from 2.56 to 2.46. Surprisingly, using cross-attention in the face model harms performance, which we attribute to overfitting. Data augmentation has a strong effect on performance, with models trained without augmentation losing 1.64 bits per vertex on average. Overall, our best model achieves a log-likelihood score of 4.26 bits per vertex, and 85% and 90% predictive accuracy for the vertex and face models respectively. Figure 14 in the appendix shows random unconditional samples from the best performing model.

Table 2 presents a comparison of different variants of the vertex model as discussed in Section 2.2. The results suggest that the proposed variants can achieve a $1.5\times$ reduction in training time with a minimal sacrifice in performance. Note that these models used different hyperparameter settings as detailed in Appendix E.

3.4. Statistics of Unconditional Model Samples

We compare the distribution of certain mesh summaries for samples from our model against the ShapeNet test set. If our model has closely matched the true data distribution then we expect these summaries to have similar distributions. We draw 1055 samples from our best unconditional model, and discard samples that don't produce a stopping token

Table 3. Modelling performance for conditional models. See Table 1 for details of bits per vertex, and accuracy scores.

Context	Bits per vertex			Accuracy	
	Vertices	Faces	Total	Vertices	Faces
None	2.46	1.79	4.26	0.851	0.900
Class	2.43	1.81	4.24	0.853	0.899
Image	2.30	1.81	4.11	0.857	0.900
+ pooling	2.35	1.78	4.13	0.856	0.900
Voxels	2.19	1.82	4.01	0.859	0.900
+ pooling	2.28	1.79	4.07	0.856	0.900

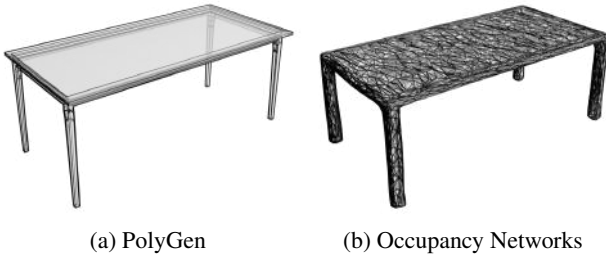(a) PolyGen   (b) Occupancy Networks

Figure 8. Comparison between a 3D mesh generated by PolyGen and a mesh obtained by postprocessing an implicit surface representation (Occupancy Networks Mescheder et al., 2019). Our model produces a more efficient representation of the 3D shape that resembles a human-constructed mesh.

within 1200 vertices, or 800 faces. We use nucleus sampling (Holtzman et al., 2019) which we found to be effective at maintaining sample diversity while reducing the presence of degraded samples. Nucleus sampling helps to reduce sampling degradation by sampling from the smallest subset of tokens that account for top-p of probability mass.

Figure 7 shows the distribution of a number of mesh summaries, for samples from PolyGen as well as the true data distribution. In particular we show: the number of vertices, number of faces, node degree, average face area and average edge length for sampled and true meshes. Although these are coarse descriptions of a 3D mesh, we find our model's samples to have a similar distribution for each mesh statistic. We observe that nucleus sampling with top-$p = 0.9$ helps to align the model distributions with the true data for a number of statistics. Figure 8 shows an example 3D mesh generated by our model compared to a mesh obtained through post-processing an occupancy function (Mescheder et al., 2019). We note that the statistics of our mesh resemble human-created meshes to a greater extent.

3.5. Conditional Modelling Performance

We train vertex and face models with three kinds of conditioning: class labels, images, and voxels. We use the same
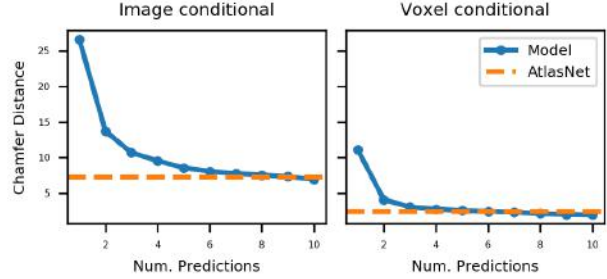

Figure 9. Symmetric chamfer distance between predicted and target pointclouds by number of predictions. Data refers to pointclouds obtained by uniformly re-sampling the target mesh.

Figure 10. Voxel conditional (blue, left) samples generated using nucleus sampling with top-p=0.9 (yellow) and ground truth meshes (blue, right).

settings as the best unconditional model: discrete vertex embeddings with no cross attention in the face model. As with the unconditional models we use 18 layers for the vertex model and 12 layers for the face model. Figures 1 and 4 show class-conditional samples. Figures 6 and 10 show samples from image and voxel conditional models respectively. Note that while we train on the ShapeNet dataset, we show ground truth meshes and inputs for a selection of representative meshes collected from the TurboSquid online object repository. Table 3 shows the impact of conditioning on predictive performance in terms of bits-per-vertex and accuracy. We find that for vertex models, voxel conditioning provides the greatest improvement, followed by images, and then by class labels. This confirms our expectations, as voxels characterize the coarse shape unambiguously, whereas images can be ambiguous depending on the object pose and lighting. However the additional context does not lead to improvements for the face model, with all conditional

face models performing slightly worse than the best unconditional model. This is likely because mesh faces are to a large extent determined by the input vertices, and the conditioning context provides relatively little additional information. In terms of predictive accuracy, we see similar effects, with accuracy improving with richer contexts for vertex models, but not for face models. We note that the accuracy ceiling is less than 100%, due to the inherent entropy of the vertex and face distributions, and so we expect diminishing gains as models approach this ceiling.

For image and voxel conditional models, we also compare to architectures that apply global average pooling to the outputs of the input encoders. We observe that pooling in this way negatively effects the vertex models' performance, but has a small positive effect on the face models' performance.

3.6. Mesh Reconstruction

We additionally evaluate the image and voxel conditioned models on mesh reconstruction, where we use symmetric chamfer distance as the reconstruction metric. The symmetric chamfer distance is a distance metric between two point sets $\mathcal{P}$ and $\mathcal{Q}$. It is defined as:

$$\mathcal{L}(\mathcal{P}, \mathcal{Q}) = \sum_{\mathbf{p} \in \mathcal{P}} \min_{\mathbf{q} \in \mathcal{Q}} (\mathbf{p} - \mathbf{q})^2 + \sum_{\mathbf{q} \in \mathcal{Q}} \min_{\mathbf{p} \in \mathcal{P}} (\mathbf{p} - \mathbf{q})^2 \quad (10)$$

For each example in the test set we draw samples from the conditional model. We sample 2500 points uniformly on the sampled and target mesh and compute the corresponding chamfer distance. We compare our model to AtlasNet (Groueix et al., 2018), a conditional model that defines a mesh surface using a number of patches that have been Transformer using a deep network. AtlasNet outputs pointclouds and is trained to minimize the chamfer distance to a target pointcloud conditioned on image or pointcloud inputs. Compared to alternative methods, AtlasNet achieves good mesh reconstruction performance, and we therefore view it as a strong baseline. We train AtlasNets models in the image and voxel conditioned settings, that are adapted to use equivalent image and voxel encoders as we use for our model. For more details see appendix D.

Figure 9 shows the mesh reconstruction results. We find that when making a single prediction, our model performs worse than AtlasNet. This is not unexpected, as AtlasNet optimizes the evaluation metric directly, whereas our model does not. When allowed to make 10 predictions, our model achieves slightly better performance than AtlasNet. Overall we find that while our model does not always produce good mesh reconstructions, it typically produces a very good reconstruction within 10 samples, which may be sufficient for many practical applications.

4. Related Work

Generative models of 3D objects exists in a variety of forms, including ordered (Nash & Williams, 2017) and unordered (Li et al., 2019; Yang et al., 2019) pointclouds, voxels (Choy et al., 2016; Wu et al., 2016; Tatarchenko et al., 2017; Rezende et al., 2016). More recently there has been significant progress using functional representations, such as signed distance functions (Park et al., 2019), and other implicit functions (Mescheder et al., 2019). There are relatively fewer examples of methods that explicitly generate a 3D mesh. Such works primarily use parameterized deformable meshes (Groueix et al., 2018), or form meshes through a collection of mesh patches. Our methods are distinguished in that we directly model the mesh data created by people, rather than alternative representations or parameterizations. In addition, our model is probabilistic, which means we can produce diverse output, and respond to ambiguous inputs in a principled way.

PolyGen's vertex model is similar to PointGrow (Sun et al., 2020), which uses an autoregressive decomposition to model 3D point clouds, outputting discrete coordinate distributions using a self-attention based architecture. PointGrow operates on fixed-length point-clouds rather than variable vertex sequences, and uses a bespoke self-attention architecture, that is relatively shallow in comparison to modern autoregressive models in other domains. By contrast, we use state-of-the-art deep architectures, and model vertices and faces, enabling us to generate high quality 3D meshes.

This work borrows from architectures developed for sequence modelling in natural language processing. This includes the sequence to sequence training paradigm (Sutskever et al., 2014), the Transformer architecture (Vaswani et al., 2017; Child et al., 2019; Parisotto et al., 2019), and pointer networks (Vinyals et al., 2015). In addition our work is inspired by sequential models of raw data, like WaveNet (van den Oord et al., 2016a) PixelRNN and its variants (van den Oord et al., 2016b; Menick & Kalchbrenner, 2019), and Music Transformers (Huang et al., 2019).

Our work is also related to Polygon-RNN (Castrejón et al., 2017; Acuna et al., 2018), a method for efficient segmentation in computer vision using polygons. Polygon-RNN take an input image and autoregressively outputs a sequence of xy coordinates that implicitly define a segmented region. PolyGen, by contrast operates in 3D space, and explicitly defines the connectivity of several polygons.

Finally our work is related to generative models of graph structured data such as GraphRNN (You et al., 2018) and GRAN (Liao et al., 2019), in that meshes can be thought of as attributed graphs. These works focus on modelling graph connectivity rather than graph attributes, whereas we model both the node attributes (vertex positions), as well

as the incorporating these attributes in our model of the connectivity.

5. Conclusion

In this work we present PolyGen, a deep generative model of 3D meshes. We pose the problem of mesh generative as autoregressive sequence modelling, and combine the benefits of Transformers and pointer networks in order to flexibly model variable length mesh sequences. PolyGen is capable of generating coherent and diverse mesh samples, and we believe that it will unlock a range of applications in computer vision, robotics, and 3D content creation.

Acknowledgements

References

Acuna, D., Ling, H., Kar, A., and Fidler, S. Efficient interactive annotation of segmentation datasets with polygon-rnn++. In *CVPR*, pp. 859–868. IEEE Computer Society, 2018.

Blender Online Community. Blender - a 3d modelling and rendering package. URL http://www.blender.org.

Castrejón, L., Kundu, K., Urtasun, R., and Fidler, S. Annotating object instances with a polygon-rnn. In *CVPR*, pp. 4485–4493. IEEE Computer Society, 2017.

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., and Yu, F. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

Chaudhuri, S., Kalogerakis, E., Guibas, L. J., and Koltun, V. Probabilistic reasoning for assembly-based 3d modeling. *ACM Trans. Graph.*, 30(4):35, 2011.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019.

Choy, C. B., Xu, D., Gwak, J., Chen, K., and Savarese, S. 3d-r2n2: A unified approach for single and multiview 3d object reconstruction. In *ECCV (8)*, volume 9912 of *Lecture Notes in Computer Science*, pp. 628–644. Springer, 2016.

Germain, M., Gregor, K., Murray, I., and Larochelle, H. Made: Masked autoencoder for distribution estimation. In *ICML*, 2015.

Google. Draco compression library. https://github.com/google/draco. Accessed: 2020-02-04.

Groueix, T., Fisher, M., Kim, V. G., Russell, B. C., and Aubry, M. A papier-mâché approach to learning 3d surface generation. In *CVPR*, pp. 216–224. IEEE Computer Society, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In *ECCV (4)*, volume 9908 of *Lecture Notes in Computer Science*, pp. 630–645. Springer, 2016.

Held, M. FIST: fast industrial-strength triangulation of polygons. *Algorithmica*, 30(4):563–596, 2001.

Holtzman, A., Buys, J., Forbes, M., and Choi, Y. The curious case of neural text degeneration. *CoRR*, abs/1904.09751, 2019.

Huang, C. A., Vaswani, A., Uszkoreit, J., Simon, I., Hawthorne, C., Shazeer, N., Dai, A. M., Hoffman, M. D., Dinculescu, M., and Eck, D. Music transformer: Generating music with long-term structure. In *ICLR (Poster)*. OpenReview.net, 2019.

Kalogerakis, E., Chaudhuri, S., Koller, D., and Koltun, V. A probabilistic model for component-based shape synthesis. *ACM Trans. Graph.*, 31(4):55:1–55:11, 2012.

Li, C., Zaheer, M., Zhang, Y., Póczos, B., and Salakhutdinov, R. Point cloud GAN. In *DGS@ICLR*. OpenReview.net, 2019.

Liao, R., Li, Y., Song, Y., Wang, S., Nash, C., Hamilton, W. L., Duvenaud, D., Urtasun, R., and Zemel, R. S. Efficient graph generation with graph recurrent attention networks. *CoRR*, abs/1910.00760, 2019.

Menick, J. and Kalchbrenner, N. Generating high fidelity images with subscale pixel networks and multidimensional upscaling. In *ICLR*. OpenReview.net, 2019.

Mescheder, L. M., Oechsle, M., Niemeyer, M., Nowozin, S., and Geiger, A. Occupancy networks: Learning 3d reconstruction in function space. In *CVPR*, pp. 4460–4470. Computer Vision Foundation / IEEE, 2019.

Nash, C. and Williams, C. K. I. The shape variational autoencoder: A deep generative model of part-segmented 3d objects. *Comput. Graph. Forum*, 36(5):1–12, 2017.

Parisotto, E., Song, H. F., Rae, J. W., Pascanu, R., Gülçehre, Ç., Jayakumar, S. M., Jaderberg, M., Kaufman, R. L., Clark, A., Noury, S., Botvinick, M. M., Heess, N., and

Hadsell, R. Stabilizing transformers for reinforcement learning. *CoRR*, abs/1910.06764, 2019.

Park, J. J., Florence, P., Straub, J., Newcombe, R. A., and Lovegrove, S. Deepsdf: Learning continuous signed distance functions for shape representation. In *CVPR*, pp. 165–174. Computer Vision Foundation / IEEE, 2019.

Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, L., Shazeer, N., Ku, A., and Tran, D. Image transformer. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4052–4061. PMLR, 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8): 9, 2019. URL https://openai.com/blog/better-language-models/.

Rezende, D. J., Eslami, S. M. A., Mohamed, S., Battaglia, P. W., Jaderberg, M., and Heess, N. Unsupervised learning of 3d structure from images. In *NIPS*, pp. 4997–5005, 2016.

Salimans, T., Karpathy, A., Chen, X., and Kingma, D. P. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

Smelik, R. M., Tutenel, T., Bidarra, R., and Benes, B. A survey on procedural modelling for virtual worlds. *Comput. Graph. Forum*, 33(6):31–50, 2014.

Sun, Y., Wang, Y., Liu, Z., Siegel, J. E., and Sarma, S. E. Pointgrow: Autoregressively learned point cloud generation with self-attention. In *Winter Conference on Applications of Computer Vision*, 2020.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *NIPS*, pp. 3104–3112, 2014.

Tatarchenko, M., Dosovitskiy, A., and Brox, T. Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs. In *ICCV*, pp. 2107–2115. IEEE Computer Society, 2017.

TurboSquid. TurboSquid. https://www.turbosquid.com/.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. W., and Kavukcuoglu, K. Wavenet: A generative model for raw audio. In *SSW*, pp. 125. ISCA, 2016a.

van den Oord, A., Kalchbrenner, N., Espeholt, L., Kavukcuoglu, K., Vinyals, O., and Graves, A. Conditional image generation with pixelcnn decoders. In *NIPS*, pp. 4790–4798, 2016b.

van den Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. Pixel recurrent neural networks. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pp. 1747–1756. JMLR.org, 2016c.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *NIPS*, pp. 5998–6008, 2017.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In *NIPS*, pp. 2692–2700, 2015.

Wu, J., Zhang, C., Xue, T., Freeman, B., and Tenenbaum, J. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *NIPS*, pp. 82–90, 2016.

Yang, G., Huang, X., Hao, Z., Liu, M., Belongie, S. J., and Hariharan, B. Pointflow: 3d point cloud generation with continuous normalizing flows. *CoRR*, abs/1906.12320, 2019.

You, J., Ying, R., Ren, X., Hamilton, W. L., and Leskovec, J. Graphrnn: Generating realistic graphs with deep autoregressive models. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5694–5703. PMLR, 2018.

Appendix

Figure 11. Examples of data augmentation and randomized rendering conditions. For each input mesh we create 50 augmentations, and render each while varying lighting, camera and material properties.

A. Data Augmentation

For each input mesh from the ShapeNet dataset we create 50 augmented versions which are used during training (Figure 11). We start by normalizing the meshes such that the length of the long diagonal of the mesh bounding box is equal to 1. We then apply the following augmentations, performing the same bounding box normalization after each. All augmentations and mesh rendering are performed prior to vertex quantization.

Axis scaling. We scale each axis independently, uniformly sampling scaling factors s_x, s_y and s_z in the interval $[0.75, 1.25]$.

Piecewise linear warping. We define a continuous, piecewise linear warping function by dividing the interval $[0, 1]$ into 5 even sub-intervals, sampling gradients $g_1, \ldots, g_5$ for each sub-interval from a log-normal distribution with variance 0.5, and composing the segments. For x and y coordinates, we ensure the warping function is symmetric about zero, by reflecting a warping function with three sub-intervals on $[0, 0.5]$ about 0.5. This preserves symmetries in the data which are often present for these axes.

Planar mesh decimation. We use Blender's planar decimation modifier (https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/decimate.html) to create n-gon meshes. This merges adjacent faces where the angle between surfaces is greater than a certain tolerance. Different tolerances result in meshes of different sizes with differing connectivity due to varying levels of decimation. We use this property for data augmentation and sample the tolerance degrees uniformly from the interval $[1, 20]$.

B. Rendering

We use Blender to create rendered images of the 3D meshes in order to train image-conditional models (Figure 11). We use Blender's Cycles (https://docs.blender.org/manual/en/latest/render/cycles/index.html) path-tracing renderer, and randomize the lighting, camera, and mesh materials. In all scenes we place the input meshes at the origin, scaled so that bounding boxes are 1m on the long diagonal.

Lighting. We use an 20W area light located 1.5m above the origin, with rectangle size 2.5m, and sample a number of 15W point lights uniformly from the range $[0, 1, \ldots, 10]$. We choose the location of each point light independently, sampling the x and y coordinates uniformly in the intervals $[-2, -0.75] \cup [0.75, 2]$, and sampling the z coordinate uniformly in the interval $[0.75, 2]$.

Camera. We position the camera at a distance d from the center of the mesh, where d is sampled uniformly from $[1.25, 1.5]$, at an elevation sampled between $[0, 1]$, and sample a rotation uniformly between $[0, 360]$. We sample a focal length for the camera in $[35, 36, \ldots, 50]$. We also sample a filter size (https://docs.blender.org/manual/en/latest/render/cycles/render_settings/film.html) in $[1.5, 2]$, which adds a small degree of blur.

Object materials. We found the ShapeNet materials and textures to be applied inconsistently across different examples when using Blender, and in many cases no textures loaded at all. Rather than use the inconsistent textures, we randomly generated materials for the 3D meshes, in order to produce a degree of visual variability. For each texture group in the mesh we sampled a new material. Materials were constructed by linking Blender nodes (https://docs.blender.org/manual/en/latest/render/shader_nodes/introduction.html#textures). In particular we use a noise shader with detail = 16, scale $= \sqrt{100 * u}, u \sim \mathcal{U}(0, 1)$, and scale draw from the interval $[0, 20]$. The noise shader is used as input to a color ramp node which interpolates between the input color, and white. The color ramp node then sets the color of a diffuse BSDF material https://docs.blender.org/manual/en/latest/render/shader_nodes/shader/diffuse.html, which is applied to faces within a texture group.

C. Transformer blocks

We use the improved Transformer variant with layer normalization moved inside the residual path, as in (Child et al., 2019; Parisotto et al., 2019). In particular we compose the Transformer blocks as follows:

$$\mathbf{R}_{\text{MMH}}^{(l)} = \text{MaskedMultiHead}(\text{LN}(\mathbf{H}_{\text{FC}}^{(l-1)})) \tag{11}$$

$$\mathbf{H}_{\text{MMH}}^{(l)} = \mathbf{H}_{\text{FC}}^{(l-1)} + \mathbf{R}_{\text{MMH}}^{(l)} \tag{12}$$

$$\mathbf{R}_{\text{FC}}^{(l)} = \text{Linear}(\text{ReLU}(\text{Linear}(\text{LN}(\mathbf{H}_{\text{MMH}}^{(l)})))) \tag{13}$$

$$\mathbf{H}_{\text{FC}}^{(l)} = \mathbf{H}_{\text{MMH}}^{(l)} + \mathbf{R}_{\text{FC}}^{(l)} \tag{14}$$
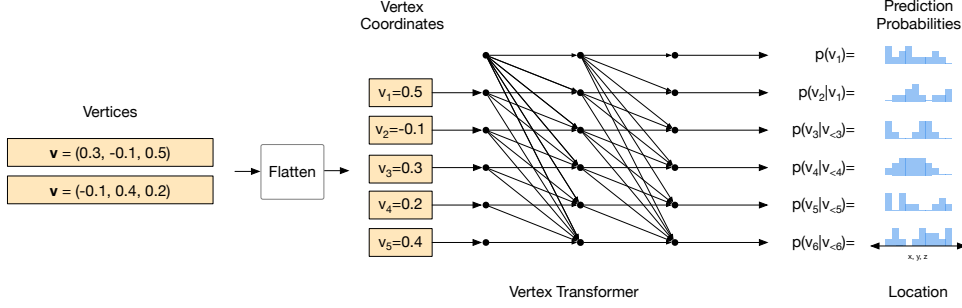
Figure 12. The vertex model is a masked Transformer decoder that takes as input a flattened sequence of vertex coordinates. The Transformer outputs discrete distributions over the individual coordinate locations, as well as the stopping token s. See Section 2.2 for a detailed description of the vertex model.
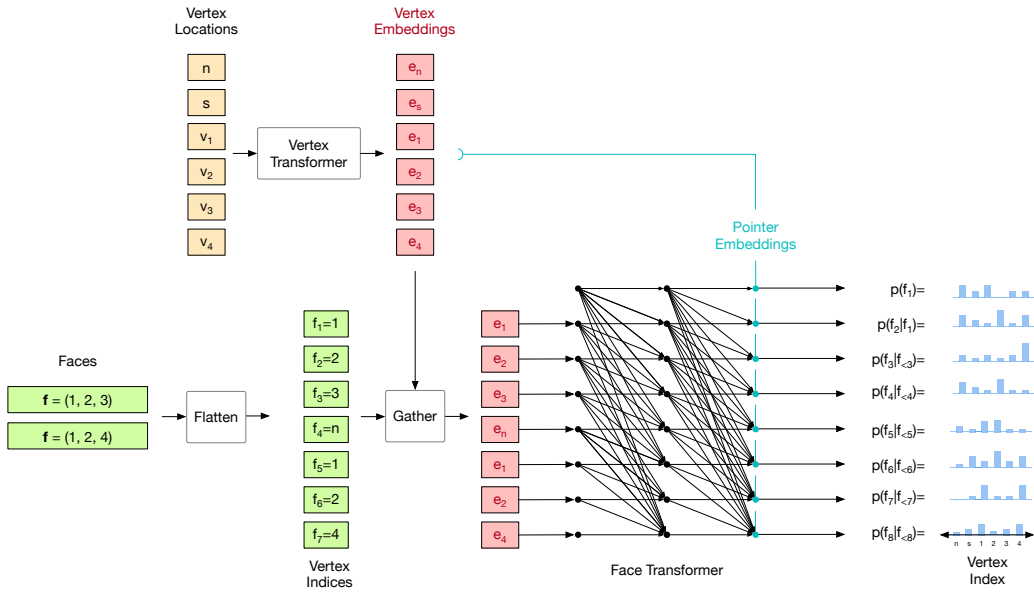

Figure 13. The face model model operates on an input set of vertices, as well as the flattened vertex indices that describe the faces. The vertices as well as the new face token n and stopping token s are first embedded using a Transformer encoder. A gather operation is then used to identify the embeddings associated with each vertex index. The index embeddings are processed with a masked Transformer decoder to output distributions over vertex indices at each step, as well as over the next-face token and the stopping token. The final layer of the Transformer outputs pointer embeddings which are compared to the vertex embeddings using a dot-product to produce the desired distributions. See Section 2.3 for a detailed description of the face model and Figure 5 in particular for a detailed depiction of the pointer network mechanism.

Where $\mathbf{R}^{(l)}$ and $\mathbf{H}^{(l)}$ are residuals and intermediate representations in the l'th block, and the subscripts FC and MMH denote the outputs of fully connected and masked multi-head self-attention layers respectively. We apply dropout immediately following the ReLU activation as this performed well in initial experiments.

Conditional models. As described in Section 2.5 For global features like class identity, we project learned class embeddings to a vector that is added to the intermediate Transformer representations $\mathbf{H}_{\text{MMH}}$ following the self-

attention layer in each block:

$$\mathbf{r}_{\text{global}}^{(l)} = \text{Linear}\left(\mathbf{h}_{\text{global}}\right) \quad (15)$$

$$\mathbf{H}_{\text{global}}^{(l)} = \mathbf{H}_{\text{MMH}}^{(l)} + \text{Broadcast}\left(\mathbf{r}_{\text{global}}^{(l)}\right) \quad (16)$$

For high dimensional inputs like images, or voxels, we jointly train a domain-appropriate encoder that outputs a sequence of context embeddings. The Transformer decoder performs cross-attention into the embedding sequence after the self-attention layer, as in the original machine translation

Transformer model:

$$\mathbf{R}_{\text{seq}}^{(l)} = \text{CrossMultiHead}\left(\mathbf{H}_{\text{MMH}}^{(l)}, \mathbf{H}_{\text{seq}}\right) \quad (17)$$

$$\mathbf{H}_{\text{seq}}^{(l)} = \mathbf{H}_{\text{MMH}}^{(l)} + \mathbf{R}_{\text{seq}}^{(l)} \quad (18)$$

The image and voxel encoders are both pre-activation resnets, with 2D and 3D convolutions respectively. The full architectures are described in Table 4.

D. AtlasNet

We use the same image and voxel-encoders (Table 4) as for the conditional PolyGen models. For consistency with the original method, we project the final feature maps to 1024 dimensions, before applying global average pooling to obtain a vector shape representation. As in the original method, the decoder is an MLP with 4 fully-connected layers of size 1024, 512, 256, 128 with ReLU non-linearities on the first three layers and tanh on the final output layer. The decoder takes the shape representation, as well as 2D points as input, and outputs a 3D vector. We use 25 patches, and train with the same optimization settings as PolyGen (Section 3) but for $5e5$ steps.

Chamfer distance. To evaluate the chamfer distance for AtlasNet models, we first generate a mesh by passing 2D triangulated meshes through each of the AtlasNet patch models as described in (Groueix et al., 2018). We then sample points on the resulting 3D mesh.

E. Alternative Vertex Models

In this section, we provide more details for the more efficient vertex model variants mentioned in Section 2.2.

In the first variant, instead of processing x, y and z coordinates in sequence we concatenate their embeddings together and pass them through a linear projection. This forms the input sequence for a 22-layer Transformer which we call *the torso*. Following (Salimans et al., 2017) we output the parameters of a mixture of 40 discretized logistics describing the joint distribution of a full 3D vertex. The main benefit of this model is that the self-attention is now performed for sequences which are 3 times shorter. This manifests in a much improved training time (see 2). Unfortunately, the speed-up comes at a price of significantly reduced performance. This may be because the underlying continuous components are not well suited to the peaky and multi-modal vertex distributions.

In the second variant we lift the parametric distribution assumption and use a MADE-style masked MLP (Germain et al., 2015) with 2 residual blocks to decode each output of a 18-layer torso h_n into a sequence of three conditional discrete distributions:

$$p(v_n|h_n) = p(z_n|h_n)p(y_n|z_n, h_n)p(x_n|z_n, y_n, h_n) \quad (19)$$

As expected, this change improves the test data likelihood while simultaneously increasing the computation cost. We notice that unlike the base model the MADE decoder has direct access only to the coordinate components within a single vertex and must rely on the output of the torso to learn about the components of previously generated vertices.

We let the decoder attend to all the generated coordinates directly in the third alternative version of our model. We replace the MADE decoder with a 6-layer Transformer which is conditioned on $\{h_n\}_n$ (this time produced by a 14-layer torso) and operates on a flattened sequence of vertex components (similarly to the base model). The conditioning is done by adding h_n to the embeddings of z_n, y_n and x_n. While slower than the MADE version, the resulting network is significantly closer in performance to the base model.

Finally, we make the model even more powerful using a 2-layer Transformer instead of simple concatenation to embed each triplet of vertex coordinates. Specifically, we sum-pool the outputs of that Transformer within every vertex. In this variant, we reduce the depth of the torso to 10 layers. This results in test likelihood similar to the that of the base model.

F. Masking Invalid Predictions

As mentioned in Section 2.2 we mask invalid predictions when evaluating our models. We identify a number of hard constraints that exist in the data, and mask the model's predictions that violate these constraints. The masked probability mass is uniformly distributed across the remaining valid values. We use the following masks:

Vertex model.

- The stopping token can only occur after an x-coordinate:

$$v_k = s \implies v_k \bmod 3 = 1 \quad (20)$$

- z-coordinates are non-decreasing:

$$z_k \geq z_{k-1} \quad (21)$$

- y-coordinates are non-decreasing if their associated z-coordinates are equal:

$$y_k \geq y_{k-1} \text{ if } z_k = z_{k-1} \quad (22)$$

- x-coordinates are increasing if their associated z and y-coordinates are equal:

$$x_k > x_{k-1} \text{ if } y_k = y_{k-1} \text{ and } z_k = z_{k-1} \quad (23)$$

Face model.

- New face tokens n can not be repeated:

$$f_k \neq n \quad \text{if} \quad f_{k-1} = n \qquad (24)$$

- The first vertex index of a new face is not less than the first index in the previous face:

$$f_1^{(k)} \geq f_1^{(k-1)}, \quad k = 1, \ldots, N_f \qquad (25)$$

- Vertex indices within a face are greater than the first index in that face:

$$f_j^{(k)} > f_1^{(k)} \qquad (26)$$

- Vertex indices within a face are unique:

$$f_i^{(k)} \neq f_j^{(k)}, \quad \forall i, j \qquad (27)$$

- The first index of a new face is not greater than the lowest unreferenced vertex index:

$$f_1^{(k)} \leq \min \left[\{v : v \leq N_V\} \setminus \{f_1^{(j)}, \ldots, f_{N_j}^{(j)}\}_{j=1}^{k-1} \right] \qquad (28)$$

G. Draco Compression Settings

We compare our model in Table 1 to Draco (Google), a performant 3D mesh compression library created by Google. We use the highest compression setting, quantize the positions to 8 bits, and do not quantize in order to compare with the 8-bit mesh representations that our model operates on. Note that the quantization performed by Draco is not identical to our uniform quantization, so the reported scores are not directly comparable. Instead they serve as a ballpark estimate of the degree of compression obtained by existing methods.

H. Unconditional Samples

Figure 14 shows a random batch of unconditional samples generated using PolyGen with nucleus sampling ant top-$p = 0.9$. The figure highlights . Firstly, the model learns to mostly output objects consistent with a shape class. Secondly, the samples contain a large proportion of certain object classes, including tables, chairs and sofas. This reflects the significant class-imbalance of the ShapeNet dataset, with many classes being underrepresented. Finally, certain failure modes are present in the collection. These include meshes with disconnected components, meshes that have produced the stopping token too early, producing incomplete objects, and meshes that don't have a distinct form that is recognizable as one of the shape classes.

layer name	output size	layer parameters
conv1	128×128×64	7×7, 64, stride 2
conv2_x	64×64×64	3×3 max pool, stride 2
		$\begin{bmatrix} 3\times3,\ 64 \\ 3\times3,\ 64 \end{bmatrix}\times1$
conv3_x	32×32×128	$\begin{bmatrix} 3\times3,\ 128 \\ 3\times3,\ 128 \end{bmatrix}\times2$
conv4_x	16×16×256	$\begin{bmatrix} 3\times3,\ 256 \\ 3\times3,\ 256 \end{bmatrix}\times2$
	256×256	spatial flatten (or)
	1×256	average pool

(a) Image encoder

layer name	output size	layer parameters
embed	28×28×28×8	embed, 8
conv1	14×14×14×64	7×7×7, 64, stride 2
conv2_x	14×14×14×64	$\begin{bmatrix} 3\times3\times3,\ 64 \\ 3\times3\times3,\ 64 \end{bmatrix}\times1$
conv3_x	7×7×7×256	$\begin{bmatrix} 3\times3\times3,\ 256 \\ 3\times3\times3,\ 256 \end{bmatrix}\times2$
	343×256	spatial flatten (or)
	1×256	average pool

(b) Voxel encoder

Table 4. Architectures for image and voxel encoders. Pre-activation residual blocks are shown in brackets, with the numbers of blocks stacked. Downsampling is performed by conv3_1, conv4_ for image encoders, and by conv3_ for voxel encoders, with a stride of 2. For AtlasNet models, we perform an additional linear projection up to 1024 dimensions before average pooling to obtain a vector shape representation.
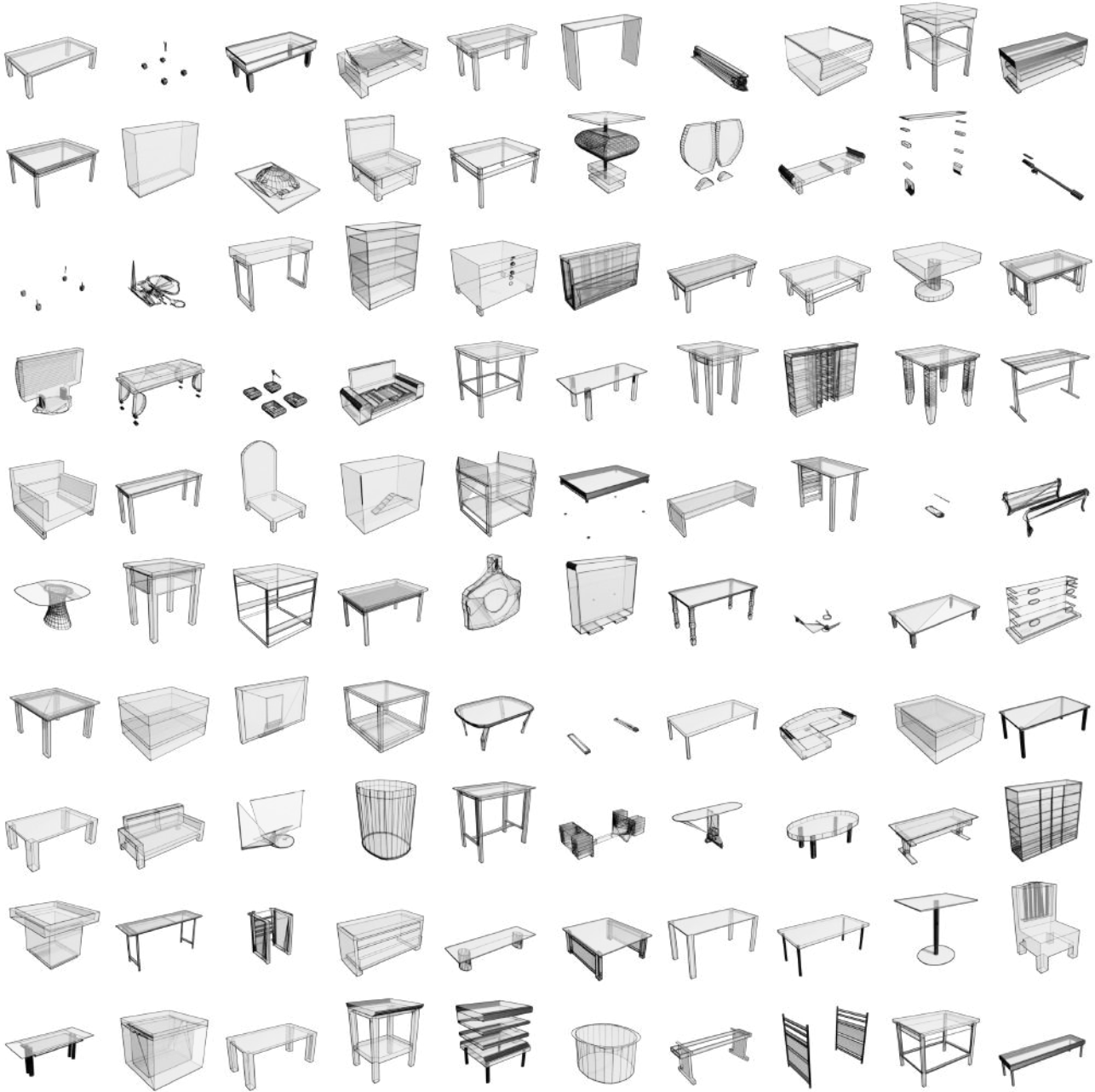
Figure 14. Random unconditional samples using nucleus sampling with top-$p = 0.9$.