Adaptive Meshes with Encoded Services

Ben Lee[1] and Stephen Brooks[1][a]
[1]*Faculty of Computer Science, Dalhousie University, 6050 University Avenue, Halifax, Canada*
{*bn628547, sbrooks*}*@cs.dal.ca*

Keywords: 3D Objects, Object Similarity, Surfaces

Abstract: 3D meshes consume more storage than is required for most applications. In this paper we use this surplus of precision to connect to a variety of online services, all while still being able to use the mesh as-is in the graphics pipeline. We describe our method to link to online services through an encoded URL in the mesh itself. We then introduce our machine learning based subdivision surface approach which uses the extra precision to also store user driven preferences. We then describe a method to store an embedding that can be used to retrieve similar meshes from an online database. Our similarity model does not rely on model categories and our embeddings are also significantly smaller, since they are encoded directly into the mesh.

1 INTRODUCTION

Meshes power games, movies, VR, and CAD, yet they often remain static after export. Our method embeds metadata into unused mantissa bits of vertex positions, enabling persistent connectivity without changing file size or format. Unlike transient GPU-based techniques, this lightweight, format-agnostic approach supports artist-driven creasing and links to online services for continual enhancement, making it suitable for entertainment and visualization workflows while adaptable to stricter CAD tolerances.

Existing metadata approaches often require custom formats, reducing interoperability. Our method leverages unused bits in standard floating-point positions, adding functionality without increasing file size or breaking compatibility with existing pipelines.

Our primary contribution is a lightweight, format-agnostic method for embedding metadata directly into unused bits of mesh vertex positions, enabling persistent connectivity without increasing file size or breaking compatibility. We demonstrate this concept through two applications—machine-learning-based subdivision surfaces and similarity-based mesh retrieval—showing how embedded metadata can guide automated refinement and facilitate dynamic online services.

This paper begins with a discussion of related work followed by an analysis of how many bits can be used without causing visual artifacts in Section 3. We then discuss how these bits can be used to link their corresponding meshes to online services. This opens the door for updating and improving meshes over time, such as in Section 5 where we apply this online service concept to subdivision surfaces with user preferences encoded. We follow this with a method of encoding online service information to support similar mesh retrieval in Section 6. This would allow meshes to be linked to an online database and similar looking meshes in a way that could ease the creation of virtual scenes.

2 PRIOR WORK

There are many methods to compress the storage requirements of a mesh. For example, Deering (1995) uses a modified Huffman encoding to store the deltas between the vertices, essentially splitting the modeling space into a grid and storing the local position in that grid. Chou and Meng (2002) compress the vertices with a codebook of codevectors, creating a lookup table for a fixed number of vectors. However, we have different aims and do not seek this level of compression. In our case, the storage requirement of the mesh will stay the same, but the information inside of it will be more compact.

A substantial body of work exists on digital watermarking for 3D meshes, aimed at ownership verification and IP protection. Techniques include covert embedding using secret keys (Cayre and Macq, 2003), spatial noise encoding with normal constraints (Yang et al., 2013), and reversible schemes for recovery (Lyu

[a] https://orcid.org/0000000274735830

et al., 2023). Surveys such as Wang et al. (2008) highlight robustness and imperceptibility as key goals.

Unlike classical watermarking, which prioritizes imperceptibility and ownership verification, our method openly encodes functional metadata for interoperability. Watermarking schemes often require secret keys and reversible embedding to restore the original mesh, whereas our approach accepts minor, visually imperceptible perturbations without guaranteeing exact reversibility—trading reversibility for persistent connectivity and zero file-size overhead. In terms of robustness, our encoding survives typical mesh operations such as rendering, subdivision, and format conversion, provided floating-point precision is preserved. Detectability is intentional: metadata is not covert but structured for optional use, avoiding the complexity of watermark extraction while ensuring compatibility with standard pipelines.

Subdivision surfaces are widely used by 3D modellers to split the polygons of a mesh into finer elements. After repositioning the newly added vertices, the mesh has higher quality geometry such as smoother edges. Traditional methods (Catmull and Clark, 1978; Loop, 1987; Dyn et al., 1990) typically position the newly made vertices through a weighted average of its neighbours.

But more recently there has been research on using other methods to update the vertex positions rather than a weighted average. Neural Subdivision (Liu et al., 2020) uses Loop subdivision's topological update of placing vertices in the midpoints of edges, but replaces the weighted average with small neural networks to determine the offset. Point cloud upsampling has been explored as well (Qi et al., 2017, 2016; Yu et al., 2018). However, this loses connectivity data so maintaining information like texture coordinates through changes in resolution is very difficult.

The subdivision method we propose in Section 5 is related to both (Liu et al., 2020) and (Yu et al., 2018) due to the fact that our machine learning model interprets our inputs as points. We use PointNet (Qi et al., 2016) as a backbone, but we use Neural Subdivision's (Liu et al., 2020) method of offsetting already subdivided points, as well as their training data generation technique. But we also augment this approach with user preferences regarding the handling of mesh creases and encode the information in the lower bits of the mesh itself.

Similar mesh retrieval is the process of retrieving an ordered collection of meshes from a database that are most similar to an input mesh. View based matching operates by rendering the mesh at various angles, and a descriptor is made from those renders (Chen et al., 2003; Su et al., 2015; Kanezaki et al., 2021; Zhirong Wu et al., 2015a). That descriptor is then used to calculate distances between the meshes, and can return the meshes with the most similar descriptors.

Our method is most similar to the approach of (Labrada et al., 2024). They use a view based convolutional autoencoder to generate embeddings of a mesh which are used to calculate distances between meshes. However, the structure of our model is smaller and the training is simplified since we do not rely on the category of the input models. Our resulting embeddings are also significantly smaller, and we encode them directly into the mesh.

3 Storing Services in Vertices

We analyze IEEE 754 floats and show that using 10–12 mantissa bits per axis introduces imperceptible error for typical rendering pipelines. This enables 30–36 bits per vertex for metadata, sufficient for linking meshes to services or storing user preferences without breaking compatibility.

For bits b_0 to b_{31} in a 32 bit IEEE floating point number, the value v is calculated as:

$$v = (-1)^{b_{31}} * 2^{(b_{30}b_{29}...b_{23})_2 - 127} * (1.b_{22}b_{21}...b_0)_2 \quad (1)$$

Where the most significant bit dictates the sign of the value, bits 30 to 23 define the exponent, and least significant bits 22 to 0 dictate the mantissa.

Since vertices can occupy any location in 3D space (positive or negative) we can ignore the sign bit. As previously discussed, the visual artifacts we aim to minimize stem from changes to the least significant bits of the mantissa. To estimate the maximum possible impact, we consider the largest offset that could result from flipping all these bits from zeros to ones, or vice versa.

Assuming we are allocating k bits from the mantissa for encoding, the largest difference on a given axis is:

$$\sum_{i=1}^{k} 2^{i-24} * 2^{(b_{30}b_{29}...b_{23})_2 - 127} \quad (2)$$

with meshes typically modelled around the origin. For a floating point value of n, the exponent is equal to $2^{\lfloor log_2(n) \rfloor}$, or floored to the nearest power of two. For a float value of n, the maximum error would be:

$$\sum_{i=1}^{k} 2^{i-24} * 2^{\lfloor log_2(n) \rfloor} \quad (3)$$

Figure 1 shows the maximum error while encoding a different number of bits k for a range of values n. The

error doubles every time n passes a power of two as more bits are used, as can be seen in the loglog graph.

This is intuitive, as the maximum possible error occurs when all bits are flipped from 0 to 1 or vice versa. Such a change effectively shifts the mantissa from a value of 1 to nearly 2 (or the reverse), which is then scaled by the exponent.

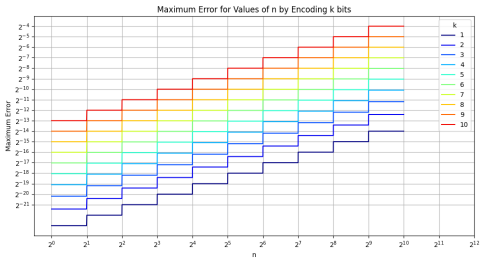

Figure 1: Maximum Error for floating point values of n, with increasing number of bits allocated for encoding k. Loglog plot (bottom).

While numerical estimates are useful, it's also valuable to understand how these changes manifest visually on 3D meshes. This can be demonstrated by embedding arbitrary data into the lower bits of the mantissa, gradually increasing the number of bits until visible artifacts appear. For this experiment, we used the first 19 digits of pi in a repeating loop across all vertices as a stand-in for actual metadata.

Figure 2 shows the relative distance of each vertex between the ground truth and a version with 12 bits per axis. As the histogram on the left shows, green represents smaller differences from the original mesh, and red and blue indicating larger differences. As mentioned, the relative error depends on the exponent of the floating point number, and therefore the distance the vertex is from the origin. The number of bits with encoded data stays constant per vertex, but the relative distance is larger the further away from the origin it is. With 12 bits per axis, it is still on the magnitude of $3.09 * 10^{-2}$ at the outside portions of the mesh (red and blue), but the majority of the vertices have a much smaller error (green) around the origin.

Based on our empirical testing with meshes centered at the origin, we found that we can readily utilize 10 to 12 bits per axis without introducing noticeable visual artifacts. A 3D vector, or coordinate typically requires 96 bits, one 32-bit float per axis, and if we make use of 10-12 bits per axis for metadata, we effectively store each vertex with 60 to 66 bits.

Empirical testing shows that encoding 10–12 bits per axis introduces a maximum positional error of $3.09 * 10^{-2}$ units at extreme distances, which is imperceptible in typical rendering pipelines and well within tolerances for entertainment and visualization workflows. For CAD or engineering contexts requiring stricter precision, fewer bits can be allocated, preserving compatibility.

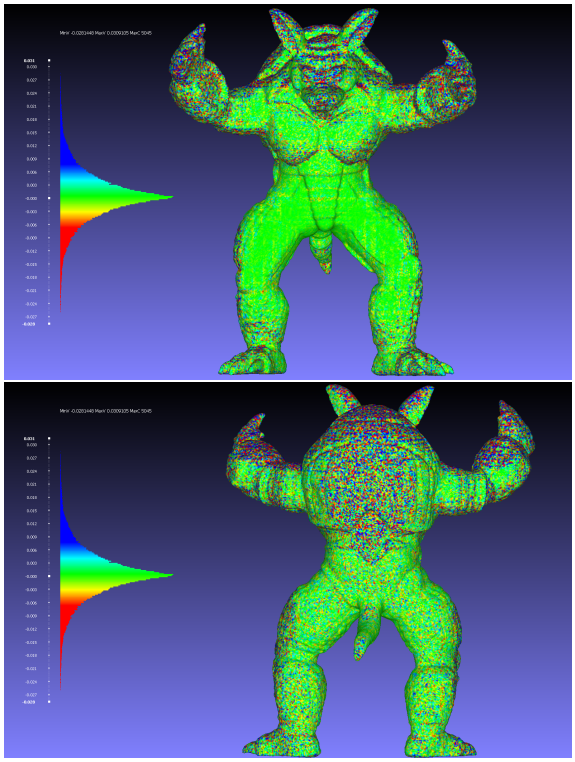

Figure 2: Relative signed distance from encoding pi in the bottom 12 bits of vertex positions

4 Distributed Services for the Mesh

If we view vertices and meshes as a basis for internet services, we can explore uses for this encoding concept that are varied and impactful. For example, if the mesh has extra metadata encoded in it, there are more attributes or features we can optionally add by requesting it from an online service. In other words, embedding URLs directly in vertex data enables persistent connectivity without external files or custom formats, unlike glTF or USD extensions that require specialized parsers and may break compatibility. This lightweight approach keeps meshes usable "as-is" while supporting dynamic services such as hardware-aware quality adaptation. Then the mesh simply needs an encoding of quality enhancements and a link to the online service that provides the data augmentation.

We can allow vertices and meshes to interface with these online services by encoding a URL within

the vertices themselves. To support multiple metadata types, we include a small header in the encoding. This header specifies the data type (e.g., URL, subdivision creasing, or similarity embedding) before the actual payload. For example, an eight-bit field can denote the type, followed by length and content fields. Services can be hosted via standard APIs.

Furthermore, a mesh does not need to be limited to a single service. As discussed, it is possible to encode many URL's in a single mesh, and metadata associated with that service may only take a few bits per vertex. As long as the metadata for each service uses mutually exclusive bits, the mesh can link to multiple services without conflict.

In a sense, this URL encoding causes the metadata to always be updating and can therefore improve over time. For example, one of the applications that will be explored is similar mesh retrieval and linking the metadata to a service will cause the mesh to link to more and more similar meshes over time as the database grows. In the following section, we describe a subdivision surface based service, which allows mesh customization and continual enhancements.

5 Encoding Subdivision Surfaces

Our encoding stores crease metadata in the mesh, enabling ML-driven subdivision without external files. It supports continuous enhancement across hardware generations without requiring artists to recreate assets.

Our machine learning-based subdivision method provides automated resolution enhancement, but it cannot always infer artistic intent, such as where edges should remain sharp or smooth. To address this, we encode creasing metadata directly into the mesh, allowing artists to guide the process without external files. This tight integration ensures that the ML model operates efficiently while respecting user-defined constraints, and the encoded URL links the mesh to an online service for continual refinement. These hard and soft edges called creases are generally discarded once the modelling software exports the mesh because meshes are typically considered static at that point.

The overall process of our subdivision method will be as follows: We first add new vertices according to Loop's method (Loop, 1987) see Figure 8 (top), but we do not offset them using a weighted average. We will instead use a machine learning model to generate the offsets. After this we use information that was pre-encoded by an artist in the lower bits of the mesh to sharpen or smooth edges where our model may misinterpret the object.

We use machine learning because we are limited by space and want to use the storage we have available for creasing. This aims to provide the correct result the majority of the time, then only requires artist involvement where it is needed.

In terms of architecture, we use PointNet (Qi et al., 2016) as our backbone because we have two goals that should be met. The first is that we want the model to be robust to mesh topology. The second goal is that we want to preserve any attributes that are currently in the mesh, such as texture coordinates across subdivision levels. We use a traditional subdivision topological update followed a machine learning step to offset the newly added points. This is a similar process to Neural Subdiv(Liu et al., 2020), though we note that their architecture requires meshes with no boundaries or holes, which does not satisfy our first goal.

To reiterate, we aim to create a subdivision surface pipeline that can interpret sharp and soft edges correctly the majority of the time, and requires minimal artist involvement.

5.1 Architecture

As mentioned, we base our architecture on PointNet (Qi et al., 2016). However, this prior work was based on classification and segmentation, so we have changed the output to be a 3D vector representing the offset to apply to the newly made vertex. We have also reduced the number of parameters in the linear layers for speed of training. Furthermore, we only input the nearest n vertices, rather than the entire cloud of vertices. But before diving into specifics, we first describe the benefit of using this architecture and why it applies to subdivision surfaces.

We use PointNet as a base because of the unordered nature of our input. Given a set of vertices, the order in which those vertices were given should not affect what surface they create. Unordered input is difficult to input into a typical multi-layer perceptron (MLP), since the model would need to implicitly learn that the order of the input vertices does not matter. But an MLP interprets the input as a high dimensional vector, so it would need to spend effort learning that the vertices (or groups of 3 dimensions) can be permuted in any way and still give the same output. It is therefore more efficient to use a different architecture than an MLP.

An overview of the architecture can be seen in Figure 3. The input is the nearest n vertices to the current position that were present in the mesh before subdividing, and the output is a 3D vector describing

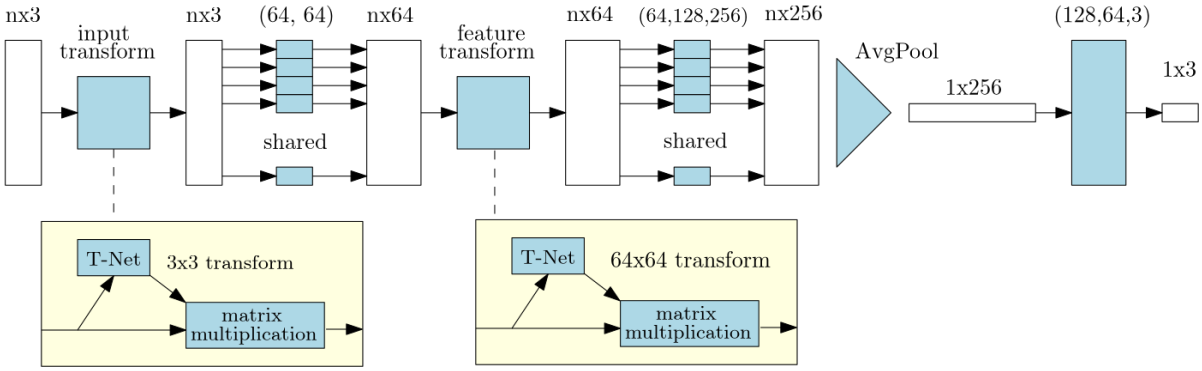

Figure 3: PointNet Qi et al. (2016) architecture, modified to have less parameters for faster training and a three dimensional output denoting the offset to apply to a vertex. Each layer except the last includes batch normalization. Brackets indicate the size of the layers. Blue indicates operations. T-Net is a smaller version of the larger architecture, consisting of two hidden layers, an average pooling then two more hidden layers which outputs a matrix, biased around the identity matrix. In our tests $n = 30$.

how that vertex should be offset from its original position. A series of small shared MLPs are used along with T-Nets, which are essentially more shared MLPs. After this an average pool is used, followed by a few linear layers to generate the final output. Each linear layer uses ReLU activation, along with batch normalization, except the last layer.

For each vertex we supply the machine learning model with the positions of the nearest $n = 30$ vertices that were present in the original mesh. The intuition being that most of the information needed to determine local curvature is within the vicinity of the vertex itself. The choice of $n = 30$ is somewhat arbitrary, however it is important that it is large enough to capture local detail, but not so large that it starts to cause memory and performance issues.

Importantly, the choice of n is not overly sensitive. We trained our model with $n = 25, 30, 35$ and found that all gave similar results, as shown in Figure 4. We can also show that $n = 30$ is not mesh dependent and can operate on meshes with varying resolutions, see Figure 7.

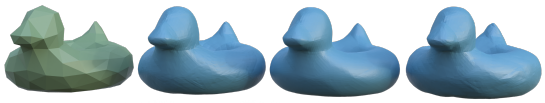

Figure 4: Original Bob mesh (left) and subdivided with models trained with $n = 25, 30, 35$ from left to right.

However, simply inputting these positions in world space would cause some issues because the model would output a different vector if the mesh were translated, rotated or scaled in any way. If any of these affine transformations were applied, it would make sense for the output to be the same, since the object's geometry is the same, and it is simply moved around in space. To address this we move each vertex and the nearest neighbouring vertices to the origin, and rotate them so the vertex's normal is facing up. This provides both translation and rotation invariance. We can achieve scale invariance by also scaling the input vertices to be within a distance of zero and one. In our case we divided by the distance to the furthest vertex. After obtaining the offset from our model, we then transform it back to the original point in space using the inverse transformation used to normalize it.

We use the Neural Subdivision method (Liu et al., 2020) of generating training data. By creating a bijective mapping between two subdivision mappings through repeated decimations. A bijective mapping means that we have a one-to-one mapping between a vertex in one subdivision level to the next. This is generated through a series of decimations of the mesh by collapsing edges until a desired vertex count is reached while keeping track of which vertex will subdivide to which. This means that a simple L^2 loss can be used between the output position and where it was in the original mesh before being decimated.

For training, we used the first 300 meshes of the SHREC15 database (Lian et al., 2015), with three subdivision levels, the lowest having 6000 vertices. We implemented this in PyTorch, using the Adam optimizer. Our model takes approximately less than hour to train on a off-the-shelf GPU.

5.2 Results

Before describing how we will be encoding information in the lower bits relating to subdivision surfaces, we can first analyze how our machine learning based method performs compared to traditional methods.

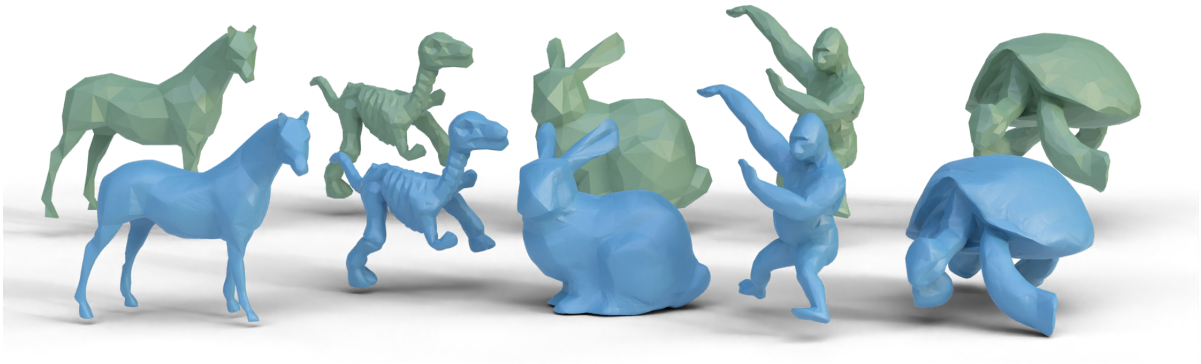

Figure 5: Original low poly meshes in green, subdivided with our model in blue

Figure 5 show some results of our model. Compared to traditional methods, ours can preserve hard edges more accurately without human involvement.

A benefit of scale invariance is that our subdivision method can be used for any number of iterations. As the resolution increases, the local neighbourhood of a vertex becomes more and more flat, and the model generally preserves the flatness of surfaces. Figure 7 shows this, even after five iterations, the model does not introduce any major artifacts, even though it has not trained past two subdivision levels.

We can quantify our results by using Hausdorff distance. In Figure 6 we take the TOSCA dataset and decimate the meshes to 1000 faces. We then subdivide them twice with our method, and Blender's Catmull Clark implementation, then calculate the Hausdorff distance between the subdivided meshes and the originals using 40000 samples. We can see that overall our initial method provides lower median distances than Catmull Clark. Two tailed T-tests assuming unequal variances were run and found that our distances were statistically significant using a confidence interval of 0.95. Over forty thousand data points were used per test, so we assume central limit theorem applies and the T-Test is a valid statistical test to use.

5.3 Creasing Encoding

Traditional subdivision methods are almost always smoothing functions, and have difficulty maintaining sharp edges. With low-poly meshes it can be inherently ambiguous whether certain edges should stay sharp, or be smoothed out. Consider the icospheres in Figure 9, there's a possibility that they should be completely smoothed out, or perhaps the author created the flat square face shown on the front side with intention and it should remain flat. Since curvature can be ambiguous, our model may also misinterpret the curvature in a certain area, like smoothing an edge when it should remain sharp.

Using the available bits in the vertices, we implemented an augmentation where the user can dictate which edges stay sharp and which can be smoothed beforehand. We based this on the traditional method of creasing. In our case, our model may sharpen edges we want smooth, or smooth edges we want sharp, so we allow the user to optionally mark vertices as either of the two by encoding a boolean value into the lower bits of the vertices. Then an edge is considered marked if it is between two tagged vertices. These vertices are marked by encoding a boolean value in a bit of the vertex, along with some information on whether it is a sharp crease, or a soft crease. In practice, this vertex tagging could be facilitated with direct vertex painting in any 3D modelling package. The encoding of the crease can then be stored into the lower bits of the mesh.

5.4 Sharp Creasing

The topology of the mesh updates in the same way on each iteration using loop subdivision, which places a new vertex at the midpoint of each edge. We then offset the newly added points with our model. We can visualize an update of two triangles in Figure 8.

If a hard crease is marked between p_c^m and p_1^m then on the surface these two triangles are approximating, the immediate angle θ between those two vertices should stay pronounced up to a certain distance depending on the strength of the crease.

Consider in Figure 8 (bottom row), in the example with the hardest edge, the influence of θ is strong, so anywhere along the crease up to p_2^m or p_0^m that angle stays the same.

However, in the smoother example, the further towards p_2^m or p_0^m from the crease one goes, the more it deviates from θ. We approximate this angle through θ by fitting two planes to $p_c^m, p_{c1}^{m+1}, p_1^m$ with p_2^m and p_0^m

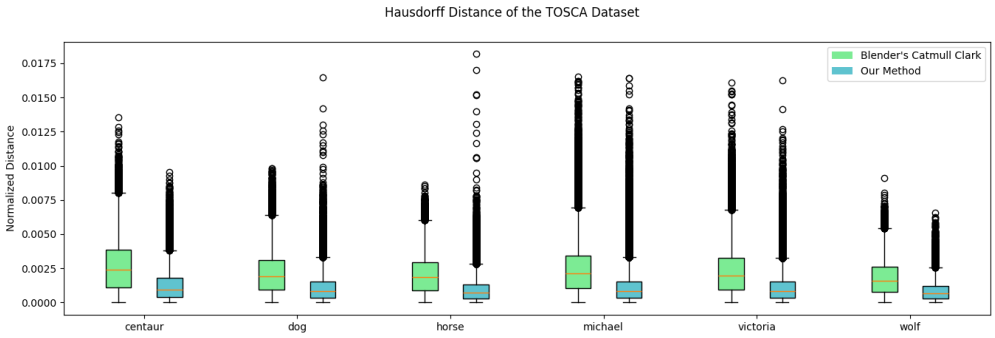

Figure 6: Mean Hausdorff distance between the original mesh in the TOSCA dataset, and subdividing a 1000 face decimation twice using Blender's Catmull Clark implementation and our method.

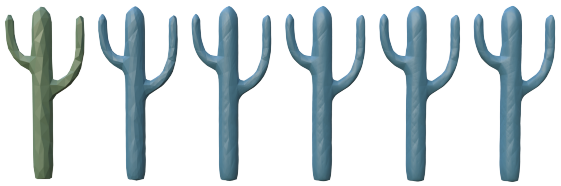

Figure 7: Subdividing for more iterations than it has seen during training does not produce visual artifacts. Cactus model from (Liu et al., 2020) source code.

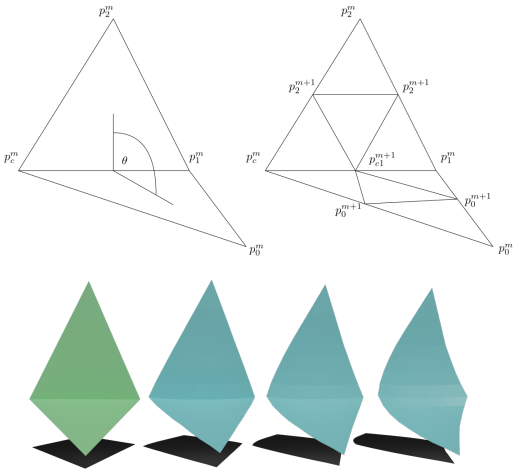

Figure 8: Subdivision of two triangles using loop subdivision's topology update. p^m indicates vertices that exist on a mesh and vertices p^{m+1} are vertices that exist in the subdivided version (top). Possible crease strengths ranging from infinitely hard to very soft (bottom).

and project the four points p_2^{m+1} and p_0^{m+1} onto this plane. We use the mean position of the input points to find a point on the plane, then singular value decomposition to calculate the normal.

We include p_{c1}^{m+1} in the plane fit in case there is any curvature introduced from the output of our model that we should keep. For example, the creased edge is a convex edge and bends outwards away from p_0^m in Figure 8 (top), which helps preserve that curvature as well.

We can further extend this approach by setting the strength of the hard edge. Instead of having a boolean value, hard or not, we can associate a value between zero and one to interpolate between the original output, and the projected points. So instead of encoding a boolean value into the mesh's lower bits to guide the subdivision creasing, we can encode a fixed precision value. For example, if we allocate four bits towards the strength of a hard crease, we can set eight different strengths with intervals of 0.125. The crease calculations can be deactivated if the strength is zero. We apply the creasing on every level of subdivision since the creasing effect will be stronger the larger the triangles are.

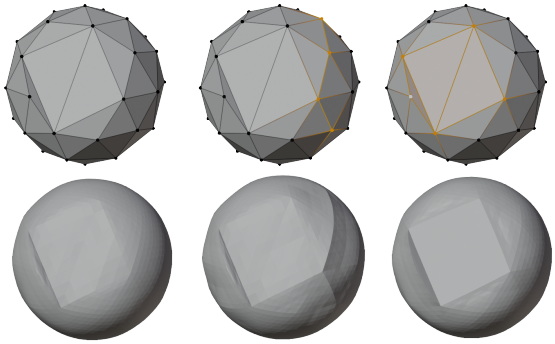

Figure 9: Modified icosphere subdivided with hard creases. Base model with highlighted edges in orange showing the creases and their subdivided counterparts below. Left most pair has no creases marked.

5.5 Soft Creasing

Softening an edge is almost an identical process as hardening an edge, except that instead of preserving θ, we flatten it. By fitting a plane between the four new vertices p_0^{m+1} and p_2^{m+1} and projecting p_c^m, p_{c1}^m

Figure 10: Softening of creases on the Stanford bunny. Edges are marked as orange if they are softened (left). Default output from model (middle) and output with softened edges (right). Note the seams around the neck and tail.

and p_1^m onto it, we can flatten θ. Figure 10, shows this method applied to the Stanford bunny around edges that may be interpreted as too sharp.

Similar to hard creases, we can adjust the strength of the softening by scaling between the original output of the model, and the plane projected point, as well as only applying the softening on later subdivisions. In practice, this user guidance would again take the form of direct user "painting" of crease values. Figure 11 shows soft creasing at varying strengths.

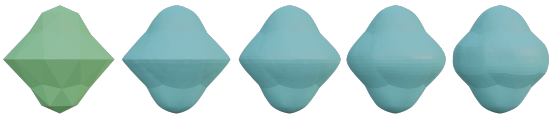

Figure 11: Original mesh in green, then soft creasing along the middle at strengths 0.0, 0.25, 0.5 and 1.0 from left to right in blue, three times subdivided.

6 Similar Mesh Retrieval

We can further take advantage of the unused bits within the 3D vertices by establishing connections between the meshes online. Scene design often requires finding visually compatible assets. We embed compact similarity embeddings and service URLs directly in meshes, enabling dynamic retrieval without extra files or format changes.

A naive way to retrieve similar meshes from a database is to send the mesh itself, and have the server analyze it and perform a search. However, meshes can be large in size or may be proprietary. It would be better to send a compressed slice of data that contains enough information needed to perform a search. We can include this in the lower bits of the mesh so it is available to anyone who comes across it.

We do this by using a view based approach. We render the mesh from a set of images and pass it through a convolutional autoencoder. The output of which provides an embedding, or a compressed slice that we encode into the mesh itself. Considering the limited number of bits available in the vertices, these embeddings are a suitable method to facilitate similar mesh retrieval.

While modern formats like glTF and USD allow high-level metadata storage, these approaches often require custom extensions and may not be supported across all pipelines. Our method encodes similarity embeddings directly in unused bits, ensuring zero increase in file size and full compatibility with existing tools. This low-level approach enables persistent, format-agnostic connectivity without breaking standard workflows, which is critical for assets shared across heterogeneous environments.

6.1 Methods

Our pipeline renders 12 fixed views per mesh, passes them through a convolutional autoencoder, and stores compact embeddings plus a service URL in vertex bits. At query time, embeddings are permuted to handle orientation and compared via Euclidean distance for retrieval.

Our approach assumes alignment to global axes for simplicity, which works well for most models. However, this assumption is not universal. Alternatives such as detecting principal directions via spectral analysis, wavelets, or smallest enclosing unaligned rectangles could reduce reliance on fixed orientations, especially for rigid models. Exploring these methods is left for future work.

Another assumption is that the software these meshes are made in, such as Blender (Blender Development Team, 2022), or Maya (Autodesk Inc, 1998), typically have a built-in axis designated as the up direction. Most models are also oriented in a way where it has an up direction, such as a computer monitor or a water bottle.

With these assumptions, we can narrow the alignment of the mesh down to four possibilities. Consider the airplane model in Figure 12, if we consider +Y as the up direction, then the nose of the plane, or its forward direction is either -X, +X, -Z or +Z. With this we can record the mesh in twelve images, which will be enough to allow us to generally avoid explicitly aligning meshes to one another.

Prior to rendering the images, we also temporarily scale the object down to a 1m cube and translate it so its bounding box is centered to the origin for normalization. We then render depth-based screenshots of the object at fixed angles. One from each of the possible forward axes (-X, +X, -Z, +Z), and four from the top and bottom. For the top and bottom, we take four so that at least one of them has the mesh's forward direction towards the top of the image.

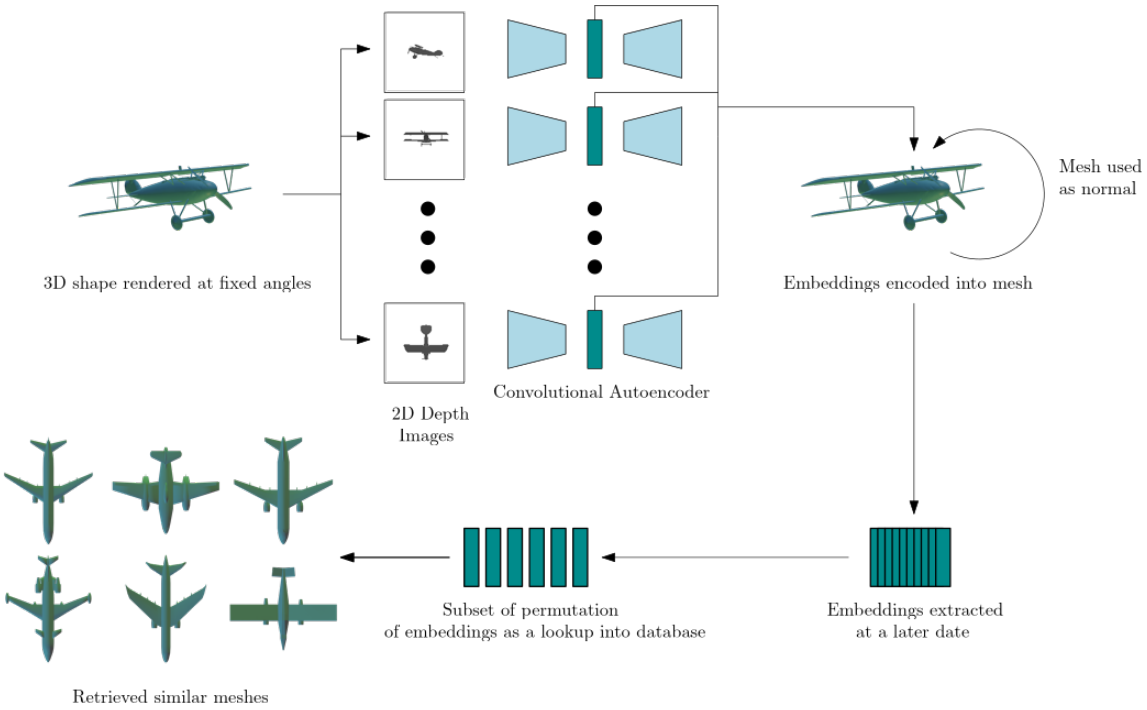

Figure 12: Overview of our mesh retrieval pipeline. A mesh is rendered at fixed angles, then those images are sent through a convolutional autoencoder generating an embedding. Those embeddings are encoded into the lower bits of the input mesh along with a URL to a mesh retrieval service. This mesh can then be used as is, indefinitely until someone wants to retrieve similar meshes. They would then send the service those embeddings as a lookup to return similar looking meshes.

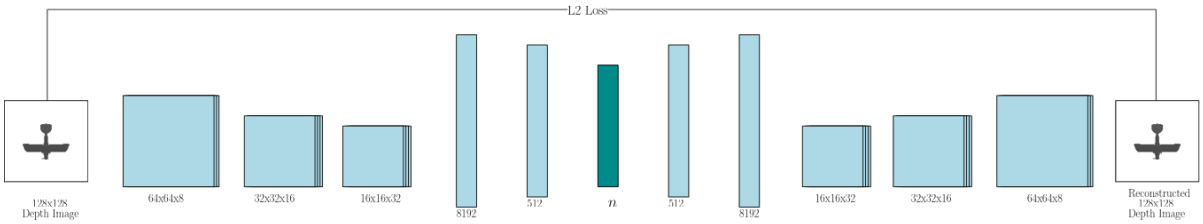

Figure 13: Architecture of our autoencoder. Each convolution (square) in the encoder is followed by a max pooling layer, mirrored by a max unpooling layer in the decoder. After the convolutions in the encoder, it is flattened and put through fully connected layers to reduce the dimensionality down to our embedding size n. Then increased back to mirror the decoder convolutions. The reconstructed output image is then compared to the input with L2 loss during training.

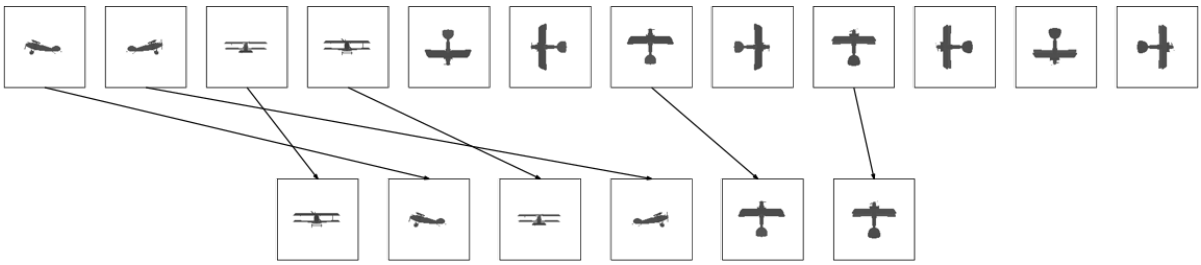

Figure 14: Example of permuting the complete set of depth embeddings (represented by their corresponding images) to create a lookup query.

From this we have twelve 128x128 grayscale renderings of our object from fixed angles. Each one of these images can then be passed on to a convolutional autoencoder to generate an embedding.

The architecture can be seen in Figure 13 and is based on a convolutional autoencoder. It begins with three convolutional layers. It is then flattened and passed through two linear layers down to an embed-

ding size n. This linear layer is the data that will be encoded into the lower bits of the mesh. The rest of the network essentially runs in reverse: two linear layers are followed by three deconvolutional layers to reconstruct the input image.

The L2 loss between the output and input images is used during training. Other approaches such as (Labrada et al., 2024) use an additional classification network that attempts to predict the category based on the embedding layer, then uses that cross entropy loss with the L2 loss in training. But our autoencoder has no notion of the categories, and should be able to generalize to other types of meshes.

With twelve depth based images per mesh, we have twelve n-float embeddings to encode in the lower bits. This sums to $n * 32 * 12$ bits or $48 * n$ bytes. In our experiments we tried n equal to 16, 32 and 64, relating to 768, 1536 and 3072 bytes respectively.

After storing the URL and embedding into the mesh, we can again use the mesh as-is indefinitely since the encoded bits would only cause imperceptible changes in the geometry. The benefit is that in the future the mesh has additional information in the form of this encoding to retrieve similar meshes, should the user wish to.

To perform the similarity based lookup, we wish to have embeddings in the order of front, right, back, left, top and bottom, where top and bottom have forward facing up. We obtain this ordering by making four lookups, permuting the twelve embeddings and taking six of them. An example of this can be seen in Figure 14. One of these will be the correct ordering, regardless of which axis was considered forward when modelling. We then use the Euclidean distance between the request embedding and those in the database, as a 6 by n vector to determine which meshes in the database are most similar. We trained on ModelNet40 (Zhirong Wu et al., 2015b) with the training and test split provided in the dataset for 300 epochs using the Adam optimizer included in PyTorch.

6.2 Results

Despite the minimal number of bits required, our method performs well in general. And our method successfully returns objects that look similar to the query mesh. In particular, objects with well defined silhouettes are very well suited to our method.

However, there can be objects within the same category that actually do not inherently look like each other, or look more similar to objects in other categories which give misleading results.

We attempt to show this with another metric, top-NmAP (Top N Mean Average Precision), popular in internet search engines. Mean average precision is based on the ranking of the entire query, so the metric is sensitive to how it orders the whole database based on a query. With internet search engines this is infeasible, so instead it takes the first N results of the query and calculates the metric based on that. In the context of internet searching, N is usually the number of items on each page.

We developed a prototype interface for our mesh retrieval that we will discuss later in this section, but we note here that it shows six items at a time. Therefore we choose N as six and twelve, relating to the mAP scores on the first and second "pages" respectively. These can be seen in Table 1 and Figure 15.

The values show that within the first six or twelve items it performs well for returning meshes in the same category. It makes sense that top six and twelve mAP score well, since they are returning the radios that look similar.

A drawback of our method can be the relative scales of objects. Prior to rendering the objects, they are scaled down to a 1x1x1 cube. If an object is especially long in one axis, such as a radio with a long antenna then it will get scaled down and the body of the radio will take up a much smaller area than if it did not have it. Generally, this means that objects with differing axis ranges, or aspect ratios are not going to match as well. This could be improved with some sort of dynamic scaling, perhaps ignoring small extensions like antennae, but this is left for future work

We can further confirm partial rotation independence by rotating the test set by random 90 degree intervals about the up axis. In Table 1, we can see that the mAP with this prior rotation does not change considerably when the rotation is not applied. The small change is most likely due to aliasing in the low resolution renders, combined with the fact that there are a small number of meshes in the dataset that do not follow our assumptions and have a different up axis than the rest, or are modeled off of the global axes.

With the networked mesh infrastructure in place, this can be presented in an interface for scene designers to use. A prototype can be seen in Figure 16. Pressing the "Get similar objects" automatically extracts the embeddings and URL from the whichever mesh is currently selected, and downloads the meshes in the database with the most similar embeddings. The retrieved objects are placed on the neighbouring circles. They automatically rotate synchronously with the reference mesh for easy comparison, and clicking on one of the similar meshes will swap it with the reference to allow for quick replacements and testing.

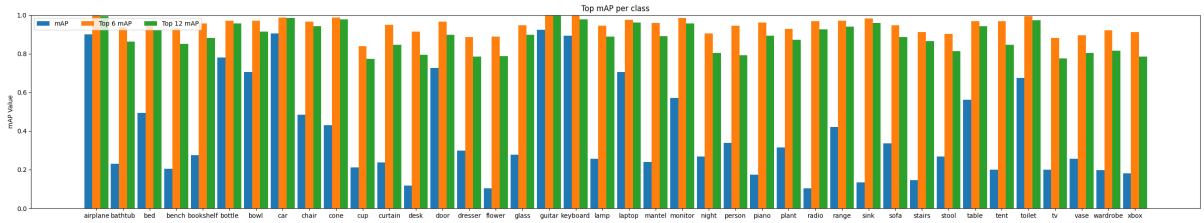

Figure 15: mAP, Top 6 mAP, and Top 12 mAP for 32 bit embeddings for ModelNet40.

mAP	16	32	64
ModelNet10	0.517	0.589	0.567
ModelNet10 (with random rotations)	0.511	0.586	0.563
ModelNet40	0.399	0.450	0.434
ModelNet10 Top 6	0.953	0.958	0.960
ModelNet10 Top 6 (with random rotations)	0.945	0.959	0.957
ModelNet40 Top 6	0.943	0.952	0.951
ModelNet10 Top 12	0.903	0.915	0.919
ModelNet10 Top 12 (with random rotations)	0.897	0.911	0.914
ModelNet40 Top 12	0.880	0.900	0.896

Table 1: Changing the embedding size between 16, 32 and 64 does not significantly change the mAP, nor does rotating the test set, showing that we can model something aligned to any of the four orientations and it does not hinder our lookup significantly.

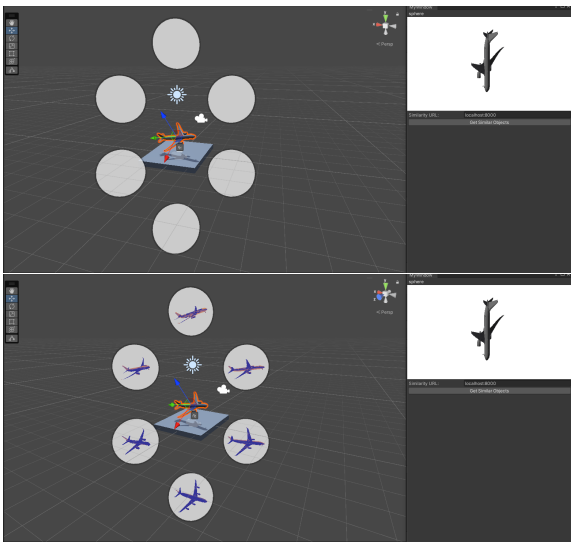

Figure 16: Prototype interface for our mesh retrieval system. A mesh can be selected, then it extracts the necessary information for the request (top). Then a request can be made to retrieve similar meshes from the database (bottom).

7 Limitations

Modifying vertex positions beyond the least significant bits can invalidate embedded metadata; however, re-encoding is straightforward since the process only requires bitwise operations on vertex coordinates. URL persistence depends on the availability of the linked service—if the server becomes unreachable, the mesh remains usable but loses dynamic functionality. This limitation could be mitigated through redundancy or caching strategies. Regarding security, URLs are openly encoded and could be spoofed if intercepted; therefore, services should employ HTTPS and authentication tokens to prevent malicious redirection.

Additionally, our current evaluation of geometric precision loss is limited to a single mesh; a more comprehensive study across diverse models and resolutions is left for future work. However, crease intent scales naturally with mesh size because both the data and its encoding grow proportionally with the number of vertices. Expanding capacity for global metadata still requires adding vertices, which may not suit all applications.

Additionally, we rely on the lower bits of floating-point positions, so the method is incompatible with file formats that truncate or compress data—making binary formats preferable, as ASCII representations often lack sufficient precision.

8 Conclusions And Future Work

We introduced adaptive meshes by encoding metadata in unused bits, enabling persistent services without format changes. Applications include ML-driven subdivision with crease hints and similarity-based retrieval. Future work may explore dynamic effects like aging or material-aware updates.

To use the decaying example; there could be a sce-

nario where a player's character ages (or decays if they were a robot) over real or accelerated time. One could also consider other aspects of an evolving object in an evolving world; for example a multiplayer game where the world becomes overgrown with moss or vegetation in real time.

There are also ways to add accessibility to the mesh, such as detecting vertices vital to the silhouette and encoding a value to increase the contrast on them for the visually impaired.

ACKNOWLEDGEMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Discovery Grant. The authors gratefully acknowledge this funding, which made this work possible.

REFERENCES

Autodesk Inc (1998). Autodesk maya. https://www.autodesk.com/ca-en/products/maya.

Blender Development Team (2022). Blender (version 3.1.0). https://www.blender.org.

Catmull, E. and Clark, J. (1978). Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355.

Cayre, F. and Macq, B. (2003). Data hiding on 3-d triangle meshes. *Signal Processing, IEEE Transactions on*, 51:939 – 949.

Chen, D., Tian, X., Shen, Y., and Ouhyoung, M. (2003). On Visual Similarity Based 3D Model Retrieval. *Computer Graphics Forum*, 22(3):223–232.

Chou, P. H. and Meng, T. H. (2002). Vertex Data Compression through Vector Quantization. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):373–382.

Deering, M. (1995). Geometry compression. In *Proceedings of SIGGRAPH '95*, pages 13–20.

Dyn, N., Levine, D., and Gregory, J. A. (1990). A butterfly subdivision scheme for surface interpolation with tension control. *ACM Trans. Graph.*, 9(2):160–169.

Kanezaki, A., Matsushita, Y., and Nishida, Y. (2021). Rotationnet for joint object categorization and unsupervised pose estimation from multi-view images. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 43(1):269–283.

Labrada, A., Bustos, B., and Sipiran, I. (2024). A convolutional architecture for 3D model embedding using image views. *The Visual Computer*, 40(3):1601–1615.

Lian, Z., Zhang, J., Choi, S., ElNaghy, H., El-Sana, J., Furuya, T., Giachetti, A., Guler, R. A., Lai, L., Li, C., Li, H., Limberger, F. A., Martin, R., Nakanishi, R. U., Neto, A. P., Nonato, L. G., Ohbuchi, R., Pevzner, K., Pickup, D., Rosin, P., Sharf, A., Sun, L., Sun, X., Tari, S., Unal, G., and Wilson, R. C. (2015). Non-rigid 3D Shape Retrieval. In *Eurographics Workshop on 3D Object Retrieval*. The Eurographics Association.

Liu, H.-T. D., Kim, V. G., Chaudhuri, S., Aigerman, N., and Jacobson, A. (2020). Neural subdivision. *ACM Transactions on Graphics*, 39(4).

Loop, C. (1987). Smooth subdivision surfaces based on triangles, master's thesis. *University of Utah, Department of Mathematics*.

Lyu, W., Cheng, L., Yin, Z., and Luo, B. (2023). Efficient reversible data hiding for 3d mesh models based on multi-lsb substitution and ring-prediction. In *APSIPA ASC*, pages 1907–1914.

Qi, C. R., Su, H., Mo, K., and Guibas, L. J. (2016). Pointnet: Deep learning on point sets for 3d classification and segmentation. *arXiv preprint arXiv:1612.00593*.

Qi, C. R., Yi, Li, Su, Hao, Guibas, and J., L. (2017). PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. arXiv:1706.02413 [cs].

Su, H., Maji, S., Kalogerakis, E., and Learned-Miller, E. G. (2015). Multi-view convolutional neural networks for 3d shape recognition. In *Proc. ICCV*.

Wang, K., Lavoue, G., Denis, F., and Baskurt, A. (2008). A Comprehensive Survey on Three-Dimensional Mesh Watermarking. *IEEE Transactions on Multimedia*, 10(8):1513–1527. Conference Name: IEEE Transactions on Multimedia.

Yang, Y., Peyerimhoff, N., and Ivrissimtzis, I. (2013). Linear correlations between spatial and normal noise in triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 19(1):45–55.

Yu, L., Li, X., Fu, C.-W., Cohen-Or, D., and Heng, P.-A. (2018). Pu-net: Point cloud upsampling network. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zhirong Wu, Song, S., Khosla, A., Fisher Yu, Linguang Zhang, Xiaoou Tang, and Xiao, J. (2015a). 3D ShapeNets: A deep representation for volumetric shapes. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1912–1920, Boston, MA, USA. IEEE.

Zhirong Wu, Song, S., Khosla, A., Fisher Yu, Linguang Zhang, Xiaoou Tang, and Xiao, J. (2015b). 3D ShapeNets: A deep representation for volumetric shapes. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1912–1920, Boston, MA, USA. IEEE.