Octree-based Graph Representation for 3D Model Classification

Keran Wang
wangk16@rpi.edu

Fu Chai
chaif@rpi.edu

Abstract

Multi-view image representation is one of the traditional methods for 3D representation, but it has an issue with depth relation. We want to explore the use of octree in finding key points in a 3D mesh that could construct a better multi-view image (in our project, we call it bitmap, which is a less complex data structure) for classification. We developed a program for 3D model classification using an octree-based graph representation followed by classification and texture generation. Our approach extracts multiple types of geometric features from 3D models using the octree data structure and represents them as graphs. We train a convolutional neural network using these graph representations as input. Also, we offer a practical application using the classification result: texture generation with the predicted label as a part of the prompt for the texture image generation, which would be applied on the surface of the model afterward. Our approach provides a promising direction for future research in 3D model classification, with potential applications in games, room design, 3D reconstruction, etc. Additionally, our project is open source on Github https://github.com/keran-w/3D-deep-learning.

Contributions: Fu Chai worked on octree graph representation and PCA. Keran Wang worked on graph representation classification and texture generation.

1. Introduction

Accurately classifying and understanding 3D models is an essential task with applications in various fields, such as computer-aided design, gaming, and virtual reality. Deep learning approaches have shown promising results in recent years but often require large amounts of data and computational resources. Additionally, effectively representing the complex geometric structure of 3D models remains a challenge.

This paper proposes an approach for model classification using an octree-based graph representation. Octrees are a hierarchical data structure allowing us to partition the 3D space into cubic regions efficiently. We extract multiple types of geometric features from 3D models using the octree. These features are then represented as a graph, where we can convert them into images, then send them to the convolutional neural network for training and classification. We also use Arcface and Focal loss functions to enhance the power of identifying [1] [2]. Eventually, we achieved over 73.5% accuracy on 40 labels in the ModelNet40 dataset. After the classification, we also apply a pre-trained diffusion model, which would take the classification result text as part of the prompt for generating a texture image, which would be applied on the surface of the 3D model. One of the contributions of our work is the exploration of different types of features obtained from the octree. We compare the performance of these features in model classification. We train a convolutional neural network using these graph representations as input and evaluate our approach on the benchmark dataset ModelNet40. While our approach did not achieve state-of-the-art performance compared to other methods, our results show the potential of using the octree-based graph representation for 3D model classification. Our work shows a potential method for 3D model classification and a practical application that the classification matters.

2. Related Work

2.1. Octree Graph Representation Method

Octrees are a hierarchical data structure allowing us to partition 3D space into cubic regions efficiently. The name "octree" comes from the fact that each node in the tree has up to eight children, corresponding to the octants of the cubic region represented by the node. Octrees are commonly used in computer graphics and geometric modeling to represent 3D objects and to perform spatial queries.

Octrees can be constructed recursively by dividing the 3D space into eight octants at each tree level. The process stops when a termination criterion is met, such as reaching a desired depth or when the number of points in a cell falls below a certain threshold. Each node in the octree represents a cubic region in 3D space and stores information such as the center of the cube and the number of points contained within the cube.

Octrees can be used to efficiently perform spatial queries, such as finding all points within a certain distance of a query point or finding the nearest neighbor of a query point. Additionally, octrees can be used to extract geometric features from 3D models, such as surface normals and curvature, by computing these features for each octant of the octree. The high efficiency of Octree makes it become valuable when the performance does matter. For example, previous work includes using octree to generate volumetric 3D outputs in a compute- and memory-efficient manner [3]. Similarly, octree has been applied to 3D model classification based on point cloud generation from octree [4]. This is also the paper that inspired us to attempt to extract different features from the model with octree to improve the classification.

In this paper, we use octrees to extract multiple types of geometric features from 3D models and to represent these features as a graph. We then use these graph representations to input a neural network for 3D model classification.

2.2. Graph Representation Classification

Graph representation is one type of method to represent 3D models, and this graph can be used to classify the model. Although there are plenty of approaches to generating graphs, there are generally two types of methods to classify these graphs. One widely used method is to apply the graph convolutional neural network [5] [6] [7]. It exploits the graph information to find features from every point's neighbors and uses the feature to build a convolutional neural network. During our literature search, we found no method using the multi-view images for classification, one of the traditional methods for 3D model classification. We want to test this method and see how well the multi-view images can utilize the features contained in the graph.

2.3. Texture Generation

Texture generation is one of the ongoing research directions for 3D deep learning. And often, the texture is generated through a diffusion model, in which the high-level idea is to add noise and then remove the noise through a deep learning model [8]. We find several publications in recent years' work in this field [9] [10] [11]. These models could generate texture using a text prompt and the provided 3D model. One significant contribution of these models is that they could identify different regions on the surface and apply different textures accordingly. In the example provided with Figure 1, we can see that the text2tex model finds the windows of a car and the zip of a bag and uses a different color than other regions [9]. However, we spent a lot of time reproducing the code and trying to use the model, but no luck since our dataset does not contain

vertex texture and a .mtl file, which prevents us from further testing. Alternatively, we used a texture-diffusion model that is fine-tuned on the stable diffusion model, which specifically generates a flat texture [12]. Then we applied a texture mapping using the generated image so our model could have texture on its surface.

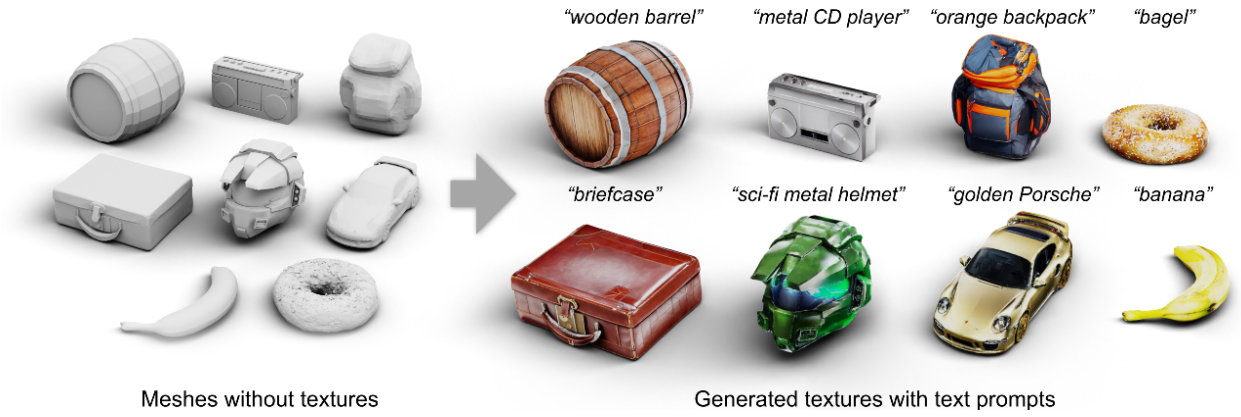

Figure 1. Examples of texture generation

3. Sampling Method

3.1. Main Approach

Our approach for extracting geometric features from 3D models is based on octree-based sampling. We first construct an octree to partition the 3D space into cubic regions, where each node in the tree represents a cubic region. For each leaf node in the octree that intersects with the input 3D model, we mark it as "moveable," inspired by pathfinding on the surface of models.

Next, we connect all the moveable leaf nodes in the octree to form a "path graph," where the nodes correspond to the moveable leaf nodes, and the edges represent the connectivity between them. Each path graph represents a disconnected component of the input 3D model. This allows us to separate multiple models in a scene and extract features for each individual model.

The path graph representation allows us to capture the surface information of the 3D models, which can be used to generate additional features for classification. We use path graph connectivity to compute additional features, such as the shortest path distance between two moveable leaf nodes and the number of neighbors for each moveable leaf node. These additional features capture the topology and connectivity of the 3D model and are combined with the geometric features to form a comprehensive feature vector for each path graph.

The resulting graph representation can be used as input to a neural network for 3D model classification. Our approach allows us to efficiently extract informative and discriminative features from 3D models. We will explain the details of the algorithm in the following sections.

3.2. Octree

The model/scene representation is splitting the space with an octree, marking the leaf nodes of the octree as moveable or not moveable. The size of an octree node is defined as the half-edge length of its corresponding cubic space. Then, the world size is defined as the size of the octree root. Each octree node contains the following information besides basic octree information (current layer, parent, children):

1. An R^3 Vector as the position of this node.
2. A Boolean value as if it is moveable.
3. A Boolean value as if it contains a moveable descendant.
4. A path graph node and edges reference.

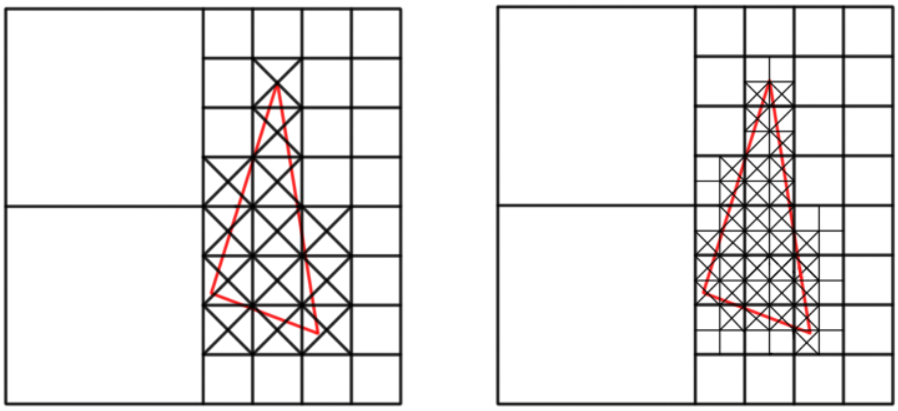

Figure 2. The different preferred layer of the same triangle

Then, the construction algorithm of adding models is as follows: for each model and for each triangular face of its mesh, all nodes which intersect with the triangle must be marked as moveable. The algorithm for checking the intersection between a triangle and a cube can be found in [13]. A depth or preferred layer needs to be specified to describe the mesh accurately. Figure 2 shows how a larger preferred layer improves accuracy. Obviously, more areas are marked as moveable on the left. The higher the preferred layer value, the more accurate the mesh representation and higher memory usage. The accuracy, which is the minimum size (edge length) of the cubic space, can be calculated as $a = 1 / 2 ^ {layer}$. For example, in a 1024 units world (edge length), to achieve accuracy with 1 unit around a mesh, 10 layers are required. Also, it is acceptable with two meshes that have different preferred layers but are added to the same graph. The following notations are used in pseudocode: *m* as the mesh, and *l* as the preferred layer, *n* as the octree node, and *t* as the triangle. The algorithm is shown as Figure 3 Algorithm 1.

Algorithm 1 Add Mesh

Require: $l \geq 0$ ▷ Layer must be positive integer

```
procedure AddMeshTriangle(n, t, l)
    if Intersect(n, t) then
        n.ContainsMoveable ← True
        if n.layer < l then
            InstantiateChildren(n)
            for all c ∈ n.Children do
                AddMeshTriangle(c, t, l)
            end for
        else
            n.Moveable ← True
        end if
    end if
end procedure
procedure AddMesh(m, l)
    for all t ∈ m.Triangles do
        AddMeshTriangle(root, t, l)
    end for
end procedure
```

Figure 3. Algorithm 1: Add mesh

3.3. Path Graph

After successfully representing the model/scene with an octree, an abstract graph is required to obtain features; an abstract graph can be created based on the octree, which is called a path graph in this paper. The idea is straightforward: leaf nodes must be adjacent to some other leaf nodes in the octree, so each moveable leaf node can be considered a vertex in the path graph, and the edges are added between each node and its adjacent neighbors. Each node is responsible for connecting six neighbors. There are two possible situations for all the neighbors Figure 4.

1. Its neighbors have the same size (depth) in the octree. In this simplest case, add edges directly, and each node will have six edges.
2. The node and its neighbors have different sizes (depth). There will be more than six edges for the larger node. However, the neighbors are defined as six nodes only. To solve this problem, all nodes with smaller sizes (greater depths) will build a connection edge to the larger one because they all have their unique neighbors set. Since all edges are

undirected, there is no difference between edges created by which node, so a larger node that did not connect itself to all others with edges created by it is acceptable.

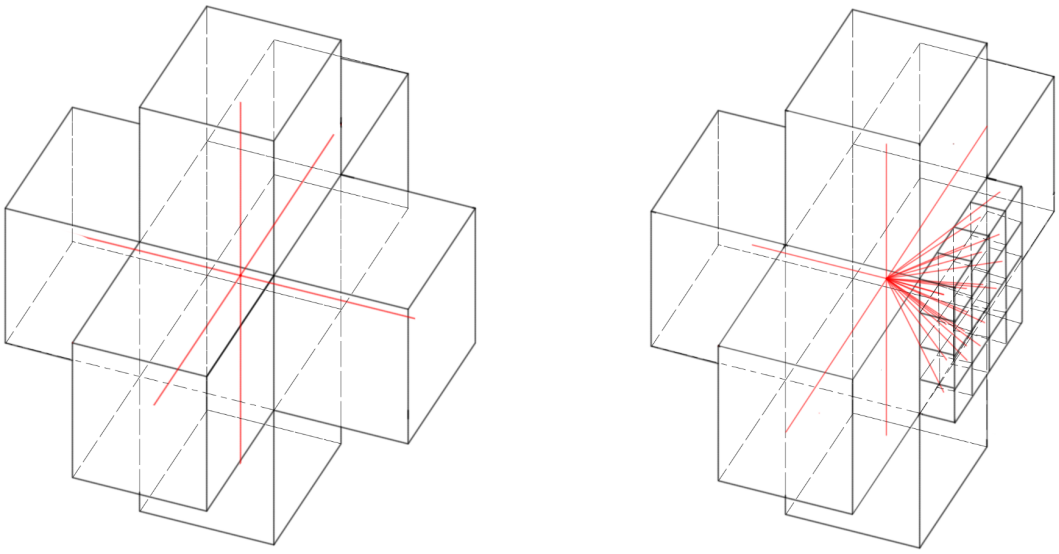

Figure 4. Path graph: (left) uniform; (right) different in size

Following notations are used in pseudocode: *PG* as the path graph. The algorithm is shown as Figure 5 Algorithm 2. After performing this algorithm, a path graph will be generated and ready to use in feature obtaining. There is no need to construct a path graph after each triangle insertion. It is possible to construct the path graph after adding all triangles because the insertion of triangles does not require a path graph. On the other hand, it will break the path graph.

The path graph now contains some disconnected components, which can be used to obtain features from multiple models in the same scene. For each disconnected component, we can run the feature-obtaining algorithm, which will be talked about in the next part of the paper. For a single mesh, we expect there will only be one component, Figure 6 gives an example.

Algorithm 2 Calculate Path Graph

```
1: procedure CALCULATEPATHGRAPH
2:     Initialize all path graph nodes:
3:     for all n ∈ octree.Leaves do
4:         if Not n.Moveable then
5:             n.PGNode ← NEW PGNODE
6:             n.PGEdge ← ∅
7:         end if
8:     end for
9:     Build edges between nodes:
10:    for all n ∈ octree.Leaves do
11:        if Not n.Moveable then
12:            for all p ∈ NEIGHBORS(n) do
13:                if p is Leaf then
14:                    e ← EDGE(N, P)
15:                    n.PGEdge ← n.PGEdge ∪ e
16:                    p.PGEdge ← p.PGEdge ∪ e
17:                end if
18:            end for
19:        end if
20:    end for
21: end procedure
```

Figure 5. Algorithm 2: Calculate path graph

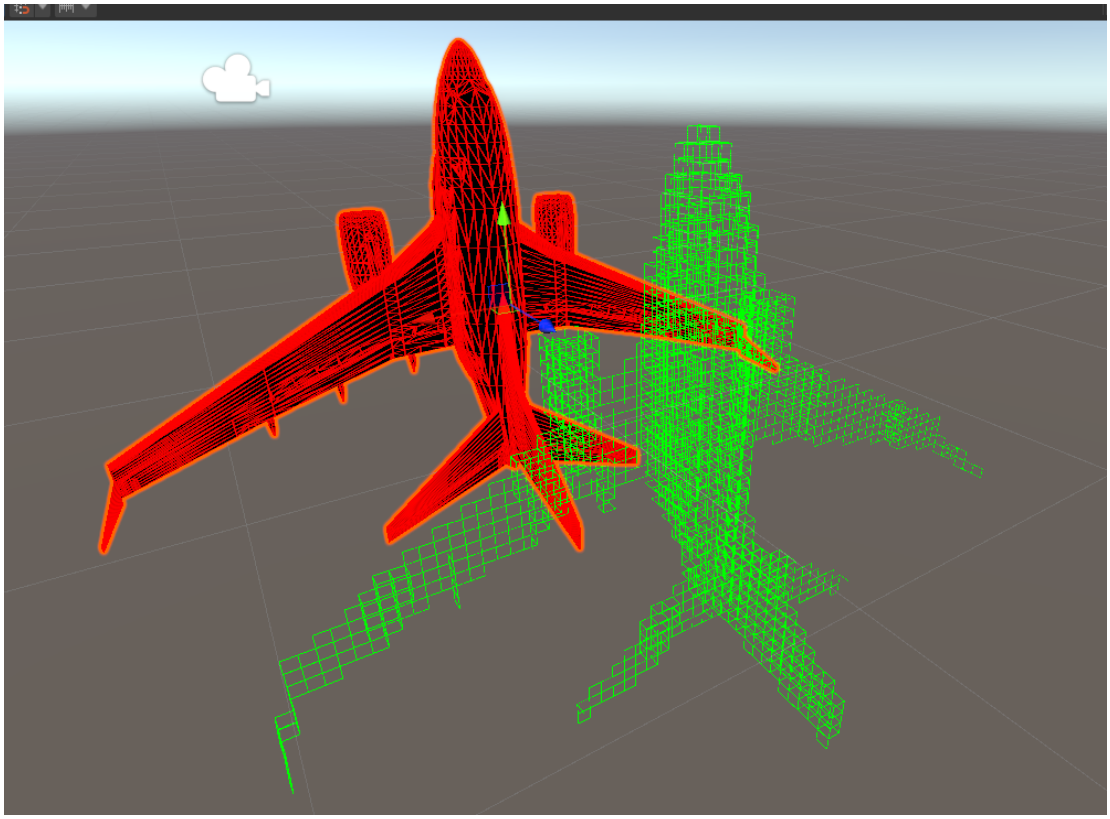

Figure 6. Path graph generated with uniform layer parameter of an airplane

3.4. Three-view Bitmap

The most straightforward feature of a component is that we can view it in three directions: positive x-axis, positive y-axis, and positive z-axis. This is known as three-view. For each vertex of the component in the path graph, we can get its position in space, then mark it in the three-view bitmap. This is simple to implement, but it is not rotation invariant. Figure 7 shows a three-view bitmap that simply maps the positions into the XY, XZ, and YZ planes of the coordinate system.

Figure 7. Three-view bitmap of an airplane

3.5. Rotation Invariant Bitmap

To overcome the issue of rotation of models, one approach is figuring out a solution to align the similar models in the same direction. One possible solution is using principal component analysis to rotate the model. Since we already obtained vertices of the path graph, we can consider it as a list of 3D data points and find three major variant directions by PCA analysis. Then we run the three-view bitmap algorithm on the rotated path graph to obtain the most representative features. Figure 8 shows a bitmap with a rotated path graph. The original model is not aligned with the XYZ axis. As models of the same type often share similar overall shapes, such as airplanes being long in the X direction (body) but thin in the Y direction (vertical tail fin), our approach can leverage this information to extract discriminative features for classification.

Figure 8. PCA rotated three-view bitmap of a rotated airplane

3.6. Adjacent Matrix Visualization

Another feature that can be obtained from the path graph is the adjacent matrix. Given a path graph for each pair of vertices, it is easy to determine whether they are connected and the distance between them. Additionally, the angle information is a key that can represent the geometry shape of the model. Let *a* and *b* become two vertices. Let *c* become the center of the path graph. The center can be calculated as the average position of all vertices. We can use three values to describe the relationship between two vertices:

1. The distance between *a* and *b*.
2. The angle *cab*.
3. The angle *cba*.

Note that all three values are rotation-invariant, which allows the same model of different rotations to provide the same feature information. Figure 9 shows a visualization of such a matrix, and we use three different color channels to represent these three values.

However, the attempt to train our model based on adjacent matrix visualization does not achieve a good result. A possible reason is that the matrix could not represent large features of the model

well, but only take care of small corners of the model. We conclude that this is not a good approach, hence we focus on bitmap representation in the following experiment.

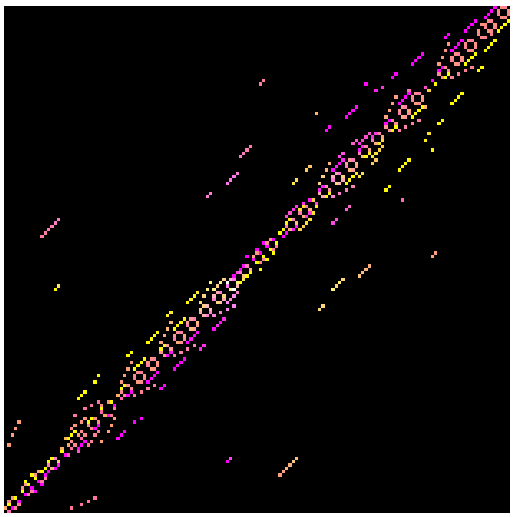

Figure 9. Visualized adjacent matrix

4. Classification Method

After obtaining the multiple-view bitmap representations from different angles using our octree-based graph representation approach, we apply deep learning techniques for classification. In this section, we introduce the model we use for classification and then discuss some challenges we faced during the process and how we overcame them.

4.1. Model Structure

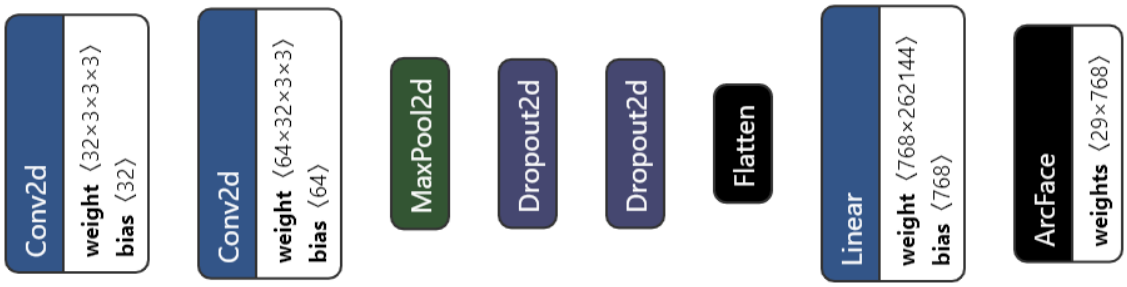

Figure 10. Model Structure

Our model architecture, depicted in Figure 10, comprises two convolutional layers and a max pooling layer. We use a linear layer followed by an Arcface layer for the output layers, which we will discuss in more detail later. To prevent overfitting, we employ a dropout function, which sets some elements of the model weights from the previous layer to 0 during training. Despite its simplicity, this CNN model has shown acceptable performance in our project, and we prioritize training speed by keeping the model simple and small.

One reason we use Arcface is to enhance the discriminative power on hard samples, which are defined as hard-to-classify samples. For example, classifying Xbox from the wardrobe is hard since they have a similar multi-view bitmap representation. Arcface considers the expected label and can learn to classify these hard samples. In our experiment, as shown later, the accuracy after applying Arcface improves by 5%, which is a significant improvement.

4.2. Loss Functions and Optimizer

In this section, we will discuss the loss functions, optimizer, and some other hyperparameters we consider in training, but we won't go into depth about this topic, and we will present the high-level ideas and challenges we encountered.

During training, a model's losses indicate the distance between its predicted and expected output. Therefore, a reliable performance measure is necessary to determine whether the model improves. Initially, we apply a cross-entropy loss and then improve it using focal loss to further improve the model's performance.

$$L_{\mathrm{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes,}$$

where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

Figure 11. Cross-entropy Loss Function

As described in Figure 11, it measures the difference between the predicted and actual probability distributions and aims to minimize this difference between them during training. As we discussed earlier, we encountered a problem with the hard samples. We eventually applied focal loss, an extension of cross-entropy loss that emphasizes hard examples during training, which increases the significant cross-entropy loss and reduces the trivial cross-entropy loss.

Optimizer aims to reduce the losses described earlier during training, and we use Adam optimizer, which is a widely used optimizer in deep learning. We try out different learning rates and find a suitable one so that the model converges in several iterations.

5. Texture Generation Method

In this section, we will show we generate texture on the surface of an object. Initially, the mesh we have does not contain any color information. In other words, we only have vertices and faces. Using vertices and faces only, we want to generate a texture and place it on the surface of this object. After we classify the object, we have high confidence about the class of the object, and we can use the label as a part of the prompt for the diffusion model to generate a texture image. Then, we will perform a texture mapping so that the object's surface has texture. As a result, we display the object using Open3D, a tool similar to OpenGL but with nearly no learning curve.

However, we don’t have time to save the object with the texture map, and displaying the object with texture will be our final step.

5.1. Texture Image Generation

As we discussed before, we have found two methods for texture generation. One is to use the prompt and the model as input, which would better identify different components on the object's surface. However, while we tried to reproduce the method shown in the paper and on Github, we encountered a problem: our data did not have a vertex texture and a Wavefront Material Template Library (.mtl) file. As a result, we switched to another method: we generated a texture using the prompt-to-image pre-trained diffusion model. Then we mapped this texture image onto the object's surface. This project used the pre-trained model "dream-textures/texture-diffusion" from hugging faces. This is the only model we found that could be used to generate texture. Although it can generate a good texture image for some prompts, like brick walls and roads, it doesn't work well for our case, as it doesn't generate a visually good texture image. We still use this model because we want to show a future direction. When the pre-trained model for generating texture images is sophisticated, it could be applied as a post-processing process for 3D model classification.

The following are a few examples of using this model. Note that "pbr" in the prompt means physically based rendering. We include this in the prompt because we want to generate texture.

5.2. UV mapping

The next step is to map the texture image on the object's surface, as we want to add realism to the 3D models. Although straightforward, our algorithm proves to be effective: we determine a vertex's position x and y coordinates on the texture image by dividing the x and y value by the

texture image's width or height. Once we acquire the vertex's position on the texture image, we can extract the texture's color at that point and assign it to the corresponding vertex on the object.

6. Experiment and Results

6.1. Dataset

We used the ModelNet40 dataset for our experiments, which contains 40 categories of different 3D models in .off file format. Each file includes information about the vertex position and faces of the model. Before is an image showing the distribution of different labels.

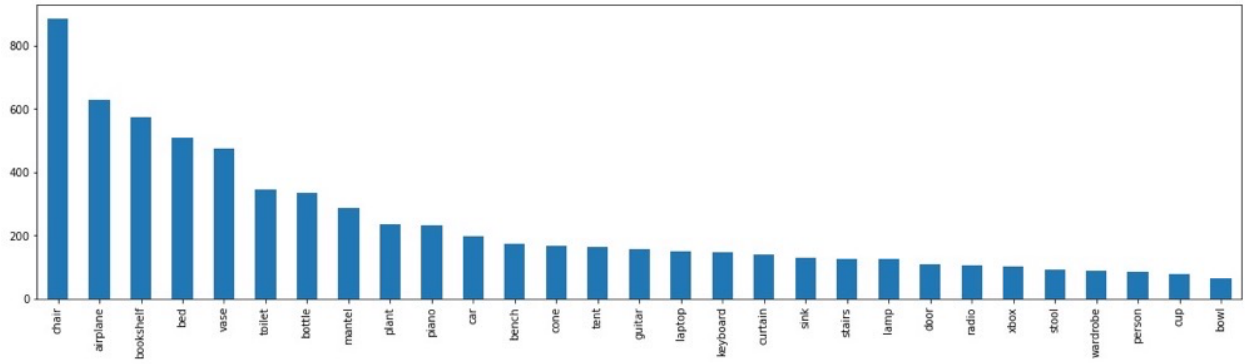

Figure 12. Dataset Label Distribution

We observe that there is a great variation in label distribution (unbalanced dataset), as shown in Figure 12, so we apply a stratified sampling method for splitting the dataset into training, validation, and test dataset, with 15% of the training dataset as the validation dataset.

6.2. Sampling Performance

We evaluate the performance (running time) of our octree-based sampling approach for 3D model classification. We evaluate our approach on a randomly generated triangle array. Additionally, we analyze the impact of different hyperparameters, such as the octree depth, on

the performance of our approach. Our results demonstrate the effectiveness and efficiency of our approach and provide insights for further improving the performance of our method. Figure 13 shows the relationship between the number of triangles and the time required to construct the path graph.

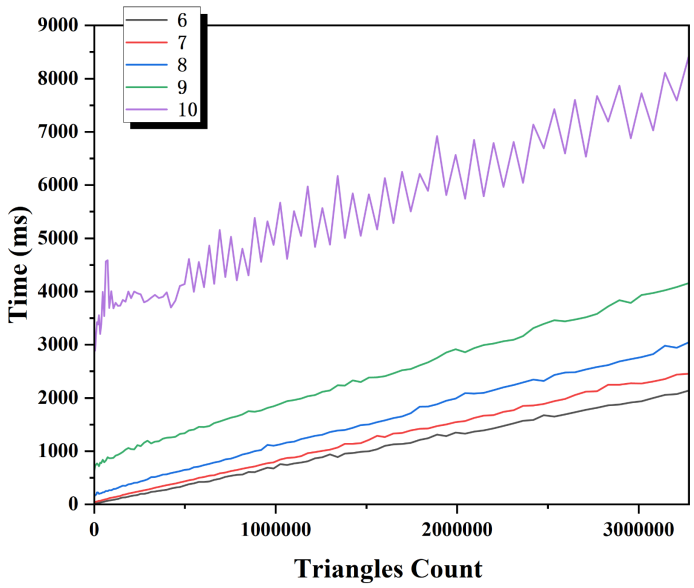

Figure 13. Time used versus triangles counts with different preferred layers.

6.3. Classification Experiment

We test our classification on layer five graph representation and layer seven representation of the modelNet40 dataset. Although the time for generating a graph representation for different layers varies greatly, our multi-view bitmap is the same with 128x128 resolution. Therefore, there is no difference in training time and inference time.

We trained the model using Google Colab with the NVIDIA Tesla P100 GPU resource, and the model took 61 seconds for each training iteration. Overall, we spent less than 20 minutes training a model, and only after a few iterations did the model converge, as shown in Figure 14.

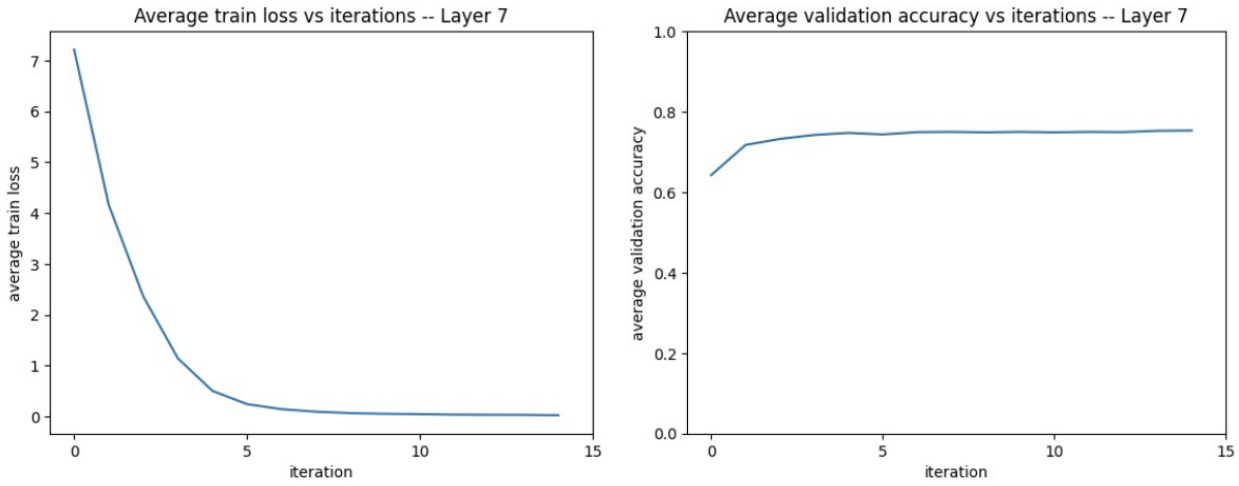

Figure 14. Average training loss and validation accuracy v.s. Iterations for 7-layer graph representation classification

Next, let's look at the classification result, discuss how the model performs on different classes, and analyze potential reasons associated with the result. We use the test dataset to evaluate the classification result, which is completely isolated from the training process and can be viewed as unseen data. First, we compute the accuracy, and the results are: for the 5-layer, the accuracy is 0.660, and for the 7-layer, the accuracy is 0.735. We see that there is a great improvement from the 5-layer to the 7-layer because the 7-layer graph representation is much more complex than the one of the 5-layer, but we also need to consider the running time as generating the 7-layer graph representation requires significantly more amount of time (shown in the section of Sampling Performance).

Finally, we compute the confusion matrix of the 7-layer and visualize it with the following heat map Figure 15.

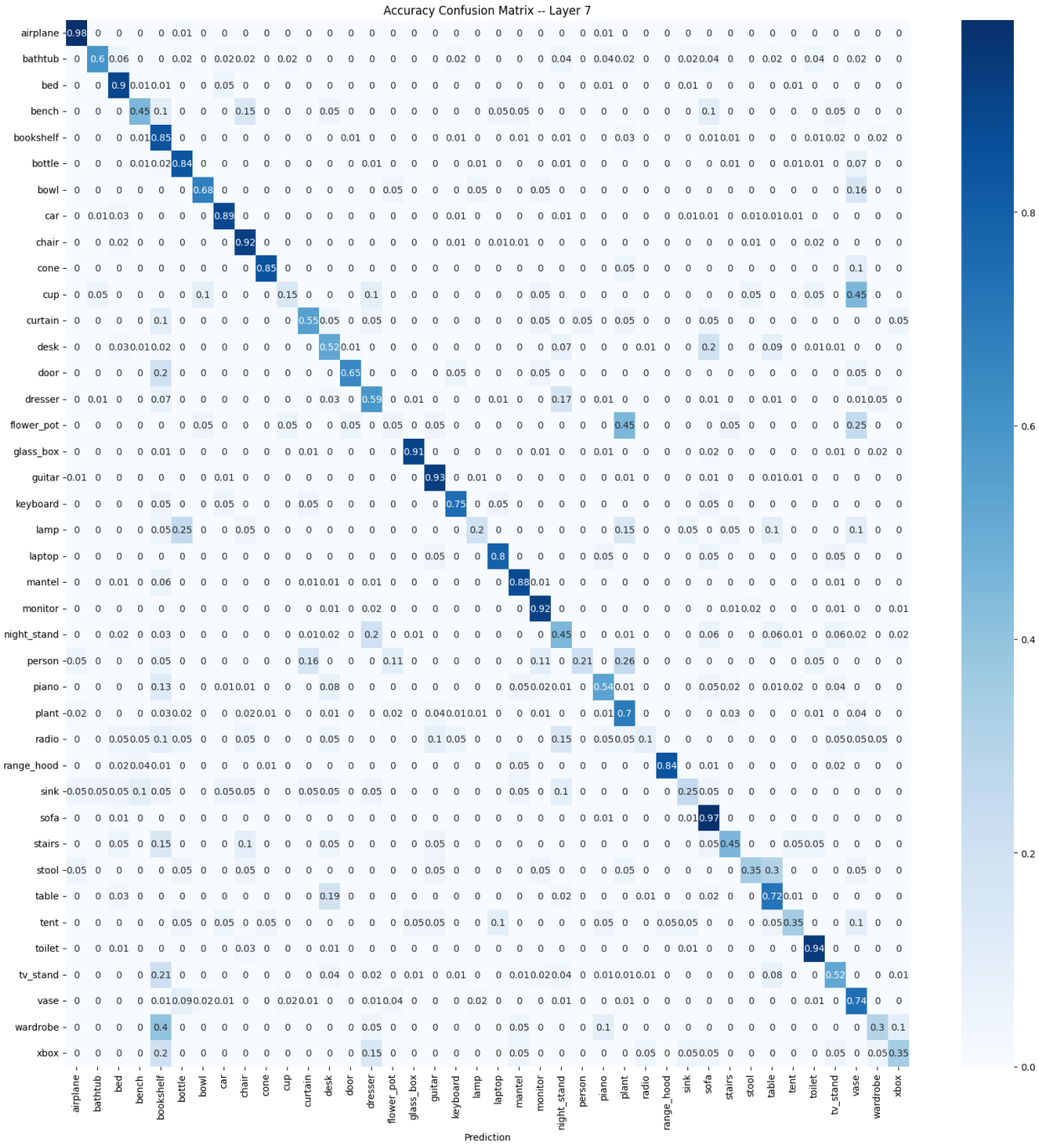

Figure 15. Accuracy confusion matrix for 7-layer graph representation

Checking the heat map above, we can interpret how the model performs on different classes of 3D models, and then we will explain why the model performs well on some classes while not on others.

We will begin to analyze the classes in which our model works well. Checking the accuracy shown in the confusion matrix, we can find that airplane (0.98), sofa (0.97), toilet (0.94), guitar (0.93) achieve a decent performance while flower_pot(0.05), cup (0.15), person (0.21) and sink (0.25). Reviewing the bitmaps (provided in supplemental materials) for each class, we can observe that the classes with a high classification accuracy have a unique shape, and it is similar among all samples. While for classes with a bad performance, the reason is either the shape is very similar to other classes or the shape is not quite similar for the same label.

6.4. Texture Generation Experiment

To test out how the texture generation performs, we will first generate texture images for all labels, so we iteratively use the prompt of every label and use the texture diffusion model to generate a texture image, such as Figure 16.

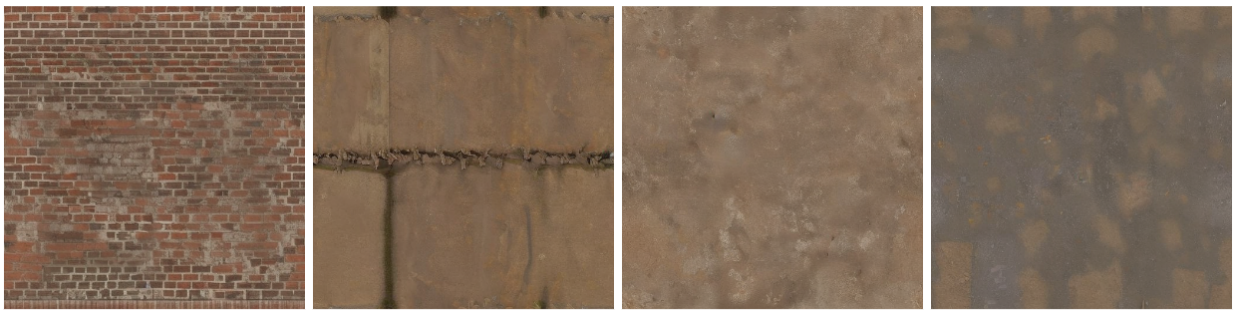

Figure 16. From left to right, the prompts are "pbr brick wall," "pbr wardrobe," "pbr vase," and "pbr person."

As we observe from the above selective results and the associated prompts, the results for the surface texture are not so good, and sometimes the model generates a totally unrelated image for our task. For example, if our prompt is "pbr person," we expect the generated texture image to be human skin, but the output texture image is definitely not human skin. There are two potential reasons. One is that this model may not be suitable for our task to generate texture on the surface

of a model. The examples provided using this model generate textures for brick walls, as shown in the example images, so it is possible that this model only works for specific flat surfaces. The other reason might be that our prompt needs to be descriptive instead of using the object's name because the descriptive texture surface will be inferred from the object's name.

The next step is to apply this texture on the object's surface to add "realism," but since the texture image generated is not real and even possibly mismatched, this step shows that different 3D models would have different textures. Below is an example where we apply the texture image on the surface of an airplane, as shown in Figure 17.

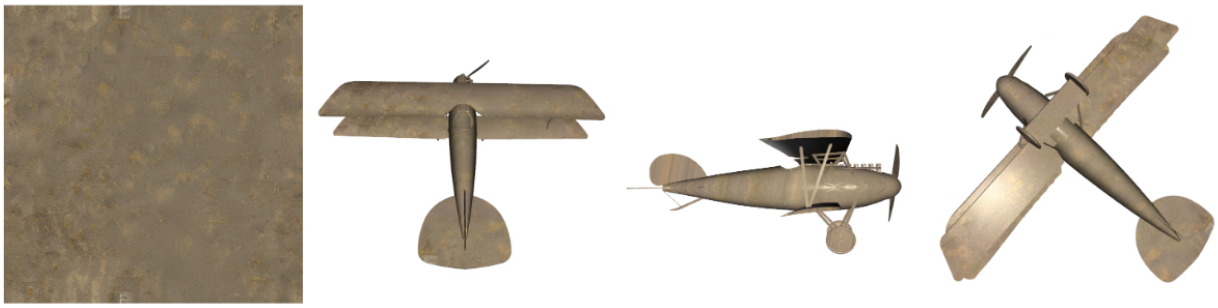

Figure 17. The first image is the texture image generated using the prompt "pbr airplane," the rest are how the airplane looks from different angles

As the example shows, the result is pleasing: we observe the texture image successfully shown on the model's surface.

7. Conclusion

In summary, we have presented an approach for 3D model classification using an octree-based graph representation. Our approach extracts multiple types of geometric features from 3D models using the octree data structure and represents them as graphs, which are then used as input to a neural network for classification. We have demonstrated that our approach can

improve classification accuracy. Additionally, we have shown a practical application of our approach in texture generation, which can be used to enhance the visual appearance of 3D models. Our approach provides a promising direction for future research in 3D model classification, with potential applications in various fields such as gaming, room design, and 3D reconstruction. Our project is open source on Github, which allows others to use and build upon our work. Overall, our approach shows the potential of using the octree-based graph representation for 3D model classification and highlights the importance of exploring different types of features for improving accuracy.

There are several avenues for further improving the performance of our approach, as well as exploring new applications of our method. One direction for future work is to investigate the impact of different hyperparameters in our approach, such as the octree depth and the number of points sampled per leaf node. Optimizing these hyperparameters can potentially improve the accuracy of our approach and make it more efficient. Another direction for future work is to explore the use of different types of neural networks for 3D model classification. While we have used a convolutional neural network (CNN) in our approach, other types of networks, such as graph neural networks (GNNs) or attention-based models, may be more suitable for processing graph representations.

Reference

[1] Deng J, Guo J, Xue N, Zafeiriou S. Arcface: Additive angular margin loss for deep face recognition. InProceedings of the IEEE/CVF conference on computer vision and pattern recognition 2019 (pp. 4690-4699).

[2] Lin TY, Goyal P, Girshick R, He K, Dollár P. Focal loss for dense object detection. InProceedings of the IEEE international conference on computer vision 2017 (pp. 2980-2988).

[3] Maxim Tatarchenko, Alexey Dosovitskiy, Thomas Brox, "Octree Generating Networks: Efficient Convolutional Architectures for High-resolution 3D Outputs," in ICCV 2017, 2017

[4] Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, Xin Tong, "O-CNN: Octree-based Convolutional Neural Networks for 3D Shape Analysis," in ACM Transactions on Graphics, Vol. 36, No. 4, Article 72, 2017

[5] Te G, Hu W, Zheng A, Guo Z. Rgcnn: Regularized graph cnn for point cloud segmentation. InProceedings of the 26th ACM international conference on Multimedia 2018 Oct 15 (pp. 746-754).

[6] Zhang Y, Rabbat M. A graph-cnn for 3d point cloud classification. In2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2018 Apr 15 (pp. 6279-6283). IEEE.

[7] Wang Y, Sun Y, Liu Z, Sarma SE, Bronstein MM, Solomon JM. Dynamic graph cnn for learning on point clouds. Acm Transactions On Graphics (tog). 2019 Oct 10;38(5):1-2.

[8] Rombach R, Blattmann A, Lorenz D, Esser P, Ommer B. High-resolution image synthesis with latent diffusion models. InProceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition 2022 (pp. 10684-10695).

[9] Chen DZ, Siddiqui Y, Lee HY, Tulyakov S, Nießner M. Text2Tex: Text-driven Texture Synthesis via Diffusion Models. arXiv preprint arXiv:2303.11396. 2023 Mar 20.

[10] Khalid NM, Xie T, Belilovsky E, Popa T. Clip-mesh: Generating textured meshes from text using pretrained image-text models. SIGGRAPH Asia 2022 Conference Papers 2022 Dec.

[11] Metzer G, Richardson E, Patashnik O, Giryes R, Cohen-Or D. Latent-NeRF for Shape-Guided Generation of 3D Shapes and Textures. arXiv preprint arXiv:2211.07600. 2022 Nov 14.

[12] Texture Diffusion https://huggingface.co/dream-textures/texture-diffusion

[13] T. V. Christensen and S. Karlsson, “High performance triangle versus box intersection checks,” in 2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010