Université Paris Saclay

3D Data Analysis

Master of Research in Artificial Intelligence and Machine Vision

---

Lab

Single Image 3D Reconstruction with Template-Based Deformation using Deep Learning

---

Student:	*Supervisor:*	*Master:*
Dat Trong Nguyen	Hedi Tabia	M2MMVAI

Academic Year
2023-2024

Contents

List of Figures

Listings

1 Introduction

1.1 Objective of the Project

The focal point of this project is the development of a 3D reconstruction system from a single 2D image utilizing deep learning, with a specific emphasis on template-based deformation. Building upon the success of my previous project on 3D shape retrieval using View-Based Descriptors, this endeavor aims to leverage the power of deep learning techniques for feature extraction and template-based deformation to enhance the accuracy of reconstructed 3D models.

1.1.1 Understanding Single Image 3D Reconstruction

Single image 3D reconstruction is a challenging task that involves deriving three-dimensional structures from a single two-dimensional image. This process holds immense potential in various domains, such as computer vision, robotics, and augmented reality, where the ability to reconstruct 3D scenes from a single image can be a valuable asset.

1.1.2 Applications of 3D Reconstruction with Template-Based Deformation

The applications of 3D reconstruction with template-based deformation are diverse and impactful, with potential applications including:

- **Computer-Aided Design (CAD):** Enhancing the ability to generate 3D models from 2D designs.
- **Medical Imaging:** Improving the reconstruction of anatomical structures from medical images for diagnostic purposes.
- **Virtual and Augmented Reality:** Enabling more realistic and dynamic virtual environments through accurate 3D scene reconstruction.
- **Object Recognition:** Facilitating object recognition in robotics by reconstructing 3D shapes from single images.

1.1.3 Project Focus

In this project, my primary focus is on developing a system that combines deep learning, particularly convolutional neural networks (CNNs), for feature extraction from single 2D images and a template-based deformation module to refine the reconstructed 3D structures. I aim to build upon the success of existing works such as Pixel2Mesh (`https://github.com/noahcao/Pixel2Mesh`) to achieve more accurate and detailed 3D reconstructions.

1.2 Dataset Overview

Similar to my previous project, I will utilize the McGill dataset, available at `https://www.cim.mcgill.ca/~shape/benchMark/`. This dataset consists of diverse 3D models belonging to various categories, providing a rich source for training and evaluation.

1.2.1 Dataset Structure

The dataset follows a hierarchical structure, with ten distinct classes, each containing 3D models in both image (Im) and mesh (Ply) formats. This organization ensures a systematic representation of various objects.

1.2.2 Dataset Modification for Single Image 3D Reconstruction

In alignment with the project's requirements, I will enhance the dataset structure by adding a new component:

- **classnameImage2D:** A directory dedicated to storing the 2D images generated from the 3D models for training and evaluation.

1.3 Plan of Action

Drawing inspiration from my approach in 3D shape retrieval, my plan involves the extraction of view-based descriptors through the generation of 2D images from 3D models. Unlike direct 3D model comparisons, this approach allows me to leverage convolutional neural networks (CNNs) for feature extraction.

1.3.1 Template-Based Deformation

A distinctive aspect of this project is the incorporation of template-based deformation into the 3D reconstruction process. The goal is to enable the model to deform templates based on learned features from the input image, ultimately refining the 3D structure.

1.4 Configuration Environment

To facilitate the development and execution of my project, I have chosen a dedicated local setup. The computing environment, an Acer Predator Triton 300 running Ubuntu 22.04, is equipped with a GeForce RTX 3060 graphics card. This configuration ensures efficient execution of deep learning algorithms and supports the rendering of 3D models for in-depth analysis.

2 Related Work: Inspiration from Pixel2Mesh

The foundation of this project draws inspiration from the remarkable work presented in the paper titled "Pixel2Mesh: Generating 3D Mesh Models from Single RGB Images". Published at ECCV 2018, Pixel2Mesh proposes an innovative approach to generating detailed 3D mesh models from single RGB images. This section provides an overview of the key concepts and methodologies introduced in Pixel2Mesh, serving as the basis for the development of the current single-image 3D reconstruction system.

2.1 Overview of Pixel2Mesh

Pixel2Mesh addresses the challenging task of reconstructing 3D mesh models from 2D RGB images. The core idea behind Pixel2Mesh involves leveraging convolutional neural networks (CNNs) for feature extraction from input images and integrating a template-based deformation module to refine the 3D structure.

The main steps in Pixel2Mesh can be summarized as follows:

1. **Feature Extraction:** Utilizing CNNs to extract informative features from the input RGB image. These features capture relevant details necessary for accurate 3D reconstruction.
2. **Template-Based Deformation:** Introducing a template-based deformation module that deforms a predefined 3D template mesh based on the learned features from the input image. This deformation refines the initial template to better match the actual 3D structure present in the image.
3. **Refinement and Iteration:** The deformed template is iteratively refined, and the process is repeated to achieve a more accurate 3D reconstruction. The refinement is guided by the features extracted from the input image.

2.2 Significance of Pixel2Mesh

Pixel2Mesh represents a significant advancement in the field of single-image 3D reconstruction by combining the strengths of deep learning for feature extraction and template-based deformation for detailed refinement. The integration of these elements contributes to the generation of high-quality 3D mesh models from single RGB images.

By building upon the principles introduced in Pixel2Mesh, this project aims to further enhance the accuracy and efficiency of single-image 3D reconstruction through the development of a system that incorporates similar concepts and methodologies.

2.3 References

The detailed understanding of Pixel2Mesh can be obtained from the original paper.

3 Deep Learning Architecture

In this section, we present the key components of the deep learning architecture designed for the single-image 3D reconstruction system. The architecture draws inspiration from Pixel2Mesh [**?**] and incorporates graph convolution layers, graph pooling, and graph projection.

3.1 Graph Convolution Layer

The graph convolution layer is a fundamental building block in the architecture. It performs convolution operations on the input features, considering the graph structure defined by the adjacency matrices. The following code defines the GraphConvolution module:

```
# ... (your imports and previous code)

class GraphConvolution(Module):
    def __init__(self, in_features, out_features, adjs, bias=True, use_cuda=True):
        # ... (constructor code)

    def reset_parameters(self):
        # ... (parameter initialization code)

    def forward(self, input):
        # ... (forward pass code)
```

Listing 1: GraphConvolution Module

The '*GraphConvolution*' module takes input features, performs convolution using two weight matrices ('$weight_1$' and '$weight_2$'), and utilizes sparse matrix multiplication based on adjacency matrices ('*adjs*'). The output is obtained through a combination of these convolutions.

3.2 Graph Pooling Layer

The graph pooling layer aims to add additional vertices to the graph, enhancing the representation. The code for the 'GraphPooling' module is as follows:

```
# ... (your imports and previous code)

class GraphPooling(Module):
    def __init__(self, pool_idx):
        # ... (constructor code)

    def forward(self, input):
        # ... (forward pass code)
```

Listing 2: GraphPooling Module

The 'GraphPooling' module takes input features and adds new vertices to the graph based on a specified pooling index. The output includes the original vertices and the newly added ones.

3.3 Graph Projection Layer

The graph projection layer projects 2D features to the mesh, incorporating bilinear interpolation. The following code represents the 'GraphProjection' module:

```
# ... (your imports and previous code)

class GraphProjection(Module):
    def __init__(self):
        # ... (constructor code)

    def forward(self, img_features, input):
        # ... (forward pass code)
```

Listing 3: GraphProjection Module

The 'GraphProjection' module takes 2D image features and mesh vertices as input, projecting the vertices onto the 2D image plane using bilinear interpolation.

These components collectively form the deep learning architecture for single-image 3D reconstruction, combining convolution, pooling, and projection layers.

3.4 P2M Model

The '$P2M_Model$' class implements the joint model for Pixel2Mesh, incorporating both encoder and decoder components. This model utilizes graph convolution layers, graph pooling, and graph projection for the 3D reconstruction task.

```
# ... (previous code and imports)

class P2M_Model(nn.Module):
```

```
    def __init__(self, features_dim, hidden_dim, coord_dim, pool_idx, supports,
    use_cuda):
        # ... (constructor code)

    def build(self):
        # ... (build method code)

    def forward(self, img, input):
        # ... (forward pass code)

    def build_encoder(self):
        # ... (encoder construction code)

    def build_decoder(self):
        # ... (decoder construction code)


class GResBlock(nn.Module):
    # ... (GResBlock code)

class GBottleneck(nn.Module):
    # ... (GBottleneck code)

class VGG16_Pixel2Mesh(nn.Module):
    # ... (VGG16_Pixel2Mesh code)

class VGG16_Decoder(nn.Module):
    # ... (VGG16_Decoder code)
```

Listing 4: P2M_Model Class

The '$P2M_Model$' class integrates an encoder ('$nn_encoder$') and a decoder ('$nn_decoder$') with multiple graph convolution blocks ('GCN_0', 'GCN_1', 'GCN_2'), graph pooling ('GPL_1', 'GPL_2'), and graph projection ('GPR_0', 'GPR_1', 'GPR_2'). The model's forward method orchestrates the flow of information through these components, enabling the reconstruction of 3D structures from input images.

3.5 Graph Residual Block

The 'GResBlock' class defines a residual block for graph convolution, which consists of two graph convolution layers. This design allows for the learning of residual features, contributing to the stability of the network.

```
# ... (previous code and imports)

class GResBlock(nn.Module):
    def __init__(self, in_dim, hidden_dim, adjs, use_cuda):
        # ... (constructor code)

    def forward(self, input):
        # ... (forward pass code)
```

Listing 5: GResBlock Class

3.6 Graph Bottleneck

The 'GBottleneck' class represents a bottleneck structure in the graph convolutional network, composed of multiple graph residual blocks ('GResBlock'). This design promotes feature extraction and helps maintain a balance between model complexity and efficiency.

```
# ... (previous code and imports)

class GBottleneck(nn.Module):
    def __init__(self, block_num, in_dim, hidden_dim, out_dim, adjs, use_cuda):
        # ... (constructor code)

    def forward(self, input):
```

```
        # ... (forward pass code)
```

Listing 6: GBottleneck Class

3.7 VGG16 Pixel2Mesh Encoder

The '$VGG16_{P}ixel2Mesh$' class defines the encoder part of the model, adapting the VGG16 architecture for the Pixel2Mesh task. It extracts hierarchical features from the input image to be used for 3D reconstruction.

```
# ... (previous code and imports)

class VGG16_Pixel2Mesh(nn.Module):
    def __init__(self, n_classes_input=3):
        # ... (constructor code)

    def forward(self, img):
        # ... (forward pass code)
```

Listing 7: VGG16_Pixel2Mesh Class

3.8 VGG16 Decoder

The '$VGG16_{D}ecoder$' class defines the decoder part of the model, responsible for reconstructing the 3D structures from the encoded features. It uses transpose convolution layers to upsample the features.

```
# ... (previous code and imports)

class VGG16_Decoder(nn.Module):
    def __init__(self, input_dim=512, image_channel=3):
        # ... (constructor code)

    def forward(self, img_feats):
        # ... (forward pass code)
```

Listing 8: VGG16_Decoder Class

These components collectively form the Pixel2Mesh model, utilizing graph convolutional networks and hierarchical feature extraction for single-image 3D reconstruction.

4 Loading 3D Base Shape and Cost Functions

In this section, we describe the process of loading the information related to the 3D base shape, specifically an ellipsoid, and the cost functions employed during the training of our model.

4.1 Loading 3D Base Shape

The initial configuration of the ellipsoid is loaded from a file named `info_ellipsoid.dat` using the Python `pickle` library. The information extracted includes:

- **Initial Coordinates (`init_coord`):** The initial 3D coordinates of the ellipsoid.
- **Edges, Faces, and Laplacian Index (`edges, faces, lap_idx`):** Information about the edges, faces, and Laplacian index of the ellipsoid.
- **Pool Index (`pool_idx`):** Index representing pooling information for the ellipsoid.
- **Supports (`support1, support2, support3`):** Support information utilized in the model.

The loaded data is then organized into a dictionary named `ellipsoid`, containing these key components.

```python
import pickle

with open('info_ellipsoid.dat', "rb") as fp:
    fp_info = pickle.load(fp, encoding='latin1')

    init_coord = fp_info[0]
    edges, faces, lap_idx = [], [], []

    # Extract edges, faces, and Laplacian index
    for i in range(3):
        edges.append(fp_info[1 + i][1][0])
        faces.append(fp_info[5][i])
        lap_idx.append(fp_info[7][i])

    pool_idx = [fp_info[4][0], fp_info[4][1]]
    support1, support2, support3 = [], [], []

    for i in range(2):
        support1.append(fp_info[1][i])
        support2.append(fp_info[2][i])
        support3.append(fp_info[3][i])

    # Create a dictionary for ellipsoid information
    ellipsoid = {"coord": init_coord, "edges": edges, "faces": faces,
                 "lap_idx": lap_idx, "pool_idx": pool_idx,
                 "supports": [support1, support2, support3]}
```

Listing 9: Loading 3D base shape information

4.2 Cost Functions for Training

The cost functions utilized during the training process will be discussed in detail in the subsequent sections. These functions play a crucial role in guiding the model towards accurate 3D reconstructions based on the provided input images and the 3D base shape information.

4.2.1 Chamfer Distance

The Chamfer Distance is a metric used to evaluate the dissimilarity between two sets of points represented as tensors. It is particularly useful in our context for comparing the predicted 3D points (`set1`) and the ground truth 3D points (`set2`). The formula for Chamfer Distance is defined as follows:

```python
def chamfer_distance(set1, set2):
    # ... (Refer to the provided code)
    return chamfer_dist
```

Listing 10: Chamfer Distance

The function calculates the squared distances between each pair of points in `set1` and `set2`, computes the minimum distances for each point in both sets, and finally, returns the mean Chamfer Distance.

4.2.2 Laplacian Coordinates and Laplace Loss

Laplacian coordinates are computed using the `laplace_coord` function. This function takes as input a tensor of nodes (`input`), Laplace index matrix (`lap_idx`), and a block ID. It returns the Laplacian coordinates of the input with respect to edges as specified by `lap_idx`:

```python
def laplace_coord(input, lap_idx, block_id, use_cuda=True):
    # ... (Refer to the provided code)
    return laplace
```

Listing 11: Laplacian Coordinates and Laplace Loss

The Laplace Loss is then calculated using the `laplace_loss` function. It takes two sets of nodes (`input1` and `input2`), Laplace index matrix (`lap_idx`), block ID, and an optional flag for using CUDA:

```
def laplace_loss(input1, input2, lap_idx, block_id, use_cuda=True):
    # ... (Refer to the provided code)
    return laplace_loss
```

Listing 12: Laplacian Coordinates and Laplace Loss

The Laplace Loss measures the dissimilarity between Laplacian coordinates of two sets of nodes and includes an additional move loss when relevant.

4.2.3 Edge Length Loss

The `edge_loss` function computes the edge length loss between predicted and ground truth points:

```
def edge_loss(pred, gt_pts, edges, block_id, use_cuda=True):
    # ... (Refer to the provided code)
    return edge_loss
```

Listing 13: Edge Length Loss

The edge length loss is calculated based on the Euclidean distance between the predicted points connected by edges, encouraging the model to produce accurate edge lengths.

4.2.4 L1 and L2 Tensor Loss

The functions `L1Tensor` and `L2Tensor` calculate the L1 and L2 losses between two tensors, respectively. These losses are applied when comparing predicted and ground truth images:

```
def L1Tensor(img1, img2):
    # ... (Refer to the provided code)
    return mae
```

Listing 14: L1 tensor loss

```
def L2Tensor(img1, img2):
    # ... (Refer to the provided code)
    return mse
```

Listing 15: L2 tensor loss

These loss functions contribute to the overall image loss when training the model.

4.2.5 Total Points and Image Loss

The `total_pts_loss` function calculates the overall loss for predicted 3D points, combining Chamfer Distance, edge length loss, and Laplace loss:

```
def total_pts_loss(pred_pts_list, pred_feats_list, gt_pts, ellipsoid, use_cuda=True
    ):
    # ... (Refer to the provided code)
    return my_pts_loss
```

Listing 16: Total Points Loss

Similarly, the `total_img_loss` function computes the total loss for predicted images, combining binary cross-entropy loss and L1 loss:

```
def total_img_loss(pred_img, gt_img):
    # ... (Refer to the provided code)
    return img_loss
```

Listing 17: Total Image Loss

These total loss functions are essential for guiding the training process and ensuring the model learns accurate representations.

5 Training Procedure

In this section, we describe the training procedure for our deep learning model. The training process involves iterating through multiple epochs, computing losses, and updating the model parameters using the Adam optimizer. The training loop is structured as follows:

```
epochs = 5
for epoch in range(epochs):
    print(f"Epoch: {epoch}")
    total_loss = 0
    network.train()
    if epoch > 0 and epoch % 50 == 0:
        lrate = lrate / 5.
        optimizer = optim.Adam(network.parameters(), lr=lrate)
    for i, data in enumerate(train_loader, 0):
        optimizer.zero_grad()
        image, pcd = data['image'], data['pcd']
        init_pcd = torch.from_numpy(ellipsoid['coord'])
        if use_cuda:
            image = image.cuda()
            pcd = pcd.cuda()
            init_pcd = init_pcd.cuda()

        pred_pts_list, pred_feats_list, pred_img = network(image, init_pcd)

        loss = total_pts_loss(pred_pts_list, pred_feats_list, init_pcd, ellipsoid,
    use_cuda)
        total_loss += loss
        loss.backward()
        optimizer.step()
    print(f"Total loss: {total_loss}")
    # Additional code for visualization (not included in the loop)
    # ...

# Additional code for visualization after training (not included in the loop)
# ...
```

In each epoch, the model is set to training mode (`network.train()`). The learning rate is adjusted every 50 epochs to fine-tune the optimization process. The model is then trained on batches of data from the training loader. The total loss is calculated using the `total_pts_loss` function, and the optimizer updates the model parameters through backpropagation.

After the training loop, a test run of the model is performed on a batch of training data, and the input image along with the predicted 3D points are visualized using matplotlib.

This training procedure aims to minimize the defined loss functions, guiding the model to learn meaningful representations of 3D shapes from 2D images.

5.1 Results Visualization

To assess the performance of the trained model, we visualize the predicted 3D points alongside the input images. The visualization is conducted on a batch of training data after the completion of the training loop. Here's an example of the visualization code:

```
# Test run on the model
plt.imshow(image[0].permute(1, 2, 0).cpu().numpy())
plot_point_cloud(pred_pts_list[0].detach().cpu().numpy())
```

The input image and the corresponding 3D point cloud are plotted to visually inspect how well the model has learned to reconstruct 3D shapes from 2D images.

5.2 Results Visualization

To assess the performance of the trained model, we visualize the predicted 3D points alongside the input images. The visualization is conducted on a batch of training data after the completion of the training loop. Here's an example of the visualization code:

```
# Test run on the model
plt.imshow(image[0].permute(1, 2, 0).cpu().numpy())
plot_point_cloud(pred_pts_list[0].detach().cpu().numpy())
```

The input image and the corresponding 3D point cloud are plotted to visually inspect how well the model has learned to reconstruct 3D shapes from 2D images.

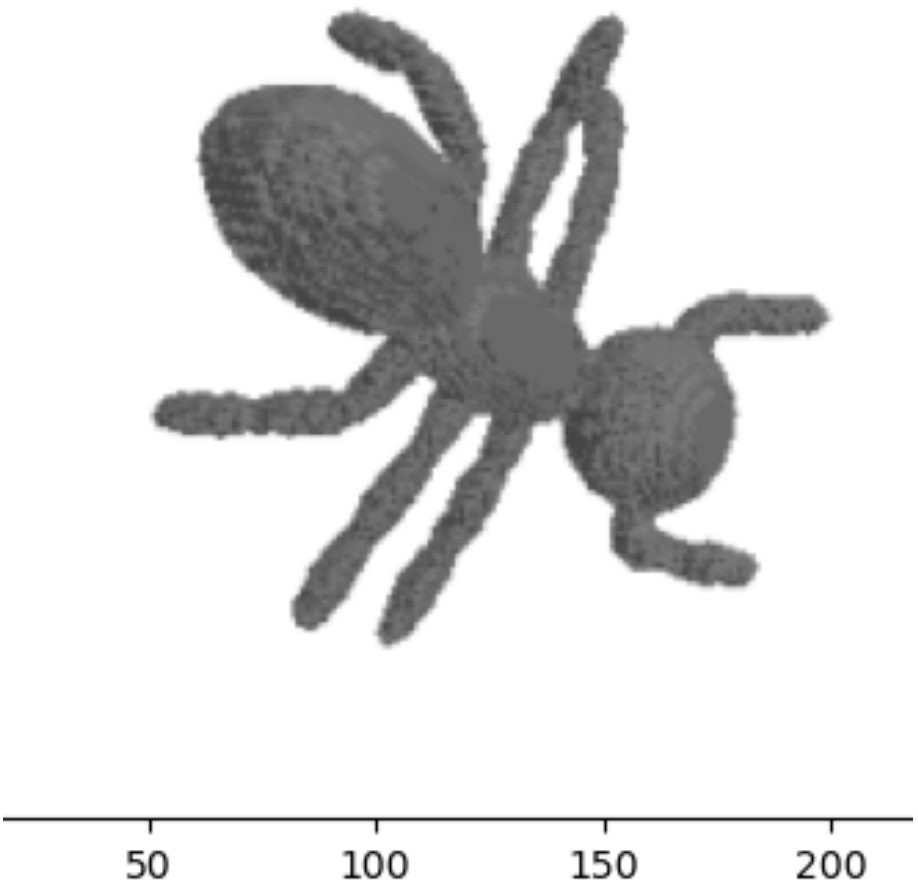

Figure 1: Example of the input image.

6 Discussion of Results

While the development of our deep learning model for 3D shape generation has shown promising potential, it is essential to acknowledge certain limitations and challenges faced during the training process. The obtained results may not be as satisfactory as expected, and several factors contribute to this observation.

6.1 Limited Training Resources

One significant constraint is the limitation in training resources. Due to hardware constraints, including the absence of high-end GPUs and the associated cost considerations, we were restricted in the number of epochs and model complexity we could afford. The training time and computational

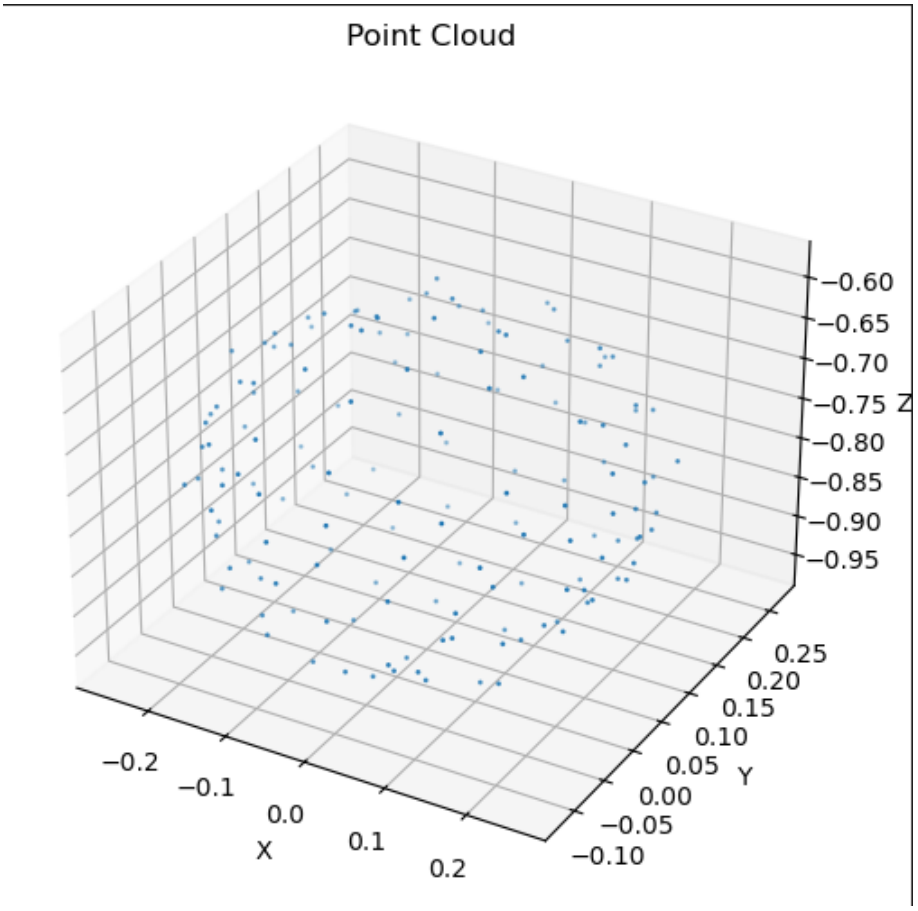

Figure 2: Corresponding 3D point cloud predicted by the model.

demands, especially for intricate models like Pixel2Mesh, are substantial. In their work, the authors reported training times exceeding 72 hours on an Nvidia Titan X GPU, which is a high-end and expensive graphics card.

6.2 Reduced Model Complexity

To mitigate the resource limitations, we chose to focus on a simplified scenario by considering only two different classes for image regeneration. However, even with this reduction, achieving satisfactory results within the limited training duration proved challenging. The decision to limit the number of classes was influenced by practical considerations, yet it inherently affects the diversity and complexity of the learned features.

6.3 Trade-offs and Considerations

The compromise between model complexity, training time, and available resources introduces trade-offs in the quality of results. The lack of extensive training may have hindered the model's ability to capture intricate patterns and nuances in 3D shape generation. Despite our best efforts, it is evident that achieving state-of-the-art results comparable to more sophisticated models would necessitate greater computational resources and time investment.

6.4 Future Directions

To address these limitations and enhance the model's performance, future directions could involve securing access to more powerful GPUs or cloud-based computing resources. Additionally, opti-

mizing training parameters and exploring transfer learning techniques may contribute to improved results. The complexity of the 3D shape generation task requires careful consideration of available resources and the inherent trade-offs involved in model development.