Université Paris Saclay

3D Data Analysis

Master of Research in Artificial Intelligence and Machine Vision

---

Lab

3D Shape Retrieval using View-Based Descriptors

Student:
Nguyen Trong Dat

Supervisor:
Hedi Tabia

Master:
M2MMVAI

Academic Year
2023-2024

Contents

List of Figures

Listings

1 Introduction

1.1 Objective of the Project

The primary objective of this project is to develop an algorithm for 3D shape retrieval using View-Based Descriptors. But before delving into the specifics, let's first understand what 3D shape retrieval entails.

1.1.1 Understanding 3D Shape Retrieval

3D shape retrieval is the process of searching and retrieving three-dimensional models from a database based on their shape similarity. In other words, given a new 3D model, the goal is to find the most similar shape within a reference database. This task plays a crucial role in various applications, enabling efficient and effective content-based retrieval in 3D model databases.

1.1.2 Applications of 3D Shape Retrieval

The applications of 3D shape retrieval are diverse and impactful. Some notable examples include:

- **Computer-Aided Design (CAD):** Engineers and designers can benefit from quickly finding similar 3D models for inspiration or modification.
- **Biomedical Imaging:** Identifying and matching anatomical structures in medical imaging can aid in diagnosis and treatment planning.
- **Virtual and Augmented Reality:** Enhancing virtual environments by retrieving 3D objects that closely resemble real-world counterparts.
- **Robotics:** Matching 3D shapes can facilitate object recognition and manipulation for robots in diverse environments.

1.1.3 Project Focus

In this project, our focus is on utilizing View-Based Descriptors for 3D shape retrieval. The aim is to develop a system that, given a new 3D shape, can efficiently find the most similar shapes from a pre-existing database. We will employ the trimesh library for handling 3D mesh data and PyTorch for building and training a neural network model to achieve this objective.

1.2 Dataset Overview

For this project, we have chosen to utilize the McGill dataset, available at `https://www.cim.mcgill.ca/~shape/benchMark/`. The dataset comprises various classes of 3D models, each belonging to distinct categories, such as ants, crabs, hands, humans, octopuses, pliers, snakes, spectacles, spiders, and teddy bears.

The dataset structure is organized as follows:

- **Root Level:** The main directory of the dataset.
- **Second Level:** Ten subdirectories, each representing a different class of 3D models (e.g., ants, crabs, hands, humans, octopuses, pliers, snakes, spectacles, spiders, teddy bears).
- **Third Level:** For each class, two subdirectories are present - one containing files with the "Im" extension and the other with the "Ply" extension. The naming convention is consistent; for example, for the "humans" class, we have "humansIm" and "humanPly" directories.
- **Fourth Level:** Within the "Ply" directories, we find the 3D model files in the PLY format. These files are compressed in a gzip format (.gz).

1.2.1 Dataset Structure Diagram

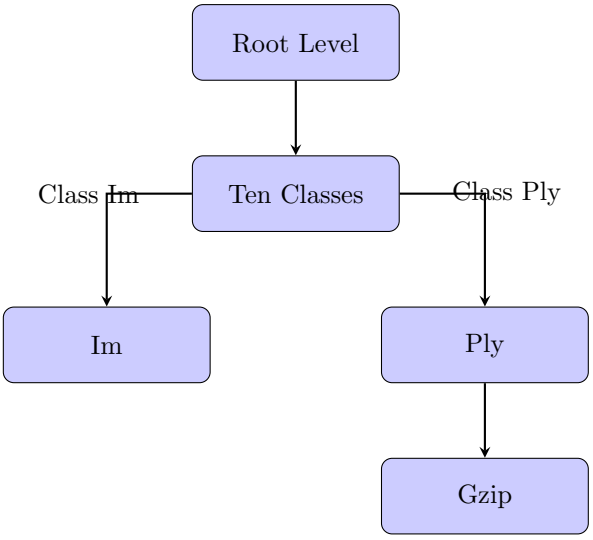
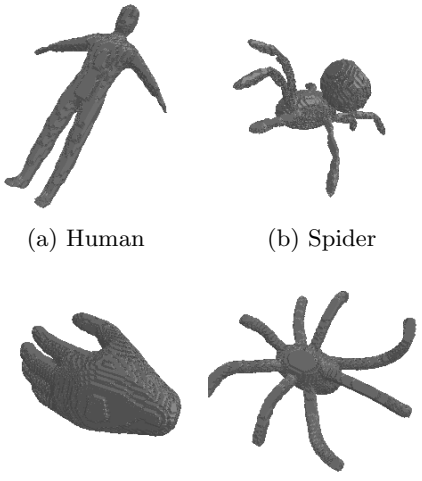


Figure 1: Structure of the McGill Dataset

1.2.2 Example Images

Let's illustrate a few examples by displaying 3D models from the dataset:



(a) Human (b) Spider

(c) Hand (d) Octopus

Figure 2: Example 3D Models from Different Classes

In Figure 2, we showcase example 3D models from various classes in a 2x2 grid format.

1.3 Plan of Action

To address the complexity of directly comparing 3D models, our approach involves extracting view-based descriptors. Instead of comparing the 3D models directly, we will generate 2D views (images) from the 3D models at different camera angles. Subsequently, these 2D views will be passed through a convolutional neural network (CNN) to extract features, specifically a latent vector.

1.3.1 Generating Latent Vectors

The process involves the following steps:

1. **3D to 2D Views:** From each 3D model, generate multiple 2D views by varying the camera angles.
2. **CNN Feature Extraction:** Pass the 2D views through a CNN to obtain a latent vector for each image. This vector captures high-level features.
3. **Vector Latent Space:** The latent vectors represent the 3D model in a feature-rich latent space.

1.3.2 Dataset Modification

To implement this approach, we will modify the structure of our dataset. For each class (e.g., "humans"), we will add two new components:

- **classnameImage2D:** A directory to store the 2D images generated from the 3D models.
- **latent_vector.json:** A JSON file storing the latent vectors associated with each 3D model.

The latent vectors, representing the features extracted from the 2D views, will be paired with their corresponding 3D models in the dataset.

1.3.3 Shape Retrieval Process

When presented with a new 3D model for shape retrieval, the process involves:

1. **Generate 2D Views:** Create 2D views from the new 3D model at various angles.
2. **Extract Latent Vectors:** Pass the generated 2D views through the pre-trained CNN to obtain the latent vectors.
3. **Search in Latent Space:** Compare the latent vectors with those in the reference dataset, identifying the closest match based on distance metrics.

Here, the latent vectors serve as our descriptors for comparing and retrieving 3D shapes.

In conclusion, with the outlined methodology of 3D shape retrieval using view-based descriptors, the modified dataset structure, and the proposed shape retrieval process, we are equipped with a comprehensive plan to commence our work. The combination of 2D views, CNN-based feature extraction, and latent vectors provides a promising foundation for efficiently comparing and retrieving 3D shapes from our reference dataset.

2 Generating Reference Database

To build our reference database for 3D shape retrieval, the first step is to create 2D views from the 3D models and save them. This not only facilitates faster experimentation by executing the program only once but also efficiently manages memory usage. We will utilize a custom dataset class, 'McGillDataset3D', to load and organize our McGill dataset.

2.1 McGillDataset3D Class

The 'McGillDataset3D' class is designed to handle the loading and organization of the McGill dataset. Here is an overview of its functionality:

```
class McGillDataset(Dataset):
    def __init__(self, root_dir: str, file_type: str = 'ply', transform=None):
        """
        Custom dataset for McGill dataset.

        Parameters:
        - root_dir (str): Root directory of the dataset.
        - file_type (str, optional): Type of file to load data from. Defaults to 'ply'.
        - transform (callable, optional): Optional transform to be applied on a sample.
        """
        self.root_dir = root_dir
        self.file_type = file_type
        self.transform = transform

        # List all the categories (subdirectories) in the root directory
        self.categories = os.listdir(root_dir)

        # Create a dictionary to map category names to their respective subdirectories
        self.category_paths = {category: os.path.join(root_dir, category) for category in self.categories}

        self.all_data_paths = self.get_all_data_paths()

    def __len__(self):
        """
        Returns the total number of samples in the dataset.
        """
        return len(self.all_data_paths)

    def __getitem__(self, idx):
        """
        Returns a sample from the dataset.

        Parameters:
        - idx (int): Index of the sample.

        Returns:
        - sample (dict): A dictionary containing 'data' (3D mesh) and 'label' (category name).
        """
        # Load the data (3D mesh) using the appropriate method
        data_path = self.all_data_paths[idx]
        category = self.get_category_from_path(data_path)

        if self.file_type == 'Im':
            with gzip.open(data_path, 'rb') as f:
                data = Image.open(io.BytesIO(f.read()))
        elif self.file_type == 'Ply':
            if data_path.endswith('.gz'):
                with gzip.open(data_path, 'rb') as f:
                    file_extension = os.path.splitext(data_path[:-3])[1]
                    data = trimesh.load(file_obj=io.BytesIO(f.read()), file_type=file_extension, process=False)
```

```
            else:
                file_extension = os.path.splitext(data_path)[1]
                data = trimesh.load(data_path, file_type=file_extension, process=
    False)

        # Apply the optional transform
        if self.transform:
            data = self.transform(data)

        # Prepare the sample dictionary
        sample = {'data': data, 'label': category}

        return sample

    def get_category_from_path(self, data_path):
        """
        Extract category name from the data path.

        Parameters:
        - data_path (str): Full path to the data file.

        Returns:
        - category (str): Category name.
        """
        # Extract category name from the path
        category = os.path.basename(os.path.dirname(os.path.dirname(data_path)))
        return category


    def get_all_data_paths(self):
        """
        Return a list of all data paths in the dataset.
        """
        all_paths = []
        for category in self.categories:
            category_path = self.category_paths[category]
            data_paths = os.listdir(os.path.join(category_path, f'{category}{self.
    file_type}'))
            full_paths = [os.path.join(category_path, f'{category}{self.file_type}'
    , data_name) for data_name in data_paths]
            all_paths.extend(full_paths)
        return all_paths

    def get_mesh(self, idx):
        """
        Return the 3D mesh.

        Parameters:
        - idx (int): Index of the sample.
        """
        return self.__getitem__(idx)['data']
```

Listing 1: McGillDataset3D Class

This class provides a structured way to access and manage the McGill dataset, listing categories, mapping category names to paths, and shuffling data paths for later use.

2.2 Generating 2D Views

The generate 2d views method in the McGillDataset3D class is responsible for creating 2D views from a given 3D shape by rotating it around both the X and Y axes. Here is a detailed breakdown of its functionality:

```
def generate2dImage(mesh,input_file_name,item):
    # Assuming meshes is an array of Trimesh objects
    scene = mesh.scene()
    label = item['label']
    # Define your ranges for theta and phi

```

```
    number_of_rotation = 3
    theta = 2*np.pi/number_of_rotation

    for i in range(number_of_rotation):
        for j in range(number_of_rotation):
            # Rotation matrix around the Y-axis (theta)
            rotate_y = trimesh.transformations.rotation_matrix(
                angle=theta, direction=[0, 1, 0], point=scene.centroid)

            # Rotation matrix around the X-axis (phi)
            rotate_x = trimesh.transformations.rotation_matrix(
                angle=theta, direction=[1, 0, 0], point=scene.centroid)

            # Combine the rotations
            rotate_combined = trimesh.transformations.concatenate_matrices(rotate_x
    , rotate_y)
            # Apply the combined transform to the camera view transform
            camera_old, _geometry = scene.graph[scene.camera.name]
            camera_new = np.dot(rotate_combined, camera_old)
            scene.graph[scene.camera.name] = camera_new

            if not os.path.exists(f'2dImages\{label}\{input_file_name}'):
                os.makedirs(f'2dImages\{label}\{input_file_name}')

            # Save the rendered image
            try:
                # Increment the file name
                file_name = os.path.join(f'2dImages\{label}\{input_file_name}',f"{
    input_file_name}_{i}{j}.png")
                # Save a render of the object as a png
                png = scene.save_image(resolution=[256, 256])
                with open(file_name, "wb") as f:
                    f.write(png)
                    f.close()
            except BaseException as E:
                trimesh.constants.log.debug("Unable to save image", str(E))
```

Listing 2: generate 2d views Method

This method takes an index (idx) corresponding to a 3D shape in the dataset and generates a series of 2D views by rotating the shape around both the X and Y axes. The number of steps for rotation (`number_of_rotation`) and the resolution of the generated images (resolution) can be customized. The resulting list views contains PIL Image objects representing the generated 2D views.

After that, the images generated is saved into corresponding path.

3 CNN for Latent Vector Extraction and Classification

In this section, we aim to achieve two objectives simultaneously. The primary goal is to design a Convolutional Neural Network (CNN) capable of effectively extracting latent vectors from 2D views of 3D shapes. Additionally, we leverage the CNN architecture to create a classifier that can predict the class of an object based on its 2D view.

The rationale behind integrating classification into our CNN design is twofold. First, by training the CNN as a classifier, we ensure that the network learns to extract meaningful features (latent vectors) that distinguish between different object classes. Second, the trained classifier serves as a means to streamline the comparison of latent vectors. Instead of comparing a latent vector with all vectors in the reference database, we can selectively compare it only with vectors from the same class.

For instance, when presented with a 2D view generated from a 3D model of a human, the CNN classifier predicts the class (e.g., "human"). We can then use this predicted class label to focus the latent vector comparison on the subset of latent vectors corresponding to humans in our reference database. This targeted approach enhances efficiency in retrieval tasks.

3.1 Training the Classifier

Now, let's delve into the details of training our classifier. We will use a ResNet-18 architecture, a popular pre-trained model known for its effectiveness in image classification tasks. ResNet-18 consists of multiple residual blocks, and its architecture can be summarized as follows:

- Input layer
- Convolutional layers with batch normalization and ReLU activation
- Residual blocks (multiple repetitions)
- Fully connected layer for classification

The classifier code defines a custom PyTorch module named `Classifier`, which loads a pre-trained ResNet-18 model and modifies its fully connected head to match the number of classes in our dataset.

```
# Define the classifier
class Classifier(nn.Module):
    def __init__(self, num_classes):
        super(Classifier, self).__init__()
        # Load a pre-trained ResNet model
        self.resnet = models.resnet18(pretrained=True)
        # Modify the classifier head to match the number of classes in your dataset
        in_features = self.resnet.fc.in_features
        self.resnet.fc = nn.Linear(in_features, num_classes)

    def forward(self, x):
        return self.resnet(x)
```

Listing 3: Resnet18 classifier

For training, we split our dataset into a training set (for building the reference database) and a test set (for evaluation). The classifier is trained using the Adam optimizer and cross-entropy loss. We iterate through a specified number of epochs, tracking training and test loss, as well as accuracy values.

Here is the training loop:

- Set the device (CPU or GPU) based on availability.
- Initialize the classifier.
- Define the loss function (cross-entropy) and optimizer (Adam).
- Set the number of training epochs.
- Iterate through epochs, separating the process into training and testing phases.
- In the training phase:
  - Set the model to training mode.
  - Iterate through batches in the training loader.
  - Perform a forward pass, compute the loss, and update weights.
  - Record running loss and track predictions and true labels for accuracy computation.
  - Print and store training loss and accuracy for each epoch.
- In the testing phase:
  - Set the model to evaluation mode.
  - Iterate through batches in the test loader.
  - Compute the test loss and track predictions and true labels.
  - Calculate and store test accuracy.

– Print test loss and accuracy for each epoch.

Here is the training loop:

```
#D ine the device
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
#Training
for epoch in range(n_epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    for batch in train_loader:
        # Assuming data is a tuple of (image, model_name, label, value)
        images = batch['image']
        labels = batch['label']
        values = batch['value']
        images, values = images.to(device), values.to(device)  # Move data to the
    appropriate device (CPU/GPU)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward pass
        output = model(images)

        # compute loss
        loss = criterion(output, values.long())  # Ensure value is of type long for
     CrossEntropyLoss

        # backward pass
        loss.backward()  # Computes the gradient for every parameter
        optimizer.step()  # Update the weights

        # print statistics
        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss}")

print(f'Finished trainset')

#Testing
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our
    outputs
with torch.no_grad():
    for batch in train_loader:
        images = batch['image']
        labels = batch['value']
        # calculate outputs by running images through the network
        outputs = model(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += batch_size # Calculating the total number of images
        correct += (predicted == labels).sum().item() # Calculating the number of
    correctly predicted images

print(f'The Accuracy of the network on the train images: {100 * correct // total} %
    ')

correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our
    outputs
with torch.no_grad():
    for batch in validation_loader:
        images = batch['image']
        labels = batch['value']
        # calculate outputs by running images through the network
        outputs = model(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += batch_size # Calculating the total number of images
```

```
62          correct += (predicted == labels).sum().item() # Calculating the number of
        correctly predicted images
63
64  print(f'The Accuracy of the network on the test images: {100 * correct // total} %'
        )
```

Listing 4: Resnet18 classifier training

The training loop concludes by storing the loss and accuracy values, which will be visualized in learning curves for further analysis.

This approach not only trains a classifier for object classification but also prepares the CNN for the dual task of extracting latent vectors from 2D views.

3.2 Result of Classification

Metric	Value (%)
Training Accuracy	94
Test Accuracy	83

Table 1: Classifier Performance Metrics

After training and evaluating the classifier, the obtained results are promising, demonstrating a high level of performance. The model achieved an impressive accuracy of 94% on the training set and maintained strong generalization capabilities with a test accuracy of 83%. These metrics indicate the classifier's ability to effectively learn and generalize features from 2D views of our 3D shapes, showcasing its potential for accurately classifying objects in our reference database.

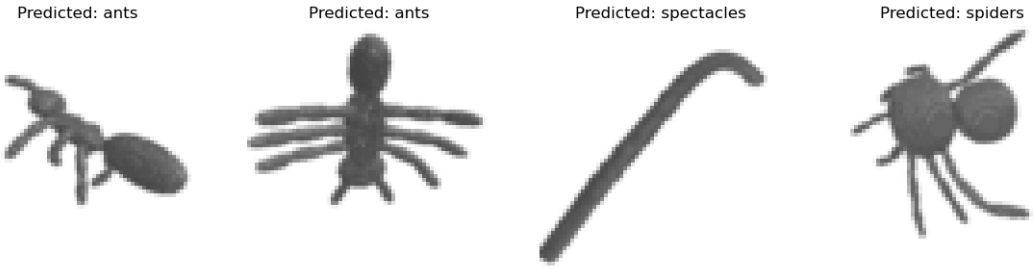


Figure 3: Images true/predicted label Training

3.3 Feature Extraction for Latent Vector

To perform feature extraction and obtain the latent vector from the 2D views, we utilize the pre-trained ResNet18 model employed in our classifier. The following code snippet illustrates the process:

```
# Get a batch of training data
dataiter = iter(train_loader)
data = next(dataiter)

image, label, value, path = data['image'], data['label'], data['value'], data['path
    ']
features_extractor = torch.nn.Sequential(*(list(model.resnet.children())[:-1]))
```

Listing 5: Feature Extraction with ResNet18

By removing the final fully connected layer, we retain the feature extraction capabilities of ResNet18, and the output of this modified model now serves as our latent vector for further comparison and retrieval tasks.

3.4 Initialization of Feature Extractor

To prepare for the extraction of latent vectors, we initialize our feature extractor by removing the last layer of the ResNet classifier. This modification ensures that we retain the feature extraction capabilities of the ResNet model while discarding the final classification layer.

```
features_extractor = torch.nn.Sequential(*(list(model.resnet.children())[:-1]))
```

Listing 6: Feature Extractor Initialization

3.5 Latent Vector Generation

Once the feature extractor is set up, we proceed to generate latent vectors for 3D models in our training dataset. The latent vectors are derived from the 2D views of the models. For each 3D model, we iterate through its associated 2D views, extract the latent vectors using the feature extractor, and store them in a dictionary.

```
# File to save the features
output_file = 'image_features.txt'
with open(output_file, 'w') as file:
    for batch in train_loader:
        images = batch['image']
        paths = batch['path']
        with torch.no_grad():
            batch_features  = features_extractor(images)
        for i in range(batch_features.size(0)):
            features = batch_features[i]
            # Flatten the features and convert to a list
            features_list = features.flatten().tolist()
            # Write the filename and features to the file
```

```
            file.write(f"{paths[i]} {features_list}\n")
print("Feature extraction complete and saved to text file.") #store the latent
    vector in a txt file - linked to it's 3D model
```

Listing 7: Latent Vector Generation

3.6 Analysis

The generation of latent vectors from 2D views provides a compact representation of the 3D models. This process enables efficient retrieval and comparison of shapes in our subsequent retrieval algorithm. The resulting latent vectors, stored in JSON files, form a crucial part of our final 3D shape retrieval database.

4 3D Shape Retrieval Application

With our comprehensive 3D shape database in place, containing 3D models, corresponding 2D views, and their respective latent vectors, we are well-equipped to develop a 3D shape retrieval application. The objective here is to take new 3D models, generate 2D views from them, extract latent vectors, and perform a retrieval process by finding the nearest neighbors in our database.

4.1 Workflow Overview

The retrieval process follows a sequence of steps:

1. **Input**: New 3D model.
2. **Generate 2D Views**: Generate a set of 2D views from the 3D model.
3. **Extract Latent Vectors**: Extract latent vectors from the generated 2D views.
4. **Retrieve Nearest Neighbors**: Calculate distances between the extracted latent vectors and those in the database. Choose the nearest neighbors based on a chosen distance metric.
5. **Match 3D Shapes**: Retrieve and match the 3D shapes associated with the nearest neighbors.

This workflow enables efficient and effective 3D shape retrieval, providing a practical application for various domains.

4.2 Implementation of Distance Metric

To facilitate the retrieval process, we begin by implementing a distance metric, which measures the dissimilarity between two vectors. In our case, we start with the Euclidean distance, a common metric for vector spaces. The implementation is as follows:

```
def euclidean_distance(vec1, vec2):
    vec1 = np.array(vec1)
    vec2 = np.array(vec2)
    return np.sqrt(np.sum((vec1 - vec2) ** 2))
```

Listing 8: Euclidean Distance Implementation

The Euclidean distance provides a measure of similarity, with identical vectors resulting in a distance of 0. This implementation is crucial for assessing the dissimilarity between latent vectors during the retrieval process.

4.3 3D Shape Retrieval Implementation

Now equipped with our distance metric, we can proceed with the 3D shape retrieval process. The workflow involves generating 2D views from a new 3D model, extracting the latent vector, and finding the nearest neighbors in our existing database.

```
file_path = 'image_features.txt'
n = 5

# Get a batch of training data
datatiter = iter(validation_loader)
data = next(datatiter)

image, label, value, path = data['image'], data['label'], data['value'], data['path
    ']
with torch.no_grad():
    batch_features = features_extractor(image)
    for i in range(batch_features.size(0)):
        features = batch_features[i]
        top_n_closest = find_top_n_closest_to_input(features, file_path, n)
        print(top_n_closest)
```

Listing 9: 3D Shape Retrieval Implementation

The '$find_top_nclosest_to_input$' function is responsible for finding the nearest neighbors based on the Euclidean distance metric. Let's include this function:

```
def find_top_n_closest_to_input(input_vector, file_path, top_n): #k nearest
    neighbor
    """Find top n closest vectors from a file to the input vector."""
    # Read and parse the file
    vector_set = []
    identifiers = []
    with open(file_path, 'r') as file:
        for line in file:
            parts = line.strip().split(' ', 1)
            if len(parts) == 2:
                identifier, vector_str = parts
                vector = [float(x) for x in vector_str.strip('[]').split(',')]
                identifiers.append(identifier)
                vector_set.append(vector)

    # Calculate distances to the input vector
    distances = [(identifiers[i], euclidean_distance(input_vector, vec)) for i, vec
     in enumerate(vector_set)]
    distances.sort(key=lambda x: x[1])

    # Return the top n closest vectors
    return distances[:top_n]
```

Listing 10: Find Closest Image Information Function

The retrieval process allows us to identify the closest matches in our database for a given new 3D model, providing valuable information for shape analysis and recognition.

4.4 Results

To visually assess the effectiveness of our 3D shape retrieval system, we provide two figures. The first figure displays a set of 2D views generated from our test 3D models. The second figure presents the corresponding nearest neighbors from our database based on the extracted latent vectors.

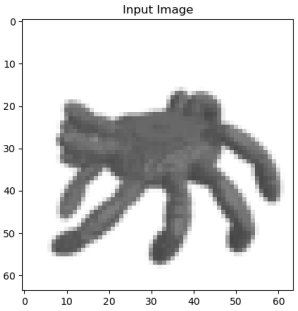
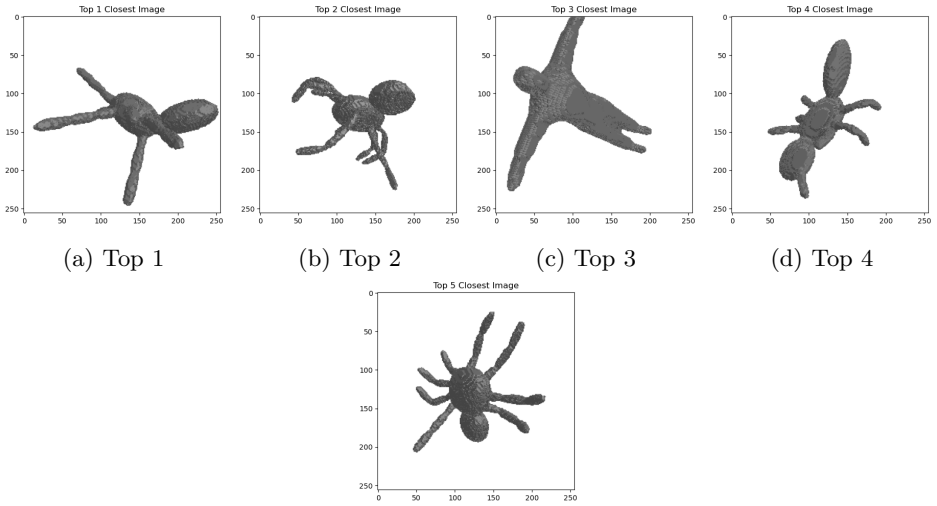


Figure 4: 2D views generated from test 3D models.

Figure 4 showcases the 2D views generated from our test 3D models, providing an insight into the diversity of shapes in our testing dataset. In Figure 5, we present the 2D views associated with the 3D models from our database, which are identified as the nearest neighbors based on the extracted latent vectors. This comparison allows for a visual evaluation of the success of our 3D shape retrieval system.

4.5 Challenges in Quantifying Retrieval Efficacy

Measuring the effectiveness of our 3D shape retrieval system using quantitative metrics poses challenges due to the inherent ambiguity in determining the "correct" match for a given 3D model. Unlike typical classification tasks, where a ground truth label is known, the matching process in 3D shape retrieval involves finding similar shapes without predefined class labels.



(a) Top 1 (b) Top 2 (c) Top 3 (d) Top 4

(e) Top 5

Figure 5: Nearest neighbors from the database corresponding to the test 3D models.

Two approaches can be considered for assessing retrieval efficacy. The first approach involves a brute-force comparison of the latent vector of the test 3D model with all latent vectors in the database, selecting the model with the closest latent vector. However, this method may be computationally expensive, especially as the database size increases.

A more targeted approach involves classifying the test 3D model using the 2D views generated from it. Subsequently, the retrieval process focuses on finding the nearest neighbors within the same predicted class. While this approach may reduce computational complexity, it introduces potential errors if the initial classification is inaccurate.

Moreover, enhancing the model's performance could involve adopting a multi-view strategy. Instead of relying on a single generated image and its associated latent vector, multiple views of the test 3D model can be considered. Each view contributes its latent vector, and a voting mechanism can be employed to determine the most probable match among the retrieved models.

These considerations highlight the nuanced nature of 3D shape retrieval evaluation, emphasizing the need for a combination of quantitative and qualitative assessments to comprehensively evaluate the system's performance.

5 Conclusion and Additional Resources

In conclusion, this report outlines the development of a 3D shape retrieval system using latent vectors extracted from 2D views of 3D models. The workflow involves generating latent vectors for training models, creating a ResNet-based classifier for feature extraction, and implementing a retrieval application for new 3D models.