Documentation of my Content Based Multimedia Retrieval Engine

User Interface

For the user interface I implemented a streamlit app which is interactive and user friendly. The app contains a sidebar which gives the user these options:

1. Shape Viewer
2. Search Engine
3. Global Statistics
4. Presentation

1. Shape Viewer

By selecting the "Shape Viewer" option, the user is found in a page that allows for the selection of a mesh category and a specific object from this category. At the top of the page there appears the option to view either the original object or the final object (after the resampling, remeshing and normalization). Finally the user can decide on whether the mesh edges will be visible or not. By pressing "Visualize Shape" a viewing area appears with the 3d mesh appearing in a gridded environment. The process of visualizing the objects is as follows:

1.1. Visualization process

The first technical task in the viewer is loading the 3D object file, which is handled using the Trimesh library. This library is well-suited for working with 3D triangular meshes and provides a straightforward way to load common 3D formats, such as .obj files.

Once a file is selected by the user, the application loads it into memory as a Trimesh object. This object contains the mesh's vertices and faces, which are essential for rendering the 3D shape later in the process. Trimesh was chosen for its simplicity and efficiency in handling 3D data.

1.2. Converting the Mesh for Visualization

After loading the 3D object, the next step is to convert it into a format that can be visualized using Plotly. The conversion involves extracting the vertices and faces (triangles) from the Trimesh object. These are passed to Plotly's `go.Mesh3d` function, which renders the 3D surface of the object within the application.

The mesh is displayed with a customizable level of transparency and flat shading, which enhances the visibility of the surface's details. Flat shading is a technique that renders each polygon with a single color, making the individual facets of the 3D shape more visible, which helps highlight the structure of the mesh.

Users also have the option to toggle the visibility of the mesh's edges. If enabled, the edges are added to the visualization as black lines that outline each face, improving the clarity of the mesh structure. The edge traces are calculated by iterating over the triangles and determining which vertices form the edges of each face.

1.3 User Interface and Rendering

Once the mesh is processed and converted, it is ready for display in the Streamlit interface. A progress bar is shown to the user during the loading and conversion processes, giving feedback on the current state of the visualization. The `plotly_chart` function is used to embed the 3D visualization directly into the app, allowing for interactive exploration of the mesh (e.g., rotating, zooming, panning and taking a screenshot).

The visualization is enclosed within a styled border, which helps delineate the viewing area and adds a visual structure to the layout. The 3D plot is displayed with gridlines on all axes, enhancing the perception of depth and the relative positioning of the object in space.

2. Search Engine

The "Search Engine" page is dedicated to the retrieval part of the assignment. Its content is simple, with just a file loader that accepts `.obj`, `.ply` and `.off` files. The input file is then objected in feature extraction and similarity search that is explained in later sections. After the retrieval is completed, the input shape along with N retrieved objects are showed.

3. Global Statistics

This page is dedicated in presenting the global statistics of the dataset in different stages of the preprocessing phase. Specifically, the original database (before the preprocessing) and the database after the resampling and remeshing. It contains statistics like: Total Shapes, Mean and Std of Face Count, Mean and Std of Vertex Count, High and Low Outliers and Watertight Object Count.

4. Presentation

In this page visualizations of the whole process will be presented along with text and explanations. From the preprocessing, to the feature extraction and the evaluation.

Preprocessing and Cleaning

To prepare the shape database for feature extraction, I conducted several preprocessing steps to ensure that all shapes met the required quality standards. This involved analyzing each shape individually, computing statistics over the entire dataset, resampling outliers, remeshing-uniform the shapes and normalizing the shapes.

1. Analyzing a Single Shape

The first step was to analyze each shape in the dataset to extract essential information. For this purpose, I developed a function that processes each 3D shape file and gathers the following data:

1. Class of the Shape: Identified from the folder structure of the dataset, where each folder represents a different category
2. Number of Vertices and Faces: Calculated by counting the vertices and faces in the mesh.
3. Type of Faces: Determined whether the mesh consists of triangles, quads, or a mix of both.
4. Axis-Aligned Bounding Box (AABB): Computed by finding the minimum and maximum coordinates along each axis, encapsulating the shape within a box.

I utilized the Trimesh library to load each 3D object file, as it provides efficient methods for handling and analyzing mesh data. By accessing the mesh properties, I obtained the counts of vertices and faces and assessed the face types. The AABB was calculated by extracting the bounds of the mesh, which are the minimum and maximum vertex coordinates along the x, y, and z axes.

Additionally, I checked whether each mesh is manifold, meaning it does not have any holes or inconsistencies, which is crucial for reliable feature extraction later on. I also flagged shapes as outliers if their vertex count was below or above predefined thresholds, indicating they might be under-sampled or overly complex.

2. Statistics Over the Whole Database

After analyzing individual shapes, I aggregated the data to compute statistics for the entire dataset. This included calculating the mean and standard deviation of the number of vertices and faces across all shapes. These statistics provide insight into the typical complexity of the shapes and help identify any significant deviations.

To identify outliers, I examined shapes that had vertex or face counts significantly lower or higher than the average. Specifically, shapes with fewer than 100 vertices or faces were considered under-sampled and marked for resampling. Conversely, shapes with excessively high counts were noted, as they could lead to increased computational load during feature extraction.

I visualized the distributions of vertices and faces using histograms. These plots display the number of shapes falling within specific ranges of vertex and face counts, making it easier to observe the overall distribution and pinpoint outliers.

3. Mesh Cleaning

To ensure the integrity and quality of the mesh data throughout the preprocessing pipeline, I implemented a comprehensive cleaning procedure. This cleaning process is crucial for removing inconsistencies, artifacts, and potential errors that could negatively impact subsequent steps like resampling and feature extraction. The cleaning is applied at multiple stages: before resampling, after resampling, and after any significant modification to the mesh data, such as remeshing.

3.1. Cleaning Procedure

The cleaning process involves several steps that address common issues found in 3D meshes. The primary function responsible for this is the clean_mesh function, which takes a mesh object as input and applies a series of cleaning operations using the Trimesh library.

The steps in the cleaning procedure are as follows:

Removal of Degenerate Faces

A degenerate face is defined as a face with zero area, which occurs when the vertices of a triangular face are either co-located or nearly collinear, causing the face to collapse into a line or point. These faces do not contribute to the surface geometry and can lead to incorrect results during mesh processing.

To identify and remove degenerate faces, the area of each triangular face is calculated using the cross product of two edge vectors. For a triangle with vertices $v1$, $v2$, and $v3$, the edges are defined as $Edge1 = v2 - v1$ and $Edge2 = v3 - v1$. The area of the triangle is proportional to the magnitude of the cross product of

these two edge vectors: `(math operation here)` A degenerate face has an area of zero, meaning that the cross product of its edge vectors results in a zero vector. Once these degenerate faces are identified, they are removed from the mesh. This cleaning operation ensures that the mesh only contains valid faces with non-zero area, which are necessary for accurate geometric representation and processing. By removing these zero-area faces, the mesh is made more efficient and free from artifacts that could hinder subsequent analysis or rendering steps.

Normal Calculation

The normal vector of a triangular face is a crucial element for operations such as rendering, shading, and geometry processing, as it defines the direction perpendicular to the face. To compute the normal for a given triangular face with vertices $v1$, $v2$, and $v3$, the first step is to determine two edge vectors by subtracting the coordinates of the vertices. These edges, `Edge1 = v2 - v1` and `Edge2 = v3 - v1`, represent two sides of the triangle. The cross product of these edge vectors:

`(math operation here)`

gives a vector n that is perpendicular to the plane of the triangle. This cross product essentially captures the area of the parallelogram formed by the edges, pointing in the direction orthogonal to the face of the triangle. The result of the cross product is then normalized

`(math operation here)`

to ensure the normal vector has a unit length, which is critical for maintaining consistency in calculations like shading. Normalization is achieved by dividing the normal vector by its magnitude, calculated as the square root of the sum of the squares of its components.

The normalized normal vector is used to ensure that the face is correctly oriented and that rendering operations, like lighting and shading, are accurate.

Merging Vertices

Merging vertices is a process used to simplify the mesh by eliminating duplicate or nearly identical vertices, which can often occur due to minor inaccuracies in the data. To identify vertices that should be merged, the Euclidean distance between pairs of vertices is calculated. If the distance between two vertices vi=(xi ,yi ,zi) and vj=(xj, yj, zj) is less than a small tolerance ϵ, they are considered duplicates and are merged.

`(math operation here)`

In the case of exact duplicates, the coordinates are identical, but for nearly identical vertices, this small threshold ϵ accounts for slight differences in position. When merging two vertices, one vertex is removed from the mesh, and any faces that referenced the removed vertex are updated to use the remaining one. In some cases, the coordinates of the two vertices can be averaged to create a new, smoothed position for the merged vertex. This merging process reduces the number of vertices without altering the overall geometry of the mesh, ensuring a cleaner, more efficient structure. This operation is particularly important for meshes with redundant data, as it optimizes the mesh for faster processing and more accurate feature extraction while preserving its visual and structural integrity.

3.2. Parallel Processing with Multiprocessing

Given the large number of meshes in the dataset, cleaning each one sequentially would be time-consuming. To expedite this process, I implemented parallel processing using Python's `multiprocessing` module. This approach allows the cleaning tasks to be distributed across multiple CPU cores, significantly speeding up the overall workflow. The `clean_dataset_parallel` function creates a pool of worker processes, with the number of processes corresponding to the number of available CPU cores. Each worker process is assigned a subset of meshes from the dataset to clean, distributing the workload efficiently. This is achieved by mapping the `clean_single_mesh` function, which is responsible for cleaning a single mesh, over the entire dataset, where each process operates independently on its assigned meshes. The function uses the formula: `Results=Pool.map(clean_single_mesh,Mesh List)` to parallelize the task, ensuring that all processes work concurrently. Once all processes have completed their tasks, the results are synchronized and collected, with any failed meshes being filtered out from the final dataset. This parallelization approach drastically reduces the time needed to clean the entire dataset, making it feasible to run the cleaning procedure multiple times throughout the preprocessing pipeline.

4. Resampling of Meshes

Resampling is a critical step in adjusting the complexity of the meshes to ensure they meet specific quality requirements for subsequent processing steps. The primary objective of this step is to refine low-resolution meshes that do not have enough detail and to simplify high-resolution meshes that are overly complex, thus bringing all meshes within a target range of vertex counts. This standardization is important for ensuring consistency across the dataset, especially for feature extraction, which can be negatively affected by meshes that are either too sparse or too dense.

To achieve this, I implemented a parallelized resampling procedure that operates across multiple CPU cores, significantly improving the efficiency of the resampling process. The core function used for this is `process_meshes_parallel`, which divides the task of resampling across multiple processes. Each mesh is resampled individually by the `process_single_mesh_resample` function, which ensures that the mesh complexity is adjusted and the resampled mesh is saved.

4.1. Adjusting Mesh Complexity

The key task in resampling is to adjust the complexity of each mesh. This is handled by the `adjust_mesh_complexity` function. The process begins by loading each mesh using the Trimesh library, which allows us to manipulate its vertices and faces. The complexity of the mesh is determined by counting the number of vertices, which serves as a proxy for how detailed or simple the mesh is. If the mesh has too few vertices (below a predefined lower bound), it is considered under-sampled and needs refinement. On the other hand, if the mesh has too many vertices (above an upper bound), it is considered too complex and needs simplification.

For meshes with a vertex count below the lower bound, I applied a **subdivision** operation. Subdivision increases the resolution of the mesh by splitting each triangular face into smaller triangles. This process effectively adds more vertices to the mesh while maintaining the original geometry. Mathematically, subdivision works by splitting each edge of the triangle and creating new vertices at the midpoint of each edge, then connecting these new vertices to form additional triangles. This operation can be expressed as:

`(math operation here)`

where $v1$ and $v2$ are the vertices of the original triangle edge. Each edge of the triangle is subdivided in this way, and new triangles are formed from the original vertices and the newly created vertices.

For meshes with too many vertices, I applied **quadric error decimation**, a technique that simplifies the mesh by reducing the number of faces and vertices while preserving the overall shape. Decimation calculates the error introduced by collapsing edges and removes the least important vertices first. In each step, a vertex is merged with one of its neighboring vertices, and the faces that contain the vertex are updated accordingly. The percentage of vertices to remove can be controlled, and in this implementation, I used a decimation factor of 30%, meaning that approximately 30% of the vertices are removed in each iteration:

```
(math operation here)
```

This operation minimizes the geometric error introduced during simplification, allowing the mesh to retain its overall shape even as the number of vertices is reduced.

4.2. Parallel Processing of Resampling

Given the large number of meshes to process, I again leveraged Python's multiprocessing module to parallelize the resampling procedure. It works by dividing the dataset into smaller chunks, with each process in the pool working on a subset of the dataset. Each process runs the `process_single_mesh_resample` function, which loads the mesh, adjusts its complexity using the `adjust_mesh_complexity` function, and then saves the resampled mesh to the designated directory.

4.3. Final Filtering

After the resampling step was completed, I encountered a few meshes that remained outliers despite the adjustments. These outliers had too too many vertices (exceeding the upper bound), indicating that the initial resampling was not sufficient to bring them within the desired complexity range.

To address this issue, I implemented a filtering process that identified and removed any meshes still classified as outliers. Specifically, I selected all meshes with vertex counts falling outside the defined acceptable range and excluded them from the dataset.

5. Remeshing of Meshes

After resampling and cleaning the dataset, the next critical step in the preprocessing pipeline was remeshing, which involves restructuring the surface of the meshes to achieve a consistent and uniform level of detail. This process, known as isotropic remeshing, redistributes the vertices and faces of the mesh more evenly across its surface while maintaining the original geometry. The remeshing step is particularly delicate, as it requires balancing the number of vertices and faces to ensure the mesh is neither too dense nor too sparse, both of which could affect subsequent feature extraction.

To automate this step across a large dataset, I used parallel remeshing to distribute the workload across multiple CPU cores, similar to previous steps like resampling and cleaning. The core function responsible for this step is the `isotropic_remesh` function, which performs the actual remeshing operation on each mesh.

5.1. Isotropic Remeshing

The goal of isotropic remeshing is to make the distribution of vertices more uniform, regardless of the original structure of the mesh. This is achieved by iteratively adjusting the edge lengths of the mesh faces so that they approach a target length, ensuring that all triangles are of similar size.

Surface Area Calculation

The first step in the isotropic remeshing procedure is to calculate the surface area A of the mesh. This value is critical for determining the target edge length for the remeshing process. The surface area is computed using the formula:

(math operation here)

where the total surface area A is the sum of the areas of all faces in the mesh. This area is used to calculate the target edge length for remeshing.

Target Edge Length Calculation

Once the surface area is known, the target edge length $L\text{target}$ is calculated based on the desired number of vertices $V\text{target}$, which we set to 5000 in this implementation. The target edge length is derived using the formula:

(math operation here)

This formula ensures that the edge lengths are adjusted proportionally to the surface area and the target number of vertices, maintaining a consistent level of detail across the mesh.

Iterative Remeshing

The isotropic remeshing process is iterative. In each iteration, the target edge length is applied, and the mesh is remeshed to redistribute the vertices accordingly. If the number of vertices after an iteration is still below the target, the edge length is recalculated using a smaller scaling factor. This process continues until the vertex count approaches or exceeds the target number of vertices. The scaling factor is decreased in each iteration, allowing finer control over the remeshing process. This iterative process ensures that the mesh reaches the desired level of detail without over-refining it.

Simplification Step (Optional)

After the remeshing step, if the mesh has a significantly higher number of faces than initially desired, an optional quadric edge collapse decimation step is applied. This decimation process simplifies the mesh by reducing the number of vertices and faces while preserving the overall geometry of the mesh. This step is mathematically represented as:

(math operation here)

5.2. Parallel Processing of Remeshing

Given the complexity of remeshing and the size of the dataset, parallel processing was essential to ensure the entire dataset could be processed in a reasonable amount of time. The `parallel_remeshing` function distributes the task across multiple CPU cores, assigning each chunk to a separate process.

Each process runs the apply_remeshing function, which applies isotropic remeshing to each mesh in its chunk. After processing, the remeshed files are saved in a designated directory. The parallelization ensures that the remeshing step is completed without bottlenecking the workflow.

5.3. Handling Edge Cases

During the remeshing process, we encountered certain edge cases where some meshes did not reach the desired target vertex count despite several iterations. For these cases, we applied additional checks to ensure that the mesh was still geometrically valid and consistent after remeshing. If a mesh could not be adequately remeshed within the allowed iterations, it was flagged for further review or removed from the dataset to maintain overall quality.

6. Normalization of Meshes

Normalization is a key step in preparing the dataset for analysis and feature extraction. It ensures that all meshes are aligned, centered, and scaled in a consistent manner, allowing for more accurate comparisons between shapes during later stages of the pipeline. The goal of normalization is to remove variations in translation, scale, and orientation that are not intrinsic to the shape itself. I applied four main steps in the normalization procedure: **centering at the origin**, **scaling to a unit cube**, **principal component analysis (PCA) alignment**, and **moment flipping**.

6.1. Centering the Mesh at the Origin

:Todo: Mention **translation** invariance

The first step of normalization is to translate the mesh so that its centroid (the geometric center of its vertices) coincides with the origin (0,0,0) of the coordinate system. This ensures that the mesh is positioned symmetrically around the origin, eliminating any bias due to its initial location.

The centroid of the mesh is computed as the mean of all its vertex coordinates:

(math operation here)

where vi represents the i-th vertex of the mesh, and N is the total number of vertices. The mesh is then translated by subtracting the centroid from each vertex position, effectively shifting the entire shape to the origin:

(math operation here)

6.2. Scaling to Fit a Unit Cube

:Todo: Mention **scale** invariance

After centering the mesh, the next step is to scale it so that it fits tightly within a unit cube (a cube with side lengths of 1) ensuring that all shapes have the same scale and allowing for fair comparisons between objects of different sizes.

To achieve this, we compute the bounding box of the mesh, which is the smallest box that completely encloses the shape. The extents of the bounding box are the lengths of its sides along the x, y, and z axes. The largest extent, Lmax, is used as the scaling factor:

(math operation here)

The mesh is then uniformly scaled so that its largest dimension fits within the unit cube. This is done by dividing all vertex coordinates by Lmax, ensuring that the shape fits within the bounds of $[-0.5,0.5]$ along

each axis:

`(math operation here)`

6.3. PCA Alignment

:Todo: Mention **rotation** invariance

`Principal Component Analysis (PCA) Alignment` was applied to ensure that the orientation of all meshes is standardized. PCA finds the primary axes along which the vertices of the mesh vary the most, and aligns these axes with the coordinate system. By aligning the mesh with its principal components, I reduce the impact of arbitrary rotations in the original data, making the mesh orientation consistent across different shapes.

he pca_align function computes the principal components (or eigenvectors) of the covariance matrix of the mesh's vertices. These eigenvectors represent the directions of maximum variance in the data. The first eigenvector $e\mathbf{1}$ corresponds to the direction of the largest variance, and the second eigenvector $e\mathbf{2}$ corresponds to the direction of the second largest variance. The third eigenvector is orthogonal to the first two and is given by their cross product $e\mathbf{3}=e\mathbf{1}\times e\mathbf{2}$.

The covariance matrix C of the vertices is defined as:

`(math operation here)`

where vi represents the coordinates of the i**-th** vertex and $\bar{v}$ is the centroid of the mesh. The eigenvectors of this matrix, $e\mathbf{1}$, $e\mathbf{2}$, and $e\mathbf{3}$, provide the directions along which the mesh will be aligned. The PCA alignment projects each vertex onto these eigenvectors to re-orient the mesh:

`(math operation here)`

This ensures that the largest variation in the mesh lies along the x**-axis**, the second largest variation along the y**-axis**, and the smallest variation along the z**-axis**. This realignment standardizes the orientation of the mesh, allowing for fair comparison between different shapes.

6.4. Moment Flipping

Moment flipping is the final step in the normalization process, and it ensures that the mesh is oriented such that its "heaviest" parts lie on the positive side of each axis. This step is important because even after PCA alignment, the mesh might still be reflected or flipped along one or more axes, leading to inconsistent orientations across the dataset.

The **moments of inertia** provide a measure of how mass (or vertex distribution) is spread relative to an axis. For each axis, the moment of inertia is computed as the sum of the squared distances of the triangle centroids from that axis:

`(math operation here)`

where xi, yi, and zi are the coordinates of the centroid of the i**-th** triangle in the mesh.

The moment_flip function calculates these moments of inertia along the x**-**, y**-**, and z**-axes**, and uses their signs to determine whether the mesh needs to be flipped along each axis. Specifically, if the moment of

inertia along an axis is negative, the mesh is flipped along that axis by multiplying all vertex coordinates along that axis by −1. This ensures that the majority of the mesh lies in the positive half-space of each axis.

The flipping operation can be described mathematically as:

```
(math operation here)
```

where Ix, Iy, and Iz are the moments of inertia along the respective axes, and the sign function returns −1 if the moment is negative and 1 otherwise. By multiplying each vertex coordinate by the corresponding sign, the mesh is flipped along the appropriate axes.

6.5 Parallel Processing for Normalization

To speed up the normalization process, I implemented parallelization using Python's multiprocessing module.

7. Feature Extraction

Feature extraction is one of the most crucial parts of the 3D shape analysis procedure. The objective of this step is to derive meaningful numerical representations (descriptors) of the shapes, which can later be used for shape comparison and retrieval. These descriptors summarize important characteristics of the 3D objects in a compact form, allowing us to quantitatively analyze and compare complex 3D geometries.

In this feature extraction process, two major types of descriptors are computed:

1. Global Descriptors: These yield a single value that summarizes a global property of the shape, such as surface area or compactness.
2. Shape Property Descriptors: These are statistical measures based on random geometric properties of the shape, such as distances between vertices or angles between random points. Since they produce distributions of values rather than a single value, we reduce these distributions to a fixed-length descriptor using histograms.

This representation will allow us to build a shape retrieval or classification engine that can efficiently compare 3D objects based on their extracted characteristics. Each descriptor provides unique insights into different aspects of the shape, and together they form a comprehensive description of the object.

7.1 Global Descriptors

The global descriptors provide single real values that capture the overall characteristics of the 3D shapes. In the feature extraction process, we compute several key global descriptors, including **volume**, **surface area**, **diameter**, **eccendricity**, **compactness**, **rectangularity**, **convexity**, **sphericity** and **elongation**. Each of these descriptors is adapted for 3D shapes and provides a different perspective on the geometry of the object.

We will now go through each global descriptor in detail, explaining how it is calculated and what geometric property it reflects.

7.1.1. Surface Area

The surface area of a 3D shape is one of the most fundamental global descriptors, as it directly quantifies the amount of surface covering the object. This descriptor is useful for comparing shapes with similar volumes but different surface textures or details.

To compute the surface area of a mesh, we sum the areas of all the individual triangular faces that make up the mesh. For each triangle, the area can be calculated using the vertices of the triangle. If the vertices of a triangle are $\boldsymbol{v1}$, $\boldsymbol{v2}$, and $\boldsymbol{v3}$, the area of the triangle is given by the magnitude of the cross product of two of its edge vectors

(math operation here)

The total surface area of the mesh is the sum of the areas of all triangles:

(math operation here)

where n is the total number of triangles in the mesh.

The Trimesh library handles the computation by summing the areas of all faces in the mesh, and it returns the total surface area. This computation is straightforward and accurate for watertight and non-watertight meshes alike, as long as the mesh is well-defined.

7.1.2. Volume

The volume of a mesh is another important descriptor that provides insight into the amount of space the shape occupies. For watertight meshes (meshes with no holes), the volume can be calculated directly using geometric methods. However, for non-watertight meshes, we need to approximate the volume using other techniques, such as voxelization. Since a few objects are not watertight, using a different technique for them would create inconsistencies. This is why I decided to use the voxelization approach for all the meshes and convex hulls as well.

Voxelization divides the 3D space into small cubic units (voxels), and the mesh is represented as a set of these voxels. Each voxel has a fixed size defined by the pitch parameter (the side length of the voxel). The total volume is then approximated by counting the number of voxels that intersect with the mesh and multiplying this number by the volume of a single voxel.

If N is the number of voxels that are filled by the mesh, and each voxel has a volume Voxel, the total volume is:

(math operation here)

Here:

1. N is the number of filled voxels.
2. pitch^3 is the volume of one voxel.

7.1.3. Compactness

7.1.4. Rectangularity

Rectangularity is a measure of how closely the shape of a 3D object approximates a rectangular box. It is computed as the ratio of the volume of the mesh to the volume of its oriented bounding box (OBB). The oriented bounding box is the smallest box (in terms of volume) that can completely enclose the mesh, but unlike the axis-aligned bounding box, it is rotated to minimize its volume and fit the mesh as tightly as possible, making the result independent of the shape's orientation.

A rectangularity value closer to 1 indicates that the shape closely resembles a rectangular box, while a value significantly less than 1 indicates that the shape is irregular or non-rectangular.

Mathematical Definition

Rectangularity is computed using the formula:

```
(math operation here)
```

Here:

1. The volume of the mesh is the actual 3D volume enclosed by the shape.
2. The volume of the OBB is the volume of the smallest enclosing oriented bounding box that fits around the shape.

The rectangularity value is always between 0 and 1 (assuming the mesh has a non-zero volume). If the shape is a perfect rectangular box, the mesh volume will be equal to the OBB volume, and the rectangularity will be 1. For more irregular shapes, the OBB volume will be larger than the mesh volume, and the rectangularity will be less than 1.

7.1.5. Diameter

The diameter of a 3D shape is defined as the largest distance between any two points on its surface. This descriptor gives insight into the size and extent of the shape by capturing the longest linear dimension that can fit within the object. The diameter provides a measure of the shape's overall "span" or reach.

In this implementation, the diameter is computed by finding the maximum Euclidean distance between pairs of vertices on the mesh's convex hull. The convex hull is the smallest convex shape that fully encloses the mesh, which reduces the complexity of the calculation compared to using all the vertices of the original mesh. Two methods are used to compute the diameter: **fast** and **slow**.

Fast Method: Vertex Sampling

The **fast** method improves computational efficiency by randomly sampling up to 200 vertices from the convex hull when the number of vertices exceeds this threshold. By reducing the number of vertices involved in the distance calculations, the method decreases the number of pairwise comparisons, which reduces the computational complexity while still providing a reasonably accurate estimate of the diameter.

In this method, the maximum distance between vertices vi and vj is calculated using the **Euclidean distance** formula:

```
(math operation here)
```

This distance is computed for all pairs of sampled vertices, and the maximum distance found is taken as the diameter of the shape. Since the number of vertices is reduced, the time complexity of this method is significantly lower, making it ideal for large meshes where a full pairwise comparison would be too costly.

Slow Method: Full Vertex Comparison

In the slow method, no vertex sampling is performed. Instead, the maximum distance is calculated between all pairs of vertices on the convex hull. This method ensures the most accurate calculation of the diameter by considering all possible vertex pairs, but at a higher computational cost.

For a mesh with n vertices, the number of pairwise comparisons in the slow method is **n(n−1)/2** which makes the time complexity $O(n^2)$. This method is more suitable for smaller meshes where a precise diameter measurement is critical and the computational cost is manageable.

7.1.6. Convexity

Convexity is a measure of how closely a 3D shape approximates its convex hull. The convex hull is the smallest convex shape that fully encloses the object. Convexity is defined as the ratio of the volume of the original mesh to the volume of its convex hull. This descriptor gives insight into how "convex" or "concave" a shape is. A convexity value close to 1 indicates that the shape is nearly convex (i.e., its volume is close to that of its convex hull), whereas a convexity value significantly less than 1 indicates that the shape has concave regions.

Mathematical Definition

Convexity is computed as:

```
(math operation here)
```

Here:

1. The volume of the mesh is the actual volume enclosed by the shape.
2. The volume of the convex hull is the volume of the convex shape that fully encloses the mesh.

The convexity value is always between 0 and 1 (assuming the mesh is non-degenerate). A convexity of 1 means that the mesh is exactly convex, while a value closer to 0 indicates significant concave features in the shape.

7.1.7. Eccentricity

Eccentricity is a measure of how elongated or stretched a 3D shape is along its principal axes. It is computed as the ratio of the largest to the smallest eigenvalues of the covariance matrix of the vertices of the mesh. The eigenvalues capture the variance of the mesh along its principal directions (which correspond to the eigenvectors). A high eccentricity value indicates that the shape is much more extended in one direction than in others, whereas an eccentricity value closer to 1 indicates that the shape is more isotropic, meaning its dimensions are more balanced.

Eccentricity is defined as:

```
(math operation here)
```

where:

1. λ1 is the largest eigenvalue of the covariance matrix (which corresponds to the direction of the greatest variance of the vertices).
2. λ3 is the smallest eigenvalue (which corresponds to the direction of the least variance).

The eigenvalues $\lambda 1$, $\lambda 2$, $\lambda 3$ are the variances along the principal axes of the shape. By taking the ratio of the largest and smallest eigenvalues, we obtain a quantitative measure of how "stretched" the shape is in the direction of its principal axes.

Covariance Matrix and Eigenvalues

The covariance matrix of the vertices captures how the coordinates vary with respect to each other. For a 3D mesh with N vertices, the covariance matrix is a 3×3 matrix that quantifies the relationships between the x-, y-, and z-coordinates of the vertices:

```
(math operation here)
```

where vi is the position vector of the i**-th** vertex and v^- is the centroid of the mesh. The eigenvalues of the covariance matrix correspond to the variance of the vertices along the principal axes, and the eigenvectors give the directions of these axes.

7.1.8. Sphericity

7.1.9. Elongation

Elongation is a measure of how "stretched" or "elongated" a 3D shape is. It is calculated as the ratio of the longest dimension to the second-longest dimension of the shape's oriented bounding box (OBB). The oriented bounding box is the smallest box that can enclose the shape, and it is rotated to fit the object as tightly as possible. Elongation gives insight into the shape's proportions, with higher values indicating a more elongated shape, while values closer to 1 indicate a shape that is more symmetrical across its major dimensions.

Elongation is computed as:

```
(math operation here)
```

Here:

1. Llongest is the length of the longest side of the oriented bounding box.
2. Lsecond longest is the length of the second longest side of the oriented bounding box.

By comparing the longest and second-longest dimensions, we get a measure of how stretched the object is along its main axis relative to its width.