Oriented Bounding Box generation

Overview

The ObbUtils library contains all Oriented Bounding Box algorithms. All Oriented Bounding Box Algorithms implement the IObbAlgorithm interface. Additionally the ObbUtils library contains an Obb-Tree implementation. The ObbTree class and ObbTreeNode classes are part of this implementation. The ObbUtils library also contains a range of useful extension methods for Obb's.

Obb Algorithms

The IObbAlgorithm interface contains the methods:

```
ObbContainer ComputeObbContainer(Vector3[] positions);

OrientedBoundingBox ComputeBoundingBox(Vector3[] positions);
```

When this method is executed, it returns a class named ObbContainer which contains the eigenvalues and bounding box computed by the implementation of the specific subclass of the interface. (The DITO-Algorithm Subclass currently doesn't implement eigenvalue computation and therefore it can't be used with the Hierarchy generation algorithm)

Obb Tree

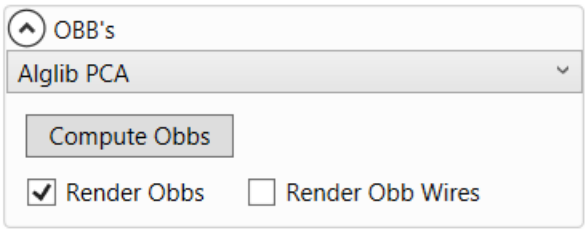
An Obb-Tree is created by using its constructor:

```
public ObbTree(Geometry3D geometry, IObbAlgorithm algorithm)
```

The constructor needs two parameters: The geometry for the Obb tree and the algorithm that is used for Obb generation. The Obb Tree is an implementation of the classical Obb-Tree algorithm. The leaves of the ObbTree contain geometry at triangle level (multiple triangles possible). The nodes of the Obb Tree are represented by the ObbTreeNode classes.

*Class diagrams …. Etc.

UI Overview

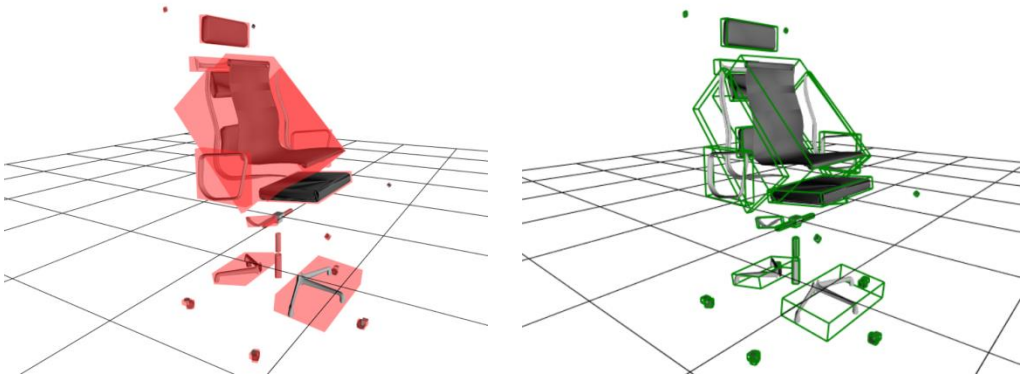


The default options are shown in the image above. The drop box allows to select one of the following algorithms:

1. Alglib (default): Use the Alglib library for eigenvalue computation and compute obbs with PCA
2. Matlab: Use Matlab for Obb computation. Requires an installed Version of Matlab. Requires installed Matlab Server. Can be problematic with larger models.
3. Ilnumerics PCA: Use the Ilnumerics library for eigenvalue computation and compute obbs with PCA.
4. DITO: Use the DITO Algorithm for Obb Computation. (Not compatible with hierarchy generation, no eigenvalue computation)

The other 2 options concern the rendering of the Obbs in the main viewport.

I. Render Obbs: Render the obbs as transformable Box models that move and rotate with the model.
II. Render Obb Wires. Render the obbs as non-transformable wires.



Symmetry detection

Overview

The Hierarchy Analysis library contains methods for detecting reflective, rotational and translational symmetries. The HierarchyAnalysis.SymmetryDetection namespace contains all Classes and methods necessary for symmetry analysis.

The static SymmetryDetector class contains different methods for symmetry detection. Symmetry detection works with a RANSAC approach for a number of trials.:

In each trial a symmetry candidate is generated. A reflective, translational or rotational transform between two or three parts (for rotations) is computed to this end. This transformation is then applied to each part. If the transformation maps this part to another part with similar eigenvalues the support counter is increased. If the support counter is above a certain hit percentage the symmetry is accepted. The best symmetry (with the biggest support count) is then returned.

The Symmetry detector is configured by a SymmetryDetectorSettings class that comprises several parameters: The thresholds for eigenvalues and centers in the symmetry detection process. The minimum support percentage for the parts defines how many part-pairs must support the symmetry for it to be accepted. The number of trials defines the number of RANSAC trials.

The SymmetryDetector class implements 3 methods for symmetry detection.

```
public static SymmetryStructure DetectDominantSymmetryTransform(List<ModelPart> parts
)
public static SymmetryStructure DetectSymmetryWithMaxSupport(List<ModelPart> parts)

public static List<SymmetryStructure> DetectAllSymmetryTransforms(List<ModelPart> par
ts)

public static List<SymmetryStructure> DetectSymmetryTransforms(List<ModelPart> parts)
```

All methods accept a List of ModelParts (with already computed obb-trees) as parameters. This is the list on which the SymmetryDetector performs its analysis. The methods return either a single symmetry structure describing the symmetry or a list of symmetry structures. A symmetry structure contains a Transformation that describes the symmetry (Transformation from each parts center to its symmetric counterpart), a SupportCount variable that contains the number of part-pairs that supported the Transformation in the RANSAC Step, a symmetry plane and the type of symmetry as an enum.

The first method (DetectDominantSymmetryTransform) scans the List of Parts for the most dominant transform. This method first uses the approach described above to find a reflective symmetry, if none is

found (to low hit percentage) it continues by searching for a translational symmetry if no translational symmetry is found it finds the best rotational symmetry.

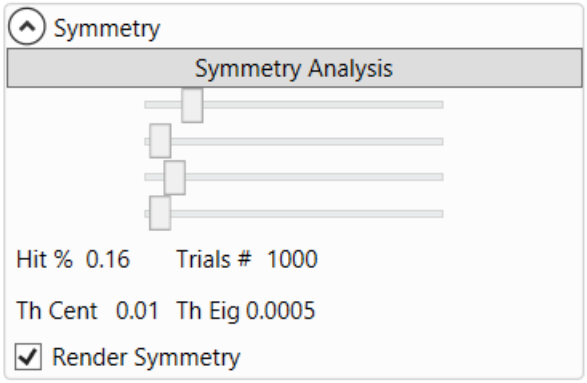
The method DetectSymmetryWithMaxSupport computes reflective, rotational and translational symmetries. It returns the symmetry that is supported best by all parts.

The method DetectAllSymmetryTransforms detects reflective rotational and translational symmetries of a model and returns them in an array.

The last method (DetectSymmetryTransforms) returns all Symmetries that exist within a model (and are supported by at least one part pair). This method is used to display all symmetries within the UI.

UI Overview

The symmetry analysis algorithm uses the settings from the obb panel for selection of the obb algorithm.



The image above shows the controls for the symmetry detection algorithm within the user interface.

The first slider is used to set the percentage of parts that must support the symmetry to be accepted. The value is shown below in the Hit % field. The second slider adjusts the number of trials for the RANSAC algorithm. The last two sliders are used to define the values for the eigenvalue threshold and center thresholds respectively. The checkbox is used to enable and disable the rendering of the symmetry model within the UI.

Contact Detection

Overview

The Hierarchy analysis library contains a namespace called HierarchyAnalysis.ContactAnalysis which comprises all classes and methods for contact detection.

The ContactAnalyzer class is the main class that performs contact detection. The ContactDetectionSettings class contains all settings for the contact detection algorithm.

Contact detection is performed by following method:

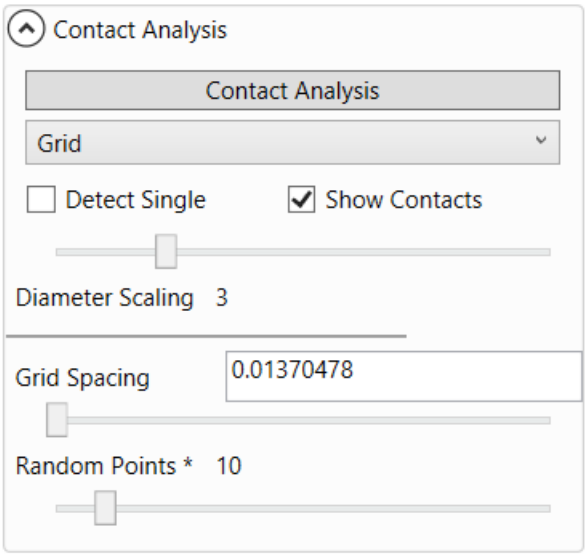
```
public static List<ModelPart> AnalyzeContacts(List<Geometry3D> geometries)
```

This method accepts a list of Geometry3D Objects (the model parts) for which contacts are detected. It returns a list of ModelParts with computed contacts and adjacent parts. Contacts are computed with the following algorithm:

First, each obb of each model part is tested with each other's model parts obb. If they overlap, the algorithm continues by traversing the obb trees of both parts to find all overlapping triangles of both parts. If two parts overlap on triangle level the algorithm refines each triangle by projecting additional points on each triangle. The way the points are projected depends on the selected Refinement algorithm (see UI Overview below). After the triangle positions are refined. The algorithm continues by measuring the distance of all triangle points from the first part to all refined points of the second part. If the distance is below a threshold (0.001% of the Model Obb diameter) each point is added to the list of contacts of both parts.

UI Overview

The contact detection algorithm uses the settings of the obb panel for selecting the obb algorithm.



The image above shows the UI Settings for the contact detection algorithm. The combobox selects the algorithm that is used for contact detection:

1. Grid: Projects a grid (aligned with the longest side of the triangle) onto all triangles of each model part and removes all points that don't lie inside the triangle. If this leads to less than 5 triangles per point, the algorithm generates a number of random points within the triangle. The number of points that are randomly generated in this case as well as the grid spacing can be set as parameters.
2. Lloyds algorithm: Uses Lloyds algorithm for triangle relaxation to generate the points within the triangles. The algorithm uses the Triangle.NET library for Voronoi diagram computation. Parameters that the algorithm accepts are: The number of random points that are generated in the initial phase, the maximum iterations for the Lloyd algorithm and the maximum of points per triangle that the Triangle.NET library is allowed to generate (Some triangle configurations force the Triangle.NET library to generate many points which can be slow.) The checkbox is experimental and tries to incorporate triangle size for the amount of random points that are generated initially.
3. Triangle.NET Smoothing: Uses the smoothing function of Triangle.NET to create a point distribution (similar to Lloyd). Parameters are the initial number of points as well as the maximum amount of iterations for the algorithm.

The option "Detect Single" stops the algorithm after detecting a single contact between two parts. If this option is deselected it will detect all contacts of the refined triangles. "Show Contacts" disables or enables the visibility of the contacts (shown as red dots). The diameter-scaling option allows to increase

the diameter for detection (the diameter of models bounding box is multiplied by this value, see above for details)

Hierarchy Generation

Overview

The hierarchy generation algorithm computes a part hierarchy that is based on the models symmetry and its part contacts. The implementation follows the Paper 3D-Model Recombination (INSERT REFERENCE) with several deviations:

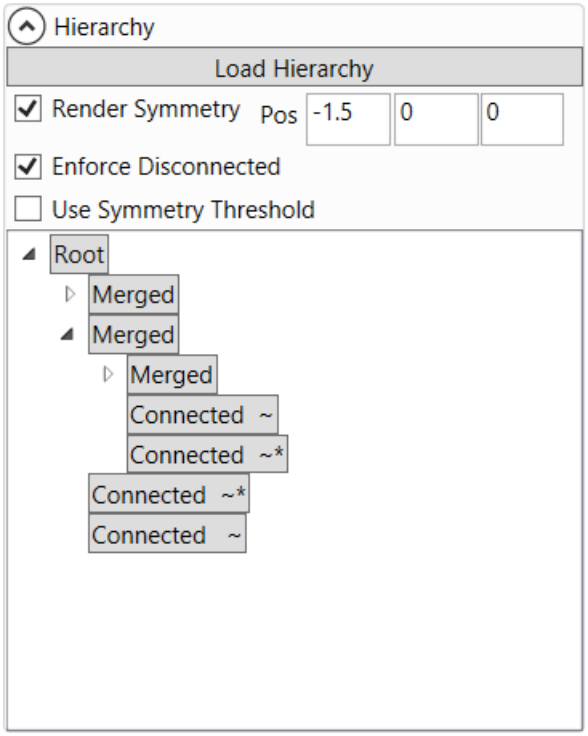
- No segmentation step is done in the beginning.
- The resampling step (sementation paragraph in the paper) is replaced by the contact refinement step decribed above. The amount of points generated is not equal to the amount of points of each individual part. (it is set per triangle)
- The algorithm uses an OBB Tree instead of an AABB Tree for contact-detection
- The support Threshold is used but set to 0% per default in the UI.

Summarized, the algorithm executes following steps:

1. Compute the Contacts of the models individual parts. (Contact-Refinement is included in this step)
2. Construct the tree by splitting each node-set by the current symmetry or if no symmetry is present (< Support threshold) use x, y od z = 0 planes (the longest side is preferred).

3. At each level merge parts again that share contacts (to ensure only connected parts within the hierarchy)

UI Overview



The image above shows the Hierarchy of a Chair model as well as the settings for the hierarchy generation algorithm within the hierarchy panel. The Load Hierarchy button starts the hierarchy generation process (The parameters for contact analysis, OBB generation and Symmetry detection within the hierarchy detection algorithm are taken from the respective panels). The Button “Render Symmetry” enables or disables rendering of the current selected symmetry node within the viewport. The three Pos fields define the coordinates where the symmetry node should be rendered within the viewport. The option “Enforce disconnected” enables or disables the node merge step of the algorithm. If it is disabled tree nodes can contain disconnected parts. The option “Use Symmetry Threshold” enables or disables the symmetry threshold from the symmetry panel. If this option is enabled and no dominant (> symmetry threshold) symmetry is found, the x,y and z = 0 planes are used for splitting, if possible. After hierarchy generation is finished, the generated tree is shown in the panel. By clicking a tree node the respective part of the tree is rendered within the viewport. Nodes can be expanded by clicking the small arrow at the left of each node. Nodes are labeled differently:

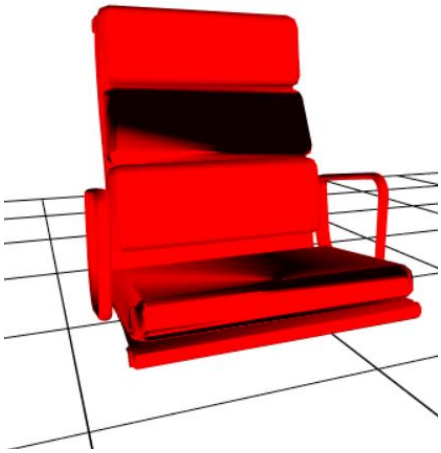
Merged – Nodes that were merged by the algorithm in the merge step

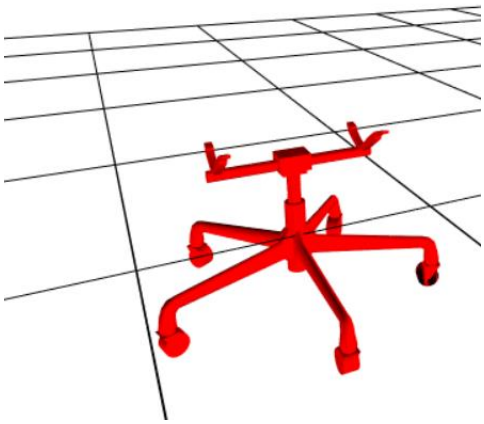
Connected – Nodes that were initially connected and have not been merged

Connected ~* The smallest connected node

Connected ~ The node was not merged because it had a similar size as the smallest connected node.

Disconnected – The node was and is disconnected from all other parts on this level.





Still missing Tasks:

- Refactor Main UI variables
- Refactor Load/Save
- Code doc
- Documentation pdf

Remaining Questions, Problems:

- Obb Tree Should implement Triangle Level Test.

- Improve initial Segmentation of Parts (remove bad groups etc.)

-Obb Tree needs to implement special cases (See RTR 3rd Edition)

- What happens with Nodes that are still disconnected on a level?

- Bad segmentation of model parts/groups leads to improper hierarchy

- Mode obb part size is used instead of part triangles size

- Model with clear symmetries work best

- Sharpdx Plane Buggy

- Models with long thin triangles -> Triangle.NET Problems (Many Points/Much Time needed)

-Generally: Models w. difficult triangles -> Problems for contact detection

-Contact detection tweaking (many random points, many iterations for good results)

- Triangle.NET undocumented, buggy

- Use different voronoi alg -> Test libs (simple Delaunay lib available and downloaded, use voronoi dual graph algorithm from wiki)

-Parameter Diameter Scale is important for Contact detection algorithm..