ProKlaue Documentation

Release 0.3.1

Enrico Reich

Sep 29, 2016

Contents

ProKlaue is a biomechanic research project at the Faculty of Veterinary Medicine of the University of Leipzig (Saxony, Germany). The primary goal is to be able to align bone models in a repeatable and deterministic way, as well as to export comparable measures of each bone model over a whole animation for further analysis.

To support the necessary work and to provide some useful routines inside the 3D-modelling and animation program *Maya Autodesk* the plugin *proKlaue* was written, which uses the Python-API of Maya and Python-Libraries for numerical computation (numpy, scipy, V-HACD). The Plugin introduces not only different scripts which are registered as commands inside Maya (for easier usage) but also a number of useful functions for general mesh-based tasks. There are a few additional functions like calculation of the convex hull, delaunay triangulation, intersection volume of multiple mesh objects and a cleanup process where one can extract a specific shell of a mesh (to eliminate possible entrapments inside a bone model caused by e.g. air, vessels or imaging errors).

CHAPTER 1

Requirements

- Maya Autodesk 2013 (or newer)
- Numpy Package for Python2.7 (at least 0.12.0)
- Scipy Package for Python2.7 (at least 0.16.0)
- V-HACD Library

1.1 Installation of Numpy & Scipy for Maya

Maya Autodesk is shipped with its own Python2.7-Installation which does not make any use of already existing Python-Installations on the system. This includes all Python-Libraries as well. So in order to make Python-Libraries work directly inside Maya, the following steps are necessary:

- Installation of Python 2.7 for the current system
- Install *Numpy* and *Scipy* (either using the Linux install repository or the *pip*-command of python, e.g. 'pip install -i https://pypi.anaconda.org/carlkl/simple numpy')
- Search subdirectory 'site-packages' in Python2.7 install path
- Select and copy directories *numpy* & *scipy*
- Go to the Maya Directory (e.g. 'C://Program Files/Autodesk/Maya2014/Python/lib/site-packages') and insert copied files

After a restart of Maya the command 'import numpy' should be usable; if not the wrong version was installed.

1.1.1 Troubleshooting

- **command pip was not found**: navigate to the python-subdirectory *Scripts* ('cd "C://Program Files/Python27/Scripts/"') and try again. Or insert Path 'Python27/Scripts' to PATH-Variable
- **Maya Error: module 'numpy' not found**: the 'numpy' directory is in the wrong Maya subdirectory. It needs to be inside Maya's Python-Directory 'lib/site-packages/'
- **Maya throws dll-Error with reference to Win32**: wrong Python-Version and consequently wrong library versions are installed. Since Maya Autodesk 2014 there is only a 64bit version. This means that Python also needs to be installed as 64bit version. It is also advisable to update 'pip' ('python -m pip install -U pip') **before** installation of the libraries. Further the packages should be installed without any optimizations to avoid conflicts.

CHAPTER 2

Installation and configuration of the Plugin

- Copy everything to 'Autodesk/maya<version>/bin/plug-ins' so that the file 'proKlaue.py' lies directly inside this directory
- Inside Maya: Windows –> Settings/Preferences –> Plugin-in Manager –> Refresh
- Search for the entry 'proKlaue.py' and check 'Loaded' and 'Auto load'

After the plugin has been successfully loaded a new shelf tab named 'ProKlaue' will appear.

At last, activate the following setting: Windows –> Settings/Preferences –> Preferences –> Setting –> Selection –> 'Track selection order'

To use the console commands directly inside Maya, one need to execute the Python command 'import maya.cmds as cmds' after each program restart (this can be done automatically by creating a file 'userSetup.py' inside the directory '/home/user/maya/version/scripts/' and write the line 'import maya.cmds as cmds' inside). All commands can then be executed by typing 'cmds.<cmd>' where *cmd* is a placeholder for the command's name.

CHAPTER 3

Commands

3.1 adjustAxisDirection

Command list Calculates the projection of all points in currently selected mesh onto a given axis. Returns the given axis or its inverse depending on the distribution of projected points: if more points lie 'above' the median (based on the range of all point) then axis itself will be returned, if more points lie 'below' median then inverse of axis is returned. Main purpose is to use this function on one of the main axes. Command only accepts 'transform' nodes. Command will only use first object in current selection.

command: cmds.adjustAxisDirection([obj], axis)

Args:

obj string with object's name inside maya

axis(a) vector as list of 3 values where object's points are projected to (e.g. [1,0,0])

Example:

```
cmds.polyPyramid()
# [u'pPyramid1', u'polyPyramid1'] #
cmds.adjustAxisDirection(a = [0,1,0])
# Result: [-0.0, -1.0, -0.0] #
```

3.2 axisParallelPlane

Command list Inserts an axis parallel plane (app) with normal vector along x-,y- or z-axis to be tangential to the minimum/maximum vertex of a selected object (in the specified axis direction), i.e. the plane has exactly one contact point with the selected object in its minimum/maximum x, y or z direction. Additionally the position of the plane is set to be below/above the mesh object's center point. The axis parallel plane can be calculated for one single time step or a complete animation. In case of a parent node with the type 'joint' the keyframes will be automatically catched (whether the transform node or joint node is selected, the behaviour will be the same). If the first keyframe in the animation is 'separated' from all other frames, i.e. the frame distance from first to second is much larger than from second to third frame, this first keyframe will be ignored in the calculation of the axis parallel plane.

see also: *centerPoint*, *altitudeMap*

command: cmds.axisParallelPlane([obj], plane = 'yz', position = 'min')

Args:

plane(p) string ('xy', 'xz', 'yz') to specifiy plane direction (default 'xz')

position(pos) string ('min', 'max') to specify if plane should be positioned at the minimal/maximal vertex position concerning the plane orientation (default 'min')

anim(a) boolean flag to indicate if plane should be calculated for the whole animation (True) or only for the current time step (default False)

returns name of created polyPlane object (name will be object name plus '_app')

Example:

```
cmds.polyTorus()
# Result: [u'pTorus1', u'polyTorus1'] #
cmds.axisParallelPlane(p = 'xz', pos = 'min')
# Result: [u'pTorus1_app', u'polyPlane1'] #
# A new plane 'pTorus1_app' is inserted which is
# positioned under the torus (-0.5 units below world origin)
```

3.3 altitudeMap

Command list Uses an axis parallel plane (app) and the object model connected to this plane to create an altitude map, i.e. a set of perpendicular distances from the faces of the object to the plane. The distance is measures from the plane to the centroid of each face. A ray is constructed from each of the faces to the plane and only those faces without any other intersection than the plane (only faces directly visible from the plane) are considered part of the altitude map. Points with a larger distance than a given threshold will be discarded.

see also: *axisParallelPlane*

command: cmds.altitudeMap([obj, plane], file = "", threshold = 10.0)

Args:

file(f) path to save altitude map to ASCII-file; if string is empty, no data will be written

threshold(t) threshold of maximum distance from plane; all points with larger distance will be discarded (default 10.0)

anim(a) boolean flag to indicate if altitude map shall be calculation for each frame (TRUE) or only for the current frame (FALSE, default). If TRUE a file name needs to be specified, because the amount of data could massively slow down maya if its only kept in work memory.

returns list of centroid points of faces, their indices in the mesh vtx-list and their distances to the plane, i.e. '[[n_1, x_1, y_1, z_1, d_1], [n_2, x_2, y_2, z_2, d_2], ...]'

Example:

```
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
cmds.xform(ro = [10, 20, 30])
cmds.axisParallelPlane(p = 'xz', pos = 'min')
# Result: [u'pCube1_app', u'polyPlane1'] #
cmds.polyTriangulate("pCube1")
# Result: [u'polyTriangulate1'] #
cmds.altitudeMap('pCube1', 'pCube1_app')
# Result: [[4, -0.3983890351647723, 0.0597721458595899, -0.3785089467837944, 0.
↪7449914933365491],
# [5, 0.019866728794979728, -0.07780045709588722, -0.5469076316145288, 0.
↪607418890381072],
# [6, 0.021764807311746848, -0.5225944965679014, -0.1788206947620432, 0.
↪16262485090905787],
```

```
# [7, 0.4192048032181355, -0.3599696226914842, 0.01564478359550836, 0.
↪32524972478547504],
# [10, -0.3964909566480051, -0.3850218936124242, -0.010422009931308688, 0.
↪300197453864535],
# [11, -0.4173067247013684, -0.08482441678052995, 0.35244215325697736, 0.
↪6003949306964292]] #
```

3.4 alignObj

Command list Calculates the eigenvectors of the current object, inserts one extra column and row and returns a 4x4 transformation matrix as 16 float-values. The eigenvectors are calculated in another function called *eigenvector*. Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

see also: *eigenvector*, *normalize*

command: cmds.alignObj([obj], axisOrder = 'yzx', fast = False)

Args:

obj string with object's name inside maya

axisOrder(ao) string to define axis order of eigenvectors (default 'yzx')

fast(f) boolean flag to indicate if calculation should use convex hull (faster but inaccurate)

Example:

```
cmds.polyTorus()
# Result: [u'pTorus1', u'polyTorus1'] #
cmds.alignObj(ao = 'xyz')
# Result: [0.609576559498125, 0.7927272028323672, -5.465342261024642e-10, 0.0, -1.
↪3544498855821985e-09, 3.520841396209562e-10, -1.0, 0.0, -0.7927272028323671, 0.
↪6095765594981248, 1.288331507658196e-09, 0.0, 0.0, 0.0, 0.0, 1.0] #
cmds.xform(m = cmds.alignObj(ao = 'xyz'))
# torus is positioned 'upwards' in x-y-plane
```

3.5 centerPoint

Command list Calculates the average of all points (vertices of mesh) of current selection. This command is useful to define the origin of an arbitrary object's local coordinate frame. Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

see also: *centroidPoint*, *axisParallelPlane*, *normalize*, *exportData*

command: cmds.centerPoint([obj])

Args:

obj string with object's name inside maya

Example:

```
cmds.polyTorus()
# Result: [u'pTorus1', u'polyTorus1'] #
cmds.centerPoint()
# Result: [-8.046627044677735e-09, -5.513429641723633e-08, -1.1246651411056519e-
↪07] #
```

```
cmds.polyPyramid()
# Result: [u'pPyramid1', u'polyPyramid1'] #
cmds.centerPoint()
# Result: [1.2363445733853951e-08, -0.21213203072547912, -1.2363447865482158e-08]
↪#
```

3.6 centroidPoint

Command list Calculates the weighted average of all triangle centroids of current selection. This command is useful to define the origin of an arbitrary object's local coordinate frame in case of irregular triangulation of the object's mesh. Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

see also: *centerPoint*

command: cmds.centroidPoint([obj])

Args:

obj string with object's name inside maya

Example:

```
cmds.polyTorus()
# Result: [u'pTorus1', u'polyTorus1'] #
cmds.centroidPoint()
# Result: [8.981527616005896e-10, -9.250593323493368e-08, -3.1820641749655423e-
↪07] #
cmds.polyPyramid()
# Result: [u'pPyramid1', u'polyPyramid1'] #
cmds.centroidPoint()
# Result: [8.125617691761175e-09, -0.20412414173849558, -8.125620434757627e-09] #
cmds.centerPoint()
```

3.7 cleanup

Command list Main purpose of this command is the extraction of the outer shell of an arbitrary object. Natural bones may have natural inclusions due to air bubbles, blood vessels or imaging errors. During animation these inclusions can obstruct a clear view when the bone model is set half-transparent and inflate computation time. To extract only the outer bone shell one needs to delete all separate inclusions and possibly cut through 'connecting tubes', i.e. small blood vessels connecting an outer shell with an inner shell. A threshold can be set to determine and delete all triangles of the bone model whose normal's distance (orthogonal distance from center point of triangle) to the next triangle is smaller than this threshold.

The command implements a button *clean* under the shelf tab 'ProKlaue', where one can find and list all shells of a currently selected transform node. The shells can then be selected to show which part of the bone model is part of this shell. A number behind each shell indicates the amount of triangles inside each set and finally one shell can be extracted, so that all other shells and their triangles will be deleted from the mesh.

The command primarily handles all the user interaction inside maya and processes output of commands *getShells* and *findTubeFaces*.

Command should only be used over the button interface

3.8 convexHull

Command list Calculates the convex hull using the scipy-library (based on *Qhull*) of the current object and creates a new transform node. The name of the new transform node will be the object's name with the suffix '_ch' Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

command: cmds.convexHull([obj])

Args:

obj string with object's name inside maya

returns name of transform node with convex hull mesh

Example:

```
cmds.polyTorus()
# Result: [u'pTorus1', u'polyTorus1'] #
cmds.convexHull()
# Result: pTorus1_ch #
```

3.9 coordinateSystem

Command list Initializes a local coordinate system for the given object consisting of 3 colored axes. Will be grouped under the given transform object. Can be used in the export-command instead of the normalized position. Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

see also: *normalize*, *exportData*

command: cmds.coordinateSystem([obj], ao = 'yzx', f = False)

Args:

obj string with object's name inside maya

axisOrder(ao) string with axis ordering of eigenvectors (default 'yzx')

fast(f) flag to indicate if covariance matrix should use the convex hull (True) or all points (False) (default True)

3.10 delaunay

Command list Calculates the 3D delaunay triangulation (scipy library) on the vertices of a mesh object to have a real 3D solid representation of a given object model. Because the delaunay triangulation in scipy returns an unsorted list of vertices for each tetrahedron, the vertices are reordered to comply with the implicit normal definition used in Maya Autodesk. The result will be the triangulated convex hull of the given object mesh.

Command currently only returns a list of strings where each string encodes the vertices of one tetrahedron (as flattened 1D list) with its correct vertex order for an implicit normal definition (all normals are pointing away from center point of tetrahedron). That means, that the result needs to be converted to matrices (nested lists) again using 'numpy.fromstring(str, sep=",").reshape(4,3)' for each string 'str' in the returned list. The reason for this behavior is the fact, that one cannot return nested lists in Maya commands and the only workaround is to cast them to strings first.

Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

command: cmds.delaunay([obj])

Args:

obj string with object's name inside maya

returns list of tetrahedrons as strings (each string represents 4 vertices with 3 points each separated by ','. Numbers [0:3] are first vertex, [3:6] second vertex, ...)

Example:

```
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
cmds.delaunay()
# Result: [u'0.5, -0.5, 0.5, -0.5, 0.5, 0.5, 0.5, 0.5, -0.5, -0.5, -0.5, 0.5', u'-
↪0.5, 0.5, 0.5, -0.5, 0.5, -0.5, 0.5, 0.5, -0.5, -0.5, -0.5, 0.5', u'0.5, 0.5, -
↪0.5, -0.5, -0.5, -0.5, 0.5, -0.5, 0.5, -0.5, -0.5, 0.5', u'-0.5, 0.5, -0.5, -0.
↪5, -0.5, -0.5, 0.5, 0.5, -0.5, -0.5, -0.5, 0.5', u'0.5, 0.5, -0.5, 0.5, 0.5, 0.
↪5, -0.5, 0.5, 0.5, 0.5, -0.5, 0.5', u'-0.5, -0.5, -0.5, 0.5, -0.5, -0.5, 0.5, 0.
↪5, -0.5, 0.5, -0.5, 0.5'] #
```

3.11 eigenvector

Command list Calculates the eigenvectors and eigenvalues of the covariance matrix of all points in current object's mesh. The eigenvector with the largest eigenvalue corresponds to the first axis defined in axis order, second largest to second axis and third largest to third axis. Command is used by *alignObj* and *exportData*. Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

The calculation of the covariance matrix is defined as:

$$C = [c_{i,j}] = \left[\left(\frac{1}{a^H}\sum_{k=0}^{n-1}\frac{a^k}{12}(9m_i^k m_j^k + p_i^k p_j^k + q_i^k q_j^k + r_i^k r_j^k)\right) - m_i^H m_j^H\right]$$

where $m^H = \frac{1}{a^H}\sum_{k=0}^{n-1} a^k m^k$ is the centroid of the convex hull with $m^i = \frac{p^i+q^i+r^i}{3}$ as centroid of triangle i and the surface of the convex hull $a^H = \sum_{k=0}^{n-1} a^k$. The area of triangle k with its is vertices $\Delta p^k q^k r^k$ is defined as a^k.

The eigenvectors and eigenvalue of C are calculated using *numpy.linalg.eigh*.

see also: *alignObj*, *exportData*

command: cmds.eigenvector([obj], ao = 'yzx', f = False)

Args:

obj string with object's name inside maya

axisOrder(ao) string to define axis order of eigenvectors (default 'yzx')

fast (f) boolean flag to indicate if calculation should use convex hull; faster but inaccurate (default False)

returns list of 9 float values corresponding to first eigenvector ([0:3]), second eigenvector ([3:6]) and third eigenvector ([6:9])

Example:

```
cmds.polyTorus()
# Result: [u'pTorus1', u'polyTorus1'] #
cmds.eigenvector()
# Result: [5.465342261024642e-10, -0.609576559498125, 0.7927272028323672, 1.0, 1.
↪3544498855821985e-09, 3.520841396209562e-10, -1.288331507658196e-09, 0.
↪7927272028323671, 0.6095765594981248] #
```

3.12 exportData

Command list Exports the specified information for a list of objects for a whole animation. Information will be written to separate files. The file names will be the object names with a possible prefix. The files itself will contain a tab-separated table with one entry/row for each animation time step. Because the whole export runs in Maya's main thread, there is currently no possibility to cancel already started export commands and one needs to wait until the export is finished which, depending on the length of the animation and number of selected objects, can take a few minutes (to speed up the export with multiple models, one can move the camera away from the scene and all objects, so the rendering step runs much faster). The current progress of the export always shows the active time frame and for each selected object the whole animation will run through once.

All the information will be tracked via maya's space locators which sometimes may show unexpected behaviour. The space locators is not directly set inside the bone model but at the normalized position, where the bone model would be after the execution of the command *normalize*. Alternatively the flag *localCoordinateSystem* allows to define an arbitrary coordinate system which will be used instead of the automatically calculated one. This coordinate system needs to be grouped under the transform node of the current object. Additionally the normalized position of each bone model is taken in world coordinates from the first keyframe > 0, which may affect the overall orientation. Lastly, in combination with joint-hierarchies it does not matter, which object is selected (the transform bone model or the parent joint node), the script will automatically combine the keyframes of both transform node and joint node, find the transform node as child of the joint node (respectively the joint node as parent of the transform node) and apply all necessary transformations to the mesh object. Requirement for this behaviour to work is the explicit hierarchical order (joint nodes should have only ONE direct transform node as child and each transform node must have an unique parent joint node; to attach multiple transform nodes under a common ancestor, one needs to use intermediate joint nodes for each level of the hierarchy).

see also: *normalize*, *coordinateSystem*

command: cmds.exportData([obj], p = "", fp = "", cm = "centerPoint", lcs = False, jh = False, f = False, tm = True, wt = True, wa = True, wr = True, ao = 'yzx')

Args:

path(p) path to target directory

filePrefix(fp) prefix of all files

centerMethod(cm) string 'centerPoint' (default) to use average of all points as position, 'centroidPoint' to use weighted mean of triangle centroid and area as position or 'centerOBB' to use center of oriented bounding box as position (only when 'align' is True)

localCoordinateSystem(lcs) flag to indicate if objects have their own local coordinate system (True) or the normalized orientation (False) shall be used for export data (default False)

jointHierarchy(jh) flag to indicate if objects are organized in an hierarchy (True) or are completely independent of each other (default False)

fast(f) flag to indicate if covariance matrix should use the convex hull (True) or all points (False) (default FALSE)

writeTransformM(tm) flag to indicate if transform matrix shall be written to file

writeTranslation(wt) flag to indicate if translation shall be written to file

writeAngles(wa) flag to indicate if projected angles shall be written to file (deprecated)

writeRotations(wr) flag to indicate if rotation values shall be written to file (deprecated)

axisOrder(ao) string with axis ordering of eigenvectors (default 'yzx')

animationStart(as) first time step where information shall be exported (default *animationStartTime* of **playbackOptions**)

animationEnd(ae) last time step where information shall be exported (default *animationEndTime* of **playbackOptions**)

animationStep(by) time difference between two animation frames (default *by* of **playbackOptions**)

3.13 findTubeFaces

Command list Calculates the triangles of current object with an orthogonal distance to the next surface below a given threshold, which indicates a tube/tunnel to connect two shells of the object with each other. All those triangles will be selected and can be deleted by the standard user action. Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

see also: *cleanup*, *getShells*

command: cmds.findTubeFaces([obj], t = 0.1)

Args:

obj string with object's name inside maya

threshold(t) threshold of orthogonal distance between triangles. All triangles closer to each other than this threshold will be selected (default 0.1)

3.14 getShells

Command list Calculates all shells of current selection and returns them as list of strings. One shell are all triangles which are connected, i.e. there exists a path from each triangle to all other triangles using common edges. Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

command: cmds.getShells([obj])

Args:

obj string with object's name inside maya

returns list of strings. Each string stands for one shell with a list of indices separated by comma and enclosed by '[...]', e.g. ['[0, 1, 2, 3, 4, ...]', '[...]', '[...]', ...]

3.15 getVolume

Command list Calculates the volume of a polygonal object by using signed volumes. All triangle faces of the mesh will be expanded to tetrahedra which are topped off at the origin (0,0,0). Sign of the volume is determined by the direction of the origin from the current face (source). Command only accepts 'transform' nodes and will only be applied to the first object of the current selection.

There is a mel-command in Maya named computePolysetVolume which calculates the volume of the selected object. The Maya command returns the exact same result (except numerical error $< 10^{-5}$) as this implementation but runs roughly 100 times slower.

Args:

obj string with object's name inside maya

returns float value with volume (based on maya's unit of length)

Example:

```
cmds.polyTorus()
# Result: [u'pTorus1', u'polyTorus1'] #
cmds.getVolume()
# Result: 4.774580351278257 #
# MEL command
computePolysetVolume();
makeIdentity -apply true -t 1 -r 1 -s 1 -n 0 -pn 1;
// pTorus2 faces = 400 //
// TOTAL VOLUME = 4.774580265 //
// Result: 4.77458 //
```

3.16 intersection

Command list Calculates an approximate intersection volume of two transform objects by intersection of the tetrahedra given by the delaunay triangulation of the convex decomposition of both objects. Because the convex hull usually is a poor approximation of the original object, the V-HACD library is used to find an approximate convex decomposition of the object itself and then use the convex parts to create triangulated tetrahedra to approximate the intersection volume by pairwise intersection tests.

So both objects will be decomposed in convex parts, every part will be triangulated with the delaunay triangulation and their tetrahedra will be intersected pairwise. The sum of the intersection volume of all pairwise tetrahedra is the sum of the intersection of both convex decompositions. To speed up the calculation there is a first evaluation which determines all intersection candidates for a tetrahedron from the first convex hull with possible tetrahedra of the second convex hull. Candidates are all those tetrahedra which lie within or intersect the axis aligned bounding box (their minimal and maximal range in each axis need to overlap). Secondly there is a more accurate collision test with all the candidates which determines which candidates actually intersect the current tetrahedron (using the axis separating theorem, see *collision_tet_tet*). Finally all remaining tetrahedra will be intersected and their intersection volume will be calculated (see *intersection_tet_tet*).

All necessary functions are implemented without any Maya commands to speed up calculation; from past experience it can be concluded that Maya commands are much slower than a fresh implementation in Python. An Interval Tree approach was tested and did not lead to performance improvements.

Command only accepts 'transform' nodes and multiple objects can be selected. The output will be a matrix with the volume of the convex hulls for each object in the main diagonal and in the upper triangular matrix are the pairwise intersections between each combination of objects. Additionally the volumes of the objects itself (not their convex decompositions) will be printed during runtime.

The parameter arguments are the same as for the command *vhacd* and will be simply handed down to the vhacd command invocation.

see also: *collision_tet_tet*, *intersection_tet_tet*, *getVolume*, *vhacd*

command: cmds.intersection([obj], kcd = True)

Args:

obj string with object's name inside maya

keepConvexDecomposition(kcd) should the convex decompositions (intermediate data) be kept (True, default) or deleted (False)

tmp(temporaryDir) directory to save temporary created files (needs read and write access). If no path is given the temporary files will be written into the user home directory

executable(exe) absolute path to the executable V-HACD file. If no path is given maya/bin/plug-ins/bin/testVHACD is used.

resolution(res) maximum number of voxels generated during the voxelization stage (10.000 - 64.000.000, default: 100.000)

depth(d) maximum number of clipping stages. During each split stage, all the model parts (with a concavity higher than the user defined threshold) are clipped according the "best" clipping plane (1 - 32, default: 20)

concavity(con) maximum concavity (0.0 - 1.0, default: 0.001)

planeDownsampling(pd) controls the granularity of the search for the "best" clipping plane (1 - 16, default: 4)

convexHullDownsampling(chd) controls the precision of the convex-hull generation process during the clipping plane selection stage (1 - 16, default: 4)

alpha(a) controls the bias toward clipping along symmetry planes (0.0 - 1.0, default: 0.05)

beta(b) controls the bias toward clipping along revolution axes (0.0 - 1.0, default: 0.05)

gamma(g) maximum allowed concavity during the merge stage (0.0 - 1.0, default: 0.0005)

normalizeMesh(pca) enable/disable normalizing the mesh before applying the convex decomposition (True/False, default: False)

mode(m) voxel-based approximate convex decomposition (0, default) or tetrahedron-based approximate convex decomposition (1)

maxNumVerticesPerCH(vtx) controls the maximum number of triangles per convex-hull (4 - 1024, default: 64)

minVolumePerCH(vol) controls the adaptive sampling of the generated convex-hulls (0.0 - 0.01, default: 0.0001)

returns string containing a nxn matrix (n: number of objects) with intersection volumes.

raises RunTimeError if volume of convex decomposition is smaller than volume of given object. Indicates that there are holes inside convex decomposition which obviously leads to incorrect results for the intersection volume. Solution is to choose smaller depth paramter

Example:

```
cmds.polyTorus()
# Result: [u'pTorus1', u'polyTorus1'] #
cmds.polyTorus()
# Result: [u'pTorus2', u'polyTorus2'] #
cmds.xform(ro = [90,0,0])
cmds.makeIdentity(a = 1, r = 1)
cmds.polyTorus()
# Result: [u'pTorus3', u'polyTorus3'] #
cmds.xform(ro = [0,0,90])
cmds.makeIdentity(a = 1, r = 1)
select -r pTorus3 pTorus2 pTorus1 ;
cmds.intersection(kcd = 0)
volume 0: 4.77458035128
volume 1: 4.77458035128
volume 2: 4.77458035128
# Result: [[ 5.97810349  1.91942589  1.92072237]
 [ 0.          5.97378722  1.91621988]
 [ 0.          0.          5.97281007]] #
```

3.17 normalize

Command list Normalize selected objects by aligning the eigenvectors of the covariance matrix to the world cordinate system and using the center point of each object as the origin of its local coordinate system.

see also: *alignObj*, *eigenvector*, *centerPoint*, *centroidPoint*

command: cmds.normalize([obj], s = 1.0, tZ = True, pZ = True, a = True, ao = 'yzx', f = False, cm = 'centerPoint')

Args:

scale(s) float value to scale all objects and their distance to the origin

transToZero(tZ) flag to indicate if objects shall be translated to world space origin (0,0,0) or remain at their position (default True)

pivToZero(pZ) flag to indicate if objects pivots shall be translated to world space origin (0,0,0) or remain at their position (default True)

align(a) flag to indicate if object shall be aligned according to eigenvectors (default True)

axisOrder(ao) string of length 3 with axis order (default 'yzx'). Will only be considered if flag 'align' is True

fast(f) flag to indicate if covariance matrix shall be computing using all points (False) or only the points in the convex hull (True). Will only be considered if flag 'align' is True (default False)

centerMethod(cm) string 'centerPoint' (default) to use average of all points as position, 'centroidPoint' to use weighted mean of triangle centroid and area as position or 'centerOBB' to use center of oriented bounding box as position (only when 'align' is True)

3.18 rangeObj

Command list Calculates the projection of all points in currently selected mesh onto a given axis. Returns a list of 3 values with min[p.x + p.y + p.z], max[p.x + p.y + p.z] and abs(max - min). Function's main purpose is to calculate the range of an object with respect to the x,y,z-axes. Command only accepts 'transform' nodes.

command: cmds.rangeObj([obj], a)

Args:

obj string with object's name inside maya

axis(a) vector as list of 3 values where object's point are projected to

Example:

```
cmds.polyPyramid()
# Result: [u'pPyramid1', u'polyPyramid1'] #
cmds.rangeObj(a = [0,1,0])
# Result: [-0.3535533845424652, 0.3535533845424652, 0.7071067690849304] #
```

3.19 vhacd

Command list Uses the V-HACD library (https://github.com/kmammou/v-hacd) to calculate an approximate convex decomposition for a number of selected or given objects. Because Maya's object representation is usually a surface mesh, it is sometimes necessary to convert the object to a solid geometry (especially when using complex objects

or boolean operations). Using convex decomposition avoids problems concerning holes or non-manifold geometry. The original object will be approximated by a finite number of convex polyhedra organized in one group per original object.

The available properties and settings of V-HACD are taken directly from the description page.

Script makes use of the *wrl2ma*-command under *maya/bin* to parse to and from different formats. Apparently there are some old parser-inconsistencies which may produce warning or error messages, which (as far as observed) have no visible effect on the produces mesh. Also the path to the directories *maya/bin* for *wrl2ma* and *plug-ins/bin* for V-HACD is guessed through the environment path variable of the operating system.

NOTE: There are observed cases where V-HACD was not able to create a real solid geometry for a given mesh (without any topological holes) but rather creates a convex decomposition where the summed volume of the convex parts is smaller than the volume of the original mesh, i.e. the decomposition creates a topologically different model where the inner area is hollow. Therefore one should always verify the output mesh and in case of a faulty decomposition choose different parameter settings (a smaller *depth*-value usually avoids this problem).

see also: *getVolume*, *intersection*

command: cmds.vhacd([obj], tmp = '~/', exe = '../maya/bin/plug-ins/bin/testVHACD', res = 100000, d = 20, con = 0.001, pd = 4, chd = 4, a = 0.05, b = 0.05, g = 0.0005, pca = False, m = 0, vtx = 64, vol = 0.0001)

Args:

tmp(temporaryDir) directory to save temporary created files (needs read and write access). If no path is given the temporary files will be written into the user home directory

executable(exe) absolute path to the executable V-HACD file. If no path is given maya/bin/plug-ins/bin/testVHACD is used.

resolution(res) maximum number of voxels generated during the voxelization stage (10.000 - 64.000.000, default: 100.000)

depth(d) maximum number of clipping stages. During each split stage, all the model parts (with a concavity higher than the user defined threshold) are clipped according the "best" clipping plane (1 - 32, default: 20)

concavity(con) maximum concavity (0.0 - 1.0, default: 0.001)

planeDownsampling(pd) controls the granularity of the search for the "best" clipping plane (1 - 16, default: 4)

convexHullDownsampling(chd) controls the precision of the convex-hull generation process during the clipping plane selection stage (1 - 16, default: 4)

alpha(a) controls the bias toward clipping along symmetry planes (0.0 - 1.0, default: 0.05)

beta(b) controls the bias toward clipping along revolution axes (0.0 - 1.0, default: 0.05)

gamma(g) maximum allowed concavity during the merge stage (0.0 - 1.0, default: 0.0005)

normalizeMesh(pca) enable/disable normalizing the mesh before applying the convex decomposition (True/False, default: False)

mode(m) voxel-based approximate convex decomposition (0, default) or tetrahedron-based approximate convex decomposition (1)

maxNumVerticesPerCH(vtx) controls the maximum number of triangles per convex-hull (4 - 1024, default: 64)

minVolumePerCH(vol) controls the adaptive sampling of the generated convex-hulls (0.0 - 0.01, default: 0.0001)

returns list of Maya group names where each group corresponds with the approximate convex decomposition of one object (structure is: group name | sub group | convex mesh)

CHAPTER 4

Misc

4.1 collision_tet_tet

Misc Module to test for collision of two tetrahedra using the axis separating theorem. Based on the paper "Fast tetrahedron-tetrahedron overlap algorithm" by F. Ganovelli, F. Ponchio and C. Rocchini.

Uses two lists of vertices for each tetrahedra. The class function Collision_tet_tet.check() returns True iff the two given tetrahedra actually intersect (or at least have common vertices). Before calling 'Collision_tet_tet.check()' the two tetrahedra need to be set with the methods **setV1**, **setV2** or **setV1V2**.

class pk_src.collision_tet_tet.**Collision_tet_tet**(*tetra1=None*, *tetra2=None*)
Initialize tetrahedra definitions.

Parameters

- **tetra1** – list of the 4 3D-vertices describing the first tetrahedron
- **tetra2** – list of the 4 3D-vertices describing the second tetrahedron

check()
Function to check if current two tetrahedra can intersect; optimized for speed.

Returns True, iff the two tetrahedra intersect (or have common vertices).

separating_plane_edge_A(*f0*, *f1*)
Helper function for tetrahedron-tetrahedron collision: checks if edge is in the plane separating faces f0 and f1.

Parameters

- **coord** – 4x4 list of lists
- **self.masks** – 4x1 list of lists with single element
- **f0** – integer
- **f1** – integer

separating_plane_faceA_1(*pv1*, *n*, *coord*, *mask_edges*)
Helper function for tetrahedron-tetrahedron collision test: checks if plane pv1 is a separating plane. Stores local coordinates and the mask bit mask_edges.

Parameters

- **pv1** – vectors between vertices of second tetrahedron and vertex V[0] of first tetrahedron (list of lists [[...],[...],...])
- **n** – normal (list [x,y,z])

- **coord** – reference to list
- **mask_edges** – reference to list with single element [m]

Returns True if pv1 is a separating plane

separating_plane_faceA_2(*n*, *coord*, *mask_edges*)
Helper function for tetrahedron-tetrahedron collision test: checks if plane v1,v2 is a separating plane. Stores local coordinates and the mask bit mask_edges.

Parameters

- **V1** – list with vertices of tetrahedron 1
- **V2** – list with vertices of tetrahedron 2
- **n** – normal
- **coord** – reference to list
- **mask_edges** – reference to list with single element [m]

separating_plane_faceB_1(*pv2*, *n*)

separating_plane_faceB_2(*n*)

setV1(*V1*)
Set list of vertices for first tetrahedron.

Parameters **V1** – list of the 4 3D-vertices describing the first tetrahedron

setV1V2(*V1*, *V2*)
Set list of vertices for first and second tetrahedron.

Parameters

- **V1** – list of the 4 3D-vertices describing the first tetrahedron
- **V2** – list of the 4 3D-vertices describing the second tetrahedron

setV2(*V2*)
Set list of vertices for second tetrahedron.

Parameters **V2** – list of the 4 3D-vertices describing the second tetrahedron

shifts = [1, 2, 4, 8]

pk_src.collision_tet_tet.**cross**(*x*, *y*)
cross product as lambda function to speed up calculations

pk_src.collision_tet_tet.**dot**(*x*, *y*)
dot product as lambda function to speed up calculations

4.2 intersection_tet_tet

Misc Module to to calculate 3D intersection of two tetrahedra. The second tetrahedron will be clipped against the first tetrahedron with a simplified sutherland-hodgman approach.

pk_src.intersection_tet_tet.**centroid_tri**(*v1*, *v2*, *v3*)
centroid of tri (3 3D-points) as lambda function to speed up calculation

pk_src.intersection_tet_tet.**cross**(*x*, *y*)
cross product as lambda function to speed up calculation

pk_src.intersection_tet_tet.**dot**(*x*, *y*)
dot product as lambda function to speed up calculation

class pk_src.intersection_tet_tet.**intersection_tet_tet**(*tetra1=None*, *tetra2=None*)
Class to initialize tetrahedra and set normals and centroids

Parameters

- **tetra1** – list of the 4 3D-vertices describing the first tetrahedron
- **tetra2** – list of the 4 3D-vertices describing the second tetrahedron

intersect()
Intersection calculation according to simplified sutherland-hodgman approach. Clip the second tetrahedron against the first one. At the end the triangles of the convex hull of all vertices will be returned

Returns list of triangles with 3D-coordinates which form the 3D convex intersection volume

setCentroid()
Calculate centroids of tetrahedron for each 4 facets.

setNormals()
Calculate normals of tetrahedron for each 4 facets.

setV1(*V1*)
Set list of vertices for first tetrahedron and set normals and centroids.

Parameters **V1** – list of the 4 3D-vertices describing the first tetrahedron

setV1V2(*V1*, *V2*)
Set list of vertices for first and second tetrahedron and set normals and centroids.

Parameters

- **V1** – list of the 4 3D-vertices describing the first tetrahedron
- **V2** – list of the 4 3D-vertices describing the second tetrahedron

setV2(*V2*)
Set list of vertices for second tetrahedron.

Parameters **V2** – list of the 4 3D-vertices describing the second tetrahedron

pk_src.intersection_tet_tet.**normalize**(*v*)
normalization of 3D-vector as lambda function to speed up calculation

4.3 misc

Misc Module with a few helper functions for plugin *ProKlaue*.

pk_src.misc.**areaTriangle**(*vertices*)
Returns the area of a triangle given its three vertices.

Parameters **vertices** – list of 3 vertices defining the triangle

Returns float value with area of triangle

Example:

```
from pk_src import misc
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
points = [[p.x, p.y, p.z] for p in misc.getPoints("pCube1")]
```

```
misc.areaTriangle([points[0], points[1], points[2]])
# Result: 0.5 #
```

pk_src.misc.**centroidTriangle**(*vertices*)
Returns the centroid of a triangle given its three vertices.

Parameters **vertices** – list of 3 vertices defining the triangle

Returns list of 3 float values representing the 3D-coordinates of the triangle's centroid

Example:

```
from pk_src import misc
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
points = [[p.x, p.y, p.z] for p in misc.getPoints("pCube1")]
misc.centroidTriangle([points[0], points[1], points[2]])
# Result: [-0.16666666666666666, -0.16666666666666666, 0.5] #
```

pk_src.misc.**getArgObj**(*syntax*, *argList*)
Method to return list of objects from argument list (throws exception if no object is given or selected).

Parameters

- **syntax** (*OpenMaya.MSyntax (api 1.0)*) – Function to syntax definition of current command
- **argList** – Argument list for command

Raises **AttributeError** – Raised when no object is selected

Returns list of object names in argument list (or from current selection)

pk_src.misc.**getFaceNormals**(*obj*, *worldSpace=True*)
Method to return all face normals. Uses the command polyInfo to access the normal information as string and parse them to a numpy array

Parameters **obj** (*string*) – name of object

Returns list of numpy arrays with the 3 float values for each normal

Example:

```
from pk_src import misc
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
misc.getFaceNormals("pCube1")
# Result: [[0.0, 0.0, 1.0], [0.0, 1.0, 0.0], [0.0, 0.0, -1.0], [0.0, -1.0, 0.
↪0], [1.0, 0.0, 0.0], [-1.0, 0.0, 0.0]] #
```

pk_src.misc.**getMesh**(*obj*)
Method to return mesh object from dag path.

Parameters **obj** (*string*) – name of object

Returns MFnMesh object (api 2.0)

pk_src.misc.**getPoints**(*obj*, *mfnObject=None*, *worldSpace=True*)
Method to return MPointArray with points of object's mesh.

Parameters

- **obj** (*string*) – name of object
- **mfnObject** (*MFnMesh*) – mesh object
- **worldSpace** (*Boolean*) – should coordinates be in world space (with transform) or in local space (without transform); default True

Returns MPointArray (api 2.0)

Example:

```
from pk_src import misc
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
cmds.xform(t = [2,1,0])
# vertices in local space (without transform)
misc.getPoints("pCube1", worldSpace = 0)
# Result: maya.api.OpenMaya.MPointArray([maya.api.OpenMaya.MPoint(-0.5, -0.5,␣
↪0.5, 1), maya.api.OpenMaya.MPoint(0.5, -0.5, 0.5, 1), maya.api.OpenMaya.
↪MPoint(-0.5, 0.5, 0.5, 1), maya.api.OpenMaya.MPoint(0.5, 0.5, 0.5, 1), maya.
↪api.OpenMaya.MPoint(-0.5, 0.5, -0.5, 1), maya.api.OpenMaya.MPoint(0.5, 0.5,␣
↪-0.5, 1), maya.api.OpenMaya.MPoint(-0.5, -0.5, -0.5, 1), maya.api.OpenMaya.
↪MPoint(0.5, -0.5, -0.5, 1)]) #
# vertices in world space (with transform)
misc.getPoints("pCube1", worldSpace = 1)
# Result: maya.api.OpenMaya.MPointArray([maya.api.OpenMaya.MPoint(1.5, 0.5, 0.
↪5, 1), maya.api.OpenMaya.MPoint(2.5, 0.5, 0.5, 1), maya.api.OpenMaya.
↪MPoint(1.5, 1.5, 0.5, 1), maya.api.OpenMaya.MPoint(2.5, 1.5, 0.5, 1), maya.
↪api.OpenMaya.MPoint(1.5, 1.5, -0.5, 1), maya.api.OpenMaya.MPoint(2.5, 1.5, -
↪0.5, 1), maya.api.OpenMaya.MPoint(1.5, 0.5, -0.5, 1), maya.api.OpenMaya.
↪MPoint(2.5, 0.5, -0.5, 1)]) #
```

pk_src.misc.**getTriangleEdges**(*triangles*)

Method to return all edges of the given triangle list as dictionary. Given triangle must use indices to point list, so the key-value pairs only contain indices to the vertices' list; key is the index of the current vertex and for each outgoing edge the value contains an index to the corresponding vertex. Values are always lists of indices.

Parameters **triangles** (*[[1, 2, 3], [1, 3, 4], ...]*) – list of triangle definitions (in index represenation, see *misc.getTriangles*)

Returns dict (keys: index of current vertex, values: list of vertices with existing edge)

Example:

```
from pk_src import misc
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
cmds.polyTriangulate()
# Result: [u'polyTriangulate1'] #
misc.getTriangleEdges(misc.getTriangles("pCube1"))
# Result: {0: [1, 2, 4, 6, 7],
        1: [0, 2, 3, 7],
        2: [0, 1, 3, 4],
        3: [1, 2, 4, 5, 7],
        4: [0, 2, 3, 5, 6],
        5: [3, 4, 6, 7],
```

```
        6: [0, 4, 5, 7],
        7: [0, 1, 3, 5, 6]} #
```

pk_src.misc.**getTriangles**(*obj*, *mfnObject=None*)

Method to return all triangles in object mesh as nested list. Length of list is equal to number of triangles for the object. Each list element is another list of 3 vertex indices which refer to the point set of the current object mesh.

Parameters

- **obj** (*string*) – name of object
- **mfnObject** (*MFnMesh*) – mesh object

Returns numpy.ndarray (e.g. array([[2,1,3],[2,3,4],[4,3,5],[4,5,6],...]) for a polygon cube)

Example:

```
from pk_src import misc
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
misc.getTriangles("pCube1")
# Result: array([[0, 1, 2],
       [2, 1, 3],
       [2, 3, 4],
       [4, 3, 5],
       [4, 5, 6],
       [6, 5, 7],
       [6, 7, 0],
       [0, 7, 1],
       [1, 7, 3],
       [3, 7, 5],
       [6, 0, 4],
       [4, 0, 2]]) #
```

pk_src.misc.**project**(*p*, *v*)

Orthogonal projection of one vector onto another.

Parameters

- **p** (*[x,y,z]*) – vector to be projected
- **v** (*[x,y,z]*) – vector where p shall be projected to

Returns orthogonal projection vector [x,y,z]

pk_src.misc.**signedVolumeOfTriangle**(*p1, p2, p3, center=[0, 0, 0]*)

Calculates signed volume of given triangle (volume of tetrahedron with triangle topped off at origin (0,0,0))

Parameters

- **p1** (*[x,y,z]*) – first point of triangle
- **p2** (*[x,y,z]*) – second point of triangle
- **p3** (*[x,y,z]*) – third point of triangle
- **center** (*[x,y,z]*) – top of tetrahedron (default (0,0,0))

Returns signed volume of tetrahedron

Example:

```
from pk_src import misc
cmds.polyCube()
# Result: [u'pCube1', u'polyCube1'] #
points = [[p.x, p.y, p.z] for p in misc.getPoints("pCube1")]
misc.signedVolumeOfTriangle(points[0], points[1], points[2])
# Result: 0.08333333333333333 #

# Volume of whole object
triangles = misc.getTriangles("pCube1")
reduce(lambda x,y: x+y, [misc.signedVolumeOfTriangle(points[tri[0]],␣
↪points[tri[1]], points[tri[2]]) for tri in triangles])
# Result: 1.0 #
```

CHAPTER 5

HowTo: Add new commands to plugin *ProKlaue*

The file *proKlaue.py* is the master-file of the plugin where all submodules will be individually imported and registered as commands. To keep the general structure and clarity one should always use separate files/modules for each new command. Additionally the file name, module name and command name should be equal to simplify the registration steps.

To add a new command *cmds.foo* implemented in the source file *foo.py* to the plugin, one only needs to add the name *foo* to the list of commands *kPluginCmdName* in *proKlaue.py* (l. 57) and it will be automatically imported, registered with syntax and button definition:

```
kPluginCmdName = ["centerPoint", "centroidPoint", "normalize", "eigenvector",
   "alignObj", "exportData", "rangeObj", "convexHull", "getShells",
   "findTubeFaces", "cleanup", "getVolume", "delaunay", "intersection",
   "adjustAxisDirection", "vhacd", "foo"]
```

The necessary parts of a new command need to be defined in a source file *foo.py*:

```
import maya.OpenMayaMPx as OpenMayaMPx
import maya.OpenMaya as om
import maya.cmds as cmds
from functools import partial

class foo(OpenMayaMPx.MPxCommand):
   windowID = "wFoo"

   def __init__(self):
      OpenMayaMPx.MPxCommand.__init__(self)

   def __cancelCallback(*pArgs):
      if cmds.window(exportData.windowID, exists = True):
      cmds.deleteUI(exportData.windowID)

   def __applyCallback(self, textF, *pArgs):
      options = {"text":cmds.textField(textF, q = 1, text = 1)}
      cmds.foo(**options)

   def createUI (self, *pArgs):
      if cmds.window(self.windowID, exists = True):
         cmds.deleteUI(self.windowID)
      cmds.window(self.windowID, title = "fooUI", sizeable = True,␣
↪resizeToFitChildren = True)
      cmds.rowColumnLayout(numberOfColumns = 2)
      text = cmds.textField(visible = True, width = 140)
      cmds.button(label = "apply", command = partial(self.__applyCallback, text),␣
↪width = 100 )
```

```
        cmds.button(label = "cancel", command = self.__cancelCallback, width = 100)
        cmds.showWindow()

    def doIt(self, argList):
        argData = om.MArgParser (self.syntax(), argList)
        text = argData.flagArgumentString("text", 0)
        print(text)

def fooCreator():
    return OpenMayaMPx.asMPxPtr( foo() )
def fooSyntaxCreator():
    syntax = om.MSyntax()
    syntax.addFlag("t", "text", om.MSyntax.kString)
    return syntax
def addButton(parentShelf):
    cmds.shelfButton(parent = parentShelf, i = "pythonFamily.png",
        c=foo().createUI, imageOverlayLabel = "foo", ann="do something")
```

In this echo-server example a command button is defined with an input text field and two buttons (*apply* and *cancel*) inside the user interface. By pressing button *apply* the text field content will be echoed to the console whereas *cancel* closes the user interface. The same effect (without the user interface) can be achieved by typing *cmds.foo(t = 'echo')*.

All emphasized lines are required definitions: class **foo** inherited from *OpenMayaMPx.MPxCommand*, its constructor **foo.__init__**, the function **foo.doIt** which will be triggered at command invocation, function **fooCreator** which just returns a pointer to a class instance, function **fooSyntaxCreator** which returns the syntax definition and function **addButton** where a shelf button under the shelf tab *ProKlaue* can be defined. If no syntax definition is needed, the function **fooSyntaxCreator** still needs to return an object of type *MSyntax*, otherwise the command registration will fail with an error message. Same applies to the function **addButton(parentShelf)**: if no button is needed, this function is still required for the registration step. See following case without syntax definition and shelf button:

```
def fooSyntaxCreator():
    syntax = om.MSyntax()
    return syntax
def addButton(parentShelf):
    pass
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

p