Implementation of Hereditary Convex Structures

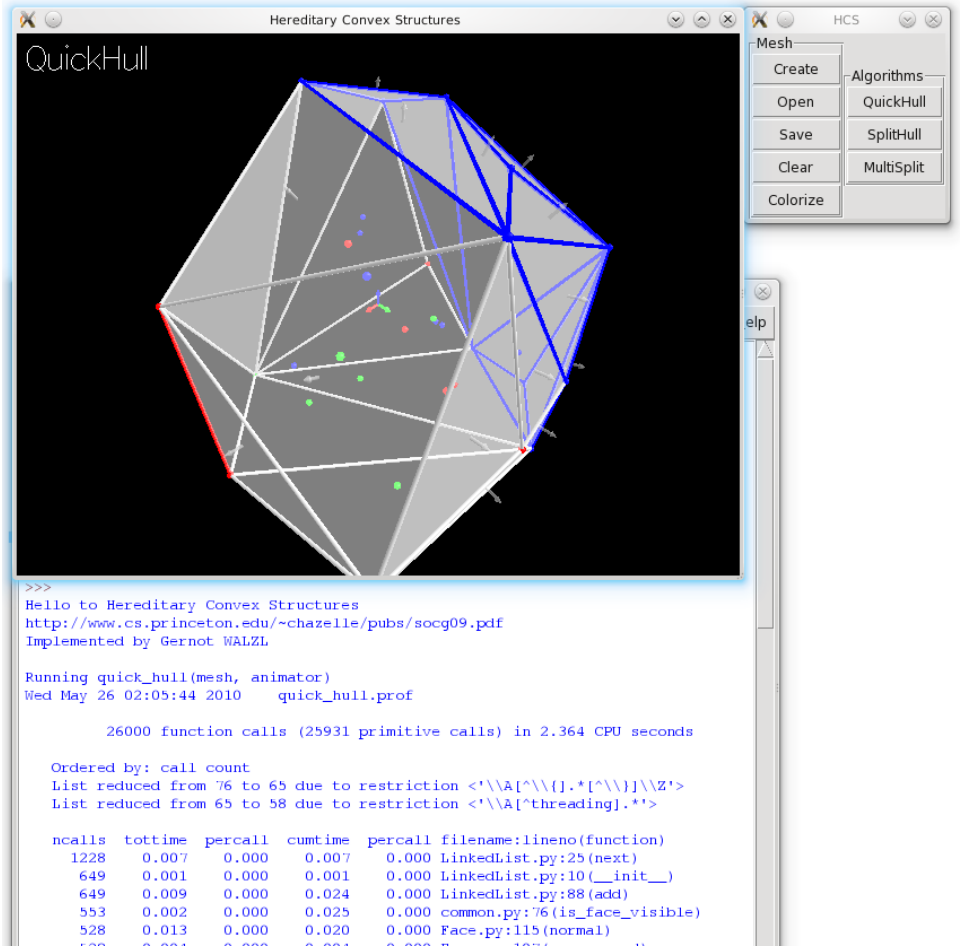
Gernot Walzl, BSc
Graz University of Technology
gernot.walzl@student.tugraz.at

2010-05-31

Abstract

This article describes the implementation of a tool to animate algorithms operating in 3-dimensional space. Good representatives for such algorithms are those introduced by B. Chazelle and W. Mulzer: *Computing Hereditary Convex Structures*.

This tool may be used to animate any algorithm operating in 3-dimensional space. It should not be necessary to implement any data structure to reach minimum running time.



Why can't we just see what everybody has to imagine?

Contents

1 Basics about Python

1.1 Why Python?

The first argument to answer this question was made on a philosophical point of view.

In compiled languages like C++ there always exist a binary and the source code, which are two different things. This is an advantage if we consider execution speed or if we want to keep intellectual property safe.

For educational purposes I find it is better, if the source code itself is executable. That means there is no difference between source and binary. This is the case in interpreted languages, like Python is. Another advantage is, that it speeds up development time. On the other hand it reduces execution speed on a constant factor.

Comparing Python to other interpreted languages we may observe the following advantages:

- Well developed object-oriented features.
- The source of Python is open.
- Runs on many platforms like Windows, Linux, Mac OS, BSD, ...
- Huge community [LQ2009:MCA], which means it is easy to get support.
- Many libraries available
  - Linear Algebra: NumPy, SciPy
  - Computer Graphics: PyOpenGL, PyGame
  - et cetera: accessing database server, serving web pages, creating reports
- C API available: possible to access libraries coded in C/C++

One of the main differences to other languages is that Python requires correct indentation of blocks. Many other languages use "{" to start a block and "}" to end one. Python identifies a block by the level of indentation. This forces the code to, at least, look structured.

It will be assumed that the reader has basic C++ or Java knowledge. A few differences to these languages will be explained in the following sub-sections. Many aspects described there can be found in the official documentation of Python [PythonDoc2010].

1.2 How to import Classes from Modules/Files

Each `*.py` file defines a module for Python. Such a module can contain many functions and classes. There are two ways to use such a function or class. The first possibility is to import the whole module. The second one is to import some functions or classes only.

Here is an example how to import the whole module:

```
>>> import math
>>> math.sqrt(20)
4.4721359549995796
```

The following example imports only one function from a module:

```
>>> from math import sqrt
>>> sqrt(20)
4.4721359549995796
```

As shown above, there is a difference in how to call the function which depends on how we have imported it. In C-terms we would say the namespaces are different.

To organize the modules hierarchically, it is possible to put them into packages. Packages in Python are equal to packages in Java. They are just folders containing some source files.

The following example imports a class called `Node` from a module called `node`, which is inside the package `data`.

```
>>> from data.node import Node
>>> a = Node()
```

A syntax as shown is not necessary in Java. By definition the name of the class has to be equal to the filename which contains it. This is not the case in Python, but many coding conventions recommend to do so.

1.3 Objects have a variable set of attributes

Objects in Python do not have a fixed set of attributes. An attribute gets appended to the object by setting it the first time. Usually this is done in the class defined constructor.

Here is an example of how it could look like:

```
>>> class Complex:
...     def __init__(self, real, imag):
...         self.real = real
...         self.imag = imag
...
```

```
>>> a = Complex(1.0, 2.0)
>>> b = Complex(3.0, 4.0)
>>> a.mutation = 5.0
>>> a.mutation
5.0
>>> b.mutation
AttributeError: Complex instance has no attribute 'mutation'
```

On a well developed object-oriented architecture it is not recommended to use such an approach. Inheritance should be the way to success.

1.4 Built-in Data Structures

To do a time complexity analysis of algorithms, it is necessary to know the running time ($T(n)$) of implemented data structures. For our Mesh data structure we will make heavily use of a list. To achieve needed running times for this data structure, it is necessary to have a list implementation with constant time for appending or deleting an element.

1.4.1 Lists

The source code for Pythons built-in list is available at [PythonSrc2010, `listobject.c`]. As it is shown in the source code, the default built-in list is implemented as array-list with memory preallocation for resizing: "The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ..."

Therefore we implement a linked-list that, for sure, has constant running time to append or delete an element.

1.4.2 Dictionaries

Dictionaries are implemented as hash maps. Let's quote [PythonSrc2010, `dictnotes.txt`]: "Dictionary operations involving only a single key can be O(1) unless resizing is possible."

1.5 Garbage Collection

Python counts the references to each object. If there are no references left, the object gets destroyed. The destruction happens exactly then when the last reference is lost. This makes the destruction of garbage deterministic, but causes problems with circular references. For example: object a references object b and object b references object a. These two objects will never be destroyed, if not at least one reference is set to `None`.

Beginning with Version 2.0 of Python, they added a Garbage Collector that is similar to the one used in Java. Object references are still counted to be deterministic in destruction. In case we have forgotten to break cyclic references, the garbage collector will do it for us. This makes it impossible to produce memory leaks, like it may happen in C++.

1.6 Multithreading and Locking

In multithreaded applications locking mechanisms are necessary to prevent race conditions (inconsistent states). Python supports a higher level threading and locking interface [PythonDoc2010, threading].

A nice feature to mention here is the "reentrant lock". These locks are easier to handle than normal locks, because a thread can not block itself. This increases the difficulty to produce unwanted dead locks (application freezes).

When acquiring a lock, it will be checked if the lock has an owner and if the current thread owns this lock. In that case, the current thread will not be blocked. If the lock has an owner which is not the current thread the current thread will be blocked. When the lock is released, another thread can gain ownership of this lock by acquiring it.

1.7 Profiling

To ensure that the implemented algorithms meet the stated running time of complexity analysis, we need a way to measure the number of function calls. This is done by profiling the running application [PythonDoc2010, profile].

1.8 pydoc

Collaboration with other people is easier with good documentation. With tools like `pydoc` such a documentation can automatically be generated by using Python source code [PythonDoc2010, pydoc]. This tool is similar to `javadoc`, but the documentation is always generated on the fly, directly from the source files.

The usage is quite easy. Simply change to the directory containing the source files. After this execute `pydoc -p 1234`. This command will start a local http server on port 1234. Now start a web-browser of your choice and open `http://localhost:1234/`.

2 OpenGL

The Open Graphics Library (OpenGL) is a specification defining a language to produce 2D and 3D computer graphics. Every graphics card company supplies a library, which allows to access the graphics card with standardized OpenGL calls. This library is usually part of the graphics card driver.

A byte compiled machine code is required to access any piece of hardware. Therefore hardware drivers have an interface to a low-level hardware-near language. This is the reason, why the C programming language is most commonly used to access hardware libraries.

2.1 Basic OpenGL Usage

OpenGL is a state machine. Each OpenGL call changes a specified state. For example: The position to draw is a state. First moving to a position and then drawing an object is different to first drawing an object and then moving to a specified position. The position is part of the model view matrix which is modified by `glTranslate` function.

Before it is possible to draw anything on screen, the environment has to be set up. This includes mode of texturing, model of lighting, register drawing function and so on. After initialization the gl main loop gets called. This causes the drawing function to be called in an endless loop. It has to clear the screen and draw everything. It could draw every primitive (triangles, points, lines) "by hand" or use a standardized toolkit like GLUT.

2.2 GLUT

The Graphics Utility Toolkit (GLUT) simplifies the use of OpenGL. It is no longer necessary to draw every triangle on our own. GLUT gives the possibility to easily draw a sphere or a cylinder with one function call. The triangles of the object to be drawn will be generated and aligned by the toolkit.

2.3 PyOpenGL

PyOpenGL [PyOpenGL2010] is a simple wrapper to use OpenGL calls in Python. It uses Pythons C API [PythonDoc2010, c-api] to forward each function call in Python to the OpenGL library. PyOpenGL also supports GLUT. Examples of famous tutorials, like NeHes OpenGL Tutorials, were ported to PyOpenGL and distributed by this project.

However OpenGL and its toolkits are still state machines and do not use an object-oriented approach. A state of the art software architecture requires object-oriented interfaces.

2.4 Object-Oriented Approach

Let's assume we want to have two OpenGL windows with different content displayed. As described before, OpenGL is a hardware library to access one graphics card, GLUT supports one main loop only and OpenGL is a state machine. One main loop only is not able to call more than one drawing function. These facts lead to the problem that we simply do not know which content we have to draw inside our drawing function. Luckily there exists a function named `glutGetWindow` which returns the current active window. By using a hash map, the reference to the corresponding object is found. This allows to forward the call to the correct drawing function.

To be less state dependent, the implemented class has methods like `draw_pipe` or `draw_text`. These methods take as parameter where to draw and an orientation to draw. The model view matrix gets stored before any other OpenGL function is called. After processing all OpenGL calls, this matrix will be restored.

This approach makes it possible to separate different contents into different objects and reduces the need to know the state before calling a function.

2.5 User Interface

2.5.1 Input

Similar to registering a drawing function, keystroke and mouse event functions are registered. When processing a button, this function gets called. It has to determine which button was pressed and what action it should perform. The keystroke function gets evaluated once a frame is drawn. In case we want to move around in the 3D environment with keystrokes, moving speed would directly depend on how many frames per second (fps) are calculated. To be performance independent, we only process the button down and button up event. This event sets a boolean variable to either true or false. The "KeyboardInputThread" checks this variable frequently at predefined time steps. The position gets adjusted at each timed cycle of the thread. This makes the moving speed independent to the performance of the application.

2.5.2 Output

To produce a 2D picture of an 3D environment, it is essential, where the camera is positioned. GLU provides a function to set the camera with 3 vectors: position of the camera, where to look at and an up-vector describing where upside is.

An example of the output is shown in figure 1.

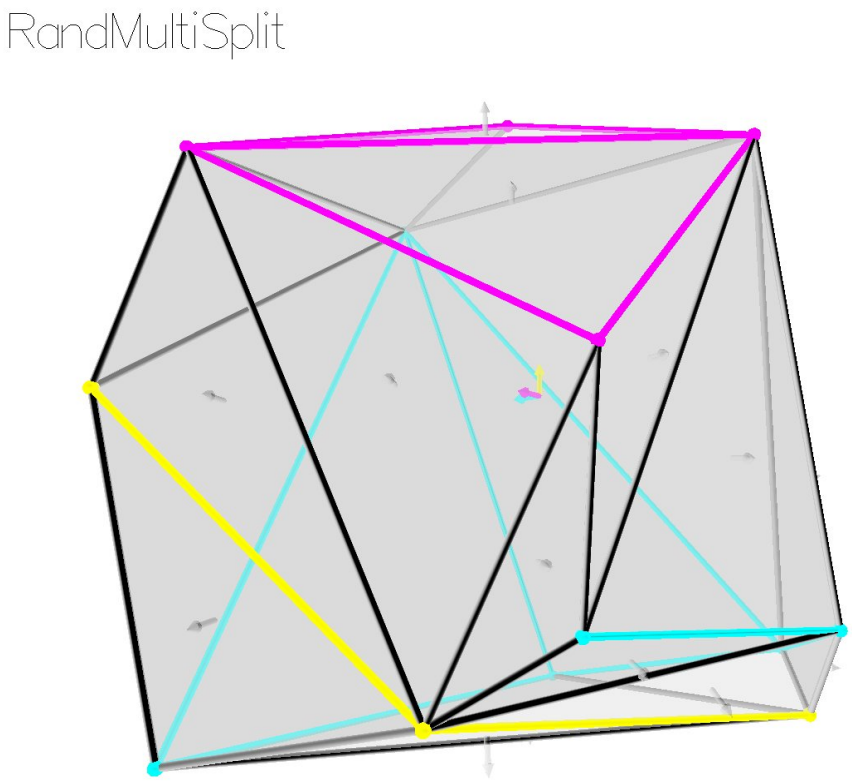


Figure 1: OpenGL Animation

2.5.3 Class Looker

This class is the glue between the input and the `gluLookAt` function. It exists mainly of 3D trigonometry. The looker is able to move around and look around. The z-axis is always up.

3 Data Structure

We process a set of n unsorted points with a given useful algorithm (see section 5). The only possibility to reach minimum computation time is that every operation on the data structure takes constant time ($O(1)$).

3.1 Linked List

A linked list consists of a variable number of elements. Each element contains some data and has a pointer to the next element. Table 1 shows a comparison between linked lists and arrays.

Operation	Linked List	Array
Indexing	$O(n)$	$O(1)$
Appending	$O(1)$	$O(n)$

Table 1: set operation times

Index operations are not used, because we always store the list element which holds corresponding data. This assures that only constant time operations are processed.

3.2 Mesh

Beginning with a comparison between polygon mesh data structures, summarized in Table 3, we observe that a Render Dynamic Mesh fits best for our needs. One problem remains in the standard Render Dynamic Mesh: Finding a face or an edge inside the structure does not work in constant time. To deal with this problem, a hash map is able to return a reference to the object found. As the reader may know, hash maps have expected constant time for this operation. For that reason the standard Render Dynamic Mesh got extended with hash maps. The improvement is shown in Table 2.

Operation	Render Dynamic	extended with hash map
add/remove node/edge/face	$O(1)$	$\sim O(1)$
find node/edge/face	$O(n)$	$O(1)$
get neighbors of node	$O(1)$	$O(1)$

Table 2: mesh operation times

Operation		Vertex-Vertex	Face-Vertex	Winded-Edge	Render Dynamic
V-V	All vertices around vertex	Explicit	V → f1, f2, f3, ... → v1, v2, v3, ...	V → e1, e2, e3, ... → v1, v2, v3, ...	V → e1, e2, e3, ... → v1, v2, v3, ...
E-F	All edges of a face	F(a,b,c) → a,b, b,c, a,c	F → a,b, b,c, a,c	Explicit	Explicit
V-F	All vertices of a face	F(a,b,c) → a,b,c	Explicit	F → e1, e2, e3 → a, b, c	Explicit
F-V	All faces around a vertex	Pair search	Explicit	V → e1, e2, e3 → f1, f2, f3, ...	Explicit
E-V	All edges around a vertex	V → v,v1, v,v2, v,v3, ...	V → f1, f2, f3, ... → v1, v2, v3, ...	Explicit	Explicit
F-E	Both faces of an edge	List compare	List compare	Explicit	Explicit
V-E	Both vertices of an edge	E(a,b) → a,b	E(a,b) → a,b	Explicit	Explicit
Flook	Find face with given vertices	F(a,b,c) → a,b,c	Set intersection of v1,v2,v3	Set intersection of v1,v2,v3	Set intersection of v1,v2,v3
Storage size		V*avg(V,V)	3F + V*avg(F,V)	3F + 8E + V*avg(E,V)	6F + 4E + V*avg(E,V)
		Example with 10 vertices, 16 faces, 24 edges			
		10 * 5 = 50	3*16 + 10*5 = 98	3*16 + 8*24 + 10*5 = 290	6*16 + 4*24 + 10*5 = 242

Table 3: summary of mesh representation operations [Wiki2010:PM]

Having a mesh data structure which is able to do all operations in constant time enables to use it for every algorithm in 3-dimensional space. If one data structure is suitable for nearly all algorithms in 3-dimensional space, it is reasonable to write a multi-purpose tool that animates many algorithms of that kind.

An example of a tetrahedron is shown in Table 4 to understand how the render dynamic mesh data structure works. A node contains a list of adjacent edges and faces of variable length.

n0	-0.5, -0.5, -0.5	e1 e2 e5	f0 f2 f3
n1	0.5, 0.5, -0.5	e0 e1 e4	f0 f1 f2
n2	-0.5, 0.5, 0.5	e0 e2 e3	f0 f1 f3
n3	0.5, -0.5, 0.5	e3 e4 e5	f1 f2 f3

(a) Node List

e0	n2 n1	f0 f1
e1	n1 n0	f0 f2
e2	n0 n2	f0 f3
e3	n2 n3	f1 f3
e4	n3 n1	f1 f2
e5	n3 n0	f2 f3

(b) Edge List

f0	n0 n2 n1	e0 e1 e2
f1	n1 n2 n3	e3 e4 e0
f2	n0 n1 n3	e4 e5 e1
f3	n2 n0 n3	e5 e3 e2

(c) Face List

Table 4: Tetrahedron as Render Dynamic Mesh

As we may observe, it is possible to deform this data structure to an inconsistent state. Of course it would not be an intention to do so, but it may happen while implementing an algorithm. To find such mistakes, the implementation of the data structure is able to check its consistency with an `is_consistent` method.

Another feature is the possibility to save and load the structure from a human-readable ASCII text file. An example is shown in subsection 3.2.5.

3.2.1 Nodes

A Node is also known as vertex or point. It consists of a position, a list of adjacent edges and a list of adjacent faces. These lists have a variable length. The hash value of a node is defined by its position only.

3.2.2 Edges

An edge consists of a start node and an end node, a face to the left and a face to the right. The hash value of an edge is defined by the given two nodes. Although a start and an end defines a direction for an edge, the hash value is direction independent.

3.2.3 Faces

A face is also known as facet or, under circumstances, triangle. It consists of three nodes: A, B, C and three edges: a, b, c. Faces have an orientation, which tells what is in front of or what is below the face. The hash value is defined by these three nodes. It is orientation independent. Nevertheless checking for equality of two faces also checks the orientation.

3.2.4 Contained Data

Each node, edge and face contains a data field. These fields are used to store algorithm specific information. For example: The `QuickHull` algorithm (see section 5.2) uses the data field of faces to store the set of conflicting points for each face.

3.2.5 File Format

The file format of this mesh data structure is quite easy to understand, if we look at the example shown in listing 1. This file describes how the tetrahedron of table 4 is stored. It starts with a description of the provided mesh. This description can be read from the `description` field of a mesh object. The keyword `nodes` indicates that following lines are nodes. It is equal with `edges` and `faces`.

```
This is a tetrahedron.

nodes:
  [0]=Node([-0.500, -0.500, -0.500])
  [1]=Node([0.500, 0.500, -0.500])
  [2]=Node([-0.500, 0.500, 0.500])
  [3]=Node([0.500, -0.500, 0.500])
edges:
  [0]=Edge(nodes[2], nodes[1])
  [1]=Edge(nodes[1], nodes[0])
  [2]=Edge(nodes[0], nodes[2])
  [3]=Edge(nodes[2], nodes[3])
  [4]=Edge(nodes[3], nodes[1])
  [5]=Edge(nodes[3], nodes[0])
faces:
  [0]=Face(nodes[0], nodes[2], nodes[1])
  [1]=Face(nodes[1], nodes[2], nodes[3])
  [2]=Face(nodes[0], nodes[1], nodes[3])
  [3]=Face(nodes[2], nodes[0], nodes[3])
```

Listing 1: tetrahedron.mesh

4 Thread Architecture

Our thread architecture decouples the computation defined by the algorithm from the visualization. Each thread can be separately analyzed by the profiler. It allows to count each function called by the algorithm thread. This means it can be experimentally determined, if an implementation of an algorithm meets stated running time.

Figure 2 shows the implemented thread architecture. Objects to share data between these threads are shown in rectangles. It has to be ensured that each of these objects works with correct data locking mechanisms.

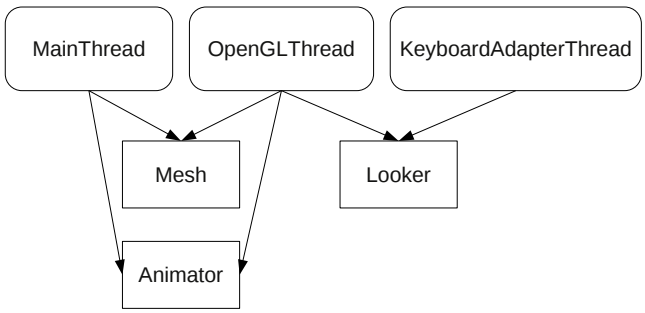


Figure 2: Thread Architecture

The main thread handles the algorithm itself. The OpenGL thread is responsible for the visualization. The keyboard adapter thread processes the user input. This is done in an own thread to have performance independent moving speed when moving around in the 3D environment with keystrokes (see section 2.5.1).

4.1 Class Animator

The Animator class is the glue between the algorithm and the visualization. It contains the following functions:

- set points to wait in algorithms source code
  An algorithm stops its execution if this function is called until continued. Usually the computation is continued after a predefined timeout.
- toggle pause
- next step
- skip animation
- set displayed text
- set displayed mesh

5 Implemented Algorithms

5.1 Basics in Computational Geometry

The Relationship between the number of edges, faces and vertices is explained in Theorem 5.1.

Theorem 5.1. *Let $conv(P)$ be a convex polytope with n vertices. The number of edges n_e of $conv(P)$ is at most $3n-6$. The number of facets n_f is at most $2n-4$.*

Proof. Euler's formula states the following relation:

$$n - n_e + n_f = 2$$

We may observe that every facet has at least 3 edges and every edge has 2 adjacent facets. This gives $2n_e \geq 3n_f$. Plugging this observation into Euler's formula we get:

$$n_e \leq 3n - 6 \qquad\qquad n_f \leq 2n - 4$$

These relations are valid for $n \geq 3$ only. □

The degree of a point p is denoted as $\deg p$. This counts the number of edges where p is a start- or an endpoint. Let P be a set of points in general convex position. As we have seen, $conv(P)$ denotes the convex hull of this set. $\deg_P p$ denotes the degree of a point p in $conv(P)$.

Lemma 5.1. *Let $conv(P_r)$ be a convex polytope with r vertices. The expected degree of $p_r \in P_r$ is bounded by 6 for all $r \geq 4$.*

Proof. Every edge gets counted in the summation of the degrees of each point $p_i \in P_r$ two times: $\sum_{i=1}^{r} \deg_{P_r} p_i \leq 2(3r-6)$ (for $r \geq 3$).

The first idea to calculate the expected value of the degree of points would be simply to take the summation over all degrees and divide them by the number of points. That would be $\frac{2(3n-6)}{n}$. This is not valid because the relation stated by Theorem 5.1 holds for $n \geq 3$ only.

To deal with this problem, we start to calculate the expected value using a tetrahedron. That means, we have 4 points set and calculate the mean over the remaining $r \geq 5$ points. The 4 points of a tetrahedron have a total degree of 12. The expected degree $\mathrm{E}[\deg_{P_r} p_r]$ is bounded as follows:

$$\begin{aligned}
\mathrm{E}[\deg_{P_r} p_r] &= \frac{1}{r-4} \sum_{i=5}^{r} \deg_{P_r} p_i \\
&\leq \frac{1}{r-4} \left(\left(\sum_{i=1}^{r} \deg_{P_r} p_i \right) - 12 \right) \\
&\leq \frac{2(3r-6)-12}{r-4} = 6
\end{aligned}$$

□

5.2 QuickHull

5.2.1 Problem

Let P be a set of n points in $\mathbb{R}^3$.
We want to compute the convex hull of P ($\mathcal{CH}(P)$).

5.2.2 Algorithm

The following algorithm is based on `ConvexHull` [deBerg2008, Chapter 11.2]. An improvement to mention is that this algorithm does not need the complete set of conflicting points for each facet. A point stays in conflict to a facet if the facet has to be deleted to add the point to the convex hull. The set of conflicting points of facet f is denoted as $P_{conflict}(f)$. An example of an conflict graph is shown in figure 3.

`QuickHull`(P)

1. Initialize a tetrahedron with 4 extremal points of P.
2. Initialize $P_{conflict}(f)$ for each facet f.
3. Let `queue` be a queue with all facets that have conflicting points.
4. While `queue` not empty:
   (a) Let f be the first facet in `queue`.
   (b) Get point $p \in P_{conflict}(f)$ with maximal distance to f.
   (c) Remove all from p visible facets (also from `queue`) starting from f and collect all conflicting points.
   (d) Triangulate open edges with point p.
   (e) Update conflict set of generated facets with collected conflicting points.
   (f) Append generated facets, which have conflicting points, to the end of `queue`.

5.2.3 Correctness

Lemma 5.2. *`QuickHull` computes the complete convex hull of P.*

Proof. The computation starts with a convex polytope, a tetrahedron. In step 4c all facets visible from a point get removed. After triangulation in step 4d the polytope stays convex. There is no possibility that this polytope will ever be else than convex.

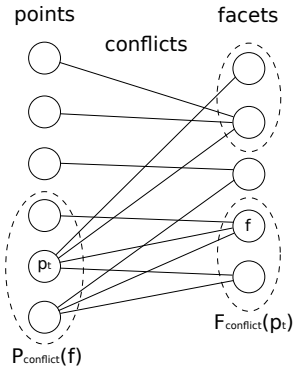


Figure 3: conflict graph

Next part of the proof is to show that the resulting convex hull is complete. This means we have to show that all points outside the polytope get processed until the queue of facets is empty. During initialization in step 2, all points outside the polytope will be in any set of conflicting points at least once. After triangulation in step 4d, the collected conflicting points are either inside or outside the newly generated polytope. If such a point is inside, it does no longer stay in conflict to any facet and will be ignored. In case it is outside, the conflicts will be analyzed in step 4e, which generates conflict sets for the newly generated facets. If all facets with conflicting points are processed, the queue is empty and we are done. □

5.2.4 Expected Time

To answer the question for the expected running time of `QuickHull` we start our investigation by analyzing the number of facets that our algorithm has to process. (see [deBerg2008, Lemma 11.3] for details)

Lemma 5.3. *The expected number of facets created during computation by `QuickHull` is at most* $6n - 20$ *(*$O(n)$*).*

Proof. The computation starts with a tetrahedron. At each iteration r one point p_r will be inserted into the intermediate convex hull $\mathcal{CH}(P_r)$. This will increase the size of the convex hull. By Lemma 5.1 the expected degree of an inserted point does not depend on the size of the convex hull. For that reason, every inserted point p_r has an expected degree bound by 6 and will therefore cause to create expected 6 facets.

The expected number number of facets created by `QuickHull` is 4 (tetrahedron) plus the summation of expected degrees.

$$4 + \sum_{r=5}^{n} \mathrm{E}[\deg_{\mathcal{CH}(P_r)} p_r] \quad \leq \quad 4 + 6(n-4) \quad = \quad 6n - 20$$

□

Lemma 5.4. *`QuickHull` runs in $O(n \log(n))$ expected time.*

Proof. Certainly the initialization takes linear time ($O(n)$). As proofed before, we have to process $O(n)$ facets. The expected running time depends on the cardinality of the set of conflicting points for each facet.

By [deBerg2008, Lemma 11.6] the expected value of $\sum_e \#(P(e))$, where the summation is over all horizon edges that appear at some stage of the algorithm, is $\leq 96n \ln n$. In this notation $\#(...)$ denotes the cardinality of a set. $P(e) = P_{conflict}(f_1) \cup P_{conflict}(f_2)$ is the union of the set of conflicting points to the edge e adjacent faces f_1 and f_2. The existence of a constant value of $\mathrm{E}[\deg_{\mathcal{CH}(P_r)} p_r]$ allows that $\sum_e \#(P(e)) = O(n \log n)$ implies the average size for one set of conflicting points. $\mathrm{E}[P_{conflict}(f)] = O(\log n)$.

These facts combined give an overall expected running time of $O(n \log n)$. □

5.3 SplitHull

5.3.1 Problem

We have an n-point set P in $\mathbb{R}^3$ in general convex position. The points $B \subseteq P$ are called *blue*. The points $R = P \backslash B$ are called *red*. $conv(P)$ denotes the convex hull of P.

Given $conv(P)$ we want to compute $conv(B)$ in $O(n)$ time.

5.3.2 Algorithm

The following algorithm was introduced by [Chazelle2009]. It works in-place. `SplitHull` removes all *red* points from the convex hull of P. As result, the remaining *blue* edges and *blue* points form a convex hull. An edge is called *blue* iff both connected points are *blue*.

`SplitHull`($conv(P)$)

1. If P contains no red points, return $conv(P)$.
2. If there exists a red point r in P for which we have $\deg_P r \leq d_0$ (with a suitable constant d_0), then return `SplitHull`($conv(P \backslash r)$).
3. Take random blue points $b \in B$ until (i) $\deg_P b \leq 6$; and (ii) there exists a blue edge e in $conv(P \backslash b)$ visible from b.
4. Call `SplitHull`($conv(P \backslash b)$) to compute $conv(B \backslash b)$.
5. Using e as a starting edge, insert b into $conv(B \backslash b)$ and return $conv(B)$.

5.3.3 Correctness

Lemma 5.5. *`SplitHull`(conv(B)) computes conv(B).*

Proof. Only *red* points are removed permanently from $conv(P)$. All *blue* points that are removed from $conv(P)$ in step 4 are inserted in step 5. The number of *blue* points stays the same.

The removal of any point of $conv(P)$ reduces the average degrees of surrounding points. This assures that all *red* points will be processed in some stage. As shown in Lemma 5.1, we can always find a point that has degree at most 6. Therefore d_0 should at least be 6. □

5.3.4 Expected Time

Lemma 5.6. *The expected time for `SplitHull`(conv(P)) is $O(n)$.*

Proof. The most important part is that every point will only be handled once.

For step 1 we simply use a counter for the number of *red* points. This will take $O(1)$ time to check for red points and $O(n)$ time to initialize once. We also initialize a list L with *red* points which have $\deg_P r \leq d_0$.

Because the point to remove has a bounded degree ($\leq d_0$), the time for removal in step 2 is constant. When the hull is altered, we update the degrees of directly connected points and the list L. After step 2 our convex hull contains at least $n/5$ pleasant *blue* points (if d_0 is large enough). Therefore we expect 5 iterations to find a pleasant *blue* point b and a corresponding *blue* edge e in step 3. The point b can be removed from the convex hull in constant time because we have a bounded degree ($deg_P b \leq 6$). To insert b in step 5 we need to know a visible edge e. The edge e helps us to determine which facets are visible from b and need to be removed. This takes constant time. □

5.3.5 Implementation Details

This algorithm requires the `find_edge` method of the mesh to have a constant running time. This is needed in step 5. During computation it may happen that edge e gets deleted. After this edge is recreated, it has another reference. This requires a hash map to be able to find edge e in constant time.
Special cases that need to be noticed when implementing `SplitHull` are:

1. A blue edge e is visible only, after the blue point b is removed from convex hull.
2. All faces may disappear.
3. With a given small $d_0(< 6)$ it may be impossible to find a suitable blue point b.

5.4 RandMultiSplit

5.4.1 Problem

Let P be a set of n points in $\mathbb{R}^3$ in general convex position. Each point $p \in P$ is colored random (uniformly and independently) with a color in $\{1, ..., \chi\}$. $c : P \to \{1, ..., \chi\}$ defines a coloring of P. For $i \in \{1, ..., \chi\}$, $C_i = c^{-1}(i)$ is the point set colored i. $F[P]$ denotes the set of facets of $conv(P)$.

Given $conv(P)$ we want to compute $conv(C_i) \forall i$ in $O(n)$ time.

5.4.2 Algorithm

The second algorithm introduced by [Chazelle2009] solves the problem described before.

`RandMultiSplit`($conv(P)$)

1. Pick a random sample $S \subseteq P$ of size n/χ and compute $conv(S)$
2. For each $p \in P$, determine a facet $f_P \in F[S]$ in conflict with p.
3. For each color i:
   (a) Insert all points of C_i into $conv(S)$.
   (b) Extract $conv(C_i)$ from $conv(C_i \cup S)$.

5.4.3 Correctness

Lemma 5.7. *`RandMultiSplit`($conv(P)$) computes $conv(C_i)$.*

Proof. Obviously all points of C_i get inserted at step 3a. The next step will extract $conv(C_i)$ by using `SplitHull`. In Lemma 5.5 we have shown that `SplitHull` works correctly. □

5.4.4 Expected Time

Lemma 5.8. *The expected time for `RandMultiSplit` is $O(n)$.*

Proof. In step 1 we have to initialize a random convex sample $conv(S)$ with n/χ points. By using `SplitHull` this can be done in n/χ time. $conv(S)$ needs to be copied χ times to capture every coloring. Therefore the first step requires $O(n)$ time.

The second step requires another algorithm (`SubsetConflictWalk`). Let's state here that this is possible in $O(n)$ time. A proof will be given in the next section.

Because we have conflicting facets f_p, each point can be inserted in expected constant time. By using `SplitHull` in step 3b, $conv(C_i)$ can be computed in $O(|C_i| + n/\chi)$ time. There are χ colors, so step 3 takes $O(n)$ time. □

5.5 SubsetConflictWalk

5.5.1 Problem

Let P be a set of n points in general convex position in $\mathbb{R}^3$. Let Q be a subset of P ($Q \subset P$).

Given $conv(Q)$ and $conv(P)$ we search for a conflict facet $f_q \in F[Q]$ for each point $q \in \{P \backslash Q\}$. $F[Q]$ denotes the set of facets of $conv(Q)$. This computation is required to run in $O(n)$ time.

5.5.2 Algorithm

The algorithm is used to solve step 2 of `RandMultiSplit`. It is the third algorithm introduced by [Chazelle2009].

$\Gamma_P(p)$ denotes the neighbors of p in $conv(P)$. An example of the neighborhood is shown in figure 4. Big points show that they are in Q and therefore in P. In the algorithm these points are usually denoted as p. Small points show that they are in P only. These points are denoted as q.

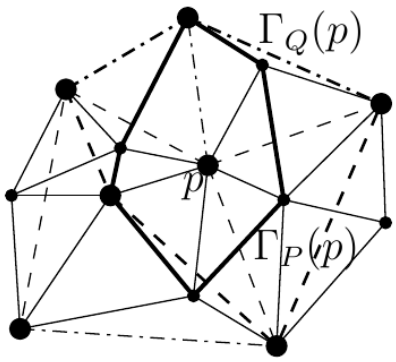


Figure 4: Neighborhood [Chazelle2009]

`SubsetConflictWalk`($conv(Q)$, $conv(P)$)

1. Let `queue` be a queue with the elements in Q.
2. While `queue` $\neq 0$.
   (a) Let p be the next point in `queue`.
   (b) If $p \notin Q$, insert p into $conv(Q)$, using a previously computed conflict facet f_p for p as a starting point.
   (c) For each neighbor $q \in \Gamma_P(p)$, find a conflict facet $\tilde{f}_q$ in $conv(Q \cup p)$.
   (d) Using the $\tilde{f}_q$'s, find conflict facets $f_q \in F[Q]$ for $\Gamma_P(p)$. if $q \in \Gamma_P(p)$ has not been encountered yet, insert into `queue`.

$\tilde{f}_q$ denotes a conflicting facet of q in $conv(Q \cup p)$. A conflicting facet $\tilde{f}_q$ can always be found in the adjacent facets of p. f_q denotes the conflicting facet of q in $conv(Q)$. The difference of $\tilde{f}_q$ and f_q is shown in figure 5.

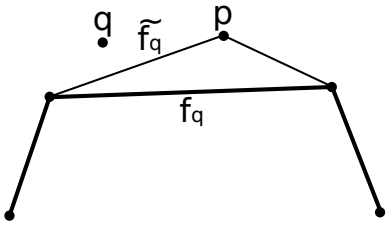


Figure 5: Difference between f_q and $\tilde{f}_q$

5.5.3 Correctness

Lemma 5.9. *`SubsetConflictWalk` computes a conflicting facet $f_q \in F[Q]$ $\forall q \in \{P \backslash Q\}$*

Proof. The `queue` gets initialized with all points of Q.

Step 2 does a breath first search over all points. It retrieves surrounding conflict facets f_q of each neighbor $q \in \Gamma_P(p)$ of p. All neighbors q that have not been encountered yet, are inserted into the `queue`. □

5.5.4 Expected Time

Lemma 5.10. *In $O(\deg_Q p + \deg_P p)$ time we can compute a conflicting facet $f_q \in F[Q]$ for every neighbor $q \in \Gamma_P(p)$ of p.*

Proof. The conflicting facets in Step 2c for $\Gamma_P(p)$ can be computed by merging the cyclically ordered lists $\Gamma_P(p)$ and $\Gamma_Q(p)$. This takes $O(\deg_Q p + \deg_P p)$ time. Because we have the degree for each point p bounded to 6 (Lemma 5.1), we can bound the total running time as follows:

$$T(n) = E\left[\sum_{p \in P} (\deg_Q p + \deg_P p)\right] \leq O(n)$$

□

6 How to extend

Our tool can easily be extended to animate any algorithm operating in 3-dimensional space. The only thing that needs to be done is filling our mesh data structure (see section 3.2). By using the `wait` method of the supplied `animator` object (see section 4.1) it is possible to define a step of the algorithms computation.

The data fields of `Node`, `Edge` and `Face` can be set freely. In case it is required to invoke any other algorithm, we have to assure that the required data fields are available. For example: To invoke `QuickHull` every `Face` needs an object of type `QHFaceData` as data. This class can be subclassed to append any additional data.

A good method for testing new implementations is to create randomized input data. Our tool stores the input mesh before invoking an algorithm. In case the computation causes an error, the input mesh will be available to reproduce the error. If no error occurs, the data will be deleted after the computation is completed.

References

[Chazelle2009] B. Chazelle and W. Mulzer, 2009:
Computing Hereditary Convex Structures, Proceedings of the 25th annual symposium on Computational geometry, 61-70,
`http://www.cs.princeton.edu/~chazelle/pubs/socg09.pdf`

[deBerg2008] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, 2008:
Computational Geometry, Springer, ISBN: 978-3-540-77973-5

[LQ2009:MCA] LinuxQuestions.org, 2009:
Members Choice Awards: Programming Language of the Year,
`http://www.linuxquestions.org/questions/2009-linuxquestions-org-members-choice-awards-91/programming-language-of-the-year-780672/`

[PythonDoc2010] Python Software Foundation, 2010:
Python documentation,
`http://docs.python.org/release/2.6.5/`

[PythonSrc2010] Python Software Foundation, 2010:
Python source code,
`http://svn.python.org/view/python/tags/r265/`

[PyOpenGL2010] *PyOpenGL*,
`http://pyopengl.sourceforge.net/`

[Wiki2010:PM] Wikipedia, 2010: *Polygon mesh*,
`http://en.wikipedia.org/wiki/Polygon_mesh`