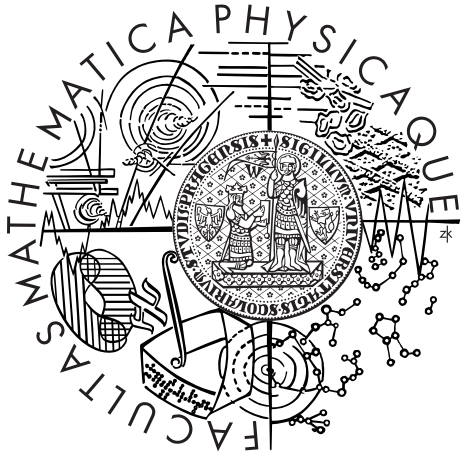
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Peter Hmíra

Generic algorithms for polygonal mesh manipulation

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Jan Kolomazník
Study programme: Informatics
Specialization: Programming

Prague 2013

I would hereby like to thank my supervisor Mgr. Jan Kolomazník. His support and encouragement have sustained me throughout writing this thesis. I would also to thank my parents for the support during the study and to my brother for being a role model for me.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Generic algorithms for polygonal mesh manipulation

Autor: Peter Hmíra

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jan Kolomazník, Kabinet software a výuky informatiky

Abstrakt: Táto bakalárska práca analyzuje algoritmy, ktoré pracujú s objemovými datami, najmä s trojuholníkovou či polygonálnou sieťou. Výsledky analýzy sú premietnuté v návrhu generickej knižnice, ktorá prijíma ľubovolnú implementáciu mesh-u, ktorá spĺňa požiadavky knižnice. Samotná knižnica je napísaná v jazyku C++ využívajúc normu C++11 s pomocou knižnice `boost`. Výber jazyka je oddôvodnený predovšetkým tým, že dôraz sa kladie hlavne na *run-time* rýchlosť a tým, že C++ nám prináša možnosti analyzovať dátove typy už počas prekladu programu. Ďalej je v práci popísaná samotná implementácia knižnice, použitie algoritmov a ich konceptov, zmysel adaptérov - nástrojov, ktoré umožňujú chod algoritmov nad takými implementáciami polygonálnych sietí, ktoré nie sú pre ne vhodne navrhnuté. Technika, akou je táto práca písaná môže byť naďalej uplatňovaná vo vývoji knižnice, teda v pridávaní ďalších algoritmov do knižnice.

Klíčová slova: polygonálny mesh, generické programovanie, algoritmy

Title: Generic algorithms for polygonal mesh manipulation

Author: Peter Hmíra

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jan Kolomazník, Department of Software and Computer Science Education

Abstract: This bachelor thesis analyses algorithms working with the volume data, especially the triangle or polygon mesh. The results of the analysis are applied in the design of the generic library which can be templated with any implementation of mesh satisfying requirements of the library. The library is written in C++ using the norm C++11 with assistance of the `boost` library. The choice of the programming language is supported by the strong emphasis on the run-time performance as well as the capabilities of C++ to analyze a templated code during the compile-time. Later in thesis is described the implemenation of the library, usage of the algorithms and their concepts, the purpose of the adapters - tools that allow to run algorithms over such an implementation of the mesh that is not properly designed for the algorithm. The technique used in the development of this library can be later applied in the library developement, thus adding new algorithms to the library.

Keywords: polygonal mesh, generic programing, algorithms

Contents

Introduction

As written in abstract, there are several libraries for 3D polygonal meshes manipulation. Each of them has already implemented or imported implementation of mesh. But what if we want to use a different mesh implementation in one of them? We have two options: either to build the algorithm from the beginning on our mesh or to forcely import our mesh implementation into the implementation of a algorithm, which can be in cases of incompatibile basic operations used in the algorithm impossible.

In fact, we are able to implement an algorithm without knowing an implementation of a given mesh. Nevertheless, in order to let the algorithm recognize the operation there have to be the specific requirements those have to be satisfied. After observing the algorithms, we will find out that the knowledge of basic operations which manipulate with the certain implementations of the polygonal mesh makes us capable of writing most of the known algorithms over the polygonal mesh.

This thesis focuses on the fact that most of implemented algorithms implement the same concept of polygonal mesh. We analyze the representative set of algorithms that supports this statement and provide a solution that utilizes the observed facts.

Algorithm Decomposition

This thesis explains how can be algorithms decomposed in smaller operations. The purpose of the decomposition is to show that some operations occur frequently and in fact, in some cases, those operations suffices for building numerous algorithms.

Goals of the Thesis

The primary goal is to design a library of generic algorithms that can be used on any implementation of the polygonal mesh. We also have to consider that the implementation may contain several absences from the viewpoint of designing the algorithm. Therefore, for each algorithm we have to provide a simple and clear concept that is required by the algorithm and build a user-friendly checker for the concept. The more simple the concepts are, the easier it is for the user to understand the technique for using the library.

The secondary goal is to design the library in terms *easy-to-expand*. In other words, whenever a new algorithm will be published, one can create the implementation inspired by the implementations of other algorithms in the library.

Generic Programming

In C++, we are able to use the programming technique that allows us to write in terms *to-be-specified-later* by using templates. In comparision with other programming languages the templates are pre-evaluated and the temporary source code is generated during the compile time. This gives us a better performance while the binaries are executed. We are thus able to write an algorithm without knowing the closer specification of an instantiated type. The only information we have to know is whether the type used as parameter contains properties that are demanded in the implementation.

The template metaprogramming technique gives us options to determine during the compile time whether a structure used as parameter contains required properties. Moreover, using this technique, we are able to expose implemented operations in the structure and generate a temporary source code based on the structure which is later merged with the rest of the source code.

In this thesis, it is shown how we can analyze an implementation of the mesh during the compile time. Therefore we are able to determine whether it satisfies the concept of the implementation of an algorithm. Based on other operations that are contained in the structure, we can temporarily build a new operation that is not generally supported by the structure. Despite the risk of inefficiences in the resulting operation, we can use the algorithms that require the generated operation.

Part I

Theoretical Background

1. 3D Object Representation

This chapter contains the definitions of various three-dimensional object representations that are commonly used in applications. A geometric query or a different operation involving the object might be more effectively formulated with one representation than another.

1.1 Overview

The 3D object representations can be divided into two major categories

- **solid** - the object in this representation contains information about inner propreties of the object such as density and elasticity. It is mostly used in software for engineering simulations.
- **surface** - this representation works only with the surface of the object ignoring the properties of the volume itself. One of the advantages of this representation is relatively simple visualisation. The surface description is entirely sufficient for graphical purposes.

1.1.1 Voxel Map

A **voxel** is a word made by joining '**Vo**lumetric' and 'Pi**xel**'. It represents the value on regular grid that forms a voxel map. The simplest representation of a voxel map is a normalized grid made of cube-shaped voxels with 0-1 values where 0 stands for the cube that is completely outside of the object and 1 represents the case of crossing the surface or inclusion by the object.

Each voxel can contain various properties such as color, material physical properties or in case of boundary voxels, a surface description. Some implementations provide information about each corner of voxel separately in order to improve the accuracy. As it can be deduced from the overview, this representation is perfect for the representation of the solid objects.

1.1.2 Implicit Surface

An implicit surface[1] is a surface defined implicitly by a function

$$surface_{f(x,y,z)} = \begin{cases} \text{on surface if } f(x,y,z) = 0 \\ \text{below surface if } f(x,y,z) < 0 \\ \text{above surface if } f(x,y,z) > 0 \end{cases} \tag{1.1}$$

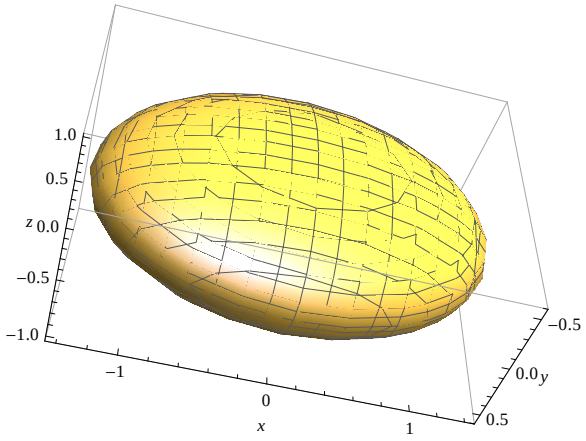


Figure 1.1: Implicit surface given by function $f(x, y, z) = \frac{1}{2}x^2 + 3y^2 + z^2 = 1$

This representation has several advantages such as efficient checking whether a point is either inside or outside and efficient search for an intersection with geometric primitives.

1.1.3 Constructive Solid Geometry

Known also as *CSG tree*[2]. This technique is used especially in the engineering software. It allows the user to construct objects using *Boolean operators*. Objects created by using this method can be used repeatedly in Boolean operators; the resulting object can be decomposed into a tree.

The leaves of the tree are typically the objects of the simple shape. However, the operators can be theoretically applied on any objects. The set of supported primitives is given by a specific software package.

Thanks to the tree structure, CSG objects come with some convenient properties. The nodes that are higher in the tree hierarchy give us an approximation that can be used in various geometric algorithms with no need to look for its descendants.

1.1.4 Point Cloud

A point cloud is a set of points in a three-dimensional coordinate system[2][3]. The set can form the surface of an object or any other three dimensional figure. Every point is indepedent from one another, with no information about its topology. They are often converted to polygon mesh or triangle mesh models.

1.1.5 Polygonal Mesh

A polygonal mesh is a collection of vertices, edges and faces that defines the shape of the surface of the object[2]. Some special implementations do not contain faces. These implementations are defined only by vertices and edges. Moreover, some cases do not consider the topology and the object is a raw set of polygons. Nevertheless, a standard implementation is expected to contain a topology information

about each vertex and face.

Each face is defined by planar polygon, in special cases by *convex polygon* or *triagle*. The polygonal mesh restricted to contain triangular faces only, is called *triangle mesh*. The main purpose of the restriction is the rendering simplification. In general case, face is defined as the set of points that can contain a hole.

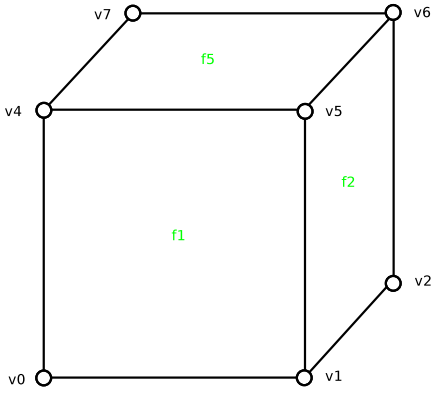
There are several techniques to represent a topology of the mesh. Each of them has its own advantages that are reflected in an efficiency of data storage and querying the surrounding elements. Several mesh representations are described in the next section.

1.2 Polygonal Mesh

The collection can be represented in a variety of ways, the main purpose of the structure is to improve the efficiency of the queries. On the other hand we face the limitations such as memory. In some representations it is impossible to query the adjacent vertices of a vertex without iteration through entire container of vertices or faces.

1.2.1 Face-Vertex Mesh

The simplest representation that contains any information about topology of the polygonal mesh. A mesh is represented by a container of vertices and the container of sets of vertices that form faces[2]. The face is represented by at least 3 vertices or more vertices that have to be coplanar.



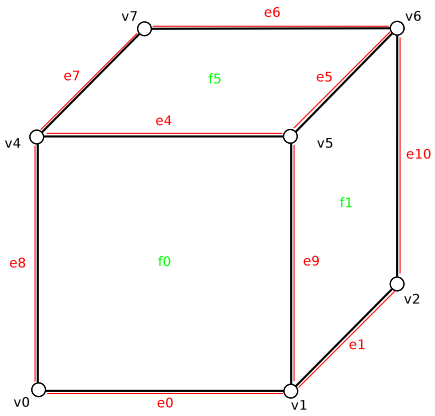
v0 v1 v5 v4	f1
v1 v2 v6 v5	f2
v4 v5 v6 v7	f5

Figure 1.2: face-vertex mesh representation

The set that forms the faces must be ordered. The preceding and following vertex in order must be adjacent to the vertex. If the structure is not extended by face-normal-attribute, the vertices have to be in clockwise/counter-clockwise order from the view of face normal. Whether the order is clockwise or counter-clockwise depends on the implementation of the mesh.

1.2.2 Winged-Edge mesh

The representation described above contains no direct information about adjacency of faces. In order to determine the adjacency of any two faces or vertices in constant complexity, we can build a structure that requires an extra storage. The winged edge structure[4] provides information capable of determining edges that belong to a vertex. It also provides information which allows us to determine the surrounding edges from the given face; in case of a triangle mesh, it returns the triplet of edges. Finally, the structure is named *winged-edge* because of its capabilty to get a face pair (wings) from a given edge.



Face list

f0	e0 e9 e4 e8
f1	e1 e10 e5 e9
f5	e4 e5 e6 e7

Edge list

e0	f0 f4	v0 v1
e1	f1 f4	v1 v2
e4	f0 f5	v4 v5
e5	f1 f5	v5 v6
e6	f2 f5	v6 v7
e7	f3 f5	v7 v8
e8	f0 f3	v4 v0
e9	f1 f0	v5 v1
e10	f2 f1	v6 v2

Figure 1.3: Winged-edge mesh

Several uncertainities occured in the above-mentioned examples.

- Let us take edge e9. In the pair of faces, which of faces f0 and f1 should be considered as the *first* and which as the *second*?
- Which of vertices v1 and v5 should be considered as the *first* and which as the *second* in the pair that forms the edge e9?

One of possible solutions is to split the edge element into two *half-edges*. The structure using, or rather based on this representation, is described in the next paragraph.

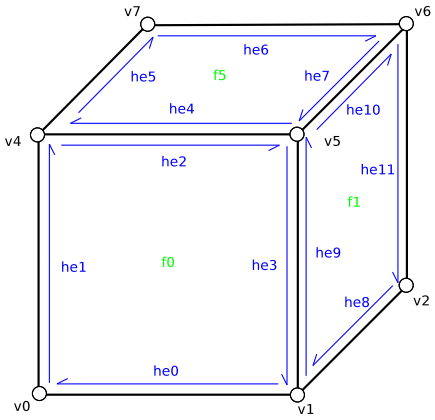
1.2.3 Half-Edge mesh

The half-edge data structure is the structure capable of maintaining the incidence information of vertices, edges and faces[5]. Each edge is decomposed in two half-edges with opposite orientations.

The structure contains the set of following information

- a vertex that is the half-edge pointing to

- a face that the half-edge belongs to
- the next half-edge in order that forms a face that half-edge belongs to
- (optional) the previous half-edge in the same order
- opposite half-edge



Half-edge list

he0	f0	v0	he1
he1	f0	v4	he2
he2	f0	v5	he3
he3	f0	v1	he0
he8	f1	v1	he9
he9	f1	v5	he10
he10	f1	v6	he11
he11	f1	v2	he8
he4	f5	v4	he5
he5	f5	v7	he6
he6	f5	v6	he7
he7	f5	v5	he4

Figure 1.4: Half-edge mesh representation

2. Operations and Algorithms

Our aim in this chapter is to describe algorithms over 3D data structures. In each description of an algorithm there is a short paragraph that summarizes required operations for running the algorithm. It is later used in hierarchical implementation of the algorithms.

The article[6] proves that the mesh representation of a hole-free solid object can be treated as a planar graph, and a polygon mesh as general unoriented graph. Therefore, we can generalize the editing functions over polygon meshes as the operations over the graphs.

2.1 Euler Operators

The *Euler operators* are the set of operators which create a polygon meshes[7]. One of the advantages of these operators is that they are invertible. In the following description paragraphs, each operator is followed by its inverted operator.

2.1.1 Make Vertex

The operation creates a vertex with no topological dependencies. It can be used either on empty mesh or mesh with already created topology.

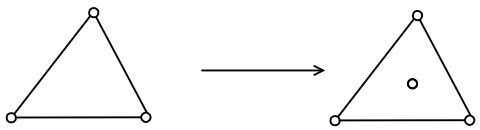


Figure 2.1: Euler operator makeV

2.1.2 Kill Vertex

The operation removes a vertex that is required not to be contained in faces or edges.

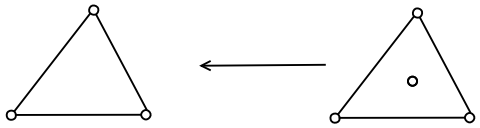


Figure 2.2: Euler operator killV

2.1.3 Make Vertex and Edge

The operation *Make Vertex and Edge* creates a vertex and connects it with an another vertex that is already contained in the mesh.

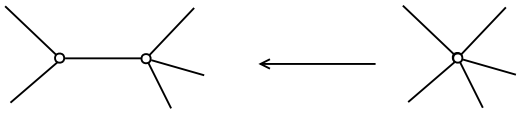


Figure 2.3: Euler operator makeEV

2.1.4 Kill Vertex and Edge

It removes a vertex and disposes the edge that is connected to the vertex.

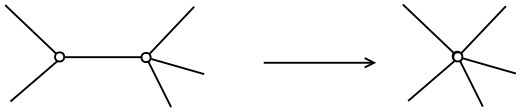


Figure 2.4: Euler operator killEV

2.1.5 Make Edge and Face

This operation creates an edge and forms a new face by splitting another one, or by forming a new face.

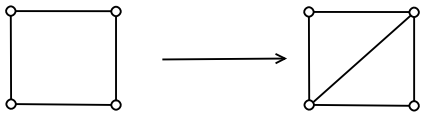


Figure 2.5: Euler operator makeEF

2.1.6 Kill Edge and Face

It removes the edge and the face that is connected to the edge.

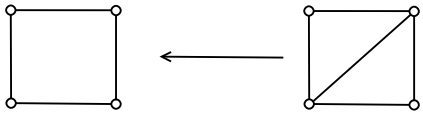


Figure 2.6: Euler operator killEF

2.1.7 Make Face and Kill Ring

It creates a face by disposing the ring because of the addition of a new edge.

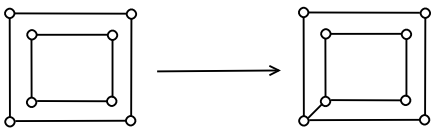


Figure 2.7: Euler operator makeFkillR

2.1.8 Kill Face and Make Ring

The operation creates a ring by disposing of a face.

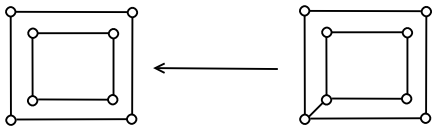


Figure 2.8: Euler operator killFmakeR

2.2 Editing Functions

Some functions are designed to change size or shape of the object. Those functions are called *editing function*. During the editing of the object, the user defines required values and the function deforms the object properly. In case of the editing function *scaling* for instance, a user defines the scaling ratio.

2.2.1 Truncate

Truncate is the operation that affects a topology of a mesh. From a given vertex, it creates a new face with surroundings of the original vertex. This is a common operation of mesh editing used e.g. in 3D editors.

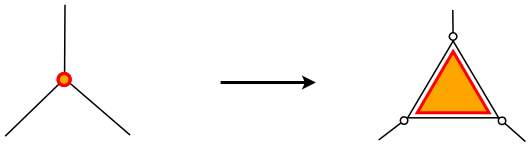


Figure 2.9: The red-marked vertex is a selected vertex to be truncated. The face filled by the orange color with the red borders is the resulting face created by the truncation.

2.2.2 Bevel

This operation is almost identical with the *truncation*. The only difference is the argument of the operation. As the truncate operation creates a new face based on the given vertex, the bevel operation creates the face from a given edge.

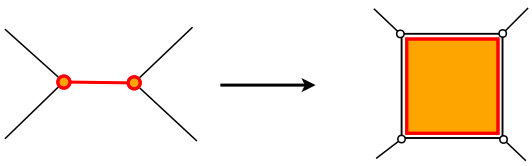


Figure 2.10: The red-marked edge is a selected edge to be beveled. As in the previous case, the resulting face is filled with the orange color.

2.2.3 Extrude

In the previous operations the argument is a vertex and then an edge. Intuitively, one can assume that there is an operation that demands a face as the argument. Operation *extrude* “pulls" the face out the object creating new faces connecting the extruded face with the resulting object.

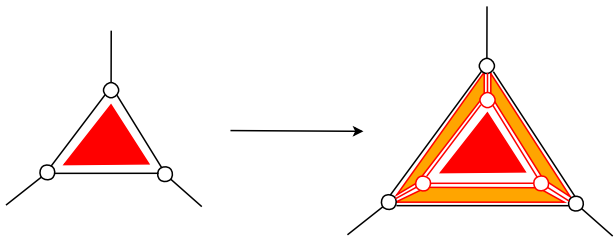


Figure 2.11: The red-filled face is a selected face to be extruded. The operation creates a set of faces (marked with orange) with the topology as shown on the picture.

2.3 Converting between Representations

This section introduces some algorithms that convert one representation to another. Each representation has its own capability.

2.3.1 Delaunay Triangulation

Let P be a set of points in the d-dimensional Euclidean space. Delanuay triangulation[8] is a triangulation such that no point $p \in P$ is inside the circum-hypersphere of any simplex in $DT(P)$. For better imagination, in case of 2-dimensional space, no point structure is inside the circumcircle of any triangle of the resulting structure.

2.3.2 Marching Cubes

Marching cubes algorithm converts a grid representation to a mesh [9]. It iterates through all grid elements and builds a new polygonial mesh. For each element (that can be considered as *cube*, cuboid or even parallelepiped) it determines whether the corners are inside or outside object. The algorithm considers only those elements which contain both categories of points; rest of them are ignored. As each element has 8 corners, each of them can be determined either as inside or outside object. In result, the element can possibly have one of $2^8 = 256$ configurations. However, some configurations can fit to another one after rotation, reflective simmetry, or sign changed case.

Finaly, all of them can be reduced to 15 unique cases. The main idea of this algorithm is to place a set of faces for each cube in order to create a polygonal mesh with a corresponding shape. In other words, each configuration refers to the configuration of new faces to be added to a mesh. However, we know that two adjoining elements share 4 corners, so the resulting faces are certainly continuous. If the corners contain any other information, except the inside/out information,

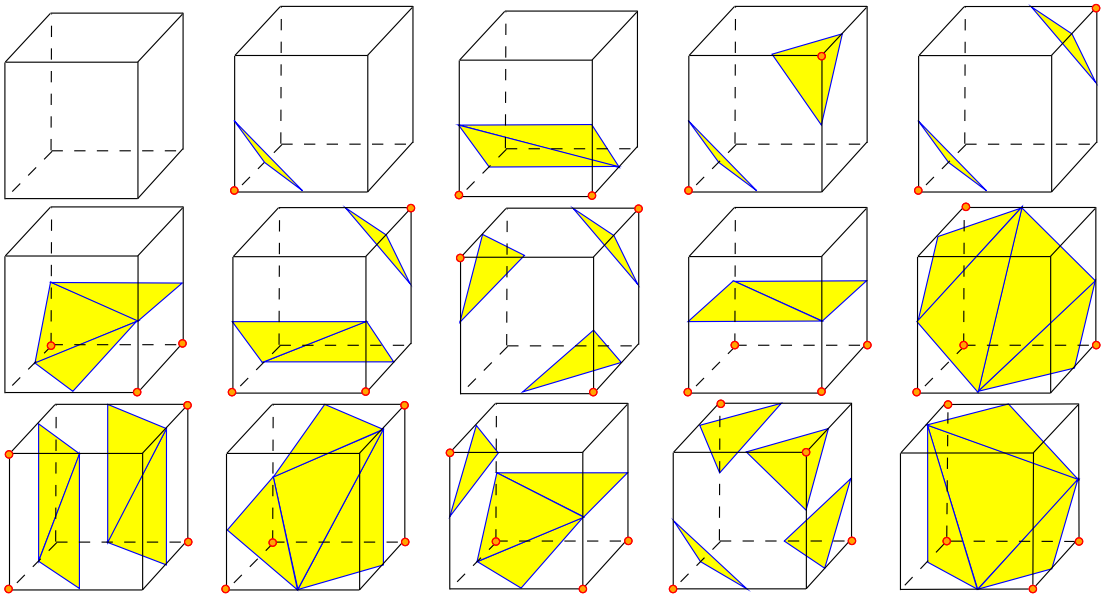


Figure 2.12: The 15 cases of marching cubes algorithm. The corners that are evaluated as inside of the object are labeled with red. The yellow triangles represent faces to be added to a mesh.

the position of the faces in the mesh can be refined. Usually, it contains the information about a color or the distance from point to surface. In that case, placing or any other processing of the face is based on linear interpolation of corner values.

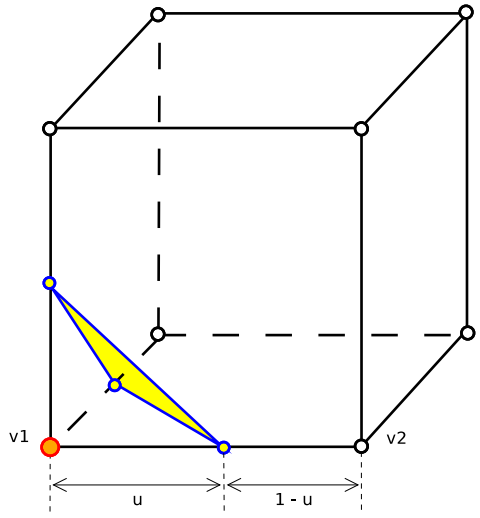


Figure 2.13: Interpolated position of face

Let $v1$ and $v2$ be the values that represent the distances from the surface to the corners. Thus, we can calculate the value of coefficient u from the following equation.

$$u = \frac{v1}{v1 - v2} \tag{2.1}$$

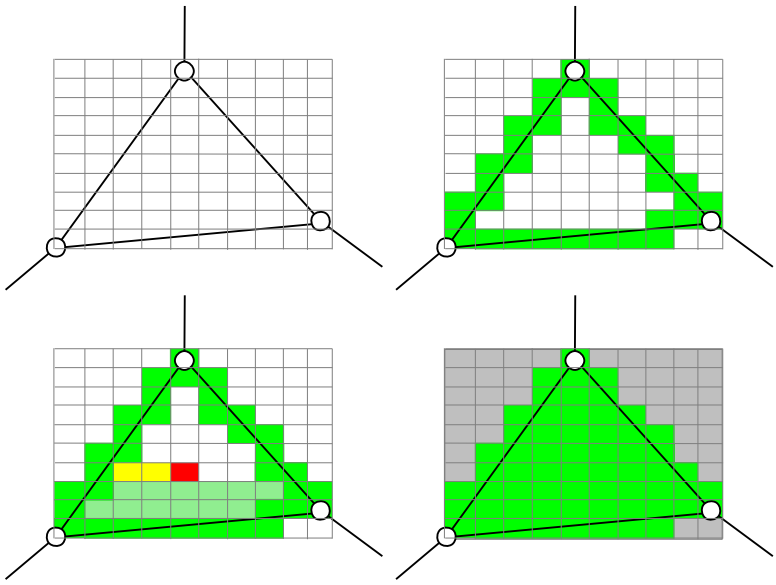


Figure 2.14: Voxelization step-by-step

2.3.3 Voxelization

Voxelization is the algorithm which is in 2D known as *rasterization*. It that constructs an initially empty 3D grid and fills the grid elements that indicate whether the element is inside the object[10]. There are several techniques how to represent a boundary element. The simpliest one is to set the boundary element as completely filled, thus we have a voxel map of values 0 and 1. The other ones store additional information in voxels that form an alias-free voxelized object[11].

This algorithm can be divided into two phases: The first phase voxelizes the faces of triangle mesh and the second one fills the created object. Admittedly, before filling the object, the algorithm has to check whether the mesh forms an enclosed surface. If it does not form an enclosed surface, the second phase of algorithm is omitted.

The first phase voxelizes faces one by one. Each face is processed similarly to the triangle rasterization in 2D. First, the algorithm voxelizes boundary edges and then runs the floodfill over the face. The filling technique of the face is processed by the line-filling algorithm in the following steps.

- On the pre-computed minimum boundary rectangle, start on the bottom and repeat for each line
- In the line, start from an arbitrary side and process the elements consequently
- After first crossing a rasterized edge, start filling the elements (entry the face)
- After second crossing a rasterized edge, stop filling (leave the face).

In the second phase, the algorithm fills the elements inside the object. The only problem is to determine whether the given voxel is inside or outside the object.

As described above, the assumption of starting with a line outside the enclosed area gives us the right method. The bounding box will be needed in this case as well. If the object cannot be wrapped the inside/outside property of element can not be determined.

Part II

Analysis

3. Libraries

We did not yet mention the implemented data structures that fulfill the described rules. This chapter introduces several libraries that are comonnly used. The following paragraphs show the corelation with the structures shown in the chapter 2.

3.1 OpenMesh

OpenMesh[12][13] is an open-source generic data structure for representing and manipulating polygonal meshes. It has been developed at the Computer Graphics Group, RWTH Aachen.

Restricting to the meshes introduced in the section 1.2, OpenMesh is considered as a half-edge structure. It is formed by the kernel that set proper attributes such as triangular restriction or allowing to remove vertices from mesh.
To fully specify a mesh, several parameters can be given:

Face type: Specifies whether to use a general polygonal mesh or a triangle mesh.

Kernel: Stores the element of the mesh internally. User chooses from the available kernels according to expected usage. For example, the decimation algorithm requires efficient deletion/insertion, thus the proper kernel for this case is the kernel based on linked list.

Traits: Traits is the class that enhances the mesh functionality such as vertices removal or adding various attributes to elements.

3.2 Trimesh

Trimesh is the library designed to read, write, and manipulate with the triangle meshes[14].

Compared to the *OpenMesh* library, Trimesh library emphasizes the efficiency and easy of use rather than the sophisticated design. The representation of the mesh is a modified face-vertex structure(see the description in the section 1.2.1). The modification consist in the addition of 3 connectivity structures:

- `neighbors` - for each vertex, all adjacent vertices
- `adjacentfaces` - for each vertex, all adjacent faces
- `across_edge` - for each face, the three faces attached to its edge. Since the faces are restricted to triangles, a given face is allowed to have only the triplet of adjacent faces.

4. Computational Complexity

This chapter describes the impact on the computational complexity by choice of the mesh representation. We will also define the limits for those we consider the operation as *effective* or *ineffective*. Later in the thesis, if the operation is marked as effective it must fulfill the defined criteria. If the mesh representation does not contain an operation that satisfies the criteria for complexity the representation does not natively support the operation.

4.1 Efficient vs. Inefficient

The efficiency is a relative expression hence we need to define what is effective. If the operation does query on the local topology only then the scanning the whole structure is considered in this case as ineffective. The acceptable complexity is the complexity of scanning the immediate surroundings.

Example:
Getting all adjacent verticies of a vertex in the face-vertex representation. (see the section 4.2)

Analysis:
Although the structure does not possess information about vertex adjacency we can acquire the demanded information by scanning the topology of the mesh. As we do not have any auxiliary structure the information can be acquired only by scanning the whole structure. Thus the only way how to solve this problem is for each face to check whether the vertex is contained in the face. Then the previous and the next vertex from the view of the demanded vertex are the ones that form the edges in the mesh.

Conclusion:
The computation complexity of the operation can be computed intuitively: For *each face* we need to check whether the demanded vertex is contained in the face so we need to check *each vertex in the face* and if the vertex is contained we are able to get an adjacent vertex in constant time.

Thus the computational complexity of the operation is $\mathcal{O}(F_{mesh} \times V_{face})$ in total. However getting the adjacent vertices is the operation querying the surroundings only the complexity of the operation on the face-vertex structure is above the effective limit.

The limits for the operation are defined in the following paragraph.

4.2 Limits

For each operation mentioned in the chapter 2 the specific limit of effectiveness has to be defined. Before we specify the limits, we classify the operations in two

categories:

- *Global operation* - the operation that can not be done without all information about entire structure
- *Local operation* - the operation that involves only the specific part of the mesh with no demans on the rest of elements in the structure

In the following analysis are involved operations that adds/removes an element(vertex, edge, face), Euler operators (see the section 2.1), advanced editing functions (see the section 2.2), the getters on the surroundings of an element and the getters on the entire structure(e.g. getting all vertices).

Add/remove primitives

Name of the operation	**Allowed complexity**	**local/global**
Add vertex	$\mathcal{O}(1)$	*local*
Remove vertex	$\mathcal{O}(V_{adjacentV})$	*local*
Create edge from vertices	$\mathcal{O}(V_{adjacentV})$	*local*
Remove edge	$\mathcal{O}(1)$	*local*
Create face from vertices	$\mathcal{O}(V_{adjacentV})$	*local*
Remove face	$\mathcal{O}(V_{adjacentV})$	*local*

The table above are shown basic operations over mesh structure. The allowed computational complexity is based on the number of adjacent vertices of the vertex involved in the operation. In case of the operation that involves more vertices the number $V_{adjacentV}$ represents the sum of all adjacent verticies of all verticies involved in the operation. E.g. in case of the operation *Create face from vertices* if the operation creates a face from the triplet of vertices the computational complexity is $3 \times V_{adjacentV}$.

Euler operators

Name of the operation	**Allowed complexity**	**local/global**
MakeV	$\mathcal{O}(1)$	*local*
KillV	$\mathcal{O}(1)$	*local*
MakeEV	$\mathcal{O}(1)$	*local*
KillEV	$\mathcal{O}(1)$	*local*
MakeEF	$\mathcal{O}(V_{adjacentV})$	*local*
KillEF	$\mathcal{O}(V_{adjacentV})$	*local*
MakeFkillR	$\mathcal{O}(V_{adjacentV})$	*local*
KillFmakeR	$\mathcal{O}(V_{adjacentV})$	*local*

Figure 4.1: The table shows the allowed complexity of the Euler operators on any mesh structure.

Editing functions

Name of the operation	Allowed complexity	local/global
Truncate	$\mathcal{O}(V_{adjacentV})$	*local*
Bevel	$\mathcal{O}(V_{adjacentV})$	*local*
Extrude	$\mathcal{O}(V_{adjacentV})$	*local*

Figure 4.2: Despite to the fact the operations listed in the table above can be created using the Euler operations a mesh may have own, faster and optimized implementation of the operation.

Getters (surroundings)

Name of the operation	Allowed complexity	local/global
Get adjacent vertices of the vertex	$\mathcal{O}(V_{adjacentV})$	*local*
Get adjacent edges of the vertex	$\mathcal{O}(E_{adjacentV})$	*local*
Get adjacent faces of the vertex	$\mathcal{O}(F_{adjacentV})$	*local*
Get vertices of the edge	$\mathcal{O}(V_{consistE})$	*local*
Get faces that the edge splits	$\mathcal{O}(F_{adjacentE})$	*local*
Get vertices of the face	$\mathcal{O}(V_{consistF})$	*local*
Get edges of the face	$\mathcal{O}(E_{consistF})$	*local*
Get adjacent faces of the face	$\mathcal{O}(F_{adjacentF})$	*local*

Figure 4.3: The operations listed above does not affect the mesh; the readable information about the structure is acquired. The allowed complexity does not permit scanning the entire structure. The operations cover operations that determines whether two elements are adjacent. E.g. the operation *dethtermine whether two vertices are adjacent* is covered by the operation *Get adjacent vertices of the vertex*.

Getters (entire structure)

Name of the operation	Allowed complexity	local/global
Get all vertices	$\mathcal{O}(V_{all})$	*global*
Get all edges	$\mathcal{O}(E_{all})$	*global*
Get all faces	$\mathcal{O}(F_{all})$	*global*

Figure 4.4: The operation that gets all elements in the structure assumes that all elements are contained in specific container that can be scanned in a linear complexity.

5. Designing the Library

There are a lot of libraries that offer the ability to represent the volume data and the basic manipulation with it. There are a lot of published algorithms that can be implemented as well.

Our goal is to create a generic set of algorithms that can be used over any implementation of mesh that satisfies a required concept. Before the algorithm is finally implemented, we must completely describe the concept of the algorithm and the behavior of the algorithm on a closely unspecified mesh. The point of this goal is *to think generally* regardless of the specification of a potentionally used mesh.

5.1 Observation

Observing the algorithms step-by-step, we can see that single steps of the algorithm are the variations of adding, removing and modifying the elements of the mesh. In addition, the algorithms also uses a capabilities of querying in mesh such as getting all adjancent vertices of given vertex or getting all vertices in a container or any iterable structure.

Such capabilities of the mesh are critical for implementing an algorithm. The question then arises, “Why do we implement the algorithms for meshes if the usage of required operations might suffice?" The answer can be relatively simple: We are generally not able to determine how is a given structure controlled without being provided the an additional information.

Thus there is an option to build a correctly working algorithm over any structure that fullfills the criteria that has been presented by the algorithm. Nowadays in C++, we are able to build a template-based interface that modifies a given structure according to algorithm requirements during a compile time.

Before using this set of algorithms, the user only defines an interface for chosen structure and passes it as a parameter.

5.2 Concepts and Algorithms

The concept is specific according to the requirements of an algorithm. Let us provide an example of an algorithm and then build a structure concept that requires the algorithm in order to work correctly.

Example:
The algorithm that flips normal on each face of a given mesh.

Solution:

Before analyzing the algorithm, we can see that input is a mesh (not closely specified) that returns modified mesh. The idea of the algorithm is simple; pick all faces and flip the normal of each one. Obviously, the mesh is required to have faces. This is not an unnecessary note, because there are several implementations of mesh without faces; e.g. vertex-edge mesh. Finally, let us summarize what *attributes* are required by the algorithm.

- faces of a mesh

Now, when we have the required attributes, we need to specify which *operations* are required.

- flipping the normal of a face
- capabilty of applying the operation on each face

Question:
What if the structure does not have a method that flips the normal of a face?

After all, the algorithm can be build without providing a method that flips a normal; it can flip the normal of a face by itself. There are two possible approaches to solving this question. The first: create the next algorithm with different requirements separately. The second: build an *adapter* that has the same functionality as a required method that flips normals and use it as method in the algorithm described above instead a required method.

Both cases require additional points of the concept instead the ommited one. In order to reverse the normal of a face manually, the following is required:

- the normal of a face
- getting the face normal
- setting the face normal

It is up to us how do we design the algorithms and the level of the implementation. The more sophisticated the adapters are, the bigger the number of cases of mesh implementation can be covered. If the compiler does not find the required methods and it is able to build an adapter, then it does not bother the user and it builds a modified method. Naturally, adapter may have own requirements therefore we can assume that some point of concepts can be replaced with different ones.

6. Algorithms Decomposition

This chapter describes the decomposition of the algorithms (see the chapter 2) so that basic operations over polygon mesh can be summarized. The word "*basic*" is relative according to the chosen level. The level chosen in this thesis is such that leads to the least count of operations in total.

If the mesh does not fulfill the concept of the algorithm, *adapters* can be used instead of the absent operators. An adapter behaves like an operator but in fact it is the algorithm that makes a desired step as a result. In the other words it is the operator implemented in terms of brute-force. In some cases, lack of an operator can not be replaced by an adapter.

The algorithms described in this chapter are only the small part of all algorithms. The emphasized algorithms are primarily the ones that are commonly used ones; the algorithms that change the mesh topology and the conversion algorithm between voxel and polygon mesh representation.

6.1 Parallel Processing

Some algorithms have a passage that is repeated for every *vertex*, or every *face*, etc. According the design of an algorithm, it can be determined whether the repetition can be processed simultaneously. If the other processed elements are affected in the loop body, the loop can not be processed parallelly.

If an algorithm have such a passage, the developer has an opportunity to accelerate the run-time of the algorithm implementation by using multiple computational threads. Since the most of processors have more than one processor core nowadays, the parallel processing of the algorithm block might significantly speed up the performing time.

If a loop has an option to be processed parallelly, in pseudocode, the loop is labelled with "**parallel**". In the implementation, if the algorithm is not implemented parallelly, the algorithm remains correct.

6.2 Converting between Representations

6.2.1 Marching Cubes

The algorithm has the specific requirements for the mesh and also for the voxel map (See algorithm description 2.3.2). However the voxel map can be in various forms some special cases of marching cubes are explained.

Algorithm 1 Marching cubes

```
 1: function MARCHING CUBES(voxelMap)
 2:     mesh := empty mesh
 3:     parallel for each voxel in voxelMap
 4:         cubeType := Determine Cube Type(voxel)
 5:         faces := Create Faces(cubeType)
 6:         for each face in faces do
 7:             Add Face To Mesh(mesh, face)
 8:         end for
 9:     end parallel for
10:     return mesh
11: end function
12:
13: function DETERMINE CUBE TYPE(voxel)
14:     configuration := initial configuration
15:                                  ▷ configuration contains all the information
16:                                        ▷ about each corner of the cube
17:                     ▷ initial configuration assumes that all corners are outside
18:     corners := Get Voxel Corners(voxel)
19:     for each corner in corners do
20:         if corner is inside the surface then
21:             Set in configuration the corner as inside
22:         end if
23:     end for
24:     return configuration
25: end function
```

From the pseudocode 1 we can conclude that the structures used as a parameters are required to support operations used in the algorithm. Thus the concept of the algorithm is designed as follows:

- *mesh* is required to support *face* structure *(line 6)*
- *mesh* is required to support adding the *faces (line 7)*
- *voxelMap* is required to support *voxel* structure *(line 3)*
- *voxelMap* is required to support getting all the *voxels (line 3)*
- *voxelMap* is required to support getting all the *corners* of each *voxel (line 18)*
- *voxelMap* and *mesh* are required to have a *cubeType* convertible to a set of *faces (line 5)*

6.2.2 Voxelization

Compared to the *Marching cubes* algorithm, the requirements for the voxel map structure are more comprehensive. As mentioned in the description (See the

section 2.3.3), the algorithm consequently voxelizes the edges, the faces and finally fills the object.

Algorithm 2 Voxelization

```
1: function VOXELIZE MESH(mesh)
2:     voxelMap := map of empty voxels
3:     parallel for each face in mesh
4:         Voxelize face(face, voxelMap)
5:     end parallel for
6:     Run Floodfill(voxelmap)
7:     return voxelmap
8: end function
9:
10: function VOXELIZE FACE(face, voxelMap)
11:     for each edge in face do
12:         Voxelize edge(edge, voxelMap)
13:     end for
14:     Run Face Floodfill(face, voxelmap)
15:     update voxelMap
16: end function
17:
18: function VOXELIZE EDGE(edge, voxelMap)
19:     v0 = First Vertex(edge)
20:     v1 = Second Vertex(edge)
21:     for each voxel between v0 and v1 do
22:         Set As Filled(voxel)
23:     end for
24:     update voxelMap
25: end function
```

7. Adapters

If the polygon mesh does not satisfy the designed concept of the algorithm then the compiler raises an error. Nevertheless, in some cases, there is an option to create an additional function that performs according the requirements for the absent operation. In C++ standard library, adapters are used to build function objects out of ordinary functions or class methods[15].

Inspired by the adapters from the C++ standard library, we create a set of adapters for our generic algorithms library. This chapter discribes the purpose of adapters in the implementation of generic algorithms supported by several examples. It is primarily used in case of the absence of any required operation used in algorithm.

7.1 Observation

Same as algorithms, adapters have specific requirements to be created. From this point of view, we can state that an adapter has a capability to substitute a point of the concept with the other ones that are required for the adapter.

That statement is supported in following paragraphs.

7.2 Get-Adjacent-Vertex-of-Vertex Adapter

If the polygonal mesh is not in a representation that contains direct information about adjacency of two vertices then the mesh can not effectively get the adjacent vertices from a given vertex. Obviously, no algorithm that contains query for adjacent vertices of a vertex can be working over such a structure.

However, using the brute-force, we can determine which ones from vertices are in adjacency with the queried vertex. In the other words, demanded vertices can be acquired after checking all faces that contain the queried vertex.

7.2.1 Concept of the Adapter

Assuming the structure is non-edge-based, we add an assumption that the structure is face-based so it supports the face structure that provides information about contained vertices in the correct order. The points of the concept are as follows:

- mesh is required to support getting all faces of the mesh
- mesh is required to support getting all vertices of a face in the mesh

Finally, we can conclude that the point "*mesh is required to support getting all adjacent vertices of a vertex in the mesh*" can be substituted by these two points in a concept of any algorithm. On the other hand, the complexity of the operation

raises from $\mathcal{O}(V_{vertex})$ to $\mathcal{O}(F_{mesh})$ where the F_{mesh} is the number of the faces in total and V_{vertex} is the expected count of the adjacent vertices.

Part III

Implementation

8. Hmira Library

The library is written in C++ using the standard *C++11*[16]. However the features for metaprogramming are not yet as advanced as the ones from *Boost*[17] we use Boost library, especially Boost Metaprogramming Library[18] and Boost Preprocessor Library[19] for metaprogramming purposes.

8.1 Designing the Concept

The important part of the generic library is the concept - the set of requirements for a template used as parameter. After the concept is satisfied, a code can be built.

Concept consists of axioms and constraints[20]. Constraints are statically evaluable predicates of the properties. Axioms are the requirements on the types that can not be statically evaluated.

8.2 C++11

The standard *C++11* comes with new features such as lambda functions, static assertions and other features that help us to pre-evaluate available constants during the compile-time.

C++11 contains the header `<type_traits>` that defines a series of classes to obtain information during the compile-time.

8.3 Boost

Boost has a metaprogramming framework `boost::mpl`[18]. During the compile-time, it is capable of evaluation logical or arithmetical expressions. Combined with `std::enable_if` or static assertion it can determine which function will be compiled and which not.

8.4 Threading Building Blocks

The library *Threading Building Blocks*[21][22] is multiplatform C++ template library for task parallelism made by Intel®. It uses C++ templates to eliminate the need to create and manage threads. The Hmira library uses the Threading Building Blocks in algorithms that have a section which can be done parallely; in the chapter 6, the section which can be done paralelly is labeled with **parallel**. E.g.: the **parallel for** does not necessarily mean that the block must be a parallel loop; it means that the block of code provides an option being done parallelly.

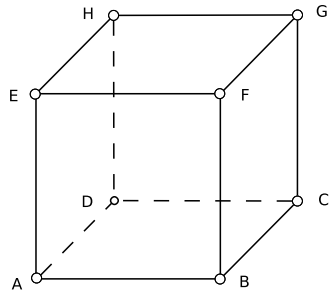
9. Traits

This chapter describes the methods of the required operations recognition by the library.

Our problem is that the class used as a template has unknown parameters and methods. Obviously, the library can not classify the purpose of the members of the class used as template by itself. The solution is to create a new class called *traits*.

Traits aggregate all useful types and methods in the way defined by the library. It is used as a parameter in the templated algorithm and when a method from the traits is called it returns the result of the method from the mesh implementation or anything that has a user put into the traits.

Example:
Implement an algorithm that generates a cube with the centroid in the origin of the standard coordinate system with edges aligned to x,y and z axes and the length of the edge is 2. Thus the coordinates of the points are:



$A = [-1, -1, -1]$
$B = [1, -1, -1]$
$C = [1, -1, 1]$
$D = [-1, -1, 1]$
$E = [-1, 1, -1]$
$F = [1, 1, -1]$
$G = [1, 1, 1]$
$H = [-1, 1, 1]$

The mesh can be assumed as polygonal.

Solution:
The code will appear as follows:

```
/**
* \param                  m the reference to the mesh
* \returns whether the operation ended successfully
*
* the mesh is assumed as empty. If the mesh is not empty
* it just adds a cube
*/
template <typename TMesh, typename TMesh_traits>
bool generate_cube(TMesh& m)
{
        typedef typename TMesh_traits::Point Point; //coordinates
        typedef typename TMesh_traits::Vertex Vertex; //vertex
        typedef typename TMesh_traits::Face Face; //face type

        auto a = Vertex(Point(-1,-1,-1));
```

```
        auto b = Vertex(Point(1,-1,-1));
        auto c = Vertex(Point(1,-1,1));
        auto d = Vertex(Point(-1,-1,1));
        auto e = Vertex(Point(-1,1,-1));
        auto f = Vertex(Point(1,1,-1));
        auto g = Vertex(Point(1,1,1));
        auto h = Vertex(Point(-1,1,1));

        TMesh_traits::add_vertex(m, a);
        TMesh_traits::add_vertex(m, b);
        TMesh_traits::add_vertex(m, c);
        TMesh_traits::add_vertex(m, d);
        TMesh_traits::add_vertex(m, e);
        TMesh_traits::add_vertex(m, f);
        TMesh_traits::add_vertex(m, g);
        TMesh_traits::add_vertex(m, h);

        TMesh_traits::create_face(m, a, b, c, d);
        TMesh_traits::create_face(m, b, c, g, f);
        TMesh_traits::create_face(m, c, g, h, d);
        TMesh_traits::create_face(m, a, d, h, e);
        TMesh_traits::create_face(m, e, f, b, a);
        TMesh_traits::create_face(m, h, g, f, e);

        return true; // the cube has been generated succesfully
}
```

Without creating an instance, in the class `TMesh_Traits` are obtained all methods and types required for the algorithm. In fact, a type can be named differently but in the traits class, it must be named following the rules specified by the interface.

Assuming we have class named `my_mesh`, the traits class will appear:

```
class my_mesh_traits
{
public: //the typenames must be visible for the algorithm
        /* TYPES */
        typedef typename Coord_type Point;
        typedef typename My_Vertex Vertex;
        typedef typename My_Face Face;
        typedef typename mesh TMesh;

        /* METHODS */
        inline static Face
        create_face(
                TMesh& m,
                Vertex a,
                Vertex b,
                Vertex c,
                Vertex d)       {/*the implementation*/}

        inline static void
        add_vertex(
                TMesh& m,
                Vertex a)        {/*the implementation*/}
};
```

10. Concepts

The example in the chapter 9 shows that the implemented algorithm will not work if the class that is used as traits misses any of the methods or parameters used in the algorithm. In order to be certain, the error is raised during the compile time.

The set of all parameters and method required by the algorithm is called *concept*. The concept specifies what must be obtained in the traits to let the algorithm work correctly. This chapter explains the purposes of specifying the concept of the algorithm and the possible methods to let the compiler check the concept.

Summarized from the code of the example in the chapter 9, we can see that the required paramters and methods are as follows:

```
        //parameters
        typename Traits::Point
        typename Traits::Vertex
        typename Traits::Face

        //methods
        Traits::Face create_face(
                Traits::Mesh& m,
                Traits::Vertex a,
                Traits::Vertex b,
                Traits::Vertex c,
                Traits::Vertex d);

        bool add_vertex(
                Traits::Mesh& m,
                Traits::Vertex a);
```

In addition, the type `Vertex` must be constructible from the type `Point`. and the type `Point` must be constructible from the triplet of `float`. From this point there are two approaches to create a class that fullfills the requirement for the constructor assuming the class does not support it by default. The first, to create an inherited class with modified constructor. The second, to build a templated wrapper that uses a class as a parameter.

```
        //building a vertex type
        //using inheriting class
        class Vertex1 : public MyVertex
        {
        public:
                Vertex1( MyPoint& p )
                {
                        this->coordinates = p;
                }
        };
        //building a vertex type
        //using a wrapper class
        template <typename TVertex, typename TPoint>
        class Vertex2 : public TVertex
        {
        public:
```

```
                Vertex2( TPoint& p )
                {
                        this->coordinates = p;
                }
        }
```

The code implements an example with an assumption that the class `MyVertex` contains a member `coordinates`.

10.1 Usage

The user can check the concept by the placing the following code before the calling of the function.

```
boost::function_requires<GenerateCubeConcept<Mesh, Traits >();
```

If the concept is not satisfied, the error code will apear as follows:

```
$ /usr/include/boost/concept/detail/has_constraints.hpp
   :32:14:   required by substitution of template<class
    Model> boost::concepts::detail::yes boost::concepts
   ::detail::has_constraints_(Model*, boost::concepts::
   detail::wrap_constraints<Model, (& Model::
   constraints)>*) [with Model = VertexAdjacencyConcept
   <Mesh, Traits > >]
```

If the checking is not processed, the compilator still throws an error. The benefit of the error message above is that the compiler always throws such an error if the concept is not satisfied. If the concept is not checked, the compiler may produce a misleading or confusing error messages that are more demanding to solve.

11. Adapters

The chapter 7 explains the purpose of the adapters. The adapter is worth using only if the conversion of the used mesh representation is more demanding than the usage of the adapter.

Adapters generally can not be treated as standard implementation of the function that an adapter replaces; the computational complexity of the adapter increases the total complexity of the algorithm so that the resulting behavior is considered as *inefficient* (see the section 4.1).

11.1 Usage

The adapters are processing the demanded operation by the *brute-force*. When the user is building the traits(see the chapter 9) for the algorithm, he puts a calling of the adapter inside the function placed to traits instead of calling the function supported by a mesh structure.

Example:
Let us have an algorithm `X` that requires iterating through all adjacent vertices from a given vertex. We have a *face-vertex* polygonal mesh(see the section 1.2.1) that does not support the *efficient*(see the section 4.1) operation capable of getting the all adjacent vertices.

How do we build a traits for the algorithm `X`*?*

Let us assume that the type representing the mesh is named `X_fv_mesh`, and the required operation of the algorithm that has to be contained in traits is as follows:

```
static std::pair<vv_iterator,vv_iterator>
get_adjacent_vertices(
        const X_fv_mesh& m,
        const vertex_descriptor v);
```

Observation:
From the declaration of the function above, it is obvious that except the `X_fv_mesh`, two additional types has to be contained in the traits.

```
typedef vertex_descriptor X_fv_mesh&::Vertex; //supported by the
   structure
typedef vv_iterator ??? //not supported by the structure
```

Thus the question arises "How do we define the `vv_iterator`" and what do we place into the implementation of the function `get_adjacent_vertices()`?

Solution:
We use an adapter `hmira::adapters::vv_adapter::vv_iterator_adapter` which allow us to call a function that returns the demanded pair of the iterators and provides a type that was required for the function; in case of this example, `vv_iterator`.

The implementation of the traits will appear as follows:

```
#include <hmira/adapters/vv_adapter.hpp>

class X_fv_mesh_traits
{
//the following block of the code has to be
//added to the traits
//...

        typedef typename hmira::adapters::vv_adapter<
                X_fv_mesh,
                X_fv_mesh_traits
        >::vv_iterator vv_iterator;

        static inline
        std::pair<vv_iterator, vv_iterator>
        get_adjacent_vertices(
                const X_fv_mesh& m,
                vertex_descriptor v)
        {
                return hmira::adapters::vv_adapter<
                                X_fv_mesh,
                                X_fv_mesh_traits
                        >::vv_iterator_adapter(m, v);
        }

//...
};
```

The class `X_fv_mesh` does not support the operation that returns the iterators that are able to iterate through the adjacent vertices of a given vertex. Instead of calling the implemented function, the function refers to an adapter. In this case `vv_iterator_adapter`.

Conclusion

The goal was to build a robust library that can be a basis for 3D editors. The robustness is based on the template-like implementation technique that allow us to let it work over any implementation of the mesh. The secondary goal was to establish a convention for the future expansion of the library that has to be simple and clear. As the concepts philosophy is clear, the library is now easy to expand with a set of algorithm.

Since most of meshes used nowadays are used in the rendering software, the implementations are commonly not appropriate for algorithms that affects the topology. That is the main reason why the implementations of algorithms has own implementation of the mesh that loads the data from an external file or has own convertors. The Hmira library provides a solution that saves a user from developing conversion software between the implementations of the mesh. In addition, it allows a user to create an adaptor for the specific operations rather than implement a convertor and the implementation of the operation separatedly.

The attached examples shows that the metaprogramming technique used in the library brings a great benefits that allow us to create any algorithm as generic. The challenging part of the implementation of the algorithm is an algorithm decomposition; as the algorithm is decomposed, the algorithm is then ready to implement using elementary operations that are contained in the concept.

As the project is developed on repository `https://github.com/hmira/hmiralib`, anyone is allowed to expand the library using the instructions contained in the appendicies. One can implement algorithms, adaptors for the operations and the traits for the mesh library.

Bibliography

[1] G. J. Agin, "Representation and description of curved objects," DTIC Document, Tech. Rep., 1972.

[2] J. Žára, B. Beneš, J. Sochor, and P. Felkel, *Moderní počítačová grafika*. Computer press, 2004.

[3] P. K. Agarwal, L. Arge, and A. Danner, "From point cloud to grid dem: A scalable approach," in *Progress in Spatial Data Handling*. Springer, 2006, pp. 771–788.

[4] B. G. Baumgart, "Winged edge polyhedron representation," DTIC Document, Tech. Rep., 1972.

[5] M. Botsch, M. Pauly, C. Rossl, S. Bischoff, and L. Kobbelt, "Geometric modeling based on triangle meshes," in *ACM SIGGRAPH 2006 Courses*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1185657.1185839

[6] B. Grünbaum, "Graphs of polyhedra; polyhedra as graphs," *Discrete mathematics*, vol. 307, no. 3, pp. 445–463, 2007.

[7] S. Havemann and D. W. Fellner, "Generative mesh modeling." Ph.D. dissertation, University of Braunschweig-Institute of Technology, 2005.

[8] L. P. Chew, "Constrained delaunay triangulations," *Algorithmica*, vol. 4, no. 1-4, pp. 97–108, 1989.

[9] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, Aug. 1987. [Online]. Available: http://doi.acm.org/10.1145/37402.37422

[10] D. Cohen-Or and A. Kaufman, "Fundamentals of surface voxelization," *Graphical Models and Image Processing*, vol. 57, no. 6, pp. 453 – 461, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077316985710398

[11] S. W. Wang and A. E. Kaufman, "Volume sampled voxelization of geometric primitives," in *Proceedings of the 4th conference on Visualization '93*, ser. VIS '93. Washington, DC, USA: IEEE Computer Society, 1993, pp. 78–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=949845.949864

[12] R. A. Computer Graphics Group, "Openmesh website," 2002. [Online]. Available: http://www.openmesh.org/

[13] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt, "Openmesh-a generic and efficient polygon mesh data structure," 2002.

[14] S. Rusinkiewicz, "Trimesh library," accessed: 2013-05-30. [Online]. Available: http://gfx.cs.princeton.edu/proj/trimesh2/

[15] V. Simonis, "Adapters and binders: overcoming problems in the design and implementation of the c++-stl," *SIGPLAN Not.*, vol. 35, no. 2, pp. 46–53, Feb. 2000. [Online]. Available: http://doi.acm.org/10.1145/345105.345122

[16] D. R. Naugler, "C++11! new features," *J. Comput. Sci. Coll.*, vol. 28, no. 5, pp. 8–8, May 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2458569.2458571

[17] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).* Addison-Wesley Professional, 2004.

[18] A. Gurtovoy and D. Abrahams, "The boost mpl library," 2002.

[19] V. Karvonen and P. Mensonides, "The boost preprocessor library," 2001.

[20] A. Sutton and B. Stroustrup, "Design of concept libraries for c++," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, A. Sloane and U. Aßmann, Eds. Springer Berlin Heidelberg, 2012, vol. 6940, pp. 97–118. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28830-2_6

[21] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2010.

[22] C. Pheatt, "Intel threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1352079.1352134

Appendices

A. Running the Implemented Examples

The library has been tested on the OS *Ubuntu 12.10* using compiler `g++ 4.7` and has been developed using the multi-platform developement tools only.

A.1 Installation

```
git clone https://github.com/hmira/hmiralib.git
```

```
mkdir build
cd build/
```

```
cmake ..
```

```
make [custom executable]
```

A.2 Running

By running a command

```
./test/[custom executable] --help
```

the program prints a message that specifies the required parameters to run.

B. Templating by a Custom Polygon Mesh

To make the library work, we must build a proper traits in order to let the library recognize the required operations.

Depending on the demanding algorithm, we must create the traits that satisfy the concept of the algorithm. If the concept is not fulfilled the compiler throws an error.

Example:

```
template <typename my_mesh>
class my_mesh_implementation_traits<my_mesh>
{
public:
        typedef typename my_mesh::point point;
        typedef typename my_mesh::normal normal;

        typedef typename my_mesh::vertex_descriptor
           vertex_descriptor;
        typedef typename my_mesh::vertex_iterator vertex_iterator
           ;

        typedef typename my_mesh::edge_descriptor edge_descriptor
           ;
        typedef typename my_mesh::edge_iterator edge_iterator;

        typedef typename my_mesh::face_descriptor face_descriptor
           ;
        typedef typename my_mesh::face_iterator face_iterator;

        typedef typename my_mesh::fv_iterator fv_iterator;
        typedef typename my_mesh::vv_iterator vv_iterator;
        typedef typename my_mesh::ve_iterator ve_iterator;

        inline static bool add_vertex(
                                vertex_descriptor a,
                                my_mesh& m)
        {
                m.my_add_vertex(a);
                return true;
        }

        inline static bool create_face(
                                vertex_descriptor a,
                                vertex_descriptor b,
                                vertex_descriptor c,
                                my_mesh& m)
        {
                m.my_make_face(a,b,c);
                return true;
        }
}
```

From here, we have a traits that is capable of generating a cube.

```
int main()
{
        typedef typename my_mesh_implementation_traits<my_mesh>
           traits;
        auto cube = generate_cube<my_mesh, traits>();
}
```

C. Expansion of the Library

C.1 Creating a new Concept

Once we have an algorithm/adapter working, we have to create a concept following the convention provided by the library. In the concept it has to be contained the rules required for running the algorithm/adaptor.

Example:
We have a method that generates a cube

```
template <typename TMesh, typename TMesh_traits>
bool generate_cube(TMesh& m)
{
        typedef typename TMesh_traits::Point Point; //coordinates
        typedef typename TMesh_traits::Vertex Vertex; //vertex
        typedef typename TMesh_traits::Face Face; //face type

        auto a = Vertex(Point(-1,-1,-1));
        auto b = Vertex(Point(1,-1,-1));
        auto c = Vertex(Point(1,-1,1));
        auto d = Vertex(Point(-1,-1,1));
        auto e = Vertex(Point(-1,1,-1));
        auto f = Vertex(Point(1,1,-1));
        auto g = Vertex(Point(1,1,1));
        auto h = Vertex(Point(-1,1,1));

        TMesh_traits::add_vertex(m, a);
        TMesh_traits::add_vertex(m, b);
        TMesh_traits::add_vertex(m, c);
        TMesh_traits::add_vertex(m, d);
        TMesh_traits::add_vertex(m, e);
        TMesh_traits::add_vertex(m, f);
        TMesh_traits::add_vertex(m, g);
        TMesh_traits::add_vertex(m, h);

        TMesh_traits::create_face(m, a, b, c, d);
        TMesh_traits::create_face(m, b, c, g, f);
        TMesh_traits::create_face(m, c, g, h, d);
        TMesh_traits::create_face(m, a, d, h, e);
        TMesh_traits::create_face(m, e, f, b, a);
        TMesh_traits::create_face(m, h, g, f, e);

        return true; // the cube has been generated succesfully
}
```

Thus the concept appear as follows:

```
template <class TMesh, class TMesh_Traits>
struct GenerateCubeConcept
{
        typedef typename TMesh_traits::Point Point; //coordinates
        typedef typename TMesh_traits::Vertex Vertex; //vertex
        typedef typename TMesh_traits::Face Face; //face type

        TMesh m;
        Vertex v;
```

```
        float f;
        Point p;

        void constraints() {

                boost::function_requires<MeshConcept<TMesh,
                   TMesh_Traits> >();

                p = Point(f, f, f);
                v = Vertex(p);
                TMesh_traits::add_vertex(m, v);
                TMesh_traits::create_face(m, v, v, v, v);
        }
};
```

After building a concept, we place a concept checker in the implementation of the algorithm.

```
boost::function_requires<GenerateCubeConcept<TMesh, TMesh_Traits>
    >();
```

C.2 Creating a new Algorithm

The algorithm should be created from the elementary operations that are commonly used in the library. As more operations that are already used in the other algorithms the algorithm uses than the robustness of the library is enhanced. In the other words, user should let himself be inspired by the other algorithms so the concepts overlap as a result.

Inspiring from the implementation C.1 we can implement an algorithm that generates a tetrahedron.

```
template <typename TMesh, typename TMesh_traits>
bool generate_cube(TMesh& m)
{
        typedef typename TMesh_traits::Point Point; //coordinates
        typedef typename TMesh_traits::Vertex Vertex; //vertex
        typedef typename TMesh_traits::Face Face; //face type

        auto a = Vertex(Point(0,0,0));
        auto b = Vertex(Point(0,0,1));
        auto c = Vertex(Point(0,1,0));
        auto d = Vertex(Point(1,0,0));

        TMesh_traits::add_vertex(m, a);
        TMesh_traits::add_vertex(m, b);
        TMesh_traits::add_vertex(m, c);
        TMesh_traits::add_vertex(m, d);

        TMesh_traits::create_face(m, a, b, c);
        TMesh_traits::create_face(m, d, a, c);
        TMesh_traits::create_face(m, d, b, a);
        TMesh_traits::create_face(m, d, c, b);

        return true; // the tetrahedron has been generated
           succesfully
```

```
}
```

From here, compared to the implementationC.1 that generates a cube, the implementation that generates a tetrahedron requires one method to have modified. Thus the concept will be as follows:

```
template <class TMesh, class TMesh_Traits>
struct GenerateTetrahedronConcept
{
        typedef typename TMesh_traits::Point Point; //coordinates
        typedef typename TMesh_traits::Vertex Vertex; //vertex
        typedef typename TMesh_traits::Face Face; //face type

        TMesh m;
        Vertex v;
        float f;
        Point p;

        void constraints() {

                boost::function_requires<MeshConcept<TMesh,
                   TMesh_Traits> >();

                p = Point(f, f, f);
                v = Vertex(p);
                TMesh_traits::add_vertex(m, v);
                TMesh_traits::create_face(m, v, v, v);
                // 3 vertices instead of 4
        }
};
```

List of Abbreviations

$V_{adjacentV}$ Number of adjacent vertices of the vertex

$E_{adjacentV}$ Number of adjacent edges of the vertex

$F_{adjacentV}$ Number of adjacent faces of the vertex

$V_{adjacentE}$ Number of adjacent vertices of the edge

$E_{adjacentE}$ Number of adjacent edges of the edge

$F_{adjacentE}$ Number of adjacent faces of the edge

$V_{adjacentF}$ Number of adjacent vertices of the face

$E_{adjacentF}$ Number of adjacent edges of the face

$F_{adjacentF}$ Number of adjacent faces of the face

$V_{consistF}$ Number of vertices that forms the face

$E_{consistF}$ Number of edges that forms the face

V_{all} Number of vertices contained in entire structure

E_{all} Number of vertices contained in entire structure

F_{all} Number of vertices contained in entire structure