UNIVERSITÀ DEGLI STUDI DI CAGLIARI

FACOLTÀ DI SCIENZE

Corso di Laurea Magistrale in Informatica

PolyCubes Optimization

Generating Coarse Quad-Layouts via Smart Polycube Quantization

Supervisor
Prof. Riccardo Scateni

M.Sc. Candidate
Gianmarco Cherchi
Matr. N. 49183

ACADEMIC YEAR 2014/2015

"We all change, when you think about it. We are all different people, all through our lives. And that's okay, that's good, you gotta keep moving so long as you remember all the people that you used to be. I will not forget one line of this, not one day, I swear . . . "

The eleventh Doctor.

Quad-Layouts are useful in a large number of applications in Computer Graphics. In particular we use them in Animation and in Game Development to describe, through the quads, the semantic parts of the characters to animate. In this thesis we propose a novel algorithm for the generation of pure quad-layouts by using optimized polycubes as an intermediate step. We have developed an algorithm that solves, in an iterative way, a mathematical model we have defined to optimize the polycube. This model is only composed of linear constraints. The resultant polycube allows us to extract a coarse quad-layout with a low number of quads. We have obtained a significant percentage of reduction of the quads' number in the quad-layout and good performances.

I Quad-Layout sono impiegati in molte applicazioni nel campo della Computer Graphics. In particolare possono essere utilizzati nell'Animazione e nello Sviluppo di Videogiochi per descrivere, mediante i domini, le diverse parti di un personaggio da un punto di vista semantico. In questa tesi viene proposto un algoritmo innovativo per la generazione di quad-layout utilizzando come step intermedio dei policubi ottimizzati. È stato sviluppato un algoritmo che si occupa dell'ottimizzazione dei policubi mediante la risoluzione, in maniera iterativa, di un modello matematico da noi definito. Tale modello è composto esclusivamente da vincoli lineari. Il policubo ottenuto come risultato consente l'estrazione di un quad-layout "coarse" che presenta un basso numero di domini. I risultati raggiunti mostrano una significativa percentuale di riduzione del numero di domini nel quad-layout e delle buone performance.

Contents

1 Introduction

Game Development and Animation are two of the key fields of Computer Graphics and today they are topics of many research projects. Animating a 3D model is one of the main goals of these disciplines.

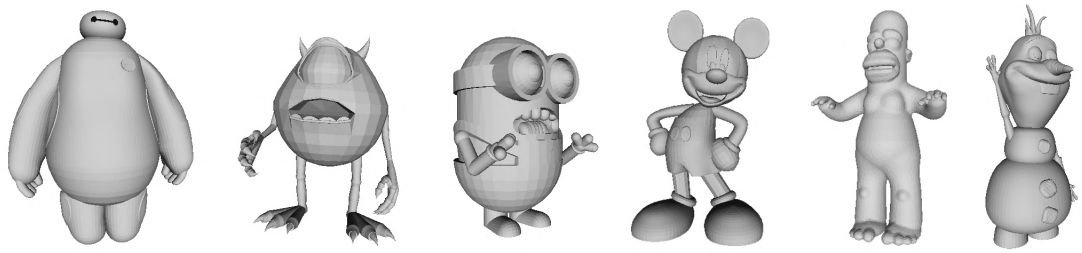


Figure 1.1: *Examples of 3D models.*

The surface of digital characters is often represented by means of hundreds (if not thousands) of small quadrilaterals glued together along shared edges. In quadrilateral-mesh (also called quad-mesh) the way this quadrilaterals are connected together is quite important for the quality of the animation: the more their edges are aligned to the features of the character the more realistic the animation will be. Moreover, it is common practice to group together quadrilaterals into bigger quadrilateral domains, each one describing a semantic part of the character. This decomposition of the surface of the character, often referred to as "quad-layout" (see *Section 2.1*), is quite important to compactly store important information describing the appearance of the shape, such as textures, material properties, fur or other tiny scale features of the surface.

Today techniques like virtual sculpting or scanning systems produce irregular meshes (without a well-define structure) that are not very useful in Computer

Graphics. The definition of a structure for an unstructured mesh, through specific operations called retopology or remeshing, is not simple (in *Section 2.1* we cite some techniques at the state of the art).

Figure 1.2: *Examples of quad-layouts obtained with different algorithms taken from [14], [2] and [15].*

The aim of this thesis is to obtain a coarse quad-layout to give a structure to an unstructured mesh. To achieve this result we have defined a pipeline (see *Section 3*) that starts from a 3D model and, thanks to its polycube representation, arrives to a final coarse quad-layout mapped to a structured quad-mesh. We propose an automatic approach for the optimization of the polycube used as an intermediate step in the pipeline. An optimized polycube makes the extraction of a coarse quad-layout possible. The obtained quad-layout can be directly mapped to the relative quad-mesh. The optimization is made possible by a mathematical model we developed with a quadratic objective function and all linear constraints (explained in detail in *Chapter 4*). Solving this model allows us to perform the alignment of vertices, edges and faces of the polycube in an efficient way. The model is built and solved several times inside an algorithm in an iterative way (as explained in *Chapter 5*). For each model we extract its polycube (with the PolyCut algorithm [10]), we perform the optimization and we extract the final optimized quad-layout. We have obtained satisfying results, illustrated in the *Chapter 6*, with significant percentage of reduction of the quads' number in the quad-layout and good performances.

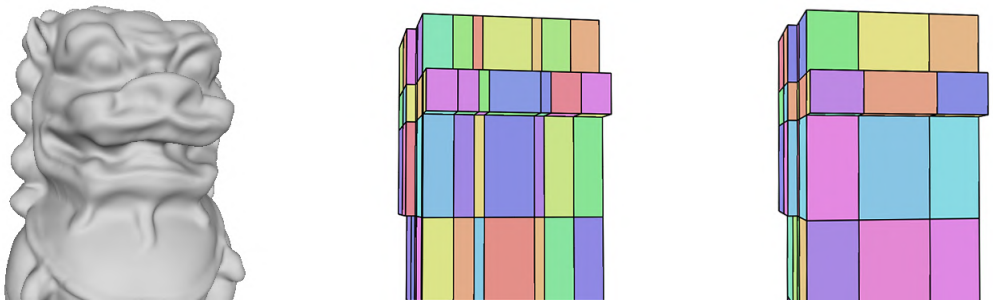


Figure 1.3: *Example of the polycube optimization: the Dragon's face in the original model (left), in the initial polycube (center) and in the optimized polycube (right).*

The rest of the thesis is organised as follows: in *Chapter 2* we will give a brief summary of the background and the state of the art useful to understand this work and the problem we want to solve, in particular we talk about quad-meshes, quad-layouts and polycubes; in *Chapter 3* we will discuss in detail the problem we want to solve and why it is important; in *Chapter 4* we will show the development of the mathematical model used in our algorithm, explaining its objective function and its constraints; in *Chapter 5* we will present the entire algorithm and the developed interactive tool to run it (with an example); in *Chapter 6* we will illustrate the obtained results with an analysis of memory and time performances; in *Chapter 7* we will draw our conclusions and in *Chapter 8* we will analyse the limitation of the algorithm and we will explain what can be done in the future to improve and continue the work. Finally, in *Appendix A*, we will show in detail some example about the code used for the creation and the optimization of the mathematical model.

2 State of the art

In Animation and in Game Development the use of *Quad-Meshes* and *Quad-Layouts* is very important. Quad-meshes are much more useful than Triangle-meshes in several fields of Computer Graphics. Different types of quad-meshes can be obtained from an initial 3D model. PolyCubes can be an innovative way to obtain optimized quad-meshes and optimized quad-layouts for different reasons. In this chapter we describe the most important features and qualities of quad-meshes, quad-layouts and polycubes.

2.1 Quad-Meshes and Quad-Layouts

In 3D Computer Graphics a *Polygon-Mesh* is a collection of vertices, edges and faces of a polyhedral object. Quad-meshes are a type of polygonal meshes in which faces are quadrilateral.

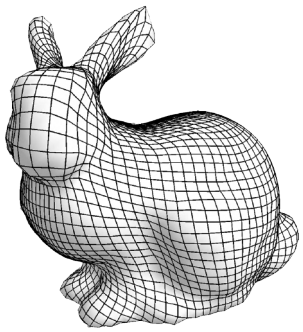


Figure 2.1: *An example of quad-mesh that represents the 3D model of the Stanford Bunny.*

In addition to the classical terminology like vertices, edges and faces, to talk about quad-meshes we must introduce some definitions like *singularity*, *chart bound-*

ary and *chart*. A "singularity" is a particular vertex of a quad-mesh that has valence (number of incidence edges) different from 4. In quad-meshes the most recurring singularities are those of valence 3 or valence 5. We talk about "chart boundaries" or "separatrices" when we have a line that connects a singularity with another singularity of the mesh. Using separatrices we can divide the mesh in quads called "charts" or "domains".

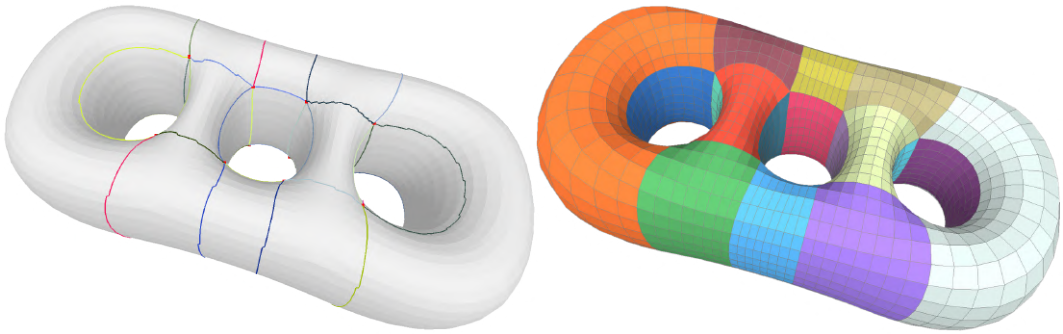


Figure 2.2: *An example of a quad-mesh with singularities and chart boundaries (left) and the same model with charts identified by different colors (right) taken from [14].*

As explained in the survey [2], based on the number and the position of singularities in a mesh, we can perform a subdivision of quad-meshes in four classes:

- *Regular meshes*: composed only by valence-4 vertices. They have no singularities and they are useless in Computer Graphics because they don't allow to perform a good distortion of the mesh.

- *Semi-regular meshes*: there are singularities. It is simple to divide this type of mesh in domains using chart boundaries. We assume that the number of domains is much less than the number of faces of the mesh. Each vertex that is inside a domain has valence 4 and each vertex that is in the chart boundary may have valence 3, 4 or 5. This class of meshes is the most interesting for several kind of Computer Graphics applications.

- *Valence Semi-regular meshes*: there are a lot of vertices with valence 4. Each semi-regular mesh is a valence semi-regular mesh but the opposite is not true. Semi-regular and valence semi-regular meshes are often included in the same class. It can be useful to subdivide these two classes because it makes it possible to define algorithms for the chart subdivision (by using semi-regular meshes) and algorithms for the minimization of the singularities number (by using valence semi-regular meshes instead).

- *Unstructured meshes*: the mesh has no structure and it is completely useless. This type of mesh is often obtained by the direct transformation of a triangle-mesh in a quad-mesh through the triangle union (two triangles become a quad, an even number of triangles is required). In this case a post-processing step is always required to give a structure to the mesh.

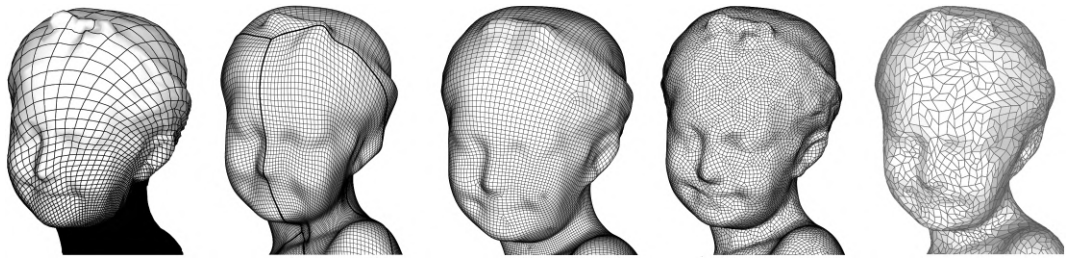


Figure 2.3: *The quad-mesh classification: Regular, Semi-regular, Valence semi-regular and Unstructured meshes taken from [2].*

Characteristics of a quad-mesh are:

- *Quad quality*: each angle of the mesh must be more or less 90° and opposite edges of each quad must have more or less the same length.
- *Regularity*: the number of singularities. This parameter changes depending on the application.
- *Good placement of irregular vertices*: this characteristic describes the placement of singularities in a mesh. A good position of singularities allows to obtain a good final quad-layout.
- *Resolution adaptivity*: in a quad mesh it is important to have the possibility to adapt the resolution of the mesh (number of vertices and quads) in the portions that require a higher number of details.

There are several application fields for quad-meshes. In Polygonal modeling and Animation it is fundamental to have a quad-mesh that has a well-defined structure (note that triangle-meshes are totally unstructured). Triangle-meshes don't guarantee a good quality animation because it is not possible to define an edge flow that matches the features of the character and its main curvatures. Quad-meshes are more difficult to create than triangle-meshes but they have more advantages. When they have a good structure, and the edges of the mesh are perfectly aligned with the main curvatures of the object, they guarantee a good quality animation

(i.e. without artifacts). Note that an unstructured quad-mesh is equivalent to a triangle-mesh and it has no advantages for the animation.

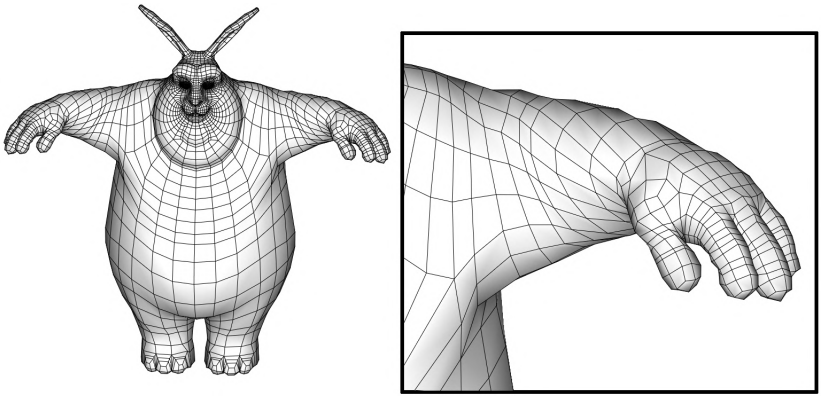


Figure 2.4: *A quad-mesh for the computer animation, taken from [2], of the "Big Buck Bunny" movie, Blender Institute 2007 [5] .*

Another application field for quad-meshes (very important for this thesis) is the mesh domain subdivision called "Quad-Layout". It is obtained by connecting the mesh singularities through chart boundaries. Having a good quality quad-layout is very important for many applications: in Game Development having a reduced but representative set of quadrangular domains allows a representation and an animation management that is much more efficient and less complex. A good quad-layout is also important in other fields of Computer Graphics because it allows to represent complex objects with a simple geometry, for example the Texture-mapping is made easier by a quadrangular chart subdivision. In general a good quad layout is obtained through a good alignment of the mesh singularities. Today this research topic is being considered not only for surface meshes but also for hex-meshes like explained in [6].

There are different approaches to compute a quad-layout. In 2011 Tarini et al. proposed a method based on the topological simplification of the cross field in input followed by a global smoothing [14]. In the same year Bommes et al. created an algorithm which detects helices in a quad-mesh and is able to remove most of them by applying a novel grid preserving the simplification operator which is guaranteed to maintain an all-quadrilateral mesh [1]. In 2012 Campen et al. present a theoretical framework and practical method for the automatic construction of simple quad-layouts on manifold surfaces based on the careful construction of the layout graph's combinatorial dual [3]. In 2014 Campen et al. present a user-assisted method for the interactive design of quad-layouts called *Dual Strip Weaving* [4]. In 2015 Usai et al. proposed a method to convert a tri-mesh in the relative quad-layout

using its curve skeleton to obtain a semi-regular quad-mesh [15]. In this thesis we propose another approach to obtain an optimized quad-layout of a mesh (tri-mesh or quad-mesh) using its polycube representation as an intermediate step.

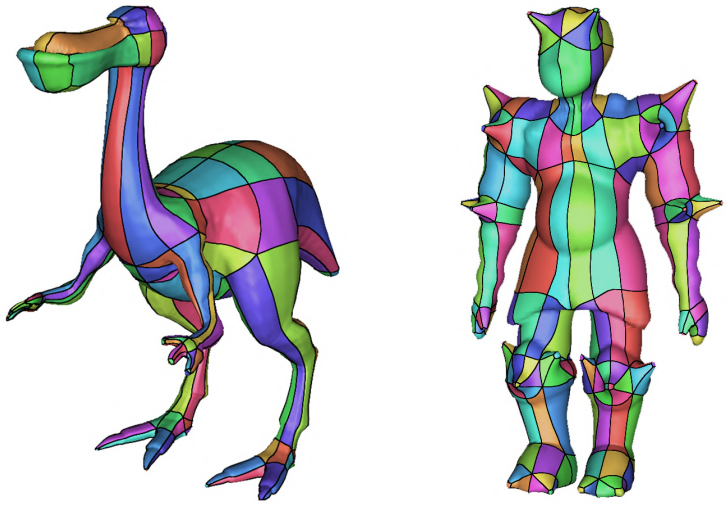


Figure 2.5: *Examples of quad-layouts taken from [15].*

2.2 PolyCubes

PolyCubes are orthogonal polyhedra made up of only axis-aligned faces. They were initially created as a support for the Texture-mapping [13]. In Computer Graphics it is very important to have a compact representation of a shape and polycubes make it possible to achieve this result by giving an explicit geometry of the analyzed model. They are characterized by three important features: axis-aligned faces, only 90° dihedral angles and planar faces. The most important property of a polycube is the ability to represent the original shape in a simple way. This allows to perform analysis and geometrical manipulation on a simple shape instead of on a complex model.

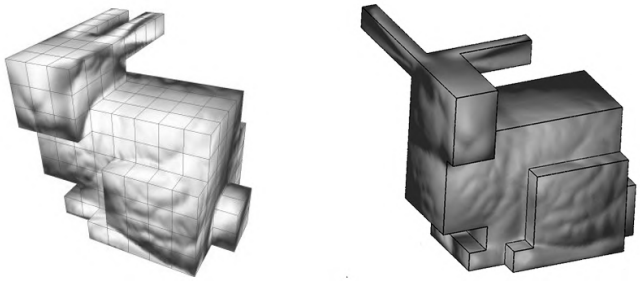


Figure 2.6: *Two examples of polycubes of the same model obtained with different algorithms, the first taken from [13] and the second taken from [10].*

Polycubes are a good combination of the original shape of the model, its simplification and topology preservation. Nowadays polycubes are a new research topic with a large number of possible applications. The first application is texture-mapping, as explained in [13]. Polycubes provide a mechanism that could be used for seamless texture-mapping with low distortion by using the surface of the polycube as the texture domain. A simplified example of this method is shown in the *Figure 2.7.*

Figure 2.7: *Example of texture-mapping described previously, taken from [13].*

Another important application of polycubes is the creation of Hexahedral meshes, important for finite element simulation. A polycube admits a trivial hex-mesh that can be generated by gridding its interior. If a volumetric parameterization between the polycube and the initial shape is available the connectivity of the hex-mesh can be mapped back to the input shape, generating an high quality hexahedral mesh [7].

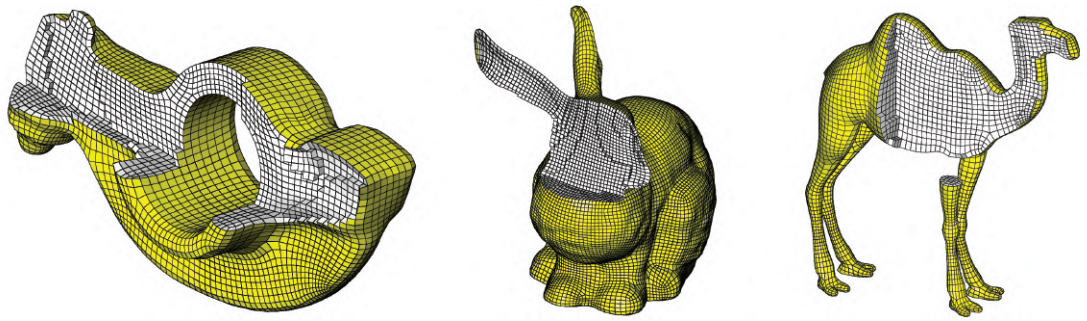


Figure 2.8: *Example of hex-meshing, taken from [7].*

There are a lot of other scenarios in which polycubes can find use, for example for the spline fitting or the mapping of a volume bound by a surface with general topology onto a topologically equivalent base domain. Another important line of research in which polycubes are useful are quad-only layouts. The coarse structure (optimized if possible) of polycubes makes it possible to obtain semi-regular quad-meshes and quad-layout in a simple way. As we will see later this is the main problem that this thesis aims to solve.

There are several algorithms to obtain polycubes with different characteristics. The first algorithm for the automatic computation of a polycube, the "*Topology-based*" paradigm, was introduced by Lin et al. in 2008 [9]. In 2009 He et al. proposed an alternative method based on the "*Divide-and-conquer*" paradigm [8]. In 2011 Gregson et al. investigated the problem of computing high-quality polycubes in the context of hex-meshing using the "*Deformation-based*" algorithm [7]. In 2013 Livesu et al. proposed an innovative algorithm for the polycube extraction, "*PolyCut*" [10], and this is the algorithm used in this thesis to compute the input polycubes for our algorithm.

In *Figure 2.9* some results of different algorithms are shown. The first is obtained with the divide-and-conquer algorithm and it can be seen that it generates too complex domains with a high number of useless corners. The second is obtained with the deformation-based approach and it produces more compact domains but unnecessary charts and artifacts. The third result, the best among these, is obtained with PolyCut and it is a good trade-off between compactness and mapping distortion.

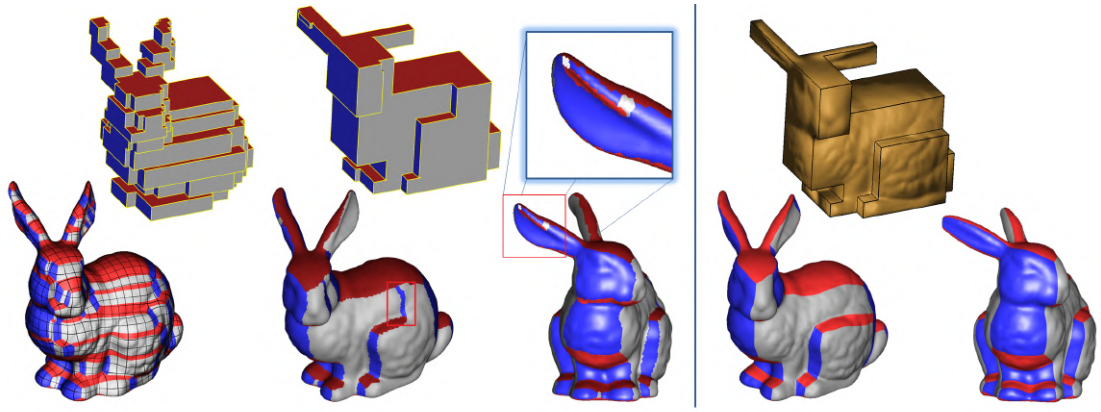


Figure 2.9: *Examples of polycubes obtained with different algorithms, taken from [10].*

A very simplified description of the *PolyCut* algorithm can be divided in four steps:

- *Initial Labelling*: segmentation of the input mesh followed by a region extraction.
- *Discrete Optimization*: the previous segmentation is optimized in order to produce a valid polycube embedding.
- *PolyCube Extraction*: polycube vertex positions are defined and regions found in the step one are projected into axis-aligned planes.

- *Parametrization*: the parametrization between the polycube and the original input mesh is computed.

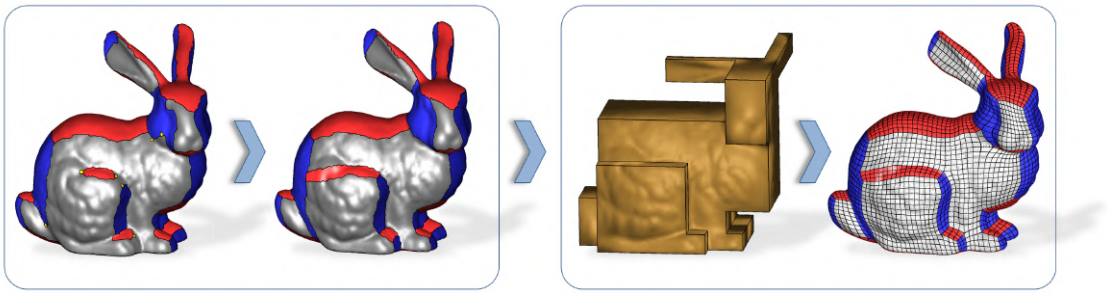


Figure 2.10: *The four steps of the PolyCut algorithm, taken from [10].*

3 Motivation

The aim of this thesis is the *PolyCube Optimization* to obtain quad-meshes and quad-layouts that are more useful for Game Development and Animation. This work is included into a pipeline as follows:

[3D model] → [polycube] → [polycube optimization] → [quad-layout]

The polycube extracted by the initial mesh can be used to obtain a quad-mesh of the initial model (that can be a tri-mesh) and the relative quad-layout. This thesis consists in the third step of this pipeline that makes it possible to create a quad-mesh topologically equivalent to the initial model and a quad-layout with the lower number of domains. The quad-layout found in the polycube can be easily mapped in the resultant quad-mesh. The optimization proposed in this work consists in the edges/faces alignment. The creation of a quad-layout in a polycube is obtained through the extension of edges and faces (like separatrices) with the purpose of identifying the quads. A simple example in the 2D space is shown in *Figure 3.1*:

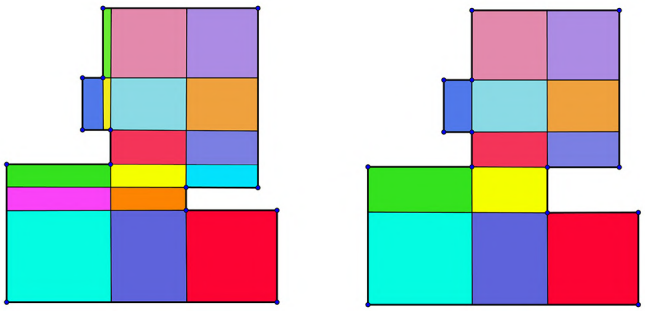


Figure 3.1: *An example in the 2D space of the initial model (left) and its optimization (right).*

As it is shown in this picture, the initial polygon (that represents a 2D approximation of the polycube relative to the mesh "Bimba") has 17 domains. If the edges alignment is performed, in particular if we align the two edges of the nape and the edge of the shoulder with the one of the chin, the model has 12 domains.

In the 3D space the problem to solve is the same. The domains' individuation is performed through chart boundaries (separatrices) and the number of quads depends on the alignment of the polycube's faces. In the following picture the polycube of the "Squirrel" model is shown: the initial number of domains is 26 and, after the purposed optimization, the final number of domains is 18.

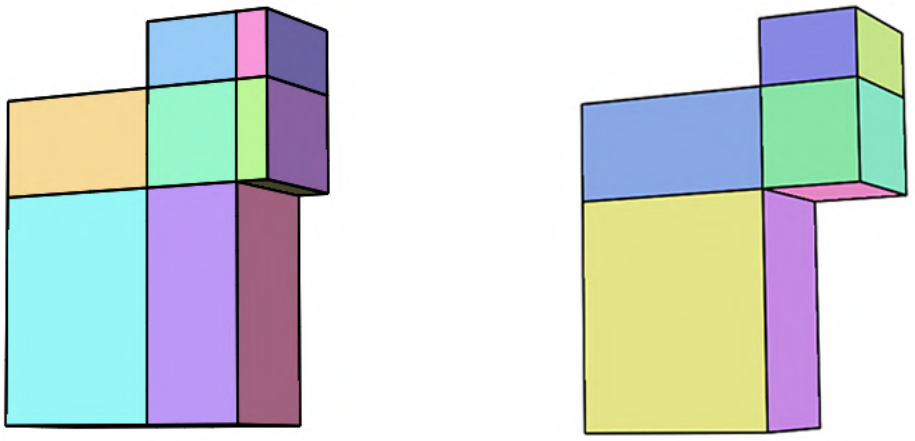


Figure 3.2: *An example in the 3D space of the initial model (left) and its optimization (right).*

Our approach to reach this goal consists in an algorithm that performs, in an iterative way, the optimization of a mathematical model appropriately built. The mathematical model is composed of an objective function (split in two parts) and only linear constraints. We decided to solve the problem through a mathematical model because it makes it possible to reach our goal in an efficient, elegant and robust way. In the next chapter the construction of the model used by our algorithm is analyzed in detail.

4 The Mathematical Model

As we said in the previous chapter, solving our problem through a mathematical model makes it possible to reach our goal in an efficient, elegant and robust way. The solution of this problem is the main part of an algorithm that computes the problem until the convergence is reached (the complete algorithm is explained in the next chapter). In this chapter we will describe the construction of the mathematical model that we used in our approach, step by step.

4.1 The Objective function

The objective function is the most important part of this model because it allows us to obtain a good trade-off between the topological preservation and the alignment required to reach our goal. We split the objective function in two parts: one takes care of the shape preservation called "E_{shape}" and the other takes care of the edges/faces alignment called "E_{align}". These two parts, that we will explain in detail, are multiplied by two scalar values α and β (varying between 0 and 1) to make it possible to work with E_{shape} and E_{align} with different weights. The final objective function is the following:

$$min\ e\ =\ \alpha \cdot E_{shape}\ +\ \beta \cdot E_{align}$$

4.1.1 Shape preservation

This portion of the objective function has the task of preserving the shape of the polycube. It is a simple attraction between the variables of the problem and the original vertices' value. If we denote with V the set of vertices of a polycube we can express the E_{shape} as follows:

$$E_{shape} = \sum_{i \in V} \left[(x_i - \tilde{x}_i)^2 + (y_i - \tilde{y}_i)^2 + (z_i - \tilde{z}_i)^2 \right]$$

In this formula x_i, y_i and z_i represent respectively the variables of the problem relative to the x-coordinate, y-coordinate and z-coordinate of the vertex i, instead $\tilde{x}_i$, $\tilde{y}_i$ and $\tilde{z}_i$ represent the original value of the vertex i coordinates. This portion of the objective function is multiplied by an α scalar smaller than the β scalar of the E_{align} portion. If we suppose $\alpha = 1$ (and $\beta = 0$ or a very small number) we clearly obtain a polycube equal to the original in input.

4.1.2 Alignment

The most important part of the problem is the portion of the objective function that we have called E_{align}. This portion takes care of the edges/faces alignment to reach our goal. In this section we will explain step by step how to find which edges and faces can be aligned and the coordinate along which to perform this alignment. As we explained earlier, one of the most important properties of a polycube is the fact that edges and faces are "axis-aligned". If we preserve this property through opportune constraints we can reduce the edges/faces alignment to vertices alignments. Indeed if we change the position of a vertex we change the position of edges and faces that include it. For this reason we describe all the alignments in terms of vertices. We denote with A the set of pairs of vertices we want to align (along one or more coordinates) and we split it in three sub-sets: A_x is the set of pairs of vertices that we want to align along the x-coordinate, A_y is the set of pairs of vertices that we want to align along the y-coordinate and A_z is the set of pairs of vertices that we want to align along the z-coordinate. We can express the E_{align} as follows:

$$E_{align} = \sum_{(i,j) \in A_x} (x_i - x_j)^2 + \sum_{(i,j) \in A_y} (y_i - y_j)^2 + \sum_{(i,j) \in A_z} (z_i - z_j)^2$$

We now explain how to compute the set A and how to split it in the three sub-set A_x, A_y and A_z. The principle on which our approach is based is "*to align local neighbour vertices to remove the largest number of misalignments*". An efficient way to find neighbour vertices is the *Voronoi Diagram*. Through the Voronoi diagram extended to the 3D space we are able to find the neighbours of a vertex by considering its Voronoi cell and the cells adjacent to it. So we consider two vertices adjacent if their Voronoi cells are adjacent. We want to align adjacent vertices along one

coordinate to perform the edges/faces alignment in the polycube. In *Figure 4.1* we show the 3D Voronoi diagram of the polycube of the "Squirrel" model taken from our interactive tool (that will be described in the next chapter). In this image the vertices of the polycube are shown in yellow, the edges in grey and the contours of the Voronoi cells in light blue.

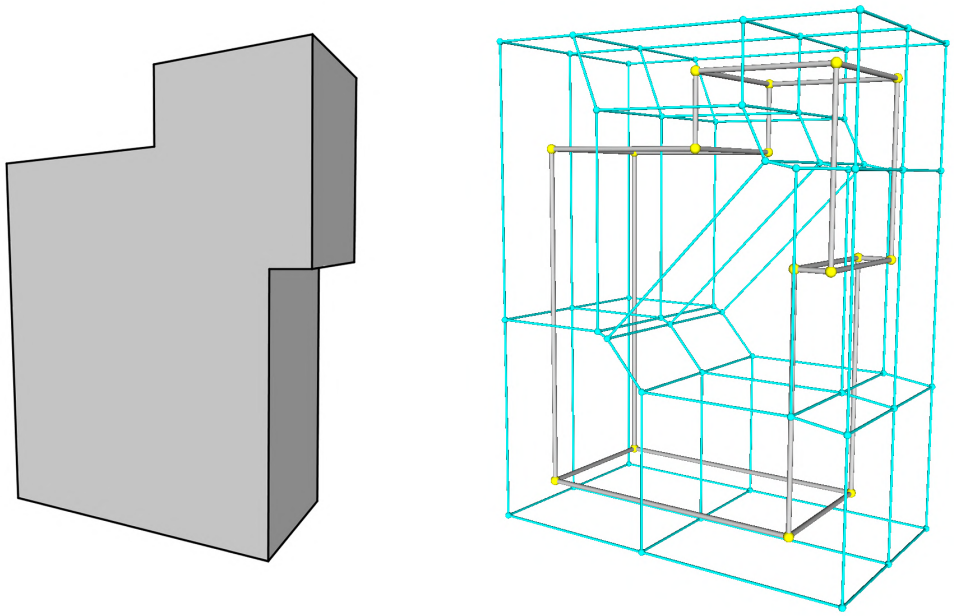


Figure 4.1: *The Squirrel model polycube (left) and its 3D Voronoi diagram (right).*

Once we found the Voronoi adjacencies we must perform a selection of them and remove those that are useless for the modeling of our mathematical problem. Obviously do not consider as adjacent vertices being end-points of the same edge. They are already aligned along one coordinate and it is not possible to align them along another coordinate without changing the edge orientation or without making the edge length equal to zero (vertex collapse).

Another case that we must reject in the modeling of our problem is the adjacency between two vertices that are end-points of two edges that have the other end-point in common (as it is shown in *Figure 4.2*). This situation is not interesting if we work with polycubes because the pairs of vertices (A,B) and the pairs of vertices (A,C) are already aligned along one coordinate and it is not possible to align the pair (B,C) without losing the axis-align property or without having an edge collapse.

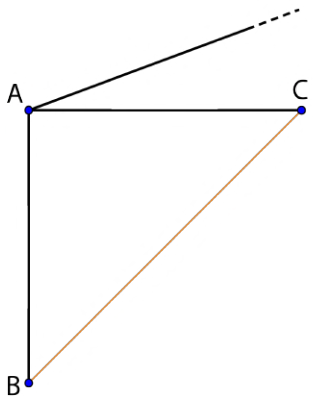


Figure 4.2: *An example of adjacency to reject.*

We are not interested in external adjacencies either. It is useless to try to align external adjacent vertices because their alignment doesn't produce any reduction of the domain's number. Identifying if an adjacency (that we can see as an imaginary edge) is internal or external to a polygon that doesn't only have convex faces is not an easy problem. To do this we must evaluate all possible cases that we can have. We can see that an edge is internal to a polygon if its direction results internal to the polygon for both its end-points. In a polycube we could have four different types of end-points for an edge representing an adjacency. We consider the dihedral angles between the faces that have the considered end-point in common to discriminate these cases. Below we show all possible cases with their relative Look-Up tables. In the following images the red arrow represents the normal of faces and the yellow vertex is the considered end-point. In the table we have, in the 1^{st}, 2^{nd} and 3^{rd} column, a "+" if the direction of the edge is concordant with the normal of the face and a "-" if not. In the 4^{th} column we have an "E" if the edge is external to the polygon and "I" if it is internal.

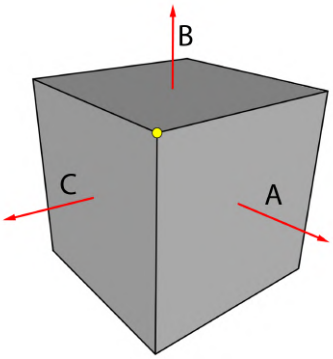


Figure 4.3: *3 convex dihedral angles.*

A	B	C	Res
+	+	+	*E*
+	+	-	*E*
+	-	+	*E*
+	-	-	*E*
-	+	+	*E*
-	+	-	*E*
-	-	+	*E*
-	-	-	*I*

Table 4.1: *Look-Up Table of case 1.*

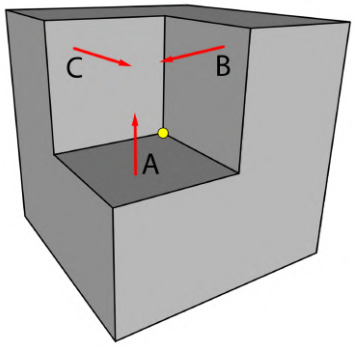


Figure 4.4: *3 concave dihedral angles.*

A	B	C	Res
+	+	+	*E*
+	+	-	*I*
+	-	+	*I*
+	-	-	*I*
-	+	+	*I*
-	+	-	*I*
-	-	+	*I*
-	-	-	*I*

Table 4.2: *Look-Up Table of case 2.*

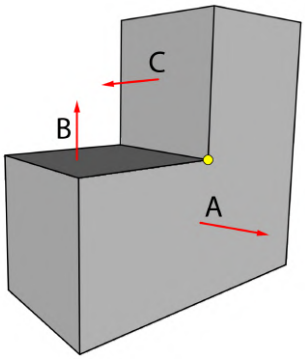


Figure 4.5: *2 convex dihedral angles and 1 concave dihedral angle.*

A	B	C	Res
+	+	+	*E*
+	+	-	*E*
+	-	+	*E*
+	-	-	*E*
-	+	+	*E*
-	+	-	*I*
-	-	+	*I*
-	-	-	*I*

Table 4.3: *Look-Up Table of case 3.*

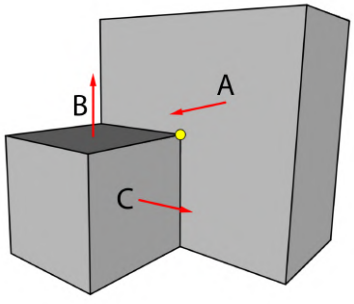


Figure 4.6: *2 concave dihedral angles and 1 convex dihedral angle.*

A	B	C	Res
+	+	+	*E*
+	+	-	*E*
+	-	+	*E*
+	-	-	*I*
-	+	+	*I*
-	+	-	*I*
-	-	+	*I*
-	-	-	*I*

Table 4.4: *Look-Up Table of case 4.*

The vertices that are end-points of the adjacencies left after the aforementioned selection (that represents possible alignments) constitute the set A. The last problem to solve to obtain the sub-sets A_x, A_y and A_z is to determine the pairs (v_i,v_j) that allow an alignment and to understand, for each pair, the coordinate along which to perform the alignment (given a couple of vertices they can only be aligned along one coordinate). If we found more than one possible alignment for the vertex v_i along the same coordinate we have to take a choice: the best thing to do is to align the vertex v_i with the vertex v_j that has the smaller *delta* along the considered coordinate.

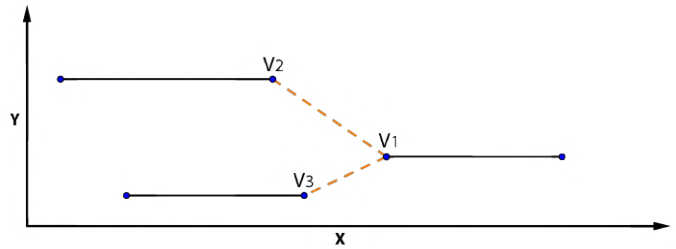


Figure 4.7: *An example of possible multiple alignments.*

In *Figure 4.7* a case of multiple possible alignments is shown. The vertex v_1 can be aligned with the vertex v_2 and with the vertex v_3 along the y-coordinate. If we insert both these pairs in the objective function the solver tries to align v_1 and v_2 and also v_1 and v_3 along the same coordinate and it probably fails in both these cases. We can absolutely make a choice and since in this case the $\Delta_y(v_1, v_3)$ is smaller than the $\Delta_y(v_1, v_2)$ we insert the pair (v_1,v_3) in the set A_y. Note that if we decide to align the vertex v_1 with the vertex v_3 along the y-coordinate it is very important not to try the alignment of v_3 with another vertex along the same coordinate. This procedure is repeated for each vertex present in the set A and, for each of them, we can only create a single pair for each coordinate (it is impossible to align a vertex with more than one other vertex along the same coordinate). So we create three sets containing the same possible alignments but sorted for different coordinates so that it is possible to choose the best one for each vertex.

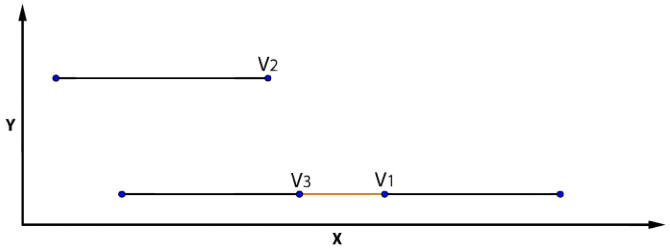


Figure 4.8: *The previous example after the alignment.*

4.2 Constraints

We have seen how to create the objective function for our mathematical model. We now explain what kind of problems we must solve through the addition of constraints. We have to preserve the property of the input polycube and we have to avoid artifacts like edge or vertex collapse. Remember that the problem is modelled and solved more than once in our algorithm (iterative approach) so it is important to preserve, through appropriate constraints, all results obtained in the previous steps. Note that in our mathematical problem all constraints are linear.

4.2.1 Collinearity of the end-points

One of the essential properties of a polycube is that its edges and faces are axis-aligned. We have seen before that in the E_{align} portion of the objective function we try to align pairs (v_i,v_j) of vertices along one coordinate. To do this the mathematical solver of our problem assigns the same coordinate to v_i and v_j. For this reason it is important to add a constraint that, for each edge of the polycube, secures that its end-points preserve the collinearity.

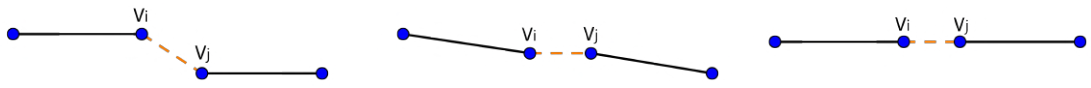


Figure 4.9: *In the first image we have two vertices that want to reach an alignment, in the second image we have the alignment without constraints and in the third image we have the correct alignment conditioned by constraints.*

We denote with E the set of pairs (v_i,v_j) that represent the end-points of the edges of the polycube and we split it in three sub-sets: $E_{\perp x}$ is the sub-set of edges perpendicular to the x-axis, $E_{\perp y}$ is the sub-set of edges perpendicular to the y-axis and $E_{\perp z}$ is the sub-set of edges perpendicular to the z-axis. We can add the following constraints to the problem:

$$\begin{aligned} \forall (i,j) \in E_{\perp x} \qquad & v_i(x) = v_j(x) \\ \forall (i,j) \in E_{\perp y} \qquad & v_i(y) = v_j(y) \\ \forall (i,j) \in E_{\perp z} \qquad & v_i(z) = v_j(z) \end{aligned}$$

where $v_i(x)$, $v_i(y)$ and $v_i(z)$ denote the variables of the problem relative respectively to the x-coordinate, y-coordinate and z-coordinate of the vertex v_i.

4.2.2 Minimum length of the edges

Another problem that we could have during the solving of the mathematical model is the collapse of the end-points of an edge. In a polycube the end-points of an edge have two equal coordinates and if there are no constraints that avoid the equality of the third coordinate we could lose an edge. To solve our problem we use an *Integer solver* that computes a resultant polycube with only integer coordinates. This is important to obtain the final quad-mesh in an easy way from the resultant polycube. For this reason the smallest length that an edge can have is 1. We denote, as explained before, E as the set of pairs (v_i,v_j) that represent the end-points of the edges of the polycube and we split it in three sub-sets: $E_{\|x}$ is the sub-set of edges parallel to the x-axis, $E_{\|y}$ is the sub-set of edges parallel to the y-axis and $E_{\|z}$ is the sub-set of edges parallel to the z-axis. We can add the following constraints to the problem:

$$
\begin{aligned}
&\forall (i,j) \in E_{\|x} \qquad if\ (v_i(x) > v_j(x))\ then\ v_i(x) - v_j(x) \geq 1\ else\ v_j(x) - v_i(x) \geq 1 \\
&\forall (i,j) \in E_{\|y} \qquad if\ (v_i(y) > v_j(y))\ then\ v_i(y) - v_j(y) \geq 1\ else\ v_j(y) - v_i(y) \geq 1 \\
&\forall (i,j) \in E_{\|z} \qquad if\ (v_i(z) > v_j(z))\ \ then\ v_i(z) - v_j(z) \geq 1\ \ else\ v_j(z) - v_i(z) \geq 1
\end{aligned}
$$

4.2.3 Vertices already aligned

As we said previously the solution to this problem is repeated more than once by the algorithm in an iterative way. With the actual formulation of the objective function we could have the following problem: at the step n the edges i and j are aligned but at the step $n+1$ the edge j is aligned with the edge k and misaligned with the edge i. Another problem that we could have is that during the problem solving two edges that were aligned in the original polycube are misaligned. To solve this problem we must preserve all alignments that were already present at the step $n-1$ by using constraints for the problem modelled at the step n. In the *Section 4.1.2* we have formalized the set A to explain the construction of the E_{align}. The set A is the set of pairs of vertices we want to align found through the Voronoi adjacencies. In this set we also have pair of vertices (v_i,v_j) already aligned in the previous step. So we define the set A', with $A' \subseteq A$, as the set of pairs of vertices already aligned for at least one coordinate. We can now add the following constraints to the problem :

$$
\begin{aligned}
&\forall (i,j) \in A'_x \qquad v_i(x) = v_j(x) \\
&\forall (i,j) \in A'_y \qquad v_i(y) = v_j(y) \\
&\forall (i,j) \in A'_z \qquad v_i(z) = v_j(z)
\end{aligned}
$$

where A'_x is the sub-set of pairs of vertices already aligned along the x-coordinate, A'_y is the sub-set of pairs of vertices already aligned along the y-coordinate and A'_z is the sub-set of pairs of vertices already aligned along the z-coordinate.

4.2.4 Avoiding the collapse of the vertices

Another important problem to avoid is the collapse of vertices that are not end-points of the same edge. In particular we want to avoid that, during the alignment, vertices that appear in a pair of the set A all reach the same coordinate value. In this case we don't have an edge collapse but the resultant polycube is not manifold (we have a vertex collapse). To solve this problem we add to our model a particular constraint that creates a plane between the two vertices that are attracting each other. We insert this plane in the middle point of the imaginary edge that represents the Voronoi adjacency between the two end-points and we limit the movement of each vertex in the semi-plane delimited by this plane. In the next figure we show the situation in which the vertex A tries to align with the vertex B and in the middle point of their "attraction-edge" the plane p is inserted. The vertex A can only move towards B in the left semi-plane and the vertex B can only move towards A in the right semi-plane. The integer solver makes sure that their coordinates are not all the same.

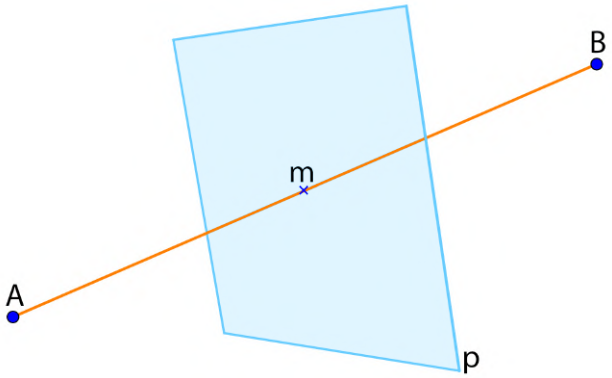


Figure 4.10: *An example of plane inserted between two vertices to avoid their collapse.*

Given the plane equation $ax+by+cz+d = 0$ and an imaginary edge between the vertices A and B, that we call e, we must create a plane passing through the middle point of e and having the same direction of e. We denote with $m = (x_m, y_m, z_m)$ the middle point of e and with $d = (d_1, d_2, d_3)$ the direction of e (it is orthogonal to the plane). The equation of the plane we are interested in is the following:

$$d_1x + d_2y + d_3z - (d_1x_m + d_2y_m + d_3z_m) = 0$$

We can now replace the x, y and z with the coordinate values of A to find in which semi-plane A is and to create two constraints: one to secure A in its semi-plane and one to secure B in the opposite semi-plane. So for each pair of vertices (v_i,v_j) of the set A previously described we perform this operation and we insert two constraints respectively for the vertices v_i and v_j:

$$\forall (i,j) \in A :$$

$$if\ (d_1\tilde{v}_i(x) + d_2\tilde{v}_i(y) + d_3\tilde{v}_i(z) - (d_1x_m + d_2y_m + d_3z_m) < 0)$$
$$then \begin{cases} d_1v_i(x) + d_2v_i(y) + d_3v_i(z) - (d_1x_m + d_2y_m + d_3z_m) < 0 \\ d_1v_j(x) + d_2v_j(y) + d_3v_j(z) - (d_1x_m + d_2y_m + d_3z_m) > 0 \end{cases}$$

$$else\ if\ (d_1\tilde{v}_i(x) + d_2\tilde{v}_i(y) + d_3\tilde{v}_i(z) - (d_1x_m + d_2y_m + d_3z_m) > 0)$$
$$then \begin{cases} d_1v_i(x) + d_2v_i(y) + d_3v_i(z) - (d_1x_m + d_2y_m + d_3z_m) > 0 \\ d_1v_j(x) + d_2v_j(y) + d_3v_j(z) - (d_1x_m + d_2y_m + d_3z_m) < 0 \end{cases}$$

where $\tilde{v}_i(x)$, $\tilde{v}_i(y)$ and $\tilde{v}_i(z)$ denote respectively the x-coordinate, the y-coordinate and the z-coordinate of the original vertex v_i while $v_i(x)$, $v_i(y)$ and $v_i(z)$ denote the variables of the problem relative to the x-coordinate, the y-coordinate and the z-coordinate of the vertex v_i in the solution.

4.2.5 Dummy vertices and edges

The last problem we want to solve is the following: if we look at the *Figure 4.11* of a test-polycube and its visualization in our interactive tool we can note that if the red vertices try to align we have no constraints that prevent the left part of the shape from going through itself.

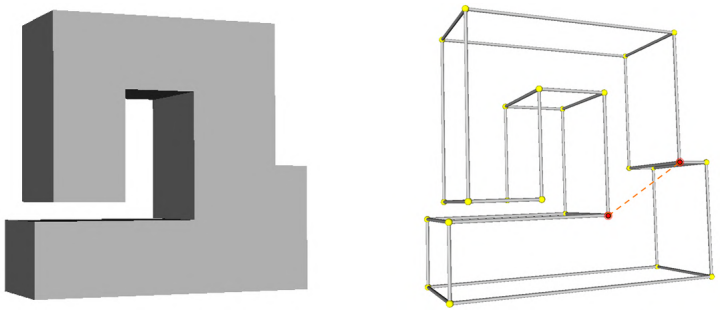


Figure 4.11: *An example of possible shape collapse.*

We now introduce the idea of *dummy vertices* and *dummy edges*. A dummy vertex is an imaginary vertex obtained by the intersection between the extension of an edge and a face (or another edge) of the polycube. A dummy edge, on the other hand, is an imaginary edge that connects an end-point of a real edge of the polycube

to the new found dummy vertex. If we compute all possible dummy vertices and edges in our input polycube we can impose, in the mathematical model, that the dummy edge length must be equal or bigger than 1. In this way we can avoid collapse between vertices and edge, vertices and faces, edges and edges. In the following figures we explain three different cases; in the first image the output of the optimization of the previous model without using the dummies, in the second the computation of the dummies in the original model (dummy edges are coloured in black) and in the third we show the optimization considering dummy constraints. As can be seen in the first picture in the upper part of the polycube we have edges collapse and in the left part of the polycube we have faces intersections. In the right image, instead, the red vertices (and relative edges and faces) are now correctly aligned.

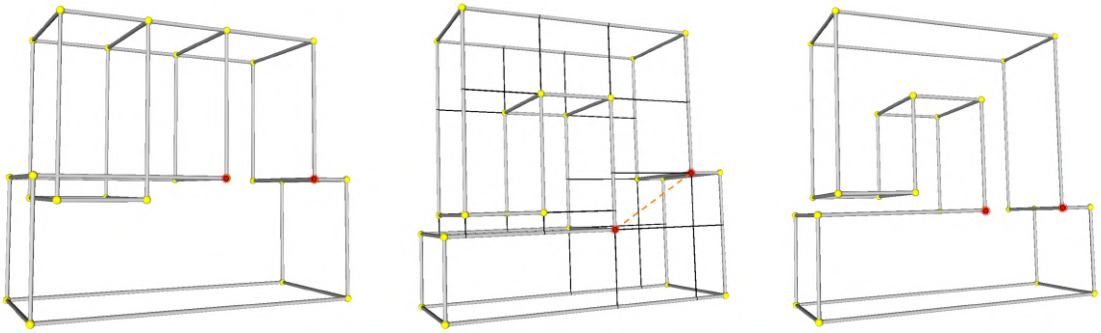


Figure 4.12: *Examples of dummies' problem.*

We define D as the set of dummy vertices. To formalize the described constraints we need to define two sets: DE and DF. DE is the set of dummy edges composed by pairs (v_i, v_j^d) when $v_i \in V$ (V is the set of the original vertices of the polycube) and $v_j^d \in D$. Now we can split the set DE in three sub-sets as follows: $DE_{\|x}$ is the sub-set of dummy edges parallel to the x-axis, $DE_{\|y}$ is the sub-set of dummy edges parallel to the y-axis and $DE_{\|z}$ is the sub-set of dummy edges parallel to the z-axis. It is now possible to secure the length of dummy edges greater than or equal to 1 with the following constraints:

$$\forall (i,j) \in DE_{\|x} \qquad if\ (v_i(x) > v_j^d(x))\ then\ v_i(x) - v_j^d(x) \geq 1\ else\ v_j^d(x) - v_i(x) \geq 1$$
$$\forall (i,j) \in DE_{\|y} \qquad if\ (v_i(y) > v_j^d(y))\ then\ v_i(y) - v_j^d(y) \geq 1\ else\ v_j^d(y) - v_i(y) \geq 1$$
$$\forall (i,j) \in DE_{\|z} \qquad if\ (v_i(z) > v_j^d(z))\ then\ v_i(z) - v_j^d(z) \geq 1\ else\ v_j^d(z) - v_i(z) \geq 1$$

These constraints on their own are not enough to obtain a solution. To solve the problem in a complete way we must attach the dummy vertices in the face or in the edge on which it lies in the original polycube (we can always use edges). To

do this we define the set DF composed of pairs (v_i^d,e_j) when $v_i^d \in D$ and $e_i \in E$ (see the *Section 4.2.1* for the definition of E and its sub-sets). We can now add the following constraints:

$$\forall (i,j) \in DF \quad \begin{cases} if\ e_j \in E_{\perp x}\ \ then\ \ v_i^d(x) = e_j(x) \\ if\ e_j \in E_{\perp y}\ \ then\ \ v_i^d(y) = e_j(y) \\ if\ e_j \in E_{\perp z}\ \ then\ \ v_i^d(z) = e_j(z) \end{cases}$$

where $e_j(x)$ denotes the x-coordinate of the e_j edge perpendicular to the x-axis, with $e_j(y)$ we denote the y-coordinate of the e_j edge perpendicular to the y-axis and with $e_j(z)$ we denote the z-coordinate of the e_j edge perpendicular to the z-axis.

4.2.6 Integer coordinates

We decided to solve our problem with an integer solver. This means that after the solution of the problem both vertices and dummy vertices have integer values of coordinates. This is important because in this way we can easily compute a quad-mesh starting from the optimized polycube. We impose a regular grid over the polycube and we obtain the quad-mesh in which the quad-layout is computed. To have this kind of result we add the last two constraints:

$$\begin{aligned} \forall i \in V \qquad & v_i(x) \in \mathbb{Z},\ v_i(y) \in \mathbb{Z},\ v_i(z) \in \mathbb{Z} \\ \forall j \in D \qquad & v_j^d(x) \in \mathbb{Z},\ v_j^d(y) \in \mathbb{Z},\ v_j^d(z) \in \mathbb{Z} \end{aligned}$$

where v_i is an original vertex of the polycube and v_j^d is a dummy vertex. In brackets we denote a specific coordinate.

4.3 The final model

We now report the complete mathematical model in extended version. Before doing this, we list all sets, sub-sets and other elements used in the problem formulation:

- V is the set of all vertices of the polycube.
- E is the set of all edges of the polycube. It can be split in $E_{\perp x}$, $E_{\perp y}$ and $E_{\perp z}$ (see *Section 4.2.1*) or in $E_{\parallel x}$, $E_{\parallel y}$ and $E_{\parallel z}$ (see *Section 4.2.2*).
- A is the set of pairs of vertices that could reach an alignment along one coordinate. It can be split in A_x, A_y and A_z (see *Section 4.1.2*).
- A' is the set of pairs of vertices already aligned along one coordinate. It can be split in A'_x, A'_y and A'_z (see *Section 4.2.3*).

- D is the set of dummy vertices of the polycube (see *Section 4.2.5*).
- DE is the set of dummy edges composed by pairs of original vertices and dummy vertices. It can be split in $DE_{\|x}$, $DE_{\|y}$ and $DE_{\|z}$ (see *Section 4.2.5*).
- DF is the set of pairs that are composed of a dummy vertex and an original edge of the polycube (see *Section 4.2.5*).
- $d = (d_1, d_2, d_3)$ is the direction of an imaginary edge that connects two vertices that we want aligned (see *Section 4.2.4).*
- $m = (x_m, y_m, z_m)$ is the middle point of an imaginary edge that connects two vertices that we want aligned (see *Section 4.2.4).*

The complete mathematical model is:

$$min\ e = \alpha\left(\sum_{i\in V}\left[(x_i-\bar{x_i})^2+(y_i-\bar{y_i})^2+(z_i-\bar{z_i})^2\right]\right)+\beta\left(\sum_{(i,j)\in A_x}(x_i-x_j)^2+\sum_{(i,j)\in A_y}(y_i-y_j)^2+\sum_{(i,j)\in A_z}(z_i-z_j)^2\right)$$

subject to:

$$\begin{array}{ll}\forall(i,j)\in E_{\perp x} & v_i(x)=v_j(x)\\ \forall(i,j)\in E_{\perp y} & v_i(y)=v_j(y)\\ \forall(i,j)\in E_{\perp z} & v_i(z)=v_j(z)\end{array}$$

$$\begin{array}{ll}\forall(i,j)\in E_{\|x} & if\ (v_i(x)>v_j(x))\ then\ v_i(x)-v_j(x)\geq 1\ else\ v_j(x)-v_i(x)\geq 1\\ \forall(i,j)\in E_{\|y} & if\ (v_i(y)>v_j(y))\ then\ v_i(y)-v_j(y)\geq 1\ else\ v_j(y)-v_i(y)\geq 1\\ \forall(i,j)\in E_{\|z} & if\ (v_i(z)>v_j(z))\ then\ v_i(z)-v_j(z)\geq 1\ else\ v_j(z)-v_i(z)\geq 1\end{array}$$

$$\begin{array}{ll}\forall(i,j)\in A'_x & v_i(x)=v_j(x)\\ \forall(i,j)\in A'_y & v_i(y)=v_j(y)\\ \forall(i,j)\in A'_z & v_i(z)=v_j(z)\end{array}$$

$$\begin{array}{l}\forall(i,j)\in A:\\ if\ (d_1\bar{v_i}(x)+d_2\bar{v_i}(y)+d_3\bar{v_i}(z)-(d_1x_m+d_2y_m+d_3z_m)<0)\\ then\ \begin{Bmatrix}d_1v_i(x)+d_2v_i(y)+d_3v_i(z)-(d_1x_m+d_2y_m+d_3z_m)<0\\ d_1v_j(x)+d_2v_j(y)+d_3v_j(z)-(d_1x_m+d_2y_m+d_3z_m)>0\end{Bmatrix}\\ else\ if\ (d_1\bar{v_i}(x)+d_2\bar{v_i}(y)+d_3\bar{v_i}(z)-(d_1x_m+d_2y_m+d_3z_m)>0)\\ then\ \begin{Bmatrix}d_1v_i(x)+d_2v_i(y)+d_3v_i(z)-(d_1x_m+d_2y_m+d_3z_m)>0\\ d_1v_j(x)+d_2v_j(y)+d_3v_j(z)-(d_1x_m+d_2y_m+d_3z_m)<0\end{Bmatrix}\end{array}$$

$$\begin{array}{ll}\forall(i,j)\in DE_{\|x} & if\ (v_i(x)>v_j^d(x))\ then\ v_i(x)-v_j^d(x)\geq 1\ else\ v_j^d(x)-v_i(x)\geq 1\\ \forall(i,j)\in DE_{\|y} & if\ (v_i(y)>v_j^d(y))\ then\ v_i(y)-v_j^d(y)\geq 1\ else\ v_j^d(y)-v_i(y)\geq 1\\ \forall(i,j)\in DE_{\|z} & if\ (v_i(z)>v_j^d(z))\ then\ v_i(z)-v_j^d(z)\geq 1\ else\ v_j^d(z)-v_i(z)\geq 1\end{array}$$

$$\forall(i,j)\in DF\ \begin{cases}if\ e_j\in E_{\perp x}\ then\ v_i^d(x)=e_j(x)\\ if\ e_j\in E_{\perp y}\ then\ v_i^d(y)=e_j(y)\\ if\ e_j\in E_{\perp z}\ then\ v_i^d(z)=e_j(z)\end{cases}$$

$$\begin{array}{ll}\forall i\in V & v_i(x)\in\mathbb{Z},\ v_i(y)\in\mathbb{Z},\ v_i(z)\in\mathbb{Z}\\ \forall j\in D & v_j^d(x)\in\mathbb{Z},\ v_j^d(y)\in\mathbb{Z},\ v_j^d(z)\in\mathbb{Z}\end{array}$$

5 Algorithms and Implementation

We have described, in the previous chapters, the problem we want to solve and the mathematical model that we want to use. In this chapter we will describe in a detailed way the algorithm that uses the mathematical model to perform the polycube optimization. We also illustrate the interactive tool that we developed to see and follow the progress of the algorithm step by step.

5.1 The final algorithm

Our final algorithm has the purpose to perform the polycube optimization described in *Chapter 3* by modeling and solving the mathematical model described in *Chapter 4*. Here is a pseudo-code version of the algorithm:

```
   Input  : A polycube P
   Output: A quad-mesh Q of the optimized polycube P'
 1 P ← loadPolyCube(model.obj);
 2 (α, β) ← readValues();
 3 repeat
 4 |   D ← computeDummy(P);
 5 |   VD ← computeVoronoiDiagram(P);
 6 |   A ← computeVoronoiAdjacencies(P, VD);
 7 |   M ← createModel(P, A);
 8 |   P' ← optimize(M, α, β);
 9 until (convergence OR max_step);
10 P' ← finalOptimization(P');
11 Q ← computeQuadMesh(P');
```

We now explain the algorithm we have developed, step by step, by referencing the pseudo-code. We use the bunny polycube model as an example to better explain the different steps of the algorithm with some references to the implementation and the library we used. Each screenshot in this section is made by using our interactive tool (described later).

Input & Output Our algorithm takes a polycube model as input and gives the quad-mesh of the optimized polycube as output (with integer coordinates). In this example we use the Bunny model and its polycube.

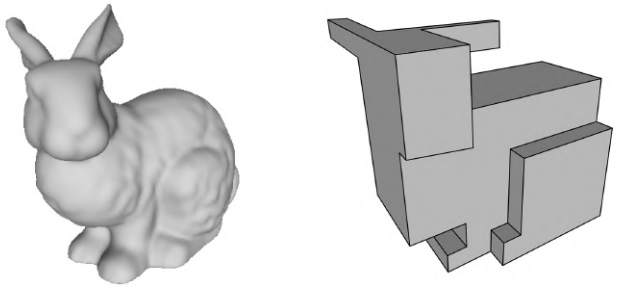


Figure 5.1: *The Bunny model and its polycube representation obtained by [10].*

loadPolyCube *(line 1)* This function loads a polycube model in .obj or .ply format. The polycube is loaded in a DCEL data structure and navigated to extract its corners and edges and to compute its faces. Polycubes in input have a number of triangles which varies from a few thousand up to hundreds of thousand. We only extract the essential information through this function. In our algorithm, we only use corners, edges and faces (a face is an axis-aligned polygon that is not necessarily convex) to decrease a big part of the computational complexity.

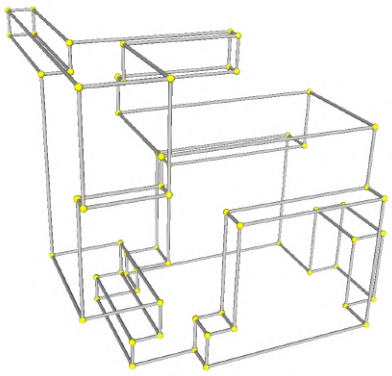


Figure 5.2: *The Bunny polycube in our interactive tool, after the essential information is extracted.*

readValues *(line 2)* As explained in *Section 4.1* we have split the objective function in two portions: one to preserve the topology of the model (E_{shape}) and one to perform the alignment (E_{align}). We have also inserted a system of weights to confer different importance to the different portions of the function. We denote the scalar number multiplied for the E_{shape} with α and the scalar number multiplied for the E_{align} with β. This function allows the user to insert the α-value and the β-value before launching the algorithm. We inserted two sliders in our tool to give the user the possibility to set these values.

computeDummy *(line 4)* This function has the task to compute dummy vertices and dummy edges (see *Section 4.2.5)*. For each edge of the polycube we trace its extension until it intersects a face or another edge of the polycube. For each intersection we create a new dummy vertex and we take note of its coordinates and of the edge/face it has intersect. The function takes the polycube as input and returns the set A with the described information. All this information is used in the construction of the model (to impose the relative constraints). In the next figure we show the dummy vertices and the dummy edges in black.

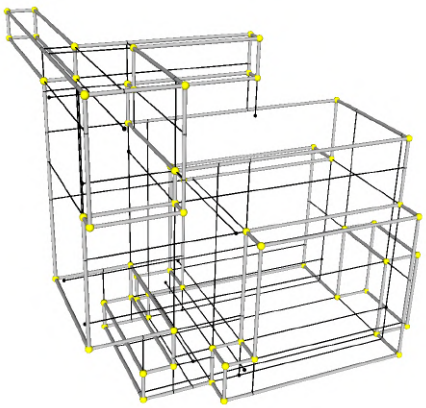


Figure 5.3: *The Bunny polycube in our interactive tool, after the dummy edges and vertices computation.*

computeVoronoiDiagram *(line 5)* This function computes the Voronoi diagram using the vertices of the input polycube as sites. To compute the diagram we use the Voro++ library [12]. Voro++ is one of the most famous open source software libraries written in C++ for the computation of the Voronoi diagram and it has an interface that can be used to carry out many types of Voronoi computations. We integrated its code in our interactive tool and, particularly, in this function. The *computeVoronoiDiagram* function takes the polycube as input and returns the Voronoi diagram as a POV-Ray file. By using a parser for this file, we read the

vertices of the Voronoi cells and their boundaries. In the next figure the Voronoi cells built around the vertices of the polycube are shown in blue.

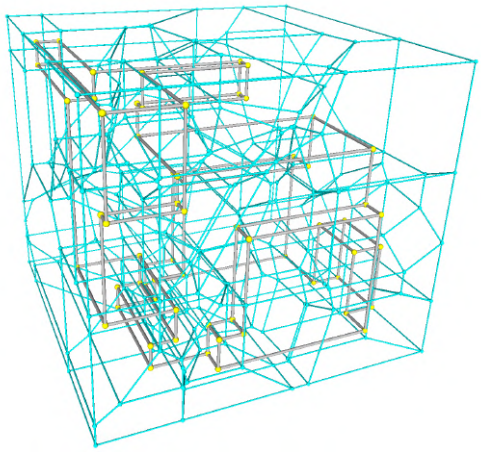


Figure 5.4: *The Bunny polycube in our interactive tool, after the Voronoi diagram computation.*

computeVoronoiAdjacencies *(line 6)* Using the Voronoi diagram computed in the previous step, this function computes all the Voronoi adjacencies of the vertices of the polycube. The function takes the polycube and the Voronoi diagram as input and returns the set A. This set is used for the construction of the E_{align} in the objective function of the mathematical model, as explained in the *Section 4.1.2*, and for the imposition of the constraints as explained in *Section 4.2.3* and *Section 4.2.4*. In the next figure we show the temporarily ignored adjacencies or the already aligned pairs of vertices in orange, the adjacencies used to try the alignment along the x-axis in red, the adjacencies used to try the alignment along the y-axis in blue and, lastly, the adjacencies used to try the alignment along the z-axis in green.

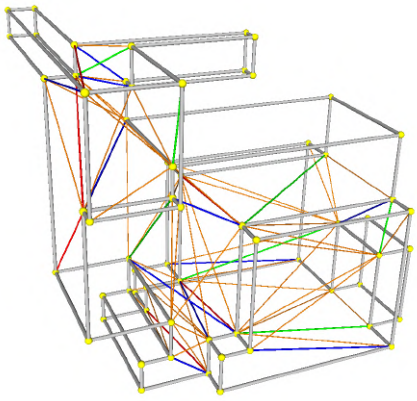


Figure 5.5: *The Bunny polycube in our interactive tool, after the Voronoi adjacencies computation.*

createModel & optimize *(line 7 and 8)* The *createModel* function has the task to automatically create the mathematical model with the objective function and the constraints described in *Chapter 4*, while the *optimize* function has the task to solve the model based on the values of α and β as explained in *Section 4.1*. To create and solve the mathematical model we used the Gurobi Optimizer 6.0 [11]. The Gurobi Optimizer is a state of the art solver for mathematical programming. The solver in the Gurobi Optimizer includes a set of different solvers for linear programming, quadratic programming, mixed-integer linear programming, quadratically constrained programming and many others. Gurobi supports interfaces for a variety of programming and modeling languages like C, C++, Java, MATLAB and others. We integrated this library (with a free license for students) in our interactive tool and in particular in this function that takes the polycube and the set A computed in the previous step as input and returns an optimized polycube. Through the Gurobi's instruction set we can easily make the objective function and the constraints (expressed in extended form and not in the matrix form) of the model and, with a single instruction, we can optimize it (for more information about the code see the *Appendix A*). In the next figure we show the first step of the optimization of the polycube used as an example model (remember that this portion of the algorithm is repeated several times until the convergence or the achievement of the maximum number of allowed steps is reached).

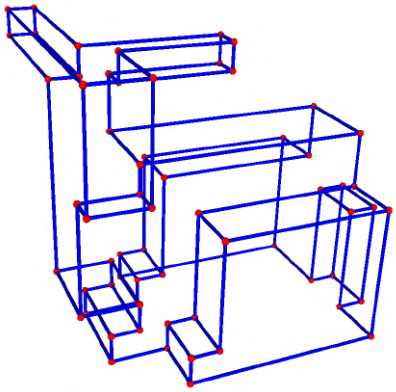


Figure 5.6: *The Bunny polycube in our interactive tool, after the first step of optimization.*

finalOptimization *(line 10)* After the repeated optimization of the model we obtain an optimized polycube as output. This polycube is topologically equivalent to the input polycube and has a greater number of aligned vertices. With the *finalOptimization* function we want to perform the last optimization step with a slightly different formulation of the problem. In particular, the aim of this function is to make the obtained polycube as similar as possible to the original shape but without losing any alignment. To reach this goal we call the *optimize* function for the last

time, passing the values of $\alpha = 1$ and $\beta = 0$. In this way the objective function of the mathematical model is transformed as follows, without weights and without the E_{align} portion:

$$min\ e = \sum_{i \in V} \left[(x_i - \tilde{x}_i)^2 + (y_i - \tilde{y}_i)^2 + (z_i - \tilde{z}_i)^2 \right]$$

In the next figure the bunny model after two steps of classical optimization and the *finalOptimization* step is shown. The resultant polycube is the final polycube.

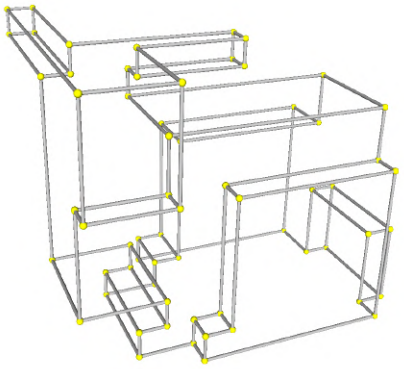


Figure 5.7: *The Bunny polycube in our interactive tool, after the final optimization step.*

computeQuadMesh *(line 11)* This function takes the final optimized polycube as input and returns the relative quad-mesh. The quad-mesh is obtained easily by superimposing a regular grid over the polycube and checking if each resultant quad is internal or external to the model. The resultant quad-mesh is saved in an .obj file and it is ready for the computation of the relative quad-layout and for the quad count.

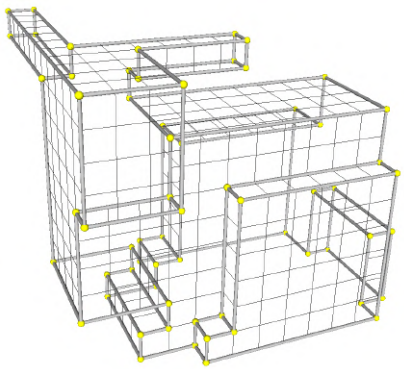


Figure 5.8: *The Bunny polycube in our interactive tool, after the quad-mesh computation.*

We have explained all the functions of our algorithm. Note that the functions *computeDummy*, *computeVoronoiDiagram*, *computeVoronoiAdjacencies*, *createModel* and *optimize* are reapeated until the verification of a logical condition:

$$(convergence\ OR\ max\_step)$$

On the occurrence of one of these conditions the optimization part of the algorithm ends and the final steps are performed. The *convergence* condition is computed by counting, in each optimization step, the number of obtained alignments (we denote it with n_a). When the n_a at the step i is equal to 0 we have achieved the convergence and the computation goes to the final steps. The *max_step* condition, on the other hand, is true when the number of allowed maximum iterations (previously decided) is reached. In this case the computation goes to the final step but the algorithm doesn't return a valid optimized polycube. In our tests, this condition was never met.

5.2 The interactive tool

We now shortly describe the interactive tool we have realized to execute and analyze the algorithm progress step by step. In the following figure the User Interface with a series of commands for the manipulation of the algorithm is shown:

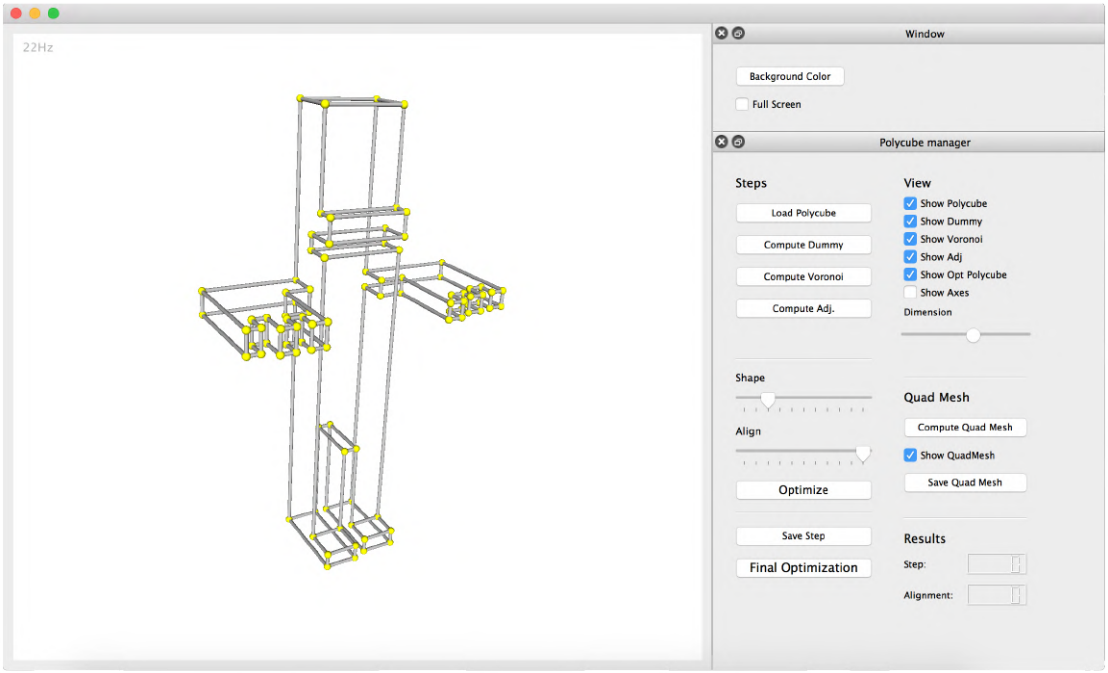


Figure 5.9: *Screenshot of the UI of our interactive tool.*

The tool and the algorithm are developed with the C++ language and the QT 5.3.1 library, using the IDE QTCreator 3.1.2 (open-source). The tool presents a big canvas in the left where the user can see the polycube (in a stylized version with only vertices and edges showing) and all transformations and auxiliary operations that the algorithm does (Voronoi diagram, dummy vertices and edges etc.). The algorithm can be run step by step by using a series of buttons and check-buttons that allows the user to decide when to perform the step and what to see about the current step and the previous steps.

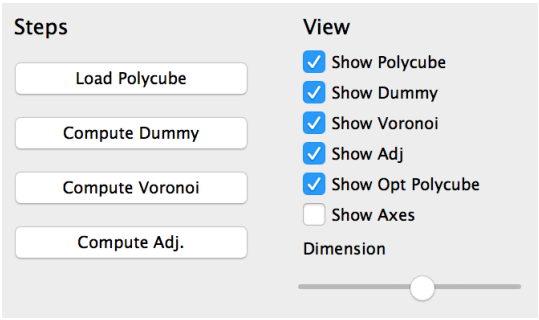


Figure 5.10: *Commands to run the algorithm step by step.*

With the optimization panel the user can choose the values about the shape preservation and the alignment (α and β values explained in *Section 4.1.* The user can later continue with the optimization step and observe the results. Two displays show the optimization step and the numbers of alignments performed in the current step. If the user is satisfied with the current optimization step he can save it and decide whether to perform another optimization step or to proceed with the final optimization.

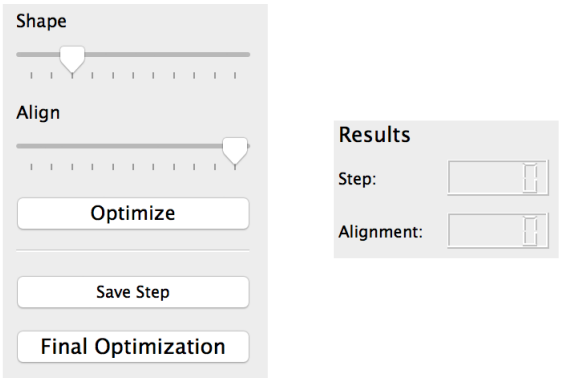


Figure 5.11: *Commands to run the optimization steps of the algorithm.*

The tool allows the user to guide the algorithm in each of its steps but it also allows him to compute the entire algorithm in a single step. The user can load the

polycube, select the desired values of *shape* and *alignment* and click the *finalOptimization* button to obtain the final result directly.

When the algorithm is finished (but also in the intermediate steps) it is possible to compute the quad-mesh of the resultant polycube and to save it in an .obj file through the dedicated panel.

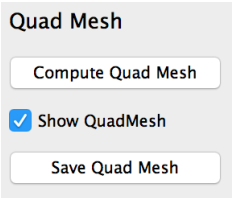


Figure 5.12: *Commands to compute the quad-mesh of the optimized polycube.*

6 Results

After explaining the problem we want to solve, the mathematical model that we modeled and the algorithm that uses it, we want to show some results with real models. The tool has been tested on several input polycubes obtained from different types of 3D models including classical 3D models, 3D scans, engineered shapes and already optimized polycubes (the "Bimba" model case). We report the results about some of them and, for each model, we report the results of the algorithm executed step by step. For each test we show the original model, the quad-layout computed on the initial polycube (obtained by the PolyCut algorithm) and the quad-layout computed on the polycube optimized by using our algorithm. Through a table we report the following data for each step of all tests: *step* is the step number (when the step number is "f" we mean the final optimization step), E_{shape} is the value of the α number used as a weight for the shape preservation in the objective function, E_{align} is the value of β number used as a weight for the alignment portion of the objective function, $\#\ align.$ is the number of alignments performed at the step i and *time* is the time used by the solver to optimize the model at the step i (in seconds). In the lower part of the table we report the data about the final results: $\#\ orig.\ quad$ is the number of original quads in the input polycube, $\#\ opt.\ quad$ is the number of quads computed in the optimized polycube, $tot.\ align.$ is the total number of the alignments performed during the optimization, $\%\ red.$ is the percentage of reduction of the quad number from the input polycube to the optimized polycube and $tot.\ time$ is the overall optimization time (in seconds).

The tests have been performed on a MacBook Pro 2015 with an Intel Core i5 dual-core with 2,7 GHz and 8 GB of RAM, so the time reported in the following tables refers to this model.

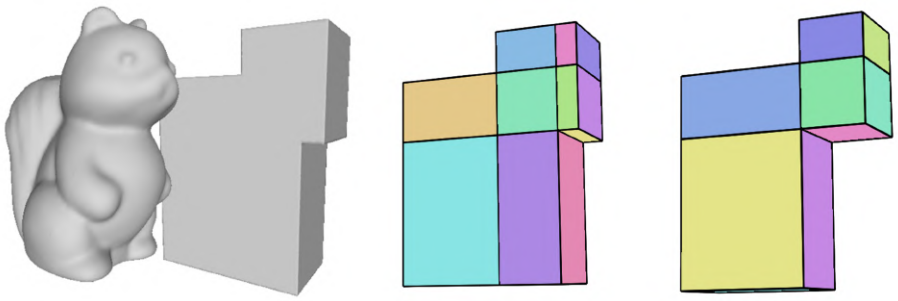


Figure 6.1: *The Squirrel model and its polycube (left), the initial quad-layout (center) and the optimized quad-layout (right).*

Squirrel				
step	E_{shape}	E_{align}	*# align.*	*time (sec)*
1	0.1	1	2	0.004688
2	0.1	1	0	0.004093
f	1	0	/	0.003679
results				
# orig. quad	*# opt. quad*	*tot. align.*	*% red.*	*tot. time (sec)*
26	18	2	30.77	0.012460

Table 6.1: *Results of the optimization of the Squirrel polycube.*

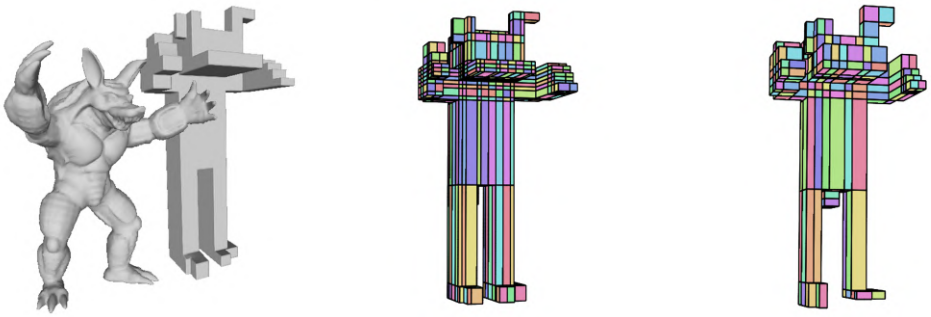


Figure 6.2: *The Armadillo model and its polycube (left), the initial quad-layout (center) and the optimized quad-layout (right).*

Armadillo				
step	E_{shape}	E_{align}	*# align.*	*time (sec)*
1	0.2	1	13	0.841866
2	0.2	1	5	0.172275
3	0.2	1	0	0.017098
f	1	0	/	1.326690
results				
# orig. quad	*# opt. quad*	*tot. align.*	*% red.*	*tot. time (sec)*
952	438	18	53.99	2.357929

Table 6.2: *Results of the optimization of the Armadillo polycube.*

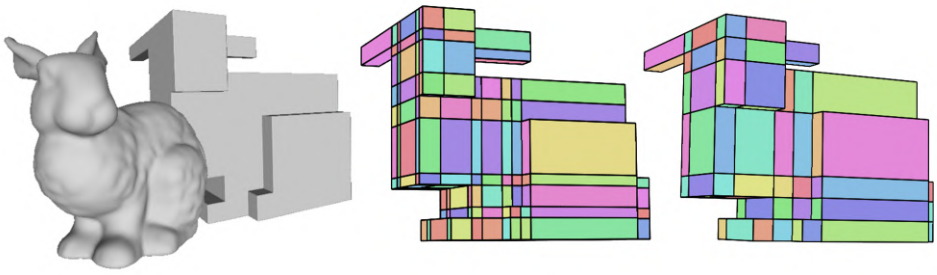


Figure 6.3: *The Bunny model and its polycube (left), the initial quad-layout (center) and the optimized quad-layout (right).*

Bunny				
step	E_{shape}	E_{align}	*# align.*	*time (sec)*
1	0.1	1	7	0.064626
2	0.1	1	3	0.033247
3	0.1	1	0	0.032253
f	1	0	/	0.105301
results				
# orig. quad	*# opt. quad*	*tot. align.*	*% red.*	*tot. time (sec)*
310	156	10	49.68	0.235427

Table 6.3: *Results of the optimization of the Bunny polycube.*

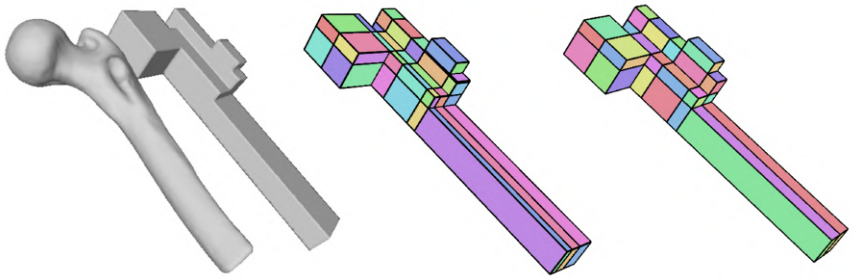


Figure 6.4: *The Bone model and its polycube (left), the initial quad-layout (center) and the optimized quad-layout (right).*

Bone				
step	E_{shape}	E_{align}	*# align.*	*time (sec)*
1	0.1	1	4	0.065363
2	0.1	1	0	0.038630
f	1	0	/	0.043966
results				
# orig. quad	*# opt. quad*	*tot. align.*	*% red.*	*tot. time (sec)*
102	80	4	21.57	0.147959

Table 6.4: *Results of the optimization of the Bone polycube.*

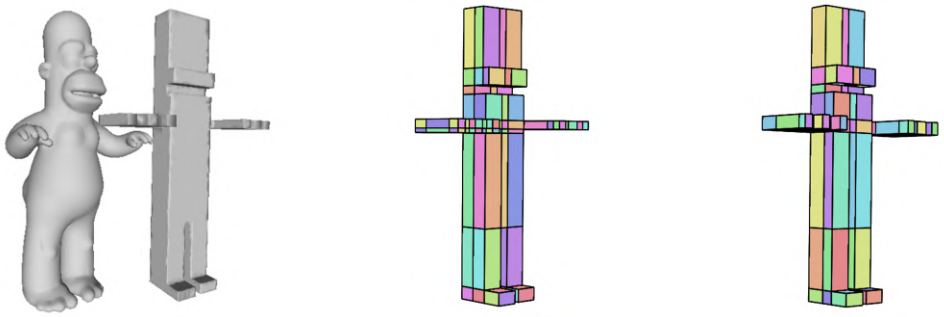


Figure 6.5: *The Homer model and its polycube (left), the initial quad-layout (center) and the optimized quad-layout (right).*

Homer				
step	E_{shape}	E_{align}	*# align.*	*time (sec)*
1	0.1	1	8	0.950295
2	0.1	1	0	0.028333
f	1	0	/	13.77570
results				
# orig. quad	*# opt. quad*	*tot. align.*	*% red.*	*tot. time (sec)*
260	202	8	22.31	14.754328

Table 6.5: *Results of the optimization of the Homer polycube.*

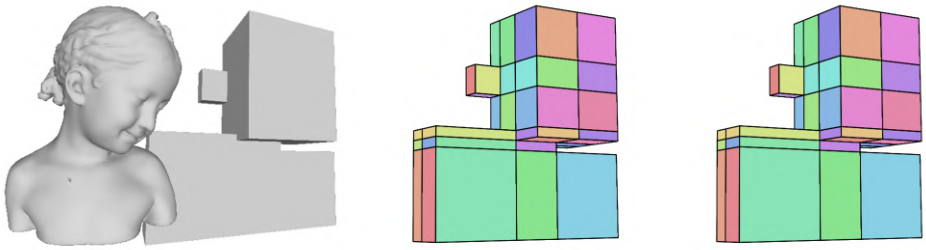


Figure 6.6: *The Bimba model and its polycube (left), the initial quad-layout (center) and the optimized quad-layout (right).*

Bimba				
step	E_{shape}	E_{align}	*# align.*	*time (sec)*
1	0.1	1	0*	0.054505
f	1	0	/	0.030028
results				
# orig. quad	*# opt. quad*	*tot. align.*	*% red.*	*tot. time (sec)*
84	84	0*	0.0*	0.084533

Table 6.6: *Results of the optimization of the Bimba polycube.*

* The Bimba polycube was already optimal before the algorithm application (all possible aligments were already done).

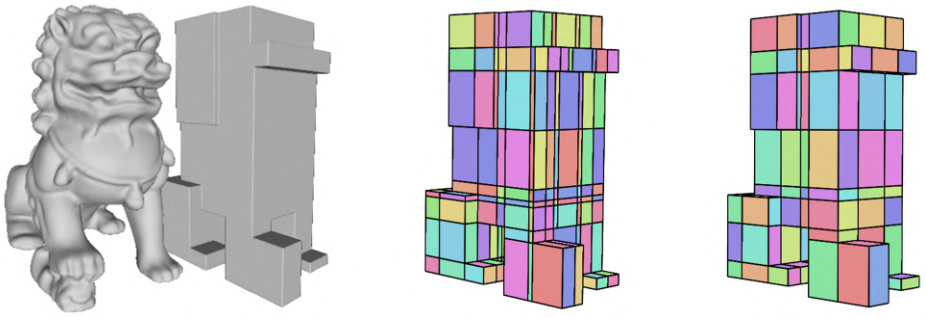


Figure 6.7: *The Dragon model and its polycube (left), the initial quad-layout (center) and the optimized quad-layout (right).*

Dragon				
step	E_{shape}	E_{align}	*# align.*	*time (sec)*
1	0.2	1	5	0.254485
2	0.2	1	2	0.101568
3	0.2	1	1	0.044433
4	0.2	1	0	0.026913
f	1	0	/	0.062079
results				
# orig. quad	*# opt. quad*	*tot. align.*	*% red.*	*tot. time (sec)*
396	210	8	46.97	0.489478

Table 6.7: *Results of the optimization of the Dragon polycube.*

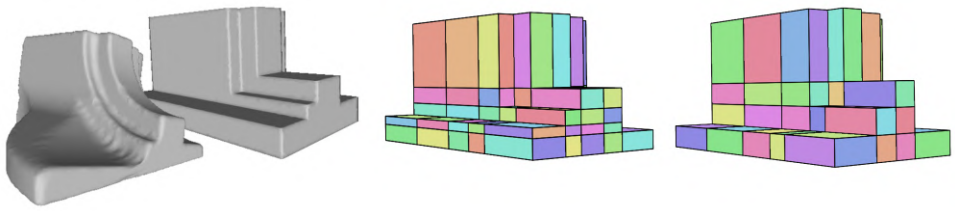


Figure 6.8: *The Shape model and its polycube (left), the initial quad-layout (center) and the optimized quad-layout (right).*

Shape				
step	E_{shape}	E_{align}	*# align.*	*time (sec)*
1	0.1	1	2	0.388350
2	0.1	1	0	0.006715
f	1	0	/	0.045978
results				
# orig. quad	*# opt. quad*	*tot. align.*	*% red.*	*tot. time (sec)*
150	112	2	25.33	0.441043

Table 6.8: *Results of the optimization of the Shape polycube.*

We show here a few significant portions of some polycubes in which the alignment of the vertices and the quads' number reduction is quite clear. In particular, in the next figures, the Dragon's face, the Armadillo's back and the Bunny's tail are shown (in the next chapter another example relative to Homer's hand is shown in *Figure 7.1*).

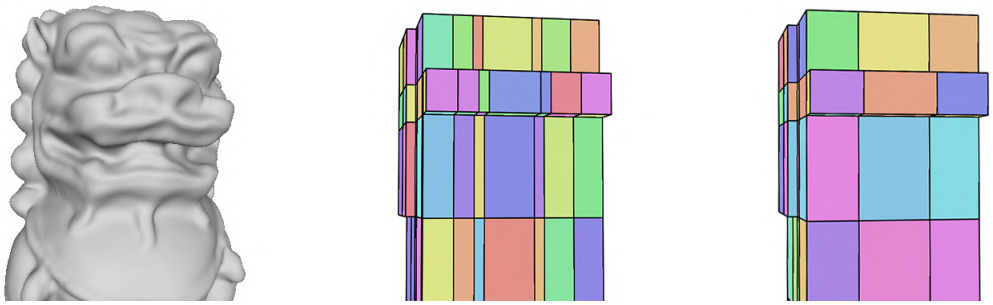


Figure 6.9: *The Dragon's face in the original model (left), in the initial polycube (center) and in the optimized polycube (right).*

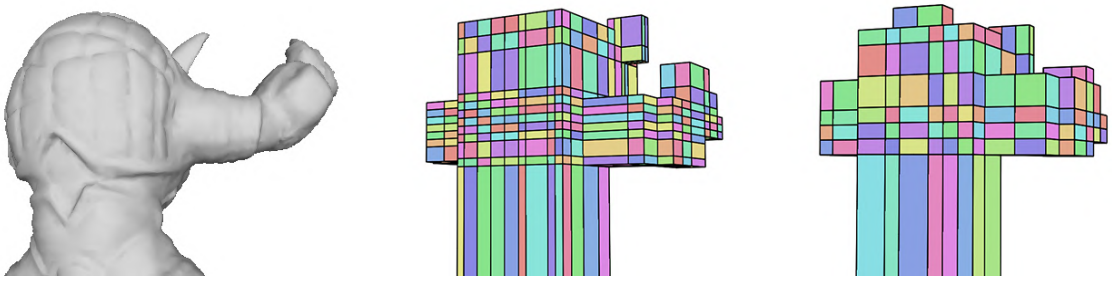


Figure 6.10: *The Armadillo's back in the original model (left), in the initial polycube (center) and in the optimized polycube (right).*

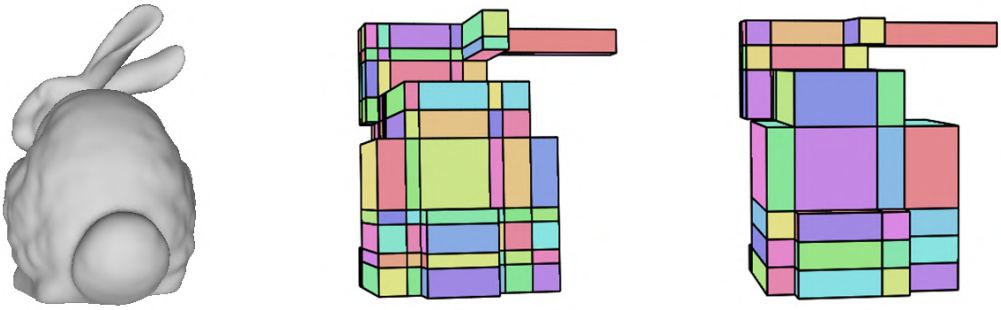
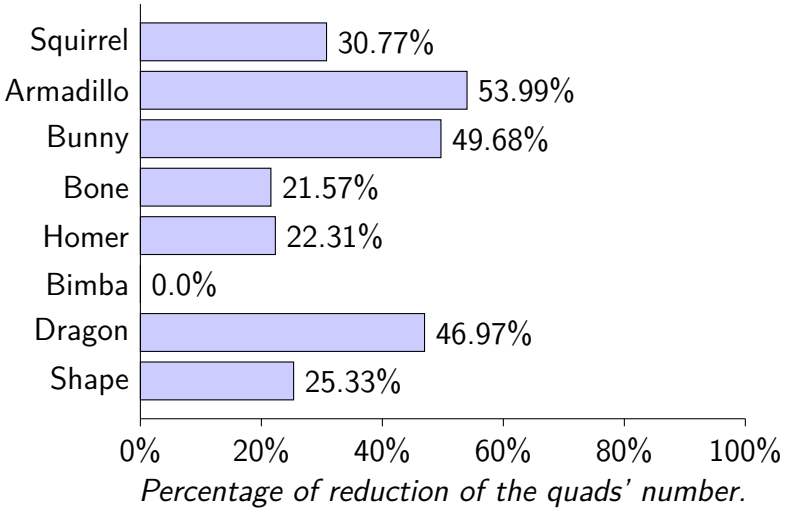
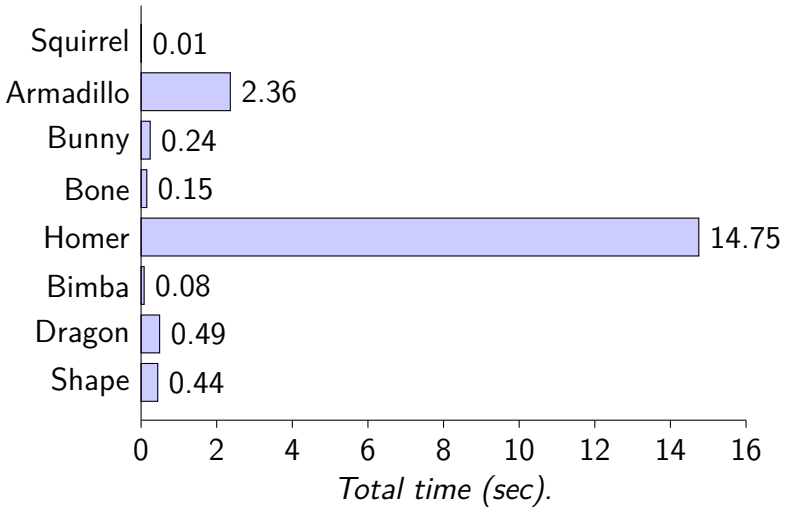


Figure 6.11: *The Bunny's tail in the original model (left), in the initial polycube (center) and in the optimized polycube (right).*

We now analyse the percentage of reduction of the quads' number in the quad-layouts (the "% *red.*" column in the tables). If we observe the quads' number in the original polycube and compare it to its optimized version (the "# *orig. quad*" and the "# *opt. quad*" columns in the previous tables) we note that the percentage of reduction is related to the initial number of misalignments in the original shape of the polycube. In the following histogram the percentage of the quads' number reduction is shown:



The execution time is another good result that we have achieved. The total time used by the algorithm to perform the optimization (the "*tot. time*" column in the previous tables) ranges from a fraction of a second to a few seconds (an exception is the Homer polycube that requires about 14 seconds). In the following histogram the total time required for the optimization (in seconds) is shown:



7

Conclusions

The purpose of this thesis was to optimize a polycube to obtain an optimized quad-layout. We have presented a polycube optimization algorithm based on the construction and the solution of a mathematical model in an iterative way. Our approach generates, in most cases, an optimized polycube that can be transformed in an optimized quad-layout. The obtained quad-layout can be easily mapped to the original model (the one from which the initial polycube has been extracted). We have obtained good results, as it is shown in the previous chapter, with a good trade-off between the shape preservation and the vertices/edges alignment. The percentage of reduction of the quads number in the quad-layout (obtained from the optimized polycube) depends on the number of misalignments of the original polycube. Higher the number of misaligned vertices and edges in the original polycube, higher the number of possible alignments that the algorithm can perform. In the previous chapter a histogram shows a summary of the percentages of reduction of the number of quads. In the following figure we show a particular case where the number of misalignments in the Homer hand's portion of the polycube decreases in a consistent way.

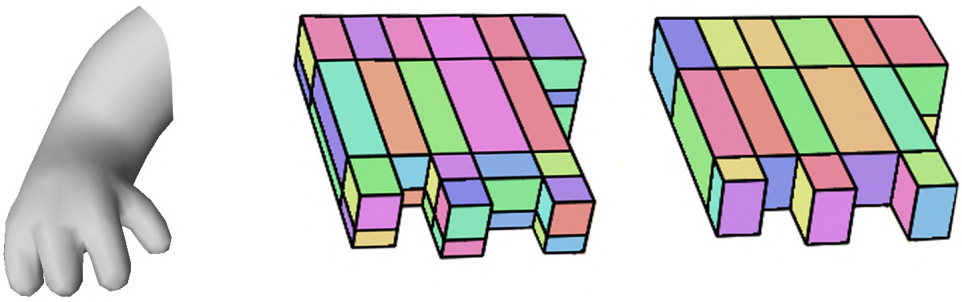


Figure 7.1: *The Homer's hand in the original model (left), in the initial polycube (center) and in the optimized polycube (right).*

Remember that the algorithm creates and solves the mathematical model several times for each input polycube. Most of the time employed by the algorithm to perform the complete optimization is used by the solver to compute the solution of the mathematical model. Working with polycubes means to work with a very small set of vertices, edges and faces and, for this reason, the steps of the algorithm (apart from the optimization steps) are computationally inexpensive. The overall time of the optimization with our algorithm ranges from a fraction of a second to a few seconds. An exception is the Homer's model that requires about 14 seconds to be optimized. We can note, by analysing the tables in the previous chapter, that the longest step is the "final optimization step". In the previous chapter a histogram shows a summary of the execution times for the different models.

Analysing the test results, the mathematical model we have built is elegant, robust and efficient to reach our goal and it can be solved in an easy way thanks to its linear constraints. We are satisfied with the achieved results. The purpose of this work has been reached with good results and good execution times (with some limitations that we will explain in the next chapter). Obviously the work is not finished. It can be continued in several ways with different application fields. In the next chapter we will describe some of these and we will show some problems of the algorithm and some possible solutions.

8 Future Works

Our method generates coarse quad-layouts of good quality for a wide set of different shapes. We observed that, in a few cases, a higher level of coarseness could be reached.

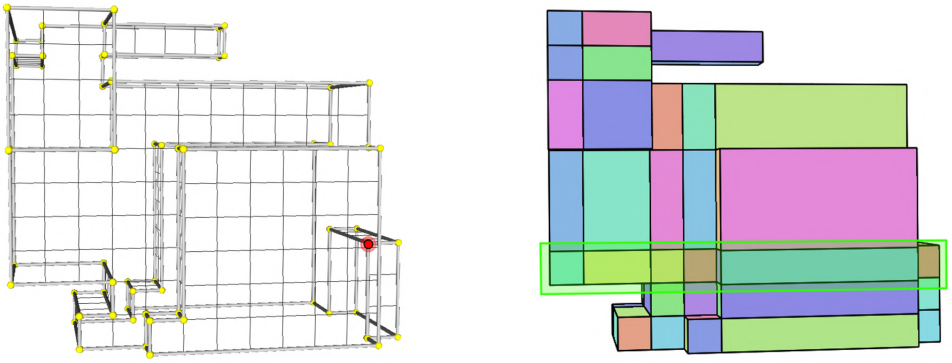


Figure 8.1: *Example of a misalignment omitted by the algorithm*

If we observe the left image of the *Figure 8.1* we can see that the red vertex is not aligned with any other vertex along the y-coordinate. This misalignment creates a set of redundant quads (highlighted by the green box of the right image) that could be deleted if the red vertex was able to perform an alignment along the y-coordinate. In *Figure 8.2* a set of possible alignments for the red vertex is shown. The problem is that in the Voronoi diagram of the vertices of the polycubes the cell of the red vertex is not adjacent with any of the cells of the green vertices. For this reason, in the mathematical model, there are no constraints or portion of the objective function that forces the alignment of the red vertex with one of the green vertices along the y-coordinate. The next step could be to find a modified

version of the Voronoi diagram that allows the algorithm to reach as many pairs of alignments as possible.

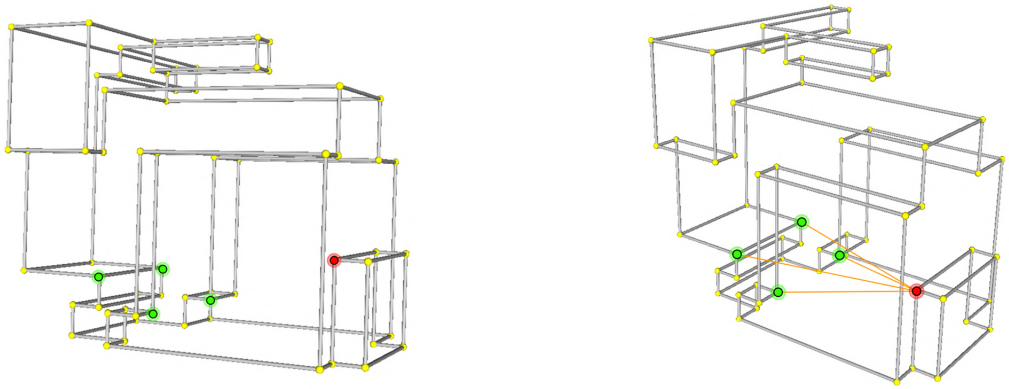


Figure 8.2: *Example of possible alignments for the red vertex*

Once solved the previous problem, we would like to test our algorithm (with the appropriate changes) in the hex-meshing field. As it is explained in [6], finding a coarse "hex-layout" (the equivalent of the quad-layout in the hex-meshes) is a new hot research topic. We have explained in *Section 2.2* that one of the applications of polycubes is the generation of hex-meshes. For this reason we would like to test if generating a hex-mesh by using an optimized polycube makes a hex-layout better than those generated through the original polycube.

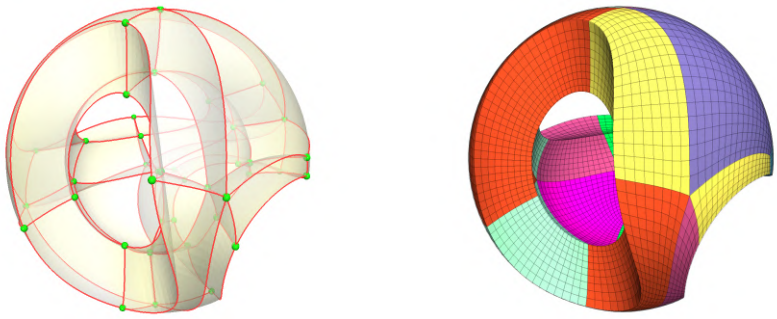


Figure 8.3: *Example of hex-mesh with the hex-layout taken from [6]*

A

Gurobi Optimizer

Gurobi Optimizer 6.0 is a state of the art solver for mathematical programming. The solver in the Gurobi Optimizer includes a set of different solvers for linear programming, quadratic programming, mixed-integer linear programming, quadratically constrained programming and many others. Gurobi supports interfaces for a variety of programming and modeling languages like C, C++, Java, MATLAB and others. We integrated this library (with a free license for students) in our C++ interactive tool. Through the Gurobi's instruction set we can easily make the objective function and the constraints of the model and, with a single instruction, we can optimize it. In this section we want to show some examples of code used to create the model and to solve it.

First of all we must create an environment where we are able to create and solve the model. To do this we declare a variable of type `GRBEnv` that representss the environment and a variable of type `GRBModel` that represents the model. Note that the model, during its creation, takes a reference to the environment in which it must be solved.

```
GRBEnv env = GRBEnv();
GRBModel model = GRBModel(env);
```

Listing A.1: Creation of the environment and creation of the model.

Next we must create the variables of the problem. We have three variables for each vertex (one for each coordinate) and three variables for each dummy vertex of the polycube (see *Section 4.2.5*). We must declare variables of type `GRBVar` and add them to the problem using the `addVar()` function. This function takes, for each variable, five parameters: `min_v` and `max_v` represent respectively the lower

value and the highest value that the variable can assume in the solution (set with the bounding box values), `obj_v` (optional) represents the initial value that the variable must assume in the objective function, `type` is the variable type that we set to `GRB.INTEGER`, and finally the `name` parameter is a string that denotes a symbolic name for the variable.

```
GRBVar v;
v = model.addVar(min_v,max_v,obj_v,type,name);
```

Listing A.2: Addition of a variable in the model.

We are ready to create the model. The first step is to create the objective function. To do this we have to create a variable of type `GRBQuadExpr` that represents a quadratic expression (we have a mathematical problem with a quadratic objective function and linear constraints). Note that the objective function is expressed in a simple and natural way. After creating the function we must add it to our model through the `setObjective()` function and update the model through the `update()` function.

```
GRBQuadExpr obj;
obj = ((v1+v2)*(v1+v2)+(v3-v4) ... );
model.setObjective(obj);
model.update();
```

Listing A.3: Example of creation of the objective function.

Adding the constraints is as simple as the creation of the objective function. Through the `addConstr()` function we can add constraints of "equality" or of "lower-than" in an easy way. After the insertion of the constraints we have to update the model with the `update()` function.

```
model.addConstr(v1-v2 <= 1);
model.addConstr(v1*v3 == 0);
model.addConstr(v4 == v5);
...
model.update();
```

Listing A.4: Examples of constraints addition.

The next and final step is the optimization. Doing this through the Gurobi solver is very easy. We just have to call the `optimize()` function and the solver

returns, in the variables of the problem previously added to the model, the solution of the optimization.

```
model.optimize();
```

Listing A.5: Optimization step.

The Gurobi solver has a system of exceptions to handle the problems during the optimization. It is important to create the environment, create the model with its objective function and its constraints, and to optimize it using a *try-catch* construct as the next listing shows:

```
try
    {
        //creation of the environment
        //creation of the model
        //creation of the objective function
        //creation of the constraints
        //optimization
    }

catch(GRBException e)
    {
        std::cout << "Error" << e.getErrorCode();
    }

catch(...)
    {
        std::cout << "Exception during the optimization";
    }
```

Listing A.6: Structure of the code.

Bibliography

[1] David Bommes, Timm Lempfer, and Leif Kobbelt. Global structure optimization of quadrilateral meshes. In *Computer Graphics Forum*, volume 30, pages 375–384. Wiley Online Library, 2011.

[2] David Bommes, Bruno Lévy, Nico Pietroni, Enrico Puppo, Claudio Silva, Marco Tarini, and Denis Zorin. Quad-mesh generation and processing: A survey. In *Computer Graphics Forum*, volume 32, pages 51–76. Wiley Online Library, 2013.

[3] Marcel Campen, David Bommes, and Leif Kobbelt. Dual loops meshing: quality quad layouts on manifolds. *ACM Transactions on Graphics (TOG)*, 31(4):110, 2012.

[4] Marcel Campen and Leif Kobbelt. Dual strip weaving: interactive design of quad layouts using elastica strips. *ACM Transactions on Graphics (TOG)*, 33(6):183, 2014.

[5] Alecu Felician. Blender institute–the institute for open 3d projects. *Open Source Science Journal*, 2(1):36–45, 2010.

[6] Xifeng Gao, Zhigang Deng, and Guoning Chen. Hex-mesh re-parameterization from aligned base domains supplemental material. *SIGGRAPH*, 2015.

[7] James Gregson, Alla Sheffer, and Eugene Zhang. All-hex mesh generation via volumetric polycube deformation. In *Computer graphics forum*, volume 30, pages 1407–1416. Wiley Online Library, 2011.

[8] Ying He, Hongyu Wang, Chi-Wing Fu, and Hong Qin. A divide-and-conquer approach for automatic polycube map construction. *Computers & Graphics*, 33(3):369–380, 2009.

[9] Juncong Lin, Xiaogang Jin, Zhengwen Fan, and Charlie CL Wang. Automatic polycube-maps. In *Advances in Geometric Modeling and Processing*, pages 3–16. Springer, 2008.

[10] Marco Livesu, Nicholas Vining, Alla Sheffer, James Gregson, and Riccardo Scateni. Polycut: monotone graph-cuts for polycube base-complex construction. *ACM Transactions on Graphics (TOG)*, 32(6):171, 2013.

[11] Gurobi Optimization et al. Gurobi optimizer reference manual. *URL: http://www. gurobi. com*, 2012.

[12] Chris Rycroft. Voro++: A three-dimensional voronoi cell library in c++. *Lawrence Berkeley National Laboratory*, 2009.

[13] Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. Polycube-maps. *ACM Transactions on Graphics (TOG)*, 23(3):853–860, 2004.

[14] Marco Tarini, Enrico Puppo, Daniele Panozzo, Nico Pietroni, and Paolo Cignoni. Simple quad domains for field aligned mesh parametrization. *ACM Transactions on Graphics (TOG)*, 30(6):142, 2011.

[15] Francesco Usai, Marco Livesu, Enrico Puppo, Marco Tarini, and Riccardo Scateni. Coarse quad layouts from curve-skeletons. *ACM Transactions on Graphics (TOG)*, in press.

List of Figures

List of Tables

Listings

Ringraziamenti

Beh finalmente posso scrivere in italiano! È giunta l'ora di fare i ringraziamenti (questo vuol dire che finalmente siamo arrivati alla fine di questa "avventura"). Già non ho voglia di scriverli, l'unica nota positiva è che posso non farli in inglese, quindi li scriverò nella mia *linguaccia* solita ben nota a tutti coloro che ora stanno leggendo questa pagina (avendo saltato tutto il resto della tesi perché non ve ne fotte nulla. Bastardi!!!). Non farò nomi perché non ho gana di scimprarmi a capire se ne ho dimenticato qualcuno, che poi vi infrascate...

Andiamo per ordine. Il primo da ringraziare è il Boss che, nonostante le sue prese per il culo, i suoi insulti o i suoi soprannomi come "*Giammi la merda umana*" o "*Giammi una buona parola per tutti*", mi ha sempre dato fiducia e soprattutto permesso di far parte del fantastico gruppo/famiglia della BatCaverna.

Tu mi tratti sempre male e soprattutto, cosa non banale e piuttosto fastidiosa, cerchi sempre di appoggiarmela. Nonostante tutto ti devo ringraziare lo stesso perché

senza di te non ce l'avrei fatta. Da Vancouver, Genova, Cagliari o da Casino dove ti trovavi hai sempre avuto tempo e voglia (forse) di darmi un consiglio o aiutarmi a ragionare... Quasi quasi, a mo' di ringraziamento e solo di ringraziamento, te la appoggerò pure io prima o poi.

Mica mi dimentico di tutti voi Cavernicoli. Grazie a tutti voi che nel corso degli anni siete entrati o usciti da questo posto lindo e accogliente che è ormai la nostra casa. Mi state/siete stati QUASI tutti simpatici e questa è una cosa abbastanza rara per me. Ringrazio chi è stato una fonte di ispirazione, chi mi ha aiutato nei momenti difficili (universitari e non) e chi ha condiviso con me scleri e incazzi supportandomi e sopportandomi. Ringrazio chi mi ha aiutato in questi anni con una spiegazione, una libreria, un ragionamento alla lavagna, un proofreading o anche un semplice consiglio. Infine, siccome non posso fare nomi, a voi darò tre pseudonimi; ci terrei in modo particolare a ringraziare voi tre che avete condiviso con me quasi tutti i giorni degli ultimi cinque anni e con cui forse (spero) avrò modo di condividere ancora tanto: grazie *Merda*, *Bolla* e *Ansia*!!!

Amici!!! Non mi dimentico sicuramente di voi! Grazie per l'AMICIZIA dimostrata in questi anni (o da sempre nel caso di qualcuno). La vita universitaria stressa molto, ma meeeeda, e quelli che possono sembrare piccoli gesti a volte sono in realtà molto importanti. Per fortuna non posso fare nomi, tanto qualcuno l'avrei dimenticato per forza, invece così se volete sentirvi ringraziati "siate ringraziati". Ci tengo a ringraziare chi mi ha ospitato una (molte più di una) volta a dormire per evitarmi un viaggio, chi mi ha minacciato più e più volte di *picchiarmi molto* (ahahahahahah aspettando...), chi mi ha fatto questo bellissimo disegno, chi mi ha consentito di svagare con un film al cinema, con una chiacchierata, con una birra, con una carrellata di cattiverie, con una gita o in generale facendomi sentire la sua presenza (tanto lo so che siete scemi → *dicasi "Presenza" il fatto di essere presente in un determinato luogo, o di intervenire, di assistere a qualche cosa* [cit. Treccani]). Grazie ancora a chi ha condiviso con me una birretta, una cena in mensa, un caffè a casa, uno yogurt, un concerto, una serata al biliardo, qualche birra, una giornata al mare e qualsiasi altra cosa. Oh mo basta, senza strollicarsi troppo, grazie e bo...

Non mi sono dimenticato di voi. Grazie alla mia Famiglia per aver sempre sempre sempre creduto in me in maniera incondizionata (anche quando io ormai avevo smesso). La cosa più importante nei momenti di difficoltà è sapere che, comunque andranno le cose, qualcuno che crede in te ci sarà sempre... Quindi GRAZIE.

Grazie a ME per esserci riuscito!!!