QUAD DOMINANT 2-MANIFOLD MESH MODELING

A Dissertation

by

MEHMET OZGUR GONEN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Wei Yan
Co-Chair of Committee,	Ergun Akleman
Committee Members,	Tracy Hammond
	Jose G Esquivel
Head of Department,	Robert Warden

May 2017

Major Subject: Architecture

ABSTRACT

In this dissertation, I present a modeling framework that provides modeling of 2D smooth meshes in arbitrary topology without any need for subdivision. In the framework, each edge of a quad face is represented by a smooth spline curve, which can be manipulated using edge vertices and additional tangential points. The overall smoothness is achieved by interpolating all four edges of any given quad across the quad surface.

The framework consists of simple quad preserving operations that manipulate the principal curves of the smooth model. These operations are all variants of a generic "Curve Split" and its inverse, "Region Collapse". By only using these sets of simple operations, it is possibly to model any desired shape conveniently. I also provide implementation guidelines for these operations.

In the results of this dissertation, I present three main applications for this modeling framework. The major application is modeling Mock3D shapes; shapes with well defined interior normals by interpolating the normals at the boundaries of the shape across its surface which can serve as a mock 3D model to mimic a 3D CGI look. As a second application, the framework can be used in origami modeling by allowing assignment of crease patterns across the surface of 2D shapes modelled. Finally, vectorization of reference photos via modeling figures by following their contours is presented as a third application.

DEDICATION

To my father Emrullah Gonen, who has wanted me to get this doctorate degree more than I have.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Ergun Akleman of the Department of Visualization and Professor Wei Yan, Jose Esquivel of Department of Architecture and Professor Tracy Hammond of the Department of Computer Science.

Professor Ergun Akleman has been the main advisor on this research, while Professor Wei Yan served as the committee chair. The origami examples shown in Section 7.2 was created by the MS student Han Wei Kung of the Department of Visualization and the example for self-folding reconfigurable structure in the same section was created by PhD student Edwin Hernandez of Department of Mechanical Engineering and published in 2016 in the Journal of Mechanisms and Robotics as cited.

All the other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by a fellowship from Texas A&M University and partially by National Science Foundation from awards NSF-EFRI-ODISSEI: Award 1240483 and NSF-CCF: Award 0917288.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND MOTIVATION

In 3D computer graphics, 3D modeling is the process of developing a mathematical representation of any three-dimensional surface of an object mostly via a modelling software. This process has challenges due to the 2D nature of the input and output devices used in the process, such as computer mouse and 2D computer displays. Creating a 3D model on the computer requires one to interact with the computer in 2-dimensions. Despite this challenge, most of the commercial modeling applications primarily support 3D platforms due to the general demand for 3D modeling in the film and video gaming industries or computer aided manufacturing. However, according to a recent market research 3D Graphics is still only 8% of the whole graphics market, while 2D graphics market such as vector, image and video constitutes the rest, i.e. more than 90%, of the graphics market [1]. Moreover, the 3D modeling market does not grow as rapidly as 2D painting/editing market.

There could be several foreseeable reasons to explain the reluctance of market share of 3D modeling, such as it could be less intuitive, more expensive and require more training than 2D. Additionally, in 3D modeling, it is harder to include all types of expressive depictions that are caused by impossible, inconsistent and incoherent shapes. Although, this can be seen as a problem for the shape modeling community, it could be an opportunity for the community to explore new areas in shape modeling research. Namely, this reluctance suggest that there exists a critical need to develop hybrid systems that can provide 3D effects along with the convenience and expressive power of 2D.

In this research, I propose a framework for modeling quad-dominant 2D meshes with arbitrary topology that can be used as an alternative for modeling 3D meshes.

These 2D meshes are quad dominant in the sense that they are composed of mostly faces with four edges. The quad face restriction is motivated by the idea of representing the faces of the model by spline patches. Since a spline patch can be curved and smooth, It's possible to obtain single view representation of complicated smooth 3D shapes by using a set of connected 2D patches.

In traditional polygonal modeling, a smooth model is obtained by applying a *smoothing* operation on a base polygonal mesh with relatively low tessellation. This smoothing operation is often times a subdivision method such as Catmull-Clark, which doubles the overall tessellation of the mesh when applied. To achieve a compelling result, this operation it applied twice or three times which may increase the polygon count by a factor of eight. Often times, this modeling approach requires the modelling artist to switch between the base mesh and its subdivided version as he keeps refining the model. The fundamental reason for this approach is that it is much more convenient for the artist to model in a lower polygonal model and let the computer handle the smoothing.

Alternatively, in geometric modeling, spline surfaces are a great way of obtaining a smooth surface. A spline surface is composed of atomic elements called *patch*es. A patch is obtained by interpolating either control points or bounding curves. Creating a 3d model via spline patches is called *patch modelling*. In this modeling approach, it is possible to obtain a smooth model bypassing the smoothing step. This is a crucial motivation point for this research.

Most spline patches are defined over interpolation $(n \times m)$ control points in four-sided rectangular form. In addition to spline patches, *Coons Patch* is also defined over bi-linear interpolation of four boundary curves. There also exists three-sided variants of the patches mentioned, however, they are obtained by using a dummy boundary condition in zero length.

Obviously, four-sided patches are very suitable representations for quad faces, since there exist a one-to-one match between the edges of the quad and sides of a four-sided patch. Consecutively, if we have a mesh whose faces are all quads, it is possible to represent it via four-sided patches throughout which results in an over all interpolated smooth mesh. This smooth mesh can be used in various applications.

The major application for this 2D modeling framework is a mock-3D shape representation that consists of texture mapped 2-complexes. The key part of this representation is that the textures that define non-conservative 2D vector fields along with thickness fields, which we call shape maps. Using shape maps, for any given mock-3D scene and a given 3D position, we can uniquely compute every 3D shape in the scene using rays emanated from the given position.

These mock-3D scenes are view dependent since the shapes of all objects in the scene depend on the positions of ray centers. Using these dynamically computed shapes, we can compute any illumination effect that requires geometry such as shadows, reflection and refraction in real time.

This representation is powerful enough to handle all types of expressively depictions from impossible renderings/shapes to incoherent or inconsistent renderings/shapes. The application I have developed for this representation turns shape modeling in to an easy-to-use, easy-to-extend 2D graphics application.

The 2D nature of the framework proposed is also suitable for designing Freeform Origami. The framework can serve as a fold pattern design tool that can provide curved crease patterns. Freeform Origami allows users to manipulate the shape of provided origami forms or user-defined forms to design crease patterns. Alternatively, well-known straight crease patterns and their curved versions can be generated using this framework. These patterns can be exported to be used by laser cutters and FEA software for coordinated fabrication and thermomechanical folding analysis in

a origami making pipeline.

Another application for the proposed framework is image vectorization, which is a commonly used technique in the graphic design community to create illustration-like images. In the image vectorization process, the graphic artist takes a photo as a base image and tries to recreate a vector representation for the underlying photo by using vector graphic tools. Gradient Mesh Tool in Adobe Photoshop is one if the common tools used in this process. However, this process is mostly manual and often very time consuming. The Gradient Mesh tool works in individual rectangular pieces and often times a single rectangular region is not adequate to represent a shape in arbitrary topology. The meshes created by the framework I propose comes in quad dominant in arbitrary topology, making it very suitable for this purpose.

2. BACKGROUND

In this chapter, I explain the theoretical grounds that this research stands on. I also give a brief overview of the related work in the literature.

2.1 Polygonal Meshes

The shapes studied in geometric modeling are mostly two- or three-dimensional. Today most geometric modeling is done with computers and for computer-based applications. Two-dimensional models are important in computer typography and technical drawing. Three-dimensional (3D) models are central to computer-aided design and manufacturing (CAD/CAM), and widely used in many applied technical fields such as mechanical engineering, architecture, geology, medical image processing and entertainment industries.

Polygonal meshes are widely used representations for 3D models in such applications. A polygonal mesh is based on the idea of cell decomposition: a complex object is represented with an assembly of simple polygonal cells. Triangles and quadrilaterals are the most common cells used in polygonal meshes. While triangle meshes are much more common in computer graphics, quite a number of tasks are better suited to quadrilaterals (quad meshes), such as texturing, compression and finite element simulation.

2.2 Topological Concepts

Topology primarily deals with the qualitative characteristics of a geometrical object rather than its quantitative dimensions [1]. The modeling operations of the framework presented in this research are topological in nature and involves the following topological concepts.

The topological concept of a *2-manifold* is the fundamental topological concept in this research. A *2-manifold* or a *2-dimensional manifold* is a topological space where every point has a neighborhood topologically equivalent to an open disk. In other words, the geometrical object locally resembles the *plane* [1].

A closed surface is a connected, closed, 2-manifold [1]. That is, it consists of a single piece and has no boundaries. A 2-manifold in general consists of a number of surfaces, each of which is homeomorphic (topologically equivalent) to a sphere with zero or more handles. The number of handles on the sphere is called the genus of the surface. Equivalently, a *genus* can be defined to be the number of holes in the surface. The genus of a 2-manifold is the sum of the genera of its component surfaces.

2.3 Mesh Representation

The 2-manifold mesh structure that is the backbone of this research has to be represented via a competent data structure. Several data structures have been proposed to represent 2-manifold mesh structures. Some of these are face-based in which mesh faces are explicitly given in consistent and oriented directions [2], while others are edge-based in which adjacency relationships around each edge are given [3] [4] [5] [6] [7] [8]. Baumgarts winged-edge structure [5] is the most well known edge-based representation, based on 10 which several variants have been proposed, including Weilers edge based structure [3], Mantylas half-edge structure [4] and Guibas and Stolfis quad-edge structure [6].

Several of the above data structures, including Weilers radial-edge structure [9], Karasicks star-edge structure [8] and Vaneceks edge-based data structure [7] can support a wide range of non-manifold surfaces. Mantylas half-edge representation [4] is one data structure that is designed to support manifold meshes. It is possible to

make the internal representation of the objects valid orientable 2-manifold structures even when the corresponding geometric shapes appear to be non-manifold [10].

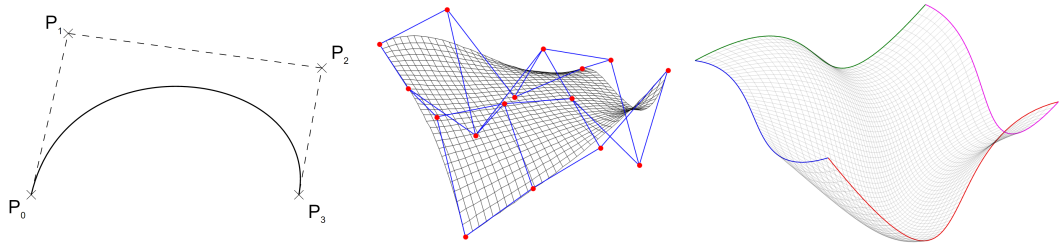
Akleman and Chen introduced a topologically robust mesh modeling approach [11] by adopting topological graph theory to computer graphics and shape modeling. Their 2-manifold mesh modeling scheme is based on a minimal set of manifold preserving operators [11] that are simpler, more intuitive and more user-friendly when compared to previously proposed schemes. The minimal set of fundamental operators that have been identified are : CreateVertex, which inserts a new vertex into the mesh, DeleteVertex, which removes an existing vertex from the mesh, InsertEdge, which inserts an edge between two existing corners of the mesh and DeleteEdge, which deletes an existing edge from the mesh [11].

2.4 Spline Representation

In mathematics, a spline is a numeric function that is piecewise-defined by polynomial functions, and which possesses a high degree of smoothness at the places where the polynomial pieces connect (which are known as nodes) [12]. The term spline is adopted from the name of a flexible strip of metal commonly used by drafters to assist in drawing curved lines. In computer graphics, parametric curves whose coordinates are given by splines are popular because of the simplicity of their construction, their ease and accuracy of evaluation, and their capacity to approximate complex shapes through curve fitting and interactive curve design.

2.4.1 Bezier Curves and Surfaces

Bezier Curve is one of the most common type of spline curve in computer graphics to model smooth curves [12]. As the curve is completely contained in the convex hull of its control points, the points can be graphically displayed and used to manipulate the curve intuitively. Quadratic and cubic Bzier curves are most common.



(a) Bezier Curve (b) Bezier Surface (c) Coons Patch

Figure 2.1: Sample of spline representations relevant to this research.

Particularly, cubic Bezier curves are in great use since the provide two end and two tangent points. They are patched together, producing a composite Bzier curve which is commonly referred to as a "path" in vector graphic applications such as Adobe Illustrator, CorelDraw. Since it is possible to break the smoothness of the composite curve at the control point at which two curves meet, paths provide great flexibility in vector graphics. In this research, cubic Bezier curves are chosen for modelling smooth shapes for this particular reason.

Four points P_0, P_1, P_2 and P_3 in the plane (or in higher-dimensional space) define a cubic Bzier curve [12]. The curve starts at P_0 going toward P_1 and arrives at P_3 coming from the direction of P_2. Usually, it will not pass through P_1 or P_2; these points are only there to provide tangent information. The explicit form of the cubic Bezier curve is:

$$B(t) = (1-t)^3 P_0 + 3(1-2)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P, 0 \leq t \leq 1 \qquad (2.1)$$

2.4.2 Coons Patch

In computer graphics, *Coons Patch* is a type of manifold parametrization that creates a smooth surface between four space curves that meet at corners [12]. Since

these four curves form a quad, it is the motivational interpolation method for this framework.

Given four space curves $c_0(t), c_1(t), d_0(t), d_1(t)$ and their condition to meet at corners as $c_0(0) = d_0(0)$, $c_0(1) = d_1(0)$, $c_1(0) = d_0(1)$, $c_1(1) = d_1(1)$, Coons first does a linear interpolation between the opposing pairs c_0 and c_1 as

$$L_c(s,t) = (1-t)c_0(s) + tc_1(s)$$

and between d_0 and d_1 as

$$L_d(s,t) = (1-s)d_0(t) + sd_1(t)$$

producing two ruled surfaces, and the bilinear interpolation on the four corner points would be

$$B(s,t) = c_0(0)(1-s)(1-t) + c_0(1)s(1-t) + c_1(0)(1-s)t + c_1(1)st$$

Then a Coons interpolation for the quad face can be written as in 2.2

$$C(s,t) = L_c(s,t) + L_d(s,t) - B(s,t) \tag{2.2}$$

2.5 3D Modeling

Although it is possible to construct a mesh by manually specifying vertices and faces, the common approach is to build meshes using a variety of tools provided by a modeling software package. *Polygonal Modeling* is the process of constructing a polygonal mesh.

2.5.1 Primitives

Primitives are three-dimensional geometric shapes you can create in a 3D-Modeling software. The primitive shapes available include spheres, cubes, cylinders, cones, planes, and many others (see Figure 2.2). You can modify the attributes of basic primitives to make them more or less complex. Many 3D modelers begin with polygon primitives as a basic starting point for their models. This technique is referred to as *primitive-up* modeling.

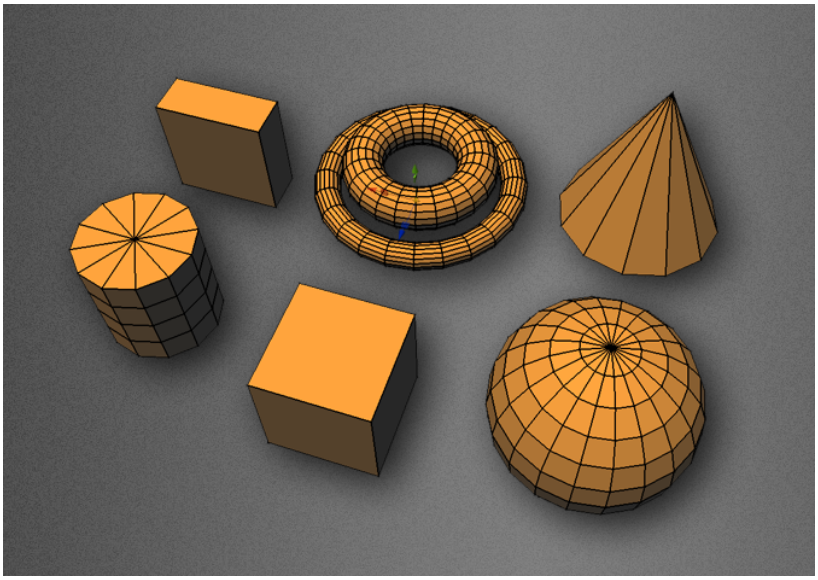
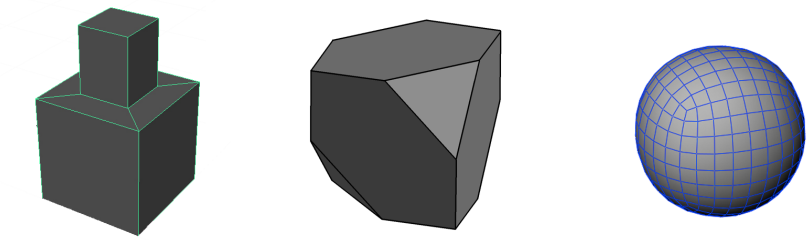


Figure 2.2: Primitives in polygonal modeling: The primitive shapes available include spheres, cubes, cylinders, cones, planes, and many others.

2.5.2 Operations

In a conventional 3D modeling frameworks, there are a very large number of operations which may be performed on polygonal meshes which modifies them topo-

logically and geometrically. Down bellow some of these operations are mentioned.



(a) Extruded cube (b) Truncated cube (c) Subdivided cube

Figure 2.3: Some sample operations in traditional polygonal modeling applied to a polygonal cube model.

- **Extrusion:** The extrusion operation, applicable to a set of edges, face, or vertices, creates a new element of same size connected to the original with a set of faces. Performing the extrude operation on a square face would create a cube connected to the surface at the location of the face. It is one of the most commonly used operations in polygonal modeling. Most modelers sculpt their initial models by performing a sequence of face extrusions on a segmented primitive.

- **Subdivision:** Subdivision is a way for modelers to add polygonal resolution to a model, either uniformly or selectively. Because a polygonal model typically starts from a low-resolution primitive with very few faces, it is almost impossible to produce a finished model without at least some level of subdivision.

  A uniform subdivision divides the entire surface of a model evenly. Uniform subdivisions are usually completed on a linear scale, meaning every polygonal

face is subdivided into four. Uniform subdivision helps to eliminate blockiness and can be used to evenly smooth the surface of a model.

- **Truncate/Bevel:** By default, the edges on a 3D model are infinitely sharpa condition that virtually never occurs in the real world. Inspected closely enough, almost every edge you encounter will have some sort of taper or roundness to it. A bevel operation takes this phenomenon into account, and is used to reduce the harshness of the edges on a 3D model: For example, each edge on a cube occurs at a 90 degree convergence between two polygonal faces. Beveling those edges creates a narrow 45 degree face between the converging planes to soften the edge's appearance and helps the cube interact with light more realistically. The length (or offset) of the bevel, as well its roundness can be determined by the modeler.

- **Refining/Shaping** Most models require some level of manual refinement via individually moving vertices around. When refining a model, the artist moves individual vertices along the x,y, or z axis to fine tune the contours of the surface. A sufficient analogy for refinement might be seen in the work of a traditional sculptor: When a sculptor works, he first blocks out the large forms of the sculpture, focusing on the overall shape of his piece. Then he revisits each region of the sculpture to fine tune the surface and carve out the necessary details. Refining a 3D model is very similar. Every extrusion, bevel, edge-loop, or subdivision, is typically accompanied by at least a little bit of vertex-by-vertex refinement. The refinement stage can be painstaking and probably consumes 90 percent of the total time a modeler spends on a piece. It might only take 30 seconds to place an edge loop, or pull out an extrusion, but it wouldn't be unheard of for a modeler to spend hours refining the nearby

surface topology (especially in organic modeling, where surface changes are smooth and subtle). Refinement is ultimately the step that takes a model from a work in progress to a finished asset.

2.6 Related Work

In this section I present some of the relevant work from the literature in regards to the methodology and application of this research.

2.6.1 Quadrangulation

In many applications meshes are generated as in triangular form. There are many methods to obtain a quad mesh from a triangular one. I classified existing quadrangulation approaches in five categories: (1) Subdivision, (2) Triangle-Combining, (3) Patch-Based, (4) Parameter Based, (5) Guiding Field Based.

2.6.1.1 Subdivision

A naive method to create quadrilateral meshes is subdivision or remeshing. Remeshing is not a practical option since it can increase the number of faces significantly. There exists two main approaches to obtain quad-dominant meshes using remeshing: (1) Vertex Insertion and (2) Corner Cutting.

Vertex Insertion: Vertex insertion is the remeshing algorithm behind popular subdivision algorithms such as Catmull Clark. Vertex insertion turns any mesh into a mesh that consists of only quadrilaterals. Therefore, Performing a Catmull-Clark subdivision [13] on any mesh will result in a whole quad mesh.

Corner Cutting: Corner Cutting is a the remeshing algorithm behind popular subdivision methods such as Chaitkin or Doo-Sabin [14]. Corner Cutting does not turn create a mesh that consists of only quadrilaterals. It increases the number of quads while keeping the existing non-quads as is.

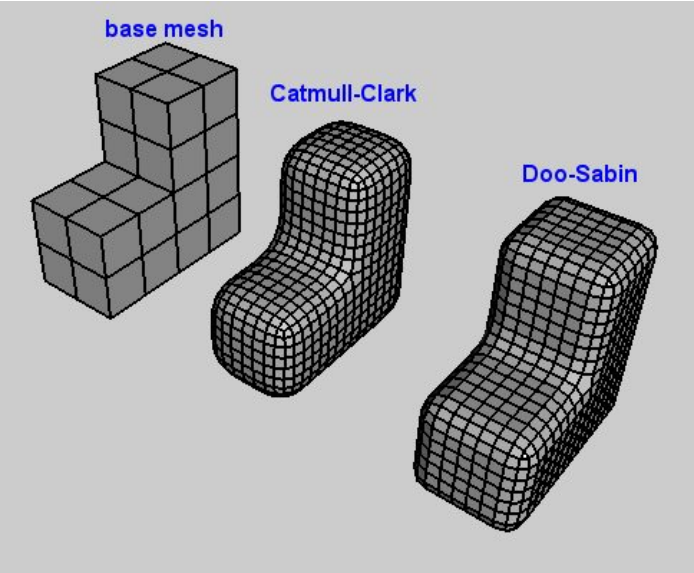


Figure 2.4: Examples of Catmull-Clark and Doo-Sabin subdivisions.

2.6.1.2 Triangle-Combining

Tri-to-quad conversion methods combines two original triangles into a quad. Naturally, these methods are expected to produce quad meshes heavily depended on the topology of the input mesh and introduce some level of irregularity. SQuad [15] is designed to improve the internal representation of meshes but can be used for tri-to-quad conversion. BlossomQuad [16] uses a combinatorial optimization algorithm to find the global optimum for conversion, however it is computationally expensive. Conversely, [17] presents a greedy approach where most eligible pairs are first identified, and remaining triangles are fused after a sequence of edge-flip operations. Figure 2.5 shows examples from these methods.

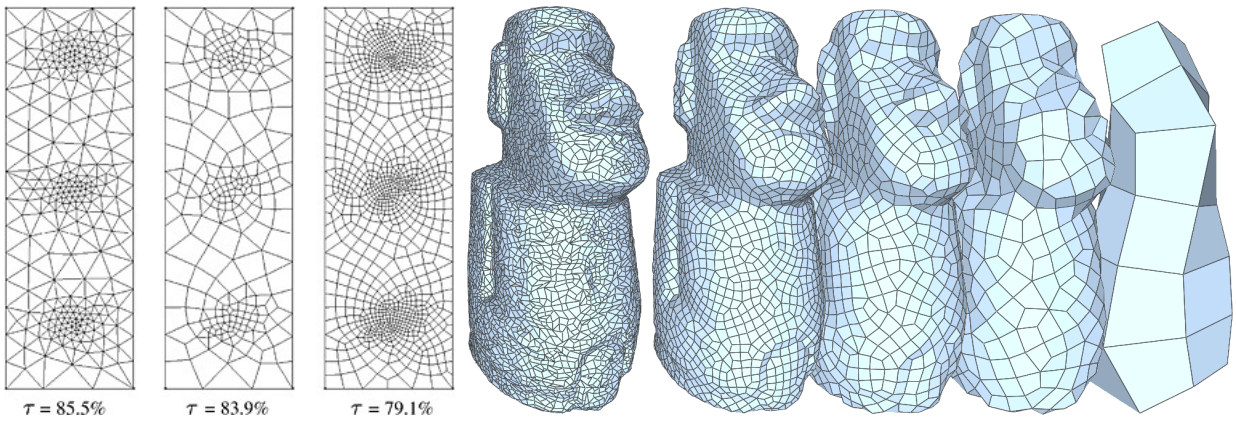


Figure 2.5: Triangle combining methods: BlossomQuad [16] (left) uses a combinatorial optimization algorithm, while Tarini et. al [17] (right) presents a greedy approach.

2.6.1.3 Patch Based Approaches

Patch-based methods work by mapping the original surface to set of square patches. The final mesh is generated by sampling the patch set as the base mesh. [18] classfies the input into flat regions by normal based clustering and extracts a coarse mesh, where high curvature regions are used in computing the base mesh. [19] generates a quad model through CC subdivision and simplifies it to a base mesh.

Gonen et al introduced sketch based modeling approach using curvature classification in [20], where 2d outline curves are represented by quadrilateral tubes.

2.6.1.4 Parameter Based Approaches

Similar to patch-based methods, there are also parametrization based methods that work by constructing a mapping of 3D surface to a 2D domain, where it can be easily tessellated into quads. [21] determines the parametrization domain topology by finding its critical points of the scalar field on the triangle mesh. There are other specialized patch quadrangulation algorithms, e.g., those that attempt to find

a topology with the fewest irregular vertices possible [22]; [23].

2.6.1.5 Guiding Field Based Approaches

Many popular algorithms generate a quad mesh from a guiding field. [24] [25], [26]. While fields can be edited by specifying locations of singularities or by controlling parameters of an optimization function, e.g., [27], [28]; [29]; [30]; [31], there is no direct relationship between a field and a resulting quad mesh, because the algorithms used to derive a quad mesh from a field are quite involved.

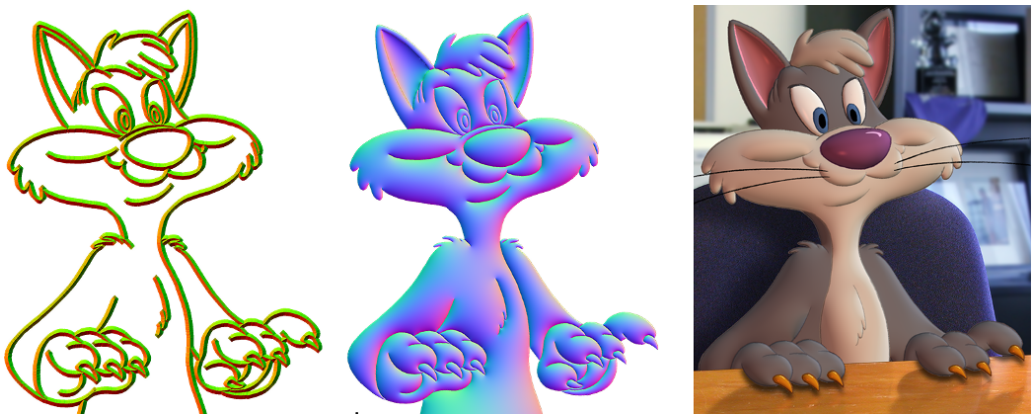
The three most related concepts for the exploration of quad mesh topologies are curve sampling [32], connectivity editing [33]; [34], and advancing fronts (paving) [35]; [36]; [37]. [32] connect the boundary vertices of a patch by curves and propose an algorithm to generate a layout and another algorithm to mutate an existing layout.

2.6.2 Mock 3D Scenes

The major application of the framework I propose is creating mock-3D scenes. There currently exists two representations that is related to this application: bas-reliefs and normal maps. However, both of them really corresponds real shapes that can exist in 3D. I present a fuzzy and view dependent representation that is suitable for global illumination while providing all the representational powers of both bas-reliefs and normal maps.

Bas-reliefs are sculptures that can be viewed from many angles with no perspective distortion as if they are just images. In other words, perspective transformation is embedded in bas-relief sculptures [38]. One problem with bas-reliefs for 2D artists is that their construction is still a sculpting process. This may not be suitable for illustrators and painters who are not interested in sculpting shapes.

Normal maps became popular when they were introduced in 1998 [40]. Although,they are mainly used as texture maps to include details to polygonal meshes,



LUMO: Illumination for cel animation

An image-based shading pipeline for 2D animation

Figure 2.6: The pipelines of previously suggested mock-3d systems: LUMO, model normal maps by diffusing 2D mormals in line drawings. Bezerra et al [39] suggested an image-based shading pipeline by inspecting the hand drawn image directly.

they can directly be used as shape representations by embedding perspective information as shown by Johnston [41]. He developed a sketch based system, called *LUMO* (see Figure 2.6), to model normal maps by diffusing 2D normals in a line drawing. Since then, only a few groups investigated the potential use of normal maps as a shape representation such as [42, 39, 43, 44]. Sun et al. [45] introduced Gradient Mesh to semi-automatically and quickly interpolate normals from edges, and Orzan et al. [46] calculate a diffusion from edges by solving the Poisson equation. Sýkora et al. [47] proposed Lazy-Brush, which can propagate scribbles to accelerate the definition of constant color regions. Finch et al. [48] build thin-plate splines which provide smoothness everywhere except at user-specified tears and creases. The underlying splines are used to interpolate normals.

Wu et al. [49] proposed shape palette, where user can draw a simple 2D primitive in the 2D view and then specify its 3D orientation by drawing a corresponding primitive. This method also performs diffusion using a thin-plate spline. Recently, Shado et al. [44] developed CrossShade, another sketch based interface to design complicated shapes as normal maps. They use an explicit mathematical formulation of the relationships between cross-section curves and the geometry. The specified cross-section is used as an extra control point to control the normals. Vergne et al. [50] introduces surface flow from smooth differential analysis, which can be used to measure smooth variations of luminance. Therefore, the author also propose to drawing the shadows and other shading effects.

3. QUAD DOMINANT SMOOTH MESH REPRESENTATION

In this chapter, I explain how a smooth mesh is represented in this framework and define the basic terminology used in this representation.

3.1 Basic Definitions

In the context of this framework, a *Quad Dominant Smooth Mesh* is the smooth geometrical surface that represents a quad dominant mesh. A *quad dominant mesh* is a mesh composed of mostly quad faces, where a *quad face* is a face with exactly four vertices and four edges connecting its these vertices. Any non-quad face in the mesh a quad dominant mesh is an *irregular face* and not legit in the framework. Exceptions to this constrain are *triangular (cap)* faces and *invisible (outer)* faces.

A *triangular face* in quad dominant mesh is an irregular face with three vertices and edges. These type of faces may be needed to create caps or saddles in the shape.

An *invisible face* is a face that is marked as invisible to be literally invisible to the user. An invisible face, also called *outer face*, is not rendered on the screen, but is internally used to preserve the 2-manifold property when representing a hole in the shape or the shape outline. Each quad dominant mesh has at least one outer face to represent the outline of the shape. Each additional outer face means an additional genus in the shape.

Any edge of an outer face is an *outer edge* in the mesh. Conversely, an *inner edge* is an edge that does not belong to any outer face, meaning shared only by visible faces.

The smooth representation of quad dominant mesh is obtain via a spline representation of a polygonal base mesh. In this representation, edges and faces are represented by spline curves and surfaces, respectively.

The spline representation of an edge is referred as a *curved edge*. A *curve* in the framework may consists of one or more curved edges. If a curve is composed of outer edges only, it is called a *boundary curve*. Conversely, a *non-boundary curve* is composed of all inner edges. A curve is *closed* if it loops back to itself and *open* if it does not. A curve whose end points touches to an outer edge may be interpreted open or closed.

3.2 Curve Table

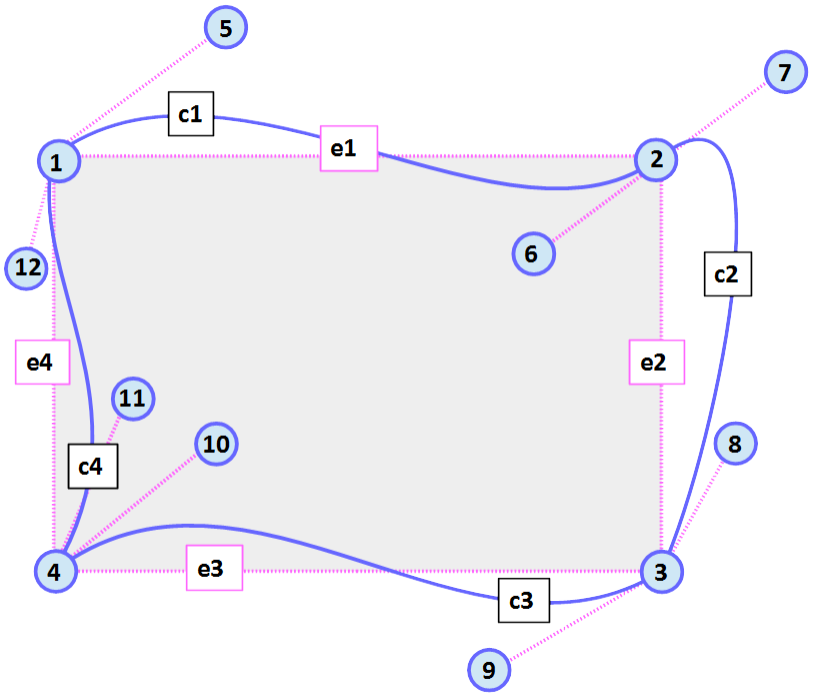


Figure 3.1: Curved edges of a quad face.

The quad dominant mesh is topologically a mesh composed of vertices, edges and faces. In the conventional geometric representation of a polygonal mesh, these terms

Edge	Curve	Start	CV1	CV2	End	Degree
e_1	C_1	v_1	v_5	v_6	v_2	3
e_2	C_2	v_2	v_7	v_8	v_3	3
e_3	C_3	v_3	v_9	v_{10}	v_4	3
e_4	C_4	v_4	v_{10}	v_{11}	v_{12}	3

Table 3.1: The curve table for the quad face.

refers to points, lines and polygons, respectively. However, a *smooth* representation of the base mesh requires extending its ordinary polygonal representation. For this, instead of lines, we can represent the edges by a spline curve that interpolates the two end-points, which correspond to vertex positions. To represent an edge as a spline curve, we use a Bezier curve of order $k > 2$. The reason is that Bezier provides continuity between adjacent curved edges by interpolating the end points. Bezier representation requires additional $k - 2$ number of *intermediate control vertices* to be stored per edge in addition to the positions of two vertices, which are actually the two end-points of the curved edge. The intermediate control vertices do not actually belong to the topology of the base mesh and are only needed for drawing the curve. On the other hand, they are needed during modeling process when an operation performed on the edges. Therefore, we use a *curve table* per edge that store the positions of all control vertices.

Figure 3.1 illustrates a quad face whose edges are represented by cubic Bezier curves that are chosen for this research. As seen in the figure, each curve interpolates the two end-points of the edge that it represents along with two additional control points that serves as tangents for the cubic Bezier curve. The table 3.1 is the curve table for the quad face in Figure 3.1. Note that each edge has a direction denoted by its *start* and *end* vertices in the table. The order of the intermediate control

vertices of the Bezier should also follow this direction for a consistent geometric representation.

3.3 Bezier Surface Representation

It is also possible to use Bezier surfaces for smooth geometric representation of a quad face for a smooth representation. A Bezier surface of degree (n, m) is defined by a set of $(n + 1, m + 1)$ control points denoted by $k_{i,j}$. In our case, we can use cubic Bezier surfaces which requires $(4, 4)$ control points. These control points can be retrieved from the four curved edges enclosing a quad face. This requires a mapping between the control points $k_{4,4}$ of the bezier representation and the set of control vertices $(v_0, v_1, ..., v_n)$ enclosing the quad.

For this mapping, we need to make use of the rotation system in the mesh. Given a quad face with its curved edges in rotation order as C_0, C_1, C_2, C_3, all the vertices from each curved edge should map to a control vertex in the $k_{4,4}$. While this mapping seems straightforward, note that if the direction of the curve does not match the given rotation order, the curve should be flipped for the mapping. This is a crucial point in the implementation.

After the control vertices of the curved edges are mapped, there will obviously be inner control vertices of the Bezier Surface remaining unmapped. Although it is possible to introduce additional control vertices for each quad face, for the sake of simplicity, we can derive the inner points from the parallelogram defined by the control points at respective corners. Table 3.2 shows a mapping of the control vertices of the quad face shown in figure 3.1. Note that the intermediate control points are computed as an average of intermediate curve points at respective corners.

After the proper mapping, a two dimensional bicubic Bzier surface that geometrically represents the quad face f can be defined as follows:

$k_{i,j}$	$k_{i=0}$	$k_{i=1}$	$k_{i=2}$	$k_{i=3}$
$k_{j=0}$	v_1	v_5	v_6	v_2
$k_{j=1}$	v_{12}	$(v_5 + v_{12})/2$	$(v_6 + v_7)/2$	v_7
$k_{j=2}$	v_{11}	$(v_{11} + v_{10})/2$	$(v_9 + v_8)/2$	v_8
$k_{j=3}$	v_4	v_{10}	v_9	v_3

Table 3.2: Mapping of bicubic Bezier surface control vertices.

$$P(u,v) = \sum_{i=0}^{4}\sum_{j=0}^{4} B_i^4(u)B_j^4(v)k_{i,j}$$

using Bernstein polynomials denoted by $B_i(t)$.

3.4 Interpolation

As previously stated, the main motivation behind the quad restriction is that it provides convenient interpolation of the data at boundaries to the interior region. The data referred here can typically represented as a n-dimensional vector, denoted as $\vec{d} =< d_0, d_1, d_2, ..., d_n >$, where each component refers to a *dimension* to be named by the application. These dimensions may refer to the data including *color* as $\vec{c}_{rgb} = (d_2, d_3, d_4)$ and *normal vector* as $\vec{n}_{xyz} = (d_5, d_6, d_7)$ depending on the context.

In this framework, each vertex v_i stores a data vector $\vec{d_i}$ to be interpolated. These data vectors should first be propagated to the boundaries of the quad region, namely to the curved edges enclosing each quad face. Given a curved edge $C(t)$ in parametric form, we can use linear interpolation to figure out the data vector $\vec{d}$ at point $C(t)$ on the curve as

$$D(t) = \vec{d_0}(1.0 - t) + \vec{d_3}t$$

where, $\vec{d_0}$ and $\vec{d_3}$ refers to the data vectors at the end points of the curve. We

can call the parametrized representation of data vectors along the boundary curve $D(t)$ as a *data curve*.

However, the *normal vector* component of the data vector is a special case since the normal is a property that depends on the surface geometry. To properly interpolate the normal vector for an interior point on the curve, we need to take the Frenet-frame of the point into account. For this, we rotate the normal vectors n_0 and n_k prior to linear blend, defined by the rotation of their initial frames. In figure 7.4 we see a demonstration of this process. The normal vectors n_0 and n_3 at both end points of the curve are first rotated as n'_0 and n'_3 prior to linear interpolation based on the Frenet frame at the intermediate point.

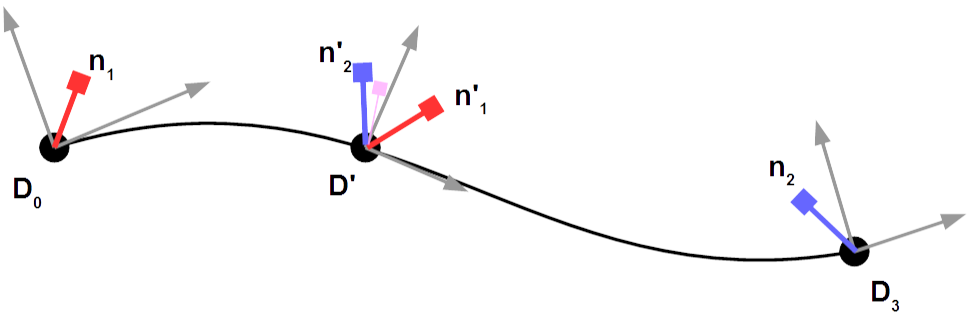


Figure 3.2: Interpolating normal vectors.

Once we have four data curves enclosing a quad face, we can use Coons Interpolation to define a data vector for any interior point in the quad region as previously mentioned. For the given the data curves $D_0(t), D_1(t), D_2(t), D_3(t)$ of a quad in the rotation order, following the equation 2.2 the Coons interpolation can then be written as

$$D'(s,t) = (1-t)D_0(s) + tD_2(s) + (1-s)D_1(t) + sD_3(t)$$

and

$$D(s,t) = D'(s,t) - (D_0(0)(1-s)(1-t) + D_0(1)s(1-t) + D_2(0)(1-s)t + D_2(1)st)$$

Since the whole surface of model composed of quad faces, we can easily compute a data vector for each surface point by this definition. Note that the direction of the data curves should again follow the rotation order.

4. QUAD-PRESERVING OPERATIONS

The core of this framework is the *quad-preserving* modeling operations. Quad-preserving operations modify meshes while preserving quad-mesh property. In other words, a quad-mesh preserving operation transform any given quad mesh into another quad mesh.

In this work, I have identified all quad-mesh preserving operations. I have observed that all of these operations can be considered as operators that split curves. Therefore, I will first introduce curve split as a general conceptual operation.

4.1 Generic Curve Split Operation

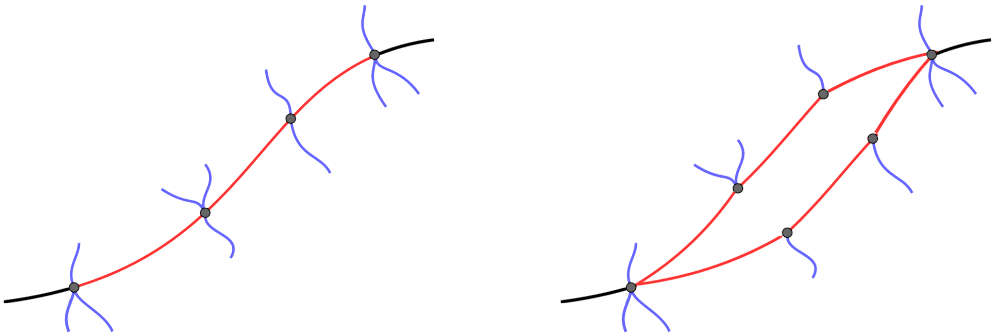
Curve split figuratively split a given closed or open curve that consists of edges. The operation creates two *split-pairs*, and an in-between region, which I refer as *split-region*.

The curve to be split can be either open and close. In this section, I will analyze compare and contrast properties of split-regions, for open and closed curves.

Figure 4.1 shows a *split-region* when an open curve is split. If the original mesh is a genus-0 2-manifold, this split creates two disconnected genus-0 2-manifold meshes with boundaries. Since this operation does not change original quadrilaterals, it is naturally quad-preserving. At this point, if we mark the split-region as an outer face, we simply create a cut in the mesh.

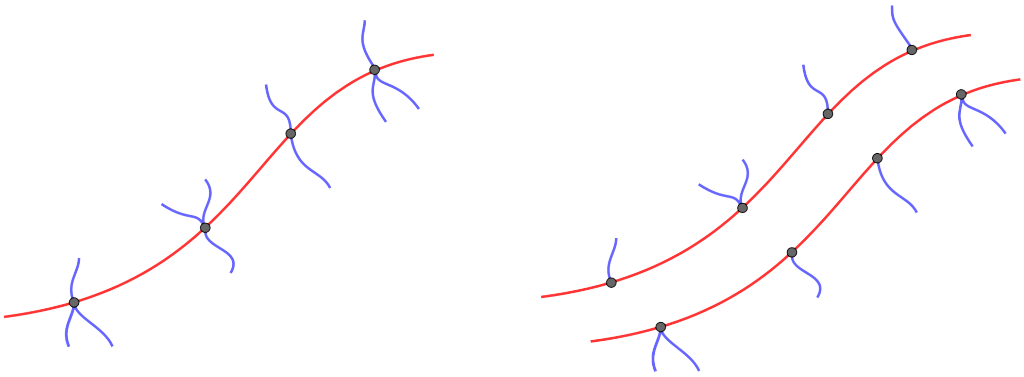
However, if we would like to keep the surface in the split region as a part of the shape, we need strategies to remesh such split-regions to obtain quadrilaterals.

There are actually two more cases: (1) An open curve with both curve ends are in boundary; (2) An open curve with one of the curve ends are in boundary. To simplify the presentation, we assume the first case is simply an open curve since the



(c) Selected open curve (d) Split region

Figure 4.1: Splitting an open curve creates a split region that is not necessarily a quadrilateral. Selected curves are painted in red.



(a) Selected closed curve (b) Split region

Figure 4.2: Splitting a closed curve creates a split region. This region disconnects the original genus-0 2-manifold mesh into two genus-0 2-manifold meshes with boundary based on Jordan's curve theorem. Note that the resulting split region is not a face. It is a region bounded by the boundaries of two genus-0 2-manifold meshes. Selected curves are painted in red.

operation disconnects the mesh into two. We also assume the second case is just a special case of open curve split.

4.2 Stitching Strategies to obtain Quad-Preserving Curve Split Operations

As discussed in the previous section, we need to quadrangulate the split-regions to preserve the quad-mesh property or not to disconnect original mesh into two.

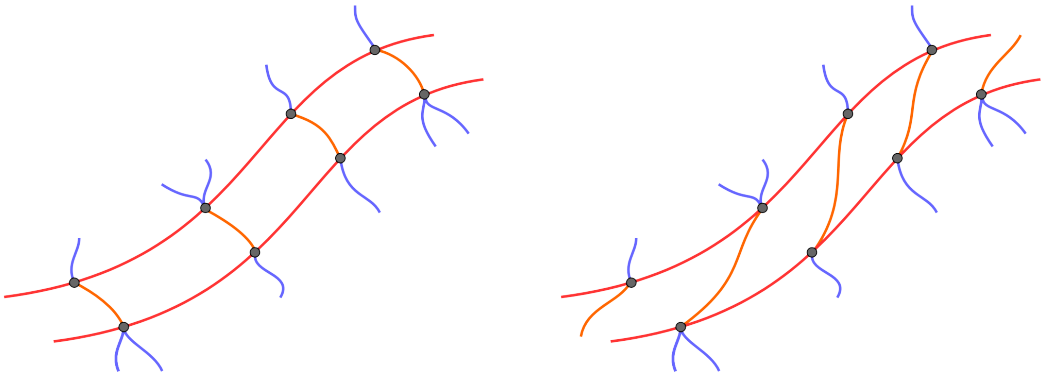
Analogically, the quadrangulation operation could be thought of dissecting the mesh along the given curve and stitching it back by inserting new curved edges between the split vertices. As shown in Figure 4.3(a) and (b), there are two possible way two stitch split-regions: (1) stitch the corresponding vertex-pairs, (2) stitch diagonally by connecting a vertex in one side to the next vertex in the other side. The concept of the direction comes from the fact that we need to assign a direction to the original curve in order to differentiate *next* and *previous* vertices. In conclusion, curve split operations can be classified into two categories: undirected and directed.

4.2.1 Quad Preserving Curve Split Operations

The discussion above demonstrate that there are three possible cases: (1) Selected curves can be either open or close; (2) The split region can either be left as a cut or quadrangulated; (3) Selection can be directed or undirected to differentiate two types of quadrangulations.

Therefore, this gives us six possible cases:

- Open Cut
- Closed Cut
- Open Directed



(c) Undirected quadrangulation (d) Directed quadrangulation

Figure 4.3: Two examples of quadrangulation of split-region: In an *undirected* quadrangulation, the edges run straight between each split vertex pairs. In the *directed* case, they run diagonally by one neighbor offset.

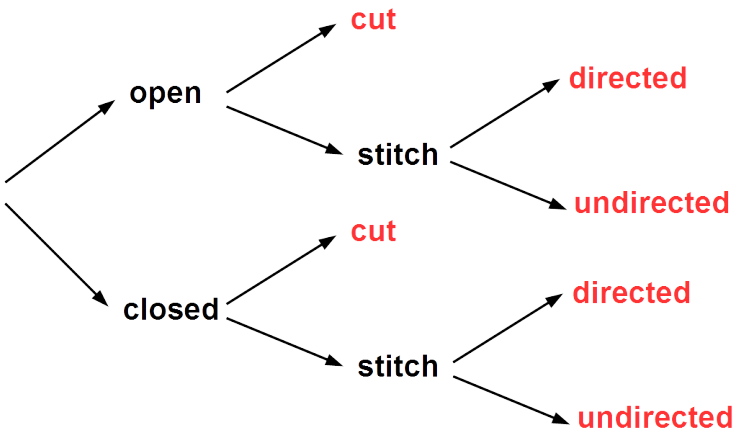


Figure 4.4: The possible cases in curve split: six types of splits are highlighted in red.

- Open Undirected
- Closed Directed
- Close Undirected

As conclusion, we have identified *six* legitimate operations that can provide topologically distinct split curve operations. The following four subsections provides a detailed description of these four operations.

4.2.1.1 Open Cut Splitting:

Performing on an open curve, this operation creates an open cut in the mesh by marking the split region as outer face. Figure 4.5 shows different case of of open cut split.

4.2.1.2 Closed Cut Split:

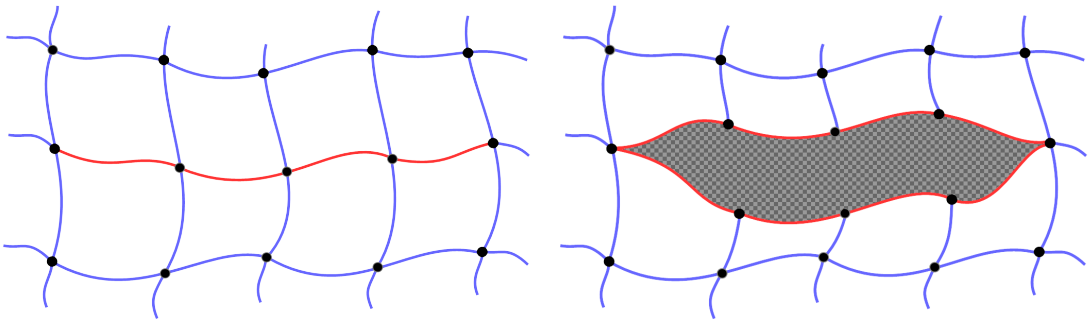
Performing on an closed curve, this operation separates the mesh into two pieces. Figure 4.5 shows different case of of open cut split.

4.2.1.3 Directed Open Curve Split:

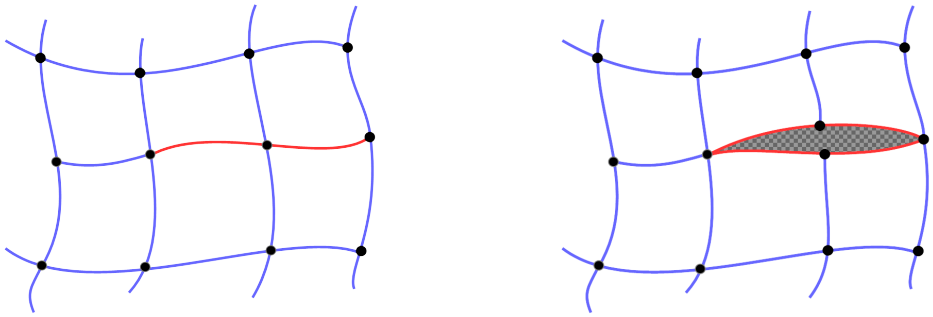
This operation can split any selected curve on the mesh by a given *direction*. It uses *directed quadrangulation* that follows the given direction, as in figure 6.1.

4.2.1.4 Undirected Open Curve Split:

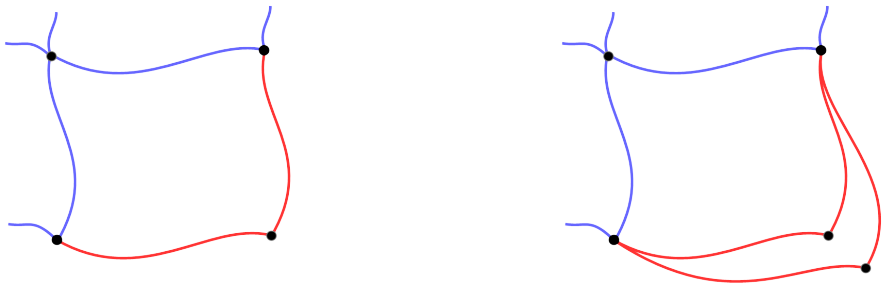
This operation is defined on open curves only. It performs a curve split on both sides of the open curve using *undirected quadrangulation*, which introduce two neighbouring split regions. Performing an *undirected quadrangulation* on these two split regions removes the edges of the original curve pointing to the end points. This results in a two-row grid with single quad faces at its end points as in 4.7.



Open curve selection Cut splitting

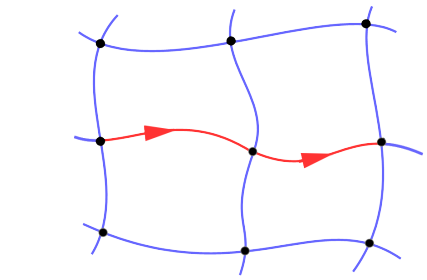


Open curve selection Cut splitting

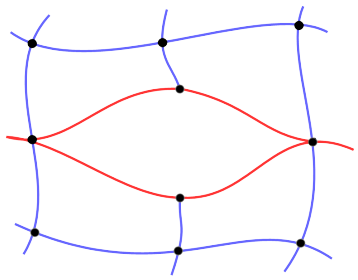


Open curve selection Cut splitting

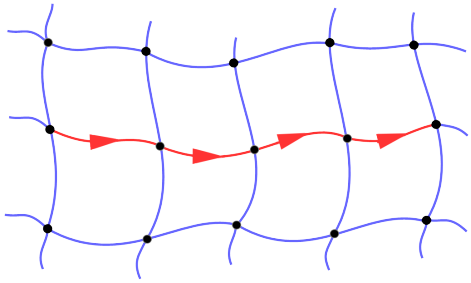
Figure 4.5: Examples of open cut splitting: In the top image since the resulting split-region is already a quad, there is no need for additional stitching.



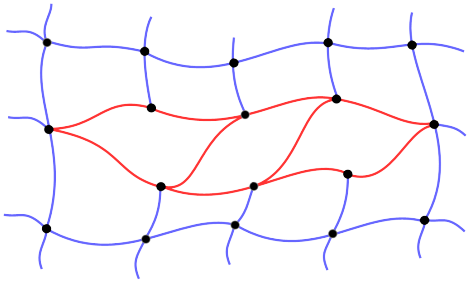
Directed open curve selection



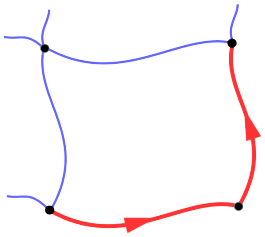
Directed open curve split



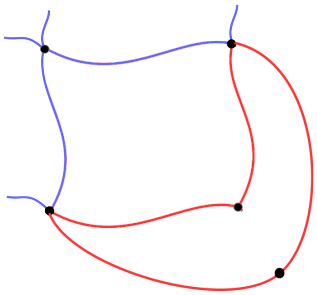
Directed open curve selection



Directed curve splitting

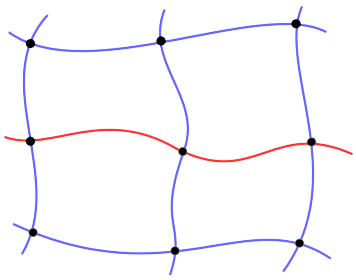


Directed open boundary curve selection

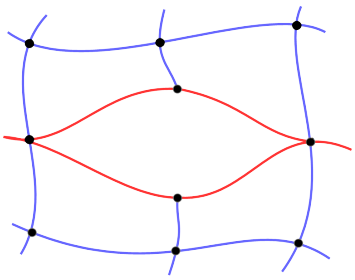


Splitting operation

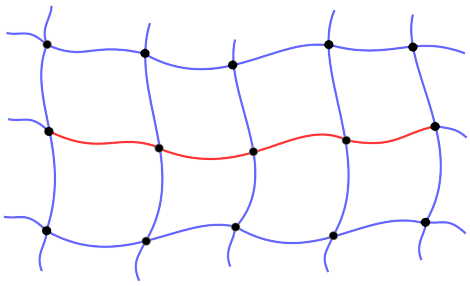
Figure 4.6: Examples of directed curve splitting: In the top image since the resulting split-region is already a quad, there is no need for additional stitching.



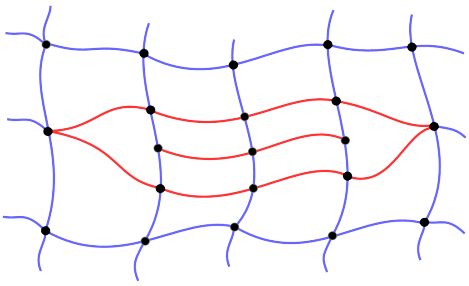
Undirected open curve selection



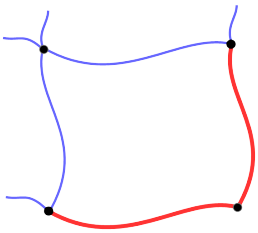
Undirected open curve split



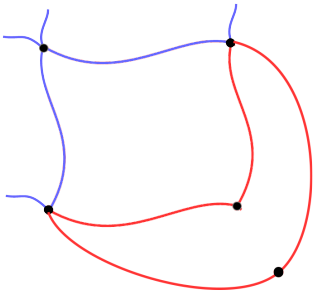
Undirected open curve Selection



Undirected open curve splitting



Undirected open boundary curve selection



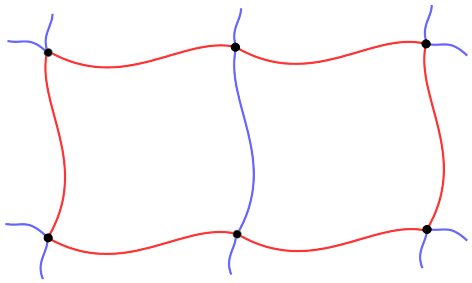
=

Undirected open curve splitting

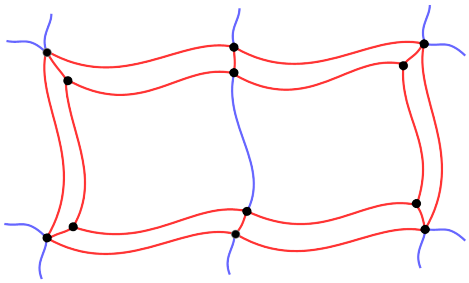
Figure 4.7: Examples of undirected open curve splitting: Note that if the curve consists of only two edges undirected and directed gives the same result and it always produces a new quadrilateral.

4.2.1.5 Undirected Closed Curve Split:

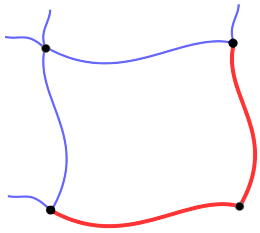
This operation performs like an *face extrusion* operation for a face enclosed by the closed curve: If the curve encloses more than one face, the result is a *group extrusion* of the faces. Again, *undirected quadrangulation* is used in the split region to match the look of a conventional face extrusion. In our case, curves can also be at the boundary of 2-manifold. We give an example for undirected closed curve split operation for boundary curves as shown in Figure 4.8.



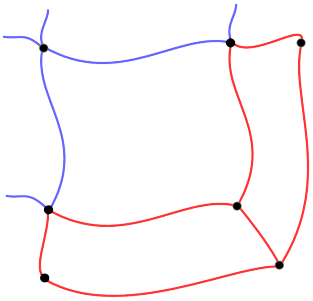
Undirected closed curve selection



Undirected closed curve splitting



Undirected closed boundary curve Selection

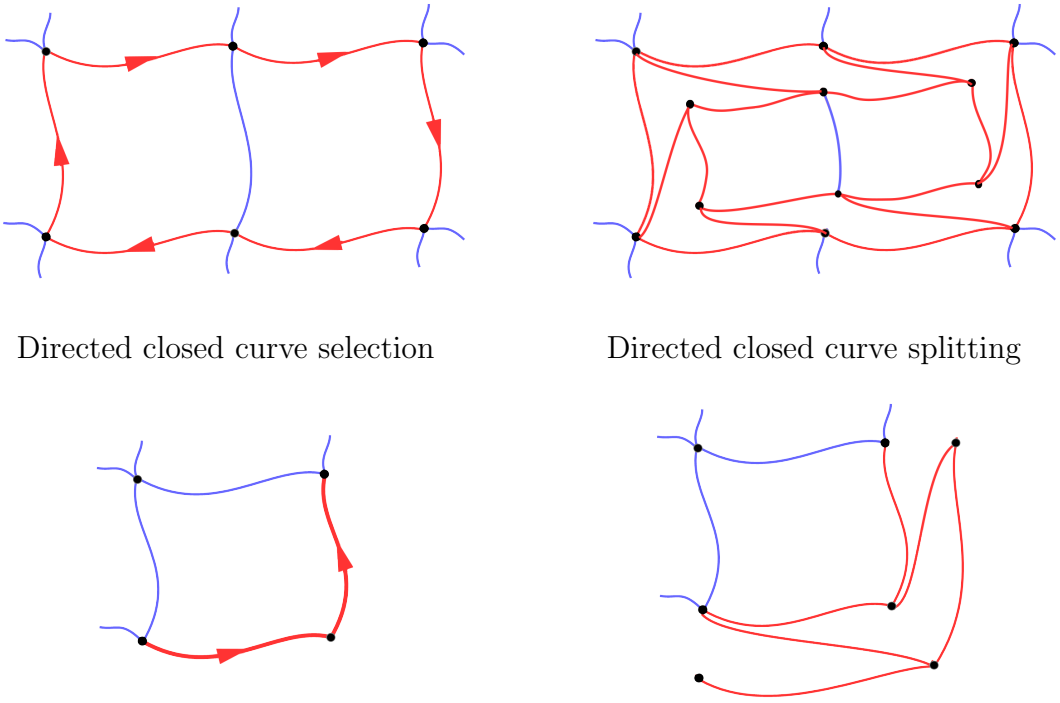


Undirected closed curve splitting

Figure 4.8: Examples of undirected open curve splitting: Note that boundary curve is not really closed, however, since its two end-points are in the boundary, we can interpret it either open or closed.

4.2.1.6 Directed Closed Curve Split:

This operation also performs like an *face extrusion* operation for a face enclosed by the closed curve as if there is a rotation. We also give an example for undirected closed curve split operation for boundary curves as shown in Figure 4.9.



Directed closed curve selection | Directed closed curve splitting

Directed closed boundary curve selection | Directed closed curve splitting

Figure 4.9: Examples of undirected closed curve splitting operation: In boundary, this operation creates a dangling edge. Note that boundary curve is again not closed, but, since its two end-points are in the boundary, we can interpret it either open or closed.

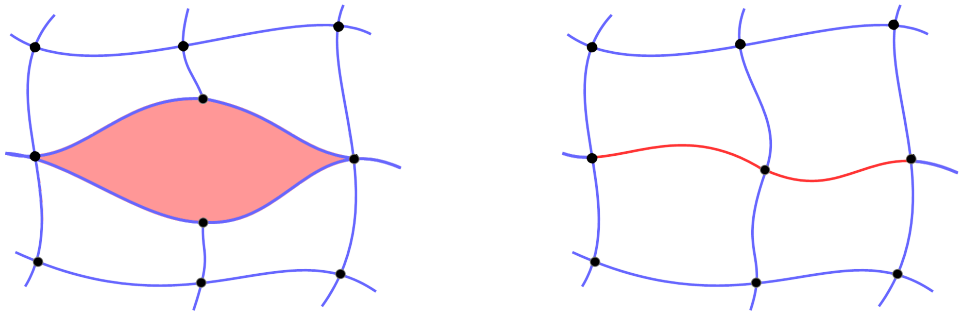


Figure 4.10: Face collapse as a region collapse operation: We select a face and collapse it. The result is not unique.

4.3 Inverse Curve Split: Region Collapse

Inverse of the curve split operations is a single operation that can collapse an entire region into a closed or open curve. Selected regions can form a ring by creating a closed curve. We simply triangulate all the faces in the region by connecting closest edges and collapse all the edges that do not cause the edges in the boundary of the region collapse. Examples are shown in Figures 4.10, 4.11 and 4.12. Note that this operation may not necessarily be unique. For instance, in Figure 4.10 if two diagonals have the same distance, the region can collapse in two different ways with equal probability.

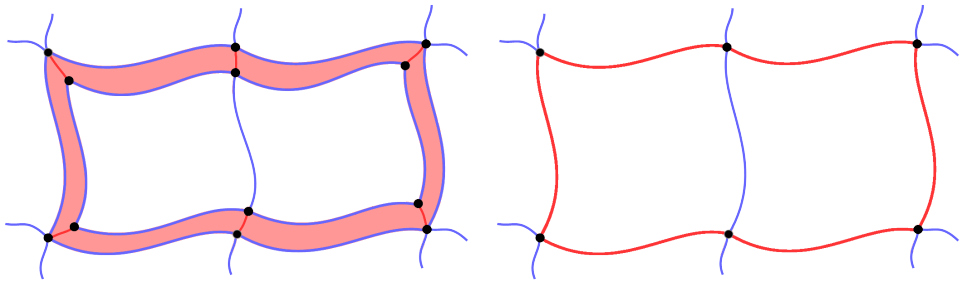


Figure 4.11: An example of ring shaped region collapse.

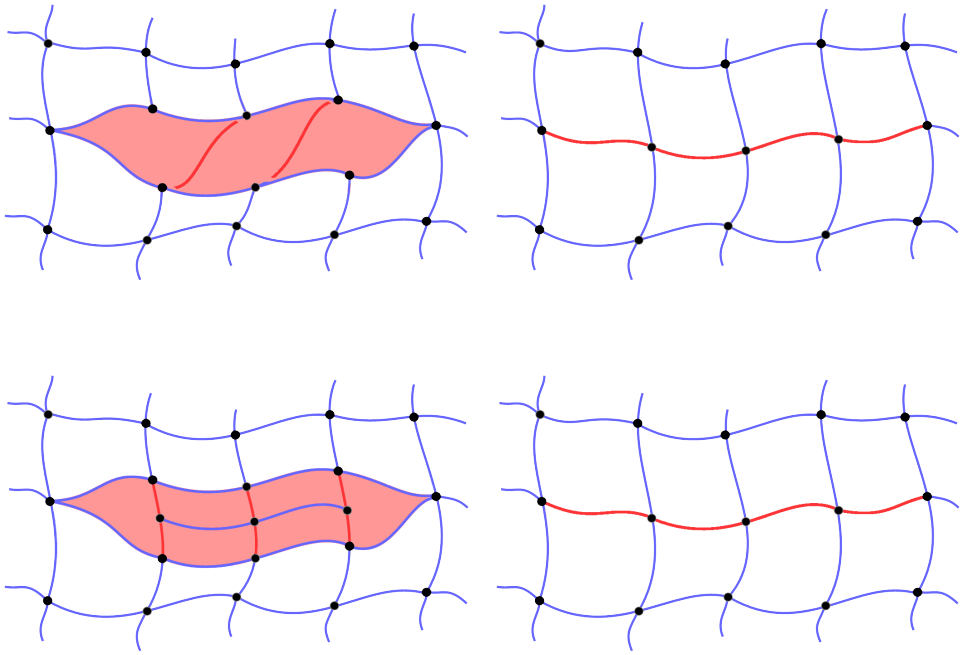


Figure 4.12: Additional examples for region collapsing.

5. HIGH LEVEL OPERATIONS

Here I provide upper level primitives and operations based on the previous chapter.

5.1 Inserting Skeletal Curves

In this framework, a boundary curve is a principal curve consisting one ore more connected outer curved edges in a smooth mesh. It constitutes the contour of the shape by separating the inner and outer faces. When a boundary curve has the same outer face in both of its sides, it is called a *skeletal curve*. A skeletal curve might be in the form of a tree or a graph, and can serve as the skeletal structure for a quad mesh generation. By interactively splitting a boundary curve, it is possible to obtain the initial mesh for the desired model easily.

5.2 Creating Primitives

Modeling a quad dominant mesh begins with a primitive mesh, as in most 3D modelling applications. I provide several primitives to user, which are composed of quad faces.

Down below, I list the type of primitives I provide in the framework. Each type presents a different method of creating a primitive. By tweaking its given parameters, users can modify the the appearance and/or the topology of the primitive created.

The procedure for creating any primitive in the framework is observed in two steps:

- First a skeletal curve for the primitive is inserted with the desired parametrize is inserted
- The skeletal curve is split desired number of times to create the primitives.

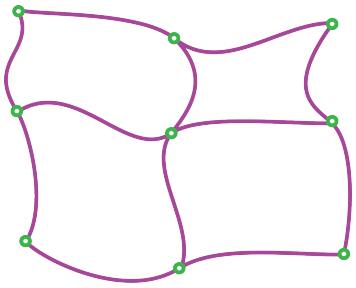


Figure 5.1: A grid of (2×2)

- **Create Grid:** The Grid is the most basic yet probably the most useful type of primitive in the framework. It is the most naive way of creating a quad mesh: a set of quad faces organized in rows and columns. The number of rows and columns along with width and height of a grid face are basic primitives to create a grid.

  Creating a grid via a curve split is fairly simple. For a grid of size $(n \times m)$, we can insert a skeletal curve consisting n number of curved edges and perform an *edge extrusion split* on the curve m times. The procedure for creating a $(2x3)$ grid is shown in Figure 5.2.

- **Create Torus:**

  This type primitive is a donut-like shape in 2D, composed of adjacent quad faces that revolves around a given origin. Topologically, it is a grid whose rows loops back to itself. Note that this mesh has a hole in the middle, which means an additional outer face in the primitive. In addition to the basic grid parameters, the user can modify the radius and percent arc of the torus. For the arcs that are below full circle, the resulting mesh becomes a grid that is bend around an origin, without a hole.

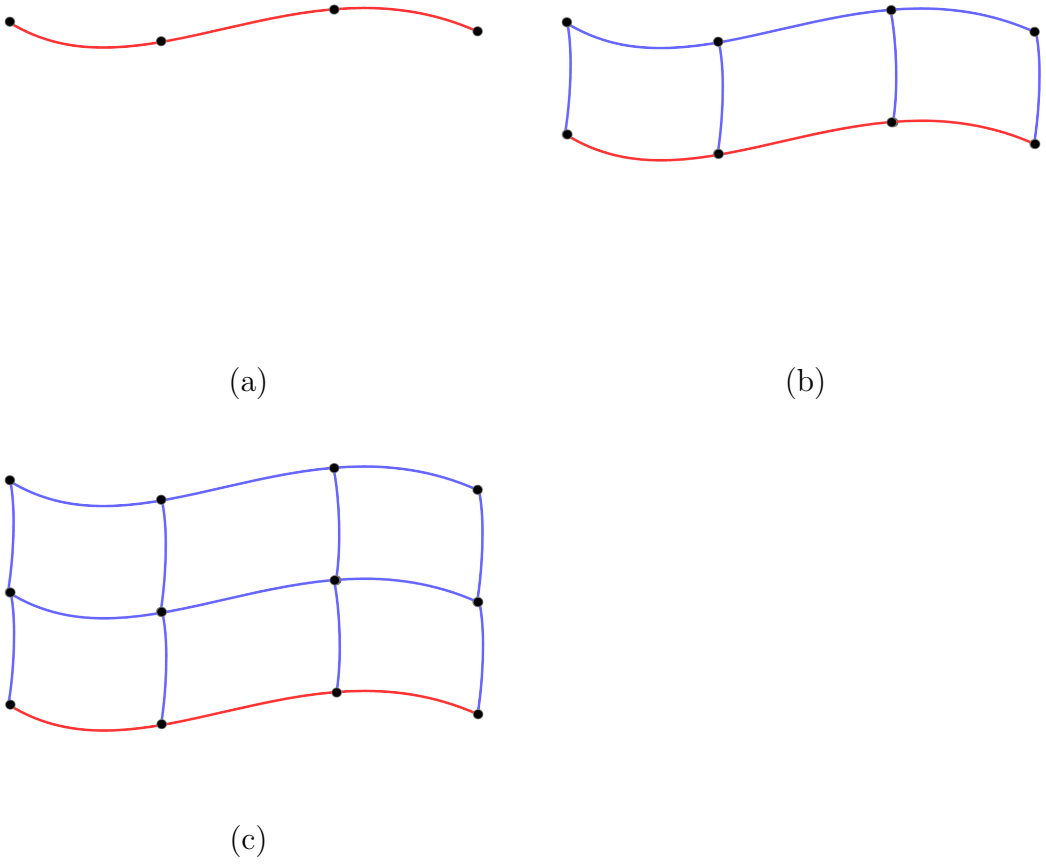


Figure 5.2: Creating a (2×3) grid: A skeletal curve with 3 Curved edges is inserted (a). Two consecutive Undirectionlan Closed Curve Split is performed in (b) and (c)

To create a torus of n segments, we need to insert a insert a skeletal curve with n curved edges and given inner radius. The curve must be closed for a full torus and open for an arc. Undirectional Closed Curve Split is performed on the skeletal curve to obtain a torus. Figure 5.3 shows the creation of a (4×4) torus.

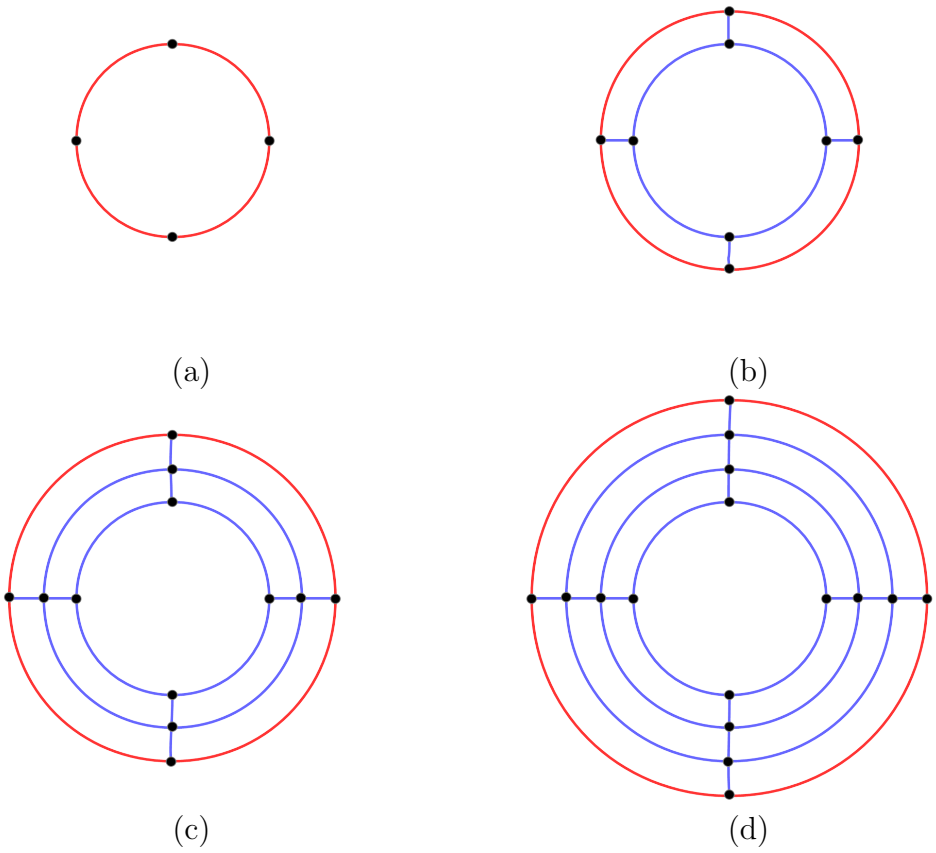


Figure 5.3: Creating a torus by splitting a skeletal curve: (a) Skeletal curve . Applying Curve Split to the skeletal curve for a torus with 4 segments shown in (a) results in a one row torus as in (b). Splitting the boundary curve in (b) and subsequently in (c) will result in a four row tours as in (d).

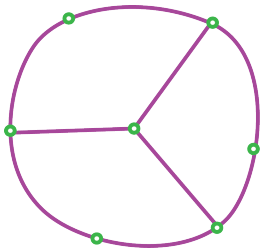


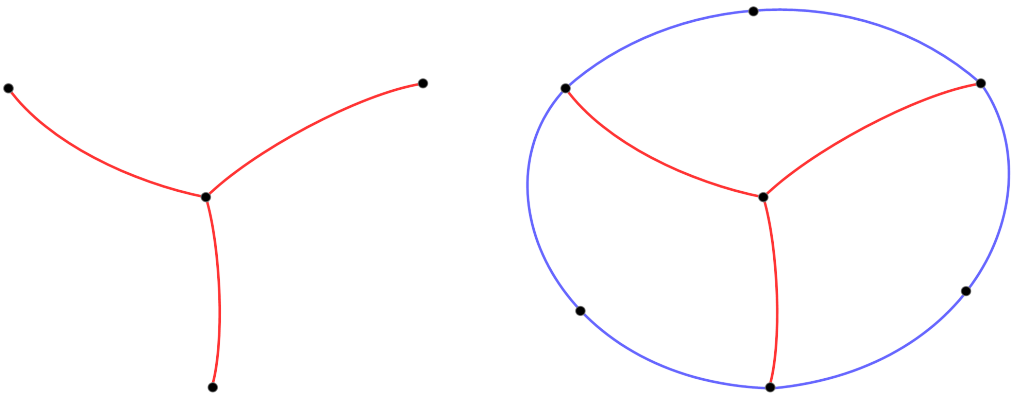
Figure 5.4: 2n-gon

- **Create 2N-gon:** This shape is a polygon with even number of sides, which is created revolving a quad face around a given origin. These faces share a vertex in the middle.

  To create a 2n-gon, we insert a skeletal curve consisting n number of curved edges revolving around the given origin. We then perform Directed Open Curve Split on it to obtain the 2n-gon as in figure 5.5.

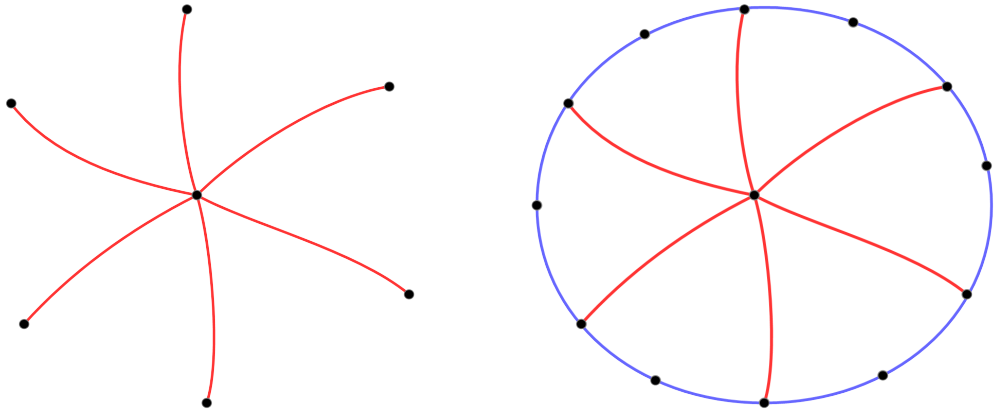
- **Convert Skeleton:**

  This primitive is the generalized form of all primitives. The skeletal curve required is expected to be given by the user and can be in graph form meaning it can have loops and/or dangling curved edges. Any type of split curve operation can be performed on the curve to obtained the desired quad mesh.



The skeletal curve for a 6-gon

Open Curve Split applied



The skeletal curve for a 12-gon

Open Curve Split applied

Figure 5.5: Creating 2n-gon by splitting skeletal curve.

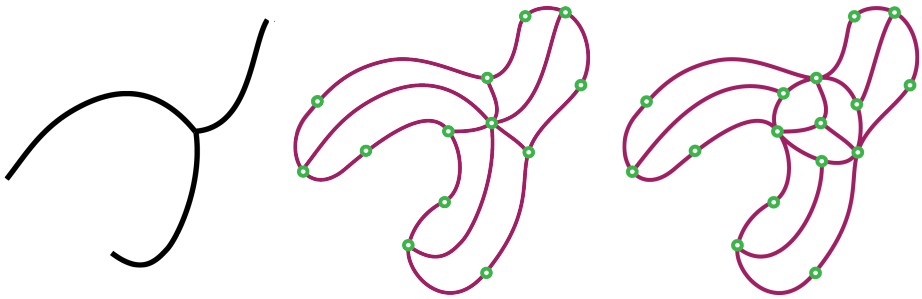


Figure 5.6: Skeleton

6. IMPLEMENTATION

In this section, I discuss how the operations explained in the previous chapters can be implemented.

6.1 Doubly Linked Face List (DLFL) Representation

DLFL is used as the underlying data structure for representing the base mesh in this research. Therefore, it is better to briefly explain the DLFL structure beforehand DLFL stands for *Doubly Linked Face List* and works based on graph rotation system. The DLFL structure consists of a list of vertices, edges and faces. Vertex, edge and face refer to the internal representations of a point in three-dimensional space, a line segment connecting two points and a sequence- of points respectively. For simplicity we can discard making explicit distinction between the internal representation and the actual geometric entity unless required. So, vertex may refer to both the geometric entity as well as the topological entity depending on the context.

For further simplicity, we include an additional entity called *corner*. A corner is a vertex-face pair, $c = v, f$, where v is one of the vertices in f. Since we want a corner to be used in a face boundary walk as $f = v_0, v_1, ..., v_{n-1}$, for the corner c_i referring to vertex v_i, we provide double way links to next and previous corners as $c_i.Next = c_{i+1}$ and $c_i.Prev = c_{i-1}$. A corner is associated with only one face, but several corners can refer to the same vertex.

Internally, each face f is represented as an ordered sequence of corners as $f = \{c_0, c_1, ..., c_n\}$, each of which contains a pointer back to the face as $c.Face = f$. Every corner also has a pointer to the vertex v it refers to as $c.Vertex = v$, and ever vertex points to a corner for a vertex traversal. An edge contains pointers to the two corners as $e = (c_0, c_1)$, each end of the edge and each belonging one of the two faces on each

side of e. Critical to this research, is the direction of the edge defined by the first and second corners as (c_0, c_1), meaning the edge runs form c_0 to c_1.

By using on the minimal set of operators $DLFL$ provides, we can easily implement the modeling operations explained in the previous chapters. Before that, we shall take a more detailed look at the minimal set of fundamental operators in the DLFL.

- $(v, f) =$ **CreateVertex**(p) creates a 2-manifold surface with one vertex v and one face f which will be referred to as a point sphere. The geometric coordinates of the vertex v are given by p which is a point in three-dimensional space. The operation effectively adds a new surface component to the current 2-manifold. The CreateVertex(o)perator is essential in the initial stage of the creation of a new mesh and creates a new surface component in the given 2-manifold. In particular, this operator is necessary when a new surface component is to be created in an empty manifold.

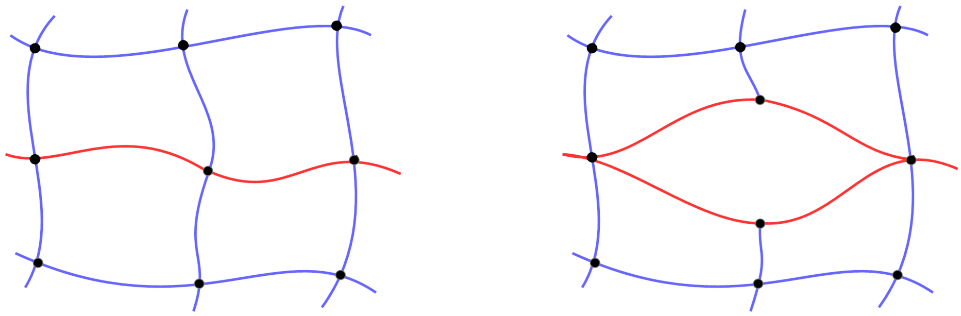
- $(v, f) =$ **DeleteVertex**(v) is the complement of the CreateVertex(o)perator. It removes a point sphere from the mesh structure. If v is not part of a point-sphere,the operator returns without making any changes to the mesh. The operation is the same as the Euler operation KV FS and effectively removes an existing surface component from the current 2-manifold. The DeleteVertex(o)perator is essential for cleaning up the mesh structure to prevent unwanted visual artifacts from appearing.

- $e =$ **InsertEdge**$(c1, c2)$ inserts a new edge e into the mesh structure between two corners c1 and c2. If InsertEdge(i)nserts an edge between two corners of the same face, the new edge divides the face into two faces without changing topology. On the other hand, if InsertEdge(i)nserts an edge between corners of two different faces (this includes the situation in which an endpoint or both

endpoints of the new edge correspond to point spheres), the new edge merges the two faces into one and changes the topology of the 2-manifold.

- **RemoveEdge**(e) deletes the edge e from the mesh structure. This is the inverse of the InsertEdge() operator. In general, if f_1 and f_2 are the faces on either side of the edge e, then deleting e combines f_1 and f_2 into a single face. But if f_1 and f_2 refer to the same face f (as will be the case if e is the result of an InsertEdge() operation between corners of two different faces), then deleting e separates f into two faces, thereby changing the topology of the mesh.

6.2 Quad Preserving Curve Splitting



Selected curved edges Vertex split

Figure 6.1: Quad preserving vertex splitting.

To implement Curve Split operations, we first explain another topological operation called *Vertex Split*. Introduced in [51], *Vertex Split* is an elementary transformation that adds an additional vertex along with two adjacent triangular faces to the

mesh, while updating the attributes of the mesh in the neighborhood of the transformation. In the context of mesh simplification, it is the inverse of *edge collapse* and is used for progressively refining a simplified mesh back to its original.

In this framework, a *quad preserving vertex split* is used to implement the Curve Split operations. The operation figuratively splits a vertex v into two vertices as v_{left} and v_{right}, and the quad dominant structure is preserved by inserting a new quad face in between the two edges e_0 and e_1 of v that are given as reference to determine the sides *left* and *right*. The main difference from [51] is that it introduces a quad face to the mesh instead of two adjacent triangles.

Based on the rotation order, the edges of v that lies from se_0 to e_1 and those from e_1 to e_0 are split as *left* and *right* edges, respectively, while the edges e_0 and e_1 are duplicated as $e_{0'}$ and $e_{1'}$ to remain on both sides. After the operation, a new quad face is created that is enclosed by edges $e_0, e_{0'}, e_{1'}, e_1$ in the rotation order. The resulting v_{left} and v_{right} lies on the diagonal of the quad.

The *vertex split* operation, as described above, can be executed as a high-level operation. Given the two corners $c_0 = v_0, f_0$ and $c_1 = v_1, f_1$, the execution for the SplitVertex(c_0, c_1) via core *DLFL* operations proceeds as follows:

- Let v be the vertex pointed by the next of c_0, meaning the vertex to be split.
- $c' =$ CreateVertex(p). This creates the additional vertex.
- Let E_l be the list of edges pointing to v from the left side, when traversed around v_s from c_1 to c_0.
- Let C_l be the corners of edges in E_l that does not point to v
- For every corner c_i in E_l, InsertEdge(c_i, c')
- For every edge e_i in E_l, RemoveEdge(e_i)

- Let c'_0 be v-next of c_0, InsertEdge(c'_0, c'). This inserts the first additional edge.

- Let c'_1 be edge-pair of c_1, InsertEdge(c'_1, c'). This inserts the second additional edge.

Now, we can discuss the implementation details for the curve split operations using the Vertex Split operation.

- Open Cut Split

- Closed Cut Split

- Open Directed Split

- Open Undirected Split

- Closed Directed Split

- Close Undirected Split

- **Directed Split**

  During interactive modeling, it is essential that the enclosed by the two split-curve-pairs conforms to the existing geometrical structure of the mesh. Therefore, the two split curves must geometrically follows the original curve. The two curves stay apart from each other at a distance to enclose the newly created split-region and meet at the end points, if there are any. For this purpose, right after the topological split, one (or both) of the split-curve-pairs is offset along the curve normal by a given distance. In this context, offsetting a curve C along its normal means that offsetting every control point p_i of C by the normal n_i of (C) at its point p_i, excluding the end points. In general form, the

offset position p' for can be calculated as $p'_i = p_i + n_i t$, where t is a parameter for the distance between split-pairs.

The *vertex split* operation can be efficiently used for executing a *Directed Curve split*. Given a curve C on a mesh, the *directed curve split* of C could be obtained performing a vertex split on each vertex in C.

However, the core operation SplitVertex(c_0, c_1) for vertex splitting is defined over the two corners pointing to the reference edges and modifies the neighboring topology by inserting new edges to the end points of these reference edges. This means that the *execution order* of the operation SplitVertex(c_0, c_1) on the all neighboring (c_0, c_1) pairs in C will effect the final topology; it is possible to obtain different results by executing the SplitVertex(c_0, c_1) in random order of (c_0, c_1) pairs. However, for the curve split operation to be consistent on output, the execution of order should strictly follow C in a given direction. This can be satisfied via a traversal on C.

Using the capabilities of the DLFL structure, we can conveniently perform a traversal on C for executing the split operation. Starting from a given initial corner c_s, the following algorithm for SplitCurve(c_s), performs the operation SplitVertex(c_0, c_1) on the consecutive corner pairs of C, using a *Depth First Search* traversal. Subsequently, this allows the C to be given in a graph form.

SplitCurveDirected$(c_0, c_1 = null)$

$C_{out} \leftarrow \{\}$
if c_1 is not null and is marked as visited **then**
  return
end if

if c_1 is not null **then**

$C_{out} \leftarrow C_{out} \cup \textsc{SplitVertex}(c_0, c_1)$

end if

$c \leftarrow c_1.Next$

repeat

$\textsc{SplitCurveDirected}(c_1, c)$

$c = VertexNext(c)$

until $c \neq c_1$;

return C_{out} =0

- **Undirected Split**

  We can again make use *vertex split* to implement an *undirected split*. The two-side split required for the undirected split operation can be executed by performing a vertex split at each vertex of C on both of its sides. For a consistent execution order for the split vertex operations, we can perform a *closed walk* along the face boundaries of C.

  After the walk, the two split regions will be under *directed quadrangulation*. To obtain the *undirected quadrangulation* required, we should *flip* the edges so that they run between the *split-vertex-pairs*. In terms of *DLFL* structure, the flipping process requires a sequence of InsertEdge() and RemoveEdge() calls throughout the both split regions. For this purpose, as we do the closed walk, we shall keep a list of *DLFL corners* of the faces created after each vertex split, namely return by the SplitVertex(). Then, we can iterate over the list to perform the edge flipping. Note that we do all the edge flips at once after all the vertex splits, since flipping edges on the go would modify the topology. Algorithm 18 outlines the implementation described.

SplitCurveUndirected(c_s)

$C_{out} \leftarrow \{\}$

$c_0 \leftarrow c_s$

repeat

$c_1 \leftarrow FindNextSelectedCorner(c_0.Next)$

if $Other(c_0) = c_1$ **then**

break

else

if c_1 loops back to c_0 **then**

$C_{out} \leftarrow C_{out} \cup$ SplitVertex$(c_0, VertexNext(c_1))$

else

$C_{out} \leftarrow C_{out} \cup$ SplitVertex(c_0, c_1)

end if

end if

$c_0 \leftarrow c_1$

until $c_0.Edge \neq c_s.Edge$;

$E_{del} \leftarrow \{\}$

for $c \in C_{out}$ **do**

$c_{nn} \leftarrow c.Next.Next$

$E_{del} \leftarrow E_{del} \cup Edge(c_{nn})$

InsertEdge(c, c_{nn})

end

for $e \in E_{del}$ **do**

RemoveEdge(e)

end

=0

- **Cut Split** In a cut split, the split region is not quadrangulated but marked as

an outer face. For implementing a cut split, we can simply perform a directed cut on C and then delete all the edges in the split region. We finally mark it as an *outer* face.

CutSplit(C)

$C_{split} \leftarrow$ SplitCurveDirected($C.c_0$)

$f.split \leftarrow C_{split}.c_0.Face$ **for** $c \in C_{split}$ **do**
| RemoveEdge($c.Edge$)

end

$f_{split}.isOuter = true$ =0

- **Boundary Split**

The main difference of this operation from the *undirected split* is that the split is performed only on the boundary side of the curve. Depending on the boundary state of the given curve, this operation proceeds in two ways.

To execute a boundary split operation on a boundary curve, we can first perform a directed split on the curve, then convert the *undirected quadrangulation* in the split region as described in the undirected split implementation. After the directed split, the split region will obviously be under *directed quadrangulation*. We can again apply edge flips for converting it to *undirected quadrangulation*, yet there will be two triangular faces on each end of the split region after the process. However, since these faces are at the boundaries, their boundary edges can simply be subdivided to convert them into quads. Following the edge subdivisions, the mid-vertices should be repositioned based on the general offset calculation.

If the given curve is a non-boundary curve, then following a directed split on the curve, we can convert the split region into an outer face by deleting all

edges in the split region and marking the remaining face in the region as outer an outer face.

The algorithm for the operation proceeds as follows:

SplitCurveBoundary(C)

  $C_{split} \leftarrow$ SplitCurveDirected($C.c_0$)

  if c is boundary **then**

    FlipEdges(C_{split})

    $v_0 \leftarrow$ SubdivideEdge($C_{split}[0]$)

    $v_0.Position = Pos(C_0) + Normal(C_0) * t$

    $v_1 \leftarrow$ SubdivideEdge($C_{split}[n-1]$)

    $v_1.Position = Pos(C_{n-1}) + Normal(C_{n-1}) * t * t$

  else

    for $c \in C_{out}$ **do**

    | RemoveEdge($c.Edge$)

  end if=0

- **Loop Split**

  To obtain a loop split, we can fist perform a directed split then on the given curve and then convert the split region to undirected quadrangulation by a sequence of edge flips, as in undirected split. Algorithm 18 is designed to handle looping cases.

6.3 Quad Preserving Region Collapse

Since region collapse is the inverse operation of the generic curve split, it can be implemented by reversing the implementation of the curve split operations. Since we used the atomic *Vertex Split* operation to implement curve split operations, we need to come up with an inverse atomic operation for region collapse. For this purpose,

we define an operation *Face Collapse*, which does the exact opposite of vertex split collapsing the given face to a single vertex. Since the vertex split operates on two neighboring edges to determine direction, we also need to indicate a direction for the face to be collapsed and it can easily done by picking a *corner* of the face.

For the given corner c, the face collapse operation takes the left and right edges of the corner and reassigns their *other* corners to be the pre-split corners respectively. The pre-split corners for these edges can be easily retrieved by using the rotation system. After the corner assignments, the given face is totally disposed. The *vertex split* operation, as described above should better be implemented as new core operation to the DLFL for efficieny. Given the corners $c = v, f$ the execution for the CollapseFace(c) can be implemented as follows:

- Let f be the face pointed by c, meaning the face to be collapsed.
- Let $e_0 \leftarrow c.PREV.E$ and $e_1 \leftarrow c.E$
- Set $e_0.c_1 \leftarrow c.PREV.PREV.PAIR$
- Set $c.PREV.PREV.PAIR.E \leftarrow e_0$
- Set $c.PREV.PREV.PAIR.V \leftarrow c.V$
- Set $e_1.c_1 \leftarrow c.NEXT.PAIR$
- Set $c.NEXT.PAIR.E \leftarrow e_1$
- Set $c.NEXT.PAIR.V \leftarrow c.V$
- DeleteVertex(c.NEXT.NEXT.V)()
- Remove f and all of its corners

Once we have the CollapseFace(c) operation in hand, the region collapse operation for a directed split region becomes fairly straight forward. Given the region as a set of corners (faces), we iterate through the set of the region and collapse them one by one.

For collapsing undirected regions, we first convert those regions by removing the undirected edges of quadrangulation between the faces of the region and inserting diagonal edges between them directionally. Once the region is converted to be directional, we can apply the region collapse operation as above.

6.4 Quad Preserving Boundary Operations

6.4.1 Inserting Skeletal Curves

The operation InsertSkeletalCurve(c_0, c_1) inserts a single curved edge with the same outer face on both of its sides, which serves as the atomic element for a skeletal curve.

6.4.2 Deleting A Quad Face

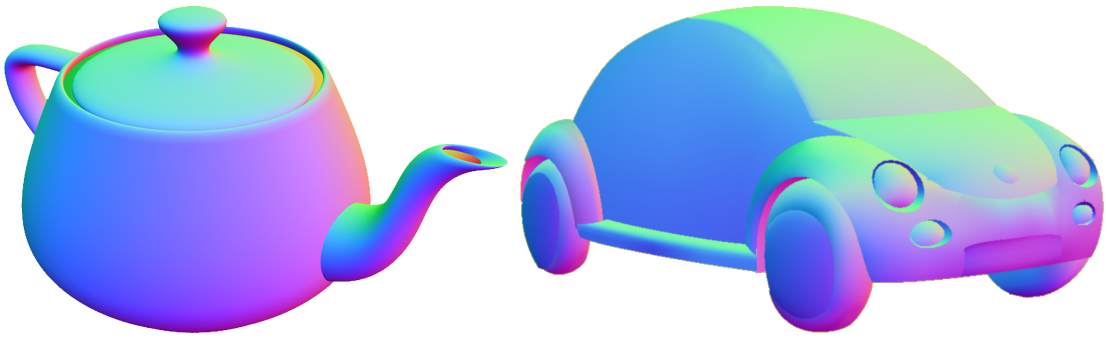
For deleting a quad face, we first iterate through all the edges of the given edge and ad the outer edges to a list. If the list turns out to be empty, we simply mark the face as an outer face and the face is dismissed being rendered on screen. Otherwise we call RemoveEdge per each edge in the list.

7. APPLICATIONS

In this part, I present three main applications of the theoretical framework I propose.

7.1 Modeling Mock-3D Shapes

A major application for this framework is modeling Mock-3D scenes. In 3D computer graphics, a 3D scene is a mathematical description in $\Re^3$ required for the final render. It essentially describes the 3D position and normal data for a given surface point and often times this data is extracted from 3D geometry representations called 3D models. However, to render an image from a fixed point of view, a mock-3D representation can serve as a 3D model by encoding the normal and position data in a 2D vector field. With the mock-3D presentation on hand, we can achieve compelling renders as 7.2.

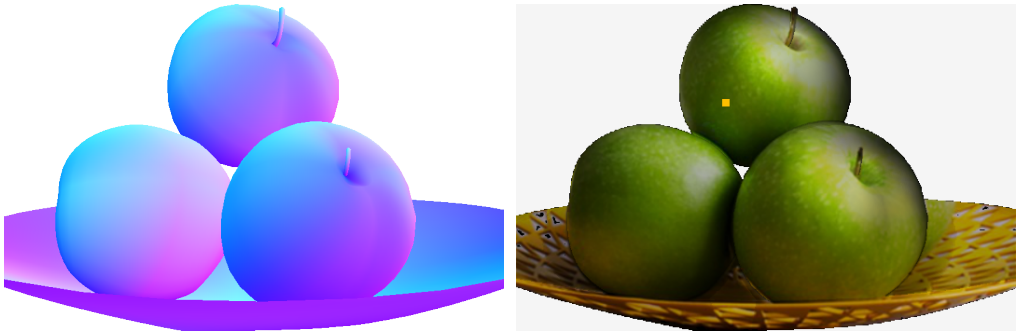


Lumo CrossShade

Figure 7.1: Examples of normal maps generated using sketch based modeling programs.

The most viable option to create a mock-3D representation is to model 2D vector

fields directly with a sketch based interface. As discussed earlier, there already exist many sketch based interface approaches, such as by Lumo [41] or CrossShade [44], that can directly be used to create such representations shown in 7.1. However, to hit the consumer market, there is a need to provide more control to users.



Mock-3D Scene Render

Figure 7.2: An sample rendering of a mock-3d scene.

The framework I present is far better suitable to model such mock-3d representations interactively. There are several reasons behind this. First of all, it allows modeling of any 2D shape. The underlying mesh may come in arbitrary topology with genus. The curve split operations previously explained provide a convenient way of modeling complicated quad mesh structures. On the other hand, the curve network that represent the mesh may allow both sharp and continues boundaries. With the help of both of these features, any 2D shape can be modeled to be used as a mock-3D Model.

7.1.1 Mock-3D Model

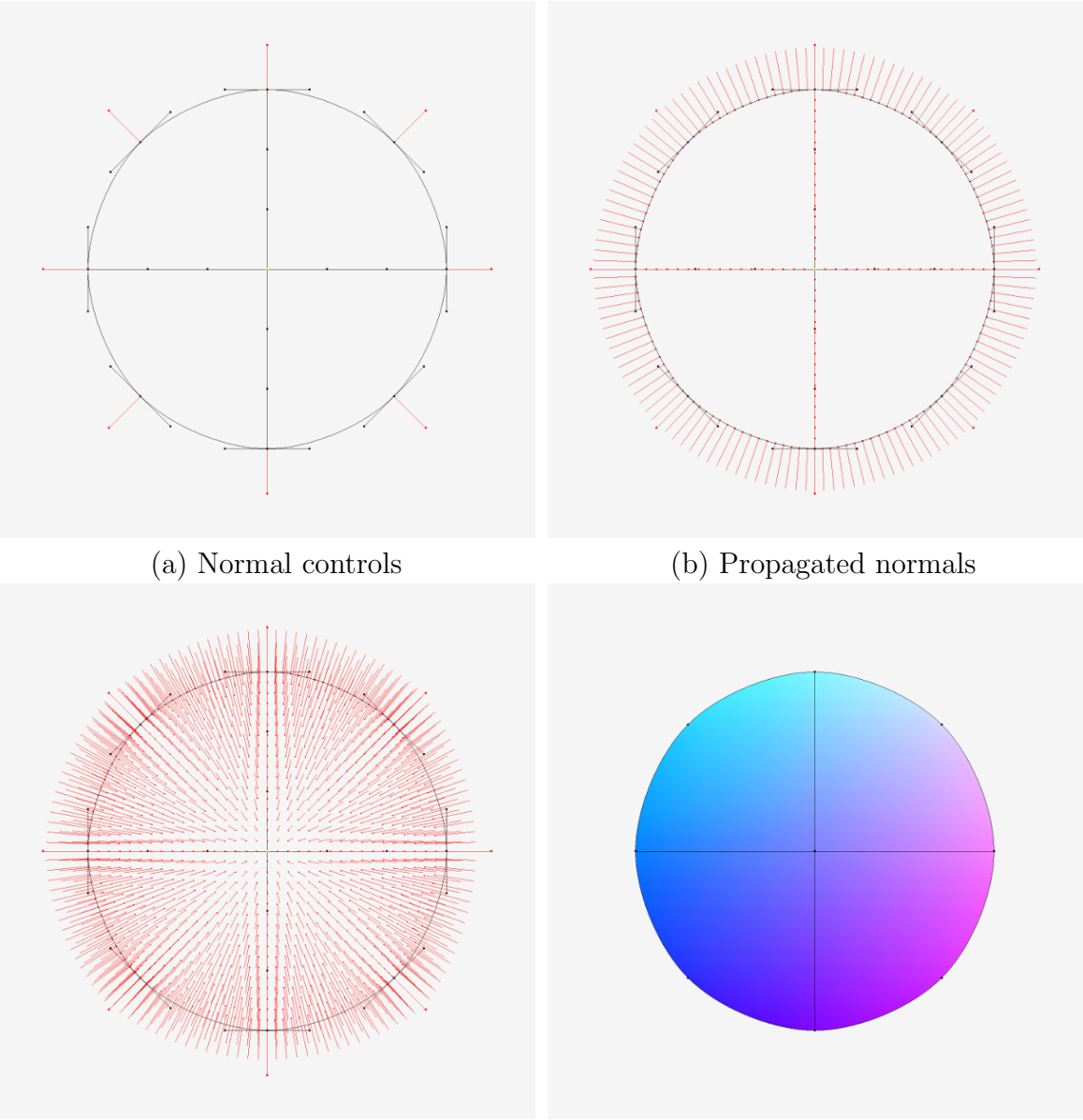
A Mock-3D model is essentially a 2D vector field that encodes the normal data per each 2D position. This vector field can be obtained by interpolating the data

at boundaries of the shapes modeled via the framework presented in this research. However, the data at boundaries do not inherently encode a 3D normal vector required by the mock-3d representation. For this purpose, we use an additional *normal vector* assigned to each vertex of the mesh referred as *vertex normal*.

A vertex normal can either be initially derived from the shape or given by the user. In our application, we initially derive and assign a normal for each vertex and then let the user adjust it via an interface control.

To derive a vertex normal for each vertex, we can use the neighboring curves to the vertex. For a Mock-3D representation, border curves constitutes the silhouette of the shape and be perpendicular to the incoming eye vector by definition. In practice, under the assumption of eye vector being parallel to the viewing plane, we can consider the normals of the boundary curves to be on the viewing plane. This can be easily achieved by converting the 2D normal of the boundary curve into a 3D vector by initializing the z component to 0. For the inner vertices, we assume that they are perpendicular to the viewing plane and simply initialize them to the normal of the viewing plane as $< 0, 0, 1 >$. Figure 7.3 (a) shows the initial normals computed for a 2×2 grid in circular form.

Once the vertex normals are initialized, they can be adjusted by user. In our application, we project vertex normals to the viewing plane as a control widget at the user interface level and let the user drag them around in 2D. Then we update the 3D vertex normal based on the new location of the projected normal. This operation involves calculation a z value based on the length of the normal widget. A full length and a zero length widget indicates $z = 0$ and $z = 1$ respectively. On a boundary curve, if the two boundary tangents of the normal are smooth, they move together to create a smooth curve and the normal to the tangent should follow the same movement for a consistent result. Therefore, when smooth tangents on a boundary



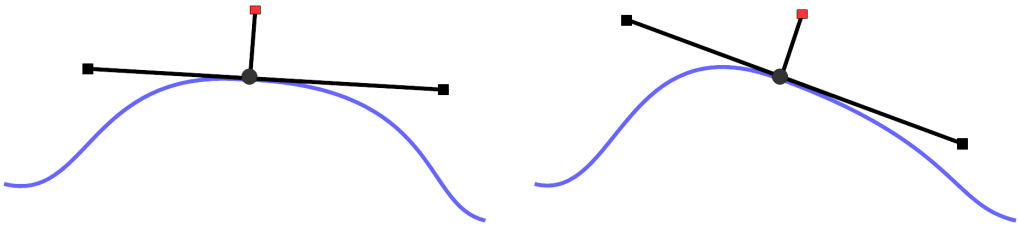
(a) Normal controls

(b) Propagated normals

(c) Interpolated normals

(d) Mapped normals

Figure 7.3: Examples of normal maps generated using sketch based modeling programs.



(a) Vertex normal (b) Rotated vertex normal

Figure 7.4: Rotation of the vertex normal based on the tangent rotation.

curve are dragged by user we figure out the rotation that the tangent went through and apply it to the normal as in Figure 7.4

To create the 2D vector field, we first propagate vertex normals over the boundary curves. Figure 7.3 (b) shows the propagated normals over the boundary curves of the sphere shape. The method for interpolating normal vectors presented in section 3.4 should be followed for this propagation. In a discreet implementation, each curve refers to an array of interpolated normals by equal parametric distances over the curves. Then, using Coonz interpolation, we can fill each face with surface normals by interpolating the normals at the boundary curves. Again, a discrete implementation of Coonz interpolation requires computing a grid of vertex normals.

In addition to the normal vectors, a mock-3D representation may also encode some color information for the shape. In 3D computer graphics, the final color of a surface point is computed by mixing several colors such as ambient, diffuse and specular based on an illumination model. This process is known as shading. A mock-3D representation can benefit from this process to create better results. To create a

finer approximation of the shading process, we need to encode the color information such as ambient, diffuse and specular colors in our *data vector*. This information is again due interpolation in the final mock-3d representation. In our application, we allow user to adjust the color of vertices individually via a channel system, which eventually updates the respective values in their data vectors.

7.1.2 Mock-3D Scene

In our implementation a Mock-3D scene is composed of one or more Mock-3D shapes stacked on top of each other as layers on a canvas. The ordering of the Mock-3D shapes on this stack is important since they can occlude each other based on this order. The shapes to the front of the stack may occlude the shapes at the back. With the help of this feature, user can create mock-3D scenes that are impossible to be represented by only one mock-3d shape. In our implementation, to provide an interactive modeling environment, we allow users to reposition a mock-3D shape on the canvas by dragging it and also reordering it on the stack by moving it up or down.

When a mock-3D scene is composed of more than one mock-3D shape, oftentimes a shape on top maybe an extension to the shape that it overlaps, such as a nose on a face. This is particularly important since modeling this kind of an extension embedded to the model would require to model it as a *dangling face* which is not supported in a manifold mesh. Although it is possible to define an operation to extrude an edge as a manifold surface, this method would cause unnecessary complexity in the mesh by adding extra edges and faces. In the case of a mock-3D application, it is more practical to represent an extended surface as an overlapping shape without increasing complexity of the original shape. However, the extension surface may need to share the data vectors with the original shape to create a smooth transition in

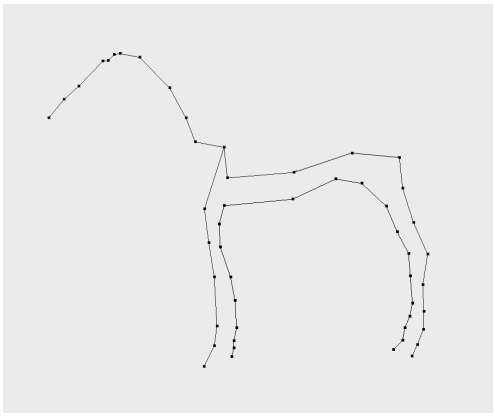
the resulting image. For this reason, we added a special operation called *sewing* that copies the data vectors from the vertices of a selected curve to the vertices of target curve. When a curve on a shape is sewed to a curve another shape, the transition two surfaces in the resulting images is ensure to be fairly smooth since both surfaces share same shading data at the point of transition.

Besides the 2D shapes, a mock-3D scene may also contain lighting information as in a 3D scene. A point light or a directional light can be represented as a 2D point on the canvas with a z-depth value and a 3D light vector respectively. In our implementation point lights are represented by dragable 2D widgets on the canvas.

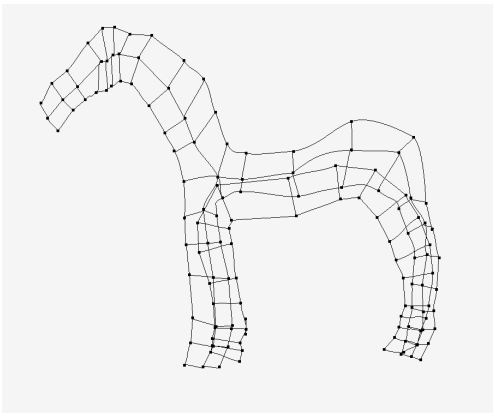
The purpose of modeling a mock-3D scene in this application is eventually achieving a rendered image that is similar to rendering of a 3D model. There are several ways to obtain this rendered image from a Mock-3D representation. Ideally, it is possible to develop a custom render engine that takes the Mock-3D scene as an input and computes a high quality final image based on this scene. In our application, we use the OpenGL render engine for creating a preliminary result. OpenGL rendering can easily be integrated to the implementation of this framework since it allows binding of the shading data (color data and vertex normal) per vertex. With the help of graphics hardware, OpenGL can also render high quality Mock3D scene at interactive rates which allows user to model a Mock3D scene by seeing the final result interactively.

7.1.3 Mock-3D Examples

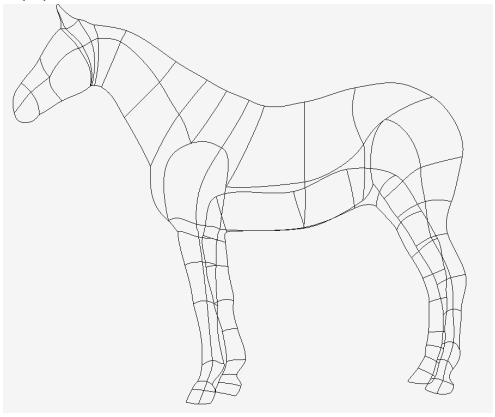
In this section I present some examples of modeled mock-3d scenes by our implementation. Figure 7.5 presents stages of creating a mock-3d model of a horse in our application. First we create two skeletal curves as in 7.5 (a) one for the body and one for the two right legs occluded by the rest of the body. Then we split the first



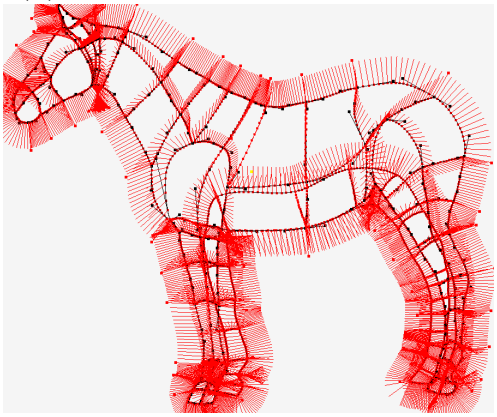
(a) Two skeletons for a horse shape



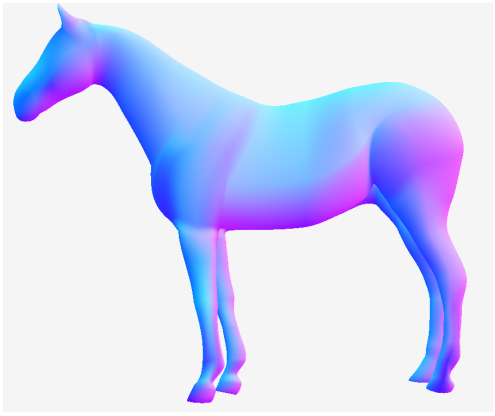
(b) Split curve applied to skeletons



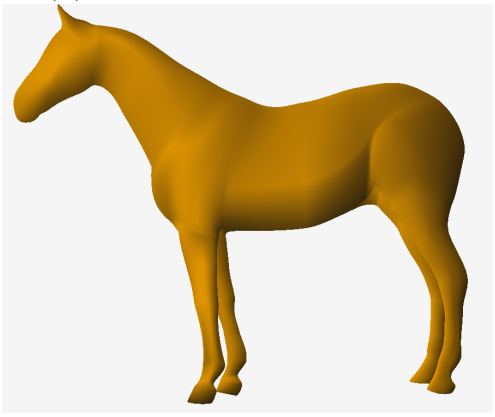
(c) Curves refined to outline a horse figure



(d) Normal vectors propagated



(e) Normal vectors interpolated as normal map



(f) A mock-3d render of the model

Figure 7.5: Stages of creating a mock-3d horse model.

curve on both sides, and second curve on one side to create a quad mesh structure to serve as the rough model. Right after the splits, we obtain two disjoint quad meshes as in 7.5 (b). We then create the outlines of a horse by moving the control points of the curves to desired positions as in 7.5 (c). When we are satisfied with the outline, we adjust the vertex normals so that we can obtain a surface that will give us the look of an horse's body. This is an iterative process that usually requires checking the preview render frequently. Note that although we use a single mesh to represent the head, body and the two left legs, we can still control the shading that distinguishes the legs and the head from the body by adjusting their normals. Figure 7.5 (d) shows the propagated normals for all the curves. In (e) and (f), we see the normal map resulting via interpolation of normals and an OpenGL render of the Mock-3d models respectively.

Figure 7.6 demonstrates how a simple cartoon character can be created by just using overlapping grid shapes. The figure contains *ten* disjoint grid shapes which eventually overlap in a way to create the Mock-3D representation of the cartoon character shown. Note that the nose of the character is also a disjoint shape from the forehead but is seamlessly blended to the character's forehead face using the *sew* tool.

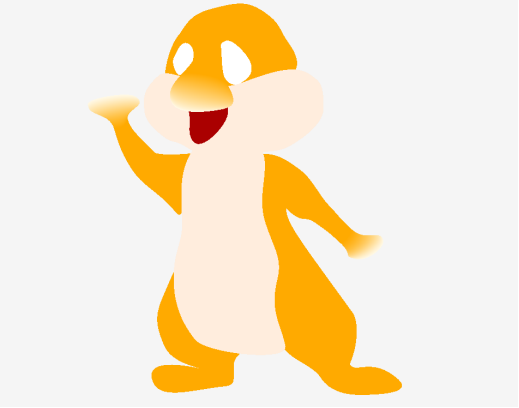
7.2 Origami Modeling

The framework can serve as a fold pattern design tool for origami modeling. An origami model is obtained by folding a flat surface down (valley) or upwards (hill) through the fold lines. These lines form a crease pattern on the surface that defines the origami model. It is possible to generate well-known crease patterns and their curved versions using this framework.

The main reason behind it is that the surface of the shape modeled is ensured to

(a) Overlapping disjoint grids

(b) Normal maps using grids

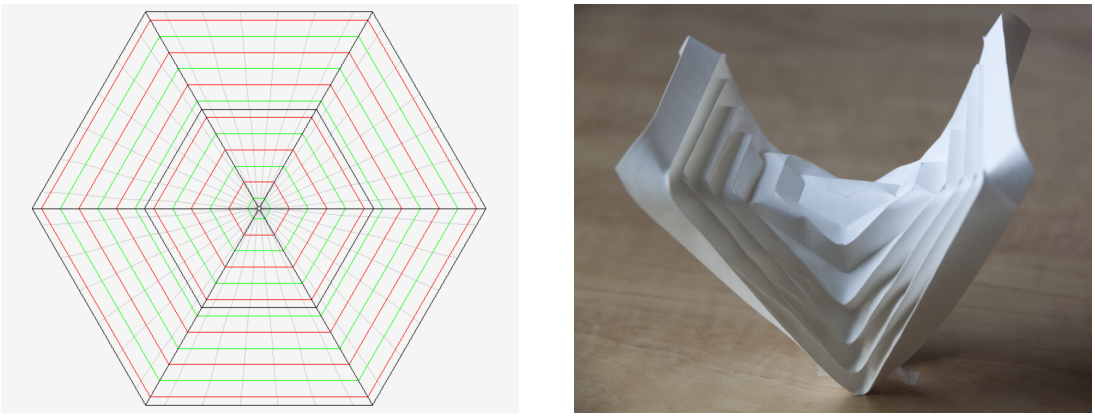


(c) Diffuse colors

(d) OpenGL render based on normal maps

Figure 7.6: A cartoon character modelled via grids.

be globally parametric. It is because every quad face in the mesh is locally represented by a parametric surface as a function of $f(u, v)$ and that we could establish a directional connectivity between adjacent quads for both parameters. The directional connectivity is achieved via the data structure that represents the underlying mesh. Once we represent each face by parametrically spaced curves in both *u,v* directions, for a given direction, we can create a given crease pattern by assigning each curve along the direction as hill, valley or flat one by one. In our system, we encode the flat, hill and valley regions via digits *0, 1, 2* respectively which provides to represent a crease pattern as a numerical array. In the application side, we developed a tool named *Assign Pattern* that assigns the input creases pattern as a numerical string to the shape along the direction indicated by a user specified *curve*.

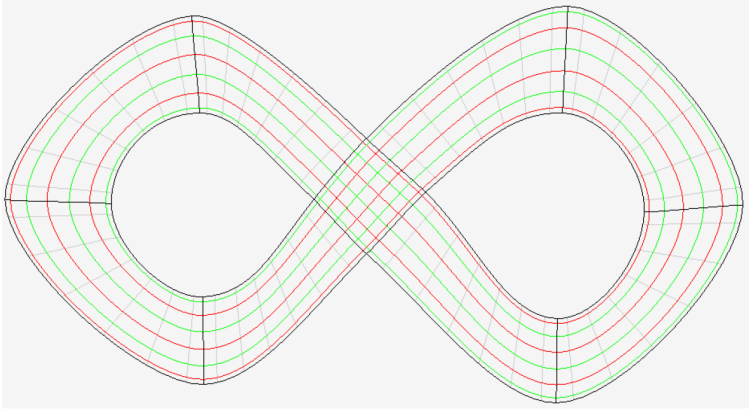


(a) A hexagonal shape via 2ngon primitive (b) Fold as hexagonal parabola

Figure 7.7: A hexagonal parabola made from the hexagon shape.

Figures 7.7 and 7.7 shows some examples of origami created by MS student H W Kung [52] as a part of his MS thesis via our application. In Figure 7.7 (a), we see a hexagonal shape modelled using the *2NGon* primitive type in the framework.

The hexagon represents *1,2* (hill-valley) fold pattern assigned in v direction from boundary edges to the center. The crease lines are color coded as *red* and *green* representing *hill* and *valley* respectively. In figure 7.7 (b) a the shape with the fold pattern from (a) is turned into a paper origami in hexagonal parabola form by folding it through the crease lines. In the process, a laser cutter was used to engrave the fold lines on paper. Figure 7.8 is a similar example of the same process demonstrating an *8* figure modelled via skeletal curve. The example again uses simple *1,2* fold pattern.



(a)A freeform shape via skeletal curve

(b)Shape in origami form

Figure 7.8: A hexagonal parabola made from the hexagon shape.

Our implementing of this framework was also used in a recent study involving patterned self-folding reconfigurable structures [53]. The fold patterns designed are converted into finite element meshes that can be analyzed in finite element analysis (FEA) software considering the thermomechanically-coupled constitutive response of the SMA material. Finite element simulations are performed to determine whether by appropriately heating the planar unfolded surface it is possible to fold it into the

desired structure. Figure 7.9 shows a self-folding torus shape via thermal stimulus. The torus shape with *(1,0)* fold pattern towards to its center was modelled and exported from our system.

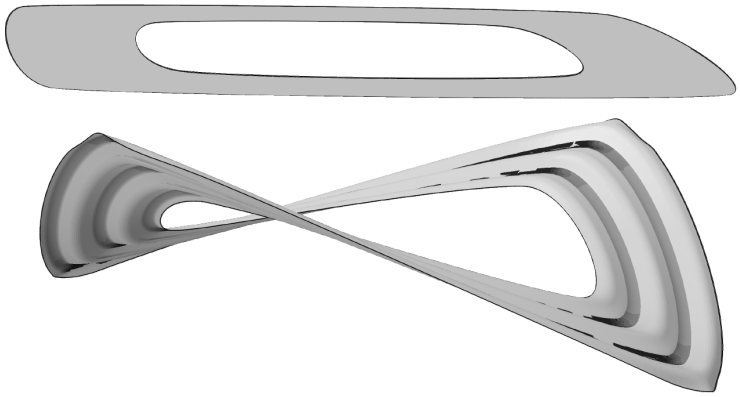


Figure 7.9: A self-folding torus shape via thermal stimulus.

7.3 Image Vectorization

Image vectorization is a commonly used technique in the graphic design community to create illustration-like images. In this process, the graphic artist takes a photo as a base image and tries to recreate a vector representation for the underlying photo by using vector graphic tools.

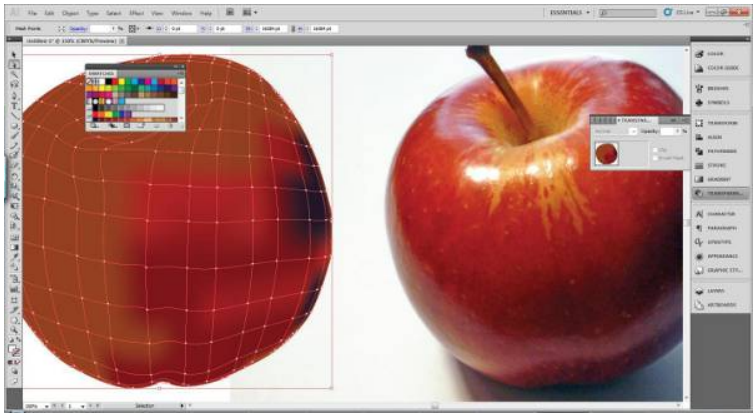


Figure 7.10: Gradient mesh tool in Adobe Photoshop being used for image vectorization.

Gradient mesh tool in Adobe Photoshop is one if the common tools used in image vectorization. It is basically a rectangular Bezier patch which creates a color field by interpolating the color values at its vertices as in seen Figure 7.10. The user can subdivide the patch in either u or v direction to additional detail. However, other operations such as edge/face extrusion or face deletion, therefore cannot have any arbitrary topology to represent any real life object. This leads users to work in disjoint pieces and manually stitch them meanwhile which ends up being a very time consuming and labor intense process.

(a) Reference image (b) Vectorized image

Figure 7.11: An example of application in image vectorization.

The meshes created by the framework I propose comes in quad dominant in arbitrary topology, making it very suitable for this purpose. The reference shape to be vectorized can easily be model as a single mesh without any need for stitching. With the operation as extrude/delete face, user can easily create openings such as nose/mount or eye openings in shapes like human face. Segment and Seam insertion operations helps to add detail to the model.

Once the modeling phase is done, vectorization step is fairly simple and automated. Each vertex of the mesh is assigned the dominant color value retrieved from the area in the reference image where the vertex maps. The Coons interpolation of the color values of the vertices give us a color filed, which is a vectorized version of the reference image. Figure 7.11 demonstrates an example for the image vectorization created in our application using a relatively low resolution mesh. The result can possibly be improved by using a finer mesh that samples the reference with better resolution.

8. CONCLUSIONS

In this dissertation, I presented a theoretical framework to model 2D meshes which are quad dominant in structure, meaning composed of mostly quadrilateral faces. The motivation behind the quad face restriction is to be able to use existing manifold parametrization methods such as Coons or B-spline patch to obtain smooth surfaces. A smooth surface is essentially composed of principle curves, and I observed that all quad-mesh preserving operations can be considered as operators that manipulates on these curves. Based on this observation, I introduced a generic *Curve Split* operation that splits a principle curve while maintaining quad dominant structure of the mesh. I derived child operations from the generic curve split operation based on the method of quadrangulation it used. I also introduced *Region Collapse* as the inverse operation of *Curve Split*. I provided implementation guidelines for all the operations.

The overall smooth 2D shape we obtain in this framework can be used in many applications. In this proposal, I presented three major applications for this framework: (1) Mock-3D scene representation (2) Image vectorization (3) Origami modeling. I have implemented a prototype modeling software in C++ to demonstrate these applications which serves as a proof of concept for the theoretical framework. I presented some preliminary examples for these applications created by this software.

On the applications side, the framework is suitable to be implemented in web or mobile based systems. Particularly, the Mock-3D scene application side, one can envision a future where static pictorial documents are converted into dynamic forms that can be accessible and continuously enriched by everybody. To reach this goal, there is a need for the development of (1) a powerful representation that supports general dynamic documents with re-renderable elements and (2) semi-automatic and

simple to use methods for turning static documents into dynamic documents. In this dissertation, I provided a theoretical infrastructure for the development of web-based systems such that, without any additional tool, people can turn their illustrations, artworks, photographs or cartoons to "html-like" documents that can dynamically be rendered, viewed or manipulated in any device. Furthermore custom offline render engines can be developed for the Mock3D application to achieve more compelling results.

Figure 8.1: A scene from "The Peanuts Movie" (c) 2015, 20th Century Fox.

In the commercial animation world, we occasionally observe stylistic hybrid attempts between classical 2D and 3D animation. "The Peanuts Movie" can be considered as an example of this style (8.1). The Mock3D modeling application we present can be improved to be turned into a Mock3D animation package to create animations in this style. This tool can help classical 2D animation artists to create 3D looking animations without worrying about shading.

One major path for future work would be extending the framework to the 3D platform for modeling smooth objects in 3D without use of subdivision or remeshing schemes. Direct use of Bezier curves in a 3D modeling framework would be a very powerful feature, since it enables convenient modeling of sharp and smooth features in a mesh simultaneously. A practical application for extension to the 3D platform is modeling the salt bodies to be used in reservoir modeling applications. To the best

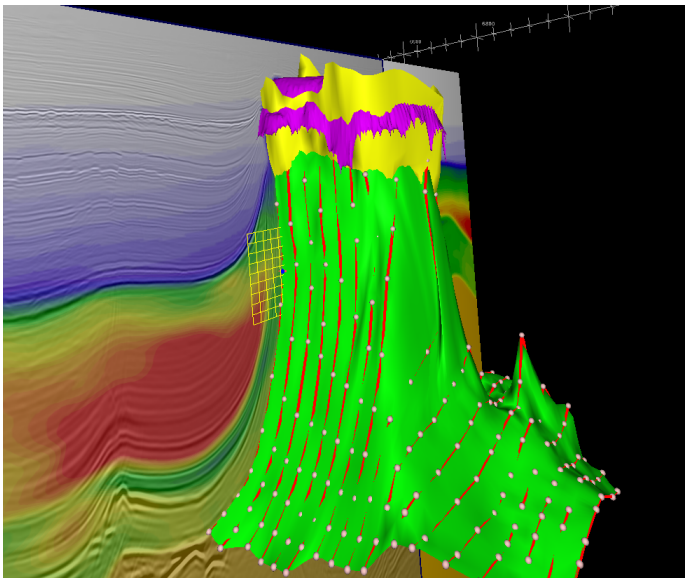


Figure 8.2: Modeling of a salt body starts with extracting silhouette curves from a seismic image.

of our knowledge, currently, there is no standard package or framework designed for salt body modelling in the industry. Engineers working in the area use combination of several methods and packages to overcome this challenging problem. Most of the time, Silhouette curves form the seismic imaging data is extracted manually and imported to a 3D modelling application to create 3D model (see figure 8.2). However, the model looses accuracy in this process since the resulting model may not follow the actual shape in between curves. Since the framework we present here builds the model directly with outline curves, it can capture all outline features of a reference image and therefore is perfectly suitable for modelling salt bodies based on seismic imagining data. Through my internship with *Hue Technology NA LLC*, a software company that provides seismic imagining solutions for oil and gas industry, I have observed that a 3D extension of this framework can significantly reduce workload and improve accuracy in the salt body modelling process.

REFERENCES

[1] V. L. Hansen, *Geometry in Nature*. Wellesley, MA: A K Peters, Ltd, 1993.

[2] G. Barequet and S. Kumar, "Repairing cad models," in *Visualization '97., Proceedings*, pp. 363–370, Oct 1997.

[3] K. Weiler, "Edge-based data structures for solid modeling in curved-surface environments," *IEEE Computer Graphics and Applications*, vol. 5.

[4] M. Mantyla, *An Introduction to Solid Modeling*. Rockville, MA: Computer Science Press, 1988.

[5] B. Baumgart, "Winged-edge polyhedron representation," Tech. Rep., Stanford University, 1972.

[6] L. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of voronoi," *ACM Trans. Graph.*, vol. 4, pp. 74–123, Apr. 1985.

[7] G. Vanecek, *Set Operations on Polyhedra Using Decomposition Methods*. Ph.D. dissertation, McGill University, Montreal, Canada, 1989.

[8] M. Karasick, *On The Representation and Manipulation of Rigid Solids*. Ph.D. dissertation, University of Maryland, College Park, MD, 1988.

[9] K. Weiler, *Topological structures for Geometric Modeling*. Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, NY, 1986.

[10] C. Hoffmann, *Geometric and Solid Modeling, An Introduction*. San Mateo, CA: Morgan Kaufman Publishers, 1989.

[11] E. Akleman and J. Chen, "Guaranteeing the 2-manifold property for meshes with doubly linked face list," *International Journal of Shape Modeling*, vol. 5, no. 2, pp. 149–177, 1999.

[12] J. C. Beatty and B. A. Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. San Mateo, CA: Morgan Kaufman Publishers, 1987.

[13] E. Catmull and J. Clark, "Recursively generated b-spline surfaces on arbitrary topological meshes," *Computer-Aided Design*, vol. 10, no. 6, pp. 350 – 355, 1978.

[14] D. Doo and M. Sabin, "Seminal graphics," ch. Behaviour of Recursive Division Surfaces Near Extraordinary Points, pp. 177–181, New York, NY, USA: ACM, 1998.

[15] T. Gurung, D. Laney, P. Lindstrom, and J. Rossignac, "Squad: Compact representation for triangle meshes," *Computer Graphics Forum*, vol. 30, no. 2, pp. 355–364, 2011.

[16] J. F. Remacle, J. Lambrechts, B. Seny, E. Marchandise, A. Johnen, and C. Geuzainet, "Blossom-quad: A non-uniform quadrilateral mesh generator using a minimum-cost perfect-matching algorithm," *International Journal for Numerical Methods in Engineering*, vol. 89, no. 9, pp. 1102–1119, 2012.

[17] M. Tarini, N. Pietroni, P. Cignoni, D. Panozzo, and E. Puppo, "Practical quad mesh simplification," *Computer Graphics Forum (Special Issue of Eurographics 2010 Conference)*, vol. 29, no. 2, pp. 407–418, 2010.

[18] I. Boier-Martin, H. Rushmeier, and J. Jin, "Parameterization of triangle meshes over quadrilateral domains," in *Proceedings of the 2004 Eurographics/ACM SIG-*

GRAPH Symposium on Geometry Processing, SGP '04, (New York, NY, USA), pp. 193–203, ACM, 2004.

[19] J. Daniels, C. T. Silva, and E. Cohen, "Semi-regular quadrilateral-only remeshing from simplified base domains," *Computer Graphics Forum*, vol. 28, no. 5, pp. 1427–1435, 2009.

[20] O. Gonen and E. Akleman, "Sketch based 3d modeling with curvature classification," *Computers & Graphics 2012*, vol. 36, no. 5, pp. 521–525, 2012.

[21] S. Dong, P.-T. Bremer, M. Garland, V. Pascucci, and J. C. Hart, "Spectral surface quadrangulation," *ACM Trans. Graph.*, vol. 25, pp. 1057–1066, July 2006.

[22] A. Nasri, M. Sabin, and Z. Yasseen, "Fillingn-sided regions by quad meshes for subdivision surfaces," *Computer Graphics Forum*, vol. 28, no. 6, pp. 1644–1658, 2009.

[23] S. Schaefer, J. Warren, and D. Zorin, "Lofting curve networks using subdivision surfaces," in *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '04, (New York, NY, USA), pp. 103–114, ACM, 2004.

[24] P. Alliez, D. Cohen-Steiner, O. Devillers, B. Lévy, and M. Desbrun, "Anisotropic polygonal remeshing," *ACM Trans. Graph.*, vol. 22, pp. 485–493, July 2003.

[25] M. Marinov and L. Kobbelt, "Direct anisotropic quad-dominant remeshing," in *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, PG '04, (Washington, DC, USA), pp. 207–216, IEEE Computer Society, 2004.

[26] S. Dong, S. Kircher, and M. Garland, "Harmonic functions for quadrilateral remeshing of arbitrary manifolds," *Computer Aided Geometric Design*, vol. 22, no. 5, pp. 392 – 423, 2005. Geometry Processing.

[27] E. Zhang, K. Mischaikow, and G. Turk, "Vector field design on surfaces," *ACM Trans. Graph.*, vol. 25, pp. 1294–1326, Oct. 2006.

[28] Y. Tong, P. Alliez, D. Cohen-Steiner, and M. Desbrun, "Designing quadrangulations with discrete harmonic forms," in *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, SGP '06, (Aire-la-Ville, Switzerland, Switzerland), pp. 201–210, Eurographics Association, 2006.

[29] J. Palacios and E. Zhang, "Rotational symmetry field design on surfaces," *ACM Trans. Graph.*, vol. 26, July 2007.

[30] N. Ray, B. Vallet, W. C. Li, and B. Lévy, "N-symmetry direction field design," *ACM Trans. Graph.*, vol. 27, pp. 10:1–10:13, May 2008.

[31] N. Ray, B. Vallet, L. Alonso, and B. Levy, "Geometry-aware direction field processing," *ACM Trans. Graph.*, vol. 29, pp. 1:1–1:11, Dec. 2009.

[32] M. Marinov and L. Kobbelt, "A robust two-step procedure for quad-dominant remeshing," *Computer Graphics Forum*, vol. 25, no. 3, pp. 537–546, 2006.

[33] S. Maza, F. Noel, and J. Leon, "Generation of quadrilateral meshes on free-form surfaces," *Computers Structures*, vol. 71, no. 5, pp. 505 – 524, 1999.

[34] C.-H. Peng, E. Zhang, Y. Kobayashi, and P. Wonka, "Connectivity editing for quadrilateral meshes," *ACM Trans. Graph.*, vol. 30, pp. 141:1–141:12, Dec. 2011.

[35] T. D. Blacker and M. B. Stephenson, "Paving: A new approach to automated quadrilateral mesh generation," *International Journal for Numerical Methods in Engineering*, vol. 32, no. 4, pp. 811–847, 1991.

[36] D. R. White and P. Kinney, "Redesign of the Paving Algorithm: Robustness Enhancements through Element by Element Meshing," in *International Meshing Roundtable*, 1997.

[37] C. Park, J.-S. Noh, I.-S. Jang, and J. M. Kang, "A new automated scheme of quadrilateral mesh generation for randomly distributed line constraints," *Comput. Aided Des.*, vol. 39, pp. 258–267, Apr. 2007.

[38] T. Weyrich, J. Deng, C. Barnes, S. Rusinkiewicz, and A. Finkelstein, "Digital bas-relief from 3d scenes," in *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, 2007.

[39] H. Bezerra, B. Feijo, and L. Velho, "An image-based shading pipeline for 2d animation," in *Computer Graphics and Image Processing, 2005. SIBGRAPI 2005. 18th Brazilian Symposium on*, pp. 307–314, 2005.

[40] J. Cohen, M. Olano, and D. Manocha, "Appearance-preserving simplification," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pp. 115–122, 1998.

[41] S. F. Johnston, "Lumo: Illumination for cel animation," in *Proceedings of the 2nd International Symposium on Non-photorealistic Animation and Rendering*, NPAR '02, pp. 45–52, 2002.

[42] M. Okabe, G. Zeng, Y. Matsushita, T. Igarashi, L. Quan, and H.-Y. Shum, "Single-view relighting with normal map painting," *Proceedings of Pacific Graphics*, pp. 27–34, 2006.

[43] H. Winnemoeller, A. Orzan, L. Boissieux, and J. Thollot, "Texture design and draping in 2d images," *Computer Graphics Forum*, vol. 28, no. 4, pp. 1091–1099, 2009.

[44] C. Shao, A. Bousseau, A. Sheffer, and K. Singh, "Crossshade: Shading concept sketches using cross-section curves," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 45:1–45:11, 2012.

[45] J. Sun, L. Liang, F. Wen, and H. Shum, "Image vectorization using optimized gradient meshes," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 11, pp. 11:1–11:7, 2007.

[46] A. Orzan, A. Bousseau, H. Winnemoller, P. Barla, J. Thollot, and D. Salesin, "Diffusion curves: A vector representation for smooth-shaded images," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3, pp. 92:1–92:8, 2008.

[47] D. Sýkora, J. Dingliana, and S. Collins, "Lazy- brush: Flexible painting tool for hand-drawn cartoons," *Computer Graphics Forum*, vol. 28, no. 2, pp. 599–608, 2009.

[48] M. Finch, J. Snyder, and H. Hoppe, "Freeform vector graphics with controlled thin-plate splines.," *ACM Transactions on Graphics (TOG)*, vol. 30, pp. 166:1–166:10, 2011.

[49] T. Wu, C. Tang, M. Brown, and H. Shum, "Shapepalettes: Interactive normal transfer via sketching.," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3, pp. 44:1–44:5, 2007.

[50] R. Vergne, P. Barla, R. W. Fleming, and X. Granier, "Surface flows for image-based shading design," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 94, pp. 94:1–94:9, 2012.

[51] H. Hoppe, "Progressive meshes," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, (New York, NY, USA), pp. 99–108, ACM, 1996.

[52] H.-W. Kung, "Curved pattern origami," masters thesis, Texas AM University, College Station, TX, 2014.

[53] E. A. P. Hernandez, D. J. Hartl, R. J. Malak, E. Akleman, O. Gonen, and H.-W. Kung, "Design tools for patterned self-folding reconfigurable structures based on programmable active laminates," *Journal of Mechanisms and Robotics*, vol. 8, no. 3, p. 031015, 2016.