Master thesis: Study of mesh generation and optimization techniques applied to parametric forms

Auteur: Vandenbergh, Julien
Promoteur(s): Béchet, Eric
Faculté: Faculté des Sciences appliquées
Diplôme: Master en sciences informatiques, à finalité spécialisée en "intelligent systems"
Année académique: 2019-2020
URI/URL: http://hdl.handle.net/2268.2/8637

Avertissement à l'attention des usagers:

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative" (BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.
University of Liège - Faculty of Applied Sciences

Master thesis
ATFE0002-1

Study of mesh generation and optimization techniques applied to parametric forms

Graduation Studies conducted for obtaining the Master’s degree in Computer science by Julien Vandenbergh

Academic promoter: Éric Béchet
Industrial promoters: Sébastien Magdelyns & Terry Moës

Academic year of 2019-2020
Abstract

This master thesis is focused on the creation of 3D meshes from parametric forms, in particular texts and fonts. The algorithm developed in this work is focused towards being integrated into a real-time rendering engine. This master thesis is done with the collaboration of the University of Liège and Deltatec.

The first step is to find a way to extract the parametric form of the text to be drawn. This extraction is based on a previous R&D done at Deltatec. It is using Direct Write and Direct 2D, both are libraries provided by DirectX.

Once the parametric forms that describe the text is extracted, a discretisation step is done to create concrete points in 2D space. Those points are then used to create a mesh by triangulating them either using Monotone Polygons Triangulation or by Constrained Delaunay Triangulation.

Once a 2D mesh is generated, depth is added through extrusion. The normal computation process is done during the extrusion but it has some pitfalls. The normal computation request a special attention to avoid lighting inconsistencies and errors.

The algorithm as a whole is timed and compared with the current in-use solution in Deltatec’s proprietary engine. The results show that the developed algorithm, despite not being optimised, is close to the current one in terms of efficiency. We could expect that the developed algorithm, once optimised, will be as efficient if not even more efficient than the current in-use. The quality of the resulting mesh was created to be equivalent to the current algorithm.

\footnote{The algorithm first split the input polygon (with holes) into monotone sub-polygons. Then it triangulate each sub-polygon.}
Acknowledgements

First, I would like to thank Terry Moës for the numerous meetings and wise advice that helped me a lot during the making of this thesis. And for his help on the redaction of this report.

Then, I would like to thank Sébastien Magdelyns for the very interesting conversations and their very interesting divergences.

I would like to thank Éric Béchet for his guidance on the parametric aspect of this work, it helped me a lot understanding more deeply the ins and outs, especially for the discretisation of such parametric forms. Moreover, thanks to its course about computational geometry and computer aided design, the majority of the algorithms seen or used during this work were already understood.

I would also like to thank Christophe Leblanc for his help on the implementation of the Monotone Polygons algorithm.

I would like to thank my family, especially my partner, for having supported my detailed conversations on the subject despite the fact that neither of them understand anything that I was saying.

Last but not least, I would like to make a special thank to my partner who supported and advised me during my years of study.
## Contents

1. **Introduction**  
   1.1 Deltatec .................................................. 8  
   1.2 Environment and Objectives  
      1.2.1 Template Editor ..................................... 9  
      1.2.2 Preliminary R&D .................................. 10  
   1.3 Work plan .................................................. 11  

2. **Definitions** .................................................. 12  
   2.1 General concepts .......................................... 12  
      2.1.1 Bézier .............................................. 12  
      2.1.2 Glyph ............................................... 13  
      2.1.3 Figure .............................................. 13  
   2.2 Licenses .................................................... 14  

3. **DirectX** .......................................................... 15  
   3.1 Direct Write ............................................... 15  
   3.2 Direct 2D .................................................. 15  
   3.3 Direct 3D .................................................. 15  
   3.4 Vertex ...................................................... 16  
   3.5 Index & vertex buffers .................................... 16  
   3.6 Culling ..................................................... 17  
   3.7 Lighting .................................................... 17  
   3.8 Existing method for mesh generation  
      3.8.1 Texture mapping ...................................... 18  
      3.8.2 Quality parameters .................................. 20  
   3.9 3D modelling ............................................... 20  

4. **Extraction of parametric forms** .................................. 21  
   4.1 Design of the Proof-of-Concept .................................. 21  
   4.2 Missing information and bugs .................................. 22  
   4.3 Fonts and their representation  
      4.3.1 TrueType & OpenType fonts  
      4.3.2 Self-intersection .................................... 22  
      4.3.3 Conventions inside/outside a glyph  
         4.3.3.1 Contours & holes .................................. 24  
      4.3.4 Shape & number of curves in a glyph ................. 26  
   4.4 Bézier curve discretisation .................................. 27  
      4.4.1 First easy approach .................................. 28  
      4.4.2 Adaptive subdivision .................................. 29  
      4.4.3 Quality parameters matching ......................... 31  

5. **2D Mesh generation** ............................................ 33  
   5.1 Tessellation formalisation .................................. 33  
   5.2 Methods and algorithms  
      5.2.1 Monotone polygons  
         5.2.1.1 Monotone decomposition  
         5.2.1.2 Triangulation of x-monotone polygons ............ 34  
         37
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Template Editor.</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Bézier curve.</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Bézier &quot;families&quot;.</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Coordinate system comparison.</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Vertex and index buffers example.</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>Shading models.</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>DirectX mesh creation.</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>Coordinate system used in texture mapping.</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td>Texture coordinates example.</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>Example of Deltatec defined qualities using DirectX solution.</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Extraction of parametric form of font.</td>
<td>21</td>
</tr>
<tr>
<td>12</td>
<td>Example of self-intersection.</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>Example of convention: ALternate and Winding.</td>
<td>24</td>
</tr>
<tr>
<td>14</td>
<td>Contour matched with its holes.</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>Enumeration order detection.</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>Example of the number of curves and lines in fonts.</td>
<td>27</td>
</tr>
<tr>
<td>17</td>
<td>Discretisation importance.</td>
<td>27</td>
</tr>
<tr>
<td>18</td>
<td>First approach to Bézier discretisation.</td>
<td>28</td>
</tr>
<tr>
<td>19</td>
<td>Execution of de Casteljau.</td>
<td>29</td>
</tr>
<tr>
<td>20</td>
<td>Adaptive Bézier discretisation.</td>
<td>31</td>
</tr>
<tr>
<td>21</td>
<td>Comparison of discretisation parameters.</td>
<td>32</td>
</tr>
<tr>
<td>22</td>
<td>Example of δ-monotone polygons.</td>
<td>33</td>
</tr>
<tr>
<td>23</td>
<td>Tagged vertices in a monotone polygon.</td>
<td>34</td>
</tr>
<tr>
<td>24</td>
<td>Subdivision into monotone polygons.</td>
<td>36</td>
</tr>
<tr>
<td>25</td>
<td>Example of Monotone Polygons triangulation.</td>
<td>38</td>
</tr>
<tr>
<td>26</td>
<td>Delaunay criterion.</td>
<td>39</td>
</tr>
<tr>
<td>27</td>
<td>Example of Sloan swapping edge to impose constraints.</td>
<td>40</td>
</tr>
<tr>
<td>28</td>
<td>Re-triangulation to impose constraints.</td>
<td>41</td>
</tr>
<tr>
<td>29</td>
<td>Comparison of Delaunay triangulation and Constrained Delaunay triangulation.</td>
<td>41</td>
</tr>
<tr>
<td>30</td>
<td>Example of quadtree with points.</td>
<td>42</td>
</tr>
<tr>
<td>31</td>
<td>Example of triangulation of a cell using a Quadtree.</td>
<td>43</td>
</tr>
<tr>
<td>32</td>
<td>Quadtree triangulation.</td>
<td>44</td>
</tr>
<tr>
<td>33</td>
<td>Non obtuse angle triangulation.</td>
<td>46</td>
</tr>
<tr>
<td>34</td>
<td>Example of the erroneous triangles.</td>
<td>48</td>
</tr>
<tr>
<td>35</td>
<td>Mapping coordinates.</td>
<td>50</td>
</tr>
<tr>
<td>36</td>
<td>Naive extrusion.</td>
<td>51</td>
</tr>
<tr>
<td>37</td>
<td>Naive extrusion: normal error.</td>
<td>52</td>
</tr>
<tr>
<td>38</td>
<td>Improved extrusion: Normal computation.</td>
<td>53</td>
</tr>
<tr>
<td>39</td>
<td>Improved extrusion: normal error.</td>
<td>53</td>
</tr>
<tr>
<td>40</td>
<td>Comparison improved and final extrusion.</td>
<td>54</td>
</tr>
<tr>
<td>41</td>
<td>Extrusion methods compared.</td>
<td>54</td>
</tr>
<tr>
<td>42</td>
<td>&quot;Hello world&quot; in 3 languages.</td>
<td>56</td>
</tr>
<tr>
<td>43</td>
<td>Comparison performances for all algorithms.</td>
<td>57</td>
</tr>
<tr>
<td>44</td>
<td>Timing of 3 sub-parts of the algorithm.</td>
<td>58</td>
</tr>
<tr>
<td>45</td>
<td>Comparison performances with or without extrusion.</td>
<td>58</td>
</tr>
<tr>
<td>46</td>
<td>Comparison performances for all categories.</td>
<td>59</td>
</tr>
<tr>
<td>47</td>
<td>Comparison performances for all quality parameters.</td>
<td>60</td>
</tr>
</tbody>
</table>
List of Tables

1    Comparison of the quality parameters between DirectX and this work. . . . . . 31
2    Algorithms comparison. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
1 Introduction

This work is made in collaboration with the University of Liège and Deltatec. The goal of this master thesis is to study and characterise the techniques to obtain a 3D mesh defined as vertices and facets in a 3D rendering engine from parametric shapes. In this work, the parametric shapes tackled are texts.

The algorithms developed, in this master thesis, as a library can be included inside a 3D real-time rendering engine such as the proprietary engine developed by Deltatec. The goal of this work is to evaluate if a replacement of the current algorithm is possible.

The algorithm takes as input a text and outputs a 3D mesh with normals, used for lighting. The quality, efficiency, and the compatibility with the current proprietary engine will be tested and compared with the current solution.

To begin, the context as well as the objective of this work is formalised in depth (See Sections 1.1, 1.2, and 1.2.1), including a short description of the work onto which this master thesis is based (See Section 4).

Following that, some general definitions are set for the frequently used terms in Section 2. A specific section is reserved for DirectX (See Section 3) to set the necessary notions and definitions used through out this report.

Next, the previous R&D used as a base for the extraction of parametric forms is described deeply with more practical definitions and notions about fonts (See Section 4). These definitions are given along their practical use in the extraction of parametric forms. Then, the discretisation of those parametric forms is described (See Section 4.4).

Then, the mesh generation algorithms are discussed, starting with the theoretical aspect (See Section 5.2). The algorithm selection is also explained (See Section 5.2.6). For the selected algorithms, the implementation details are given (See Section 5.3), closely followed by the texturing of the resulting mesh (See Section 5.4).

After that, the extrusion process is described with all the pitfalls for the normals computations (see Section 6).

The performances of both the developed algorithms and the current DirectX solution are compared (see Section 7).

Just before concluding, the possible improvements and the future features are discussed (see Section 8).

1.1 Deltatec

Deltatec is an electronic and computer solution design company based in Liège. Out of the 70 people working at Deltatec, 50 work on hardware and software design. Its domains are real-time video processing for spatial, televisual and industrial activities, as well as real-time video generation with augmented reality and virtual reality.

My master thesis supervisors were T. Moës, Technical Leader, and S. Magdelyns, Senior Project Manager.

1.2 Environment and Objectives

The goal of this master thesis is to study and characterise the techniques to obtain a mesh defined as vertices and facets in a 3D rendering engine from parametric shapes. The work is an R&D and the industrialisation of the proposed solution is not considered as a requirement. This industrialisation will be conducted by Deltatec design team if the results of the R&D are satisfying.
The focus of this work is to build an algorithm that generates meshes for texts. This algorithm will be treated as a library to be included inside Deltatec’s proprietary 3D engine. This means that the algorithm must remain agnostic of the internal data representation of the engine.

As the 3D engine is developed for Windows only, the solution must also be developed and tested only for Windows. This proprietary 3D Engine is built above DirectX; the currently used version of DirectX is the version 9.0c, June 10 update. Thus, notions about it are required to better understand the context of this work.

During this work, choices will be made. These choices are guided by the fact that the work aims at being integrated into proprietary code used for commercial products with high added value.

The engine already has a solution to generate meshes for texts; it uses a solution from DirectX. Only a limited control is available over the current solution via parameters.

A replacement solution is desired by Deltatec for many reasons:

- DirectX 9 is getting old and deprecated. A transition to DirectX 12 is desired by Deltatec. But no solution exists in DirectX 12 for mesh generation of text. An another solution is thus required before going to DirectX 12.

- The current solution is a black box, a replacement solution will allow a full control over the code.

- It will improve the algorithmic knowledge in house about mesh generation.

In other words, this work will allow to improve this control by adding the ability to modify the algorithm itself.

1.2.1 Template Editor

Template Editor is a facade GUI for Deltatec’s engine, it is used to interact with the engine. A specific feature was created to allow Template Editor to import a custom mesh as a vertex buffer and a index buffer (See Section 3.5 for more details). With that feature, we can compare in the engine and side by side both the mesh generated by the current DirectX solution and the ones built during this R&D.
1.2.2 Preliminary R&D

The starting point of this work is a preliminary R&D done by someone at Deltatec, prior to my work. The objective of this preliminary R&D was to characterise a sector making it possible to obtain a parametric description of a text as a Proof-of-Concept (PoC). The result was a small application designed to display a text in a simple 2D environment based on the parametric description of this text and without any processing or reasoning.
1.3 Work plan

This work follows a plan set up jointly by Deltatec and myself:

1. Take back the previous R&D about parametric form extraction from fonts.

2. 2D Mesh generation:
   (a) Problem formalisation
   (b) State-of-the-art (documentation)
   (c) Algorithm selection:
       i. Choose between 1 and M method(s)
       ii. Choose between 1 and N algorithm(s)
   (d) Implementation of the chosen algorithm(s)
   (e) Conclusion:
       i. Benchmark:
           A. Performances
           B. Precision/Quality
       ii. Compatibility with current engine/applications

3. Texturing

4. 3D Mesh generation (extrusion)

5. Bevelling [Optional]:
   (a) Problem formalisation
   (b) State-of-the-art (documentation)
   (c) Algorithm selection:
       i. Choose between 1 and M method(s)
       ii. Choose between 1 and N algorithm(s)
   (d) Implementation of the chosen algorithm(s)
   (e) Conclusion:
       i. Benchmark:
           A. Performances
           B. Precision/Quality
       ii. Compatibility with current engine/applications

The bevelling is marked as optional because this is not a mandatory step towards mesh generation. But it is an attractive feature to have, knowing that it is not supported by the current solution. The bevelling is also optional because the previous steps are already sensed as large amount of work.
2 Definitions

Some important definitions need to be set before beginning because they are used throughout this entire report.

2.1 General concepts

2.1.1 Bézier

A Bézier curve, also shortened to Bézier, is a parametric curve defined by its control points (at least two control points). The control points, also shortened to CP, define the overall shape of the curve. The curve always starts at the first control point and ends at the last control point. In-between those, the control points are not always interpolated by the curve.

In practice, the most common types of Bézier are quadratic and cubic Bézier, respectively defined by three and four control points.

In this work, only cubic Bézier will be used due to the interface exposed by Direct 2D (See Section 3.2). Given that information, we can take a look at all general forms taken by a cubic Bézier. These forms can be split into three "families" of cubic Bézier.

1. The first is the simpler and has a "U" shape. This happens when the quadrilateral formed by the ordered control points is convex without self-intersection. An example can be seen on Figure 3 on the left.

2. The second has a "V" shape. This can be formed from any "U" shape Bézier, by swapping the second and third control points, meaning there is a self-intersection in the polygon CP1-CP2-CP3-CP4. An example can be seen on Figure 3 on the middle.

3. The third and last has a "S" shape. This can be formed from any "U" shape Bézier, by taking the second or third control points and put it on the other side of the line passing by CP1 and CP4 (first and last control points). An example can be seen on Figure 3 on the right.

Figure 2: Quadratic Bézier (blue), Cubic Bézier (black), Control points (orange).
2.1.2 Glyph

Generally speaking, a glyph is a drawing, but commonly in typography, it represents a letter. For example, "i" is a single glyph while "Hello" is composed of 5 glyphs.

2.1.3 Figure

A figure is a sub-part of a glyph. It represents an ordered list of continuous geometrical elements (lines, Bézier, ...) defining either a contour or a hole. It can be opened or closed. For texts, each figure must be closed as it defines a part of the outline. For example:

- "i" is composed of 2 figures (2 contours), one for the point and one for the vertical bar.
- "o" is also composed of 2 figures, one for the contour and one for the hole.
2.2 Licenses

The open-source licenses are used to regulate the use of open-source projects.

It is not always necessary to reinvent the wheel, thus a pre-existing code can be taken from an open-source project and optionally modified to fit the needs. They come in many different types and it is important to know them because some are not acceptable for industrial products. Here is a list of the ones that were be encountered during this work:

- **GPL**: This is the most restrictive license, it forces the user of GPL code to put all the code associated or interacting with GPL code as GPL. That means open-source with restrictions such as the impossibility to use in paid products. In many Deltatec’s contexts (commercial products), such as the one this work is inscribed in, this license enforces too much constraints so that GPL-licensed must be rejected.

- **LGPL**: This license is a more relaxed version of GPL. The restrictions impose that the modifications on the LGPL code must be published and that any portion of LGPL code must be re-compiled without compromising the usability and functionality of the final project.

- **BSD-3**: The code is free to use as wished with the only soft constraint that all contributions need to be cited.

- **MIT**: The code is free to use as wished with the only soft constraint that only the initial source need to be cited.

The GPL license is the only one totally prohibited. The LGPL license requires specific development in production that increases the cost of development with respect to BSD and MIT. Therefore, both last are chosen in priority when possible.
3 DirectX

DirectX is a collection of APIs developed by Microsoft for handling multimedia. It is only available on Microsoft platforms and the currently supported versions range from 9 to 12. This section contains the definition of APIs and concepts used throughout this work.

3.1 Direct Write

Direct Write is an API specifically used when handling high-quality texts. Let us define four useful interfaces exposed by Direct Write, the little link represents a clickable link to the official documentation:

- IDWriteFactory: This interface is the root factory interface for all DirectWrite objects. It is used to create all subsequent DirectWrite objects.
- IDWriteTextFormat: The IDWriteTextFormat interface describes the font and paragraph properties used to format text and it describes locale information.
- IDWriteTextLayout: The IDWriteTextLayout interface represents a block of text after it has been fully analyzed and formatted.
- IDWriteTextRenderer: This interface represents a set of application-defined callbacks that perform rendering of text and decorations such as underlines. It is implemented by the user and not implemented directly by Direct Write.

3.2 Direct 2D

Direct 2D is an API responsible for hardware-accelerated high-quality rendering text and other 2D objects such as user interface elements (UI). It is used, for example, in GPS. Direct 2D is also provided by DirectX.

The only useful interface of Direct 2D is ID2D1SimplifiedGeometrySink that defines an interface that, when inherited, describes a geometric path containing only cubic Bézier curves and/or lines.

3.3 Direct 3D

Direct 3D is an API from DirectX which is used to render three-dimensional graphics. It can be hardware-accelerated by a compatible graphic card.

Direct 3D uses general rendering concepts such as culling, index and vertex buffers, texture mapping, lighting, ... . The necessary concepts for this work are explained in the following sections.

The atomic storage unit for rendering is the vertex and can contain many different types of data such as coordinates, normals, tangent, ... . Direct 3D is used extensively by Deltatec’s proprietary engine.

3 https://docs.microsoft.com/en-us/windows/win32/direct2d/direct2d-portal
3.4 Vertex

In this work, the vertices are represented as:

- A point: \((x, y, z)\)
- A normal: \((N_x, N_y, N_z)\)
- A texture coordinate: \((T_u, T_v)\)

As described in the structure of a vertex, it contains a normal. This is because the normals are not saved in the facets but at the vertices.

Note: This structure can lead to erroneous normals on some facets because the normals are interpolated to obtain the normal at each point of the facet. These errors are discussed in more details in Section 6.

The coordinate system used by DirectX is left-handed (See Figure 5) with the X axis going from left to right and Y going up.

![Coordinate system comparison](https://docs.microsoft.com/en-us/windows/win32/direct3d9/images/leftrght.png)

3.5 Index & vertex buffers

In rendering, the data used to draw can have multiple forms but the one used in this work is composed of two buffers:

- Vertex buffer: This buffer is a contiguous chunk of memory that contains definitions of vertices.
- Index buffer: This buffer is a contiguous chunk of memory that is always associated to a vertex buffer to define polygons based on the vertices of this vertex buffer based on their index.
3.6 Culling

The culling is an optimisation done in rendering to avoid drawing faces that are not visible. Based on the winding order (order of enumeration) of each face, it can be determined if the face is facing away from the camera. If the face is facing towards the camera, the face is rendered normally but if it is not, the face is not rendered at all.

3.7 Lighting

The lighting of a mesh can be done in multiple ways. The flat shading is a simple approach to lighting, the color of the polygon is uniform and computed for a single vertex (or the centroid of the polygon).

An improvement can be done using the Gouraud shading. This method computes the colors at each vertex then it interpolates that color for each pixel of the polygon (typically triangles). This technique remains pretty fast but does not give good results if the light does not hit the vertices.

An alternative is to use the Phong shading. This method interpolates the normals for each pixel, then the color is computed for each pixel. This method is significantly more expensive but gives good results wherever the light hits.

Figure 5: Vertex and index buffers example in 2D for clarity.
3.8 Existing method for mesh generation

Direct3D 9 provides an algorithm to generate a mesh for a given text, font and size. But this algorithm is a black box which provides only a small amount of control through parameters. This algorithm is not able to bevel the generated mesh.

3.8.1 Texture mapping

Texture mapping is a method to map a texture onto a mesh. Let us define the texture space defined by $u$ and $v$ both between 0 (included) and 1 (included). Let the $u$ axis going from left to right and $v$ from top to bottom. A visual representation can be seen in Figure 8. Each vertex of a mesh contains mapping coordinates that are used to determine the color for each pixel of the faces it defines by interpolating the mapping coordinates.

The Figures 8 and 9 use the DirectX 9 axis system.
As an example for a simple color gradient texture applied on a triangle, see Figure 9. The three vertices that define the triangle contains each a point in 3D space (only show in 2D for ease of view) as well as a mapping coordinate. The triangle in gray on the left represent the mapping of the triangle on the right in texture space.
3.8.2 Quality parameters

The quality of the resulting mesh can be controlled through a parameter which bounds the error between the curve and its approximation. The quality of the resulting mesh will entirely depend on the quality of the discretisation. In Deltatec’s engine, the value of this parameter is fixed by four different qualities: low, normal, high, and best. These values were fixed as a result of experience and practical use.

![Figure 10: Example of Deltatec defined qualities using DirectX solution.](image)

3.9 3D modelling

In 3D modelling, the shape of the triangles used has an importance depending on the case:

- Lighting: Regular triangles are prettier when lit when they do not form a plan. This is due to the interpolation of normals used to compute the color of each triangle.
- Bevelling: When bevelling a mesh, it is to be expected that if the triangles are regular, it is easier to compute the bevel.
4 Extraction of parametric forms

As already introduced earlier, the base of this work is a preliminary R&D. The method used is based on Direct Write and Direct 2D (See Section 3.1 and 3.2).

4.1 Design of the Proof-of-Concept

The PoC uses the entities defined above for the following purposes (See Section 3.1 and 3.2):

- IDWriteFactory: Used to create the text layout and format.
- IDWriteTextFormat: Used to create the text layout. Contains the size and properties of the text.
- IDWriteTextLayout: Used to draw the text. Contains the text to be drawn.
- IDWriteTextRenderer: Act as a proxy to intercept the drawing commands and redirect them into the sink.
- ID2D1SimplifiedGeometrySink: Gets all the data required to draw the text. Can only get cubic Bézier curves or lines.

![Diagram](image)

Figure 11: Structure: Extraction of the parametric form of font for a given text. Inputs (green), outputs (orange).

When a text needs to be drawn, the text is given to the text layout and the resulting information can be found in the sink.
4.2 Missing information and bugs

As the previous R&D was only to check if it was possible to extract easily the font information using existing libraries, some bugs were present. Some information were still undetermined because not relevant at that time. Here is a non-exhaustive list of bugs and limitations of the existing implementation.

- All letters are drawn using the same X offset (on top of each other). This is resulting from the inter-letter spacing being unused despite being given by Direct 2D.
- The outline drawn for some letters is not closed, the last line is missing (visually only). The last point need to be connected to the starting point if a flag is set at the end of the definition of a figure.
- No data is saved nor processed. The data is used just-in-time to draw on the test application.
- The different curves and lines composing the outline are not differentiable.
- The font cannot be changed dynamically which helps a lot when debugging.

Quickly after the start of this work, the inter-letter spacing was used properly, allowing the letters to be displayed next to each other and glyphs were closed (missing last line).

Many accessibility features were added to speed up the future testing, debugging, as well as validating the understanding of gathered information. These features allow to show or hide dynamically each element gathered such as control points, lines, polylines, and curves. These additions were used almost continuously throughout the whole work.

4.3 Fonts and their representation

The fonts can be represented using multiple formats or conventions and they observe some properties and constraints. This section aims at explaining the necessary information and their use in this work.

4.3.1 TrueType & OpenType fonts

These are two common standards for fonts representation. The format used by those representation is fixed: TrueType uses exclusively lines and quadratic Bézier where OpenType can in addition use cubic Bézier.

It has to be noted that whatever the representation of the font, Direct 2D will convert it to only lines and cubic Béziers.

4.3.2 Self-intersection

For fonts, a convention of no self-intersection is observed. This means that the figures used to describe the font must never cross another part of itself (See Figure 12). As seen often during researches related to fonts, a self-intersecting font is an incorrectly defined font. This is because if a figure intersect itself, the interior and exterior of the curve cannot be defined anymore. Many algorithms assume that the no self-intersection property is respected. These algorithms might have unknown behaviour if this property is not respected.

---

4.3.3 Conventions inside/outside a glyph

There exist two conventions to define if a point is inside or outside the glyph. The convention used is given by Direct 2D when describing every figure. Both can be used to describe the same glyph but not at the same time. In practice during this work, only the winding definition is implemented because all the tested fonts and texts use it.

**Alternate**  A point is defined as inside the glyph if any ray starting at that point passes through an odd number of line segments (from the same glyph). An example is shown in Figure 13.

**Winding**  A point is defined as inside the glyph if any ray starting at that point has its number of intersection with clockwise oriented figures minus its number of intersection with counter-clockwise oriented figures is positive. An example is shown in Figure 13.

**Comparison alternate and winding**  The winding convention contains the alternate convention because if a shape can be determined as alternate it also can by winding. But the opposite is not always true (e.g. if a point is inside two concentric figures, it cannot be defined as inside the glyph with Alternate), an example is show in Figure 13. The most central circle can not be described as inside by alternate. This can be explained because winding contains the enumeration order which is not contained in alternate.
4.3.3.1 Contours & holes

It is useful for later that each figure is labelled with either "contour" or "hole". Knowing this allows to triangulate each contour and its holes as independent pieces of a glyph. It is necessary to know for each contour all the holes that are directly inside it —It should be noted that for a hole in a contour in a hole in a contour, the first hole is not directly inside the last contour. This necessity applies only when using Monotone Polygons (See later in Section 5.2.1) because it requires that the triangulation is done polygon (with holes) by polygon. It cannot triangulate two polygons at once.

The matching of a contour to its holes can be done by first detecting the rectangular bounding box of each figure: smallest axis-aligned rectangle for which the figure is fully inside. We match each hole to the contour which entirely contains it and which is the closest to the hole. As the first condition is necessary but not sufficient, the latter condition is used as a tie breaker if multiple contours entirely contain a hole.

The contours and holes are defined as polyline, given that definition, the minimum distance between any point \( p \) of a hole to a contour can be found by searching for the smallest distance between \( p \) and all the line segments forming the contour.

The tiebreaker is not an optimal solution when matching \( n \) nested contours to their respective holes. But this algorithm is used because it has the advantage of being less complex and it has a similar efficiency when \( n \) is small. As this work is focused on figures extracted from characters, we can always expect that the number of nested contours \( (n) \) is small. Furthermore, the bounding boxes criterion already filters out almost all contours, reducing the tie breaker usage to very rare occurrences (no occurrence at all for latin characters in the following standard fonts: Tahoma, Comic Sans MS, Courier New, Arial, and Times New Roman).

An optimal algorithm for the tie breaker exists and is described in Chapter 2 of Computational Geometry: Algorithms and Applications [11]. This algorithm matches every holes to their respective contours all at once, as opposed to sequentially searching for the next contours matching a specific hole.

Erratum: See Appendix [C]
In the example shown in Figure 14, the bounding boxes are named $B_1$, $B_2$ and $B_3$. The only hole is $B_3$, it must be matched as directly inside either $B_1$ or $B_2$. Let us search which contour is closest to $B_3$.

For any point $p$ of $B_3$, search for the minimum distance between $p$ and all the points of $B_1$ and $B_2$. The contour which is the closest is matched to $B_3$, thus $B_3$ is matched as directly inside $B_2$.

**Figure 14: Contour matched with its holes: Inner green is not matched as directly inside red.**

**Enumeration order** In practice, for texts, Direct 2D only uses the Winding convention. This means that the order of enumeration of points defines a contour or a hole. This enumeration order is constant once set, it can be either clockwise or counter-clockwise.

To detect if a figure, seen as an ordered list of points, is in clockwise or counter-clockwise order many methods exist but one stands out by its simplicity and its efficiency:

- Take a point which is guaranteed to be on the convex hull. This point can be the lowest, leftmost point (or any other such extreme).
- Then compute the cross product of the edges before and after it from the ordered list of points.
- The sign of $z$ of the cross product gives the order of enumeration: clockwise or counter-clockwise.

---

\[ \text{How do I find the orientation of a simple polygon?} \quad \text{http://www.faqs.org/faqs/graphics/algorithms-faq/} \]
The reason that such a point works is because this point has a convex internal angle (strictly less than $\pi$). This technique can be applied to any point from a convex polygon, to detect its order of enumeration, because any point is by definition on the convex hull.

If the figure is composed of a mix of lines and Béziers, only the extremities are kept for this computation. The definition of the curve does not impact the order of enumeration.

The main advantage of this method over the others is that a single extreme point needs to be found to make a single computation.

The figure with the biggest rectangular bounding box is as a contour (cannot be a hole). The order of that biggest figure defines the order for all other figures in the same glyph. For example, if the biggest figure has a clockwise order, then all clockwise figures in the glyph are contours where the counter-clockwise figures are holes.

### 4.3.4 Shape & number of curves in a glyph

During researches and tests about the extracted glyphs, we realised that the number of curves that define a letter is quite high. This means that the size of the curve is small with respect to the size of the letter it defines. This also means that the letter tends to have many small details. But it does apply only for fonts that are curvy.

In all the encountered fonts, the level of detail of the curved area of a glyph is such that a glyph is often described with a large number of small segments or Béziers. Luckily, for optimisation reasons, the straight segments are described as straight line instead of Bézier even if very long.
4.4 Bézier curve discretisation

As stated before, a Bézier curve is a parametric curve defined by its control points. As triangulation algorithms take as input a set of points, the parametric form cannot be used directly. The discretisation is thus mandatory as an intermediate step between the extraction of the glyph information and the triangulation.

Moreover, once the discretisation is done, the visual quality of the resulting 2D mesh is not impacted by the algorithm used. This is because the discretised points define the outline of the shape that will be triangulated. This 2D shape will be the same despite being defined with different triangles.
The interface providing the parametric forms can exclusively give lines or cubic Béziers. The discretisation of a line is only composed of the endpoints of the line. The discretisation of the Bézier will be discussed in more details because it is not as easy.

The size chosen for the font is fixed to 64em because a size is required by Direct Write (by the IDWriteTextFormat, see Section 3.1) to extract the parametric information. But the size has no impact because once the mesh is generated, the vertices coordinates are normalised between -1 and 1 to be used as vectorial coordinates.

4.4.1 First easy approach

A Bézier curve can be approximated by a single line, this crude approximation is starting at CP1 (starting control point) and ending at CP4 (ending control point). This approximation can be improved by introducing a single intermediary point at the middle of the curve. This particular point is chosen because it is easy and efficient to compute. See Figure 18 for a comparison on the three "families" of Bézier.

Obviously the results of the 1-line approximation is not accurate especially for "U" and "V" Bézier. In fact, the inaccuracy increases with the size of the curve, leading to unusable results when the curve is relatively big compared to the letter in which it resides.

We could think that the same issue happens for the 2-lines approximation but it is significantly better in the worst cases of the 1-line approximation. And it is not worst in other cases. This can be visualised easily if we take a look at the comparison for the three "families" of cubic Bézier (See Figure 18).

The 2-lines approximation will thus become our "low" quality. It is quick and the number of points created is low. It can be used when the text is small because the details are too small to be seen. As it will be shown later, it matches the in-use "low" quality (See Section 4.4.3).

Figure 18: First easy approach of Bézier (black) discretisation: 1-line (dotted red), 2-lines (dashed purple).
4.4.2 Adaptive subdivision

Given that the curvature of a Bézier is neither constant nor linear, it is not trivial to discretise it with a given error bound. Ideally, the algorithm would reach the error bound with the least number of points. But to reach this, no efficient method exists. The method used will thus achieve the error bound without constraint about the number of points used to reach it.

I used an adaptation of the well-known algorithm of de Casteljau [2].

Original de Casteljau The idea of the algorithm is to recursively split the curve into two curves with the same number of control points. The point at which the curve is split is on the curve and is recorded.

The algorithm takes as input a curve \( c \) with \( d \) control points and returns a list of discretised points. The algorithm also use a maximum recursive depth \( r (> 0) \). The algorithm is as follows:

- If \( r == 0 \): stop the recursion.
- Split the curve at the mid point \( m \) on the curve, such that \( m = \frac{CP_1+3CP_2+3CP_3+CP_4}{2} \). The computation is for cubic Bézier only but similar computations are done for other number of control points.
- Do a recursive call on the first half of the curve with \( r - 1 \) instead of \( r \).
- Save \( m \) at the end of the list of saved point.
- Recursive call on the second half of the curve with \( r - 1 \) instead of \( r \).

Figure 19: Execution of de Casteljau. \( P_{3,0} \) is \( m \). \( P_0, P_{1,0}, P_{2,0}, P_{3,0} \) determine the first half of the curve and \( P_{3,0}, P_{2,1}, P_{1,2}, P_3 \) determine the second half of the curve for their respective recursive call.

Source: [https://i.stack.imgur.com/3G5Ws.png](https://i.stack.imgur.com/3G5Ws.png)
Modified de Casteljau The only modification is the stop criteria and it uses two new parameters, in addition of \( r \) which remains the same:

- **Angle error:** The threshold angle between two consecutive approximation lines. Used to avoid big angles.
- **Distance error:** If the distance between the segments CP1-CP4 and CP2-CP3 is below the threshold distance, CP1-CP4 can be used as approximation of the curve.

In practice, \( r \) must not stop the recursion if the other parameters are set appropriately. The modified algorithm is as follows:

- If \( r == 0 \): stop the recursion.
- Split the curve at the mid point \( m \) on the curve.
- **Stop the recursion if the first half of the curve can be approximated by a line:**
  - The curve can be approximated if the distance between CP1-CP4 and CP2-CP3 is smaller than the distance threshold
  - Or, if the angle formed by CP1-M-CP4 (where M is the mid point of the curve) is smaller than the angle threshold.

Else do a recursive call on the first half of the curve (still with \( d \) newly computed control points) with \( r - 1 \) instead of \( r \).

- Save \( m \) at the end of the list of saved point.

- **If the second half of the curve can be approximated by a line, stop the recursion. Else do a recursive call on the second half of the curve (still with \( d \) newly computed control points) with \( r - 1 \) instead of \( r \).**

The results of this modified de Casteljau can be viewed in Figure [20] for three different set of parameters. These are the final set of parameters that were chosen; the explanation of this choice is described in detail below in Section 4.4.3.
4.4.3 Quality parameters matching

As stated before, both the algorithm of DirectX and the one developed in this work use parameters to control the quality of the discretisation. For the current solution in Deltatec’s engine, four distinct qualities are fixed. The goal is to get approximately the same quality for the developed algorithm despite using other parameters. The comparison between the results obtained can be seen in Figure 21 and the detailed values are shown in Table 1.

As explained in Section 4.4.1, the "low" quality is only based on the geometric definition of the font. Thus it does not have any parameters.

<table>
<thead>
<tr>
<th>Quality</th>
<th>DirectX Deviation</th>
<th>2-lines approximation</th>
<th>Adaptive subdivision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max subdivisions</td>
<td>Distance threshold</td>
<td>Angle threshold (radian)</td>
</tr>
<tr>
<td>Low</td>
<td>0.1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Normal</td>
<td>0.01</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>High</td>
<td>0.005</td>
<td>15</td>
<td>0.1</td>
</tr>
<tr>
<td>Best</td>
<td>0.001</td>
<td>15</td>
<td>0.005</td>
</tr>
</tbody>
</table>

Table 1: Quality parameters matching between DirectX and 2-lines approximation & Adaptive subdivision.
Figure 21: Comparison of discretisation parameters. The font used is Comic Sans MS.
5 2D Mesh generation

5.1 Tessellation formalisation

This section describes the steps used to convert a pair text-font to a 2D mesh.

As each mesh generation algorithm takes as input a set of points (discretised points) and some constraints (the polylines describing the glyph) to generate a mesh, the choice of algorithm used is based on the shapes of the triangles and the efficiency of the algorithm. The shape of the triangles does not matter in 2D if the mesh is unlit as explained in Section 3.9. But the triangulation algorithm must be selected accordingly to the end goal which is a 3D mesh.

5.2 Methods and algorithms

5.2.1 Monotone polygons

Let us first define monotone polygons such that if we have δ as an arbitrary line, a polygon is δ-monotone if any perpendicular line to δ cuts the polygon in at most two points (See Figure 22).

![Figure 22: The two polygons on the left are not δ-monotone, the two on the right are. Green and blue lines represent respectively one and two intersections. Red lines represent three or more intersections. Source: https://en.wikipedia.org/wiki/File:M-polygon.svg](https://en.wikipedia.org/wiki/File:M-polygon.svg)

The triangulation of a monotone polygon is based on the triangulation of any convex polygon. For any convex polygons, the triangulation can be achieve by creating triangles from a point to any two consecutive other points. The resulting triangulation is fan shaped. The same fan-shaped approach can be used on monotone polygons with some adaptation to take into account that the polygons are monotone and not convex. This technique is simple and efficient as described below.

https://en.wikipedia.org/wiki/Monotone_polygon
But in practice, almost no polygons will be monotone. Fortunately, there exists an efficient algorithm to split any polygon (with holes) into a set of monotone polygons.

The following algorithms for monotone decomposition and monotone triangulation are both from the course MECA0524-1 CAD & GEOMETRIC ALGORITHMS, BÉCHET ERIC AT ULIÈGE.

5.2.1.1 Monotone decomposition

The algorithm to split any polygon into a set of x-monotone sub-polygons is as follows:

- Tag every vertices (See Figure 23):
  - Start: Both edges are on the right and $\alpha < \pi$.
  - End: Both edges are on the left and $\alpha < \pi$.
  - Split: Both edges are on the right and $\alpha > \pi$.
  - Merge: Both edges are on the left and $\alpha > \pi$.
  - Regular: One edge on either side.

- Sort the vertices in lexicographical order (first on x then on y) based on their coordinates and put them in a priority queue $Q$. For example, the queue for the polygon in Figure 24 will be: 2, 1, 14, 12, 3, 11, 4, 13, 10, 6, 5, 9, 7, and 8.

- Initialise an empty binary search tree $T$ of edges referred to as the status.

- While $Q$ is not empty:
  - Pop vertex $v_i$ from $Q$.
  - Call the appropriate procedure to handle $v_i$ depending on its tag. Knowing that for each edge $e_j$ an associated vertex $corr(e_j)$ exists and is defined (no vertex creation) during the algorithm.

![Figure 23: Tagged vertices in a monotone polygon. "S": Start, "E": End, "M": Merge, "R": Regular. There is no split in this polygon.](image-url)
Appropriate procedure for $v_i$ depending on its type:

- **Start:** Insert $e_i$ in $T$ and $corr(e_i) = v_i$.

- **End:**
  - If $corr(e_{i-1})$ is a Merge vertex: Add the diagonal $v_i$ to $corr(e_{i-1})$.
  - Anyway, delete $e_{i-1}$ from $T$.

- **Split:**
  - Find $e_j$ in $T$ such that $e_j$ is directly below $v_i$.
  - Add the diagonal $v_i$ to $corr(e_j)$.
  - Insert $e_i$ in $T$ and $corr(e_i) = v_i$.

- **Merge:**
  - If $corr(e_{i-1})$ is a Merge vertex: Add the diagonal $v_i$ to $corr(e_{i-1})$.
  - Delete $e_{i-1}$ from $T$.
  - Find $e_j$ in $T$ such that $e_j$ is directly below $v_i$.
  - If $corr(e_i)$ is a Merge vertex: Add the diagonal $v_i$ to $corr(e_i)$.
  - Set $corr(e_j) = v_i$.

- **Regular:**
  - If the interior of the polygon lies above $v_i$:
    - * If $corr(e_{i-1})$ is a Merge vertex: Add the diagonal $v_i$ to $corr(e_{i-1})$.
    - * Delete $e_{i-1}$ from $T$.
    - * Insert $e_i$ in $T$ and $corr(e_i) = v_i$.
    - Else:
      * Find $e_j$ in $T$ such that $e_j$ is directly below $v_i$.
      * If $corr(e_j)$ is a Merge vertex: Add the diagonal $v_i$ to $corr(e_j)$.
      * Set $corr(e_j) = v_i$.

---

7 The addition of edges is done inside the initial polygon thus creating the separations required to define the monotone sub-polygons.

8 Directly below refers to a vertex having a y coordinate smaller than the edge and having a x coordinate between the x coordinate of the start and end point of the edge.
This algorithm is a sweep line algorithm because it scans the plan (from left to right). Its complexity is $O(n\log(n))$ because:

- Building $Q$: $O(n\log(n))$.
- Every vertex is processed in at most $O(\log(n))$ (search in $T$) and there are $n$ vertices. So a total complexity of $O(n\log(n))$ for the vertex processing.
5.2.1.2 Triangulation of x-monotone polygons

The triangulation algorithm for monotone polygons is as follows:

- Tag each vertex as being on the top chain or the bottom chain. The chains of a monotone polygon is such that while traversing the chain, the points are enumerated in their natural order (x monotonically increasing or decreasing). This can be done efficiently by traversing the definition of the monotone polygon. If the next neighbouring vertex has a bigger x coordinate, the current vertex is on the top chain. If the next neighbouring vertex has a smaller x coordinate, the current vertex is on the bottom chain.
- Sort the vertices in lexicographical order (first on x then on y) based on their coordinates.
- Initialise an empty stack $S$ and push $v_0$ and $v_1$, reps. the first and second leftmost vertices.
- For $i$ from 2 to $n-2$
  - If $v_i$ and the top of the stack are on different chains:
    * Pop all vertices from the stack and create a diagonal between each of those and $v_i$, except the last one (still popped out).
    * Push $v_{i-1}$ and $v_i$ on $S$.
  - Else:
    * Pop one vertex from $S$.
    * Pop the other vertices one by one and make a diagonal between those and $v_i$, as far as possible without crossing. When the diagonal cannot be done, stop.
    * Push the last popped vertex and $v_i$ on $S$.
- Create a diagonal between $v_{n-1}$ and all the remaining vertices in the stack except the first and last.

---

9 The diagonal is made if it is fully inside the polygon and does not intersect with any edge of the initial polygon.
This algorithm is also a sweep line algorithm. Its complexity is $O(n)$ because:

- The ordering of vertices is $O(n)$ because the polygon is monotone.
- The loop is done $n - 3$ times and in the worst case, $2n - 4$ vertices are pushed on the stack. Thus the number of popped vertices cannot exceed this values either. Each operation is in constant time.

5.2.1.3 Full algorithm

The resulting complexity of the full algorithm is $O(n \log(n))$. This technique is efficient but there are no guarantee about the shape of the resulting triangles. In practice, there will be long and thin triangles as this can be seen on Figure 25.

As we can see on the Figure 25, the shapes of the triangles resembles the shapes of the triangles from the triangulation of DirectX (See Section 3.8). We can assume that DirectX is using the same algorithm as a base.

5.2.2 Constrained Delaunay

The Constrained Delaunay Triangulation is most of the time a two part algorithm which begins with a Delaunay Triangulation (unconstrained) and ends with the addition of constraints. The addition of constraints is required because the Delaunay Triangulation does not ensure the presence of any edges.

5.2.2.1 Delaunay triangulation

The Delaunay triangulation of a set of points is the triangulation of the convex hull of the set of points which follows the Delaunay criterion (See below).

The resulting triangulation does not ensure that any specific edges are present and the triangulation can contain unwanted triangles. An example of Delaunay triangulation is given in
The Delaunay criterion is responsible for the shape of the triangles. The complexity of the Delaunay triangulation is $O(n \log(n))$, for $n$ vertices as input.

The algorithm is an iterative algorithm for which the points are added one at a time. After each insertion, the Delaunay criterion is satisfied.

**Delaunay criterion**  The criterion is such that no point $P$, from the set of triangulated points, is inside the circumcircle of any triangle in the triangulation. This criterion maximises the minimum angle of all triangles. The shape of triangles tends to be equilateral as much as possible. This avoids triangles that are long and thin.

It is possible to impose the Delaunay criterion on any arbitrary triangulation but this has a complexity of $\Omega(n^2)$. It is thus better to create the triangulation from scratch instead of improving an existing one.

**5.2.2.2 Adding constraints**

There exist many algorithms to impose constraints to a given Delaunay triangulation, only the two most commonly cited ones will be described. These algorithms are based on the same two steps: Imposing edges and removing the outer triangles.
S.W. Sloan [4] For each edge that must be in the triangulation, also called constraint, the algorithm works as follows:

- Create a queue of edges that intersect the constraint $c$.
- Pop the first edge $e$ from the queue.
- If $e$ is a diagonal of a convex quadrilateral formed by both triangles adjacent to $e$, replace $e$ in the triangulation by the other diagonal of this quadrilateral. See Figure 27 for a visual representation of the swap.
- Else, put back $e$ at the end of the queue.
- Loop until the queue is empty.
- When the constraint it in the triangulation, lock it in place and impose the criterion on each modified edge. This ensures that the criterion is respected as much as possible.

The theory shows that the queue always contains at least an edge that can be swapped and thus removed from the queue. Therefore, at each iteration, an edge can be removed from the queue, guaranteeing that the algorithm will stop.

Figure 27: Example of Sloan swapping edge to impose constraints.

Source: S.W. Sloan [4]

M.V. Anglada [5] The idea is to find the polygon made of triangles which intersect a constraint then to triangulate twice, once for the points on one side of the constraint and one for the other side. This will impose the constraint while maintaining the Delaunay criterion as fully as possible. The two polygons that are re-triangulated might not be convex, the removal of unwanted outer triangles must be done before merging back the result into the mesh.

To remove the outer triangles, it is enough to go around the boundary edges (contours or holes) to be able to reject them.
All in one algorithm  It exists more involved algorithms that can impose constraints during the triangulation. These are not detailed here because of their involved nature, such an algorithm is described by Domiter, et al [6]. This paper describes the algorithm used in Fast Poly2Tri (See Section 5.3.3).
5.2.3 Quadtree

It is possible to use the well-known quadtree\(^{10}\) structure (See definition below) to create a triangulation of a set of points and edges.

5.2.3.1 Quadtree structure

A quadtree is a data structure in which each internal node has exactly four children. Each of these children represent a quadrant of a 2D space. The quadtree follows a set of rules:

- Each cell has a maximum capacity. When it is reached, the cell is split in four (recursive subdivision).
- The children cells might be squares, rectangles, or any arbitrary shape.

Quadtrees have different types according to the type of data they represent (point, edge, region, ...). The complexity of the creation of a quadtree of \(n\) points and a depth \(d\) is \(O((d+1)n)\). The depth is dependent on the distribution of the points.

![Example of quadtree with points.](https://en.wikipedia.org/wiki/Quadtree#/media/File:Point_quadtree.svg)

---

\(^{10}\)https://en.wikipedia.org/wiki/Quadtree
5.2.3.2 Quadtree balancing

A quadtree is balanced if any two neighbouring cells differ at most by a factor of two in size. The quadtree balancing algorithm searches for neighbouring cells that are not balanced and expand them.

The complexity of the balancing of a quadtree of size \( n \) and depth \( d \) is \( O((d + 1)m) \), where \( m \) is the number of cells after the balancing.

5.2.3.3 Quadtree triangulation

The triangulation is done by triangulating each cell of the quadtree as follows:

- If the quadrant is empty (no point and no edge): Triangulate into two triangles using the four corners.
- If the quadrant contains at least one point or segment of edge: Triangulate using the four corners, the point(s), and the extremities of edge segments.

The cost to generate a mesh of a quadtree is \( O(S_p \log^2(U)) \), where \( S_p \) is the sum of the perimeters of all edges and \( U \) is the length of the side of the root of the quadtree (typically the rectangular bounding box around the points).

![Diagram of triangulation of a cell using a Quadtree.](image)

Figure 31: Example of triangulation of a cell using a Quadtree.
Original point (Red), original edges (Green), added points (Blue), added edges (Blue).
The triangulation using a quadtree has three stages:

- Create a quadtree $T$ from the set of points and edges (constraints).
- Optional: Balance $T$. The balance can be omitted to reduce the number of triangles. The balance is done when dealing with finite elements analysis.
- Triangulate all the cells that are inside the constrained shape.

The result of such a triangulation can be seen in Figure 32. The resulting computational complexity of the full algorithm is $O(((d+1)n) + S_p \log^2(U)) = O(dn + S_p \log^2(U))$.

This technique is efficient because the quadtree is a structure optimised to store points. But the algorithm adds far too many points, creating a higher number of points and triangles. Moreover, the number of triangles is depending on the alignment of the grid with respect to the points. The higher number of points is especially a problem when in a real-time rendering setting such as Deltatec’s engine uses.

It might be possible to apply some kind of custom post-processing to remove points and triangles by merging them together but this would defeat the purpose by adding more complexity and inefficiencies, along with many specific cases.
5.2.4 No discretisation triangulation

An idea was proposed by someone at Deltatec. This was just an idea but was very interesting because it was not mandatory to discretise the curves and the number of triangles might be low. This idea was as follows:

- Pick a point \( p \) (following some policy) in the glyph.
- From the glyph, the points, and the triangles already present, insert the biggest triangle using \( p \).
- Repeat until the whole glyph is covered.

This idea gives a mesh that does not depend on any discretisation and we could expect that the number of triangles is low because only the biggest triangle is added at each step.

But after some researches, this idea was not viable. The detection of the biggest triangle is not easy. This detection will most likely use line-Bézier intersection computation which itself requires discretisation.

Moreover, the shape the triangles would have been unpredictable which is a problem for the bevelling.

This idea seems to be unfeasible in practice without discretisation which is the goal of the approach.

5.2.5 Miscellaneous & exotic

There exist many more algorithms to triangulate a given set of points but we have already covered the main ones. The uncovered algorithms are mainly focusing criteria for specific situations or needs. These criteria are not useful for us because the Delaunay criterion is by far the most important for us.

An example of those criteria is: No obtuse angle in the final triangulation [8].

The algorithm, "Linear-Size Nonobtuse Triangulation of Polygons", is as follows:

- Stage 1: Packs the domain with non-overlapping disks, tangent to each other and to the sides of the domain. The disk packing is such that each region not covered by a disk has at most four sides (either lines or arcs). Edges are added between the center of the disks and the tangency points on the boundaries.
- Stage 2: Each sub-polygon created in stage 1 is triangulated.

The main steps of this algorithm are shown in Figure [33]. The complexity of this algorithm is \( O(n \log(n)) \) for simple polygons and \( O(n^{3/2}) \) for polygons with holes.
This criteria is not useful for us and the complexity of the overall algorithm is worse than the previously described algorithms.

5.2.6 Algorithm(s) selection

To choose the algorithm or algorithms that will fit the best our use case, a quick comparison must be done regarding their complexity, advantages, and drawbacks.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Implementation</th>
<th>In-use load</th>
<th>Triangles shapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constrained Delaunay</td>
<td>$O(n \log(n))$</td>
<td>Complex</td>
<td>Low</td>
<td>Almost equilateral</td>
</tr>
<tr>
<td>Monotone polygons</td>
<td>$O(n \log(n))$</td>
<td>Simple</td>
<td>Low</td>
<td>Unconstrained (long &amp; thin)</td>
</tr>
<tr>
<td>Quadtree</td>
<td>$O(dn + S_p \log^2(U))$</td>
<td>Simple</td>
<td>Potentially high</td>
<td>Regular (isosceles)</td>
</tr>
</tbody>
</table>

Table 2: Algorithms comparison.

According to the Table 2, Delaunay seems to be the best algorithm with no drawback except the implementation complexity. Something that is not shown in the table is the constant factor in the complexity. This constant factor is not relevant for asymptotic behaviour but might have a big impact given that $n$ is small for our practical case — $n$ is typically between 25 (lowest quality) and 200 (best quality). It is to be expected that this constant factor is bigger for Delaunay than for Monotone Polygons but it is unknown by how much. This assumption is based on the fact that an additional criterion is added for Delaunay but not for Monotone Polygons. This criterion can not come for free. Both algorithms need to be compared more deeply before taking a decision.

As discussed earlier, the shape of the triangles matters so the ideal solution would be to use Delaunay in every cases, but if the difference of performance is big enough, Monotone Polygons might be used for 2D unlit texts, while Delaunay would be used for the other cases.

Quadtree has too many problems compared to the other two, it will not be implemented.
5.3 Implemented algorithms

As mentioned in section 5.2.6, both Constrained Delaunay and Monotone Polygons will be implemented and compared, this section describes the implementation of those algorithms.

5.3.1 Triangulated elements

The triangulation is not made glyph by glyph but rather by group of figures, each group is composed of one contour and all its holes (See Section 4.3.3.1). This distinction is required by Monotone Polygons to work properly. This is not required by Constrained Delaunay but it is not useless either. By reducing the number of points triangulated at once, the overall efficiency of the triangulation cannot decrease. Let us prove it knowing that both algorithms run in $O(n \log(n))$.

Let us define $n_i$ as the number of points in a group of figures, thus the total number of points $N$ is such that:

$$N = \sum_{i} n_i$$

We can show that:

$$\sum_{i} n_i \log(n_i) \leq N \log(N)$$

The entire demonstration can be found in Appendix B.

5.3.2 Monotone Polygons

The literature about this algorithm is not easy to find nor detailed. This can be explained because the researches on the subject were done around early 90’s. There is no available open-source project ready to use because it is considered as an easy to implement algorithm. Moreover, Delaunay is often used instead because the resulting triangles have better shapes (the more equilateral triangles as possible). Therefore, I implemented this algorithm by myself as no existing implementation was suitable.

During the development, a bug occurred leading to some triangles being erroneous. This error happens when the algorithm must check if a diagonal is fully inside the polygon (not crossing any edge and all points of the diagonal must be inside the polygon). This check can be done efficiently but my implementation has some edge cases that do not work. In all the paper I could find, this specific part was never explained in details, so I could not pinpoint the mistake easily. As this error is minor and this work is a R&D, it was not fixed. A fix can be done by doing an inefficient search for intersection in addition to a test to detect if the diagonal is fully inside. From this brute force version, we know that the algorithm is correct. For the benchmarks, given the low number of edge cases, the efficient but erroneous version will be used. The error does not impact on the timing results.
The implementation is not optimised towards performances nor memory consumption. This can still be done if the R&D leads to production. See Section 8.1 for more details about the possible optimisations that can be done.

5.3.3 Constrained Delaunay

Given that Delaunay is a well-known and heavily used algorithm, many implementations (open-source or not) are available. After some researches, here are the available possibilities\textsuperscript{11} the small is the direct link to the website:

- **GMSH**
  - License: GPL.

- **CGAL**
  - License: LPGL & GPL depending on the modules. The modules needed are under GPL.
  - Purchase potential (module by module). One-shot fee.

- **Triangle**
  - License: Custom: Seems to be old or modified version of GPL.
  - Old but robust product.

- **Fade2D**
  - Purchase with maintenance contract, bug fixes, and support. The module wanted is in the base package.

\textsuperscript{11}As these researches are about the constrained version of the algorithm, Delaunator (a popular open-source implementation) is excluded unless no satisfactory constrained Delaunay implementations exists. In such case, it might be used to get an unconstrained triangulation into which the constraints are added a posteriori.
• Poly2Tri (GitHub)
  – License: BSD-3.
  – Not optimised code.

• Fast Poly2Tri (GitHub)
  – License: MIT.
  – Highly optimised.

Given the available possibilities, I cannot use GMSH or CGAL (See Section 2.2). Triangle is also excluded for the same reason. Fade2D comes with much more than the algorithm wanted and it is not free. Given that the last two possibilities are from GitHub, thus free, with MIT or BSD licenses, the choice is easy because both have correct documentation but Fast Poly2Tri has the advantage of being optimised.

5.4 Texturing

To apply any texture other than a plain color, the mesh must have correct texture coordinates. To avoid recomputing the mapping coordinates, they are computed only once before ending the algorithm.

To compute the coordinates themselves, it is required to compute the rectangular bounding box around the mesh, then compute the relative position of each point inside that bounding box.

Let us define the set of points $P$ of the mesh. The bounding box is defined by the pair of points $(\text{Min}_x, \text{Min}_y)$ & $(\text{Max}_x, \text{Max}_y)$, such that:

$$Min_x = \min_{p \in P} \{ p.x \}$$

$$Max_x = \max_{p \in P} \{ p.x \}$$

$$Min_y = \min_{p \in P} \{ p.y \}$$

$$Max_y = \max_{p \in P} \{ p.y \}$$

Then to compute the mapping coordinates $(T_u, T_v)$ of a point $p \in P$:

$$T_u = \frac{p.x - Min_x}{Max_x - Min_x}$$

$$T_v = \frac{p.y - Min_y}{Max_y - Min_y}$$

In practice, because of the inverted Y axis, as described in Section 3.8.1, $T_v$ is computed as:

$$T_v = 1.0 - \frac{p.y - Min_y}{Max_y - Min_y}$$
Figure 35: Mapping coordinates.
6 3D mesh generation: Extrusion

Until now, the generated mesh is only in two dimensions, but we want to have texts as a three dimensional mesh. While going from 2D to 3D, the computation of normals is not trivial anymore. In 2D, the normals are all the same and perpendicular to the plane. For 3D meshes, this is not as straightforward because neighbouring faces are not necessarily in the same plane.

The output of the triangulation algorithms is saved inside a structure which saves all the necessary information to later be able of generating compact and optimised vertex and index buffers.

The resulting triangles are unique and already ordered to be in clockwise order. This is the standard used by DirectX to do its back face culling (See Section 3.6). If the standard is the opposite for culling, any two index of each triangle must be swapped.

6.1 Naive approach

Let us define that the initial 2D face is the front face. We can then duplicate the front face and call that duplicated version the back face. The z coordinate of all the points in the back face are set to -1 to create a gap between the two faces. The normals of the vertices in the back face are the opposite of their respective vertices in the front face. And to avoid having two faces with a gap between them, we create triangles forming a ribbon linking the front and back face together. What is called a ribbon here can be in practice multiple ribbons. For example, "I" have only one contour so it has only a single ribbon, "D" as two ribbons, one for the contour and one for the hole, "B" as three ribbons. Each figure leads to a ribbon.

![Figure 36: Naive extrusion, top and front view: Front face (red), back face (green), ribbon (blue).](image)

This implementation has \(2N\) vertices at the end, where \(N\) is the number of vertices in the front face. This is quite an efficient implementation of the extrusion. But there is problem with the light on the ribbon. As anticipated, because the normals are at the vertices, they are interpolated on the triangles of the ribbon leading to a gradient of shadow. This gradient will happen whatever the position of the light. This behaviour was predictable and predicted as this is a common error.
Figure 37: Erroneous gradient on the ribbon of "D". The ribbon must not have a gradient on the left. The ribbon must be lit on the right.

6.2 Improved version

To solve the issue with interpolated normals, new vertices are created to have dedicated normals for the ribbon. The front and back faces are created, as for the naive approach. Then, all the vertices (both from the front and back faces) are duplicated to form the "ribbon face". The ribbon is then created using only vertices from the ribbon face.

Let us define $P_n \in F_{\text{front}}$, $P_{nd} \in F_{\text{back}}$ matching vertices on different faces. Let us define the same for the ribbon face $R_n \in R_{\text{front}}$, $R_{nd} \in R_{\text{back}}$, where the vertex $R_n$ (resp. $R_{nd}$) as the same point coordinate as $P_n$ (resp. $P_{nd}$). Let us define $QN(\cdot, \cdot, \cdot, \cdot)$ a function taking four vertices as input and returning the normal of the quadrilateral passed as input. See Figure 38 for a visual representation.

Let us define:

\[ v = P_2 - P_1, \quad u = P_3 - P_1 \]
\[ QN(P_1, P_2, P_3, P_4) = \frac{v \times u}{||v \times u||} \]

\[ \text{Normal}(R_n) = \text{Normal}(R_{nd}) = \frac{QN(R_n, R_{nd}, R_{n+1}, R_{n+1d}) + QN(R_{n-1}, R_{n-1d}, R_n, R_{nd})}{2} \]
Figure 38: Improved extrusion: Normal computation.

$QN(R_{n-1}, R_{n-1d}, R_n, R_{nd})$ in blue, $QN(R_n, R_{nd}, R_{n+1}, R_{n+1d})$ in turquoise, and $\text{Normal}(R_n)$ & $\text{Normal}(R_{nd})$ in green.

The number of vertices has doubled from the naive approach, increasing the total number of vertices to $4N$.

This almost fully solves the problem with normals. An error still occurs when a hard edge is encountered around the ribbon, the average of the two normals creates the same visual result as in the naive approach but around the ribbon not between the front (or back) face and the ribbon. An edge is hard if the angle formed by both neighbouring faces is small. The definition of small is done through a threshold which can be modified if required. The erroneous edges in Figure 39 are hard edges. The edges around the ribbon in the curved part of the "D" are all soft edges. It does not happen between the front (or back) face and the ribbon because the vertices are already duplicated to avoid the gradient. This gradient is a consequence of two hard edges bounding the ribbon.

Figure 39: Erroneous gradient around hard edge of "D". On the left, the top of "D" should not be lit. On the right, the top of the "D" should be lit.
6.3 Final version

To solve the problem of hard edges, a check must be made during the creation of the ribbon to detect the hard edges. Once found, the vertices at that edge must be duplicated and the normal of the quadrilaterals are set only based on the normal of the quadrilateral itself (not using the average of neighbouring quadrilaterals). Care must be taken to avoid duplicating too many vertices. The total number of vertices is $4N + 2H$, $H$ being the number of hard edges along the ribbon. The final number of vertices for a worst case scenario is thus $4N + 2H = 4N + 2N = 6N$

![Figure 40](image1)

Figure 40: The "D" in front is using the improved version of the normals where the "D" behind is using the final version of the normals. The difference around the corner is well defined when these are compared side to side.

![Figure 41](image2)

Figure 41: Comparison of extrusion methods: naive (top), improved (middle), and final (bottom). The left hand side stack is rendered using Gouraud shading, while the right hand side stack is rendered using Phong shading. See Section 3.7 for more details.
7 Performance assessment

7.1 Benchmark protocol

To be able to retrieve useful information to analyse, a good benchmark protocol must be created. The timings must be compared for many different fonts and for multiple categories of texts. An exhaustive list of the tested parameters can be found in Appendix \[A.1\].

The fonts were chosen during a meeting with my industrial supervisors, their choices were directed by the most common fonts used by their clients. The texts chosen follow the same reasoning and are there to test typical cases.

The following fonts were tested:

- Tahoma
- Arial
- Comic Sans MS
- Times New Roman
- Courier New

The categories of texts were defined as follows:

- Character:
  - All letters of the latin alphabet (lower and uppercase), all digits, and all the common symbols found on the Belgian keyboard layout (108 total)
  - The 26 most used Arabic characters
  - The 20 most used Chinese characters
  - The 20 most used Japanese characters

- Small words and typical cases:
  - "Hello world" — "Deltatec"
  - "Tirs cadrés" — "DELTACAST"
  - "Fautes"
  - "Cartons rouges"
  - "Cartons jaunes"
  - "Au revoir"
  - "Julien Vandenbergh"
  - "Terry Moës"
  - "Sébastien Magdelyns"
  - Same words translated to Arabic (11 total)
  - Same words translated to Chinese (11 total)
  - Same words translated to Japanese (11 total)

- Long texts:
  - Lorem ipsum [...] (About 128 characters long)
Figure 42: "Hello world" in Arabic (top), Chinese (middle), and Japanese (bottom).

For each algorithm (including DirectX), quality parameter, font, and text (for all categories of text), the full mesh generation algorithm is performed and its performances is timed. The test is actually done more than once (typically 1000 times) at a time and only the average is considered. The part of the code which is tested is only composed of the mesh creation, it starts when giving the text to be processed and stops as soon as the mesh is returned. The setup is not included in the times provided as it can be done only once for multiple meshes. The setup is made of the selection of font and quality parameters.

Another benchmark is done by generating images (both filled mesh and wire framed mesh) of each of the texts to be able to check visually the resulting quality. Examples can be found in the appendix A.2. These images are not generated for the DirectX algorithm because they can directly (and dynamically) be visualised in Template Editor. Moreover, these images are only used to visually check the results of the developed algorithm.
7.2 Results

7.2.1 Overall performance comparison

The first comparison is to compare the average results by each algorithm over the whole benchmark. As we can see on Figure 43, DirectX is about two times quicker than both Constrained Delaunay Triangulation and Monotone Polygons Triangulation. As anticipated, the results obtained are worst than the ones of DirectX (See Section 3.8), which can be explained by a better optimised code. But it should be noted that a factor of two is quite small, which let us think that it should be achievable.

![Comparison performances for all algorithms.](image)

Figure 43: Comparison performances for all algorithms.

7.2.1.1 Detailed cost of developed code

To have a little bit more insights about the code developed, a similar benchmark was done to compute the time taken by the different steps: extraction of the parametric form, the discretisation, and the triangulation.

The results are clear, the discretisation is not significant compared to other parts (<0.1%). The extraction of the glyph information is bound by the performance of Direct 2D, but it amounts around 5% which is not a bottleneck. Almost all of the time is passed inside the triangulation (~ 95%), this must be the main focus when optimising.
7.2.1.2 Extrusion

As the extrusion was not optimised, a full benchmark was done without it to have a sense of the importance of the time loss. This was a real discovery to see that the extrusion is actually responsible of more than 50\% of the whole time spent. As a comparison, Figure 45 shows the differences between with extrusion and without.

Figure 44: Detailed timings of the 3 sub-parts. Discretisation is $\leq 0.1\%$ and thus not visible on this graph.

Figure 45: Comparison performances with or without extrusion. DirectX always has extrusion.
As a big surprise for everyone, the unextruded version implemented is faster than DirectX in all considered situations. This information is very important because it means that with some optimisation to the extrusion, there is hope that the performances are similar or even better than those of DirectX. As discussed in more depth in Section 8.1, the extrusion can heavily be optimised.

### 7.2.2 Category comparison

To be sure that the trends are not influenced by the category of texts, a comparison focused on the difference between algorithms given each category. It does not seem that the trends are influenced by the category, they look the same as in the Figure 43.

![Figure 46: Comparison performances for all categories.](image)
7.2.3 Quality parameters comparison

The same trends check is made on the quality parameters. The results are identical to the category of texts, meaning that the algorithms behave all in the same way whatever the quality parameters used with respect to each other.

![Figure 47](image)

**Figure 47:** Comparison performances for all quality parameters.

7.2.4 Font comparison

The same trends check can be done with the same results on the fonts. We can thus conclude that no algorithm has a preferred or undesirable setting.

![Figure 48](image)

**Figure 48:** Comparison performances for all tested fonts.
7.3 Compatibility with current engine

During the development of the algorithm, the conventions were matching between those used and the ones of DirectX. The mesh generated are totally compatible with the current engine. The algorithm can be included as a library to input directly the information into the engine if required.
8 What’s next

The follow up of this work would be composed of two main parts: the optimisations and the bevelling.

8.1 Possible Optimisations

As stated multiple times, almost no optimisation was made, especially for the extrusion. This section is thus reserved entirely to detail the possible optimisations.

8.1.1 Complexity improvements

While thinking about the possible optimisations, it might be useful to check if possible improvements can be made solely based on the computational complexity. But in this case, the algorithms used are all at their lower bound.

8.1.1.1 Discretisation

The discretisation cannot be improved because it is already the most efficient algorithms for discretisation of Bézier curves.

8.1.1.2 Triangulation

Both triangulation algorithms implemented are already at the lower bound of the computational complexity. No further improvements can be made.

8.1.1.3 Extrusion

The extrusion is not optimised for many reasons — discussed below, but it cannot be improved from the points of view of the computational complexity because the computational complexity is bound to the size of the output.

8.1.2 Glyph extraction into the sink

The possible optimisations are mainly towards a better management of the memory. As of now, the discretised points generated are saved into a dynamic data structure which leads to many dynamic memory allocations. The number of discretised points cannot be known in advance because it depends on the input text, font, and quality parameters chosen, but a minimum can be guessed to reduce the number of reallocation by using reserved memory.

The memory footprint can also be reduced while maintaining an efficient detection of the order of enumeration (clockwise or counter-clockwise). Currently, a list of key points is saved during the extraction of the parametric form, then used to determine the order of enumeration once the figure is completed. This list can be reduced to only three points. This would both reduce the memory footprint but also the dynamic allocations because the number of key points is also unknown.

8.1.3 Monotone Polygons

The optimisations are also mainly towards a better management of the memory. Almost all the data containers used are dynamic which is not necessary. Moreover, the ownership of that dynamic memory is not managed well, leading to extra copies to ensure that the memory is still reachable.
At many places, map and set containers (from the STL) are used where their unordered version can be used, increasing the efficiency of the algorithm. The complexity would become $O(1)$ compared to $O(\log(n))$ for the most used operations on those containers.

8.1.4 Extrusion

The memory is dynamically allocated for all the duplicated vertices which is not necessary because the lower and upper bound of the memory used is known: The lower bound is $4N$ and the upper bound is $6N$, as described in Section 6.3. There exist at least two possible solutions with each their advantages and drawbacks:

- Allocate a single memory block of size $6N$ (in RAM) and record the number of vertices inside it. Once the extrusion is done, resize the memory block to fit the actual size used (or upload it to GPU memory (DirectX memory)).
  - Advantages: There is no dynamic memory allocation.
  - Drawbacks: The data must be copied to a smaller buffer in RAM or copied to GPU memory. It cannot be written directly to GPU memory because the size is unknown and the full size of the buffer needs to be used if allocated in GPU memory.
- Allocate $6N$ directly in GPU memory (DirectX memory) and duplicate all the vertices as if all edges are hard but with the same computation of normals as in the final version of the extrusion.
  - Advantages: There is no dynamic memory allocation and the information is directly set in GPU memory (DirectX memory).
  - Drawbacks: It can use up to $2N$ vertices more than required. If $N$ is big, it can become a problem for GPU memory.

8.1.5 Multi-threading

The triangulation algorithms cannot benefit directly from multi-threading because of their sequential nature. But as the triangulation and extrusion is done by independent group of figures, these can be easily parallelised to decrease the overall time needed to generate the full mesh.

Some modifications would be required to efficiently merge the results into a single mesh.

8.2 Bevelling

The time was too short to tackle the bevelling but a quick overview of the existing methods revealed a potential entry point.

The technique is called edge chamfering and consists of a subdivision of the hard edges into a small face that will be used to interpolate the normals, creating a smooth gradient effect. This effect is the same as the problem with our naive approach to extrusion but this time in a controlled manner to have a smooth lighting. The hard edges are modified not to be round but to give the effect of being round with lighting.
To improve this effect with the addition of a round corner, we could iteratively apply the chamfering until the corners are round enough, given some stopping criteria.

This technique might not be the way to go but it is nonetheless an entry point to the bevelling of meshes.
9 Conclusion

This thesis propose an implementation for the generation of 3D mesh. The algorithm takes as input a text and outputs a 3D mesh with normals that can be used directly inside a 3D rendering engine.

The extraction of parametric forms and their discretisation is efficient and gives qualities equivalent to the current solution.

The mesh can be generated by either the Monotone Polygons Triangulation algorithm or by Constrained Delaunay Triangulation algorithm. Both of them are efficient and have close timings. But, in practice, the Constrained Delaunay Triangulation algorithm will be used because of the shape of the resulting triangles are more equilateral. We also found that the algorithm that could be used by DirectX is probably Monotone Polygons. The extrusion can be optimised to improve drastically its performances. Given the results obtained by this work, it is promising to use them to replace the current solution.

The solution developed during this thesis can be included, once optimised, into Deltatec’s proprietary engine.
A Appendix: Benchmark

A.1 Benchmark detailed variables

A.1.1 Algorithms

As discussed before, 3 algorithms were used:

- Monotone polygons
- Delaunay
- DirectX 9 (unknown method which is most probably Monotone polygons)

A.1.2 Fonts

The fonts used were the following ones:

- Tahoma
- Arial
- Comic Sans MS
- Times New Roman
- Courier New

A.1.3 Characters

Latin

³, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, °, _, A, Z, E, R, T, Y, U, I, O, P, ¬,
², & e, ",, , (, §, é, !, ç, â, ), ¬, a, z, e, r, t, y, u, i, o, p,
¬, $, q, s, d, f, g, h, j, k, ,m, ü, µ, w, x, c, v, b, n, ,, ;, :, =,
<, {, @, #, ¬, {, }, ], [, ' , , ¬, \\

Figure 50: Arabic characters.

Figure 51: Chinese characters.
A.1.4 Small words and typical cases

Latin

"Hello world!", "Au revoir", "Deltatec", "DELTACAST",
"Julien Vandenberghe", "Terry Moës", "Sébastien Magdelyns",
"Tirs cadrés", "Cartons jaunes", "Carton rouges", "Fautes"

Arabic

"سَيُومَ كَرِيْت"، "إِقْرَأْنَيْنَا فَنَافِرِعْنَاهُ"، "عَمَّلَ صَبِيلَاً امْتَلَأَ"، "قَمَتَ امْتَلَأَنِ"، "إِقْرَأْنَا إِلَى"، "مَعَابِدْ امْتَلَأَتْ"، "نَكِيْدَةُ امْتَلَأَتْ"، "عَمَّلَ صَبِيلَاً امْتَلَأَتْ"، "فَارِضَ النَّافِرِيْت"، "عَمَّلَ صَبِيلَاً امْتَلَأَتْ"، "إِقْرَأْنَيْنَا فَنَافِرِعْنَاهُ"، "عَمَّلَ صَبِيلَاً امْتَلَأَ"، "قَمَتَ امْتَلَأَنِ"، "إِقْرَأْنَا إِلَى"، "مَعَابِيدْ امْتَلَأَتْ"، "نَكِيْدَةُ امْتَلَأَتْ"، "عَمَّلَ صَبِيلَاً امْتَلَأَتْ"، "فَارِضَ النَّافِرِيْت"، "عَمَّلَ صَبِيلَاً امْتَلَأَتْ"، "قَمَتَ امْتَلَأَنِ"، "إِقْرَأْنَا إِلَى"， "مَعَابِيدْ امْتَلَأَتْ"، "نَكِيْدَةُ امْتَلَأَتْ"， "عَمَّلَ صَبِيلَاً امْتَلَأَتْ"， "فَارِضَ النَّافِرِيْت"， "عَمَّلَ صَبِيلَاً امْتَلَأَتْ"， "قَمَتَ امْتَلَأَنِ"， "إِقْرَأْنَا إِلَى"، "مَعَابِيدْ امْتَلَأَتْ"، "نَكِيْدَةُ امْتَلَأَتْ"．

Figure 53: Arabic small words and typical cases.

Chinese

"你好世界", "再見", "台達", "三角洲演員", "朱利安·范登伯格", "特里·莫斯",
"塞巴斯蒂安·馬格德林斯", "射門", "黃牌", "紅牌", "犯規"

Figure 54: Chinese small words and typical cases.

Japanese

"ハローワールド", "さようなら", "デルタテック", "デルタキャスト", "ジュリアン・ヴァンデンバー", "デリー・モース", "セバスチャン・マグデリン", "ショットショット", "イエローカード", "レッドカード", "ファウル"

Figure 55: Japanese small words and typical cases.

A.1.5 Long text

The only long text used was: "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua."

A.2 Samples of generated tessellations
Figure 56: Example of generated images. Top left is using Monotone polygons. Top right is using Constrained Delaunay. At the bottom both are using Constrained Delaunay to show the result on non-latin characters.
Appendix: Demonstration of complexity reduction

Let us define \( n_i \) as the number of points in a group of figures, thus the total number of points \( N \) is such that

\[
N = \sum_{i} n_i
\]

We can show that:

\[
\sum_{i} n_i \log(n_i) \leq N \log(N)
\]

Let us defined:

\[
f_i = \frac{n_i}{N}
\]

and thus

\[
\sum_{i} f_i = 1
\]

\[
\sum_{i} n_i \log(n_i) = \sum_{i} f_i N \log(f_i N)
\]

\[
= \sum_{i} [f_i N (\log(f_i) + \log(f_i N))]
\]

\[
= \sum_{i} f_i N \log(f_i) + \sum_{i} f_i N \log(N)
\]

\[
= N \sum_{i} f_i \log(f_i) + N \log(N) \sum_{i} f_i
\]

\[
= N \log(N) - |N \sum_{i} f_i \log(f_i)|
\]

\[
\leq N \log(N)
\]

This is true because

\[
N \sum_{i} f_i \log(f_i) \leq 0
\]

Where \( N \geq 0, f_i \in [0, 1] \) thus \( \log(f_i) \in ]-\infty, 0] \) and \( f_i \log(f_i) \leq 0 \). We can then conclude that the hypothesis is true.
The previously used tie breaker was to match the contour with the biggest shared area with the hole. This approach is efficient but has a major flaw. If the Figure 14 is slightly modified (See Figure 57), the tie breaker outputs the wrong contour.

As we can see on Figure 57, the sizes of the bounding boxes are: $B_1$: 5x5, $B_2$: 6.5x6.5, $B_3$: 2x2. The previous tie breaker will match $B_1$ to $B_3$ because $\frac{B_3}{B_1} \geq \frac{B_1}{B_2}$. If we use the new tie breaker which matches to the closest contour to the hole, it will always outputs the correct contour $B_2$. 

Figure 57: Erroneous tie breaker case.
References


