Parameterized Leaf Shape Generation

Cem Kalyoncu

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of

Master of Science
in
Computer Engineering

Cyprus International University
November 2009
Lefkoşa, North Cyprus

Approval of the Institute of Science

Asst. Prof. Mehmedali Egemen
Director (acting)

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Assoc. Prof. Dr. Hasan Demirel
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Asst. Prof. Dr. Erbuğ Çelebi
Supervisor

Examining Committee

1. Assoc. Prof. Dr. Hasan Demirel _____
2. Asst. Prof. Dr. Erbuğ Çelebi _____
3. Asst. Prof. Dr. Devrim Seral _____

ABSTRACT

Leaves are an important part of plants in nature. In this study, we have introduced a method and parameters to build an artificial leaf. Our proposed model consists of simple and intuitive parameters that can easily be understood by a graphic designer. The model is based on positional information system where graphical methods are used. We also implemented the model with an application that allows the end users to design, save and build leaves for their graphic design projects. Although there are other studies working on plant generation, our study is the only one that defines a leaf completely which can produce usable results. The resultant leaf images can either be used as texture in plant generation tool or as a clip-art in graphic design projects. We have also demonstrated the use of generated leaves in various works.

Table of Contents

Chapter 1 – Introduction

In this study, we have studied on leaf generation and representation problems of computer graphics.. In this chapter, we have defined the problem, our solution approach, previous research on the subject and technologies we have adopted to reach proposed objectives.

1.1. Problem Definition

Nowadays computer generated scenes are used to achieve artificial realism. Human made objects are created with incredible accuracy. However, creating nature scenes are still a challenge; this is caused by the complexity and variability of nature. This issue can also be seen in organic objects like trees and plants. Manually creating a plant or a tree takes too much time. Therefore, tools to generate these organic objects are introduced. Solutions are proposed by both academic and commercial researchers. The existing research focuses into constructing trees or plants, specially 3D structures which are considered more than 2D silhouettes or textures of the objects. Most importantly, leaf or bark textures of trees or plants are left to the designer to be found and placed. While its relatively easy to take a photo of a tree bark and unfold it using simple graphic editors, photographing a leaf is harder; most leaves are curved leading to non linear perspective and lighting imperfections. It requires specially crafted studio to achieve good results. Moreover, a camera with macro functionality is required. Even with these equipments, it requires access to the desired plant. After the photo of the leaf is taken, its background should be removed too. Moreover, in the light of todays hardware improvements, bump or displacement

maps are used in 3D projects. However, obtaining these maps requires expensive equipments to extract displacement maps of real life objects. Leaves have slight difference in height, therefore it requires high quality hardware to capture these maps accurately. However, slight height variance creates huge difference in lighting, therefore, it cannot be omitted. These difficulties lead to scenes having repeatedly same leaf, reducing the believability and quality of the generated scene.

Shape of a leaf gives feelings of nature, growth and forest. In graphic design, specifically in printed media, leaf silhouettes are used extensively to translate these positive feelings. Although many leaf silhouettes and “clip arts” are available, it is hard to find a matching clip art for a specific design. Moreover, most of the better quality clip arts are not available freely. This leads to designers creating leaves that they require by themselves. This task is long and daunting.

The problems that are related with computer graphics can be solved with a tool that allows us to create leaf textures or shapes. For such a tool one must formalize the visual aspects of a leaf and parameterize them. This parameterizing process allows to design a leaf once to create any number of similar leaves. Moreover, defined leaf can act as a blueprint of that leaf type and these blueprints can become a library. After parameters are defined, a tool can be created and a graphic designer who is equipped with it can create the texture, silhouette, or bump

Figure 1.1: *L-System Trees*

map of any leaf that he requires with no lighting or perspective imperfections and without the need of actual plant.

1.2. Goals

We have two primary goals in this study. First, is to examine various leaves and build a formalized list of parameters that can define a leaf. These parameters should be sufficient to create a leaf shape without the need of any external help and their numbers should be small to reduce complexity.

Second aim of this study is to create an easy to use application that implements this model which can solve the problem of creating leaves. The application itself should address as many problems as possible. Our application should allow both bitmap and vector versions of a designed leaf, while supporting to save and load leaf structures for later use. This tool should also be capable of creating different leaves from the same set of parameters allowing variance that can easily be observed within the nature. Moreover, to target larger audience, the application should be easily understood and should not require any specific skill.

1.3. Previous Works

The generation/simulation of organic structures is an interesting subject of computer graphics. Many algorithms and methods are developed for this field ([1], [2], [3], [4]). These algorithms and methods can be divided into two categories. The first method is to follow the nature closely to achieve biologically realistic results. The simulation approach helps us to understand the nature better. The other method of creating an image of an organic structure is to generate only visible elements.

Many researchers have worked on this subject of tree and plant creation using both methods. However, the effort on creating leaf textures and shapes are limited [5].

In an attempt to create a method to simulate leaf venation pattern, thus creating a leaf texture, a group of researchers used biological methods [6]. In that research, it is shown that the leaf venation pattern is closely related to shape of the leaf blade. Therefore, with the input of the leaf shape, growth parameters, and venation parameters user can have a biologically identical leaf pattern. This research helps us to understand growth pattern of leaf veins. However, their approach has serious drawbacks. First of all, the requested parameters are difficult to understand; therefore, a user without prior knowledge on the subject cannot use the system effectively. Since the parameters are numerical, it is hard to create the desired leaf pattern. Moreover, system requires an existing leaf blade shape and attractors to grow. Finally, the result is black and white; therefore, it must be colored manually.

Another method of generating organic structures is to use fractals. Since plants tend to repeat themselves, using fractal methods for plant and leaf growth can be used. Best example of this method is presented by L-System fractals [4]. Using L-System Fractals many researches tried to create trees and plants. Figure 1.1 shows trees that are generated with this method. Oppenheimer, also shown that leaves can be generated using fractals. However, only a number of leaves can be modeled this way. However, not every plant has leaves possessing self similarity.

A plant modeling system using positional information was proposed by [1]. In this system, user is asked to supply simple and graphical (curves to define plant organs, placements, etc...) parameters to create the desired plant shape. This research

has proven that positional information can be used in organic modeling to reduce the required knowledge level to use the system. Moreover, with this method, it is possible to create visually appealing plant models with less computational power. Their research is quite similar to our study, however their field is plant models and not leaves.

[7] has proposed a method of creating 3-D geometry for leaves. Their system divides a leaf into polygons to allow deformations to be made. Therefore, using this system, 2-D leaf textures, such as the ones generated by our project, can be deformed and rendered in 3-D scenes.

[8] shows an algorithm to combine edges of a leaf blade. Without using this algorithm, compound leaf blades cross each other and create artificial looking sharp edges. This algorithm combines these blades smoothly; therefore, it helps to create better looking compound leaves. However, after our research, we have found that ordinary Bézier union and smoothing works for leaves. Using a well known method instead of complex methods should be valued. However, Bézier operations can be performed if output of the system is Bézier paths, instead of a bitmap image.

[5] proposed a method to reconstruct leaves using a scanned image along with a user input. However, the proposed method only covers external shape of the leaf and focuses on constructing mesh structure rather than image of it.

XFrog [9] is a commercial application which is well-known and widely used for plant modeling. This application possesses the traits of positional information system. Although this system is powerful and widely used, it has no support to

generate leaves shapes or images. On the other hand, this application requires readily available leaf textures to be effective.

1.4. Standards Used

1.4.1. XML

In this study, we are required to choose a method to represent leaf data. Our study aims to create an application that can be used along with other applications. Therefore, we have chosen XML to save files for the reasons explained below.

Extensible Markup Language (XML) is a human-readable, machine-understandable, general syntax for describing hierarchical data, applicable to a wide range of applications such as databases, e-commerce, web development, searching. XML is a very flexible text format that contains structured information, that have some structures, that contain contents like words, pictures and have some indication of what role that content plays.

XML is a meta-language for describing markup languages, mechanism to identify structures in a document, that are defined as a standard by the XML specification. XML provides a facility to define tags, that enable the definition, transmission, validation, and interpretation of data between applications and between organizations, and the structural relationships between semantics and a tag set. All of the semantics of an XML document will either be defined by the applications that process them or by style sheets.

XML has an extensibility that allows you to define and share your own markup that may have meaningful names for all your information items. XML is also heavily used for enclosing or encapsulating information in order to pass it between different computing systems that can be either internal or external such as vendors, customers, partners. By its system interoperability, knowledge transfer can be easier between different computing teams.

1.4.2. Vector Graphics and SVG

Our application is required to create usable output. We have decided to use vector output for quality. To make the output scalable and usable in many applications while being standard compliant, we have used SVG (Scalable Vector Graphics).

Vector graphics is employed for the creation of digital images through a sequence of commands or mathematical statements that place geometrical primitives such as points, lines, curves, and shapes or polygons in a given 2D or 3D space. They are made up of any individual objects and they have individual properties assigned to them such as color, fill, and outline. Vector graphics are resolution independent because they can be output to the highest quality at any scale. Vector-oriented images can be resized, stretched, and rotated without loosing quality. They are stored as vectors which look better on devices such as monitors and printers with higher resolution and often require less disk space. Moreover, it is easy to modify vector images. However, it is not possible, at least not logical, to represent every image as a vector graphic. Sometimes textures are used to achieve more realism. On graphics design vector-oriented images have more advantages than bitmaps because of

realizability and being easy to edit. Commonly used design applications that use vector images includes Inkscape (SVG), Corel Draw (CDR), Adobe Illustrator (AI), Freehand (FH), and Adobe Flash (SWF, FLA). [10]

SVG is a language for describing two-dimensional graphics and graphical applications in XML. Because SVG is text-based, it is easy to create. A sample SVG document is displayed in Appendix B. Key features include shapes, text and embedded raster graphics, with many different painting styles. SVG has been developed as a standard format by the World Wide Web Consortium (W3C) for displaying vector graphics on the Web. SVG enables Web documents to be smaller, faster and more interactive. Being a vector format, it does not suffer resolution problems.

SVG is well-supported in the majority of modern browser, with active development and rapid improvement in both interoperability and performance. SVG images can be scaled up or down to fit proportionally into any size display because of resolution and device independently. SVG images can be rendered with different CSS styles for each environment. They work well across a range of available bandwidths. SVG makes it possible for designers to escape the constant need to update graphics by hand or use custom code to generate bitmap images.

SVG is used in many business areas including Web graphics, animation, user interfaces, GIS (Geographic Information Systems) and mapping, embedded systems, graphics interchange, print and hard-copy output, mobile applications and high-quality design. [11]

1.4.3. OpenGL

Open Graphics Library (OpenGL) is an application programming interface that is used to define 2D and 3D computer scenes. It is a cross-platform API that is generally considered to set the standard in the computer industry when it comes to this type of interaction with 2D computer graphics and has also become the usual tool for use with 3D graphics as well. Moreover, OpenGL is available for almost all programming languages.

OpenGL is a collection of several hundred functions providing access to all the features offered by graphics hardware. The graphics hardware may comprise varying degrees of graphics acceleration, from a raster subsystem capable of rendering two-dimensional lines and polygons to sophisticated floating-point processors capable of transforming and computing on geometric data. Internally, it acts as a state machine that is a collection of states that tell OpenGL what to do. Using the API, you can set various aspects of the state machine, including such things as the current color, lighting, blending, and so on. When rendering, everything drawn is affected by the current settings of the state machine. OpenGL possess the following features.

- 2D image scaling
- Rendering 3D objects including spheres, cylinders, and disks
- Texture mapping and automatic mipmap generation from a single image
- Support for curves surfaces through NURBS
- Support for tessellation of non-convex polygons
- Special-purpose transformations and matrices
- Illumination, blending and transformations

The OpenGL draws primitives subject to a number of selectable modes. Each primitive is a point, line segment, polygon, or pixel rectangle. Primitives are defined by a group of one or more vertices. A vertex defines a point, an end point of an edge,or a corner of a polygon where two edges meet. Data, consisting of positional coordinates, colors, normals, and texture coordinates, are associated with a vertex and each vertex is processed independently, in order, and in the same way.

The OpenGL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, anti-aliasing methods, and pixel update operators. [12], [13].

1.4.4. PNG

Vector graphics have its strengths, however, in some projects raster (bitmap) images are preferred. In addition, since widget engine does not support vector images, our user interface should contain bitmap images. Because of its advantages, we have chosen Portable Network Graphics (PNG) image format in our project.

PNG is a bitmapped image format that employs lossless data compression. PNG was created in response to the GIF licensing debacle and is optimized for graphics use on the Internet and other on-line services. PNG supports image depths up to 24-bit and provides a better lossless compression than that found in GIF files. Moreover, PNG has alpha channel and losses compression which is not supported by JPG.

PNG has three main advantages over GIF; alpha channels, variable transparency, gamma correction and control of image brightness. PNG also

compresses better than GIF in almost every case. PNG provides a useful format for the storage of intermediate stages of editing. Since compression method used is fully lossless and supports up to 48-bit true color or 16-bit gray scale saving, restoring and re-saving an image will not degrade its quality, unlike standard JPEG. The PNG specification leaves no room for implementors to pick and choose what features they'll support; the result is that a PNG image saved in one application is readable in any other PNG-supporting application. [14]

1.5. List of Symbols and Abbreviations

In this section, we added the list of abbreviations used in this document.

- **API:** Application Programming Interface
- **GGE:** Gorgon Game Engine
- **GIF:** Graphics Interchange Format
- **GIMP:** GNU Image Manipulation Program
- **GRE:** Gorgon Resource Engine
- **JPEG:** Joint Photographic Experts Group
- **LZMA:** Lempel-Ziv-Markov Chain Algorithm
- **NURBS:** Non-Uniform Sational B-Spline
- **OpenGL:** Open Graphics Library
- **PNG:** Portable Network Graphics
- **SCC:** Cascading Style Sheets
- **SVG:** Scalable Vector Graphics
- **UML:** Unified Modeling Language
- **XML:** Extensible Markup Language
- **XSD:** Extensible Markup Language Schema
- **OpenAL:** Open Audio Library

Chapter 2 – Leaf Anatomy and Morphology

To generate a leaf shape and texture we have analyzed leaves in the nature. This chapter defines leaf anatomy and morphology. This chapter is the key to understand rest of this document.

2.1. Terminology

This section describes the terminology used in leaf anatomy and morphology. We have also mentioned terminology used in our study. In Figure 2.1, leaf organs are illustrated.

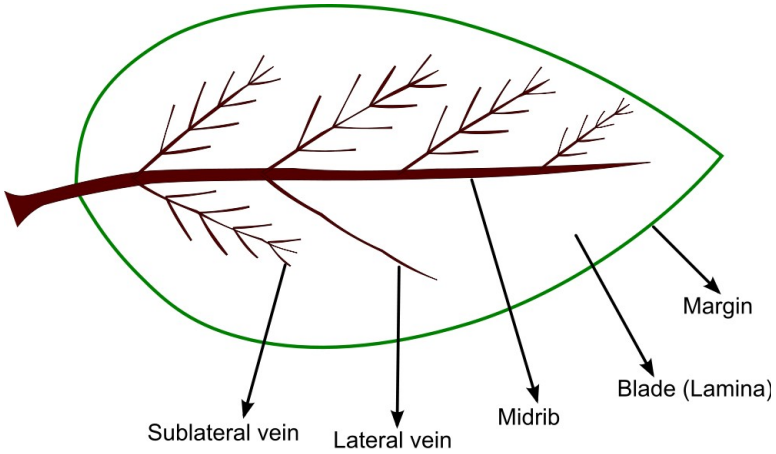


Figure 2.1: *Leaf organs*

- **Midrib:** is the first vein level, its directly connected to the stem
- **Secondary veins:** veins that are growing from the midrib
- **Lateral vein:** veins that are growing from the both sides of midrib
- **Blade (Lamina):** leaf surface
- **Tip:** either the top section of the blade or the last sub leaves that grows from the tip of the midrib

- **Margin:** The edge of a leaf
- **Lobe:** A rounded or pointed section of a leaf
- **Sinus:** The area between the lobes of a leaf
- **Leaflet:** is a part of a compound leaf that resembles a smaller leaf.
- **Leaf level:** definition of a vein branch or sub leaves of a main leaf where they are created and adjusted by the user
- **Leaf, sub leaf:** created by the system by considering the given parameters and they consist of set of Bézier paths to be drawn

2.2. Leaf Shapes

There are many different leaf shapes in the nature. They are categorized by botanists for easy identification. Figure 2.2 shows different types of leaves. The following is the list of common leaf shapes.

- **Acicular:** needle shaped (Figure 2.2 - a)
- **Falcate:** hooked or sickle shaped (Figure 2.2 - b)
- **Orbicular:** broad and circular (Figure 2.2 - c)
- **Rhomboid:** diamond shaped (Figure 2.2 - d)
- **Elliptic:** broad, oval shaped leaf (Figure 2.2 - e)
- **Obovate:** egg shaped, wide at tip (Figure 2.2 - f)
- **Hastate:** triangular with base loves (Figure 2.2 - g)
- **Spatulate:** spoon shaped (Figure 2.2 - h)
- **Lanceolate:** pointed at both ends (Figure 2.2 - i)

- **Obcordate:** heart shaped, pointed base (Figure 2.2 - j)
- **Ocate:** egg shaped, wide at base (Figure 2.2 - k)
- **Cordate:** heart shaped, pointed tip (Figure 2.2 - l)
- **Flabelate:** fan shaped
- **Deltoid:** triangular leaf

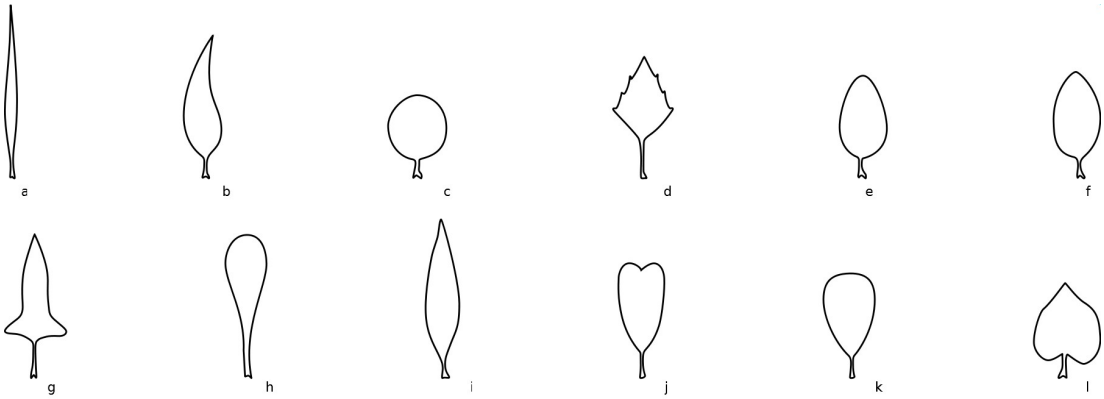


Figure 2.2: *Various leaf shapes*

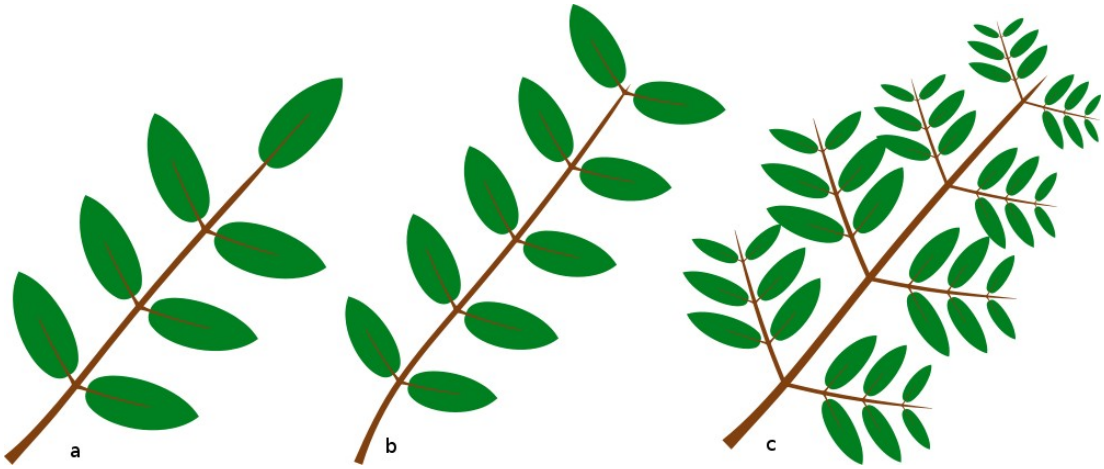


Figure 2.3: *Pinnate leaves, a. odd pinnate, b. even pinnate, c. bipinnate*

2.3. Leaflet Arrangements

Compound leaves has leaflets which are located on the midrib. Different leaves has different leaflet arrangements. The most common arrangements are pinnate, palmate, alternate and rosette. In pinnate arrangement, leaflets sprout from both sides

of the midrib. Figure 2.3 shows pinnately compound leaves. Pinnate leaves are classified as odd pinnate or even pinnate according to the number of leaves sprout from the top. Bipinnate leaves have leaflets that are also pinnately compound. In alternate arrangement leaflets alternately sprout from the sides of midrib. In palmate arrangement, leaflets are radiated from the base of the leaf. Leaflets can vary in size and shape. Figure 2.4 shows alternate and palmate arrangements.

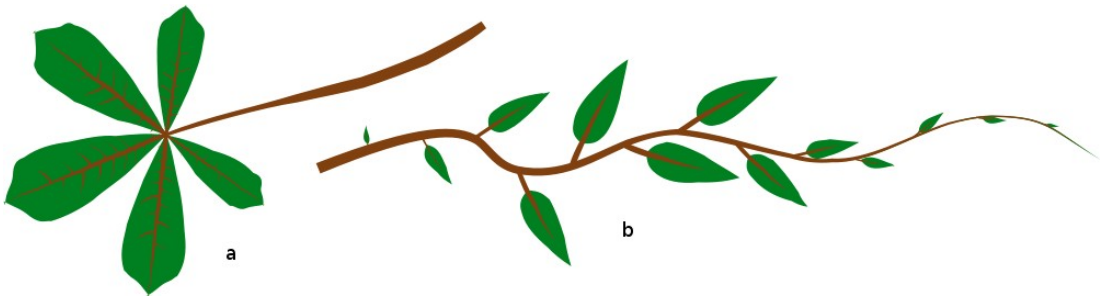


Figure 2.4: *a. Palmate and b. alternate arrangement*

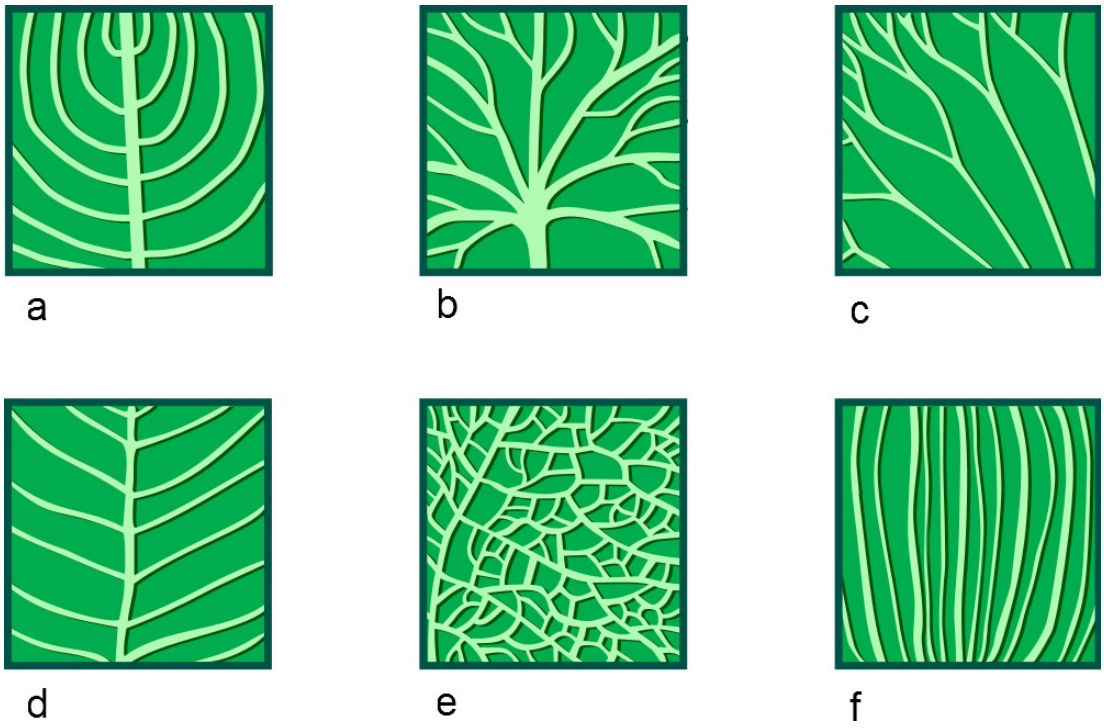


Figure 2.5: *Leaf venation patterns: a. arcuate, b. dichotomous, c. palmate, d. pinnate, e. reticulate, f. parallel*

2.4. Venation Patterns

Leaves have veins to supply water and minerals to its cells. There are different distribution patterns of veins. Venation pattern modifies texture of the leaf. Figure 2.5 shows common leaf patterns that are listed below.

a) **Arcuate:** secondary veins bends towards the tip

b) **Dichotomous:** veins branch symmetrically

c) **Palmate:** veins distributed from a central point

d) **Pinnate:** secondary veins are paired oppositely

e) **Reticulate:** smaller veins form a network

f) **Parallel:** veins are parallel to each other

2.5. Margin Types

The leaf margin is the boundary area extending along the edge of the leaf. There various types of margins. Which are illustrated in Figure 2.6. Some of the margins are defined in the following:

Entire type has a smooth edge with neither teeth nor lobes. These leaves are rather simple. Lemon tree leaf is an example to this type.

Toothed leaves have a saw like margin with small tooth that can vary in size, from very small to medium; in sharpness, from needle-like to soft; and in shape from rounded to points. Toothed margin is further divided into following types:

- *Serrate:* teeth facing to the tip
- *Dentate:* teeth facing outside the leaf

- *Crenate:* round and broad teeth
- *Incised:* deeply cut with sharp, irregular teeth

Lobed leaves have some type of indentation toward the midrib that can vary in profundity and shape, rounded or pointed. In our system, generally, lobes are not defined using margin shapes, they are an extension of the veins. [15], [16], [17]

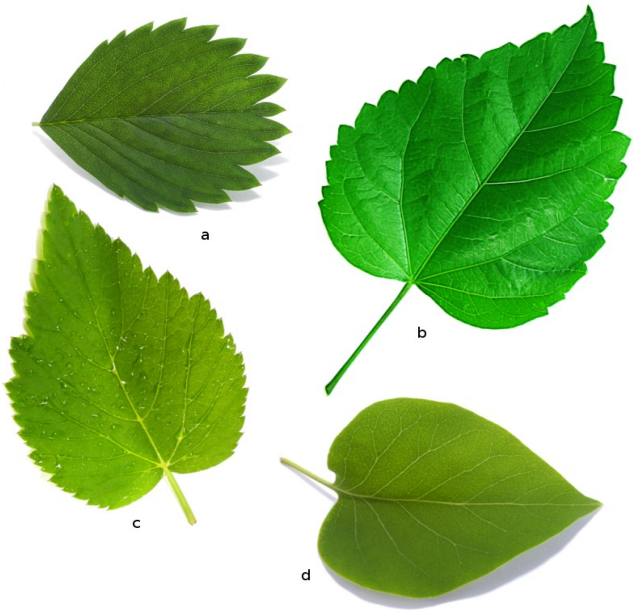


Figure 2.6: *Various margin shapes; a. serrate, b. crenate, c. doubly serrate, d. entire*

Chapter 3 – Leaf Modeling

We have proposed a method using positional information system method with specifically chosen parameters to create a leaf. In a positional information system, an organic object is represented as positional parameters which allow user to specify different values for different positions [1]. These parameters are represented with functions or better yet, curves. In this research, we have decided that using Bézier curves will allow defining positional parameters easily.

We have observed different leaves and searched for common parameters that exist in them. As the result of this observation, we conclude that a leaf should be represented as a hierarchy of smaller leaves and/or veins that have different values for parameters and a method to distribute them within its parent. While choosing these parameters, we have taken care to keep number of parameters less, while still having the ability to represent different types of leaves. Keeping number of parameters less helps users to learn the system faster and reduces the time required to generate the leaf.

We have categorized these parameters into five different groups. It is important to note that some of the parameters discussed in the following sections are Bézier curves. These parameters either denote a shape, or a value that can change along the leaf. In the following list parameter hierarchy is displayed.

- **Hierarchy**
  - Sub levels
  - Name

- **Shape**
  - Strength
  - Length/strength
  - Curvature
  - Maximum length
  - Straighten
  - Blade shape
  - Show blade
  - Vein shape
  - Vein length
  - **Margin**
    - Margin shape
    - Margin repeat
    - Margin size relation
      - *none*
      - *blade size*
      - *blade strength*
      - *margin strength*
    - Margin strength
    - Leaf strength relation
- **Placement parameters**
  - Distribution method
    - *free*
    - *opposite*
    - *alternate*
    - *circular*
    - *top/down*
  - Spacing
  - Angle
  - Tip

- **Randomization**
  - **Bézier**
    - Curvature
    - Blade shape
    - Vein shape
    - Margin shape
  - **Value**
    - Strength
    - Length
    - Margin repeat
    - Angle
    - Spacing

3.1. Hierarchy

There are tree basic leaf types: simple, compound and lobed. All these leaf types has a hierarchy of smaller leaves and/veins. *Simple* leaves have veins inside a single leaf blade. *Compound* leaves have leaflets which are actually simple. *Lobed* leaves have multiple blade fragments that are combined together to form complete leaf blade. They also have veins reaching to the lobes. For the sake of simplicity, leaflets also represent lobes in this document; unless specified otherwise. A careful observation shows that we have leaflets (a simple leaf is also a leaflet) and veins. Since leaflets and veins are bent and placed similarly (even used together), we have decided to have only one type of object; leaf level.

A leaf level is the basic template, blueprint, of a vein or leaflet. It is defined as a set of parameters. It can have a blade and/or vein can define a leaflet or vein. A leaf level may also have **sub leaf levels**. These levels are placed on this level according to

placement parameters defined in Section 3.4. Sub levels are useful to create leaflets or venation patterns. How the leaf is affected by the number of sub leaves can be observed from Figure 3.1. In this figure “a” shows a simple leaf having only one leaf level, it has both blade and vein. “b” shows another simple leaf which has lateral venation pattern. Leaf “c” and “d” is a compound leaf. The leaflets of leaf “d” is an example of lobed leaves. All the leaves given in Figure 3.1 have only one parent and one sub level. However, by the definition of our system, this hierarchy does not necessarily mean that each leaf level could have only one sub leaf; they can have any number of sub levels. Moreover, a leaf level can be parented by more than one level. Our system does not allow recursions. Apart from other visual or arrangement parameters, every leaf level has also a **name** to identify itself.

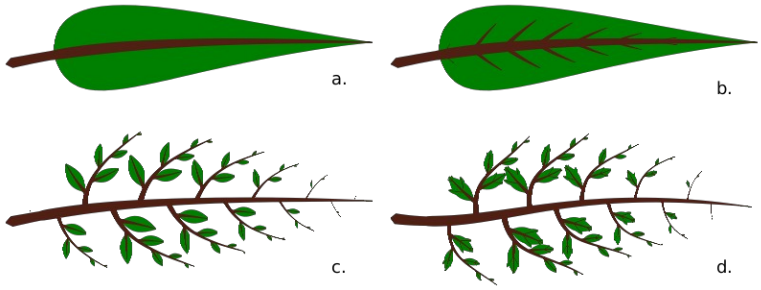


Figure 3.1: *Difference in number of leaf levels*

3.2. Shape parameters

Shape parameters specify overall shape of the leaf, these parameters are curvature, length, blade and/or vein shape and the change of curvature over the parent leaf level. Margin parameters, which also affect the final shape of the leaf by adding details, is discussed in Section 3.3.

In nature, size and length of leaflets or veins within a leaf varies. Often this variance is systematic and affects both length and size together. For instance, consider a compound leaf. Generally leaflets at the end of the leaf are smaller than the ones at the middle. To simulate this behavior, we have implemented **strength** property for each sub leaf. It is a property of a sub leaf, not a parameter. Therefore, we can define strength as a property of a sub leaf which affects other parameters by decreasing their sizes. These parameters are blade shape, leaflet or vein length, vein thickness and size of margins (if strength is chosen as size affecting parameter). Its value ranges from 0 (no strength), to 1 (full strength). For instance, consider two leaflets of the same type, one having strength of 0.5 and other 0.25. The first leaflet's length, blade and vein size will be twice the second one.

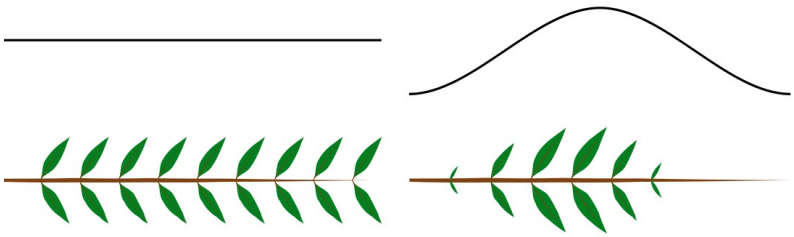


Figure 3.2: *Effect of different strength parameters*

Strength parameter directly affects the strength of the sub leaves that are connected to the leaf level that it is defined for. This parameter is specified with a curve. The x-axis of this parameter is the distance from the start of the leaf's main curve while y-axis specifies strength amount. The strength of a sub leaf, sprouting from a point, is the product of the strength property of this sub leaf and strength parameter's value at that point. Therefore, strength is a cumulative property of a sub leaf which is modified by strength parameters of the parent leaf levels. Figure 3.2 shows the effect of this parameter.

Although the size and the length of a leaflet or vein is generally proportional, there can be exceptions**. Length per Strength parameter** is a positional parameter which allows user to specify different lengths (y-axis) for different strengths (x-axis). Starting value of this parameter instructs length to be directly proportional to strength. We have implemented this parameter, so that user may choose how strength affects the length. Instead of proportional effect, it can be modified to set length of every sub leaf to the same value while strength still affects the thickness / size of blade or vein.

Main curve of a sub leaf is used to bend both blade and vein. Leaflets or veins that are under a sub leaf sprout from the main curve therefore, it also affects placement of those sub leaves.

Curvature parameter is the base curve that is used to calculate the main curve. It is modified by properties of a sub leaf to form the main curve. These properties are strength, length and straighten. It is also rotated and moved to adjust alignment on the parent leaf. Effect of this parameter is extensive since changing this parameter bends the current leaf level and any leaf levels that are connected to it. This shape is not always used fully. If the strength of a sub leaf is less than full value, only a portion of this curve is used. The amount that will be used is specified by length/strength parameter value for the given sub leaf strength. Figure 3.3 shows the effect of this parameter.

Maximum length parameter specifies the length of the current leaf level at full strength. It is specified as the percentage of the canvas width. This parameter modifies the size of curvature instead of just stretching it.

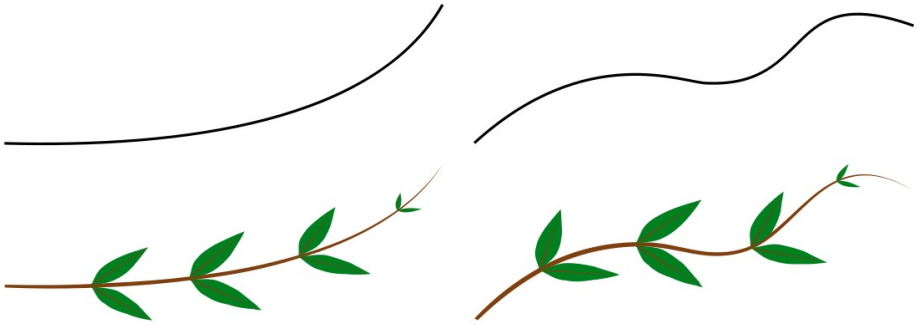


Figure 3.3: *Different curvature parameters and their results*

Many compound leaves have changing curvatures. Especially, the leaflets close to the tip is less curved than others. To address this behavior we have implemented **straighten parameter** which affects the shape of the curvature by morphing the given curve to a straight line. The value (y-axis) of the parameter at the given sprout point (x-axis is mapped to parent leaf's main curve) determines the amount of the effect. This parameter is useful to change curvature of sub leaves along the parent leaf. Figure 3.4 shows its application.

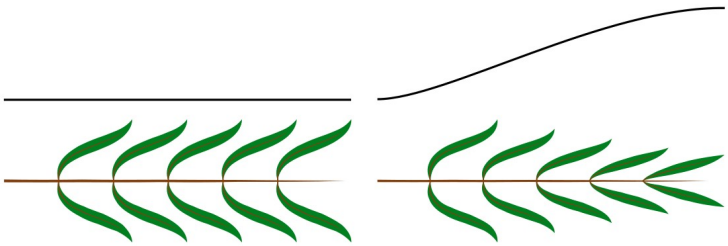


Figure 3.4: *Effect of straighten parameter*

Blade shape parameter specifies the shape (curve) of the leaf blade, and it is bent over main curve to produce final leaf blade for the current sub leaf. This shape is not the final shape of leaf blade, there can be multiple sub leaves that affect the blade shape of the leaf. Their combined shapes produces lobed or compound leaf

shapes. Moreover, margin parameters can affect the actual blade of the leaf. In Figure 3.10, a lobed leaf is shown which has multiple sub leaves that affect the final leaf blade. Figure 3.5 compares different blade shapes. The users are free to show or hide blade of a leaf level by changing **show blade** parameter.

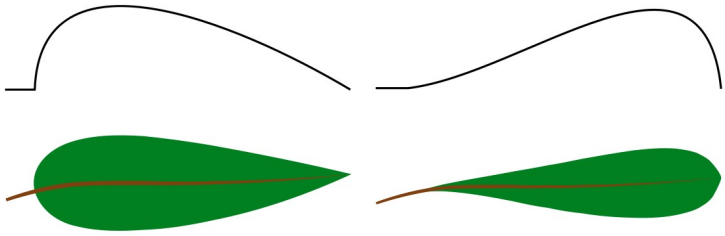


Figure 3.5: *Different blade shapes*

Vein shape parameter modifies the outline shape / thickness of vein. This parameter is a curve that is bent by main curve to become final vein shape for the sub leaf that it belongs. It is also important to note that leaflets always sprout from the main curve, not from the edge of the vein. The Vein of a leaf level can be hidden by setting **show vein** parameter to false. Figure 3.6 shows different vein shapes.

Vein length parameter changes the coverage percentage of the vein to the main curve. It also affects the placement of the sub leaves as they should sprout from the vein.

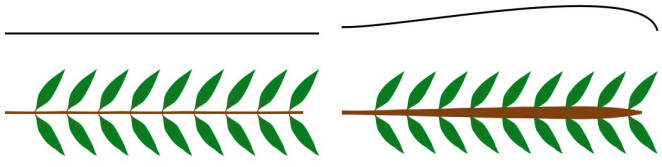


Figure 3.6: *Different vein shapes*

3.3. Margin Parameters

Margin is an important and distinguishing feature of a leaf. Figure 2.6 shows different margin properties of different plants. Leaf margins are repeated along the leaf blade, therefore we have implemented a system where user specifies the repeated shape and how it should be repeated. Margin parameters can be enabled or disabled. When it is disabled, the "*entire*" margin shape is used.

Margin shape parameter specifies the repeated shape (curve). This curve specifies only one part of the entire margin. This shape is resized according to margin size relation parameter. **Margin repeat** parameter is used to specify the times that the margin shape is repeated. The number of repeats can be relative to the sub leaf strength reducing the number of repeats.

Size of the margin shape can be changed along the leaf blade. For this purpose we have introduced **Margin size relation** parameter which specifies the property of the current leaf level that should be used to scale margin shape. This parameter should act after the repeating process. There are four options. In the option "*none"* the margin size is not modified. *Blade size* option scales the margin to the size of the current sub leaf's blade at the specified point. *Blade strength* option resizes margin shape according to the leaf level strength (the value of the parameter at that point not the cumulative strength property) at that point. Final option, *margin strength*, allows user to specify the margin size with another parameter. These options are represented to allow user to select and existing parameter which is already defined, instead of defining a new one. **Leaf strength relation** allows how much strength of the current sub leaf affects the size of margin. This parameter is a percentage value.

3.4. Placement Parameters

These parameters control how a leaf level is placed on the parent level. These parameters help to control the shape of compound and lobed leaves as well as leaf venation patterns.

Distribution method determines how the sub leaves (or veins) are organized on the parent leaf. This affects placement of leaflets in a compound leaf, lobes in a lobed leaf and venation pattern. There are six distribution methods.

The first distribution method is *free* distribution. In ,this method, leaflets or veins sprout from opposite side (top/bottom) of the parent leaf is not aligned. This method is exactly the same as opposite if there is no randomization. In *opposite* method, opposing sub leaves are placed at the same point. This method is used to create pinnately distributed leaflets or venation patterns. *Alternate* method places sub leaves alternately to opposing sides. This method is used for alternate arrangements. In *circular* method, sub leaves are placed in different angles sprouting from the same point. This method can be used to obtain rosette, palmate and digitate shaped leaves; palmate and rotate venation patterns. Angle parameter is ineffectual in this method since the distribution method is responsible from angle calculation. In *top* and *bottom* distribution method, sub leaves are placed to one side of the parent leaf. Figure 3.7 compares different types of distribution methods.

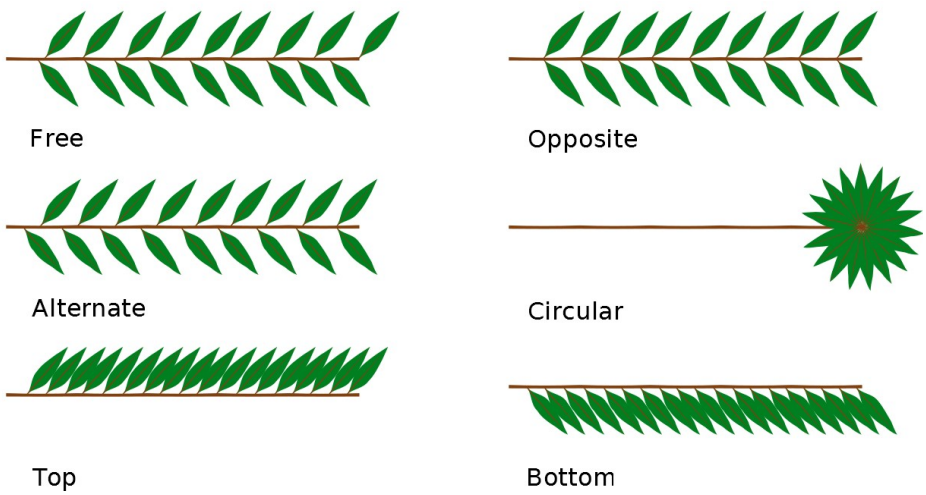


Figure 3.7: *Different distribution methods*

In distribution methods other than circular, sub leaves are placed along the parent sub leaf's main curve. The distance between sprout points is controlled by **spacing parameter**. Spacing parameter is defined as a curve where x-axis denotes the distance from the start of the parent leaf's main curve while y is the distance that should exist between leaflets or veins. This parameter also affects circular distribution method by determining the angle between sub leaves. Spacing can relative to the strength of the sub leaf. If it is relative, the number of sub leaves placed depends on the strength; otherwise number of sub leaves that are placed on the same leaf level is always same, regardless of its length or size. In Figure 3.8 effect of spacing is shown.

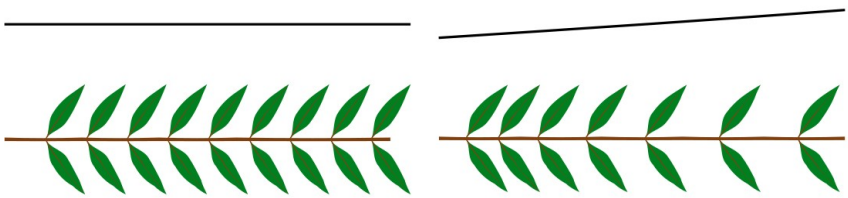


Figure 3.8: *Effect of spacing parameter*

Angle parameter controls the angle between sub leaf and its parent; because it is a curve, different angled sub leaves can exist on same parent. Since the shape (main curve) of the sub leaf is not a straight line, the actual angle between main curves of sub leaf and parent can be different than the determined value. Figure 3.9 shows the effect of angle parameter.

Leaf levels can also be specified to have **tips**. The tip is a leaflet that sprouts from the end of the current sub leaf. All the positional parameter values are taken from the last point of the curves.

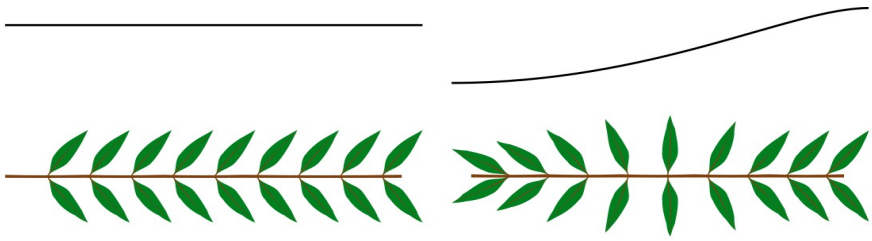


Figure 3.9: *The effect of changing angle parameter*

3.5. Randomization

Most plants in nature are disordered and their features are imperfect. This behavior can also be observed in leaves. If we take two leaves from the same plant, we can easily notice that they are different, even if they are at the same size. To simulate this property, we have introduced randomization to the system. In our system a sub leaf has properties (parameter values specific to that sub leaf) that are defined as curves or values. Therefore, there are two types of randomization. First one affects the shapes (curves) and called **Bézier randomization**. This type of randomization affects curvature, blade, vein, and margin shape. Since these curves

are defined with Bézier paths, they contain end points and control points. These points are defined as a rectangular area rather than a single point so that the exact control point can be randomly selected within that rectangular area. This method allows easier control of the randomness. Figure 3.10 compares two leaves that are randomly generated from the same set of parameters.

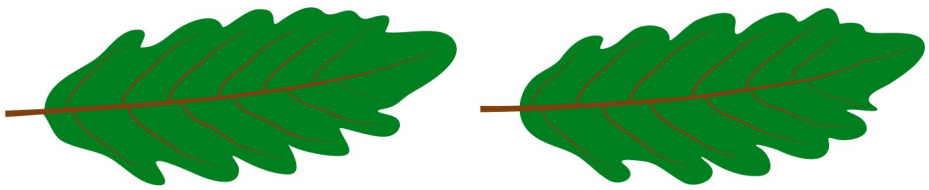


Figure 3.10: *Different leaves created using same parameters*

Value randomization method is used for angle, spacing, length, margin repeat, and strength values. In this method, parameters are not directly affected. Instead, the calculated values for a specific leaflet/vein are altered. This randomization method requires four different parameters and allows two different amount of randomization to be specified. First parameter specifies the chance of using first randomization method. Second and the third parameters are the range of first type and last parameter specifies the amount of change that can occur for the second type. This method allows the finer control of randomness. For instance, it is possible to set range of the second type to zero, to have a regular leaf with few disoriented leaflets. Figure 3.11 shows the application of this method.

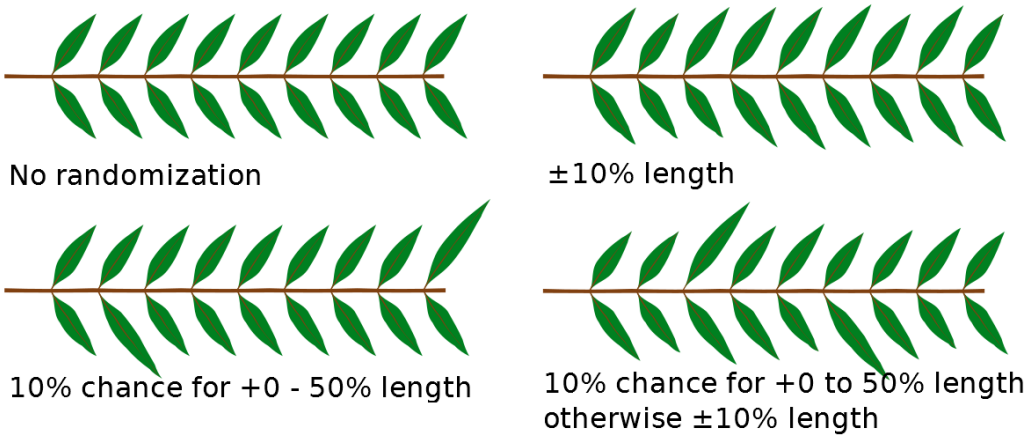


Figure 3.11: *Value randomization*

Chapter 4 – Bézier Class and Operations

4.1. Introduction

In this study, leaves are formed and defined using Bézier curves. Moreover, every step that are performed to generate the leaf modifies Bézier paths to perform its functionality. Therefore, this chapter defines Bézier curves and its implementation in the project. In section 4.2, general information regarding Bézier curves is given. Section 4.3 defines how they are used within this project as well as the terminology used and operations that are performed on them.

4.2. Bézier Curves

Bézier curves were originally introduced by Paul de Casteljau in 1959. However, they became a famous shape only when Pierre Bézier, French engineer at Renault, used them to design automobiles in the 1970's. Bézier curves are now widely used in many fields such as industrial and computer-aided design, vector-based drawing, font design and 3D modeling. [18]

4.2.1. Bézier Curves in Vector Graphics

In vector graphics, curves are a common shape. In most vector systems, curves are defined with Bézier curves, where they are mostly cubic Bézier curves. However, some applications use quadric Bézier curves for simplicity. In the following list some of the systems that use Bézier splines to represent curve primitives are displayed.

- TrueType fonts (quadric)

- Type 1 fonts (cubic)
- SVG Format (cubic)
- Inkscape, Gimp, Corel Draw/Photopaint, Photoshop (cubic)

4.2.2. Cubic Bézier Curve

This type is the most commonly used Bézier curve. This is a spline of third order and defined by four points: two endpoints (nodes, anchor points) (P_1, P_4) and two control points (P_2, P_3). The control points do not lie on the curve itself but define its shape. The curve, shown in Figure 4.1, starts at P_1 goes towards P_2 and arrives at P_4 coming from the direction of P_3 to create a smooth curve whose endpoints will be P_1 and P_4. The curve should have the additional property that the slope of the tangent line leaving P_1 should be the same as the slope of the line connecting points P_1 and P_2. P_2 fact, the reason point P_2 is called a control point is that the position of P_2 relative to P_1 determines the slope of the curve as it leaves point P_1 and starts bending towards point P_4. In general, the curve need not go through or even be near either P_2 or P_3. [19]

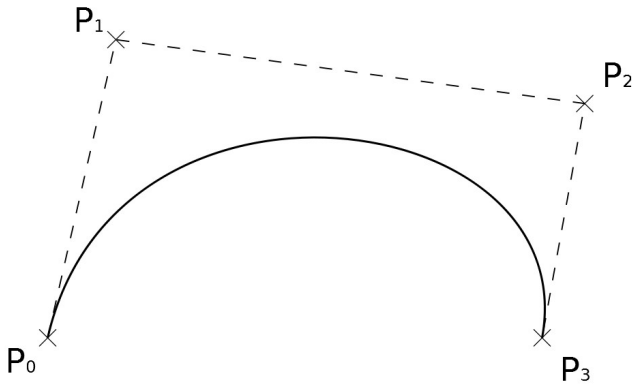


Figure 4.1: *A cubic Bézier curve*

Equation of the cubic Bézier curve is defined in Illustration 4.1.

$$B(t) = (1-t)^3 \cdot P_1 + 3 \cdot t \cdot (1-t)^2 \cdot P_2 + 3 \cdot t^2 \cdot (1-t) \cdot P_3 + t^3 \cdot P_4$$

Illustration 4.1: *Bézier Curve formula*

Bézier equations are parametric equations in variable t, and are symmetrical with respect to x and y. The parameter t, varying in interval $[0, 1]$, cuts the segment $P_1 - P_4$ into intervals, according to the wanted accuracy. When $t = 0$, the result is $B(0) = P_1$. For $t = 1$, the result is $B(1) = P_4$. The Bézier curve is tangent to the segment of line $P_1 - P_2$ at the start and $P_3 - P_4$ at the end. The curve remains within the convex hull of the control points.

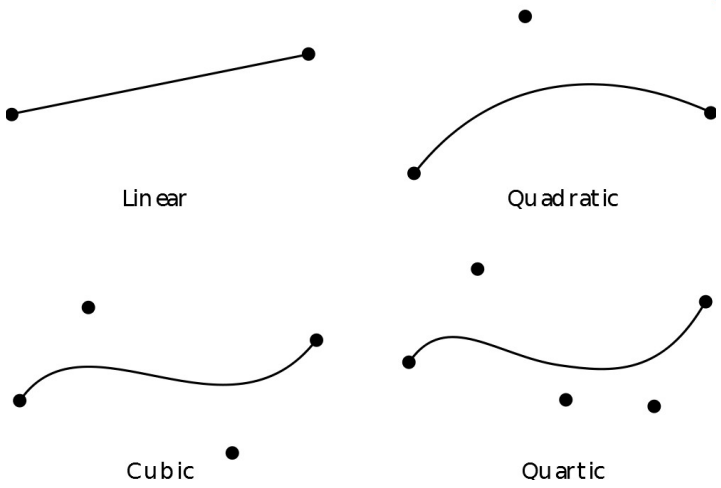


Figure 4.2: *Four different degree Bézier curves*

4.2.3. Bézier Curves of Higher Degrees

Bézier curves of any degree can be defined. Figure 4.2 shows sample curves of one through four. A degree n Bézier curve has $n + 1$ control points (P_0, P_1, ..., P_n) and its formula is defined in Illustration 4.2. [20]

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} \cdot (1-t)^{n-i} \cdot t^i \cdot P_i$$

Illustration 4.2: *Bézier Curve formula for degree n*

4.3. Application

Since Bézier curves are used commonly in this project, it is clear that we should design a system to handle Bézier curves and operations from a central system. This system will allow us to progress faster and increases the readability and maintainablity of the source code.

To represent and modify Bézier curves, we have implemented a C++ class. This class is responsible for all Bézier operations including transformations, save, load and drawing. To increase usefulness, we have implemented a system where a Bézier path is defined as a set of curves which are connected to each other. We have created three classes to handle specific data and tasks. *`Point`* class stores information of a single point. This can be an endpoint (on the curve) or a control point. *`Segment`* class stores information of a single Bézier curve. Since last point of a segment is the start point of the next one, end points are stored in *`Bezier`* class and only refered within a segment. This allows a changing point that is common to two segments to affect both segments. *`Bezier`* class is the Bézier path which contains multiple segments. Moreover, a Bézier path can have randomization information that are used while creating a leaf. This information is stored in *`Point`* class.

4.3.1. Terminology

Before starting to define the operations that are performed by a Bézier path, we will explain the terminology that is used in our system.

- **Bézier Curve**: always means cubic Bézier spline

- **Segment**: is an individual Bézier curve. However, in the actual system every segment is connected therefore the start point of a segment is the end point of its preceeding matrix
- **Bézier path**: is a series of segments, can be further defined as closed or open. A closed path defines an area and can be filled
- **t:** is the period of the Bézier spline which is defined as $P = B(t)$
- **Endpoint, node:** start and end point of a Bézier curve, or in other words control points of the curve that lie on the curve
- **Control points**: are the points between two end points of a segment which do not lie on the curve itself, just affect its shape
- **Place**: is a variable that is defined as the distance from the beginning of the path. Its important to note that a path can contain more than one segments therefore place is does not have the same value with t
- **Value curve**: is a Bézier path which has only one Y-axis value for each X-axis value (one-to-one function). It can be tought as a simple horizontal curve that does not wraps around itself

4.3.2. Bezier Class Properties

As it has been discussed before, a leaf is a set of Bézier paths which are formed by manipulating other paths. These manipulations are handled by the `Bezier` class. Following operations can be performed on a Bézier path: translate, scale, rotate, mirror, combine, finding value for a given X on a value curve, finding segment t (period) at a given place, offset, duplicate, randomize, break (adding a new segment), determining slope/angle at a given place or period, reducing Y difference, save XML, load XML and draw. All these functions are performed in vector space and are not affected by resolution or aliasing. [21]

4.3.3. Translate, Scale, Rotate and Mirror

Translate, rotate and mirror are basic transformations in computer graphics. To apply these transformations, every point on the Bézier path is transformed. Translate function moves control points by the specified X and Y distance. Scale function resizes the given curve by the given ratio for X and Y axis. This transformation uses `(0,0)` point as origin. It also modifies randomization data in the same manner. Rotate function rotates control points by the given angle and uses the specified origin. To mirror a curve, there are two mirror functions, horizonal and verticle. Mirror functions determine the effective width or height of the curve and mirror it using center point as origin. Except scaling, these functions does not modify randomization data.

4.3.4. Combine

This function combines two Bézier paths together. This operation basically adds segments of the given path to the current one. This combination can happen in two

ways. In the first one, method starts adding segments from the first segment of the second path. Second method is mirrored combination; in this method, segments of the second path are added in reverse order. This method allows us to create a closed path. When adding the first segment to the current path, first point (or last if mirrored) of the second path and the last point of the first path is checked. If they are at the same location system proceeds to adding segments. However, if they are not overlapping, a new segment is created between these two points; this segment is a straigth line. The resultant path preserves randomization data.

This function is used to combine margin shapes together without mirror option. It is used with mirror option to create closed paths of blade and vein shapes.

4.3.5. Finding Y Value at Given X

This function returns value for the given X-axis value. This is used to determine parameter value at a given point. For this function to work properly the Bézier path in question should be a value curve. Otherwise, the return value of this function is not determinative, because there will be more than one value for a given X. Although the task of finding Y value for a given X seems trivial, it is not the case for splines, because there is no function defined as $Y = f(x)$. It is defined as $P(x, y) = B(t)$. Therefore, we should approximate the value of t where P_x value gets close to the requested X-axis value. We use enhanced binary search method to achieve this quickly. This method estimates the correct t value from the rate of x movement over t value change. According to our calculations, after the 5^{th} step, error rate is around E^{-5} which is quite satisfactory.

4.3.6. Finding Segment and Period at Given Place

These functions cannot be separated because a period is valid only for one segment. Finding the segment at a given place is trivial since the starting place of every segment is buffered. This function only checks which segment that place is in and returns that segment.

The second function calculates the period by summing the distances between interpolated points of the Bézier curve. This system uses adaptive interpolation technique where increment in `t` value is set to a value where the distance between interpolated points will become 0.25 – 0.5 pixels. This function could further be optimized by Bézier subdivision. However, this requires further study of speed and error rate comparison of two methods.

4.3.7. Determine Slope/Angle

These functions calculate the slope at the given period or place on the path and either it returns the slope or the angle depending on the request. These functions use first derivative of Bézier curve function to determine the slope at the given period. If a place on the path is given instead of period, period and the segment is found as the first step. After this step, slope calculation function returns `y/x` value without further calculation. Angle function uses arc-tangent to calculate the angle and returns the angle. The following is the first derivative of a Bézier curve. Derivation is made over the variable `t`. Illustration 4.3 shows the first derivative of Bézier Curves. [22]

$$B'(t) = -3\cdot(1-t)^2\cdot P_0 + 3\cdot(1-4\cdot t+3\cdot t^2)\cdot P_1 + 3\cdot(2\cdot t-3\cdot t^2)\cdot P_2 + 3\cdot t^2\cdot P_3$$

Illustration 4.3: *First derivative of Bézier Curve formula*

4.3.8. Offset

This method is the key operation for building a leaf. It offsets a curve (**base curve**) using the values of the second curve (**offset parameter**). This operation also allows a static and curve based multiplier to be specified. This multiplier and y axis of the offset curve is used to determine offset amount. Additionally, it is possible to specify amount of base curve to be used. Lastly, this function can create mirrored output. This operation can also be defined as envelop function. It bends the x-axis of offset parameter curve then returns the result.

Leaf blade and vein shape are formed from blade shape and vein shape parameters. They are offset and wrapped around the main curve. Margin shape is also applied with the same method, but using modified blade shape as base curve.

Operation starts with calculating transformation value which calculates curve distance from x-axis of the offset parameter. This value will be used to map x-axis of the offset parameter to the base curve. While calculating this value, move rate (amount of the base curve that will be used) is also used. After these calculations offset parameter curve is divided so that it will include segments on the base curve. This allows us to only consider segments of the offset parameter.

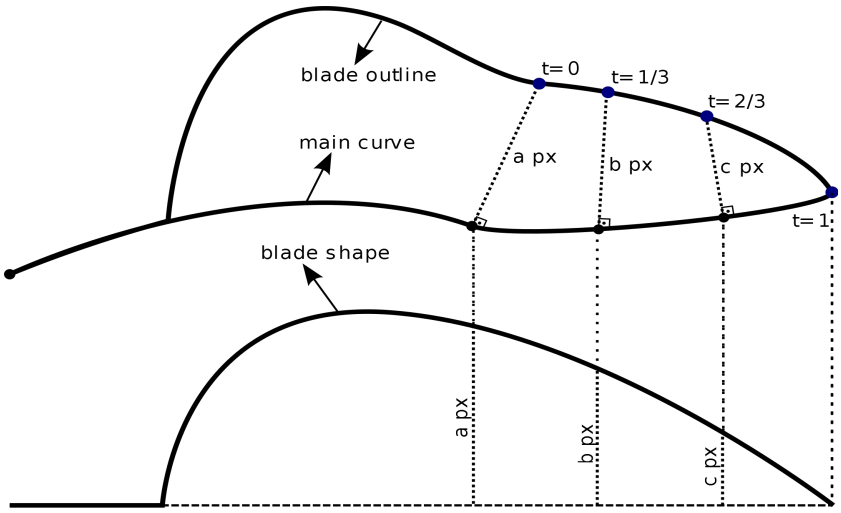


Figure 4.3: *Bézier offset function applied to leaf blade*

For every segment on offset parameter, algorithm creates a temporary segment. To calculate points of temporary segment, start and end points and two points in the middle of the offset segment is taken. Y-axis values of these curves are unmodified offset amount. Then the x-axis value of these points are calculated. With x-axis value destination points that are on the base curve is obtained, transformation value is used for this purpose. After this calculation, points on the base curve are determined. Using the slope of the base curve at the found points, angles which are perpendicular to the base curve at those points are calculated (if mirror option is chosen -90° is used instead of 90°). Then these points are moved by the modified offset amount rotated by the calculated angle. Offset amount is modified by multiplying with static multiplier and curve multiplier at the x-axis value of offset curve points. All these four points are on the new segment that should be formed. Using these points, we are able to estimate control points of this temporary segment. Illustration 4.4 is used to determine control points.

$$d_1 = P_1 - P_0, \quad d_2 = P_2 - P_1, \quad d_3 = P_3 - P_2$$

$$t_1 = \frac{d_1}{(d_1 + d_2 + d_3)},$$

$$t_2 = \frac{(d_1 + d_2)}{(d_1 + d_2 + d_3)}$$

$$C_1 = \frac{3t\cdot(t_2-1)\cdot t_2^2\cdot P_0 - (t_2-1)\cdot t_2^2\cdot(P_0-P_1) + t_1^3\cdot(P_0-3t_2\cdot P_0+2t_2\cdot P_0-P_2+t_2^2\cdot P_3) - t_1^2\cdot(P_0-3t_2\cdot P_0+2t_2^3\cdot P_0-P_2+t_2^3}{3\cdot(t_1-1)\cdot t_1\cdot(t_1-t_2)\cdot(t_2-1)*t_2}$$

$$C_2 = \frac{-(t_2-1)^2\cdot t_2\cdot(P_0-P_1) + t_1^3\cdot((t_2-1)^2\cdot P_0-P_2-t_2\cdot P_3+2t2^2\cdot P_3) + t_1\cdot((t2-1)^2\cdot(1+2t_2)\cdot P_0-P_2+t_2^3\cdot P_3) - t1^2\cdot((2-3t_2+t_2^3)\cdot P_0-2P_2}{3\cdot(t_1-1)\cdot t_1\cdot(t_1-t_2)\cdot(t_2-1)*t_2}$$

Illustration 4.4: *Determining control points*

After all temporary segments are obtained, the base curve is modified to have the obtained segments. Figure 4.3 describes this operation while the following is the formal definition of the discussed algorithm.

- For every segment in our offset curve,
- Let $B_{offset}(t)$ function be Beziér function of the current segment, $B'(t)$ function to be the target function
- $P_0 = B(0),\ P_1 = B(1/4),\ P_2 = B(3/4),\ P_3 = B(1)$
- X-axis values of P_0, P_1, P_2, P_3 is calculated
- Using x-axis values, positions, points (Q_0, Q_1, Q_2, Q_3) and angles ($\alpha_0, \alpha_1, \alpha_2, \alpha_3$) of these points on the base curve is calculated
- These points (Q_0, Q_1, Q_2, Q_3) are moved by offset amount (y-axis value of P_0, P_1, P_2, P_3) perpendicular to the curve resulting points (Q_0', Q_1', Q_2', Q_3')
- Let $B'(t) = (1-t)^3\cdot P'_0 + (1-t)^2\cdot t\cdot P'_1 + (1-t)\cdot t^2\cdot P'_2 + t^3\cdot P'_3$ then
- $P'_0 = Q_0'$
- control points are calculated as follows

$$P'_1 = \frac{3t\cdot(t_2-1)\cdot t_2^2\cdot P_0 - (t_2-1)\cdot t_2^2\cdot(P_0-P_1) + t_1^3\cdot(P_0-3t_2\cdot P_0+2t_2\cdot P_0-P_2+t_2^2\cdot P_3) - t_1^2\cdot(P_0-3t_2\cdot P_0+2t_2^3\cdot P_0-P_2+t_2^3\cdot P_3)}{3\cdot(t_1-1)\cdot t_1\cdot(t_1-t_2)\cdot(t_2-1)*t_2}$$

$$P'_2 = \frac{-(t_2-1)^2\cdot t_2\cdot(P_0-P_1) + t_1^3\cdot((t_2-1)^2\cdot P_0-P_2-t_2\cdot P_3+2t2^2\cdot P_3) + t_1\cdot((t2-1)^2\cdot(1+2t_2)\cdot P_0-P_2+t_2^3\cdot P_3) - t1^2\cdot((2-3t_2+t_2^3)\cdot P_0-2P_2+2t_2^3\cdot P_3)}{3\cdot(t_1-1)\cdot t_1\cdot(t_1-t_2)\cdot(t_2-1)*t_2}$$

- $P'_3 = Q_3'$

4.3.9. Duplicate

This operation creates a complete copy of this Bézier path. A curve and its duplicate are independent of each other. Moreover, destruction of the newly created `bezier` object is in responsibility of the callee.

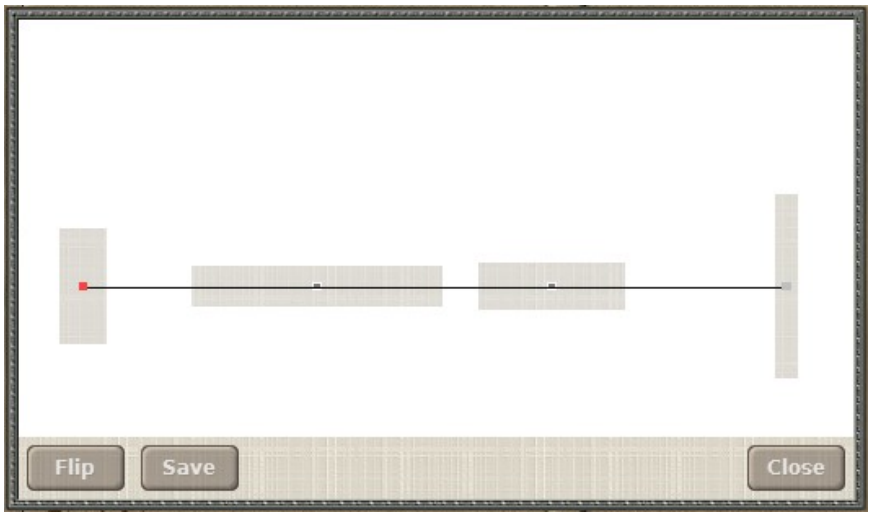


Figure 4.4: *Defining ranges for randomization*

4.3.10. Randomize

This function operates on all points and control points of the Bézier path and randomizes them according to their randomization range. Randomization information is defined as range independent of the point itself, therefore, using this function second time creates a more randomized path. Figure 4.4 shows defined ranges for randomization. The gray rectangles define the boundaries for the given point.

4.3.11. Segment Division

This operation divides a Bézier segment into two segments. It can either performed on the curve which does not modify the shape of the path or an outside point can be used which became a part of the curve. First function uses Bézier curves ability to be divided from any point on the curve. This property is unique to Bézier

splines. To find new segments, the method of finding point at the given period is applied without using the formula. This involves in finding tangent lines and their corresponding points. First and last first-tier points (determined on initially calculated tangent lines) are used as the first and last control points of the first and second segment. Second tier points (obtained by intersecting a secondary tangent line with previous ones) are used as the last control point of the first segment and first control point of the second segment. This method does not use approximation methods. Therefore, its efficient and precise.

Second method to break a curve into two segments is inserting a new point. This function sets the second point of the first segment and the first point of the second segment to the given point. Then it assigns missing control points to -10 and +10 of the inserted point. This function also checks existing control points, if they go beyond the bounding box of the newly created segments, they are moved inside it. This prevents formation of a highly deformed path.

4.3.12. Reducing Y Difference

This function is used to morph a given curve to a straight line. This function takes amount of reduction as parameter. When this parameter is 1, the curve is reduced to a line. This task is done by subtracting $amount \times y$ from the y-axis of every point and control point in the path. This function affects this path, does not create a new one.

4.3.13. Save XML

This method creates XML string to be saved into leaf definition file. This function takes its save name as parameter. Firstly, the starting point of the path is saved (*firstpoint* tag). Point objects put *x*, *y*, *randx* and *randy* (randomization range) into separate attributes. After saving first point, all segments are saved one by one. *Segment* class saves its first and second control points as well as second end point. First end point is not saved because it is the last point of the previous segment (for the first segment it is the first point which is saved by *Bezier* class).

4.3.14. Load XML

This method uses XML Parser libraries to read information that is stored within an XML structure. Loading method is similar to saving, first point is read, new segments are created and asked to load themselves using the data supplied by the XML file. To make the system more adaptable, loading operation does not make any checks. It reads the information it can find and uses default values for missing data. Moreover, any data which is not supposed to be present is ignored.

4.3.15. Draw

This function draws the given path on a given canvas. Canvas object is defined within Gorgon Graphics Library. It is an image object and can normally be drawn on a layer. But also, it allows its data sources to be modified. Since Bézier curve is a spline we have to determine points and draw lines between them. Normally these points are separated by a fixed amount of period. This results variable distances between points. When the point distance increases, curve looses its smoothness. It can be fixed by increasing number of points, however, this slows down the drawing

process. These problems can be solved by adaptive subdivision method. In this method, a Bézier curve is divided into smaller curves recursively. Given a Bézier curve, this algorithm checks whether its close to a straight line. This check is done by calculating two points on the curve and check their distance to the line passing between first and last point of the curve. In our method, we have used the angle difference to calculate error rate, instead of calculating point to line distance If this distance is small enough a line is drawn. However, if its not, this curve is divided into two smaller curves. To speed up the process, our algorithm does not actually break the curve, instead, it calculates period range for smaller curves. And these newly created curves are checked separately.

The method used in this study is first proposed by [23]. Using this method, a curve of 600 pixel length is drawn using 120 lines. Without using adaptive subdivision, we should use 300 lines to achieve a smooth curve. This algorithm is shown in Illustration 4.5.

1. For a given curve of P_0, P_1, P_2, P_3 and δ which is the threshold angle of an almost straight line
2. Function $B(t)=P_0\cdot t^3+P_1\cdot t^2\cdot(1-t)+P_2\cdot t\cdot(1-t)^2+P_3\cdot(1-t)^3$
3. For $t_s=0, t_e=1$ calculate $\text{divide}(t'_s, t'_e)$
4. where function divide(te, ts) defined as

   a. $p_0=B(t_s),\ p_1=B(\frac{t_e-t_s}{3}+t_s),\quad p_2=B(2\times\frac{t_e-t_s}{3}+t_s)\ p_3=B(t_e)$

   b. if $|\widehat{p_0,p_1}-\widehat{p_1,p_2}|+|\widehat{p_1,p_2}-\widehat{p_2,p_3}|>\delta\Rightarrow$

      for $t'_s=t_s,\ t'_e=\frac{t_e-t_s}{2}+t_s$ $\text{divide}(t'_s, t'_e)$

      for $t'_s=\frac{t_e-t_s}{2}+t_s,\ t'_e=t_e$ $\text{divide}(t'_s, t'_e)$

      else

      define the line $B(t_s),\ B(t_e)$

Illustration 4.5: *Bézier subdivision algorithm*

After these points are calculated, this point list is sent to `DrawLines` function. This function takes an array of points and draws an anti-aliased line with variable thickness. It is very important to note that this function takes an array of floating point numbers, not integer values. This method is used to prevent discontinuous edges between lines.

This function uses reverse rotation method to draw variable thickness line. It basically rotates and translates a rectangle of the given height and length to the place where the line should be drawn. This method creates perfectly anti-aliased, variable thickness lines.

Chapter 5 – Implementation

5.1. Introduction

We have built an application to validate our model and provide a useful tool for graphic designers. This application is written in C++ and uses object orientated programming methodologies. We have used Gorgon Widget engine to provide user interface. This system is chosen because it is extensible and based on C++ and object oriented methodologies.

In this chapter, implementation details of the application is explained. In section 5.2, the underlying engine of our application is defined. Widget engine system that is used within this project is introduced in section 5.3. Fourth section discusses how the leaf data and its representation is stored. In the fifth section, leaf building process is explained. The next section details the process of rendering, drawing, of a leaf. In section seven, user interface design is described with details. The last section describes how save/load and export features work.

5.2. Gorgon Game Engine (GGE)

Gorgon Game Engine is a system that is written in C++ to meet the requirements of a game or game related application. A game that is written using this system is shown in Figure 5.1. GGE has a dual-layer architecture that supports many functionality including graphics (2D, 3D hierarchically layered system including clipping, transformations, ordering, raw and prepared image support), sound (including fast 3D positional surround support), video and multimedia, keyboard and

mouse input (organized and managed within graphic layer system), mouse pointer system, interval events, animations (both image based and time based effects), graphic effects and resource support with graphics, sound and data. Dual layer system allows easy adaptation where back-end layer is responsible to communicate with low level API where a fixed front-end is used by programmers. Its primary goal is to be useful without sacrificing speed.

Gorgon Resource Files are used within this system. This binary file type is extensible and can preserve both backward and forward compatibility. Gorgon Resource Engine (GRE) is a sub module of game engine that allows loading these files. These files contain a directory system where resources are placed hierarchically. Moreover, any resource type can contain other resource types. Resources also have commands/actions and properties related with that resource. This file architecture allows partial load or easy export and import support because a file only contains some information (file type and signature) and the root directory. Currently there are six basic resources: text, image (uncompressed, PNG, JPEG (no alpha support) and LZMA compression), data array (can contain text, number, color, point, or rectangle), sound (uncompressed or lossless LZMA compression), bitmap fonts, and animations. Gorgon Widgets File is also this type of file that supports more and complex objects.

Gorgon Game Engine uses OpenGL to access graphics card. This system is hardware accelerated. It means any transformations, light and opacity calculations are made by graphics card rather than CPU. It also uses graphic card's memory to store images. This allows faster drawing of images. These abilities improve the speed

of the system while layer system increases its usefulness. GGE graphics system contains main layer initially. This layer can contain other layers, however, nothing can be drawn on it. There are different types of layers which can be grouped as graphical (2D and 3D), input and hierarchy layers. These layers can contain any layer as its sub layer and can be ordered within its parent. For instance, to draw an image with colorizing support one must use `Colorizable2DLayer`.

OpenAL (Open Audio Library) is a system to access sound resources. This library is supported by Creative Labs. This library is an open system and can be used with any sound card. GGE supports wave type audio system which can be used either as background music or positional audio. Sound engine has its own garbage collection system which removes wave controllers that are not needed anymore. Sound resource, which can be loaded by GRE, can directly be played with this system.

Both keyboard and mouse input system reads data from operating system calls; however, they are presented with two different methods. Keyboard input uses a bubbling event system. In this system keyboard input requesters register their event handlers. When an input is to be distributed, the last request takes precedence. If this requester does not use the key that is pressed, that key stroke is propagated to the next handler.

Mouse input system is tied to the layer architecture. A layer can be target of mouse input and produces events depending on it. These event handlers define a region where the mouse events originated from, is given to them. This system also passes the event to the layer below if it is not used.

Pointer system reads pointer information, which may contain many pointers with their types and hot-spot information, from the given resource directory and is responsible of displaying, moving, changing and restoring pointer. It has a multi-consumer method to display pointers. This method allows pointer to be changed by an object while another change request is made by another call. These requests can be canceled with any order. The pointer specified by the latest active request is shown. If there is no request, this sub system displays default pointer. This system also hides in game pointer and shows operating system pointer when the game looses its focus or pointer moves out of window.

Figure 5.1: *A game written with GGE*

There is a generic animation system within this system. This system can either work discretely (frame based) or continuously (time based). Animation resources that are loaded from resource files are managed by discrete animation controller. This allows frame based events, speed adjustments, pause, frame control, playing direction, and looping. On the other hand, continuous animation system is used by

effect animations. These effects contain layer mover/sizer, tinting, flipping animation, and counting numbers. The number of these effects can easily be increased since there is no need to re-implement animation control mechanism.

5.3. Gorgon Widgets

This is the widget library that is used within this project. It is built over Gorgon Game Engine. This library is also written in C++. Its prime objective is to provide customizable and easy to use user interface. In this widget system, every visual aspect of a widget is specified by widget file. A widget file is a type of Gorgon Resource File. Widget definitions within this file is read and used as blueprints to create widgets. If the organization of this file is made according to widget registry system, it can provide easy creation of them.

Every widget in this system is based on `IWidgetObject` class which unifies all widgets and handles common tasks such as move and resize. This system allows easy implementation of new widgets. A widget called Bézier Control is created over this system to be used within this project. This control uses Bézier Editor and ordinary button to accomplish its task. A button is used to open editor while displaying current path on it. Bézier Editor, which also uses other widgets to help displaying editor frame, action buttons, uses Interactive Bézier object. Interactive Bézier system uses pointer events to modify Bézier path. Curve editor also handles keyboard events such as arrow keys to help modifying the path.

5.4. Leaf Data and Representation

We have designed the system with object oriented methodology and we have used model view controller approach where the system can work without a user interface. The classes implemented for leaf generation, create instance of objects that are designed to represent the generated leaf. These classes are responsible from exporting and drawing generated leaf.

`LeafBlueprint` class defines a leaf. It is used to perform leaf generation as well as save and load functions. This class holds all leaf levels in a list to perform these functions. `StemLevel` class defines a leaf level. It contains all the parameters defined for a leaf. This class is also responsible from building sub leaves.

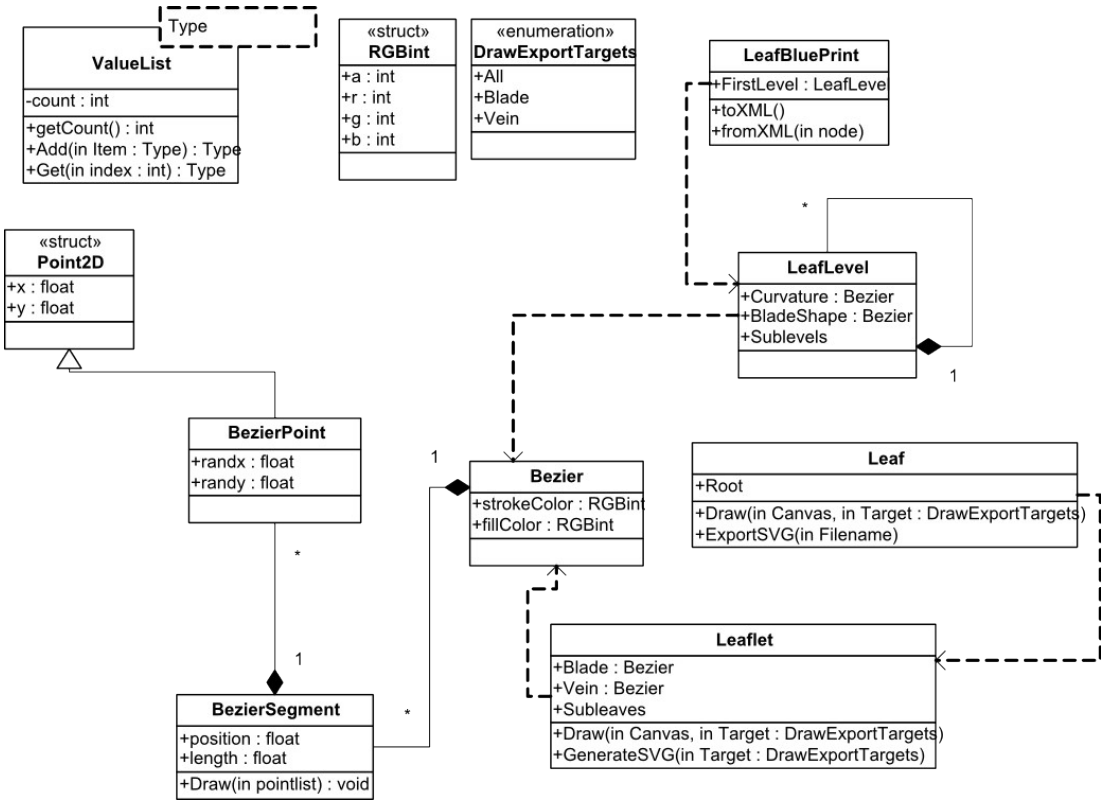


Illustration 5.1: *UML Diagram of the class architecture*

Leaf class is the root class of the generated leaf. Some of its important methods are export and draw functions. *Subleaf* class represent a leaflet or vein. It is generated by *Subleaf* class. It contains blade and vein paths as well as sub leaves that are connected to it. Moreover, an internally used method to draw the leaf level it represent on a given canvas. This method should only accessed from other sub leaves and the *Leaf* class.

5.5. Generating a Leaf

A leaf can be created by calling Generate function of *LeafBlueprint* class. This function creates an empty *Leaf* class and calls build leaf function of the first sub leaf which is the midrib.

5.5.1. Build Function

Build function of *Subleaf* is called to create shape and adjust sprout points and parameters of its sub levels. It calls build functions of sub levels and stores their resultant classes in a collection. This collection will be used later, when the leaf is requested to draw or export itself. *Build* function only generates one leaflet or vein. However, it calls *BuildSubleaves* function which calls Build functions of its sub levels to build its own sub leaves.

Build function requires strength of the current sub leaf its building as well as its location on the parent leaf, sprout point (coordinates), rotation and whether it should be mirrored. The values of these parameters are given by the parent leaf; however, midrib has no parent. Therefore, default values (1 for strength, 0 (start) for location, left-middle of the canvas for point, 0° for rotation and false for mirror) are assigned

for it. If the given strength value is too low to produce a valid output (less than 1/1000) this function does not produce a result and returns an empty sub leaf. Notice that this behavior is expected by the caller and no error is generated. After parameters are checked, strength value is randomized using value randomization method (see subsection 4.3.10).

Second operation of this function is to create main curve. Main curve is derived from curvature parameter, so the curvature specified for this leaf level is duplicated. Then this newly created Bézier path is scaled to fit the length specified with maximum length parameter. For this operation, its current length is calculated and resize (see subsection 4.3.3) function of *Bezier* class is called with the requested size modification. After resizing, main curve is randomized using Bézier randomization. Straighten operation takes place as the next step. This operation uses y difference reduction function of *Bezier* class (see subsection 4.3.12). If mirroring is requested, Bézier path is mirrored vertically (see subsection 4.3.3). Vertical mirror is used because curvature parameter is specified horizontally. After mirroring, main curve is rotated around its first point by the amount specified by its parent. As the next step of this operation, main curve is moved to the requested point. The first point of the curve is used as the base. Offset operation on Bézier curves are not exact. They produce an estimation. Therefore, to increase the quality, application splits the main curve to create additional five segments. As the last step, amount of main curve to be used is calculated using length/strength parameter with the current strength. Only this amount of main curve will be used. Therefore, if the strength is low, this sub leaf will be shorter. Randomization is also applied to this value.

The next operation is to create blade of this sub leaf. This operation is performed only show blade option is chosen for this level. For this task, main curve and Shape parameter is duplicated twice; once for upper side, other for lower. Duplicated shape parameters are randomized and wrapped around main curves. Wrapping operation is performed by Bézier offset function (see subsection 4.3.8). After this operation, two halves of the blade is formed. If margin parameters are specified, these halves are put into further modification. Otherwise they are directly combined to form blade shape for this sub leaf.

Margin parameters modify top and bottom side of the leaf independently except repeat count. As the first step, repeat count is calculated. The value of the parameter is taken and if it is specified to be relative to strength, it is multiplied with current strength value. Then repeat count is randomized according to the randomization range defined for this leaf level. Second step is to create repeated margin shape. In this step, basic margin shape is duplicated and they are combined together. This operation is repeated as much as the repeat value. This new Bézier path is duplicated for the bottom side of the blade. After that, these paths are randomized independently. As the next step, previously formed top and bottom blade shapes are offset with their margin curves. While offsetting, margin size relation parameter is used to determine factor of the operation. This factor is supplied to offset function as well as curves. After this step, top and bottom side blade shapes are combined and shape of the current sub leaf is formed.

Creating vein shape is quite similar to blade shape generation. Firstly, main curve and vein thickness parameters are duplicated. After this, randomization is

applied to new vein curves. In the next step, offset operation is performed on main curves using vein curves as offset parameters. Finally, modified curves are combined to form vein shape of this sub leaf. These operations take place only if the user chooses to show vein for this leaf level.

Both blade and vein paths are stored in the generated *Subleaf* class.

The last operation (excluding cleanup) while building a sub leaf is calling *BuildSubleaves* function. Main curve, sub leaf that is being generated, strength of the current sub leaf, sprout point, rotation angle and mirror values are given to this function.

5.5.2. BuildSubleaves

This function determines build parameters and calls build functions of sub leaves that will be under this sub leaf. It separately processes every leaf level connected to the current one. This function acts on distribution method of these sub leaves.

If distribution method is **free**, top and bottom side of the sub leaf is processed separately. Firstly, distance factor is calculated by dividing the length of the main curve to maximum height of the parameter. This factor is used to transform value of the spacing parameter to actual distance that should exist between sub leaves. From the start of the main curve until the end, every point is checked if there can be a sprout point at that location. For this purpose, the distance from the last sub leaf is calculated. This distance is modified by current strength if spacing is set to be relative to the strength of the sub leaf. By this way, if relation exists between strength

and distance, there will be less number of sub leaves when the strength is low. If the distance from the last sub leaf sprout point is greater than the spacing requirement calculated by transforming the value of spacing parameter at that point using distance factor, a sprout point is designated. Spacing randomization is taken into account while this comparison is made. Strength and angle for this sprout point is calculated using parameters of sub leaf level. Moreover, the angle of the main curve at that point is calculated. After this step, angle value is either added or subtracted from curve angle depending on mirror and whether working for top or bottom of the sub leaf. This is the same for determining whether to mirror the sprouting sub leaf. After these values are calculated, they are passed to `Build` function of the sub leaf level. When top side is finished, same procedure is applied to bottom side. Illustration 5.2 the formalized method for free distribution model. Since top and bottom sides are handled separately, randomness may cause sub leaves at top and bottom to be in different positions even in different numbers. Without randomization, this method produces exactly the same result as opposite distribution.

1 $distance_{factor} = \frac{Max_{ParameterValue}}{MainCurve_{length}}$

2 $parameter_{factor} = \frac{Max_{ParameterWidth}}{MainCurve_{length}}$

3 $distance = 0$

4 *For every point in main curve*

   1 $distance = distance + 1 \times distance_{factor}$

   2 $distance = distance \times strength$, if relative spacing is chosen

   3 Place a sprout point , if $distance > Spacing(point_{position} \times parameter_{factor})$

Illustration 5.2: *Formal definition of free distribution*

Opposite distribution is quite similar to free distribution. In this method, when a sprout point is found, two sub leaves are placed; one for top section and another for the bottom. This method uses one loop to handle both sections. Synchronized handling mechanism results top and bottom leaves to sprout from the same point and become opposite of each other.

Similar to previous distribution mechanism, **alternate distribution** also places sub leaves to top and bottom sides of the main curve. Placing sub leaves alternately to top and bottom is its prime difference from opposite distribution. This method places a sub leaf at half of the distance calculated from spacing parameter since it places only one sub leaf per sprout point.

Circular distribution method is quite different from other methods. Instead of distributing sub leaves along the main curve, it places every sub leaf at the end of the vein giving different sprout angles for each. This method creates fan like shape. Moreover, it does not use angle parameter at all. Firstly, factor to transform spacing value to angle is calculated. After this, every angle starting from -180° to 180° is checked. If the angle difference between current angle and previously generated sub leaf is more than angle distance calculated from spacing parameter, a new sub leaf is placed. All other parameters are calculated similarly; distance is replaced by angle. When the build function of the sub leaf level is called its given the tip of the vein as the sprout point, location is determined from the angle (-180° will become 0 while 180° will be 1), the angle is directly taken from loop. Mirror parameter is set when the angle is below 0°.

Top and **bottom** distribution methods only places sub leaves to one side. Similar to alternate distribution they work at half spacing.

5.6. Rendering Leaf

Our application supports internal rendering. Although, this rendered cannot provide very high quality results, it helps to see the leaf while adjusting parameter. Our renderer supports anti-aliasing and sub pixel accuracy; therefore, the result is not far from high quality vector rendering applications. In addition to filled rendering, our application displays blade and vein outlines, allowing user to see parts of the leaf clearly.

As it is discussed before, a generated leaf contains Bézier paths that are ready to draw. To draw outlines of these paths, Draw function of `Bezier` class is used. This function build a point list and uses line drawing algorithm to draw outline. However, rendering a leaf is more complicated than this. It requires closed Bézier paths which will be filled. This task is accomplished by building point list of Bézier paths, and passing this list to draw polygon function.

`DrawPolygon` function takes an array of points and color information to fill the given polygon. This function uses enhanced scan line algorithm to fill the given curves. This scan line algorithm uses anti-aliasing method similar to Wu's algorithm. Moreover, its speed is enhanced by sorting mechanism which allows straight forward looking of line crossing to improve efficiency. In this scan line operation, every row of the image falling within the bounding box of the polygon is taken. For every row, draw flag is reset and all pixels of that row is checked. If that pixel contains a line

boundary, draw flag is toggled. If draw flag is set for any pixel, requested color is given to that pixel. Line boundaries are treated slightly different. In case of a boundary, algorithm checks how much of that pixel falls within the polygon and paints the given color with that intensity. This allows anti-aliased, smooth edges.

5.7. User Interface

On of the most important aspect of a design application is its user interface. Generally, design applications offer many tools that is aimed to assist the user. These tools are often packed together, so the user can access them faster, forming complex interfaces. Therefore, it is a trade-off between use speed and simplicity.

In our application, we have used streamlined and simple interface to reduce the learning curve. Firstly we have a menu bar where generic commands exists. There are three panels, first one is leaf level control. In this panel leaf levels that exist in this leaf definition is listed. In addition, more leaf levels can be added or removed from here. Second panel contains parameters. We have used a tabbed interface to group parameters. Moreover, parameters are sorted by their importance, therefore, the parameters that the user will change most will be at the top. Last panel shows results. A sample screen showing user interface is displayed in Figure 5.2.

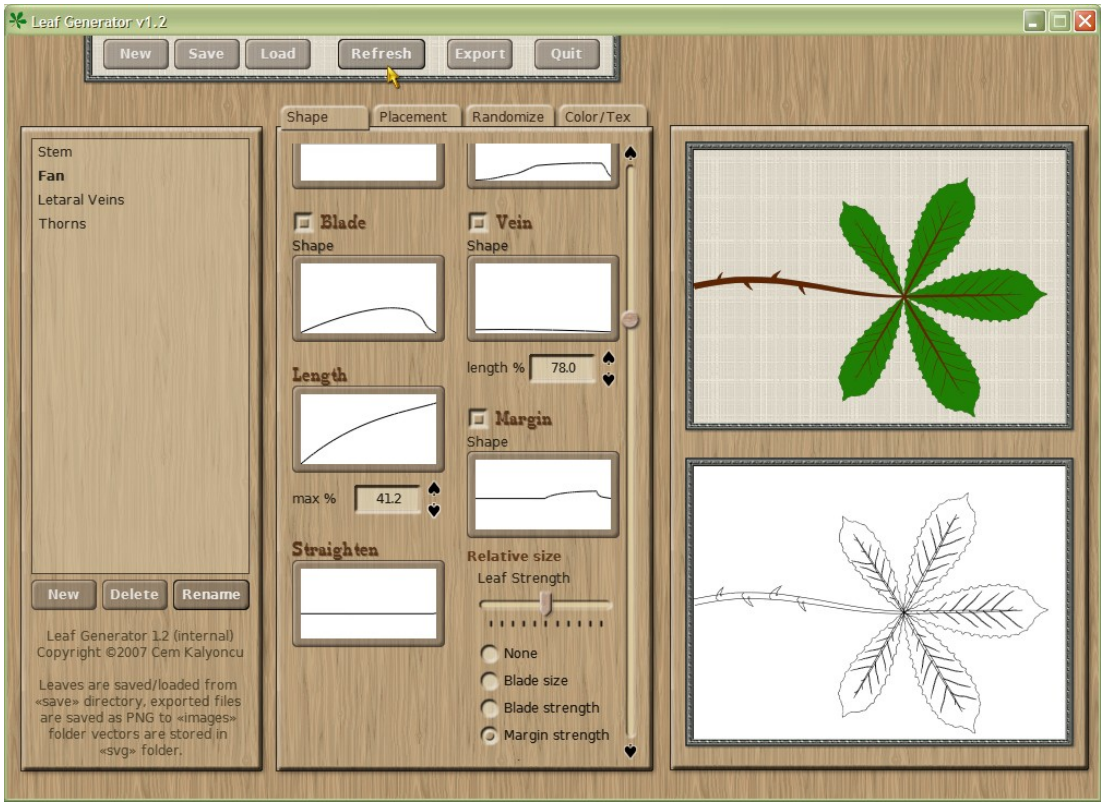


Figure 5.2: *Main interface and shape parameters*

In our application UI class is the sole accessor to the user interface. The rest of the application is unaware of the user interface. This allows us to create derivative application such as a console based batch processor.

5.7.1. Main Interface

Menu bar, leaf level list and results panels can be defined as main interface. Results panel is invisible when application is opened, it is shown when the user request to see the leaf. Moreover, file, message and rename dialogs are considered to be in this category. They are simple dialog frames that have widgets on them. To make the coding easier, we have implemented functions to show these dialogs whenever they are needed.

Menu bar button events are handled by *UI* class. However, only task of user interface system is to call a function to perform the requested task. Some tasks are outside the scope of leaf generation. Such operations are performed by *Application* class. *Application* class handles application termination, creating a new leaf (uses template save file), and loading a leaf. Its also responsible from keeping current leaf class.

5.7.2. Leaf Parameters

Leaf parameters are shown and allowed to be edited on the parameters panel. This panel is hidden when the application starts and shown when a leaf level is selected. When a leaf level is selected *SetSelected* function is called which adjusts all controls to show parameter values of the given sub leaf. Since some parameters are defined as curves, *Bezier* classes of these parameters are given to Bézier controls to be shown and edited. Bézier editors automatically modify curves without need of events. However, other controls fire change event to notify its value is changed. These events are handled by *UI* class and selected leaf level is modified accordingly.

5.7.3. Bézier Control and Editor

Solely for the purpose of editing parameters in this project, we have created a new widget, Bézier control. This widget is derived from button and uses its image property to show current Bézier shape. This is possible by using size attribute of *Bezier* class which modifies. When the button is clicked Bézier editor is shown which is displayed in Figure 5.3.

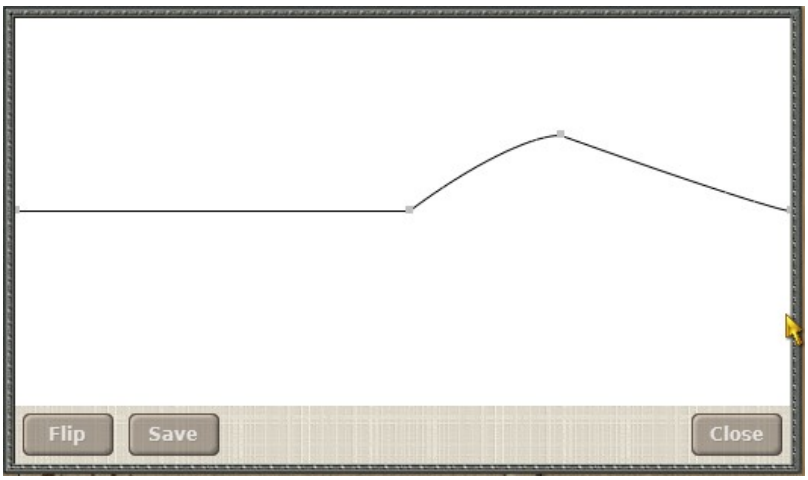


Figure 5.3: *Beziér Editor*

Bézier editor uses interactive Bézier system to allow user to edit the given path. Interactive Bézier class extends normal `Bezier` class and adds edit support using mouse events. Interactive Bézier class basically handles mouse events within a given canvas and modify Bézier path that it extends. Although it could be done, there is no keyboard editing support in this system. Interactive Bézier system changes mouse pointer when it is over an endpoint or control point. It also allows user to select Bézier point and move them. Moreover, entire curve can be scaled or moved. This system also supports restrictive movement for Bézier value parameters which should have exactly one y-value for each x-axis.

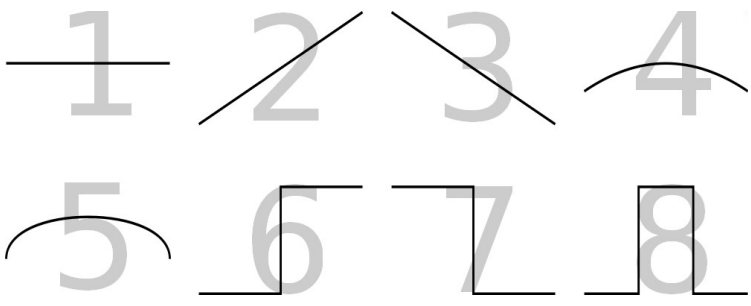


Figure 5.4: *Keyboard shortcuts for curve templates*

On top of interactive Bézier system, editor adds keyboard support, templates, save, and flip controls. Keyboard events are provided by frame widget. Keyboard event handler has two purposes, first one is to detect arrow keys. Arrow keys modify selected node by changing its position or positions of its control points. This effect is controlled by modifier keys (shift, control, alt). If no modifier key is pressed along with the arrow keys, node and its control points are modified. Pressing shift will modify previous, control will modify following control point. Alt modifier allows user to move only the node itself. Second responsibility of keyboard handler is to allow a template to be chosen by the user. Keys “1” through “8” can be used for this purpose. Figure 5.4 shows usable templates. Buttons are not used for templates because there is space constraint on the dialog frame. Flip button mirrors entire path vertically. Clicking on the save button save the current Bézier path to *parameter.svg* file. Using this system allows us to demonstrate usage of parameters easily. To save the SVG file, the target file is opened, SVG header is saved, export function of Bézier path is called and file is closed. The architecture of this system makes this task trivial. Last button is close which closes the editor frame. After closing the dialog, close event is fired. This is used by Bézier control to update the image on the button.

5.8. Save/Load and Export

This project supports save/load and export functions. Save/load functionality allows us to save leaves for later usage while export function allows usage of generated leaf images in other applications. This system is designed to be extensible. Therefore, the save method should also be easily extended. We have defined an XML

format for the generated leaves. XML save files can easily be modified while preserving backward and forward compatibility. XML Schema of a leaf file is shown in Appendix A.

5.8.1. XML Format

Leaf XML has a root tag of *leaf*. Within the root tag, parent tag of *stems* contains all vein levels (using *stem* tag) including the root level. For every *stem* tag, id and sub-vein list, and parameters are saved. Sub-veins are saved by their vein-id, so it is possible to save recursive vein-levels with this method. In the leaf XML, all parameters are saved by their names. This method is used so that if another transition function is implemented, it can used without changing the structure of the XML file. Enumerated values are saved with their corresponding integer values. Since Bézier parameters are complex objects, we have created a method to save these curves.

A Bézier curve is represented by *type="bezier"* attribute. *Bezier* tags also have *segments* attribute, which denotes the number of segments exists within this curve. The *firstpoint* tag is the first tag and represents start point of the curve. After this tag segments are saved with *segment* tags. Each segment tag has three points, control point 1 (denoted by *controlpoint id="1"*), control point 2 (denoted by *controlpoint id="2"*), and end point (denoted by *endpoint* tag). Points contain *x*, *y*, *randx*, *randy* (used for randomization) attributes along with a possible id attribute.

5.8.2. Save

This system saves XML data by propagating save request to all leaf levels. These levels save their data by either directly creating XML code or asking a Bézier path to save itself. Bézier path also calls XML save method of its points. This propagation method allows us to change the system easily. For instance, when Bézier randomization is implemented, save method of points is changed to fit this new property. Without further modification, system was able to save and load randomization information that is present on points.

Save function resides in `LeafBlueprint` class (see Section 5.4). This class opens the target file and creates XML header including XSD (XML Schema Definition) target. After this preparation, every leaf level is given chance to create their save strings. Simple variables are saved by the class itself while Bézier parameters are saved by `bezier` as mentioned before (see section 4.3.13). After each level builds its XML save string, this code is saved to the file and its closed by leaf blueprint.

5.8.3. Load

Our application uses “XMLParser” library by Frank Vanden Berghen to read XML data. This XML parser library is designed to be used in C++ applications and follows object oriented programming methods.

When a load request is made, the requested file is sent to parser which returns the root element. The root element contains leaf levels. These leaf levels are sent to newly created leaf levels to be loaded. When a leaf level receives an XML node to be loaded, it checks every parameter for existence, inexistent parameters are left with

their default values. If a parameter is a simple type, it is read and put into the corresponding variable immediately. However, Bézier parameters are sent to Bézier load function where it is loaded. This method is similar to save and allows us to change the system a lot easier.

Although XML schema definition for leaf XML specifies that all parameters are required, default values for missing parameters are used and no error is generated to preserve back compatibility. This allows future version of the program to be able to use save files from previous versions without causing problems.

5.8.4. SVG Export

One of the most important aims of this project is to be useful. Usefulness cannot be achieved without being able to transform the output of your application to a common format. To address this, we have built SVG export support into our application. SVG is defined in section 1.4.2.

SVG called is made to the generated leaf. As defined in Section 5.4 `Leaf` class is responsible for post generation operations. When an export call is made, the given file is opened and a plain SVG header is written to it. Then the export call is propagated to the midrib, the first sub leaf. Every sub leaf saves its blade and vein paths as SVG path node. Path node contains fill, stroke attributes as well as path data. This data is represented as commands and points as parameters to these commands. Commands are separated by spaces. This structure is detailed below. [24]

- **M** (move) command moves the start of the path to the given coordinate, it takes a point as parameter, its used as `Mx,y`

- **C** (curve to) command creates a curve segment. It requires three points. It is a Bézier curve segment and actually requires four control points. The actual first point is taken from the last point of the path. Its usage is *Cx1,y1 x2,y2 x3,y3*.

After a sub leaf writes its blade and vein data, it asks its sub leaves to write their blade and vein paths. When every sub leaf is written to file, *Leaf* class writes the end marker and closes the file.

Chapter 6 – Conclusion and Results

6.1. Conclusion

In this study we have determined set of parameters and an algorithm that is based on Bézier curves to reconstruct a leaf. The parameters can define almost any leaf. It is possible to represent simple, compound and lobed leaves as well as different leaf margins and venation patterns. Using Bézier curves, it is possible to create any leaf shape. Irregularities in the leaf shapes are achieved by using Bézier randomization method. Margins can be specified separately then repeated by the system. This leads to an easier usage. This system only lacks reticulate venation pattern which is added to future works category.

We have implemented our model with an application. This application can be used by graphic designers to supply them with leaf images whenever they require. They can either use an existing leaf definition or design it the way they want. Our application allows its user to edit leaf parameters and see the results in the same window. Since leaf images created almost instantly, designing leaves become an easy task. Using randomization, they can create many leaves of the same type to enrich their projects.

6.2. Advantages

The application created as the result of this study allows graphic designers and botanists to design images with ease. After designing process these leaf definitions can be used to create any number of leaf images. In addition, these definitions can

form a library where anyone can access leaf definitions of the plant they require. Randomization also allows different leaf images to be generated from the same set of parameters. This helps us to create more believable scenes. Consider a tree that is added to a garden scene, having different leaves makes the tree more realistic.

Apart from 3D design, our application can generate leaves that can be used in printed media or web graphics. For instance, many different leaves can be stacked to create a background image. A company may choose a leaf as their logo, ability to export vector graphics allow these images to be free of transformation problems. This ability also benefits printed media where the quality and design size can vary from few centimeters to meters.

6.3. Results and Discussions

In this section we have demonstrated results of our study. Since our study focuses on visual aspects of leaves, we have displayed these results visually. We have created several leaves and displayed their advantages and applications in the real world projects.

Apart from ease of use, best advantage of our system is producing vector images. As it has been discussed (see Section 1.4.2), vector images maintain their quality when transformed. In Figure 6.1 a photo of a leaf is compared to vector leaf that is created with our application. In this comparison, raster (bitmap) image looses its quality when enlarged while quality of vector version is constant. Moreover, a vector image can easily be edited. If desired, colors can be changed or the shape can be modified.

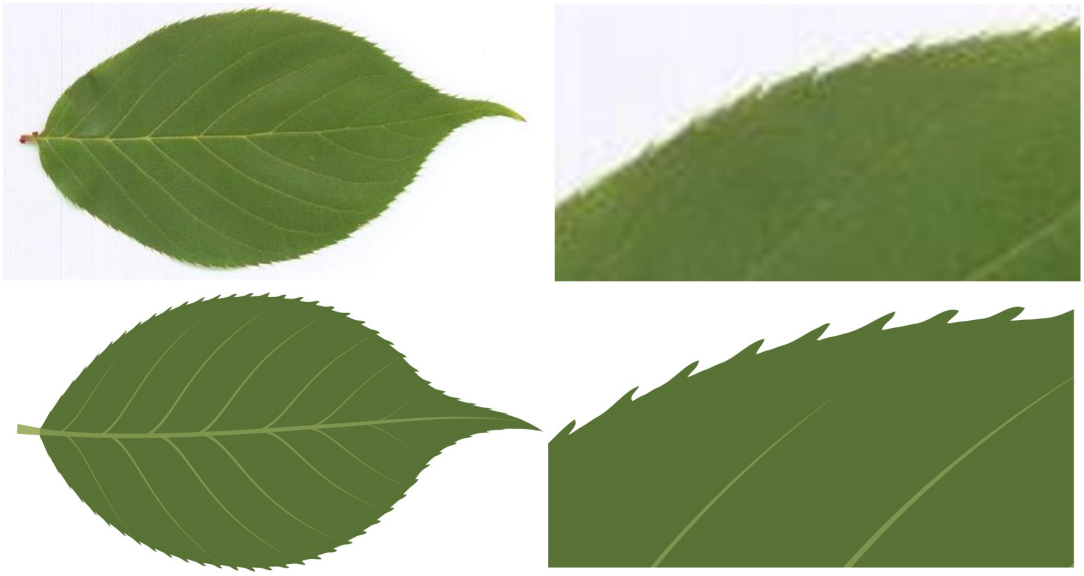


Figure 6.1: *Comparison of raster (top) and vector (bottom) images of a leaf at different zoom levels. Raster image is a photograph, while vector version is created by our application*

In Figure 6.2, a generated leaf is used as an icon. Highlight, gradient and shadow is added to create more appealing image. This image is still in SVG form and various sized icons can be exported using it. This is eastern redbud tree leaf, its shape is cordate with entire margin and it has palmate veins.

Figure 6.2: *A generated leaf used as an icon.*

A complex leaf is shown in Figure 6.3. This image is taken from a medicinal plant called Cannabis. Its extract is used prescription based medicine to cure glaucoma, vomiting and pain. However, this plant is also used to make drugs called marijuana. Therefore, its production is regulated in many countries. Because of this,

photographs of this plant cannot easily be taken. We are bound to images that can be found on the Internet. However, using our system, leaf definition of this plant can be made public and any required images can be produced over this template. To illustrate this purpose we have designed this leaf in our application and generated two leaves from this definition. In Figure 6.4, outline version of the leaf is shown. Both of the generated leaves are displayed in Figure 6.5, smaller version is used for outline version. Two leaves seem identical, however, they have small differences between them. A careful examination will reveal differences.

Figure 6.3: *A cannabis leaf*

Figure 6.4: *Cannabis leaf outline*

We have used cannabis leaf in several graphic design projects to demonstrate how they can be used in these kind of projects. First one is a concept design for a company called Cannabis Institute Labs and shown in Figure 6.6. For this project a logo (both colored and black and white) and a letterhead paper is designed.

Figure 6.5: *Cannabis leaves generated by our application*

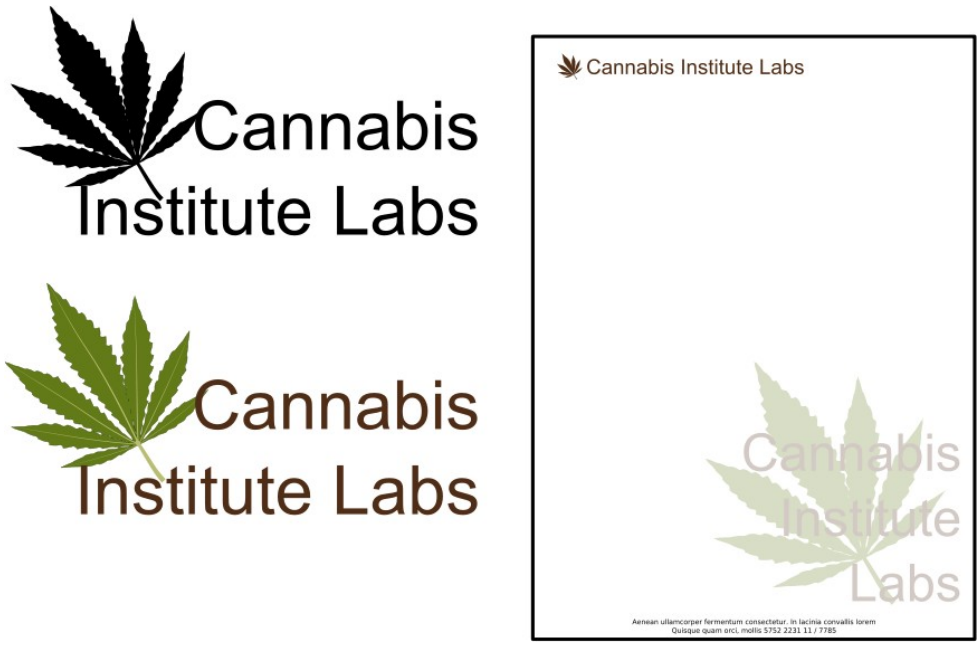


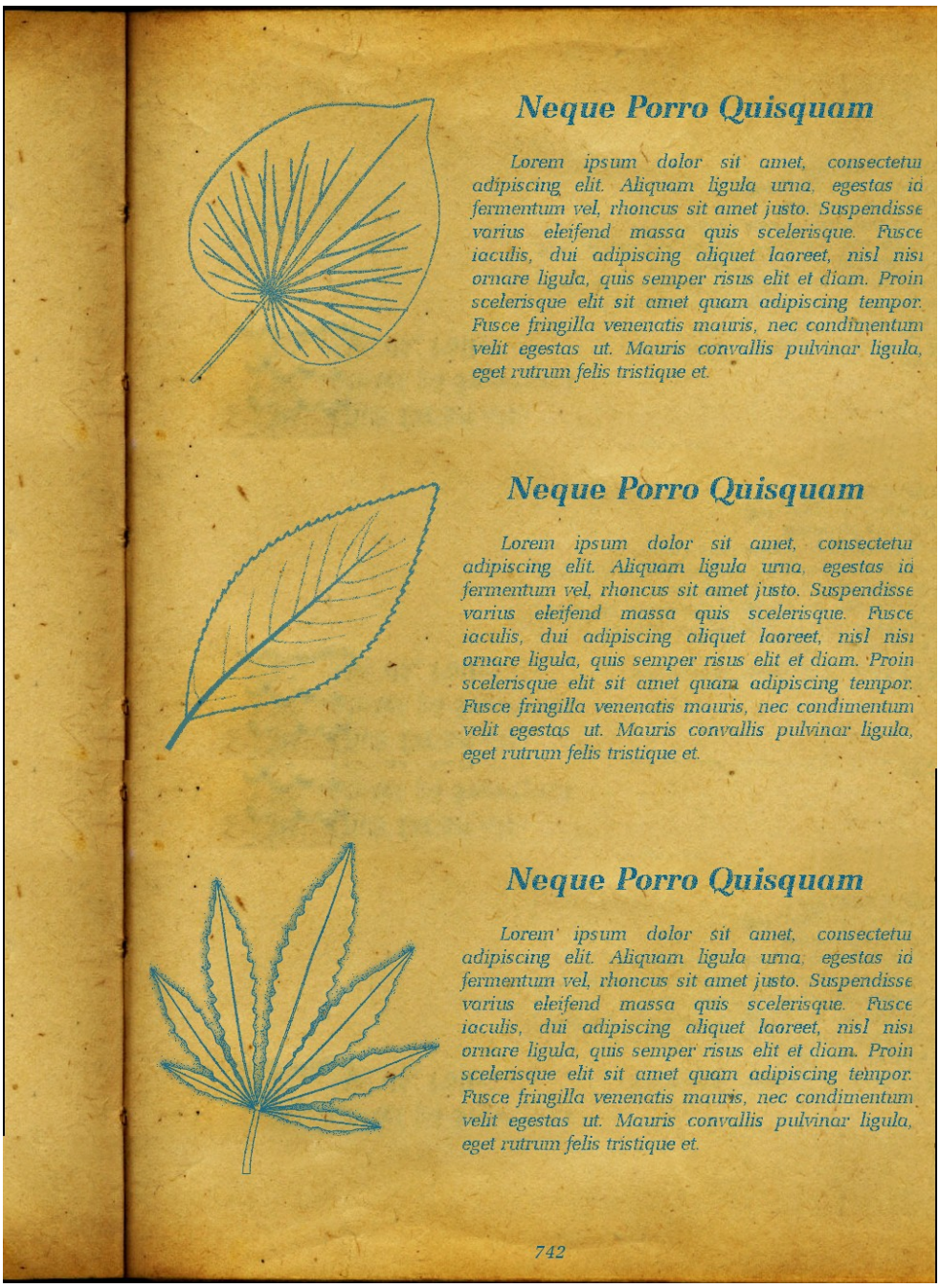
Figure 6.6: *Concept design for Cannabis Institute Labs*

Second project is to embed our leaves into an existing image. In this case we have created a burnt wood effect which can be observed in Figure 6.7.

Figure 6.7: *Leaves are embedded in photograph*

Our third project featuring same set of leaves is a compilation project shown in Figure 6.9. In this project we have placed outline versions of our leaves on an old looking book page. Leaves are colored in blue and dissolve effect is used on them. Moreover, we have added text to complete the project. We have used GIMP for this project. The result of this project is displayed in Figure 6.9.

Figure 6.8*: An artificial scene featuring a wall and an ivy plant*



Neque Porro Quisquam

Lorem ipsum dolor sit amet, consectetu adipiscing elit. Aliquam ligula urna, egestas id fermentum vel, rhoncus sit amet justo. Suspendisse varius eleifend massa quis scelerisque. Fusce iaculis, dui adipiscing aliquet laoreet, nisl nisi ornare ligula, quis semper risus elit et diam. Proin scelerisque elit sit amet quam adipiscing tempor. Fusce fringilla venenatis mauris, nec condimentum velit egestas ut. Mauris convallis pulvinar ligula, eget rutrum felis tristique et.

Neque Porro Quisquam

Lorem ipsum dolor sit amet, consectetu adipiscing elit. Aliquam ligula urna, egestas id fermentum vel, rhoncus sit amet justo. Suspendisse varius eleifend massa quis scelerisque. Fusce iaculis, dui adipiscing aliquet laoreet, nisl nisi ornare ligula, quis semper risus elit et diam. Proin scelerisque elit sit amet quam adipiscing tempor. Fusce fringilla venenatis mauris, nec condimentum velit egestas ut. Mauris convallis pulvinar ligula, eget rutrum felis tristique et.

Neque Porro Quisquam

Lorem ipsum dolor sit amet, consectetu adipiscing elit. Aliquam ligula urna, egestas id fermentum vel, rhoncus sit amet justo. Suspendisse varius eleifend massa quis scelerisque. Fusce iaculis, dui adipiscing aliquet laoreet, nisl nisi ornare ligula, quis semper risus elit et diam. Proin scelerisque elit sit amet quam adipiscing tempor. Fusce fringilla venenatis mauris, nec condimentum velit egestas ut. Mauris convallis pulvinar ligula, eget rutrum felis tristique et.

742

Figure 6.9: *Libram of leaves, our leaves are place on an old book image*

We have also added more generated leaves displayed in Figure 6.10. In Figure 6.8, Figure 6.11 and Figure 6.12 more 3D scenes are shown.

Figure 6.10: *Leaf images generated by our application*

Figure 6.11: *A 3D scene using two leaves that are generated by our system, plants are designed in XFrog*

Figure 6.12: *A 3D scene using a leaf texture generated by our application, vine is created using Ivy Generator*

6.4. Future Works

There are still features that we have decided to design and implement. Since a project is never truly finished, more parameters, methods, and algorithms are planned to be added. The following list is the summary of the features that are planned as our future works.

- Color and texture mapping will provide more ways to customize the look of the generated leaf
- Different methods for vein placement will be searched and added as necessary
- Aging of the leaf will be added to cover all the necessary leaf textures to create a complete tree
- Decaying of leaf because of age and possible diseases
- Better rendering method will be devised
- Implementation of bump and/or displacement map generation for 3D usage
- A wrapper to create completely randomized leaves where the parameters are defined by the application
- A wrapper that will allow user to specify leaf parameters through specifying biological data (e.g. the amount of the water that the plant consumes)
- Smoothing leaf blade using Bézier union and smooth operations

References

[1]: Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, Brendan Lane, The use of positional information in the modeling of plants, SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, 2001

[2]: Bernd Lintermann and Oliver Deussen, Interactive Modeling of Plants, IEEE Comput. Graph. Appl., 1999

[3]: Katsuhiko Onishi and Shoichi Hasuike and Yoshifumi Kitamura and Fumio Kishino, Interactive modeling of trees by using growth simulation, VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology, 2003

[4]: Aristid Lindenmeyer, Mathematical models for cellular interaction in development, J. Theoret. Biology, 1968

[5]: XAO Yu-kun, LI Yun-feng, ZHU Qing-sheng, LIU Yin-bin, Modeling Leaves Based on Real Image, Journal of Shangai University, 2004

[6]: Adam Runions, Martin Fuhrer, Brendan Lane, Pavol Federl, Anne-Gaëlle Rolland-Lagan, Przemyslaw Prusinkiewicz, Modeling and visualization of leaf venation patterns, SIGGRAPH '05: ACM SIGGRAPH 2005 Papers, 2005

[7]: Shenglian Lu and Chunjiang Zhao and Xinyu Guo and Changfeng Li, Modeling Curled Leaves, ICIG '07: Proceedings of the Fourth International Conference on Image and Graphics, 2007

[8]: Mark S. Hammel and Przemyslaw Prusinkiewicz and Brian Wyvill, Modelling compound leaves using implicit contours, CG International '92: Proceedings of the 10th International Conference of the Computer Graphics Society on Visual computing: integrating computer graphics with computer vision, 1992

[9]: , XFrog Web Site, , http://www.xfrog.com/

[10]: Various authors, Vector graphics, 2009, http://en.wikipedia.org/wiki/Vector_graphics

[11]: Chris Lilley, About SVG, 2004, http://www.w3.org/Graphics/SVG/About

[12]: Jackie Neider, Tom Davis, and Mason Woo , OpenGL Redbook, 1993

[13]: Richard S. Wright, Benjamin Lipchak, Nicholas Haemel, OpenGL SuperBible, 2004

[14]: Greg Roelofs, Introductiohn to PNG Features, 2009, http://www.libpng.org/pub/png/pngintro.html

[15]: David Beattie, John White, Leaf Identification, 1999, http://cas.psu.edu/docs/CASDEPT/Hort/LeafID/Default.html

[16]: Debivort, Leaf morphology, 2006 (http://commons.wikimedia.org/wiki/File:Leaf_morphology_no_title.png)

[17]: Christopher Brickell, The Royal Horticultural Society, 1996

[18]: Various authors, Bézier Curve, 2009, http://en.wikipedia.org/wiki/Bézier_curve

[19]: Andés Iglesias, Bezier Curves and Surfaces, 2001

[20]: R. Winkel, Generalized Bernstein Polynomials and Beizer Curves, 2001

[21]: Dr Thomas Sederberg, BYU Bézier curves, 2003

[22]: M. S. Floater, Derivatives of Rational Bezier Curves, 1991

[23]: Maxim Shemanarev, Adaptive Subdivision of Bezier Curves, 2005, http://www.antigrain.com/research/ adaptive_bezier

[24]: David Eisenberg, SVG Essentials, 2002

Appendix A. XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="leaf" targetNamespace="http://darkgaze.org/leaf.xsd"
elementFormDefault="qualified" xmlns="http://darkgaze.org/leaf.xsd"
xmlns:mstns="http://tempuri.org/leaf.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:NS="http://darkgaze.org/leaf.xsd">
  <xs:complexType name="point">
    <xs:attribute name="x" type="xs:decimal" />
    <xs:attribute name="y" type="xs:decimal" />
    <xs:attribute name="randx" type="xs:decimal" />
    <xs:attribute name="randy" type="xs:decimal" />
  </xs:complexType>
  <xs:complexType name="controlpoint">
    <xs:attribute name="id" type="xs:positiveInteger" />
    <xs:attribute name="x" type="xs:decimal" />
    <xs:attribute name="y" type="xs:decimal" />
    <xs:attribute name="randx" type="xs:decimal" />
    <xs:attribute name="randy" type="xs:decimal" />
  </xs:complexType>
  <xs:complexType name="limitbreakrandomize">
    <xs:attribute name="rla" type="xs:decimal" />
    <xs:attribute name="rlbc" type="xs:decimal" />
    <xs:attribute name="rlbl" type="xs:decimal" />
    <xs:attribute name="rlbh" type="xs:decimal" />
  </xs:complexType>
  <xs:complexType name="bezier">
    <xs:sequence>
      <xs:element name="firstpoint" type="point" />
      <xs:element name="segment">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="controlpoint" minOccurs="2" maxOccurs="2"
type="controlpoint" />
            <xs:element name="endpoint" type="point" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="type" type="xs:string" fixed="bezier" />
    <xs:attribute name="segments" type="xs:positiveInteger" />
  </xs:complexType>
  <xs:element name="leaf">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="stems">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="stem" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="curvature" type="bezier" />
                    <xs:element name="power" type="bezier" />
                    <xs:element name="shape" type="bezier" />
                    <xs:element name="thickness" type="bezier" />
                    <xs:element name="stemspacing" type="bezier" />
                    <xs:element name="straighten" type="bezier" />
                    <xs:element name="angle" type="bezier" />
                    <xs:element name="length_power" type="bezier" />
                    <xs:element name="margin" type="bezier" />
                    <xs:element name="marginstrength" type="bezier" />
                    <xs:element name="maxlength" type="xs:decimal" />
                    <xs:element name="veinlength" type="xs:decimal" />
                    <xs:element name="distribution" type="xs:positiveInteger" />
                    <xs:element name="hasblade" type="xs:boolean" />
                    <xs:element name="hasvein" type="xs:boolean" />
```

```
                    <xs:element name="relativespacing" type="xs:boolean" />
                    <xs:element name="hasmargin" type="xs:boolean" />
                    <xs:element name="marginsizeleafpower" type="xs:decimal" />
                    <xs:element name="marginsizerelation"
type="xs:positiveInteger" />
                    <xs:element name="relativemarginrepeat" type="xs:boolean" />
                    <xs:element name="marginrepeat" type="xs:decimal" />
                    <xs:element name="tip" type="xs:boolean" />
                    <xs:element name="anglerandomize"
type="limitbreakrandomize" />
                    <xs:element name="spacingrandomize" type="limitbreakrandomize"
/>
                    <xs:element name="lengthrandomize"
type="limitbreakrandomize" />
                    <xs:element name="marginrepeatrandomize"
type="limitbreakrandomize" />
                    <xs:element name="levelstrengthrandomize"
type="limitbreakrandomize" />
                    <xs:element name="substems">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="substem" maxOccurs="unbounded"
type="xs:IDREF" />
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="id" type="xs:ID" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="version" type="xs:string" />
    </xs:complexType>
    <xs:key name="key2">
      <xs:selector xpath=".//NS:stem" />
      <xs:field xpath="@id" />
    </xs:key>
    <xs:keyref name="stemsubstems" refer="key2">
      <xs:selector xpath=".//NS:substems" />
      <xs:field xpath="NS:substem" />
    </xs:keyref>
  </xs:element>
</xs:schema>
```

Appendix B. Sample SVG Document

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="20cm" height="16cm" viewBox="0 0 512 256"
xmlns="http://www.w3.org/2000/svg" version="1.1">
<g id="rootblade">
 <path stroke="#000000" fill="#008020" stroke-width="0.00" d=" M2.57,249.50
C25.34,249.42 48.10,248.35 70.81,246.68  C81.21,245.91 91.59,245.02 101.97,244.04
C102.39,237.10 95.34,233.50 91.71,228.56  C78.62,214.27 77.42,192.72 81.60,174.67
C84.12,167.96 87.58,161.57 91.86,155.82  C103.32,140.97 120.19,129.95
138.59,126.04  C157.70,121.14 177.68,119.90 197.33,121.15  C218.99,122.54
240.59,127.16 260.64,135.54  C289.43,147.22 309.60,171.93 328.89,195.17
C339.30,206.45 354.00,211.94 366.22,220.86  C366.22,220.86 366.22,221.86
366.22,221.86  C354.28,230.58 340.10,236.90 331.12,249.08  C315.10,273.12
301.28,299.89 277.20,317.07  C260.77,328.93 242.21,337.84 222.87,343.81
C204.00,349.65 184.20,352.92 164.42,352.56  C144.44,352.94 124.14,346.07
108.51,333.60  C102.81,328.86 97.81,323.24 93.83,316.99  C86.03,300.06
84.13,278.35 94.72,262.14  C97.47,256.65 103.80,251.80 102.06,245.04
C91.68,246.02 81.29,246.91 70.89,247.68  C48.15,249.35 25.37,250.42 2.57,250.50  "
/>
</g>
<g id="rootvein">
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M0.37,248.20
C23.35,248.19 46.33,247.17 69.25,245.55  C92.26,243.92 115.23,241.68 138.16,239.23
C160.89,236.81 183.60,234.17 206.32,231.69  C252.03,226.70 297.90,222.33
343.91,221.65  C297.96,223.40 252.22,228.57 206.60,234.11  C183.90,236.87
161.22,239.73 138.51,242.33  C115.55,244.95 92.55,247.32 69.51,249.03
C46.50,250.73 23.44,251.79 0.37,251.80  " />
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M117.10,243.35
C115.96,244.09 114.83,244.84 113.72,245.60  C112.60,246.36 111.50,247.14
110.41,247.93  C109.20,248.82 107.99,249.71 106.81,250.62  C104.67,252.26
102.59,253.98 100.54,255.73  C102.53,253.91 104.60,252.18 106.71,250.50
C107.89,249.57 109.07,248.65 110.28,247.76  C111.37,246.96 112.47,246.18
113.59,245.41  C114.71,244.64 115.84,243.89 116.98,243.17  " />
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M117.36,243.55
C115.06,246.25 112.87,249.05 110.78,251.91  C108.76,254.68 106.85,257.51
105.01,260.39  C103.09,263.43 101.26,266.53 99.53,269.69  C96.15,275.85
93.13,282.20 90.41,288.69  C92.94,282.12 95.84,275.69 99.14,269.48  C100.84,266.29
102.63,263.15 104.53,260.09  C106.34,257.17 108.26,254.32 110.27,251.55
C112.37,248.66 114.58,245.86 116.91,243.16  " />
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M100.53,267.51
C100.31,267.87 100.09,268.24 99.87,268.61  C99.60,269.07 99.33,269.53 99.06,269.99
C98.78,270.46 98.52,270.93 98.25,271.39  C97.76,272.24 97.29,273.09 96.80,273.93
C97.26,273.07 97.74,272.22 98.22,271.37  C98.48,270.90 98.75,270.44 99.01,269.97
C99.27,269.50 99.53,269.03 99.80,268.57  C100.01,268.20 100.23,267.82
100.44,267.45  " />
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M117.68,243.60
C115.61,248.42 113.82,253.35 112.25,258.35  C110.70,263.31 109.38,268.34
108.26,273.42  C107.13,278.56 106.20,283.74 105.44,288.95  C104.68,294.15
104.09,299.38 103.64,304.61  C103.19,309.83 102.87,315.05 102.67,320.28
C102.68,315.04 102.87,309.81 103.22,304.58  C103.57,299.33 104.07,294.08
104.75,288.85  C105.43,283.62 106.30,278.40 107.39,273.23  C108.47,268.12
109.76,263.05 111.32,258.05  C112.88,253.02 114.70,248.06 116.82,243.23  " />
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M107.87,274.65
C106.56,277.73 105.32,280.84 104.13,283.96  C102.93,287.08 101.80,290.22
100.72,293.37  C99.63,296.54 98.59,299.74 97.61,302.94  C95.63,309.36 93.86,315.83
92.16,322.33  C93.69,315.79 95.44,309.30 97.35,302.87  C98.31,299.65 99.32,296.44
100.36,293.25  C101.41,290.08 102.50,286.92 103.63,283.78  C104.76,280.62
105.95,277.49 107.17,274.37  " />
  <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M116.83,242.89
C114.15,239.02 111.68,235.00 109.41,230.87  C107.12,226.69 105.04,222.40
103.14,218.03  C101.28,213.73 99.59,209.36 98.06,204.92  C95.03,196.11
92.61,187.09 90.66,177.98  C92.90,187.01 95.47,195.97 98.64,204.72  C100.22,209.11
101.95,213.45 103.86,217.72  C105.78,222.05 107.88,226.30 110.17,230.45
C112.43,234.55 114.87,238.55 117.51,242.42  " />
```

```
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M101.85,214.85
C100.59,212.59 99.36,210.31 98.16,208.02  C96.98,205.78 95.82,203.53 94.69,201.27
C93.52,198.95 92.39,196.61 91.27,194.26  C89.09,189.64 86.99,184.98 85.05,180.26
C87.12,184.92 89.22,189.58 91.45,194.17  C92.58,196.51 93.75,198.83 94.94,201.14
C96.10,203.39 97.29,205.62 98.51,207.83  C99.76,210.09 101.04,212.34 102.35,214.56
" />
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M116.92,242.93
C114.58,241.06 112.31,239.12 110.12,237.09  C107.92,235.07 105.79,232.97
103.73,230.81  C101.67,228.64 99.68,226.40 97.75,224.11  C93.97,219.60
90.45,214.87 87.16,209.99  C90.57,214.77 94.19,219.42 98.03,223.87  C99.98,226.13
102.00,228.34 104.07,230.48  C106.14,232.62 108.27,234.70 110.46,236.71
C112.66,238.73 114.91,240.67 117.23,242.54  " />
 <path stroke="#000000" fill="#804010" stroke-width="0.00" d=" M116.97,242.90
C116.27,242.62 115.56,242.32 114.86,242.02  C114.03,241.66 113.19,241.29
112.37,240.90  C111.54,240.52 110.72,240.13 109.90,239.73  C108.41,239.00
106.94,238.24 105.47,237.45  C106.95,238.20 108.45,238.93 109.95,239.64
C110.77,240.03 111.59,240.42 112.42,240.79  C113.25,241.16 114.08,241.53
114.92,241.89  C115.62,242.19 116.32,242.49 117.02,242.78  " />
</g>
</svg>
```