UNIVERSITY OF CALGARY

Interactive Visualization and Modeling of Plant Structures in Virtual Reality

by

Jeremy Adam Hart

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

June, 2020

© Jeremy Adam Hart 2020

Abstract

Virtual reality facilitates a level of immersion impossible with two-dimensional interfaces. Through it, users are able to interact with virtual environments while gaining many of the benefits of interacting within the physical world, most notable of which for my work is the improved perception of depth and the ease of three dimensional interactions. I investigate how virtual reality impacts the study and modeling of plant structures by introducing two applications: Virtual Network Explorer (ViNE) and Tree Wand. ViNE has been designed to aid in the study of complicated networks, such as the intricate vascular structure of flower heads. It allows the user to immerse themselves inside networks, and segment their various components. In Tree Wand a controller is used as a brush to guide the growth of a tree, combining the user's input with a biologically motivated algorithm to simulate a tree's development. In addition, it provides an interface for the user to manipulate the algorithm's parameters, allowing a wide variety of tree forms to be represented. Taken together, these applications demonstrate the usefulness of virtual reality in the visual analysis and modeling of plants.

Table of Contents

Abstract . . . ii
Table of Contents . . . iii
List of Figures . . . vi
1 Introduction . . . 1

I Virtual Network Explorer 3
2 Problem position . . . 4
3 Background . . . 6
3.1 Vasculature visualization and segmentation . . . 6
3.2 Viewing volumetric data in 3D . . . 8
3.3 Viewing interaction in virtual reality . . . 10
4 Design . . . 12
5 Implementation . . . 15
5.1 3D manipulation . . . 15
5.1.1 Design . . . 15
5.1.2 Translation . . . 17
5.1.3 Rotation . . . 17
5.2 Depth cueing . . . 19
5.2.1 Design . . . 19
5.2.2 Method . . . 23
5.2.3 Results . . . 26
5.3 View saving . . . 28
5.3.1 Design . . . 28
5.3.2 Method . . . 29
5.4 Painting . . . 30
5.4.1 User interface . . . 31
5.4.2 Color storage on the mesh . . . 32
5.4.3 Brush intersection . . . 35
5.5 Undo and redo . . . 37
5.6 Selective visibility . . . 38
5.7 Optimization . . . 42
5.7.1 Parallelization . . . 42
5.7.2 Synchronization of color data . . . 44
5.7.3 Rendering . . . 48
6 Application examples and results . . . 50
6.1 Gerbera . . . 50
6.2 Helianthus . . . 55
6.3 Craspedia . . . 58
6.4 Animated flythrough . . . 60

II Tree Wand 62

7 Problem position 63
8 Background 64
8.1 Modeling in virtual reality 64
8.2 Tree modeling 66
9 Generative algorithm 71
9.1 Tree growth in nature 71
9.2 Tree data structure 73
9.3 Basic growth model 75
9.4 Competition for space 77
9.4.1 Influences on growth direction 79
9.5 Competition for light 81
9.5.1 Extended Borchert-Honda 81
9.5.2 Shadow propagation 84
9.5.3 Grid representation 87
9.6 Branch radius 89
9.7 Internode lengths 90
9.8 Algorithm 91
10 Tree appearance 94
10.1 Branch modeling 94
10.1.1 Skeleton generation 94
10.1.2 Mesh generation 96
10.1.3 Texturing 101
10.2 Leaves 106
10.2.1 Leaf positioning 106
10.2.2 Leaf orientation 107
10.3 GPU acceleration 110
11 VR interface design 114
11.1 Distributing markers 114
11.2 Pruning 116
11.3 Controller menu 117
11.4 Modifying parameters 118
11.4.1 Parameter windows 118
11.4.2 Sliders 120
11.4.3 Texture selectors 121
11.4.4 Influence triangle 122
12 Results 123
12.1 Generating trees 123
12.2 Performance 132
13 Conclusion 135
13.1 Contributions 135
13.2 Future work 137

A The HTC Vive controller 140

B The parameter windows 141
B.1 Tree form window 141
B.2 Appearance window 143
B.3 Light window 145
Bibliography 147

List of Figures and Illustrations

5.1 Spindle and Wheel transformations. (a) Initial configuration; (b) Yaw/Roll; (c) Pitch; (d) Scaling . . . 15
5.2 Three depth cueing scenarios with fixed gradient based on distance from viewer. Red lines indicate the beginning and end of the depth cueing gradient. (a) The model is at the ideal scale and distance: depth cueing creates noticeable contrast across the model. (b) The model is too small and close: depth cueing has little effect on the original color. (c) The model is too large and far away: most of the model has faded into the background, and the parts that haven't are still dark. . . . 20
5.3 The blue trapezoid indicates the view frustum, and the dotted lines indicate the closest and furthest points within it. (a) and (b) use the distance to the red concave object to determine the range of the depth cueing. (c) and (d) use the distance to the yellow convex hull. The change in depth cueing between (a) and (b) is significant, while the change between (c) and (d) is slight. . . . 21
5.4 An iteration of the Quickhull algorithm. (a) The geometry at the end of the previous iteration. (b) Randomly select a face, and the furthest point in the direction of its normal. (c) Remove all faces for which the point lies in the direction of their normal. (d) Connect boundary edges to new point using a triangle fan. . . . 22
5.5 Finding the closest point on a triangle. The closest point to each edge is shown using the same color. The closest point to the triangle will be the closest of these three to E. . . . 25
5.6 Comparison of depth cueing turned off and on. Left side: No depth cueing. Right side: Depth cueing. Top: Ideal lighting angle for visibility. Bottom: Poor lighting angle, only ambient shading visible. . . . 26
5.7 View of the outside of the flower head. Left: No depth cueing. Right: Depth cueing. . . . 27
5.8 Painting controls for the left and right controllers . . . 31
5.9 Color wheel for color selection. The currently selected color slice is extruded upwards, and an indentation is left underneath the thumb's position. . . . 32
5.10 Three options for color storage on a mesh. (a) Unpainted (b) Per-pixel (c) Per-face (d) Per-vertex. . . . 33
5.11 Color wheel indicating the colors made invisible using contrasting radial lines. 39
5.12 Noise in center of flower head removed using selective visibility. . . . 40
5.13 A single vein of a gerbera flower head, and its children, segmented apart from the others. . . . 41
5.14 Comparison between single and multi-threaded pipleines for OpenVR (a) Single-threaded pipeline in scenario where painting takes long enough that `WaitGetPoses()` misses its frame's synchronization point. (b) Multi-threaded pipeline where rendering and querying tracking information operates on one thread, while painting operates on another. The painting time no longer causes synchronization points to be missed. . . . 44

5.15 Visualization of single, double and triple-buffering for rendering pipelines. (a) Two threads with a single shared buffer. Each thread needs to wait for the other thread to finish using it before it can gain access. (b) Two threads with two shared buffers. While neither thread needs to wait for access, the render thread can become stuck with an older version despite several completed writes. (c) Two threads with three shared buffers. Neither thread needs to wait for access, and the render thread always receives the most recent completed changes. 45
5.16 A comparison of the timelines for using `glSubBufferData` and PinnedMemory to transfer geometry data. "Submit n" refers to the initiation of a draw call on the CPU using the data in buffer n. The central region indicates of each diagram indicates when each thread is using each buffer. (a) `glBufferSubData` is used to explicitly upload the data with "Upload n". The CPU is exclusively used to read and write from the three buffers. (b) The GPU directly accesses the pinned memory storage as directed by the CPU read thread. 47

6.1 "Unwrapping" of each component of a gerbera flower head layer by layer. . . 51
6.2 A Gerbera flower head separated into its components. (a) Bracts. (b) Ray Florets. (c) Disk and trans florets. (d) Floret veins. (e) Centripetal veins. (f) Isolated vein overlaid on top of centrifugal veins. Components in bottom row are larger relative to the middle row for improved visibility. 52
6.3 An isolated gerbera vein. Brown centrifugal veins originating from the flower's stem. Red centripetal vein trails up and back towards the center of the flower head. Yellow floret veins connect the ray and disk florets to the brown and red veins. Green bract veins connect to the centrifugal veins. 54
6.4 Helianthus components. (a) Whole flower head. (b) Ray florets. (c) Base. (d) Disk florets. (e) Floret veins. (f) Main veins. (g) Isolated vein. 56
6.5 Isolated vein from the Helianthus mesh. Brown centrifugal veins branch into green centrifugal veins and orange bract veins. 57
6.6 (a) Craspedia flower head. (b) Florets. (c) Floret veins. (d) Main veins. . . . 58
6.7 (a) Unsegmented flower head. (b) Fully segmented flower head. (c) Vasculature with surrounding noise. (d) Vasculature without surrounding noise. 59
6.8 A frame from a flythrough of the Gerbera flower head, rendered in Blender and colored in ViNE. 60
6.9 Frames from Blender animation. 61

8.1 Pythagoras tree 66

9.1 Twig morphology 71
9.2 (a) Branching angles measure the angle between the previous internode's heading, and the main or lateral shoots' headings. (b) The divergence angle between successive branches of a tree with a main branching angle of 0. This angle is often 137.5°: known as the "golden angle". Branches are numbered in order of lowest to highest. 73

9.3 Two modes of branching. 76
9.4 Circular markers being assigned to buds. The color of the marker matches the view cone of the bud it is assigned to. Grey circles represent the occupation zone within which markers are removed. 78
9.6 Influences affecting the direction of internode development. Depicts a strong weighting towards environmental influence. The tropism angle is 90 degrees from horizontal. 80
9.5 How to calculate the tropism vector associated with a heading, $\hat{\boldsymbol{h}}$. 80
9.7 Light accumulation and vigor distribution. Light collected at the buds is propagated downwards to the base, then redistributed up in the form of vigor according to Equation 9.7. 82
9.8 Shadow propagation methods. (a) The height-based shadowing from Palubicki et al. which calculates the degree of shadow solely based on height. (b) The distance-based shadowing used in Tree Wand, which calculates the degree of shadow using distance to the cell. 85
9.9 Fine grids accessible from hash table. Each entry in the hash table uses Equation 9.14 to map the three dimensional index to a pointer to a fine grid with the corresponding color. 88
9.10 Illustration of TreeFold and TreeUnfold higher-order functions applied to a tree. f is the function applied to the nodes, "+" is used in a fold to combine children, and "b" is the base value for f. 92

10.1 The smoothing of internode connections using rational Bézier curves. A quadratic rational Bézier curve is constructed around every corner between a point halfway along the first internode, the node between, and a point halfway along the second internode. 95
10.2 Two photos of trees depicting the shape of branching points. (Left) A photo of a simple branching pattern. The region where the two branches merge is only impacted by the trunk and the branch. (Right) A photo of a complicated branching pattern. Several branches impact the way their neighbours merge into the trunk. 96
10.3 A branch modeled using a generalized cylinder consisting of a circle swept along a quadratic Bézier curve. The vector $\hat{\boldsymbol{b}}_x(u)$ (green) along with the tangent (red) are used to generate a frame for the circle along the curve. . . 98
10.4 Progression of increasing branch radius left to right. Generalized cylinders work well with smaller radii, but result in a less smooth appearance with larger radii with the intersection becoming more apparent. 101
10.5 Texture coordinates assigned using u and v parameters from generalized cylinder. The checkerboard texture is non-uniformly stretched across the generalized cylinders. 102
10.6 Spacing for a rational Bézier curve with a weight of 3 for the center point. (a) Directly using the u parameter from Equation 10.10. (b) Using Equation 10.13. 105
10.7 Using the formula from Equation 10.13 to assign texture coordinates. (Left) The model from Figure 10.5 instead (Right) A linear general cylinder. 106

10.8 Rotations around the axes of a leaf. Rotation around $\hat{\boldsymbol{h}}$ occurs when the petiole twists, rotation around $\hat{\boldsymbol{r}}$ results when it bends up and down, and rotation around $\hat{\boldsymbol{u}}$ occurs when it bends left and right. 108
10.9 Leaf reorientation with different elasticities. (a) Leaves with initial orientations (elasticity 0). (b) Leaves with elasticity 0.05 for all rotations. (c) Leaves with elasticity 0.1 for all rotations 110
10.10 OpenGL graphics pipleines. (a) A simplified version of the OpenGL pipeline as of OpenGL 4.6. (b) The OpenGL pipeline when using tessellation 111
10.11 Using tessellation shader to create generalized cylinder. The tessellation control shader determines dimensions of the abstract patch, and the tessellation evaluation shader maps each vertex in the abstract patch to a point in clip space. 112

11.1 Using the brush to paint a new branch on a tree. 115
11.2 Pruning of a branch on the tree. A partial press of the trigger previews nodes that would be pruned, while a full press removes them. 116
11.3 Controller menu functions counterclockwise from the green icon: (1) Appearance window (2) Tree form window (3) Light window (4) Turn pruning on/off (5) Save the current tree (6) Load saved tree (7) Start new tree. . . . 117
11.4 Controller interacting through the window by pointing at a slider. 119
11.5 A slider controlling the leaf size. 120
11.6 Icon selector for selecting the tree's bark texture. 121
11.7 Triangle for setting influence weights. The area of each of the red, green and blue sub-triangles are proprotional to the weight of that influence. 122

12.1 A tree with a well defined main trunk with a tropism of 90 degrees enabled for the branches beyond the first order. Left: without leaves; Right: with leaves. 124
12.2 A tree with two well established main branches. 125
12.3 A columnar tree. 126
12.4 A small tree with winding branches. 127
12.5 A large oak tree. 128
12.6 Maple tree demonstrating opposite branching. 130
12.7 A comparison of trees generated from three different methods for determining internode length. All three sets of tests were made using $\lambda = 0.6$. Left: Constant internode length. Middle: Internode length based on vigor. Right: Internode length based on branch order. 131
12.8 Graph relating the number of internodes to the time taken for a growth step. 133
12.9 Graph relating the number of internodes multiplied by the number of markers to the growth step duration. 134

A.1 The HTC Vive's controller. The controller can be tracked in 3D using the HTC Vive Lighthouse base stations. Trigger: Button pressed by the pointer finger which can be pressed down partially or fully. Controller clicks on a full press. Grip button: Pressed with middle and ring finger using an action similar to the user closing their fist. Menu button: A button that can be assigned a function within a VR application. Trackpad: A trackpad that tracks the position of the user's thumb. It can be used as a button when pressed down. Steam menu button: A button that opens up the Steam Home menu. Cannot be used for VR applications. 140

B.1 The tree form window. 141
B.2 The appearance window. 143
B.3 The light window. 145

Chapter 1

Introduction

Branching networks are ubiquitous in the natural world. They are present in a tree's branches, roots, and the veins within leaves. They are found everywhere from the veins that transport resources in organisms, to branching river deltas. It can often be difficult to analyze branching networks, as their very nature predisposes them towards complexity; each additional order of branching can increase the number of elements within it exponentially.

A tree may have tens of thousands of branches, with an even larger number of leaves. Amidst all of this complexity is organization, a pattern that governs how a tree develops in a way that maintains the particular character of its species. A tree is too complex, however, to distill this pattern by analyzing it in its entirety; to consider each of its branches individually. It is necessary to simplify it by either isolating part of the tree, or viewing it as a higher level abstraction.

The vasculature within flower heads exhibits a similar level of complexity. Each ray and disk floret on the flower head's surface connects to a vast network of veins which provide a path towards the stem. Unlike a tree, however, the vasculature within flower heads both splits and merges. The work I present in this thesis helps manage complexity in both the analysis of plant form, and the generation of it.

The increased availability and recent advancements in virtual reality have created a number of opportunities for innovative new approaches to analysis and modeling. Headsets provided by virtual reality systems allow for improved depth perception through binocular displays, which makes it easier to set objects apart based on their distance. In addition, the ability to view objects from different perspectives in virtual reality can align more closely to how a physical object would be studied in real life. The modeling of geometry also benefits

from improved depth perception, but it benefits most strongly from the use of a controller with three dimensional tracking and orientation, providing it with four additional degrees of freedom when compared to a computer mouse.

My thesis is divided into two parts, corresponding to the two applications I've developed, to examine the benefits virtual reality offers in the analysis and modeling of plant forms.

The first part of my thesis covers Virtual Network Explorer (ViNE), an application designed to assist in understanding complex vascular structures in plants. I have focused the design on the case study of analyzing the vasculature hidden within flower heads. ViNE also provides the functionality for manually segmenting and annotating data.

The second part of my thesis outlines Tree Wand, an application I designed to model trees within virtual reality. This application allows the user to express the complexity of a tree by using a three dimensional trackable controller as a brush to outline the tree's general form. Tree Wand is built on a biologically motivated algorithm which increases the plausibility of the generated trees. Additionally, it provides an interface for interactively modifying the parameters used to generate the trees from within virtual reality, allowing users to capture a variety of different tree forms.

Part I

Virtual Network Explorer

Chapter 2

Problem position

The external beauty of flowers can be easily appreciated, but beneath their surface lies rich and complex structures that are far less accessible. Hundreds of veins twist and turn their way through a flower head smaller than a fingertip, and within these networks are clues to the mechanisms that drive the flower's development. To unveil these clues, however, it is necessary to have the ability to visualize and interpret the vasculature.

There are a number of obstacles in the way of understanding a flower's vasculature structures. These networks are obscured not only by the flower's outer layers, but also by the internal tissues within which the veins are embedded. To be able to visualize the vasculature, it must first be isolated from the surrounding tissues.

Unlike that of leaves, vasculature in flower heads is not localized to a single plane. This presents challenges when using imaging methods which can only visualize a planar slice at a time. Difficulties can also arise when trying to view the three-dimensional vasculature on a two dimensional monitor, as the projection hinders the ability to perceive depth.

In addition, vasculature can be very dense. With hundreds of branching connections, it is challenging to follow and understand the path of a single vein without being overwhelmed by the surrounding complexity.

There are, however, some commonly used techniques to manage the complexity. Enhancement of depth can help to distinguish overlapping structures. Important structures can be tagged to help the user quickly identify them from their surroundings, such as marking the position of veins in cross sections [49]. The components of a dataset can also distinguished from each other through segmentation. These components are often distinguished by some structural or material property. For example, one way to segment a scan of a human

body is to separate it into bones, muscle, veins, and other organs. Once an image has been segmented, the groups can be visualized differently from each other, such as through differing colors. In addition emphasizing differences between the components of a data set, a segmentation can also be used to determine which components are to be visualized.

The objective of this part of the thesis was to develop a tool for exploring and analyzing the vasculature in plants, particularly that of flower heads. To effectively visualize the vasculature the user [1] must be able to easily interpret the three-dimensional shape of the veins and view the vasculature isolated from the rest of the data set. The segmentation process should be easy to use, easy to learn and efficient. For these reasons, I have chosen to explore using virtual reality as a platform.

While flower heads are the main application domain being targeted, many of these same problems occur in other contexts. Vasculature and network structures can be found throughout plant organisms. Plant roots form tree structures, being obscured by soil instead of plant matter. For an example outside of plant biology, neurons and bronchi also form complex networks. Their study may be aided by this tool as well.

[1] The intended user being a biologist, student, or educator.

Chapter 3

Background

3.1 Vasculature visualization and segmentation

Kang et al. [49] analyzed the vascular patterns of the plant *Arabidopsis thaliana* using a procedure presented by Donnelley et al. [28]. Kang et al. sectioned the plant into slices using a microtome. This procedure uses a diamond knife to cut the plant into 2 μm thick planar slices, which can then be viewed using bright field microscopy. Using this method, samples were taken along the main stem in a region around the terminal shoot. Veins were identified by the expression of the ATBH8::GUS marker gene. The vasculature of *Arabidopsis* ran mostly perpendicular to the transversal planes of the slices, which resulted in most of the veins being depicted as cross sections.

To reconstruct the vasculature, the authors stepped through photographs of these slices on a computer, and manually made linkages between the veins in adjacent slices. In this manner, they obtained a topological representation of the vascular network. While requiring a large investment of manual labor and expertise in inferring structure from the two dimensional slices, this method has the advantage of clearly identifying primary veins in early stages of development through the use of the ATHB-8::GUS reporter and generating high resolution images.

Dhondt et al. [26] explored the use of high resolution X-ray computed tomography to create *in vivo* scans of *Arabidopsis* seedlings. The non-invasive nature of these scans allowed them to track the development of the seedlings over multiple stages with a limited effect on growth. They segmented the model into different components using thresholding and a watershed algorithm [53], though none of these components were vasculature. Lee et al. [55] used a similar method, scanning an *Antirrhinum* flower with optical projection tomography

[93], and using a region growing method [36] to segment the vasculature.

Common techniques between these two studies are the use of thresholding and watershed or region growing algorithms. The methods used by these studies aligns with typical methods used for segmentation described by Long et al. [61]. Segmenting while viewing an individual image slice at a time is still the most common method for biologists [61, 81, 102], though it can be a time consuming and laborious process. This indicates that despite the advancements in 3D rendering, there is still something appealing to analyzing volumetric data through 2D slices; Long et al. suggests that this may be attributed to the simplicity of viewing a two dimensional cross-sectional display.

Automatic segmentation methods have seen a certain amount of success, and are almost necessary when analyzing exceptionally large and complex volumetric data, which can be terabytes in size. However, they can result in a significant number of false negatives or false positives [60, 1]. These errors must be found and corrected manually, which reduces the amount of time saved by the automatic segmentation, and necessitates interactive methods for quickly identifying and correcting improperly segmented areas.

A number of tools have been developed with the intention of providing a manual or semi-automatic method of segmentation. Tools like this can be useful for either segmenting the entire dataset, or for validating and correcting output from automated methods.

Virtual finger [81] operates on 3D image stacks, using a mouse and monitor as its interface. Neurons and vascular structures can be traced by estimating the depth of curves the user draws on the screen. Their user study testing this technique demonstrated a significant improvement in segmentation time compared to working with individual image slices.

NeuroLucida [8], the most cited tool in neuron tracing publications, operates using a 2D interface as in virtual finger, and uses the volumetric data to estimate the depth of the cursor. In addition, it includes an array of automatic segmentation and flood filling features for tracing neuron networks, attempting to match the thickness of the curve segments to

that of the neurons. These automated algorithms have been designed towards the specific case of analyzing neuron networks, and does not see as much use outside of that domain. It is not clear how it would be applied to plant vasculature, and its commercial nature does not allow for easy comparisons.

MorphoNet [56] is a web browser based tool that is used to view and interact with 3D meshes extracted from segmented data. Using meshes instead of volumetric data — which can be large and computationally expensive to render — reduces the rendering cost for a less optimized browser environment. Using the 3D interface, the user can further segment the data into groups, and correct minor segmentation errors. The tool sets itself apart by also tracking segmented components over time to represent the development of the model, for example tracking lineage of divided cells.

In a recent work, Usher et al. [102] developed a neuron tracing tool through which the user interacts with a three dimensional image stack in virtual reality. Volume rendering techniques are used to visualize the data, allowing it to be viewed using a transfer function or as an isosurface. Volume rendering is typically requires more memory and computation time than rendering polygonal meshes, and rendering for virtual reality requires higher frame rates than other applications to avoid motion sickness. This is addressed by only visualizing a small 256^3 region of the image stack in focus, and using a paging system alongside sparse 3D textures to update the GPU with any new data needed. Neurons are traced by placing linked nodes along the path of the neurons. This tool was found to be intuitive, and users were able to learn how to use it within 10 minutes.

3.2 Viewing volumetric data in 3D

3D image stacks, or volumetric data, are often stored as a set of images with each voxel, a 3D pixel, having an "intensity". Different materials typically have different intensities which are dependent on the method used to obtain the data.

By interpolating between voxels, volumetric data can encode a continuous field of values. An “isosurface” can be defined on a continuous function [112] by defining a boundary where the function is equal to a constant value, called the iso-value. In the context of scanned data, isosurfaces can indicate where one material transitions to another.

Ray casting based methods can be used to visualize iso-surfaces by finding the intersection between it and the camera ray. Using ray marching, the continuous function is sampled at increments along the camera ray and the surface is defined between the first two consecutive samples which have values falling on opposite sides of the isovalue [57]. The intersection process can be accelerated by using hierarchical data structures to inform the sample placements [58].

An isosurface can also be extracted as a triangular mesh using the marching cubes algorithm [112, 64]. In this algorithm, the isosurface is represented as triangles between adjacent voxels that lie on different sides of the isovalue. Depending on which of a voxel’s neighours are on the inside or outside the isosurface, one of 15 templates is used to triangulate the surface between them.

Aside from rendering the volume as a surface, ray tracing based techniques can use a transfer function to collect information from multiple voxels in a single camera ray [30]. The transfer function maps intensity values to a color and opacity, in addition to other possible material properties, and is evaluated at multiple samples along the ray, accumulating the shading information until a certain accumulated opacity is reached. This technique is more computationally expensive than rendering a surface due to the number of samples contributing to the final ray color, however the increased power of GPUs has made it more viable for use in real time applications with larger volumetric data [113].

3.3 Viewing interaction in virtual reality

A popular approach to designing user interface interactions is to rely on “metaphors” [14]: an analogue typically of interactions in the real world that can be used as a basis for understanding the interaction. An example of a metaphor in a 2D interface would be the use of buttons — relying on the knowledge that buttons are intended to be pressed — or a slider resembling that on a sound mixer.

Virtual reality and 3D interaction tools allow for a much wider range of metaphors than that feasible with a keyboard, mouse and monitor. They also open the possibility for metaphors to be more direct: for example, the user can pick up an object by reaching out to touch it object and squeeze the controller to hold it.

While virtual reality affords more freedom in choosing metaphors, they are still limited by the interaction devices being used. Most virtual reality systems provide controllers. A controller is a device that is carried in the user’s hands and is used to interface with the application. Those designed for virtual reality typically have their position and orientation relative to the user tracked and displayed within the simulation. The controllers used by the HTC Vive and Oculus Rift VR systems, are among the most accessible options, but they can’t capture accurate finger or hand positions, nor provide the user with more than crude physical feedback such as vibration. Targeting consumer VR devices at this time excludes metaphors that rely on finger or hand dexterity, such as rotating a pen or small object in your hand using your fingers.

Fundamental operations for viewing virtual models are rotation, translation and scaling. Most head mounted displays for virtual reality, including the HTC Vive and Oculus Rift experimented with in this research, can track the head position and orientation as it moves through virtual space, offering a natural means of effectively applying the inverse of these transformations. However, due to space restrictions and ease of use, it is often beneficial to transform the model directly as well. There is an abundance of past work outlining

interaction methods and metaphors that can be used to achieve these effects under the limitations imposed by using a controller.

One-Hand Grab with Two-Hand Scaling [68] utilizes the metaphor of an object being held in the user's hand. Any translation or rotation applied to the controller while the object is being grabbed is applied directly to the object as well. Scaling the object requires the use of two hands, and modifies the object's scale in proportion to the change in distance between the hands.

Scaled-world Grab [72] utilizes a perspective illusion to seamlessly interact with far away objects. When an object in a scene is interacted with, the entire scene is scaled relative to the user's head placing the object within arm's reach. Manipulation is handled similarly to One-Hand Grab with Two-Hand Scaling. Once the object has finished being manipulated, the scene is returned to its original scale. Due to the scaling transformations being relative to the user's head, they occur almost imperceptibly, with the only noticeable effect occurring from change in parallax between the eyes.

The 3-DOF hand method [71] separates the rotation and translation functionality between the two hands. In this method, one hand rotates and the other translates; the motivation being to reduce the chance of accidental rotations.

The Handle-bar metaphor simulates holding a handle-bar connected to the scene with both hands. Rotations, translations and scalings of the object are represented by the rotation, translation and scaling of the line between the controllers. This metaphor was introduced by Song et al. [96], and extended in the Spindle and Wheel metaphor [18] by also simulating rotation around the axis of the handlebar.

Chapter 4

Design

ViNE was designed with two main goals in mind: it should be easy to use, and intuitive to learn — two traits which frequently correlate. Not only does ViNE have to compete against the numerous new tools introduced for segmentation and analysis, but also the established tools and usage patterns that researchers have grown familiar with over their careers. An interface that is quick to learn, and easy to use can more quickly demonstrate advantages over more familiar methods.

Virtual reality helps achieve both of these goals. Virtual reality allows metaphors to be used that more closely mimic real life, which can reduce the time required to learn the interface, and allow integral tasks such as viewing a dataset from different perspectives to be achieved in a similar manner as with a physical object.

Of the many available virtual reality systems, those incorporating smartphones — such as Google Cardboard — are the most accessible. However they lack the processing speed and memory to handle large datasets, and the absence of controllers limits the types of interactions they can support. For this reason, I have targeted desktop virtual reality systems such as the HTC Vive and Oculus Rift. These systems utilize trackable controllers, and can support larger amounts of memory and compute power through the connected desktop computer.

I use the OpenVR [104] API developed by Valve Software to interface between the desktop application and virtual reality system. OpenVR supports a variety of different devices, including the HTC Vive and Oculus Rift, allowing multiple virtual reality systems to be used without needing a different interface for each. While I have included support for other devices, my design has been focused primarily on the HTC Vive, and the interface presented

in this thesis is that which was designed for that platform.

OpenVR is compatible with graphics APIs OpenGL, DirectX and Vulkan, as well as being supported in rendering engines such as Unity and Unreal Engine. While Unity and Unreal Engine can streamline the development process, using graphics APIs directly gives developers more control over performance. To this end, I implemented ViNE in C++ with OpenGL.

As a reference object for design and performance, a scan of a gerbera flower head was used. This scan was performed at the Canadian Light Source using synchrotron-based phase contrast imaging. The gerbera model is 14 GB with dimensions of 3248x3248x1420. Two other scans have featured in this thesis: Helianthus which had dimensions of 2066x2148x1348, and Craspedia which had dimensions of 945x1982x1954.

All data used in this thesis was scanned at the Canadian Light Source using the synchrotron-based phase contrast imaging on beamline 05B1-1 [1][111].

In Usher et al.. [102], using volumetric rendering techniques allowed for at most 256x256x256 sections of their data to be viewed at a time. This limitation makes it difficult to interpret how parts of the dataset fit in a wider context, and while it may not significantly affect segmenting, it would hinder the analysis of the data. To allow for faster rendering of the image stacks, I chose to render the data as a triangular mesh instead, similar to Leggio et al.. [56]. To convert an image stack into a triangular mesh, I first find an isovalue in the image that captures as much of the vasculature as possible while limiting the surrounding plant tissue. Next, I use OpenVDB's [29] marching cubes implementation to create a triangular mesh freom the isosurface. This may require downsampling of the image stack to keep a manageable size of mesh. For the gerbera model, the resulting mesh had 18 million vertices and 36 million faces after being downsampled by a factor of two along each dimension.

ViNE's primary means of interacting with the model is through painting. This can

[1] Data was obtained through the efforts of Teng Zhang, Mik Cieslak, Paula Elomaa, Przemysław Prusinkiewicz, and CLS researchers Jarvis Stobbs and Chitra Karunakaran

be used for the purpose of tagging points of interest, as well as segmenting the model's components by assigning them distinct colors. Like Leggio et al.. [56], I also provide an interface for selectively hiding segmented components.

As being in virtual reality is normally a solitary experience, and conveying the experience to others can be difficult, I introduced a function that saves the user's point of view, allowing others to view the model from that perspective.

Chapter 5

Implementation

5.1 3D manipulation

Room-scale virtual reality, provided by many virtual reality devices, allows the user to walk around a model and view it from different perspectives. While this is useful for analyzing a model, it is limited by the inability to change scale and the size of the walking space; furthermore, the optimal viewing angle may occur in uncomfortable or impossible positions, such as lying on the ground. To address these shortcomings, I have provided controls within ViNE for translating, rotating and scaling the object.

5.1.1 Design

To implement these transformations, I have selected a modified version of the Spindle and Wheel method [18]. This approach attempts to emulate the user grabbing a handle-bar embedded in the scene, and manipulates the scene relative to how the handle-bar moves.

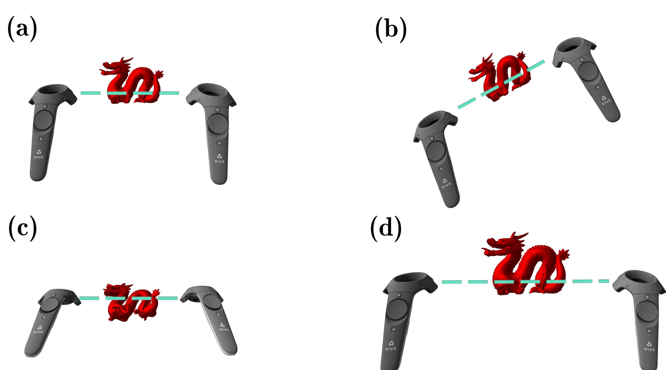


Figure 5.1: Spindle and Wheel transformations. (a) Initial configuration; (b) Yaw/Roll; (c) Pitch; (d) Scaling

A controller is enabled to transform the scene when the “grip” button is pressed. When both grip buttons are pressed, the behaviour follows that outlined in Figure 5.1: modeling the handle-bar metaphor and providing means for translation, rotation and scaling.

If only one grip button is pressed, the controller’s movement will only result in translations. Translations are the most commonly used transformation, and it is difficult to just apply a translation without unintended rotation or scaling through the Spindle and Wheel approach.

The implementation of the handle-bar metaphor largely follows that outlined by Cho et al[18]. Yaw and roll rotations, relative to the handle-bar lying along the x-axis, are performed by changing the direction of the line between the controllers. Changing the distance between the controllers while the grip buttons are pressed changes the scale of the object proportionally.

I introduced a small modification with respect to pitch transformation: the rotation applied is equal to the average change in pitch, as opposed to the change in pitch of the dominant hand. Introducing the concept of a dominant hand which is only relevant to one form of rotation is counter-intuitive, and can confuse users accidentally trying to make pitch rotations with the non-dominant hand. In contrast, when pitch is controlled by the average orientation, users of ViNE have effectively utilized the feature even without having it described to them.

A further modification I have made concerns translation. Bringing a far away object closer can be tedious, requiring many cycles of grabbing it and letting go; analogous to dragging an unsliding object towards yourself with a rope. Adding quickly decaying momentum to the object after it had been let go allowed this interaction to become much smoother. This was attempted with rotation as well, however when the model was large it created the impression of the room spinning causing several users to express discomfort from experiencing nausea.

5.1.2 Translation

While an object is being grabbed, a translation is applied equal to change in position of the controller grabbing it.

After the grip button has been released, the object retains its momentum. To calculate the momentum at this moment, we need to calculate the controller's velocity. The average velocity of the controller between frames i and $i-1$ is approximated using a finite difference:

$$\boldsymbol{v}_i = \frac{\mathrm{X}_i - \mathrm{X}_{i-1}}{\Delta t}\,, \tag{5.1}$$

where $\boldsymbol{v}_i$ is the velocity, X_i is the position at time i, and Δt is the time between frames i and $i-1$. Using this formula can result, however, in the object "flying off" on unintended trajectories due to precision issues with estimating the controller's position over short distances. To address this shortcoming, I instead use a smoothed velocity, $\bar{\boldsymbol{v}}_i$, calculated as the average velocity over N frames:

$$\bar{\boldsymbol{v}}_i = \frac{\mathrm{X}_i - \mathrm{X}_{i-N}}{N\Delta t}\,. \tag{5.2}$$

For the purpose of the application, $N = 5$ was sufficient to make the momentum more stable.

To allow the momentum to quickly decay, I set the velocity of the object in subsequent frames to be:

$$\boldsymbol{v}_i = a\boldsymbol{v}_{i-1}\,, \tag{5.3}$$

where a is set to 0.98 in the application.

5.1.3 Rotation

Orientations of all entities in ViNE, including models and virtual reality devices, are stored as quaternions to give a compact and efficiently composable representation.

This 3D viewing method includes two forms of rotation: yaw/roll, as in Figure 5.1(a), and pitch, as in Figure 5.1(b). They are treated separately because the yaw/roll component of the transformation is derived solely from the controller's positions, whereas the pitch around the handle-bar is also dependent on the controllers' orientations.

Yaw and Roll

For the first form of rotation, we need to find the rotation between the line connecting the controllers from one frame to the next. Let L and R be the positions of the left and right controller. The line connecting them at frame i will then be $\overrightarrow{\mathrm{LR}}_i$.

The smallest rotation between this vector in two successive frames is about the axis:

$$\boldsymbol{r}_i = \overrightarrow{\mathrm{LR}}_i \times \overrightarrow{\mathrm{LR}}_{i-1} \,. \tag{5.4}$$

The angle between them can be calculated using the equation:

$$\theta_i = \arccos\left(\frac{\overrightarrow{\mathrm{LR}}_i \cdot \overrightarrow{\mathrm{LR}}_{i-1}}{\left\|\overrightarrow{\mathrm{LR}}_i\right\| \left\|\overrightarrow{\mathrm{LR}}_{i-1}\right\|}\right) \,. \tag{5.5}$$

The rotation for yaw and roll can therefore be represented by the quaternion $[\frac{\theta_i}{2}, \quad \boldsymbol{r_i}]$.

Pitch

To find the pitch rotation, the rotation between the previous controller orientation and the current one is used, represented by the unit quaternions $\boldsymbol{q}_i$ and $\boldsymbol{q}_{i-1}$, with the following equation:

$$\Delta\boldsymbol{q}_i = \boldsymbol{q}_i \boldsymbol{q}_{i-1}^{-1} \,. \tag{5.6}$$

However, we only want to interpret rotations around the $\overrightarrow{\mathrm{LR}}$ axis. There wouldn't be a clear meaning in the handlebar metaphor for the user's hands to rotate in any direction but around the bar. There are many possible ways to map an arbitrary rotation to a change in pitch, but there are two constraints that should be upheld. Let $\Delta\boldsymbol{q}_i = [s_i, \quad \boldsymbol{r}_i]$, and let the

rotation we apply to the object be $\Delta \boldsymbol{q}'_i$. When $\boldsymbol{r}_i$ is perpendicular to $\overrightarrow{\mathrm{LR}}_i$, the rotation $\Delta \boldsymbol{q}'_i$ should be equal to the identity; when it is parallel, $\Delta \boldsymbol{q}'_i$ should be equal to $\Delta \boldsymbol{q}_i$.

A mapping that satisfies these constraints weights the amount of rotation by the dot product between $\boldsymbol{r}_i$ and $\overrightarrow{\mathrm{LR}}_i$ using the following formula:

$$\begin{aligned} \theta &= \arccos(s_i) \\ w &= \frac{\boldsymbol{r} \cdot \overrightarrow{\mathrm{LR}}_i}{\|\boldsymbol{r}_i\| \left\|\overrightarrow{\mathrm{LR}}_i\right\|} \\ \Delta \boldsymbol{q}'_i &= \left[\cos(w\theta), \quad \sin(w\theta) \frac{\overrightarrow{\mathrm{LR}}_i}{\left\|\overrightarrow{\mathrm{LR}}_i\right\|}\right] \end{aligned} \tag{5.7}$$

This equation smoothly blends between the two constraints. The final rotation used for the roll interaction is the average of the $\Delta \boldsymbol{q}'_i$ calculated for each controller.

5.2 Depth cueing

While being able to view the model in virtual reality aids in depth perception, it can still be difficult to distinguish depth at times, especially in the case of particularly complicated vasculature or when viewing parts of the mesh which only recieve uniform ambient lighting. To address this, I've explored using depth cueing to enhance the perception of depth.

5.2.1 Design

Depth cueing emphasizes differences in depth by making objects darker the farther they are away. This results in an effect that looks similar to the object being illuminated by a light near the viewer. In this model, a monotonically increasing function f of distance to the shaded element, d, with a range of $(0, p]$, for $p \in [0, 1]$, is used to blend between the shaded color and the background color.

$$C_{final}(d) = (1 - f(d))C_{init} + f(d)C_{background}\,, \tag{5.8}$$

Setting p to be 1 results in some parts of the model completely disappearing into the background. I experimented with different values of p to ensure that the whole model is visible while maintaining as much contrast as possible, finding a value of 0.85 struck an effective compromise.

Using a fixed function f that does not consider the scale and distance of the mesh behaves poorly in a number of configurations. At the ideal scale and distance to the user different depths are easily distinguishable. If the scale or distance is much smaller than the ideal, only a small range of the color gradient will be used. If the scale or distance is much larger, most of the model will appear close to the background color; not only losing the benefit of depth cueing, but also much of the lighting or color information. These configurations can be seen in Figure 5.2.

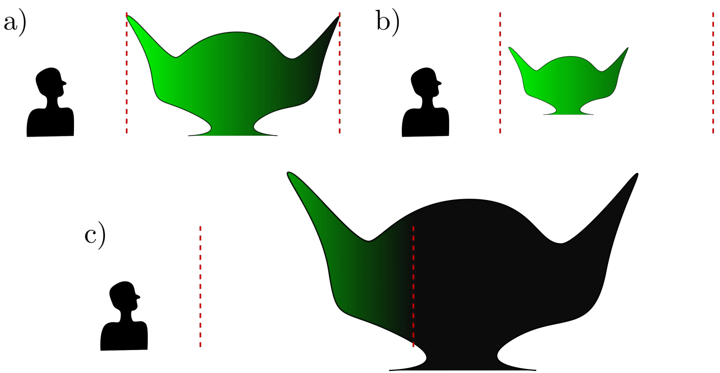


Figure 5.2: Three depth cueing scenarios with fixed gradient based on distance from viewer. Red lines indicate the beginning and end of the depth cueing gradient. (a) The model is at the ideal scale and distance: depth cueing creates noticeable contrast across the model. (b) The model is too small and close: depth cueing has little effect on the original color. (c) The model is too large and far away: most of the model has faded into the background, and the parts that haven't are still dark.

A better method would utilize the full color gradient for depth cueing in the majority of scenarios, not just when the model is at the right scale and distance. A method that offers

clearest perception of depth would have the furthest visible portion of the model be colored as close to the background color as desired, and the closest visible portion of the model to be colored normally. However, while this may be ideal for a static image, it would result in rapidly changing colors as new closest or furthest parts of the model appear or disappear from view, as can be seen in Figure 5.3 (a) and (b).

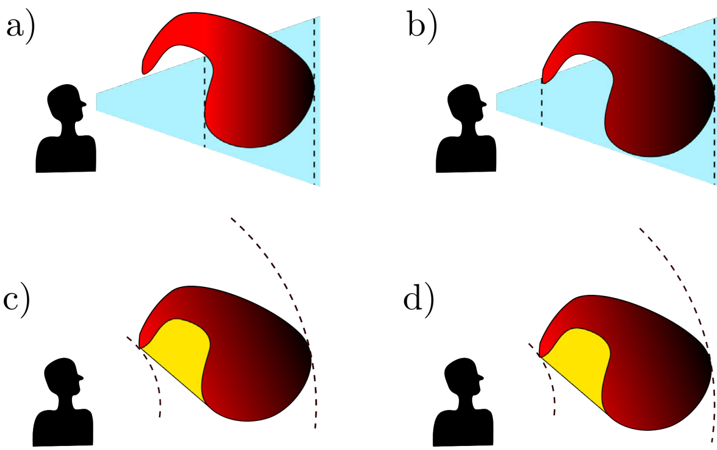


Figure 5.3: The blue trapezoid indicates the view frustum, and the dotted lines indicate the closest and furthest points within it. (a) and (b) use the distance to the red concave object to determine the range of the depth cueing. (c) and (d) use the distance to the yellow convex hull. The change in depth cueing between (a) and (b) is significant, while the change between (c) and (d) is slight.

We want a method that uses the optimal range of the depth cueing gradient, while also being as temporally consistent as possible. If the distance to the model is used instead, this will not change with the viewpoint. While this comes at the cost of utilizing less of the depth cueing gradient in some situations I consider this an acceptable tradeoff for temporal continuity.

Measuring the distance to the model can be expensive, possibly requiring millions of distance calculations even if just measuring distance to its vertices. I explored using bounding

volumes to reduce this cost.

Using a bounding sphere as a bounding volume utilized a small portion of the depth cueing gradient in situations where the sphere made a poor approximation for the shape of the model. Disk and capsule-shaped objects had particularly poor contrast across their smallest axes.

A convex hull can better conform to the shape of an object than other bounding volumes such as spheres and bounding boxes, and approximates disk-like and capsule-like shapes well. Tests for the closest and furthest distance to a convex hull are much expensive than for a sphere, but we can accelerate them by using an approximate convex hull.

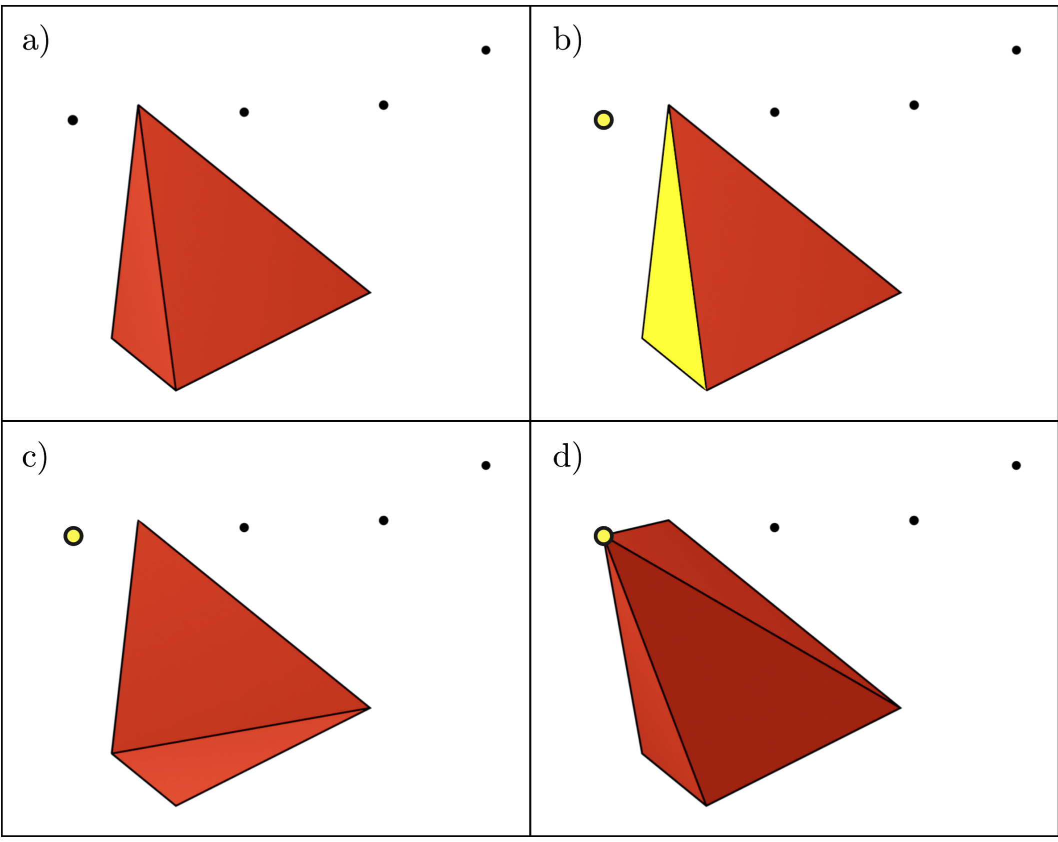


Figure 5.4: An iteration of the Quickhull algorithm. (a) The geometry at the end of the previous iteration. (b) Randomly select a face, and the furthest point in the direction of its normal. (c) Remove all faces for which the point lies in the direction of their normal. (d) Connect boundary edges to new point using a triangle fan.

The algorithm I chose to compute the convex hull is the implementation by Barber et al. [3] of the Quickhull Algorithm [21]. This algorithm iteratively converges on the convex hull of a set of points. It begins with a tetrahedron created from four randomly points in the model chosen such that no triangle is degenerate. On each iteration of the algorithm, a triangle is randomly selected. The points that are currently outside of the convex hull are then iterated over, and the point which has the largest dot product with the triangle's normal is added to the convex hull. The triangles on the convex hull are then iterated over. If the vector between a point on the triangle and the new point lies in the direction of the triangle's normal, that triangle is removed from the mesh. A ring of boundary edges will be left over, and the new point is connected to the convex hull using a triangle fan around this ring. This process is visualized in Figure 5.4.

In my experimentation with several different meshes I found that the algorithm can be stopped fairly early and produce a coarse representation of the convex hull. The reduced vertex count compared to the original model helps accelerate the computation of the distance from the user to the convex hull. As an example, the gerbera model has over 18 million vertices, but a convex hull with only 274 is close enough not to notice a difference during use of the program.

5.2.2 Method

Before rendering, the closest and furthest distances from the user's position in virtual space to the convex hull are calculated.

I treat the user's viewing position as the average position of their two eyes, E. When E is inside the convex hull, the distance to the closest point on it is 0. To test if the user is inside the convex hull, I trace a ray in an arbitrary direction, and count the number of intersections that occur between it and the triangles of the convex hull.

An intersection between the ray and a triangle can be calculated by solving the following equation [94]:

$$\mathrm{E} + t\hat{\boldsymbol{d}} = \mathrm{A} + \beta\overrightarrow{\mathrm{AB}} + \gamma\overrightarrow{\mathrm{AC}}\,, \tag{5.9}$$

where A, B and C are the triangle's vertices, t is the distance along the ray, $\hat{\boldsymbol{d}}$ is the ray's direction, and β and γ are the barycentric coordinates of A and B. This equation finds the intersection between the ray and triangle's plane, expressing it in terms of the triangle's barycentric coordinates and the progress along the ray [94]. Equation 5.9 can be rewritten as the linear system:

$$\begin{bmatrix} \overrightarrow{\mathrm{BA}}_x & \overrightarrow{\mathrm{CA}}_x & \hat{\boldsymbol{d}}_x \\ \overrightarrow{\mathrm{BA}}_y & \overrightarrow{\mathrm{CA}}_y & \hat{\boldsymbol{d}}_y \\ \overrightarrow{\mathrm{BA}}_z & \overrightarrow{\mathrm{CA}}_z & \hat{\boldsymbol{d}}_z \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \overrightarrow{\mathrm{EA}}_x \\ \overrightarrow{\mathrm{EA}}_y \\ \overrightarrow{\mathrm{EA}}_z \end{bmatrix}\,, \tag{5.10}$$

and solved using inversion or Cramer's rule [94]. The ray intersects with a triangle if t, γ and β are all positive, and $\beta + \gamma < 1$.

If the number of intersections is 0 or 2, E is outside of the convex hull, as the ray either never intersected, or it entered and left. If the number is 1, then E is inside the convex hull, with the one intersection causing the ray to leave. The number of intersections cannot be greater than 2 for a convex object.

If E is not inside the convex hull, the closest point to it lies on the convex hull's surface. I break this problem up by finding the closest point to E on each triangle of the convex hull generated by Quickhull, and returning the closest of those points.

To find the closest point on an individual triangle, I first calculate the barycentric coordinates of the E projected onto the plane of that triangle, E_p. The barycentric coordinates of E_p can be solved for by using the triangle's normal as the ray direction, $\hat{\boldsymbol{d}}$, in Equation 5.10. The projected point can be expressed using the barycentric coordinates: $\mathrm{E}_p = \mathrm{A} + \beta\overrightarrow{\mathrm{AB}} + \gamma\overrightarrow{\mathrm{AC}}$.

If E_p lies within the triangle, it is the closest point to E. If it does not, I find the closest point to E_p on each of the triangle's edges, and use the closest of those points instead. The closest point on an edge can be found by projecting E_p onto the edge and clamping it at its vertices, as illustrated in Figure 5.5.

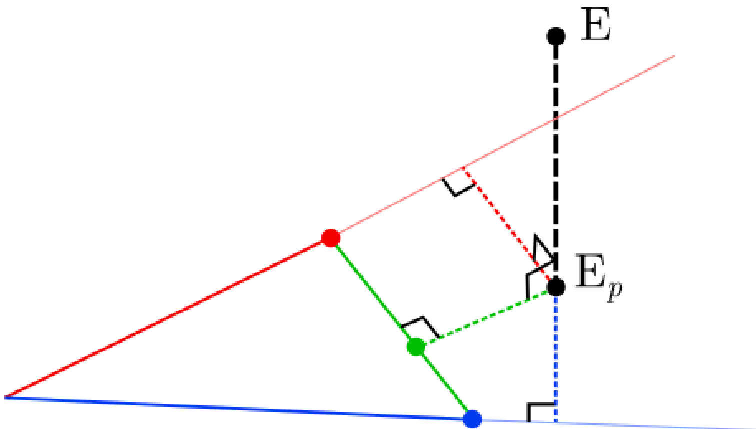


Figure 5.5: Finding the closest point on a triangle. The closest point to each edge is shown using the same color. The closest point to the triangle will be the closest of these three to E.

Calculating the furthest point is simpler, and only requires iterating over vertices, as the furthest point will always be a vertex. To justify this with a short sketch of a proof by contradiction, suppose that the furthest point lay somewhere on the triangle that was not a vertex. Consider the closest edge to this point. Moving parallel to this edge in at least one direction will result in a further point from E, contradicting our assumption.

Through experimentation, I found the best results to occur when the depth cueing is calculated using an exponential function; a simplified model of how fog behaves in the real world. Let d be the distance to the fragment, d_c be the distance to the closest point on the convex hull, d_f be the distance to the furthest point, and s be a parameter to modulate how quickly the depth cueing increases. The following function is used in Equation 5.8 to modify the color of a fragment after shading

$$f(d) = e^{s\frac{d-d_c}{d_f-d_c}}. \tag{5.11}$$

5.2.3 Results

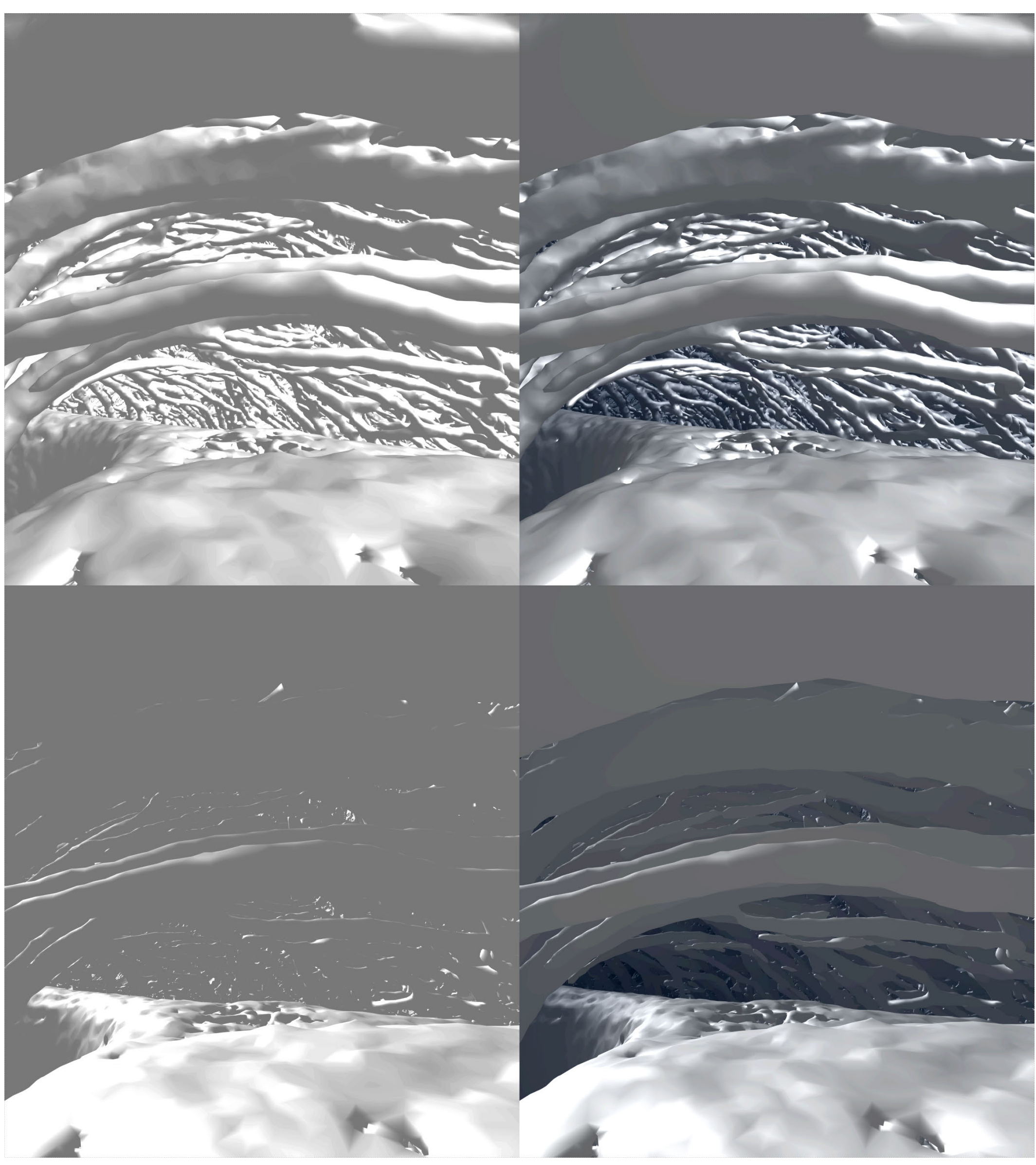


Figure 5.6: Comparison of depth cueing turned off and on. Left side: No depth cueing. Right side: Depth cueing. Top: Ideal lighting angle for visibility. Bottom: Poor lighting angle, only ambient shading visible.

Figure 5.6 shows comparison of depth cueing under well lit and poorly lit conditions. Depth cueing has the most impact in situations when the user is viewing the parts of the model that don't receive direct illumination. Without depth cueing there is little to distinguish different parts of the mesh, forcing the user to rotate the model for better lighting conditions. Under well lit conditions, depth cueing has a smaller effect, but having less emphasis on the vasculature further away helps reduce clutter.

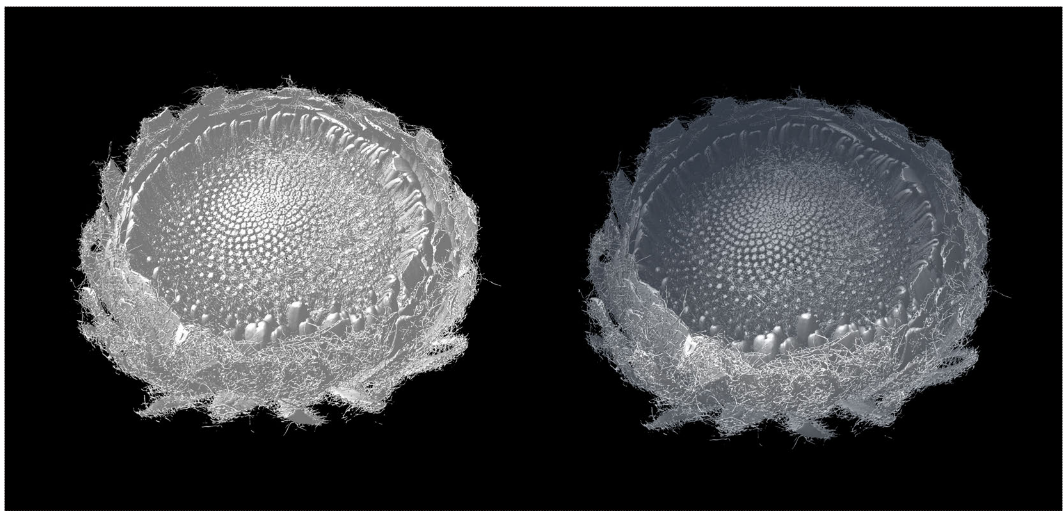


Figure 5.7: View of the outside of the flower head. Left: No depth cueing. Right: Depth cueing.

When viewing the outside of the mesh as in Figure 5.7, it becomes arguable whether a benefit exists. From this view there are few nearby structures that differ significantly in depth, and the depth cueing darkens more uniformly around areas of the screen.

User response to this feature was mixed. Some users found the manner in which the depth cueing changed to be somewhat unintuitive. The mental model of further away objects darkening conflicts with the way in which the closest and furthest distances change as the distance between the user and objects changes; an object being brought closer will not cause it become brighter, while the user may expect it to.

I attempted using image-space horizon-based ambient occlusion [5] as an alternate means

of adding contrast when the mesh is viewed at angles without direct lighting (Figure 5.6) bottom left). While it helped to distinguish veins under these conditions, it also increased rendering time by too large of a factor to maintain the necessary frame rate.

5.3 View saving

When users analyzed flower heads with early versions of ViNE, collaborating with others was a common difficulty. While an external monitor displayed what the user saw, when they wanted to share the experience with someone else they needed to give them the headset. During this transfer, it was difficult for the next user to place themselves in the same position and orientation, complicating the process of finding the features or perspective the original user intended to show. This was also a problem when a single user wanted to find the same features or perspectives again for future sessions.

5.3.1 Design

With these challenges in mind, I've provided a means to save and load "viewpoints". When the right trackpad is pressed, the current view is saved. Saved views can be loaded using the keyboard.

While the saved view could match the head's orientation exactly, when a new user puts on the HMD and loads that view, their head won't necessarily be in the same orientation. If the second user's head is tilted to the side relative to the first user's, a view that was intended to present a flower's base horizontal to the ground would end up tilted to the side instead. When a user is in virtual reality, they have a concept of which direction up is in regardless of how their head is oriented, which is not preserved when saving or loading a view using this method.

To address this, I only consider a portion of the head's orientation, specifically the horizontal direction the user is facing. By limiting the orientation to the viewer's horizontal

facing, the perceived direction of up is preserved regardless of the tilt or inclination of the head when saving or loading.

A drawback to the implementation I've chosen is that the object of focus will not always be directly within the user's view when loaded. However, with our goal of sharing the experience of being where the user was when the view was saved, this will be better matched by preserving the direction of vertical. This also allows the user who saved the view to provide the same directional cues they would have when they were wearing the headset, such as "look up" or "look right".

5.3.2 Method

I describe this section using matrices instead of quaternions, as I have in the rest of my thesis, due to how ubiquitous they are for coordinate frame transformations.

To save the view, I describe the position and orientation of the mesh with respect to the view direction projected onto the horizontal plane, $\hat{\boldsymbol{d}}$, and the position of the viewer, V To do this I transform the mesh into a "saved view space" space, where $\hat{\boldsymbol{d}}$ is directed down the $-\hat{\boldsymbol{z}}$ axis. The rotation component of this transformation is described by the following rotation around the $\hat{\boldsymbol{y}}$ axis

$$\theta = \text{atan2}(\hat{\boldsymbol{d}}_x, \hat{\boldsymbol{d}}_z)$$

$$R_{view} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{5.12}$$

while the translation component of this transformation is the translation matrix that takes V to the origin.

To transform the mesh into saved view space, I use the following equations:

$$\mathrm{P}_{saved} = T_{view}R_{view}\begin{bmatrix}\mathrm{P}\\1\end{bmatrix}\mathrm{R}_{saved} = R_{view}R_{mesh}\,, \tag{5.13}$$

where P is the mesh's position, and R_{mesh} is its orientation. P_{saved} and R_{saved} are saved in a file to describe the viewpoint.

To load the view, I first calculate R_{view} and T_{view} for the new horizontal view direction using Equation 5.12 and **??**. I then use the inverse transformation to place the model in world space relative to the viewer:

$$\mathrm{P} = (T_{view}R_{view})^{-1}\begin{bmatrix}\mathrm{P}_{saved}\\1\end{bmatrix}\mathrm{R}_{mesh} = R^{-1}_{view}R_{saved}\,. \tag{5.14}$$

5.4 Painting

In ViNE, painting is one of the main methods of interaction with the polygonal mesh generated from the image stack. One of the main purposes of painting is to segment and annotate data stacks. In the interest of this, the painting functionality has been designed with a focus on helping to distinguish or draw associations between different parts of the mesh.

5.4.1 User interface

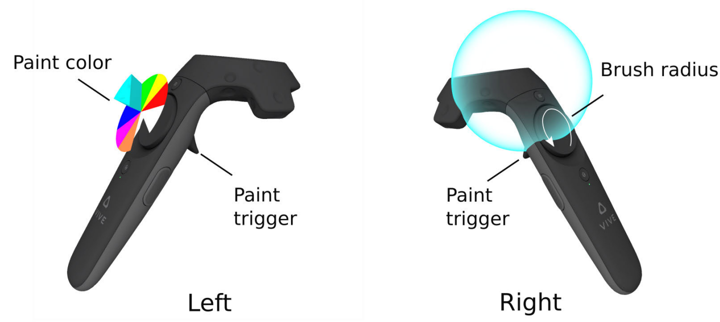


Figure 5.8: Painting controls for the left and right controllers

The brush, represented as a partially transparent sphere around the controller, indicates the region within which the mesh can be painted and the color which will be painted with. The triggers on the HTC Vive controllers register the degree to which they have been pressed. A full press of the trigger is required to paint with the brush. A partial press will display the sphere, but not paint within it. This allows the user to preview the region that they will paint.

The user can change the radius of the brush by making circles with their thumb on the right trackpad. Motion in a clockwise direction makes the brush larger, while a counter-clockwise direction makes it smaller, with a full rotation in either direction doubling or halving the radius.

The color of the brush is chosen with the color wheel assigned to the left controller. The color wheel is displayed on top of the left trackpad, scaled up to 1.5 times the size of the trackpad to help improve visibility. The thumb's position on the trackpad is mapped directly to the color wheel, leaving a small depression in the color wheel where the thumb is located. The active color is set as the color of the slice the thumb lies within, indicated by having that slice extruded vertically. While the thumb's location on the trackpad can be estimated to an extent by the extruded wedge, it becomes much more ambiguous when the thumb is

near the center. Adding the depression to explicitly indicate the thumb's position solved this problem.

Figure 5.9: Color wheel for color selection. The currently selected color slice is extruded upwards, and an indentation is left underneath the thumb's position.

5.4.2 Color storage on the mesh

Segmenting and tagging operations are achieved by coloring the mesh. There are three possible methods that could potentially be used to color the mesh: setting a color per-pixel in textures mapped to the mesh, per-face, or per-vertex.

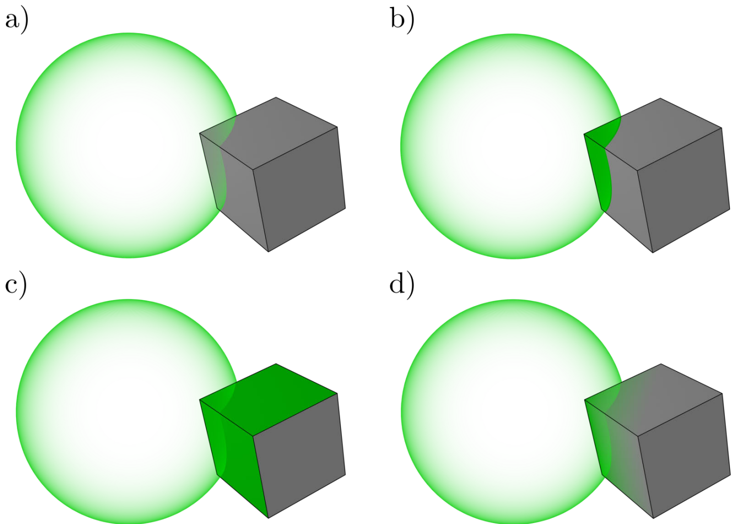


Figure 5.10: Three options for color storage on a mesh. (a) Unpainted (b) Per-pixel (c) Per-face (d) Per-vertex.

Storing colors in a texture yields the most accurate mapping of the brush onto the mesh. Adjusting the resolution of the texture allows the brush to be arbitrarily precise in regards to how much of the face is painted. This is particularly beneficial when segmenting a mesh which has been zoomed in on to a large extent; portions of large triangles could be filled in to whatever degree desired. On the other hand, this requires a large amount of data to be stored, which would not only impact memory, but data transfer time from the CPU to GPU. To provide a rough lower bound on the memory usage, if we assume that each triangle spans at least 3 texels the absolute minimum amount of memory required would be 3 times the number of faces, as well as at least two floating point numbers for each of the vertices for the texture coordinates. In addition, the operation of identifying the texels to be painted is the most computationally expensive of the three options, requiring each texel mapped to a face to be tested for intersection against the brush.

Storing colors on a per-face basis requires less memory than per-pixel. The brush-to-face intersection test is also less computationally expensive than in the uv mapped texture approach, while still being more expensive than per-vertex. The biggest drawback with

storing information per-face is the poor support that OpenGL provides for this type of data association. OpenGL is primarily designed to assign data per-vertex via vertex attribute arrays, or per draw call via uniforms. The most common approach to store per-face information in OpenGL is to forgo indexed drawing and use vertex attribute arrays to duplicate the data for every vertex on the face. All of the data associated with each vertex would be duplicated a number of times equal to its valence: for an typical mesh this would fall in the range of 3-6. Due to the size of the meshes ViNE is intended to work with, this requires a substantial amount of extra memory.

Storing colors per-vertex has the lowest memory cost, cheapest intersection test — a straight distance calculation — and can be accelerated easily and efficiently with spatial subdivision data structures. The biggest disadvantage occurs when the brush is intersecting with the mesh, but does not contain any vertices, resulting in nothing being painted. This scenario becomes more common when interacting with large faces, such as on a mesh zoomed in on closely.

While each of these options have their own advantages, I chose to store colors per-vertex. What improvement to the user-experience the other options may offer is offset by the performance cost they incur. As will be discussed in section 5.7, to work with the size of data ViNE is intended to, the memory usage of the color information must be as small as possible. The drawback that would occur when painting large triangles is also not as significant as it could be given that the preferred method of mesh generation, marching cubes, will always create similarly sized triangles and in practice it has been rare for a user to interact with the mesh at a scale where this becomes a problem.

To further reduce the memory cost, I've chosen to store the color in the form of a one byte index into a color table. Compared to storing it as RGB, a byte index reduces the amount of data to store, and modify, by two thirds. In addition, this allows colors to be easily reassigned by adjusting the map on the color table.

5.4.3 Brush intersection

Given that the brush is spherical, determining which vertices lie inside of it can be expressed as a distance calculation:

$$\|\mathrm{C} - \mathrm{V}\| \leq R\,, \tag{5.15}$$

for a controller position of C, vertex position V and brush radius of R. The square root for the length calculation can be cut by instead evaluating:

$$(\mathrm{C} - \mathrm{V}) \cdot (\mathrm{C} - \mathrm{V}) \leq R^2\,. \tag{5.16}$$

Due to the number of vertices in meshes used in ViNE, determining which points are within the brush radius can be computationally expensive, even with a complexity of $O(n)$. Using a spatial subdivision data structure can reduce the number of distance calculations required. I have chosen to use a K-D tree[1].

Storage requirements for the K-D tree are minimized by creating the KD-Tree in-place — using no extra space on top of the data stored in it — using Algorithm 5.1. Each recursive invocation of the function finds a pivot in the middle of its range, such that every point for which the dth coordinate is less than that of the pivot is on the left side, and everything with a dth coordinate greater than or equal to it is on the right side. This function is then called recursively on each half of the array, with the next dimension compared instead.

In the algorithm, NTH_ELEMENT is the C++ standard library function which sorts a range of elements such that the index in the second argument contains what that element would have been in a fully sorted array, and all elements smaller appear to the left of it, while those larger appear to the right.

[1] Code courtesy of Andrew Owens

Algorithm 5.1 Create KD-tree

```
1: procedure CreateKDTree(vertices, d, start, end)
2:     size ← end − start
3:     if size ≤ 1 then return
4:     end if
5:     mid ← start + size/2
6:     procedure CompareDimension(a, b)
7:         return a[d] < b[d]                 ▷ Compare coordinates for the dth dimension
8:     end procedure
9:     nth_element(start, mid, end, CompareDimension)
10:    d ← (d + 1)  mod 3                     ▷ Change to the next dimension
11:    CreateKDTree(vertices, d, start, mid)
12:    CreateKDTree(vertices, d, mid + 1, end)
13: end procedure
```

To find the points that are within a given radius, for each "level" of the K-D tree, the distance along that level's dimension to the pivot is computed. If the distance is greater than R, the points beyond it can be ignored, and only the other side is recursively tested.

5.5 Undo and redo

Despite having the ability to preview the areas to be painted, the user is likely to make errors during the segmentation process. Many use cases require precise painting to avoid nearby structures, and an accidental press of the paint trigger can overwrite a large amount of work.

The undo or redo functionality allows the user to segment more freely without having to worry about possible mistakes. The undo implemented in ViNE reverses any changes between when the painting trigger is first pressed and when it is released, which I shall refer to as a painting operation. In addition, it is a multi-level undo, allowing the user to undo the last 20 painting operations.

Given the large size of the data stacks, and of the 3D triangular meshes generated from them, it is important to efficiently store the changes made while painting. The less memory used to store each change, the more past states can be stored for the multi-level undo.

There are several ways the changes made during a painting operation could have been tracked. Storing the color of the entire mesh would be a simple, but impractical solution, as most painting operations will only modify the color of a small subset of the total vertices.

Tracking the index and color change of each vertex painted on each frame of the painting operation is another way to represent the changes made. However, depending on how long the painting operation lasts, this could require even more space than the previous option. Given the short distance that the brush is likely to move between successive frames, there is a high likelihood that the brush will intersect with the previous region that was painted, resulting in duplicated copies of the changes being added from one frame to the next. The less the brush has moved between frames, the more wasteful this would become.

To limit the changes being tracked to a subset of all of the vertices, and remove the possibility of duplication, I have decided to store the changes in C++'s map data structure, which is typically implemented as a binary search tree [99]. The vertex's index within the geometry buffer is used as the key, and its corresponding value stores both the old and new

color, allowing the same map to be used to both undo and redo an action.

In addition to each modified vertex only being tracked once per painting operation, the map data structure automatically sorts the key-value pairs in the order of the keys, the vertex indices. Applying the undo action by iterating over the sorted map of changes allows the array to be written to in sequence, improving cache utilization when compared to modifying the array in a random order.

To implement a multi-level undo, I maintain two stacks: an undo stack storing the map of changes made in painting operations, and a redo stack storing the changes for painting operations that have been undone. When a painting operation is completed, the map of its changes is pushed onto the top of the undo stack, removing the bottom element if the stack's size exceeds the maximum of 20. When the undo button is pressed, the changes at the top of the undo stack are iterated over, reverting each index in the map to its old color. The map of changes is then popped from the undo stack and pushed to the top of the redo stack. When the redo button is pressed, the same procedure is followed, but with the two stacks switched, and the new color being written instead of the old color. The redo stack is cleared when a new painting operation begins.

5.6 Selective visibility

In the analysis of complex structures and data, it is often helpful to selectively show or hide detail, a feature also present in Leggio et al. [56]. Analyzing the venation patterns of a flower head is easier without them being obscured by the surrounding florets and bracts. However the bracts may still add useful context as to where the vasculature exists within the flower head.

In addition, datasets are rarely perfectly clean, as shown in Figure 5.12, and in the process of converting a volume to a marching cubes mesh, it is difficult — and frequently impossible — to find an isovalue that captures only the desired features. Other plant structures and

Figure 5.11: Color wheel indicating the colors made invisible using contrasting radial lines.

noise may also be captured with the isovalue. Noise not only makes viewing the mesh more difficult, but also makes it more challenging to segment it without accidentally capturing the noise.

To address these objectives, I've allowed the visibility of anything that has been painted to be selectively toggled on and off. The user controls this by selecting a color from the trackpad and then pressing down on the trackpad until it clicks, hiding everything that had been segmented with that color. The same method is used to reveal it again.

Selective visibility can be used for the removal of unwanted noise from an data set as in Figure 5.12.

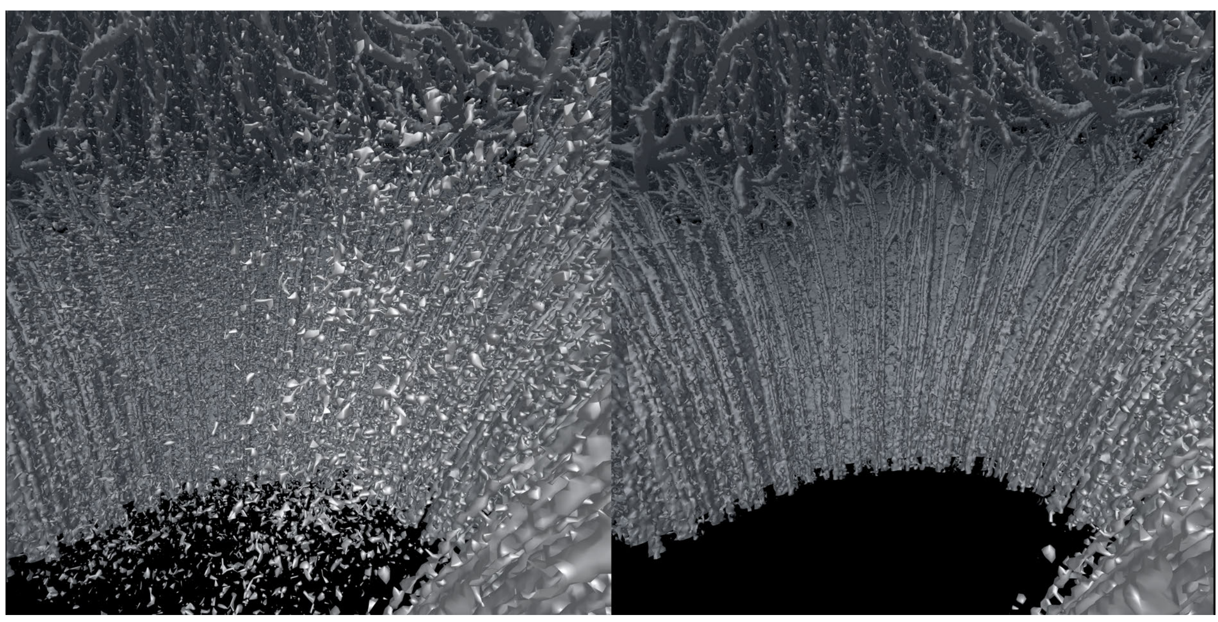


Figure 5.12: Noise in center of flower head removed using selective visibility.

This functionality can also be used to hide or reveal features on a dataset which has had its components segmented using different colors. It can also be useful to isolate a specific vein along with all of its children to better interpret the form of an individual vein. A segmentation applying both is depicted below:

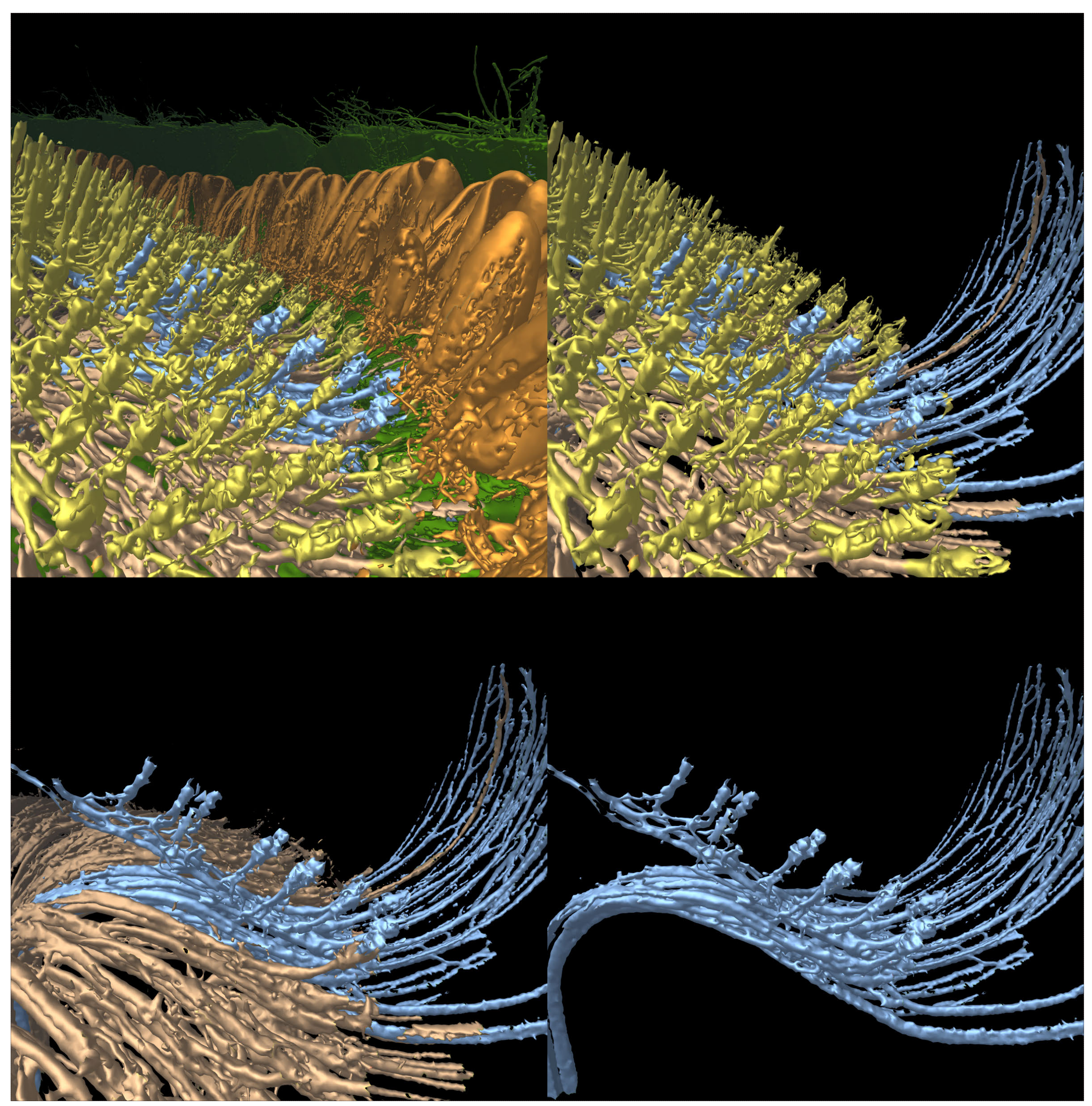


Figure 5.13: A single vein of a gerbera flower head, and its children, segmented apart from the others.

In addition to controlling visualizations, vertices hidden using this feature cannot be painted; which reduces errors by removing the possibility of accidentally segmenting something that has already been correctly segmented.

5.7 Optimization

To keep a consistent point of reference, I use the gerbera dataset for all timing benchmarks in this section. The mesh generated from it has 18 million vertices, and 36 million triangular faces. I have set this as the target number of triangles that ViNE should be able to render at VR frame rates.

Rendering and painting are the most time consuming operations in ViNE by several orders of magnitude. To render the gerbera mesh takes between 14 and 27 milliseconds depending on a number of variables, most notably how much of the mesh is within view. In VR applications, it is important to hit as high frame rates as possible to reduce the degree of motion sickness. While VR applications target 90 frames as the ideal minimum frame rate, features such as asynchronous reprojection [103] provided by OpenVR can reduce the impact of missed frames. It achieves this by warping the previous frame to align with the most recent head position; resulting in a more comfortable experience to users at lower frame rates such as 45 frames per second.

A VR application targeting 90 frames per second requires the entirety of a frame's work to be completed in 11ms, while 45 frames per second requires it to be completed within 22ms. Given the range of rendering times for the gerbera model, the rendering calls need to be initiated as close to the completion of the last as possible to have a chance of fitting into either of these ranges.

5.7.1 Parallelization

OpenVR was designed with a specific pipeline in mind to ensure that the tracking information for the controllers and headset is as up to date as possible, and render calls can complete in time to submit to the headset for the next frame. This pipeline relies on a strategy called a "running start" [107]. In the running start, the device tracking information is obtained 2ms before the headset's Vertical Sync (VSync) — the point at which the frame is presented on

the headset. This tracking information contains the estimated position of the controllers and headset three VSyncs in the future (equivalently 24ms: 2ms to the next VSync and 11ms to each after that) to account for rendering time and transmission time to the headset.

A programmer using OpenVR must call the `WaitGetPoses()` function to access the tracking information for devices, and to accommodate the running start strategy, this function will always wait to return 2ms before the next VSync point. As soon as the tracking information is obtained, it is ideal to immediately start rendering with that new information to ensure that it is as accurate as possible by the time the frame is displayed on the headset. Any additional expensive work done on the thread querying for tracking information and rendering runs the risk of either starting the rendering too late to finish in time, or just missing the synchronization point at 2ms before VSync, causing the function to hold the thread for an entire frame as illustrated in Figure 5.14(a).

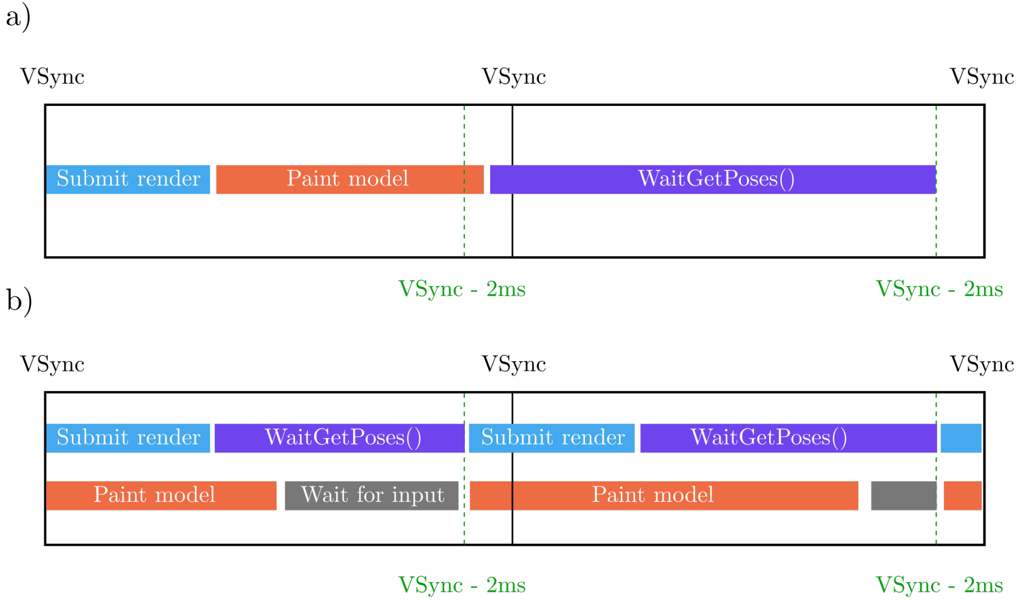


Figure 5.14: Comparison between single and multi-threaded pipleines for OpenVR (a) Single-threaded pipeline in scenario where painting takes long enough that `WaitGetPoses()` misses its frame's synchronization point. (b) Multi-threaded pipeline where rendering and querying tracking information operates on one thread, while painting operates on another. The painting time no longer causes synchronization points to be missed.

For these reasons, I have separated the work load between two threads, one which will query tracking information and initiate render calls, and the other which is responsible for all of the expensive CPU work: notably the painting and undo/redo functionality. This has the added benefit that if a large region is painted and updating the color array spans several frames, the render thread can keep rendering frames to the headset as quickly as they can be rendered using the most recent completed data.

5.7.2 Synchronization of color data

Having the painting and rendering operations occuring on separate threads helps maintain a high and consistent frame rate, but consideration must be given to how the data is

synchronized between the two threads. The painting thread writes changes to a color array, and the rendering thread reads from a color array when it renders the mesh. Often the writing operation is short, but it can span multiple frames when large portions of the mesh are painted. The gerbera mesh has 18 megabytes of color information, which can take over a second to modify. As discussed earlier, rendering operations can take in the range of 14 to 27ms.

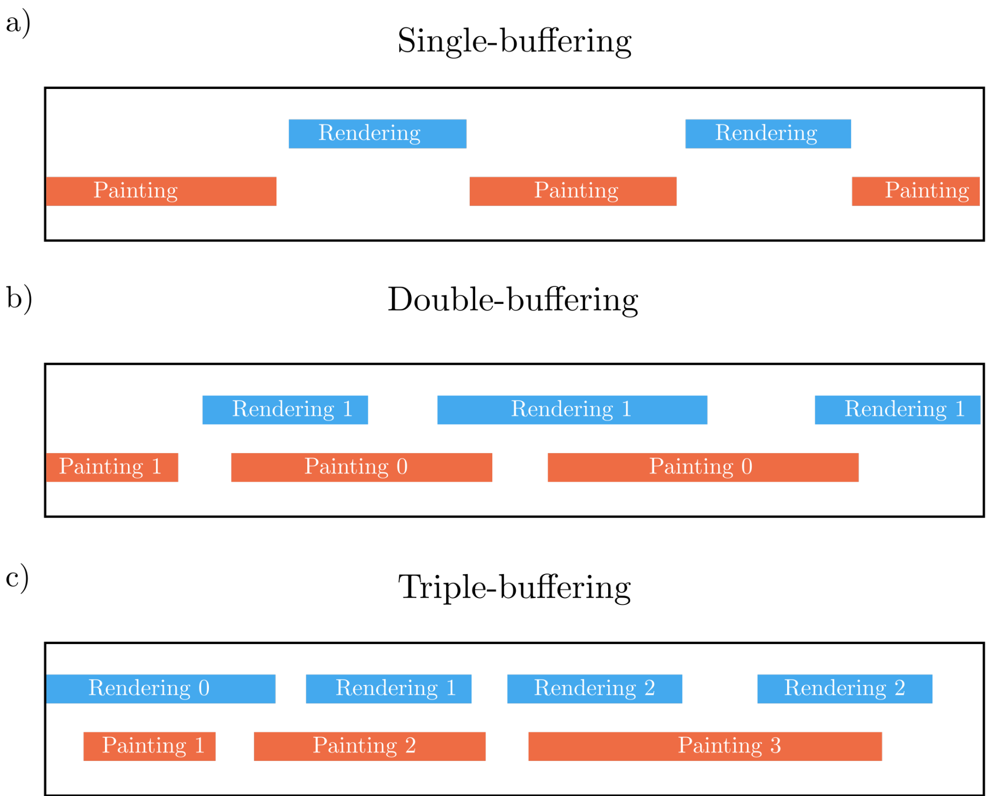


Figure 5.15: Visualization of single, double and triple-buffering for rendering pipelines. (a) Two threads with a single shared buffer. Each thread needs to wait for the other thread to finish using it before it can gain access. (b) Two threads with two shared buffers. While neither thread needs to wait for access, the render thread can become stuck with an older version despite several completed writes. (c) Two threads with three shared buffers. Neither thread needs to wait for access, and the render thread always receives the most recent completed changes.

To avoid race conditions, guards must be put in place to ensure that the rendering thread is never reading from the same memory that the painting thread is writing to as this is undefined behavior. Due to the long duration of both the reading and writing operations, taking turns to access a single buffer of color data would require both threads to spend time idling while the other finishes, defeating the purpose of multi-threading. A double buffer [45, 35] allows both threads to always have access to a buffer when they needs it, but the render thread may lag behind the painting thread, as shown in Figure 5.15.

Using triple-buffering [45, 35], the writing thread always writes to the oldest buffer not currently being read from, and the reading thread always reads from the last buffer that has finished being written to. Not only do reading and writing threads always have immediate access to a buffer, but the most up to date buffer will always be accessible to read. While this does require the largest memory commitment, it only requires 54 megabytes for the gerbera model, which is a small cost for the effect the reduced latency has on the user's experience.

The next consideration is how to upload the changed data to the GPU, which operates on its own thread outside of my direct control. I compared two options, uploading the changes directly using `glBufferSubData`, or allocating the buffer using "pinned memory".

The function `glBufferSubData` requires the calling thread to create a copy of a region of the geometry data that will be uploaded to the GPU at some point before a required draw call is made. While it could be called on the painting thread to keep the cost of the copy away from the rendering thread, that would require having an OpenGL context for both threads, which has been shown to be less efficient than using a single context [45]. Using this method in ViNE, I call `glBufferSubData` to copy the region of data that has been modified. Depending on the indices of the vertices painted, this may be anywhere from a very small, or very large portion of the whole buffer.

In regards to the other method, "Pinned memory" is a region of non-pageable memory which can be accessed directly by both the CPU and GPU without the need for an explicit

transfer. Data can also be transferred at higher bandwidths to the GPU than with other sections of RAM. In this implementation, the triple buffer is allocated directly in pinned memory. The rendering thread does not need to make an explicit transfer, it just needs to specify the index to the most recently completed buffer when initializing the render.

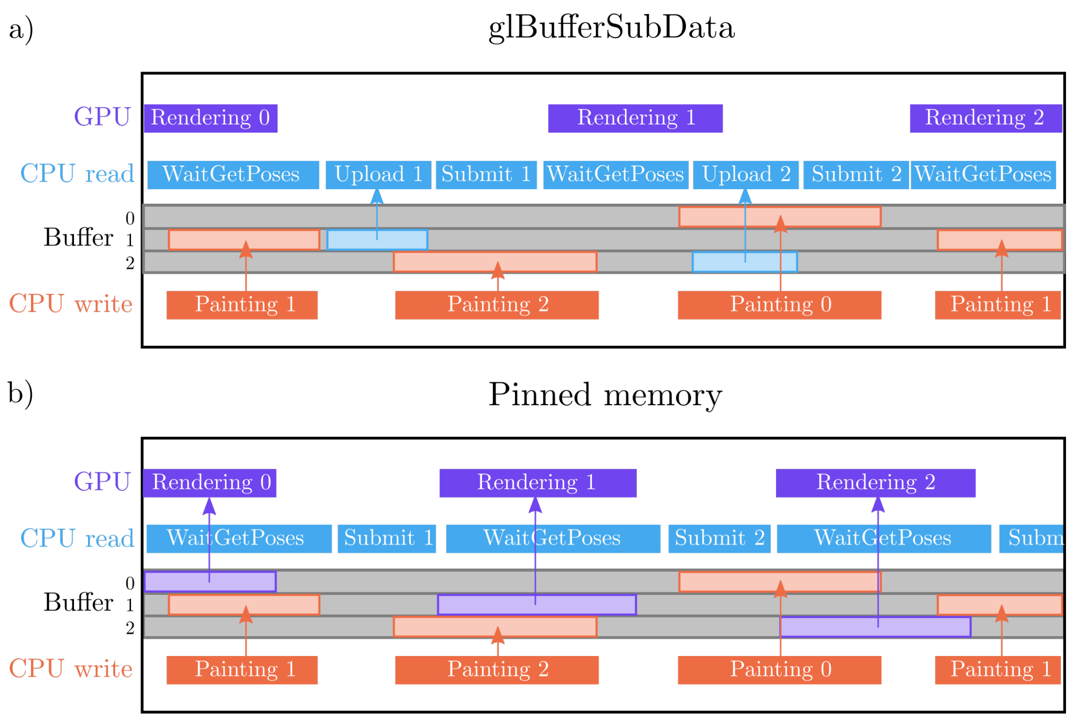


Figure 5.16: A comparison of the timelines for using `glSubBufferData` and PinnedMemory to transfer geometry data. "Submit n" refers to the initiation of a draw call on the CPU using the data in buffer n. The central region indicates of each diagram indicates when each thread is using each buffer. (a) `glBufferSubData` is used to explicitly upload the data with "Upload n". The CPU is exclusively used to read and write from the three buffers. (b) The GPU directly accesses the pinned memory storage as directed by the CPU read thread.

Having implemented both of these methods, I found the difference between the two to be slight. To compare the methods, a sequence of painting and controller inputs was recorded for the gerbera mesh and played back using both implementations. `glBufferSubData` took 3% longer than using pinned memory to complete the sequence of inputs. When I profiled

these two methods using NVIDIA Nsight I found that the time the GPU spends rendering is slightly longer using pinned memory, however the `glBufferSubData` operation causes `WaitGetPoses` to miss the synchronization point more often. In addition, when painting large portions of the mesh the `glBufferSubData` function call took up to 90 ms to complete, causing 8 frames to be missed. By comparison, when using pinned memory the performance was not affected at all by painting the mesh. These results are consistent with those obtained by other performance comparisons [45, 34]. Due to the more consistent frame rate using pinned memory, this is the method I use in ViNE.

5.7.3 Rendering

Even with the improvements obtained in subsection 5.7.1 and subsection 5.7.2, rendering times of 14 to 27ms make it impossible to complete each frame within the 11ms required to achieve the ideal frame rate of 90 frames per second. In fact, at 27ms, the program will not be able to maintain the fallback of 45 frames per second either.

A measure that greatly reduces the rendering time is enabling back-face culling. This setting automatically discards faces which are oriented away from the viewer, as determined by a clockwise winding order. Removing back-faces has advantages and drawbacks; while a mesh generated from marching cubes is usually closed, except for at the boundaries of the 3D volume, using ViNE's selective visibility can create holes in the mesh, exposing the back-faces. In exchange, the maximum rendering time for the gerbera mesh was reduced to 18ms, a reduction by 33%.

Another large time expense that can be improved is one inherent in all virtual reality applications, rendering everything twice to display to each eye. This requires OpenGL to execute every stage of the graphics pipeline multiple times, including reading the color data from pinned memory.

The `GL_NV_stereo_view_rendering` extension provided by NVIDIA can be used to reduce the amount of redundant work. In the basic OpenGL pipeline, a vertex shader

is first executed on every vertex in the mesh, with the primary purpose of transforming vertices from model space into clip space. The fact that this transformation differs from each eye is the reason why two executions of the entire pipeline are needed. The `GL_NV_stereo_view_rendering` extension allows a vertex position to be output for each eye, and sent to be rendered to separate halves of the framebuffer. With this extension, only a single render call is needed, which means the vertex data only needs to be transferred from the pinned memory to the GPU once.

With the back-face culling and the `GL_NV_stereo_view_rendering` extension applied, the range for rendering times becomes 10 to 14ms. While these render times will often still exceed the required time for 90 frames per second, in practice most of the frames are delivered fast enough to update at 90 frames per second, and only 20-30% of the time will a frame be missed, dropping it to 45 frames per second. While it would be ideal to always display at the maximum frame rate, with asynchronous reprojection, missed frames have a small impact on the user experience.

Chapter 6

Application examples and results

Using ViNE, I segmented 3D image stacks of three flower heads: Gerbera, Helianthus and Craspedia.

6.1 Gerbera

The gerbera flower head took approximately 7 hours to fully segment. At the time of segmenting, I had not yet introduced the selective visibility feature, which increased the total time required.

The gerbera flower head was segmented into six components: the bracts, ray florets, disk and trans florets, centrifugal veins (growing outwards from stem towards bracts), the floret veins which connect to the ray disk and trans florets, centripetal veins (grow inwards and upwards from where they branch off of the centrifugal veins), and a single isolated vein traced from start to finish. These components are depicted in Figure 6.2.

Segmenting large uniform areas in the gerbera head, such as the regions filled with floret veins worked quickly with ViNE. It was possible to paint broadly in the correct region, and fix the small number of errors that came from it. An area that proved challenging to segment was the voronoi diagram-like structure that the tips of the floret veins pass through to enter the disk florets. When I segmented this structure I needed to precisely paint around each of the floret veins, a task that was time consuming, and would have benefited greatly from the selective visibility feature. Due to the ambiguity of what part of the flower head this structure represents, I have not displayed it along with the other components.

The largest time sink, however, was locating all of the missed veins. The inside of the flower head was complicated enough that many small areas which had not yet been painted

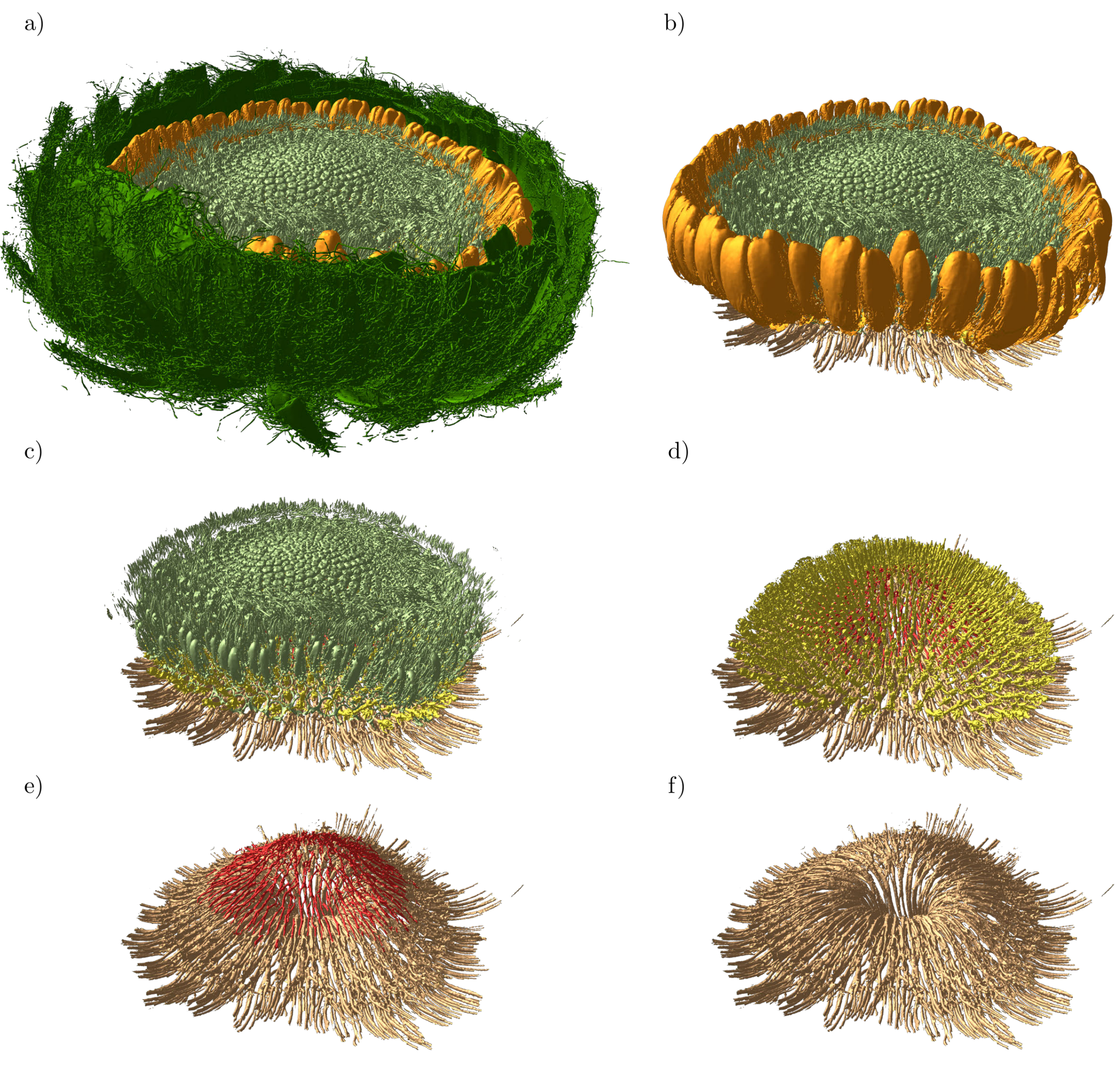


Figure 6.1: “Unwrapping” of each component of a gerbera flower head layer by layer.

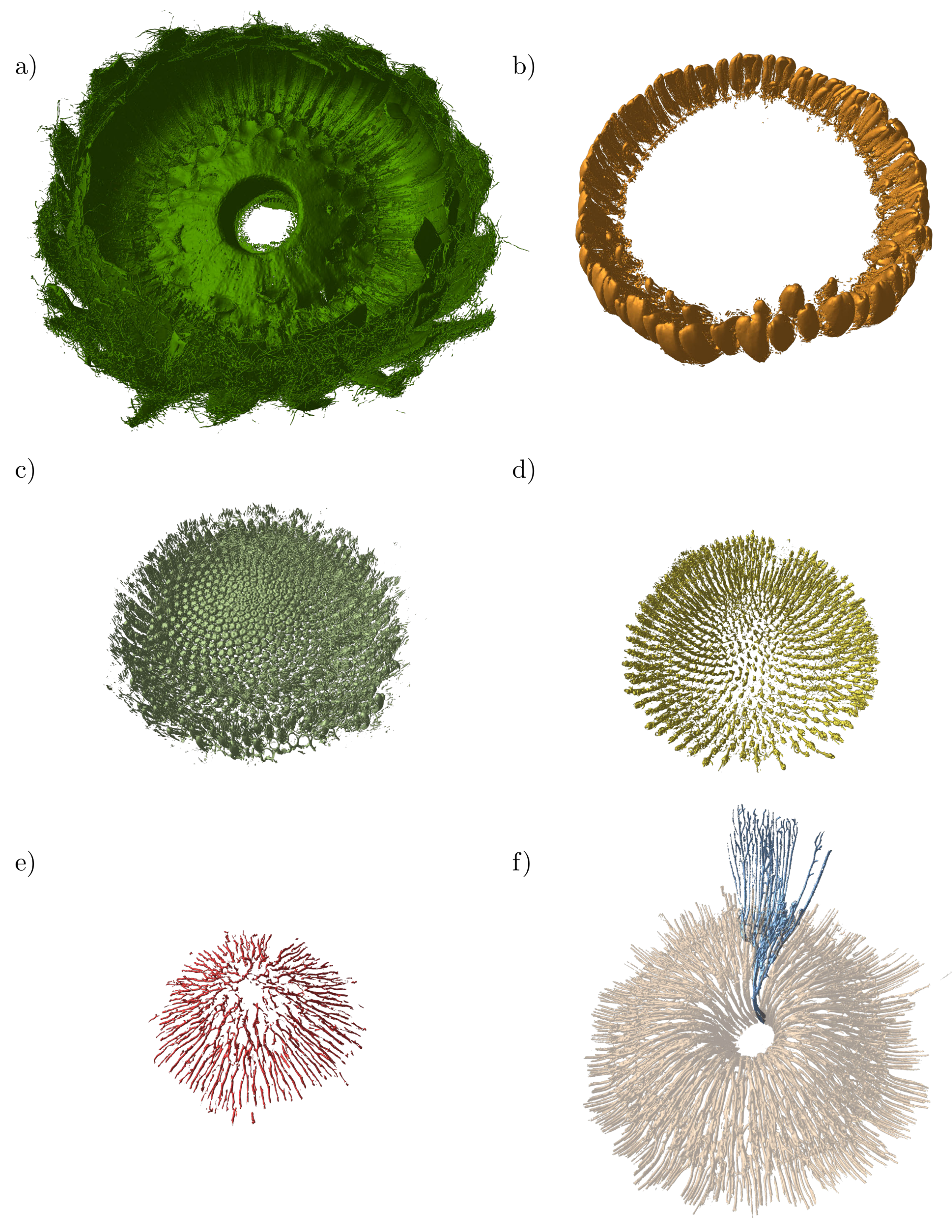


Figure 6.2: A Gerbera flower head separated into its components. (a) Bracts. (b) Ray Florets. (c) Disk and trans florets. (d) Floret veins. (e) Centripetal veins. (f) Isolated vein overlaid on top of centrifugal veins. Components in bottom row are larger relative to the middle row for improved visibility.

were hidden by surrounding veins, a task that selective visibility would have aided in.

The main veins of the gerbera flower head travel from the stem towards the bracts in a smooth trajectory. floret veins connect to ray and disc florets above, joining with the main veins.

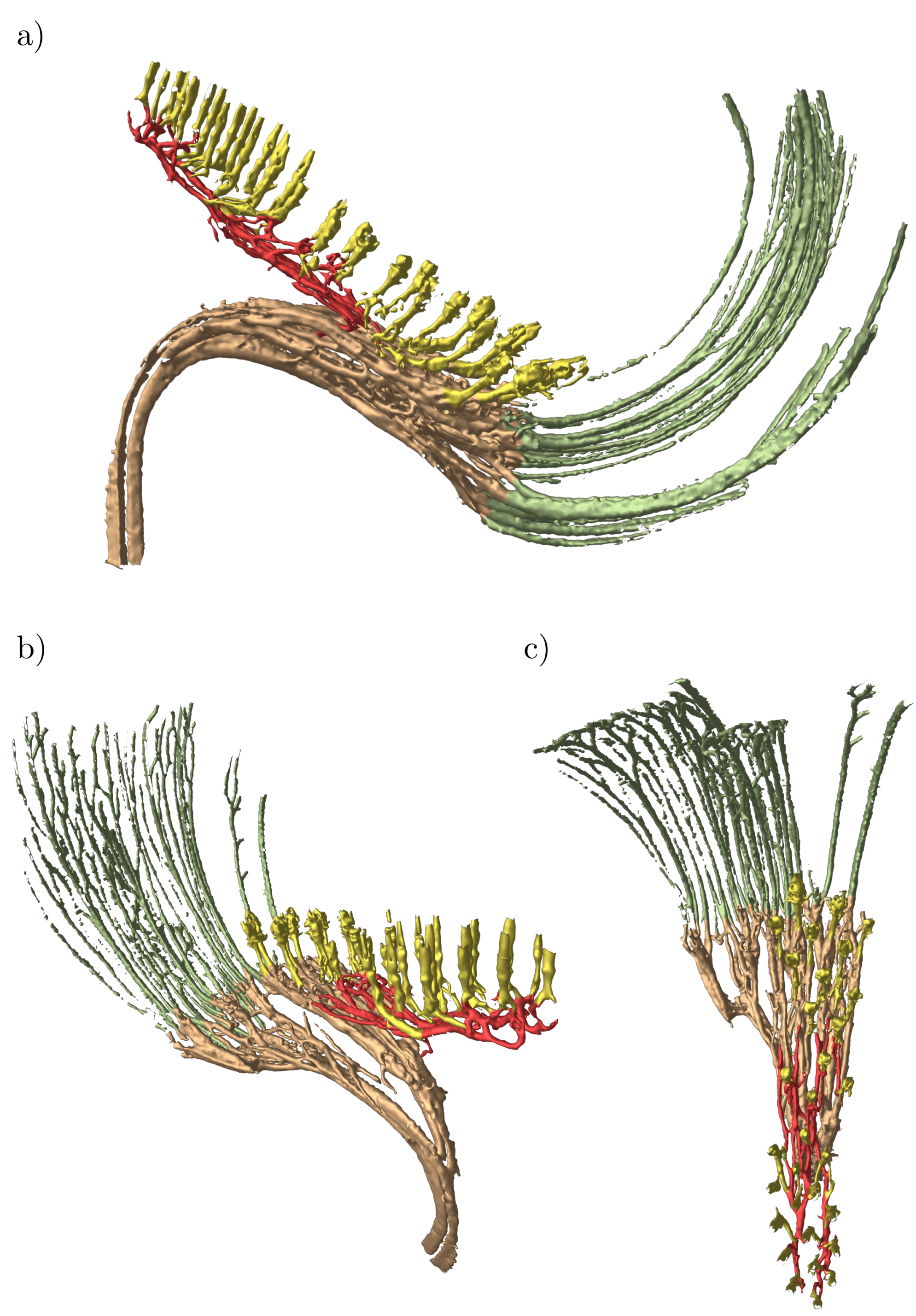


Figure 6.3: An isolated gerbera vein. Brown centrifugal veins originating from the flower's stem. Red centripetal vein trails up and back towards the center of the flower head. Yellow floret veins connect the ray and disk florets to the brown and red veins. Green bract veins connect to the centrifugal veins.

6.2 Helianthus

The Helianthus mesh consists of 32 million vertices and 64 million faces, which puts it beyond the range that ViNE was optimized for with the gerbera mesh. The frame rate remained steady at 45 frames per second, though rarely reached 90. Rendering times for this model took up to 17ms.

The Helianthus model took 4 hours and 15 minutes for me to segment. Helianthus has a larger number of veins than ferbera, but I attribute the faster segmentation time to the selective visibility feature, which helped by removing obscuring plant tissue, as well as allowing large regions of vasculature to be segmented at a time by hiding already segmented components from being painted.

The majority of the segmentation was completed within the first 2 hours. Most of the remaining time was spent tracing the vasculature into the ray florets. The brush's spherical shape was ill suited towards painting veins in a wide and narrow environment. Many of the veins inside the ray florets were also several times wider than they are thick, which may be the result the scanned data not properly distinguishing adjacent veins, requiring me to paint back and forth as well as upward.

While the experience was still comfortable, the frame rate drop made it difficult to segment continuously for more than 45 minutes at a time without discomfort.

Compared to Gerbera in Figure 6.3, the vasculature in Helianthus is more disorganized: most adjacent main veins have smaller veins connecting them. I found it more difficult to identify the difference between floret veins and the connections between main veins, typically needing to follow them to the end to determine if they connect to a disk floret. Additionally, the floret veins shown in Figure 6.5 have a different character to those in gerbera. The path to the disk florets branches extensively, beginning from multiple points along the centrifugal veins. This contrasts with gerbera where most of the floret veins connect to a small set of long centripetal veins making their way towards the center.

Figure 6.4: Helianthus components. (a) Whole flower head. (b) Ray florets. (c) Base. (d) Disk florets. (e) Floret veins. (f) Main veins. (g) Isolated vein.

Figure 6.5: Isolated vein from the Helianthus mesh. Brown centrifugal veins branch into green centrifugal veins and orange bract veins.

6.3 Craspedia

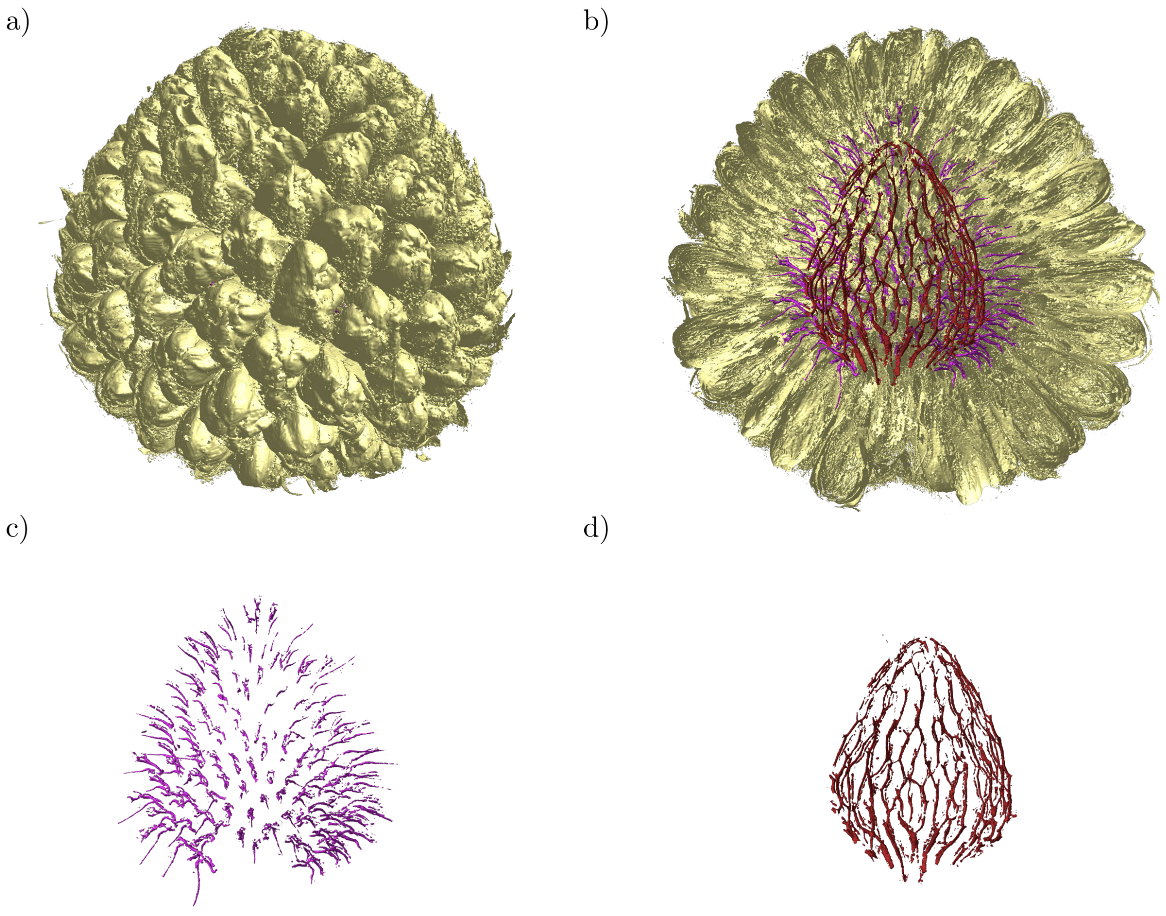


Figure 6.6: (a) Craspedia flower head. (b) Florets. (c) Floret veins. (d) Main veins.

The Craspedia mesh has 14 million vertices, and 28 million faces, making it the smallest of the three models. This mesh was most consistently rendered at 90 frames per second, and caused no discomfort while segmenting. Rendering times for this mesh took up to 9.3ms.

Segmentation of Craspedia took under two hours. As with gerbera, this model was also segmented before selective visibility was implemented. After the feature was implemented, I segmented it again to evaluate the difference. Segmentation with selective visibility took 20 minutes. I attribute most of this difference in time to the amount of noise surrounding the

vasculature. Removing this noise without selective visibility required painting around each vein wherever noise was found. On the second segmentation, however, I segmented the veins on their own, and then hid them to remove all of the noise with a larger brush.

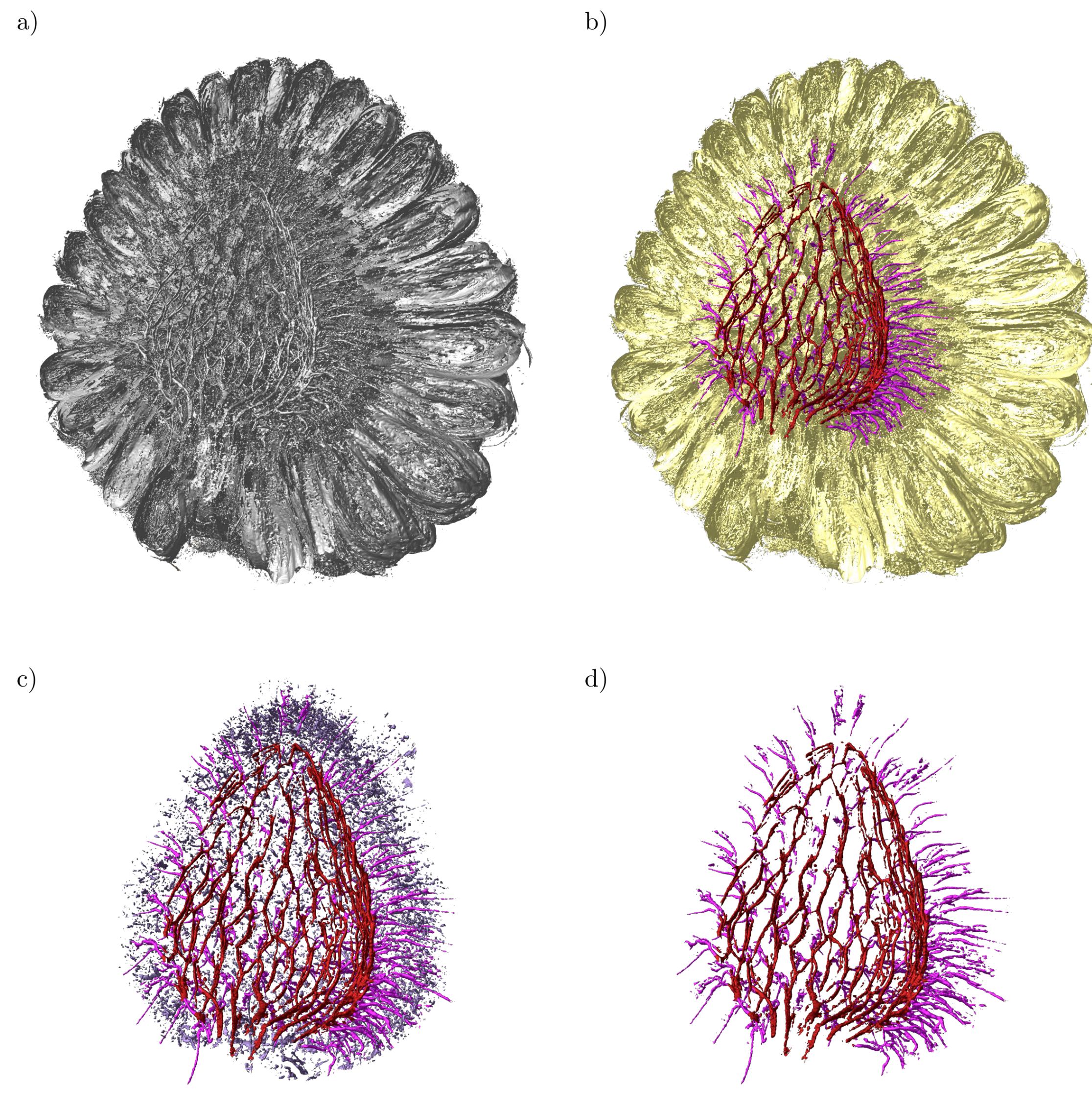


Figure 6.7: (a) Unsegmented flower head. (b) Fully segmented flower head. (c) Vasculature with surrounding noise. (d) Vasculature without surrounding noise.

6.4 Animated flythrough

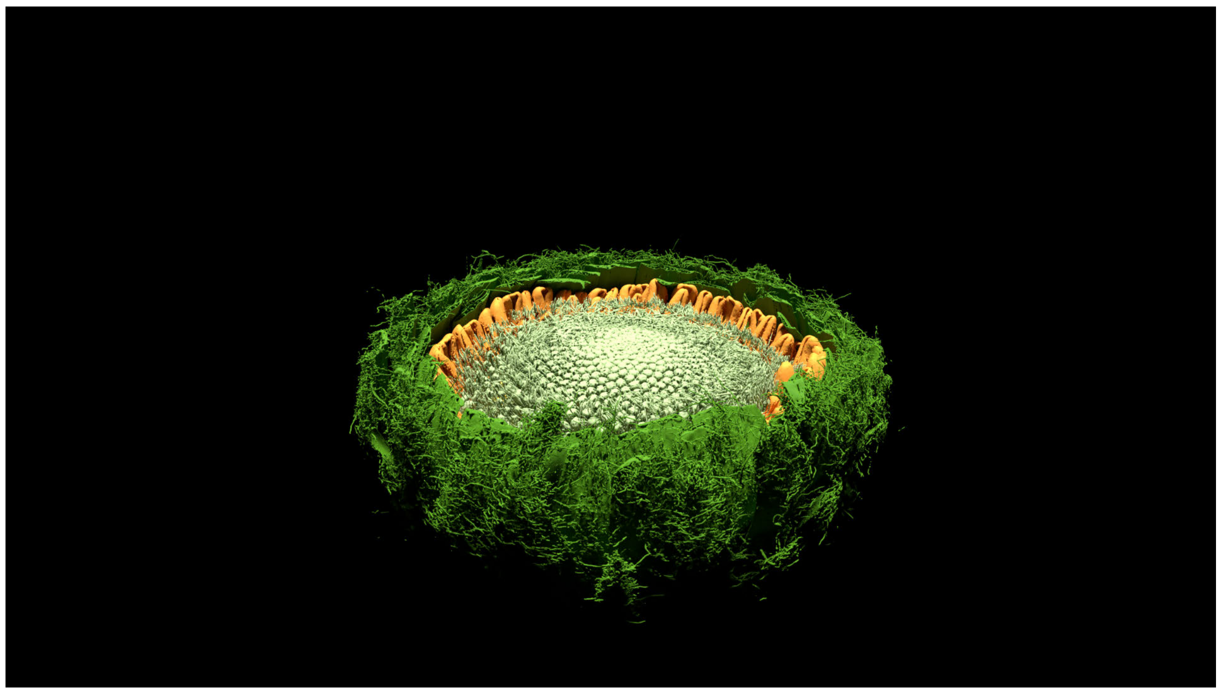


Figure 6.8: A frame from a flythrough of the Gerbera flower head, rendered in Blender and colored in ViNE.

Images cannot adequately capture the experience of viewing these plant models in virtual reality; the feeling of depth and immersion is lost, and it can be difficult to infer the structure from a static image. I created an animated flythrough of the Gerbera flower head to capture its shape better through motion.

I exported the Gerbera segmentation to a ply file with colors stored at the vertices. In the geometry processing software suite Meshlab [19], I used the "Smooth: Laplacian Vertex Color" filter to smooth out the sharp color transitions at the boundaries of segmented components.

I imported that model into Blender [9], and used the software to animate the camera along a path entering through the top of the flower head, circling underneath the main veins, panning over the floret veins, and exiting back out the top. Images from this animation can

be found in Figure 6.8 and Figure 6.9.

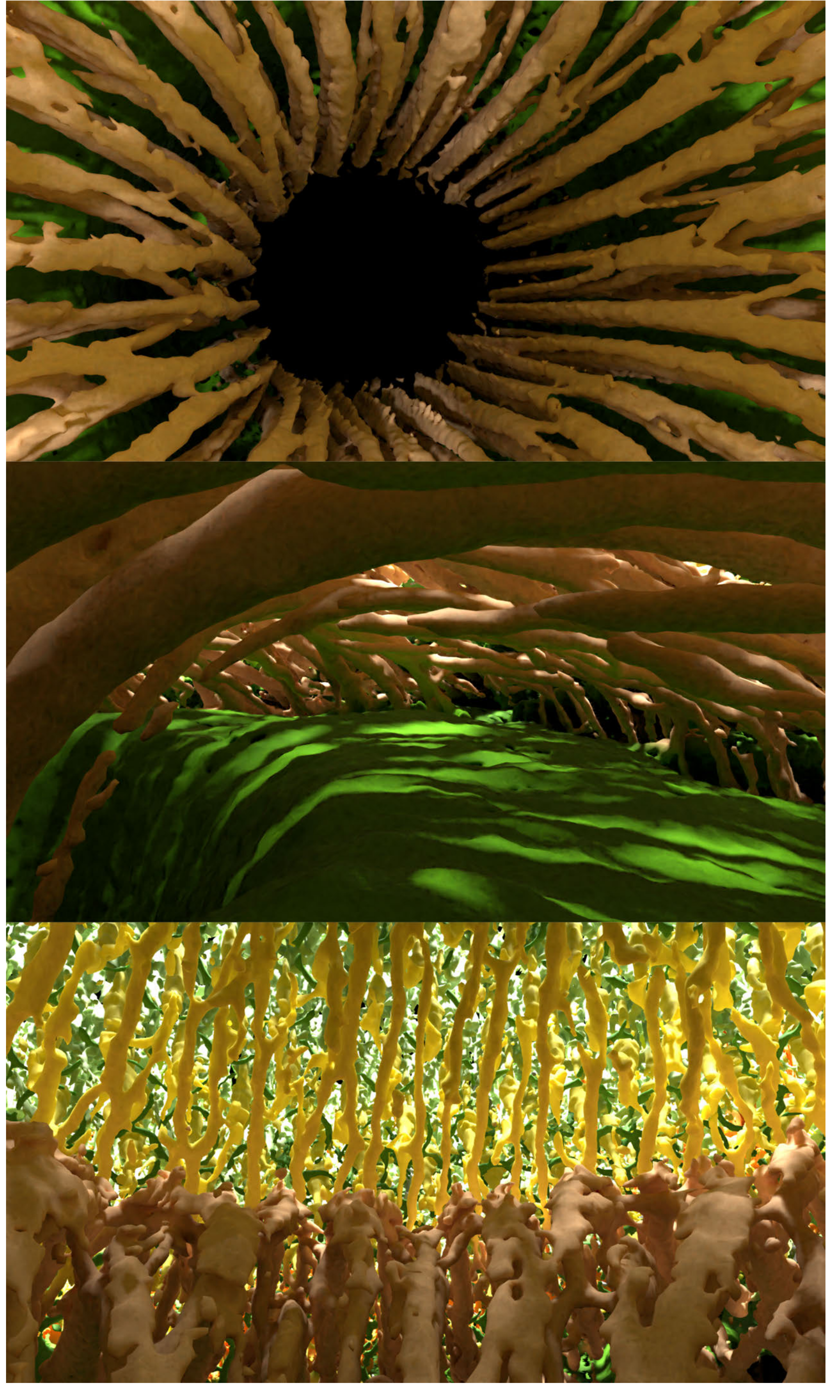


Figure 6.9: Frames from Blender animation.

Part II

Tree Wand

Chapter 7

Problem position

Generating the geometry of trees manually is prohibitively time consuming. Large trees can have tens of thousands of branches, and manually placing each is laborious. In painting, it is common for an artist to approximate a tree by the overall shape of its crown, or by drawing its more prominent branches explicitly, but masking the finer branches among more vaguely depicted bundles of leaves, creating a suggestion of the tree's form.

A modeling tool for creating trees should offer a similar range of specificity; allowing the modeler to focus in detail on the significant aspects of the tree's shape, while broadly defining parts which are less necessary to be specified explicitly. However, it is important that the details filled in for the modeler not only look plausible, but also have an appearance similar to that of the type of tree they wish to depict. Biologically motivated algorithms for generating trees typically have a set of parameters which are used to express a wide variety of tree forms. Tree Wand should also provide an easy method of manipulating these parameters within virtual reality.

Much of my work extends the methods developed by Longay et al. in TreeSketch [63]. They provided an application for modeling biologically plausible tress on a tablet device. However, their platform limited their interactions to two dimensions, which impaired the user's ability to specify the depth the tree's branches.

Due to the limited variation in coniferous trees, I have chosen to focus Tree Wand on modeling deciduous trees. Coniferous trees vary little compared to deciduous trees, and can be more easily described by a set of parameters, as opposed to an virtual reality interface allowing for more precise control.

Chapter 8

Background

8.1 Modeling in virtual reality

Virtual reality provides an ease of interacting with the environment in three dimensions that aids in many aspects of modeling. While 2D manipulation methods must restrict the interaction to at most two dimensions, potentially inferring the depth through context, applications in virtual reality can interact directly in 3D using trackable controllers. In addition to the extra degrees of freedom, perception of depth in virtual reality is improved by the parallax of two displays in the headset, parallax from small movements of the head, and the ability to sense your hand positions through proprioception.

However, despite these advantages, modeling in virtual reality has its own challenges. The additional degree of freedom can become a drawback when the user wants their interactions to be restricted along one or two dimensions. In addition, a user working with a mouse can use a table to support their hand, and the mouse will not move unless force is applied. Interactions in virtual reality, by comparison, are typically unsupported which has been demonstrated to reduce drawing accuracy [2]. These factors together can make precise modeling more difficult.

Applications for virtual reality modeling have a long history, predating the widespread availability of consumer VR devices in 2016. Clark et al. [20] developed a modeling interface in 1976 for use with a head-mounted display (HMD) and "wand" held in the user's hand. Meshes in this application were represented as B-Spline patches, and their shape could be manipulated by grabbing and moving the control points using the wand.

A very different approach was taken by CavePainting [50]: representing objects using a set of polygonized brush strokes instead of as a connected mesh. CavePainting was developed in

the Cave Automatic Virtual Environment (CAVE), a HMD-free virtual reality environment placing the user inside a room with images projected on the four walls. The user carries a prop that they would use as a brush to paint strips in the air. This application was designed with a focus on creating art, as opposed to assets to be used for other applications. Recent applications such as Google's Tilt Brush [37] have used this style of modeling. It has become common enough in VR applications that there has also been work generating meshes from polygonized brushstrokes painted in virtual reality [88].

McGraw et al. [69] proposed a technique for generating surfaces along the path of a moving curve between two controllers. An open or closed Hermite curve is defined between two trackable controllers, with the direction of the tangent vector at each endpoint being tied to the controller's orientation. As the controllers move, the curves from one frame to the next are connected to each other to form a swept surface. In addition to swept surfaces, McGraw et al. provided a method to distribute cloned meshes, such as cubes, along the curve between the controllers, using a rotation minimizing frame for orientation. Swept surfaces similar to those presented there have also been used in commercial applications such as Gravity Sketch [95].

In the realm of biology, the application LifeBrush [25] uses sketching in virtual reality to initialize simulations. A spherical brush distributes proteins within a simulation environment; placing them in the unoccupied spaces inside the brush's radius. In addition, a sculpting interface is used to model the shape of individual proteins as well as the environment surrounding them, allowing for addition and subtraction of volume to local regions of the mesh. The iterative nature of sculpting can help address the imprecision of interactions in virtual reality: small repeated modifications allows for errors to be quickly corrected. Sculpting also makes the foundation of the virtual reality modeling program Oculus Medium [108].

8.2 Tree modeling

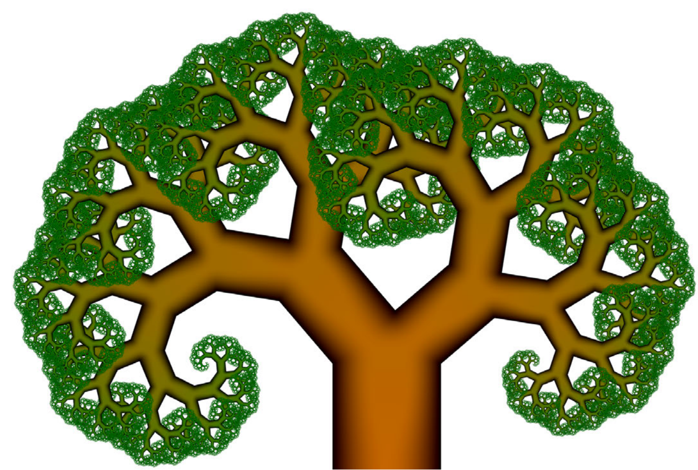


Figure 8.1: Pythagoras tree

There is a rich history of using mathematics to model treelike structures. Before the onset and availability of computers, recursively applied patterns provided a natural way to reduce a tree's complexity to a small set of easily applicable rules. An early example of a recursive pattern generating a treelike form is the Pythagoras Tree (Figure 8.1), developed by Albert E. Bosman in 1942 [12]. Many other examples of biological forms created from recursive patterns can be found in The Fractal Geometry of Nature [67] and The Algorithmic Beauty of Plants [84]. Due to the self-similar nature of fractals, many tree models can be concisely expressed using L-systems, a formal grammar which applies symbolic replacement rules to an initial axiom, and visualized geometrically based on the interpretation of those symbols [84].

Many other early attempts at modeling trees follow simple recursively repeating patterns. Honda [44] generated a tree structure using a fractal in which each branch splits into two child branches, branching at specified angles on the most horizontal plane the parent branch lies on — unless one of the branching angles is 0 and remains vertical, in which case each successive branch off the main trunk will differ from the last by a divergence angle. While the model

managed to create treelike structures with a simple set of rules, the self-similarity can become apparent at closer inspection, presenting an unnatural degree of regularity. Oppenheimer [78] addressed the strict-similarity of fractal-based models by introducing random perturbations to the branching angles, resulting in a less uniform, more organic looking form.

However, a tree's form cannot just be described with recursively repeated patterns, there are other factors that come into play. Branches can inhibit each other's growth, and shade, possibly from the tree itself, promotes shedding of branches [110, 91]. Sachs describes this as a "[competition for] environmental space, or, more specifically, for light." [90]. Internal signaling mechanisms can also inhibit the growth of branches [91, 90], notably the hormone auxin has been found to impede development of lower branches to enforce "apical dominance" [110].

Borchert et al. [11] modeled the effect of resource distribution on the development of buds by attempting to capture the manner in which water flux is distributed between an internode's children. Transpiration in the leaves causes water to flow towards them, resulting in an internode having more water directed towards it the more leaves it has as children. Apical dominance is modeled using a flux ratio, representing the extent to which the main branch is favored when water is distributed to it. The amount of water flux reaching an internode determines its "vigor" which will trigger the development of buds if above a threshold.

Several works have incorporated a competition for space into their models. Ulam defined patterns on polygonal grids that would grow to adjacent cells to occupy space, but reject any cells that would cause it to touch an occupied cell [101]. These rules resulted in a number of treelike patterns. A similar strategy was applied to fractal tree models [44], detecting when internodes have grown too close to each other, and shedding the one with the lowest vigor.

Rodkaew et al. [73] adopted particle driven method for generating trees. Particles, representing terminal buds, are distributed within a volume which determines the shape

of the tree's crown, and then moved according to attractive forces pulling them towards nearby particles and the base of the tree, merging when two particles become close enough. The paths traced by these particles becomes the branching structure of the tree.

Runions et al. [89] introduced the space colonization algorithm, a method with similarities to Rodkaew et al. which instead works from the base of the tree upwards. Attraction points are placed within the shape of the tree's eventual crown, which are assigned to the closest internode on the tree. Each internode with an attraction point assigned to it creates a new internode, growing towards the average position of the attraction points assigned to it, branching if the internode is not a terminal bud. When an attraction point falls within a "kill distance" of an internode, it is removed from the simulation. A variety of different tree forms could be expressed by modifying the density of the attraction points and the kill distance.

Palubicki et al. [79] builds off of Runions et al. by introducing buds as the only point of growth on the tree, each bud only having a cone within which attraction points are visible. Palubicki et al. also defined an alternative growth model to using attraction points, expanding the Borchert et al. [11] vigor model to consider the illumination of buds — the main phenomenon that space colonization was originally intended to capture — as well as using the light gradient to influence the direction of growth. They also introduce an interface for painting attraction points using a tablet.

While the previous methods are almost all procedural, there are also methods for tree modeling which focus on providing creative control over a tree's form.

Prusinkiewicz et al. [85] developed a method for manipulating a tree's shape through a set of function curves defining internode length, branching angle and main trunk shape in relation to the distance along the tree's main axis. The function curve specifying internode length serves as an approximation of the tree's silhouette so long as the first order branches are straight and protrude horizontally, and the higher order branches are relatively short in

length.

Boudon et al. [13] generates bonsai trees by specifying a hierarchy of envelopes within which the tree will be grown. Each envelope is a surface of resolution, and defines a "silhouette" encapsulating all of the envelopes at the next level of the hierarchy, with the bottom level envelopes being filled with leaves.

Pirk et al. [83] defined a framework for incorporating the effect of wind into tree growth models. The modeler can specify wind direction and strength, which may change over the lifetime of the tree, enacting a lasting plastic deformation to the tree's structure.

A method for generating trees from two dimensional sketches was provided by Chen et al. [17] as a part of the Xfrog modeling suite. Chen et al. infers the 3D shape of the sketch by attempting to match it to the set of parameters within a library of tree parameters which could have most closely reproduced that tree. After matching the shape of the sketch, the tree then continues developing for several iterations, or until it fills the shape of a crown drawn by the user and rotated into a surface of revolution. While this software typically works at interactive rates, larger trees may take 50 seconds or longer to generate.

SpeedTree [47] is a vegetation modeling program used by animation studios and video game developers. It has been used in over 40 films, and has been integrated into many environments such as Maya, Unreal Engine 4 and Unity. SpeedTree allows the modeler to procedurally introduce branches one order at a time, and manipulate them by modifying the position of nodes placed along each branch. It also allows for more direct control of features such as the detailed bumps and ridges in the tree's trunk . A more in depth analysis of how SpeedTree works is unfortunately not possible due to its proprietary nature.

TreeSketch [63] is a tree modeling application designed for tablets, built on the tree modeling techniques used in Palubicki et al [79]. Touch gestures are used on the tablet to paint attraction points for the space colonization algorithm. While the light gradient isn't used to affect the direction of growth, it is used with the extended Borchert-Honda model

to determine bud activation and promote shedding of branches exposed to too much shade. TreeSketch also allows for pruning and bending of individual branches. The depth of the brush is either set to where it intersects with the plane that passes through the base of the tree spanned by the vertical axis with respect to the tree vector horizontal in screen space. If the brush begins at a branch, the drawing plane is translated to include that branch. While this may often accommodate the modeler's intentions, there is not an direct way to control the depth of the brush.

Chapter 9

Generative algorithm

9.1 Tree growth in nature

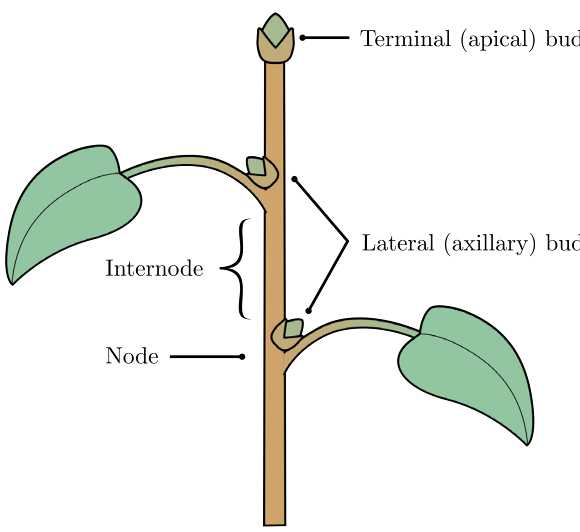


Figure 9.1: Twig morphology

The primary means by which a tree grows from season to season is not through the continual elongation of their internodes, but through the introduction of new cells at its meristems. A meristem is a region on a plant consisting of undifferentiated cells which can divide, and later form the basis of plant organs. The apical meristem, which can be found in lateral and terminal buds, facilitates growth by initiating new internodes and lateral buds. After the initial period of growth has ended, internodes enter dormancy and their length remains fixed [110].

The second category of meristem that affects tree form is the lateral meristem, which

is responsible for producing the plant tissue that increases the diameter of the tree. After their initial period of growth, internodes do not grow in length, but they continue to grow radially [110, 91].

A bud may either lay dormant, produce a long shoot, or produce a short shoot. A long shoot, is — as its name suggests — longer, but also has the capability to produce new lateral branches. A short shoot is substantially shorter, and can only produce leaves and main internodes. Which form a bud will produce may be the result of the environment and internal signaling, and likely an interplay between the two [110].

The order of a branch — the number of lateral branchings which must be taken to reach it — also has an effect on shoot length, though it is typically less dramatic than the difference between short shoot and long shoots. The higher the branch's order, the shorter its average length [110, 27].

The availability of water, nutrients and light, as well as hormonal signaling ~~have been suggested to~~ affect tree form. *Apical dominance* is a phenomenon observed in trees where the main shoot will inhibit development of lateral shoots. The hormone auxin plays a role in this regulation [110]. In addition to the factors determining which buds produce new shoots, insufficient resources given to branches cause them to be shed.

The main branching angle is the angle between the main shoot's and its parent internode's headings, while the lateral branching angle is the angle between the lateral shoot and its parent internode's headings. Branching angles tend to be consistent across a tree, with the main branching angle often being small, while the lateral branching angle tends to be larger, up to 90°[74]. The divergence angle is the angle between successive buds on the tree. This may not be the angle between successive branches, as not every bud will produce a shoot, and branches may be shed.

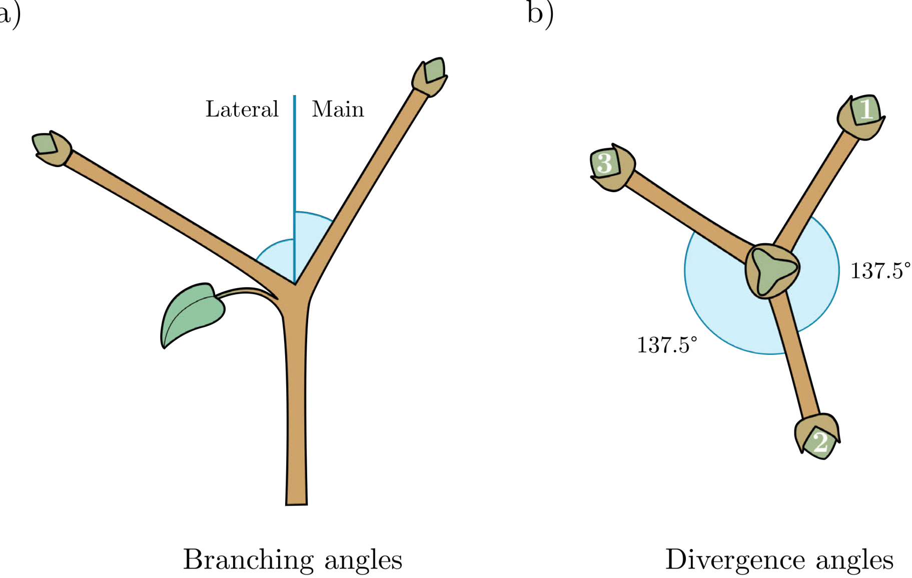


Figure 9.2: (a) Branching angles measure the angle between the previous internode's heading, and the main or lateral shoots' headings. (b) The divergence angle between successive branches of a tree with a main branching angle of 0. This angle is often 137.5°: known as the "golden angle". Branches are numbered in order of lowest to highest.

9.2 Tree data structure

In Tree Wand, I represent the trees using a pointer-based tree data structure, with each node containing pointers to its children, a pointer to its parent, and some data that is stored at the node.

```
class Node2{
        std::unique_ptr<Node2> children [2];
        Node2* parent;
        NodeData data;
};

class Node3{
        std::unique_ptr<Node3> children [3];
        Node3* parent;
        NodeData data;
};
```

I use two different trees for my data structures, one for two-way branching and another for three-way branching. This allows the representation to be kept concise. The main branch continues along the child in the first index, with the second and third indices represent the lateral branches.

An internode is defined between two successive nodes of the tree. Information which is associated with an internode, such as a branch's radius, is stored in the node at its tip. A unique pointer holds ownership over the data it points to, and will automatically call the child's destructor and delete its associated data when the unique pointer is reassigned to a different object or has its destructor called. This aligns with the tree data structure presented by Sutter [97]. Inside of the `NodeData` struct I keep track of the node's position, its up vector, radius, vigor, accumulated light, weighted accumulated light (multiplied by the lambda values of its children) and the node's illumination.

While it is common to model trees with turtle geometry, partially due to how it can simplify implementations, I have decided to represent the positions of all the tree's nodes in the global coordinate frame. Using turtle geometry to find the position of any node on the tree requires a frame transformation for every internode from it to the tree's root, any

operation on each end node of the tree would then require a number of transformations equal to the number of internodes in the tree. In comparison, storing the node's global position requires a frame transformation to be done only when a new internode is added to the tree.

I have chosen to model buds implicitly within my tree data structure. Each pointer in the list of children either points to its child internode, or is a null pointer, indicating that a bud exists there. While this requires a small amount of extra code to address, it reduces duplication of information, as a large amount of information associated with the buds is the same at its parent node, such as its position and illumination.

9.3 Basic growth model

I model tree development starting with a single internode with an terminal bud at the top. At each growth step, some portion of the tree's buds will produce new internodes in the direction of their heading.

The initial heading of a bud is determined by the divergence angles and branching angles of the tree, as pictured in Figure 9.2. I use a moving frame to keep track of the direction of the lateral branch for each internode by using a vector perpendicular to its heading, denoted $\hat{\boldsymbol{l}}$. This vector can be used along with the internode's heading, $\hat{\boldsymbol{h}}$ to calculate the initial direction for the lateral and main buds:

$$\hat{\boldsymbol{b}}_{main} = \cos(\theta_{main})\hat{\boldsymbol{h}} - \sin(\theta_{main})\hat{\boldsymbol{l}} \tag{9.1}$$

$$\hat{\boldsymbol{b}}_{lateral} = \cos(\theta_{lateral})\hat{\boldsymbol{h}} + \sin(\theta_{lateral})\hat{\boldsymbol{l}} \tag{9.2}$$

For each consecutive internode, $\hat{\boldsymbol{l}}$ is first rotated by the divergence angle around previous internode's heading, and then by the rotation between the previous and the next internode's headings, $\hat{\boldsymbol{h}}_{prev}$ and $\hat{\boldsymbol{h}}_{next}$, respectively.

$$
\begin{aligned}
\boldsymbol{q}_{div} &= \left[\cos\left(\frac{\theta_{div}}{2}\right), \quad \sin\left(\frac{\theta_{div}}{2}\right)\hat{\boldsymbol{h}}_{prev}\right] \\
\theta_h &= \arccos\left(\hat{\boldsymbol{h}}_{prev} \cdot \hat{\boldsymbol{h}}_{next}\right) \\
\boldsymbol{q}_{head} &= \left[\cos\left(\frac{\theta_h}{2}\right), \quad \sin\left(\frac{\theta_h}{2}\right)\frac{\hat{\boldsymbol{h}}_{prev} \times \hat{\boldsymbol{h}}_{next}}{\left\|\hat{\boldsymbol{h}}_{prev} \times \hat{\boldsymbol{h}}_{next}\right\|}\right] \\
\hat{\boldsymbol{l}}_{next} &= \boldsymbol{q}_{head}\boldsymbol{q}_{div}\hat{\boldsymbol{l}}_{prev}\bar{\boldsymbol{q}}_{div}\bar{\boldsymbol{q}}_{head}
\end{aligned}
\tag{9.3}
$$

Thus far these examples have been outlined for trees with "alternate branching". "Opposite branching" is another branching pattern, which occurs in trees such as maple. For trees with opposite branching, each node can produce three buds, a lateral bud on either side, with the main bud growing between them, as shown in Figure 9.3.

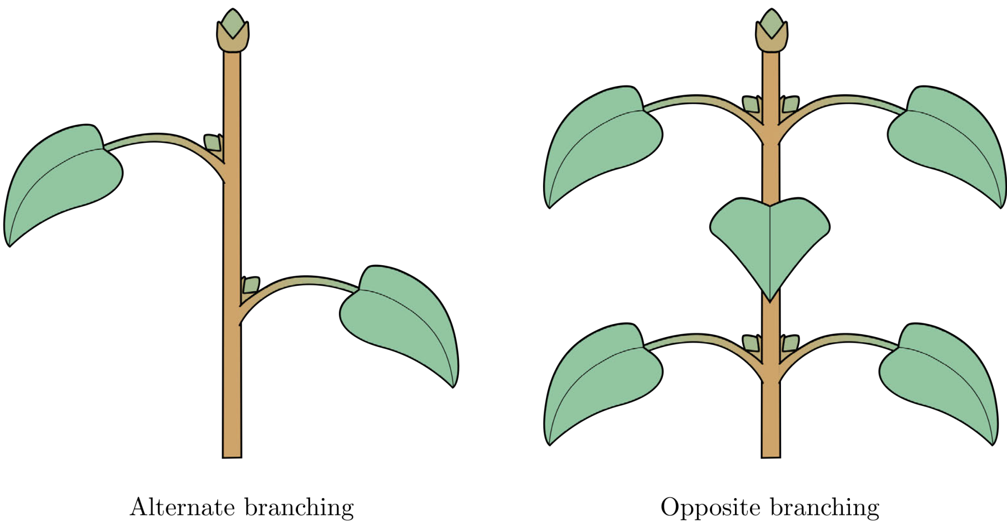


Figure 9.3: Two modes of branching.

The approach used for alternate branching is modified slightly to handle the opposite branching pattern. The main bud's initial heading will match that of the previous internode, whereas Equation 9.2 can be used for the two lateral buds, flipping the sign of the sine term between them so they branch in opposite directions.

9.4 Competition for space

Framing tree growth in terms of competition for space attempts to capture several phenomena that impact tree form. The first is the availability of space; a tree cannot grow where there is no room to, either due to the surrounding environment or the tree itself. The second is availability of light; the denser the packing of branches and leaves, the less light they will receive.

While competition for light is modeled more explicitly in the following section, having the ability to specify the available space for a tree to grow provides an effective means for interactively controlling the tree's shape, as well as modeling the effects of light in a local neighborhood.

As in previous implementations of the Space Colonization algorithm [89, 79, 63], the available space for the tree to grow is indicated using a random distribution of "markers". An *occupancy zone*, a sphere around each node on the tree with a radius provided as a parameter to the algorithm, removes any markers within it to indicate the node has claimed that region of space.

During each of the tree's growth steps, nearby markers are assigned to buds. Each bud has a limited range at which markers are visible. The distance between the marker and the bud must be less than or equal to the *view radius* — which specifies how far away marker particle can be — and it must fall inside a cone defined by the *view angle*. Together these form the bud's *perception volume*, as presented by Palubicki et al. [79].

While in previous work, the marker is assigned to the closest bud [89, 79, 63], I have made a small modification, by assigning it to the bud with the smallest distance relative to the length of the internode it would produce as shown below:

$$d_{modified} = \frac{d}{l}\,, \tag{9.4}$$

where d is the distance from the bud to the marker, and l is the length of internode the bud would produce.

In previous approaches, markers are assigned to the closest buds because the internode produced by the closest bud can more quickly occupy that space. However, a longer internode grows at a faster rate, which indicates a greater ability to fill nearby space than shorter shoots. Increasing the likelihood of markers being assigned to lower order buds makes trees with well defined main trunks easier to generate.

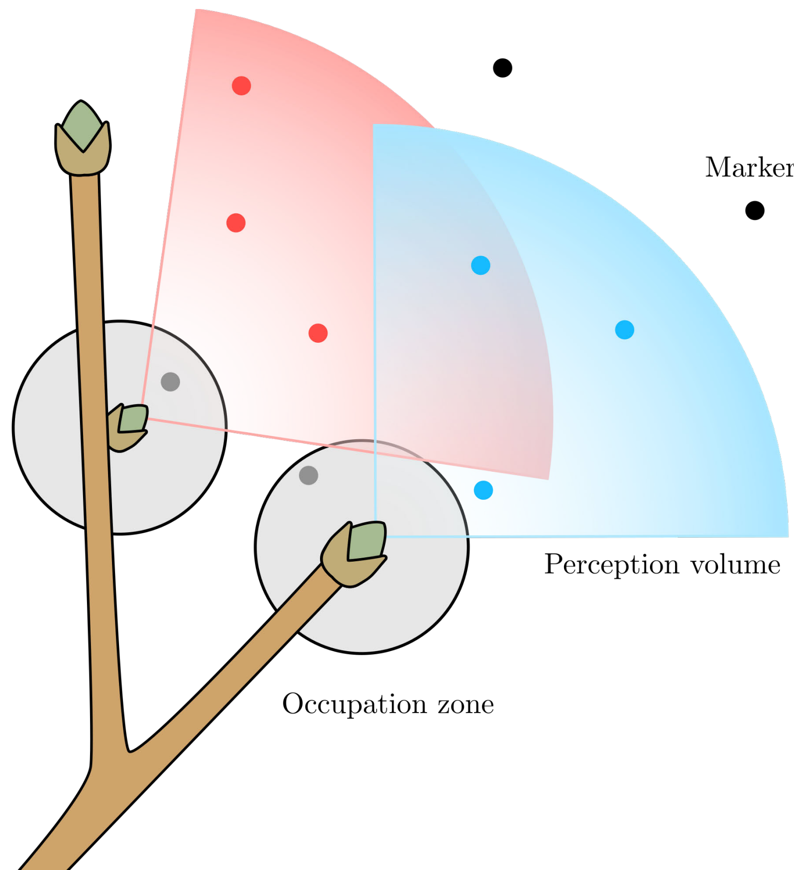


Figure 9.4: Circular markers being assigned to buds. The color of the marker matches the view cone of the bud it is assigned to. Grey circles represent the occupation zone within which markers are removed.

If a single attraction marker has been assigned to a bud with sufficient vigor, that bud will produce an internode. Otherwise, the bud remains dormant.

9.4.1 Influences on growth direction

As in Longay et al. [63] there are three influences that impact the direction that a new internode will grow towards. Each of these influences is characterized by a vector indicating the direction that influence encourages the internode to grow in. The first is its “heading”, which is the initial heading of the bud it develops from, which will be represented by $\hat{\boldsymbol{h}}$. This encapsulates a resistance to bending.

The second influence is “tropism”, $\hat{\boldsymbol{t}}$, specifically gravitotropism, which is the tendency for internodes to gradually adjust their heading towards a specific angle from vertical. Common variations of this are orthotropism, in which internodes have a tendency to grow opposite to the direction of gravity, and plagiotropism, in which internodes have a tendency to lie on the horizontal plane.

When the tropism angle is not 0°or 180°, there are an infinite number of directions the internode could point in that would place it at that angle from vertical. As in Longay et al. [63], I choose the vector that the bud will grow towards for the tropism influence to be the vector requiring the smallest change in direction. This vector lies on the plane spanned by $\hat{\boldsymbol{h}}$ and the y axis, $\hat{\boldsymbol{y}}$, and can be found with the following construction:

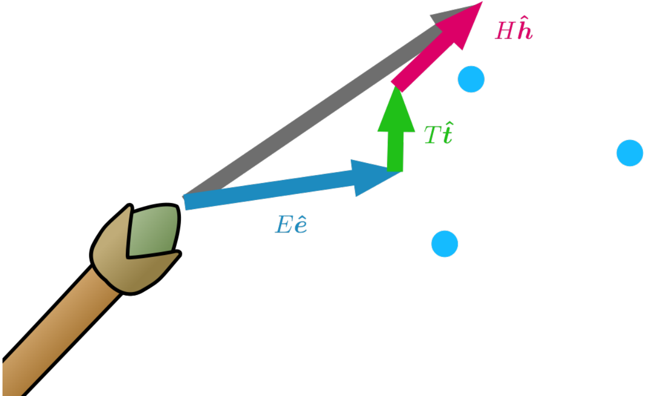


Figure 9.6: Influences affecting the direction of internode development. Depicts a strong weighting towards environmental influence. The tropism angle is 90 degrees from horizontal.

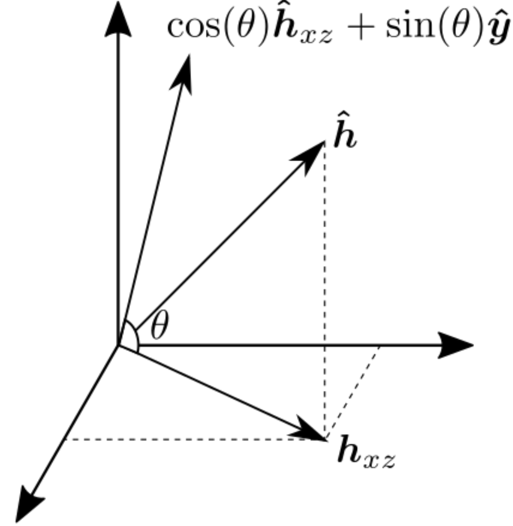


Figure 9.5: How to calculate the tropism vector associated with a heading, $\hat{\boldsymbol{h}}$.

The final influence is the "environment", $\hat{\boldsymbol{e}}$, which is the vector towards the average position of the markers assigned to that bud, representing a tendency to grow towards empty space.

Each of these influences are assigned weights, and combined in a linear combination before being normalized, as shown in Figure 9.6. This is expressed with the following equation:

$$\hat{\boldsymbol{d}} = \frac{H\hat{\boldsymbol{h}} + T\hat{\boldsymbol{t}} + E\hat{\boldsymbol{e}}}{\left\| H\hat{\boldsymbol{h}} + T\hat{\boldsymbol{t}} + E\hat{\boldsymbol{e}} \right\|}, \tag{9.5}$$

where H, T, and E are configurable constants which weight the impact of each influence. The final position of the internode's endpoint is found by multiplying the direction $\hat{\boldsymbol{d}}$ by the bud's internode length, and adding it to the bud's initial position.

9.5 Competition for light

While aspects of competition for light are partially modeled through competition for space, longer range effects, such as the shadow a canopy casts on lower branches, need to be addressed more explicitly.

9.5.1 Extended Borchert-Honda

The Extended Borchert-Honda model by Palubicki et al. [79] provides a means of controlling the distribution of resources within a tree. In this model, light Q is collected at the tree's buds and accumulated down the tree to its base. The total light at the base is then converted into vigor, a measure of the plant's resources, by the formula $v_{base} = \alpha Q_{base}$ where α is a user configurable parameter.

At each branching point, vigor is distributed proportionally to the amount of light a child contributed compared to the total light contributed by all of the internode's children. Additionally, a parameter λ is used to bias the distribution of resources towards or away from the main axis, according to the following equation:

$$v_{main} = v\frac{\lambda Q_{main}}{\lambda Q_{main} + (1-\lambda)Q_{lat}} \tag{9.6}$$
$$v_{lat} = v\frac{(1-\lambda)Q_{lat}}{\lambda Q_{main} + (1-\lambda)Q_{lat}}$$

While a bud's vigor is below a user specified threshold, it becomes dormant, and won't produce new shoots.

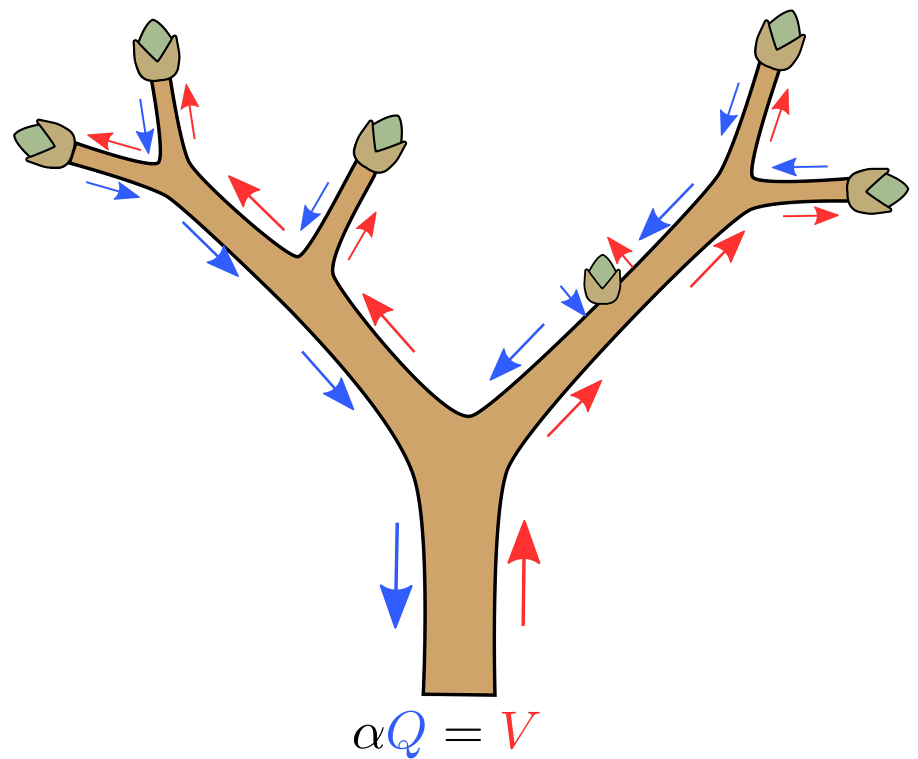


Figure 9.7: Light accumulation and vigor distribution. Light collected at the buds is propagated downwards to the base, then redistributed up in the form of vigor according to Equation 9.7.

As in Longay et al. [63], a bud only becomes activated when its vigor is above a user provided threshold, and if it has at least one marker assigned to it from the space

colonization algorithm. When $\lambda > 0.5$, the Extended Borchert-Honda model enforces apical dominance, resulting in a more excurrent form, characterized by an established main trunk. When $\lambda < 0.5$, the main trunk becomes less established and lateral branches become more numerous.

Equation 9.7 can be extended for three-way branching by assigning the a weight of $(1-\lambda)$ the second lateral bud, and adding it to the denominator for the sum of light reaching the branch

$$\begin{aligned} v_{main} &= v\frac{\lambda Q_{main}}{\lambda Q_{main} + (1-\lambda)Q_{left} + (1-\lambda)Q_{right}} \\ v_{left} &= v\frac{(1-\lambda)Q_{left}}{\lambda Q_{main} + (1-\lambda)Q_{left} + (1-\lambda)Q_{right}} \\ v_{right} &= v\frac{(1-\lambda)Q_{right}}{\lambda Q_{main} + (1-\lambda)Q_{left} + (1-\lambda)Q_{right}} . \end{aligned} \tag{9.7}$$

This formula still maintains the property that when $\lambda > 0.5$ the main bud is favored over the laterals. When $\lambda = 0.5$, each bud is distributed vigor directly proportional to the amount of light they contributed, and when $\lambda < 0.5$, the lateral buds are favored.

In addition to controlling bud activation, vigor is also used to control the shedding of branches. When a node's vigor falls below the number of its internode children, it is removed from the tree. Shedding and limiting bud activation are two means by which the tree's canopy can reduce the presence of lower branches.

While shedding plays an important role in a tree's appearance, it can result in work the modeler has done being undone as a result of shadow and changes in the distribution of vigor. As in Longay et al. [63], I've added a parameter κ to modulate the light reaching a bud through the formula:

$$Q = L^{\kappa} , \tag{9.8}$$

where L is the raw value of light calculated at the bud's location. The parameter κ removes the impact of light entirely when it is 0, and reduces the impact of small changes in light when $\kappa > 1$. This requires a larger change in the amount of light a bud receives before it is shed.

9.5.2 Shadow propagation

Before allowing light to have an effect on the growth, it's necessary to estimate the amount of light reaching various parts of the tree. While there are accurate methods for computing global illumination using ray tracing and path tracing, they typically require a large investment of compute time to remove noise, especially with complicated winding structures such as trees. When using Blender with the gerbera mesh from Part 1 it required forty minutes per frame of GPU compute time to reduce noise to acceptable levels in more obscured areas of mesh. Provided the 11ms window per frame in virtual reality, and the desire for interactive rates feedback while modeling, a faster — if less accurate — method is needed.

I use an approach based on that of Palubicki et al. [79], which approximates the illumination on a grid. Each node on the tree is produces a shadow which cascades down in a pyramid shape to lower levels of the grid. A parameter q_{max} defines the pyramid's height, and the width and depth at each level of the pyramid q starting from the top are $1 + 2q$.

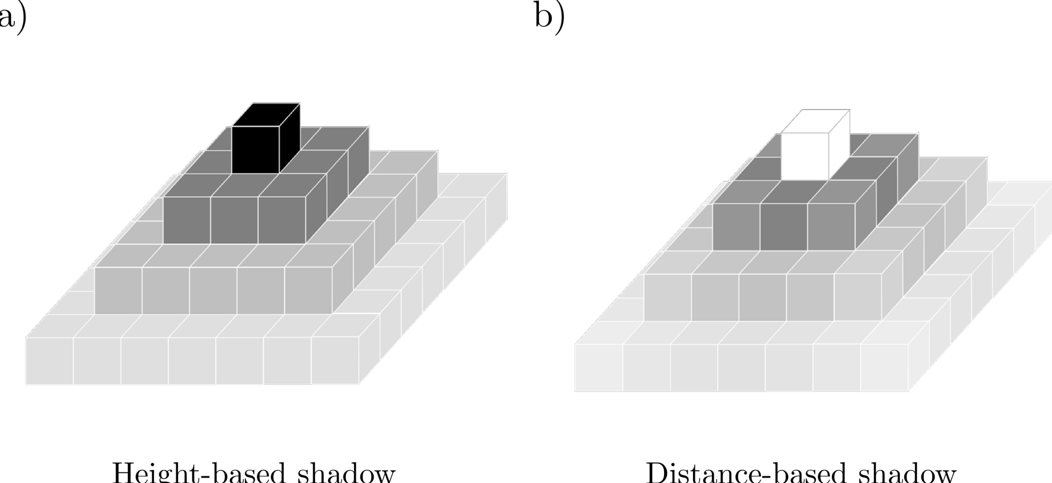


Figure 9.8: Shadow propagation methods. (a) The height-based shadowing from Palubicki et al. which calculates the degree of shadow solely based on height. (b) The distance-based shadowing used in Tree Wand, which calculates the degree of shadow using distance to the cell.

In Palubicki et al., the amount of shadow in a cell is calculated using a height-based approach:

$$\alpha\beta^{q}\,, \tag{9.9}$$

where $\alpha > 0$ and $\beta \in (0, 1)$ are user defined parameters which control the amount of shadow produced by a bud, and the rate of falloff. This formula is illustrated in Figure 9.8(a). In my experimentation with this method, the shadow had too strong of an impact horizontally, particularly at the corners. I instead decided to use a distance-based version of the formula, incorporating all the offsets from the node's index, instead of just q

$$\alpha\beta^{\sqrt{q^2+i^2+j^2}}\,, \tag{9.10}$$

where i and j are the offsets from the node's index in the x and z directions. The resulting shadow intensities are illustrated in Figure 9.8(b). One final modification made was to not

apply shadow to the cell the node occupies. There's a clear relationship that nodes in higher cells shadow the nodes in lower cells; however this relationship is made ambiguous between nodes in the same cell. As a result I found this made it easier to control the effect of a canopy of internodes impeding the development in the tree below. When shadowing was applied to the cell the internode occupied, increasing the amount of shadow would often reduce the density of the internodes within that cell, which would lessen the amount of shadow on internodes in cells below.

See below for an implementation of this algorithm:

Algorithm 9.1 Propagate shadow

```
1: procedure PropagateShadow(x, y, z, q_max, α, β)
2:     for q ← 1 to q_max − 1 do
3:         for i ← −q to q do
4:             for j ← −q to q do
5:                 grid[y-q][x+i][z+j] ← αβ^sqrt(x^2+y^2+z^2)
6:             end for
7:         end for
8:     end for
9: end procedure
```

It should be noted that while the distance-based shadows may make for a better approximation, they are also considerably more expensive. Using height-based shadows, the calculation only needs to be made q_{max} times, once for each level, compared to $\frac{q_{max}(q_{max}+1)}{2}$ times for the distance-based method.

This calculation can be made more efficient by using a look-up table, given that all the values in the pyramid are fixed provided an α, β and q_{max}.

Through experimentation I found that a cell with dimensions equal to $d \times 3d \times d$ where d is the base internode length, allowed for the generation of appealing tree forms. Setting the vertical dimension of the cell to 3 times the other two allowed for the shadows to be concentrated more below the internodes instead of being spread over a wide area.

9.5.3 Grid representation

Using the above algorithm with a q_{max} of 12 requires 78 cells to be written to per node. This requires that the illumination information be stored in a data structure which is efficient to modify. The density of the pyramids, and short distance between nodes producing them will also result in a dense distribution of data, with relatively few holes.

Using a voxel grid to store the light values would allow for efficient cache usage, as much of 9.1 would access the grid sequentially. Unfortunately, due to the variability in where the tree can grow, a voxel grid representation runs the risk of either being wasteful in its usage of space, or requiring multiple expensive resizing operations.

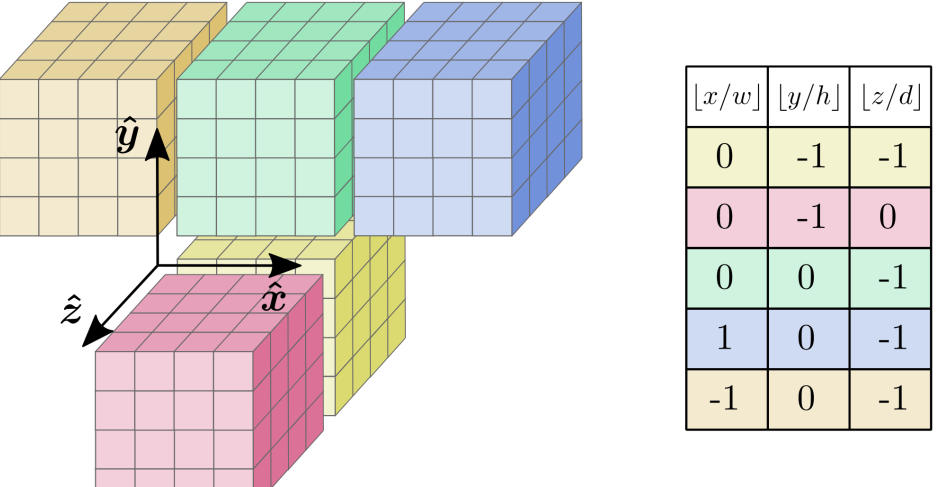
An octree data structure [70] starts with a cube enclosing the total area, which can be recursively subdivided into 8 equal sized cubes. Octrees are commonly used for multiresolution rendering, and while an Octree could end up representing the illumination information more efficiently than a voxel grid depending on the distribution of shadow, it requires many levels of indirection to access the value of a single cell. An octree that could represent a grid of 128x128x128 cells, would require seven levels of indirection to access a single cell, and this would still only allow a small region for trees to be grown.

Instead, I've chosen to use a two-tiered data structure. Space is partitioned into a coarse grid, with a fine grid nested inside each cell. The domain of the coarse grid is not explicitly represented, instead being stored as a hash table that maps an index representing a coarse cell's region of space to a pointer to a fine grid. The fine grid, and the pointer to it stored in the hash table, is only created when the shadow propagation attempts to write a value inside of the coarse cell it's contained within.

The 3D coarse grid index for a node is calculated by the equation

$$\begin{aligned} i &= \lfloor x/w \rfloor, \\ j &= \lfloor y/h \rfloor, \\ k &= \lfloor z/d, \rfloor\,, \end{aligned} \tag{9.11}$$

where w, h, d are the width, height and depth of each coarse cell, and x, y, and z are the world coordinates for the node. A coarse cell is created the first time an index hashes to that location.



$\lfloor x/w \rfloor$	$\lfloor y/h \rfloor$	$\lfloor z/d \rfloor$
0	-1	-1
0	-1	0
0	0	-1
1	0	-1
-1	0	-1

Figure 9.9: Fine grids accessible from hash table. Each entry in the hash table uses Equation 9.14 to map the three dimensional index to a pointer to a fine grid with the corresponding color.

To map the triplets of spatial indices to a single unique index that can be used as a key in the hash table, I chose to use Szudzik's pairing function [98]

$$S(a,b) = \begin{cases} a * a + a * b + b, & \textbf{if } a \geq b \\ a + b * b, & \textbf{otherwise}\,. \end{cases} \tag{9.12}$$

This function operates on two natural numbers, not integers, an issue that can be fixed by alternating positive and negative integers

$$N(a) = \begin{cases} 2a, & \textbf{if } a \geq 0 \\ -2a - 1, & \textbf{otherwise} \end{cases}. \tag{9.13}$$

To make a pairing function for three indices, I apply Szudzik's pairing function twice, resulting in the final hash function:

$$H(i, j, k) = S(S(N(i), N(j)), N(k)) \tag{9.14}$$

The fine grid inside each of the coarse cells is stored as a fixed size flat array of floating point values indicating illumination. The index for the fine grid can be calculated by dividing the remainder of the floor operation in Equation 9.11 by the dimensions of a cell in the fine grid, and taking the floor of the result.

This method gives some of the benefits of a voxel grid, while allowing for a smaller investment of memory occupied by empty space.

9.6 Branch radius

The radius of branches in trees often follows Da Vinci's Rule, which states that the sum of the cross-sectional areas of the children internodes is equal to the cross-sectional area of the parent [65]. This can be expressed more simply with the rearranged formula:

$$r^2 = r_1^2 + r_2^2\,,$$

where r is the radius of the parent internode, and r_1 and r_2 are the radii of the children. However, this doesn't always match what is observed in nature, where often an exponent between 2 and 3 is more appropriate[65], resulting in the following modified formula:

$$r^n = r_1^n + r_2^n\,, \tag{9.15}$$

where n is a configurable parameter.

9.7 Internode lengths

In Longay et al. [63] internode lengths were constant. However, in nature, the length of internodes tends to decrease with increasing branching order [27]. I have explored two different methods of controlling this, and implemented both in Tree Wand.

I first took a direct approach, and expressed the internode length as a function of the branching order. In Diao et al. [27], the average internode length decreases most significantly between lower branching orders, and becomes more uniform at higher orders. After trying several functions, I obtained the best results when having the first order start at a user specified maximum length, and setting each successive order's length to be halfway between the previous order's length and the minimum length. This is modeled by the following equation:

$$l = l_b + (\alpha - 1)l_b * \frac{1}{2^k}\,, \tag{9.16}$$

where k is the order of the branch, l is the final length, l_b is the minimum length, and αl_b is the maximum length, with α as a user configurable parameter.

While this expressed the qualities I looked for, it was designed to replicate what was observed in nature, and lacked biological motivation. To provide a more biologically motivated mechanism, I also explored making the internode's length a function of the vigor of the bud that produced it. This method operates on the assumption that an internode receiving more resources will develop into a longer internode. It can also explain the decrease in length over branching order through apical dominance; in the Extended Borchert-Honda model, a λ value greater than 0.5 will cause a tree to distribute more of its vigor to lower order branches.

I model the increase of segment length as a function of vigor by using the ratio between the vigor of the internode's parent bud, v, and the maximum vigor of any bud on the tree, v_{max}:

$$l = (1 - \frac{v}{v_{max}})l_b + \frac{v}{v_{max}}\alpha l_b\,. \tag{9.17}$$

This formula is inspired by that used in Longay et al. [63] for determining the number of internodes produced by a bud in a growth step. However, the increased internode length also affects the modified distance used to assign markers to buds Equation 9.4. As a result of using this method, buds with higher vigor can be assigned markers which would otherwise go to closer buds with less vigor.

In my experimentation with this method, it consistently produced a dominant main trunk when the λ value for apical dominance was greater than 0.5.

I evaluate these two methods further in chapter 12.

9.8 Algorithm

Taking all the preceding elements together, we have the algorithm for generating a tree:

Algorithm 9.2 Tree Growth Step

```
 1: procedure TreeGrowthStep(tree, markers)
 2:     RemoveMarkersInOccupationZone(tree, markers)
 3:     buds ← LocateActiveBuds(tree)                 ▷ subsection 9.5.1
 4:     AssignMarkersToBuds(markers, buds)            ▷ section 9.4
 5:     GrowInternodes(tree, buds)                    ▷ subsection 9.4.1
 6:     PropagateShadow(tree)                         ▷ Algorithm 9.1
 7:     AccumulateLight(tree)                         ▷ subsection 9.5.1
 8:     DistributeVigor(tree)                         ▷ Equation 9.7
 9:     ShedInternodes(tree)                          ▷ subsection 9.5.1
10:     CalculateDaVinciRadius(tree)                  ▷ section 9.6
11: end procedure
```

The way in which markers input to this algorithm are obtained is described in section 11.1.

To make the Algorithm 9.2 agnostic to the underlying tree structure, I have implemented it using higher-order functions defined on the tree. In particular, I use the three higher-order functions: TreeMap, TreeFold [46] and TreeUnfold, the latter two being illustrated in Figure 9.10.

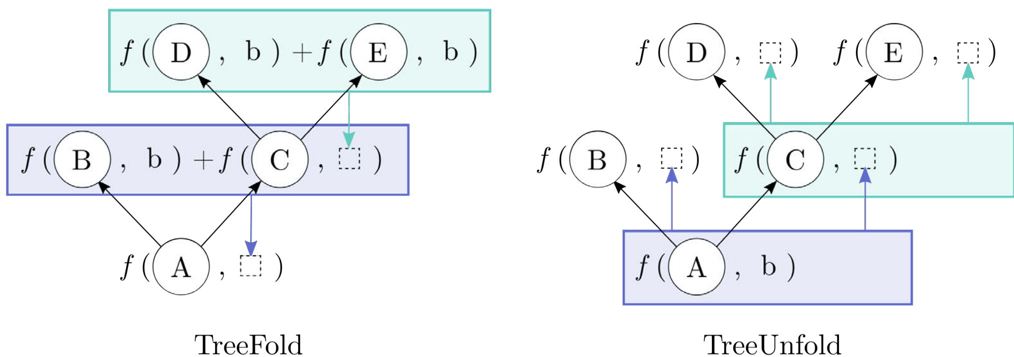


Figure 9.10: Illustration of TreeFold and TreeUnfold higher-order functions applied to a tree. f is the function applied to the nodes, "+" is used in a fold to combine children, and "b" is the base value for f.

TreeMap is the simplest of the three, and applies a function to every node on the tree. TreeFold propagates information from the tree's terminal internodes to its root. It takes two functions as arguments: a function `f` which operates on a node using information obtained from its children, and a function "+" that combines the return values of `f` for the children of a node.

TreeUnfold propagates information from the tree's root to its terminal internodes. A function `f` operates on each node of the tree, sending the return value of its execution to its parent as an additional argument.

Having defined the programming constructs I have used, I will now describe each function in Algorithm 9.2.

RemoveMarkersInOccupationZone maps a function to each node which iterates

over the list of markers, flagging any that are within the node's occupation zone for removal. LocateActiveBuds maps a function over the tree which identifies any buds, represented by null pointers, which have sufficient vigor to be activated. AssignMarkersToBuds iterates over the list of markers, and finds the closest bud that has it within the bud's perception volume. GrowInternodes adds the buds which have had a marker assigned to them to the tree, and grows them according to the influences in subsection 9.4.1.

PropagateShadow uses Algorithm 9.1 to produce a pyramid of shadow underneath the endpoint of each new internode on the tree. AccumulateLight uses a TreeFold to accumulate the light from buds to the tree's root. DistributeVigor calculates the tree's vigor from the accumulated light at the root, and uses a TreeUnfold to distribute it throughout the tree according to Equation 9.7. ShedInternodes removes internodes on the tree which have less vigor than their number of children. CalculateDaVinciRadius uses a TreeFold to evaluate Equation 9.15 for each node, by having f return each branch's squared radii.

In Tree Wand, this algorithm is run on a separate thread from the rendering, as in section 5.7, to prevent the simulation time from affecting the frame rate.

As the end result of each growth step, this algorithm produces a tree expressed using the data structure in section 9.2. While this describes the structure of the tree's internodes, further steps must be taken to visualize it.

Chapter 10

Tree appearance

10.1 Branch modeling

10.1.1 Skeleton generation

The algorithm outlined above generates a network of nodes connected by internodes which represent the tree's form. Additional steps need to be taken to transform the collection of segments into something more closely resembling a tree.

The first issue is that the junctions where internodes meet can have a sharp and jagged appearance. I've chosen to address this by creating a Bézier curve around each node, using the node's position, and the midpoints of the internodes on either side as the control points. This generates a set of composite Bézier curves with G_1 continuity for any path from root to leaf. There is, however, only G_0 continuity between internodes which share a parent.

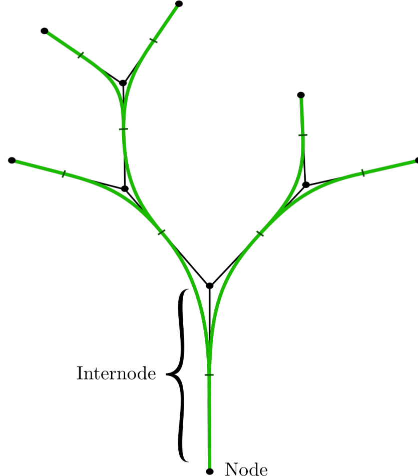


Figure 10.1: The smoothing of internode connections using rational Bézier curves. A quadratic rational Bézier curve is constructed around every corner between a point halfway along the first internode, the node between, and a point halfway along the second internode.

Depending on the length of internodes, and the tree being modeled, a Bézier curve may result in too gradual of a bend. Using a rational Bézier curve instead allows the sharpness of the bend to be controlled by increasing the central control point's weight relative to the other two.

We can express the rational Bézier curve using the equation:

$$\boldsymbol{N}(u) = (1-u^2)\begin{bmatrix}\frac{1}{2}(\mathrm{P}_{i-1}+\mathrm{P}_i)\\ 1\end{bmatrix} + 2(1-u)u\begin{bmatrix}\mathrm{P}_i\\ w\end{bmatrix} + u^2\begin{bmatrix}\frac{1}{2}(\mathrm{P}_i+\mathrm{P}_{i+1})\\ 1\end{bmatrix}, \tag{10.1}$$

for $u \in [0,1]$, where P_i is the position of the node at the junction of the two internodes, and w is the weight assigned to the center point. $\boldsymbol{N}(u)$ is a point defined in homogenous coordinates, and can be projected back to Cartesian space by dividing by the fourth coordinate.

10.1.2 Mesh generation

Figure 10.2: Two photos of trees depicting the shape of branching points. (Left) A photo of a simple branching pattern. The region where the two branches merge is only impacted by the trunk and the branch. (Right) A photo of a complicated branching pattern. Several branches impact the way their neighbours merge into the trunk.

In creating an accurate mesh around an arbitrary tree structure, there are a number of challenges. Many complex arrangements must be supported to be capable of capturing the variety of forms seen in nature. For branches with small radii, the branch has very little effect on the shape of its parent. However, a branch with a large radius integrates into its parent over a wide region. In Figure 10.2(Left), the large branch blends into the trunk most of the way to the bottom of the image, however the small branches coming out of it cause little deviation to its shape.

A method for generating a mesh for the branching point on the left of Figure 10.2 was provided by Bloomenthal et al. [10] by using a “ramiform”, a parameterization of a

topological prism. However, this method only works when a branching point has three points of entry; any other branches within the region of the ramiform's parameterization, as in Figure 10.2(right), would cause it to fail.

MacMurchy [66] presented a method of using low-polygon templates, followed by subdivision. This method could handle a larger number of arrangements than that by Bloomenthal et al., but would still struggle with modeling the branching in the right image of Figure 10.2. It also would incur a performance cost compared to just using generalized cylinders, especially considering that any addition of branches downstream will update the a branch's radius requiring otherwise unchanged branches to be recomputed, requiring the newly subdivided geometry to also be uploaded to the GPU.

With efficiency being particularly important, and no clear efficient solution found that handles all cases nicely, I have chosen to solely use generalized cylinders to represent the tree. In a section 10.3 I demonstrate how they can be transferred to the GPU with a minimal amount of data required.

A generalized cylinder is a parameterized surface generated by sweeping a planar curve defining the cross section along a *carrier curve* determining its path. In the context of modeling tree internodes, I use a circle as the cross section and transport it along the tree's skeleton. As a drawback, this requires multiple overlapping meshes at locations where an internode has more than one child.

A generalized cylinder is parameterized by two values, u for the position along the rational Bézier curve, and v for the position on the circle. The center of the circle lies along the skeleton, and can be evaluated using $\mathrm{N}(u)$.

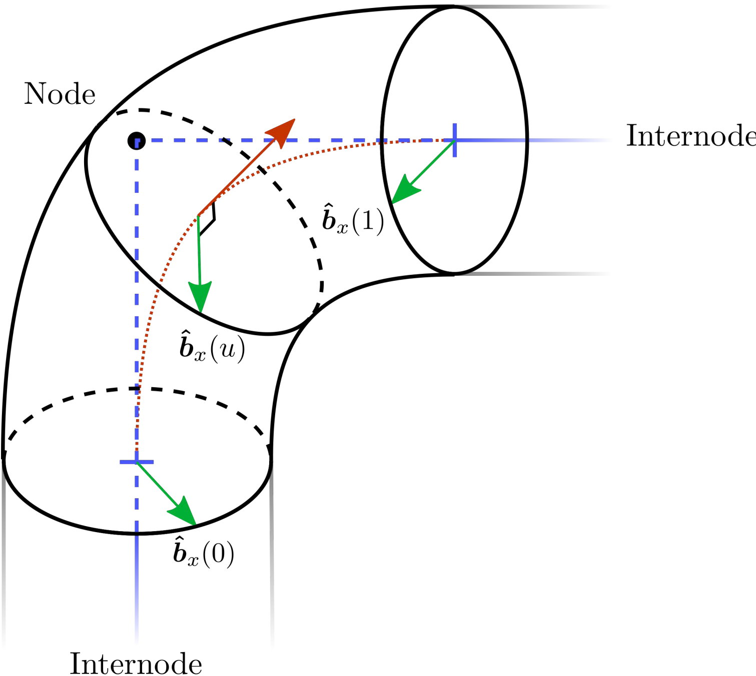


Figure 10.3: A branch modeled using a generalized cylinder consisting of a circle swept along a quadratic Bézier curve. The vector $\hat{\boldsymbol{b}}_x(u)$ (green) along with the tangent (red) are used to generate a frame for the circle along the curve.

To frame the circle along the curve, I orient it perpendicular to the curve's tangent, which can be calculated through calculating the derivative (either numerically or analytically) of the curve at u.

Aside from the plane it lies on, two other basis vectors are needed to frame the curve: $\hat{\boldsymbol{b}}_x$ and $\hat{\boldsymbol{b_y}}$. With the constraint that the three basis vectors are all orthogonal to each other, it suffices to specify one of $\hat{\boldsymbol{b}}_x$ or $\hat{\boldsymbol{b}}_y$ to build a unique frame. While a circle is rotationally invariant, when it is discretized, the ends of the generalized cylinder should line up with the previous and next cylinders to avoid gaps between internodes. In addition, the twist around the cylinder should be minimized to keep the cylinder looking smooth at low triangle counts.

Frenet frames use a curve's tangent and acceleration as the two basis vectors that define

its orthogonal frame, and have been applied in the modeling of trees by Bloomenthal et al. [10]. However, there are several limitations to Frenet frames: the frame becomes undefined along straight lines and inflection points, the latter of which will cause the frame to undergo an instantaneous rotation [85]. Given that the branching structure I've defined is only G_1 continuous at boundaries, this frame would become discontinuous between every segment of the spline.

Another method of framing curves is using a parallel transport frame [40, 85], which transports a frame along the curve such that its heading aligns with the tangent of the curve and it does not twist. This frame has also also been used extensively in the modeling of trees [85, 89, 62]. Typically this frame is calculated by sequentially stepping along the curve in small increments and rotating around the right and up vectors to update the frame at each point. However, this method conflicts with optimizations I perform in which the vertices of a generalized cylinder segment are calculated in parallel (section 10.3).

By taking advantage of the fact that a quadratic rational Bézier spline lies on a plane, it is possible to calculate the parallel transport frame at any point along a curve segment with a single rotation, provided an initial frame at N(0). On a planar curve, any rotation of the tangent must be with respect to the axis perpendicular to the plane, otherwise the curve would leave the plane. As all of the rotations occur around the same axis, they are no longer order dependent. As a result, the rotation to update the frame for an arbitrary point $0 \leq u \leq 1$ along the curve can be be combined into a single rotation of an angle equal to that between $\boldsymbol{N}'(0)$ and $\boldsymbol{N}'(u)$.

Rodrigues' rotation formula can be used to rotate a vector about an axis:

$$\hat{\boldsymbol{v}}_f = \hat{\boldsymbol{v}}_i \cos(\theta) + (\hat{\boldsymbol{k}} \times \hat{\boldsymbol{v}}_i) \sin(\theta) + \hat{\boldsymbol{k}}(\hat{\boldsymbol{k}} \cdot \hat{\boldsymbol{v}}_i)(1 - \cos(\theta))\,, \tag{10.2}$$

where $\hat{\boldsymbol{v}}_i$ and $\hat{\boldsymbol{v}}_f$ is the vector before and after the rotation, and with $\hat{\boldsymbol{k}}$ being the axis of rotation and θ being the angle. We can use this to rotate the frame's $\hat{\boldsymbol{b}}_x$ basis vector:

$$\cos(\theta) = \hat{\boldsymbol{N}}'(0) \cdot \hat{\boldsymbol{N}}'(1), \tag{10.3}$$

$$\sin(\theta) = \sqrt{1 - \cos(\theta)^2}, \tag{10.4}$$

$$\hat{\boldsymbol{k}} = \frac{(\mathrm{P}_1 - \mathrm{P}_0) \times (\mathrm{P}_2 - \mathrm{P}_1)}{\|(\mathrm{P}_1 - \mathrm{P}_0) \times (\mathrm{P}_2 - \mathrm{P}_1)\|}, \tag{10.5}$$

$$\hat{\boldsymbol{b}}_x(u) = \hat{\boldsymbol{b}}_x(0)\cos(\theta) + \left(\hat{\boldsymbol{k}} \times \hat{\boldsymbol{b}}_x(0)\right)\sin(\theta) + \hat{\boldsymbol{k}}(\hat{\boldsymbol{k}} \cdot \hat{\boldsymbol{b}}_x(0)(1 - \cos(\theta)). \tag{10.6}$$

This does still require the frame at the start of the curve segment, N(0). While its individual curve segments each lie on a plane, the composite rational Bézier spline does not. The frame at the start of each curve segment must be calculated sequentially, updating the frame from one curve segment to the next by rotating the frame by the angle between $\boldsymbol{N}'(0)$ and $\boldsymbol{N}'(1)$ on that segment.

To calculate the final basis vector for the frame, we can use the cross product

$$\hat{\boldsymbol{b}}_y(u) = \hat{\boldsymbol{b}}_x(u) \times \hat{\boldsymbol{N}}(u). \tag{10.7}$$

The remaining variable to determine is the radius of the cylinder. Since the radius will vary across the tree, the generalized cylinder will have to blend between internodes with different radii. While a linear blend could be used, this would create C_1 discontinuity where the general cylinders join. The smoothstep function based on a cubic hermite curve has a derivative of 0 at the beginning and end, which will maintain C_1 continuity.

$$s(u) = \begin{cases} 0 & \textbf{if } u < 0 \\ 3u^2 - 2u^3 & \textbf{if } 0 \leq u \leq 1 \\ 1 & \textbf{if } u > 1 \end{cases}, \tag{10.8}$$

which can be used to blend the two radii:

$$r(u) = (1 - s(u))r_i + s(u)r_{i+1}, \tag{10.9}$$

where r_i and r_{i+1} are the radii of the first and second internode the generalized cylinder is defined on.

With the all of the elements together, we have the equation for the generalized cylinder:

$$\mathrm{GC}(u,v) = \mathrm{N}(u) + r(u)\cos(\theta v)\hat{\boldsymbol{b}}_x(u) + r(u)\sin(\theta v)\hat{\boldsymbol{b}}_y(u) \tag{10.10}$$

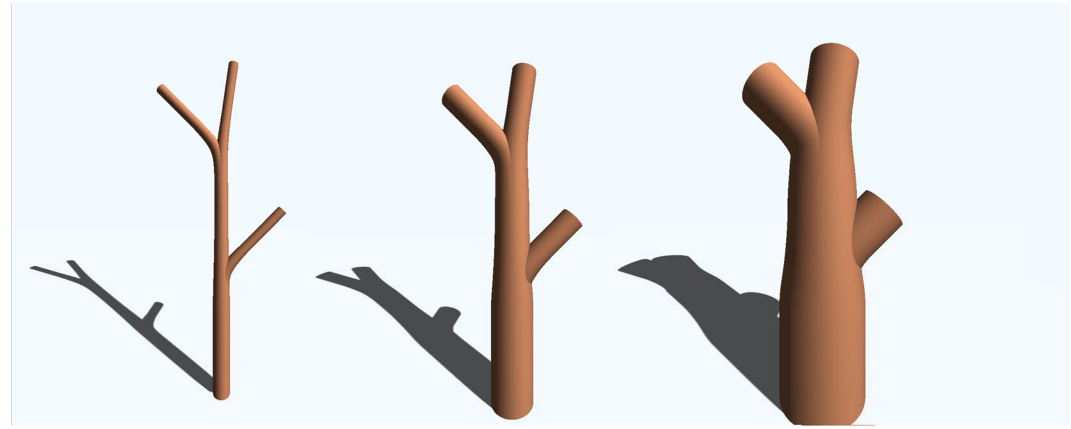


Figure 10.4: Progression of increasing branch radius left to right. Generalized cylinders work well with smaller radii, but result in a less smooth appearance with larger radii with the intersection becoming more apparent.

Using generalized cylinders in this manner to model trees results in a treelike appearance when the ratio between the radius and internode length is small. However, when it is larger, the intersections between the cylinders become more apparent with a smaller region to blend as shown in Figure 10.2. Figure 10.4 depicts a simple tree constructed using generalized cylinders with increasing radii. While not ideal, I consider this to be an acceptable approximation.

10.1.3 Texturing

There are several techniques available for texturing branching objects. Solid textures are 3D textures in which the the object is embedded [80]. When a part of the object is being shaded, its 3D position is to calculate the coordinates into the texture. Unfortunately solid textures

work best for objects which are isotropic or uniformly anisotropic in the global coordinate system, whereas a tree's bark follows the direction of the internodes.

Texture mapping is an approach which maps a two dimensional texture onto the surface of an object [42]. This mapping is defined by texture coordinates assigned to the vertices of a mesh, which denote from where on the texture the vertex receives its color. The texture coordinates inside faces are defined by a affine combination of the texture coordinates of the vertices, typically weighted by barycentric coordinates. I have chosen to use texture mapping to apply a bark texture to the tree, because it allows the texture to follow the directionality of the internodes, performs quickly, and can be easily applied to parametric shapes.

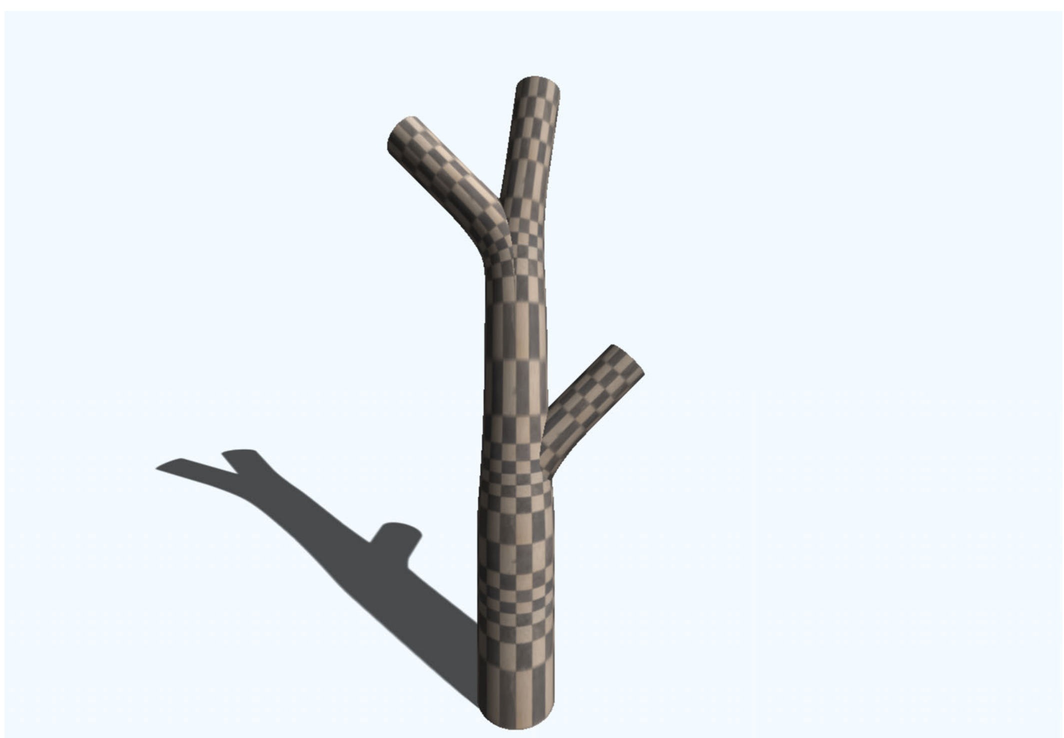


Figure 10.5: Texture coordinates assigned using u and v parameters from generalized cylinder. The checkerboard texture is non-uniformly stretched across the generalized cylinders.

Using two different parametric shapes to construct the generalized cylinder gives a natural starting point for texture coordinates. Using the parameters directly works for the v

coordinate parameterizing the circle, however Bézier splines are not arc length parameterized — with rational Bézier splines being even less so — resulting in the texture being stretched vertically non-uniformly as seen in Figure 10.5.

To texture the cylinder without unnecessary stretching, it is necessary to have a function determining the distance s along a curve given the parameter u, or a method for determining u given s

$$s = d(u) \tag{10.11}$$

$$u = d^{-1}(s)\,. \tag{10.12}$$

There are methods for approximating the arc length of a rational Bézier spline using optimization [15] and iterative methods [38], however no closed form solution. Due to the frequency this quantity needs to be evaluated, as well as optimizations in later sections which require it to be computed in parallel for multiple u values on a single curve, a less computationally expensive method is desired which doesn't require access to the arc length caclulated at previous points along the spline.

While Bloomenthal et al. [10] provided a means of texturing branching points through different parameterizations of the ramiform, that is impossible in this approach due to the lack of continuity between the children of a branching point, and continuity along the each linear path down a tree is the best that can be achieved.

Being that the texture coordinates are solely for visual effect, I've developed a rough, yet efficient approximation for Equation 10.11. This method is based on the observation that the spacing of a quadratic rational Bézier spline changes a relatively small amount depending on the angle between the 3 points. The magnitude of a rational Bézier spline's derivative with an angle of 180°between equal length segments at $u = 0.5$ changes by 7% at an angle of 135°, 43% at 90°, and 160% at 45°. High branching angles are uncommon in tree models, particularly those greater than 90°, which would minimize the error's impact.

The strategy uses a 1D rational Bézier spline as a means to estimate the segment length

of a curve regardless of its control points' configuration. With a 180°angle between segments, this method provides exact solution, becoming less accurate the further it bends. We can define the spline such that its three control points lie at 0, $\|P_1 - P_0\|$ and $\|P_2 - P_1\|$ with the following equation:

$$D(u) = \frac{2u(1-u)w\,\|P_1 - P_0\| + u^2\,\|P_2 - P_1\|}{(1-u)^2 + 2*(1-u)uw + u^2}\,, \tag{10.13}$$

where w is the weight of the central control point. This function returns a scalar value which is used to approximate $d(u)$ in Equation 10.11.

To prove that this forms an exact solution when there is a 180°angle between segments, suppose that the curve has been been translated and rotated such that its first control point is at the origin, and the remaining two control points are along the x-axis. Translation and rotation do not affect the length of the curve because it is an affine combination of its control points. The equation for the curve will then be:

$$B(u) = \begin{bmatrix} \frac{2u(1-u)w\|P_1-P_0\|+u^2\|P_2-P_1\|}{(1-u)^2+2(1-u)uw+u^2} \\ 0 \\ 0 \end{bmatrix} \tag{10.14}$$

The length of the curve at any point will be equal to the x coordinate, which is equal to Equation 10.13. This formula applied to various control point configurations can be seen in Figure 10.6.

Length ratio = 1		Length ratio = 1.5	
(a)	(b)	(a)	(b)

Figure 10.6: Spacing for a rational Bézier curve with a weight of 3 for the center point. (a) Directly using the u parameter from Equation 10.10. (b) Using Equation 10.13.

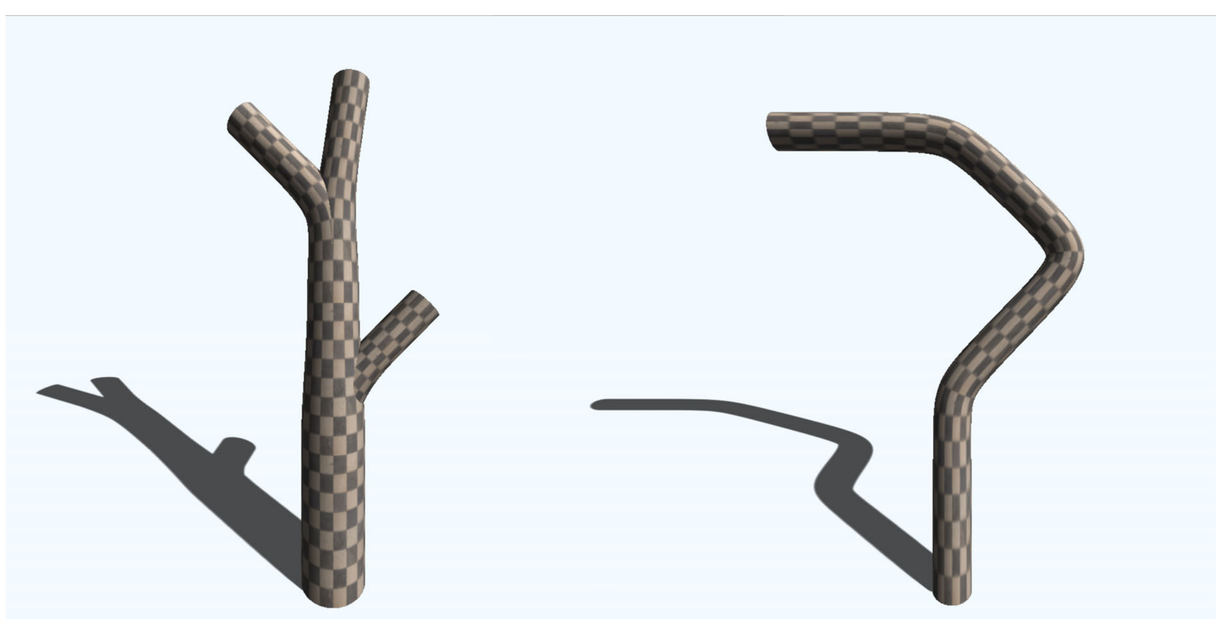


Figure 10.7: Using the formula from Equation 10.13 to assign texture coordinates. (Left) The model from Figure 10.5 instead (Right) A linear general cylinder.

10.2 Leaves

10.2.1 Leaf positioning

In nature, leaves appear below the lateral buds on a new shoot when it first develops [110]. However, in the context of a tree modeled procedurally by a user in Tree Wand, it is not apparent when each internode on the tree was developed. I have chosen to adopt a similar approach to that used in TreeSketch [63], by positioning the leaves beneath buds on internodes with a radius that is below a certain threshold. In addition to lateral buds, I also place leaves on the main buds. While this is not biologically accurate, it results in a fuller appearance for the tree.

In Tree Wand, I give a leaf placed below a lateral bud the same initial orientation that an internode produced by that bud would have.

10.2.2 Leaf orientation

Leaf orientation has a significant impact on a tree's appearance. The orientation of leaves affects the amount of light that they are able to absorb, with a vertical orientation of the leaf normal allowing it to absorb the most light [106, 31]. However, this is not always advantageous, as too much light absorption can result in a leaf overheating [31]. In addition, the less vertical the leaf's orientation in the canopy, the deeper light penetrates to a tree's lower leaves [51], which can explain why leaves at the top of a tree's canopy tend to be less vertically oriented than those lower down [86]. The inclination of leaves also varies between species, with some trees such as *Alnus incana* having a strong bias towards vertically oriented leaf normals, whereas trees such as *Betula pendula* are distributed much closer to horizontal [86]. These distributions are not solely dependent on the leaf's initial orientation; a leaf can reorient itself to absorb more or less light through the bending and twisting of its petiole [76, 75].

Let a leaf's initial orientation be described by a heading vector, $\hat{\boldsymbol{h}}$, right vector, $\hat{\boldsymbol{r}}$, and up vector $\hat{\boldsymbol{u}}$ which corresponds to the leaf's normal.

Longay et al. [62] presented a method for orienting leaves using a set of torques applied to each axis, which direct the leaf's normal vector closer to a tropism vector at a specified angle from vertical. These torques are applied over a number of iterations to obtain the final orientation.

The method I have chosen operates similarly to that used in Longay et al., and is based on a model designed by Przemysław Prusinkiewicz in Virtual Laboratory [32].

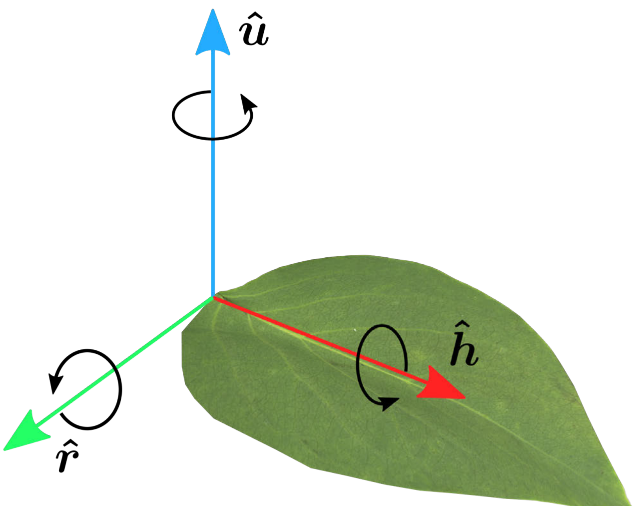


Figure 10.8: Rotations around the axes of a leaf. Rotation around $\hat{\boldsymbol{h}}$ occurs when the petiole twists, rotation around $\hat{\boldsymbol{r}}$ results when it bends up and down, and rotation around $\hat{\boldsymbol{u}}$ occurs when it bends left and right.

I reorient the leaf to bring $\hat{\boldsymbol{u}}$ closer to vertical using three rotations: one about the leaf's $\hat{\boldsymbol{h}}$ vector to represent twisting of the petiole, and two about its $\hat{\boldsymbol{r}}$ and $\hat{\boldsymbol{u}}$ vectors to represent bending of the petiole. While the bending rotations around $\hat{\boldsymbol{r}}$ and $\hat{\boldsymbol{u}}$ could be combined into one, a petiole's cross section is not circular, and its ability to bend around each axis may differ. Keeping the two rotations distinct allows for the magnitude of the rotations around each axis to be controlled separately.

To determine the magnitude of rotation in each circumstance, I use two factors: how far away the target basis vector, $\hat{\boldsymbol{b}}$, is away from the vertical axis, $\hat{\boldsymbol{y}}$, and how much rotating around the rotation axis will bring the vector closer to vertical. A measure of the first can be made by using $\left\|\hat{\boldsymbol{b}} \times \hat{\boldsymbol{y}}\right\|$. This results in a value that increases in magnitude as the angle between $\hat{\boldsymbol{b}}$ and $\hat{\boldsymbol{y}}$ increases, up to 90°, after which it will decrease. The decrease may be justified by the fact that the bottom side of the leaf is also able to absorb light, though less effectively than the top [75].

I use the same metric to weight the rotation based on the orientation of the rotation axis, $\hat{\boldsymbol{v}}$. When $\hat{\boldsymbol{v}}$ is perpendicular to $\hat{\boldsymbol{y}}$, $\hat{\boldsymbol{b}}$ can be directly rotated to $\hat{\boldsymbol{y}}$. In contrast, when they are parallel, $\hat{\boldsymbol{b}}$ will be constrained to the planeperpendicular to $\hat{\boldsymbol{y}}$. Modulating the magnitude of rotation by $\|\hat{\boldsymbol{y}} \times \hat{\boldsymbol{v}}\|$ will result in no rotation when rotating around that axis does nothing, and the most rotation when it has the largest effect.

Combining these factors together we obtain the angle of rotation by the following equation:

$$\theta = e \left\| \hat{\boldsymbol{b}} \times \hat{\boldsymbol{y}} \right\| \|\hat{\boldsymbol{v}} \times \hat{\boldsymbol{y}}\| \ , \tag{10.15}$$

where e is the elasticity of the rotation around that axis. The elasticity of each axis is user controllable parameter, with an elasticity of 0 disabling that rotation.

We can use this formula for our three rotations:

- A bending rotation around the $\hat{\boldsymbol{u}}$ axis making $\hat{\boldsymbol{h}}$ as vertical as possible,
- A bending rotation around the $\hat{\boldsymbol{r}}$ axis making $\hat{\boldsymbol{u}}$ as vertical as possible, and
- A twisting rotation around the $\hat{\boldsymbol{h}}$ axis making $\hat{\boldsymbol{u}}$ as vertical as possible.

The first rotation does not directly reorient $\hat{\boldsymbol{u}}$ to vertical, however it reorients the leaf to make the $\hat{\boldsymbol{r}}$ vector more horizontal, allowing the second rotation to rotate $\hat{\boldsymbol{u}}$ closer to vertical.

To remove the impact of the order of rotations, they are done iteratively in small increments, updating the frame after each rotation. The number of iterations is a modifiable parameter, and typically between 10 and 20 iterations are enough to produce results resembling those of leaves in nature.

The petiole is visualized using a generalized cylinder as in section 10.1. The first control point, p_0 lies at the bud's location, the second at $p_1 = p_0 + l\hat{\boldsymbol{h}}_{init}$ and the third

at $p_2 = p_1 + l\hat{\boldsymbol{h}}_{final}$, where l is a parameter controlling the distance between control points. While the path the petiole would have taken while bending is unlikely to lie on this plane, due to the number of leaves on the tree this difference is difficult to notice.

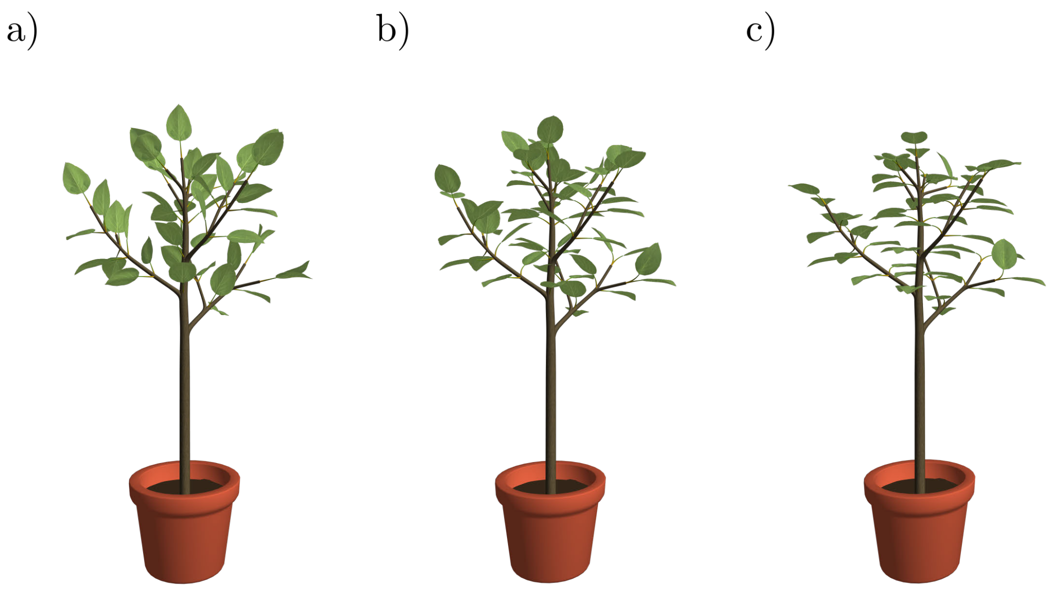


Figure 10.9: Leaf reorientation with different elasticities. (a) Leaves with initial orientations (elasticity 0). (b) Leaves with elasticity 0.05 for all rotations. (c) Leaves with elasticity 0.1 for all rotations

In contrast to Longay et al's [62] approach, the method I have used modulates the rotation around each axis by a factor representing how closely that axis can rotate the target vector to vertical. However, this comes at the cost of allowing non-zero tropism angles. Extending this method to allow for arbitrary tropism angles requires further investigation.

10.3 GPU acceleration

While in ViNE the geometry being rendered doesn't change, in Tree Wand, both the geometry and the number of triangles are frequently changing. New triangles are constantly

added and removed with the addition and shedding of internodes, and old geometry is updated with changing radii according to Da Vinci's rule (section 9.6). Not only is it computationally expensive to generate the geometry, but the modified data can span large regions in the geometry buffers and thus require a large amount of bandwidth to be transferred from the CPU to the GPU. Kohek et al. [52] found that when implementing self-organizing tree models on the GPU, that the data transfer time between the CPU and GPU was a significant factor in performance.

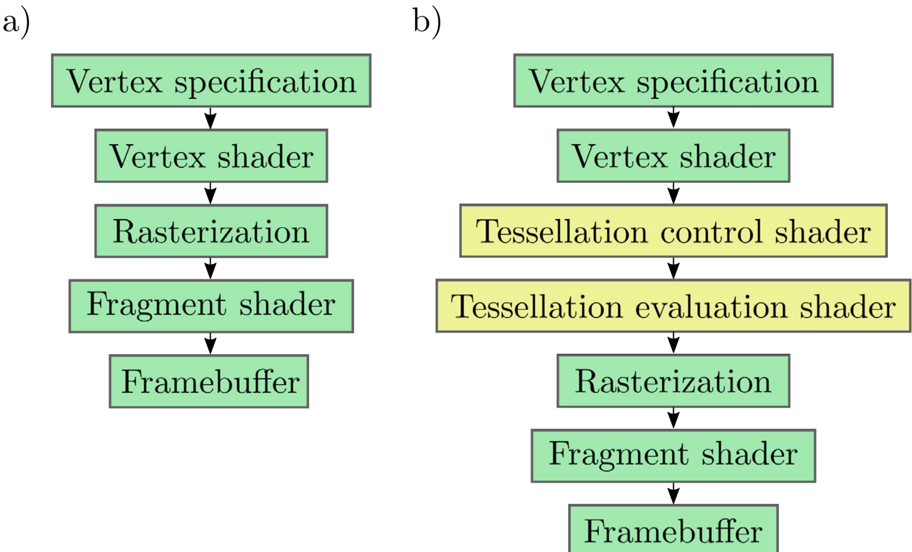


Figure 10.10: OpenGL graphics pipleines. (a) A simplified version of the OpenGL pipeline as of OpenGL 4.6. (b) The OpenGL pipeline when using tessellation

Using OpenGL's tessellation shaders, I can significantly reduce the amount of information required to be sent to the GPU. The tessellation shader constructs surfaces from input "patches" of vertex information, which are used to determine how to map a 2D abstract patch — which can have the topology of a subdivided line, triangle or quad — into an output patch defined in 3D space. The tessellation pipeline includes two new stages compared to the standard OpenGL 4.6 pipeline: the tessellation control shader, and tessellation evaluation shader. The tessellation control shader receives an array filled with the output from the vertex shader invocations for each vertex in its associated input patch. This shader is

responsible for specifying the level of subdivision of the output patch, as well as forwarding any necessary information to the next stage of the pipeline. The tessellation evaluation shader is invoked separately on each vertex in the abstract patch, and is responsible for evaluating the position of that vertex on the output patch. Each invocation of this shader is provided the unique coordinates of its corresponding vertex in the abstract patch, for a quad this ranges from (0, 0) in the bottom left to (1, 1) in the top right.

The abstract patch coordinates can be input directly into Equation 10.10 to map the abstract patch into the shape of a generalized cylinder. The remaining components of the formula can be passed in through vertex data. Each of the three vertices in the input patch store the position of one of the control points, with the first and third vertex also storing $\hat{\boldsymbol{b}}_x(0)$ and $\hat{\boldsymbol{b}}_x(1)$, while the middle vertex stores the w value of the middle control point, and the texture coordinate at the beginning and end of the cylinder. Using tessellation shaders in this way allows an entire general cylinder to be represented using 18 floating point values, or 64 bytes.

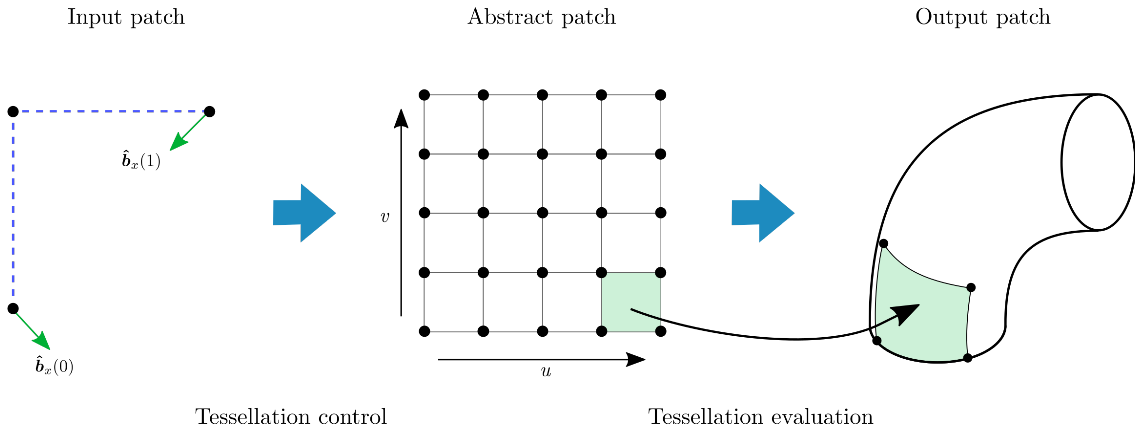


Figure 10.11: Using tessellation shader to create generalized cylinder. The tessellation control shader determines dimensions of the abstract patch, and the tessellation evaluation shader maps each vertex in the abstract patch to a point in clip space.

To keep the geometry looking smooth even when the branches become very thick, I use the tessellation control shader to linearly blend between the level of subdivision for the output patch based on the radius of its associate internodes. The level of subdivision can be set independently for the top and bottom edges of the quad, to prevent holes between adjacent patches. I use the radius of the first control point to determine the level of subdivision at the bottom edge of the output patch, and the radius of the third control point for the top edge. When the the control point's radius is equal to the tip radius, the edge is subdivided into 10, when it is 50 times the tip radius, the edge is subdivided into 64 — the maximum supported by tessellation shaders. I set the subdivision levels for the left and right edges of the quad to be the greater of the top and bottom's subdivision levels.

Chapter 11

VR interface design

Tree Wand makes it possible for the tree's development to be guided by the path of the "wand" — the trackable controller. When the user paints with the wand, it leaves behind a trail of "fairy dust" — the markers — towards which the tree will grow.

The algorithm I use for generating trees is controlled by a set of parameters which can be used to express a variety of tree forms. I have provided an interface to allow the user to manipulate these parameters from within virtual reality by interacting with three different types of widgets. The parameters are divided between three windows which can be placed around the user for easy viewing and interaction. A menu inspired by the color wheel in ViNE has also been provided to allow the user to select functions such as saving and loading models, as well as opening and closing windows.

Modeling in Tree Wand becomes much easier with the ability to translate, rotate, and scale the model. As the metaphor used for these interactions in ViNE [96] was both intuitive and easy to use, I have decided to continue using it as is in Tree Wand.

11.1 Distributing markers

The controller is used as a wand to paint attraction markers to represent the available space for the tree to grow. This is similar to the method used by Longay et al. [63] on an iPad, in which the markers were sketched using the touch screen, however, since the controllers can be tracked in three dimensions, it is not necessary to rotate the view to paint at a different depth. In addition, the enhanced perception of depth in virtual reality provides a better impression of the tree's shape, making it easier to grow a branch from a specific location on the tree.

Figure 11.1: Using the brush to paint a new branch on a tree.

Having the ability to model the tree at different levels of detail was one of the main goals of Tree Wand. It is important to provide the user with the ability to directly specify a tree's shape when needed, as well as model more broadly when it's not. A way in which TreeSketch [63] achieved this was by modulating the size of the brush by the amount of pressure used on the touch screen.

Similarly I allow the user to control the level of precision of their modeling by modifying the wand's radius. The radius of the wand is modified in the same manner as in ViNE: for each clockwise circle the user makes with their thumb, the wand's radius doubles — halving when this is done counter-clockwise. With an Oculus Rift, the radius is instead modified with the analog control stick, increasing when pointed up, and decreasing when pointed down.

I use Poisson darts [22] to distribute the markers within the brush's radius, rejecting any points that fall within a certain radius of another marker. I set this radius to be equal to the radius of the occupation zone.

After the markers have been distributed with the wand, they remain for 60 growth steps

before they disappear.

11.2 Pruning

A modeler will often want part of the tree to be changed or removed; this may be due to user error, an evolving design, or the procedural elements of the modeling not generating the tree the user envisioned. To facilitate this, I provide a mode for pruning branches instead of growing them.

Figure 11.2: Pruning of a branch on the tree. A partial press of the trigger previews nodes that would be pruned, while a full press removes them.

While in pruning mode, the brush is colored red instead of green. When the trigger is fully pressed, any internodes with a midpoint that falls inside the brush's radius are removed; which will also remove any of their children as a result. While the trigger is partially pressed, Tree Wand previews the nodes that would be removed by coloring all of their associated branches red.

11.3 Controller menu

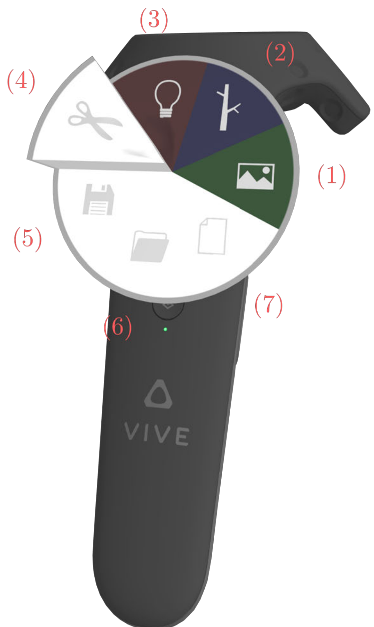


Figure 11.3: Controller menu functions counterclockwise from the green icon: (1) Appearance window (2) Tree form window (3) Light window (4) Turn pruning on/off (5) Save the current tree (6) Load saved tree (7) Start new tree.

The menu appears over the trackpad of the left controller, designed similarly to the color wheel in ViNE. Each menu option corresponds to a wedge in the wheel, labeled with an icon indicating its function. When the menu option is hovered over , its wedge is extruded upwards. Clicking on the trackpad activates the functionality of the associated wedge.

The red, green and blue menu options shown in Figure 11.3 toggle their associated parameter window open and closed. The color of the button matches the color of the parameter window to remove ambiguity. Parameter windows are described in the following section. The save button saves both the current layout and parameters of the tree to a file which stores the tree using the bracketed string representation [59] in a file named "default.tree". The load button loads the geometry and parameters from the same file.

Lastly, the clear button will remove all of the internodes from the tree, leaving the base.

11.4 Modifying parameters

11.4.1 Parameter windows

There is far more flexibility with where menus and windows are situated within virtual reality than there are within a 2D interface. A major consideration is what the window should be bound to. The simplest model is to have the window bound to nothing, in what is called a "floating menu" [105, 24]. A floating menu is often interacted with either directly by touch with a controller or the user's hand, or by using the controller as a "laser pointer" to point at the location being interacted with. Directly touching the menu however can become difficult when the menu is out of reach, which can make a pointer-based interface more ideal [24]. Floating menus are common interfaces, appearing a variety of virtual reality games such as Beat Saber [6] as well as being the main means of interaction with the HTC Vive's home menu.

An alternative is to have the menu bound to one of the user's controllers, and manipulated by touch with the other. This allows both controllers to work together to select an item on the menu, and subverts the issue with touching the floating menu, since the user's hand will always be within reach. A user study comparing these two menus determined that a menu bound to the user's hand could be manipulated more accurately [72].

While having the menu bound to a controller may be more accurate, I have chosen to adopt a floating menu. A floating menu can afford to be larger, allowing it to represent more information at a time. When manipulating the parameters of tree models, I found that I spent more time looking at the parameters, determining which should be changed, than I did changing them. The limited space of an controller bound window would make surveying the parameters more difficult. In addition, this is a form of interaction a user of the HTC Vive is likely to have been exposed to through the home menu, and they are more likely to

be familiar with using it.

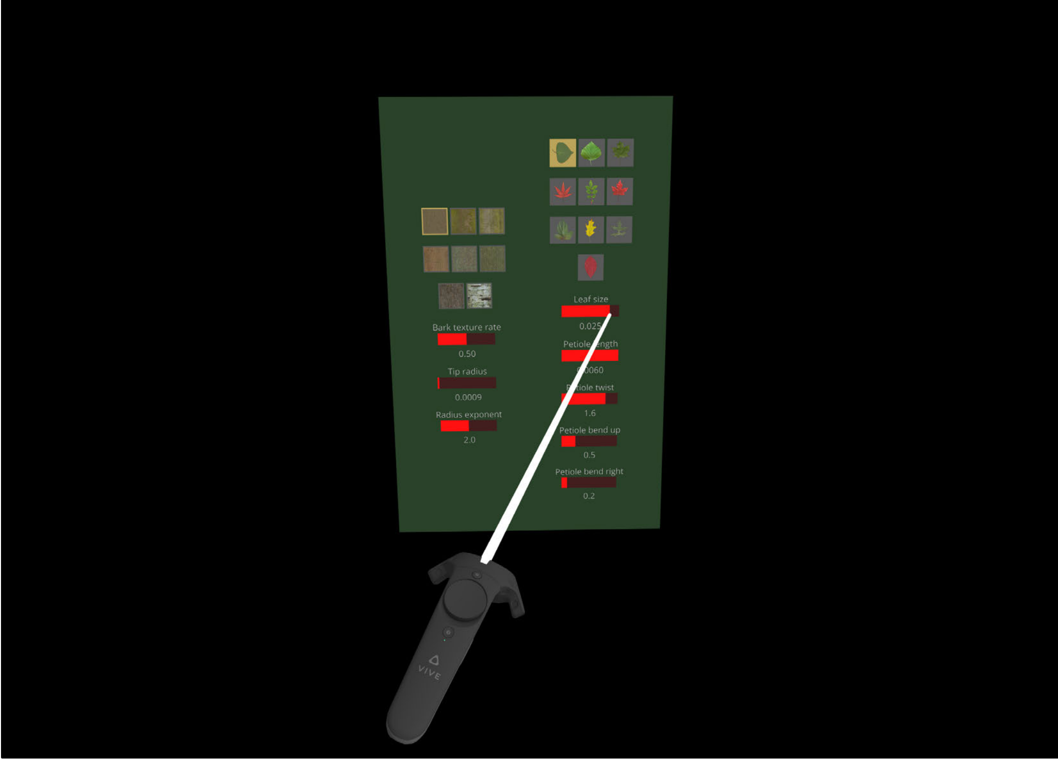


Figure 11.4: Controller interacting through the window by pointing at a slider.

The parameters for TreeSketch VR have been separated into three categories: appearance, tree form and shadowing. Each category of parameters appears in its own window, which can be opened and closed by selecting the corresponding wedge in the menu. When the window is opened, it is offset by one meter from the controller's position in the direction of the controller's heading, with an orientation such that it faces the controller as much as possible while keeping its up vector pointing up.

When a controller points at a window, a white ray is cast from the towards the window in the direction of the controller's heading. The end of the ray indicates cursor's position when the controller interacts with the window's components. Pressing the trigger button begins an interaction with the component the controller is pointing at, and releasing the trigger ends it. While the controller is pointing at the window, it cannot produce markers

or apply view transformations to the scene.

The windows can be repositioned by holding the grip button on a controller while pointing towards it, its new position is calculated the same as the initial placement: one meter ahead of the controller's position, with an orientation facing the controller.

There are three forms of components inside the windows: sliders, icon selectors, and the influence triangle.

11.4.2 Sliders

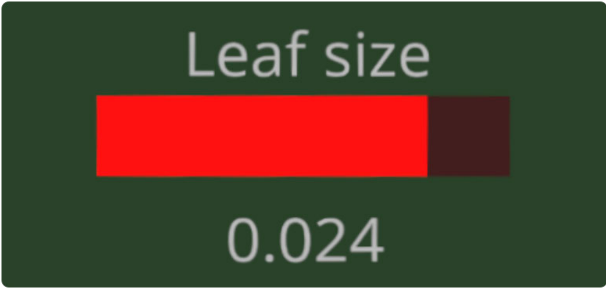


Figure 11.5: A slider controlling the leaf size.

A number of parameters are monitored by sliders, which consist of a label indicating its name, a red bar, and a label indicating the parameter's current value. When the red bar is selected, it can be moved between the left and right edge according to the cursor position.

11.4.3 Texture selectors

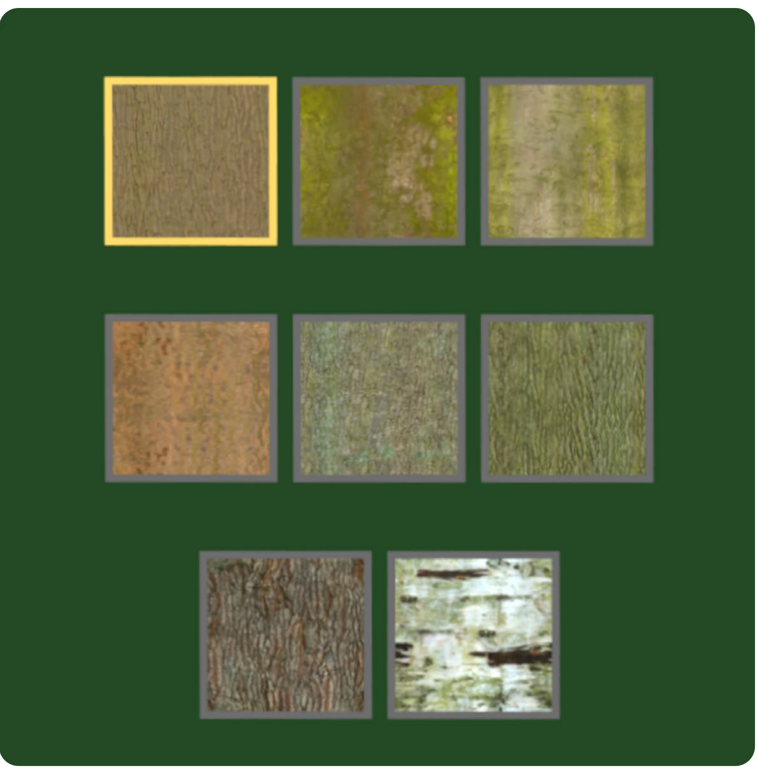


Figure 11.6: Icon selector for selecting the tree's bark texture.

Icon selectors display a grid of images which can be selected when hovered over with the cursor. The currently selected icon is indicated with a yellow highlight around the image. The icon selectors are used to choose the bark texture on the tree, as well as the type of leaves that will grow on it.

11.4.4 Influence triangle

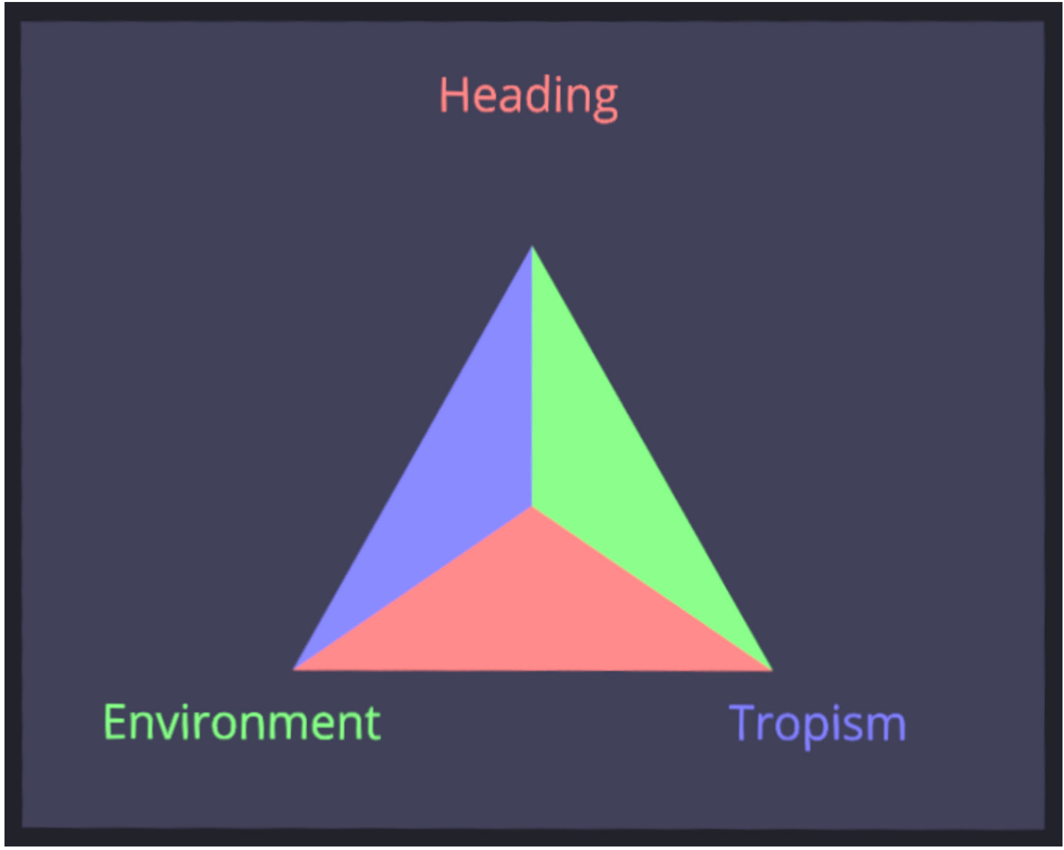


Figure 11.7: Triangle for setting influence weights. The area of each of the red, green and blue sub-triangles are proprotional to the weight of that influence.

The influence triangle is used to blend between the three influences affecting internode growth direction: heading, environment and tropism. A triangle is used here instead of a slider due to the relationship between the parameters in which increasing one will decrease the others.

The point at the center of the triangle is used to manipulate the three parameters, with the value assigned to each calculated from the barycentric coordinates of the point. The barycentric coordinate associated with a corner of the triangle in regards to a point inside it is equal to the area of the triangle formed between that point and the other two corners of the triangle divided by the area of the whole triangle. I use this property to visualize the weight of each influence as seen in Figure 11.7.

Chapter 12

Results

In this chapter I show examples of trees that I've modeled using Tree Wand and discuss how they have been created. I then illustrate the effect that the two different methods for determining internode length I presented in section 9.7 had on generating trees. Lastly, I close this chapter by analyzing the performance of Tree Wand and discussing the limitations on the size of trees it can produce.

12.1 Generating trees

Here I present a selection of trees I have modeled inside of Tree Wand. Each tree was modeled quickly, the fastest being finished in under a minute, while the longest took under 10 minutes.

Figure 12.1: A tree with a well defined main trunk with a tropism of 90 degrees enabled for the branches beyond the first order. Left: without leaves; Right: with leaves.

The tree in Figure 12.1 was generated with two motions of the wand. The vertical trunk was drawn first, then the brush was quickly swept up the trunk allowing each of the lateral branches to start around the same time. In addition to the modifiable parameters from the menu, timing of when markers appear can also play a large role in a tree's form. When this motion was made more slowly, the lowest branches consumed the attraction points and continued all the way to the top of the tree.

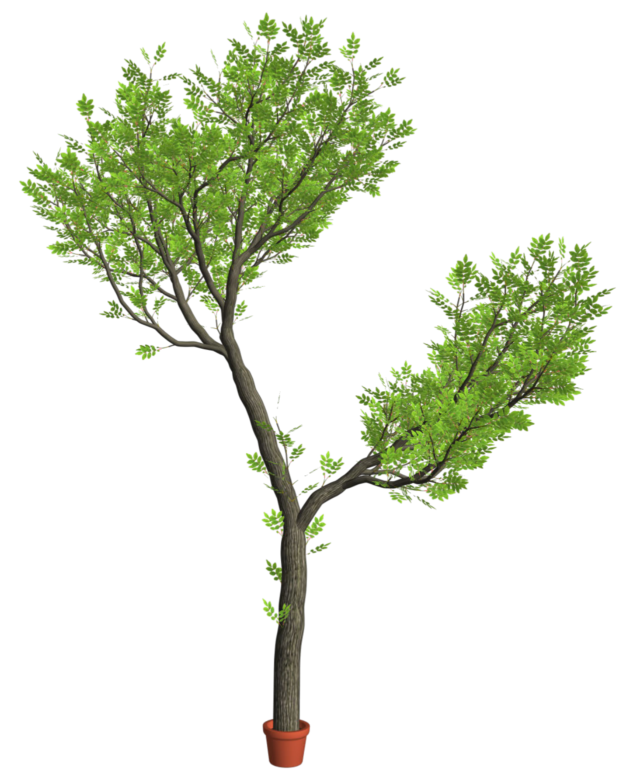


Figure 12.2: A tree with two well established main branches.

Using a smaller brush also enabled more precise control of how a tree branched. Figure 12.2 was created by drawing a "Y" shape with a small brush, followed by using a larger brush on the top. By decreasing the exponent for the shadow decay, the upper canopy casts a shadow inhibiting growth on the branch below.

Figure 12.3: A columnar tree.

Columnar trees can be created by narrowing the view angle and applying a vertical tropism. I also found that they required a sufficiently high environment influence to draw them towards empty space, without which their vertical branches overlap too much and become unrealistically dense and indistinguishable from one another.

Figure 12.4: A small tree with winding branches.

The tree in Figure 12.4 was created by setting a larger lateral and main branching angle, a tropism of 90°, a high environment weight, and an increased view angle of 180°. While modeling this tree I used very broad motions with the wand, filling the desired shape with fairy dust, leaving the tree generation algorithm to dictate most of the details. After the majority of the tree form was determined, I used a smaller radius with the wand to add more internodes to areas I wanted to be more developed.

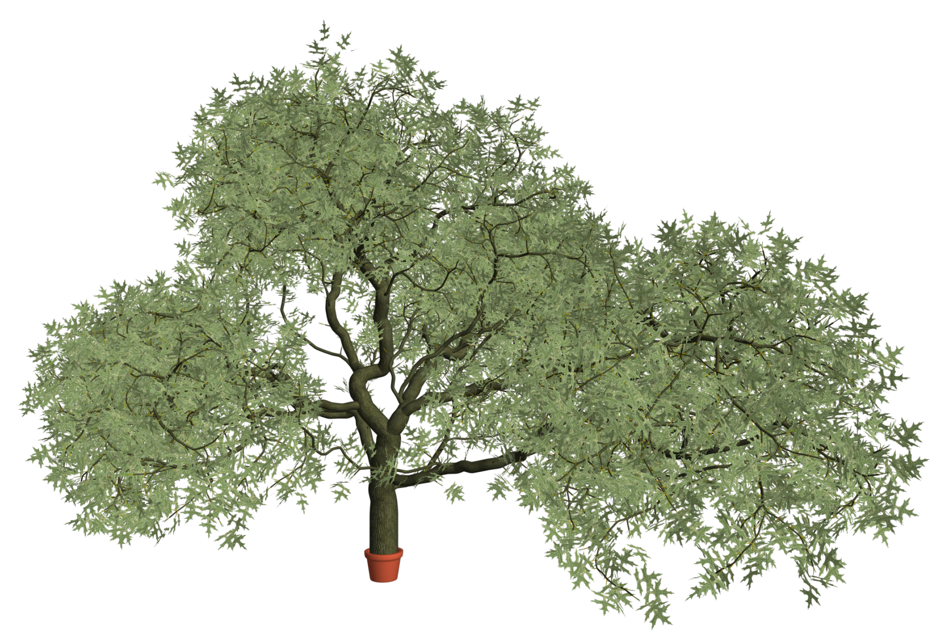


Figure 12.5: A large oak tree.

I used a similar set of parameters as in Figure 12.4 to model an Oak tree. To generate a different shape of tree, I set the λ value to 0.4, and decreased the influence of the horizontal tropism. I modeled this tree more deliberately than in Figure 12.4, using a broad brush to

grow the tree towards three concentrations of leaves.

I modeled the Manitoba Maple tree in Figure 12.6 as an example of a tree with opposite branching. I used the maple tree's lateral branching angle of 45°[110], and simulated the winding appearance of its main branches by setting a main branching angle of 12°, and increasing the environment influence. I started with an apical dominance of 0.6, and lowered it to 0.4 to create denser leaves at the tips.

I also explored the impact of the two methods I proposed for determining internode length in section 9.7 on trees generated with Tree Wand. In Figure 12.7 I show examples of trees generated by each method when using the wand in the same location above the root.

Both methods made it significantly easier to develop an established main trunk when compared to a constant internode length. Qualitatively, I found the difference between their results to be subtle, suggesting this formula effectively captures the phenomenon of decreasing intenode lengths in nature. For this reason, I decided to use the vigor model as the default for Tree Wand, using it to generate all of the results in this section.

The use of 3D trackable controllers helped immensely in modeling the trees in this chapter. When using Tree Wand I never experienced difficulties estimating the depth of the brush. Making small modifications to the tree becomes much easier when knowing the exact position of the brush, and reduces the chance of having an unintended branch grow towards the brush.

One of the most significant problems encountered while modeling trees using Tree Wand was the inability to grow a new branch in a highly shadowed region due to a combination of the buds having insufficient vigor to be activated, and shedding. This was of particular issue because the final branch once it has completed growing may not be shed as a result of leaves further out receiving more illumination. This complication arises from the fact that a tree's history in Tree Wand is not necessarily representative of the order in which the branches would have been grown in nature. Tree Wand facilitates a modeling approach where the user develops one part of the tree at a time, as opposed to iteratively adding internodes to every

Figure 12.6: Maple tree demonstrating opposite branching.

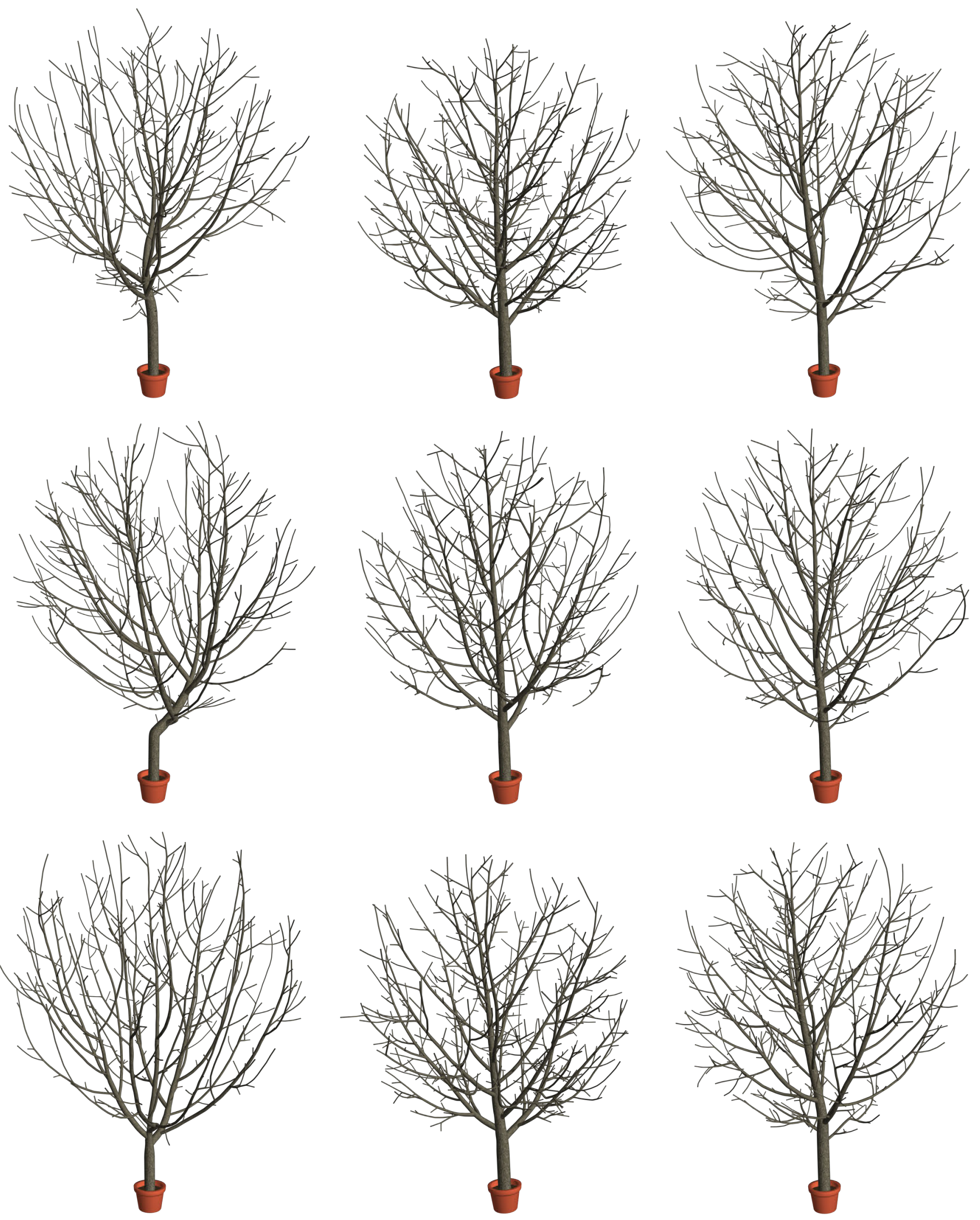


Figure 12.7: A comparison of trees generated from three different methods for determining internode length. All three sets of tests were made using $\lambda = 0.6$. Left: Constant internode length. Middle: Internode length based on vigor. Right: Internode length based on branch order.

region that may grow at once. This issue imposes inconvenient limitations on the order in which the user draws elements, which can limit the creative process.

12.2 Performance

Tree Wand was tested on a Windows desktop with an Intel® Xeon® CPU E5-1620 v3 processor running at 3.50GHz with 4 cores, a NVIDIA TITAN X (Pascal) graphics card, and 32GB of ram.

The frame rate of Tree Wand remains stable at 90 frames per second (fps) up to when the tree has 22,000 internodes, at which point occasional frames are missed and asynchronous reprojection is used to fill them in. When the maximum and minimum degrees of tessellation used in section 10.3 are reduced from 10 and 64 to 6 and 32, the frame rate doesn't drop until the tree has 32,000 internodes. When leaves are removed, the maximum number of internodes before a frame rate drop increases to 61,000.

Whether or not the wand is currently drawing has a minimal impact on the frame rate as most of the computational work is done on the non-rendering thread. When the tree has 40,000 internodes, the rendering thread is occupied for 5ms with the data transfer, however this data transfer occurs infrequently due to Tree Wand's real bottleneck.

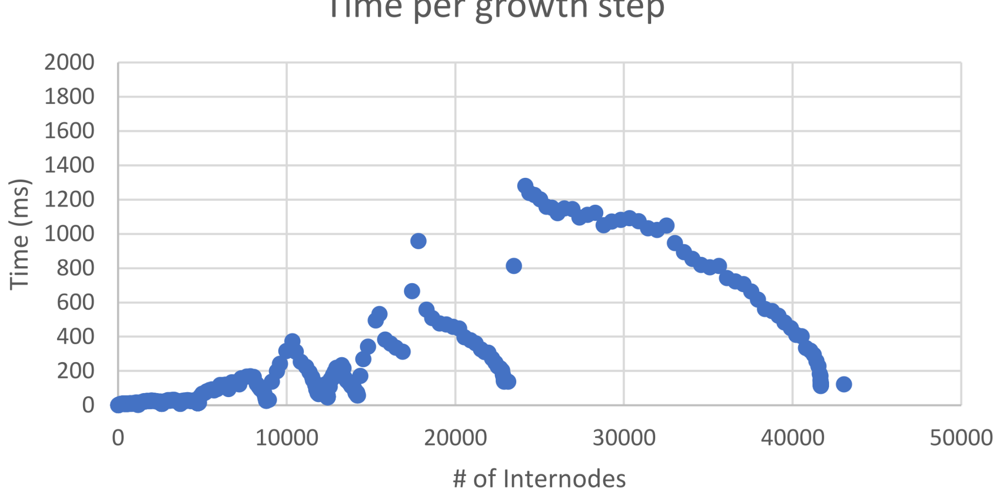


Figure 12.8: Graph relating the number of internodes to the time taken for a growth step.

While rendering times do not vary below 22,000 internodes, the amount time between the user painting markers and the tree generation thread finishing a new tree growth step becomes an issue at a smaller internode count. Figure 12.8 compares the number of internodes to the growth step duration. While the user will not suffer motion sickness like they would for rendering if the growth steps take more than 11ms, it becomes much less interactive when it falls below 30fps: 33ms per frame. This loss of interactivity begins to occur around 5000 internodes, but while Figure 12.8 shows a general increasing trend, it is not tightly correlated.

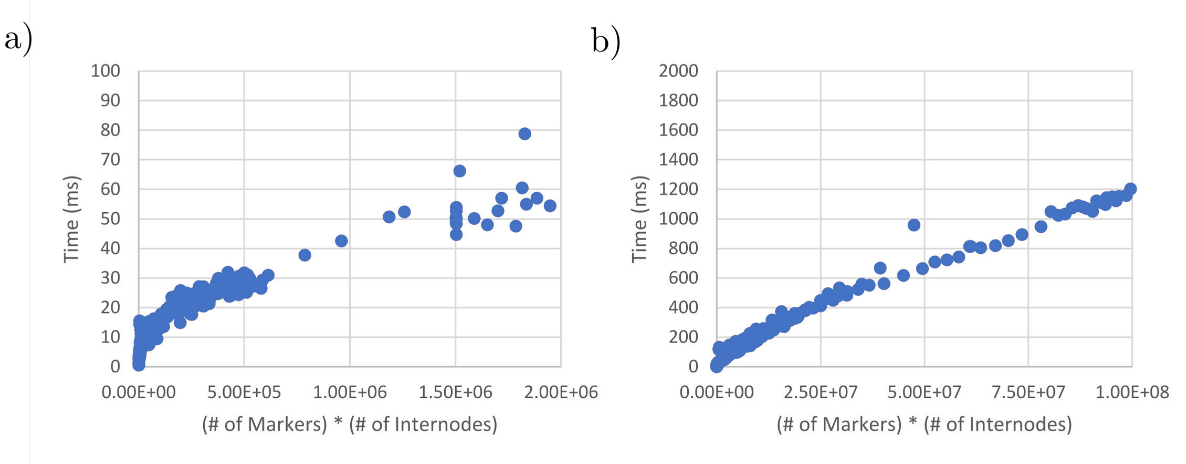


Figure 12.9: Graph relating the number of internodes multiplied by the number of markers to the growth step duration.

Plotting the number of internodes multiplied by the number of markers against the duration of the growth step shows a very strong correlation, and highlights the source of the slowdown. Since Tree Wand does not use a spatial subdivision data structure to find nearby buds in the AssignMarkersToBuds function in Algorithm 9.2, the complexity of this operation is $O(bm)$ where b is the number of buds, and m is the number of markers. When bm is greater than 500,000, Tree Wand becomes less interactive. For higher values of bm, the duration of a growth step can take over a second, as illustrated in Figure 12.9.

With this said, out of the examples I've shown, only modeling the Oak (Figure 12.5) and the Manitoba Maple tree (Figure 12.6) did I begin to encounter difficulties with modeling at an interactive rate.

Chapter 13

Conclusion

13.1 Contributions

As a part of this thesis, I developed ViNE, an application for the visualization, analysis and segmentation of plant structures within virtual reality. For this application I have presented a pipeline that enables large volumetric data to be rendered efficiently in virtual reality through polygonization using marching cubes. I have applied techniques to allow large meshes consisting of 30 million triangles to be painted while maintaining virtual reality frame rates. I have also included a number of features that benefit the analysis of plant structures such as the ability to save viewpoints and toggle the visibility of segmented components, as well as a temporally continuous model for depth cueing.

I have also developed Tree Wand, a virtual reality application for modeling trees inspired by TreeSketch [63]. I provided an interface for guiding the development of a tree by spreading fairy dust and manipulating parameters using floating windows. To facilitate the tree's growth, I modified the biologically motivated algorithm used in TreeSketch to support varying internode lengths. To visualize the tree, I have used the GPU to accelerate the generation of the tree's mesh, and developed a method to texture the tree without noticeable stretching.

While I did not have the opportunity to run a user study, both applications have received encouraging feedback. ViNE and Tree Wand have been used by colleagues within the University of Calgary, as well as visitors from other universities, numbering approximately 20-30 for each application. The users were able to learn both applications quickly, and with minimal instruction. Users who had experience with both Tree Wand and TreeSketch appreciated the impact that virtual reality had on modeling trees, finding it much easier and

more intuitive to create 3D tree structures within a virtual reality environment than when restricted to a 2D view.

In addition to the experience of colleagues and visitors, an early version of ViNE was presented at the P2IRC symposium in 2018 to approximately 30 researchers working with plants in various capacities. These researchers had a wide range of familiarity with virtual reality, some owning VR devices themselves, while others having had no exposure to VR or handheld controllers. Despite this, nearly every user was able to quickly learn the software and use it proficiently within minutes. The majority of these users required little to no direction, some figuring out how to use ViNE before I had the opportunity to explain it. The reception towards ViNE was very positive, with several researchers expressing interest in seeing it used as an educational tool, as it allows for the exploration of biological structures with an easily learned interface.

Among the few who struggled, most of the difficulty was found with the controllers: the grip buttons on the HTC Vive controller can be difficult to find while a user is unable to see their hands. Another source of difficulty for some users was not reaching far enough with the brush to paint the object in front of them. While only a couple of people had this problem, those that did would attempt this multiple times. This may be due to challenges with depth perception, but I suspect that it may have to do with an incorrect mental model of how ViNE works, with the user making the assumption that the brush will paint the parts of the model that it overlaps with from their viewpoint, akin to how a user would paint on a two-dimensional monitor.

These experiences have given insight into the usefulness of VR for exploring and designing 3D models. The improved perception of depth, and the ability to easily view models from different perspectives through the user's head movements greatly aided in the interpretation of complicated structures such as trees and vasculature. Trackable controllers opened up new options for the design of intuitive metaphors, and provided an ease of interaction in

three dimensions that a mouse cannot reproduce.

Through my work, I have demonstrated how the ability to be immersed within a virtual environment opens up new ways to view and interact with plant structures, as well as providing the new and unique experience of being able to stand inside of a flower head.

13.2 Future work

While ViNE and Tree Wand succeeded in their goals, there are a number of areas for future work that would be interesting to explore.

Both applications received a significant amount of feedback from the people they were demonstrated to, however no formal user study has been conducted. A user study would help to place the benefits of this work on a more solid footing.

Users in ViNE highlight veins by segmenting a 3D mesh into its separate components, however it can often be useful to have a node graph representing the vasculature as well, as was done by Usher et al. [102]. A graph representation of the vasculature can more easily represent the topology of the network. The datasets ViNE had been tested on also had veins broken at points, possibly as a result of the polygonization. A graph representation could bridge these gaps, making interpretation of the vasculature structure more straightforward.

While using a mesh generated by marching cubes helped meet the minimum frame rates needed for VR, this decision came with some drawbacks. Often an isovalue which allowed the vasculature to be clearly represented in one area would result in too much noise in a different part of the flower head. Volumetric rendering techniques which use isovalues or transfer functions could address this, as they can interactively change how the dataset is visualized. A transfer function can also use translucency to depict the exterior of the flower head without hiding its vasculature. Using volumetric rendering, Usher et al. [102] was limited to 256x256x256 windows into the dataset at a time, but future advancements in techniques and technology may make it possible for large datasets to be viewed in their

entirety at the necessary frame rates.

In Tree Wand the user grows an isolated tree in an empty environment. Having the ability to load terrain and urban structures within Tree Wand would give the user the opportunity to design trees with consideration to their intended surroundings. This would also open the possibility of modeling the interactions between a tree and its environment, such as restricting a tree's growth to be outside of buildings. A similar extension would be to allow multiple trees to be grown simultaneously, having nearby trees affect each other's development.

Modeling at interactive rates was an issue when there were a large number of buds and markers. Incorporating a spatial subdivision data structure could reduce the complexity of assigning buds to markers, and improve the performance considerably. Longay et al used a spatial grid in TreeSketch [63], however it would be worthwhile to explore other options as well, such as a K-D tree used in subsection 5.4.3 or the two-tiered grid used in subsection 9.5.3. Spatial acceleration structures would be a necessity for supporting the growth of multiple trees while keeping the modeling process interactive.

Using generalized cylinders to model internodes in Tree Wand was fast and efficient, but they could not accurately model branching points and would poorly resemble a real branch's appearance with larger ratios between the radius and curve length. In subsection 10.1.2 I discussed methods that could generate more realistic looking trees, but were more computationally expensive. The performance cost of using these techniques could be mitigated by adopting a hybrid approach: using the current method for thin branches and a more expensive one for thick branches. There are typically much fewer thick branches in a tree than thin ones, which would result in a much smaller impact to the program's performance, while improving the tree's appearance where it's most noticeable. In addition to the performance cost, while these methods were an improvement on generalized cylinders for representing branching points, they could not support arbitrary configurations. Addressing

this limitation may require other techniques to be explored.

The shadow propagation model I have used for this thesis is a coarse approximation, which takes no influence from the internode's shape; internodes of very different radii produce the same amount of shadow. In addition to this, the depth that the shadows propagate is finite. It would be interesting to see how tree forms may change with a more accurate lighting model which better incorporates the propagation of light through a tree.

Finally, there are features within TreeSketch [63] that ViNE could benefit from which have not yet been implemented within it. TreeSketch provided an undo function which could continuously reverse changes, instead of grouping them together. Several other features such as the ability to bend branches and a selective growth mode which restricts the tree's growth to the children of a single internode would help the user to assert more control over the tree's form.

Appendix A

The HTC Vive controller

Figure A.1: The HTC Vive's controller. The controller can be tracked in 3D using the HTC Vive Lighthouse base stations. Trigger: Button pressed by the pointer finger which can be pressed down partially or fully. Controller clicks on a full press. Grip button: Pressed with middle and ring finger using an action similar to the user closing their fist. Menu button: A button that can be assigned a function within a VR application. Trackpad: A trackpad that tracks the position of the user's thumb. It can be used as a button when pressed down. Steam menu button: A button that opens up the Steam Home menu. Cannot be used for VR applications.

Appendix B

The parameter windows

B.1 Tree form window

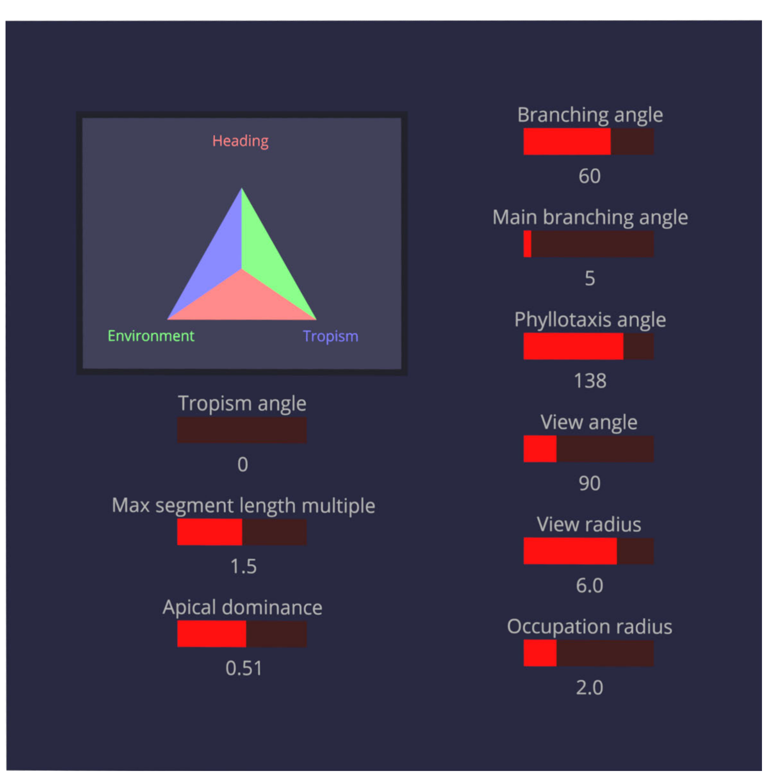


Figure B.1: The tree form window.

- Influence triangle: Determines the weight of the three influences in subsection 9.4.1. The size of the colored triangle determines the weight of the influence with the same color.
- Tropism angle: Angle from vertical for tropism influence.
- Max segment length multiple: Determines the maximum length of an internode, the value of α in section 9.7.

- Apical dominance: Sets the value of λ in subsection 9.5.1.
- Branching angle: The lateral branching angle from section 9.3.
- Main branching angle: The main branching angle from section 9.3.
- Phyllotaxis angle: The divergence angle from section 9.3.
- View angle: The angle of the perception volume in section 9.4.
- View radius: The radius of the perception volume in section 9.4.
- Occupation radius: The radius of the occupation zone in section 9.4.

B.2 Appearance window

Figure B.2: The appearance window.

- Bark texture select: Selects the bark texture used for the tree.
- Bark texture rate: Determines how stretched vertically the texture is on the tree.
- Tip radius: The radius of the end internodes of each branch.
- Radius exponent: The exponent used for section 9.6 to calculate the radius of the parent branch from its children.
- Leaf texture select: Selects the leaf texture and its associated geometry

- Leaf size: The scale of the leaves. If this is 0, no leaves are produced.
- Petiole length: The length of the petioles attached to the leaf.
- Petiole twist: The elasticity around the leaf's heading vector in subsection 10.2.2.
- Petiole bend up: The elasticity around the leaf's right vector in subsection 10.2.2.
- Petiole bend right: The elasticity around the leaf's up vector in subsection 10.2.2.

B.3 Light window

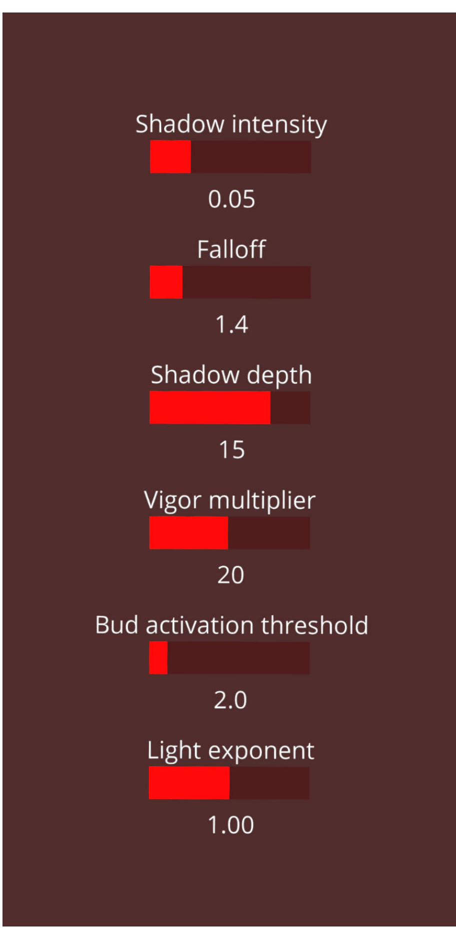


Figure B.3: The light window.

- Shadow intensity: The α value that scales the intensity of the shadow from subsection 9.5.2.
- Falloff: The β value that determines how quickly the shadow falls off from subsection 9.5.2.
- Shadow depth: The number of cells downward the shadow propagation pyramid extends, q_{max} from subsection 9.5.2.
- Vigor multiplier: The α value from subsection 9.5.1 that is used to convert light into vigor.

- Bud activation threshold: The amount of vigor a bud requires to become active. From subsection 9.5.1.

- Light exponent: The exponent that converts the illumination a bud receives into light. κ from subsection 9.5.1.

Bibliography

[1] Acciai, L., Soda, P., and Iannello, G. Automated Neuron Tracing Methods: An Updated Account. *Neuroinformatics 14*, 4 (2016), 353–367.

[2] Arora, R., Kazi, R. H., Anderson, F., Grossman, T., Singh, K., and Fitzmaurice, G. Experimental evaluation of sketching on surfaces in VR. *Conference on Human Factors in Computing Systems - Proceedings 2017-May* (2017), 5643–5654.

[3] Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software* (December 1996).

[4] Barrera Machuca, M. D., Asente, P., Stuerzlinger, W., Lu, J., and Kim, B. Multiplanes: Assisted freehand VR sketching. *SUI 2018 - Proceedings of the Symposium on Spatial User Interaction*, October (2018), 36–47.

[5] Bavoil, L., Sainz, M., and Dimitrov, R. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 Talks* (New York, NY, USA, 2008), SIGGRAPH '08, Association for Computing Machinery.

[6] Beat Games. Beat Saber, 2018.

[7] Beucher, S. Watersheds of functions and picture segmentation. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings 1982-May*, June 1982 (1982), 1928–1931.

[8] Bioscience, M. Neurolucida.

[9] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[10] Bloomenthal, J. Modeling the Mighty Maple. *Computer Graphics (ACM) 19*, 3 (1985), 305–311.

[11] Borchert, R., and Honda, H. Control of Development in the Bifurcating Branch System of Tabebuia rosea: A Computer Simulation. *Botanical Gazette 145*, 2 (1984), 184–195.

[12] Bosman, A. E. *Het wondere onderzoekingsveld der vlakke meetkunde.* Parcival, 1957.

[13] Boudon, F., Prusinkiewicz, P., Federl, P., Godin, C., and Karwowski, R. Interactive design of bonsai tree models. *Computer Graphics Forum 22*, 3 (2003), 591–599.

[14] Carroll, J. M., Mack, R. L., and Kellogg, W. A. Interface Metaphors and User Interface Design. In *Handbook of Human-Computer Interaction*, M. Helander, Ed. User Interface Institute, 1988, ch. 3, pp. 67–85.

[15] Cattiaux-Huillard, I., Albrecht, G., and Hernández-Mederos, V. Optimal parameterization of rational quadratic curves. *Computer Aided Geometric Design 26*, 7 (2009), 725–732.

[16] Chandrasekaran, S. Implementing Circular Buffer in C. `https://embedjournal.com/implementing-circular-buffer-embedded-c/`, 2014. [Online; accessed 27-May-2020].

[17] Chen, X., Neubert, B., Xu, Y. Q., Deussen, O., and Kang, S. B. Sketch-based tree modeling using Markov random field. *ACM Transactions on Graphics 27*, 5 (2008).

[18] Cho, I., and Wartell, Z. Evaluation of a bimanual simultaneous 7DOF interaction technique in virtual environments. *2015 IEEE Symposium on 3D User Interfaces, 3DUI 2015 - Proceedings* (2015).

[19] Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F., and Ranzuglia, G. MeshLab: an Open-Source Mesh Processing Tool. In *Eurographics Italian Chapter Conference* (2008), V. Scarano, R. D. Chiara, and U. Erra, Eds., The Eurographics Association.

[20] Clark, J. H. Designing Surfaces in 3-D. *Communications of the ACM 19*, 8 (1976), 454–460.

[21] Clarkson, K. L. Applications of random sampling in computational geometry, II. *Proceedings of the 4th Annual Symposium on Computational Geometry, SCG 1988* (1988), 1–11.

[22] Cook, R. L. Stochastic sampling in computer graphics. *ACM Transactions on Graphics (TOG) 5*, 1 (1986), 51–72.

[23] Coxeter, H. S. M. *Introduction to geometry.* John Wiley & Sons, New York, London, 1961.

[24] Dachselt, R., and Hübner, A. A survey and taxonomy of 3D menu techniques. *12th Eurographics Symposium on Virtual Environments, EGVE 2006* (2006), 89–99.

[25] Davison, T., Samavati, F., and Jacob, C. LifeBrush: Painting, simulating, and visualizing dense biomolecular environments. *Computers and Graphics (Pergamon) 82* (2019), 232–242.

[26] Dhondt, S., Vanhaeren, H., Van Loo, D., Cnudde, V., and Inzé, D. Plant structure visualization by high-resolution X-ray computed tomography. *Trends in Plant Science 15*, 8 (2010), 419–422.

[27] Diao, J., Lei, X., Wang, J., Lu, J., Guo, H., Fu, L., Shen, C., Ma, W., and Shen, J. Quantifying the variability of internode allometry within and between trees

for Pinus tabulaeformis Carr. Using a multilevel nonlinear mixed-effect model. *Forests 5*, 11 (2014), 2825–2845.

[28] Donnelly, P. M. D., Ario Bonetta, D., Tsukaya, H., and Engler, R. E. D. Cell Cycling and Cell Enlargement in Developing Leaves of Arabidopsis. *Developmental biology 215* (1999), 407–419.

[29] Dreamworks Animation. *OpenVDB*, 2012.

[30] Drebin, R. A., Carpenter, L., and Hanrahan, P. Volume rendering. *SIGGRAPH Comput. Graph. 22*, 4 (June 1988), 65–74.

[31] Falster, D. S., and Westoby, M. Leaf size and angle vary widely across species: What consequences for light interception? *New Phytologist 158*, 3 (2003), 509–525.

[32] Federl, P., and Prusinkiewicz, P. Virtual laboratory: An interactive software environment for computer graphics. *Proceedings - Computer Graphics International, CGI 1999* (1999).

[33] Feng, J., Cho, I., and Wartell, Z. Comparison of Device-Based, One and Two-Handed 7DOF Manipulation Techniques. *SUI '15: Proceedings of the 3rd ACM Symposium on Spatial User Interaction* (2015). Technique Review.

[34] Filipek, B. Persistent Mapped Buffers, Benchmark Results. `https://www.bfilipek.com/2015/01/persistent-mapped-buffers-benchmark.html`, 2015. [Online; accessed 27-May-2020].

[35] Filipek, B. Persistent Mapped Buffers in OpenGL. `https://www.bfilipek.com/2015/01/persistent-mapped-buffers-in-opengl.html`, 2015. [Online; accessed 27-May-2020].

[36] Freixenet, J., Muñoz, X., Raba, D., Martí, J., and Cufí, X. Yet another survey on image segmentation: Region and boundary information integration. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 2352* (2002), 408–422.

[37] Google. Tilt Brush by Google., 2015.

[38] Guenter, B., and Parent, R. Computing the Arc Length of Parametric Curves. *IEEE Computer Graphics and Applications 10*, 3 (1990), 72–78.

[39] Hand, C. A survey of 3D interaction techniques. *Computer Graphics Forum 16*, 5 (1997), 269–281.

[40] Hanson, A. J. Quaternion Gauss Maps and Optimal Framings of Curves and Surfaces. *Technical Report*, 518 (1998).

[41] Hart, J. C. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *Visual Computer 12*, 10 (1996), 527–545.

[42] Heckbert, P. S. Survey of Texture Mapping. *Proceedings - Graphics Interface*, May (1986), 207–212.

[43] Hisao Honda, P. B. T., and B., J. Computer Simulation of Branch Interaction and Regulation by Unequal Flow Rates in Botanical Trees. *American Journal of Botany 68*, 4 (1981), 569–585.

[44] Honda, H. Description of the form of trees by the parameters of the tree-like body. *Journal of Theoretical Biology 31* (1971), 331–338.

[45] Hrabcak, L., and Masserann, A. Asynchronous Buffer Transfers. In *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. Hoboken : CRC Press, 2012, 2012, ch. 28, pp. 391–414.

[46] Hughes, J. Why functional programming matters. *Computer Journal 32*, 2 (1989), 98–107.

[47] Interactive Data Visualization. Speedtree, 2020.

[48] Kahlen, K., Wiechers, D., and Stützel, H. Modelling leaf phototropism in a cucumber canopy. *Functional Plant Biology 35*, 10 (2008), 876–884.

[49] Kang, J., Tang, J., Donnelly, P., and Dengler, N. Primary vascular pattern and expression of ATHB-8 in shoots of Arabidopsis. *New Phytologist 158*, 3 (2003), 443–454.

[50] Keefe, D., Feliz, D., Moscovich, T., Laidlaw, D., and La Viola, J. CavePainting: A fully immersive 3D artistic medium and interactive experience. In *Proceedings of the Symposium on Interactive 3D Graphics* (Jan. 2001), pp. 85–93. ~~2001 Symposium on Interactive 3D Graphics ; Conference date: 19-03-2001 Through 21-03-2001.~~

[51] King, D. A. The functional significance of leaf angle in eucalyptus. *Australian Journal of Botany 45*, 4 (1997), 619–639.

[52] Kohek, Š., and Strnad, D. Interactive synthesis of self-organizing tree models on the GPU. *Computing 97*, 2 (2014), 145–169.

[53] Kornilov, A. S., and Safonov, I. V. An overview of watershed algorithm implementations in open source libraries. *Journal of Imaging 4*, 10 (2018).

[54] Koroleva, O. A., Davies, A., Deeken, R., Thorpe, M. R., Tomos, A. D., and Hedrich, R. Identification of a new glucosinolate-rich cell type in arabidopsis flower stalk. *Plant Physiology 124*, 2 (2000), 599–608.

[55] Lee, K., Avondo, J., Morrison, H., Blot, L., Stark, M., Sharpe, J., Bangham, A., and Coen, E. Visualizing plant development and gene expression in three dimensions using optical projection tomography. *Plant Cell 18*, 9 (2006), 2145–2156.

[56] Leggio, B., Laussu, J., Carlier, A., Godin, C., Lemaire, P., and Faure, E. MorphoNet: an interactive online morphological browser to explore complex multi-scale data. *Nature Communications 10*, 1 (2019).

[57] Levoy, M. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications 8*, 3 (1988), 29–37.

[58] Levoy, M. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics (TOG) 9*, 3 (1990), 245–261.

[59] Lindenmayer, A. Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology 18*, 3 (1968), 300–315.

[60] Liu, Y. The DIADEM and beyond. *Neuroinformatics 9*, 2-3 (2011), 99–102.

[61] Long, F., Zhou, J., and Peng, H. Visualization and analysis of 3D microscopic images. *PLoS Computational Biology 8*, 6 (2012), 1–7.

[62] Longay, S. Interactive Procedural Modelling of Trees and Landscapes. 160.

[63] Longay, S., Runions, A., Boudon, F., and Prusinkiewicz, P. TreeSketch : Interactive Procedural Modeling of Trees on a Tablet. *The proceedings of the Eurographics Symposium on Sketch-Based Interfaces and Modeling* (2012), 107–120.

[64] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 163–169.

[65] MacDonald, N. *Trees and networks in biological models.* Chichester (UK) Wiley, 1983.

[66] Macmurchy, P. The Use of Subdivision Surfaces in the Modeling of Plants. Master's thesis, The University of Calgary, ~~4~~ 2004.

[67] Mandelbrot, B. B., and Wheeler, J. A. The Fractal Geometry of Nature, 1983.

[68] Masliah, M. R., and Milgram, P. Measuring the allocation of control in a 6 degree-of-freedom docking experiment. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (2000). ~~One handed grab plus scale.~~

[69] McGraw, T., Garcia, E., and Sumner, D. Interactive swept surface modeling in virtual reality with motion-tracked controllers. *Proceedings - Sketch-Based Interfaces and Modeling, SBIM 2017 - Part of Expressive 2017*, ~~1~~ (2017).

[70] Meagher, D. Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. *Rensselaer Polytechnic Institute*, Technical Report IPL-TR-80-111 (1980).

[71] Mendes, D., Fonseca, F., Araujo, B., Ferreira, A., and Jorge, J. Mid-air interactions above stereoscopic interactive tables . *2014 IEEE Symposium on 3D User Interfaces (3DUI)* (2014).

[72] Mine, M. R., Brooks, F. P., and Sequin, C. H. Moving objects in space: Exploiting proprioception in virtual-environment interaction. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., p. 19–26.

[73] Morse, F. W., Rodkaew, Y., Chongstitvatana, P., growth modeling, . . . , S. S. . . . ., 2003, U., Siripant, S., and Lursinsap, C. Particle systems for plant

modeling. *Plant growth modeling and applications Proceedings of PMA03 2*, November (2003), 169–175.

[74] Murray, C. D. The physiological principle of minimum work applied to the angle of branching of arteries. *Journal of General Physiology 9*, 6 (1926), 835–841.

[75] Niinemets, Ü., and Fleck, S. Petiole mechanics, leaf inclination, morphology, and investment in support in relation to light availability in the canopy of Liriodendron tulipifera. *Oecologia 132*, 1 (2002), 21–33.

[76] Niklas, K. J. Gravity-Induced Effects on Material Properties and Size of Leaves on Horizontal Shoots of Acer saccharum (Aceraceae). *American Journal of Botany 79*, 7 (1992), 820.

[77] Okabe, M., Owada, S., and Igarash, T. Interactive design of botanical trees using freehand sketches and example-based editing. *Computer Graphics Forum 24*, 3 (2005), 487–496.

[78] Oppenheimer, P. E. Real time design and animation of fractal plants and trees. *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1986 20*, 4 (1986), 55–64.

[79] Palubicki, W., Horel, K., Longay, S., Runions, A., Lane, B., Měch, R., and Prusinkiewicz, P. Self-organizing tree models for image synthesis. *ACM Transactions on Graphics 28*, 3 (2009).

[80] Peachey, D. R. Solid Texturing of Complex Surfaces. *SIGGRAPH '85 19*, 3 (1985), 279–286.

[81] Peng, H., Tang, J., Xiao, H., Bria, A., Zhou, J., Butler, V., Zhou, Z., Gonzalez-Bellido, P. T., Oh, S. W., Chen, J., Mitra, A., Tsien, R. W., Zeng, H., Ascoli, G. A., Iannello, G., Hawrylycz, M., Myers, E., and

Long, F. Virtual finger boosts three-dimensional imaging and microsurgery as well as terabyte volume image visualization and analysis. *Nature Communications 5*, May (2014), 1–13.

[82] Perlin, K., and Hoffert, E. M. Hypertexture. *SIGGRAPH Comput. Graph. 23*, 3 (July 1989), 253–262.

[83] Pirk, S., Niese, T., Hädrich, T., Benes, B., and Deussen, O. Windy trees: Computing stress response for developmental tree models. *ACM Transactions on Graphics 33*, 6 (2014).

[84] Prusinkiewicz, P., and Lindenmayer, A. The Algorithmic Beauty of Plants (Przemyslaw Prusinkiewicz and Aristid Lindenmayer). *SIAM Review 34*, 1 (1990).

[85] Prusinkiewicz, P., Mündermann, L., Karwowski, R., and Lane, B. The use of positional information in the modeling of plants. *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2001* (2001), 289–300.

[86] Raabe, K., Pisek, J., Sonnentag, O., and Annuk, K. Variations of leaf inclination angle distribution with height over the growing season and light exposure for eight broadleaf tree species. *Agricultural and Forest Meteorology 214-215* (2015), 2–11.

[87] Rekdal, O. B. Academic urban legends. *Social Studies of Science 44*, 4 (2014), 638–654.

[88] Rosales, E., Rodriguez, J., and Sheffer, A. Surfacebrush: From virtual reality drawings to manifold surfaces. *ACM Transactions on Graphics 38*, 4 (2019).

[89] Runions, A., Lane, B., and Prusinkiewicz, P. Modeling trees with a space colonization algorithm. *Natural Phenomena* (2007), 63–70.

[90] Sachs, T. Self-organization of tree form: A model for complex social systems. *Journal of Theoretical Biology 230*, 2 (2004), 197–202.

[91] Sachs, T., and Novoplansky, A. Tree Form: Architectural Models do not suffice. *Journal of Plant Sciences 91*, 5 (1995), 1689–1699.

[92] Sachs, T., and Novoplansky, A. Tree Form: Architectural Models do not suffice. *Journal of Plant Sciences 91*, 5 (2012), 1689–1699.

[93] Sharpe, J. Optical Projection Tomography. In *Advanced Imaging in Biology and Medicine: Technology, Software Environments, Applications*, C. W. Sensen and B. Hallgrímsson, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 199–224.

[94] Shirley. Fundamentals of Computer Graphics. *Fundamentals of Computer Graphics* (2009).

[95] Sketch, G. Gravity sketch, 2017.

[96] Song, P., Goh, W. B., Hutama, W., Fu, C.-W., and Liu, X. A handle bar metaphor for virtual object manipulation with mid-air interaction. *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012).

[97] Sutter, H. Leak Freedom in C++... By Default. `https://github.com/CppCon/CppCon2016/blob/master/Presentations/Lifetime%20Safety%20By%20Default%20-%20Making%20Code%20Leak-Free%20by%20Construction/Lifetime%20Safety%20By%20Default%20-%20Making%20Code%20Leak-Free%20by%20Construction%20-%20Herb%20Sutter%20-%20CppCon%202016.pdf`, 2016. [Online; accessed 27-May-2020].

[98] Szudzik, M. An elegant pairing function. In *Wolfram Research (ed.) Special NKS 2006 Wolfram Science Conference* (2006), pp. 1–12.

[99] The C++ Resources Network. std::map. http://www.cplusplus.com/reference/map/map/, 2020. [Online; accessed 27-May-2020].

[100] The Khronos Group Inc. Tessellation. https://www.khronos.org/opengl/wiki/Tessellation, 2019. [Online; accessed 27-May-2020].

[101] Ulam, S. On some mathematical problems connected with patterns of growth of figures. In *Mathematical Problems in the Biological Sciences, Volume 14; Volume 20*, R. E. Bellman, Ed. Proceedings of Symposia in Applied Mathematics, 1962, ch. On some ma, pp. 215–224.

[102] Usher, W., Klacansky, P., Federer, F., Bremer, P. T., Knoll, A., Yarch, J., Angelucci, A., and Pascucci, V. A Virtual Reality Visualization Tool for Neuron Tracing. *IEEE Transactions on Visualization and Computer Graphics 24*, 1 (2018), 994–1003.

[103] Valve Corporation. Introducing SteamVR Motion Smoothing. https://steamcommunity.com/games/250820/announcements/detail/1705071932992003492, 2018. [Online; accessed 27-May-2020].

[104] Valve Corporation. *OpenVR*, 2020.

[105] Van Teylingen, R., Ribarsky, W., and Van Mast, C. D. Virtual data visualizer. *IEEE Transactions on Visualization and Computer Graphics 3*, 1 (1997), 65–74.

[106] Verheof, W. Light Scattering by Leaf layers with Application to Canopy Reflectance Modeling: the SAIL Model. *Remote Sensing Of Environment 141* (1984), 125–141.

[107] Vlachos, A. Advanced VR Rendering by Alex Vlachos. https://www.youtube.com/watch?v=J07G38_pxU4&list=PLF68GURdN-D0g0pjKY7CARtl19y3qK_3P, 2015. [Online; accessed 27-May-2020].

[108] VR, O. Oculus medium, 2017.

[109] Walls, J. R., Coultas, L., Rossant, J., and Henkelman, R. M. Three-dimensional analysis of vascular development in the mouse embryo. *PLoS ONE 3*, 8 (2008).

[110] Wilson, B. F. *The Growing Tree*, 1 ed. University of Massachusetts Press, 1970.

[111] Wysokinski, T. W., Chapman, D., Adams, G., Renier, M., Suortti, P., and Thomlinson, W. Beamlines of the biomedical imaging and therapy facility at the Canadian Light Source - Part 2. *Journal of Physics: Conference Series 425*, PART 7 (2013), 5–9.

[112] Wyvill, G., McPheeters, C., and Wyvill, B. Data structure for soft objects. *The Visual Computer 2*, 4 (1986), 227–234.

[113] Zhang, Q., Eagleson, R., and Peters, T. M. Volume visualization: a technical overview with a focus on medical applications. *Journal of digital imaging 24*, 4 (2011), 640–664.