

**M A S A R Y K
U N I V E R S I T Y**

FACULTY OF INFORMATICS

**Procedural generation of flower
pigmentation patterns**

Master's Thesis

VENDULA NĚMCOVÁ

Brno, Spring 2023

**M A S A R Y K
U N I V E R S I T Y**

FACULTY OF INFORMATICS

Procedural generation of flower pigmentation patterns

Master's Thesis

VENDULA NĚMCOVÁ

Advisor: RNDr. Jan Byška, Ph.D.

Department of Visual Computing

Brno, Spring 2023

MUNI
FI

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Vendula Němcová

Advisor: RNDr. Jan Byška, Ph.D.

Acknowledgements

First of all, I would like to thank my supervisor RNDr. Jan Byška, Ph.D. for the many supportive consultations, insightful suggestions, debugging ideas, and frequent proofreading. I would also like to thank Nuno Correia, Art.D. for fostering the thesis during my stay abroad and for his kind feedback on the initial design prototype. Additional thanks belong to Mgr. Antonín Zita, M.A., Ph.D. for the text corrections and valuable insights about writing.

Furthermore, I would like to thank Bc. Vlastimil Blažek for his grain of salt and valuable critical input on the text and application. I also thank my parents for the supportive lunches and sauna conversations and Thuy Trang Hanová for bringing sunny weather. My final thanks and yet another mistargeted promotion belong to the game Omnibullet for offering me creative distraction.

Abstract

The thesis follows a recently published paper from algorithmic botany titled Modeling flower pigmentation patterns (Ringham et al., 2021). The original implementation of the modeling method is reviewed and redesigned into a new application with improved UI, which aims to be more accessible to vegetation artists. Using the developed application, users can design procedural pipelines that generate color textures for existing 3D models of flowers. The pipelines are created in a node editor typical to 3D modeling software, where the nodes represent individual patterning operations.

Keywords

procedural texture generation, algorithmic botany, user interface design, C++, ImGui, WebGPU

Contents

Introduction	1
1 Creation of vegetation assets	3
1.1 3D modeling and photogrammetry	3
1.2 Software dedicated to vegetation	4
1.2.1 Modeling methods	4
1.2.2 Materials and texturing	5
1.2.3 UI and UX techniques	5
1.3 Conclusion	7
2 Background	8
2.1 Modeling method	8
2.2 Vasculature	9
2.2.1 Algorithm	10
2.2.2 Example	12
2.2.3 Vein width	13
2.2.4 Modification for coalescing veins	13
2.2.5 Implementation	13
2.2.6 3D extension	14
2.3 Positional information	14
2.3.1 Diffusion	15
2.4 Random influences	18
2.4.1 Random vertex selection	18
2.4.2 Stochastic diffusion	19
2.5 Reaction-diffusion	19
3 Analysis of an existing implementation	22
3.1 Description	22
3.2 Shortcomings	24
3.3 Conclusion	25
4 Design of a new implementation	26
4.1 UI and UX requirements	26
4.2 Suitable UI and UX approaches	27
4.3 Proposed design	28
4.3.1 Nodes	30

4.3.2	Simulation control	34
5	Implementation	35
5.1	Design implications	35
5.2	Building elements	35
5.2.1	Nodes	36
5.2.2	Parameters	41
5.2.3	Operations	41
5.2.4	Drawables	44
5.3	Element interaction	45
5.3.1	Parameter synchronization	45
5.3.2	Node property updates	45
5.3.3	Visualization updates	46
6	Results	47
7	Discussion and future development	52
	Conclusion	53
	Bibliography	54
A	Initial design prototype	60
B	Node pipelines of case study patterns	68
C	External attachments	71

List of Figures

1.1	Components in PlantFactory and SpeedTree	6
1.2	Parameters and curves in PlantFactory and SpeedTree . .	7
2.1	Examples of floral patterns modeled by Ringham et al. . .	9
2.2	Types of leaf growth	11
2.3	Algorithm for simulating leaf venation	12
2.4	Diffusion computation on a mesh	16
2.5	The development of an activator-inhibitor system	21
3.1	Sidebar with tools and settings in the rdpq program . . .	23
3.2	Mesh view in the rdpq program	23
3.3	The pigmentation modeling process as a pipeline	24
4.1	The graph editor in Blender	27
4.2	The Unity shader graph	28
4.3	The implemented user interface	29
4.4	The individual stages in a venation pipeline	31
4.5	The individual stages in a <i>Digitalis purpurea</i> pipeline . . .	33
5.1	Diagram of main classes	37
5.2	Diagram of node editor classes	38
6.1	Example effects of venation parameters: <i>kill distance</i> and <i>growth speed</i>	48
6.2	Reproduced patterns: <i>Primula</i> $\times$ <i>polyantha</i> and <i>Arctostaphylos uva-ursi</i>	49
6.3	Reproduced pattern: <i>Digitalis purpurea</i>	49
6.4	Reproduced pattern: <i>Platycodon grandiflorus</i>	50
6.5	Approximated pattern: <i>Dianthus barbatus</i>	50
6.6	Approximated pattern: <i>Phalaenopsis Chian Xen Leopard Stripes</i> .	51
6.7	Approximated pattern: <i>Mimulus guttatus</i>	51
A.1	UI prototype: <i>Kohleria</i> — Mesh node	60
A.2	UI prototype: <i>Kohleria</i> — Paint node	61
A.3	UI prototype: <i>Kohleria</i> — Diffusion node	61
A.4	UI prototype: <i>Kohleria</i> — Gradient node	62
A.5	UI prototype: <i>Kohleria</i> — Reaction-diffusion node at $t = 1$.	62
A.6	UI prototype: <i>Kohleria</i> — Reaction-diffusion node at $t = 10$.	63
A.7	UI prototype: <i>Kohleria</i> — Reaction-diffusion node at $t = 20$.	63

A.8	UI prototype: <i>Kohleria</i> — Reaction-diffusion node at $t = 50$	64
A.9	UI prototype: <i>Kohleria</i> — Result node	64
A.10	UI prototype: <i>Platycodon</i> — Venation node at $t = 1$	65
A.11	UI prototype: <i>Platycodon</i> — Venation node at $t = 10$	65
A.12	UI prototype: <i>Platycodon</i> — Venation node at $t = 50$	66
A.13	UI prototype: <i>Platycodon</i> — Venation node at $t = 80$	66
A.14	UI prototype: <i>Platycodon</i> — Venation node at $t = 95$	67
A.15	UI prototype: <i>Platycodon</i> — Result node	67
B.1	Node pipeline: <i>Primula</i> $\times$ <i>polyantha</i> and <i>Arctostaphylos uva-</i> <i>ursi</i>	68
B.2	Node pipeline: <i>Digitalis purpurea</i>	68
B.3	Node pipeline: <i>Platycodon grandiflorus</i>	69
B.4	Node pipeline: <i>Dianthus barbatus</i>	69
B.5	Node pipeline: <i>Phalaenopsis Chian Xen Leopard</i>	70
B.6	Node pipeline: <i>Mimulus guttatus</i>	70

Introduction

Algorithmic botany is an interdisciplinary field that combines natural sciences with informatics to study plant morphology and growth. The main objective is to model, simulate, and visualize realistic plant representations. When formulating these virtual models, biology provides a theoretical basis that is examined through computer science, whose perspectives, in turn, help understand and describe botanical phenomena that have not yet been explained.

The new findings, as well as computer-generated models of plants, greatly benefit fields like agriculture, forestry, and environmental modeling, because simulating plant behavior with software enables rapid and well-controlled experiments. Moreover, algorithmic botany also has direct applications in computer graphics, visualization, and art, as it enables the creation of believable virtual environments.

The study of plant morphology has a long history, but it has been pushed forward by mathematics and computational techniques only since the 20th century. One of the first to successfully apply this approach was Alan Turing, who mathematically described the principles of biological morphogenesis in 1952 [1]. Another milestone was the formal grammar *L-system* developed by Aristid Lindenmayer in 1968 [2] to describe the behavior of plant cells. The grammar was later extended in collaboration with Przemyslaw Prusinkiewicz to model the growth of higher plants [3]. Currently, a significant part of algorithmic-botany research is done at the University of Calgary in the Biological Modeling and Visualization research group (BMV) led by Prusinkiewicz. The group has been active since the 1990s and has published many papers, theses, and other materials focusing on various aspects of plant modeling [4].

Modeling flower pigmentation patterns is a recent paper [5] written by the BMV researchers Ringham et al., which concentrates on the colorful patterning of flowers. It reviews the flower pigmentation patterns found in nature and classifies them by the supposed mechanism of their creation. Based on existing research, the authors devise several methods for reproducing the patterns algorithmically. Furthermore, they implement them in an interactive graphical application.

This thesis aims to implement the modeling methods in a new application with an improved user interface, which would be more accessible to artists who model vegetation for computer games and visualization. The application should let the artists design and procedurally generate color textures for existing 3D models of flowers.

The first chapter reviews the approaches and software that are currently utilized for creating vegetation assets in computer games, cinematography, and visualization. Chapter 2 describes the pigmentation-modeling method [5] in mathematic and algorithmic details. Chapter 3 analyses the application implemented by Ringham et al. [5] and comments on how it can be adjusted for practical artistic use. Chapter 4 proposes a new, improved application design, and Chapter 5 describes its technical implementation. Finally, the last two chapters discuss what the new application can achieve and how it could be further improved. The practical part of the thesis constitutes the newly implemented application for creating flower textures.

1 Creation of vegetation assets

This chapter provides an overview of vegetation modeling approaches and software used by artists in practice. The main focus is given to the game industry and related commercial domains to select methods that have proven sufficiently effective, user-friendly, versatile, and capable of achieving realism. The main goals are:

- To identify the methods of structuring and representing information related to organic modeling.
- To observe the user interfaces and interaction methods used for organic modeling.
- To find which kinds of functionality are important for the users.
- To assess the capabilities and limitations of full-blown vegetation toolkits regarding the synthesis of stylized and realistic textures.

1.1 3D modeling and photogrammetry

This section overviews the two most common techniques for creating plant assets in general-purpose 3D software.

One way to create plant geometry is by manual 3D modeling, most commonly digital sculpting, which is a technique similar to working with clay. First, the artists create a base mesh by, e.g., composing spheres or modeling a low-polygon mesh. Then they modify the mesh with sculpting brushes that increase detail, carve, push, pull, smooth, or otherwise alter the surface.

Another method is photogrammetry, where the artists digitalize real-world plants. First, a plant is viewed from different angles and captured in multiple high-resolution photographs. Subsequently, significant surface points are identified across the pictures and reconstructed into a three-dimensional point cloud. Finally, the points are connected to form a surface.

In both cases, the resulting mesh usually has a high polygon count and must be optimized for real-time applications. This is done by simplifying the model into a low-poly version and “baking” the details

into textures, such as normal, displacement, and ambient occlusion maps.

Typically, the materials use color textures scanned from natural plants to achieve a realistic look. Alternatively, stylized vegetation may be hand-painted by the artist.

1.2 Software dedicated to vegetation

This section describes complex software¹ dedicated to modeling vegetation, covering the industry-leading *SpeedTree* [6] and *PlantFactory* [7].

SpeedTree and PlantFactory are procedural modeling toolkits for creating 3D models of diverse vegetation and other organic self-repeating structures. Both provide integrations into selected 3D creation suites and allow exporting into multiple widely used formats. In addition to the modeling toolkits, the companies offer collections of pre-modeled plants, usually grouped by type or biome. These plants are fully textured and model the species found in nature.

1.2.1 Modeling methods

The plant modeling methods seem driven or at least closely inspired by L-systems [7, 8], with each software using a different ratio of procedural generation and interactive control. The plant is decomposed into a hierarchy of its parts (organs), such as the roots, branches, stalks, or leaves. Each plant part has a template that generates multiple instances of the plant geometry. The templates have several defining properties, such as length, profile, attachment angle, or the distribution and density of occurrence within its parent organ. The property values can be randomized or manually adjusted per instance to disrupt regularity and repetition between the generated plant parts.

To provide more artistic control over the plant, both SpeedTree and PlantFactory allow adjusting the geometry. For example, the user can draw branch skeletons by hand and alter the shape and surface of individual plant parts. To achieve the latter, PlantFactory allows users

1. The information presented below was deduced from online software documentation, promotional materials, and instructive videos without any personal experience or closer knowledge of the software and thus might be imprecise or incomplete.

to create custom meshes by parametrically modifying basic geometric shapes. On the other hand, SpeedTree focuses more on manual fine-tuning via sculpting techniques.

1.2.2 Materials and texturing

The SpeedTree [8] plant geometry is combined with PBR materials. They can be created by assembling textures of multiple types, such as color/albedo, opacity, normal, subsurface, and height maps. The textures can be loaded from external files or created from a simple gradient or noise. SpeedTree then allows basic adjustment of the texture colors, such as modifying brightness, contrast, saturation, and color tone. The materials can reflect the season and have several tinted variants, e.g., to decrease the homogeneity of leaves. The SpeedTree documentation implies that users primarily work with textures obtained by scanning natural plants. It is, however, unclear whether such textures are made available in SpeedTree or need to be imported from other sources.

PlantFactory [7, 9] provides a rich, node-based system for creating procedural materials. It is similar to shader graphs in standard 3D creation suites (see Sec. 4.2 for examples) and features hundreds of nodes, including various noises, fractals, and math and color operations. The nodes can even incorporate information from the plant model, such as shape, maturity, or the current season. Upon plant export, all procedural material properties can be baked into bitmap textures or converted to vertex colors.

In both SpeedTree and PlantFactory, the materials can differ on their front and back sides. Furthermore, it is possible to create texture atlases, which store multiple texture variants in one picture for faster rendering.

1.2.3 UI and UX techniques

The main workflow in SpeedTree [8] and PlantFactory [7, 10] consists of manipulating a component hierarchy, adjusting parameters, and watching the 3D model change. SpeedTree represents the component hierarchy as a tree with symbolic icons, whereas PlantFactory displays it as a graph of feature nodes. The graph can consist of a few



Figure 1.1: Component hierarchy in PlantFactory (top) and SpeedTree (bottom). Adapted from [10, 8].

plant-organ nodes in simpler models, but it can also be extended with various utility nodes to achieve more advanced designs. Figure 1.1 shows both UI variants.

In both toolkits, many model properties are specified via curves. These curves control how the properties change along the plant organ or according to the placement within its parent. Curves can, for example, dictate the profile, distribution, density, resolution, or pruning intensity. Other model properties are specified via value ranges or as categorical options. The property values typically do not have clear units, and the ranges are not unified. However, the naming is usually intuitive, using words common to 3D creation suites or borrowing botanical vocabulary. Figure 1.2 shows examples of several properties.

PlantFactory supports saving custom components and property settings into a personal library, facilitating the reuse of previous work.

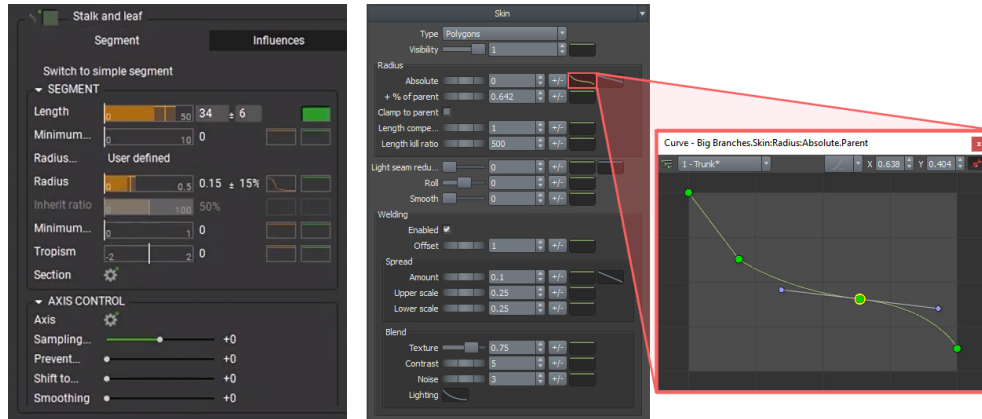


Figure 1.2: Parameters and curves in PlantFactory (left) and SpeedTree (right). Adapted from [10, 8].

1.3 Conclusion

The chapter briefly overviews the vegetation-modeling techniques and hints at the modeler's capabilities and preferences. The dedicated software employs principles and structures inspired by algorithmic botany. It systematizes plant modeling and heavily utilizes procedural techniques to simplify and automate the user workflow. Nevertheless, customization and manual control remain vital. Even though the software is complex and uses extensive parametrization, this has not negatively affected its adoption.

The materials typically reproduce or mimic realistic surfaces. Apart from scanning natural plants, there also seems to be some demand for procedurally created materials. The software supports importing external textures and incorporating them into complex materials for individual plant organs.

These observations suggest that a botanically-motivated method for the procedural generation of flower textures may have an artistic application. Depending on the future maturity of the method's technical realization, user interest, and software extensibility, the method might theoretically be integrated with the existing software. In any case, the generated textures could be imported as an external resource.

2 Background

This chapter paraphrases the paper on *Modeling flower pigmentation patterns* by Ringham et al. [5]. It briefly touches upon its biological and morphological background and explores the utilized algorithms and mathematical concepts, providing the theoretical basis for the following chapters.

The color of flowers is known to be primarily determined by biological pigments. However, how these pigments disperse and concentrate still needs to be fully understood. Studies suggest that some pigmentation patterns are affected by genetics or random influences, and others are formed through reaction-diffusion processes. Other missing pieces of knowledge may be, in part, supplemented [5] by theoretical morphogenesis.

Ringham et al. [5] build on the existing theories and identify several mechanisms that may affect pigmentation. Four of them are chosen for modeling: the vascular system (Fig. 2.1 C), the position within the flower (Fig. 2.1 A, B, D, and area in E), reaction-diffusion processes (Fig. 2.1 spots in E and F), and random phenomena (Fig. 2.1 G). Following some previous attempts to model pigmentation in animals [11, 12, 13], they also consider that these mechanisms can be combined. For example, the reaction-diffusion process may change depending on the local properties of plant tissue — in effect, positional information. This assumption facilitates both the analysis of floral patterns and their algorithmic simulation. The authors propose a method for pigmentation modeling built around these four mechanisms.

2.1 Modeling method

The flower is represented by a 3D triangular mesh that is taken as input. The patterns are formed by morphogens — pigment-inducing substances, which are quantized on the mesh vertices. The morphogen concentrations are initialized interactively and then modulated by the procedural application of the patterning mechanisms. This modeling stage can be repeated so that the outcome of one patterning mechanism can become the input for another. Finally, a user-defined color map transforms the morphogen concentrations into colorful pigments.



Figure 2.1: Examples of floral patterns modeled in [5]: (A) Gradual color change, (B) discrete color change, (C) pigmentation associated with veins, (D) regular spots, (E) irregular spots and area guiding pollinators, (F) irregular spots suggesting reaction-diffusion, (G) random pattern due to DNA transpositions. Adapted from [5].

For the simulation to perform well, the mesh should approximate the surface with dense, almost isotropic, and uniform triangles. Ringham et al. [5] achieve these qualities by modeling a rough mesh in a 3D mesh editor Blender [14], increasing the resolution using Catmull-Clark subdivision, and triangulating the result using the Instant Meshes software [15]. During pigmentation modeling, the mesh is represented using the half-edge data structure.

The following sections provide fundamental insight into the computational models of each mechanism affecting pigmentation.

2.2 Vasculature

Many flower species display patterning that appears to be associated with the vasculature (Fig. 2.1 C). Based on flower observation and phylogenetic studies [16, 17], Ringham et al. [5] suppose that veins may locally affect diffusion or be the site of morphogen production. Hence, a vein system is modeled to drive and influence the patterning

process. Runions et al. [18] previously proposed an iterative geometric algorithm for simulating leaf venation. Ringham et al. [5] employ this algorithm, extending it from the plane into 3D.

The following subsections are digests of the original paper by Runions et al. [18]

2.2.1 Algorithm

The algorithm is inspired by botanical theories, particularly *the canalization hypothesis* [19]. According to this hypothesis, the veins develop in response to the distribution of plant hormone auxin. The hormone produced in the leaf blade flows toward the leaf stalk, changing the traversed tissue. The cells with a higher auxin flow become more conductive and, in turn, attract more transport. Gradually, narrow canals reminiscing riverbeds are formed that eventually differentiate into vascular strands.

Runions et al. [18] revisit previous computational models [20, 21] and design a simple algorithm producing veins that look more natural. It is based on a hypothetical feedback loop underlying vein development: The leaf growth creates space for new auxin sources and veins. The auxin sources cause the formation of veins, which then “absorb” the closeby sources, channeling auxin away. A repeated simulation of these processes generates the whole vascular system. The algorithm emulates them with the following steps:

1. The leaf grows by a small amount.
2. New auxin sources are inserted in suitable places.
3. New veins grow toward the auxin sources.
4. Auxin sources too close to veins are destroyed.

The algorithm supports three kinds of leaf growth depicted in Figure 2.2: *Marginal*, in which the leaf grows on its border; *uniform*, in which the growth is uniform across the entire leaf; and a limited version of *anisotropic non-uniform* growth, in which the initial leaf shape is deformed over time. The marginal and uniform growth is modeled by scaling the leaf edge or blade using a scaling factor, which linearly increments with time. The anisotropic non-uniform growth

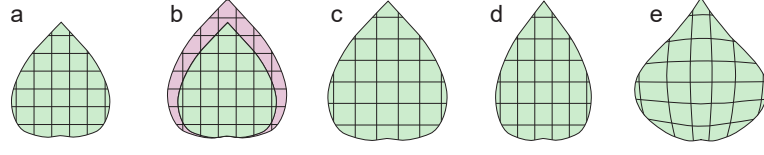


Figure 2.2: An initial leaf shape (a) and the results of its: (b) marginal growth, (c) uniform isotropic growth, (d) uniform anisotropic growth, and (e) non-uniform anisotropic growth. Taken from [18].

is modeled by user-defined functions that control the growth rate in the horizontal and vertical directions. The initial leaf shape and the growth method are specified as the algorithm's input.

The venation is represented by a directed acyclic graph $\mathcal{G} = (V, E)$. The set of vertices V symbolizes vein nodes, which approximate small segments of the veins. Adjacent vein nodes are connected by edges $e \in E \subset V \times V$. The edges are oriented from the older vein nodes towards their younger successors. An initial vein node (or multiple) is specified as the algorithm's input.

Auxin sources are represented by a set of points S scattered in the leaf tissue. Their placement depends on the local neighborhood: new auxin sources can form at least a *birth distance* b_s away from the already existing sources and at least a birth distance b_v away from existing vein nodes. Similarly, established auxin sources disappear once a new vein node appears within a *kill distance* d_k .

The auxin sources drive the process of vein creation. Each source influences the vein node closest to it. If a vein node v has such influencers, it will grow a child node v' towards the respective auxin sources. The child node's position will be given by:

$$p(v') = p(v) + D \frac{\vec{n}}{\|\vec{n}\|}, \text{ where } \vec{n} = \sum_{s \in S(v)} \frac{p(s) - p(v)}{\|p(s) - p(v)\|} \quad (2.1)$$

and where p denotes position, D is an inter-node distance parameter, and $s \in S(v)$ are the auxin sources influencing v .

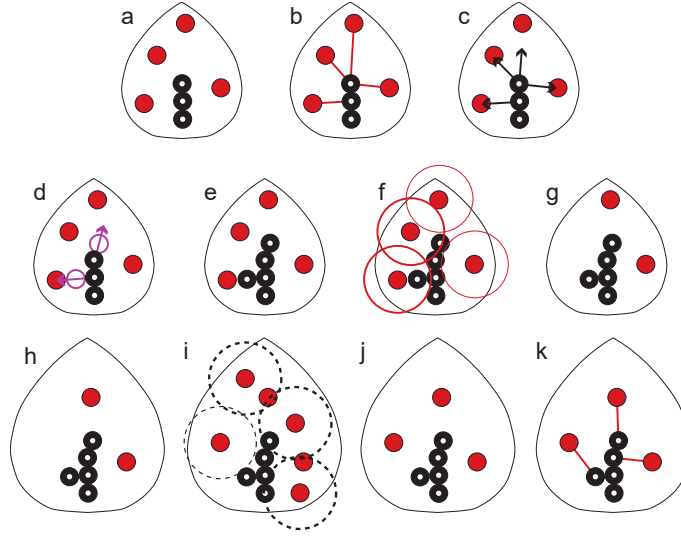


Figure 2.3: An illustration of the algorithm for simulating leaf venation. Taken from [18].

2.2.2 Example

Figure 2.3 illustrates a single loop of the simulation: In the first depicted step (a), there is a chain of three vein nodes, each represented by a black disc, and four auxin sources represented by red discs. Each vein node is assigned its influencing auxin sources (b). The directions towards these influencers (c) are combined, and the direction of new vein growth is obtained (d). Consecutively, new vein nodes develop (e). Because the venation has changed, the neighborhood of auxin sources is tested for the proximity of vein nodes (f). The auxin sources within the kill distance from veins are destroyed (g; only the vein centers are considered). The leaf grows by its margin (h). Candidates for new auxin sources are generated and tested for proximity to other sources and veins (i). The only suitable auxin source is added (j). The steps (j) and (k) correspond to (a) and (b), marking the beginning of a new iteration.

2.2.3 Vein width

To complete the vascular system, the individual vein segments need to be assigned width. The veins connected to the leaf stalk are naturally the widest, and as they branch, their successors get progressively thinner and thinner. Runions et al. [18] roughly model this effect using Murray's law [22]. It states that the radius of a vessel before a branching point is related to the radii of its successors by $r_{parent}^n = r_{child1}^n + r_{child2}^n$. The exponent n is taken as an input parameter for greater modeling flexibility. The widths are computed from the youngest (thinnest) veins to the oldest.

2.2.4 Modification for coalescing veins

The described algorithm generates open vascular systems where the veins have free ends. However, in nature, veins can also connect into loops, forming a closed system. Runions et al. [18] propose a second variant of the algorithm that simulates such systems. The key change is that more than one vein can grow toward and reach the same source. Therefore, an auxin source s is modified to influence multiple vein nodes instead of the closest one. A vein node v is influenced by s if it belongs to the *relative neighborhood* of s :

$$\forall u \in V \ \|p(v) - p(s)\| < \max\{\|p(u) - p(s)\|, \|p(v) - p(u)\|\}, \quad (2.2)$$

where p denotes position. The set of such veins v is denoted $V(s)$. Another modification concerns auxin source removal: The source must remain in place until it is reached by all the veins it influences. Once a vein node v enters the kill distance dk of source s , the vein nodes in $V(s)$ are tagged. When a tagged node develops a successor, the tag is passed down to it. The auxin source s is removed after all the tagged veins have reached it or left its kill distance dk .

2.2.5 Implementation

The previously described algorithm requires frequent spatial searching and distance calculations: Proximity information is used for inserting and deleting auxin sources and determining the location of new veins. In the later iterations of the algorithm, there can be hundreds

of auxin sources and thousands of vein nodes to process. Therefore, effective proximity testing is needed. Runions et al. [18] propose to employ Voronoi diagrams [23, 24].

In open vasculature systems, the unoptimized algorithm performs $\mathcal{O}(c(k+n))$ distance calculations for c candidate auxin sources, k existing sources, and n existing vein nodes. Additional $\mathcal{O}(kn)$ calculations are done when determining the influence of auxin sources on veins. In the improved version, a Voronoi diagram is constructed for the set S of auxin sources and another for the vein nodes V . The diagrams can be recreated in each iteration in $\mathcal{O}(n \log n + k \log k)$ time or acquired even faster by incrementally growing the initial diagrams [18, 24]. When a candidate auxin source is considered, its nearest neighbors are found in the Voronoi diagrams and tested for proximity. These tests take $\mathcal{O}(c(\log n + \log k))$ time for c candidate auxin sources because the nearest neighbor search takes logarithmic time [23]. Similarly, the nearest veins to be influenced by k auxin sources are found in $\mathcal{O}(k \log n)$ time. Their distances are stored and reused during auxin source removal.

Fast simulation of closed vasculature systems also depends on the effective computation of relative neighborhoods. This optimization is omitted because it is more complicated and not applied in the thesis. The interested reader may refer to the original paper by Runions et al. [18].

2.2.6 3D extension

Ringham et al. [5] state they have extended the vasculature algorithm into 3D by taking inspiration from Hädrich et al. [25]. In this extension, the auxin sources and the initial veins are positioned on the flower mesh. The positions of new veins are determined in 3D space and then projected onto the mesh surface. However, Ringham et al. [5] provide no other details, especially regarding the implementation.

2.3 Positional information

Some pigmentation patterns are closely related to their position within the flower. For example, the color in the flower center may differ

from the margins (Fig. 2.1 A and B), or certain flower areas might be highlighted to facilitate pollination (Fig. 2.1 spotted area in E). The pigments may form discrete regions (Fig. 2.1 B and D) or change in smooth transitions (Fig. 2.1 A).

Ringham et al. [5] capture positional effects using manual input and diffusion. The modeler may draw on the flower mesh to interactively specify the initial concentrations of pigment-inducing morphogens. These can then be adjusted by diffusion, which spreads the morphogens around, blurring the drawing. The obtained positional information – spatially varying morphogen concentrations – can be directly transformed into pigment or combined with other patterning mechanisms to simulate more complex effects.

2.3.1 Diffusion

Ringham et al. [5] propose a new implementation of diffusion that operates on a triangulated mesh and supports anisotropic spread and random influence. The method is derived from Fick's laws of diffusion and discrete differential geometry. The following description is a digest of the original paper by Ringham et al. [5].

The diffusing substance flows from regions of high concentration to those of low concentration. This concept is formalized by the first Fick's law as follows:

$$J = -D\nabla c, \quad (2.3)$$

where J is a vector that denotes the diffusion flux, ∇c is the gradient of concentration c , and D is the diffusivity term determining how fast the substance spreads. Because of the flux, the average substance concentration in region R changes over time. The change is described by:

$$\frac{dc(R)}{dt} = \frac{\dot{m}(R)}{\text{area}(R)}. \quad (2.4)$$

Here, $\dot{m}(R)$ is the rate of mass flow into the region R , defined by:

$$\dot{m}(R) = - \oint_{\delta R} J \cdot \hat{n} dl, \quad (2.5)$$

where δR is the region boundary, dl is an element of the boundary, and $\hat{n}$ is a normal vector to dl .

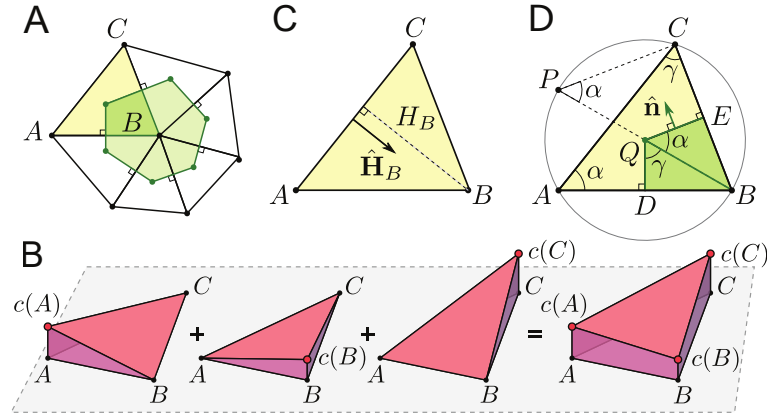


Figure 2.4: (A) The triangles sharing vertex B (white and yellow) and the Voronoi polygon $V(B)$ (green). (B) The concentration gradient in $\triangle ABC$ is obtained from gradients contributed by individual vertices. (C) The altitude segment H_B and unit vector $\hat{H}_B$. (D) The fragment of Voronoi polygon $V(B)$ that is contributed by $\triangle ABC$. Taken from [5].

Ringham et al. [5] adapt these differential equations for computation on the mesh, which they assume to be a Delaunay triangulation [24] of its vertices. The concentration of the diffusing morphogen is sampled at the vertices and denoted as $c(B)$ for a given vertex B . The concentration gradient ∇c and the flux J are calculated for each mesh triangle. To obtain the change in concentration, the authors utilize Voronoi polygons [24]. A Voronoi polygon $V(B)$ approximates the neighborhood of vertex B using fragments of the adjacent triangles (Fig. 2.4 A). These triangles conjointly affect the mass flow $\dot{m}(V(B))$ into the Voronoi polygon $V(B)$, which then determines the change in concentration $dc(B)$.

The concentration gradient in a triangle is approximated using the concentration at individual vertices. Figure 2.4 B illustrates the concentration values as vertex elevation. From this elevation, partial gradients become apparent: If $c(A)$ and $c(C)$ were 0, the concentration $c(B)$ would spread from B “down” along the altitude segment H_B . Considering all vertices together, the concentration gradient in the

whole triangle ABC is given by:

$$\nabla c(\triangle ABC) = \frac{c(A)}{\|H_A\|} \hat{H}_A + \frac{c(B)}{\|H_B\|} \hat{H}_B + \frac{c(C)}{\|H_C\|} \hat{H}_C, \quad (2.6)$$

where $\|H_B\|$ is the length of altitude H_B and $\hat{H}_B$ is a unit vector towards B parallel with H_B [5]. This expression can be simplified [5] using the vector N perpendicular to ABC such that $\|N\| = 2 \cdot \text{area}(\triangle ABC)$:

$$\nabla c(\triangle ABC) = -(c(A)\vec{BC} + c(B)\vec{CA} + c(C)\vec{AB}) \times \frac{N}{N \cdot N}. \quad (2.7)$$

If the diffusion spread is isotropic, the diffusivity D is a scalar constant and the flux J can be obtained directly from equation 2.3:

$$J(\triangle ABC) = -D \cdot \nabla c(\triangle ABC). \quad (2.8)$$

If it is anisotropic, Ringham et al. [5] treat D as a tensor defined by a pair of orthogonal eigenvectors λ_1 and λ_2 , which specify the diffusivity in two directions $\hat{\lambda}_1$ and $\hat{\lambda}_2$. The total flux is then obtained as a sum of the fluxes along both directions:

$$J(\triangle ABC) = -(\nabla c(\triangle ABC) \cdot \hat{\lambda}_1) \lambda_1 - (\nabla c(\triangle ABC) \cdot \hat{\lambda}_2) \lambda_2. \quad (2.9)$$

The mass is assumed to flow into B through the edges of Voronoi polygon $V(B)$. For triangle ABC , we thus consider the segments EQ and DQ depicted in Figure 2.4 D. The rate of mass flow through EQ is:

$$\dot{m}_{EQ} = J(\triangle ABC) \cdot \hat{n} \|\vec{EQ}\| = -\frac{1}{2} J(\triangle ABC) \cdot \vec{BC} \cot \alpha. \quad (2.10)$$

Using the same approach for $\dot{m}_{DQ}$, the rate of mass flow from the triangle ABC into $V(B)$ is:

$$\begin{aligned} \dot{m}(\triangle ABC \rightarrow V(B)) &= \\ \dot{m}_{EQ} + \dot{m}_{DQ} &= \frac{1}{2} J(\triangle ABC) \cdot (-\vec{BC} \cot \alpha + \vec{AB} \cot \gamma). \end{aligned} \quad (2.11)$$

The total rate of mass flow $\dot{m}(V(B))$ into $V(B)$ is obtained by summing the contributions of all triangles sharing vertex B .

The rate of concentration change at vertex B is proportional to the area of Voronoi polygon $V(B)$ (eq. 2.4). This area is again computed considering all polygon fragments in triangles adjacent to B . The contribution of a triangle ABC is given by:

$$\text{area}(\triangle ABC \rightarrow V(B)) = \frac{1}{8}(\vec{BC} \cdot \vec{BC} \cot \alpha + \vec{AB} \cdot \vec{AB} \cot \gamma). \quad (2.12)$$

The detailed derivation of equations 2.10–2.12 can be found in [5].

2.4 Random influences

Flower pigmentation can be affected by random influences of several kinds. DNA transpositions [26] can create irregular stains, as seen in Figure 2.1 G. The morphogen concentrations can be randomly inhomogeneous, causing the appearance of patterns during reaction-diffusion processes or disrupting pattern symmetry. Diffusion itself is a stochastic process due to the random movement of molecules.

Ringham et al. [5] account for randomness in the following ways: The morphogen concentrations may be initialized to distinctive values at randomly selected mesh vertices. The concentration values, in general, may also be chosen randomly. Finally, Ringham et al. [5] propose a stochastic extension for the diffusion model discussed in section 2.3.

2.4.1 Random vertex selection

Random vertex selection is achieved by enumerating all mesh vertices and selecting from them by randomly generated indices. Thanks to the required mesh properties described in section 2.1, uniformly distributed indices select vertices of approximately uniform distribution. [5] The rejection sampling method can be used to limit the probability of choosing vertex $v \in V$ by a custom probability density function $f : V \rightarrow \langle 0, 1 \rangle$. For example, a pattern may involve irregular spots that are densely scattered around the flower center but disappear towards the margins. In such a case, a random index preselects a candidate vertex v , and the value $f(v)$ is tested against another random number p generated from a uniform distribution. If $f(v) > p$, the vertex v is included in the final selection. [5]

2.4.2 Stochastic diffusion

Ringham et al. [5] adjust their computational model of diffusion to consider the random movement of molecules. First, they decompose the rate of mass flow through edge EQ ($\dot{m}_{EQ}$ from equation 2.10) by the individual contributing vertices of triangle ABC . For each vertex (e.g., B), they decompose the equation further and separate the morphogen concentration $c(B)$, the diffusivity, and the terms determined by the triangle geometry. The latter two are jointly denoted as Z_B , and the equation is rewritten as: $\dot{m}_{EQ}(B) = c(B)Z_B$. Here, Ringham et al. [5] convert masses into molecule numbers using model parameter Ω and utilize the *Chemical Langevin Equation*. [27] They end up with a random variable $\dot{M}_{EQ}(B)$ describing the average mass flow rate through edge EQ contributed by vertex B over time interval Δt :

$$\dot{M}_{EQ}(B) \Delta t = c(B) Z_B \Delta t + \sqrt{\frac{1}{\Omega} c(B) |Z_B|} \Delta t \mathcal{N}(0, 1), \quad (2.13)$$

where $\mathcal{N}(0, 1)$ represents a number from a normal distribution with $\mu = 0, \sigma = 1$.

2.5 Reaction-diffusion

Reaction-diffusion systems [1] are mathematical models of morphogenesis which describe the evolution and interaction of two and more morphogens. They can be represented by systems of partial differential equations:

$$\frac{du_i}{dt} = R(u_1, \dots, u_n) + D_i \nabla^2 u_i, \quad (2.14)$$

for $i = 1, \dots, n$ morphogen types. The term $D_i \nabla^2 u_i$ symbolizes the diffusion of morphogen u_i into nearby cells. $R(u_1, \dots, u_n)$ describes how different morphogen types in one cell affect each other. For example, the abundance of morphogen u_i may result in a local decrease in morphogen u_j , or some portion of morphogen u_k may transform into u_l . Reaction-diffusion processes lead to the emergence or development of waves and self-organizing patterns.

These mathematical models were proposed by Turing [1]. He described systems where the initial morphogen concentrations are nearly

homogeneous, but applying reaction-diffusion equations causes a periodic pattern to emerge. Other reaction-diffusion systems were proposed later, including the *activator-inhibitor* and *activator-substrate* systems by Gierer and Meinhardt [28] and the model by Gray and Scott [29]. In the activator-inhibitor model, patterns emerge from homogeneity, like in Turing's. In contrast, the activator-substrate and Gray-Scott systems develop patterns from the inhomogeneities that are initially present.

In their flower case studies, Ringham et al. [5] utilize Turing's system and activator-inhibitor and activator-substrate systems. The chosen equations for Turing's system are:

$$\begin{aligned}\frac{du}{dt} &= \rho(\alpha - uv) + D_u \nabla^2 u, \\ \frac{dv}{dt} &= \rho(uv - v - \beta) + \nabla \cdot D_v \nabla v,\end{aligned}\tag{2.15}$$

where $\nabla \cdot D_v \nabla v$ is an extension describing the anisotropic diffusion of u_v [30, 5].

The activator-inhibitor equations are adapted from the work of Ding et al. [31], who have analyzed the dotted pattern in *Mimulus guttatus* on a molecular level and reproduced it computationally. The version used by Ringham et al. [5] is given by:

$$\begin{aligned}\frac{da}{dt} &= \rho_a \frac{a^2}{h + \kappa} + \rho_{a0} - \mu_a a + D_a \nabla^2 a, \\ \frac{dh}{dt} &= \rho_h a^2 + \rho_{h0} - \mu_h h + D_h \nabla^2 h,\end{aligned}\tag{2.16}$$

where a represents the concentration of the activator and h the concentration of the inhibitor (proteins NEGAN and RTO, respectively).

The activator and inhibitor [28] are two kinds of morphogen, each diffusing at a different rate. The activator is self-enhancing, i.e., it tends to increase at places with locally larger concentrations. The inhibitor is produced in response to activator growth and quickly spreads into the neighborhood of an activator peak. Because of the inhibitor, the activator concentration on the peak boundary drops again, forming new local maxima nearby.

Although the patterns in Turing's and the activator-inhibitor models can spontaneously emerge from even morphogen concentrations, it

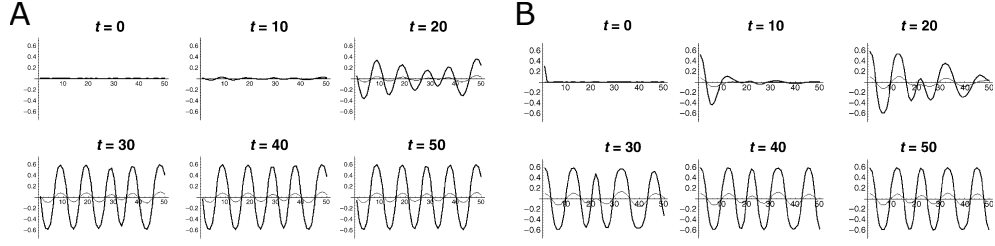


Figure 2.5: The development of a sample activator-inhibitor system in time. The concentrations at initial time $t = 0$ are even in (A) and uneven in (B). Taken from [32].

is also possible to initialize the concentrations unevenly. The intermediate reaction-diffusion pattern will be irregular but will stabilize over time in the same periodic pattern that would emerge from homogeneity [1, 32]. Figure 2.5 illustrates this on a sample activator-inhibitor system. Ringham et al. [5] use this temporary effect to generate dotted patterns that are irregularly spaced.

For the activator-substrate system [28], Ringham et al. [5] use the following equations:

$$\begin{aligned}\frac{da}{dt} &= \rho_s a^2 + \rho_{a0} - \mu_a a + D_a \nabla^2 a, \\ \frac{ds}{dt} &= -\rho_s a^2 + \rho_{s0} - \mu_s s + D_s \nabla^2 s,\end{aligned}\tag{2.17}$$

where a corresponds to the activator and s to the substrate concentrations.

3 Analysis of an existing implementation

To support the research on *Modeling flower pigmentation patterns* described in Chapter 2, Ringham et al. [5] implemented their method as a program called *rdpg* (reaction-diffusion pattern generator). This chapter aims to review and evaluate the program and outline its possible improvements.

Although the paper [5] describes the implementation and is supplemented with a video [33] demonstrating the *rdpg* functionality, I have not found any public build or source code repository. Therefore, the following analysis of the program is not based on direct experience but on available descriptions.

3.1 Description

The program is written in C++ using OpenGL 4.6 graphics and the Dear ImGui 1.8 user interface library. The implementation considers the simulation speed: the computation is parallelized and can run on the CPU or GPU. Another important aspect is the ease of work with different reaction-diffusion models. To facilitate experimentation, Ringham et al. [5] implement a scripting system that allows specifying the reaction-diffusion equations during runtime.

The user interface comprises a top menu bar, a mesh view, and a left sidebar with tools and settings. The top menu bar has two items: *File* and *Options*. The mesh view covers most of the screen and displays the input mesh as a solid surface or a wireframe (Figure 3.2 A). The mesh can be rotated by the mouse. The left sidebar (Figure 3.1 A) contains several sections with settings and a button for starting and pausing the reaction-diffusion simulation (Figure 3.1 B). The *Parameters* section exposes the parameters from reaction-diffusion equations; Figure 3.1 C suggests the activator-substrate system is in use. The *Painting* section configures interactive input. The user can either paint morphogen concentrations (Figures 3.1 D and 3.2 B) or specify the anisotropic diffusivity (Figures 3.1 E and 3.2 C). The last *Rendering* section controls the mesh view (Figure 3.1 F). The user can choose the displayed morphogens and their color mapping, adjust the mesh and window rendering, and set the number of simulation steps per frame.

3. ANALYSIS OF AN EXISTING IMPLEMENTATION

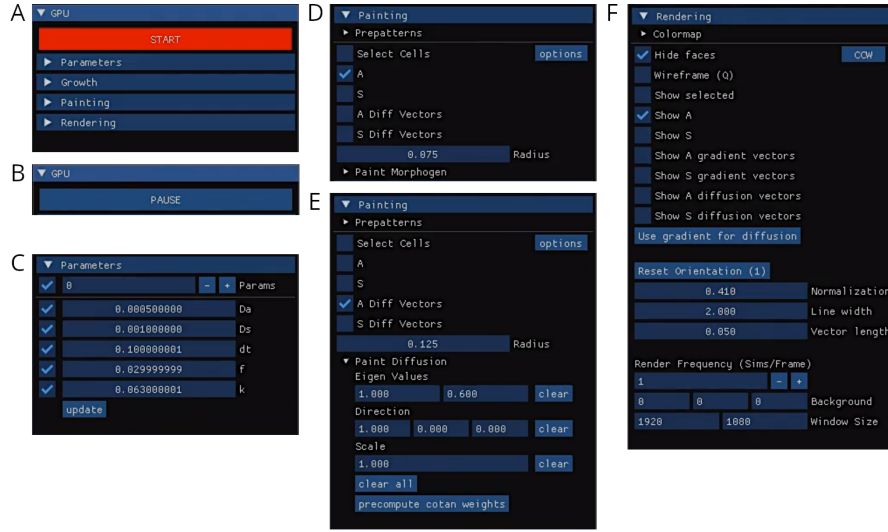


Figure 3.1: Sidebar with tools and settings in rdpG [33]. (A) Sidebar with a button for simulation control and collapsed sections. (B) Control button during simulation. (C) Reaction-diffusion parameters. (D) Parameters for painting morphogen concentrations. (E) Parameters for painting anisotropic diffusivity. (F) Parameters for adjusting the mesh view. Author's archive.

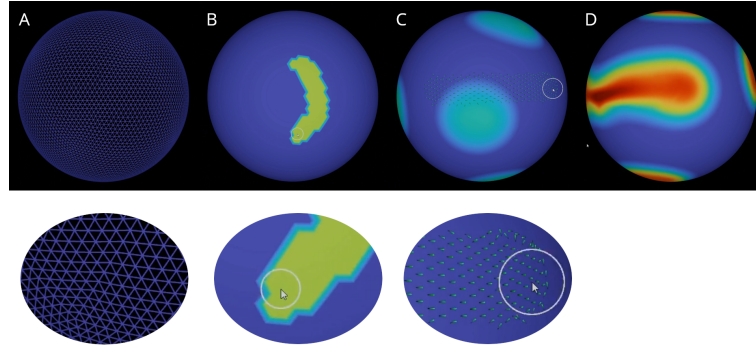


Figure 3.2: Mesh view in rdpG [33]. (A) Mesh wireframe, (B) painting morphogen concentrations, (C) painting anisotropic diffusivity, (D) adjusted normalization when mapping concentration to colors. Author's archive.

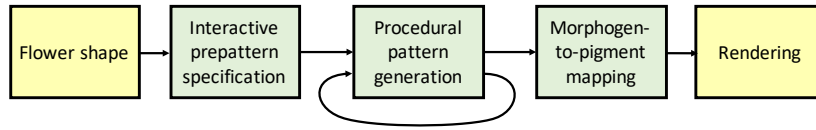


Figure 3.3: The pigmentation modeling process as a pipeline. Ringham et al. implement the stages in green. Taken from [5].

Unfortunately, the video [33] does not show all settings and features. Namely, the sidebar (Figure 3.1 A) suggests the simulation is computed on the GPU, but there is no visible switch to the CPU. Similarly, the *Parameters* section shows no option to choose the reaction-diffusion type or specify the equations. In addition, some sections only appear collapsed, so they cannot be studied — this includes the *Growth* section, the *Prepatterns* and *Paint Morphogen* subsections in *Painting*, the mesh cell selection in *Painting*, and the *Colormap* subsection in *Rendering*.

3.2 Shortcomings

Although the *rdpg* program appears functional and fast, it has several shortcomings. The main one concerns the user workflow. While Ringham et al. [5] describe the pigmentation modeling process as a pipeline (Figure 3.3), the program does not support this thinking. Instead, the modeling stages somewhat blend: the interactive input (painting on the mesh) can be interleaved with the reaction-diffusion simulation, and the morphogen-to-pigment mapping is done continuously. Moreover, the settings always reflect only the current state; the previous configurations and the simulation progress are not recorded. While such application design enables real-time experimentation, it is inconvenient when users want to modify, analyze, or reproduce their once-created pigmentation patterns.

Another downside of the program is its focus. Ringham et al. [5] describe in their paper how vasculature, diffusion, reaction-diffusion, and randomness can work together to create diverse pigmentation patterns. However, *rdpg* seems to prioritize reaction-diffusion over the other mechanisms. The diffusion settings influence the reaction-

diffusion process, but it is unclear how diffusion can be used alone and whether it can be stochastic. A similar problem is with the vasculature modeling, which does not seem to be handled by the program.

Finally, the user interface has some issues. The most important problem is that settings for unrelated logic are displayed together. For example, the *Rendering* section contains a “Use gradient for diffusion” button that implies changing the simulation instead of just visualization. Similarly, the *Painting* section has an option to “Select Cells,” which may logically differ from painting concentrations and diffusivity. Another problem is confusing representation: different painting and visualization options are accompanied by checkboxes, while checking and using all seems impractical, if not impossible. Lastly, the simulation control is rather elementary — it is only possible to hit start and pause. The user has no precise, configurable control and does not know the simulation progress.

3.3 Conclusion

The modeling method has great potential to be used by artists for creating procedural textures. However, the *rdpg* program seems to be designed primarily for research purposes. Its design supports interactive experimentation but does not utilize the structured nature of pigmentation modeling. While the theoretical method is procedural and exactly parametrizable, these qualities are lost in practice. Separating the modeling stages, leveraging the pipeline composition, and representing information better could enable the method to be used creatively. In the following chapter, I propose a new application design that improves the identified problems.

4 Design of a new implementation

This chapter presents a new application design aiming to make pigmentation modeling more accessible to an artistic audience. The design focuses on ease of use and the pipeline nature of the modeling method.

4.1 UI and UX requirements

The pigmentation modeling method described in Chapter 2 and its existing implementation analyzed in Chapter 3 are rather complex and technical. To generate a pigmentation pattern, the users must combine different operations and adjust their parameters. The parameters are numerous, and slight changes in one may result in a dramatic difference in the resulting pattern. For these reasons, users will need to experiment before they understand the system and will make frequent mistakes and amendments before they adjust the pattern to their liking.

To secure good usability and address the issues mentioned in Section 3.2, I have formulated the following requirements for the application and user interface:

- It is possible to see the whole procedural pipeline and its steps — the contributing operations.
- New operations can be added or removed at all times.
- It is clear how each operation affects the result.
- Every operation can be adjusted at all times.
- Adjusting one operation does not require reconfiguring another.
- Parameter adjustment has an immediate effect.
- The progress of time-dependent operations is visible.
- It is possible to control the simulation of time-dependent operations with fine precision.
- The procedural pipeline can be saved.
- The previously created pipeline can be loaded and rerun to produce the same or comparable pattern.

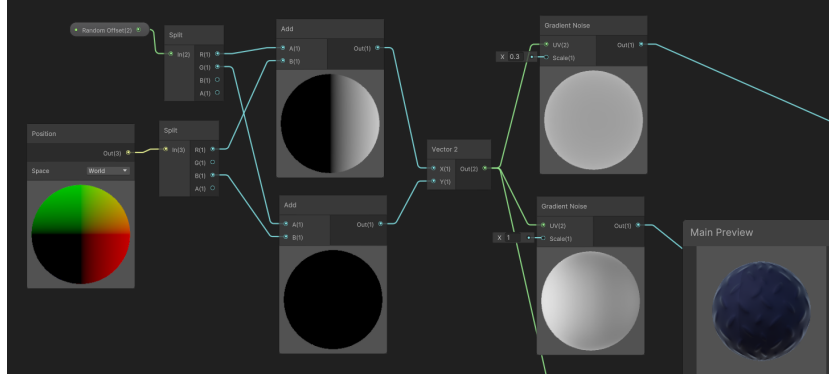


Figure 4.2: The Unity shader graph [35] with miniature previews of intermediate material state. The final material is displayed in the bottom right corner. Author’s archive.

vasculature and simulating reaction-diffusion processes take some time, and it is desirable to associate interactive painting more closely with the rest of the pipeline. Other changes were needed to provide adequate visual feedback on what each operation does. Unfortunately, Blender shows only the resulting material and offers no preview of the intermediate state. In comparison, Unity improves the user experience by supplementing nodes with miniature previews, as shown in Figure 4.2. Nevertheless, the slowly-formed and detailed pigmentation patterns ask for even closer inspection.

4.3 Proposed design

The proposed design introduces several kinds of nodes. The most important represent venation, diffusion, and reaction-diffusion. These are called *simulation nodes* and depend on time. Other nodes capture manual painting input, select mesh vertices, and transform values. The remaining nodes delimit the pipeline: it starts with a single mesh node and ends with the output vertex-color or color-texture node.

The nodes can be assembled in a graph editor at the bottom part of the application window (Fig. 4.3 B). It contains a menu bar for node creation (Fig. 4.3 C) and a canvas where the nodes are placed. The nodes and pins are colored by type to hint about their intended usage.

4. DESIGN OF A NEW IMPLEMENTATION

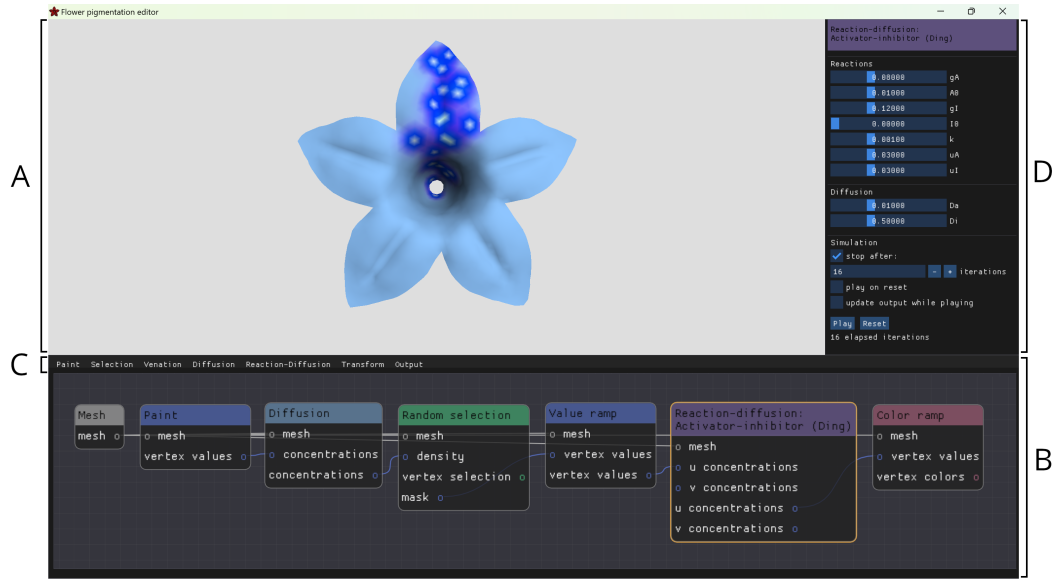


Figure 4.3: The implemented user interface. (A) The main view with the decorated mesh, (B) node graph editor with (C) node menu bar, (D) sidebar with tools and parameters. Author's archive.

Above the editor, the application displays the mesh being decorated with a pigmentation pattern (Fig. 4.3 A). The mesh view changes based on the selected node. For example, when the user selects a venation node, they see veins growing through the mesh, and when they select a diffusion node, they see dissolving colors. The view is refreshed in real-time upon simulation progress or operation adjustment. The parameters and options related to the selected node are displayed in the right sidebar (Fig. 4.3 D).

Appendix A contains an early prototype of the user interface, which models two case studies presented by Ringham et al. [5], roughly interpreting them in terms of interconnected operation nodes. The implemented user interface is shown in Figure 4.3. The nodes included in the final design are described in the following section.

4.3.1 Nodes

Mesh

The *Mesh node* represents the 3D model designated for texturing. When the node is selected, the sidebar allows the user to choose a predefined flower model or load a custom mesh. The main view displays the mesh as a wireframe (Fig. 4.4 A, 4.5 A). The node has a single output property — the *mesh* object, which appears as an input property in all other nodes. The Mesh node is created upon application start in a single copy and cannot be deleted.

Painting

The *Paint node* lets the user interactively specify positional information. The node's single input is the mesh, which is displayed in the main view as a colored surface or a wireframe. Initially, the mesh contains no positional information and appears black. By moving a brush over the mesh, the user can assign float values to the vertices under the cursor. The affected area then shifts to red according to the painted value (Fig. 4.4 B, 4.5 B). The brush radius, painted values, and mesh view can be configured in the sidebar. The Paint node outputs the resulting *vertex values*.

Selection

The selection nodes let the user select specific vertices on the mesh. They have two output properties in common: *vertex selection* and a *mask*. The mask is formed by vertex values, where the selected vertices are assigned 1 and the unselected 0. These vertex values are utilized for visualization similar to the Paint node — selected vertices are displayed in red, and the rest of the mesh is black (Fig. 4.5 D). The main view can show a solid surface or a wireframe.

The *Threshold Selection node* chooses vertices according to their assigned float value. It provides a configurable threshold parameter and takes vertex values as input (e.g., obtained from the Paint node). The selection contains the mesh vertices with values above the threshold.

The *Random Selection node* executes the algorithm described in Section 2.4.1 and chooses vertices according to an acceptance probability

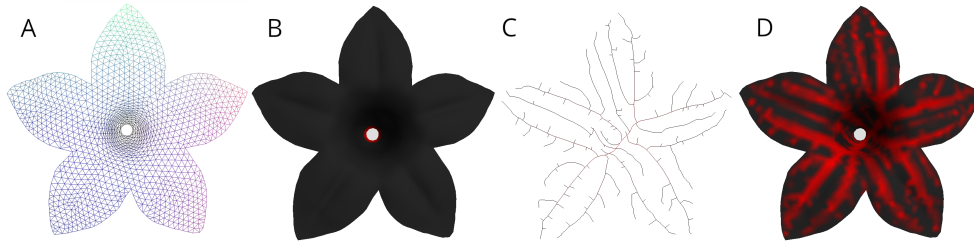


Figure 4.4: The individual stages in a venation pipeline. The main view as displayed in (A) the Mesh node, (B) the Paint node and Threshold Selection node (specifying the root vertices; visualization is the same), (C) the Venation node, (D) the Venation Imprint node. Author’s archive.

function. The function is specified as input vertex values ranging from 0 to 1. The node selects a given number of vertices, each with the probability corresponding to its assigned value. The user can configure the number of selected vertices in the sidebar. Another *maximum attempts* parameter prevents long or endless execution when the density function is poorly defined.

Venation

The editor menu offers two nodes related to vasculature patterns. The *Venation node* is a simulation node that generates a vein system. The *Venation Imprint node* “prints” an existing vein system onto the mesh, approximating vein segments with nearby vertices.

The Venation node takes input *root vertices*, which are specified as vertex selection and form the origin of the vein system. The veins are generated according to the parameters from Section 2.2 and outputted as a *venation* object. The main view displays a skeleton of the vein system with vein thickness encoded in red-black (Fig. 4.4 C).

The Venation Imprint node takes input venation and outputs vertex values between 0 and 1. The sidebar includes several parameters that adjust how strongly the veins are projected to the vertex values — practically, how contrasting the imprint is. The values are used for visualization like in other nodes with similar output (Fig. 4.4 D).

Diffusion

The *Diffusion node* simulates the diffusion process as described in Section 2.3.1. The node input is the initial morphogen *concentrations* represented as vertex values, which are then diffused and outputted in the same form. The concentrations are again visualized in red and black on the mesh surface (Fig. 4.5 C). In the sidebar, the user can set the diffusivity to control the blurring effect.

Reaction-Diffusion

The editor menu lists several nodes that simulate reaction-diffusion processes. Each node implements a different reaction-diffusion model for two morphogen types. The concentrations of participating morphogens are inputted and outputted as two sets of vertex values. Unlike in other nodes, the concentration input is optional; the nodes can automatically generate the initial values if none are specified. The sidebar exposes the parameters for the reaction equations and the diffusivity of each morphogen. The main view shows the mesh surface with morphogen concentrations visualized in colors (Fig. 4.5 E). The palette may differ by the system, but typically, the values are normalized and encoded in two color channels.

Transformation

In a *Transformation* dropdown, the editor menu provides the *Value Ramp node* and the *Color Ramp node*. They serve for mapping input vertex values to different vertex values or RGB *vertex colors*, respectively. The transformation works as follows:

- The input values are normalized to the range $\langle 0, 1 \rangle$.
- In the sidebar, the user configures two or more stop values that map to specific values or colors.
- When an input value lies between two defined stops, its new value or color is obtained by linear interpolation.
- Input values preceding the first stop or succeeding the last stop are clamped.

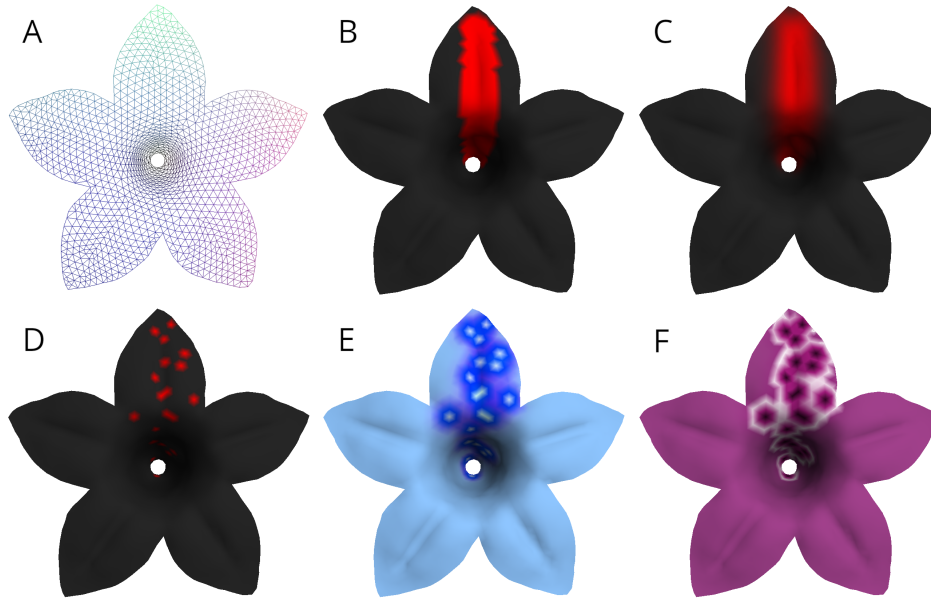


Figure 4.5: The individual stages in a *Digitalis purpurea* pipeline. The main view as displayed in (A) the Mesh node, (B) the Paint node, (C) the Diffusion node, (D) the Random Selection node (random distribution is provided by C), (E) the Reaction-Diffusion node — Activator-Inhibitor (Ding), (F) the Color Ramp node. Author’s archive.

This transformation should be flexible enough to suit a wide range of applications. The main view displays the mesh surface with the new color mapping (Fig. 4.5 F) or the vertex values depicted in red-black.

Output

The *Output* dropdown in the editor menu offers two nodes for exporting the pigmentation pattern. They both take vertex colors as input and have no output property. The *Colored Mesh node* lets the user re-export the mesh extended with per-vertex color information. Alternatively, the *Color Texture node* writes the colors into a 2D image based on the mesh UV coordinates. In the sidebar, the user can specify the destination file and the size of the color texture. The main view displays the mesh as a colored surface.

4.3.2 Simulation control

All simulation nodes have a special sidebar section dedicated to simulation control. The interface supports two control approaches. The first is interactive — the user watches the simulation run and presses the *Pause*, *Play*, or *Reset* buttons whenever they see fit. Alternatively, the user can preprogram the simulation to pause after a given number of iterations. The elapsed iterations are counted and displayed at the bottom of the section.

Additional parameters establish the resetting and updating behavior. The simulation is reset upon clicking the *Reset* button and when a node input property or a critical parameter changes. In such moments, it may or may not be desirable to start playing the simulation again. For example, restarting in-between consecutive parameter changes is wasteful, but restarting upon the button click might be comfortable. The user can specify their current preference using a checkbox parameter.

Another parameter affects the resetting of dependent nodes. The updates of output properties can be suspended during the simulation, which is useful when two simulation nodes depend on each other. In such a case, the output property of the first node would be frequently recomputed as the simulation progresses, causing the second node to reset repeatedly. In contrast, running the second operation only after the first one stops can avoid the needless resetting overhead and result in a smoother user experience.

5 Implementation

This chapter describes the technical implementation of the new application introduced in Chapter 4. The application is written in C++, utilizing the Standard C++ Library [36], the Boost C++ Libraries [37], and the GLM library [38] for linear algebra. The graphics are realized via the WebGPU API [39] using the Dawn WebGPU implementation [40] and the GLFW library [41] for window and input management.

5.1 Design implications

The design of the user interface and interaction style implies several requirements for technical implementation. First, the operations performed by nodes should be able to run independently and in multiple copies with different configurations. Apart from the dedicated node, each operation should be associated with a set of parameters and a visualization of the mesh state. The execution of node and visualization logic should be separated from the time-dependent operations for the application to run smoothly. Nevertheless, the parameters should be shared between these execution layers, being editable from the UI but directly and immediately affecting the operation. It should be possible to control the operation progress from the simulation nodes, and the visualization should be able to read the operation changes.

The operation nodes in the graph editor should be organized in an acyclic, directed graph. Upon each frame, the nodes should be processed in their reverse topological order so that all output properties are up-to-date before they are referenced from the dependent nodes. When a critical input property changes, the operation should be reset, unless specified otherwise (see Section 4.3.2).

5.2 Building elements

As the previous section suggests, the design leads to a specific structure of building elements. These are the operations, parameters, nodes, operation visualizations, and the graph editor. Additional necessary elements, such as a shader program, camera, and input manager, handle

the standard application logic. Figure 5.1 shows how all these elements translate into classes and how they are related. Nonetheless, the class names slightly differ from the nomenclature used so far: `NodeView` roughly corresponds to the graph editor, `IDrawable` represents the operation visualization — a drawable 3D object, and `ObjectProgram` is a shader program that renders such objects.

The boxes in bold generalize several specialized classes, constituting operation-specific parameters, nodes, drawables, and logic. The operation logic is not restricted to any class hierarchy, which enables it to be implemented without much knowledge about the surrounding application. In contrast, the nodes, parameters, and drawables implement predefined interfaces that allow them to communicate with each other.

5.2.1 Nodes

The graph editor is implemented in several custom classes that model its parts. Every type of node in the UI corresponds to a specific `Node` class and has one or more properties represented by a `Pin` class. Two pins of different nodes can be attached by a connection, which is modeled by the `Link` class. The editor itself is described by the `NodeView` class, which stores and manages all the nodes and links. Figure 5.2 shows a simplified class diagram of the mentioned classes.

The above classes internally utilize the `ImGui` library [42] and its `Node Editor` extension [43] to render the UI. Most user interactions, such as editor navigation, node movement, and drawing links, are already handled by the `Node Editor` in the background. The following sections discuss only the code I implemented, encapsulating the data and providing application-specific logic.

Pin class

The `Pin` class includes information about a node property and stores its value. The `PinType` attribute specifies the property data type and the `PinKind` attribute declares whether it is input or output. The property value is stored in a smart pointer and accompanied by a version mark, which is used for tracking changes.

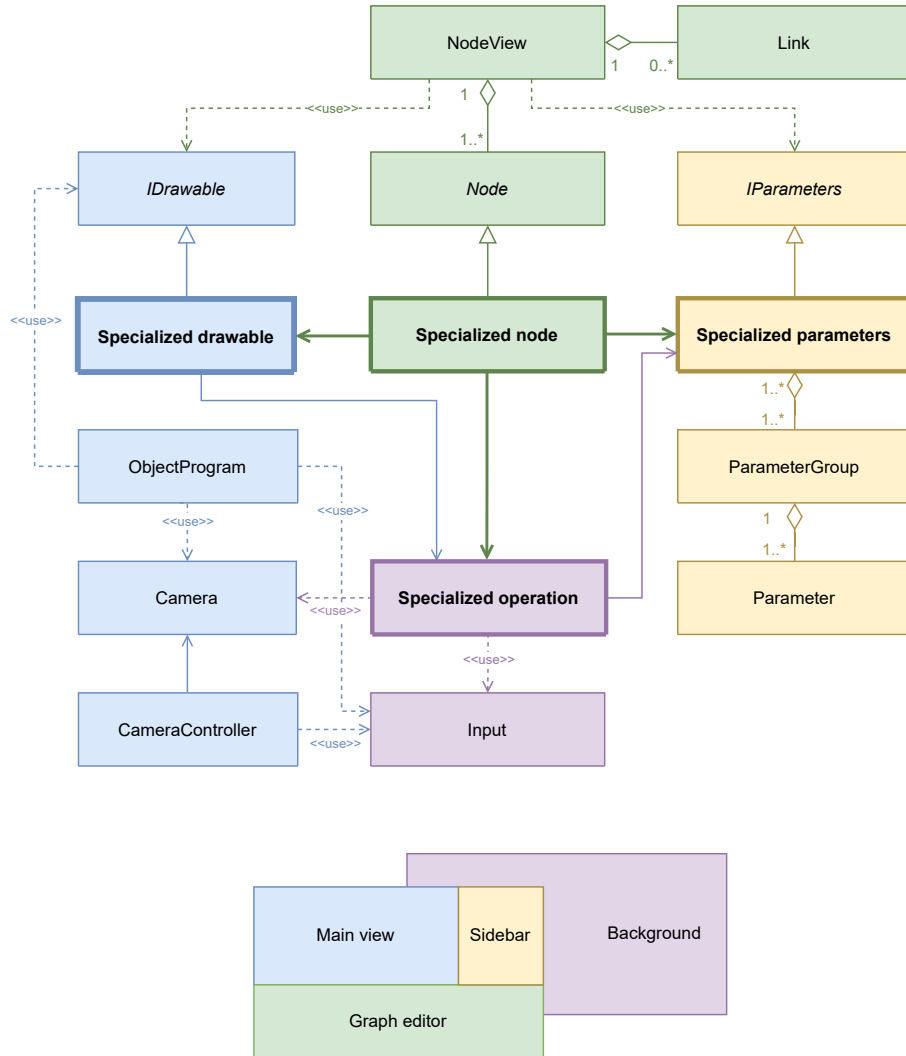
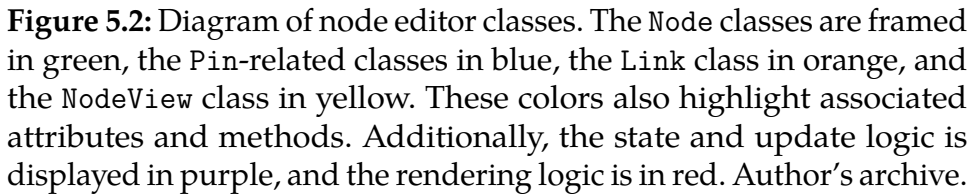


Figure 5.1: Diagram of main classes, their relationships, and their role in the application. The blue classes are closely related to the main application view, the yellow classes are primarily associated with the sidebar, the green classes compose the graph editor, and the purple classes represent background logic. The classes in bold generalize several specialized classes. Author's archive.



Node class

The abstract `Node` class (Fig. 5.2, middle) defines the common node functionality and provides several interface methods designated for implementation in its specialized subclasses. Among other attributes, it contains two collections of input and output Pins. A specific pin can be accessed by its name via convenience methods (Fig. 5.2, blue Node methods).

The most important Node methods serve for rendering (Fig. 5.2, red) and managing the state of the associated operation (Fig. 5.2, purple). The `render` method lays out the node visuals for the graph editor, and the `renderSidebar` method displays the node tools and parameters in the sidebar. The 3D visualization for the main view is obtained through the `getDrawable` method, which is implemented in Node subclasses.

The `update` method manages the operation state and is decomposed into several steps. First, the input links received as method arguments are compared with the current node inputs. Based on the pin values and versions, it is determined which pins were disconnected and which should be updated. If such pins are detected, their values are cleared or, respectively, copied from their linked counterparts. After that, the node and its associated operation are reset. Because the resetting logic closely depends on the operation kind, the `reset` method is left to be implemented by the subclasses. During reset, the node typically recreates the operation, reevaluates it, and updates the output properties. Finally, the output properties have their version marks recomputed.

SimulationNode class

`SimulationNode` extends the `Node` class with data and methods specific to time-dependent operations. Most importantly, it owns a thread executing an abstract `simulate` method with operation logic. Because the simulation is typically divided into repeated discrete steps, the subclasses implement it via a convenience `iterate` method. This method invokes the simulation according to the control parameters and counts the completed iterations.

`SimulationNode` also provides direct control over the simulation playing state. Internally, the state is kept in an atomic `shouldCompute` flag, which can be set and cleared using the `startSimulation` and `stopSimulation` methods. The UI elements for controlling the simulation (Sec. 4.3.2) are appended to the sidebar in an overload of the `renderSidebar` method.

Specialized node classes

The nodes executing specific operations are direct descendants of the `Node` and `SimulationNode` classes. The `SimulationNode` is inherited by classes representing the Venation node, Diffusion node, and Reaction-Diffusion node (Sec. 4.3.1). The `ReactionDiffusionNode` class is templated and can be instantiated with different reaction-diffusion systems. The remaining nodes derive their classes from the `Node` class, executing their operations synchronously.

The individual classes do not introduce new concepts but only implement the interfaces of their parents. Therefore, their description is omitted, while the noteworthy logic of individual operations is covered in Section 5.2.3.

NodeView class

The node editor is maintained by the `NodeView` class. Its primary task is to create, store, and destroy nodes and manage their linking and selection. `NodeView` also renders the node menu, editor canvas with nodes, and sidebar with associated tools and parameters. Additionally, it exposes the selected node's drawable for rendering with the `ObjectProgram`.

The nodes and links are stored in a directed graph [44]. Initially, the graph contains only a single *Mesh Node*, but the user can add new nodes by selecting from the node menu. Additionally, a new edge can be inserted into the graph by drawing a link between the pins of two nodes. The edge leads from the outputting node into the receiving one and is described by a `Link` instance. If the input pin is reconnected, its previous link is automatically destroyed. Any node and link can be deleted from the graph through a context menu, and the delete key can remove the selected node.

5.2.2 Parameters

The parameter classes serve as a communication medium between the nodes and the operations. They are stored by the nodes, rendered and edited in the sidebar, and read within the operations.

The parameters are built from simple blocks, which are assembled into operation-specific container classes. The `Parameter` class represents a primitive floating-point, integer, or boolean value, a semantically bounded radians value, or a composed color value. Each variant is associated with a suitable UI element provided by Dear ImGui [42], which displays and controls the parameter value.

5.2.3 Operations

The following sections provide technical details about the synchronous and simulated operations executed by the eponymous nodes. The Selection and Transformation operations are omitted as they are technically simple. For more context, please see Section 4.3.1, which explains all operation logic from a high-level perspective.

Mesh

The *Mesh operation* loads a 3D model and builds its internal mesh representation. The model is read from a PLY file [45] using the MSH_PLY library [46]. Like in the implementation of Ringham et al. [5], the mesh is represented by the half-edge data structure retaining neighborhood information. I implemented this data structure from scratch to fit several requirements. Firstly, the structure is read-only because the application does not change the mesh. Secondly, the imported mesh data are preprocessed. This is necessary because one vertex may be imported multiple times if shared between triangles where its normals or UV coordinates differ. Without preprocessing, the triangles would not be considered adjacent, which would be inconvenient for operations working with a connected surface. In order to avoid this, the neighborhood information is created on another data layer, where the vertices are united by position. The original data are kept for rendering and export. Finally, the mesh uses two sequences of rendering indices — one for the triangular solid surface and one for the linear wireframe. The latter is automatically computed.

Paint

The *Paint operation* associates the mesh vertices with float values based on user input. The mesh is painted by a virtual brush, whose radius and strength (painted value) are specified as operation parameters. The painting is triggered by a mouse drag received from the Input class (Fig. 5.1) via an observer binding. Subsequently, the mouse position is converted into a world-space ray that is cast toward the mesh to select vertices. If it hits any triangle, all adjacent vertices inside the brush radius are painted with the brush value. The brush radius is taken relative to the average edge length of the mesh triangles.

The updates of the Paint node output are postponed while the user paints to improve application responsiveness and prevent dependent nodes from frequent resetting.

Venation

The *Venation operation* implements the algorithm for generating open vasculature described in Section 2.2, extending it to 3D as hinted by Ringham et al. (Sec. 2.2.6). Initially, I implemented continuous uniform growth, but it produced cluttered vein systems as new auxin sources were added at every step across the whole surface. Consequently, the final application uses marginal growth, producing cleaner and more realistic veins. It is implemented discretely: the flower starts growing from the triangles with initial veins and gradually extends with neighboring triangles until the mesh is complete. The auxin sources are pre-generated at the simulation start and “activated” as their containing triangles appear in the growing flower.

Unfortunately, the Voronoi diagrams used by Runions et al. [18] for optimizing the algorithm (Sec. 2.2.5) lose their beneficial time complexity in 3D [24]. Therefore, I use a different optimization based on the mesh neighborhood information. Like the auxin sources, the vein nodes are associated with the triangles that contain them. The triangles are utilized for searching the neighborhood when inserting auxin sources and determining the projected positions of new veins. Furthermore, the auxin-vein distances are computed incrementally, always considering only newly added veins.

All operation parameters specifying distances are again taken relative to the average edge length of the mesh triangles.

Venation Imprint

The *Venation Imprint operation* approximates the generated vein system by float vertex values. It traces the triangles bridged by vein segments and measures the distance between their vertices and the segment. Gradually, minimal distances are accumulated for each vertex, creating an approximate imprint. For realistic appearance, the distances are weighted by inverse vein thickness, so thinner veins have less influence than thicker ones. Finally, the result is post-processed to regulate the value contrast.

Diffusion

The *Diffusion operation* implements the isotropic deterministic version of the algorithm by Ringham et al. [5] described in Section 2.3.1. Additionally, the operation maintains information about the minimum and maximum concentration values to facilitate later visualization.

Reaction-Diffusion

The *Reaction-diffusion operation* simulates the development and interactions of two morphogens. It combines the previously mentioned diffusion with a system of reaction equations as described in Section 2.5. In the implementation by Ringham et al. [5], these equations can be specified by the user at runtime (Sec. 3.1). I have decided not to implement this feature because it is technically demanding and would add unnecessary complexity for the users. Instead, the users are offered a predefined set of reaction-diffusion systems to explore and directly utilize. The provided systems include the Gray-Scott model and three versions of the activator-inhibitor model: the one utilized by Ding et al. [31], the one utilized by Ringham et al. [5], and another one imposing upper limits on the individual functions.

The models are implemented as separate parametrized classes with methods executing the reactions. They are then used as template arguments for a general `ReactionDiffusion` class, which combines

the reactions with diffusion and simulates the system behavior using the numerical midpoint method. This design makes it easy to add new models, avoiding the need to implement whole new operations and nodes.

Output

The *Output operations* allow exporting the created pigmentation pattern. *Colored mesh* takes a vector of colors and exports them as vertex properties along with the mesh. The data are written using the MSH_PLY library [46] into a PLY file [45]. The second *Color texture* operation exports only the color data into a 2D texture image. It uses a dedicated shader program, which renders the colors into a plane according to the UV coordinates of the mesh. Subsequently, the framebuffer data are copied to the CPU and saved as a PNG file [47] using the LodePNG library [48].

5.2.4 Drawables

Drawables are classes that visualize the operation effects (Sec. 5.2). They implement the `IDrawable` interface, which prescribes several getter methods for rendering: the primitive topology (such as a triangle or line list), number of rendered elements, position buffer, normal buffer, index buffer, and optionally, UV and data buffers. The last data buffer contains 3-component values used as colors.

The interface is implemented by the `Mesh` class and several specialized visualization classes. The mesh uses the line list topology, being rendered as a wireframe, and provides no data buffer — the `ObjectProgram` shader substitutes it with the normal buffer. Another two `IDrawable` implementations form the base classes for specialized usage. The first is `MeshColorDrawable`, which uses the existing mesh interface and supplements it with an additional data buffer with colors. It also allows switching the topology between triangles (solid surface) and lines (wireframe). The second base class is `VertexValuesDrawable`, which extends `MeshColorDrawable` with logic for visualizing vertex values.

The specialized drawables are associated with specific node operations. Most of them trivially derive `VertexValuesDrawable` or ex-

tend `MeshColorDrawable`, providing the colors directly or obtaining them by custom transformations. As the only exception, `Venation` provides an independent `IDrawable` implementation. It reads the venation graph described in Section 5.2.3 and gradually builds its 3D wireframe. The drawable regularly updates its buffers as the operation produces new veins.

5.3 Element interaction

The building elements introduced in Section 5.2 need to communicate. Namely, the parameters controlled from the UI must be synchronized with the simulations running in background threads, and the visualizing drawables need to know how the simulation data evolves. Furthermore, the nodes need to share data along existing property links.

5.3.1 Parameter synchronization

The parameters instance is stored and managed in the node, while the operation receives a read-only reference. The values of individual parameters are represented by the `std::atomic` class from the C++ Standard Library [36], which ensures that value reads and writes are properly ordered and do not interfere. The same approach is used for the attributes controlling simulation, as mentioned in Section 5.2.1.

5.3.2 Node property updates

The communication between nodes uses the smart `std::shared_ptr` pointer from the C++ Standard Library [36]. This class allows sharing the pointed value between multiple copies of the pointer, destroying the value only when the last pointer is removed. Additionally, the pointers can safely reference one value from different threads [49].

The shared pointers store operation objects and property values. They are beneficial because they allow the nodes to update independently of one another and their operations. For example, let there be a `Paint` node and a dependent `Diffusion` node, and suppose the `Diffusion` simulation is running. When the user paints on the mesh, the `Paint` node's output property changes. At that moment, different

entities must hold different versions of the painted data. Specifically, the Diffusion node's input should not be directly rewritten but only marked as outdated so that the node can react to the change. Also, the diffused paint should not be changed mid-simulation, but the simulation should finish its current iteration and then be gracefully reset. Using copied shared pointers can achieve such independence — the original paint values can be referenced while needed.

5.3.3 Visualization updates

Lastly, it is necessary to establish communication between a simulation and its drawable. This concerns Diffusion, Reaction-diffusion, and Venation. The first two are relatively straightforward because their simulations only change the values in a vector of stable size and memory location. Considering that, the drawables read from the vector without synchronization. Such access is theoretically incorrect, potentially leading to read-write race conditions. However, it is more tolerable for the visualization to be occasionally imprecise than laggy due to waiting for correct data. The inconsistencies will likely be small and thus unrecognizable to the user in a constantly changing picture.

The visualization of Venation is slightly complicated because the simulation incrementally enlarges the vein graph, which can eventually contain high numbers of veins. The drawable has to reflect this growth, extending the vertex buffers with new veins. In order to reduce buffer reallocations, the drawable allocates twice the space that is needed upon each resize, reserving the rest for future additions. Another optimization considers that once a vein segment is created, it never moves or disappears. Therefore, the buffers need not be entirely rewritten but only extended. The Venation operation supports such incremental updates by keeping track of the newly added vein nodes. It stores them in a wait-free queue [50], which is concurrently read by the drawable. Nevertheless, this optimization applies only to vein positions and indices. Unfortunately, the vein thickness constantly changes in the whole vein graph, so the buffer with its color encoding must be repeatedly rewritten.

6 Results

This chapter provides an overview of the flower pigmentation patterns that can be created in the newly implemented application. First, I explored the parametric space of venation patterns (Fig. 6.1). Then I tried to reproduce the case studies presented by Ringham et al. [5] (Fig. 6.2–6.7). The following pages compare the obtained visuals, showing significant modeling steps and final patterns. The simulation times are listed in Table 6.1, and the used nodes are given in Appendix B.

Figures 6.2–6.4 show patterns that were closely reproduced using the same principles and similar parameters as Ringham et al. [5]. The patterns in Figure 6.2 combine painting and diffusion; the reproductions use only the deterministic variant. The *Digitalis purpurea* pattern (Fig. 6.3) was obtained by painting, diffusion, random vertex selection, and reaction diffusion. The *Platycodon grandiflorus* pattern was created using venation and venation imprint.

Figures 6.5–6.7 show the patterns that were roughly approximated. In the reference *Dianthus barbatus* pattern (Fig. 6.5), a painted arc is blurred by anisotropic diffusion that spreads faster along the veins. In contrast, the approximation multiplies the painted arc with the vein imprint and diffuses them. A similar difference regards the *Phalaenopsis* pattern (Fig. 6.6): in the reference, the veins control the reaction-diffusion parameters, while the approximation translates them into initial morphogen concentrations. The last pattern, *Mimulus guttatus* (Fig. 6.7), is directly generated by reaction-diffusion. Its simulation area is confined in the reference, whereas the approximation utilizes the whole mesh and later clips the result.

While the current implementation can only approximate the last three patterns, it can be easily extended to support them fully by allowing the parameters to use multiple values differing by vertex. The initial design prototype in Appendix A shows a possible UI for this feature (Fig. A.5–A.8): such parameters become optional node properties, using vertex values obtained via node links or a uniform value set from the sidebar.

Currently, the application also cannot handle 3 of the 10 case studies presented by Ringham et al. [5]: *Kohleria*, *Trichoglotys*, and *Phalaenopsis Nankung's 4.55PM*. These patterns could be also easily supported

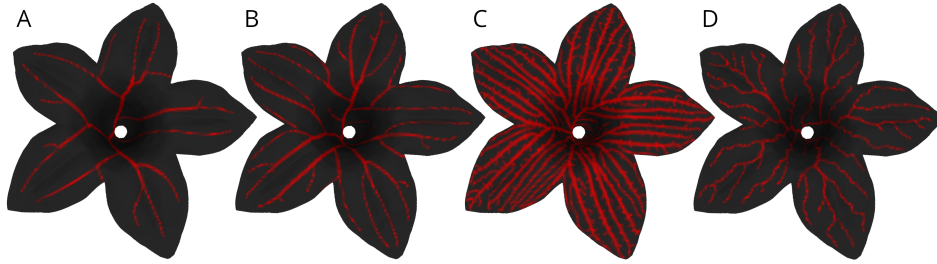


Figure 6.1: Example effects of venation parameters. (A–C) Growth speed of 1 iteration per increment and varied *kill distance* of 10, 5, and 1. (D) Growth speed of 5 iterations per increment.

by implementing additional reaction-diffusion models, a new node exposing the diffusion concentration gradient, and a node for generating random vertex values.

Table 6.1: The iterations and time (m:s) required by the simulation nodes. Node types that were not used in the pipeline are dashed. The computations were performed using the AMD Ryzen 7 5800H 3.20 GHz 8-core CPU, running in debug build under Windows 11. The mesh for all case studies contains 21,394 triangles.

Name	Fig.	Diffusion		Reaction-diff.		Venation	
		Steps	Time	Steps	Time	Steps	Time
<i>Primula</i>	6.2	113	0:04	–	–	–	–
<i>Arctostaphylos</i>	6.2	122	0:05	–	–	–	–
<i>Digitalis</i>	6.3	71	0:03	185	0:43	–	–
<i>Platycodon</i>	6.4	–	–	–	–	126	0:33
<i>Dianthus</i>	6.5	550	0:21	–	–	105	0:16
<i>Phalaenopsis</i>	6.6	–	–	3,305	12:46	163	0:14
<i>Mimulus</i>	6.7	–	–	10,276	39:47	–	–

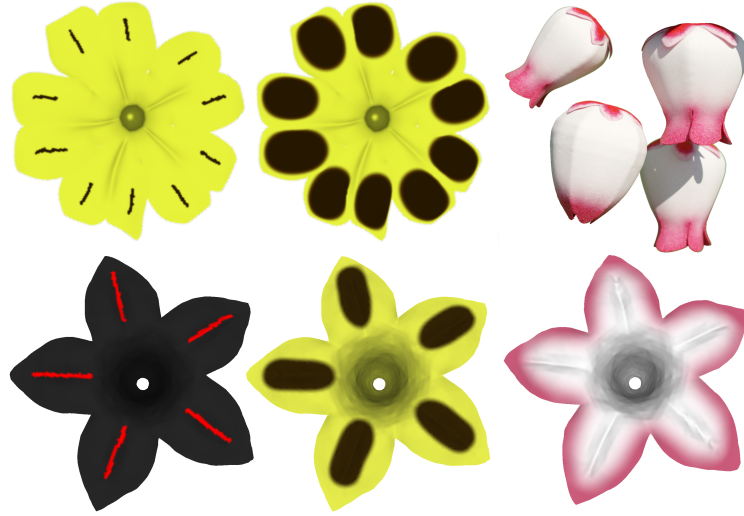


Figure 6.2: Reproduced patterns (bottom row): *Primula* $\times$ *polyantha* (left and center) and *Arctostaphylos uva-ursi* (right). From the left: painted marks, two final diffused patterns. Top row adapted from [5].

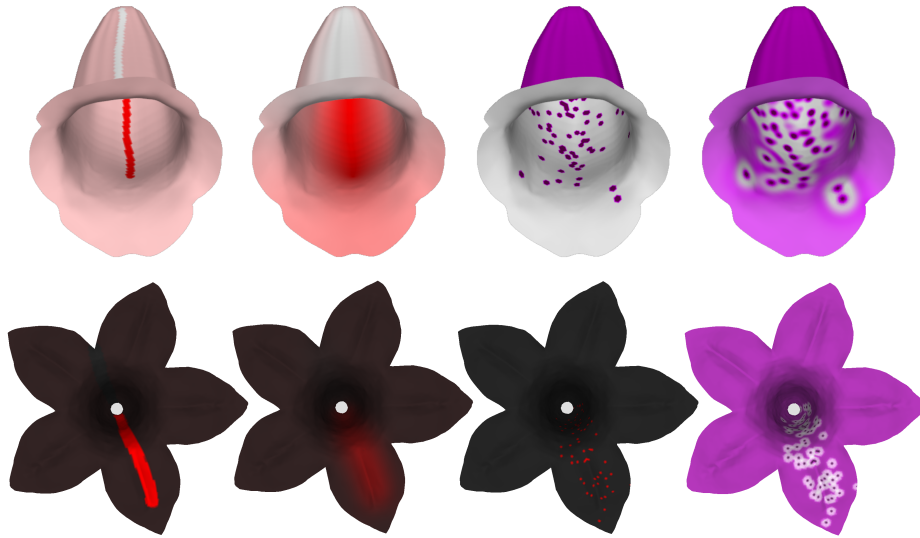


Figure 6.3: Reproduced pattern (bottom row): *Digitalis purpurea*. From the left: painted mark, diffusion, random vertex selection, and final reaction-diffusion pattern. Top row taken from [5].



Figure 6.4: Reproduced pattern (bottom row): *Platycodon grandiflorus*. From the left: venation, venation imprint (bottom only), and final pattern. Top row adapted from [5].

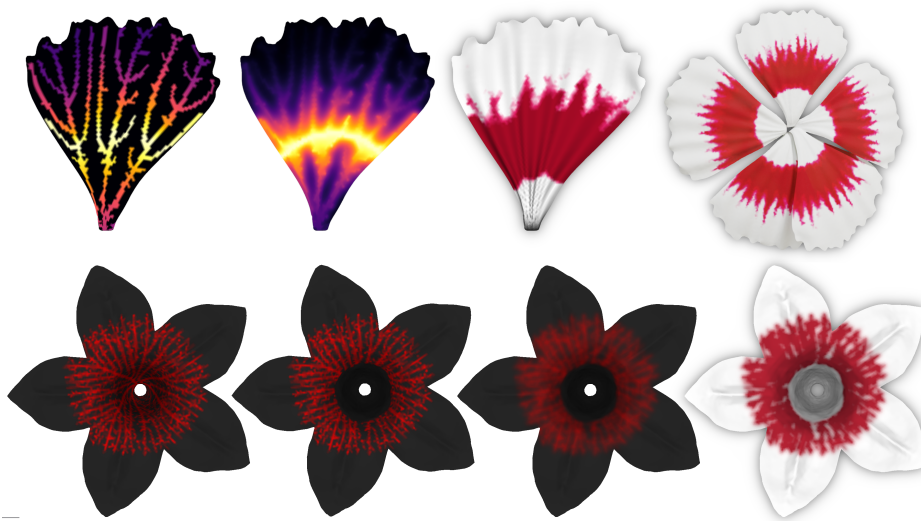


Figure 6.5: Approximated pattern (bottom row): *Dianthus barbatus*. From the left: venation, multiplication clip, diffusion, and final pattern. Top row was adapted from [5] and shows anisotropic diffusion spreading faster along the veins.

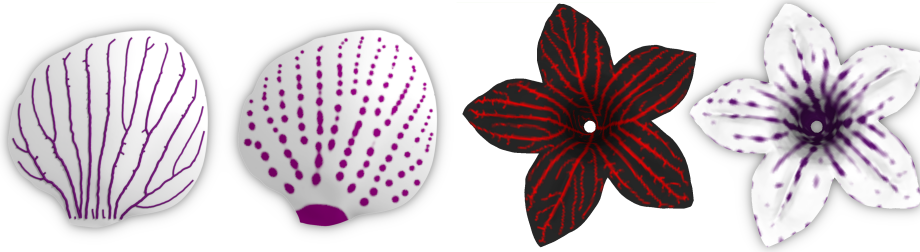


Figure 6.6: Approximated pattern (right half): *Phalaenopsis Chian Xen Leopard Stripes*. The first and second images show venation controlling reaction-diffusion, the third and fourth images show venation imprint post-processed by reaction-diffusion. Left half taken from [5].



Figure 6.7: Approximated pattern (bottom row): *Mimulus guttatus*. From the left: A reaction-diffusion pattern, multiplication clip, and final pattern. The top row was taken from [5] and shows a reaction-diffusion pattern simulated only in the pink mesh area.

7 Discussion and future development

The application's primary goal was to cover all patterning mechanisms and make them available from a new user interface. The implementation was approached breadth-first, focusing on the bigger picture. Consequently, the individual application parts could be revisited in more detail and further improved, extended, and optimized. This chapter discusses significant development opportunities that were not mentioned before.

Several possible improvements regard the main patterning operations. Firstly, diffusion could be extended to support anisotropic and stochastic spread. Venation might also generate closed vasculature patterns, and its visualization could use a 3D tubular skeleton instead of the current linear wireframe. Additionally, the set of supported reaction-diffusion systems could be extended, and the parametric space could be explored to identify the range of useful values. The effect of different initial concentrations should be studied as well — a few experiments suggest that choosing suitably small values may result in significantly faster pattern emergence.

From the user's perspective, an important feature yet to be implemented is the ability to save the created node pipeline. Nevertheless, some design questions must be decided first, such as whether to forbid or track parameter adjustments during the simulation and how to ensure pipeline compatibility between application versions. Additionally, the users might appreciate the support of alternative mesh and texture formats. When these known tasks are addressed, further development would greatly benefit from user testing with the target audience.

Finally, the application could be optimized in multiple respects. For example, diffusion and reaction-diffusion computations could be executed in parallel for every triangle and vertex, which could be straightforwardly achieved using the OpenMP directives [51]. In venation, the vein creation and auxin source removal might be parallelized similarly, possibly requiring a switch to the Parallel Boost Graph Library [52]. Furthermore, the painting operation could use a spatial indexing structure for faster raycasting. Lastly, the source code should be refactored to reduce indirection and memory allocations.

Conclusion

The thesis studied a recently published paper from algorithmic botany focusing on *Modeling flower pigmentation patterns* [5]. The goals were to evaluate the authorial implementation of the modeling method and develop a new application with an improved user interface. The motivation was to make the method more accessible to artists who model vegetation for computer games and visualization.

Reviewing current vegetation modeling approaches and software suggested that the pigmentation modeling method may align well with the artistic workflow. The software extensively utilized procedural techniques, included many modeling parameters, and worked with complex, structured information. These qualities are also natural to pigmentation modeling. Although they may seem too technical for creative use, the reviewed software is widely adopted.

Nevertheless, the existing program implementing the method had several significant usability problems. Most importantly, the modeling workflow did not correspond to the pipeline nature of the method, and its stages blended. Furthermore, the user's actions and modeling state were not explicitly represented. Overall, the user interface needed improvement in many aspects.

The newly developed application addresses these identified problems. It makes the modeling state explicit and separates the individual modeling stages, representing them by nodes composed in a graph. The nodes implement all pigmentation-patterning mechanisms and additional utility tasks. This type of user interface is commonly utilized in 3D modeling suites and vegetation modeling toolkits. It allows the users to think about creation as a structured process and lets them design various procedural pipelines. The created color patterns can be included in the mesh or exported as a 2D image texture.

The application can reproduce or approximate most case studies demonstrated in the paper. The remaining case studies were considered in the design phase and could be supported in the future after implementing new nodes or extending the current functionality. Other possible improvements include the optimization and parallelization of computations. Finally, the achieved user experience could be tested on the target audience and refined according to the acquired feedback.

Bibliography

1. TURING, Alan. The Chemical Basis of Morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* [online]. 1952, vol. 237, no. 641, pp. 37–72 [visited on 2023-03-14]. Available from: <https://royalsocietypublishing.org/doi/pdf/10.1098/rstb.1952.0012>.
2. LINDENMAYER, Aristid. Mathematical models for cellular interactions in development. I and II. *Journal of Theoretical Biology*. 1968, vol. 18, no. 3, pp. 280–315. ISSN 0022-5193. Available from DOI: [https://doi.org/10.1016/0022-5193\(68\)90079-9](https://doi.org/10.1016/0022-5193(68)90079-9).
3. PRUSINKIEWICZ, Przemysław; LINDENMAYER, Aristid. The Algorithmic Beauty of Plants. In: *The Virtual Laboratory*. Springer New York, NY, 1990. Available from DOI: <https://doi.org/10.1007/978-1-4613-8476-2>.
4. *Algorithmic Botany: Publications* [online]. Calgary: Biological Modeling and Visualization research group [visited on 2023-03-14]. Available from: <http://algorithmicbotany.org/papers/>.
5. RINGHAM, Lee; OWENS, Andrew; CIESLAK, Mikolaj; HARDER, Lawrence D.; PRUSINKIEWICZ, Przemysław. Modeling Flower Pigmentation Patterns. *ACM Transactions on Graphics*. 2021, vol. 40, no. 6. ISSN 0730-0301. Available from DOI: 10.1145/3478513.3480548.
6. *SpeedTree — 3D Vegetation Modeling and Middleware*. Lexington, South Carolina, United States: Interactive Data Visualization, Inc., 2023. Available also from: <https://store.speedtree.com>.
7. *PlantFactory: Overview*. Exton, Pennsylvania, United States: E-ON Software, Sarl c/o Bentley Systems Inc., 2023. Available also from: <https://info.e-onsoftware.com/plantfactory>.
8. *SpeedTree: Modeler 9 Documentation*. Lexington, South Carolina, United States: Interactive Data Visualization, Inc., 2022. Available also from: <https://docs9.speedtree.com/modeler/doku.php>.

BIBLIOGRAPHY

9. SEEBACHER, Daniel. *Using vertex colors from PlantFactory & PlantCatalog in other apps*. Exton, Pennsylvania, United States: E-ON Software, Sarl c/o Bentley Systems Inc., 2023. Available also from: <https://info.e-onsoftware.com/blog/using-vertex-color-sets-from-plantfactory-and-plantcatalog-plants-in-other-apps>.
10. DANIEL SEEBACHER, e-on software. *PlantFactory: Understanding the Autogrowth node. Part 1 - Overview*. YouTube, 2020. Available also from: <https://www.youtube.com/watch?v=a7Kzb4xXlH4>.
11. TURK, Greg. Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion. *ACM SIGGRAPH Computer Graphics*. 1991, vol. 25, no. 4, pp. 289–298. ISSN 0097-8930. Available from doi: 10.1145/127719.122749.
12. FOWLER, Deborah R.; MEINHARDT, Hans; PRUSINKIEWICZ, Przemyslaw. Modeling Seashells. *ACM SIGGRAPH Computer Graphics*. 1992, vol. 26, no. 2, pp. 379–387. ISSN 0097-8930. Available from doi: 10.1145/142920.134096.
13. MEINHARDT, Hans. The Algorithmic Beauty of Sea Shells. In: *The Virtual Laboratory*. Springer Berlin, Heidelberg, 2009. Available from doi: <https://doi.org/10.1007/978-3-540-92142-4>.
14. *Blender: The Free and Open Source 3D Creation Suite*. Blender Institute, Amsterdam, the Netherlands: Blender Foundation, 2023. Available also from: <https://blender.org>.
15. JAKOB, Wenzel. *Instant Meshes*. 2023. Available also from: <https://github.com/wjakob/instant-meshes>.
16. SCHWINN, Kathy; VENAIL, Julien; SHANG, Yongjin; MACKAY, Steve; ALM, Vibeke; BUTELLI, Eugenio; OYAMA, Ryan; BAILEY, Paul; DAVIES, Kevin; MARTIN, Cathie. A small family of MYB-regulatory genes controls floral pigmentation intensity and patterning in the genus *Antirrhinum*. *The Plant Cell*. 2006, vol. 18, no. 4, pp. 831–851. Available from doi: 10.1105/tpc.105.039255.
17. SHANG, Yongjin; VENAIL, Julien; MACKAY, Steve; BAILEY, Paul; SCHWINN, Kathy; JAMESON, Paula; MARTIN, Cathie; DAVIES, Kevin. The molecular basis for venation patterning of pigmentation and its effect on pollinator attraction in flowers of

- Antirrhinum. *The New Phytologist*. 2011, vol. 189, no. 2, pp. 602–615. Available from doi: 10.1111/j.1469-8137.2010.03498.x.
18. RUNIONS, Adam; FUHRER, Martin; LANE, Brendan; FEDERL, Pavol; ROLLAND-LAGAN, Anne-Gaëlle; PRUSINKIEWICZ, Przemyslaw. Modeling and Visualization of Leaf Venation Patterns. *ACM Transactions on Graphics*. 2005, vol. 24, no. 3, pp. 702–711. ISSN 0730-0301. Available from doi: 10.1145/1073204.1073251.
 19. SACHS, Tsvi. The Control of the Patterned Differentiation of Vascular Tissues. In: WOOLHOUSE, H.W. (ed.). *Academic Press*, 1981, vol. 9, pp. 151–262. *Advances in Botanical Research*. ISSN 0065-2296. Available from doi: [https://doi.org/10.1016/S0065-2296\(08\)60351-1](https://doi.org/10.1016/S0065-2296(08)60351-1).
 20. MITCHISON, Graeme J. The dynamics of auxin transport. *Proceedings of the Royal Society London B*. 1980, vol. 209, no. 1177, pp. 489–511. ISSN 0080-4649. Available from doi: <https://doi.org/10.1098/rspb.1980.0109>.
 21. GOTTLIEB, Marc E. Angiogenesis and Vascular Networks: Complex Anatomies from Deterministic Non-Linear Physiologies. In: *Growth Patterns in Physical Sciences and Biology*. Ed. by GARCIA-RUIZ, Juan Manuel; LOUIS, Enrique; MEAKIN, Paul; SANDER, Leonard M. Boston, MA: Springer US, 1993, pp. 267–276. ISBN 978-1-4615-2852-4. Available from doi: 10.1007/978-1-4615-2852-4_29.
 22. MURRAY, Cecil D. The Physiological Principle of Minimum Work. *Proceedings of the National Academy of Sciences*. 1926, vol. 12, no. 3, pp. 207–214. Available from doi: 10.1073/pnas.12.3.207.
 23. PREPARATA, Franco P.; SHAMOS, Michael Ian. Computational Geometry: An Introduction. In: *Monographs in Computer Science*. Springer New York, NY, 1985. Available from doi: <https://doi.org/10.1007/978-1-4612-1098-6>.
 24. OKABE, Atsuyuki; BOOTS, Barry; SUGIHARA, Kokichi. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. USA: John Wiley & Sons, Inc., 1992. ISBN 0471934305. Available also from: <https://dl.acm.org/doi/abs/10.5555/135734>.

25. HÄDRICH, Torsten; BENES, Bedrich; DEUSSEN, Oliver; PIRK, Sören. Interactive Modeling and Authoring of Climbing Plants. *Computer Graphics Forum*. 2017, vol. 36, no. 2, pp. 49–61. Available from DOI: <https://doi.org/10.1111/cgf.13106>.
26. RUBELJ, I. Specialized Recombination. In: *Brenner's Encyclopedia of Genetics (Second Edition)*. Second Edition. San Diego: Academic Press, 2013, pp. 498–500. ISBN 978-0-08-096156-9. Available from DOI: <https://doi.org/10.1016/B978-0-12-374984-0.01449-2>.
27. GILLESPIE, Daniel. The Chemical Langevin Equation. *Journal of Chemical Physics*. 2000, vol. 115, pp. 297–306. Available from DOI: [10.1063/1.481811](https://doi.org/10.1063/1.481811).
28. GIERER, Alfred; MEINHARDT, Hans. A theory of biological pattern formation. *Kybernetik*. 1972, vol. 12, pp. 30–39. Available from DOI: <https://doi.org/10.1007/BF00289234>.
29. GRAY, Peter; SCOTT, Stephen K. Autocatalytic reactions in the isothermal, continuous stirred tank reactor: Oscillations and instabilities in the system $A + 2B \rightarrow 3B$; $B \rightarrow C$. *Chemical Engineering Science*. 1984, vol. 39, no. 6, pp. 1087–1097. ISSN 0009-2509. Available from DOI: [https://doi.org/10.1016/0009-2509\(84\)87017-7](https://doi.org/10.1016/0009-2509(84)87017-7).
30. ANDREUX, Mathieu; RODOLÀ, Emanuele; AUBRY, Mathieu; CREMERS, Daniel. Anisotropic Laplace-Beltrami Operators for Shape Analysis. In: *Computer Vision - ECCV 2014 Workshops*. Springer, 2015, vol. Lecture Notes in Computer Science 8928, pp. 299–312. Available from DOI: https://doi.org/10.1007/978-3-319-16220-1_21.
31. DING, Baoqing; PATTERSON, Erin L.; HOLALU, Srinidhi V.; LI, Jingjian; JOHNSON, Grace A.; STANLEY, Lauren E.; GREENLEE, Anna B.; PENG, Foen; JR., H.D. Bradshaw; BLINOV, Michael L.; BLACKMAN, Benjamin K.; YUAN, Yao-Wu. Two MYB Proteins in a Self-Organizing Activator-Inhibitor System Produce Spotted Pigmentation Patterns. *Current Biology*. 2020, vol. 30, no. 5, pp. 1–13. Available from DOI: <https://doi.org/10.1016/j.cub.2019.12.067>.

32. MIURA, Takashi; MAINI, Philip K. Periodic pattern formation in reaction–diffusion systems: An introduction for numerical simulation. *Anatomical Science International*. 2004, vol. 79, pp. 112–123. Available from doi: 10.1111/j.1447-073x.2004.00079.x.
33. RINGHAM, Lee; OWENS, Andrew; CIESLAK, Mikolaj; HARDER, Lawrence; PRUSINKIEWICZ, Przemyslaw. *Modeling Flower Pigmentation Patterns: Supplementary video*. University of Calgary, 2021. Available also from: http://algorithmicbotany.org/papers/pigmentation_patterns.sigasia2021.opt.mp4.
34. *Unity: Real-Time Development Platform*. San Francisco, California, United States: Unity Technologies, 2023. Available also from: <https://unity.com/features/shader-graph>.
35. *Shader graph: Build beautiful, dynamic shaders using a node-based visual authoring tool*. San Francisco, California, United States: Unity Technologies, 2023. Available also from: <https://unity.com/features/shader-graph>.
36. *C++ Standard Library*. 2023. Available also from: https://en.cppreference.com/w/cpp/standard_library.
37. DAWES, Beman; ABRAHAM, David; RIVERA, Rene; BOOST COMMUNITY, the. *Boost C++ Libraries*. 2023. Available also from: <https://www.boost.org/>.
38. CREATION, G-Truc. *OpenGL Mathematics (GLM)*. 2005. Available also from: <https://github.com/g-truc/glm>.
39. W3C. *WebGPU*. Ed. by NINOMIYA, Kai; JONES, Brandon; MAXFIELD, Myles C. 2023. Available also from: <https://www.w3.org/TR/webgpu/>.
40. DEVELOPERS, "WebGPU native". *Dawn: A WebGPU implementation*. 2019. Available also from: <https://github.com/hexops/dawn>.
41. GEELNARD, Marcus; LÖWY, Camilla. *GLFW*. 2019. Available also from: <https://www.glfw.org/>.
42. CORNUT, Omar. *Dear ImGui*. 2023. Available also from: <https://github.com/ocornut/imgui>.

43. CICHON, Michał. *Node Editor in ImGui*. 2019. Available also from: <https://github.com/thedmd/imgui-node-editor>.
44. SIEK, Jeremy; LEE, Lie-Quan; LUMSDAINE, Andrew. *The Boost Graph Library (BGL)*. 2001. Available also from: https://www.boost.org/doc/libs/1_82_0/libs/graph/doc/index.html.
45. TURK, Greg. *PLY — Polygon File Format*. 1994. Available also from: <http://paulbourke.net/dataformats/ply/>.
46. HALBER, Maciej. *msh — Maciej's Toolbox*. 2022. Available also from: <https://github.com/mhalber/msh>.
47. GROUP, PNG Development. *Portable Network Graphics: An Open, Extensible Image Format with Lossless Compression*. Ed. by ROELOFS, Greg. 2022. Available also from: <http://www.libpng.org/pub/png/>.
48. VANDEVENNE, Lode. *LodePNG*. 2018. Available also from: <https://lodev.org/lodepng/>.
49. *std::shared_ptr*. 2023. Available also from: https://en.cppreference.com/w/cpp/memory/shared_ptr.
50. BLECHMANN, Tim. *Boost.Lockfree*. 2011. Available also from: https://www.boost.org/doc/libs/1_82_0/doc/html/lockfree.html.
51. OPENMP. *OpenMP: The OpenMP API specification for parallel programming*. 2023. Available also from: <https://www.openmp.org/>.
52. EDMONDS, Nick; GREGOR, Douglas; LUMSDAINE, Andrew. *The Parallel Boost Graph Library*. 2009. Available also from: https://www.boost.org/doc/libs/1_82_0/libs/graph_parallel/doc/html/index.html.

A Initial design prototype

Figures A.1–A.15 show an initial design prototype of the user interface. It models two case studies presented by Ringham et al. [5], *Kohleria* and *Platycodon grandiflorus*, roughly interpreting them in terms of interconnected operation nodes.

The nodes and parameters used in the prototype are only tentative and have changed in the final design and implementation.

The flower images with intermediate patterns are taken from the paper [5] and the supplementary video [33], except for the resulting *Platycodon* pattern (Fig. A.15), which was imitated according to a realistic render from the paper.

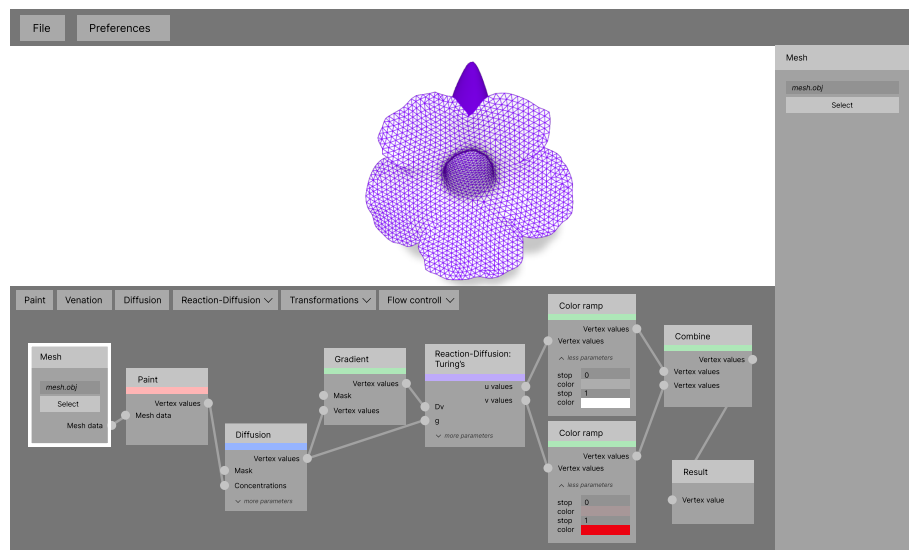


Figure A.1: UI prototype: *Kohleria* case study. Mesh node. Author's archive.

A. INITIAL DESIGN PROTOTYPE

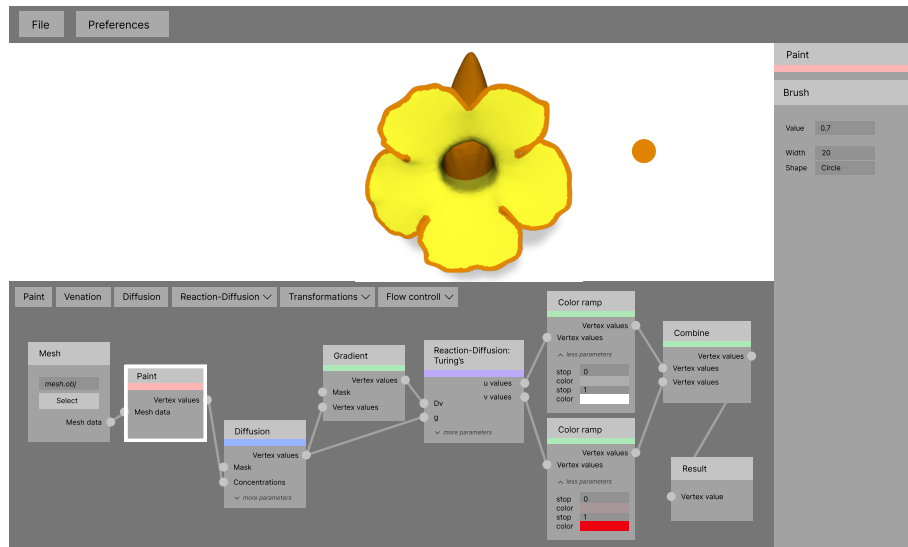


Figure A.2: UI prototype: *Kohleria* case study. Paint node. Author's archive.

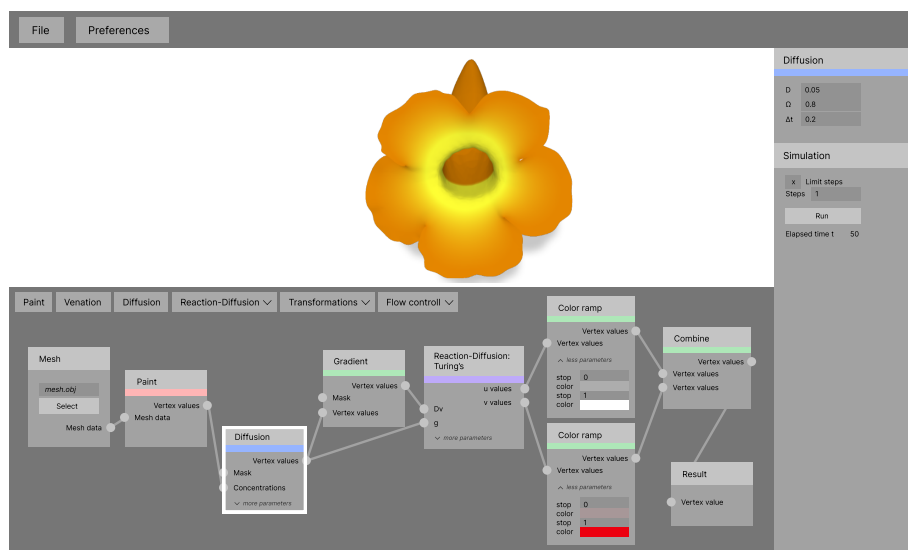


Figure A.3: UI prototype: *Kohleria* case study. Diffusion node. Author's archive.

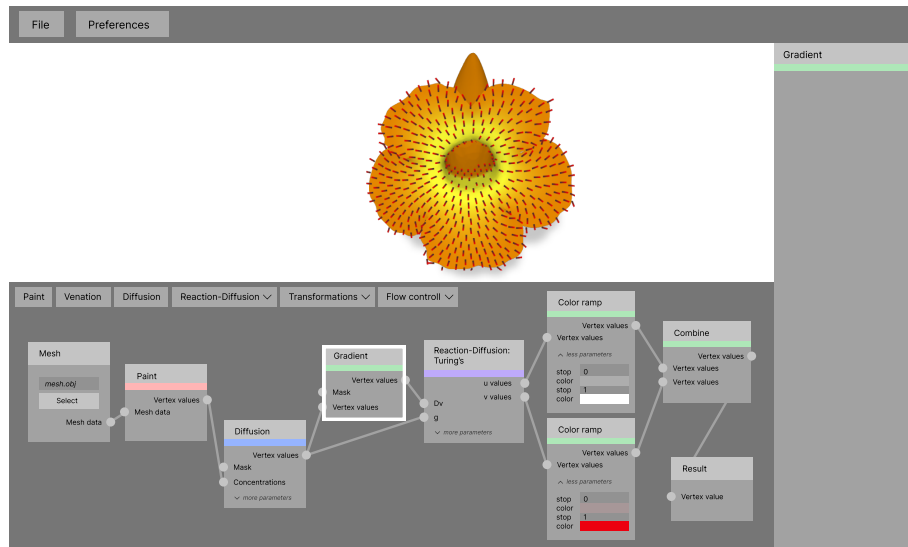


Figure A.4: UI prototype: *Kohleria* case study. Gradient node. Author's archive.

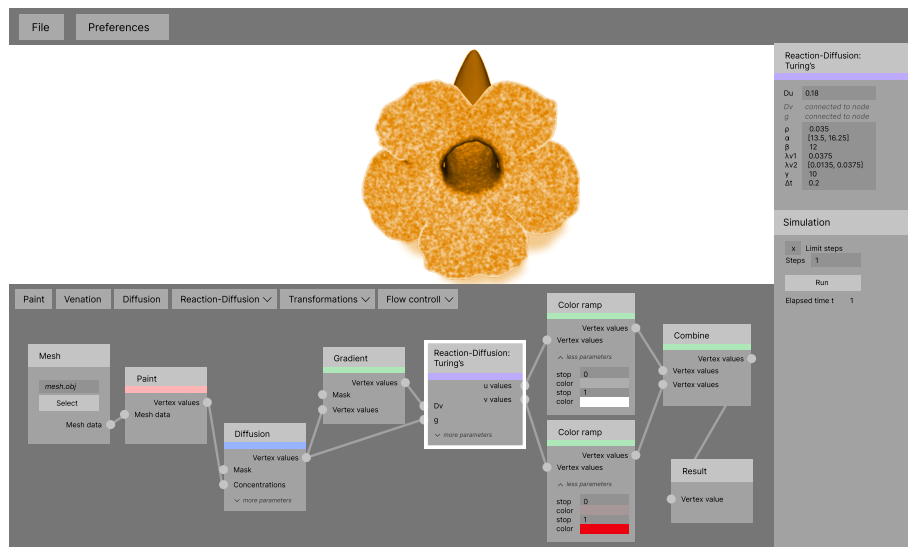


Figure A.5: UI prototype: *Kohleria* case study. Reaction-diffusion node at simulation time $t = 1$. Author's archive.

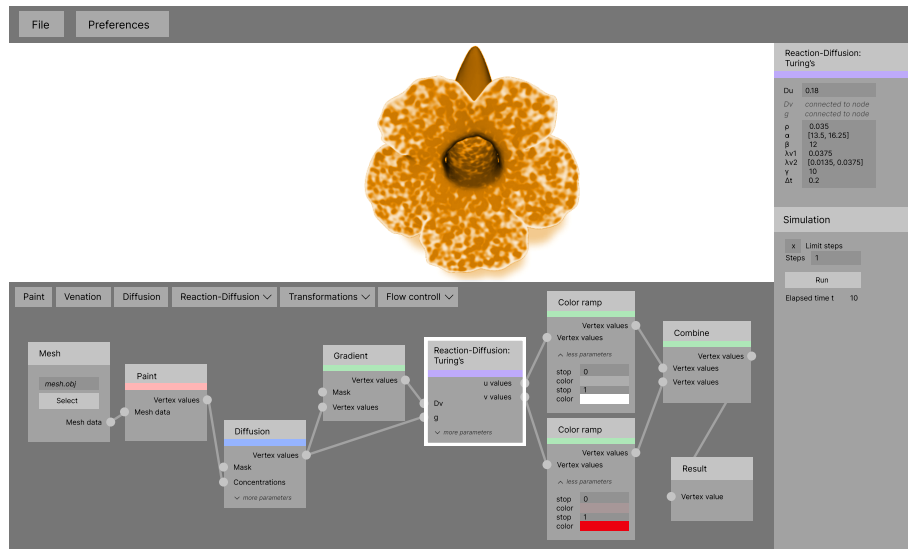


Figure A.6: UI prototype: *Kohleria* case study. Reaction-diffusion node at simulation time $t = 10$. Author's archive.

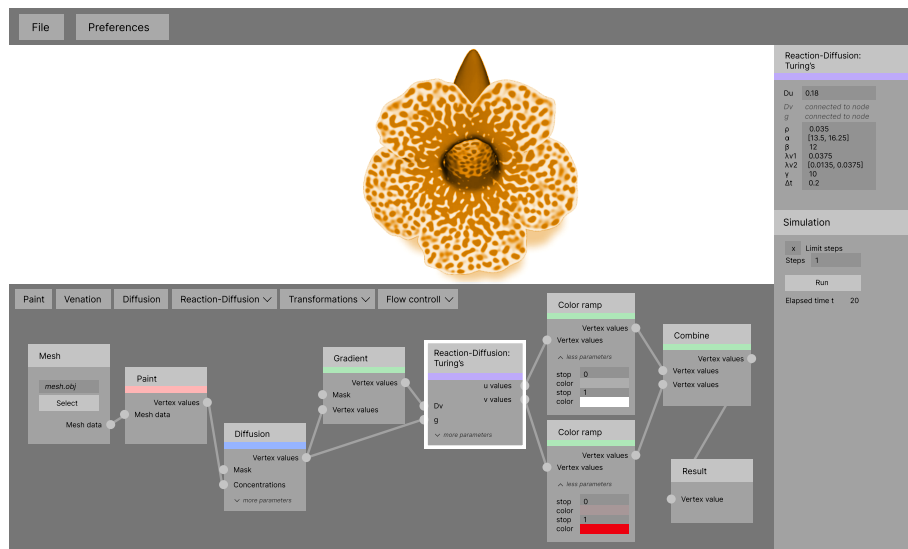


Figure A.7: UI prototype: *Kohleria* case study. Reaction-diffusion node at simulation time $t = 20$. Author's archive.

A. INITIAL DESIGN PROTOTYPE

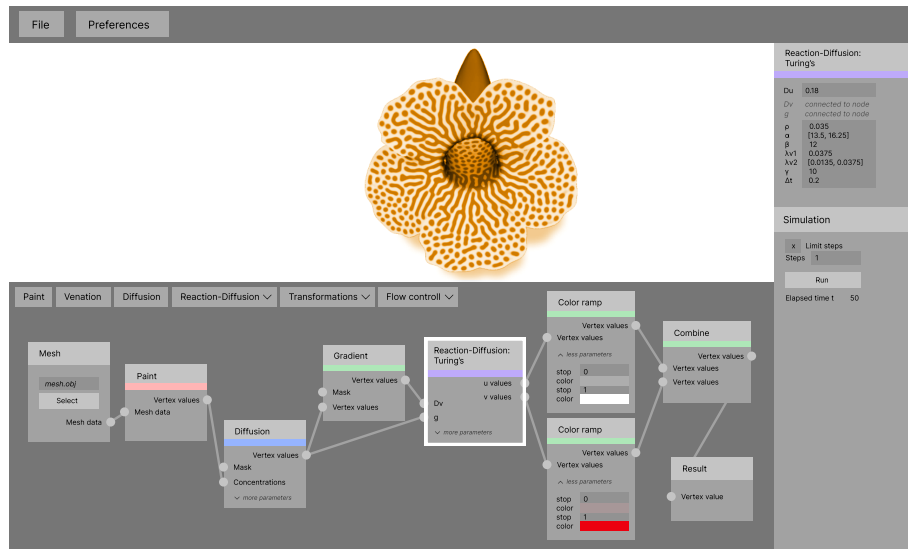


Figure A.8: UI prototype: *Kohleria* case study. Reaction-diffusion node at simulation time $t = 50$. Author's archive.

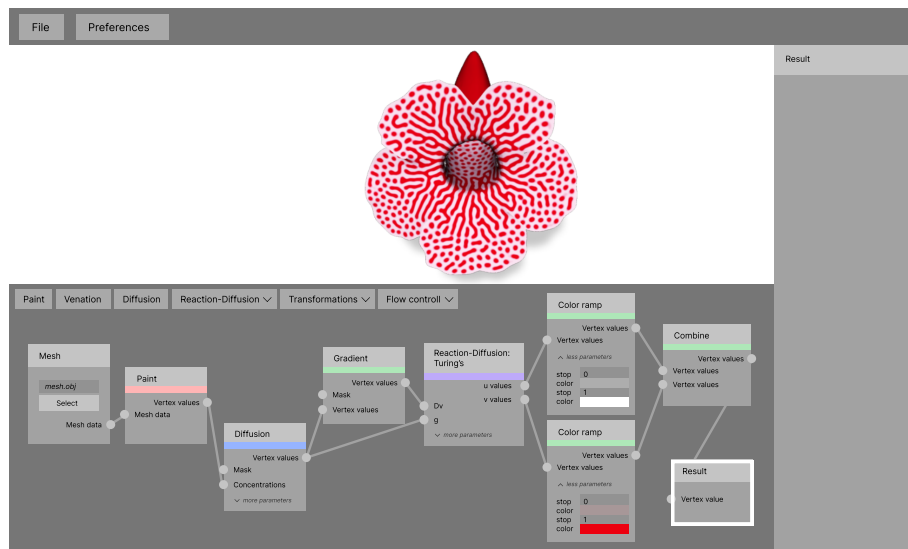


Figure A.9: UI prototype: *Kohleria* case study. Result node. Author's archive.

A. INITIAL DESIGN PROTOTYPE

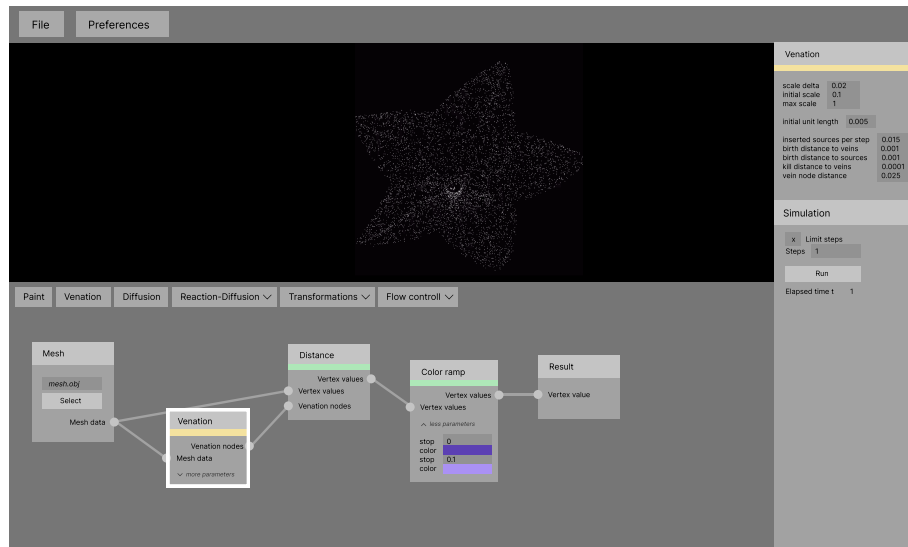


Figure A.10: UI prototype: *Platycodon grandiflorus* case study. Venation node at simulation time $t = 1$. Author's archive.

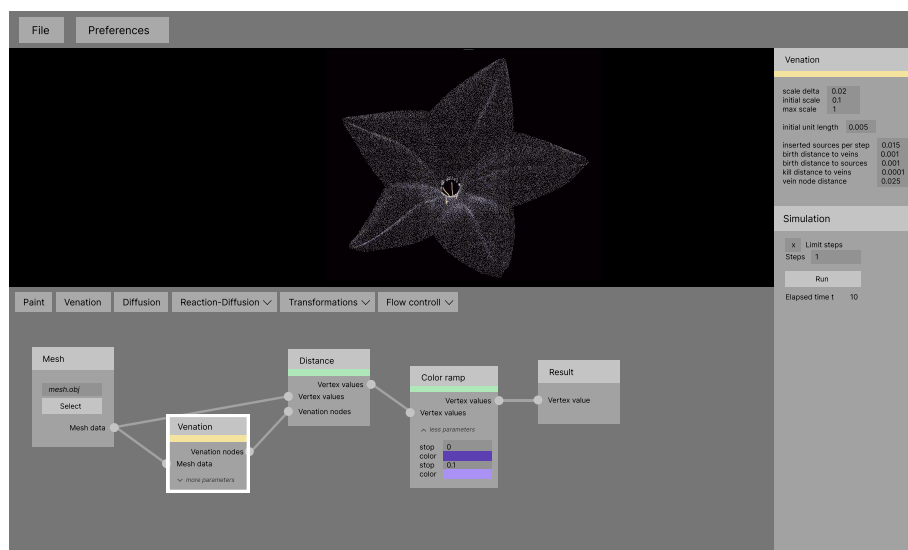


Figure A.11: UI prototype: *Platycodon grandiflorus* case study. Venation node at simulation time $t = 10$. Author's archive.

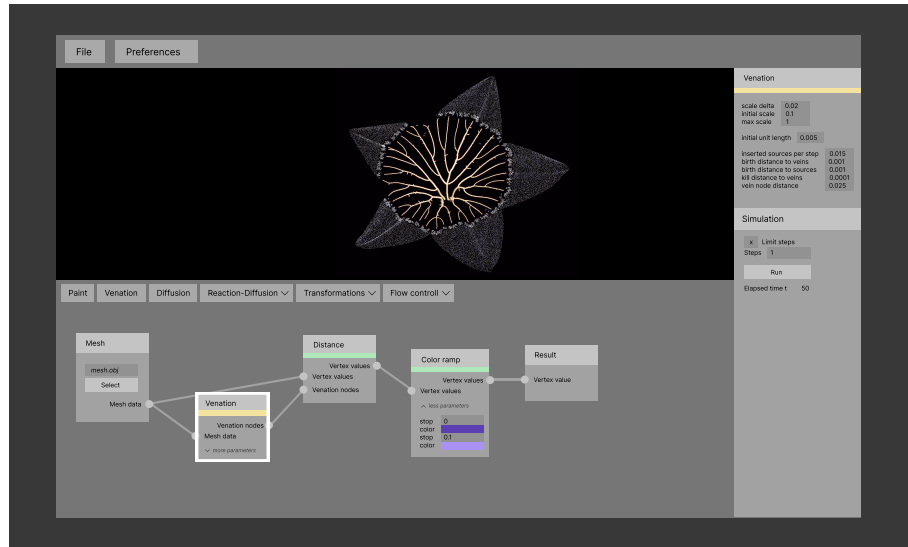


Figure A.12: UI prototype: *Platycodon grandiflorus* case study. Venation node at simulation time $t = 50$. Author's archive.

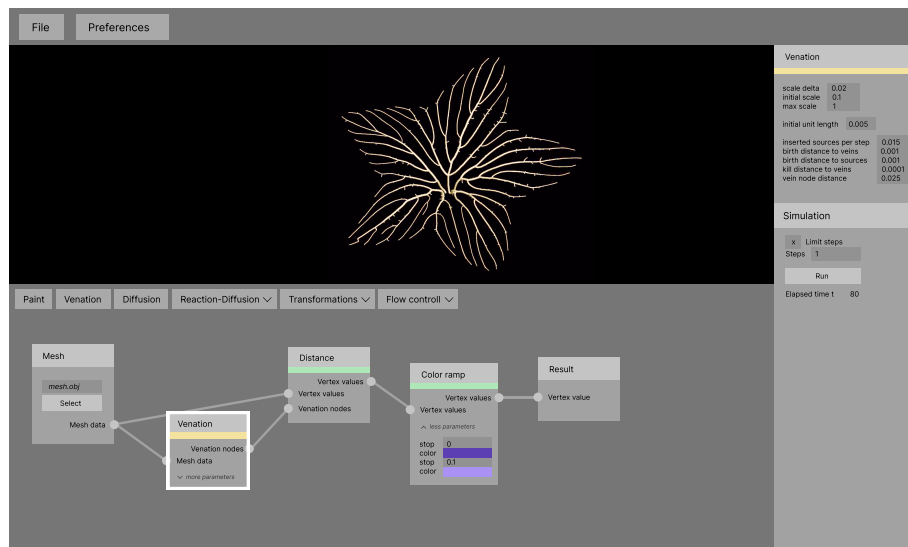


Figure A.13: UI prototype: *Platycodon grandiflorus* case study. Venation node at simulation time $t = 80$. Author's archive.

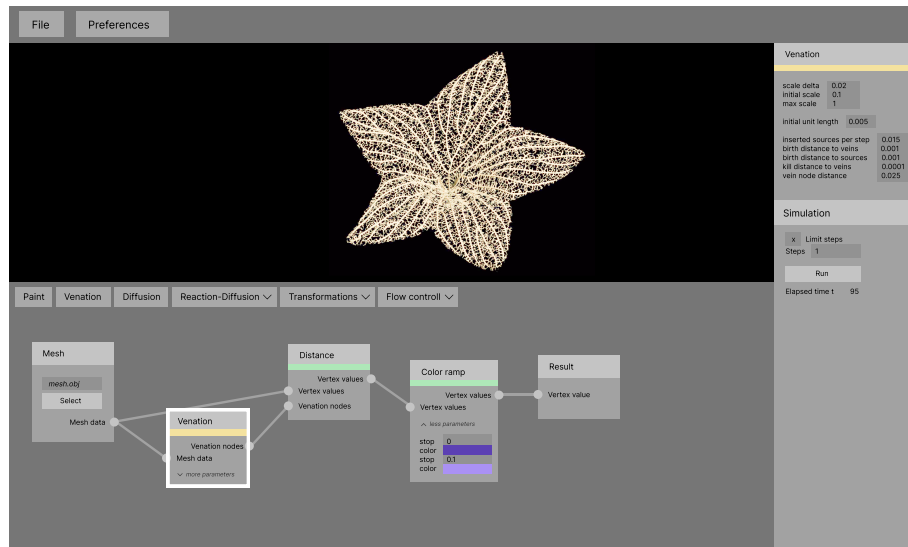


Figure A.14: UI prototype: *Platycodon grandiflorus* case study. Venation node at simulation time $t = 95$. Author's archive.

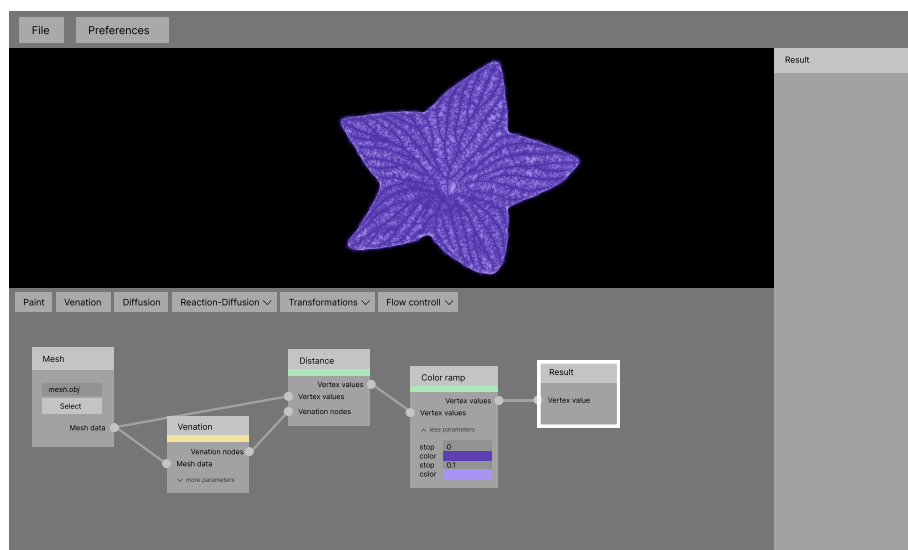


Figure A.15: UI prototype: *Platycodon grandiflorus* case study. Result node. Author's archive.

B Node pipelines of case study patterns

This Appendix provides additional information about the reproduced and approximated pigmentation patterns presented in Chapter 6. Figures B.1–B.6 show the node pipelines generating the patterns, and their captions specify the parameters of the involved simulations. Figures B.4 and B.6 use a tentative Multiply node, which takes two vectors of vertex values and multiplies them component-wise.

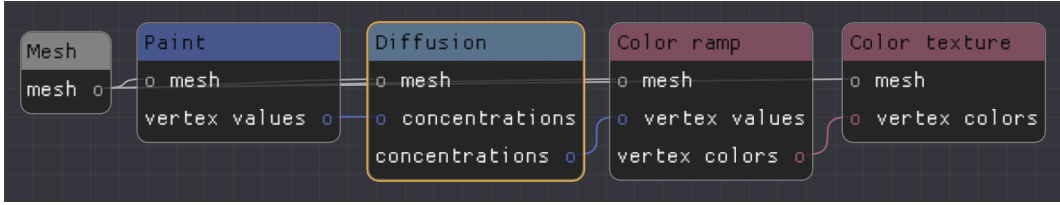


Figure B.1: Node pipeline: *Primula* $\times$ *polyantha* and *Arctostaphylos uva-ursi* (Fig. 6.2). Diffusion uses $spread = 0.1$.

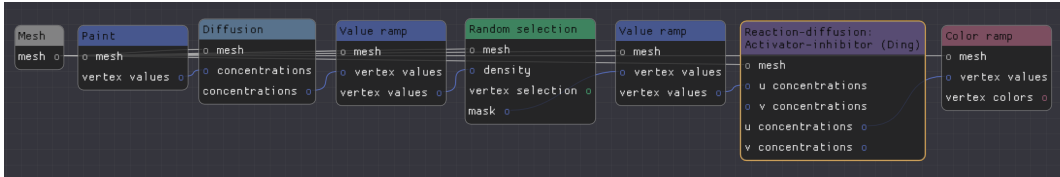


Figure B.2: Node pipeline: *Digitalis purpurea* (Fig. 6.3). Diffusion uses $spread = 0.31$, Random Selection picks 66 vertices, and Reaction-Diffusion uses Activator-Inhibitor model according to Ding et al. [31] with $gA = 0.08$, $A0 = 0.01$, $gI = 0.12$, $I0 = 0.000001$, $k = 0.001$, $uA = 0.03$, $uI = 0.03$, $Da = 0.01$, $Di = 0.5$.

B. NODE PIPELINES OF CASE STUDY PATTERNS

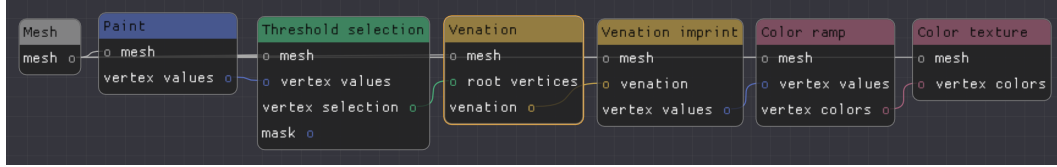


Figure B.3: Node pipeline *Platycodon grandiflorus* (Fig. 6.4). Venation is generated with 1 iteration per growth step, 0.4 sources per triangle, birth distance to veins = 1, birth distance to sources = 1, kill distance to veins = 1, vein segment length = 0.8, thickness of newest vein = 0.001, vein thickness build-up = 3.

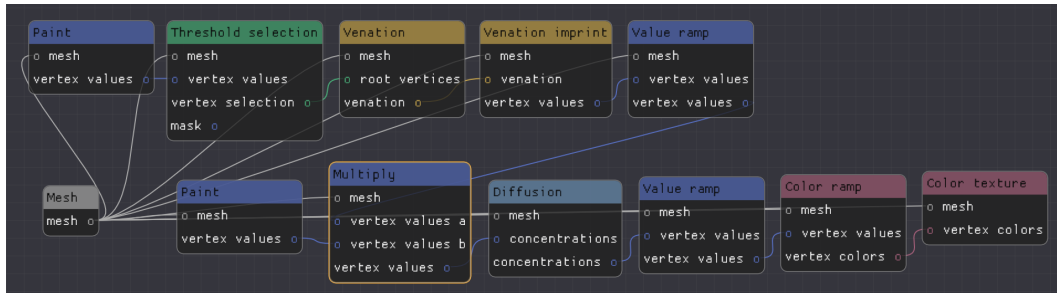


Figure B.4: Node pipeline: *Dianthus barbatus* (Fig. 6.5). The Multiply node takes input from the Value Ramp node in the upper row and the Paint node in the lower row. Venation is generated with 1 iteration per growth step, 6.3 sources per triangle, birth distance to veins = 0.6, birth distance to sources = 0.6, kill distance to veins = 0.6, segment length = 0.6, thickness of newest = 0.1, thickness build-up = 3 Diffusion uses spread = 0.001.

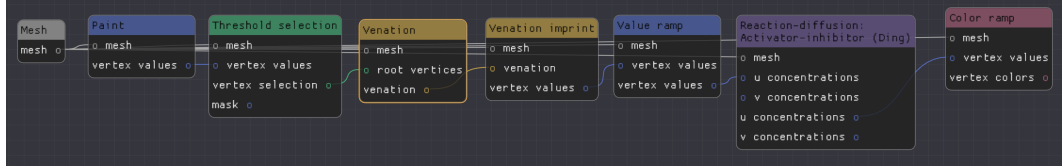


Figure B.5: Node pipeline: *Phalaenopsis Chian Xen Leopard Stripes* (Fig. 6.6). Venation is generated with 1 iteration per growth step, 0.3 sources per triangle, birth distance to veins = 2, birth distance to sources = 2, kill distance to veins = 2, segment length = 0.6, thickness of newest = 0.001, thickness build-up = 3. Reaction-Diffusion uses Activator-Inhibitor model according to Ding et al. [31] with $gA = 0.08$, $A0 = 0.01$, $gI = 0.12$, $I0 = 0.000001$, $k = 0.001$, $uA = 0.03$, $uI = 0.03$, $Da = 0.01$, $Di = 0.5$.

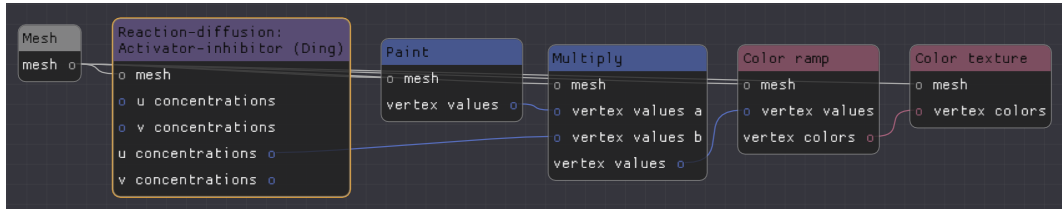


Figure B.6: Node pipeline: *Mimulus guttatus* (Fig. 6.7). Reaction-Diffusion uses Activator-Inhibitor model according to Ding et al. [31] with $gA = 0.08$, $A0 = 0.01$, $gI = 0.12$, $I0 = 0.000001$, $k = 0.001$, $uA = 0.03$, $uI = 0.03$, $Da = 0.01$, $Di = 0.5$.

C External attachments

The thesis comes with three additional external attachments:

- A set of 2D textures generated by the case-study node pipelines.
- The source code of the newly developed application.
- An executable Windows binary of the application.

The application is controlled by a mouse. The mesh can be rotated by holding the wheel button and zoomed by scrolling. The node canvas can also be zoomed by scrolling and moved by dragging with the right mouse button. Right-clicking a hovered node or link displays its context menu. Any parameters overflowing the sidebar area can be scrolled into view.