Models of Morphogenesis and Generative Art

Matthew McConnell - 30094710

University of Calgary, Fall 2024

1 Introduction

Morphogenesis refers to biological or chemical processes that cause an organism to develop its shape. Examples of this can include leaf venation, coral growth, and many other observed patterns in our natural world. Morphogenesis has been used extensively in many computational settings, including those dealing with *generative art* [1]. We explore two models of morphogenesis: Reaction-Diffusion [7] and a Fast Simulation of Laplacian growth [4], with a specific aim of tweaking these models towards a generative art lens. This project implements and analyzes computationally-based models of *morphogenesis* as a standalone digital system that can generate real time patterns. Underlying mathematical and algorithmic concepts are explored in great detail, as well as a description of underlying implementation and UI controls that govern our system. We explore and analyze the various changes in outputs that small tweaks to various parameters can cause on the systems (**seen in a supplemental appendix at the end of this report** (AppendixA)). This project is not inherently a presentation of novel "research", as such, it is structured in a technical-report manner, rather than a traditional research paper.

1.1 Generative Art and Morphogenesis

The definition of *art* is fluid and easily debated, as it encompasses an extensive range of mediums and forms, serving as means for individual human expression. Defining *generative* art can have similar complications, however, for purposes of this project we will use Galanter's definition [1]. "Generative art refers to any art practice in which the artist uses a system, such as a set of natural language rules, a computer program, a machine or other procedural invention, that is set into motion with some degree of autonomy, thereby contributing to or resulting in a completed work of art" [1]. The key idea that separates *generative art* from traditional art is the construction or use of some type of tool that the artist does not control in it's entirety. Typical generative art communities include computer/electronic music, computer graphics and animation, glitch art/live coding, industrial design and architecture and artificial intelligence [1]. A common aspect of generative art is its frequent use of algorithms and computational models that simulate natural phenomena. These computationally focused methods often produce organic, complex, dynamic and "aesthetic" patterns, making *morphogenesis* a compelling underlying framework for digital art.

Morphogenesis has long inspired computational approaches to creativity. As early as the **1950s**, people (notably **Alan Turing**) explored the use of underlying chemical and biological systems [8] to model some form of computational processes. Turing introduced ideas of chemical basis for morphogenesis, and pioneered the first *reaction diffusion* systems [8], with some patterns being dubbed "Turing patterns". These biological and chemical inspirations extend into modern computer graphics-based disciplines, where techniques such as fractals, recursion, and other rule-based methods mimic the principles seen in morphogenesis. Models of morphogenesis allow us to connect science and art in a mathematically grounded and organic way, allowing artists to "simulate" natural growth and self-organization. This connection gives artists the tools needed to explore randomness, order, chaos, emergence and growth, all themes that are commonly found in

natural systems. Generative art has and will continue to evolve, leveraging new technology and mathematical concepts to extend the limits of computational creativity.

2 Mathematics and Algorithms

2.1 Reaction-Diffusion - Gray Scott Model

In the *Gray Scott Model* [7] [3], the simulation handles two virtual chemicals (A and B), reacting and diffusing on a two-dimensional grid. Chemical **A** is added at a given *feed* rate and chemical **B** is removed at a given *kill* rate. The reaction between the two is governed by the simple idea that two **Bs** convert an **A** into a **B**, similar to **B** reproducing, using **A** as food. The chemicals also *diffuse* in such a way that uneven concentrations spread out across the grid. Importantly, we set chemical **A** to diffuse faster than **B**. These simple chemical reaction-based rules are approximated by using two numbers at each cell in our two-dimensional grid, one for each of the local concentrations of **A** and **B** respectively. It's worth noting that the individual particles themselves are not simulated [4], as we might see in a more *kinetics* based approach. When a grid of thousands of cells are simulated in this manner, large scale patterns can emerge.

There are two primary equations which govern how the grid is updated over time, concerned with the concentrations of both **A** and **B** at each grid cell.

$$A' = A + (D_A \nabla^2 A - AB^2 + f(1 - A))\Delta t$$
$$B' = B + (D_B \nabla^2 B + AB^2 - (k + f)B)\Delta t$$

A' and B' represent our new values after a step of the simulation, with A and B representing our previous values. D_A and D_B represent our *diffusion* rates for both **A** and **B** and can easily be varied. Next, we have our two-dimensional *laplacian functions*:

$$\nabla^2 A$$
$$\nabla^2 B$$

which provide the difference between the average of nearby grid cells and the current cell. These *laplacian* functions allow us to simulate diffusion, allowing both **A** and **B** to become more like their neighbors. We also have our reaction term:

$$AB^2$$

dictating the chance that one **A** and two **Bs** will come together. We know that **A** is converted to **B**, as such this amount is subtracted from **A** and added to **B**. Finally, we have both our *feed* and *kill* terms. The *feed* term is quite simple, we add chemical **A** at a given rate f, scaled by $(1 - A)$ such that **A** can never exceed 1. On the other hand, the *kill* term:

$$-(k + f)B$$

is subtracted from our second equation to remove **B**. The *kill* term is both scaled by **B** (so that it doesn't provide negative values) and has f added so that the resulting *kill* rate is never less than the feed rate.

2.1.1 The Mysterious Laplacian

The resulting behavior of this simulation is heavily dependent on our two *laplacian* functions, which govern how cells "convert" to be more like their neighbors. There are many ways for performing a *laplacian*, however, for our purposes, we consider a 3x3 convolution [7] with a center weight of -1, adjacent neighbor weights of 0.2 and diagonal neighbors of 0.05:

$$\begin{matrix} 0.05 & 0.2 & 0.05 \\ 0.2 & -1 & 0.2 \\ 0.05 & 0.2 & 0.05 \end{matrix}$$

with these *weighted sums* of chemical **A** and **B** being computed for all **8** neighbors.

2.2 Laplacian Growth - Dielectric Breakdown

Dielectric Breakdown occurs when an electrically insulating material (dielectric) is subjected to a high enough voltage, and suddenly becomes a conductor, allowing a vast amount of current to flow through it. We can apply this natural phenomena in a computation simulation, by using electrical potentials on a regular two-dimensional grid to create branching patterns, resembline those of traditional *Diffusion Limited Aggregation* [2].

2.2.1 Dielectric Breakdown

This approach is similar to *dielectric breakdown*, attempting to simulate the branching patterns that occur in electrical discharge [4]. We can generalize a *dielectric breakdown* simulation in three steps; 1. Calculate the electric potential ϕ on a two-dimensional grid according to some boundary condition, 2. Select a grid cell as a "growth site" according to this ϕ and 3. Add the new growth site to the boundary condition [4]. One iteration of these three steps attaches a particle to our structure, and when this process repeats over large number of iterations, a growth structure or aggregate is obtained.

2.2.2 Fast Simulation of Laplacian Growth

Kim et. al [4] proposed a faster and more memory efficient *laplacian growth* simulation than traditional *dielectric breakdown* models. Computing **step 1** (calculating ϕ) dictates a disproportional amount of the computation. In order to circumvent this step, we can constrain the interior of the aggregate to be a perfect insulator. *Dielectric breakdown* occurs when an insulator is converted into a conductor, however, this "fast" approach simulates the opposite case, where a conductor converts into an insulator.

The *fast simulation of laplacian growth* works as follows: We initialize the algorithm, by first inserting a point charge somewhere in our grid (typically the center). Next, we locate the candidate sites around the charge, and calculate the potential at each candidate site:

$$\phi_i = \sum_{j=0}^{n} (1 - \frac{R_1}{r_{i,j}})$$

where j is the index of a point charge, $r_{i,j}$ is the distance between grid cell i and point charge j and n is the total number of point charges. Then, for each iteration of our algorithm, we will randomly select a new growth site p_i according to:

$$p_i = \frac{(\Phi_i)^\eta}{\sum_{j=1}^{n} (\Phi_j)^\eta}$$

where

$$\Phi_i = \frac{\phi_i - \phi_{min}}{\phi_{max} - \phi_{min}}$$

is our normalized potential values (according to the current minimum and maximum potentials). Then, a new point charge is added at the growth site, all candidate sites have their potentials updated:

$$\phi_i^{t+1} = \phi_i^t + (1 - \frac{R_1}{r_{i,t+1}})$$

where ϕ_i^{t+1} corresponds to the potential at position i at timestep $t+1$ and ϕ_i^t is the potential at the same point at the previous timestep. Finally, we calculate the potential at the new candidate sites using the same method as in our initialization. Repeating these iterative steps allows complex branching structures to emerge.

3 Implementation Details

3.1 Reaction Diffusion

This simulation takes place on a two-dimensional grid of size 1024 x 1024, mapped to a texture which is pasted on a plane. Both the underlying simulation logic determining the chemical growth and the rendering is controlled via a *compute shader* written in **HLSL**. Handling various UI adjustments and speed of the simulation is implemented as a C# script. **4 iterations** of the algorithm are computed before the display is updated, helping to optimize real-time performance. This simulation is optimized at a extremely high FPS (>> 60) on a 1024 x 1024 grid. Sebastian lague's simulation of Reaction-Diffusion [5] was incredibly helpful during the development of this model, particularly during the implementation of the compute shader.

3.1.1 Main Algorithm

The algorithm is first initialized with a small amount of chemical **b**, (setting grid $G_{curr}[i][j].y = 1$) in various shapes such as a triangle or three diagonal squares (see initial concentration maps). Then, for each iteration of the algorithm, we will get the current values of **A** and **B**, calculate their respective *laplacians*, apply our governing equations to obtain **A'** and **B'**, and update the concentrations, storing this updated state in a new grid, G_{new}.

// Initial Concentration of Chemical B
$G[i, j] \leftarrow InitializeGrid()$;
foreach *iteration* **do**
  $a, b \leftarrow G[i, j]$;
  $laplacianA, laplacianB \leftarrow CalculateLaplacians()$;
  $reaction \leftarrow a * b * b$;
  $A_{diff} \leftarrow (D_a * laplacian_A - reaction + feed * (1 - a))$;
  $B_{diff} \leftarrow (D_b * laplacian_B + reaction - (kill + feed) * b)$;
  $G_{new}[i, j] \leftarrow (a + A_{diff}, b + B_{diff})$;
  // Render colors
  $G_{display} \leftarrow UpdateColor(colorMode, G_{curr}, G_{new})$;
end

3.1.2 Colors

There are **6** distinct color modes. First, the default color mode present in a **HLSL** shader:

$$G_{display}[i, j] = G_{curr}[i, j]$$

The remaining color modes use a $float4$ containing various color properties:

$$Color_A = G_{next}[i, j].x$$

$$Color_B = G_{next}[i, j].y$$

$$Color_W = G_{next}[i, j].z$$

$$Color_Z = G_{next}[i, j].w$$

Using these various color properties inherent to the grid, we can obtain our second color, black and white:

$$G_{display}[i,j] = Color_A - Color_B$$

Our third color mode: *power function* is a bit more advanced, utilizing a pool of colors to generate a scalar value, and then applying a non-linear transformation (*pow*) to adjust the contrast. Additionally, we add an offset to ensure the range stays positive, invert the "gradient of colors" and introduce additional scaling factors for each color.

$$G_{display}[i,j] = \left(1 - |1 - (v + 1 - \max(Color_W, Color_Z) \cdot 350)|^{0.6}\right) + \begin{pmatrix} Color_W \\ 0 \\ 0 \\ 0 \end{pmatrix} \cdot 400$$

Our fourth color mode: *gradient power function* generates a particular color by blending values into a *float4* representing an **RGBA** value, swapping various channels and applying a power function. We initialize a color vector, where each **RGBA** channel is individually set and scaled,

$$Color = \begin{pmatrix} Color_Z \cdot 350 \\ Color_W \cdot 250 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} Color_W \\ 0 \\ 0 \\ 0 \end{pmatrix} \cdot 400$$

and then reorder the color channels and apply a power function to non-linearize it:

$$Color = Color_{GBRA}, Color.g = Color.g^2 \cdot 6, Color = min(1, max(0, Color))$$

and finally apply it to our display grid:

$$G_{display}[i,j] = Color$$

Our fifth color mode attempts to represent a "fiery glow" in which we first obtain the intensity:

$$I = Color_A - Color_b$$

add a glow by saturation

$$G = saturate(Color_W \cdot 6.0 + Color_Z \cdot 3.0)$$

and then map this intensity to specific **RGB** channels:

$$Color = \begin{pmatrix} saturate(I \cdot 20 + G \cdot 5 \\ saturate(I \cdot 0.8 + G \cdot 0.5 \\ saturate(G \cdot 0.25 \\ 1.0 \end{pmatrix}$$

Our final color mode attempts to represent a cool-blue or icy setting, in which we again obtain the intensity

$$I = Color_A - Color_B$$

add a glow by saturation

$$G = saturate(Color_W \cdot 100.0 + Color_Z \cdot 100.0)$$

and map them to specific **RGB** channels:

$$Color = \begin{pmatrix} saturate(I \cdot 0.5 + G \cdot 0.9 \\ saturate(I \cdot 0.3 + G \cdot 0.6 \\ saturate(G \cdot 5 \\ 1.0 \end{pmatrix}$$

However, before mapping to our display grid, we apply a power function to create an attempt at sharp contrast for the icy look:

$$G_{display}[i,j] = \begin{pmatrix} R \\ B \\ G \\ 2.0 \end{pmatrix}^{(1.0,0.2,0,1.0)}$$

Some of these colors were taken from **Sebastian Lague's** reaction-diffusion simulation [5], others were generated by playing around with various **HLSL** functions and properties.

3.1.3 Initial Concentration Maps

Pre-built *initial concentration maps* are included, which can drastically affect the growth of the simulation. These *initial concentration maps* are loaded in during the initialization of the algorithm, before iterating. Instead of inserting a small amount of chemical **B** at the center, we can vary the initial concentration to take the form of particular shapes:

- **single square** - Add a small amount of **B** as a square at the center of the grid.
- **single circle** - Add a small amount of **B** as a circle at the center of the grid.
- **3 diagonal squares** - Add a small amount of **B** as a square, and then place 3 in a diagonal pattern across the grid.
- **line** - Add a large amount of **B** in one direction, and a small amount in another, creating a vertical line or column.
- **triangle** - Add a small amount of **B** as a triangle at the center of the grid.
- **15 random points** - Add a small amount of **B** at 15 random points across the grid.

The mathematical concepts for these *initial concentration maps* are rooted in basic geometry, and thus omitted from further explanation.

3.1.4 Extension - Directional Bias

To add additional complexity into this model, a *directional bias* extension was implemented. The basic idea is to affect the "overall" shape of the simulation to grow along particular axes or directions. This is achieved by varying the *kill rate* across the grid, such that the chemicals tend to grow along these invisible "axes", creating an invisible pattern governing our chemical pattern. These "directions" or "axes" include;

- **linear strips** (Horizontal) - Vary the kill rate based on a particular width of segment, promoting growth in horizontal strips.
- **linear strips** (Vertical) - Vary the kill rate based on a particular width of segment, promoting growth in vertical strips.
- **linear strips** (Diagonal) - Vary the kill rate based on a particular width of segment, promoting growth in diagonal strips.
- **radial** - Vary the kill rate based on the current distance from the center, normalized over the maximum distance.
- **sinusoidal** - Vary the kill rate based on a sinusoidal frequency, divided by a particular wavelength.

The *strength* of the bias, and the *number of segments* (when applicable) are controllable via UI elements, and can easily be adjusted to provide differing patterns.

3.2 Laplacian Growth

This simulation takes place on a two-dimensional grid of size 400 x 400. The underlying simulation logic depending on the mathematics mentioned above is controlled via C# scripts. The rendering (including color modes) are implemented via a *compute shader* in **HLSL**. **5** iterations of the algorithm are computed on the C# side, after which, our two dimensional aggregate map is converted into a one dimensional aggregate buffer and then fed to the *compute shader* for display. This provides a decent balance between performance and real-time growth, allowing larger branching structures to emerge. This simulation is optimized to run in real-time at roughly **30** FPS, on a 400 x 400 grid. There is a significant decrease in performance as the aggregate becomes large (due to the sheer number of candidate sites to consider), as such, a maximum iteration count of **10000** iterations is enforced.

3.2.1 Main Algorithm

The algorithm is initialized by 1. inserting a point charge, 2. locating the candidate sites around the charge and 3. calculating the potential at each candidate site. Then, for each iteration of the algorithm we 1. randomly select a growth site, 2. add a new point charge at this growth site, 3. update the potential at all candidate sites, 4. add the new candidate sites surrounding the growth site and 5. calculate the potential at each of the new candidate sites.

```
// Initialize Data Structures
M = int[400, 400];
C = map[Vector2 : float];
P = List < Vector2 > ();
M[initialSeeds] = 1;
foreach i, j ∈ M do
    C_new = map[Vector2 : float];
    C_new ← AddCandidateSites(i, j, C);
    CalculatePotentials(C_new.keys);
end
// Iterate Algorithm
foreach iteration do
    G ← SelectGrowthSite();
    M[G.x, G.y] = 1;
    P.Add(Vector2(G.x, G.y));
    UpdatePotentials(G);
    C_new = map[Vector2 : float];
    C_new ← AddCandidateSites(G);
    CalculatePotentials(C_new.keys);
end
```

Note that M is our aggregate map, C are our candidate sites, P is a point charge, C_{new} are new candidate sites and G is the growth site currently under consideration.

3.2.2 Colors

Similar to the reaction diffusion model, we have a few colors modes for our *laplacian growth* simulation. First, we have a simple white on black color mode, where any grid cells with value **1** in our aggregate map M are colored white (black background). Next, a black on white color mode, where any grid cells with value **1** in M are colored black (white background). Finally, we have a gradient color mode, which colors particles according to their distance from the initial seed. If any *initial point charge* map does not contain a single seed, the initial seed (in terms of color)

is set to the center of the grid.

$$Color_{gradient} = lerp(Color_A, Color_B, ||distance||)$$

where $||distance||$ corresponds to the normalized distance between the currently particle and the maximum distance in the grid. Both $Color_A$ and $Color_B$ can be set via UI controls.

3.2.3 Initial Point Charge Maps

Pre-built *initial point charge maps* are included, which can drastically affect the growth of the particles in the simulation. These *initial point charge maps* are loaded in during the initialization of the algorithm, before iterating. Instead of inserting a singular point charge into the center of the grid, we can vary the initial *point charges* to take the form of particular shapes:

- **single point charge** - Add a single point charge at the center of the grid.
- **random single point charge** - Add a single point charge at a random location on the grid.
- **20 random point charges** - Add **20** point charges at random locations on the grid.
- **square** - Add point charges that make up the outline of a square.
- **triangle** - Add point charges that make up the outline of a triangle.
- **circle** - Add point charges that make up the outline of a circle.
- **cross** - Add point charges along the horizontal and vertical axes that make up a cross, or graph-axis shape.

The mathematical concepts for these *initial point charge maps* are rooted in basic geometry, and are located in a separate helper file.

3.2.4 Extension - Growth Towards Mouse

As with the reaction diffusion model, we add some additional complexity into this model in the form of a *Directed Growth* extension. The basic idea is for the algorithm to be more likely to pick particles that are at a shorter distance from the current mouse position. This is achieved by applying a "directional bias" in our *CalculatePotentials()* function, where we first obtain the distance from a current *candidate site* and a *target point* (the mouse position, converted to grid position). We then normalize this distance by dividing out the maximum distance of the grid, and add the inverse of this normalized distance to our ϕ calculation multiplied by some scaling factor k.

$$\phi = \phi + \frac{1}{bias} \cdot k$$

The algorithm still selects the *growth site* as mentioned in the theory sections, to maintain the aggregate branching structure, we are simply encouraging the algorithm to pick values that are closer to a target position by increasing certain ϕ values. This is essential, as we want the algorithm to still select particles that are further away from our target position, but not as frequently as those who closer. The scaling factor k is currently set to **5**, and is not controllable via UI elements. This is intentional, as particular scaling values can drastically effect the growth of the particles, making it incredibly trivial to destroy the aggregate branching patterns. The pattern will "grow towards" the mouse position when the user holds the **left mouse button** down, and is more pronounced the closer the mouse position is to point charges.

4 The System

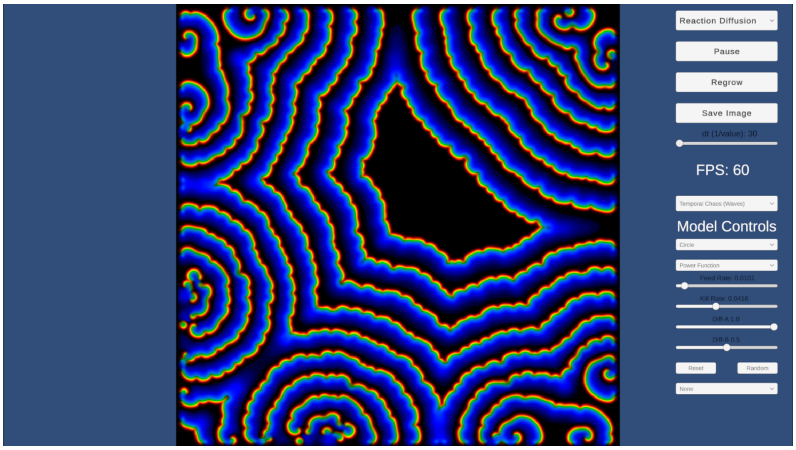
This system has been implemented in the **Unity** game engine, using a combination of **C#** scripts as well as compute shaders written in **HLSL**. It is optimized to run in *real-time*, although the Reaction Diffusion model is more optimized as it is entirely parallelized. UI controls are implemented via **Unity's** built in UI system. Furthermore, a compute shader "helper" library was taken from *Sebastian Lague* [5], to help **HLSL** integrate better with Unity.

4.1 General Simulation Controls

General simulation controls include (from top to bottom):

- Model Selection (Dropdown) - Select Reaction-Diffusion or Laplacian Growth
- Play/Pause (Button) - Start or Stop Simulation
- Regrow (Button) - Regrow pattern, maintaining parameters
- Save Image (Button) - Save image to local directory (found in Unity data files in the `Output Images` directory, see code access)
- dt (Slider) - Change the **dt** of the simulation to speed up or slow down the model.
- Presets (Dropdown) - Select a preset pattern (specific to model)
- Reset (Button) - Reset the model to initial parameters
- Random (Button) - Randomly set parameters

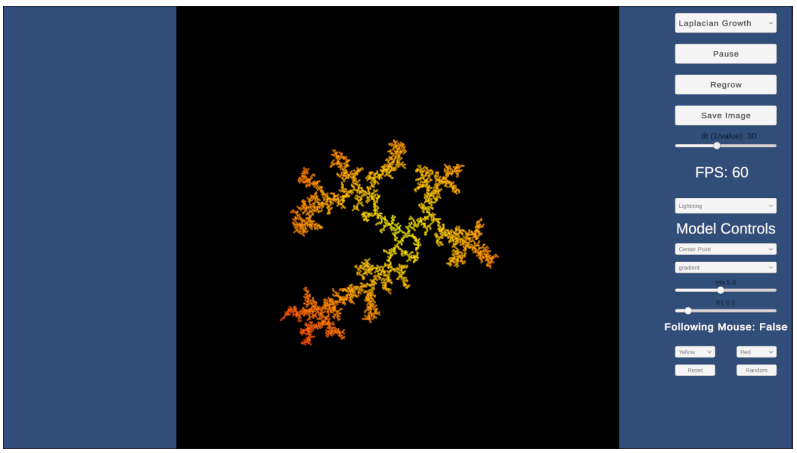
4.1.1 Reaction Diffusion



Reaction Diffusion controls include (from top to bottom):

- Initial Concentration Map (Dropdown) - Select the initial concentration map or shape
- Color Modes (Dropdown) - Select the color mode
- Feed Rate (Slider) - Vary the Feed Rate
- Kill Rate (Slider) - Vary the Kill Rate
- Diff-A (Slider) - Vary diffusion rate A
- Diff-B (Slider) - Vary diffusion rate B
- Directional Bias (Dropdown) - Select the shape of the directional bias (included are sliders to vary number of segments and strength of bias)

4.1.2 Laplacian Growth



Laplacian Growth controls include (from top to bottom):

- Initial Seed Map (Dropdown) - Select the initial seed locations
- Color Modes (Dropdown) - Select the color mode
- eta (slider) - Vary eta
- R1 (slider) - Vary R1
- Following Mouse (text) - Indicates whether or not the particles will cluster towards the mouse (hold down left mouse button to enable)
- Color Picker (Dropdowns) - Pick the two colors which form a gradient (Only available in gradient color mode)

It is rather difficult to highlight the power of the implemented system in a technical report. As such, we encourage readers to play around with the released build; varying parameters, observing animations and creating their own generative art.

5 Conclusions and Future Work

This project aims to explore the connection between *morphogenesis* and *generative art*, utilizing a real-time pattern generation system, rooted in models of morphogenesis. By showcasing computational models such as *Reaction-Diffusion* and *Laplacian growth*, this project highlights the underlying power of morphogenesis to serve as a foundation for generative art. Mathematical concepts and algorithms have been provided in-depth, alongside general system and implementation details. Through parameter adjustments and real-time simulations, we demonstrate the importance of the underlying theory, and showcase the power that minor changes of these parameters have on the output patterns. A release system has been provided, allowing readers to explore these models and parameters in real-time.

As generative art is an inherently broad topic, there are many avenues for future work. Possible extensions for the reaction-diffusion model include *flow fields*; applying an underlying force to the particles on the grid such that they move in addition to the chemical rules, *mouse input*; allowing the user to control how the chemical is added to the simulation and *style maps*; feed and kill rates are no longer static, but instead vary across the grid. New and improved *color modes*, *sizing* of chemicals and more modern *rendering techniques* could also be considered as relatively trivial additions to this model. Possible extensions for the laplacian growth model include *size*; varying the sizes of point charges, *directional bias*; additional movement forces on the point charges enabling them to accumulate in more predictable ways and *shapes*; changing the underlying

shapes of the particles. A further obvious addition would be more careful consideration towards optimization, attempting to parallelize this model in such a way that it can run exclusively on a shader. Finally, other models of patterning such as *traditional diffusion limited aggregation*, *space colonization*, *differential growth* or *slime-mold simulations* could easily be included as additional models in the system, adding depth into the underlying growth patterns that are being represented.

6 Code Access

Access to the Unity Project (including underlying scripts, algorithms and code) is available on github at: `https://github.com/matthew-mcc/Generative_Art_Unity` underneath the "Reaction-Diffusion" project directory. Note, both the laplacian growth and reaction diffusion models are implemented in this directory, that's just the name of the project. The primary files (located under `Assets/Scripts`) include 1. `RD_Simulation_Shader.compute`: the compute shader for reaction diffusion, 2. `RD_Simulation.cs`: it's governing C# script, 3. `Fast_Laplacian.cs`: the primary Laplacian Growth implementation, 4. `Fast_Laplacian_Compute.compute`: the corresponding rendering code and 5. `Simulation_Handler.cs`: which manages all UI elements and various simulation properties.

A release version of the project has been built to run on windows. Attached (in D2L submission and unzipped on github) is a .zip file of a release build. Simply extract the contents to a directory and run the executable. Finally, images are saved to the path:

`%appdata%/LocalLow/Default-Company/Reaction-Diffusion`: in an `Output_Images` directory. Unity's default file storage system creates temporary files in the appdata directory, and any saved images can be found there.

References

[1] Philip Galanter. "Generative art theory". In: *A companion to digital art* (2016), pp. 146–180.

[2] Thomas C Halsey. "Diffusion-limited aggregation: A model for pattern formation". In: *Physics Today* 53.11 (2000), pp. 36–41.

[3] Grzegorz Kepisty. "Generative art: reaction-diffusion and Chladni figures". In: (Sept. 2020).

[4] Theodore Kim et al. "Fast Simulation of Laplacian Growth". In: *IEEE Computer Graphics and Applications* 27.2 (2007), pp. 68–76. DOI: `10.1109/MCG.2007.33`.

[5] Sebastian Lague. *Complex behaviour from simple rules: 3 simulations*. en.

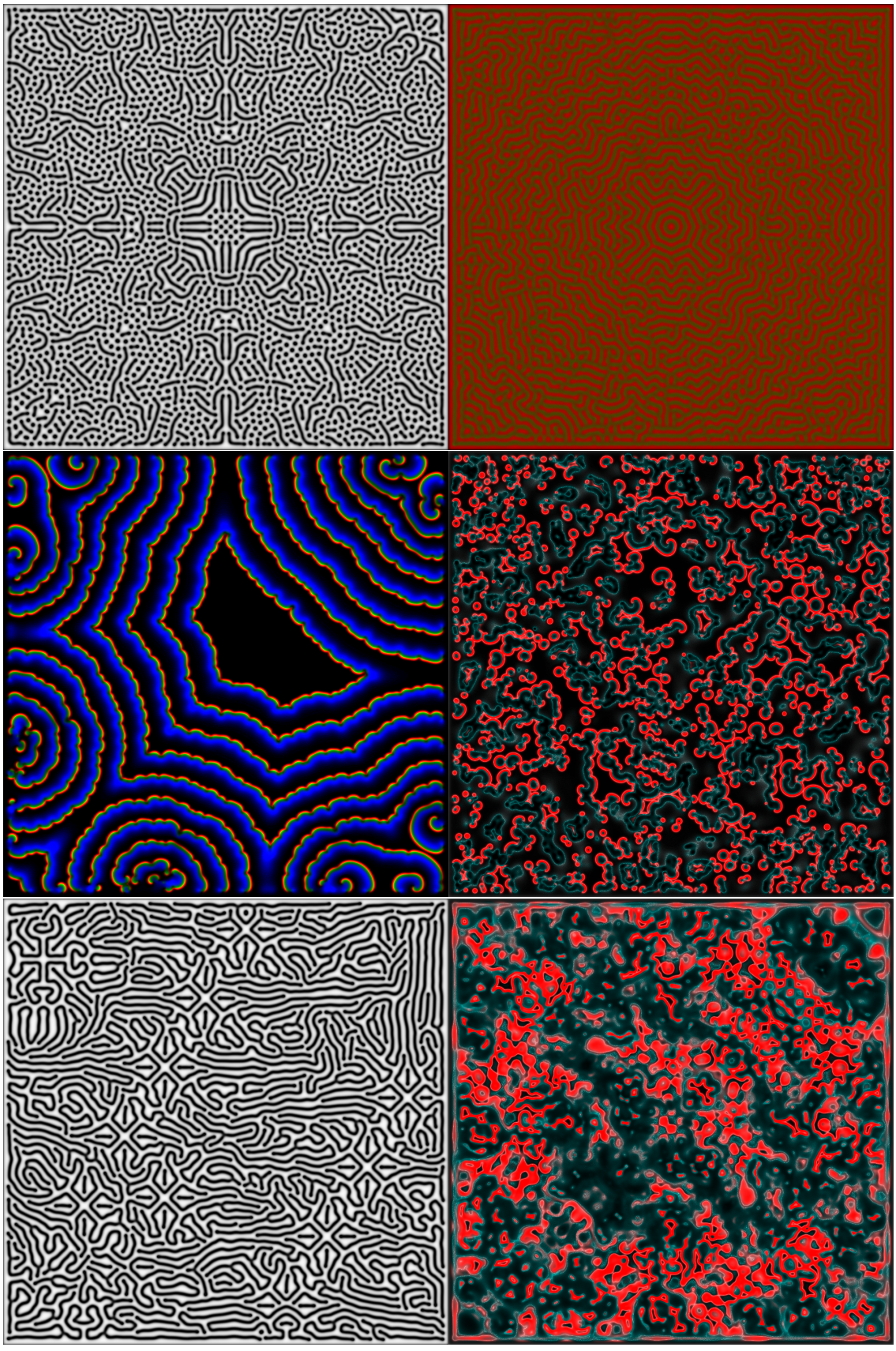
[6] Robert Munafo. *Pearson's Classification (Extended) of Gray-Scott System Parameter Values*. en. `https://mrob.com/pub/comp/xmorphia/pearson-classes.html`. Accessed: 2024-12-14.

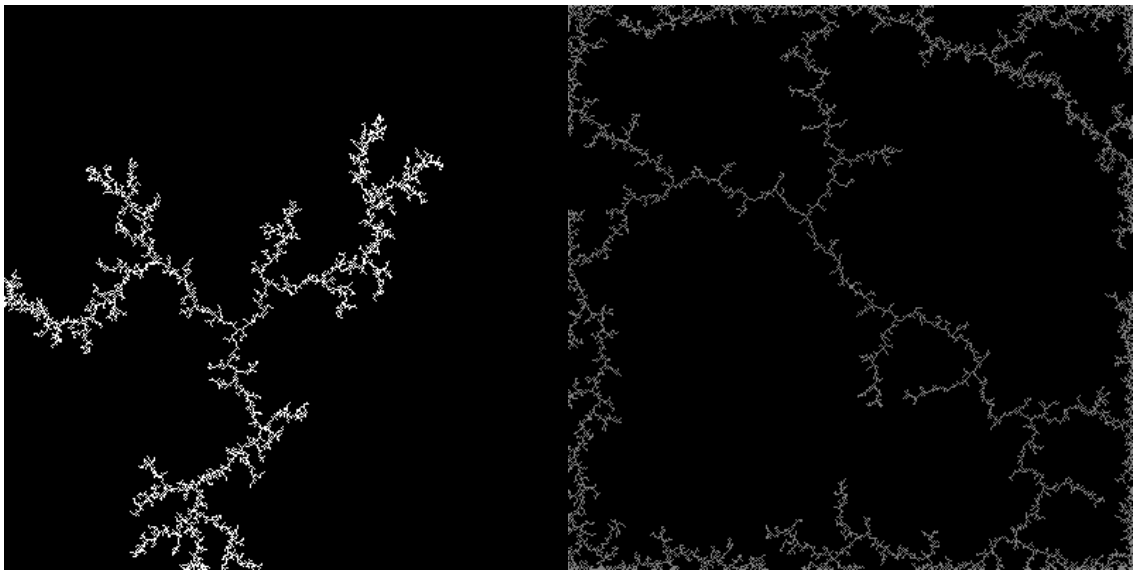
[7] Karl Sims. *Reaction-Diffusion Tutorial*. `https://www.karlsims.com/rd.html`. Accessed: 2024-12-14.

[8] A M Turing. "The chemical basis of morphogenesis". en. In: *Philos. Trans. R. Soc. Lond.* 237.641 (Aug. 1952), pp. 37–72.

7 Art Exhibit

Below is a collection of favorite images that were generated strictly through the use of the presented system.





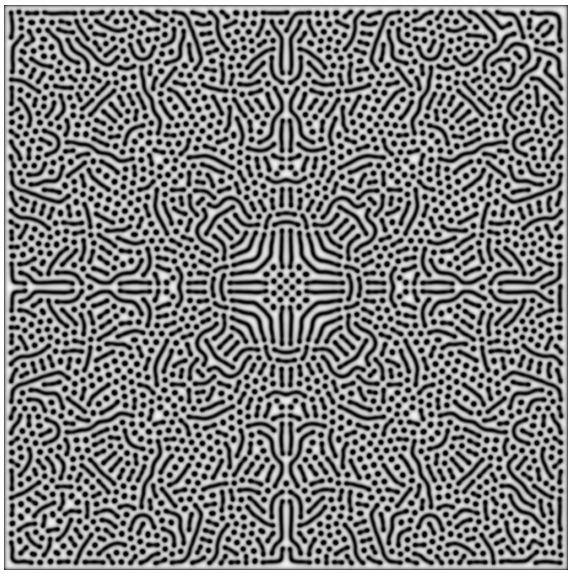
A Parameter Analysis

It's not inherently beneficial to compare our *reaction diffusion* model against our *laplacian growth* model, as they present differing underlying characteristics (growth/diffusion vs. branching patterns). This section will present various "patterns" that can be generated using the two models, that might be commonly found in literature [3], [8], [6]. An image of a pattern and a corresponding set of parameters, extensions, initial maps, etc... will be provided alongside a description of why the parameters are working the way they are to generate a "relatively" known pattern.

We will first start with a subset of patterns generated by the **reaction diffusion** model. In all cases, D_A is set to 1 and D_B is set to 0.5. F represents feed rate, K represents kill rate, C_i represents the initial concentration map and $Color$ represents the color mode. Classification of patterns is provided via the *Pearson-Classification* method [6].

Next, we will examine a subset of patterns generated by the **laplacian growth** model. Unfortunately, there is minimal literature regarding commonly seen aggregate branching patterns. As such, I will provide insight into the specific value of η and the initial seed map C_i, and why the pattern is forming in the manner it does.

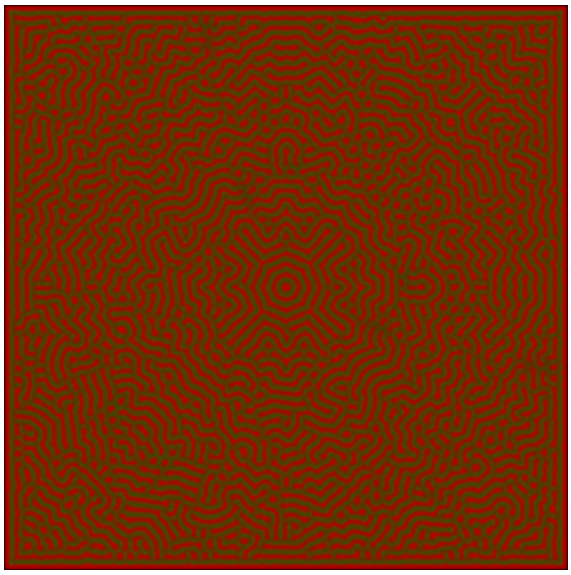
A.1 Mitosis



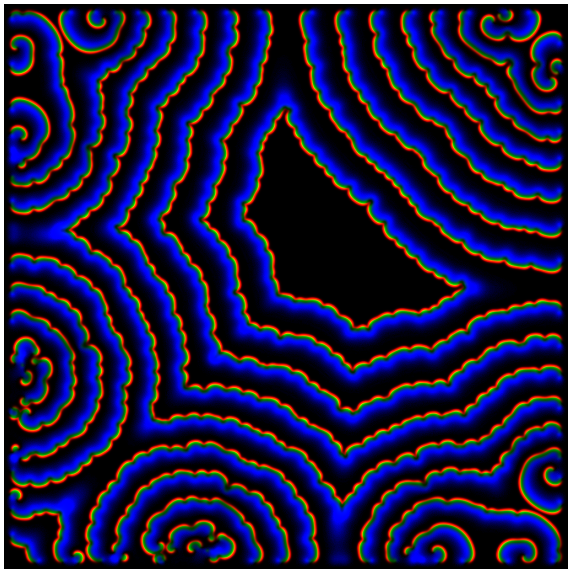
($F = 0.0367$, $K = 0.0620$, C_i = square, $Color$ = black/white). This pattern showcases a pattern of **stable solitons** [6], animating a form of mitosis [7], and is primarily composed of spots spread out across the grid. After the space is filled, these solitons often rearrange into grid-like structures [6] [3], or combine together to create tendrils. This pattern is better visualized via animation, as we can see the "cells" splitting and "growing" in real time. K and F are selected in such a way that a *steady state* is achieved to ensure a balance of chemical reactions such that the concentration of **A** is depleted locally wherever **B** forms, maintaining the spots across the grid. Eventually, all movement stops in this pattern (when the steady-state is reached). These patterns belong to the Pearson-Class **lambda** [6].

A.2 Maze

($F = 0.0220$, $K = 0.0505$, C_i = square, $Color$ = default). This pattern showcases a maze-like structure, with perturbing labyrinths and corridors. Although not fundamentally a "maze" (as there is no connectivity), this presents a visual representation of various corridors and pathways. This pattern is primarily made up of stripes, often branching and create multiple "disjoint" sets [6] to create the illusion of pathways. The pattern grows out in all directions, similar to how a "ring" or "coral" might form, rather than compositing small spots or stripes that turn into the tendril like structures [6]. The simulation "almost" reaches a steady state, however, glitches of "pulses" of Chemical **B** can be observed, in which the reaction is trying to create more of these branches, but unsuccessful due to the selection of K. This is a type **kappa** pattern according to the Pearson-Classification [6] and could also be considered a "Turing pattern" [8].

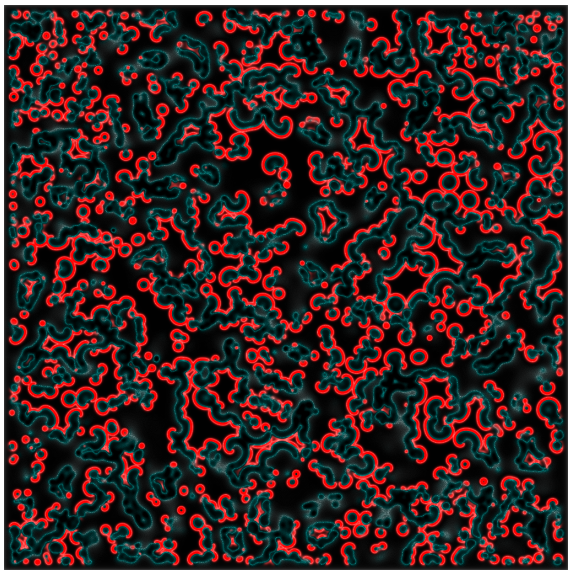


A.3 Waves (Temporal Chaos)



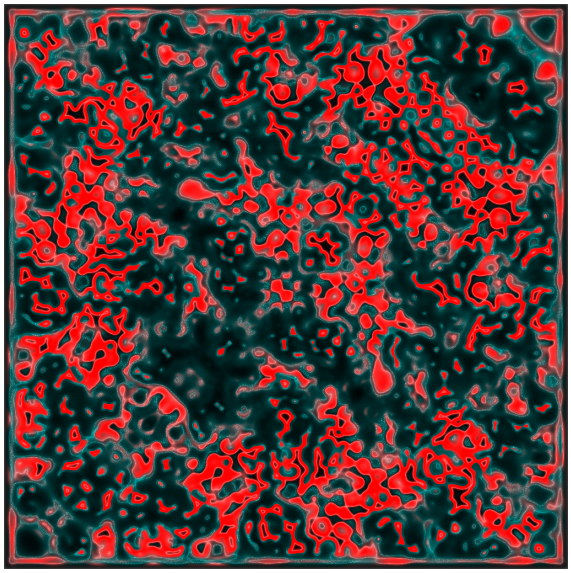
($F = 0.0101$, $K = 0.0416$, C_i = circle, $Color$ = power function). This pattern showcases waves that animate over time, becoming more pronounced as the waves reach the boundaries of the grid. Interestingly enough, even though no explicit boundary conditions were varied, this selection of F and K allow the waves to "bounce back" from the boundaries, in turn creating reactions moving inward (towards the initial concentration) rather than outward. This pattern represents a type of "Spatial-temporal chaos" [6], primarily composed of spots that grow out and connect together to form rings. These "rings" will grow until they contact something else (in our case, the boundary), which they will then "bounce back" towards the center. This is a type **alpha** pattern according to the Pearson-Classification [6].

A.4 Sustained Spirals



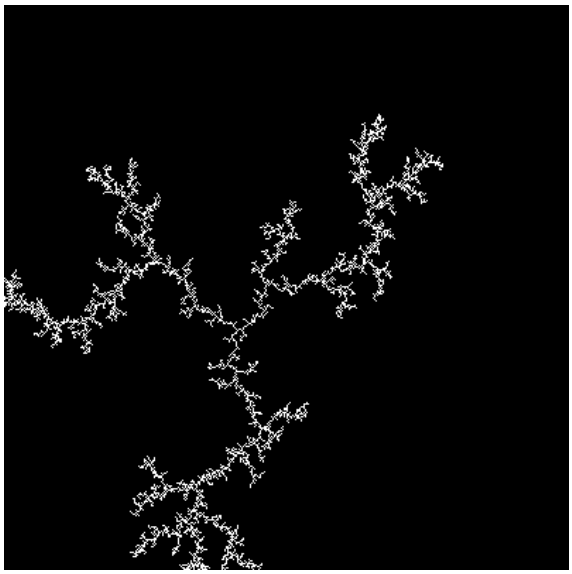
($F = 0.0142$, $K = 0.0474$, C_i = triangle, $Color$ = gradient power function). This pattern showcases a type of "sustained spirals" [6], similar to what might be observed of chemical reactions in a petri dish. The initial seed of the concentrations is essential to the shape of these sustained spirals, and is essential in producing self-sustaining and long-lived spirals. If an unfit initial concentration is selected, the spirals will quickly die out with the pattern quickly becoming uniform with only one chemical present in the simulation. This is a type **xi** pattern according to the Pearson-Classification [6].

A.5 Red Waves on Blue Ocean



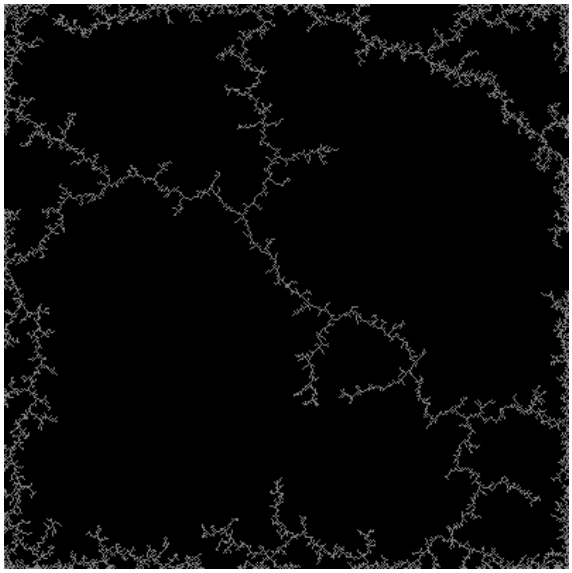
($F = 0.0136$, $K = 0.0390$, C_i = circle, $Color$ = gradient power function). This pattern showcases a topological view of a simulated "ocean", with red values representing the interaction of waves. As in our "waves" pattern (see above), we observe a type of Spatial-temporal chaos, where localized red and blue states differ at different times. We can see periodic pink voids [6] that open up suddenly, and quickly fill in as they are overpowered by our red and blue colors. This is classified as a type **beta** pattern according to the Pearson-Classification [6].

A.6 Lightning



($\eta = 6.8$, $R1 = 4$, C_i = random seed, $Color$ = black/white). This pattern showcases a sort of "lightning" spreading out from our initial seed. The aggregate tends to favor long branches rather than shorter clusters, due to the relatively high η value. This is readily apparent when visualizing the animation, as we can see aggregate particles spread out quickly from the center, and only begin to "cluster" aggressively once a majority of the grid has been filled out. This pattern favors a "depth-first" exploration, moving away from other particles instead of closely clumping together. This pattern is generated only when there is a singular initial seed, as this choice of η aggregates particles away from each other, if we have multiple initial seeds, the lightning structure falls apart.

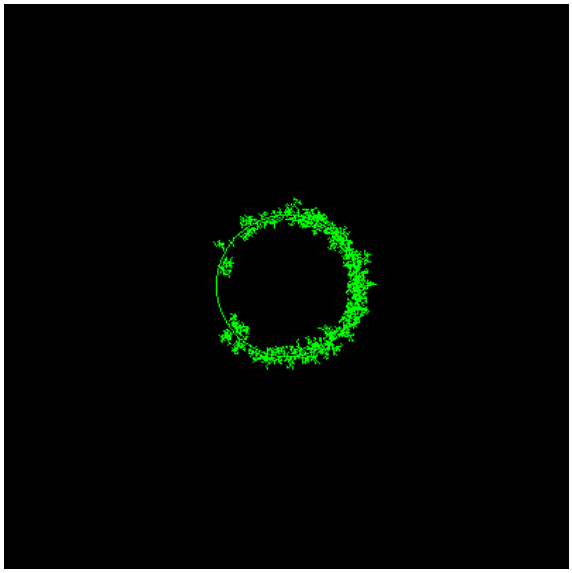
A.7 Cracked Stone



($\eta = 10$, $R1 = 2$, C_i = center point, $Color$ = grey (gradient). This pattern showcases a sort of "fractured" stone, in which the aggregate branches outwards towards the border. This pattern also showcases a type of "depth-first" exploration, with the high η value dictating particles move away from the initial seed before clustering. This pattern is generated from an initial center seed, as to provide equal spacing from the center to the border for each edge of the aggregate. Once

the aggregate runs out of space to grow outwards, it begins clustering along the border. However, it still prefers to aggregate in long branches rather than deep clusters.

A.8 Holiday Wreath



($\eta = 2$, $R1 = 0.5$, C_i = circle, $Color$ = green (gradient). This pattern showcases a sort of "holday wreath", where particles cluster around an initial circular ring of seeds. With a low η value, we can see that the aggregate tends to cluster together more aggressively, rather than forming long and outward branching patterns (as we have seen in our previous patterns). Letting this pattern run for higher iterations showcases the destruction of branching patterns that can emerge when there is a large amount of relatively centralized initial seed, particularly in tandem with a low η value.