Slime Mold Particle Simulation on UE4

Issei Mori
Computer Science and Engineering
University of California, Santa Cruz
Santa Cruz, California, USA
imori@ucsc.edu

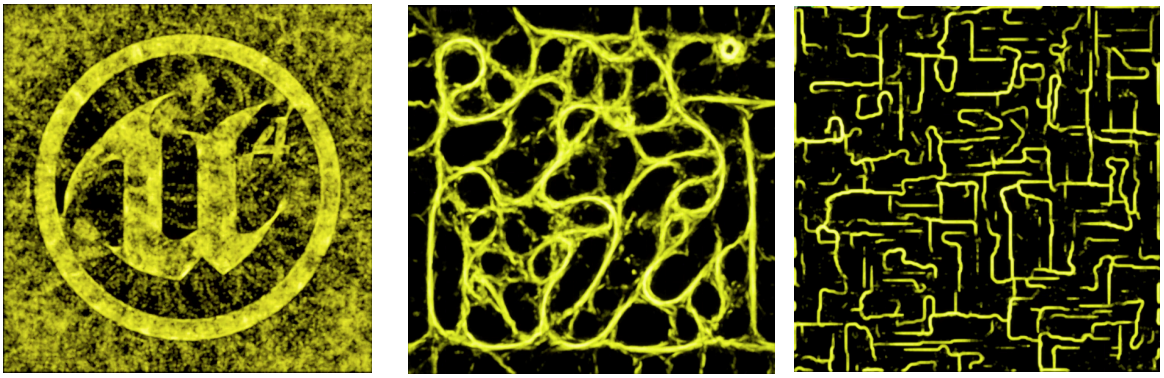

Figure 1: Left: The simulation of the Slime Mold model using an Unreal Engine logo as a food source data map with a high weight. Notice that it forms the logo while keeping the slime-mold-like pattern. Center: The same simulation with 1,000,000 particles. Notice that it produces a highly detailed simulation. Right: The simulation with low sensor distance value. Notice that it no longer looks like a slime mold, however, it forms an interesting pattern.

ABSTRACT

Slime Mold is a type of fungi, which grows efficiently to reach as many food sources as possible. There is a known algorithm for how they grow, and we can simulate the growth of a slime mold with an agent-based particle simulation. This relatively new algorithm for a particle simulation has been applied to many fields, and computer graphics is one of those fields. Simulating and visualizing a slime mold produces an interesting pattern that could be used for texture, particle effects, and etc. We have implemented the slime mold particle simulation using Unreal Engine and explored how each parameter changes the pattern of the particles as well as the limitation with the current Unreal Engine version 4.23.

CCS CONCEPTS

Computing methodologies → Modeling and simulation

KEYWORDS

Generative Art, Particle Simulation

[*] Article Title Footnote needs to be captured as Title Note
[†] Author Footnote to be captured as Author Note

WOODSTOCK'18, June, 2018, El Paso, Texas USA

1 Background

In 2010, Jeff Jones presented the complex emergent pattern formation of slime mold using a simple particle-like agent in his paper [Jones 2010]. In the paper, he explains the way each particle decides its direction based on other particles and the given data and analyzes how each parameter in the algorithm affects the pattern formation. This paper has motivated other artists to explore the effects, using a different platform. Sage Jenson implemented the model which runs in real-time using GLSL with a dedicated GPU. He presented multiple examples on his website and showed how different initial particle positions and the data map produces distinct forging patterns. In the same year, Moritz published a tutorial of the slime mold model for Houdini, a 3D animation software. He modified the algorithm to work in 3D space. After reading these articles, we felt that extending this model to a real-time interactive space may result in an interesting experience. While these two previous

implementations work nicely on their selected platforms, they do not provide real-time interactive experience to the users. We felt that it would be nice if the users can interact with the model by adding particles or changing the parameters in real-time. For these reasons, we chose to implement the slime mold model using Unreal Engine and its built-in Niagara Editor. Niagara is a fully-programmable lightweight node-based VFX editor released in 2018 and allows artists to create VFX effects with less coding knowledge than the previous VFX editor in Unreal Engine, Cascade. Implementing the slime mold model with Niagara allows anyone using Unreal Engine to use the effects in their scenes. Moreover, it has the potential to be extended to a VR application.

2 Methodology

Our implementation is based on the algorithm explained by Jones and Jenson, however, a few modifications were required due to the limitation of the Niagara system. We will discuss the algorithm explained in their paper and the implementation we used.

2.1 Algorithm

Jones explains the algorithm of the slime mold growth as following. Every particle is on a trail map which consists of a given static data and the density of the agents themselves. First, each particle spawned randomly with an initial velocity tries to take a step to the direction that it's facing to as longs as it will not hit the boundary. Then in the Sensory stage, each particle samples trail map values in the area specified with a Sensor Width that is Sensor Offset Distance away from the current Agent Position (Figure 2). It also samples two other areas that are Sensor Angle rotated from the forward. Then the agent selects the direction of the sample area which has the highest trail map value and faces the direction. These operations are repeated every frame.

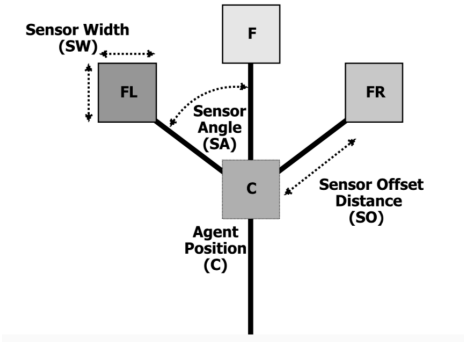

Figure 2: Each particle agent picks three candidate directions based on the given Sensor Width, Sensor Angle, Sensor Offset Distance, and Agent Position [Jones 2010]

2.2 Implementation

In order to implement the algorithm using Unreal Engine Niagara editor, we had to make a few modifications to the algorithm due to some limitations and for performance optimizations. First, instead of storing all the particle location data into a data structure, a Scene Capture camera captures the particles in a scene and store the 2D render as a texture. This texture is updated every frame and can be accessed from the Niagara editor. Each particle has opacity less than one and the particles are on a black background plane, therefore the intensity value of the captured texture indicates the density of the particles at each position. The particles are spawned within a 2000x2000 Unreal-Unit grid, but the resolution of the texture is set to 200x200. This averages the intensity of 10x10 pixels into a pixel and works as a filter. Jones and actually sample each pixel with Sensor Width by Sensor Width area, but we found that using the low-resolution texture can easily substitute the process. Simulating a slime mold model requires a data map, which represents the food source that slime mold tries to collect. Since this data map is a static data, we set a simple texture input in the Niagara Editor. This data map texture is resized to 200x200 and added with the captured density texture when a particle samples three directions. Here, two parameters Data Weight and Deposit Weight are available to users to modify. The sampling and keeping particles within a grid are coded using the Niagara Editor (Figure 3).

Parameters:
N: Number of particles
1: Spawn N particles randomly with a random initial velocity
2: FOR every frame:
3: FOR every particle:
4: Sample at three sensor directions
5: Pick the one has the highest density
6: If particle is out-bounds:
7: Move to the other side of the grid

Figure 3: Algorithm of Slime Mold Model Optimized for Unreal Engine Niagara System.

2.3 Parameters

A total of 11 parameters are gathered to one place for users to easily modify and see the effect in real-time. The first parameter is Spawn Count and represents a number of particles to be spawned. The next is Color, which can be used to modify the color and the intensity of the particles. Three parameters, Sensor Angle, Sensor Distance, and Move Distance are used to determine the three sampling locations. Lifetime defines a lifetime of all the particles. Users can also set Opacity and Particle Size of the particles. Also, Data Texture is available for users to set their own data map, and they can also determine the contribution of the data map and the density map using Data Weight and Deposit Weight. All the float value parameters also provide the ability to change the value over time using a curve editor, which allows users to key-frame the parameter values to create a unique animation. In addition to these 11 parameters, users can also change the resolution of the scene capture texture by going into a Texture Render Target window.

3 Results

Given the ability to change the 12 parameters easily in real-time, we have investigated how each parameter affects the simulated pattern of the slime mold model.

3.1 Sensor Angle and Sensor Distance

We have invested how Sensor Angle and Sensor Distance affect the pattern by simulating the combinations of different Sensor Angle values and Sensor Distance values with other parameters locked. In figure 4, the x-axis is Sensor Angle in degrees, and the y-axis is Sensor Distance in Unreal Engine unit distance. We can observe that lower SA values produce more complex structures of the lines while higher SA values produce a wider gap between curves. Also, lower SD values produce smaller loops while higher SD values produce larger loops. We can observe that higher SD and SA are required to form a realistic slime-mold-like structure, however, lower SD and lower SA values produce interesting patterns.

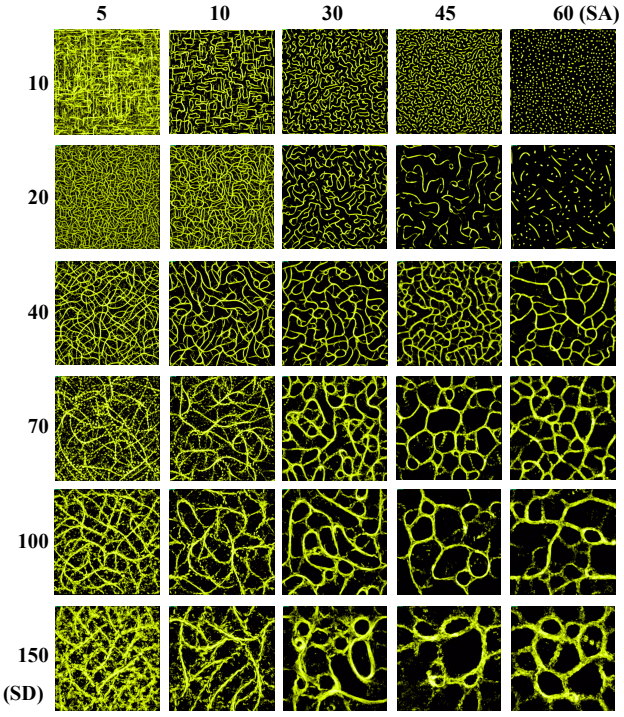

Figure 4: Comparison of the simulation using 5 different Sensor Angle values and 6 different Sensor Distance values. Notice that lower SA produces more complicated pattern and higher SD produces thicker lines.

3.2 Move Distance

Move Distance (MD) is how far each particle moves every frame. Therefore, higher MD values move the particles faster. In figure 5, we have captured the simulation with MD 20 after 3 seconds and MD 200 after 1 second. As two pictures look similar, we can observe that higher MD values let the particles form a certain structure faster than it would with lower MD values. However, higher speed means that the pattern disappears faster also. If one wishes to keep the structure the same for a long period of time, MD has to be a very small value.

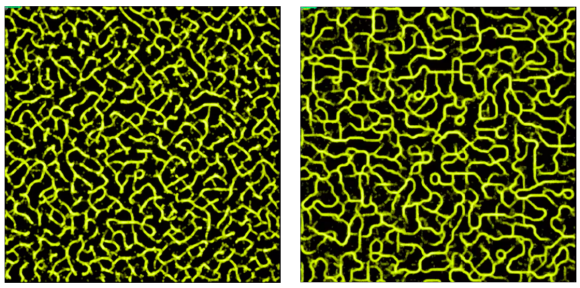

Figure 5: Left: Move Distance 20 after 3 seconds. Right: Move Distance 200 after 1 second. Notice that two patterns look similar, indicating that Move Distance is essentially the play speed.

3.3 Particle Size

Particle Size (PS) decides the size of each particle. This is independent of Move Distance, however, larger particle size means that it magnifies the density because it paints a larger area. This parameter needs to be balanced with Opacity to prevent the result to be too bright or too dark. In figure 6, we have captured the simulated model with 4 different PS, 5, 10, 20, 40. Not only higher PS produces thicker lines, it also affects the structure of the model. This is because the particle size changes the density map with our implementation. Also, higher PS values blur the curves and create a more natural-looking pattern.

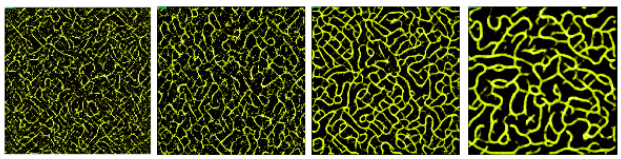

Figure 6: From the left, Particle Size 5, 10, 20, 40. Notice that the size of particles also affects the structure of the model.

3.4 Data Texture and Weights

Data Texture (DT) is a static texture represents a food source that slime mold looks for and move towards. Since the sampling process looks at both the density map and data map, we have added Data Weight and Deposit Weight, which decides the contribution of the data map and density map respectively. In figure 7, we have captured the transition between two weight values using a data texture. In the Data Weight 60% and Deposit Weight 40%, we can observe both slime-mold-like patterns as well as the provided data texture. Users can key-frame the transition of two weights to create an animation with any data texture.

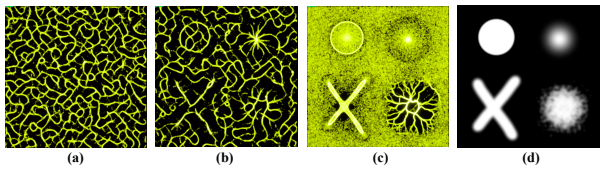

Figure 7: (a) Deposit Weight 100%. (b) Deposit Weight 40% and Data Weight 60%. (c) Data Weight 100% (d) The data texture. Notice that in the transition, we can observe both slime-mold-like pattern and the data texture.

3.5 Scene Capture Texture Size

Scene Capture Texture Size sets the resolution of the texture used to capture the particles for the density map. Increasing the texture size impacts the performance of the simulation, however, users can change the size of the texture as they wish to. In figure 8, we captured the same simulation with three different Scene Capture Texture Size, 100x100, 200x200, 2000x2000. We can see that lower texture size produces rough curves while higher texture size produces smooth curves and more loops. 2000x2000 is the same size as the actual simulation, therefore the texture does not perform any filtering and sensor process might not work accurately. One-fifth or one-tenth of the simulation size is preferred.

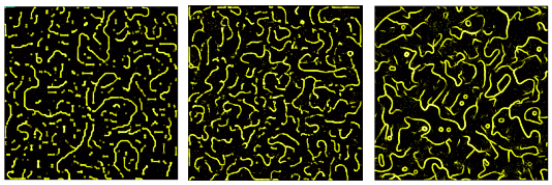

Figure 8: From the left, Scene Capture Texture Size 100x100, 200x200, 2000x2000. Notice that the size of the texture affects the formation of the model.

3.6 Applying the Slime Mold Particles to a Geometry

Since the resultant simulation is in a 2D space, we can use the particle effects as a 2D texture to apply to any geometry by using another Scene Capture Camera. Here, users can set any color or texture as a background instead of the back background used to capture the density. In figure 9, we apply the texture to a 3D model of a giraffe with a brown color background. Mapping the particles onto a geometry requires that the model is nicely UV-mapped.

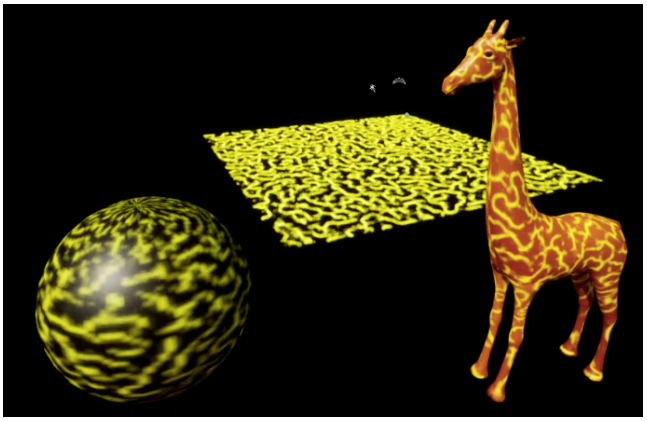

Figure 9: Applying the texture created by simulating slime mold particles to geometries. Notice that the giraffe has a non-black background color. Users can change the color and texture to be overlaid by the particles texture.

4 Future Work and Limitations

This Slime Mold particle simulation project was heavily constrained by the fact that Niagara is still in a beta version. The biggest and unexpected challenge was that there is no way to store the particle location data into any kind of data structure. If Niagara supports writing to a texture or 2D array, it would be easier to create a density grid without a Scene Capture Camera. Also, if this is supported, the extension of this model to a 3D space would be possible. There actually is a way to store data linked to each particle ID, however, this unique particle ID is not supported with GPU simulation, which is required when it has to deal with a texture. Luckily, the official statement of the support for unique particle ID on GPU was released and expected to come with a near-future update of Unreal Engine. The official release of Niagara may introduce some type of data structure that can be useful. For the current limitations, we will have to wait for their update.

REFERENCES

[1] Jeff Jones, 2010. Characteristics of Pattern Formation and Evolution in Approximations of Physarum Transport Networks. In *Proceedings of Artificial Life 16: 127–153 (2010)*

[2] Sage Jenson, physarum. 2019. https://sagejenson.com/physarum

[3] Moritz, Physarum Slime Mold, 2019. https://entagma.com/physarum-slime-mold/