TECHNISCHE UNIVERSITÄT DRESDEN

IXLAB

Faculty of Computer Science Institute of Software and Multimedia Technology

Junior Professorship in Immersive Media

Student Research Project

Particle Simulations on 2D Manifolds: Implemented for Physarum Models on the Surface of Objects in a dynamic 3D Environment

Joram Brenz

Course: INF-D-950
Matriculation number: 4504598
Matriculation year: 2015

to achieve the academic degree

Diplom Informatik (Dipl.Inf.)

Supervising professor
Matthew McGinity

Submitted on: 12th August 2022

TECHNISCHE UNIVERSITÄT DRESDEN

IXLAB

Faculty of Computer Science Institute of Software and Multimedia Technology

Junior Professorship in Immersive Media

Statement of authorship

I hereby certify that I have authored this Student Research Project entitled *Particle Simulations on 2D Manifolds: Implemented for Physarum Models on the Surface of Objects in a dynamic 3D Environment* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. During the preparation of this thesis I was only supported by the following persons:

Matthew McGinity

Additional persons were not involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 12th August 2022

Joram Brenz

Contents

List of Figures

1 Introduction

Physarum Slime Mold is studied because of its fascinating emergent intelligence apparent in its navigation and interactions with its environment. As its emergent properties can be realised in surprisingly simple models, it is often studied through simulation - leading to a whole branch of research known as "physarum computing".

These simulations however typically take place in highly abstracted environments. Ignoring features like elevation, curvature and normals that surfaces in a 3D world have, and which are important for studying the effect of outside influences like gravity, lighting, collisions, etc.

In this project we develop a method of simulating the physarum on arbitrary 2D manifolds and transfering particles between manifolds, allowing for the simulation of physarum growth in virtual environments that resemble the real world.

Further, real-time simulation and immersive display technology allow for the construction of a Virtual Reality Physarum Laboratory, in which humans can manipulate and influence the environment and simulation directly.

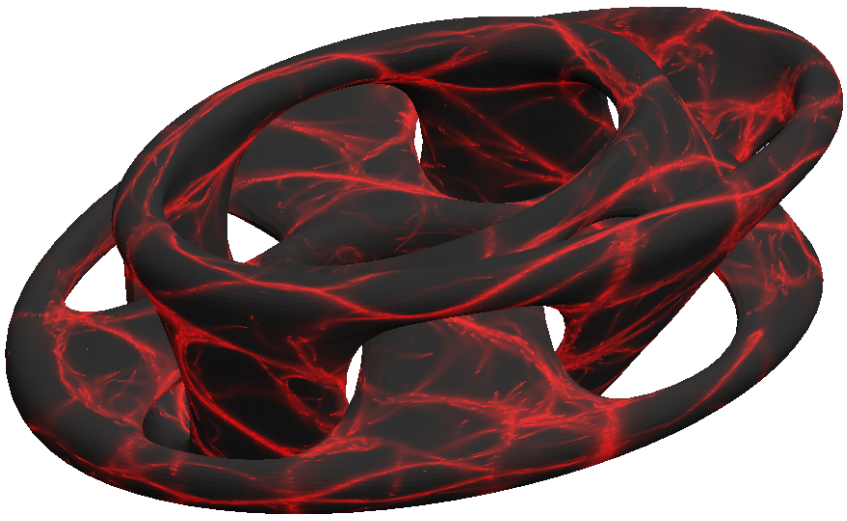

Figure 1.1: Physarum on complex Surface[1]

2 Background

2.1 Manifolds and Euclidean Space

Surfaces of 3D objects are an example of what mathematicians call 2D manifolds. They have no border and do not self intersect. In theory manifolds can be either closed or infinite, but for this work we will only be talking about finite ones. [16]

A euclidean space is a space that is not curved in a higher dimension. This is what we normally use for anything geometry related. The 3D space that our object is in will be euclidean, but in contrast to a plane, the surface of the object will be a non-euclidean 2D space, because it is curved in the 3rd Dimension. [12]

2.2 Homogeneous Coordinates

In developing the formulas for our method we sometimes represent points and vectors with an additional dimension. The coordinate for that dimension is always 0 for direction- and 1 for point-vectors.

This is an application of Homogeneous Coordinates [15] often used in computergraphics to allow for the expression of any Affine Transformation [8] with a single matrix multiplication. We use this for coordinate space transformations.

2.3 Particle Simulations

There are many people using the term "Particle Simulation" refering to different kinds of particle-based simulation methods, but finding a clear definition was more difficult than anticipated.

In the article "Computer Simulations in Science" Winsberg lists equation-based simulations and agent-based simulations as the two main types of computer simulations. Regarding equation-based simulations he continues to make a distinction between field- and particle-based simulations. But later notes that the individuals of agent based simulations are quite similar to particles. [19]

There are cases where the individuals of agent based simulations are straight up called particles. Like it is the case for self-propelled particles [18].

Field based methods are often used in computational fluid dynamics [10] or similar fields and examples include the finite element method [14] and finite difference method [13].

Examples for particle based methods are molecular dynamics [17], discrete element method [11] or autonomous agents [9].

Physarum Models are interesting in that they make use of particles and a field. The field evolves according to diffusion equations, but in addition to that is influenced by the particles. The particles act as indipendent agents, basing their decisions on the local field and their own state. Notably there is no direct interaction between particles.

This also shows already what kind of simulations our method will be applicable for. Namely field based simulations, and models with agent based particles where the interaction between them can be expressed as field based particle interactions.

As a side note, our method should also work for cellular automata as they share many similarities with how field based models are discretised.

2.4 OpenGL / WebGL

Many of the well established and widely available GPU APIs are designed to be used for classic rendering tasks. This was particularly noticable when creating a browser based prototype.

Compute shader support for browsers will become available in the future with the implementation of the vulkan based webgpu standart, but with the current webgl 2.0 standart, which is based on opengl es 3.0, the only way to do general computing is using the vertex or fragment shaders and rendering the results into frame buffer textures.

The limitations in available texture formats and access patterns dictate to a high degree the structure of the solution, but it was possible to map the problem in a way to create a reasonably fast implementation.

2.5 Unity3D

Unity3D is a game engine that will be used for the project to make integration with existing vizualitation and interaction technology easier (see next section). It does provide computer shader support but to efficiently integrate the compute shaders for the simulation with the rest of the rendering pipeline - especially mesh data, texture mappings, shadow maps, etc. - will probably still pose some challenges quite similar to the implementation with WebGL.

2.6 Visualization and Interaction in AR

With the use of 3D scanning, the simulation could be run on real physical objects. The simulation could then be interacted with by manipulating the real object and observed by overlaying the simulation using an AR headset.

3 Related Work

3.1 Physarum Simulations

The Physarum Slime Mold has been studied for its emerging behaviour and potential in optimizing transport networks by a number of people. [todo: cite people here]

For this work I will employ the model from Jones' article "Characteristics of pattern formation and evolution in approximations of physarum transport networks" [4] as described by Jenson [3].

Both of them do a great job of describing the model, so I will just give a short summary.

The model consists of a trail map and freely moving particles or agents. The particles alter their direction based on the trails in front of them and leave fresh trails behind them while moving.

Essentially there are three rules that describe the interaction of these components:

- particle movement: The particles are always moving with a constant speed. They sense the trail intensity at three angles in front of them and adjust their direction to follow the highest concentration.
- trail deposition: The particles leave behind a trail.
- trail diffusion and decay: The trails diffuse and decay over time.

These simple rules result in a fascinating emerging behaviour where the particles create a network of trails.

4 Design Methodology

4.1 The Objects

There are different ways to describe the surface of an object. The most common explicit approach is to use a triangle mesh, so that's what we will do here.

To distinguish discretized curved surfaces and actual edges, surface normals are often annotated to the corners of the triangles. Using this information to consider the curvature of each triangle would most definitely improve the quality of the simulation. However it also considerably increases the complexity of the calculations and can to some degree be compensated for by remeshing with smaller triangles. So to start with, all triangles will be considered flat. Since this decision does not influence the general architecture of the solution it should be possible to add later.

4.2 The Particles

To describe the particles, some information needs to be stored for each of them. Such as:

position: triangle_id and uv coordinates

The position of a particle could be given either in 3D space (world space or model space) or in regard to its position on the triangle mesh. Since it is easier to derive the 3D position from a given position on the triangle mesh than to find the nearest triangle for a 3D position, the position of the particles will be stored using a triangle_id and 2D coordinates for the precise position on that triangle.

From the different options for the triangle coordinates the uv coordinates will be used, because:

- they scale nicely when the triangles corner points move,
- it's easy to check out-of-triangle condition,
- they can be used pretty straighforward with the trail texture (for sensor readings, trace deposition and render pass)

orientation: angle

For the orientation, either an angle or direction vector could be used. Triangular functions can be used to translate between angle and a vector in orthogonal space.

The direction vector (in uv space) is needed when applying the linear offset and when moving the particle. The rotational offset for the sensors is given as an angle, but could be translated to a matrix before passing to the shader.

For our implementation we decided to store the orientation as an angle, because it only needs a single field and that turned out to be benefitial when dealing with the limits of the webgl implementation.

velocity: implicit

For the physarum simulation no velocity needs to be stored, because it is a common constant for all particles.

If the particles had individual velocities there would be two possible cases. Either the movement is coupled to the orientation, in which case a scalar would be enough to describe it, or the movement is independent from the orientation, in which case a vector would be required.

The scalar could also be combined into the orientation when storing the direction vector instead of the angle.

4.3 Frames of Reference

For the simulation, multiple coordinate frames are used. Here is how they interact and what they are used for.

4.3.1 World Space

All objects and particles of the simulation exist in one 3D World. Positions in this world are described using 3D karthesian coordinates. These world coordinates are what all the other frames of reference are based in.

4.3.2 Model Space

Each model has a model space, used to describe the vertices of its mesh relative to an anchor point. This decouples the shape of the objects from their location.

4.3.3 Triangle uv Spaces

To describe the position of points on a triangle, triangle uv coordinates are used. These use two sides of the triangle as base vectors. Figure 4.1

Triangle uv coordinates can be translated into texture uv or world coordinates using the triangles corner positions in the corresponding target space t.

$P_t = (A1_t - A0_t) * u + (A2_t - A0_t) * v$

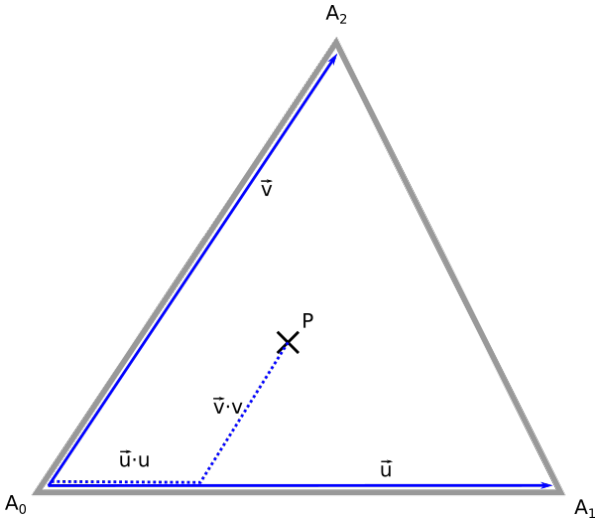

Figure 4.1: UV Space

uv Space Rotation

To unify different cases of triangle orientations it is usefull to rotate which vectors are used as base vectors for the uv space. (see Crossing Edges section)

From Figure 4.2 the following formula for rotating one corner anti-clockwise can be derived:

$$u_b = v_a$$
$$v_b = 1 - u_a - v_a$$

This can then be written as a transformation matrix:

$$\begin{bmatrix} u_b \\ v_b \\ 1 \end{bmatrix} = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ -1.0 & -1.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} u_a \\ v_a \\ 1 \end{bmatrix}$$

As to be expected when rotating a triangle, the third power of this matrix is the identity. Also, multiplying by this matrix twice is the same as multiplying with the inverse of it.

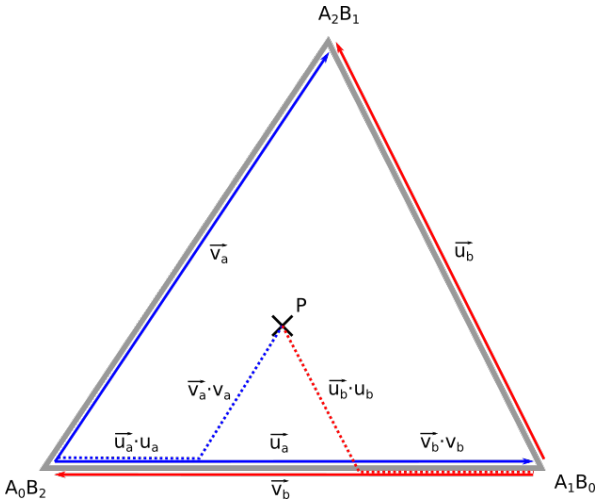

Figure 4.2: UV Space Rotation

4.3.4 Triangle ph Spaces

The introduction of orthogonal ph spaces is necessary, because the trigonometry functions only work consistently with orthogonal base vectors of the same length (in regards to world space).

Also, working with base vectors that are normalized makes it easier to have particles move at a consistent speed no matter which triangle they are on.

To get the normalized base vectors $\overrightarrow{p_n}$ and $\overrightarrow{h_n}$, first $\overrightarrow{v}$ is projected onto $\overrightarrow{u}$ to get $\overrightarrow{p}$, then $\overrightarrow{p}$ is substracted from $\overrightarrow{v}$ to get $\overrightarrow{h}$. Both of which are then normalized.

To transform coordinates from uv space to normalized ph space the following transformation matrix can be used:

$$\begin{bmatrix} p_n \\ h_n \end{bmatrix} = \begin{bmatrix} |\overrightarrow{u}| & |\overrightarrow{p}| \\ 0 & |\overrightarrow{h}| \end{bmatrix} * \begin{bmatrix} u \\ v \end{bmatrix}$$

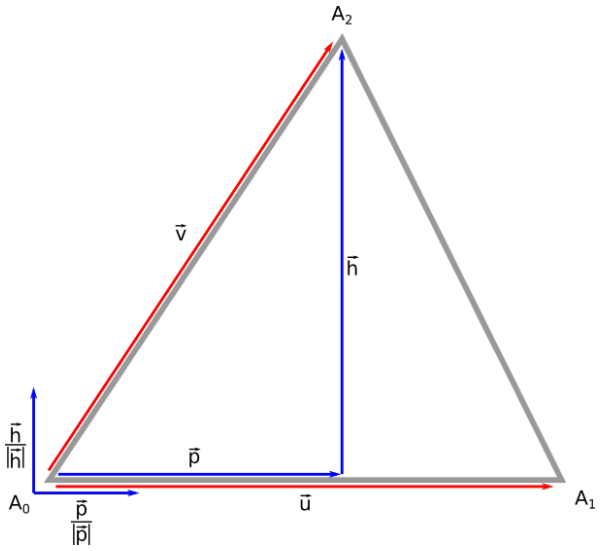

Figure 4.3: [normalized] ph Space

4.4 Crossing Edges

Since a manifold is supposed to locally resemble Euclidean space near each point [16] it has to be possible to represent the position of a point near the edge in either triangles coordinate system and since they are part of the same local euclidian space, there should be a linear relationship between them.

This can be imagined as folding the two triangles flat along the common edge, so that they end up in the same plane while preserving the length of their sides.

With some rotating of the uv spaces if necessary, it is always possible to reduce the problem to the situation shown in Figure 4.4. Here the Particle P has left the inside of triangle A and crossed the edge into triangle B. Therefore the coordinates $(u, v)_a$ (where u<0) have to be transformed into the coordinates $(u, v)_b$.

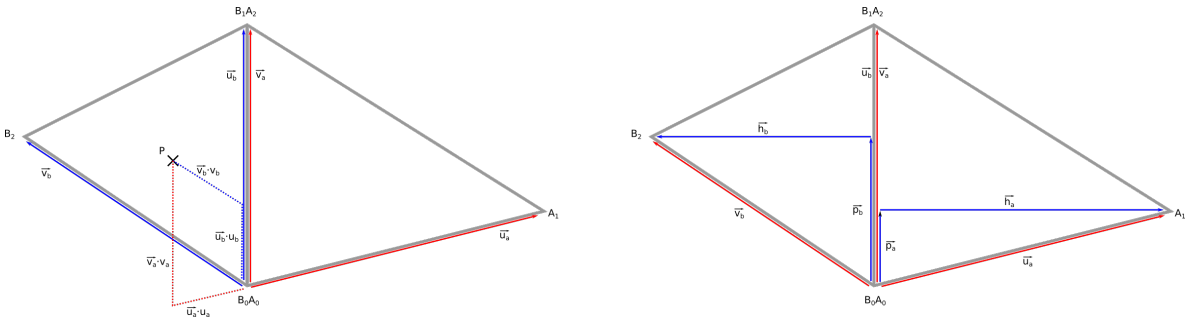

Figure 4.4: Geometry of Crossing Edges

The new and the old position are the same just with different base vectors.

$$u_a\overrightarrow{u_a} + v_a\overrightarrow{v_a} = u_b\overrightarrow{u_b} + v_b\overrightarrow{v_b} \tag{4.1}$$

The vectors can be split into projection and height vectors.

$$u_a\overrightarrow{u_a} = u_a\overrightarrow{p_a} + u_a\overrightarrow{h_a} \tag{4.2}$$
$$v_b\overrightarrow{v_b} = v_b\overrightarrow{p_b} + v_b\overrightarrow{h_b} \tag{4.3}$$

therefore

$$u_a\overrightarrow{p_a} + u_a\overrightarrow{h_a} + v_a\overrightarrow{v_a} = u_b\overrightarrow{u_b} + v_b\overrightarrow{p_b} + v_b\overrightarrow{h_b} \tag{4.4}$$

Now since the height vectors are orthogonal to the others which are all parallel, this equation can be split into two independent ones,

$$u_a|\overrightarrow{h_a}| = -v_b|\overrightarrow{h_b}| \tag{4.5}$$
$$u_a|\overrightarrow{p_a}| + v_a|\overrightarrow{v_a}| = u_b|\overrightarrow{u_b}| + v_b|\overrightarrow{p_b}| \tag{4.6}$$

and solved for u_b and v_b.

$$v_b = -\frac{|\overrightarrow{h_a}|}{|\overrightarrow{h_b}|}u_a \tag{4.7}$$
$$u_b = \frac{|\overrightarrow{p_a}|}{|\overrightarrow{u_b}|}u_a - \frac{|\overrightarrow{p_b}|}{|\overrightarrow{u_b}|}v_b + \frac{|\overrightarrow{v_a}|}{|\overrightarrow{u_b}|}v_a \tag{4.8}$$
$$= \frac{|\overrightarrow{p_a}|}{|\overrightarrow{u_b}|}u_a + \frac{|\overrightarrow{p_b}|}{|\overrightarrow{u_b}|}\frac{|\overrightarrow{h_a}|}{|\overrightarrow{h_b}|}u_a + v_a \tag{4.9}$$
$$= \left(\frac{|\overrightarrow{p_a}| + \frac{|\overrightarrow{h_a}|}{|\overrightarrow{h_b}|}|\overrightarrow{p_b}|}{|\overrightarrow{u_b}|}\right)u_a + v_a \tag{4.10}$$
$$\tag{4.11}$$

The resulting equations can be combined into a transformation matrix.

$$\begin{bmatrix} u_b \\ v_b \\ 1 \end{bmatrix} = \begin{bmatrix} k_2 & 1 & 0 \\ k_1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} u_a \\ v_a \\ 1 \end{bmatrix}$$

where

$$k_1 = -\frac{|\overrightarrow{h_a}|}{|\overrightarrow{h_b}|}$$
$$k_2 = \frac{|\overrightarrow{p_a}| - k_1|\overrightarrow{p_b}|}{|\overrightarrow{u_b}|}$$

how to decide which edge to cross

While triangle uv coordinates represent unique positions when inside the triangle, identical coordinates outside the triangle may need to be mapped to different neighbouring triangles and positions depending on which edge was crossed to get there.

So to always be able to apply the correct transformations it is necessary to also know from where a position was reached. By using a direction vector to do a line intersect test with all three triangle edges, the one that was actually crossed can be determined.

Now since the size of the areas where these errors can occur is dependent on how far outside the triangle the position is, a faster approach may be used without much problems, when the step size of the motion is significantly smaller than the size of the triangles.

4.5 Texture Mapping

To be able to render to every triangles texture in one pass they have to be part of one big texture.

When using a regular pattern, the position of each triangle in the texture can be calculated from its id.

However to achieve better results on models with highly varying triangle sizes it will be preferable to create a texture map and store the texture uv coordinates for each triangle corner with the triangles.

4.6 Data Model

From the sections above we can collect what our implementations data model should generally look like.

```
Particles[id]
    position
        uint triangle_id
        vec2 triangle_uv
    [vec2 velocity]
    [float orientation]

Triangles[triangle_id]
Edges[edge_index]
uint triangle_id
uint edge_index
```

How the mesh is stored will most likely be determined or strongly influenced by the framework used for the implementation. Generally there are two ways to store meshes, either indexed or not.

```
Triangles[id]
    ivec3 vertex_ids

Vertices[id]
    vec3 model_space_position
    [vec2 texture_uv]

or

Triangles[id]
    Vertices[3]
        vec3 model_space_position
        [vec2 texture_uv]
```

Using a seperate vertex array and having the triangles index into it reduces the memory footprint when points can be reused for multiple triangles. Also by keeping the points indexed it is easier to upload new positions at runtime. However storing points directly in the triangle array reduces the number of necessary texture lookups.

When including uv coordinates the value of indexing becomes even more difficult to predict. Depending on the texture mapping only some or even no points will also share their uv coordinates.

How the data is finally grouped, including whether edges can be part of the mesh data or need their own buffer will also be restricted by the available storage types. OpenGL textures will only allow 4 times 32bit per entry and no mixing of float and int. (There is functionality to interprete uint32 as float.) StorageBlocks (or a similar technology) on the other hand would allow for structs with mixed data types.

4.7 Transfer of Particles between Meshes

So far we talked about the simulation on one manifold or mesh. However in our virtual environment we want the physarum to be able to spread over multiple objects.

For this we need to detect which objects are touching and where. Then we can redirect the affected particles to go from the surface of one object to that of another.

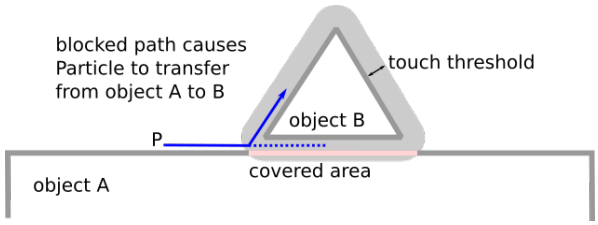

Figure 4.5: Particle Transfer

Adding a threshold distance can be necessary because, due to uneven surfaces and imprecisions of the physics engine, objects won't always exactly touch.

4.7.1 Selecting Particles with SDF

Generally we want to transfer particles upon entering a covered area. When a particle enters a covered area by moving we can detect this by comparing the situation before and after the step. However, particles can also end up in a covered area due to movement of the respective objects or after a transfer if the covered areas of the two objects don't line up exactly.

A more stable transfer condition can be designed, by replacing the binary concept of being inside or outside a covered area with a value for how far into, or away from a covered area a particle is. This is called a signed distance. The gradient of the signed distance in the direction of a particles movement tells us whether the particle is entering or leaving the covered area.

For environments with many static objects it is inefficient to recompute the signed distance for every step since it remains constant for each location on the objects as long as the objects don't move. Instead we can compute once what is called a signed distance field and use it to look up the signed distance whenever we move a particle.

4.7.2 Target Location

We want to make the trajectory of particles across objects just as seamless as when crossing triangle borders on a mesh. So first we need to find the nearest triangle on the target object to our current particle position. Depending on the implementation this is necessary for calculating the signed distance in the first place and can be stored as an additional value in the signed distance field.

We then need to find the folding line which is the line of intersection from the plane the particle is on to the plane the target triangle is in. Using this we get a formula quite similar to the one for crossing triangle borders. Except for, since the target plane extends in two directions from the line of intersection we receive two possible target locations. At this point we can use the signed distance field again to determine which location is less covered.

5 Implementation

During the realization of this project two solutions have been implemented.

The first one which uses HTML, JavaScript and WebGL is more of a standalone demo / proof of concept prototype. The second one uses the Unity3D Game Engine and was designed to also be compatible with possible future works. While they share most of the core functionality some features, like simulations with multiple objects and transfering particles between them, are only available on the Unity implementation.

5.1 Technology/Framework Options

5.1.1 WebGL/HTML/JavaScript

WebGL is an OpenGL based rendering API for the web. There are two major versions: WebGL [1.0] and WebGL 2.0 . The most important difference for this project being the availability of configurable framebuffers. So to be able to render to a texture instead of the screen we needed to use WebGL 2.0 .

While the official specification for version 2.0 is still work in progress, it states that the standard tries to "conform closely to the OpenGL ES 3.0 API" [2]. Based on that, webgl2.0 has already been implemented on almost any modern browser for years with very few incompatibilities remaining, none of which where a problem for this project.

OpenGL3 introduced the shader based rendering pipeline and OpenGL ES 3.0 defines a subset that drops most of the legacy support. It is mostly used in low power devices like smartphones and can be expected to define the minimum functionality available for any device running a modern browser.

One of the main reasons to go for a web based implementation is it's crossplatform compatibility. The same application can be served to Linux, Windows, Mac, Android and IOs devices. There is no need for compilation, but using a built script based on the npm modules html-minifier-terser, strip-json-comments-cli and inline-scripts we managed to pack the entire application into a single html file for easier distribution.

Another more personal reason to start with a WebGL implementation was that it is a framework that I was already used to working with. So while I knew that in the end a Unity Implementation would probably be preferable, I think it was the right decision to start with a webgl prototype.

Doing so allowed me to reduce the sources of errors and focus on one problem at time. Also the additional iteration resulted in a cleaner result for the second implementation.

There were no major problems with implementing the proposed formula for our method in the prototype. But there were some bugs caused by smaller mistakes like sign errors. And while causing unnecessary trouble they also created some interesting new patterns:
(The bottom right image shows the correct behaviour.)

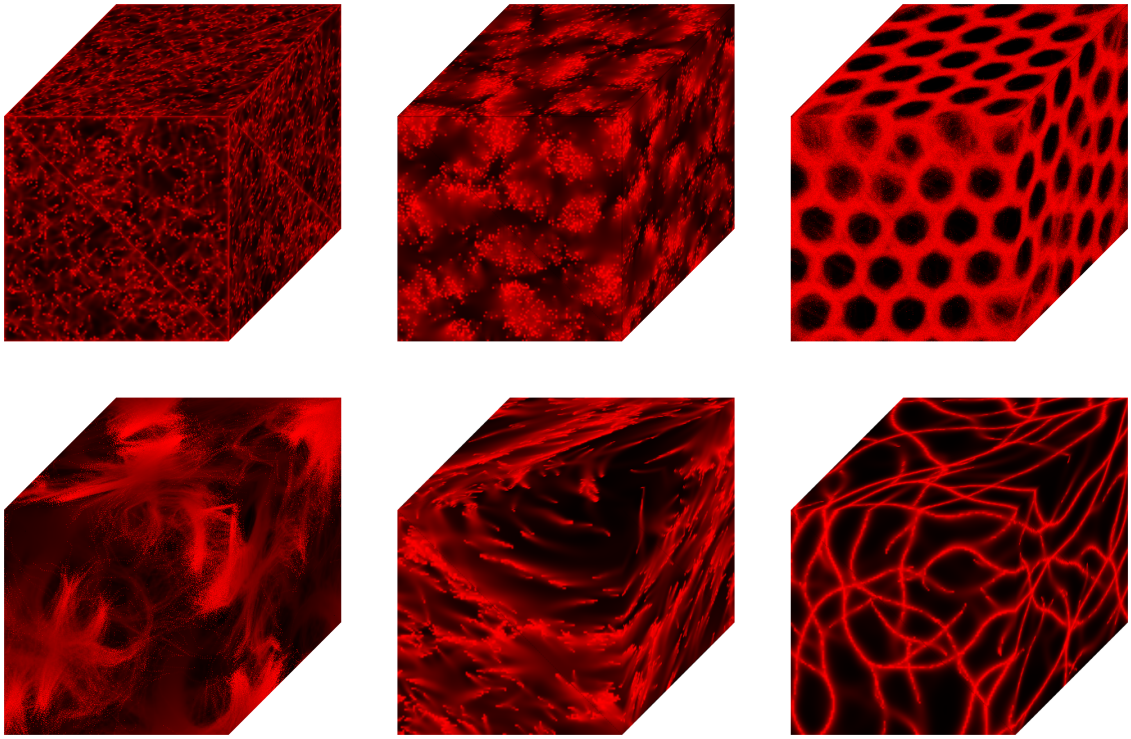

Figure 5.1: Beautiful Bugs

5.1.2 Unity 3D

The Unity Engine is designed for making Computer Games. This also makes it a useful tool for creating 3D visualisations and simulations in general.

In contrast to WebGL, which is only a graphics API, Unity is a framework with a lot of additional functionality already built in. There is built in support for loading meshes. The editor allows to move and interact with objects. Properties can be changed in the inspector at runtime. And there are APIs and templates for integrating VR or AR functionality. All this has the potential to add a lot of interactivity to the simulation without much additional work.

Using the inspector for manipulating parameters instead of creating a dedicated gui makes it easy to add new options. To solve problems with properties that should not be changed at runtime or only can be used in certain combinations we added a custom inspector class that disables properties based on those conditions.

The rendering pipeline is highly integrated with the framework to allow for all the features a game would typically need and is already set up to work like this out of the box. This made it a bit harder to find information on how to do custom shader calls. But once I figured out how to call rendering from a script instead of having to use the main unity update loop and how to use layers to render only a specific mesh without the original scene in the background, it was pretty uncomplicated to translate the webgl solution into unity.

In contrast to webgl which uses glsl, unity uses a shader language called HLSL (high level shader language) which was designed for directX but can be crosscompiled to work with several graphics APIs including vulkan, metal, opengl and webgl. In addition to the classic graphics shaders, it also supports using hlsl for writing compute shaders. Using compute shaders some of the shaders can be rewritten to be cleaner and probably more performant. However, not all features are available on every build target. So by making use of some of the more general computing focused features we will loose compatibility with older architectures.

5.2 Data Access and Organization

WebGL

To avoid the bottleneck of copying data between main system memory (RAM) and GPU memory (VRAM) everything relevant to the simulation must be stored on the GPU. In WebGL shaders we have read access to uniforms and textures. Information from Vertex buffers (VBOs) is only made available in a very limited way through the draw call. Other OpenGL features like Storage Blocks and similar are not available. The only output of the shader are the colors drawn to the renderbuffer. Using webgl 2.0 it is possible to bind one or more textures to the renderbuffer. However when using more than one texture they have to be of equal dimensions.

Also textures have their own set of limitations. Namely they can only have up to 4 channels and all of them have to have the same data type. This has multiple implications. To keep the number of memory calls low related data should be stored in a single pixel. For particles this means we will use an angle and not a direction vector for the orientation. Also the triangle id will be stored as a float, reducing the maximum possible number of triangles to 2**24.

Another limitation of textures is the size. All dimensions of a texture are limited seperately, so to store more data, arrays need to be broken into lines (2D instead of 1D). 3D textures could store even more data, but can't be rendered to. (Or only one layer at a time.)

Since textures can't be bound to a framebuffer and read at the same time, both the trailmap and particle texture have to be duplicated and used alternatingly.

HLSL

While the verion of glsl used for webgl was designed in line with the classic staged rendering pipeline, HLSL offers more features that are suited for general computing tasks. With Structured-Buffers custom structs can be used to define the layout of the data. There are StructuredBuffer types for read, write and append access. The last one being especially usefull for collecting data from some but not all threads, as is the case for the particle transfer.

So for the Unity implementation we will only keep the trailmap as a texture and use the structured compute buffers for particles and edges instead.

Since C# and HLSL are both C based languages it was even possible, with some precompiler statements, to outsource the struct definitions into a seperate file that gets included in both the HLSL shaders and the C# part of the Unity implementation.

Mesh API Compute Shader Access

When the work on the Unity implementation for this project started, direct access to the vertex and index buffers of a mesh was only available on the alpha version. There was no official documentation but only an example project [7] and a google document [5] describing the feature. It was later officially released with Unity 2021.2.

Instead of having to manually copy and update mesh data to a custom buffer this feature allows direct access to the same mesh buffers used by the framework. To be able to it the type of the buffers has to be adjusted. Also, since the buffers are managed by the framework and their location could change without notice, the uniform bindings for the shaders have to be updated each time.

```
// on initialization
mesh.vertexBufferTarget |= GraphicsBuffer.Target.Raw;
mesh.indexBufferTarget |= GraphicsBuffer.Target.Raw;

// each update
GraphicsBuffer vertexBuffer = mesh.GetVertexBuffer(0);
shader.SetBuffer("_VertexBuffer", vertexBuffer);
GraphicsBuffer indexBuffer = mesh.GetIndexBuffer();
shader.SetBuffer("_IndexBuffer", indexBuffer);
```

On the shader side there will probably be an api for accessing these buffers available at some point. But for now we had to write our own library. The index data can be formatted in different ways. Only the more common uint16 formatting has been implemented. There two 16bit indices are packed into a 32bit field. Then those fields are spaced out every four 32bits. Probably because of some hardware restrictions. For the vertex data, stride and offset can be read out in C# and passed to the shader as uniforms.

```
// make sure mesh index format is uint16
Debug.Assert(mesh.indexFormat == UnityEngine.Rendering.IndexFormat.UInt16);

// pass vertex buffer stride and vertex attribute offset[s]
int vertex_buffer_stride = mesh.GetVertexBufferStride(0);
int uv_offset = mesh.GetVertexAttributeOffset(UVAttribute);
shader.SetInt("_VertexBufferStride", vertex_buffer_stride);
shader.SetInt("_VertexBufferUVOffset", uv_offset);
```

5.3 Initialization

5.3.1 Models

Several models have been used. They can be found in a common folder outside of the implementations. Most of the development was done with a simple six sided cube. The "torus" model [1] by sonic art was used as a more complex shape for testing and demonstration. The cube consists of 12 triangles, the torus of 13,824 quads. In addition to that unity builtins like the sphere and pill were used for testing. A bigger "office" model was considered as an environment, but was not used yet as it contains open edges.

In Unity the loading of the model file is done by the framework and many file formats are accepted. For the webgl implementation a custom load function for .obj files was written. It only handles vertex positions and faces. Additional information like texture, normals or submeshes are ignored. Also only triangles or quads are accepted for faces. Since the webgl solution does not implement moving the camera or the object, automatic scaling is applied to fit the object into the viewport.

For the simulation we need a way to look up how the triangles are connected. In the mesh this information is only implicitely included. By comparing pairs of corners of the triangles an index is built that contains which edge of one triangle is connected to which edge of another triangle. This fails if the mesh contains open edges, or if more than two triangles meet in one edge.

5.3.2 UV Mapping

Each triangle needs a dedicated area on the trailmap texture. Since there is no image data or similar prepared beforehand this texture mapping does not need to be part of the model and can be decided at runtime.

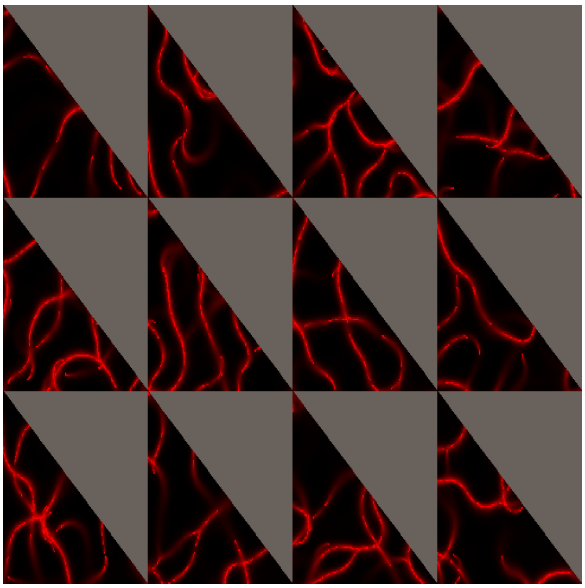

Figure 5.2: implicit grid based uv mapping

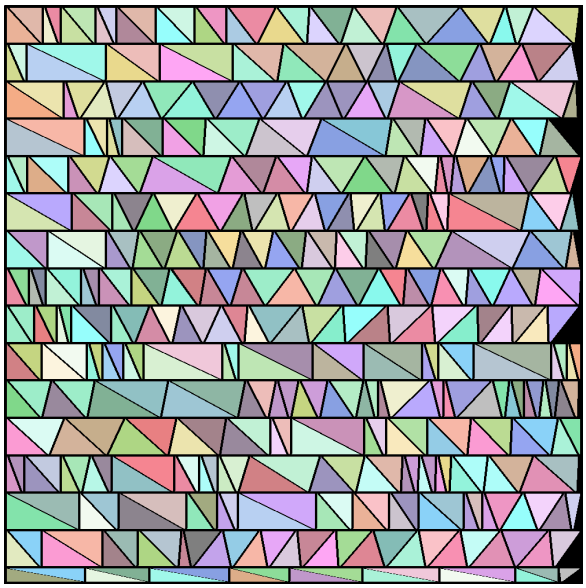

Figure 5.3: explicit auto uv mapping

For the webgl implementation a simple implicit approach has been used, where each triangle is mapped to a cell on a grid based on its index. This is shown in Figure 5.2 Obviously the grid cells could be split diagonally for a more compact packing. But more importantly this mapping does not retain the shape or relative size of the triangles. Distortion and varying pixel density is mostly a problem for the diffusion calculation, but also creates problems with interpolation. One idea was to adjust the size and form of the triangle inside the cell, but this only works if they are all about the same size or again a lot of space would be wasted.

By calculating the mapping beforehand and supplying the uv coordinates to the shader as part of the vertex data, like it is normally done for texture mapping, a way more efficient packing while retaining shape and relative size can be achieved. A "auto uv" script for creating such mappings has been written in python as a prototype and the result is shown in Figure 5.3. But the developed algorithm was relatively slow and abandoned upon realizing that Unity does already provide an import setting for models to create lightmap uvs that satisfies the requirements.

Adding a border between not connected triangles can help avoid false interpolation.

5.3.3 Particles

The initial placement of the particles is done as a uniform random distribution, where each location on the mesh is equally likely to be chosen. Other initial placement strategies can produce more interesting effects in the early phase of the simulation, but this seemed like the most neutral approach.

The selection of a triangles of the mesh, and a rotation between 0 and 360 can be done independently without a problem. For the uv values however, if their sum is greater than 1 the particle isn't actually on the triangle anymore.

The first attempt at solving this, was to select one of the values first and restrict the range of the second value accordingly. This had the problem, that values were evenly distributed for the first axis, but as the cross section of the triangle changes, the density of particles increased towards one of the tips of the triangle.
To avoid this, the final solution instead chooses random values between 0 and 1 for u and v independently and if the sum is greater than 1 replaces both with one minus the respective value. This means the space of possible positions starts as a parallelogram of twice the size and is effectively cut diagonally, moving the additional triangle on top of the original.

For the selection of the triangle, all triangles are currently chosen with equal probability. This only works for meshes where all triangles are about the same size. For a more general solution triangles should be weighted by their area.

5.4 Using Shaders

Vertex and Fragment Shaders

The two shader types that can be assumed to be available on any modern GPU are vertex shader and fragment shader - they are always used as a pair. The vertex shader gets its input from the vertex and index buffers. The fragment shader calculates colors and sometimes depth values for

fragments, that are then blended and output to the renderbuffer. Between them is the rasterization stage that creates a varying number of fragments depending on the size and position of the rendered primitives (mostly triangles). Figure 5.4 shows how depending on the position (xyz or uv) chosen for the vertices in the vertex shader different fragments are generated for each primitive.

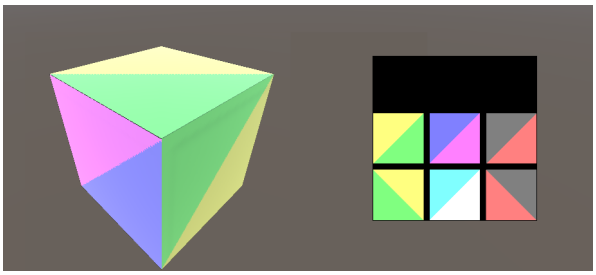

Figure 5.4: Drawing to Screen vs Drawing to Texture

Compute Shaders

There are some parts of our simulation where we don't really need this separation into multiple stages. With webgl we use the vertex shader just to set up the correct fragments in this case and do all the actual calculation in the fragment shader.

With compute shaders we can completely avoid the unnecessary overhead of using the rendering pipeline.

On the other hand, if a problem is structured in a way that fits the vertex shader / fragment shader model very well, we may want to keep using them to take advantage of more optimized data paths in hardware.

Instanced Rendering

Instanced Rendering allows to draw the content of the Index and Vertex Buffer multiple times. Since all necessary data is stored on the GPU, placing a single triangle in those buffers, and using the instance ID to look up the actual triangle data, can replace a normal indexed draw call.

This gives us a way to get the index of the triangle that a vertex and then fragment is part of, which is not really possible when using instanced drawing, because one vertex in the vertex shader could be used for multiple primitives in the rasterization. In Figure 5.4 the instance id has been used to select a common color for all vertices of the same triangle.

Also the webgl implementation needs a texture with the mesh data and using instanced rendering we can avoid having to duplicate that data for the index and vertex buffer.

List of Shaders

The following is a list of all the shaders that are part of the simulation.

- Particle Shader

- Deposition Shader
- Diffusion Shader
- Visualization Shader
- SDF Shader
- Transfer Shaders

5.4.1 Common Functions

Since all of the above mentioned shaders work on the same data structures there were a lot of the same helper functions needed for all of them. Most of them are related to accessing the content of the textures or buffers and doing coordinate space transformations.

In webgl the shader code is passed as a single string, so the common code is just another string that gets prepended to the source code of each shader. But in Unity it was possible to use #include and other preprocessor directives to organize a custom library.

The MeshAccess.cginc module contains functions for reading mesh data from the Index and Vertex Buffer.

The Manifold.cginc module contains functions that use the information about connected edges to allow transformations from one triangle to another and functions for translating between texture uv, triangle uv, triangle nh and 3D model space in general. Also there are helper functions in there for normalizing out of triangle locations by automatically changing to the correct neighbouring triangle up to a configurable number of EdgeCrossAttempt.

The Debug.cginc module uses a computer buffer bound in append mode to store debug output that can then be read from C# and printed to the terminal for the developer.

The Random.cginc module contains a collection of hash, random and distribution functions that are used to create reproducable but pseudo random behaviour.

5.4.2 Particle Movement

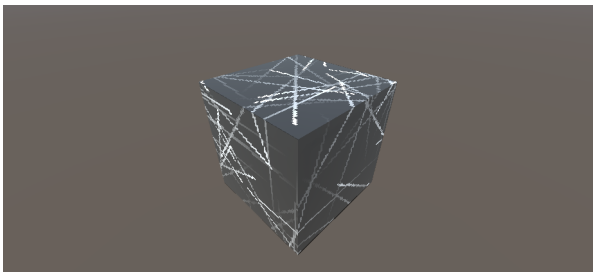

Figure 5.5: Particles Crossing Edges

The particle move shader needs to be executed for every particle. So for the vertex / fragment shader version, the vertex shader creates one big primitive that covers the whole viewport. This way one fragment is created for each pixel of the renderbuffer, and the main calculations are done in the fragment shader. This is the situation where it certainly makes sense to replace

the vertex / fragment shader combo with a compute shader. Also since we decided to use a StructuredBuffer for storing the particles in Unity, we definitely have to use a compute shader because a StructuredBuffer can't be rendered to.

Using the common functions for buffer access and triangle uv normalization the basic particle move shader, without particle transfer, boiles down to the following steps:

```
// read
t_particle p = _Particles[i_particle];

// adjust orientation
float d_left   =
    read_sensor(p.triangle_id, p.triangle_uv, p.orientation + _SensorAngle)
    + gaussian_random(0u+i_particle)*_NoiseWeight;
float d_middle =
    read_sensor(p.triangle_id, p.triangle_uv, p.orientation               )
    + gaussian_random(1u+i_particle)*_NoiseWeight;
float d_right  =
    read_sensor(p.triangle_id, p.triangle_uv, p.orientation - _SensorAngle)
    + gaussian_random(2u+i_particle)*_NoiseWeight;

if ((d_middle < d_left) || (d_middle < d_right)) {
    // using sign was suggested by MM
    p.orientation += sign(d_left - d_right) * _RotationSpeed;
}

// angle -> direction
float2 direction = mul(orthogonal_to_triangle_uv(p.triangle_id),angle_to_dir(p.orient

// step
p.triangle_uv += _ParticleSpeed * direction;

// normalize
normalize_location(p.triangle_id, p.triangle_uv, direction);

// direction -> angle
p.orientation = atan2(mul(triangle_uv_to_orthogonal(p.triangle_id), direction));

// writeback
_Particles[i_particle] = p;
```

5.4.3 Trail Deposition

Conceptually the trail deposition is really straightforward. For each particle the current position is read, the corresponding texture uv position is calculated and a predefined amount is added to the trail texture at that pixel.

One slightly annoying bug was that gl_PointSize is not initialized to the same value on all platforms. But what actually makes this interesting is, that the effect of multiple particles at the same location

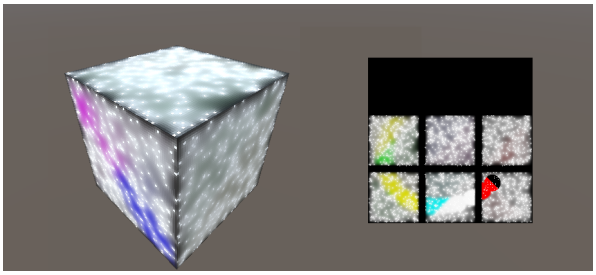

Figure 5.6: Deposition and basic Diffusion

is supposed to accumulate.

With the rendering pipeline a blend mode can be configured to add fragments onto the existing value in the framebuffer instead of replacing it. In webgl we had to add a dependency on the extension EXT_float_blend to allow for blending on a float framebuffer, but this seems to be well supported.

In Unity, since particle movement was already done as a compute shader, deposition was too. The InterlockedAdd function was used to achieve an atomic increment. Since InterlockedAdd only works for integers we changed to counting the number of particles on one channel of the trailmap and doing the diffusion on another channel.

Using a seperate channel to count particles could be done for the graphics shaders too, if it turns out to have additional advantages. But for now there didn't seem to be any benefit in backporting that to the webgl version.

5.4.4 Trail Diffusion

The trail diffusion is one shader that actually uses the vertex / fragment shader model. Only the pixels of the texture that are used for a triangle need to be updated. This is easiest done by using the vertex shader with the triangles to create fragments everywhere a triangle is mapped to. The vertex shader could also be used to calculate intermediate values that are the same for all pixels of the same triangle, like how to translate positions that extend into a neighbouring triangle. But that would require some adjusting of the common functions for coordinate translation.

The diffusion itself is done in the fragment shader, by blending the previous value with an average for the nine neighbouring pixels and applying a decay. When using a seperate particle count channel, new traces are added accordingly.

To allow diffusion across triangle borders the position for the surrounding samples, is first expressed in triangle uv coordinates, normalized, and then translated into texture uv coordinates. A screen space derivative is used to calculate the triangle uv delta corresponding to a one pixel offset.

5.4.5 Visualization

The visualization is kept really simple for now. Using the standart model-view-projection matrices a 3D view of the object[s] is rendered.

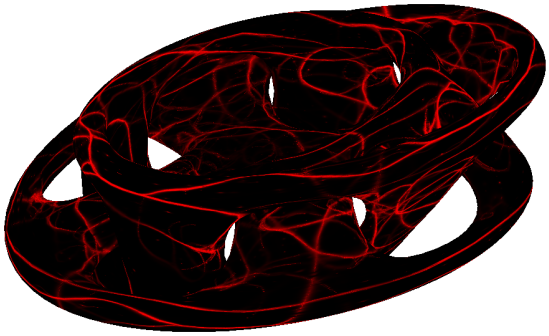

Figure 5.7: Torus without Shading

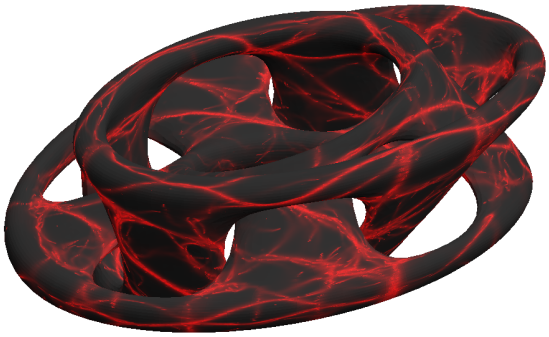

Figure 5.8: Torus with Shading

In webgl, the view matrix is fixed as there are no camera controls yet. The draw call is instanced to be able to load the mesh data from texture. And simple shading using calculated surface normals and one global directional light is added to make it easier to tell the shape of the object. (see Figure 5.7, Figure 5.8) Clamping the triangle uv values before calculating texture uv values reduces visible borders like the ones on the first cube in Figure 5.1.

In Unity visualization is done using a modified standart shader. The only change being that it uses the correct set of uv coordinates instead of UV0.

5.5 Signed Distance Fields

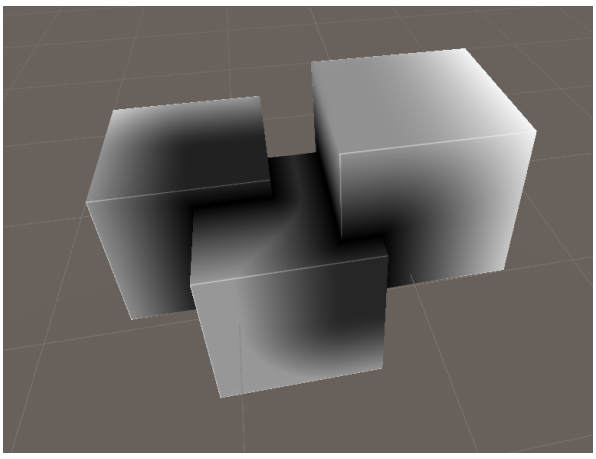

Figure 5.9: Signed Distance Fields

A signed distance field is used to express the distance between any arbitrary point in space and a given object. For our simulation we only need the values of the signed distance fields on the surface of the objects. Since calculating the distance values is not trivial, it made sense to store them in a texture. This way they only have to be recomputed, when the objects move. The same uv values as for the trailmap are used.

For simple objects like cubes or spheres, it is possible to come up with closed formulas. Combining those to construct bigger objects works fine for positive, but not for negative distances, which

is acceptable for many applications. But for the physarum simulation we are mainly interested in the negative distances.

So instead a shader was written to calculate the signed distance from any point on one object to a given second object. The mesh data for the first object is passed using the index and vertex buffer bound to the draw call. The mesh data for the second object is accessed with the custom mesh access functions. Using the vertex shader, the triangles of the first object are laid out in uv space and fragments covering every point on its surface are created.

For each fragment, the signed distance is determined by looping through all triangles of the second mesh and taking the distance with the lowest absolute value. The distance to a triangle is calculated by choosing the nearest point on the plane that the triangle is on and projecting it onto the edges if it is ouside the area of the triangle, then measuring the absolute distance to that point. The sign is added depending on whether the fragment is facing the front or the backside of the triangle.

So the triangle that's nearest to the fragment also decides whether the fragment is inside or outside the object. This works well in most cases, but there is a case for objects with sharp edges, where the nearest point can be on an edge where a triangle from the front meets a triangle from the back of the object. So we have to add a rule that if multiple triangles have the same absolute distance, if any one of them is facing the fragment with its front the fragment has to be considered outside the object (positive distance) and only if all them face the fragment with their backs the fragment can be inside (negative distance).

When there are more than two objects in the simulation the signed distance field on each object has to consider the distance to multiple other objects. This can be done by running the shader once for every other object and introducing blending to the results. Relevant for the simulation is the object with the most overlap, so the signed distance with the lowest value (highest negative value) should be kept.

Since at first it seemed like writing to the depth buffer from the fragment shader didn't work, a depth test was implemented by reading the previous distance from the texture and returning the minimum of the new distance and the old one. This was not without problems either. As it turns out, at least on my system, in contrast to integer textures, float textures can not be sampled from and rendered to at the same time. So optional buffering using a second texture was implemented. Then later after some other changes, using the built in depth testing turned out to work after all. The solution had probably something to do with changing ZTest from Less to LEqual in addition to enabling ZWrite, but it is unclear how that effected it exactly. For the inspector a conditionally grayed out script property was used to show that applying the buffer fix is only available/necessary when using texture based depth testing.

When working with multiple objects, to be able to transfer the particles to the correct target, it is important to not only keep track of which triangle but also of with object the distance was measured to. This means in addition to the signed distance, the SDF texture also stores a mesh and triangle id, which have to be kept consistent with the ones used for the simulation. To make the particle transfer faster it was also considered to store information about the triangle uv of the exact nearest point on the triangle and angles or transform matrices. But seen as the particle transfers only happen along very thin lines, it doesn't seem efficient to calculate and store those values for everywhere on the surface. This of cause also depends on how static the objects in the simulation are and how often the sdf texture needs to be recomputed.

By sampling the resulting SDF texture, particles can be made to avoid areas with negative distance.

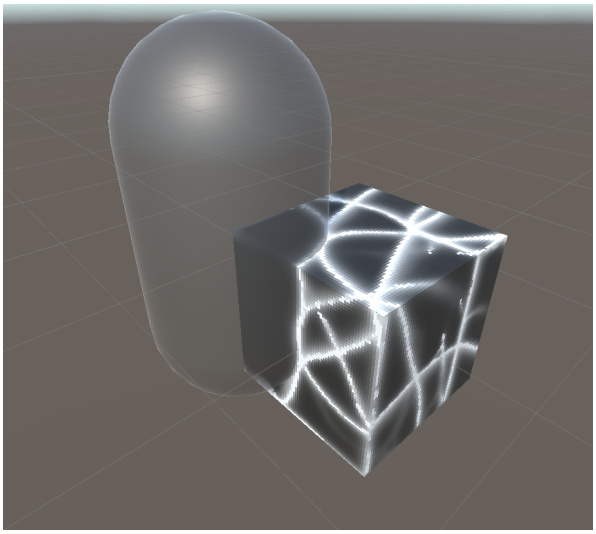

Figure 5.10: Particles using SDF to avoid Covered Area

5.5.1 Depth Value Mapping

When using the built in depth testing of the render pipeline it is not possible to use the signed distance value directly as the depth value.

Most graphicscards nowadays support integer and float depth values up to 32bit. The type of the depth channel is part of the texture declaration but it does not have to match the type of the result set in the fragment shader. Typically the fragment shader returns a fixed or float between 0 and 1 (clipspace) which is then mapped to [minint...maxint] if the texture is configured for integer depth values. [6] Even when using a texture with float depth values, the values still have to be in the clipspace range.

So we needed to find a way to distinctly map the signed distance values which can be any floating point value to the range of 0 to 1, while keeping the order relation. In mathematical terms, we need a function f that satifies the following conditions:

$$f : [-inf, inf] \rightarrow [0, 1]$$

$$\begin{aligned} a < b &\implies f(a) < f(b) \\ a = b &\implies f(a) = f(b) \\ a > b &\implies f(a) > f(b) \end{aligned}$$

Since we are working with discrete number representations, the first problem to encouter is that there are more values between -inf and inf than between 0 and 1. So we have to loosen our conditions and accept an ambiguous assignment:

$$\begin{aligned} a < b &\implies f(a) \leq f(b) \\ a > b &\implies f(a) \geq f(b) \end{aligned}$$

Now due to the way floating point values are formatted the [0, 1] range makes up just under a quarter of all floating point values. So in average between 4 and 5 input values have to share the same output value. This is a loss of 2 to 3 of 24 bits in precision. But how they are actually distributed depends on the chosen function.

The default mapping deployed by the graphics pipeline uses a multiplicative inverse (1/z) that is scaled using values derived from the near and far clipping planes. This works well for the area between the clipping planes but completely fails outside.

```
// default mapping for clipping space using near and far clipping planes
(1.0 - d * _ZBufferParams.w) / (d * _ZBufferParams.z);
```

A logistic function may appear as a valid candidate at first. But it approaches the limits so fast that with the discretization to floating point numbers it already completely flats out for values of d outside [-709, 36].

```
// mapping using logistic function
1/(1+exp(-d));
```

A function that has a similar shape but approaches the limits comparatively slowly is arctan. The following function completely flats out for values of d outside of about $\pm 2^{52}$.

```
// mapping using atan
(1/3.141592653589793) * atan(d) + 0.5;
```

But not only values with a high absolute value, also values with a small absolute value are hard to represent. Floating point numbers have their highest density around 0 and can get as close as 2^{-149}. At 0.5 the floating point numbers are way less dense, so all values in the range of about $[-2^{-54}, 2^{-53}]$ are mapped to exactly 0.5. And this are only the absolute limits of what can be represented. Way before an increasing number of input values are already mapped to the same output value.

That being said, the arctan is probably completely sufficient for what we need in our implementation. However using bit level manipulation it is possible to create a mapping with an almost perfectly even distributed precision over the complete range of floating point values, dropping at most 3 significant bits.

```
// mapping using bit structure of float32 as defined in IEEE754
uint bits = asuint(d);
bits ^= (bits & 0x80000000) ? 0xFFFFFFFF : 0x80000000;
bits = (bits >> 2);
return asfloat(bits) / 2.0;
```

Figure 5.11 shows how by rearranging the different ranges in the floating point format and with a bitshift, all floating point numbers can be mapped to the range [0, 2[with only loosing exactly 2 bits of precision for every possible value. The division by 2 at the end is done as a normal floating point operation so the hardware can take care of the special cases and possibly necessary conversion from normalized to denormalized format. For normalized numbers a division by 2 is just a decrease by 1 in the exponent, for denormalized numbers it is another bitshift. Therefore each input value looses 2 or 3 bits in the process, meaning always 4 or 8 successive input values share the same output value.

For all functions nan should probably be mapped to nan again, which can be done with a surrounding if, but also the signed distance should not be nan in the first place.

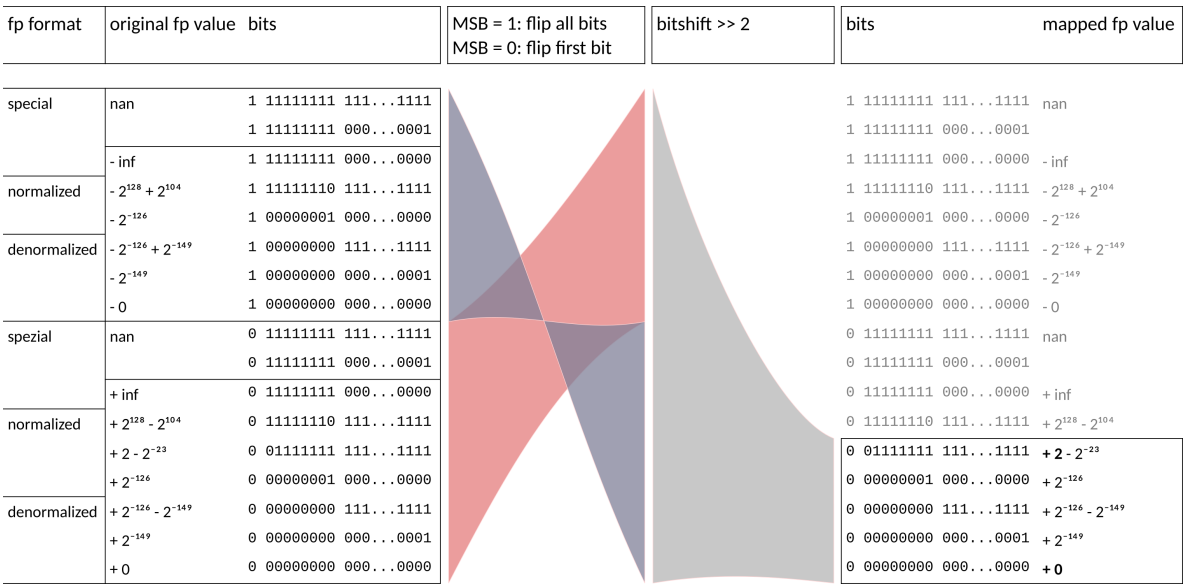

Figure 5.11: Mapping Floating Point Values from [-inf:+inf] to [0:2[

All of the shown functions were implemented and the last one is actively used. In comparision it can be said that:

- Clipping is a field-tested method, easy to implement and works well in supported range but totally fails outside.
- Logistic function takes a lot to compute and quickly reaches numerical limits.
- Atan is way more numerically stable than the logistic function, and significantly faster due to better hardware support.
- Bitshift has the most uniform coverage of the theoretical input range, and is in theory way easier to compute than logistic or atan. Although on some hardware atan only uses a single cycle, so the bitshift function may actually be slightly slower. Also it is specific to the floating point format and therefore less universal.

5.6 Particle Transfer

5.6.1 Particle Selection

The test for selecting particles that need to be transfered to another object is implemented in the particle move shader:

```
// check if inside covered area
float sdf_distance_now = read_sdf(triangle_id_now, triangle_uv_now).r;
if (sdf_distance_now < 0) {
    // check if heading into or just leaving
    float sdf_distance_then = read_sdf(triangle_id_then, triangle_uv_then).r;
    if (sdf_distance_now < sdf_distance_then) {
        // add particle to particles that need to be transfered
```

```
            _ParticleIndices.Append(i_particle);
        }
}
```

Here a stack is used to accumulated the indices of particles that need to be moved. Using this array of indices, the following kernels can be run with the minimal number of necessary thread groups.

5.6.2 Transfer Strategies

Five transfer strategies have been implemented and can be selected in the inspector.

- No Transfer - There is no transfer of particles between objects. Currently they are disabled and vanish.
- Respawn - There is no transfer of particles between objects. Instead they are randomly respawned on the same object using the same rules as for the initial placement. This is implemented on the GPU and can be slow if many particles need to be respawned.
- No Defragmentation - Particles are transfered to other objects, but keep their place in memory. So particles from anywhere in the buffer can be on any mesh.
- Sequential Defragmentation - Particles are transfered to other objects. When this happens they are also moved to keep the particles of the same mesh in the same buffer window. This happens on the GPU but with a single thread.
- Parallel Defragmentation - Same as Sequential Defragmentation, but parallelized.

5.6.3 Buffers

When running the simulation for multiple objects a single buffer is used for all the particles. All previous particle shaders still need to be calculated seperately for each object to have the correct meshes and textures bound, but this way no expensive buffer resizing has to be done, when particles move from one mesh to another as the total number of particles stays the same.

Now we can either run the particle shaders for each mesh on all particles and just ignore the ones that are not part of the current mesh (No Defragmentation Strategy) or we can run them on flexible sized subsets of the particle buffer, if we make sure that all particles that are on the same mesh are next to each other in the buffer. (Defragmentation Strategy)

Since particles could go from any one to any other object and shaders only allow to bind a fixed number of particles, the transfer has to be done in two parts. One that has access to the mesh data of the first object and one that has access to the mesh data of the second object. In between the position and orientation of the particle are exchanged in world space and to store that information two Transfer buffers are used.

The ParticleIndices Buffer was already shown in the Particle Selection section. It is used to select which particles following shader kernels should be run for.

A TranferIndices Buffer is used for Parallel Defragmentation.

5.6.4 Transfer Out

In the transfer_out shader transfers are created for all the previously selected particles. Target mesh and triangle are read from the sdf and stored in the transfer. Position, direction and surface normal are translated into world space and stored too. Then the transfer gets added to the TransferOut buffer.

The TransferOut buffer becomes the TransferIn buffer for the next mesh. The TransferIn buffer should be empty at this point and becomes the new TransferOut buffer.

5.6.5 Transfer In

The transfer_in shader is the first shader run for each mesh. For every transfer it first looks at the target mesh. If the transfers destination is not the current mesh it just gets copied over to the TransferOut buffer.

Otherwise the attached information is used to calculate the new position and trajectory for the particle on the target triangle as described in the methods chapter.

Afterwards the TransferIn buffer is cleared, by setting the associated buffer count to zero.

5.6.6 Defragmentation

The ParticleBuffer is used as a ringbuffer, where the windows for each mesh can wrap around. The Defragmentation happens after the transfer_out of one object and the transfer_in of the next object. It makes sure that all particles that are currently in transfer are passed on to the next mesh.

For this to work all particles that stay on the mesh have to come first, and the particles that leave the mesh are copied to the end of the meshs buffer window. Then the border of the buffer windows is moved so that the particles are considered inside the buffer window of the next mesh.

Sequential Defragmentation

The size of the TransferBuffer can be used to tell how much the buffer window border will be moved. All particles that need to be moved out will have an entry in the TransferBuffer, but some of them may already be on the right side of the new buffer window border. And all particles in the reassigned area that should stay with the mesh need to be copied in front of the border.

The Sequential Defragmentation uses a single thread to build pairs of particles by stepping through both lists of candidates until it finds the next qualifying one, then swapping them, until it reaches the end. The number of those particles can vary but they will always match.

Parallel Defragmentation

The Parallel Defragmentation uses multiple threads to first check for candidates and then swap particles in parallel. This is done in two shader calls.

The defragmentation_parallel_scan builds the two lists of particles that need to be swapped. The indices of particles that should stay with the mesh but are in the reassigned area are pushed to the ParticleIndices buffer. The indices of transfers where the corresponding particle is not yet in the reassigned area are pushed to the TransferIndices buffer.

The defragmentation_parallel_swap takes the particles from both lists for each index and swaps them.

In both defragmentation versions the transfers need to be updated with the new index of their particle.

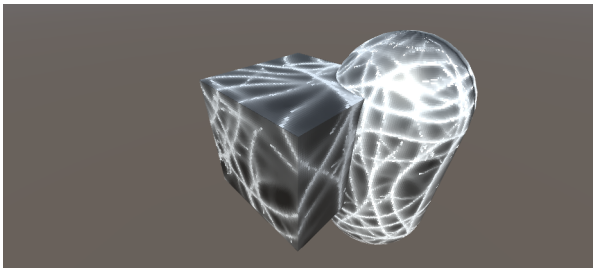

Figure 5.12: Particle Transfer

6 Results and Evaluation

6.1 WebGL Solution

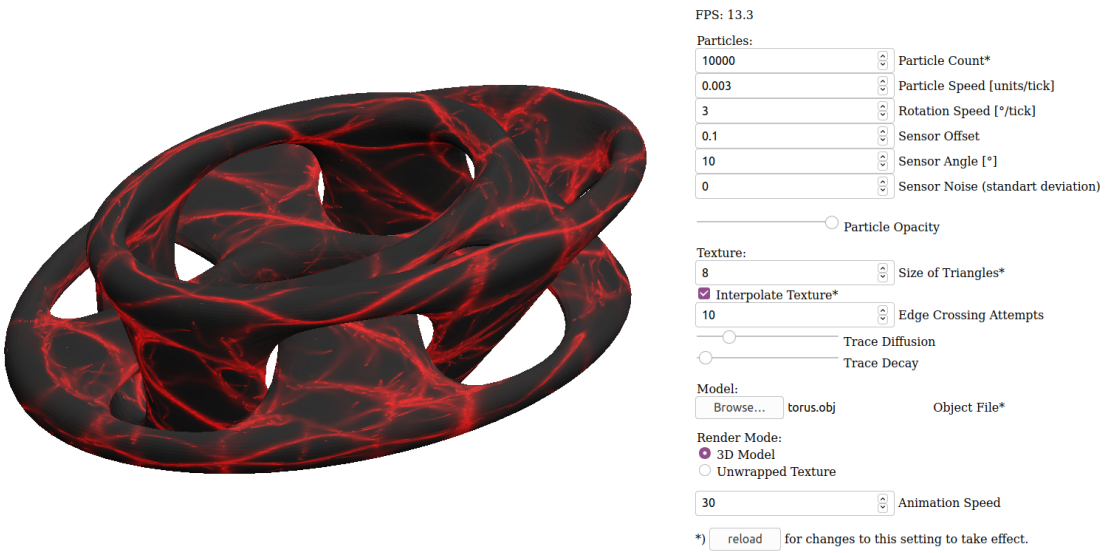

Figure 6.1: WebGL UI

The WebGL Implementation was written in HTML, JavaScript and GLSL using the WebGL 2.0 API. It was tested to run in Firefox, Opera and Chrome on Linux, Windows, Android and Mac. There is no camera control, but loaded objects are automatically scaled to fit the viewport. Meshes can be loaded from obj files containing triangles or quads. A simple grid based UV mapping is applied automatically. The uv mapping can be examined by switching the render mode to show the unwrapped texture. The animation speed is bound to the fps and can be limited using the Animation Speed setting. Most of the parameters can be adjusted at runtime, but some changes need a restart of the simulation to take effect. These are marked with an asterisk.

The following parameters are available to adjust the behaviour of the simulation:

- **Particle Count** - The total number of particles in the simulation.

- Particle Speed - The distance a particle travels in one simulation step. For reference, the object is scaled to have a radius of 1.
- Rotation Speed - The angle in degrees that a particle can change it's direction per simulation step.
- Sensor Offset - The offset from the particle to the points on the texture that is samples to decide its next step.
- Sensor Angle - The angle between the direction the particle is heading and the right or left sensor.
- Sensor Noise - The standart deviation of a gaussian / normal distributed noise, that is added to the reading of each sensor.
- Particle Opacity - The opacity that is used for blending the newly deposited trails onto the trailmap texture.
- Size of Triangles - The cell size for the grid based uv mapping in pixels.
- Edge Crossing Attempts - The maximum number of triangle to triangle transformations that will be done for one triangle uv normalization. If set too low particles that crossed two many borders in the same simulation step will temporarily be at an invalid location. If on average the particle does not have to cross more borders per step than the set number of edge crossing attempts, the normalization will be completed in a following step.
- Trace Diffusion - The blending factor between the previous value of a trailmap pixel and the average of its surrounding pixels. The diffusion always propagates one pixel per simulation step. This can be used to control how the decay over time relates to the decay over distance.
- Trace Decay - A multiplicative factor for how much of the trails remains from one simulation step to the next.

6.2 Unity Solution

The Unity Implementation is structured in a modular way that aligns with the concepts set by the framework. Positioning objects and providing them with meshes is a native part of the engine. Scripts, shaders and textures are used to add the simulation. This makes it easy to integrate the physarum simulation into other unity projects.

Multiple objects that are part of the same simulation are nested inside a common parent object. The parent has a Script component that coordinates the simulation for the multiple objects. Figure 6.2 It is used to attach the shaders, show debug information, control the simulation, and also contains the following global settings:

- Transfer Buffer Size - The size of the Transfer buffer determines how many particles can be moved between textures at the same time. A runtime error will be thrown if this is set too low.
- Transfer Strategy - The strategy for transfering particles as explained in the implementation chapter.

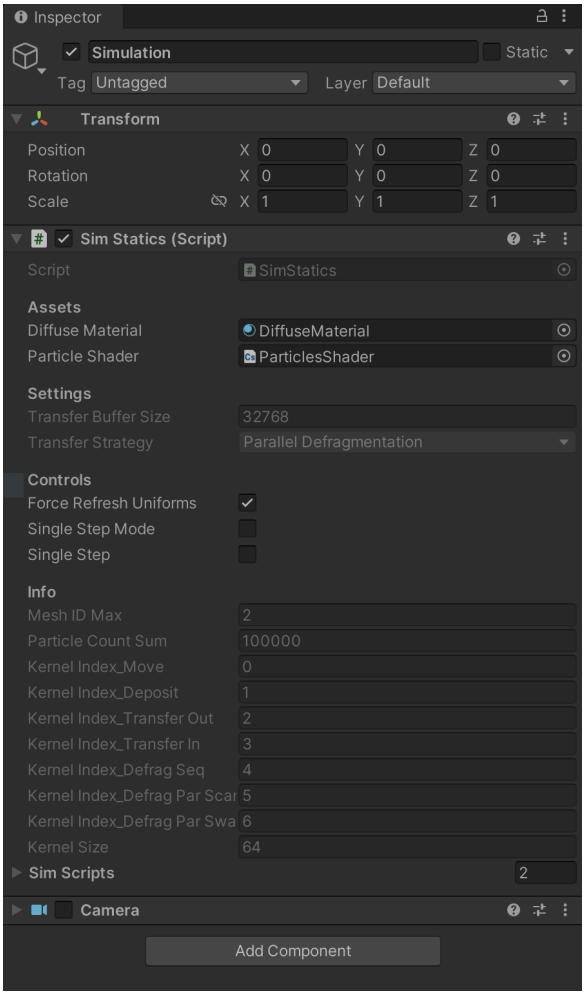

Figure 6.2: Global Simulation Parameters

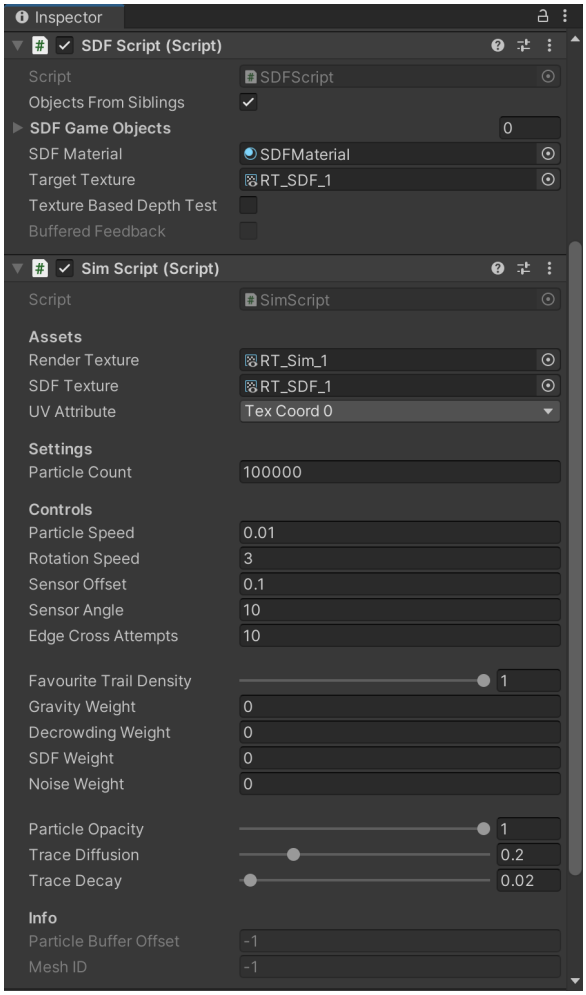

Figure 6.3: SDF and Simulation Parameters

The forceRefreshUniforms option makes sure that changes to parameters are immediately uploaded to the uniforms while the simulation is running.

Each child object has two scripts attached. Figure 6.3 The SDF Script for generating and updating a sdf texture, and the Sim Script responsible for the actual simulation.

The SDF Script can be configured to calculate the SDF to a specific list of objects or all siblings. A material and target texture have to be attached. If there are problems with using the builtin depthtest of the graphics shaders, a texture based depth test can be enabled. And when using the texture based depth test there is also the option to enable buffering for better compatibility with float textures. Both of this comes with a performance cost.

For the simulation in addition to the parameters already present on the WebGL Solution the following additional parameters are available:

- **Favourite Trail Density** - Instead of prefering the direction with the highest sensor reading the particles will prefer a trail density that is closest to this value.
- **Gravity Weight** - A factor for increasing the attractivenes of locations based on their height in world space.

- Decrowding Weight - A factor for decreasing the attractivenes of locations with many particles at that exact location. This uses the particle count channel of the trail texture.
- SDF Weight - A factor for increasing or decreasing the attractivenes of locations based on how close they are to other objects.

It also worked with the VR template and a Valve Index Headset. But for an interactive VR experience there should be actual ingame parameter controls not just the inspector, and also grabbing and moving stuff etc.

6.3 Limitations of the Method

One important limitation of our method has to be addressed.

There is no index to find particles based on their position. This makes efficient collision detection or remeshing impossible. So particles can only interact with each other indirectly using the texture, and the mesh topology has to be static. (No adding / removing of triangles).

However, the positions of vertices can change, so moving objects or deforming parts of an object by using skinned meshes should be possible. An application for this would be tracking the fingers of a hand in a VR context.

6.4 Performance

The two implementations have different feature sets, so the following results can not be taken for directly comparing the used technologies. But it can tell us what kind of performance we can expect for different combinations of parameters, hardware and technology.

The testing was done on a Razer Blade Late 2016 laptop with 16GB RAM and an Intel Corei7-6700HQ processor. In addition to the integrated Intel(R) HD Graphics 530 GPU of the laptop an external Thunderbolt3 GPU enclosure with an NVIDIA GeForce GTX 980 was used. The external display connected to the nvidia GPU was 1080p 50Hz.

The WebGL solution was run on firefox 103.0.2 (64bit). And for the Unity solution Unity 2021.2.3f1 was used. Everything was done on a Windows 10 operating system. The Unity Solution was tested in the editor. Compilation would probably increase the performance, but wasn't feasable as the inspector is needed for the parameter controls.

A set of parameters that worked well for all tested szenarios was using 500000 particles and a texture size of 1024x1024. Figure 6.4 shows how the solutions compare at these parameters.

Unsurprisingly using a smaller model and faster GPU increased the performance and vice versa. Interestingly the webgl implementation performs better in both cases using the intel GPU. On the nvidia GPU the unity implementation continues to scale, while the webgl implementation is locked at 50 fps. As this is exactly the refresh rate of the connected monitor this most likely means that the browser is using some kind of vsync by default.

As a side note, with the same parameters a Samsung Galaxy S10e (Exynos 9 9820 Processor, ARM-Mali G76 MP12 GPU) managed to run the webgl implementation with 12fps.

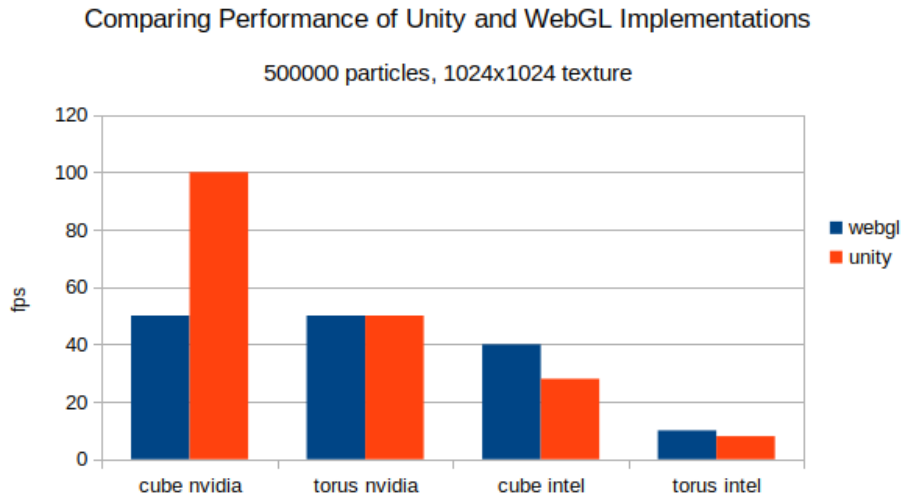

Figure 6.4: Performance Comparison of Unity and WebGL Implementations

Another thing to look at is how the same setup performs for different problem sizes.

There are hard limits for the maximum particle count and texture size for both implementations. The webgl implementation has a javascript issue that currently prevents it from working with more than about 500000 particles, and webgl throws an error when trying to create a texture above 2GB so the biggest successfully used texturesize was 16384x16384. The unity implementation works up to 8000000 particles before reaching the maximum thread group count. And Unity started to randomly crash when using textures above 4096x4096.

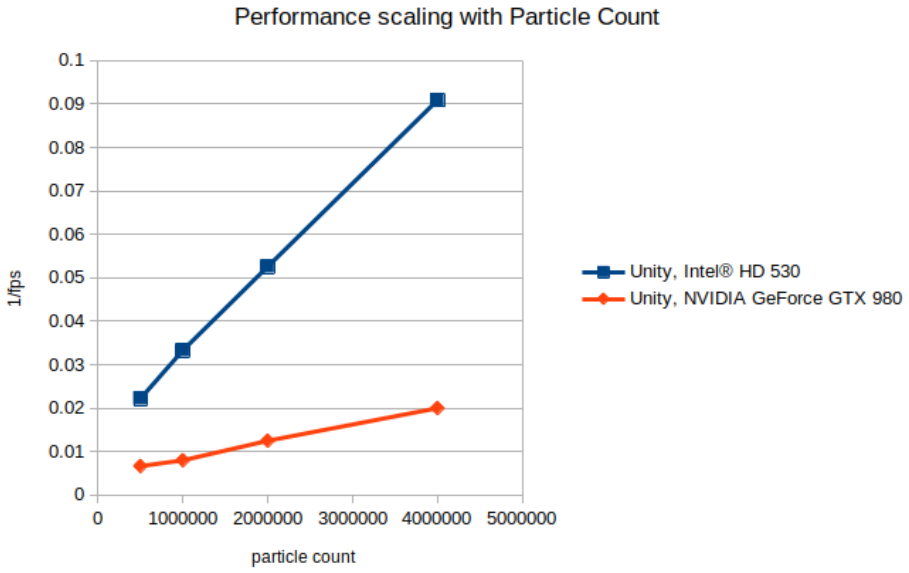

Figure 6.5: Performance scaling with Particle Count

Because of the low particle limit on webgl only the Unity Implementation was used for Figure 6.5. The Graph shows a linear relation between the particle count and the amount of time needed to compute one frame.

Figure 6.6 shows the performance at different texture sizes. It seems like for small texture sizes there are other factors dominating the performance. Then for larger textures the performance decreases. There does not seem to be a clear linear relation between the number of pixels and the time it takes to compute a frame. A linear relation would be expected for the diffusion. But many other parts of the system are influenced by the texture size too, like for example the rendering. So this graph will show a superposition of those seperate effects. Timing the different parts of the simulation seperately may be required to get a deeper understanding.

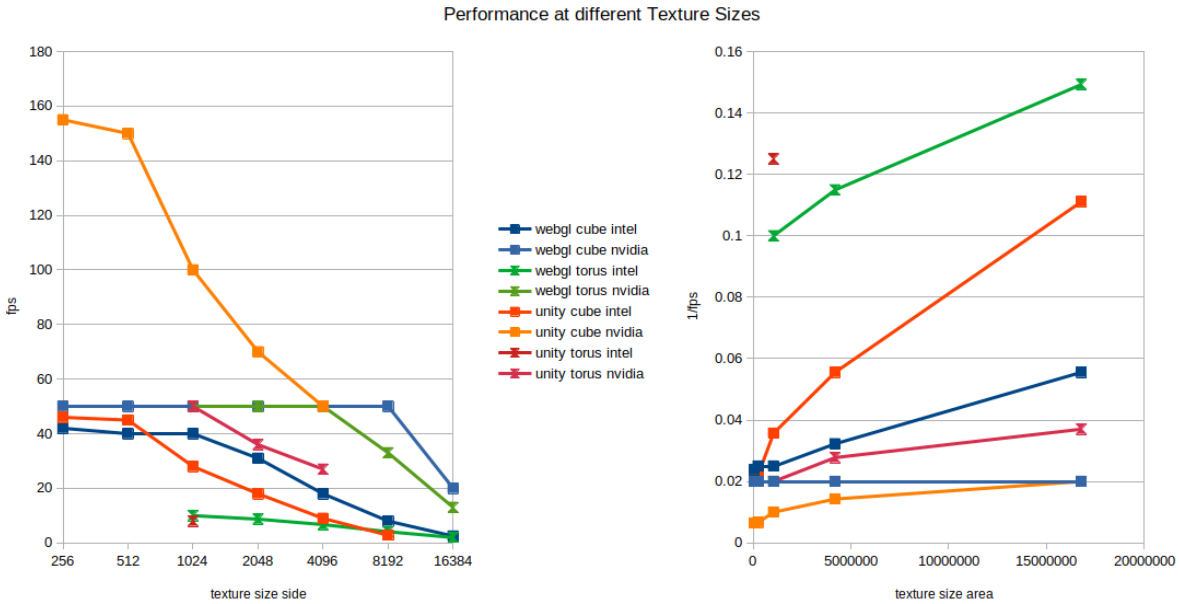

Figure 6.6: Performance at different Texture Sizes

Not shown in the graphs is the effect of changing the edge crossing attempts. All of them use a value of 10 edge crossing attempts. Changing this gives a speedup that is heavily dependend on the number and size of triangles in the object. For the cube model there is almost no effect, while on the torus model lowering this from 10 to 1 gives a speed up of up to 2.5x. But it also limits the maximum usable particle speed.

Further tests have to be done for the simulation with multiple object. In particular on how the sdf calculation and different particle transfer strategies scale with the number of objects and the frequency of their movements. For now it can be said that there was no visible timing difference between the defragmentation strategies when only using 2 or 3 meshes.

7 Conclusion and Future Work

We have successfully implemented a physarum model on the surface of multiple objects in a dynamic 3D environment using the proposed method for particle simulations on 2D manifolds.

The Physarum Model shows the characteristical highway formations but left on its own it tends to collapse into few rings after a while, only to some degree restricted by the topology of the object, like the holes in the torus. This reinforces the need for ways to interact with the particles. The implementation created for this work is the first step on the path to the proposed construction of a Virtual Reality Physarum Laboratory. The implementations show acceptable performance, but may need further optimization to match the framerates required for virtual reality devices, when used within a room scale environment.

Work that could be done as part of creating this Virtual Reality Physarum Laboratory in the future includes:

Interactions

- Implementing support for skinned meshes would allow to run the simulation on the tracked hands or even full body of the user.
- Think about how distributed simulation could work in a multi user environment. The Simulation is deterministic in theory but also chaotic in nature.
- Live calculated lightmaps and shadows could be used to interact with particles.
- Add a way to paint on the trail texture.
- In VR it should be possible to move and rotate objects with motion tracked controllers.
- Create the option to add markers and create an SDF like "distance to next marker" channel.
- A food inspired reward system could incentivise particles to visit certain spots. How recently particles have eaten could also influence the composition of their traces, their speed, or their behaviour.
- Physics simulation could be explored in combination with the simulation. For instance there could be a marble track, where the marble moves particles around, blocks their paths, etc.

Visual Effects

- Using the trail texture to render glow or a bump map instead of just the surface color could make the simulation more visually interesting.
- Using a scanned model of a real room for a VR environment could make the simulation feel more real / increase immersion.
- More elaborate initialisation options for the particles can create interesting patterns at the beginning of the simulation. For example when all particles start at one point.

Technical Features

- Adding ways to handle open edges in meshes would make it easier to try the simulation on new meshes.
- If an object does not have a suitable uv mapping for the simulation, there may be a way to automatically trigger the create uv for lightmap on import option.
- Implement uint32 index format for mesh access.
- Find a way to have SDF recognize self intersection of meshes.
- Implement a camera controls for webgl.
- Add parameter to adjust the distance to where the neighbours for diffusion are sampled.

Optimizations

- Investigate if screen space derivatives can be used to reduce the number of texture lookups for diffusion.
- Consider the use of mipmapping to enable diffusion to propagate faster / further per tick.
- Surface normals can be used to reduce the number of triangles required to approximate curved surfaces.
- SDF and simulation can be run on different timings. This would allow to slow down sdf refresh rate.
- Try going back to the vertex/fragment shader model for trail deposition.
- Precalculate factors for translation matrices and store them in a buffer. This is already set up to some degree in the Manifold.cginc file but was not completely implemented. This would definitely improve performance and change not much else.
- Use the vertex shader to precalculate translation matrixes for sampling across edges for diffusion.
- Only consider objects with overlapping bounding boxes when calculating SDF.
- Store orientation of particles as vector, since storage limitation was mostly related to webgl. This gets rid of a few conversions and allows to have a variable velocity if needed.

Bug Fixes

- For the initial particle distribution consider the area of triangles for correctly weighted selection.
- The diffusion leaks black pixels on some uv borders. This is a result of the clamp_triangle_uv not being implemented completely. The problem to figure out is how to the scaling from uv coordinates to pixel sizes in that function.
- The transfer strategy NoTransfer should disable particle selection, to stop them from vanishing.
- Fix javascript issue that stops webgl version from using more then about 500000 particles. This should be as easy as not using a function call with spread syntax.
- The forceRefreshUniforms option is more of a workaround. With custom editor panels it should be possible to have a callback an only update the Uniforms on actual changes.

Bibliography

[1] sonic art. *Torus 3D*. URL: https://www.turbosquid.com/3d-models/shape-sculpture-art-3d-1595712.

[2] Dean Jackson and Jeff Gilbert. *WebGL 2.0 Specification*. URL: https://registry.khronos.org/webgl/specs/latest/2.0/.

[3] Sage Jenson. *Physarum*. Feb. 2019. URL: https://sagejenson.com/physarum.

[4] Jeff Jones. "Characteristics of pattern formation and evolution in approximations of physarum transport networks". In: *Artificial Life* 16 (2 2010), pp. 127–153. ISSN: 1064-5462. DOI: 10.1162/artl.2010.16.2.16202.

[5] Aras Pranckevičius. *Unity 21.2 Mesh API Compute Shader Access*. URL: https://docs.google.com/document/d/1_YrJafo9_ZsFm4-8K2QlD0k3RgwZ_49tSA84paobfcY.

[6] Nathan Reed. *Depth Precision Visualized*. URL: https://developer.nvidia.com/content/depth-precision-visualized.

[7] Keijiro Takahashi. *NoiseBall6*. URL: https://github.com/keijiro/NoiseBall6.

[8] Wikipedia. *Affine Transformations*. URL: https://en.wikipedia.org/wiki/Transformation_matrix#Affine_transformations.

[9] Wikipedia. *Autonomous Agent*. URL: https://en.wikipedia.org/wiki/Autonomous_agent.

[10] Wikipedia. *Computational fluid dynamics*. URL: https://en.wikipedia.org/wiki/Computational_fluid_dynamics.

[11] Wikipedia. *Discrete Element Method*. URL: https://en.wikipedia.org/wiki/Discrete_element_method.

[12] Wikipedia. *Euclidean Space*. URL: https://en.wikipedia.org/wiki/Euclidean_space.

[13] Wikipedia. *Finite Difference Method*. URL: https://en.wikipedia.org/wiki/Finite_difference_method.

[14] Wikipedia. *Finite Element Method*. URL: https://en.wikipedia.org/wiki/Finite_element_method.

[15] Wikipedia. *Homogeneous Coordinates*. URL: https://en.wikipedia.org/wiki/Homogeneous_coordinates#Use_in_computer_graphics_and_computer_vision.

[16] Wikipedia. *Manifold*. URL: https://en.wikipedia.org/wiki/Manifold.

[17] Wikipedia. *Molecular Dynamics*. URL: https://en.wikipedia.org/wiki/Molecular_dynamics.

[18] Wikipedia. *Self-propelled Particles*. URL: https://en.wikipedia.org/wiki/Self-propelled_particles.

[19] Eric Winsberg. "Computer Simulations in Science". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2019. Metaphysics Research Lab, Stanford University, 2019.