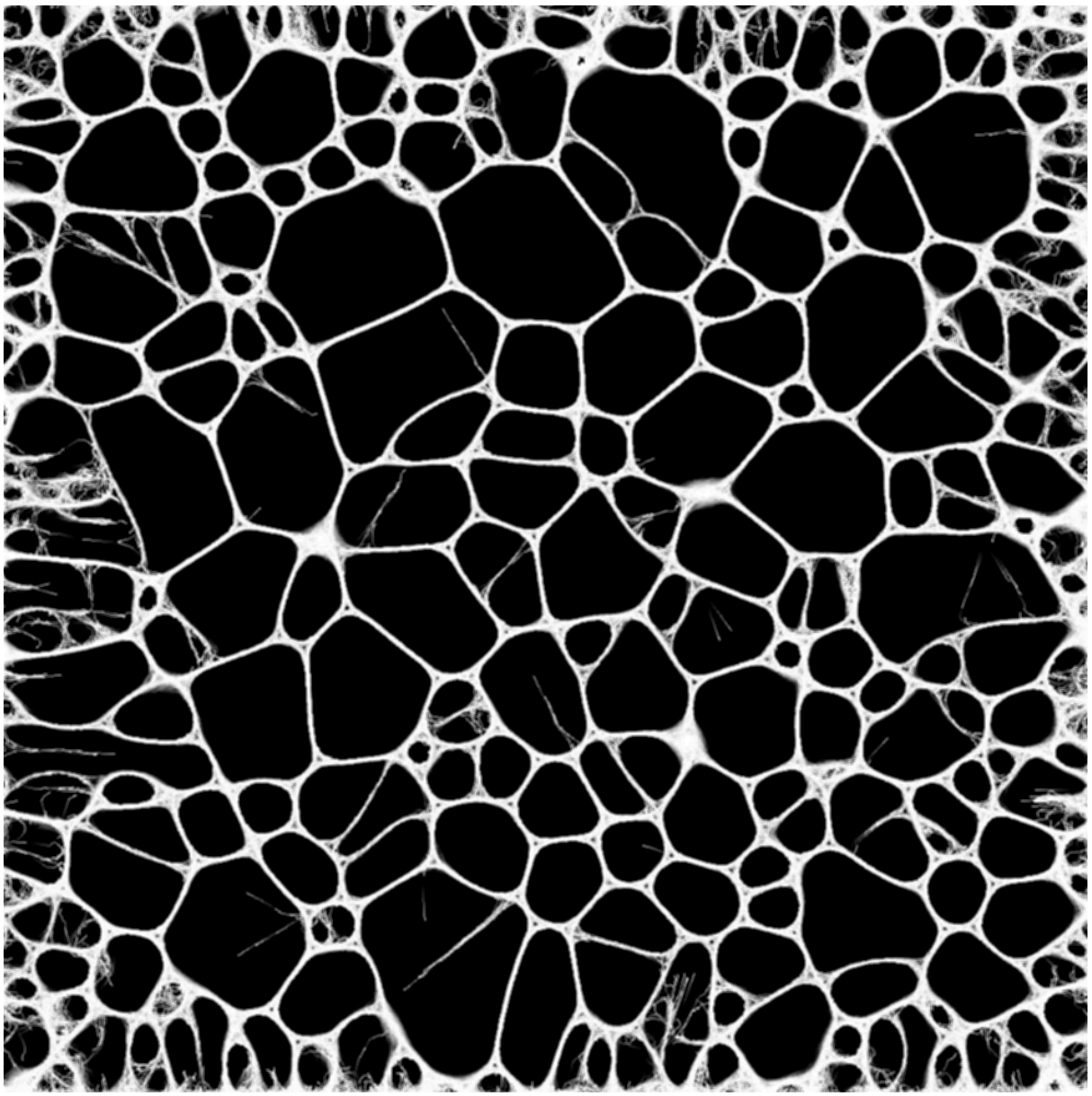


Physarum Simulation

IT-GFS vom 09.01.2023

Finn Agethen, TGM12.1

Inhaltsverzeichnis

1. Einführung in die C-Programmiersprache

Die C-Programmiersprache wurde in den 1970er Jahren von Dennis Ritchie entwickelt und ist seitdem eine der am weitesten verbreiteten Programmiersprachen. Abgesehen von Pointern und Memory Management zählt C als eine einsteigerfreundliche Programmiersprache. Dies liegt daran, dass C im Gegensatz zu C++ und anderen modernen Programmiersprachen fast keine abstrakten Datentypen hat. Im Folgenden findet sich eine kleine Einführung von den Konzepten, die zum Verstehen des Codes der Physarum-Simulation erforderlich sind. Eine weitaus ausführlichere Einführung findet man unter dem link: https://www.tutorialspoint.com/cprogramming/index.htm.

Programmstruktur

```c
#include <stdio.h>

int main(){
      printf("Hello, World!\n");
      return 0;
}
```

Die erste Zeile mit dem Befehl “#include” gibt an, dass die Standard-Eingabe-/Ausgabe-Bibliothek (stdio.h) in das Programm eingebunden wird. Bei Bedarf können auch mit dem “#define”-Befehl konstante, globale Variablen deklariert werden. Die “main”-Funktion ist der Einstiegspunkt jedes C-Programms. Die “printf”-Funktion wird aus der stdio-Bibliothek importiert und gibt den Text "Hello, World!" in der Konsole aus. Die “return 0”-Anweisung gibt an, dass das Programm erfolgreich beendet wurde.

Variablen

```c
int ganzZahl = 42;
float kommaZahl = 3.57;
double präzisereKommaZahl = 3.5742;
char zeichen = 'a';
```

Es gibt verschiedene Variablentypen. In diesem Beispiel gibt es:

- “**int**” (Integer) für ganze Zahlen
- “**float**” (floating point number) für reelle Zahlen
- “**double**” für reelle Zahlen mit doppelter Präzision
- “**char**” (character) für Buchstaben und andere Zeichen

Operatoren

In C gibt es verschiedene Arten von Operatoren, die für verschiedene Arten von Operationen verwendet werden können. Hier ist eine Liste einiger der häufigsten Operatoren in C und eine kurze Erklärung ihrer Verwendung:

- Arithmetische Operatoren:
  - "**+**": Addition
  - "**-**": Subtraktion
  - "*****": Multiplikation
  - "**/**": Division
  - "**%**": Modulo (Gibt den Rest einer Division zurück)
- Vergleichsoperatoren:
  - "**==**": Gleichheit
  - "**!=**": Ungleichheit
  - "**>**": Größer als
  - "**<**": Kleiner als
  - "**>=**": Größer als oder gleich
  - "**<=**": Kleiner als oder gleich
- Logische Operatoren:
  - "**&&**": Logisches UND
  - "**||**": Logisches ODER
  - "**!**": Logisches NICHT
- Zuweisungsoperatoren:
  - "**=**": Zuweisung
  - "**+=**": Addition und Zuweisung
  - "**-=**": Subtraktion und Zuweisung
- Inkrement- und Dekrement-Operatoren:
  - "**++**": Inkrement (Erhöht den Wert um 1)
  - "**--**": Decrement (Verringert den Wert um 1)

If-Statements

```c
if (num > 0) {
    printf("Die Zahl ist positiv.\n");
} else if (num < 0) {
    printf("Die Zahl ist negativ.\n");
} else {
    printf("Die Zahl ist 0.\n");
}
```

Durch If-Statements können Entscheidungen getroffen werden. Wenn die Behauptung in den Klammern wahr ist, wird der Code innerhalb der geschweiften Klammern aufgeführt und danach bis zum Ende des If-Statements gesprungen.

In diesem Beispiel wird überprüft, ob “**num**” größer als 0 ist. Wenn ja, wird **"Die Zahl ist positiv."** ausgegeben. Wenn “**num**” kleiner als 0 ist, wird **"Die Zahl ist negativ."** ausgegeben. Andernfalls wird **"Die Zahl ist 0."** ausgegeben.

Loops

Es gibt verschiedene Arten von Schleifen (loops): for, while, do-while.
In diesen Beispielen wird die for- und die while-Schleife verwendet.

For-loops werden in C verwendet, um eine Schleife für eine bestimmte Anzahl von Iterationen auszuführen.

```
int i;
for (i = 0; i <= 10; i++){
      printf("%d\n", i);
}
```

In diesem Beispiel wird die Schleife von 0 bis 10 iteriert und jeder Wert in der Schleife wird in der Konsole ausgegeben.

While-loops werden in C verwendet, um eine Schleife solange auszuführen, bis eine bestimmte Bedingung erfüllt ist.

```
int num = 5;
while (num > 0) {
    printf("%d\n", num);
    num--;
}
```

In diesem Beispiel wird die Schleife solange ausgeführt, bis “**num**” kleiner oder gleich 0 ist. Jeder Wert von “**num**” wird in der Schleife ausgegeben und” **num**” wird um 1 verringert.

Structs

Structs in C sind Datenstrukturen, die aus verschiedenen Datenfeldern bestehen, die unter einem gemeinsamen Namen zusammengefasst werden. Structs werden häufig verwendet, um komplexe Daten zu speichern und zu organisieren.

```c
struct Position{
      int x;
      int y;
};

struct Position pos;
pos.x = 5;
pos.y = 10;
```

In diesem Beispiel wurde ein Struct namens “**Position**” definiert. Dieser neue Datentyp setzt sich aus zwei Integern zusammen. Durch den Punkt-Operator (“**.**”) kann auf die einzelnen Variablen zugegriffen werden.
Nachdem der neue Datentyp definiert wurde, wird in diesem Beispiel eine Variable names “**pos**” erstellt und ihre Datenfelder mit Werten initialisiert. Der “**x**”-Wert wird auf 5 und der “**y**”-Wert auf 10 gesetzt.

Funktionen

```c
int add(int a, int b){
      int c = a + b;
      return c;
}

int zahl1 = 5;
int zahl2 = 13;

int ergebnis;
ergebnis = add(zahl1, zahl2);

printf("Ergebnis: %d\n", ergebnis);
```

Wenn eine Funktion aufgerufen wird, wird der in ihr stehende Codeblock ausgeführt. Eine Funktion kann auch, wie in diesem Beispiel, Parameter verlangen, welche in der Funktion gebraucht werden. Hier werden in die Funktion zwei Integer als Parameter gegeben. In der Funktion werden diese Zahlen addiert und das Ergebnis

wird von der Funktion mit dem “**return**”-Keyword zurückgegeben. Dieser Wert wird von der Variable “**ergebnis**” abgefangen und anschließend ausgegeben.

2. Was ist OpenGL?[1, 2, 3]

“OpenGL ist eine [...] Anwendungsprogrammierschnittstelle zum Rendern von 2D- und 3D-Vektorgrafiken.”[1]
Mithilfe der Programmiersprache glsl (GL Shading Language), welche ein Teil von OpenGl und im Grunde eine Vereinfachung der C-Programmiersprache mit Vektoren ist, können Programme auf dem Grafikprozessor mit drastisch verbesserter Performance ausgeführt werden. Diese Programme nennt man Shader. Diese Shader können nicht nur Rechnungen durchführen, sondern auch Grafiken rendern.

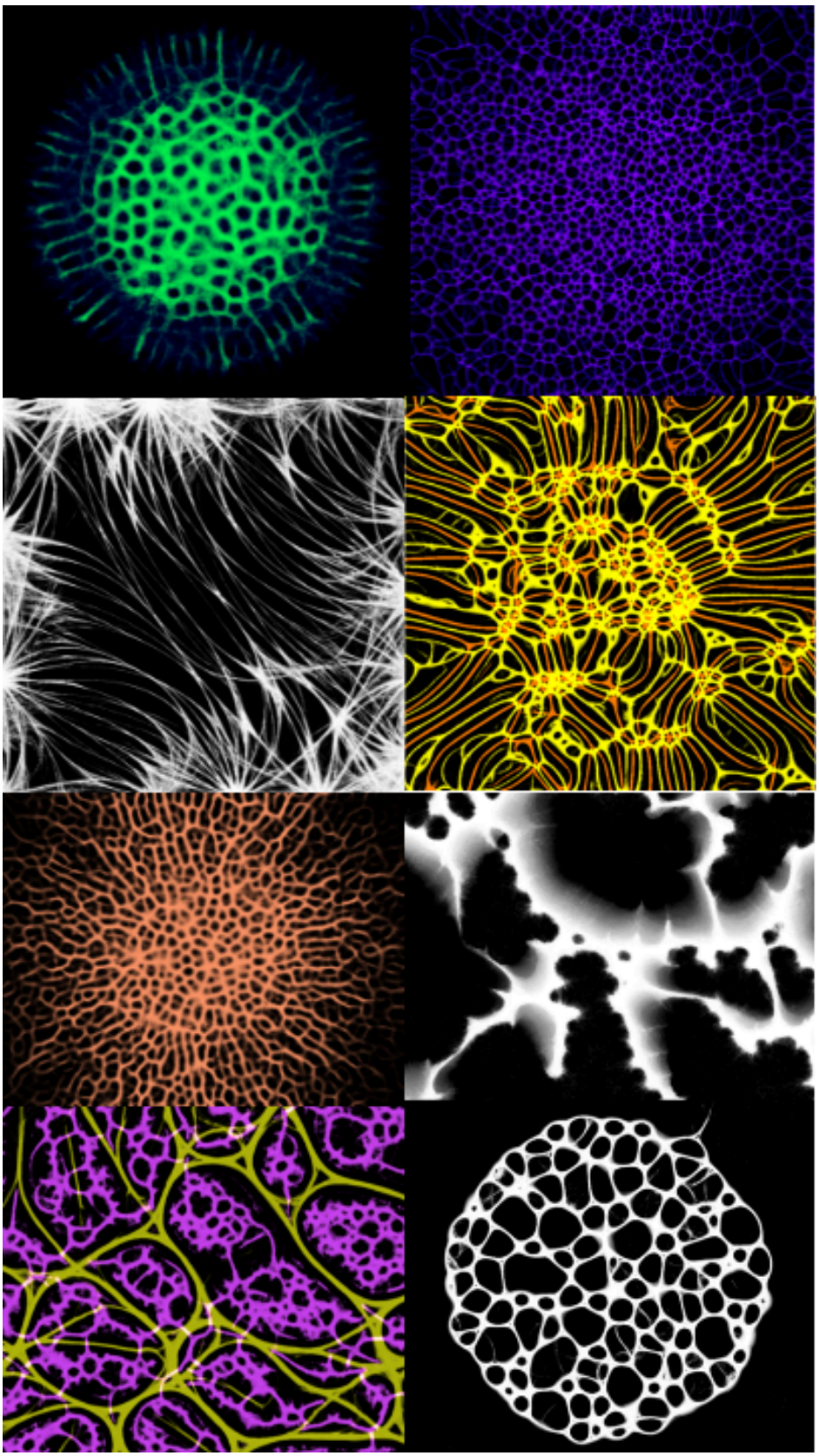
Mit den reinen OpenGL Funktionen werden meist nur die Formen bestimmt und mit den Shadern kommuniziert bzw. ausgeführt.
Die einfachste Konfiguration von Shadern, um Grafiken zu rendern, besteht aus einem Vertex-Shader und einem Fragment-Shader. Ein weiterer häufig verwendeter Shader ist der Compute-Shader, welcher nur zu schnellen Berechnungen auf dem Grafikprozessor verwendet wird.
OpenGL bietet aber noch viel mehr, welches man unter den Quellen 2 und 3 nachlesen kann.

3. Physarum[4, 5, 6, 7]

Physarum polycephalum ist eine Gattung der Schleimpilze, der nachgewiesen[4] wurde, dass sie sich auch ohne Gehirn an die Position von Nährstoffquellen erinnern und sogar optimale Pfade zu dieser finden kann.
Dieses Verhalten wurde in Simulationen und Experimenten untersucht und könnte möglicherweise Anwendungen in der Logistik, im Transportwesen und in anderen Bereichen haben. Es wurden auch bereits Versuche von Forschern durchgeführt, die zeigen, wie ein Schleimpilz ein effizientes Eisenbahnsystem in Tokyo entwirft[5].

Es wurden ebenfalls Forschungen zu den verschiedenen Mustern unter verschiedenen Bedingungen, die der Schleim annimmt, betrieben. In der Studie[6] des Herausgebers “MIT Press“ wurde eine Simulation des Schleims beschrieben und an welche Regeln diese sich halten muss. Die Simulation hier orientiert sich an den Regeln dieser Studie.
Im Internet gibt es schon zahlreiche Simulationen von Schleimen, nicht nur von Forschern, sondern auch von Künstlern wie zum Beispiel Georgios Cherouvim[7].

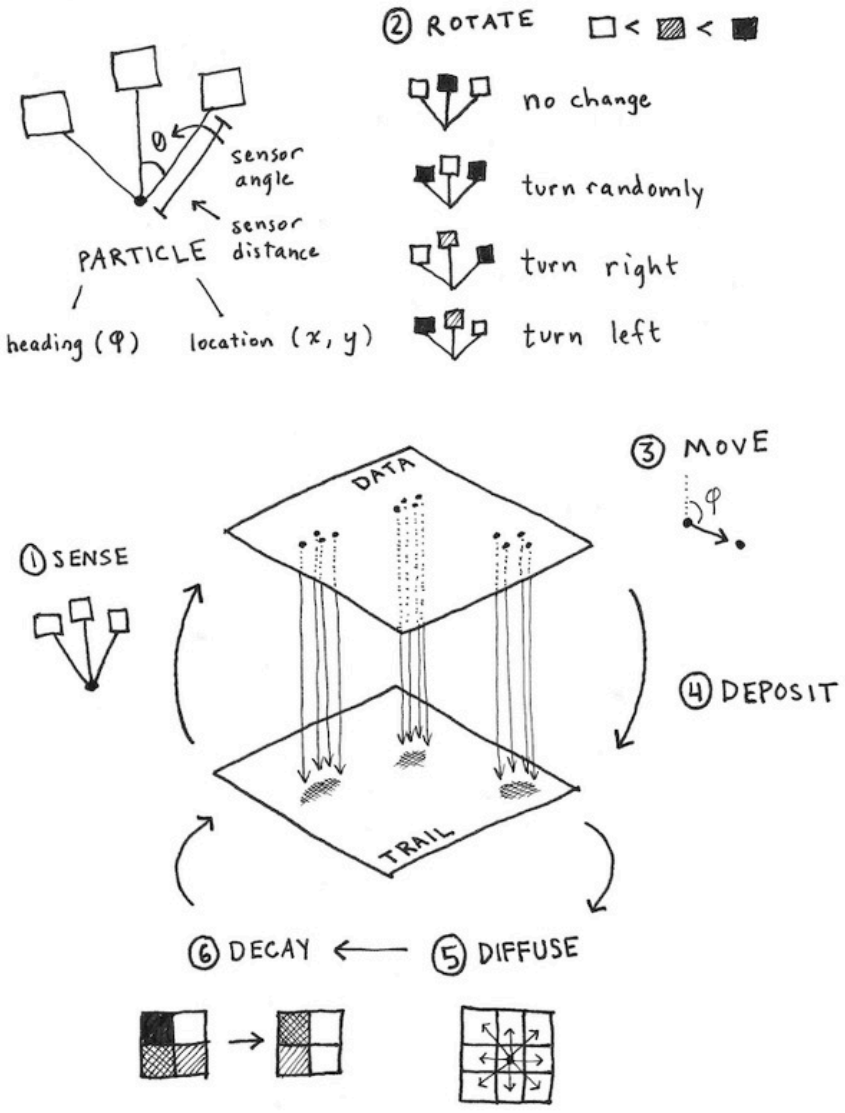
4. Simulationsbeispiele



Die Bilder wurden mit unterschiedlichen Einstellungen und unterschiedlicher Simulationsdauer aufgenommen.

5. Erklärung des Simulation-Codes[8]

Diese Simulation ist in der Lage bis zu drei verschiedene Schleim-Spezies mit unterschiedlichen Eigenschaften zu simulieren. Diese verschiedenen Spezies versuchen sich ebenfalls zu meiden.
Zu Beginn der Simulation werden die Partikel in einer vorher angegebenen Form generiert. Der Standort sowie die Richtung der Partikel werden in einer passenden Datenstruktur gespeichert und im Laufe der Zeit angepasst. Ein Partikel bzw. Agent hat drei Sensoren, einen in einem bestimmten Abstand vor ihm und auf beiden Seiten von diesem Sensor in einem bestimmten Winkel einen weiteren. Wenn der Agent sich bewegt, hinterlässt er eine Spur, welcher andere Agents folgen. Diese Spur wird in einer anderen Datenstruktur gespeichert und verblasst nach bestimmter Zeit.



Algorithmus-Bild[8]

Im obigen Bild ist der Algorithmus der Physarum-Simulation veranschaulicht.
Im ersten Schritt hält der Agent mit seinen Sensoren nach anderen Agents bzw. ihrer Spur Ausschau. Seine Richtung verändert sich im nächsten Schritt, je nachdem welcher Sensor die höchste Aktivierung hat.
Im dritten Schritt bewegt sich der Agent in seine neue Richtung und platziert im vierten Schritt seine Spur.
Im fünften Schritt verbreitet sich diese Spur auf benachbarte Felder.
Der letzte Schritt lässt die Spur des Agents ein wenig verblassen, bis sie eventuell verschwindet.
Diese Schritte wiederholen sich, solange die Simulation anhält.

In den folgenden Auszügen handelt es sich lediglich um den Simulations-Teil des Programms, welcher in glsl geschrieben wurde. Es handelt sich hierbei um einen Compute- und Fragment- Shader.

```
struct Agent{
      float x, y, angle;
      int speciesIdx; // 3 species supported
};

struct SpeciesSettings{
      int spawnMode;
      float sensorSize,
            sensorOffsetDistance,
            sensorAngle,
            turnSpeed,
            moveSpeed,
            r, g, b;
};

struct SimulationSettings{
      float agents,
            s1inp, s2inp, s3inp,
            fps, fpsoff,
            avoid,
            blurRadius,
            trailWeight,
            diffuseWeight,
            decayRate;
};
```

```
layout(binding = 1, rgba32f) uniform image2D trailMap;
layout(binding = 2, std430) buffer agents{
	Agent dataMap[];
};
layout(binding = 3, std430) buffer speciesSettings{
	SpeciesSettings settings[];
};
layout(binding = 4, std430) buffer simulationSettings{
	SimulationSettings simSettings;
};
```

Ganz oben im Programm definieren wir drei Datentypen. Zwei, in denen wir die Einstellungen für die Simulation und die verschiedenen Schleim-Spezies deklarieren und einen, der die Informationen der Agents deklariert. Zu diesen Informationen gehören die Position, die Richtung als Radian und die zugehörige Spezies.
Anschließend wurden vier Variablen deklariert, welche verwendet werden, um Daten von der Anwendung an den Shader selbst zu übergeben.
Diese Daten können dann über “**trailMap**”, “**dataMap**”, “**settings**”, “**simSettings**” abgegriffen werden.
Die Variable “**trailMap**” speichert alle Spuren der Agents.
Die Variable “**dataMap**” speichert alle Informationen zu den Agents.
Die Variablen “**settings**” und “**simSettings**” speichern die Einstellung für die Schleim-Spezies und die Simulation.

```
void main(){
	ivec2 id = ivec2(gl_GlobalInvocationID.xy);

	// Update Agents
	Agent agent = dataMap[id.x];
	SpeciesSettings config = settings[agent.speciesIdx];

	vec3 speciesMask = vec3(
		(agent.speciesIdx == 0) ? 1.0 : 0.0,
		(agent.speciesIdx == 1) ? 1.0 : 0.0,
		(agent.speciesIdx == 2) ? 1.0 : 0.0
	);

	// Stear based on the sensors
	float sensorAngleRad = config.sensorAngle * (PI / 180.0);

	float weightForward = sense(agent, speciesMask, 0.0);
	float weightLeft = sense(agent, speciesMask, sensorAngleRad);
```

```
float weightRight = sense(agent, speciesMask, -sensorAngleRad);

uint random = hash(int(agent.y) * imgSize.x + int(agent.x) +
                   hash(id.x + time));
float randomSteerStrength = scaleToRange01(random);

float turnSpeed = config.turnSpeed * 2 * PI;

// Do nothing
if (weightForward > weightLeft && weightForward > weightRight){
     dataMap[id.x].angle += 0;
}
// Left or right randomly
else if (weightForward < weightLeft && weightForward < weightRight){
     dataMap[id.x].angle += (randomSteerStrength - 0.5) *
                            2.0 * turnSpeed;
}
// Right
else if (weightRight > weightLeft){
     dataMap[id.x].angle -= randomSteerStrength * turnSpeed;
}
// Left
else if (weightLeft > weightRight){
     dataMap[id.x].angle += randomSteerStrength * turnSpeed;
}

// Update positions
vec2 direction = vec2(cos(agent.angle), sin(agent.angle));
vec2 newPos = vec2(agent.x + direction.x * config.moveSpeed,
                   agent.y + direction.y * config.moveSpeed);

// Bounce of wall
if (newPos.x < 0.0 || newPos.x >= imgSize.x ||
    newPos.y < 0.0 || newPos.y >= imgSize.y){
     random = hash(random);
     float randomAngle = scaleToRange01(random) * 2 * PI;

     newPos.x = min(imgSize.x - 1.0, max(0.0, newPos.x));
     newPos.y = min(imgSize.y - 1.0, max(0.0, newPos.y));

     dataMap[id.x].angle = randomAngle;
}
```

```
    else {
        // Leave a trail
        vec3 oldTrail = imageLoad(trailMap, ivec2(newPos)).rgb;
        imageStore(trailMap, ivec2(newPos), vec4(min(vec3(1.0),
                    oldTrail + speciesMask *
                    simSettings.trailWeight), 1.0));
    }

    dataMap[id.x].x = newPos.x;
    dataMap[id.x].y = newPos.y;
}
```

In der “**main**”-Funktion wird jeder Agent einzeln aktualisiert. Zuerst wird der zu aktualisierende Agent mit seinen Informationen in der Variable “**agent**” gespeichert und eine “**speciesMask**” für ihn erstellt, welche später zur Identifizierung der zugehörigen Spezies wichtig ist. Anschließend wird die Aktivierung der drei Sensoren in der später beschriebenen “**sense**”-Funktion berechnet und gespeichert. Danach wird die Richtung des Agents anhand des folgenden Verfahrens verändert:

- Ist die Aktivierung des Sensors in der Mitte am stärksten, wird die Richtung nicht verändert.
- Ist der Wert in der Mitte kleiner als die Aktivierung links und rechs, wird wird die Richtung in einem bestimmten Grad zufällig verändert. (Dieser zufällige Wert wurde zuvor mit einer Hash-Funktion generiert und für diesen Anwendungszweck mit einer weiteren Funktion auf das Intervall [0, 1] zugeschnitten)
- Ist die Aktivierung rechts stärker als links, wird die Richtung nach rechts verändert und vice versa.

Wenn die Richtung bestimmt wurde, wird als nächstes die neue Position berechnet. Falls diese außerhalb des Simulationsbereichs fallen würde, wird sie innerhalb der Grenzen gesetzt und eine komplett neue Richtung zufällig bestimmt.
Wenn die neue Position feststeht, wird die alte mit der Aktuellen überschrieben.

```
float sense(Agent agent, vec3 speciesMask, float sensorAngleOffset){
      float sensorAngle = agent.angle + sensorAngleOffset;
      vec2 sensorDir = vec2(cos(sensorAngle), sin(sensorAngle));

      ivec2 sensorCenter = ivec2(vec2(agent.x, agent.y) + sensorDir *
                        settings[agent.speciesIdx].sensorOffsetDistance);

      float sum = 0.0;
      int sensorSize = int(settings[agent.speciesIdx].sensorSize);
      for (int offsetX = -sensorSize; offsetX <= sensorSize; offsetX++){
            for (int offsetY = -sensorSize; offsetY <= sensorSize;
                offsetY++){
                  int sampleX = int(min(imgSize.x - 1.0,
                                        max(0.0, sensorCenter.x + offsetX)));
                  int sampleY = int(min(imgSize.y - 1.0,
                                        max(0.0, sensorCenter.y + offsetY)));

                  sum += dot(speciesMask * 2 - 1,
                       imageLoad(trailMap, ivec2(sampleX, sampleY)).rgb);
            }
      }
      return sum;
}
```

In der “**sense**”-Funktion geben wir einen Wert zurück, der die Stärke der Spuren im Radius des Sensors wiedergibt. Dazu übergeben wir den Agent, die zugehörige Spezies und den Versetzungswinkel des Sensors in die Funktion.

In der Funktion wird zuerst mithilfe geometrischer Funktionen und Vektorrechnung die Mitte des Sensors bestimmt. Anschließend werden die Werte aller Pixel innerhalb des Sensor Radiuses addiert. Diese Werte sind, wenn von derselben Spezies positiv und andernfalls negativ. Die Summe der Werte wird zum Schluss von der Funktion zurückgegeben.

```
void diffuse(ivec2 coord){
      vec3 sum = vec3(0.0);
      vec3 originalCol = imageLoad(trailMap, coord).rgb;

      int radius = int(simSettings.blurRadius);
      // Blur
      for (int offsetX = -radius; offsetX <= radius; offsetX++){
            for (int offsetY = -radius; offsetY <= radius;
                offsetY++){
                 int sampleX = int(min(imgSize.x - 1.0,
                                       max(0.0, coord.x + offsetX)));
                 int sampleY = int(min(imgSize.y - 1.0,
                                       max(0.0, coord.y + offsetY)));
                 sum += imageLoad(trailMap,
                                  ivec2(sampleX, sampleY)).rgb;
            }
      }

       vec3 blurredCol = sum / float((radius * 2 + 1) *
                                     (radius * 2 + 1));
      blurredCol = originalCol * (1.0 - simSettings.diffuseWeight)
                   + blurredCol * simSettings.diffuseWeight;

      imageStore(trailMap, coord.xy,
                 vec4(max(vec3(0.0),
                 blurredCol - simSettings.decayRate), 1.0));
}
```

Die “**diffuse**”-Funktion führt den fünften und den sechsten Schritt des Algorithmuses aus. Um das Verbreiten der Spur zu simulieren, bekommt jeder Pixel den Durchschnittswert der benachbarten Pixel.
Ähnlich wie in der “**sense**”-Funktion werden hierzu alle Werte der Pixel in einem bestimmten Radius addiert. Danach wird der Durchschnitt berechnet und dem Pixel mit den übergebenen Koordinaten im Parameter “**coord**” zugeschrieben. Bevor aber dies passiert, wird der neu bestimmte Wert noch mit der am Anfang gespeicherten Originalfarbe “**orininalCol**” verrechnet, um mehr Kontrolle über das verschwommene Bild zu haben.
Am Ende wird noch der sechste Schritt ausgeführt, indem ein bestimmter Verblassungs-Wert von dem Pixel abgezogen wird, um das Verblassen der Spur nachzuahmen.

6. Mögliche Erweiterungen[8, 9, 10]

Dieses Projekt ist keinesfalls abgeschlossen, sondern kann ganz im Gegenteil noch erweitert werden. Zum Beispiel könnten Nahrungsquellen, natürliche Selektion bzw. Evolution hinzugefügt oder die Geschwindigkeit des Schleims anhand der Dichte verändert werden.
Eine weitere Idee von Sage Jenson[8] wäre es, ansteckendes Verhalten zu programmieren, wobei die Eigenschaften der Agents verändert werden, wenn sie sich an infizierten Partikeln anstecken. Andere spielerische Erweiterungen wären ein Bild- oder Video-Modus, indem die Partikel das Bild formen, wie in der Version von KHY[9]. Natürlich könnte man die Simulation auch in einem 3D Umfeld programmieren, wie es unter anderem der User AtwoodDeng[10] auf Github gemacht hat.

7. Quellen

1. https://en.wikipedia.org/wiki/OpenGL
2. http://www.opengl-tutorial.org/
3. https://learnopengl.com/
4. https://www.pnas.org/doi/10.1073/pnas.2007815118
5. https://www.youtube.com/watch?v=GwKuFREOgmo
6. https://uwe-repository.worktribe.com/output/980579
7. https://ch3.gr/sugarscape-and-slime-mold/
8. https://cargocollective.com/sagejenson/physarum
9. https://www.youtube.com/watch?v=rW9ZsO6LYdk
10. https://github.com/AtwoodDeng/Physarum