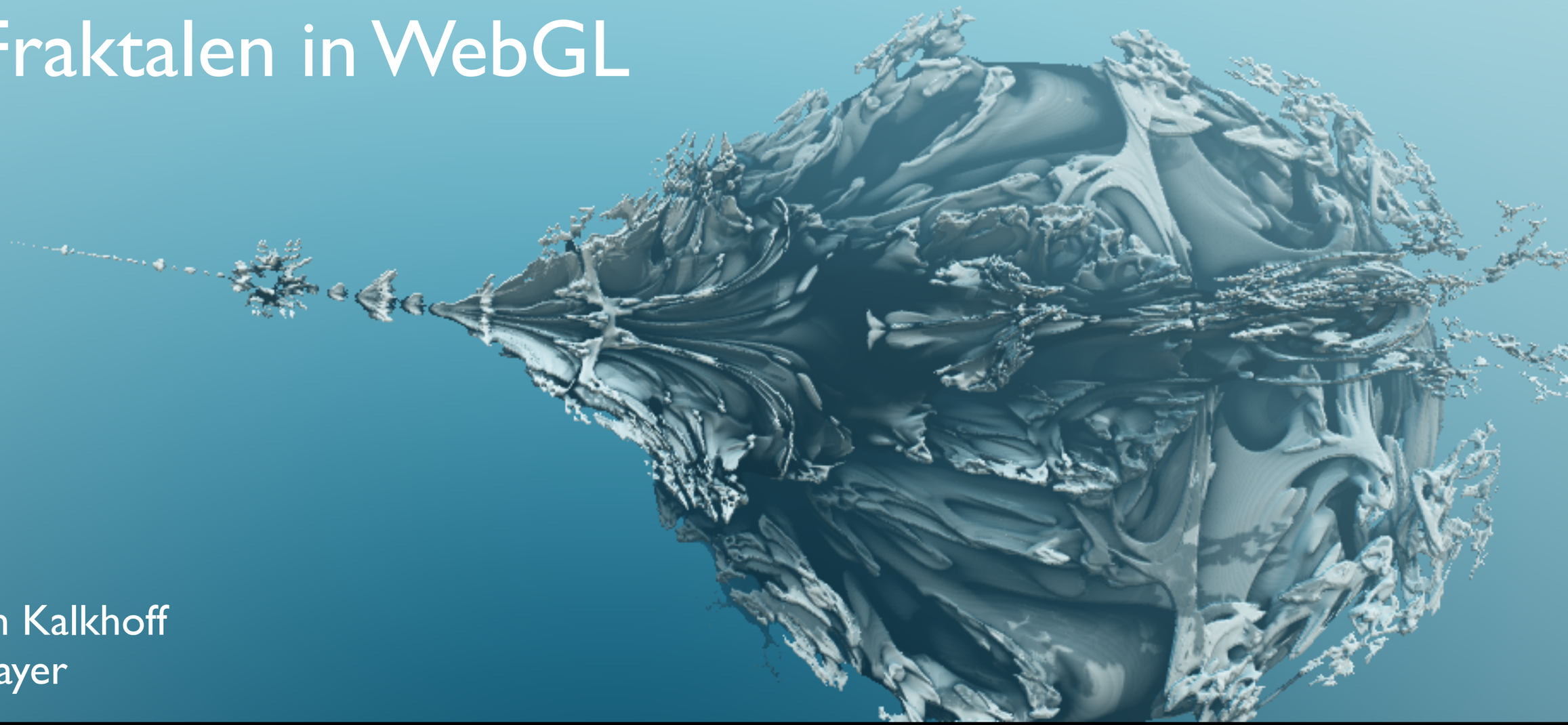
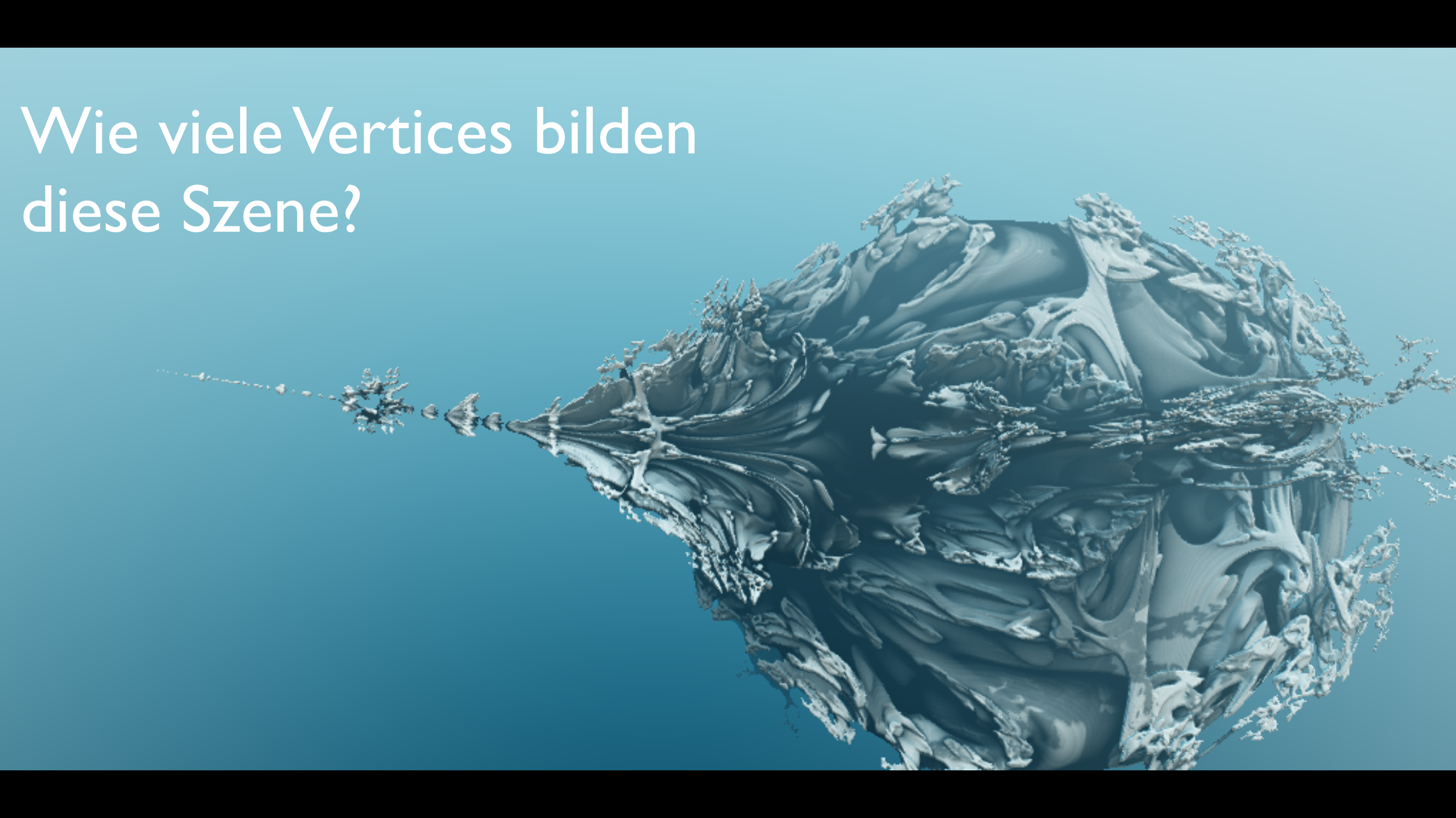
Interaktives Rendern von 3D Fraktalen in WebGL



Sebastian Kalkhoff
Tobias Bayer

Wie viele Vertices bilden diese Szene?



Wie viele Vertices bilden diese Szene?

4

Inhalt

- Zielsetzung
- Fraktale
- Darstellungsmoeglichkeiten
- Implementierung
- Optimierung
- Ergebnis

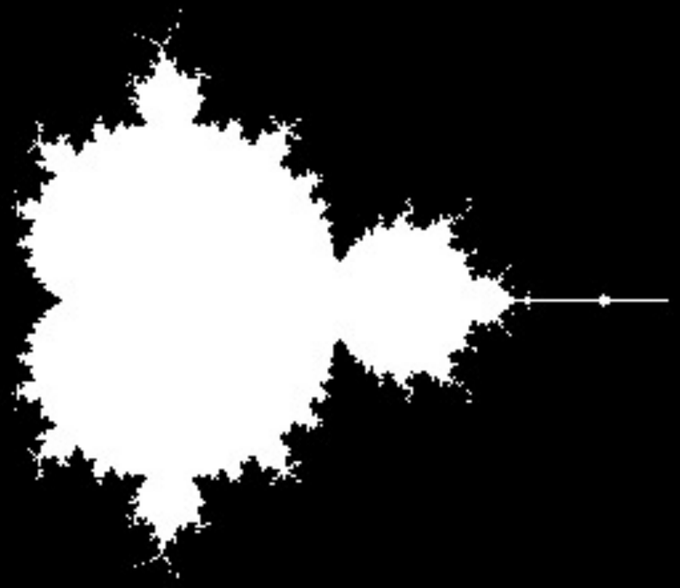
Zielsetzung

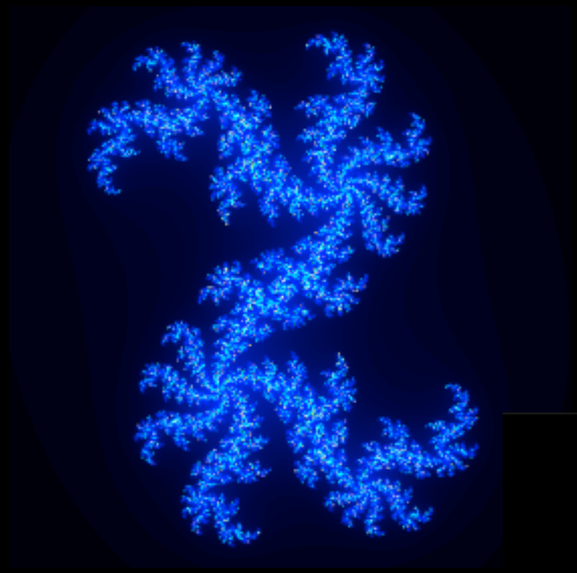
- Darstellung von verschiedenen 3D Fraktalen in WebGL
- frei bewegliche Kamera
- Interaktivitaet (20-30fps auf HD3000)

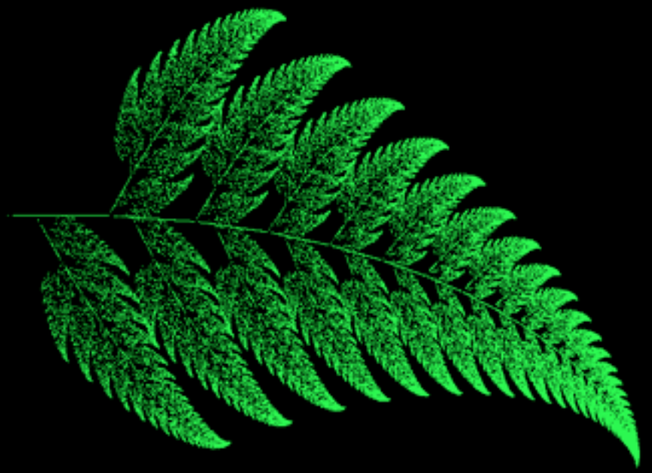
Was sind Fraktale?

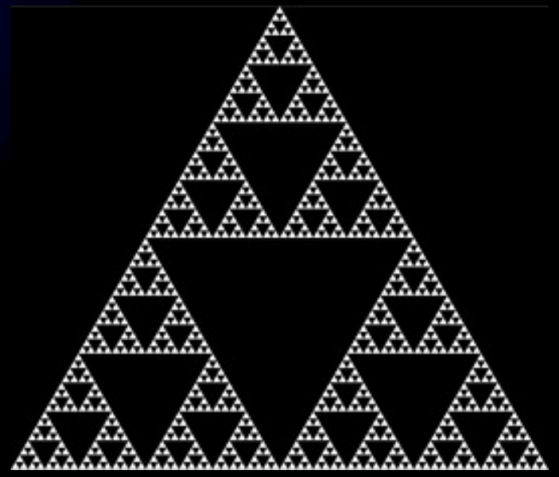
- “natuerliche oder kuenstliche Gebilde, die keine ganzzahlige Dimensionalitaet besitzen”
- im 2D/3D Raum unendlich detailierte Geometrien
- ergeben sich meist aus simplen rekursiven Funktionen

2D Fraktale

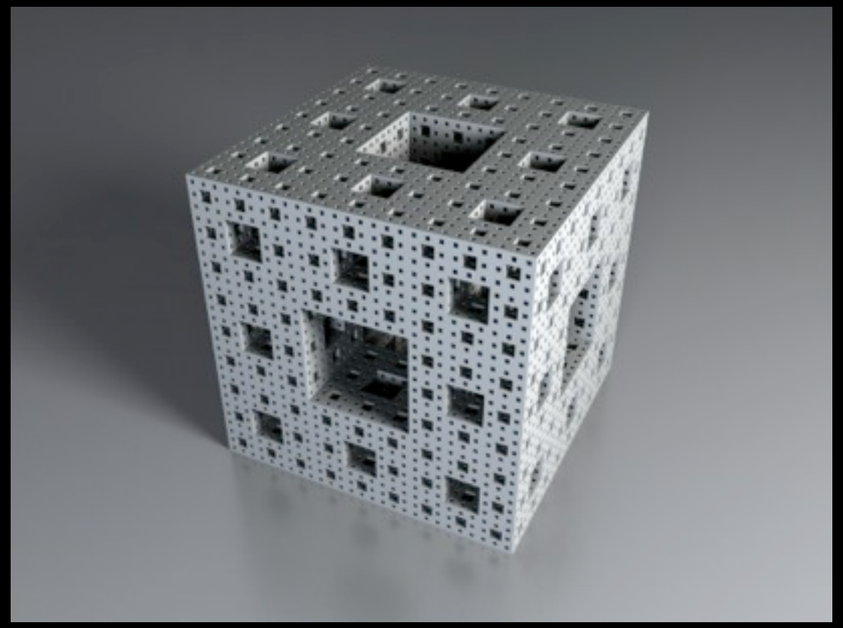


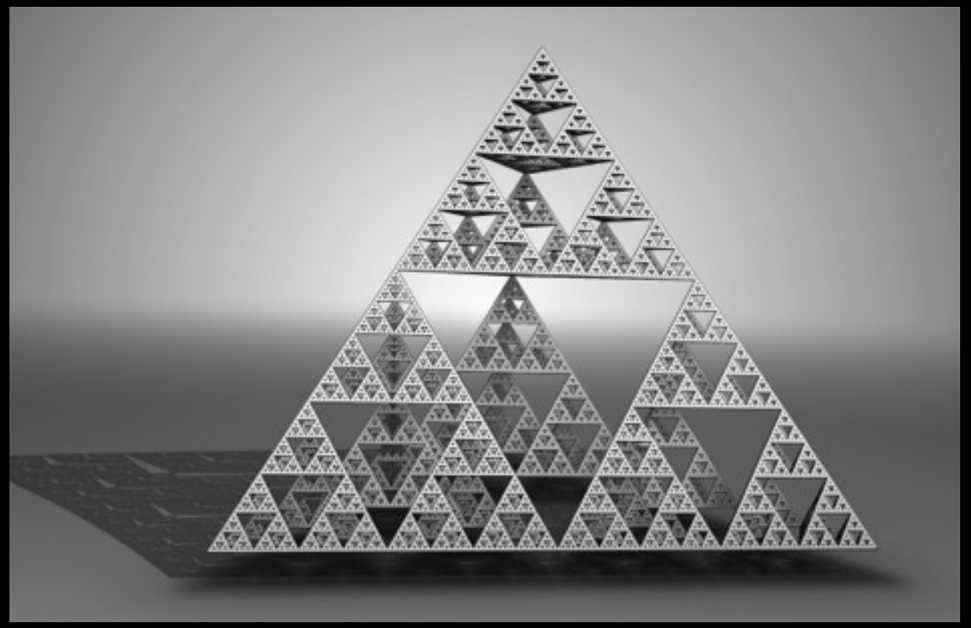


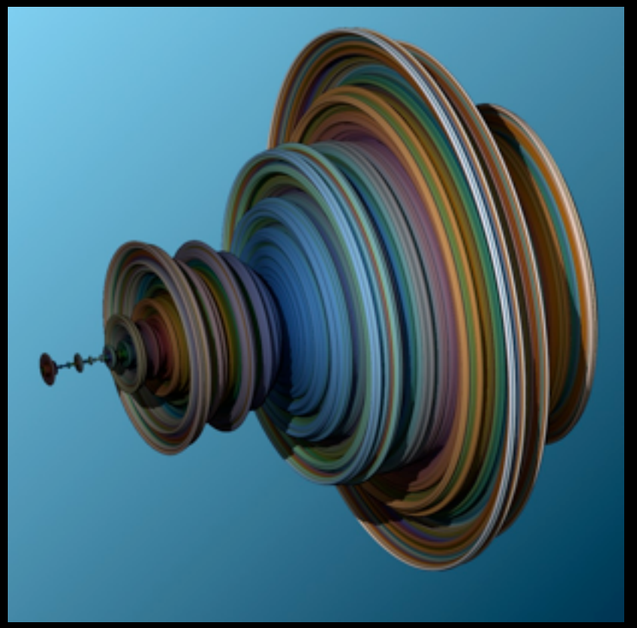




3D Fraktale







Warum Fraktale?

- visuell interessant
- verschiedene Darstellungsmoeglichkeiten
- Prozedural generierbar

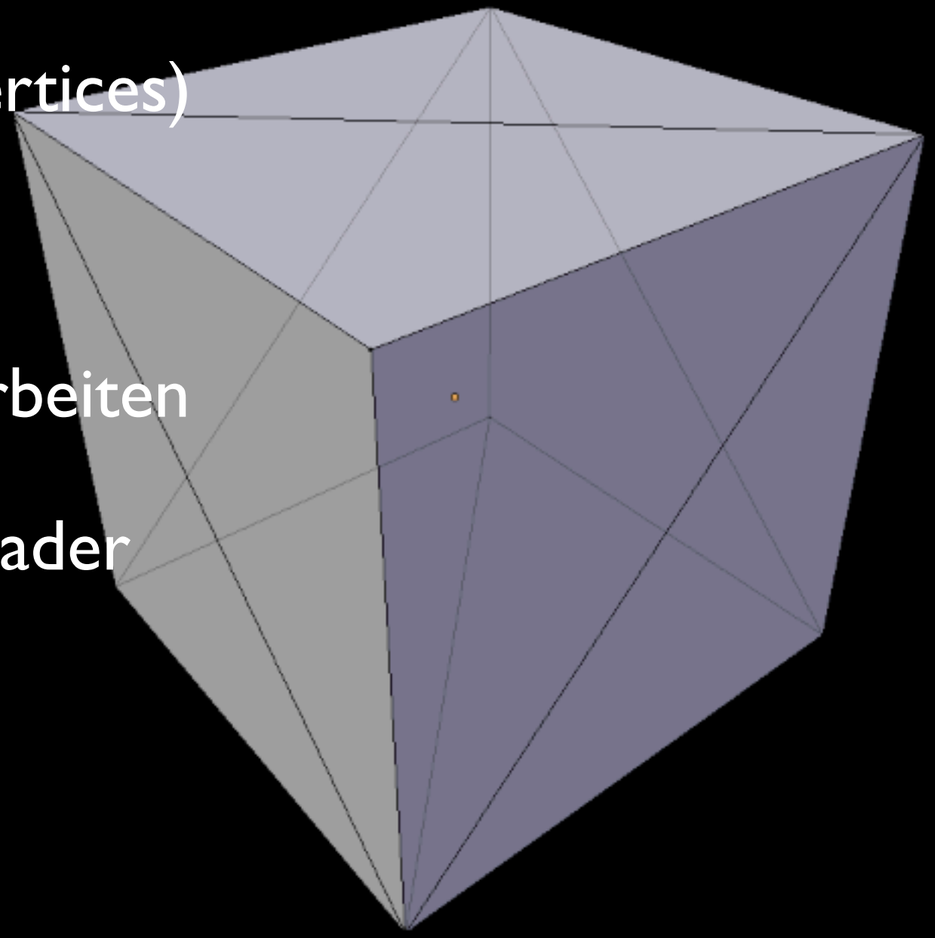
=> optimal fuer WebGL (Traffic)

Fraktale - Darstellung

- Polygondarstellung
- Voxeldarstellung
- Darstellung ueber Raymarching

Polygon

- Form definiert ueber 3 bis n Eckpunkte (Vertices)
- jede Flaeche besteht aus 3 Vertices
- Hardware kann Polygone sehr schnell verarbeiten
- Die Darstellung in WebGL erfolgt ueber Shader



Shader

- Programme die auf der GPU ausgefuehrt werden
- Drei Arten von Shadern
  - Vertex Shader
  - Geometry Shader
  - Fragment Shader
- 1 Vertex- & 1 Fragment Shader bilden ein Shader Programm
- GLSL Syntax ~ C Syntax

Shader

- Programme die auf der GPU ausgefuehrt werden
- Drei Arten von Shadern
  - Vertex Shader transformiert Vertices
  - Geometry Shader erzeugt zusaetzliche Geometrie (Tessellation)
  - Fragment Shader verarbeitet jedes Fragment (Pixel)
- 1 Vertex- & 1 Fragment Shader bilden ein Shader Programm
- GLSL Syntax ~ C Syntax

Shader

- Programme die auf der GPU ausgefuehrt werden
- Drei Arten von Shadern
  - Vertex Shader — transformiert Vertices
  - ~~Geometry Shader~~ — in WebGL nicht vorhanden
  - Fragment Shader — verarbeitet jedes Fragment (Pixel)
- 1 Vertex- & 1 Fragment Shader bilden ein Shader Programm
- GLSL Syntax ~ C Syntax

Vertex Shader Beispiel

```
void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

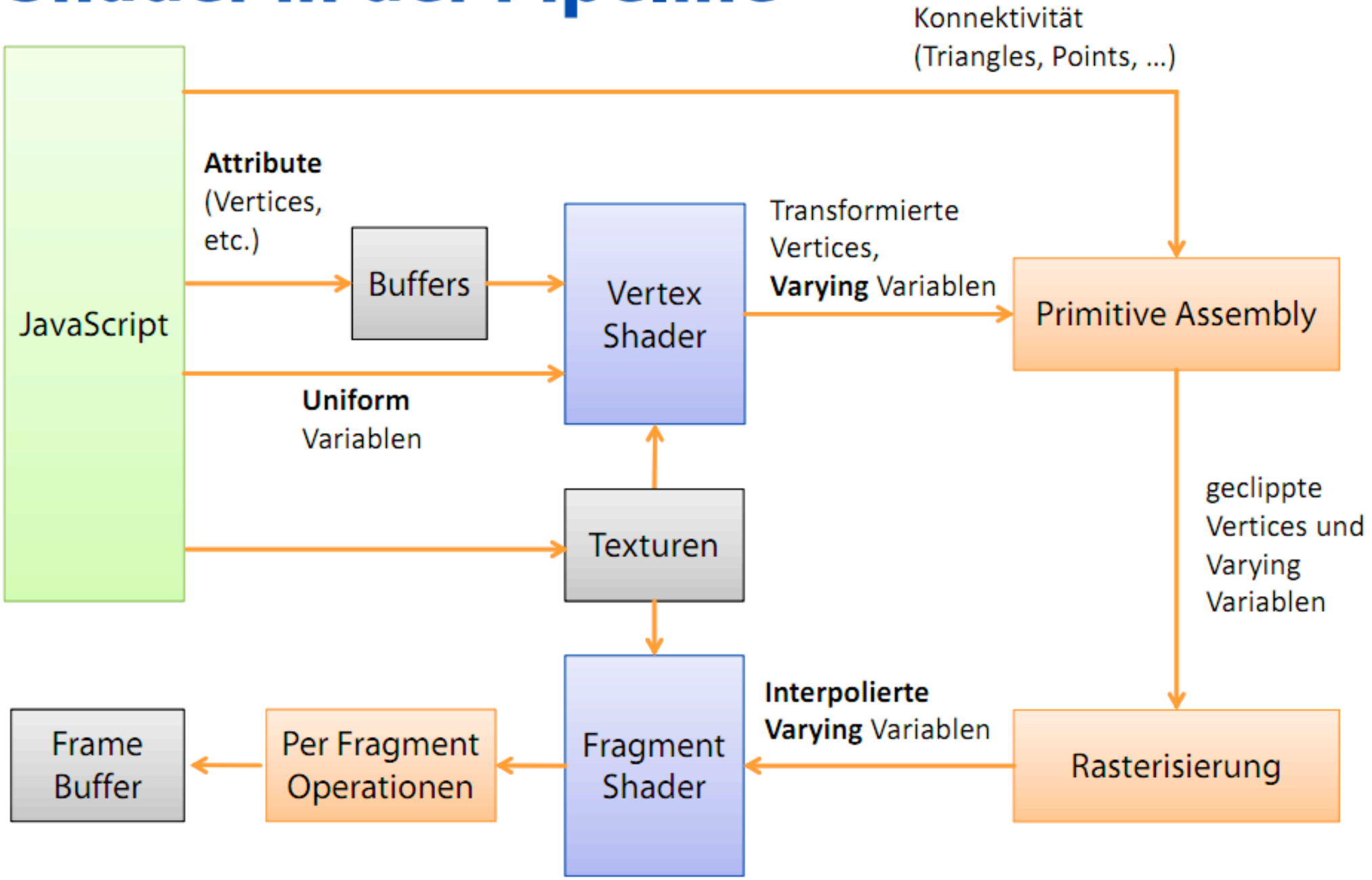
Fragment Shader Beispiel

```
void main(void)
{
   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Datenaustausch Shader

- verschiedene Datentypen: (int, float, vec2, vec3, vec4, mat3, mat4)
  - Pro Drawcall : Uniforms
  - Pro Vertex : Attributes
  - von Vertex zu Fragment : Varyings (perspektivisch interpoliert)
- Texturen als Input : Sampler
- Texturen als Output : Render to Texture

Shader in der Pipeline



[Fuhrmann I I]

Vorteile Polygondarstellung

- bewaehrt und schnell
- viele hilfreiche Funktionen vorhanden (WebGL Standard)
- ausfuehrlich dokumentiert

=> einfach zu benutzen, sofern Vertices vorhanden sind

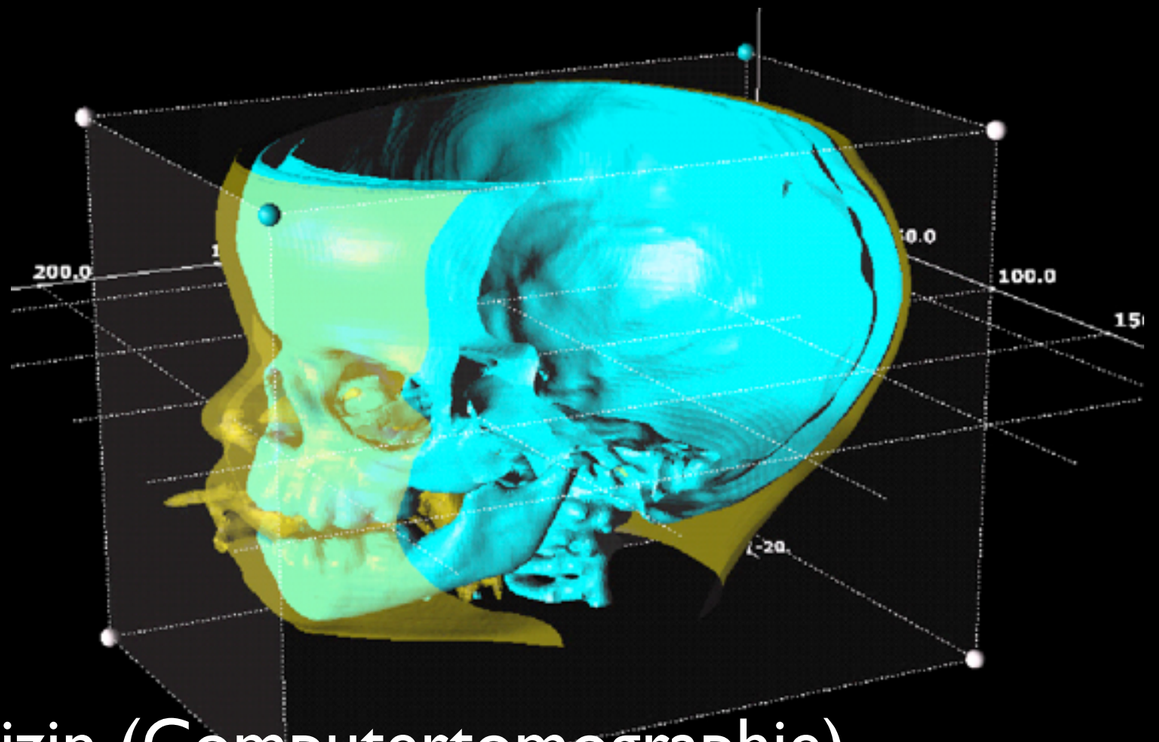
Nachteile Polygondarstellung

- Hohe Detailstufen = hoher Speicherbedarf
- Level of Detail durch Polygone festgelegt
- Vertices fuer komplexe Geometrien muessen erst erstellt werden

  Fraktal triangulieren?

  => ungeeignet

Voxel



- volumetrischer Pixel
  - Farbe
  - Position im Raum
- Findet oft Anwendung in der Medizin (Computertomographie)
- Standard fuer 3D Scans

Vorteile Voxeldarstellung

- effizient zu rendern durch Baumstruktur (Octree)
- einfaches Level of Detail durch Baumstruktur
- Detailgrad nur durch den Speicher begrenzt

Nachteile Voxeldarstellung

- extrem hoher Speicherbedarf bei geringer raeumlicher Aufloesung
  - 512^3 Wuerfel aus RGBA 8 bit pro Kanal ~ 512MB
- Datenstruktur muss gefuellt werden
  - Fraktal muss dreidimensional gerastert werden
- Neuaufbau der Baumstruktur bei Aenderung der Geometrie

=> ungeeignet fuer Fraktale

Raymarching mit Distance Fields

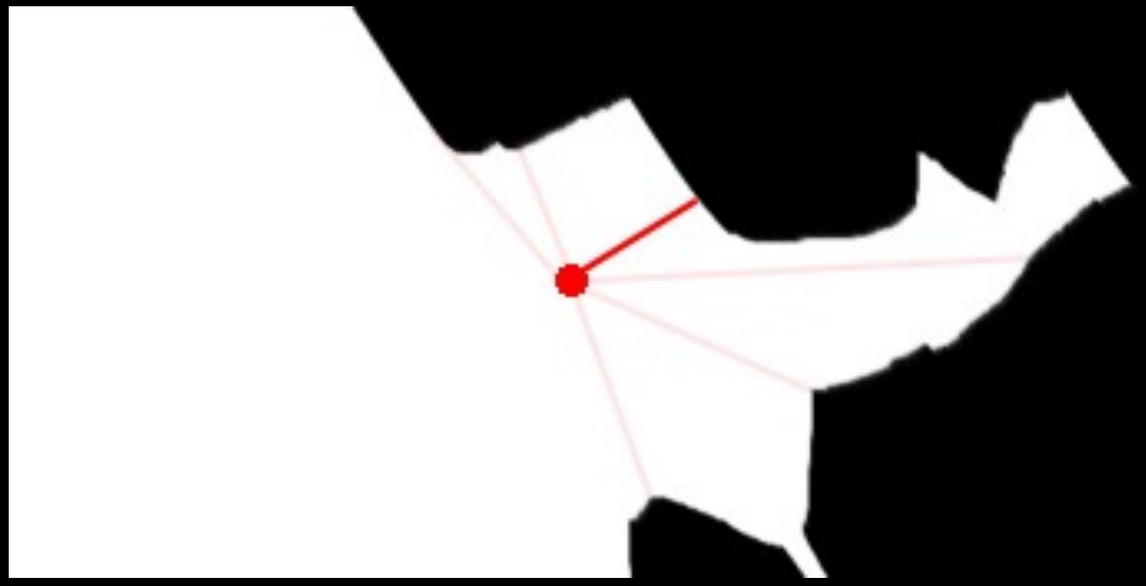
- Eine Art von Raycasting
- Verfahren zum Approxmieren eines Volumens (mind. 1 Ray pro Pixel)
- ideal parallelisierbar  GPU !
- Schrittweite der Rays werden durch Distance Fields bestimmt
- Distance Field gegeben durch den Distance Estimator
- minimum Distance als Abbruchbedingung

Raymarching mit Distance Fields

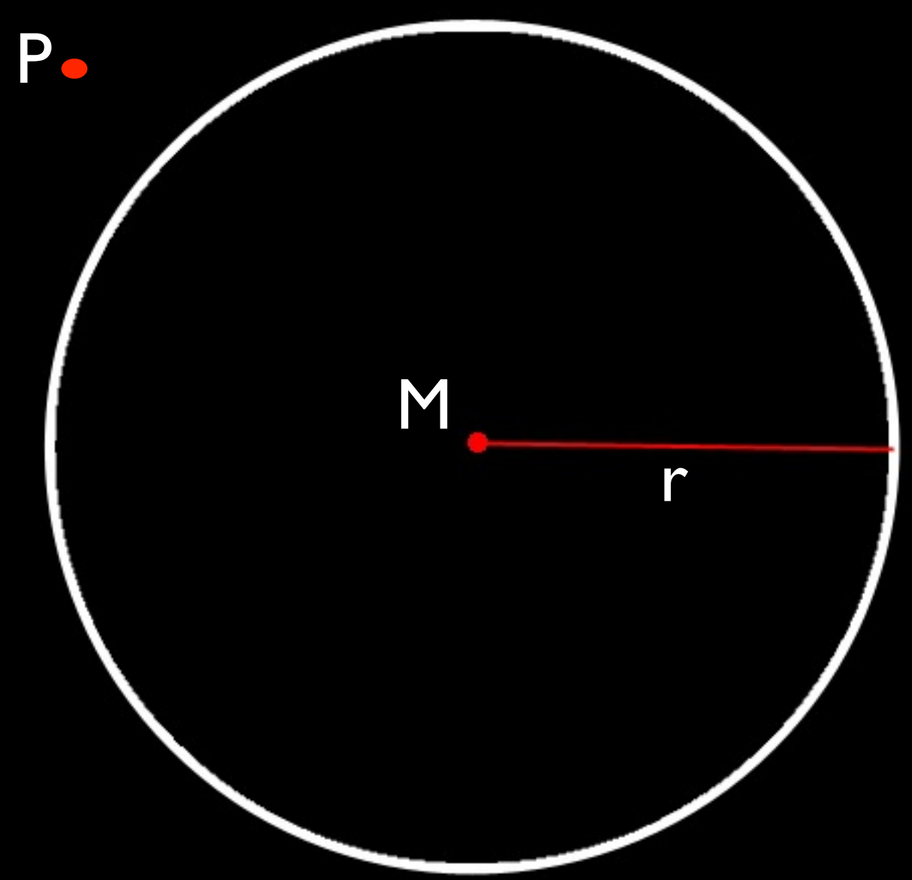
- Eine Art von Raycasting
- Verfahren zum Approxmieren eines Volumens (mind. 1 Ray pro Pixel)
- ideal parallelisierbar  GPU !
- Schrittweite der Rays werden durch Distance Fields bestimmt
- Distance Field gegeben durch den Distance Estimator
- minimum Distance als Abbruchbedingung

Distance Estimator

- Eine Funktion f(x,y,z), die jedem Punkt im Raum einen Abstand zu der von der Funktion beschriebenen Geometrie zuweist.
- Man erhaelt immer den **kuerzesten Abstand** zu der Geometrie



Distance Estimator



Bsp. DE(Kreis) :

- definiert durch Mittelpunkt und Radius

Der Abstand von einem Punkt P zum Mittelpunkt kann einfach berechnet werden.

d = length(P - M)

Zieht man nun den Radius ab erhaelt man die kuerzeste Distanz zum Kreis.

DE(Kreis) = f(x,y) = d = length(P - M) - r

Distance Estimator

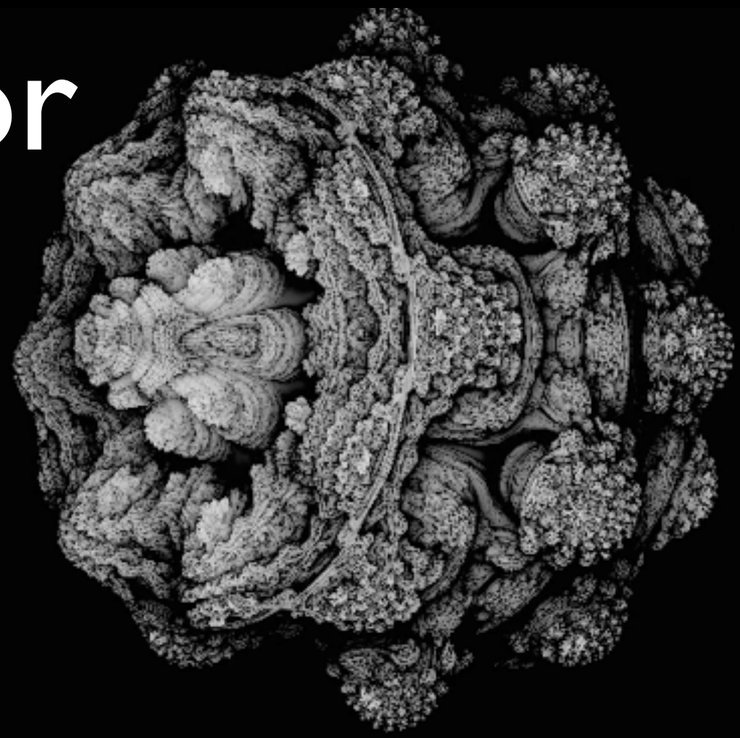
- Fraktale lassen sich sehr gut durch Distance Estimator darstellen
- Es kann unendlich* viel Detail ohne grossen Speicheraufwand dargestellt werden

Distance Estimator

- Fraktale lassen sich sehr gut durch Distance Estimator darstellen
- Es kann unendlich* viel Detail ohne grossen Speicheraufwand dargestellt werden

* unendlich ist hier durch die float Genauigkeit begrenzt

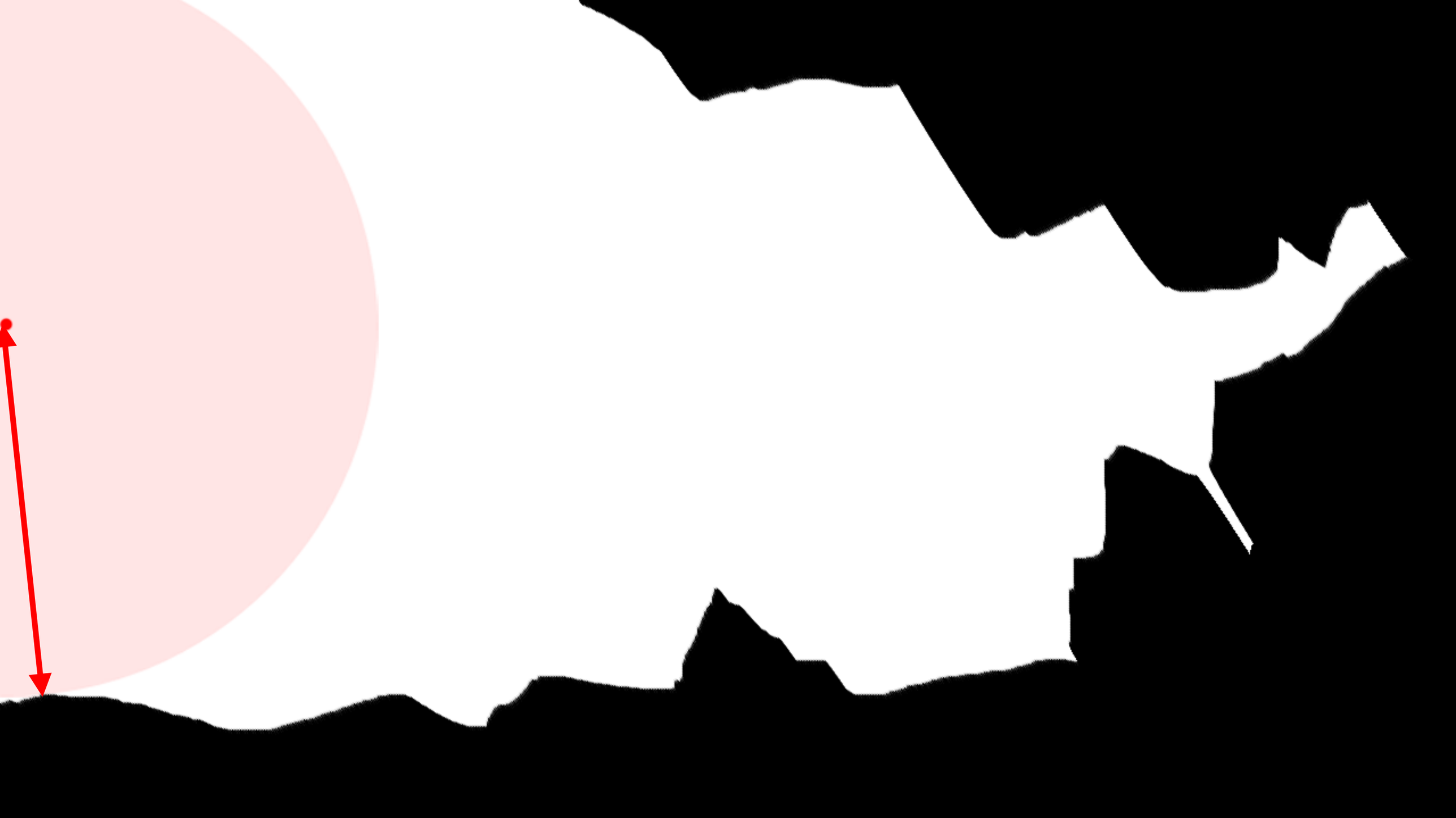
Distance Estimator

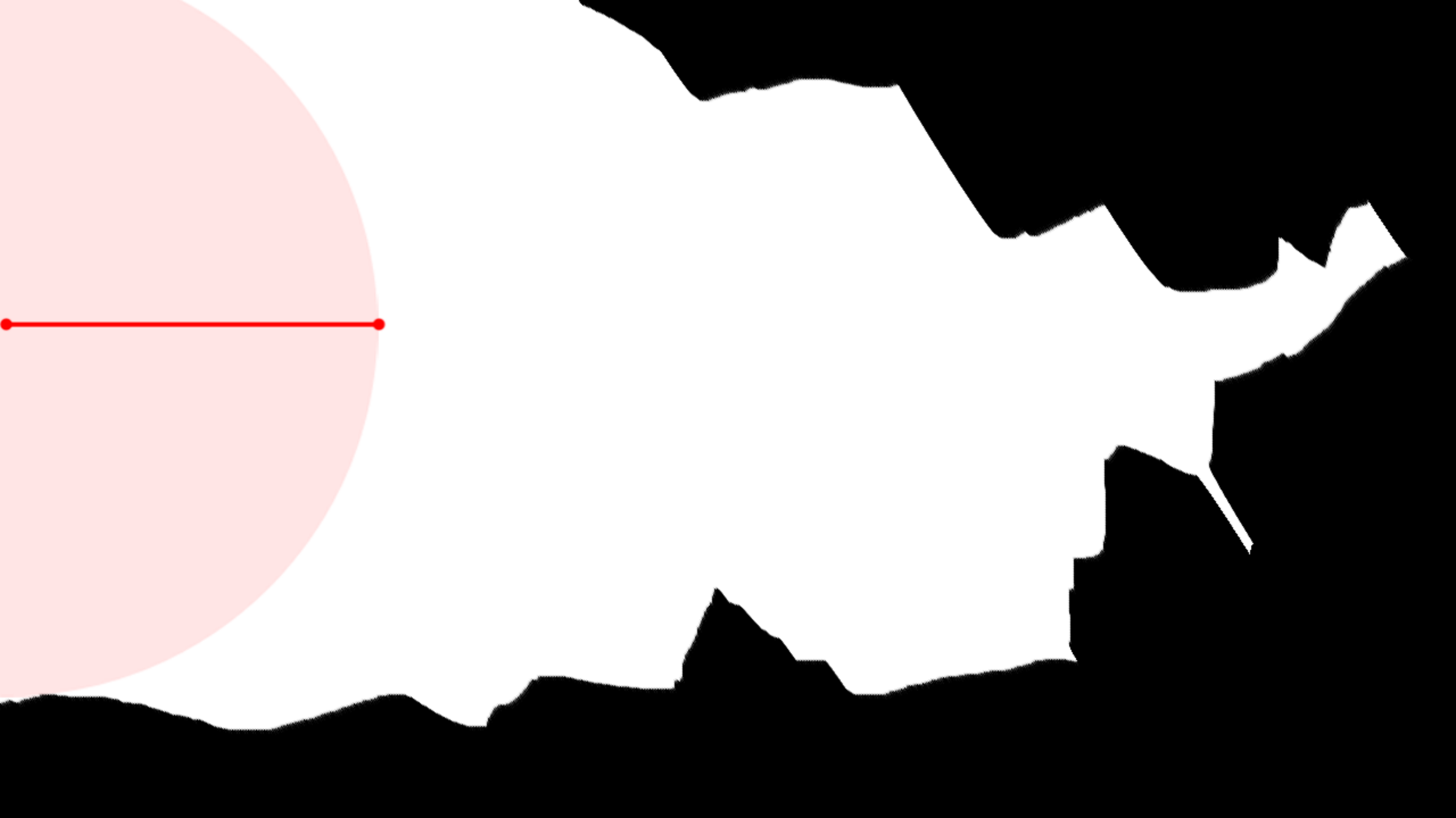


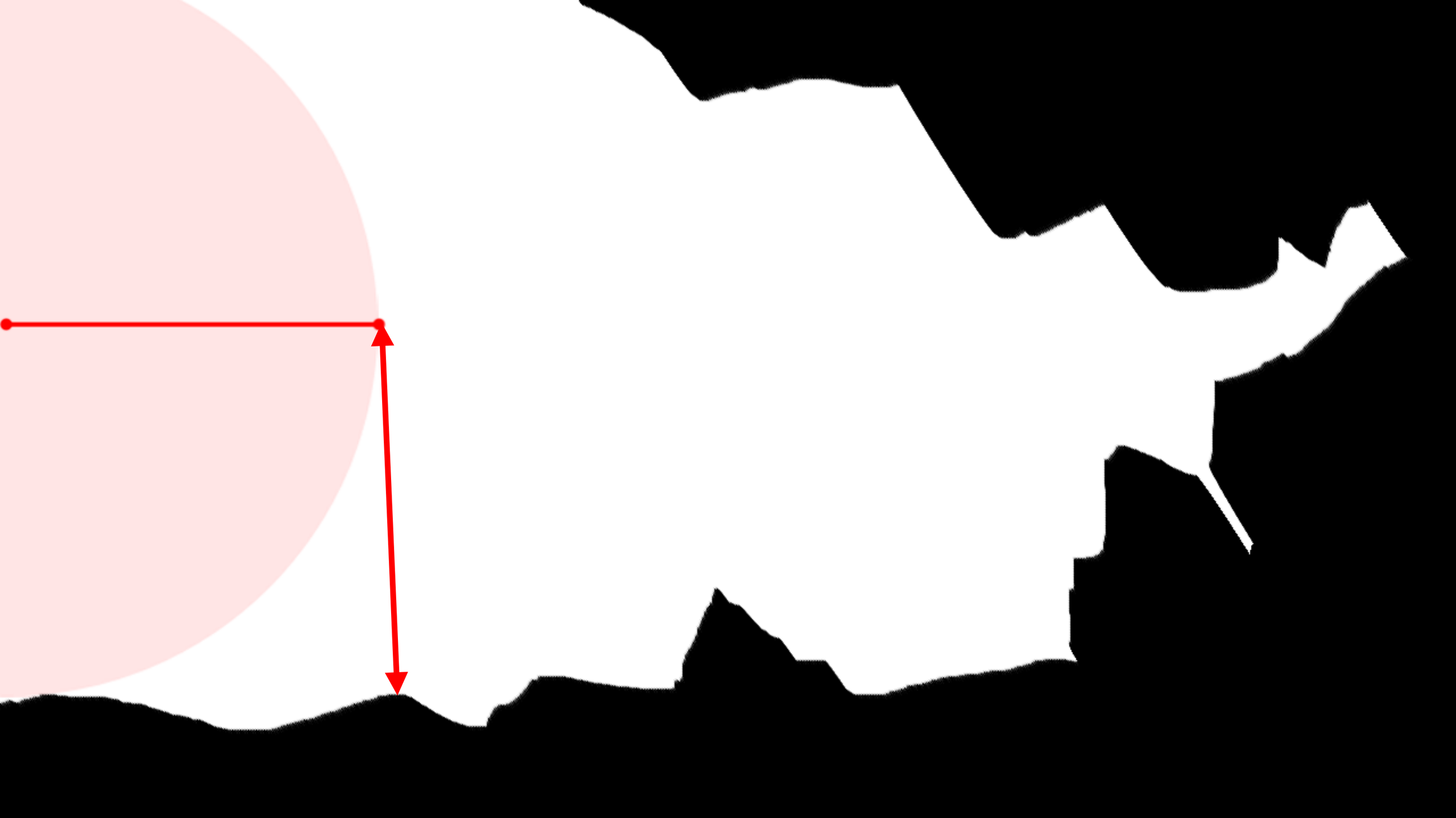
```
float distanceEstimatorMandelBulb(vec3 pos) {
   vec3 z = pos;
   float dr = 1.0;
   float r = 0.0;
   for (int i = 0; i < Iterations ; i++)
   {
       r = length(z);
       if(r > Bailout) break;
       dr =  pow( r, uPower - 1.0) * uPower * dr + 1.0;

       // convert to polar coordinates
       float theta = acos(z.z / r);
       float phi = atan(z.y, z.x);

       // scale and rotate the point
       float zr = pow( r,uPower);
       theta = theta * uPower;
       phi = phi * uPower;

       // convert back to cartesian coordinates
       z = zr * vec3(sin(theta) * cos(phi), sin(phi) * sin(theta), cos(theta));
       z += pos;
   }
   return 0.5 * log(r) * r / dr ;
}
```

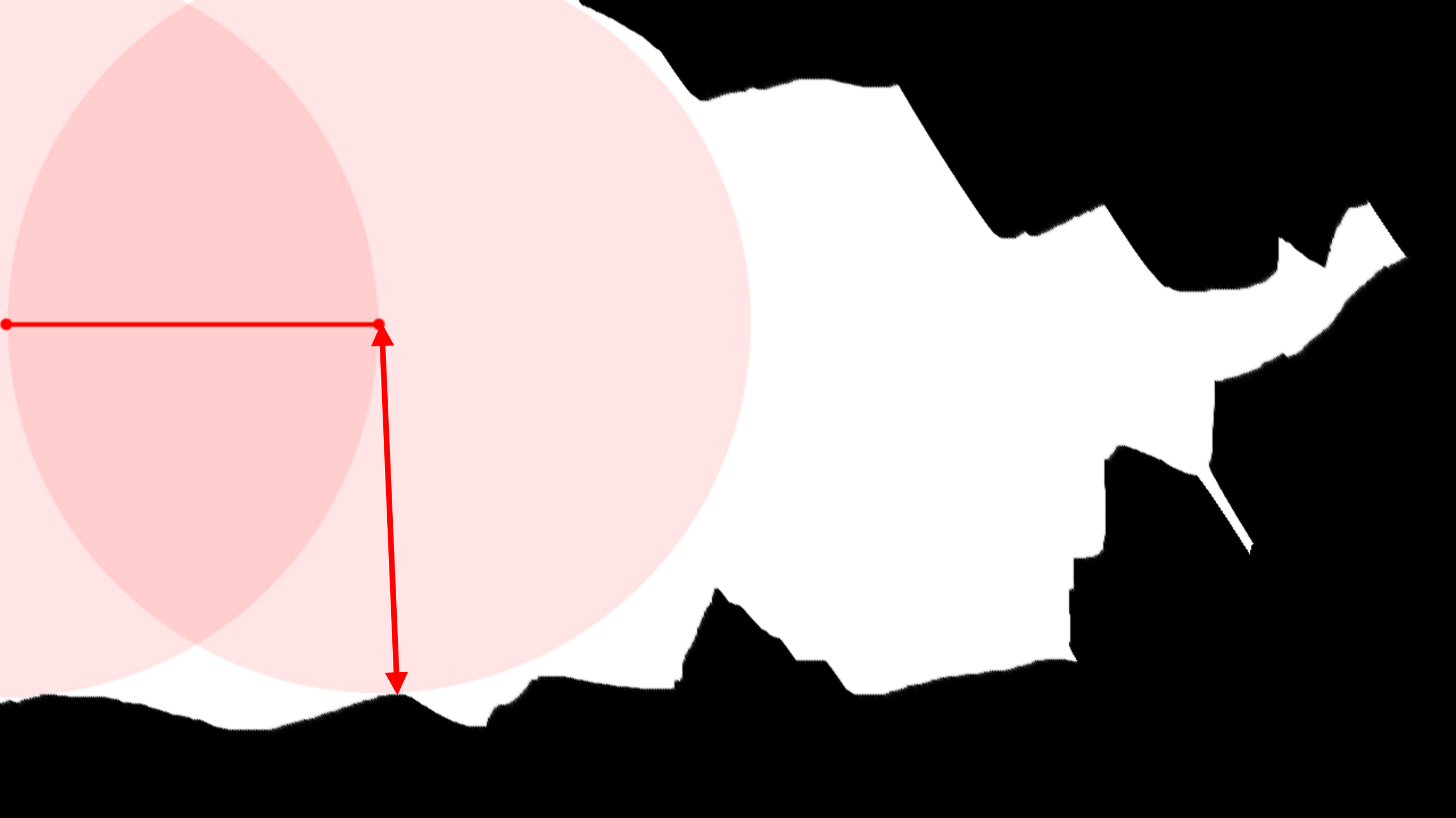
Raymarching mit Distance Fields

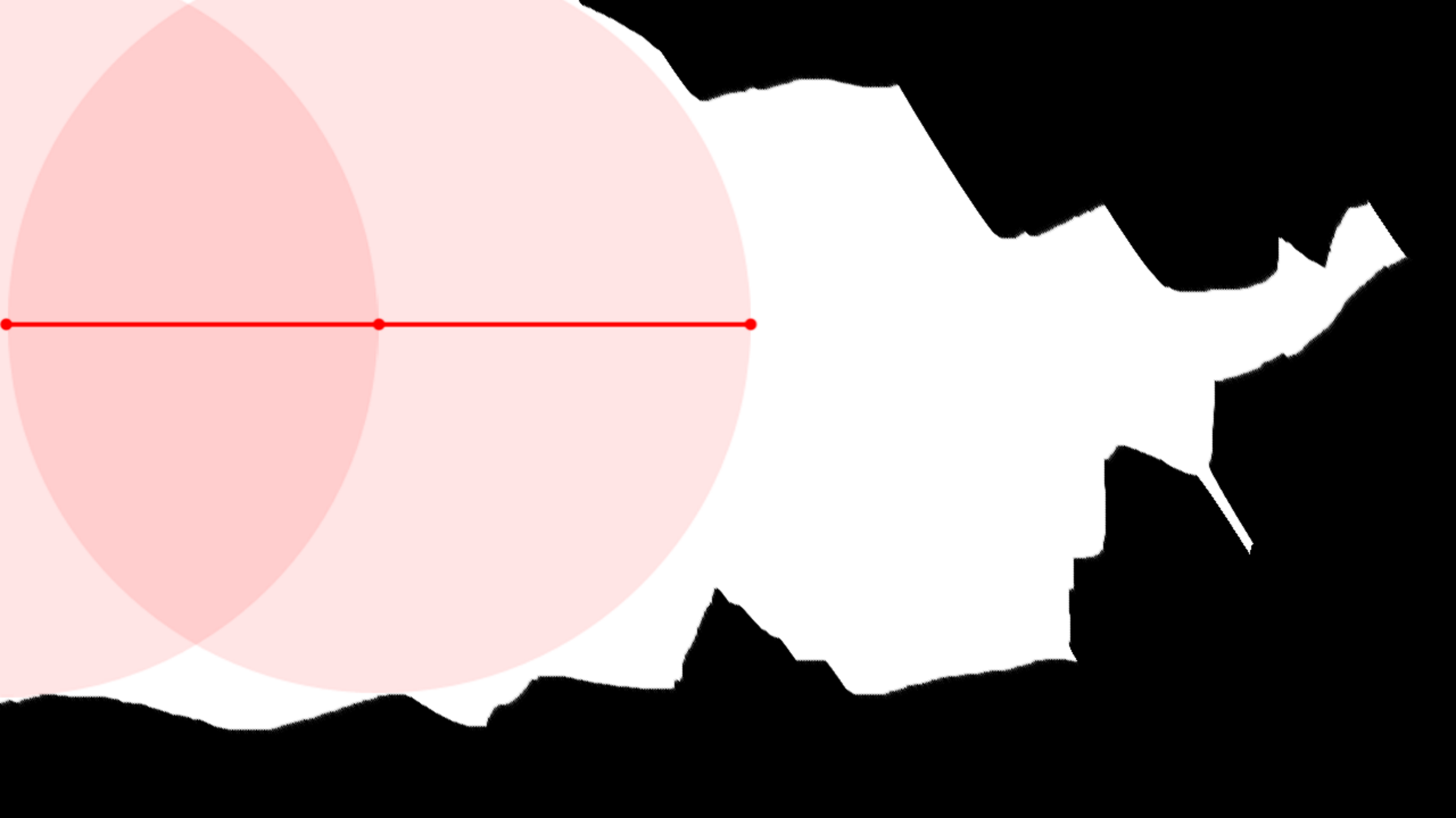
- Eine Art von Raycasting
- Verfahren zum Approxmieren eines Volumens (mind. 1 Ray pro Pixel)
- ideal parallelisierbar  GPU !
- Schrittweite der Rays werden durch Distance Fields bestimmt
- Distance Field gegeben durch den Distance Estimator
- minimum Distance als Abbruchbedingung

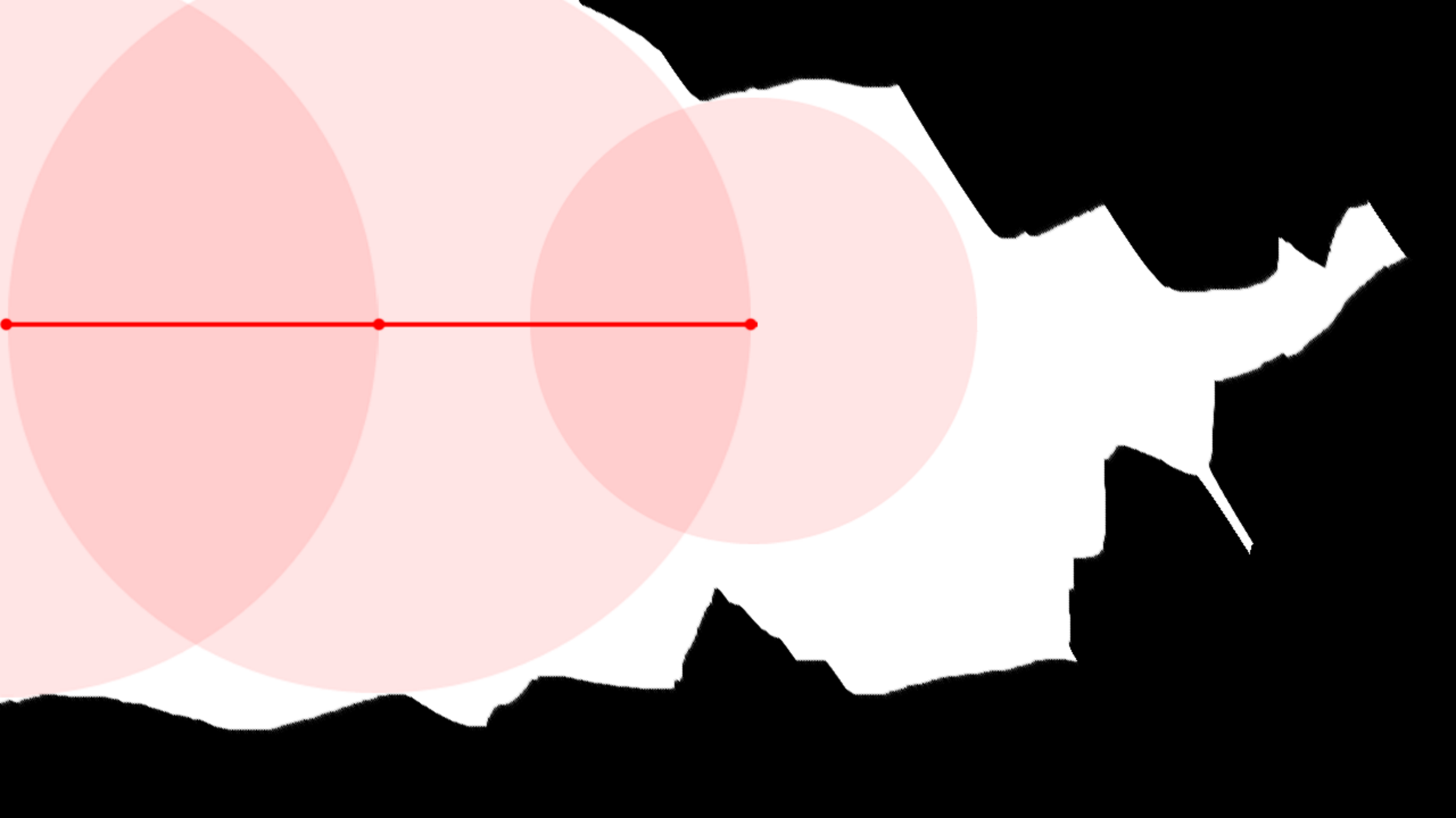


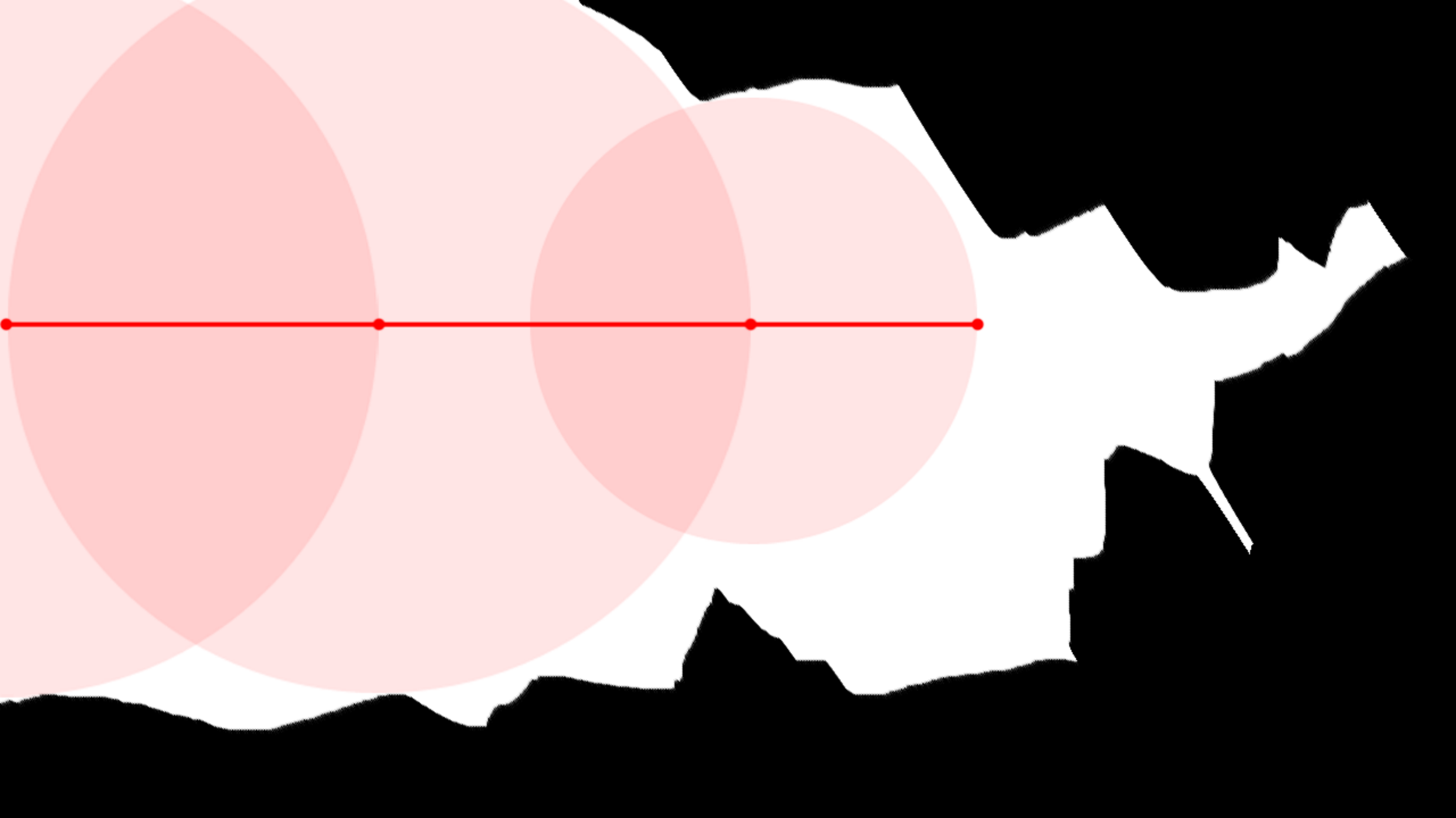


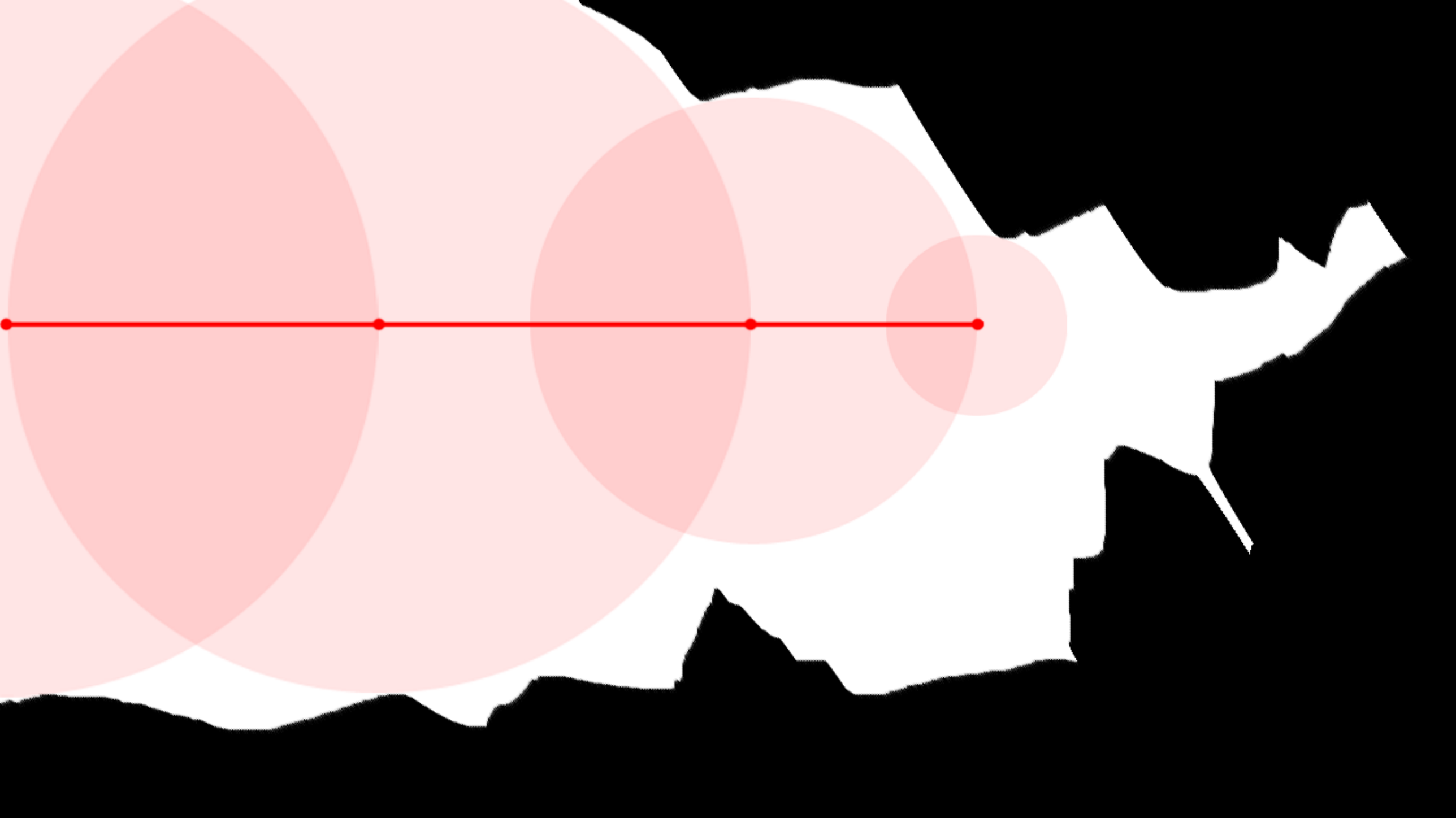


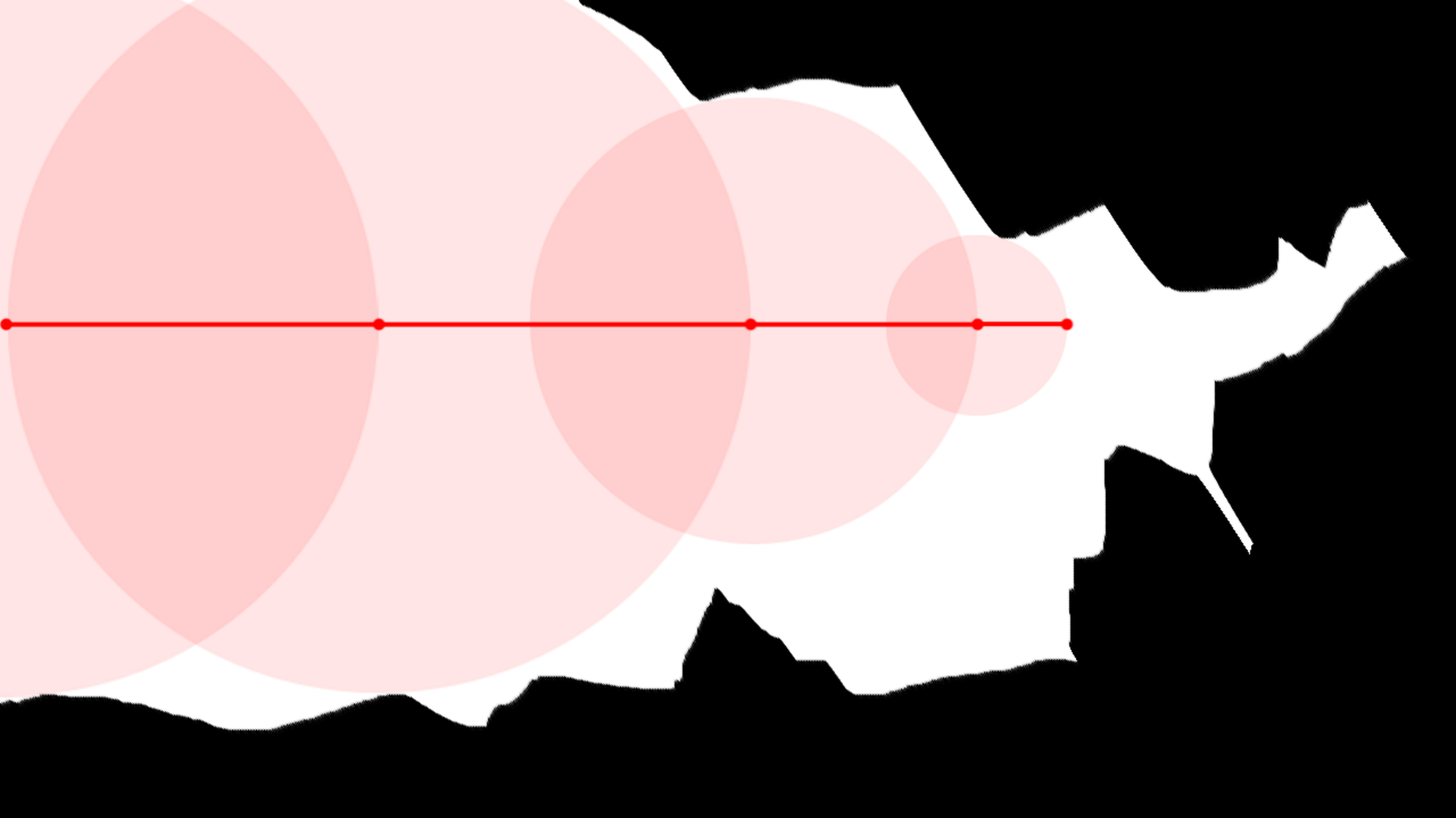


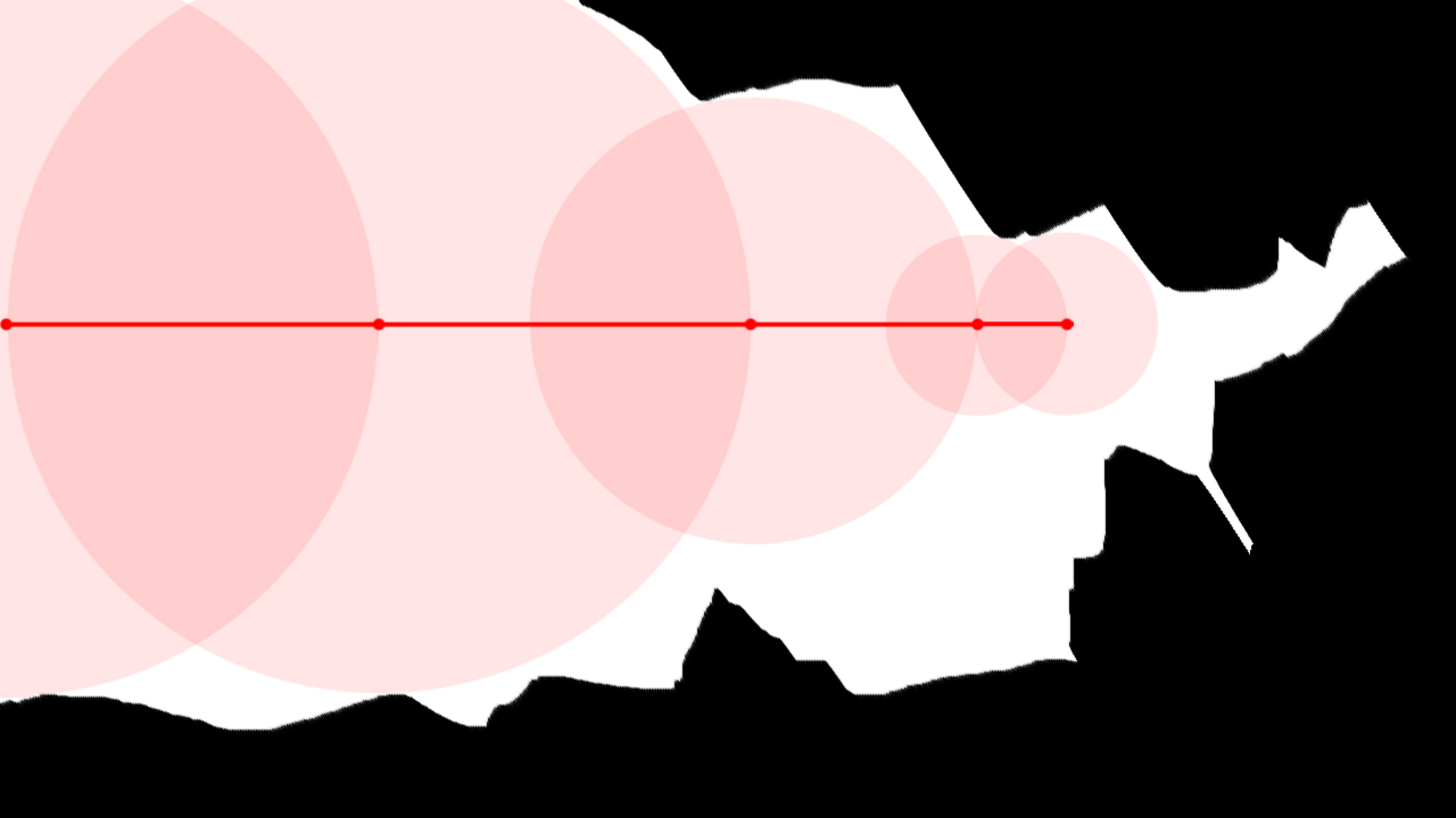


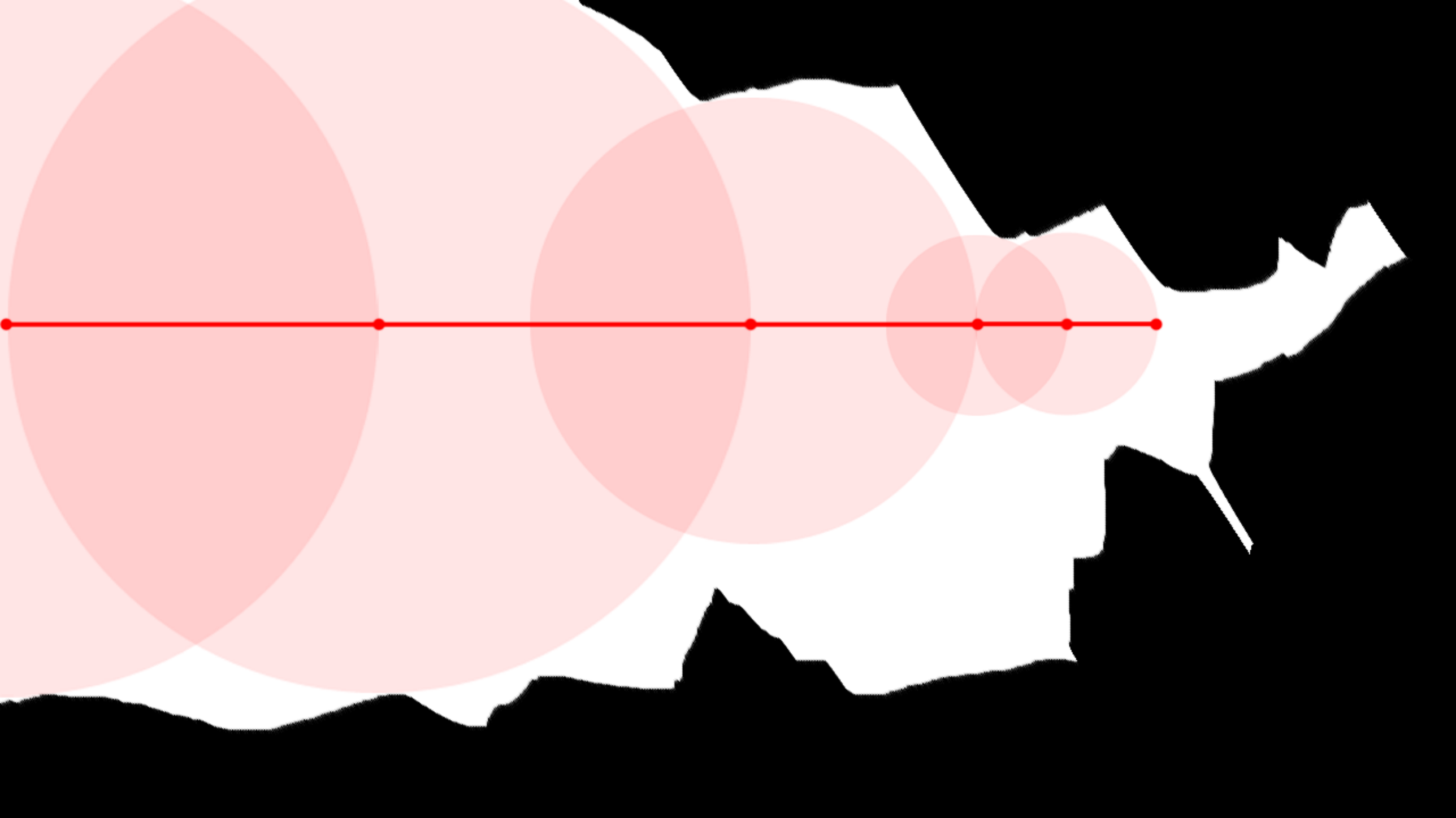


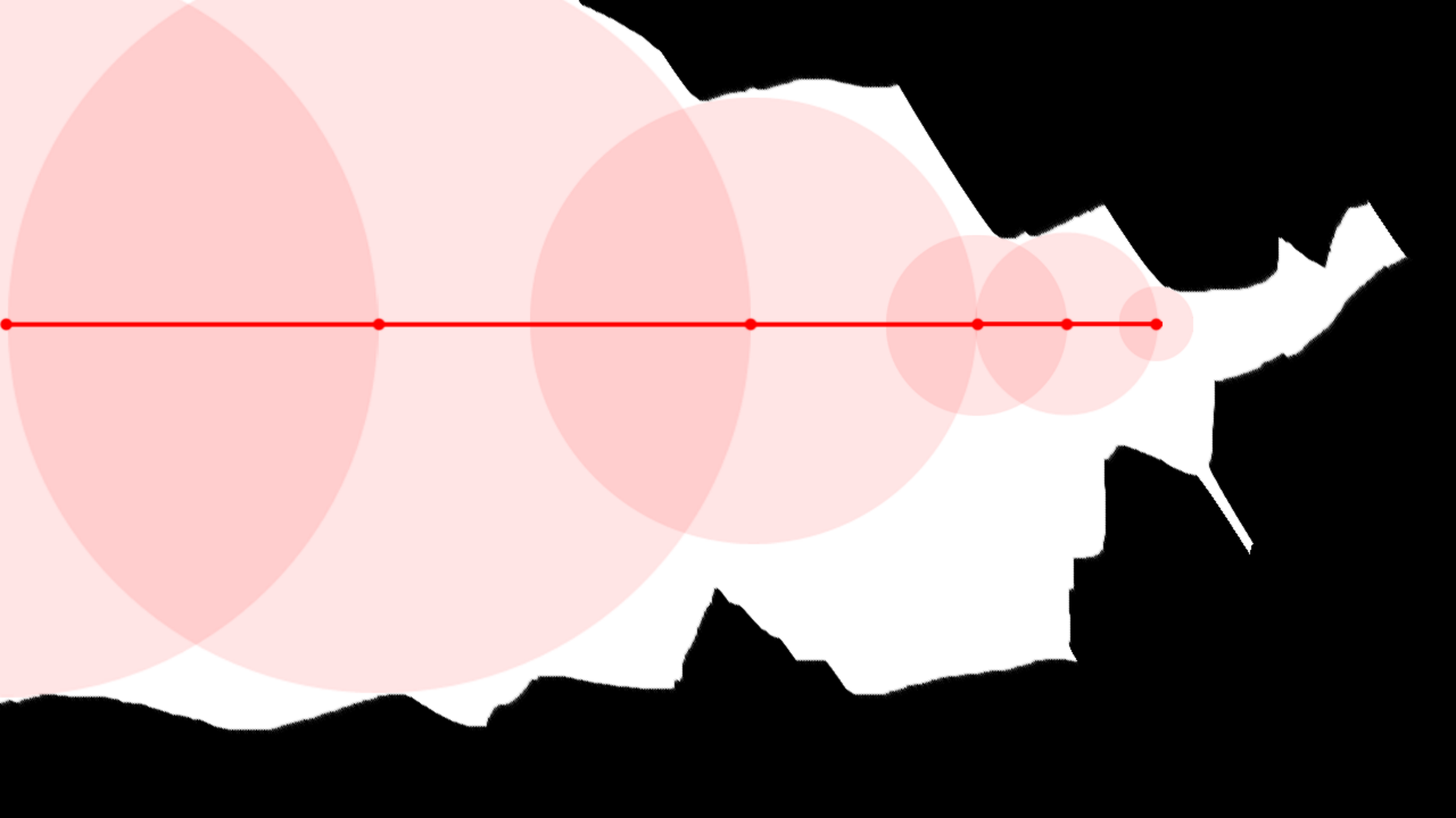


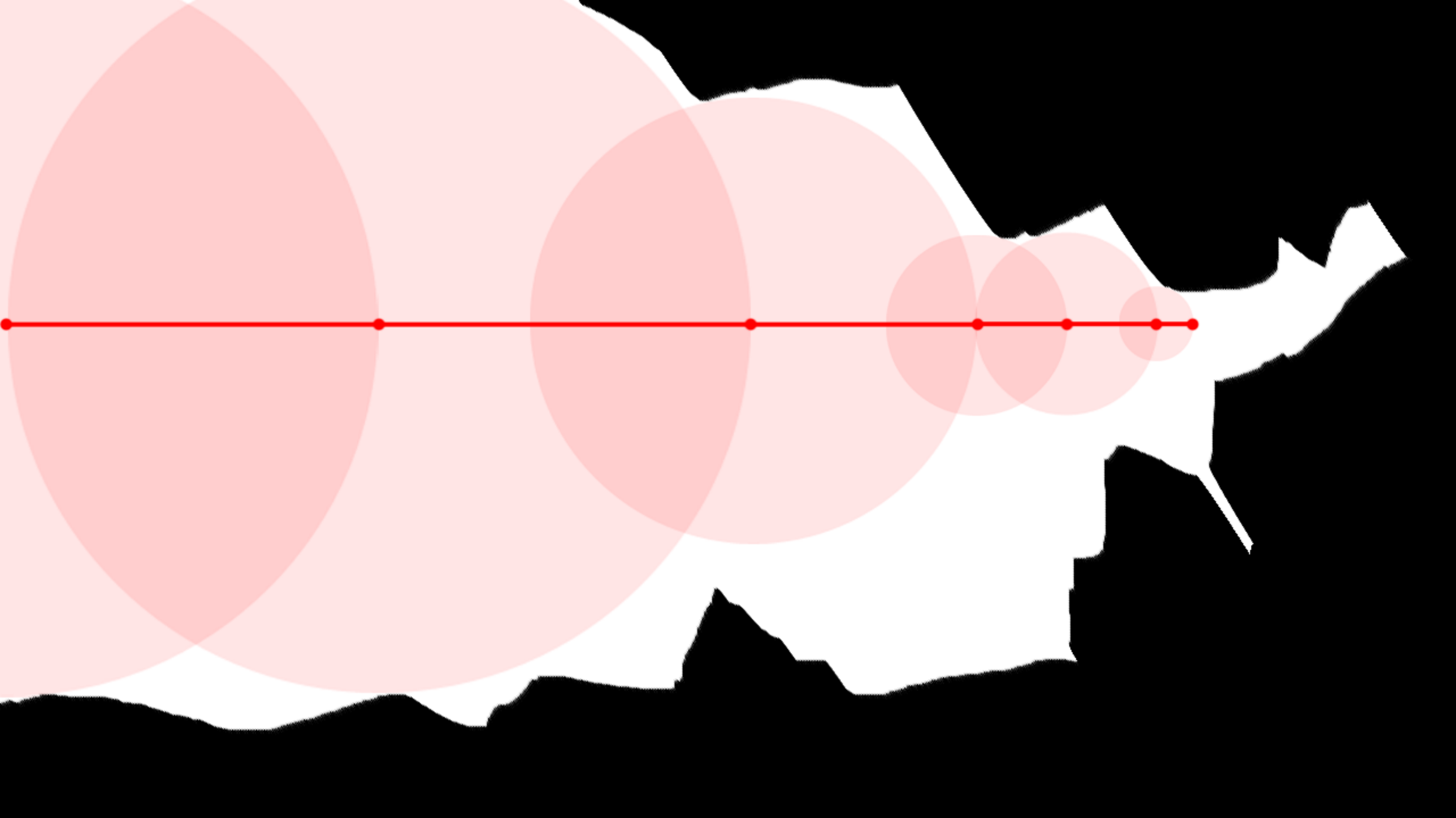


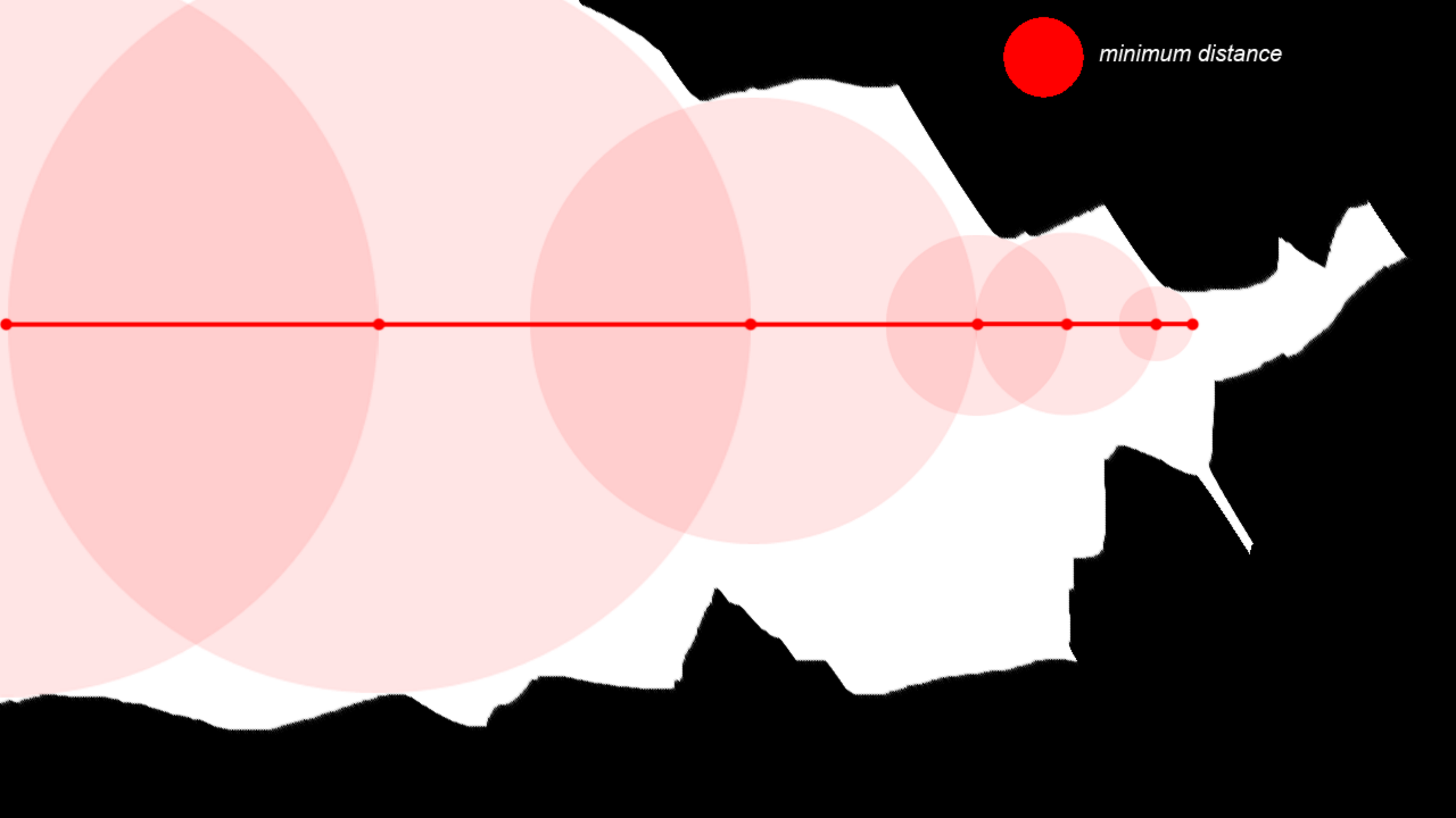






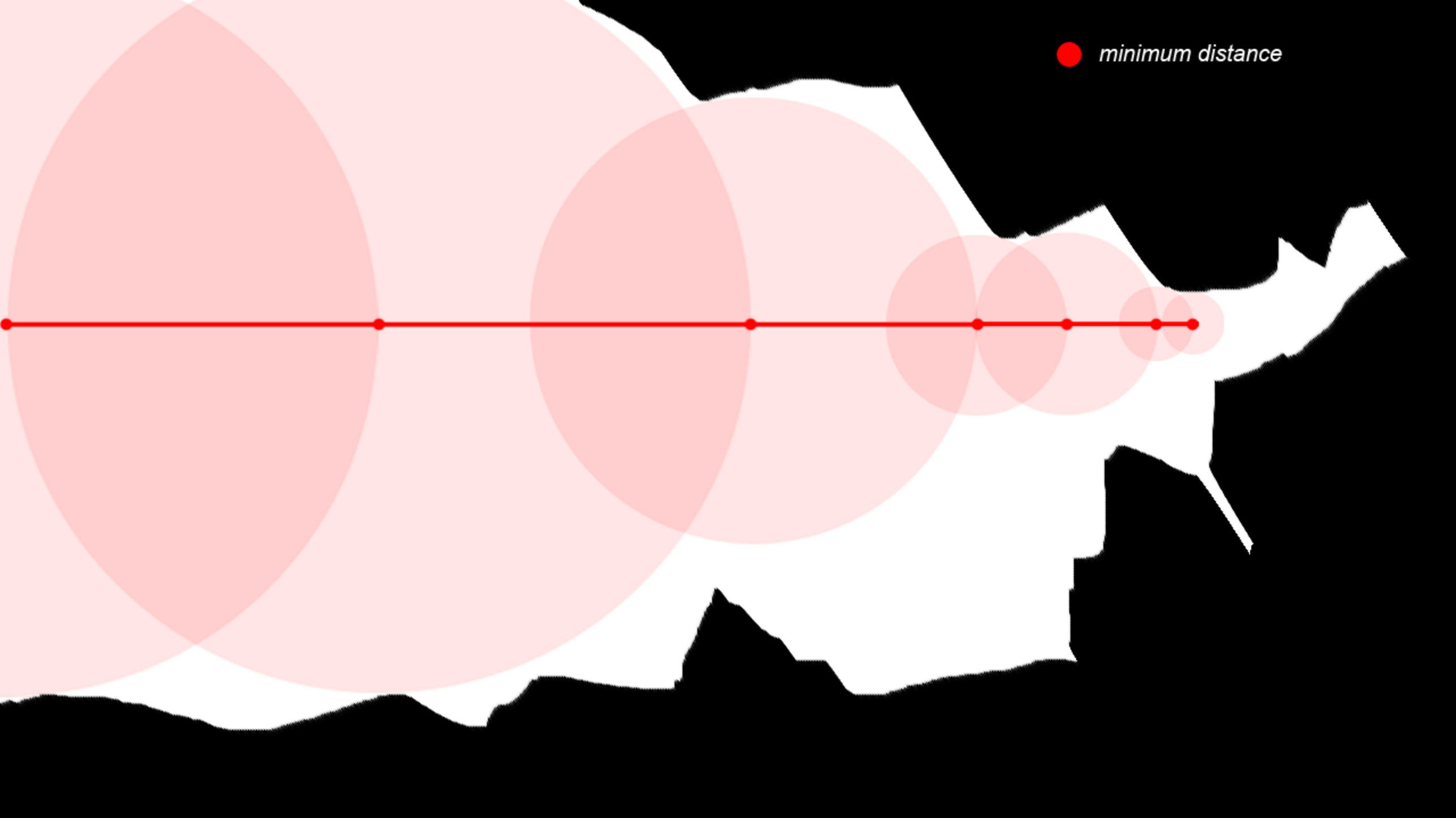


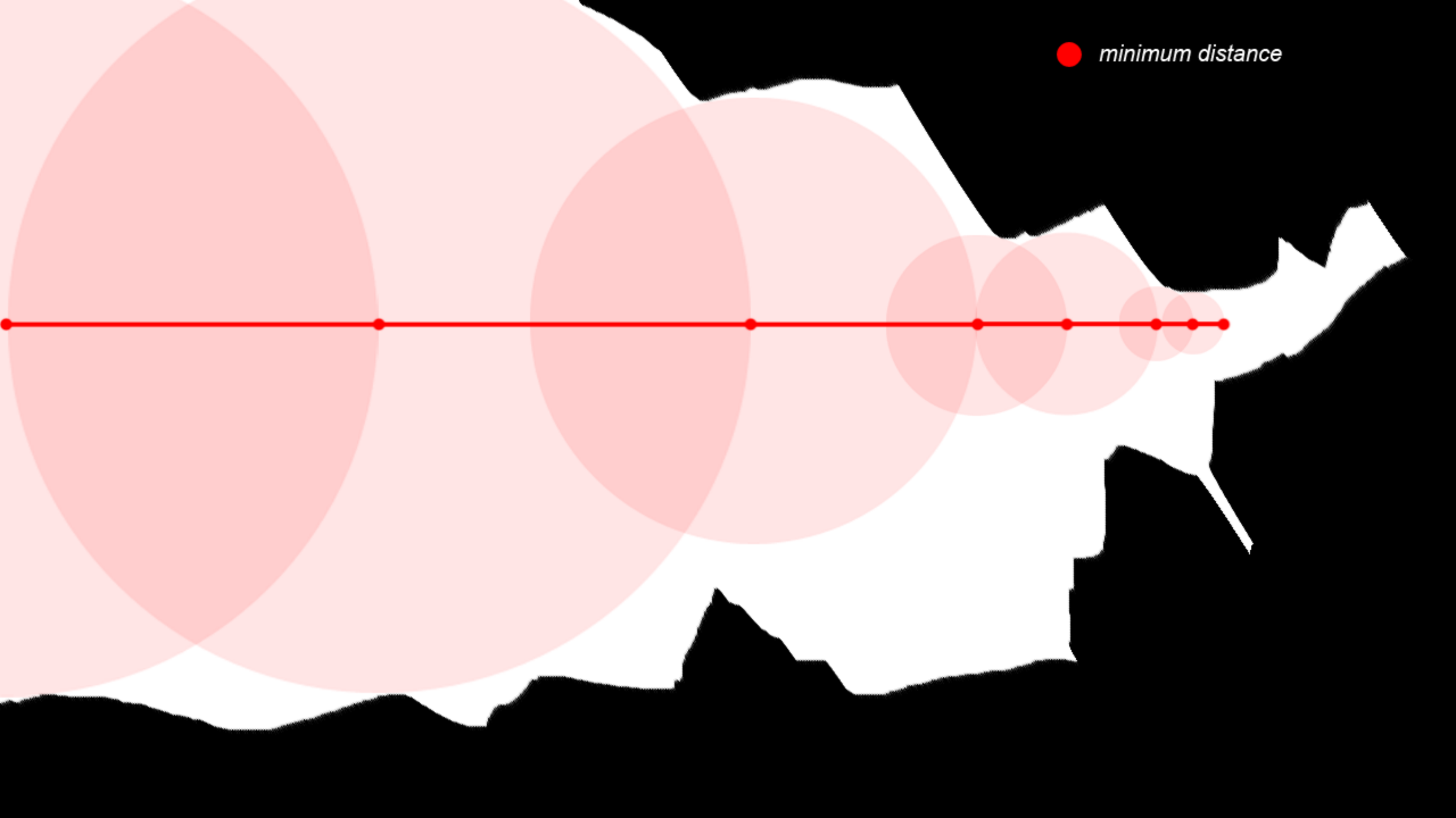


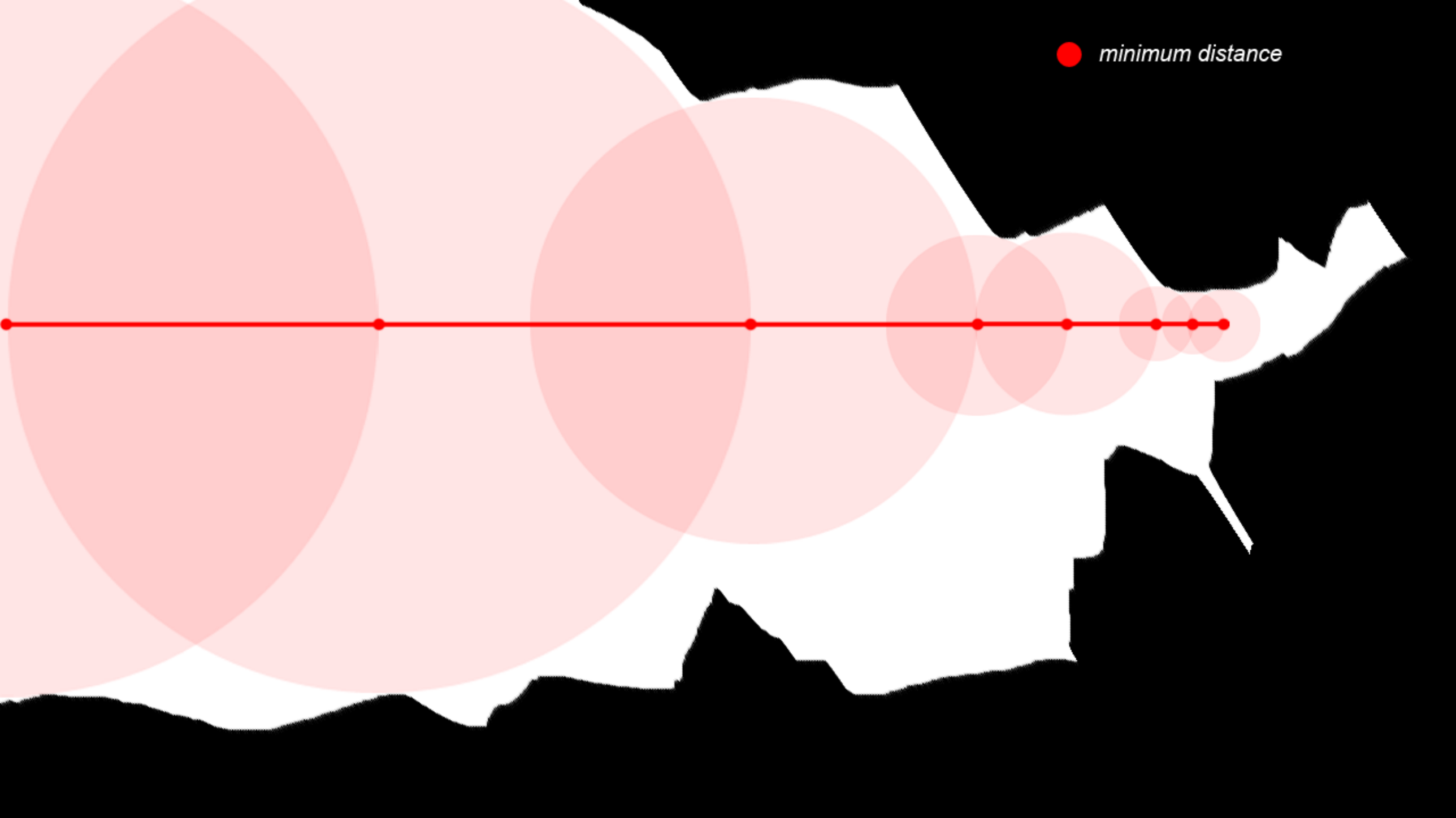

minimum distance

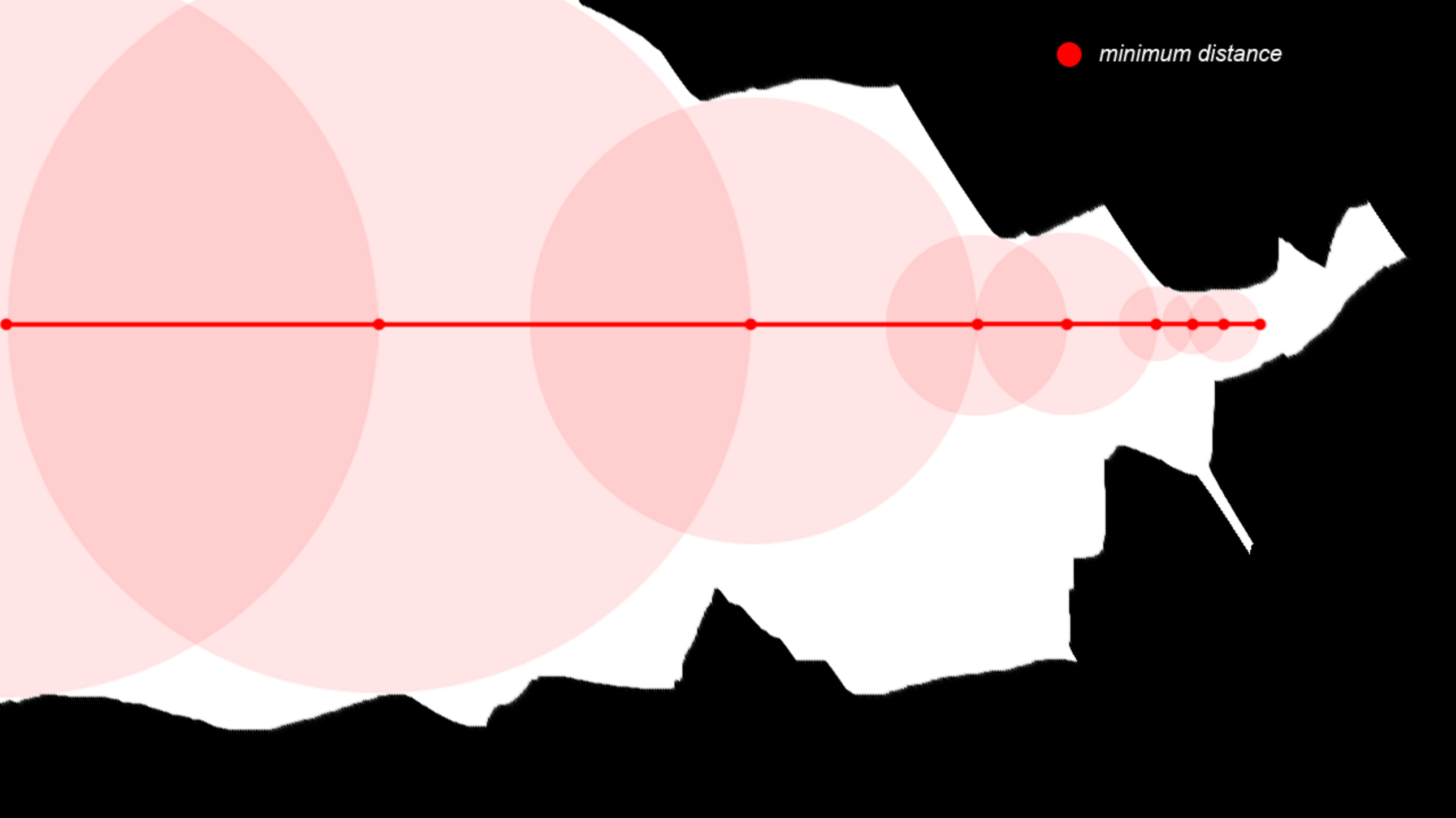
Raymarching mit Distance Fields

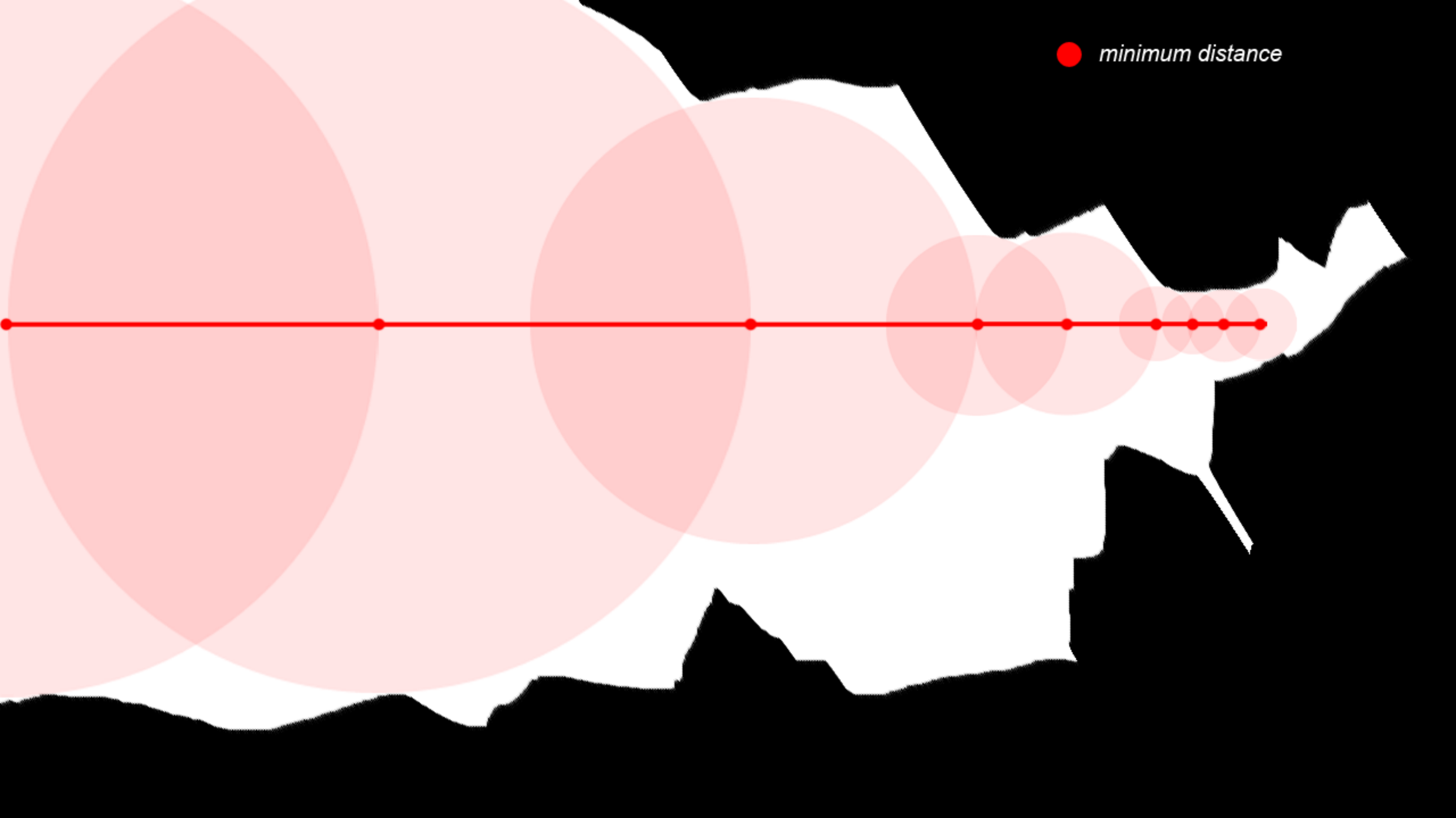
- Genauigkeit abhaengig von der Minimum Distance
- hoehere Genauigkeit = hoeherer Rechenaufwand
- Anzahl der Schritte steigt stark an

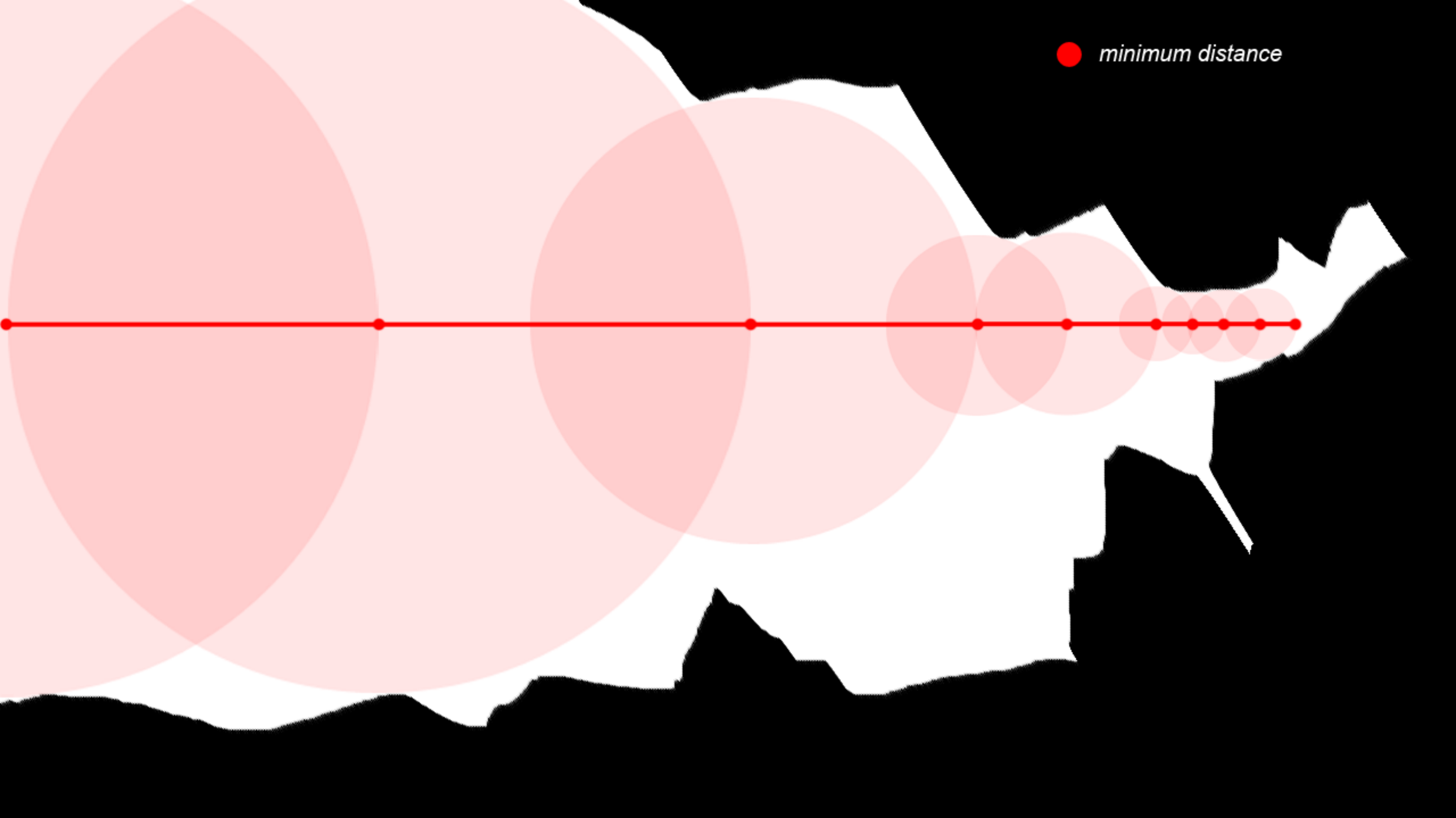

minimum distance

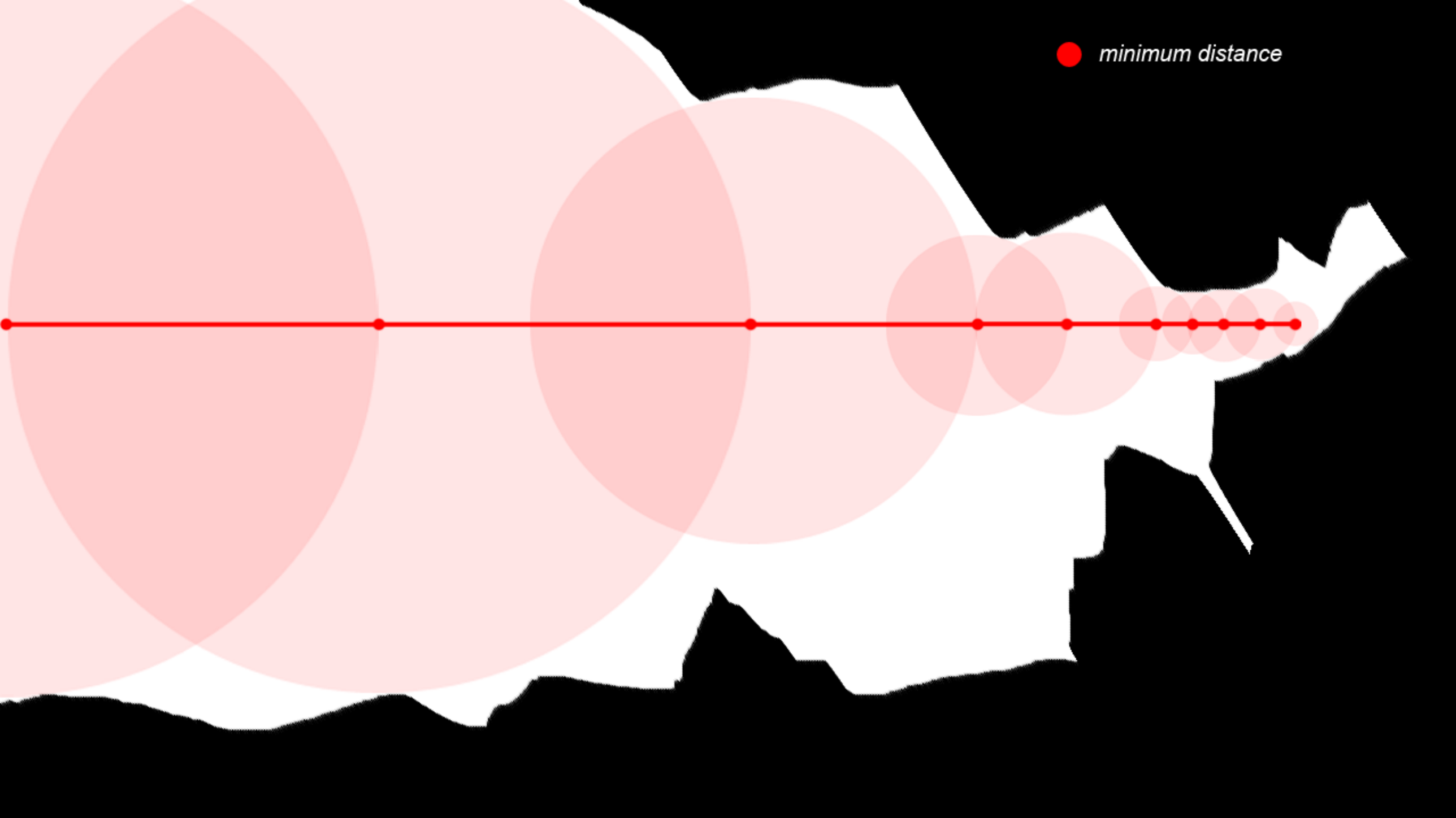

minimum distance

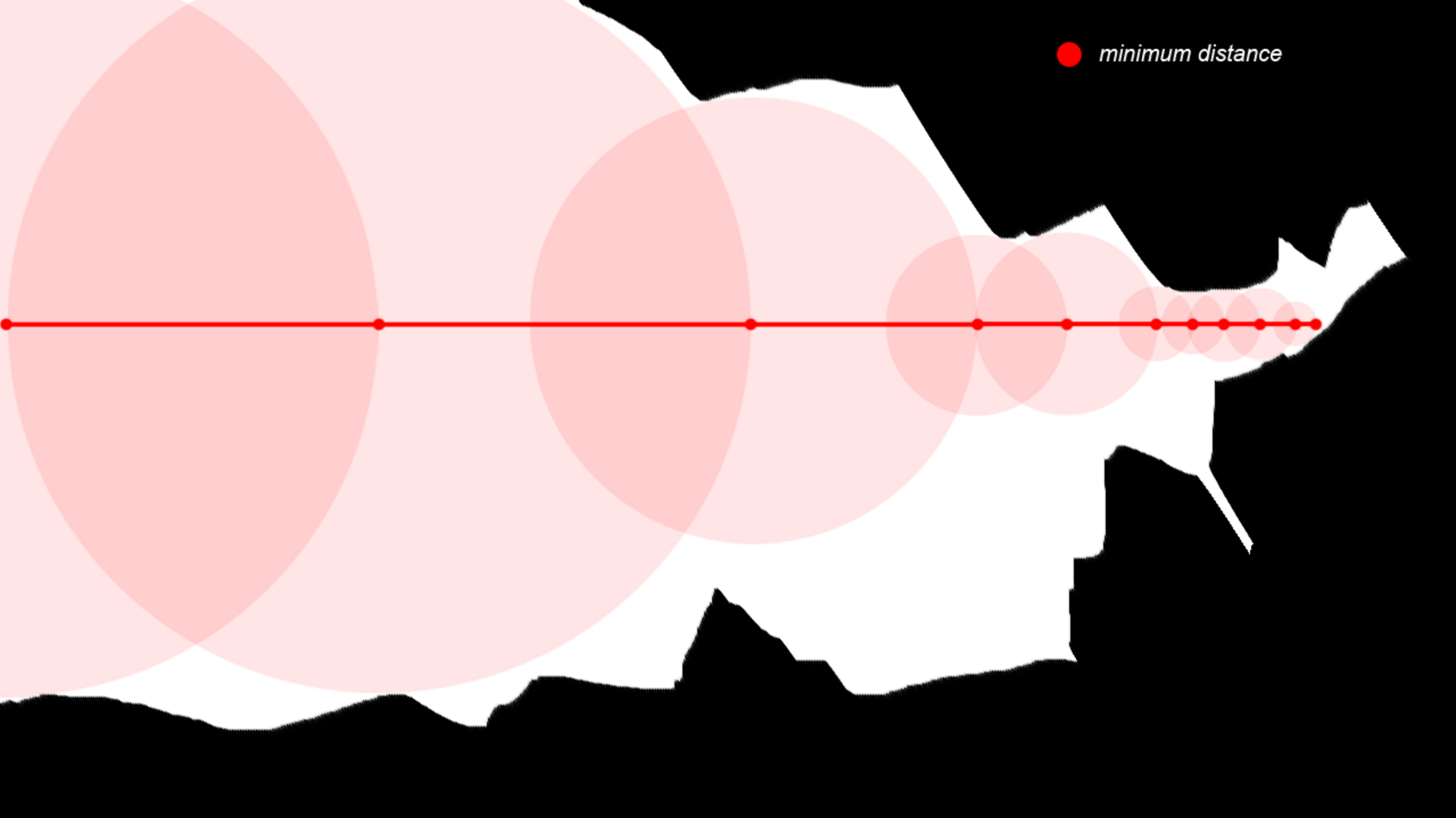

minimum distance


minimum distance


minimum distance


minimum distance


minimum distance


minimum distance

Raymarching mit Distance Fields

```
const float minimumDistance = 0.00001f;
const float maximumDistance = 1000.0f;

float getDistance(vec3 position, vec3 direction)
{
   float distance = DE_Geometrie(position);
   float totalDistance = distance;

   while (distance > minimumDistance && totalDistance < maximumDistance)
   {
      vec3 point = position + totalDistance * direction;
      distance = DE_Geometrie(point);
      totalDistance += distance;
   }
   return totalDistance;
}
```

Raymarching mit Distance Fields

```
const float minimumDistance = 0.00001f;
const float maximumDistance = 1000.0f;

float getDistance(vec3 position, vec3 direction)
{
   float distance = DE_Geometrie(position);
   float totalDistance = distance;

   while (distance > minimumDistance && totalDistance < maximumDistance)
   {
      vec3 point = position + totalDistance * direction;
      distance = DE_Geometrie(point);
      totalDistance += distance;
   }
   return totalDistance;
}
```

Raymarching mit Distance Fields

```
const float minimumDistance = 0.00001f;
const float maximumDistance = 1000.0f;

float getDistance(vec3 position, vec3 direction)
{
   float distance = DE_Geometrie(position);
   float totalDistance = distance;

   while (distance > minimumDistance && totalDistance < maximumDistance)
   {
      vec3 point = position + totalDistance * direction;
      distance = DE_Geometrie(point);
      totalDistance += distance;
   }
   return totalDistance;
}
```

Raymarching mit Distance Fields

```
const float minimumDistance = 0.00001f;
const float maximumDistance = 1000.0f;

float getDistance(vec3 position, vec3 direction)
{
   float distance = DE_Geometrie(position);
   float totalDistance = distance;

   while (distance > minimumDistance && totalDistance < maximumDistance)
   {
      vec3 point = position + totalDistance * direction;
      distance = DE_Geometrie(point);
      totalDistance += distance;
   }
   return totalDistance;
}
```

Raymarching mit Distance Fields

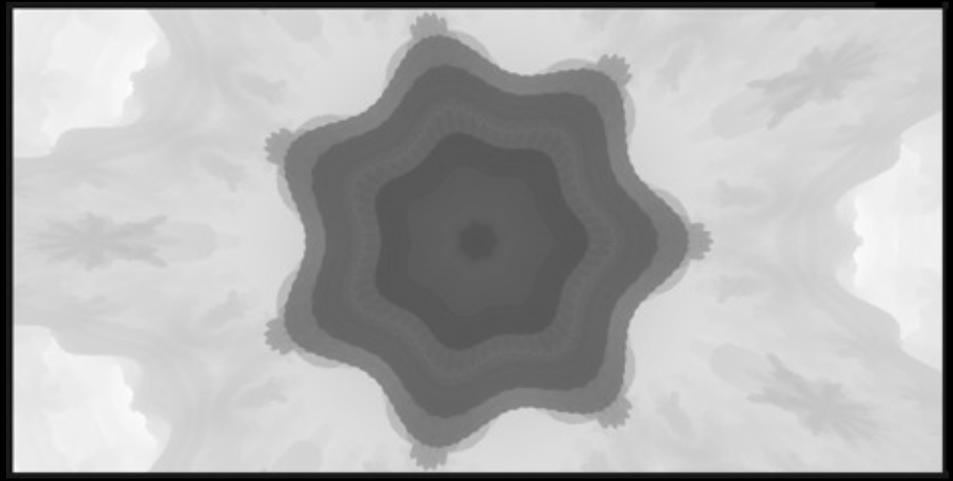
```
const float minimumDistance = 0.00001f;
const float maximumDistance = 1000.0f;

float getDistance(vec3 position, vec3 direction)
{
   float distance = DE_Geometrie(position);
   float totalDistance = distance;

   while (distance > minimumDistance && totalDistance < maximumDistance)
   {
      vec3 point = position + totalDistance * direction;
      distance = DE_Geometrie(point);
      totalDistance += distance;
   }
   return totalDistance;
}
```

Raymarching mit Distance Fields

```
const float minimumDistance = 0.00001f;
const float maximumDistance = 1000.0f;

float getDistance(vec3 position, vec3 direction)
{
   float distance = DE_Geometrie(position);
   float totalDistance = distance;

   while (distance > minimumDistance && totalDistance < maximumDistance)
   {
      vec3 point = position + totalDistance * direction;
      distance = DE_Geometrie(point);
      totalDistance += distance;
   }
   return totalDistance;
}
```

Raymarching mit Distance Fields

- liefert ein Tiefenbild
- die Oberflaechenbeschaffenheit laesst sich aus den Tiefeninformationen rekonstruieren



Raymarching mit Distance Fields

PROBLEM:

- keine dynamischen Schleifen in WebGL Shadern !


```
while (distance > minimumDistance && totalDistance < maximumDistance)
{
  ...
}
```

Raymarching mit Distance Fields

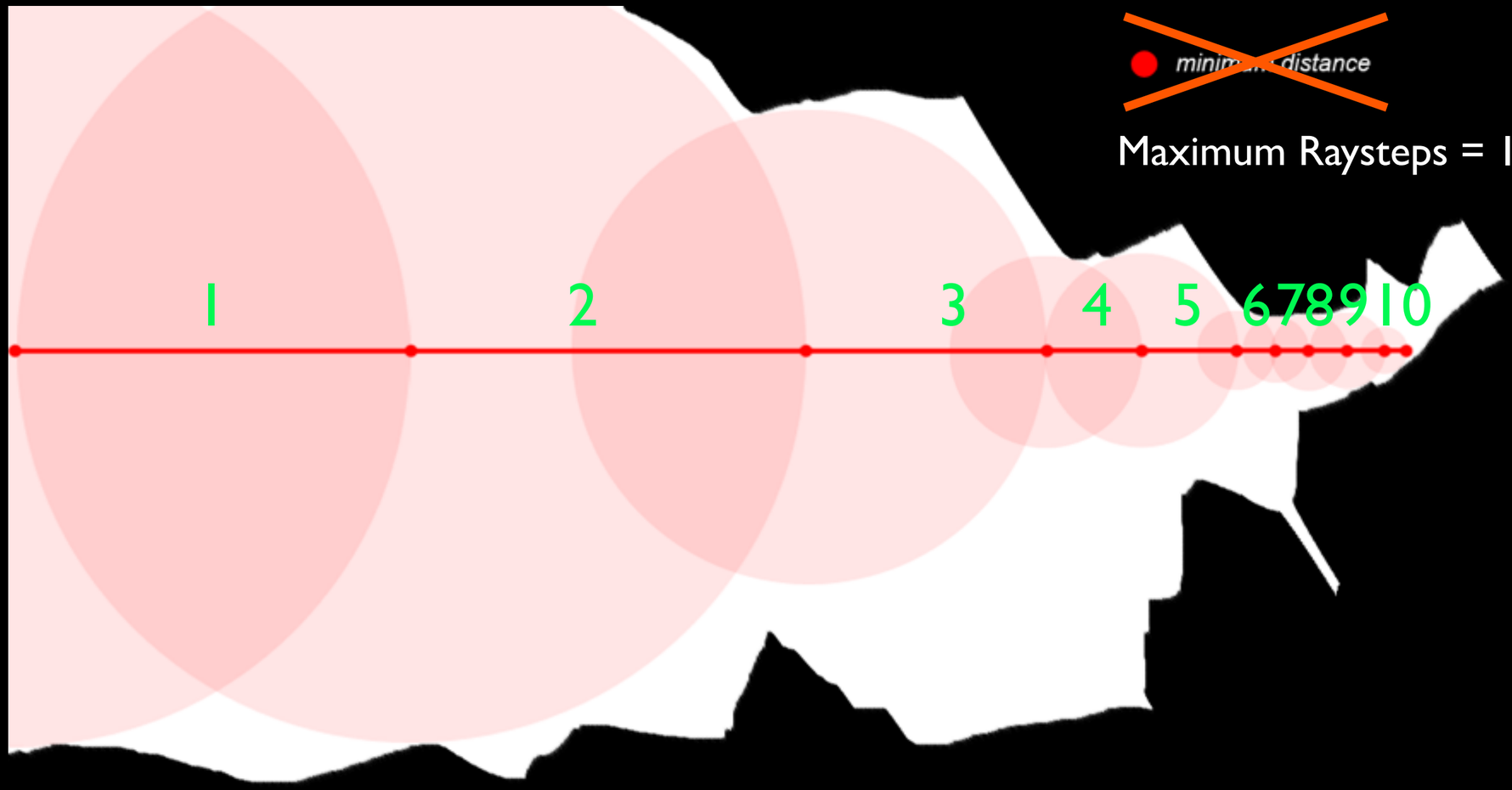
LOESUNG:

- Anzahl der Rayschritte als Abbruchbedingung

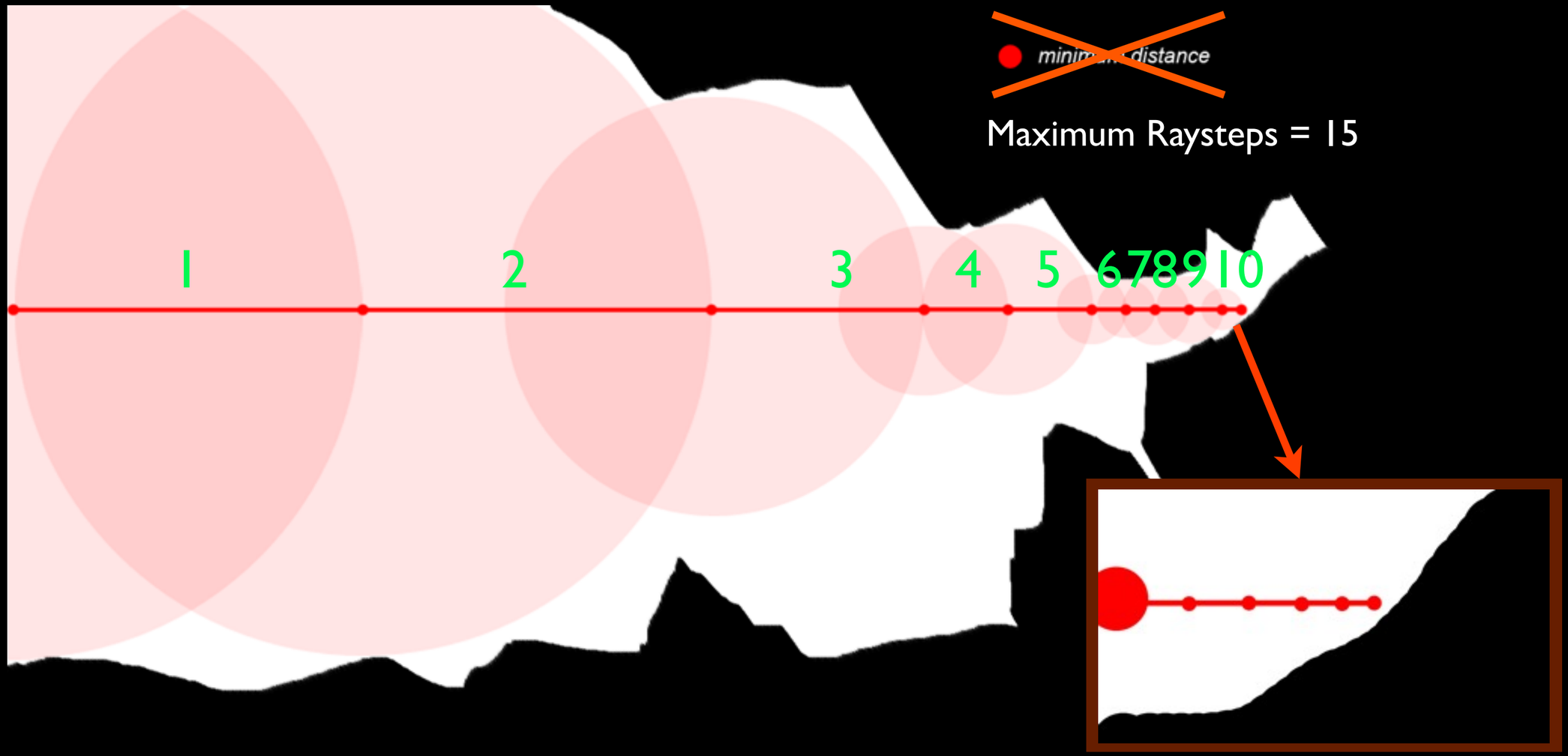
konstant

```
for (step = 0; step < maximumRaysteps; step++)
{
  ...
}
```

Raymarching mit Distance Fields



Raymarching mit Distance Fields



Raymarching mit Distance Fields

LOESUNG:

- Anzahl der Rayschritte als Abbruchbedingung
- funktioniert auf neuerer Hardware:
  - versteckte dynamische Abbruchbedingung
  - minimum Distance als zusaetzliche Abbruchbedingung

```
for (step = 0; step < maximumRaysteps; step++)
{
  ...
  if (distance < minimumDistance) break; // Treiber/GPU abhaengig!
}
```

Raymarching mit Distance Fields

OPTIMIERUNG:

- Clarity Function [Hart89] `f(d) = α*d^ß`

`α = Skalierung`
`d = Abstand`
`ß = Exponent`

- `minimumDistance` abhaengig von `totalDistance`:

```
dynMinDistance = minimumDistance * totalDistance * 10.0;
```

Raymarching mit Distance Fields

OPTIMIERUNG:

- Clarity Function [Hart89] `f(d) = α*d^ß`

`α = Skalierung`
`d = Abstand`
`ß = Exponent`

- `minimumDistance` abhaengig von `totalDistance`:

```
dynMinDistance = minimumDistance * totalDistance * 10.0;
```

> weniger Details auf weit entfernten Oberflaechen (Level of Detail)
> weniger Aliasing
> bessere Performance

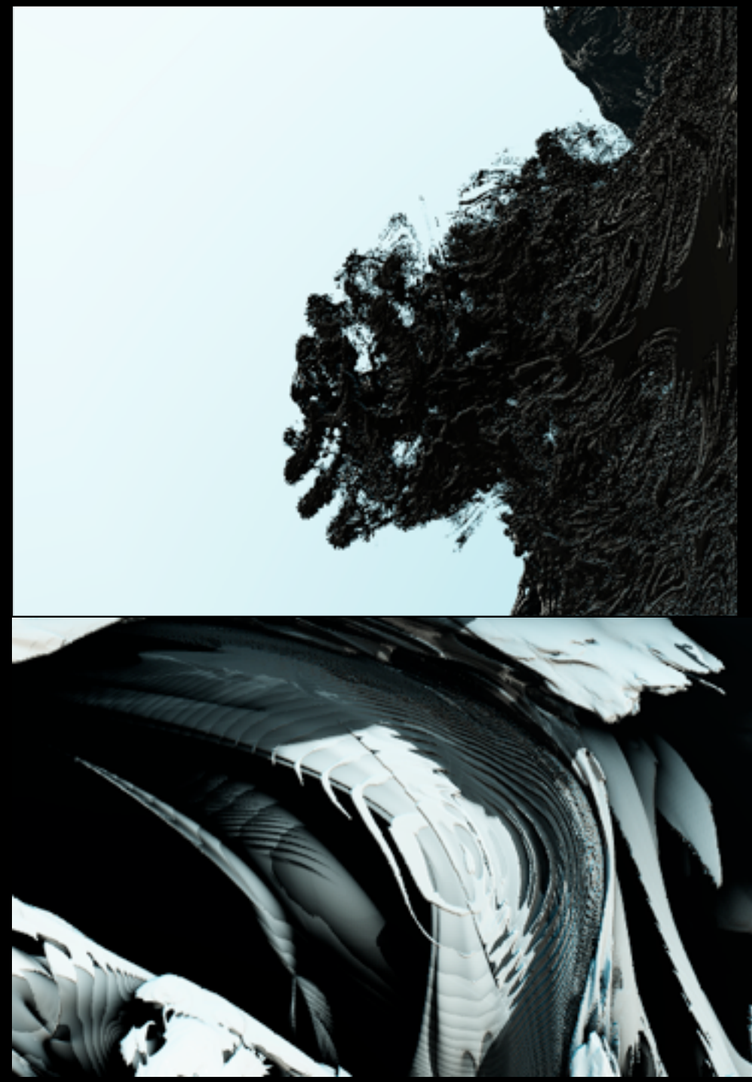
Raymarching mit Distance Fields

ARTEFAKTE:

- Overstepping
  - Ungenauigkeit bei der Fraktal Distance Estimation

  - besonders auffaellig bei der Schattenberechnung

  => der Ray durchdringt die Oberflaeche



Raymarching mit Distance Fields

ARTEFAKTE:

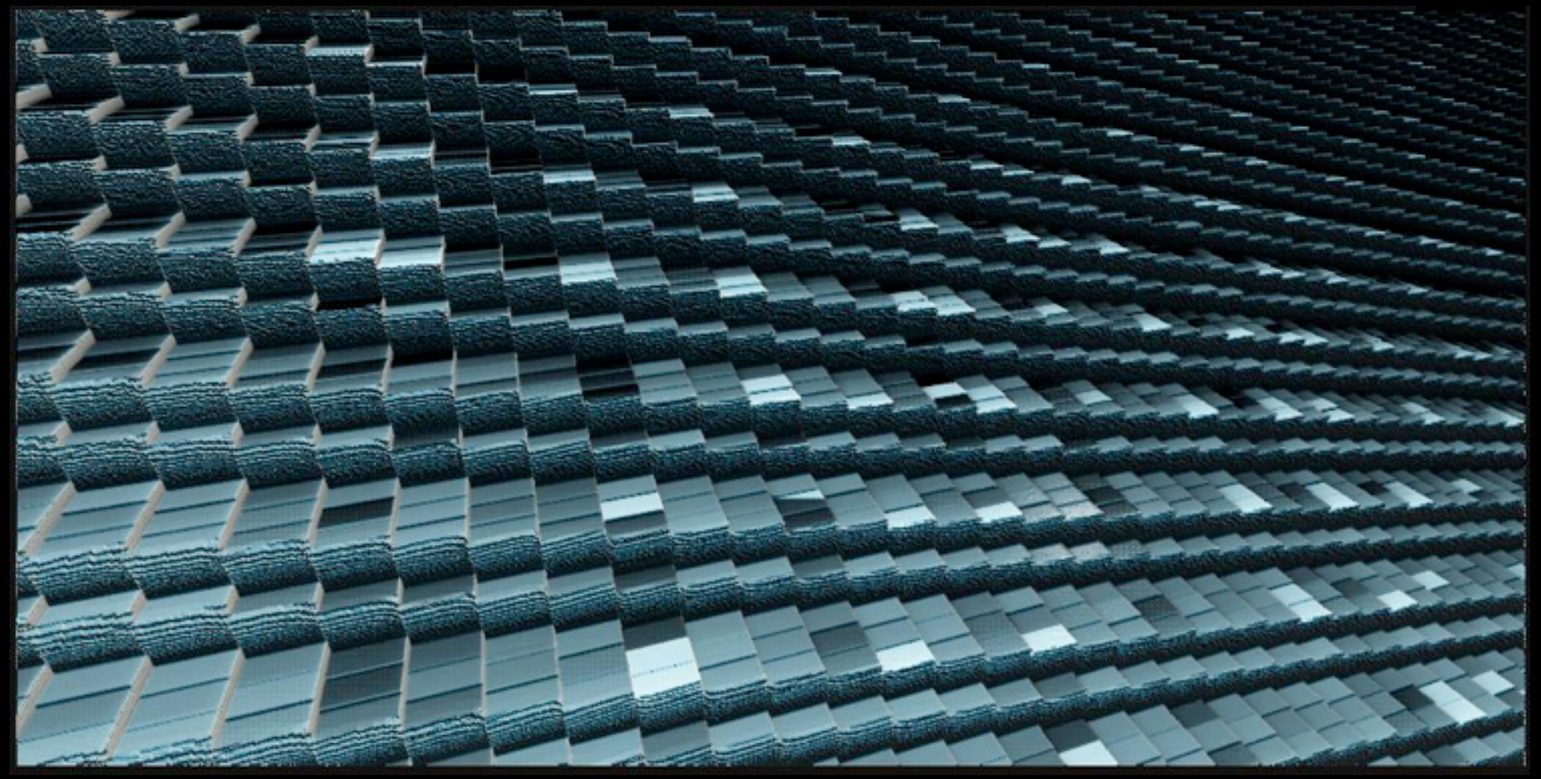
- Erreichen von Genauigkeitsgrenzen

=> Erreichen der float Genauigkeit

=> begrenzte Genauigkeit des Distance Estimators durch die `minimumDistance`

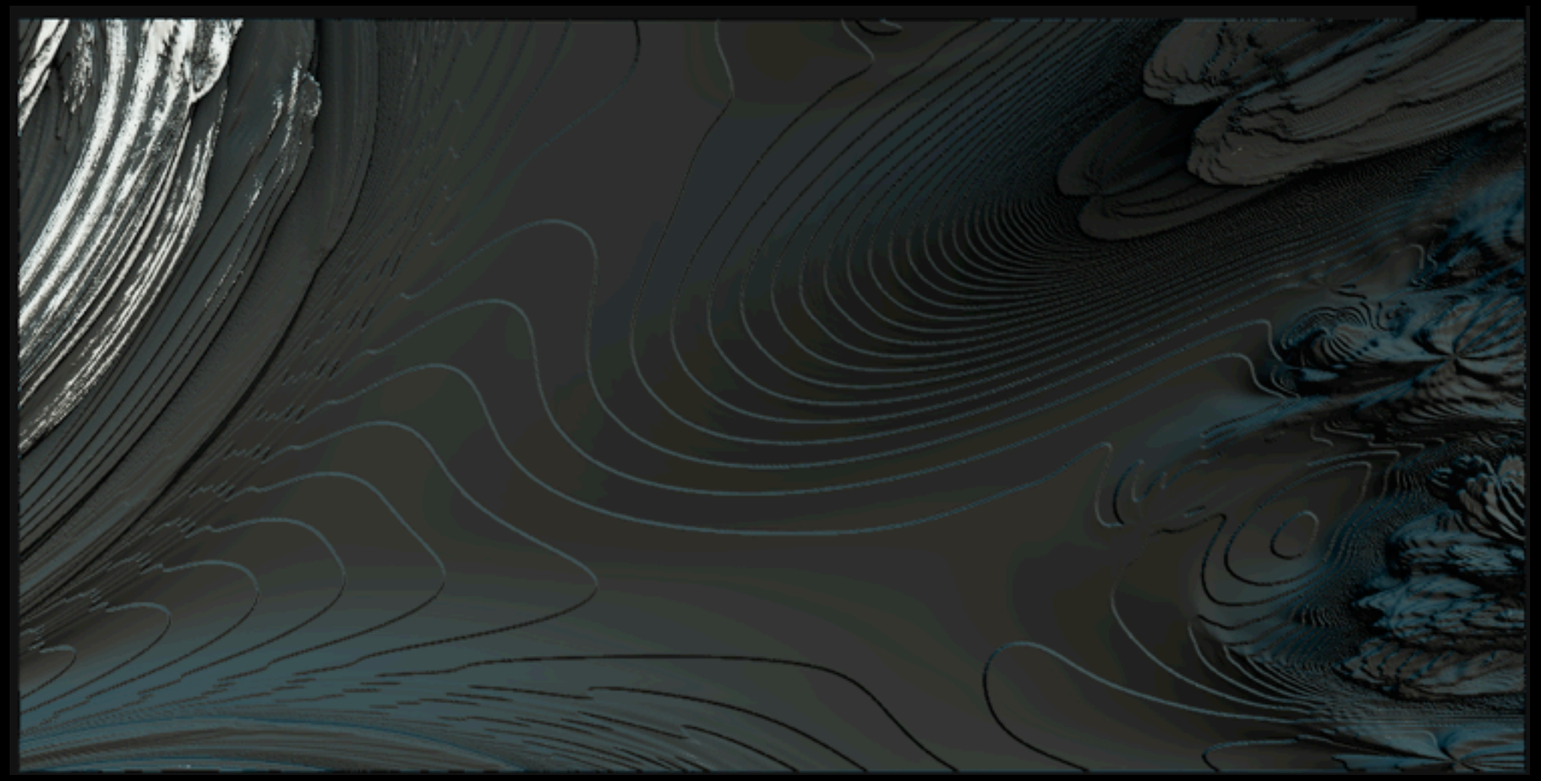
Raymarching mit Distance Fields

- sichtbare float Genauigkeit sehr dicht an der Oberflaeche



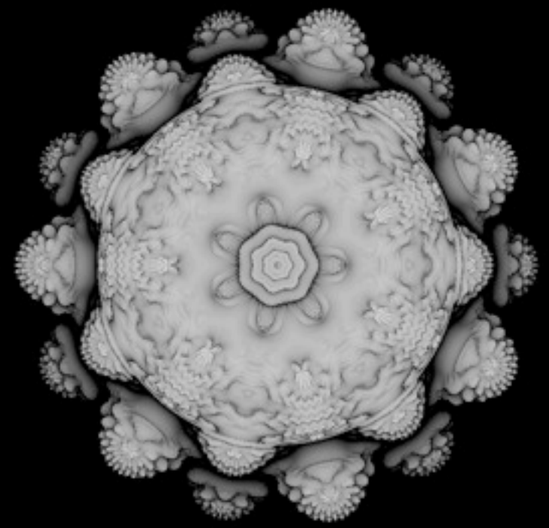
Raymarching mit Distance Fields

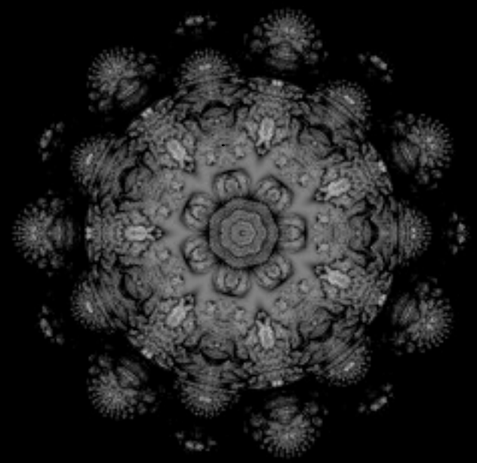
- sichtbare Ungenauigkeit des Distance Estimators (Scheibenbildung)



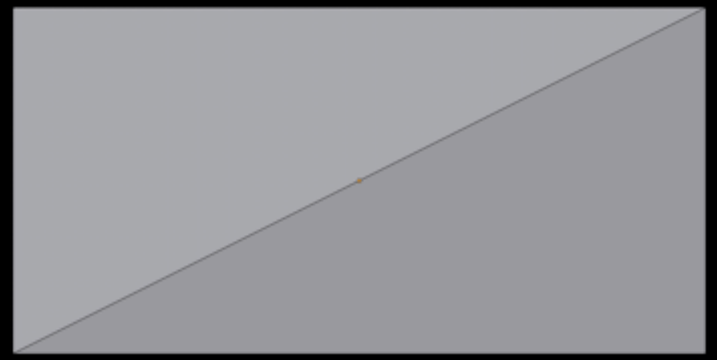
Implementierung

- Erster Softwareprototyp in C# (CPU)
  - orthogonale Projektion (1 Ray pro Pixel)
  - Visualisierung der Schritte pro Pixel als Grauwert
  - nicht parallelisiert (1 Thread)
  - nicht interaktiv
  - Rechenzeit (512 x 512px) ~20 min (0,0003 fps)
    CPU : i7 870 @ 3,4 GHz (4 Cores / 8 Threads)



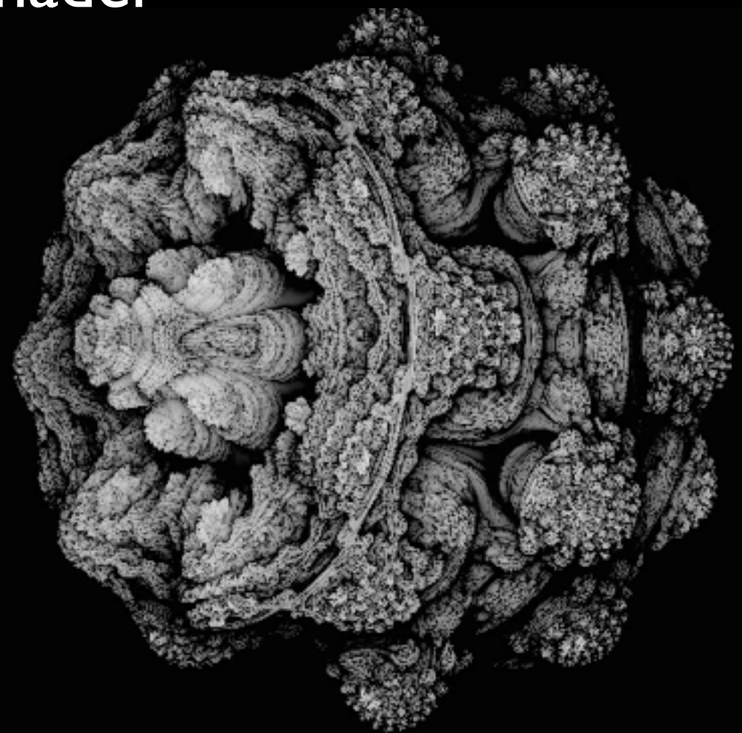


Implementierung

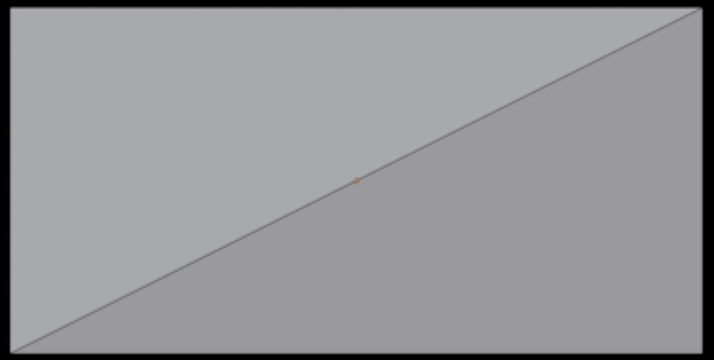


- Implementierung in GLSL (Fragment Shader)
  - 1 Quad bildschirmfuellend dargestellt (4 Vertices)
  - Berechnung erfolgt ausschliesslich im Fragment Shader
  - 1 Ray pro Fragment parallel (512 GPU Cores)
  - Interaktiv
  - Rechenzeit (512 x 512px) < 16ms (>60fps)

    GPU : NVidia GTX 580 (512 Cuda Cores)

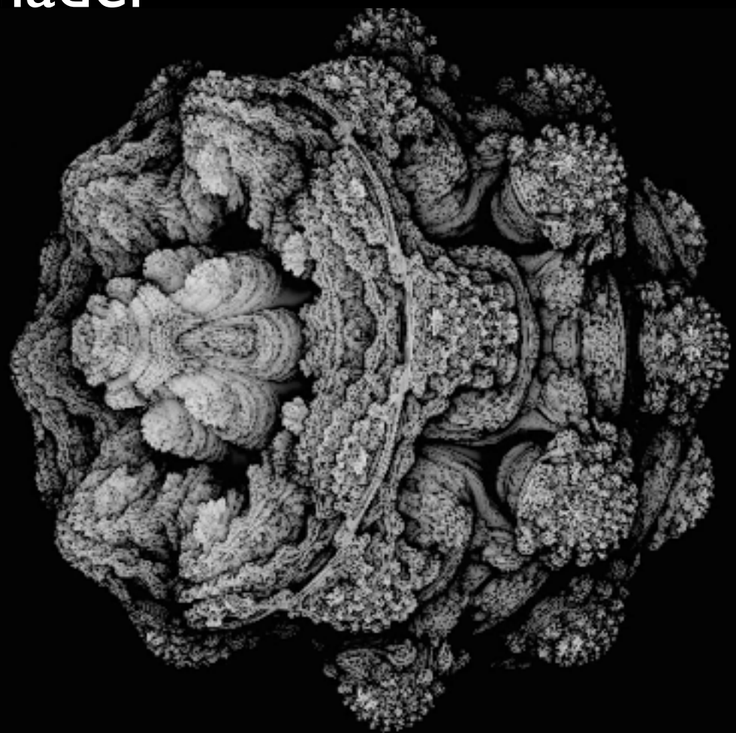


Implementierung



- Implementierung in GLSL (Fragment Shader)
  - 1 Quad bildschirmfuellend dargestellt (4 Vertices)
  - Berechnung erfolgt ausschliesslich im Fragment Shader
  - 1 Ray pro Fragment parallel (512 GPU Cores)
  - Interaktiv
  - Rechenzeit (512 x 512px) < 16ms (>60fps)

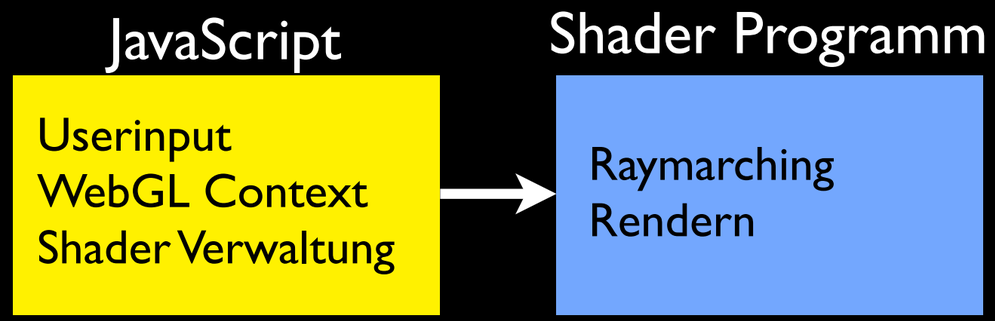
    GPU : NVidia GTX 580 (512 Cuda Cores)



VIDEO

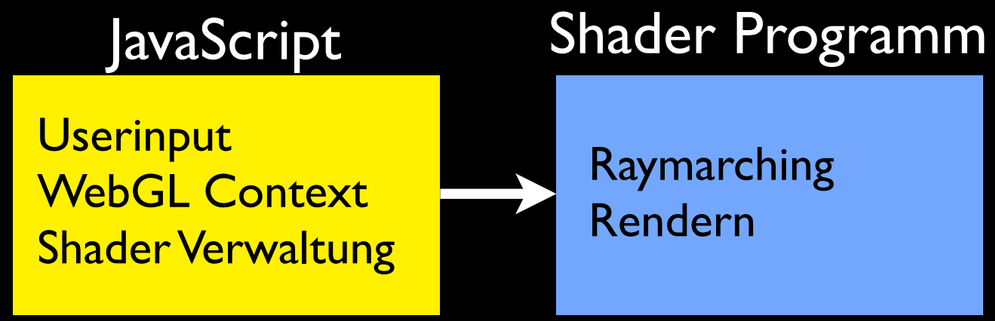
Implementierung - Entwicklung

Erster Entwurf



Implementierung - Entwicklung

Erster Entwurf



Beleuchtung / Schattierung?

Implementierung - Entwicklung

- Ambient Occlusion :
  - Schattierung durch Umgebungsverdeckung
  - je mehr Geometrie um einen Punkt exisitiert, desto dunkler wird dieser dargestellt
  - approximiert indirekte Beleuchtung

Implementierung - Entwicklung

- Ambient Occlusion :
  - aehnlicher Effekt beim Raymarching
  - praktisch kostenloses AO durch Anzahl der Raysteps

Implementierung - Entwicklung

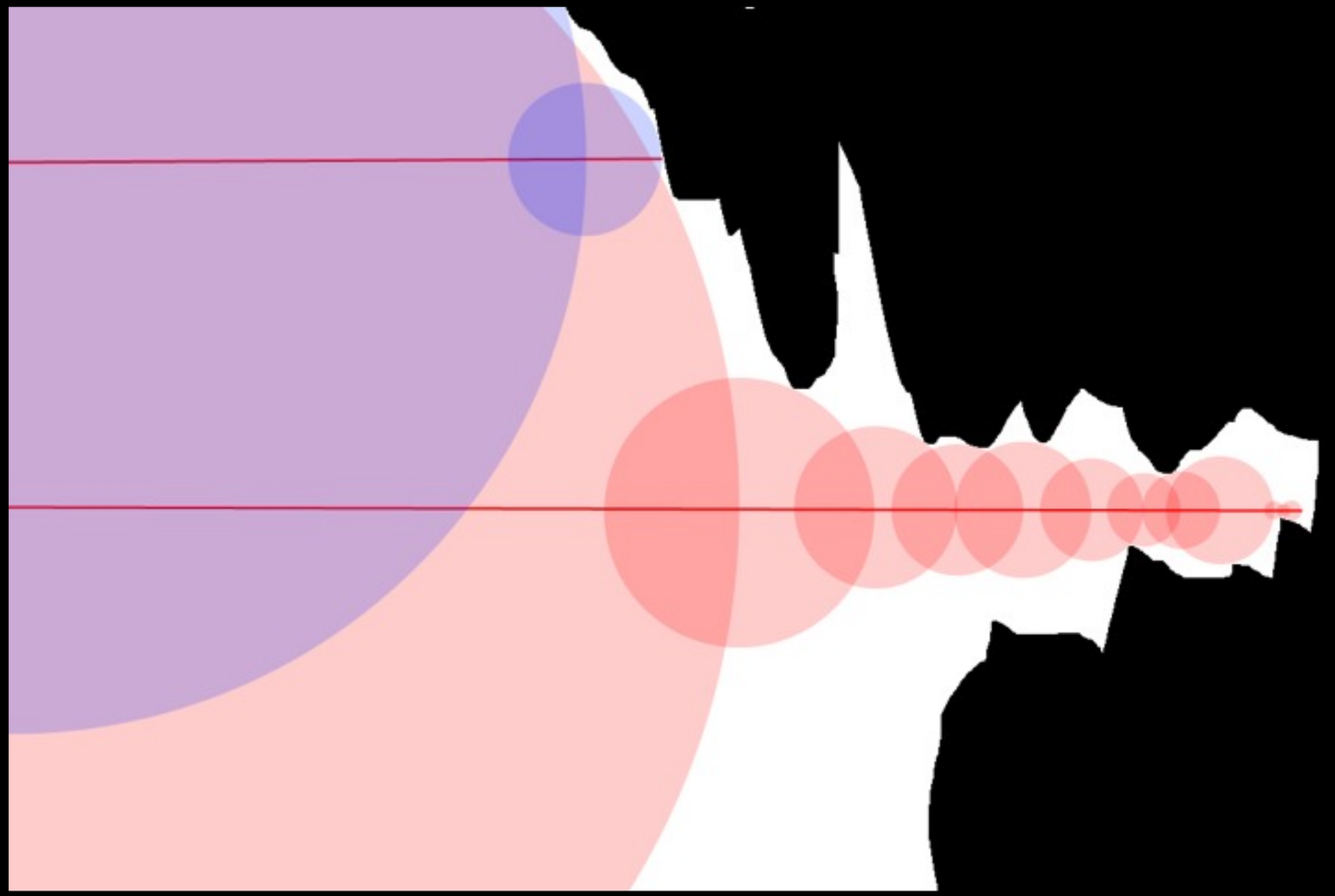
- Ambient Occlusion :
  - aehnlicher Effekt beim Raymarching
  - praktisch kostenloses AO durch Anzahl der Raysteps

Warum?

Implementierung - Entwicklung

knapp an Geometrie vorbei
=> verdeckt
=> viele Schritte

direkt auf Oberflaeche
=> nicht verdeckt
=> wenig Schritte



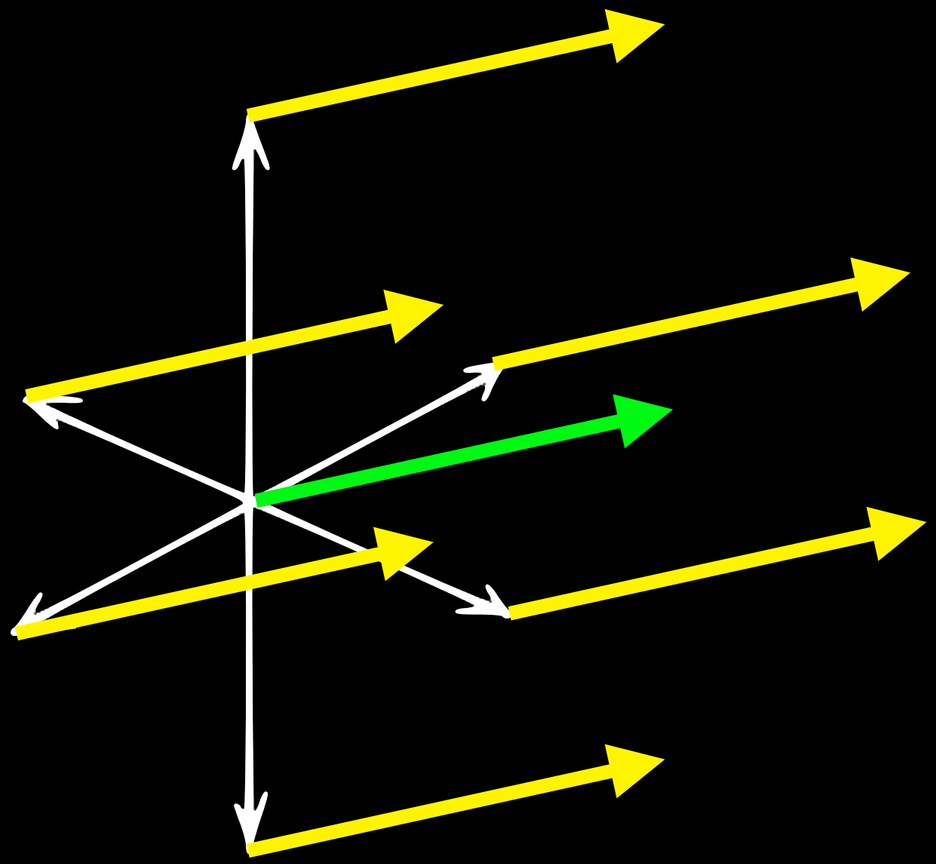
Implementierung - Entwicklung

- Implementierung von Phong Shading
  - Ambient (Umgebungslicht)
  - Diffuse (Lambert)
  - Specular (Glanz)

=> Oberflaechennormalen werden benoetigt

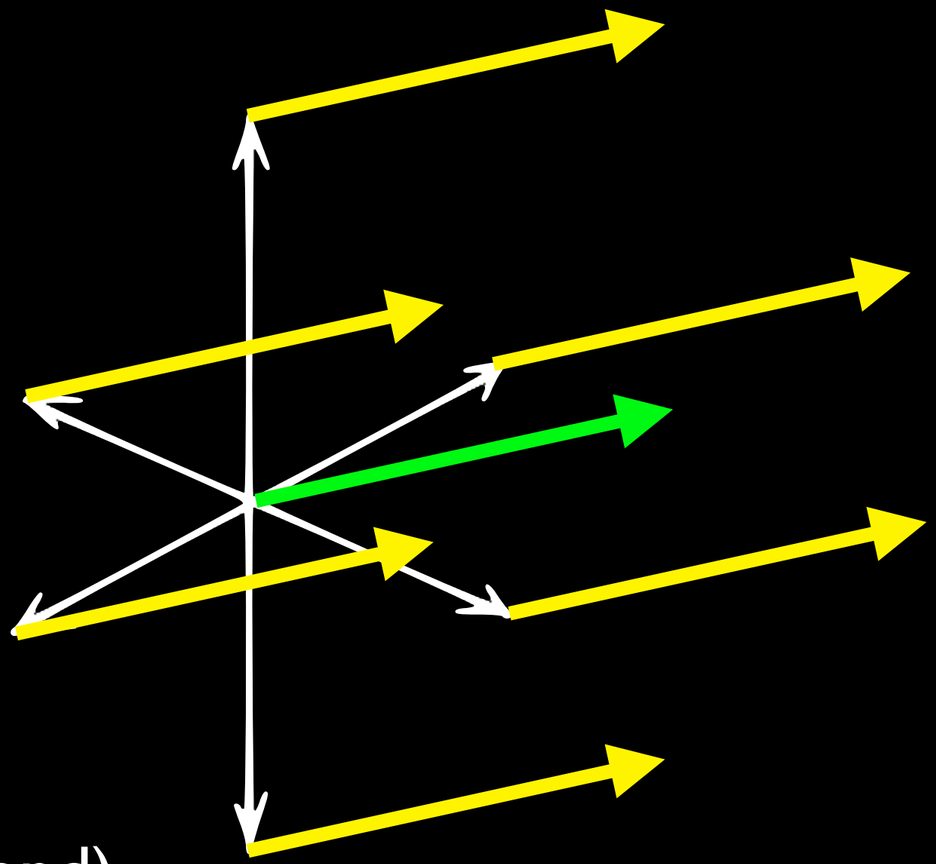
Berechnung der Normalen

- Erster Ansatz :
  - pro Ray 6 weitere Rays losgeschickt
  - jeweils um ein delta in alle Richtungen verschoben
  - Berechnung der Steigung durch Umgebungspunkte



Berechnung der Normalen

- Erster Ansatz :
  - pro Ray 6 weitere Rays losgeschickt
  - jeweils um ein delta in alle Richtungen verschoben
  - Berechnung der Steigung durch Umgebungspunkte



=> sehr genau, aber zu langsam (7 facher Aufwand)

Berechnung der Normalen

- Zweiter Ansatz
  - Ableitung der Tiefeninformationen in X- und Y-Richtung
  - `dFdx()` / `dFdy()` Funktionen in GLSL

Berechnung der Normalen

- Zweiter Ansatz
  - Ableitung der Tiefeninformationen in X- und Y-Richtung
  - `dFdx()` / `dFdy()` Funktionen in GLSL

=> schnell, aber zu ungenau, da automatische blockweise Berechnung

VIDEO

Berechnung der Normalen

- Dritter Ansatz :
  - Berechnung der Normalen aus den umliegenden Tiefenwerten
  - Zugriff auf benachbarte Pixel noetig

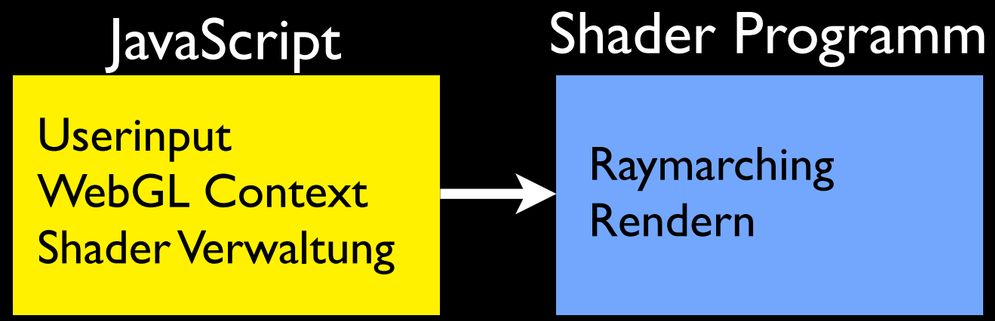
=> Tiefeninformation in Textur rendern

=> 16 bit float Rendertarget fuer hoeher Praezision (Extension)

=> Umstellung des Programmablaufs

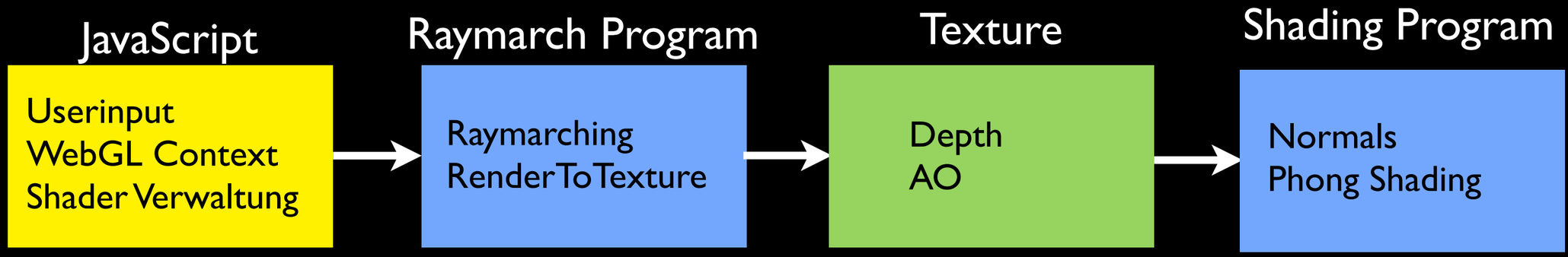
Implementierung - Entwicklung

Erster Entwurf



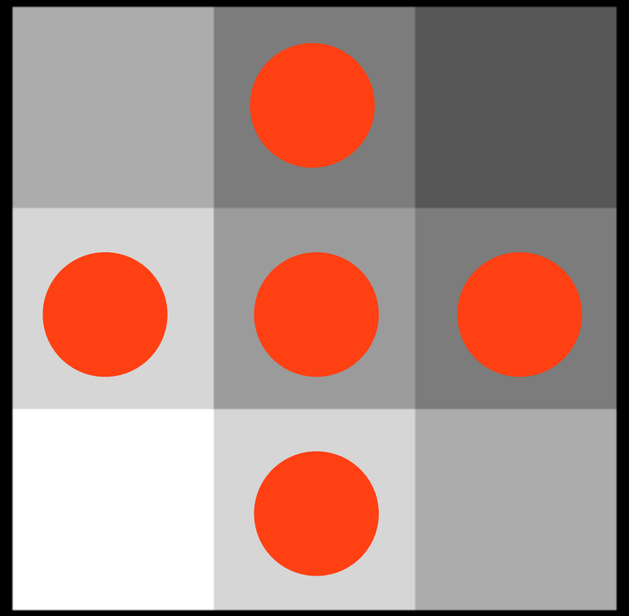
Implementierung - Entwicklung

<u>Zweiter Entwurf</u>



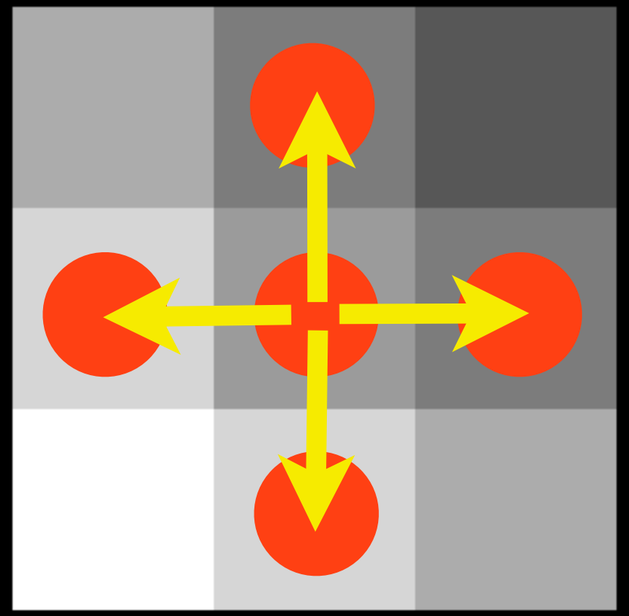
Implementierung - Entwicklung

- Berechnung der Normalen :
  - Tiefensample des aktuellen Fragments
  - 4 umliegende Tiefensamples
  - Rekonstruktion der Weltpositionen



Implementierung - Entwicklung

- Berechnung der Normalen :
  - Tiefensample des aktuellen Fragments
  - 4 umliegende Tiefensamples
  - Rekonstruktion der Weltpositionen
  - Tangentenvektoren bilden
  - Kreuzprodukte bilden
    - Normal = Mittelwert der Kreuzprodukte



Implementierung - Entwicklung

OPTIMIERUNG :

- Speichern der Position anstatt der Tiefe (Viewspace)

=> Rekonstruktion der Postion aus der Tiefe entfaellt

  - Tiefe entspricht `length(position)`
  - Ray Richtung entspricht `normalize(position)`

Implementierung - Entwicklung

OPTIMIERUNG :

- Speichern der Position anstatt der Tiefe (Viewspace)

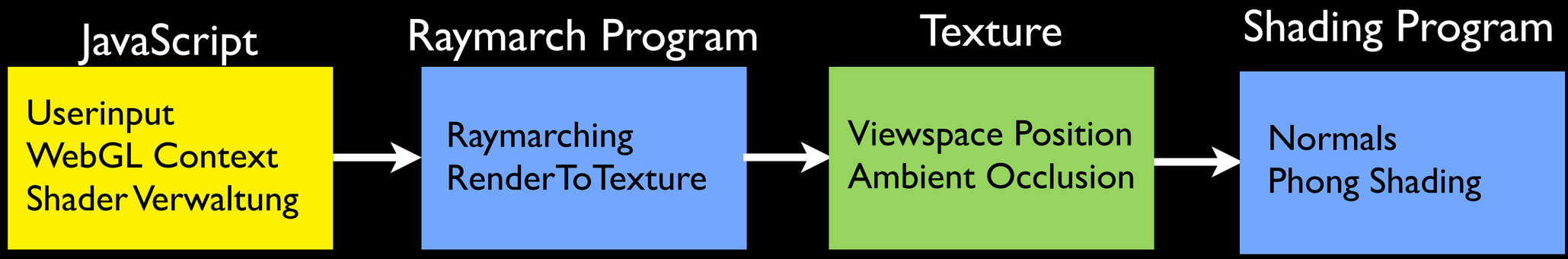
=> Rekonstruktion der Postion aus der Tiefe entfaellt

  - Tiefe entspricht `length(position)`
  - Ray Richtung entspricht `normalize(position)`

=> alle fuer Phong Shading erforderlichen Daten vorhanden!

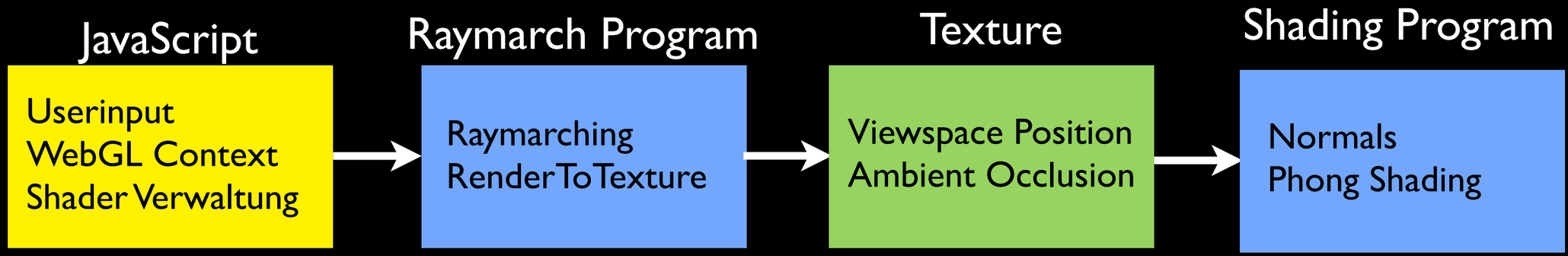
Implementierung - Entwicklung

<u>Dritter Entwurf</u>



Implementierung - Entwicklung

Dritter Entwurf

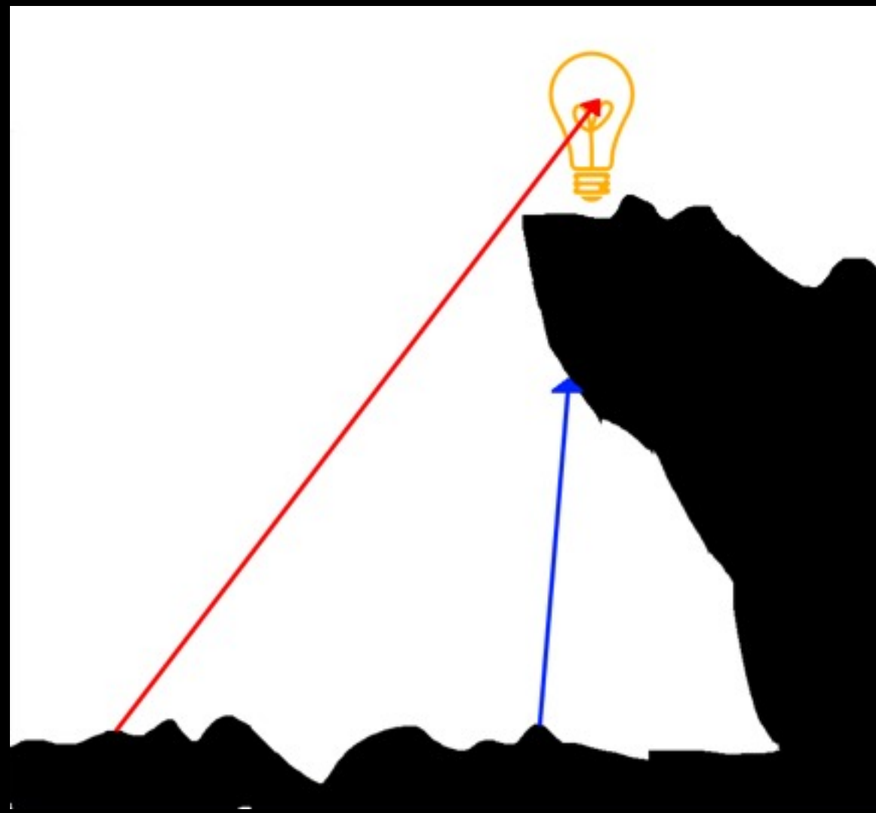


Schattenwurf?

Implementierung - Entwicklung

- Berechnung von Schlagschatten/Schattenwurf :
  - von der Oberflaeche ein Ray Richtung Lichtquelle

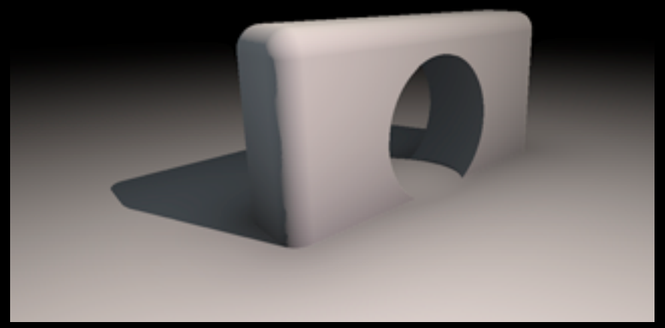
```
if ( length(ray) < distanceToLight )
{
    //Object is in Shadow
}
```



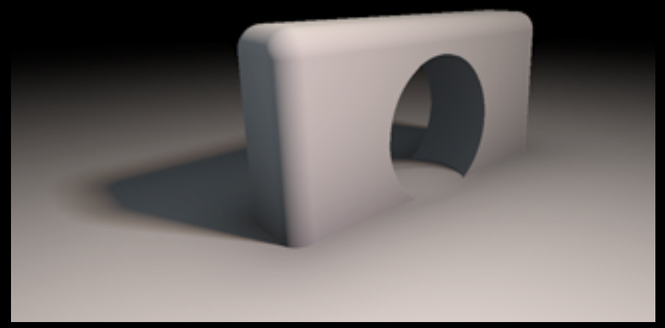
=> liegt ein Objekt auf dem Weg zur Lichtquelle, liegt der Punkt im Schatten

Implementierung - Entwicklung

- Berechnung von Schlagschatten/Schattenwurf :
  - weiche Schatten / Penumbra [Quilez10]

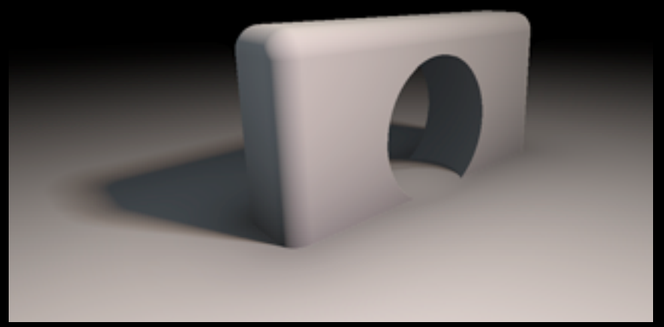


ohne Penumbra



mit Penumbra

Implementierung - Entwicklung



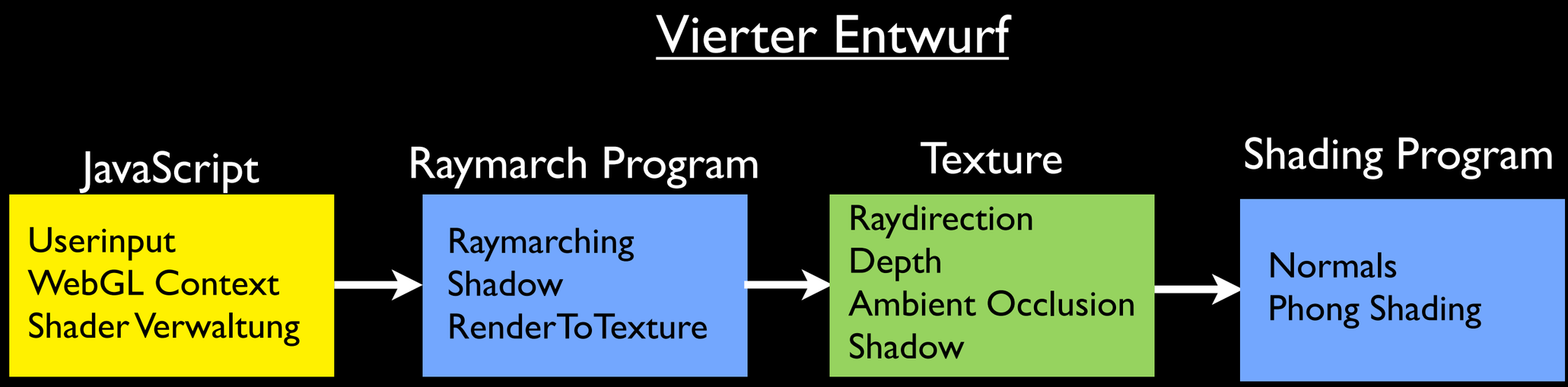
Penumbra hellt den Schatten an den Raendern auf
und berechnet sich wie folgt :

```
penumbra = min( penumbra, hardness * distance/totalDistance);
```

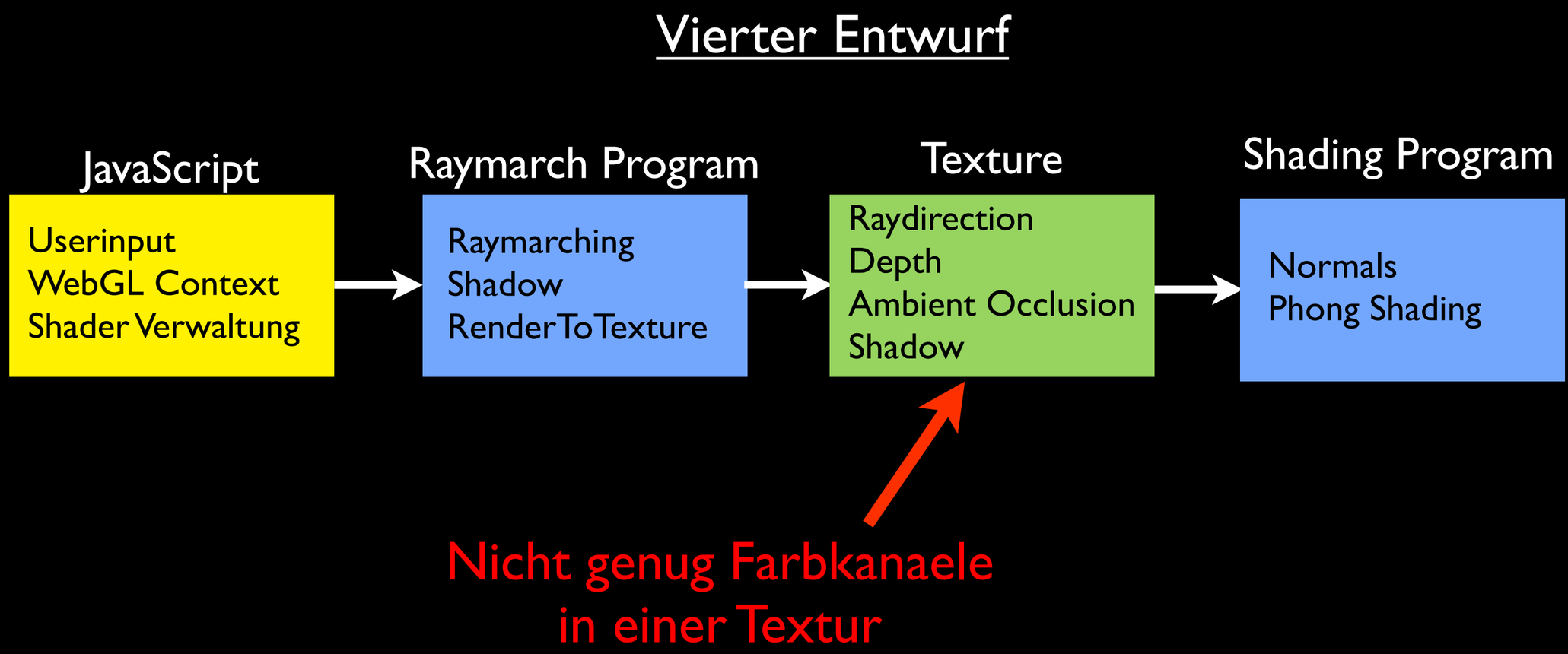
Je naeher der Strahl an der Geometrie vorbeigeht, desto dunkler ist der Schatten.

=> Naehe zur Geometrie durch Raymarching gegeben

Implementierung - Entwicklung



Implementierung - Entwicklung



Implementierung - Entwicklung

LOESUNG:

- Multiple Rendertargets
  (gleichzeitig Daten in verschiedene Texturen schreiben)

Implementierung - Entwicklung

LOESUNG:

- Multiple Rendertargets
  (gleichzeitig Daten in verschiedene Texturen schreiben)

=> wird von WebGL nicht unterstuetzt

Implementierung - Entwicklung

NEUE LOESUNG:

- aggressives Packen der Daten in die einzelnen Kanaele

Daten pro Pixel:

```
float ambientOcclusion
float shadow
vec3 raydirection
float depth
```

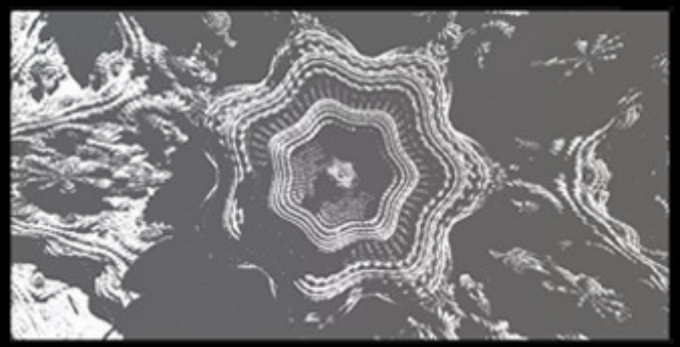
=> Speichern in die RGBA Kanaele

Implementierung - Entwicklung

NEUE LOESUNG:

- aggressives Packen der Daten in die einzelnen Kanaele
=> Multiplizieren von Schatten und Ambient Occlusion

shadowing

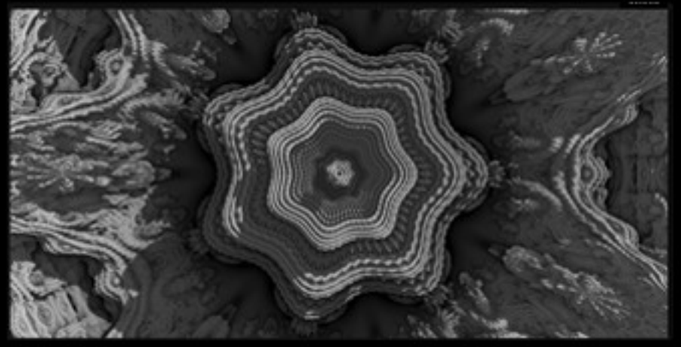


X

ambient occlusion

=

premultplied shadows



Implementierung - Entwicklung

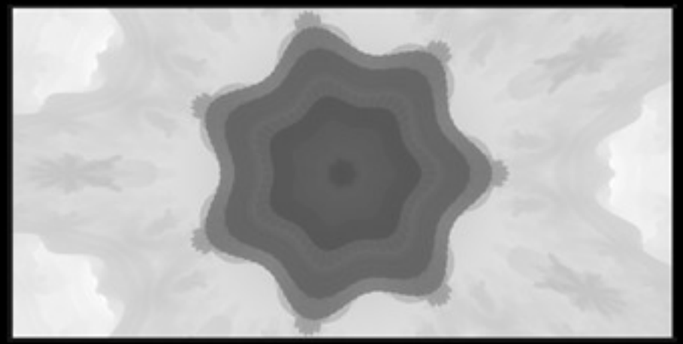
NEUE LOESUNG:

- aggressives Packen der Daten in die einzelnen Kanaele
=> normalisierte Strahlrichtung mit Tiefe multiplizieren

normalize (ray direction)

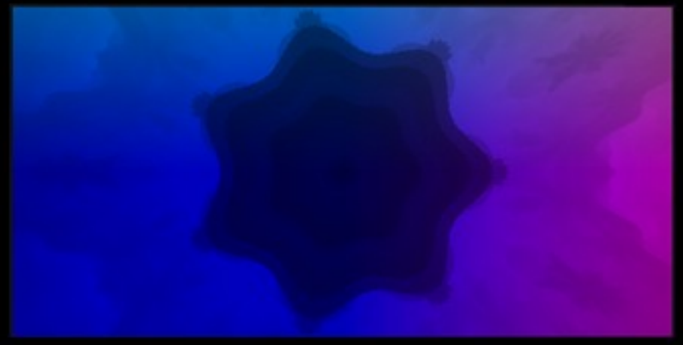
X

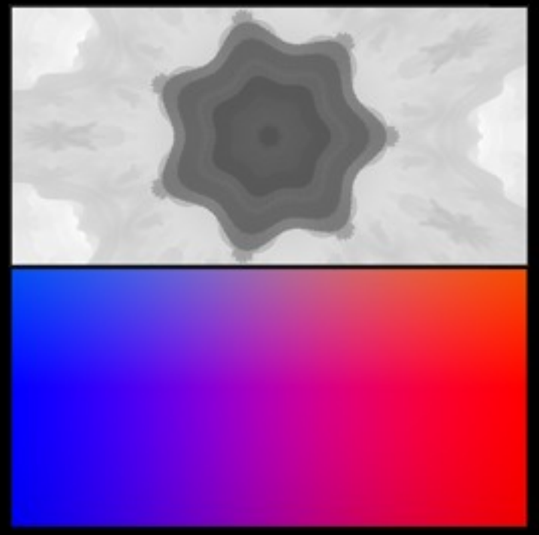
depth

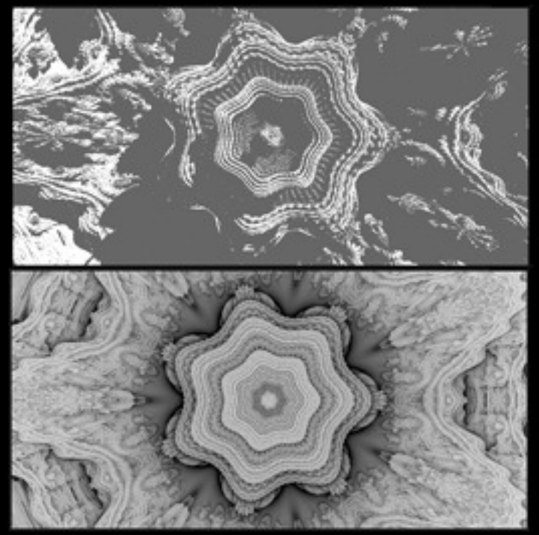


=

ray with length

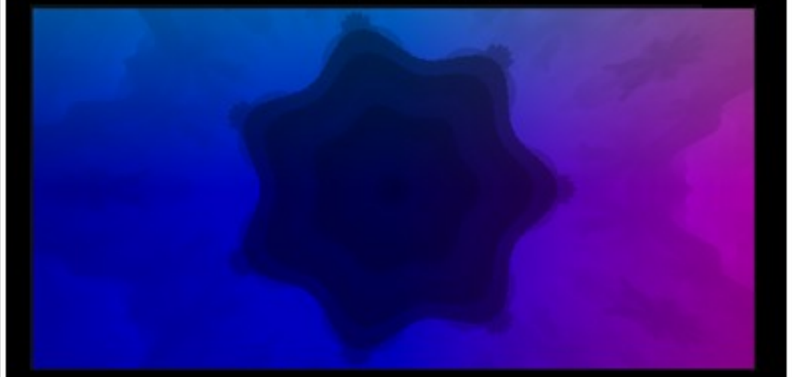






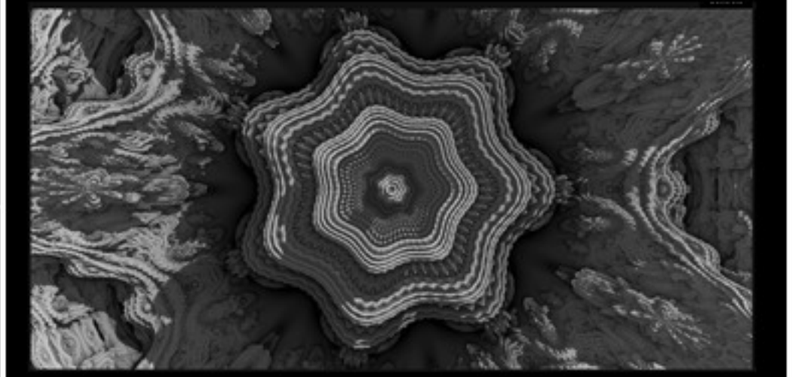
RGB

ray with length

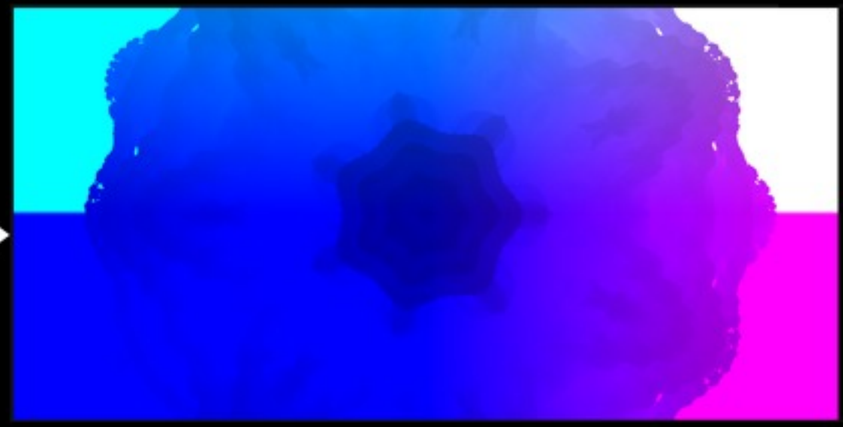


Alpha

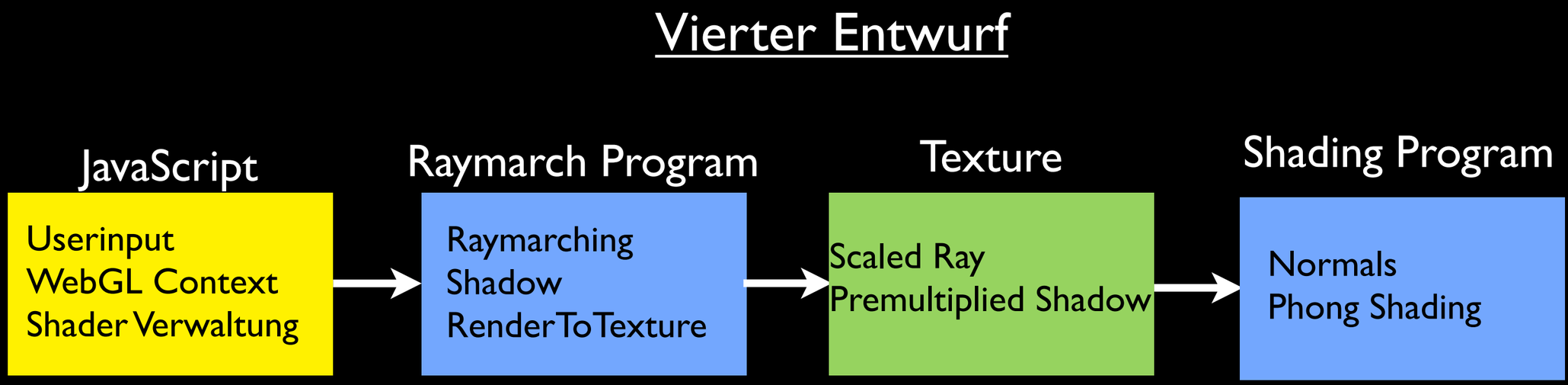
premultiplied shadows



G Buffer

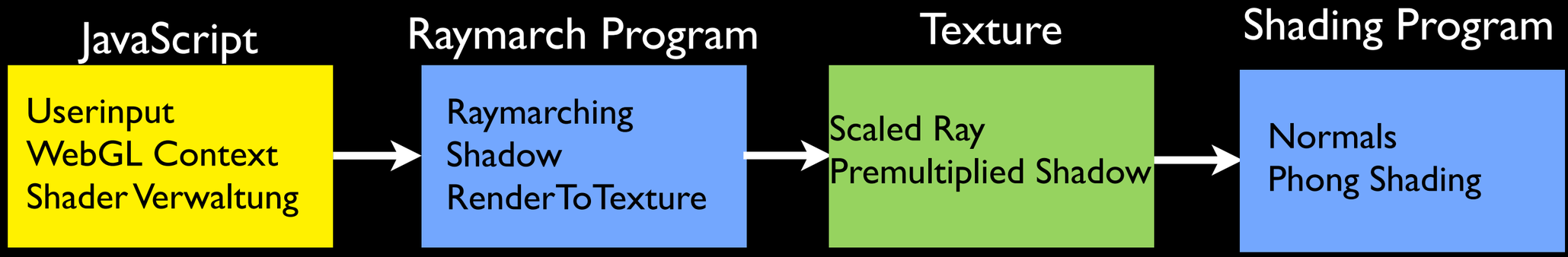


Implementierung - Entwicklung



Implementierung - Entwicklung

Vierter Entwurf



bessere Tiefenwirkung?

Implementierung - Entwicklung

LOESUNG:

- Approximation von Lichstreuung durch Distance Fog

- Berechnung des Distance Fogs ueber die Entfernung

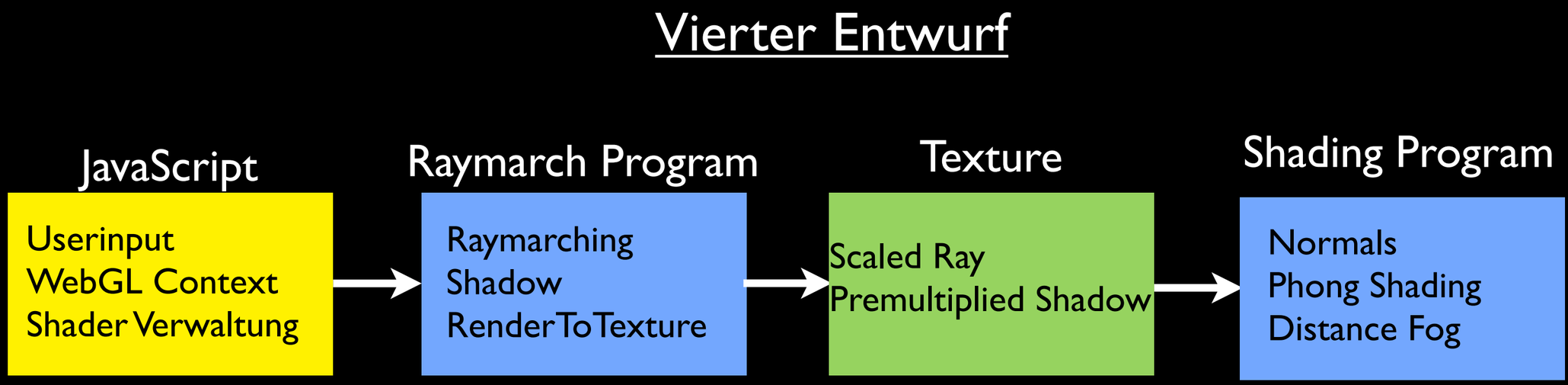
```
// fogScale is user defined
float fogFactor = fogScale * pow(depth, 0.75f);
```

- Blenden der Geometrie mit der Hintergrundfarbe anhand des `fogFactor`

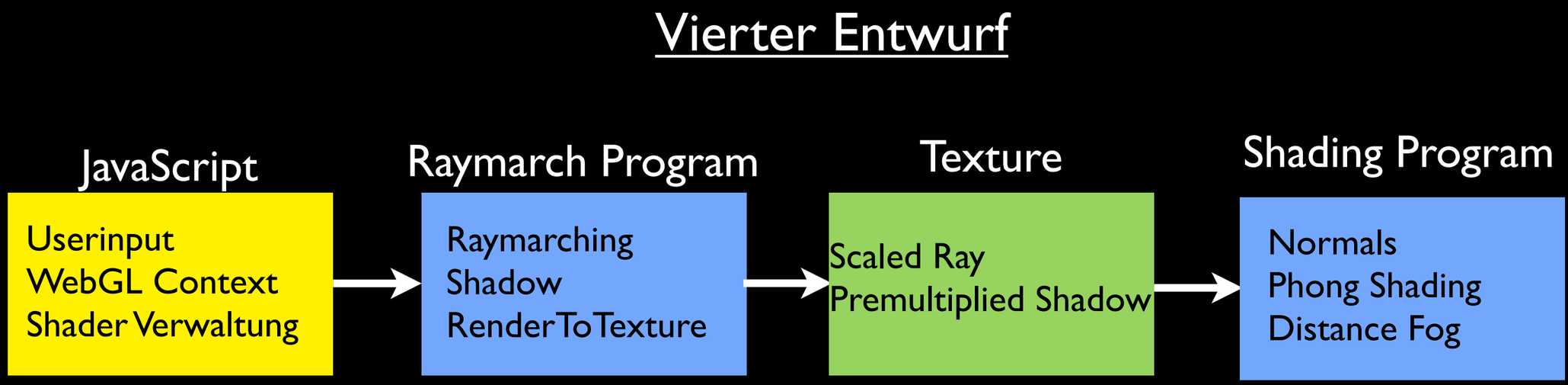
=> einfache Berechnung mit gutem Ergebnis

Implementierung - Entwicklung

Implementierung - Entwicklung



Implementierung - Entwicklung



Anti Aliasing?

Optimierung - Bildqualitaet

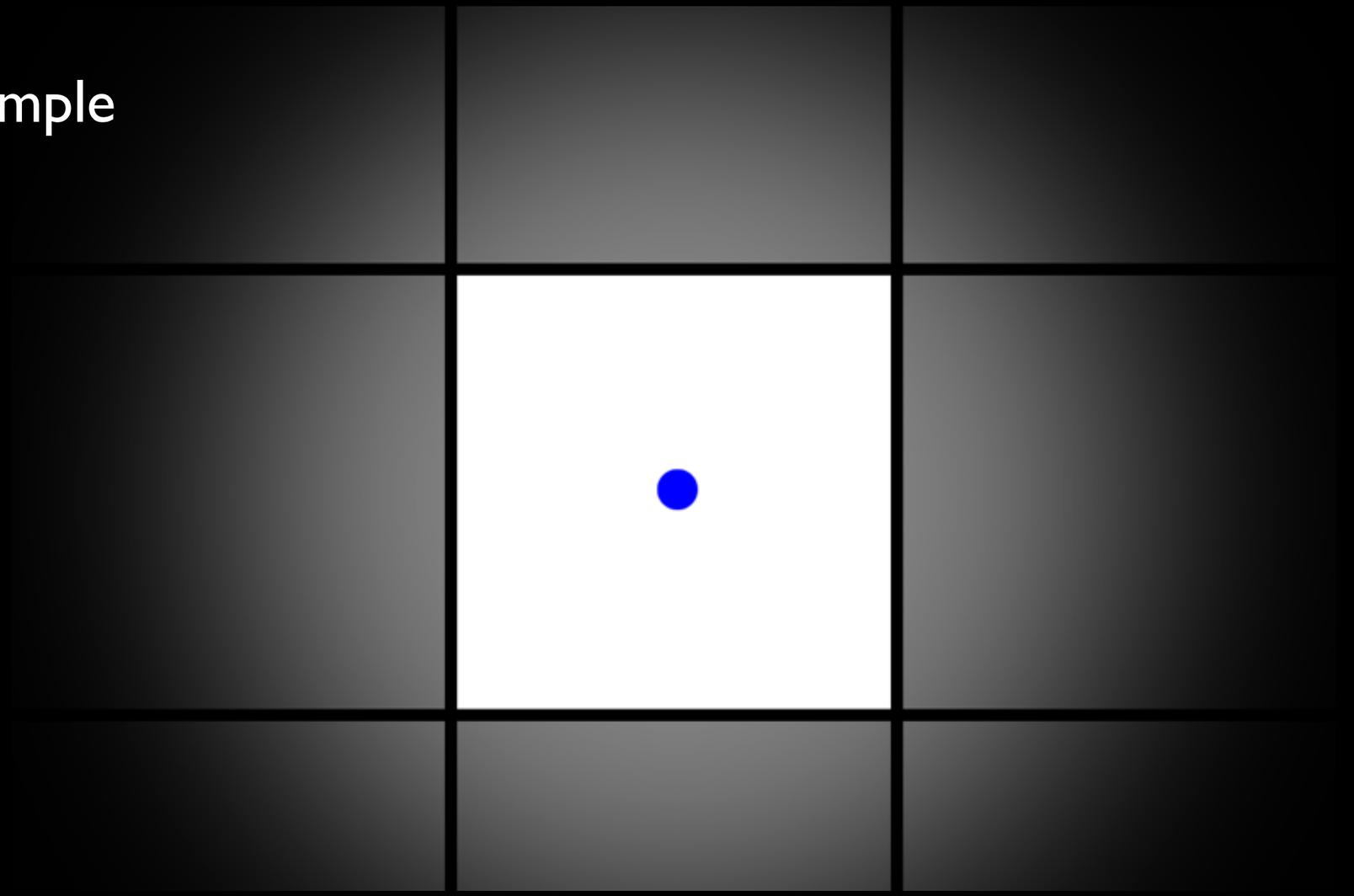
LOESUNG :

- Supersampling
  - mehr als ein Ray pro Pixel
  - Raydirection um einen Subpixeloffset verschoben
  - Subpixeloffset gleichmaessig auf Pixelflaeche verteilt

  => Mittelwert aller Rays pro Pixel bilden Ergebnis

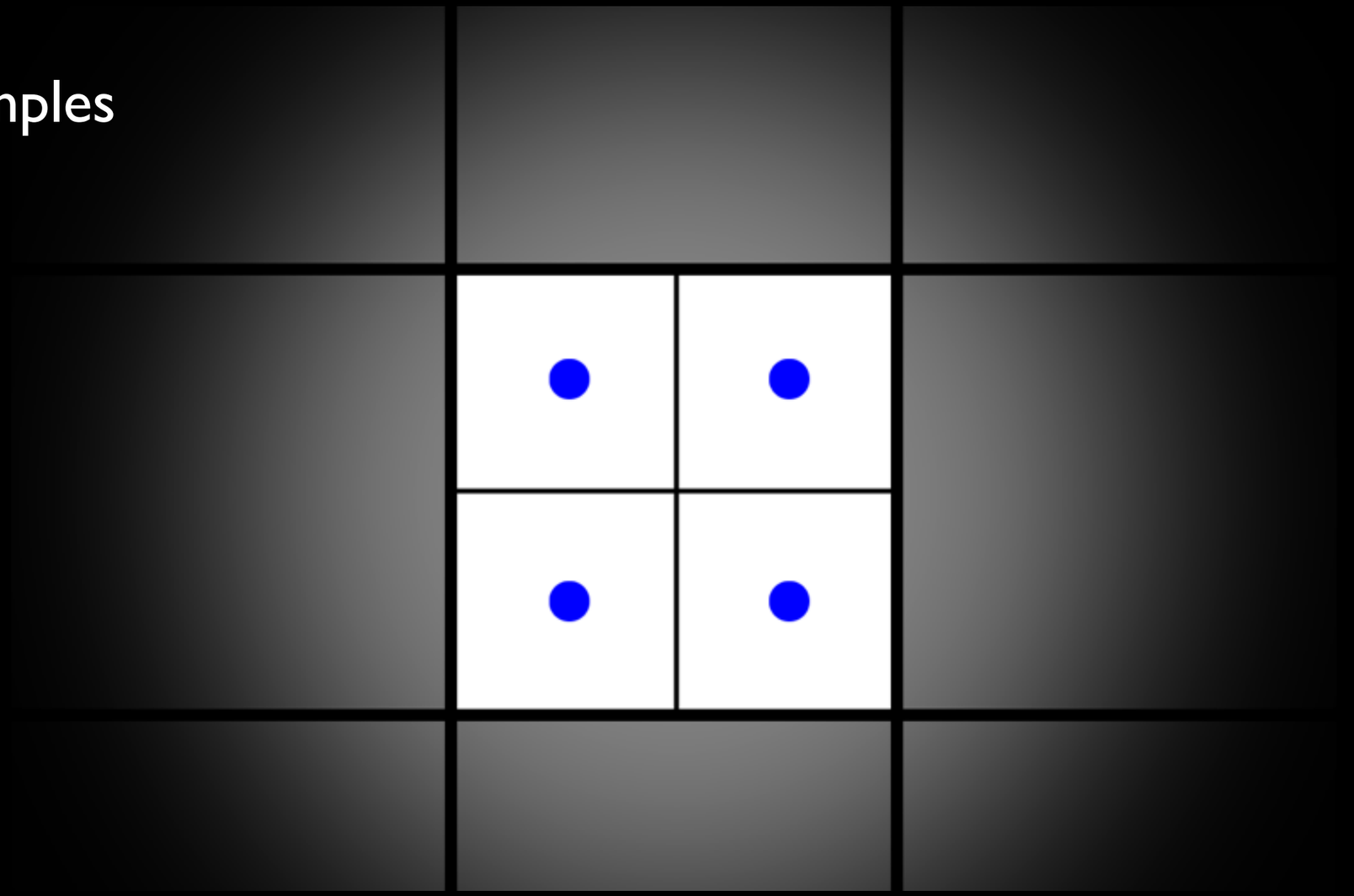
Optimierung - Bildqualitaet

1 Sample



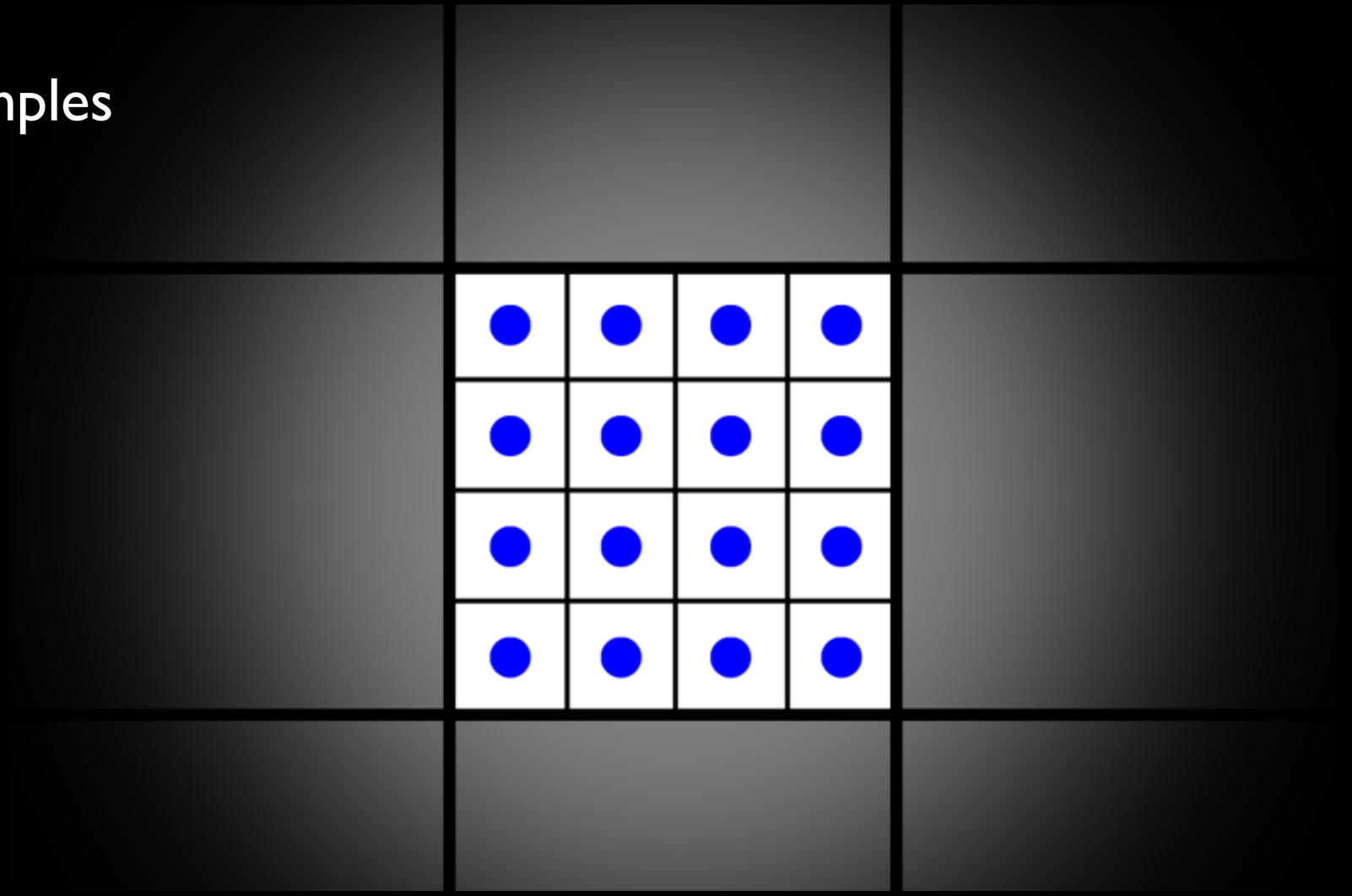
Optimierung - Bildqualitaet

2x2 Samples



Optimierung - Bildqualitaet

4x4 Samples



Optimierung - Bildqualitaet

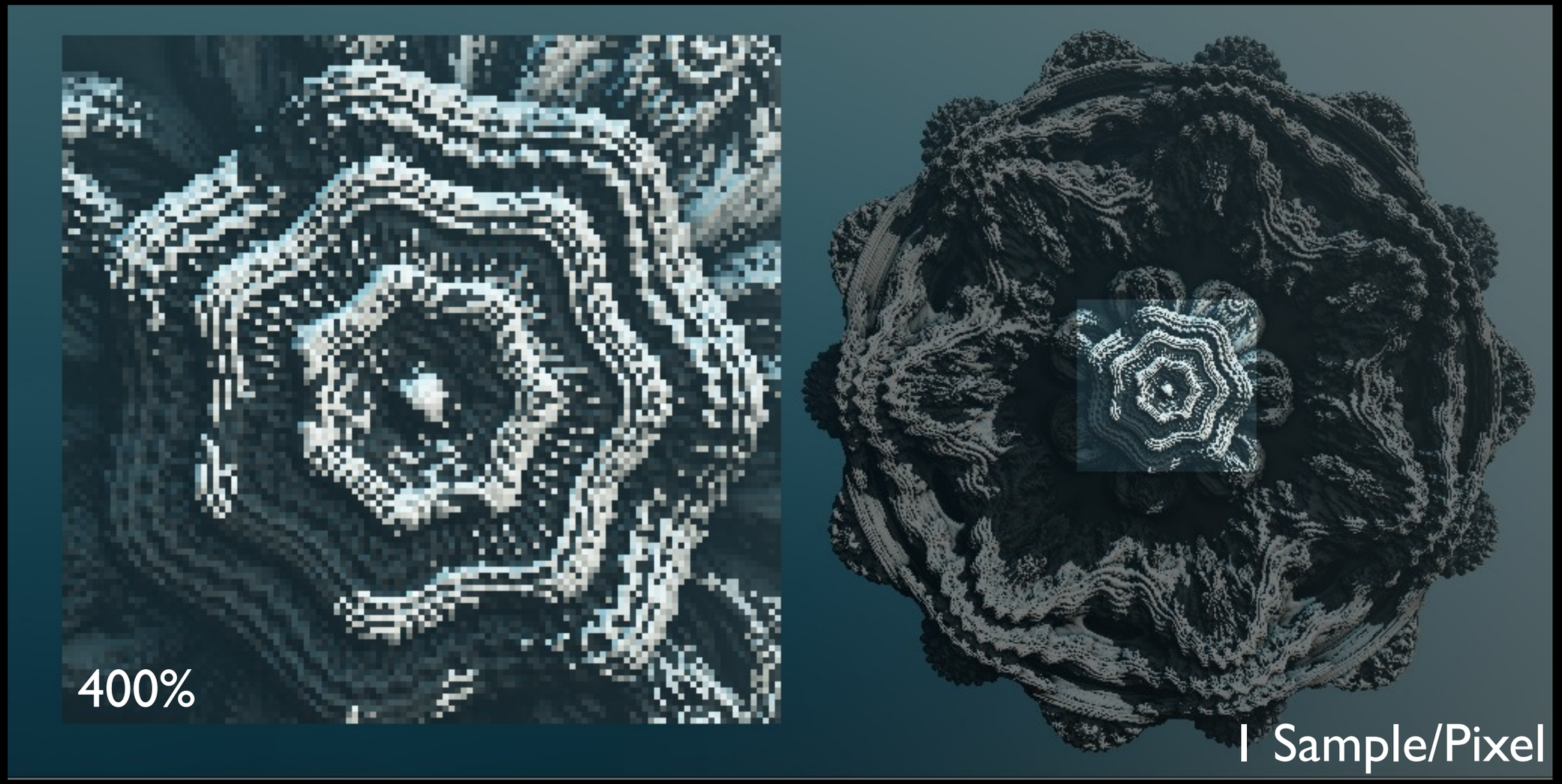
- Die Aliasfehler nehmen mit zunehmender Samplezahl ab
- Der Rechenaufwand steigt deutlich an

  => Aufteilung auf mehrere Frames (Zwischenergebnis darstellen)

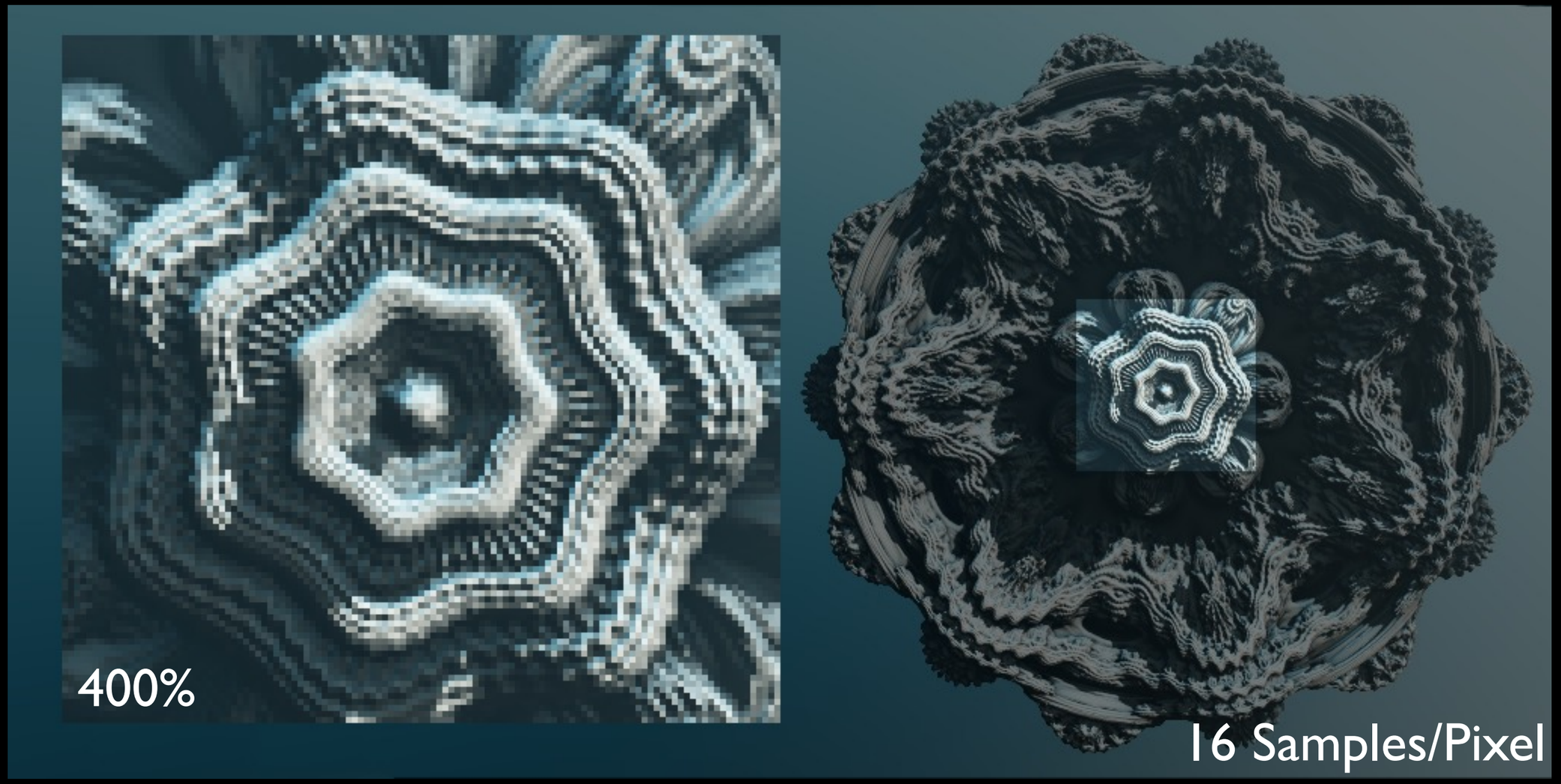
  => Buffer fuer Zwischenergebnis benoetigt

```
while (!moving && samples < maxSamples)
{
 renderNextSample();   //mix old and new values
 samples++:
}
```

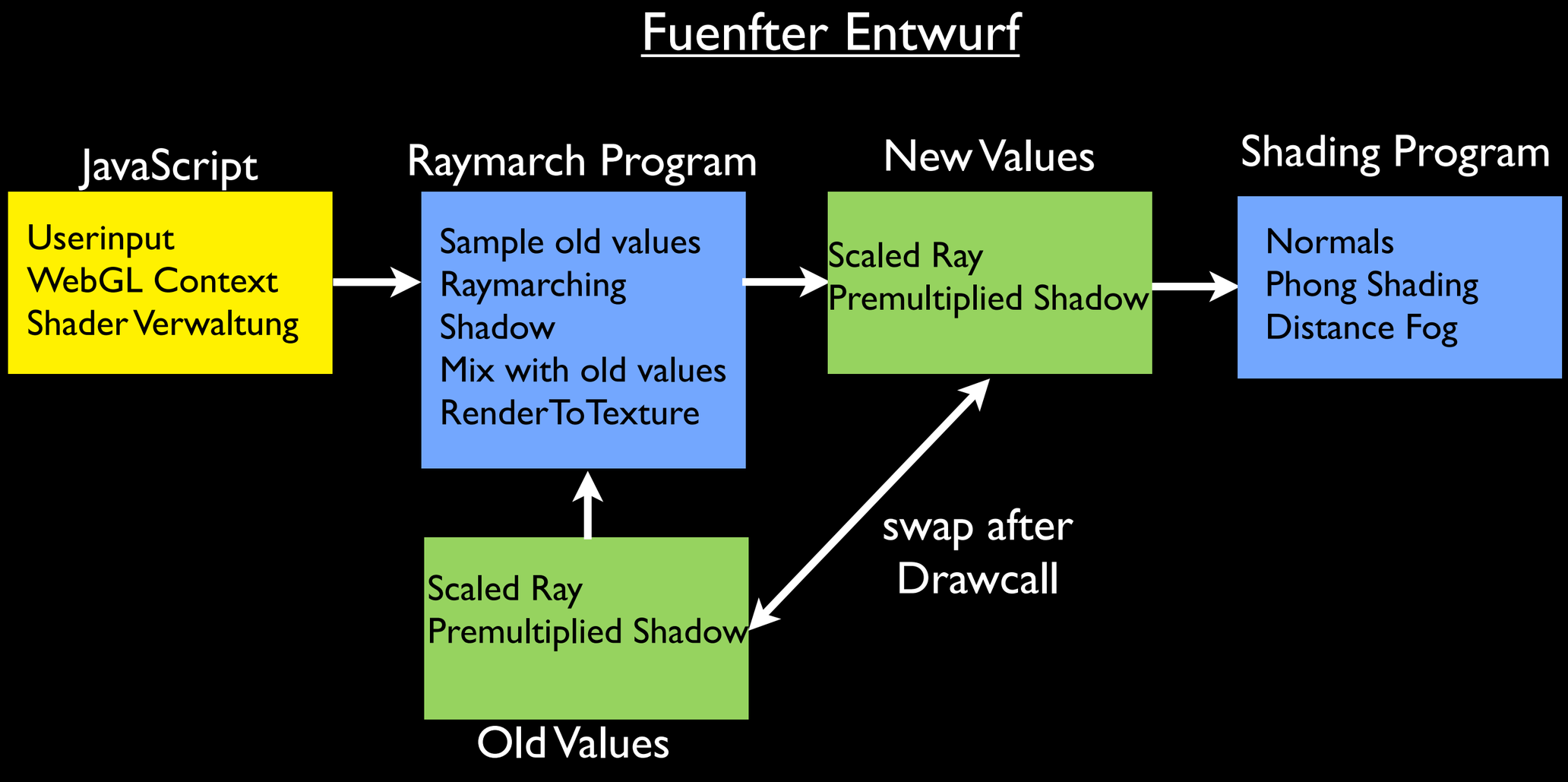
Optimierung - Bildqualitaet



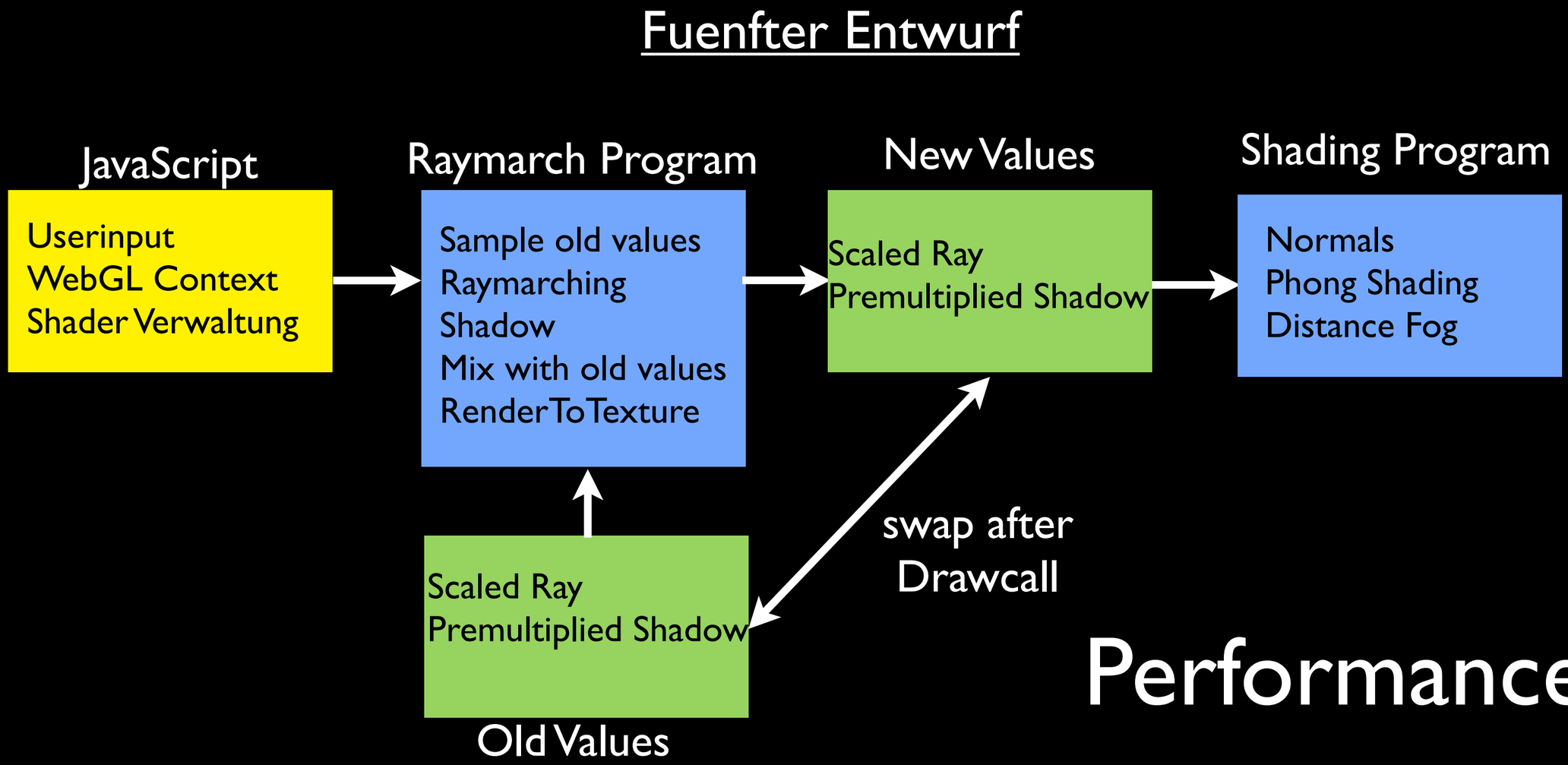
Optimierung - Bildqualitaet



Optimierung - Bildqualitaet



Optimierung - Bildqualitaet



Performance?

Optimierung - Performance

- Bildqualitaet sehr gut
- Massives Inputlag auf Low End Hardware

  => nicht interaktiv genug

- Rechenaufwand pro Frame zu hoch

  => Reduzierung der Rays pro Frame (weniger Pixel pro Frame)

- Aufloesung des Ausgabebildes soll bestehen bleiben !

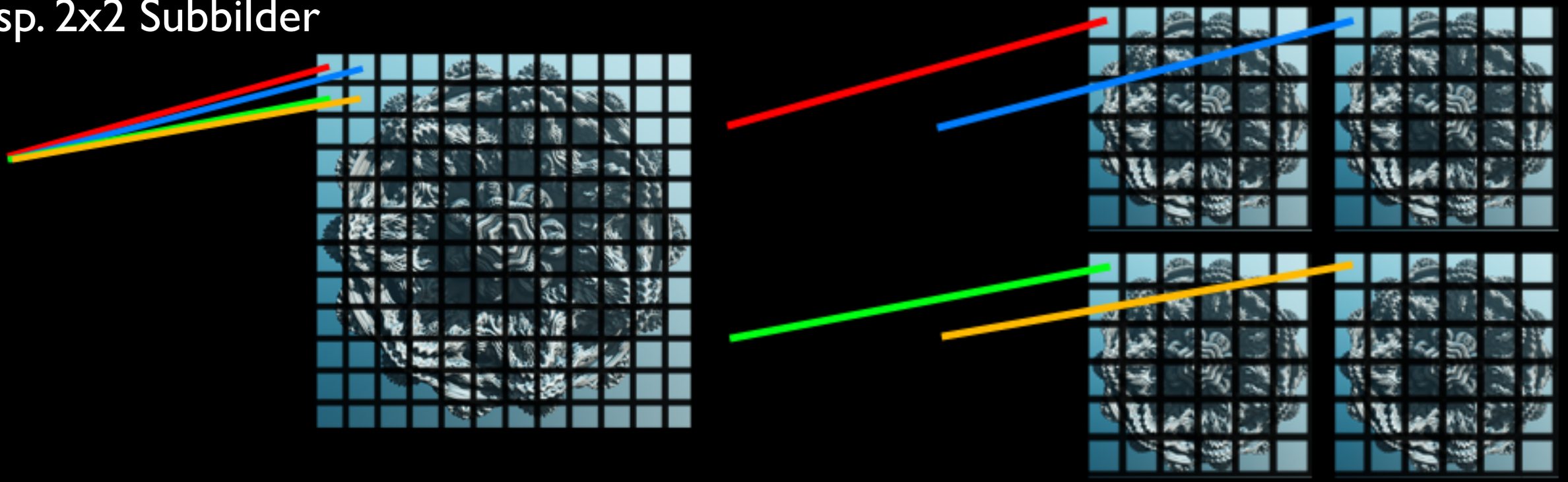
Optimierung - Performance

- Aufteilung des Bildes in Subbilder
- Pro Frame nur ein Subbild
- “Stempeln” des Subbildes
- Zusammensetzung des Gesamtbildes aus Subbildern

Optimierung - Performance

Aufteilung des Bildes in Subbilder

Bsp. 2x2 Subbilder



Optimierung - Performance

Aufteilung des Bildes in Subbilder

- Rasterisierung des Bildes

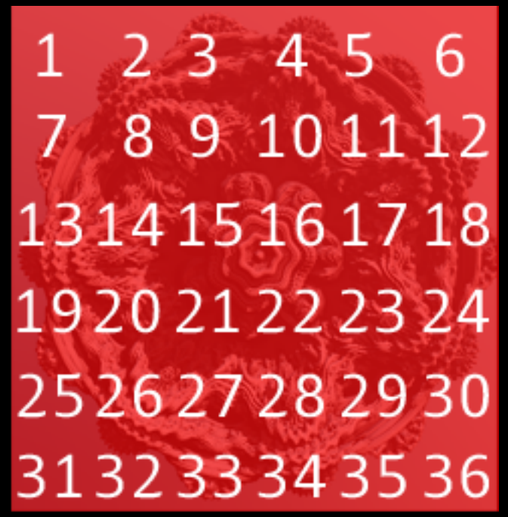
- ermoeglicht LowRes Darstellung

- veringert den Rechenaufwand pro Frame

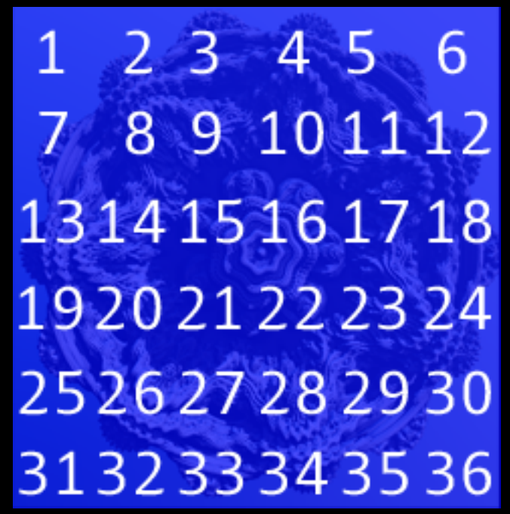
=> besser Reaktionsgeschwindigkeit

Optimierung - Performance

1. Frame



2. Frame

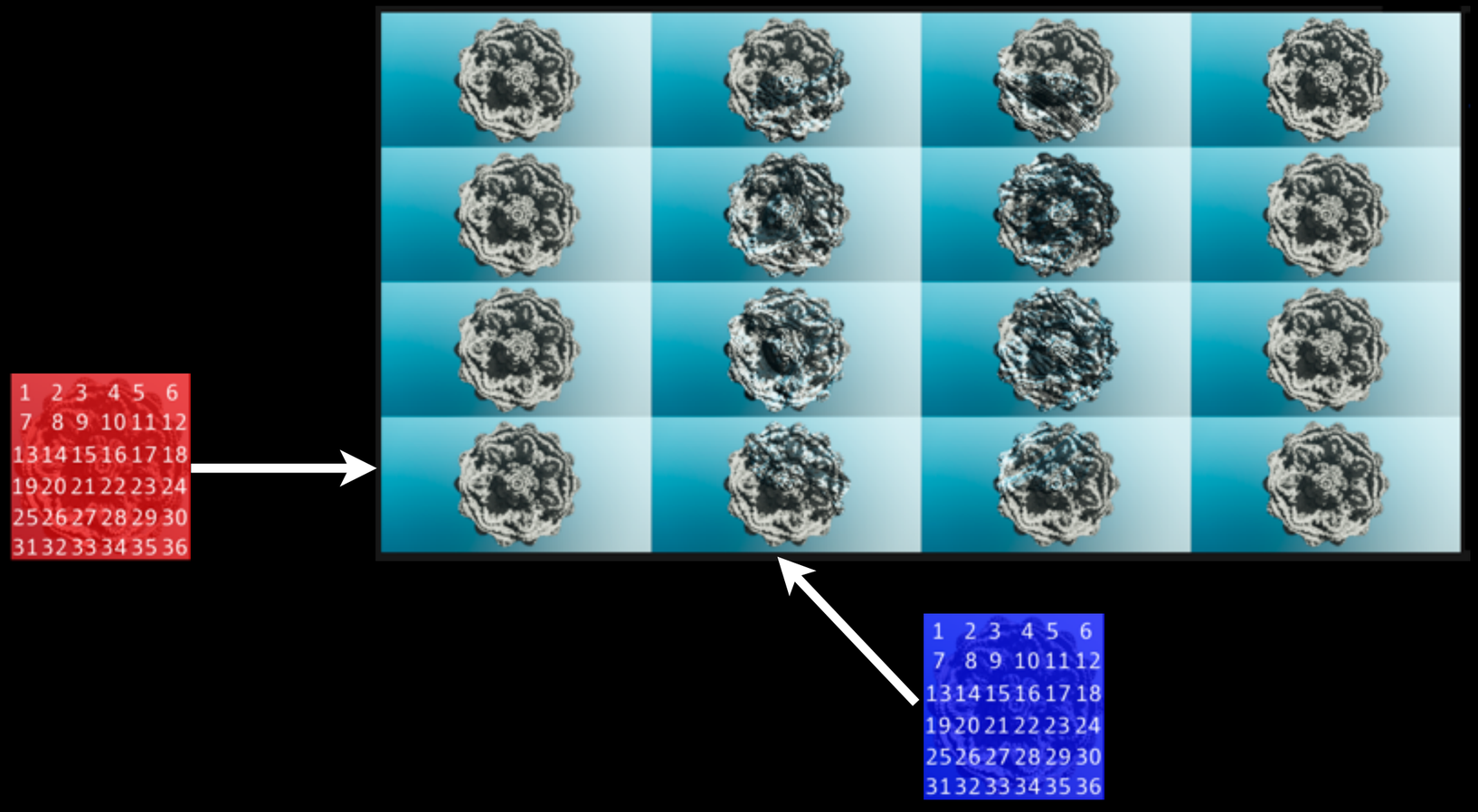


Optimierung - Performance

- Aufteilung des Bildes in Subbilder
- Pro Frame nur ein Subbild
- “Stempeln” des Subbildes
- Zusammensetzung des Gesamtbildes aus Subbildern

Optimierung - Performance

"Stempeln" des Subbildes

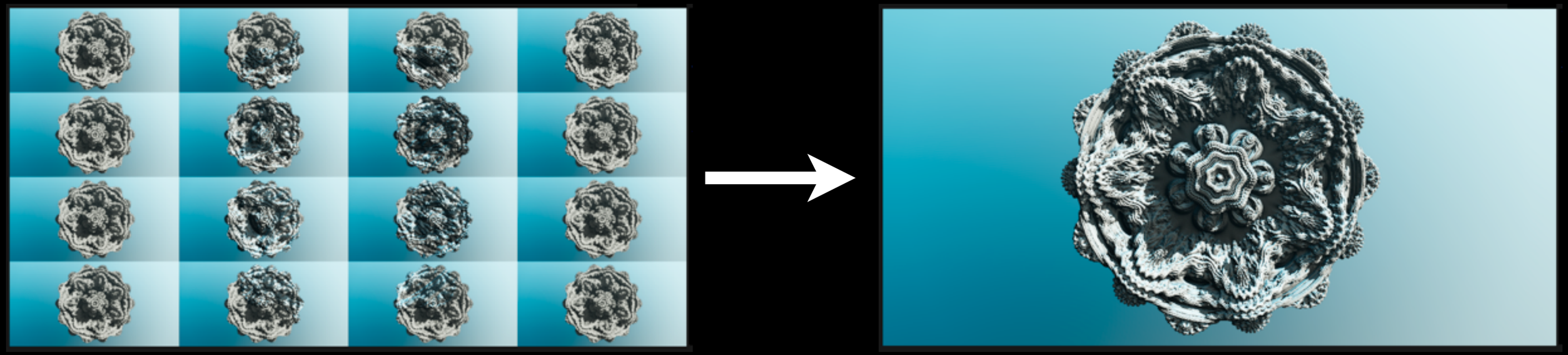


Optimierung - Performance

- Aufteilung des Bildes in Subbilder
- Pro Frame nur ein Subbild
- “Stempeln” des Subbildes
- Zusammensetzung des Gesamtbildes aus Subbildern

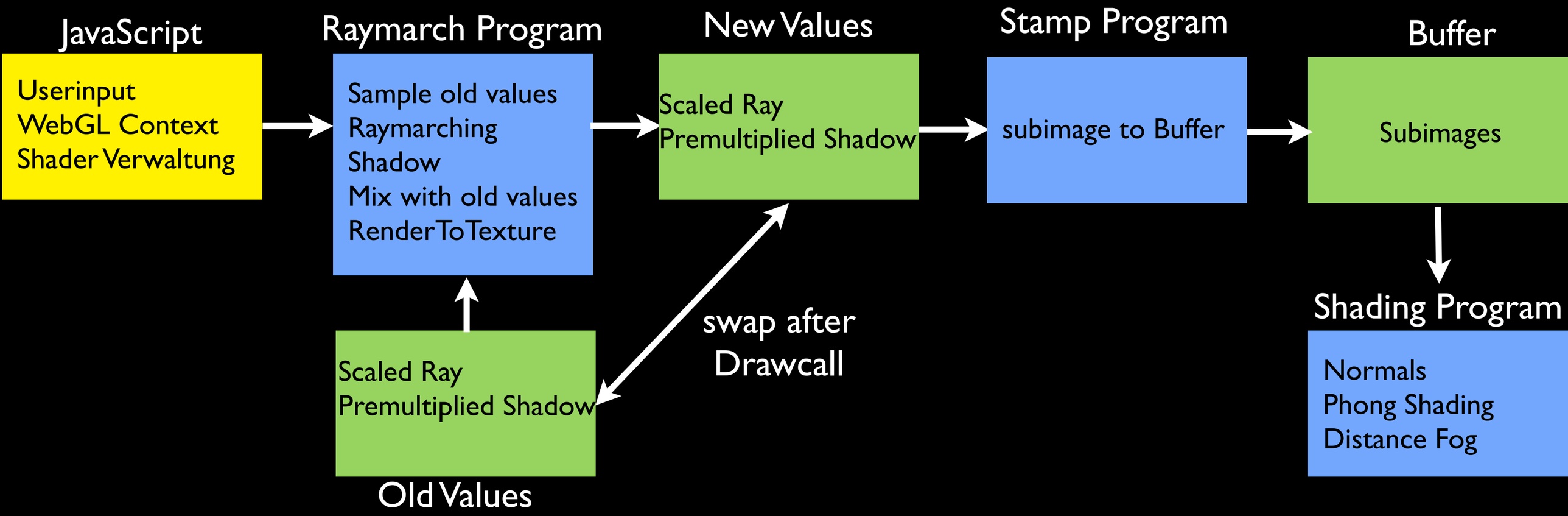
Optimierung - Performance

Zusammensetzung des Gesamtbildes aus Subbildern



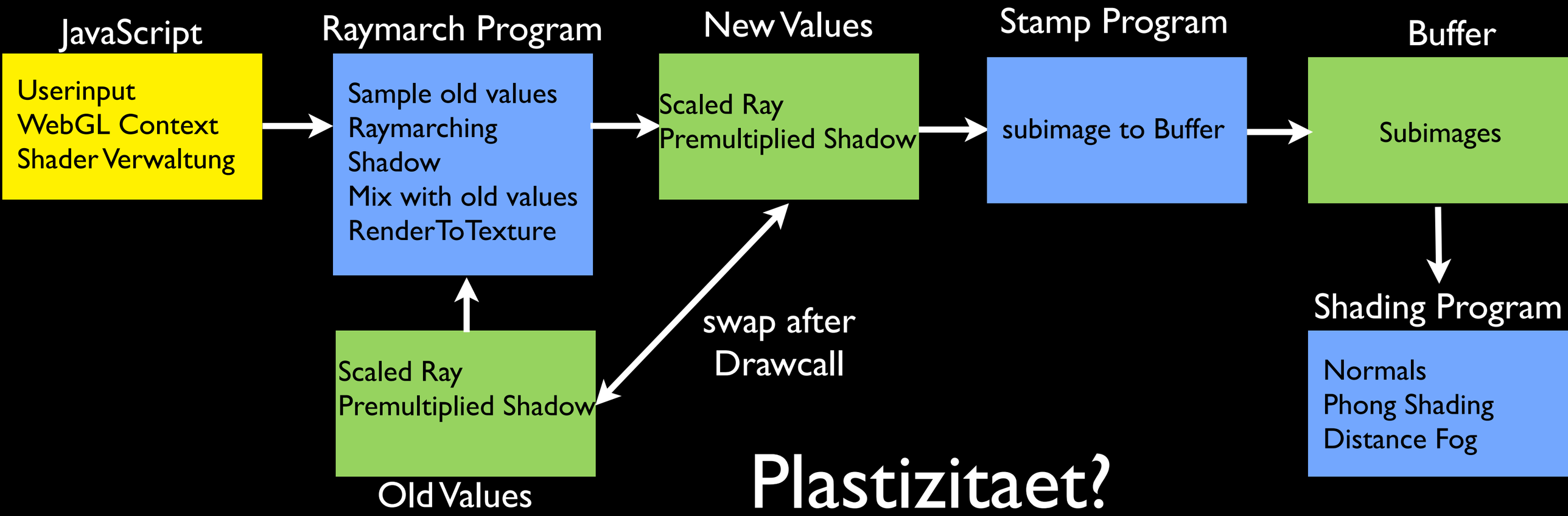
Optimierung - Bildqualitaet

Sechster Entwurf



Optimierung - Bildqualitaet

Sechster Entwurf



Plastizitaet?

Post Processing

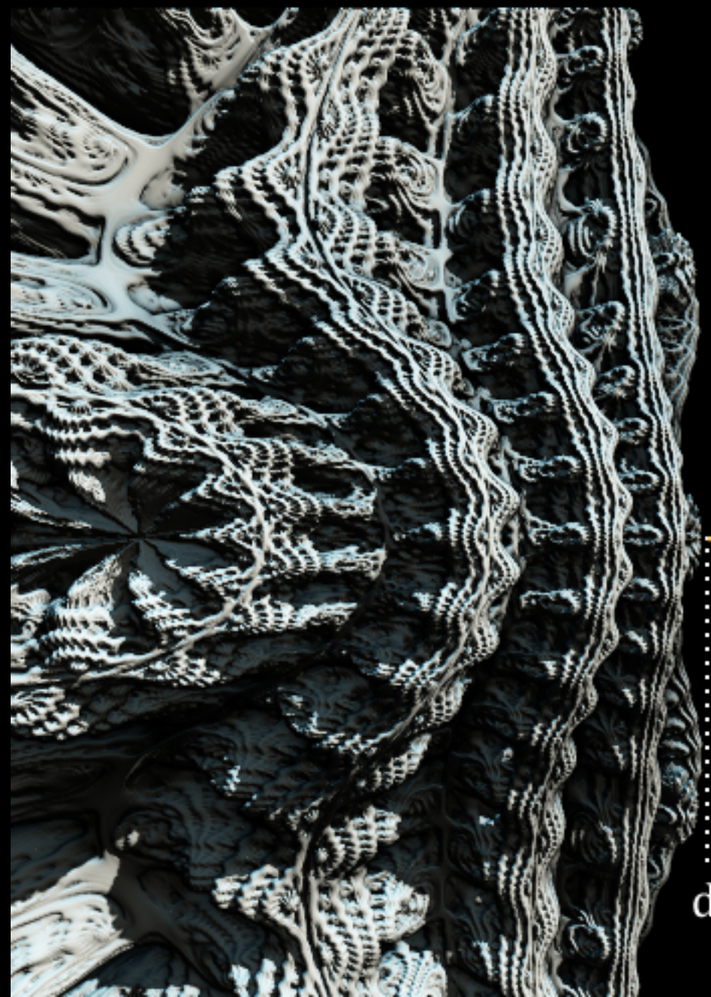
LOESUNG :

- Plastizitaet durch Tiefenunschaerfe
  => realistischer Eindruck wird verstaerkt

- als Post Process ueber Shader realisierbar [Upitis12]
  => Farbe und Tiefe als Input
  => Unschaerfekreisdurchmesser bestimmt Grad der Unschaerfe

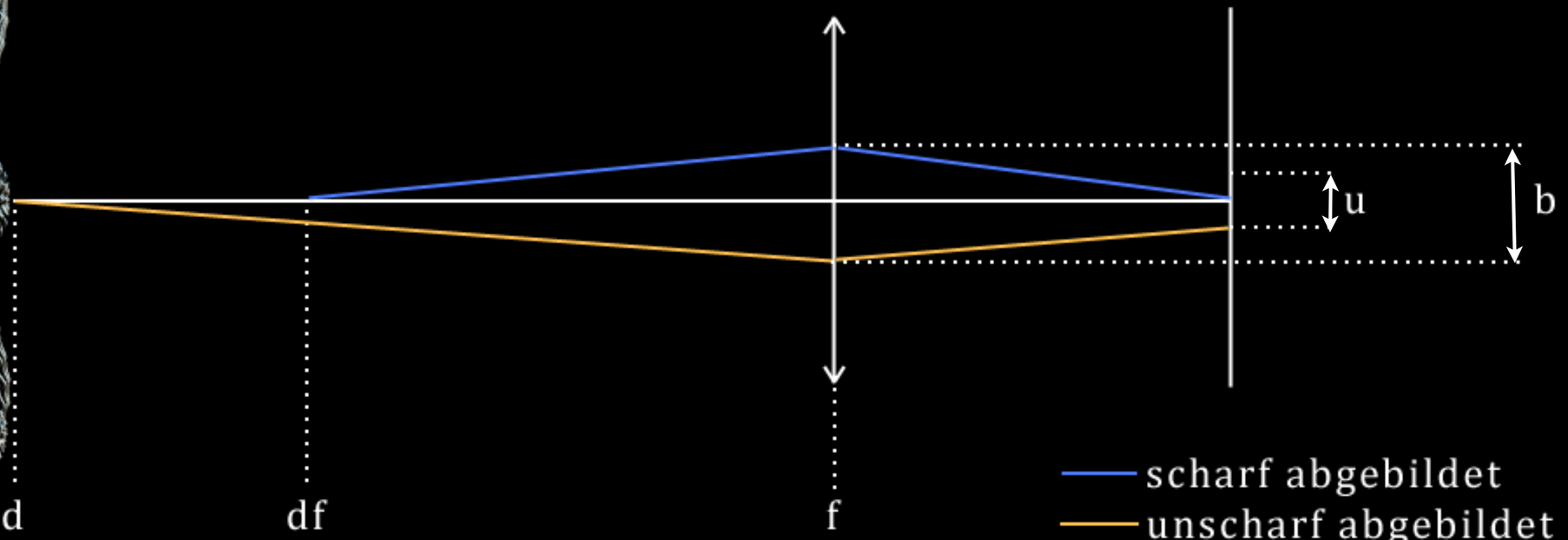
- Verschiebung der Schaerfeebene

Unschaerfekreisberechnung



$$u = b * \frac{|d - df|}{d} * \frac{f}{df - f}$$

u: Durchmesser Unschaerfekreis
b: Durchmesser Blende
d: Tiefe des akt. Fragments
df: Tiefe der Schaerfeebene
f: Brennweite

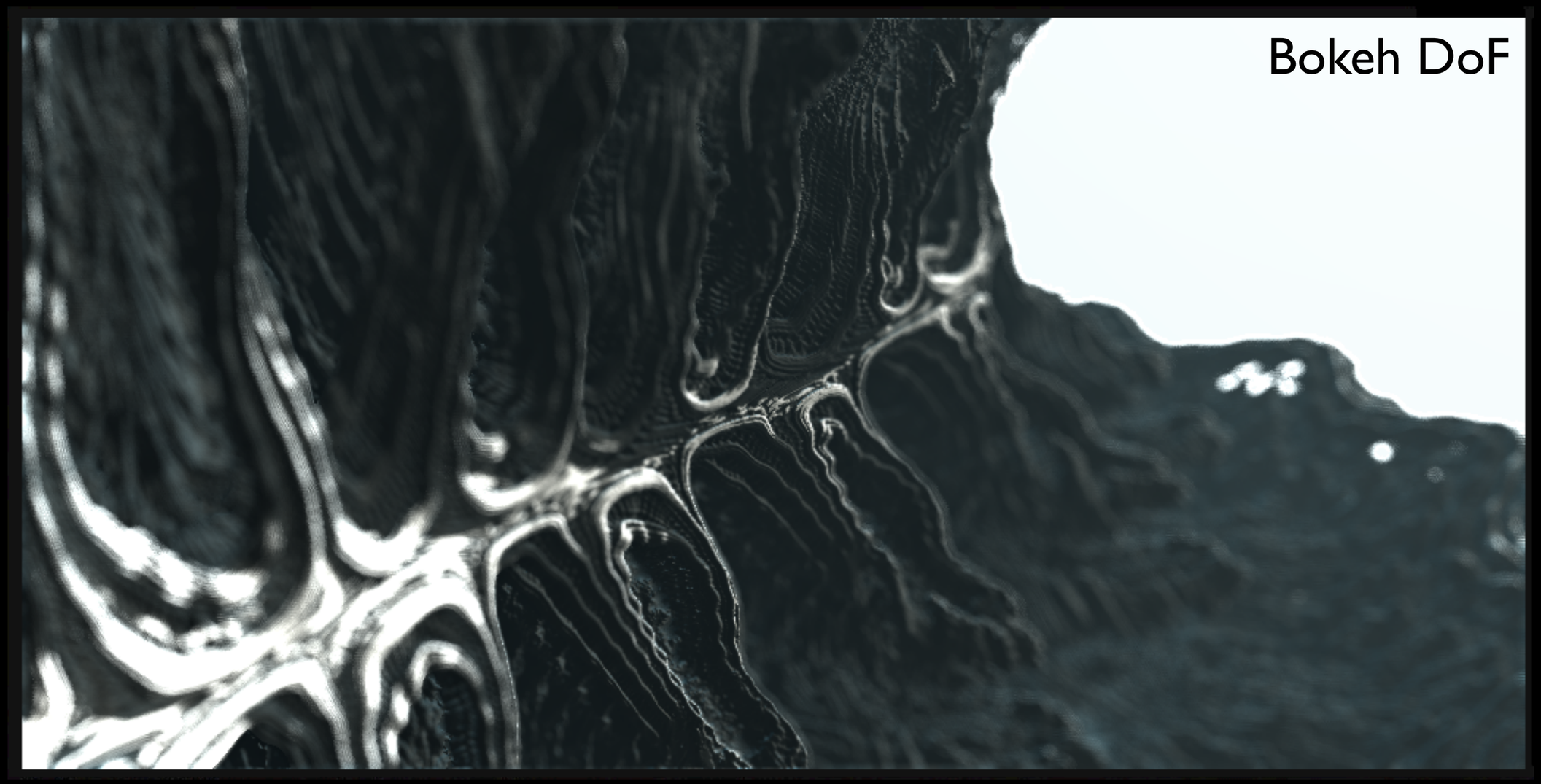


Depth of Field

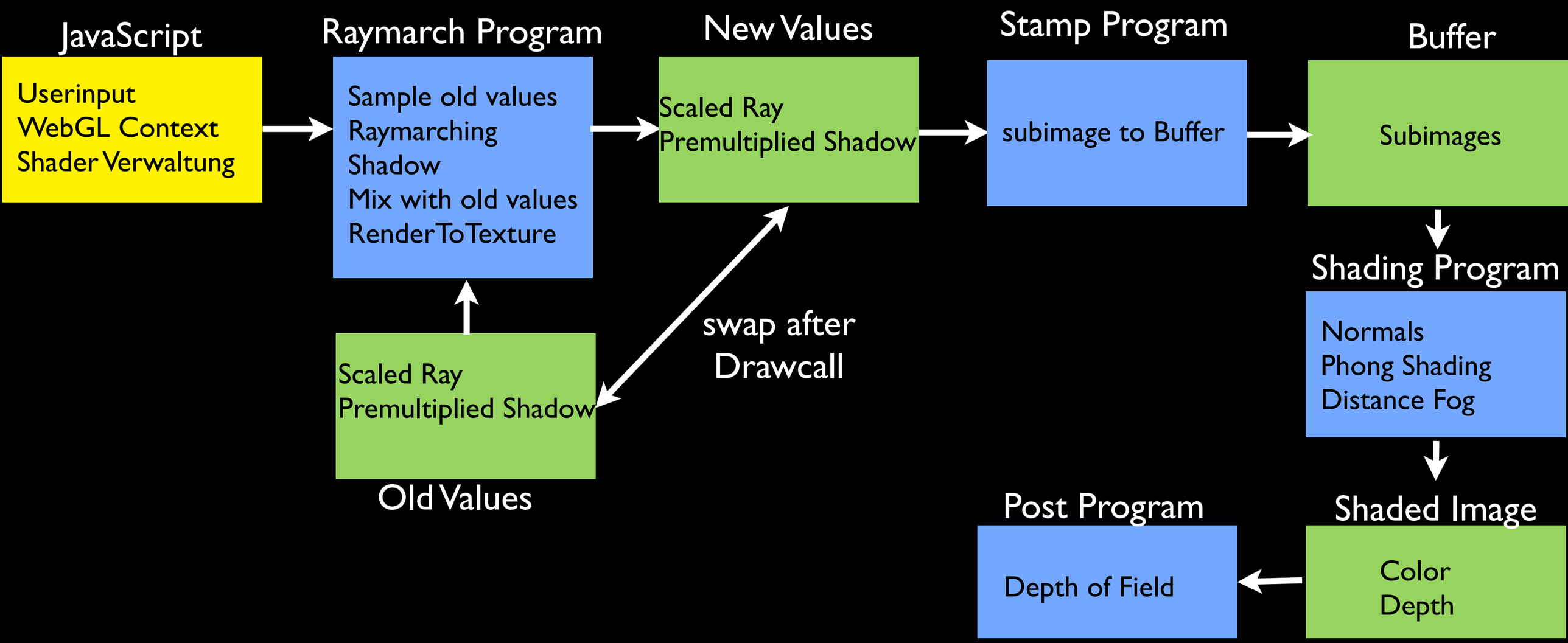
```
for each pixel
{
  float radius = calculateCOC(); //Unschaerfekreis berechnen

  for each sample
  {
    color += colorInRadius(radius);
    //gibt zufaelligen Farbwert im Radius des Unschaerfekreises
    um den Pixel zurueck
  }

  color /= NUM_SAMPLES; //Farbwert normalisieren
}
```

Post Processing

Post Processing



Finaler Entwurf



Rueckblick

- Darstellung von verschiedenen 3D Fraktalen in WebGL
- frei bewegliche Kamera
- Interaktivitaet (20-30fps auf HD3000)

Ergebnis

- Darstellung von verschiedenen 3D Fraktalen in WebGL
- frei bewegliche Kamera
- Interaktivitaet (20-30fps auf HD3000)

Ergebnis

- Darstellung von verschiedenen 3D Fraktalen in WebGL
- frei bewegliche Kamera
- Interaktivitaet (20-30fps auf HD3000)
- Shading
- Anti Aliasing
- Distance Fog
- Post Processing (Exposure, DoF)
- Undersampling (subimages)
- bewegliche Lichtquelle
- Schattenwurf

Schwierigkeiten

- WebGL noch kein finaler Standard !
  - abhaengig von Browser/OS
  - nur sehr limitierter Umfang an GL Funktionen
    - Extensions
  - schwer zu debuggen (keine/wenige Fehlermeldungen)

=> experimentelle Arbeitsumgebung

Verbesserungsmoeglichkeiten

- stochastische Verteilung der Anti Aliasing Samples
- variable Ausgabeaufloesung (variable Subimageanzahl)
- automatische Regulierung der Kamerageschwindigkeit (DE in JS)
- Branching fuer Browser- / OS-Kompatibilitaet

DEMO

projekte.sinnpirat.de/fraktal

Unterstuetzt zur Zeit nur Google Chrome unter Windows

Quellen

[Fuhrmann11] - Arnulph Fuhrmann, Seminar Themen der Computergrafik - Vorlesungsfolien, 2011

[Quilez10] - Inigo Quilez, Free penumbra shadows for raymarching distance fields 2010

[Hart89] - John C. Hart , Ray Tracing Determenistic 3D Fractals, 1989

[Upitis12] - Martin Upitis, GLSL depth of field with bokeh, 2012