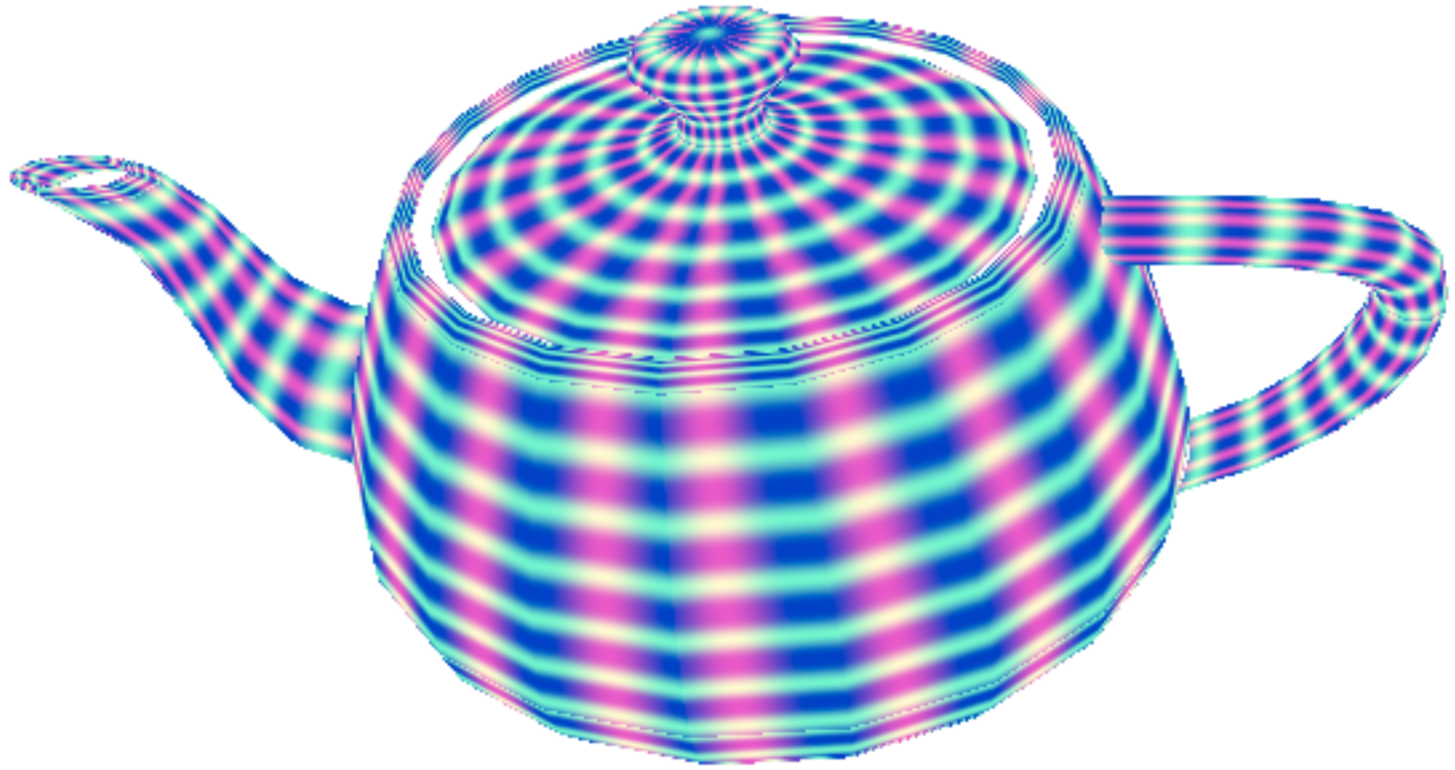
TNM084
Procedural images

Ingemar Ragnemalm, ISY



Lecture 4

More noise

Voronoi (cellular) noise including distance mapping

Transformations of other things than geometry

Anti-aliasing

Dugga results

Many had not very high points. Don't give up, it was the very first. You can improve it on the retake.

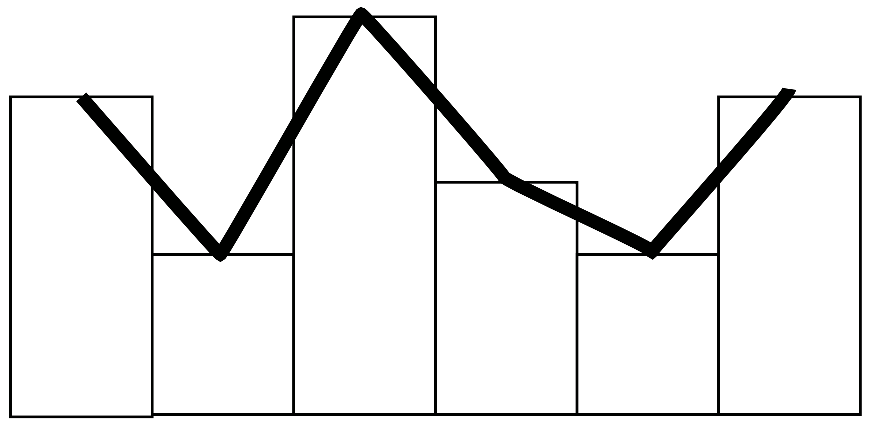
Any score on 2.5 or higher should be considered good (but not too good to improve)!

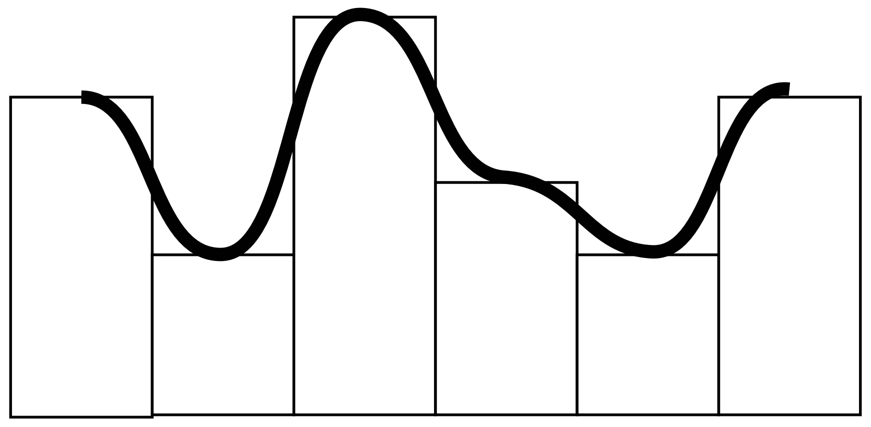
Common misconception in the dugga

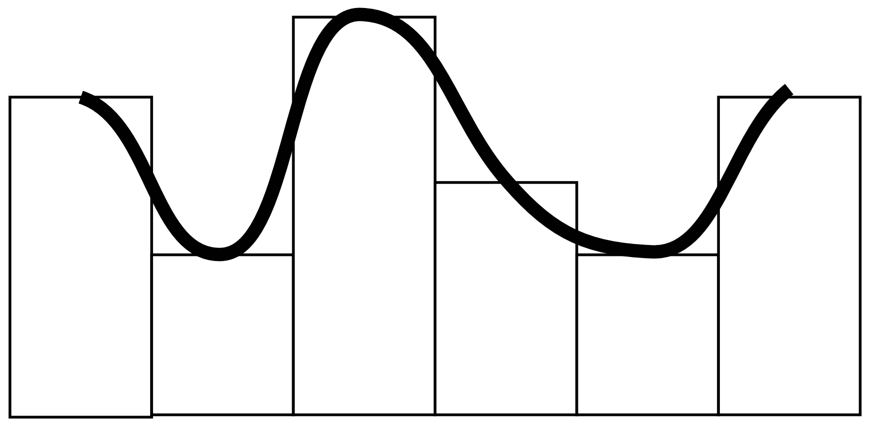
Filtered white noise (a.k.a. value noise) is not less smooth than gradient noise!

Consider a smoothstep between each sample.

Simple, and smooth.


Linear


Smoothstep


Cubic spline

A few hints

C continuity is about continuous derivatives (= same angle and speed). G is about proportional ones (= same angle).

Interpolations (esp cubic spline) approximates *sinc*.

Low-pass filtering (Gaussian) is something else.

Parallel patterns are calculated as functions of x and y, *pixelwise*. Fragment shader friendly is the point.

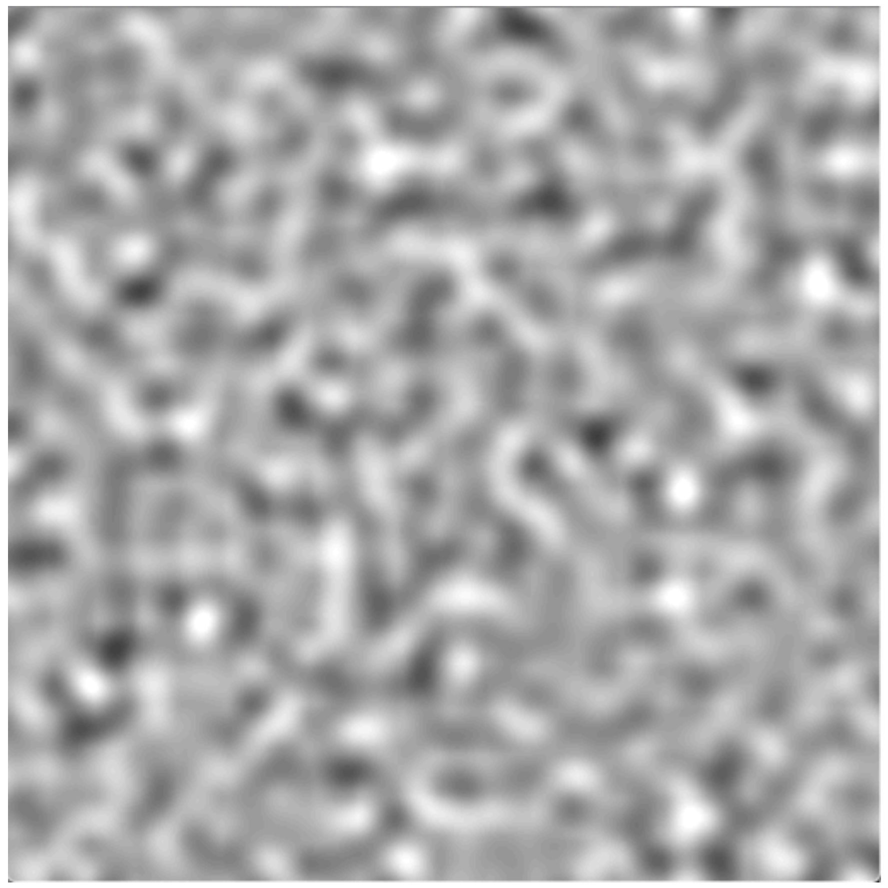
More noise

Some more comments on gradient noise

Voronoi noise

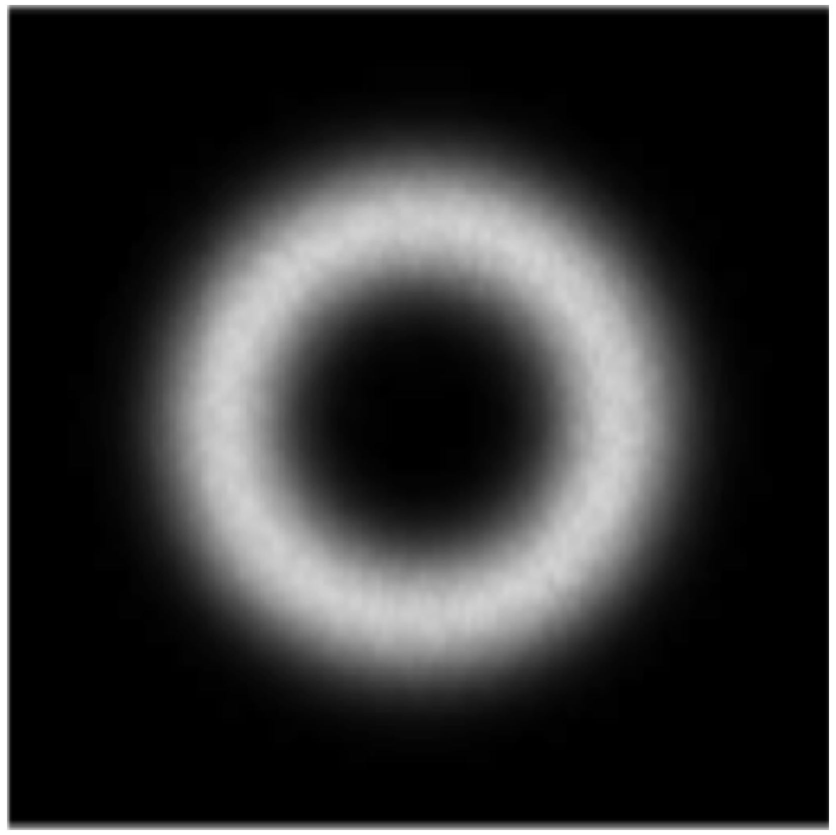
Double gradient noise

Add 2 or 4 different gradient noise functions
to avoid the zero crossing artifact

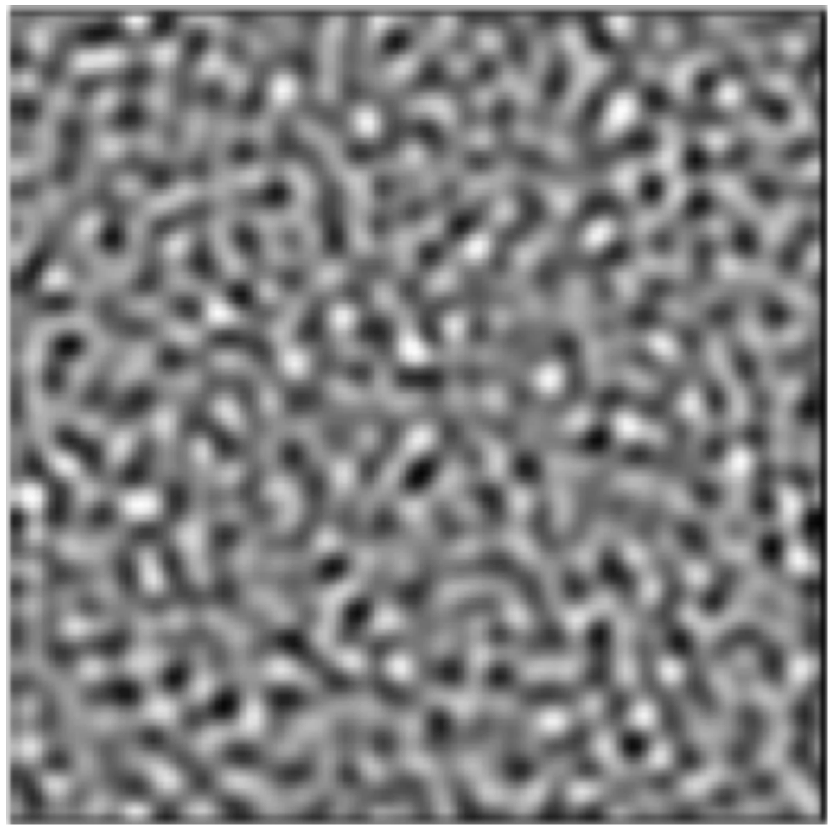


Fourier spectral synthesis

Random number in frequency space



Random numbers here, tuned to the desired frequencies



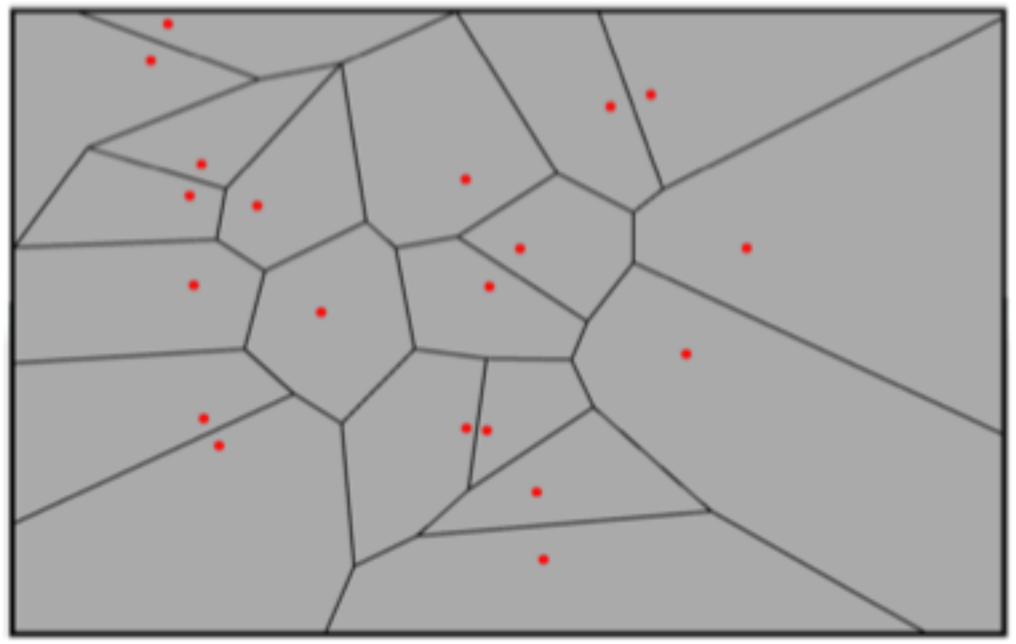
produces a perfect noise in O(NlogN) time

Voronoi (cellular) noise

Random tesselation of space into polygonal patches.

Based on Voronoi diagrams

Voronoi diagram: A subdivision of space into regions, each region being the area closest to each if a set of seed points:

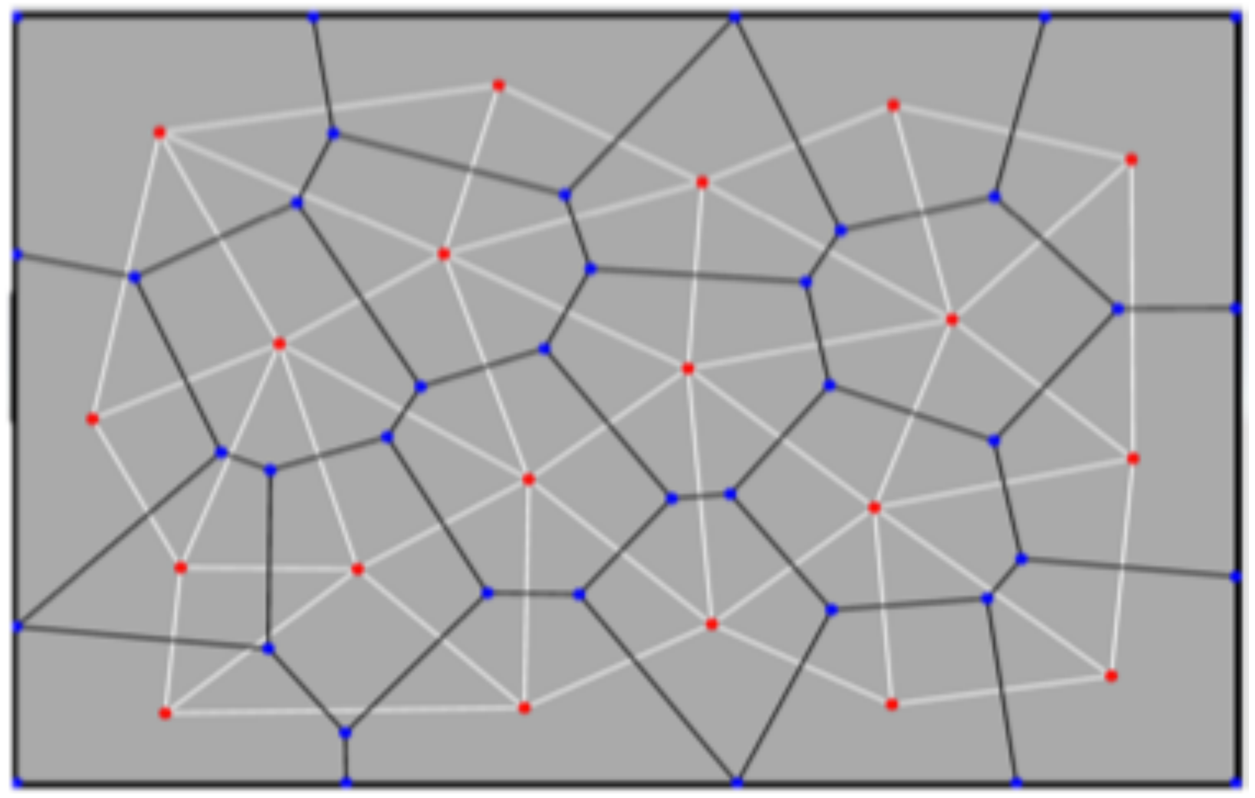


Many pictures following are from Marcus Dahlquists excellent project report from 2019

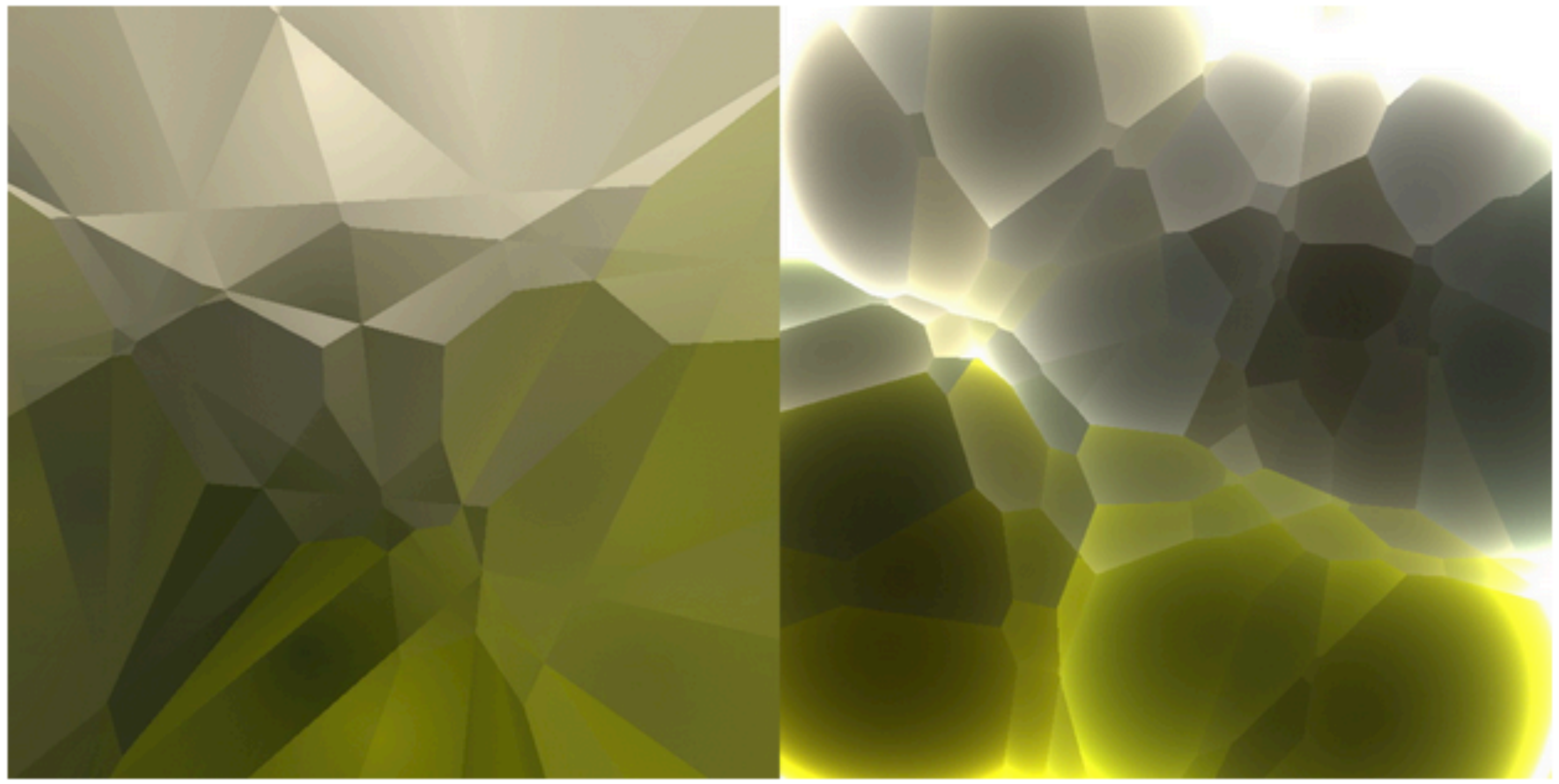
Voronoi diagrams and Delaunay triangulations

The Delaunay triangulation is a dual to the Voronoi diagram

Branches are only between points with touching Delaunay polygons

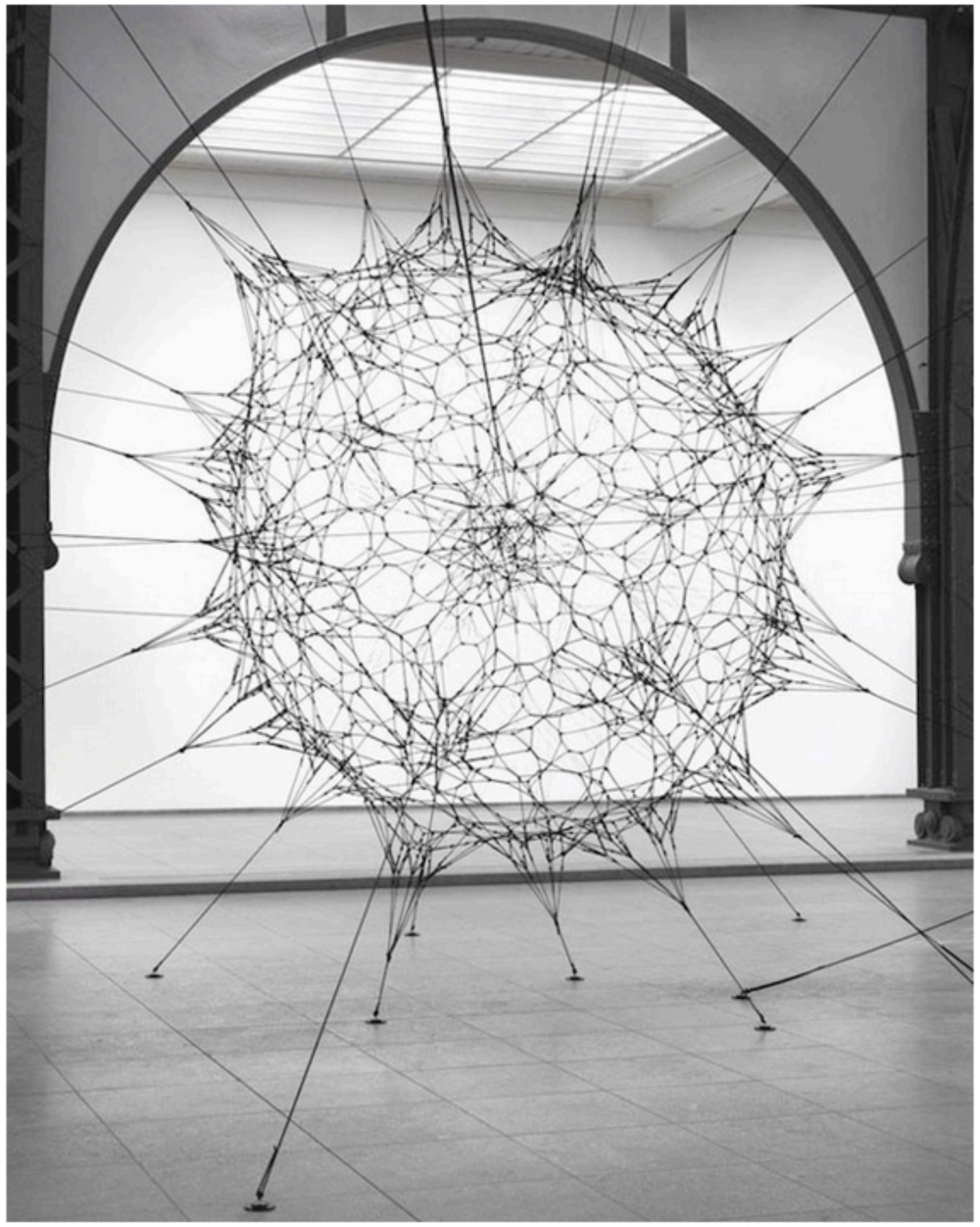


Examples usages of Voronoi noise

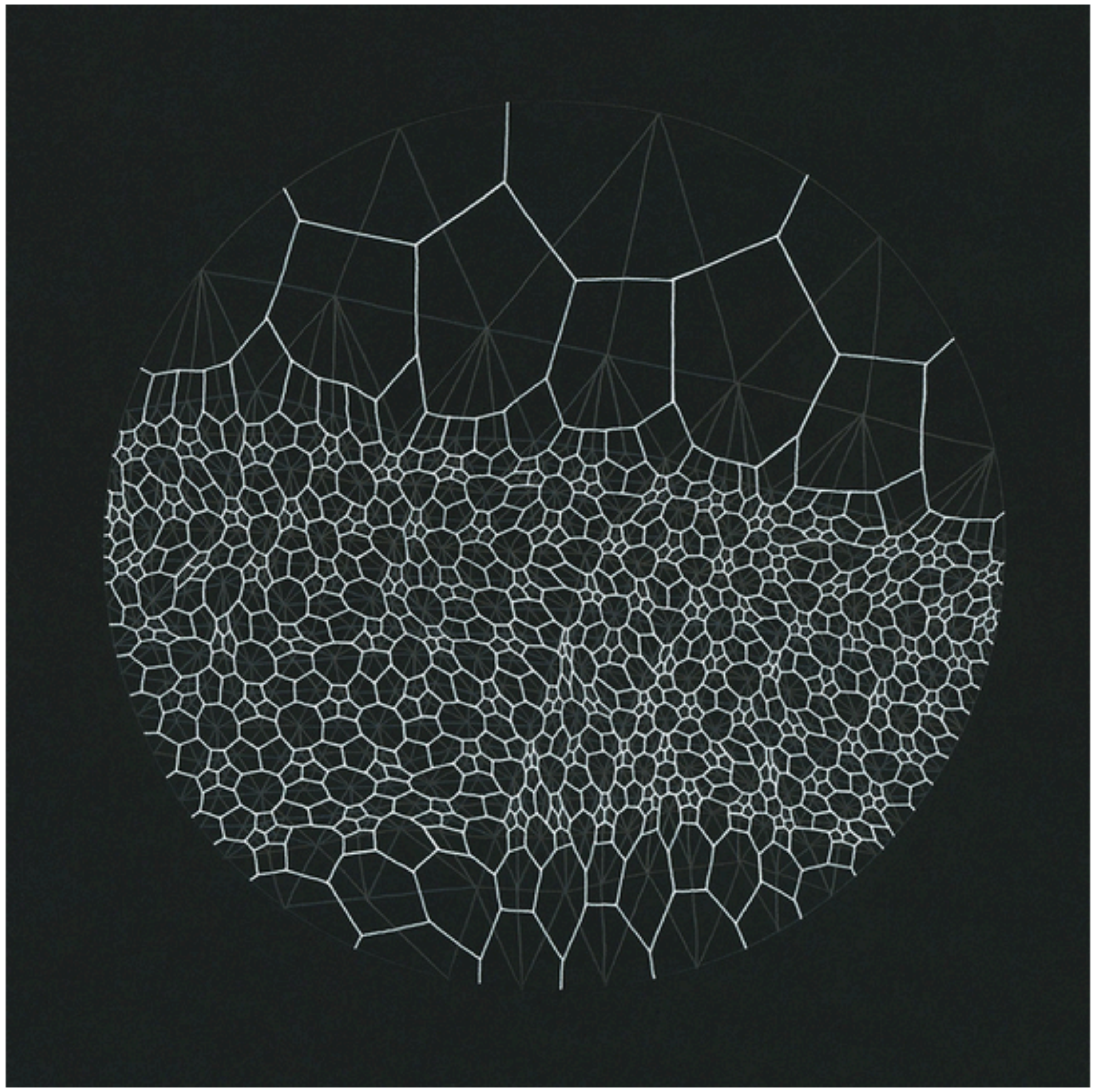

Extended Voronoi - Leo Solaas (2011)

From Book of shaders

Examples usages of Voronoi noise


Cloud Cities - Tomás Saraceno (2011)

Examples usages of Voronoi noise


Accretion Disc Series - Clint Fulkerson

Trivial implementation

For all points (pixels)

find the closest point in the seed list.

Acceptable for small sets of seeds

Complexity grows rapidly with larger sets

Possible accelerations:

- Build the Voronoi diagram explicitly from geometry, sorting points in lists
- Run a distance transform
- Do it in parallel in a fragment shader

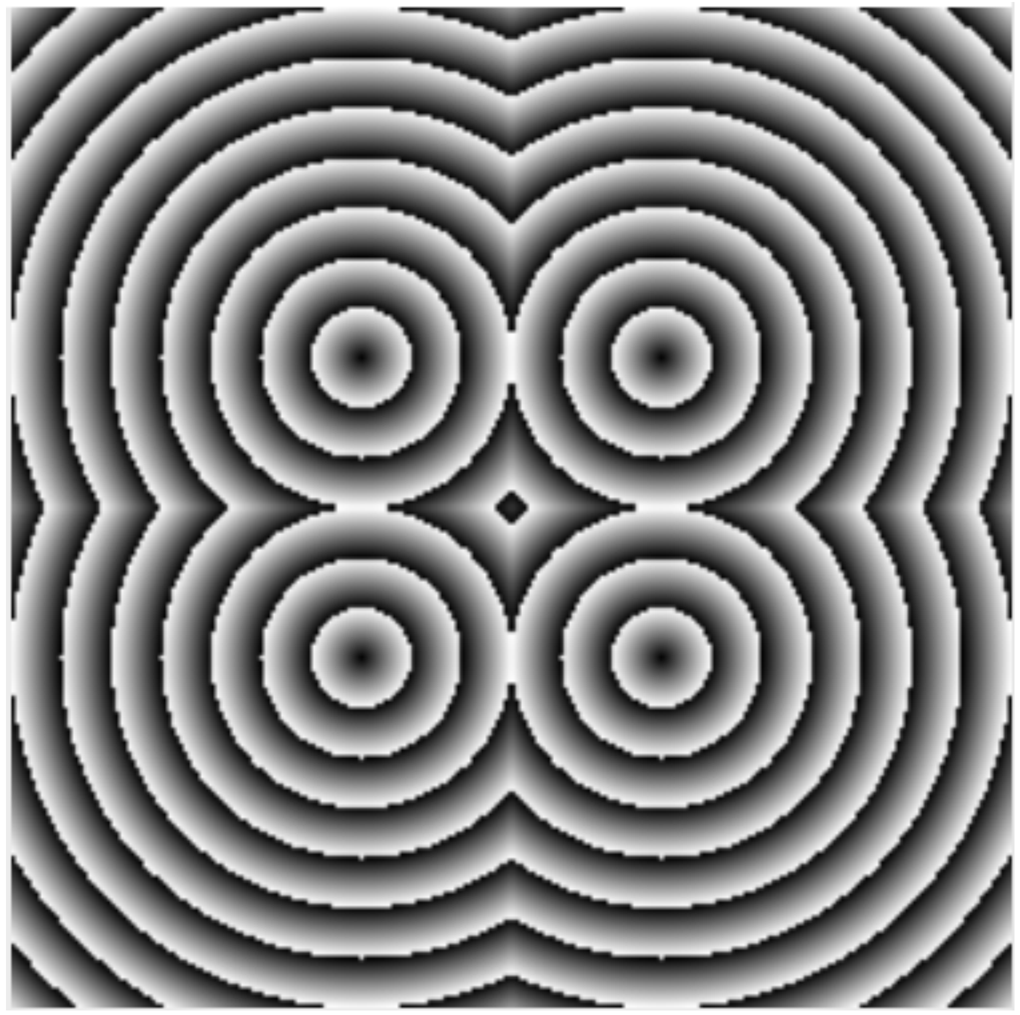
Distance maps

= Distance transforms

= Distance fields

An image where each pixel holds the distance to the nearest object (seed) pixel.

Can be extended to hold a pointer to the seed (= a Voronoi diagram)



Origin

First published in 1966 by Rosenfeld & Pfalz with simple metrics

Very fast sequential implementation!

Later refined to better metrics

1980: The Euclidean Distance Transform by Danielsson including an efficient parallel algorithm, "Jump flooding".

2011: Gustavson & Strand made the "Anti-aliased EDT" for sub-pixel precision. Vector-based version (suitable for Voronoi diagrams) by myself 2017.

Sequential implementation

In 2D, 3 or 4 scans over the image.

Non-eucliean (Rosenfeld 1966) needs only two scans.

4-scan EDT by Danielsson 1980

Symmetric version (Ragnemalm 1991):

(-1,-1)	(0,-1)
(-1,0)	(0,0)

(0,-1)	(1,-1)
(0,0)	(1,0)

(-1,0)	(0,0)
(-1,1)	(0,1)

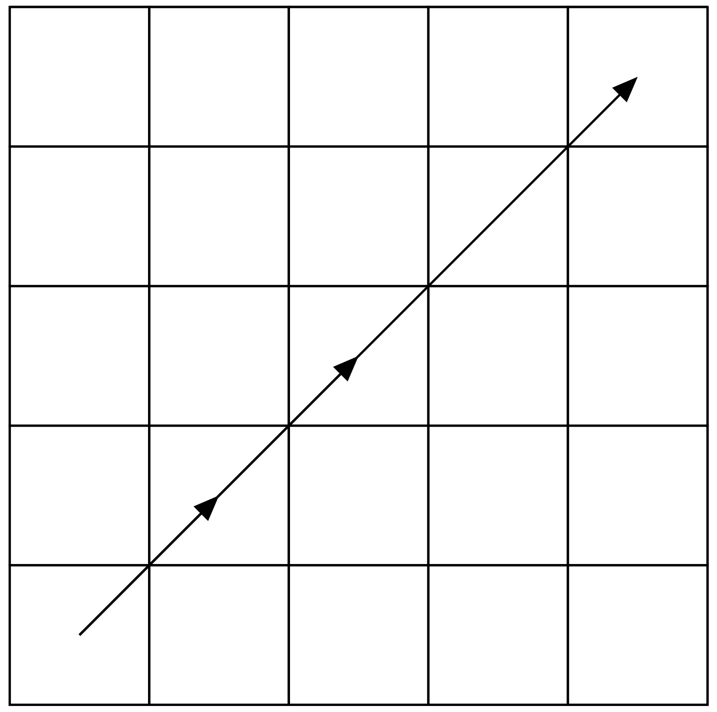
(0,0)	(1,0)
(0,1)	(1,1)

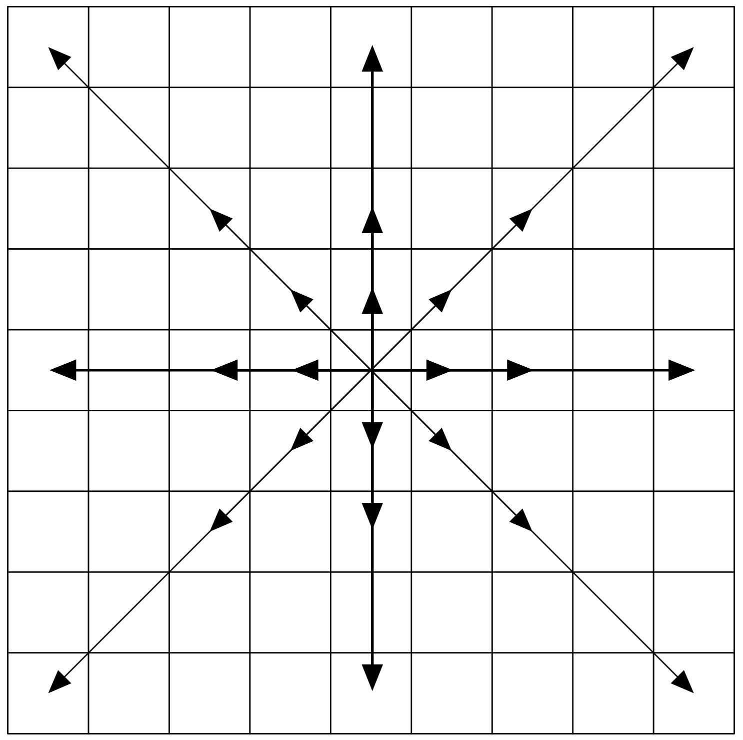
Parallel implementation - Jump flooding

Danielsson 1980

Takes steps of increasing length

Simple implementation, very fast on GPUs





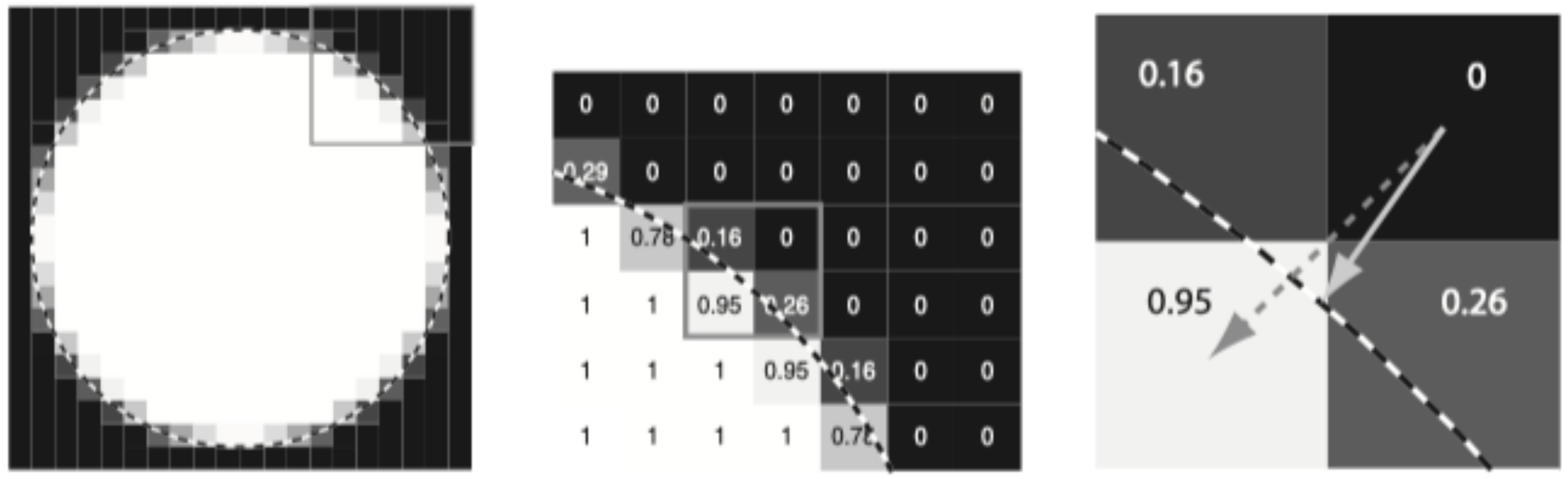
Anti-aliased EDT

Gustavson & Strand 2011

Adds an offset due to the intensity of edge pixels.

Approximates the edge location from the grayscale value.

Produces a very smooth distance map.

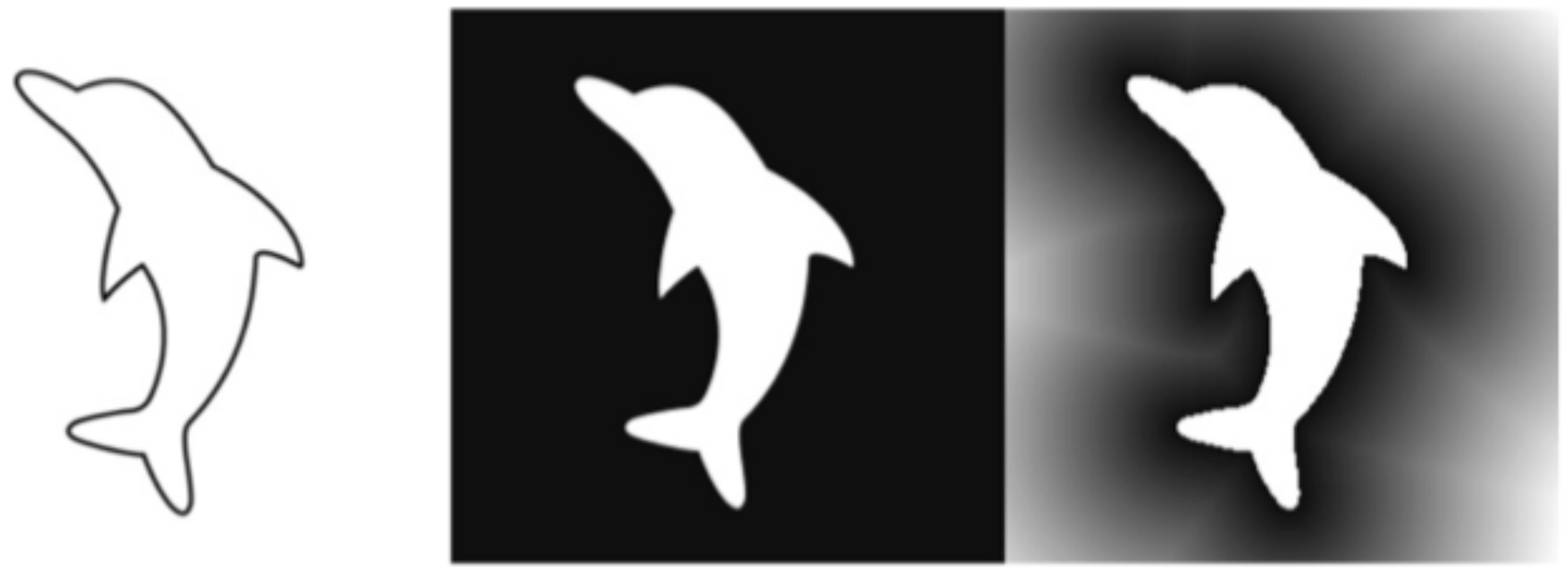


(a) (b) (c)

Stefan's test image and its distance map

Very nice and smooth.

The smoothness opens for more applications



Applications of distance maps

Surprisingly many!

- Graphical effects around objects, like glow
- Voronoi diagrams (and its applications)
- Edge smoothing
- Morphological operations
- Acceleration of ray marching
- Advanced bump mapping variants

and more!

Relevance for this course

Acceleration of large Voronoi noise

Adding effects like glow

For procedural animations, create data for good movement paths

Base for interesting patterns

Can be used for anti-aliasing (later this lecture)!

More...

Back to the Voronoi noise

A full distance transform is the general solution to an arbitrary large set of points with arbitrary positions

Can you cheat it?

If you put restrictions on the placement of seeds, you can force

- compact Voronoi polygons
- faster computation by ignoring far away seeds

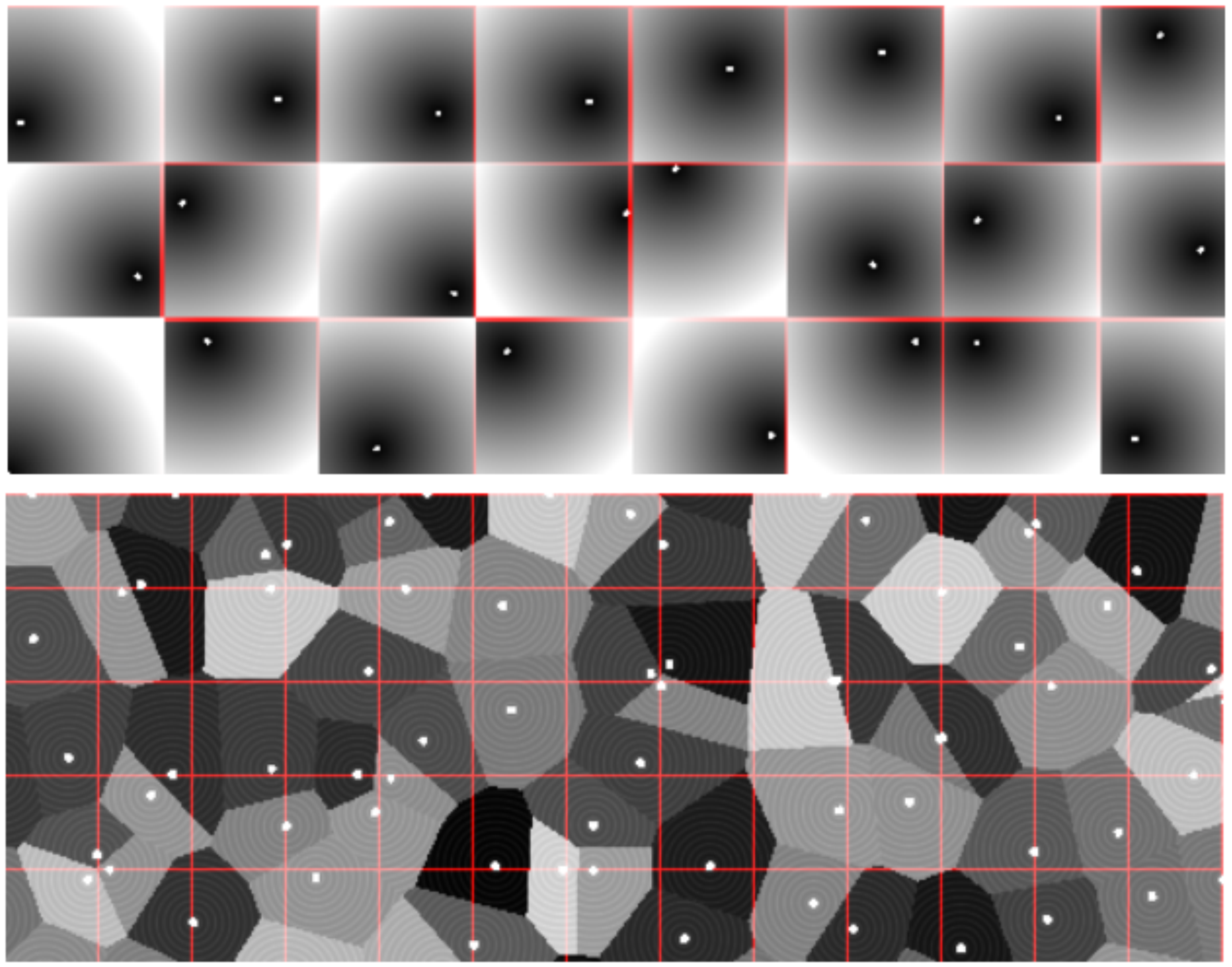
Tiling

One popular restriction is tiling.

Each seed is placed in a random position in a specific tile, square space, where no other seed goes.

- We only have to check the 8 closest neighbors. (Almost.)
- The result tends to be relatively "relaxed" with compact Voronoi areas.

However, the grid restriction limits the pattern to certain axis-dependency.

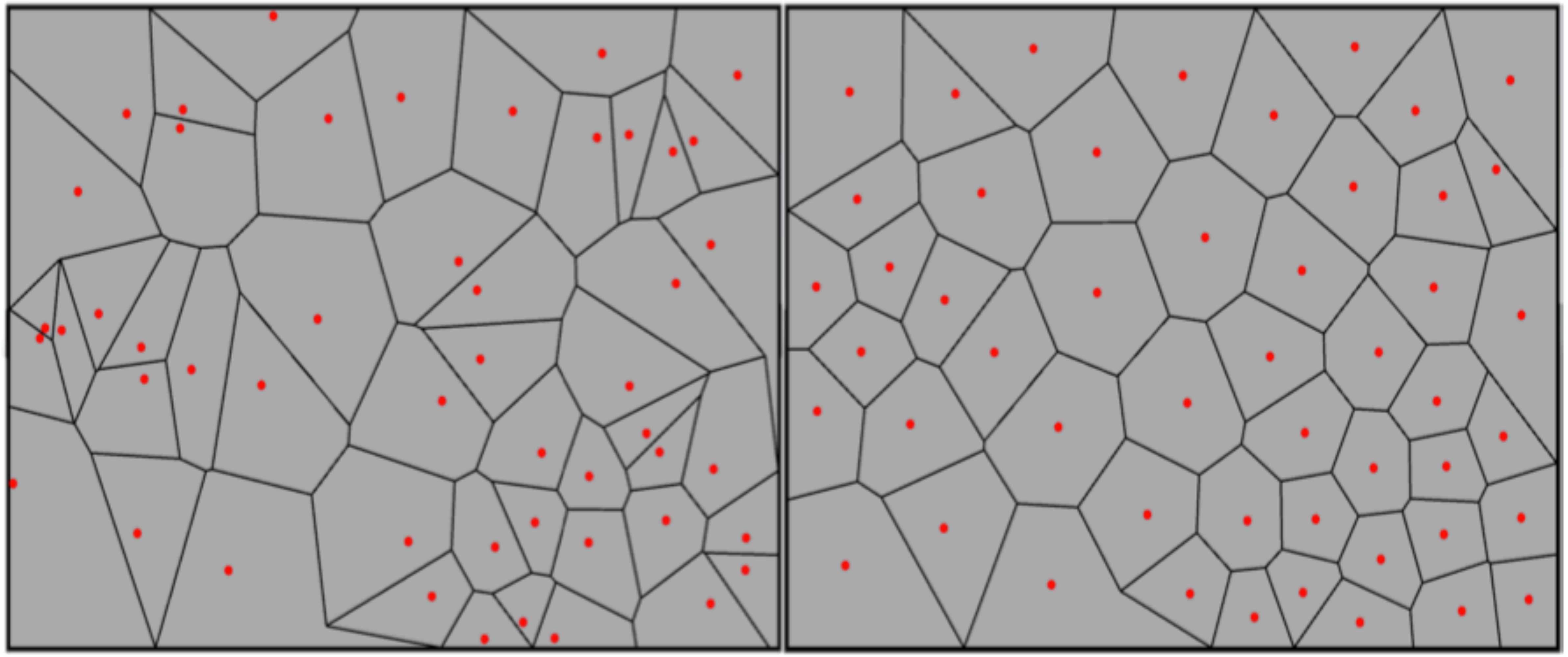


From Book of shaders

Refinement of Voronoi noise

We can make it totally random

or make sure that it is visually pleasing for our application

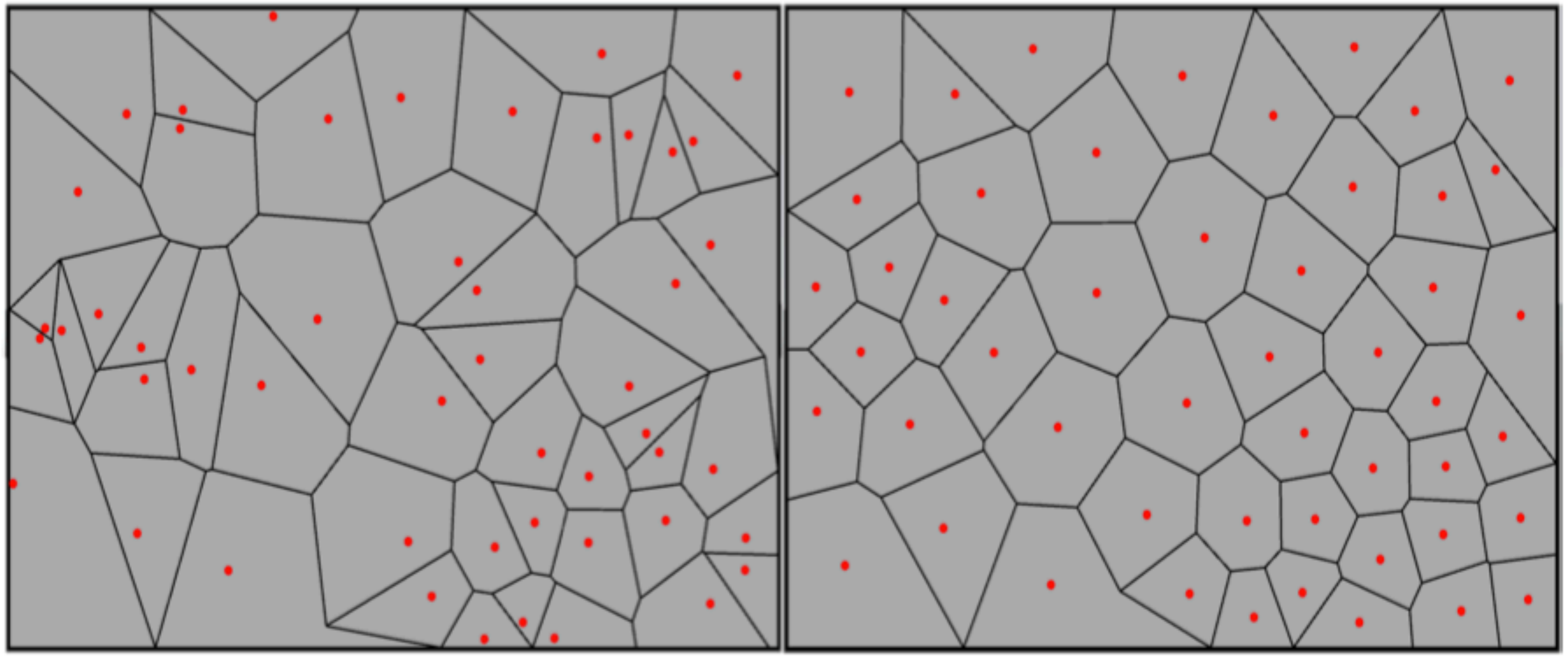


Lloyd's relaxation

Moves the seeds to better locations

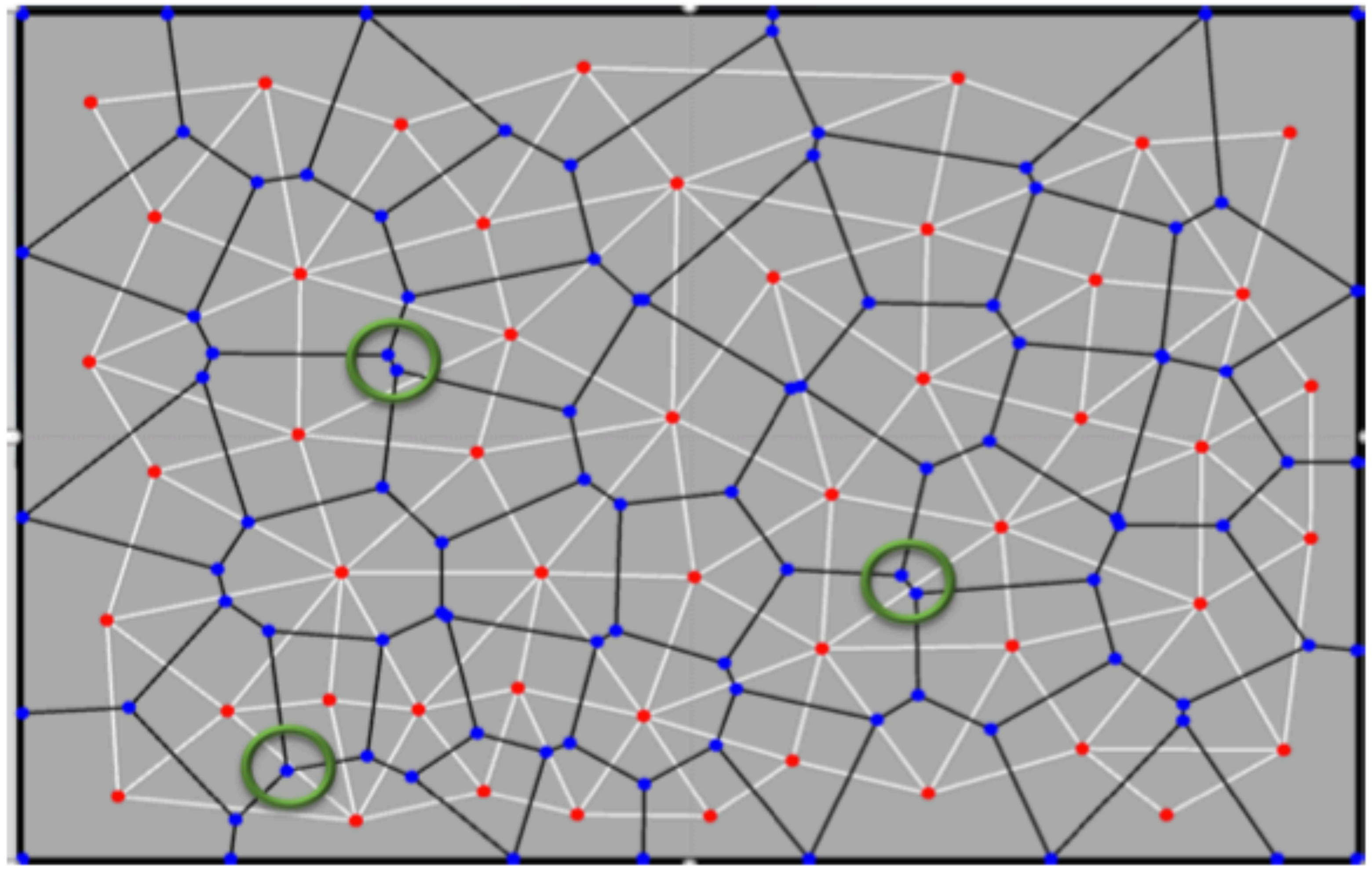
Push away close seeds from each other

Lloyd 1982



Going even further?

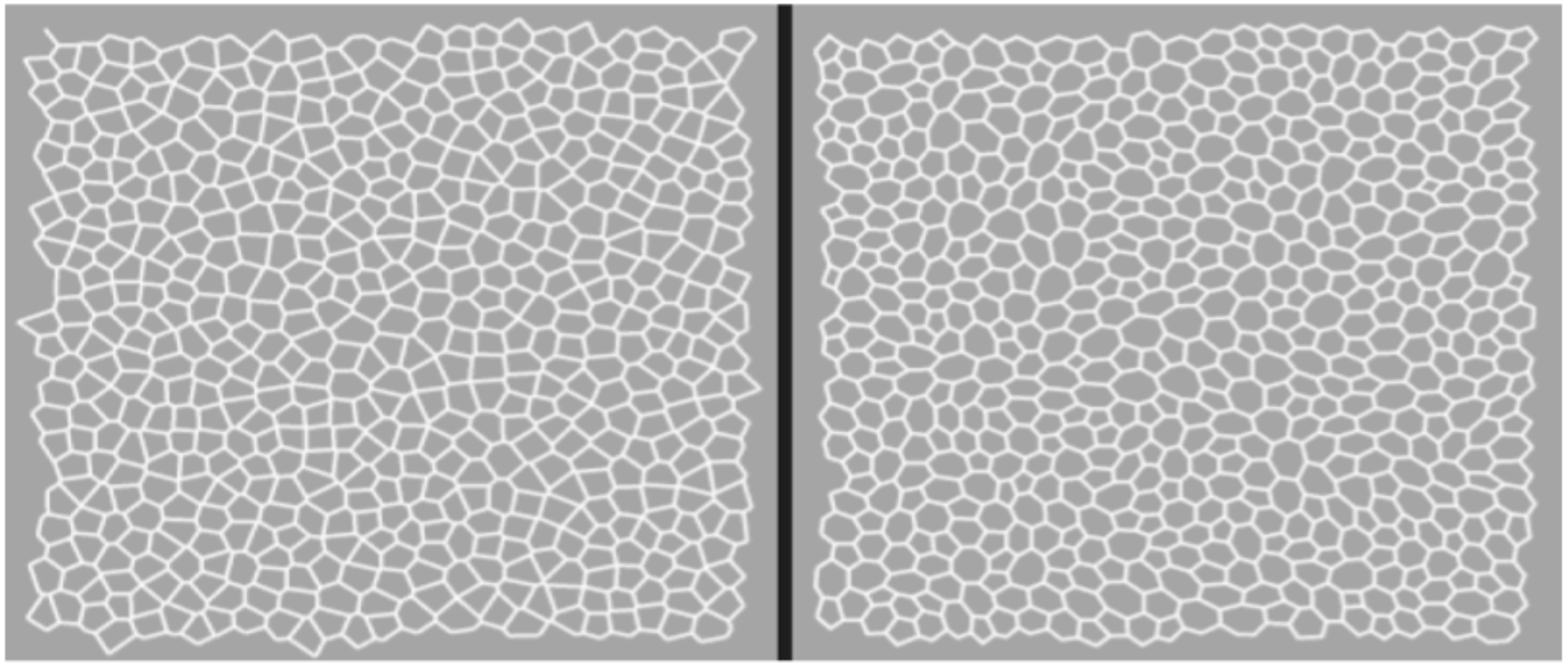
You might still dislike certain features in the Delaunay tesselation



Dahlquist's procedural maps

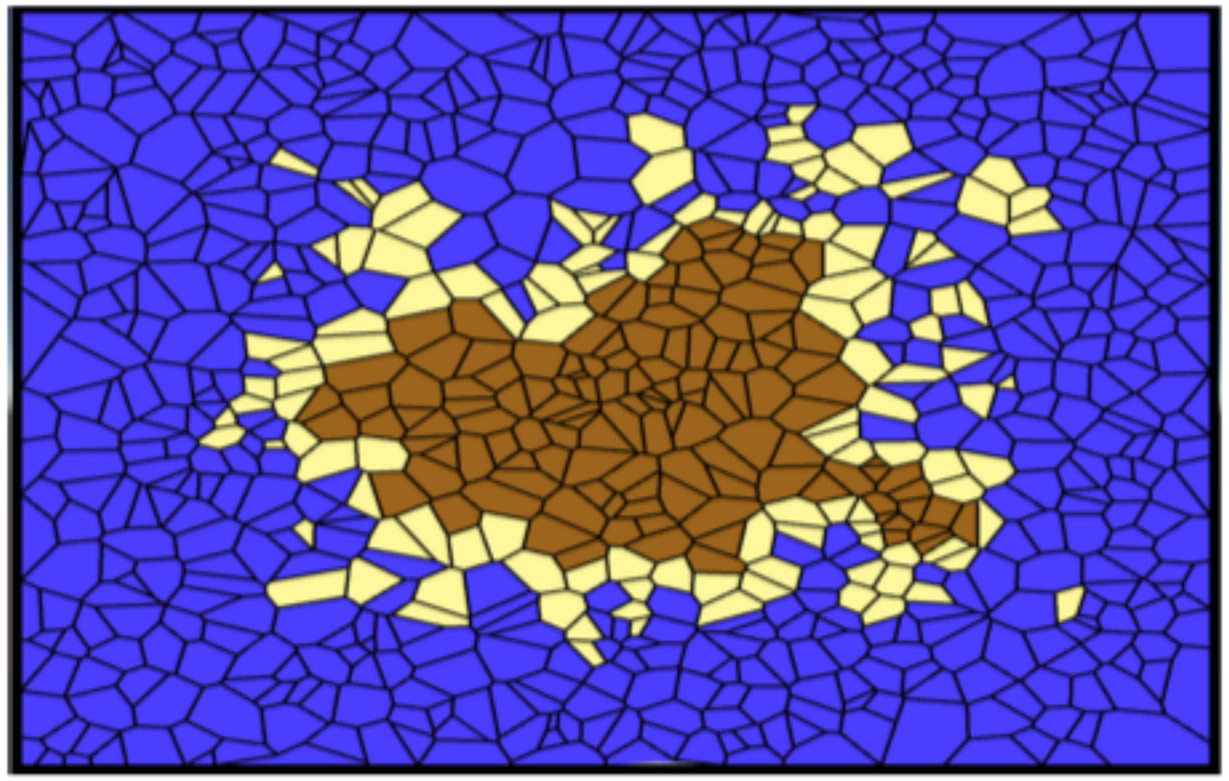
His goal was to create procedural maps usable for e.g. games

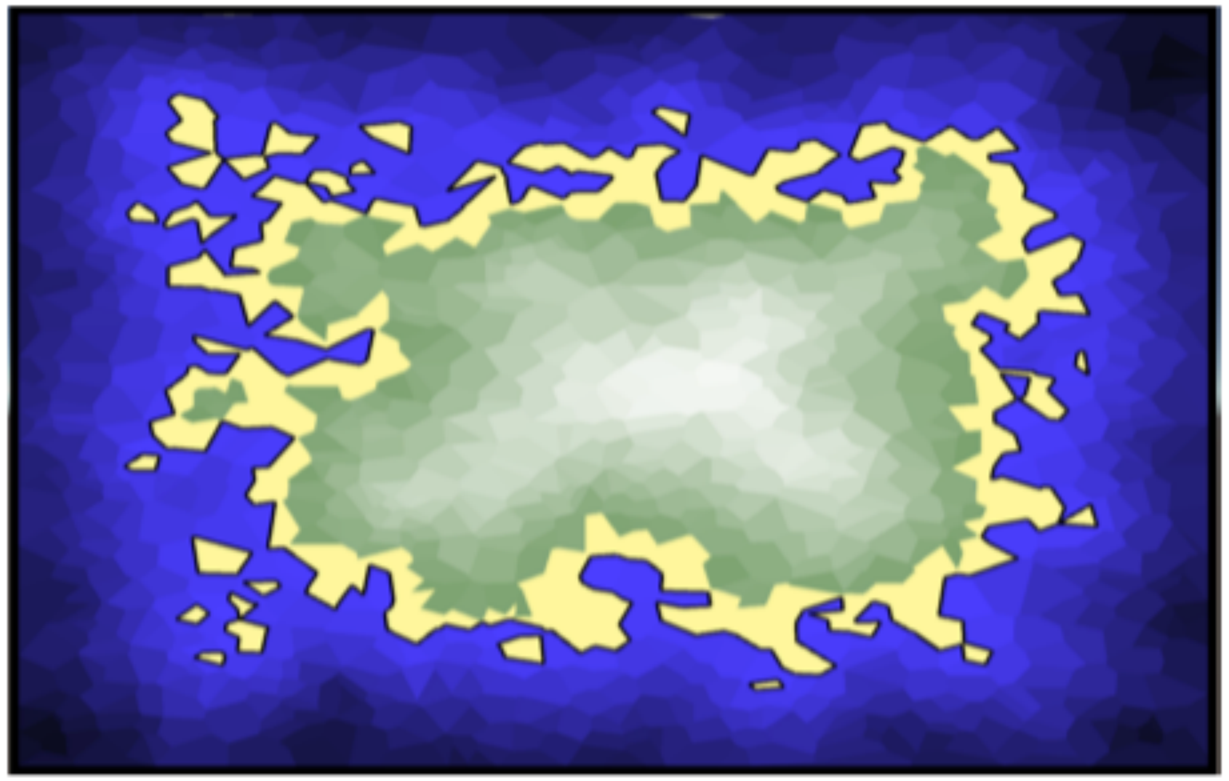
Used the above relaxations. Not true Delanay/Voronoi in the end.



Height data for land and water etc

Multi-level Perlin noise (FBM) tuned by distance to edge

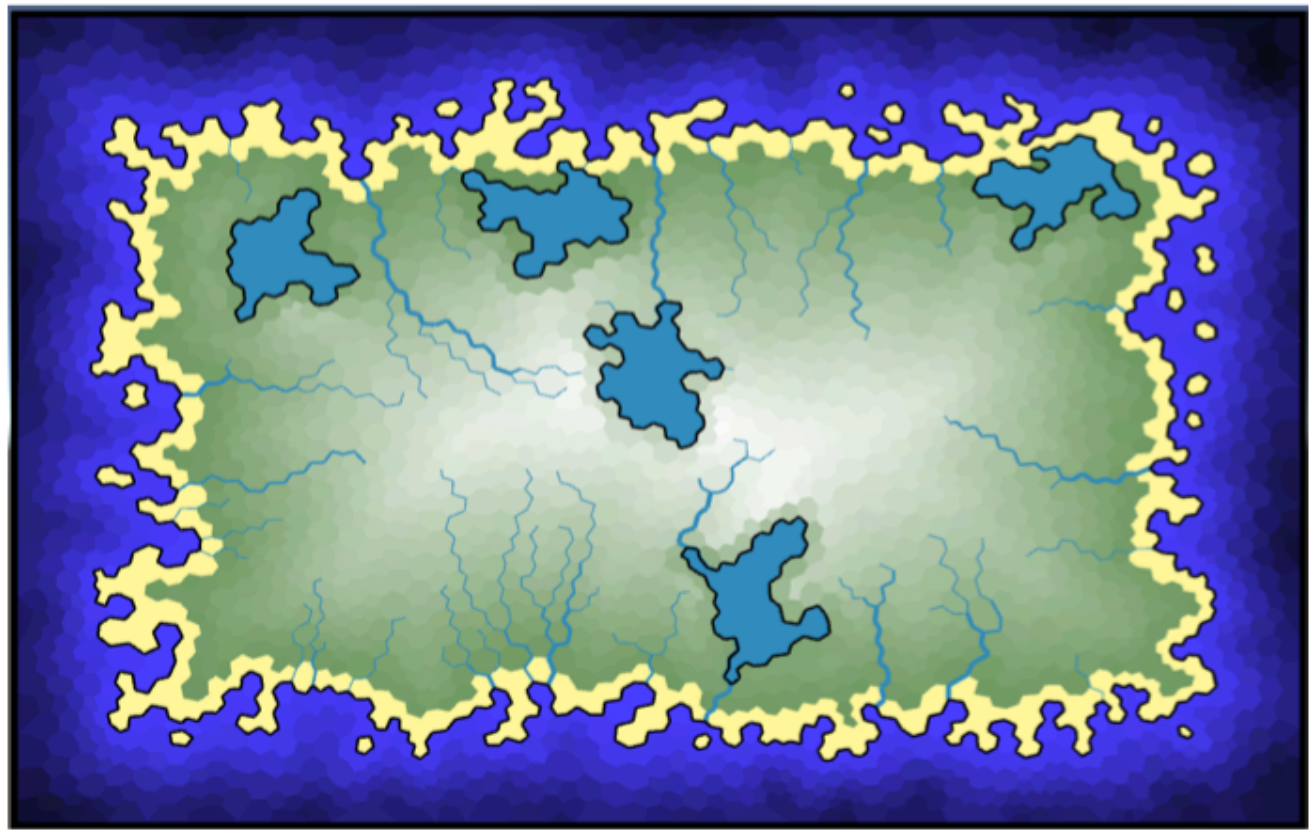




Lakes and rivers

Lakes found as isolated low areas

Rivers created somewhat arbitrarily



One more time: Transformations

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

applied elsewhere

Color transformation

You can transform colors with a matrix!

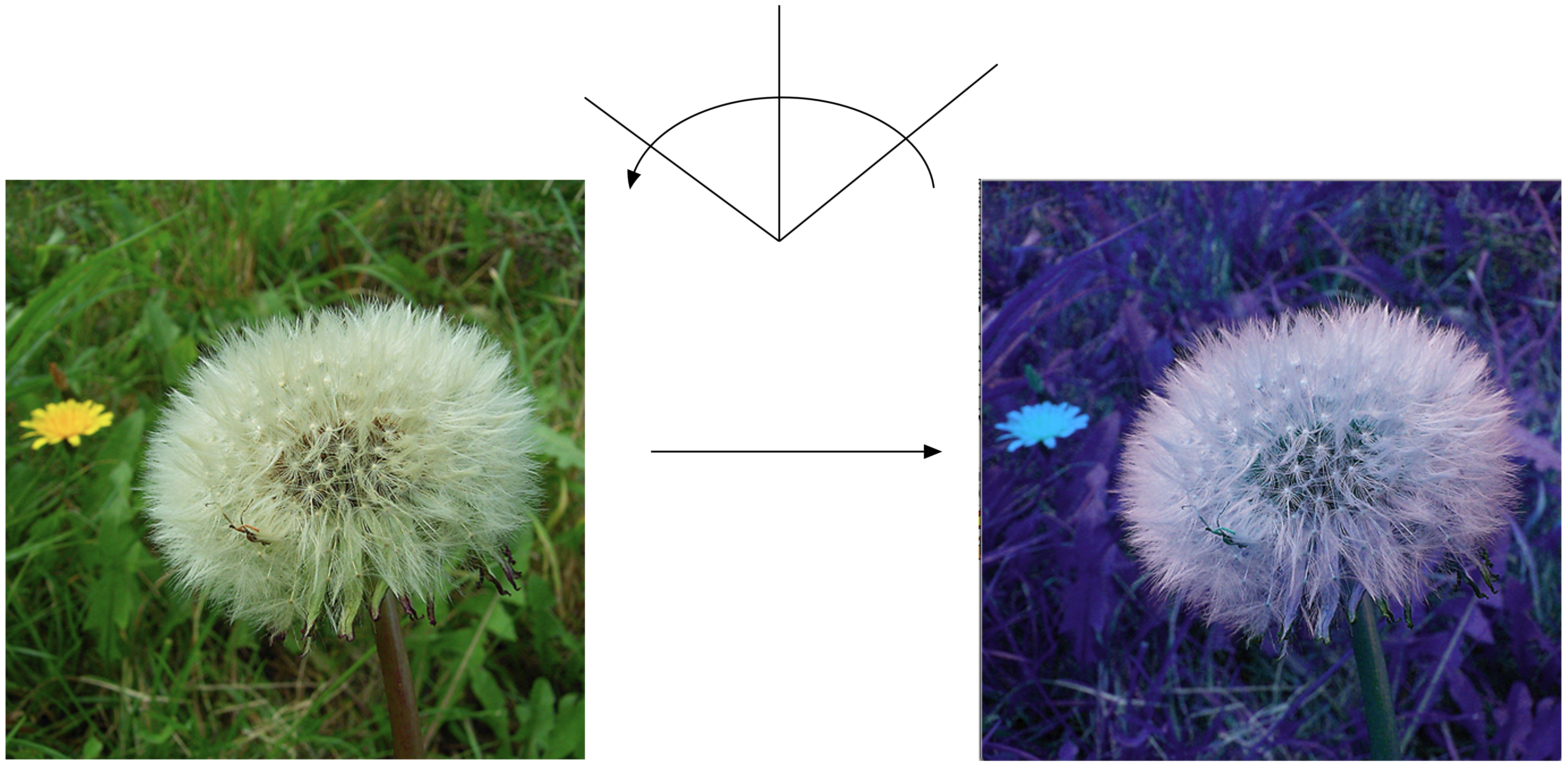
Example: Transform between color formats:

```
// YUV to RGB matrix
mat3 yuv2rgb = mat3(1.0, 0.0, 1.13983,
                    1.0, -0.39465, -0.58060,
                    1.0, 2.03211, 0.0);

// RGB to YUV matrix
mat3 rgb2yuv = mat3(0.2126, 0.7152, 0.0722,
                   -0.09991, -0.33609, 0.43600,
                    0.615, -0.5586, -0.05639);
```

Rotating colors

```
    mat4 colorMatrix;
    colorMatrix = Rz(time); // Rotate around blue
    colorMatrix = Rx(time); // Rotate around red
    colorMatrix = ArbRotate(SetVec3(1,1,1), time); // Rotate around rgb axis
    glUniformMatrix4fv(glGetUniformLocation(program, "colorMatrix"), 1,
GL_TRUE, colorMatrix.m);


uniform mat4 colorMatrix;

void main(void)
{
    vec4 color = texture(tex, texCoord);
    outColor = colorMatrix * color;
}
```

Rotates in RGB space



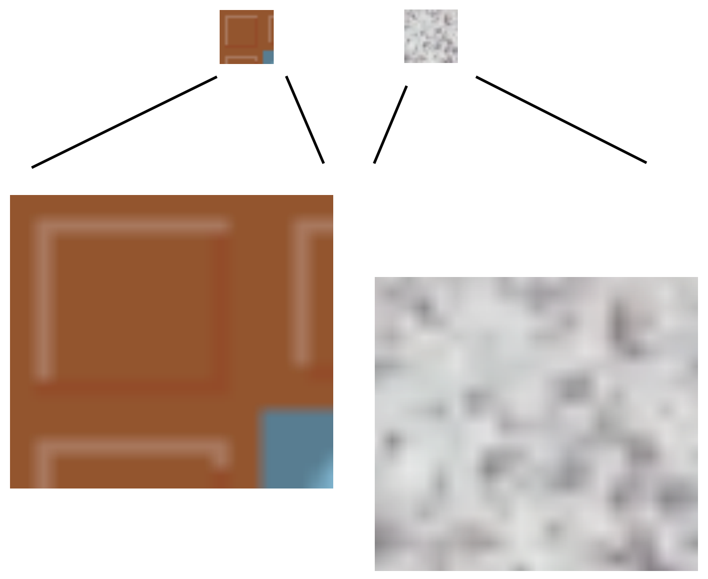
Texture coordinate transformations

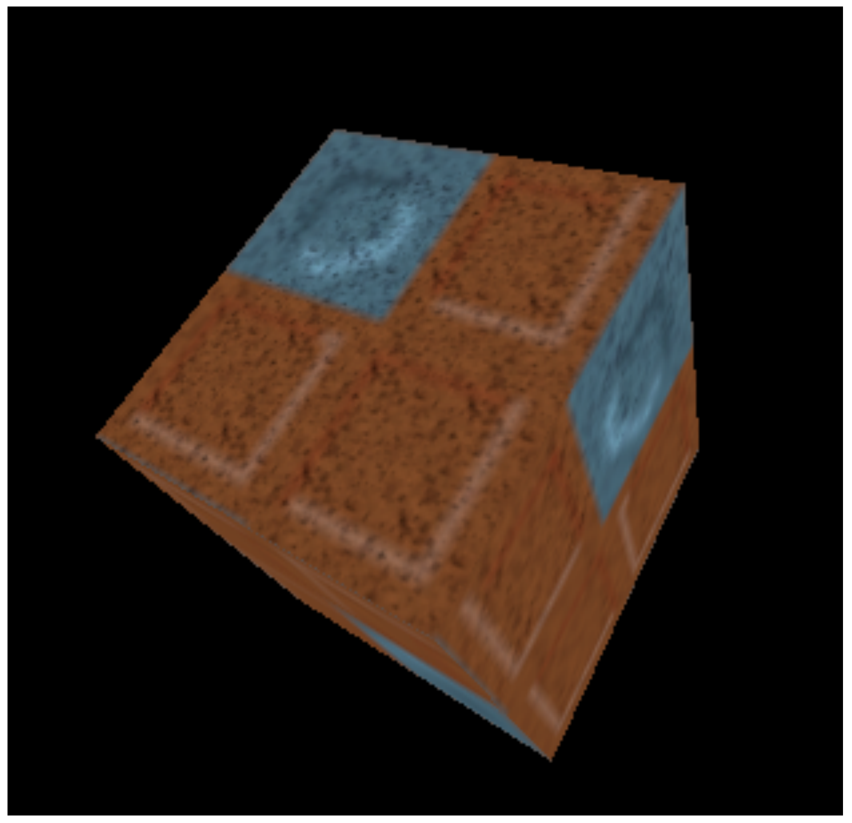
Don't take texture coordinates as something static that you can't change!

Detail textures

Combine a high frequency texture with a low frequency one

Application of *multitexturering.*

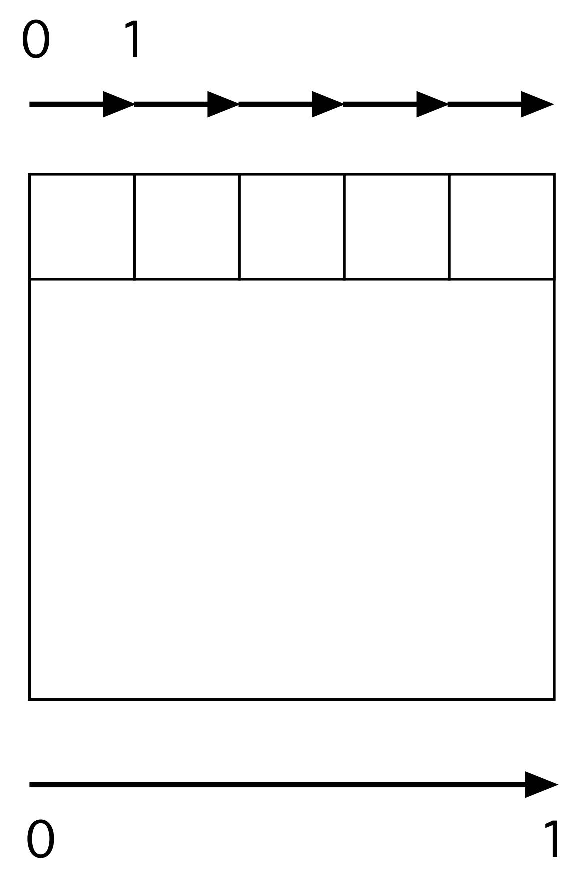




Coordinates

Detail texturing is performed with different coordinates per texture:

Low frequency texture: 0 to 1
High frequency texture: 0 to detailLevel
(with repeating texture)



Scrolling textures

Offset a texture by another, moving texture.
Very good way to make procedural water!

The demo is much, much more interesting...

Scrolling textures

Combine textures, move textures, mix textures, affect texture coordinates.

In the example fragment shader:

Access texture 1, offset by time.
Access texture 2, offset by texture 1.

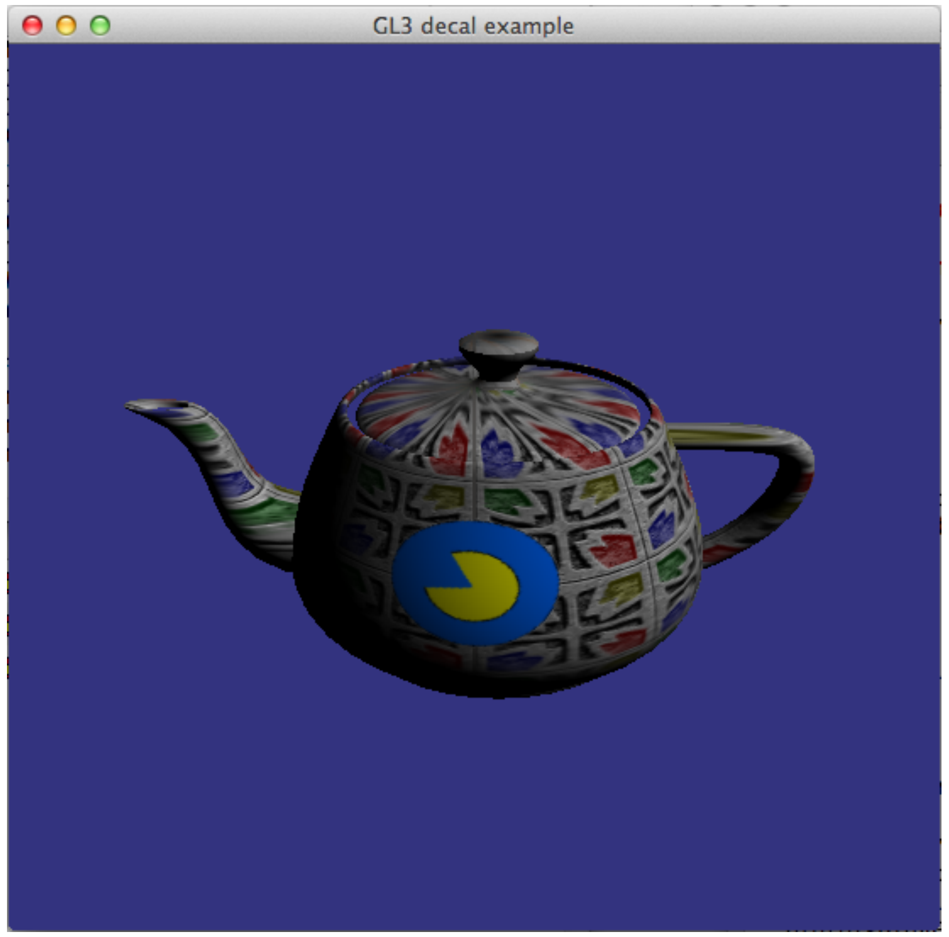
```
void main(void)
{
  float time=iTime*0.0001;

  vec4 t1 = texture(tex1, texCoord + vec2(time * t1scalex,
                    time*t1scaley));
  vec4 t2 = texture(tex2, texCoord + vec2(t1)  * t2scale);

  outColor = t2;
}
```

Conclusion: We do not need to map 1:1 with the texture coordinates.
Different textures/functions can be mapped differently!

Example: Decal

- Utah Teapot
- Spherically mapped surface texture
- Linearly mapped decal with GL_CLAMP_TO_EDGE

Place as you please



Texture placement

Models usually to have pre-generated texture coordinates.

You can easily scale and translate (which many did in lab 1)

But... why not do this with a matrix?

Use a rotation matrix to rotate the texture!

Apply the matrix on texture coordinates!

Texture matrix

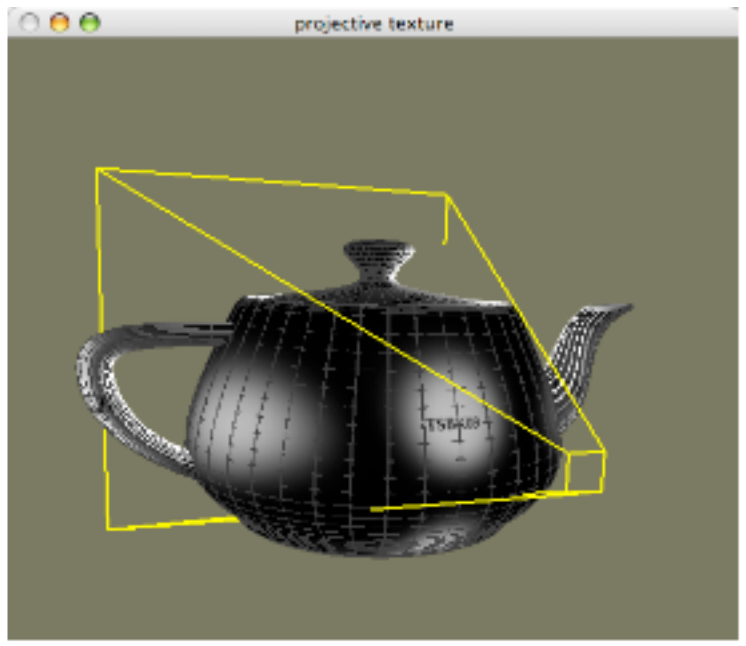
Simple example with texture matrix and multipass texturing

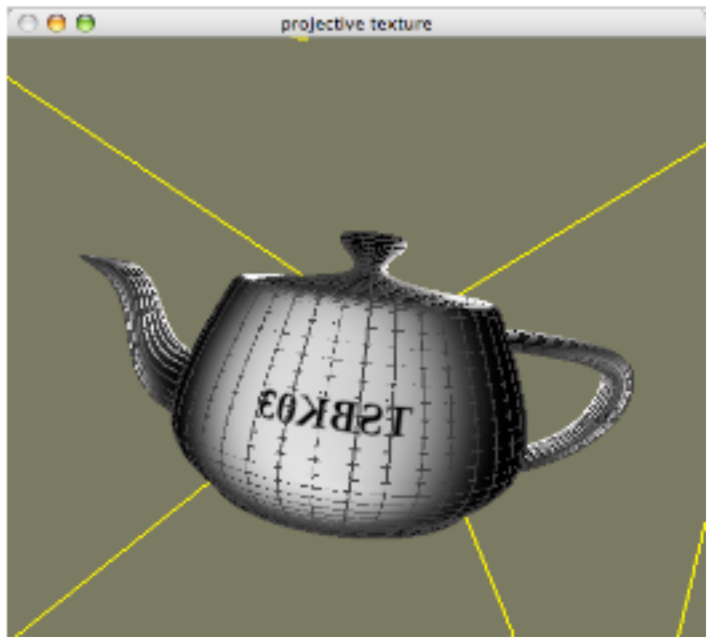
We can use any transformations for moving selected parts of the texture

Projected textures

Special case of texture placement: Projection matrix!

Used for shadow generation (especially shadow mapping).





More on shape generation and shaders

Usually a function that tells whether we are inside or outside a shape.

Simple case: A circle

```
radius = sqrt((x - center_x)^2 + (y-center_y)^2)
if (radius < circleradius)
  pixel = 1;
else
  pixel = 0;
```

What is wrong with this?

Avoid if-statements in shaders when possible

Bad habit for SIMD = bad habit for GLSL

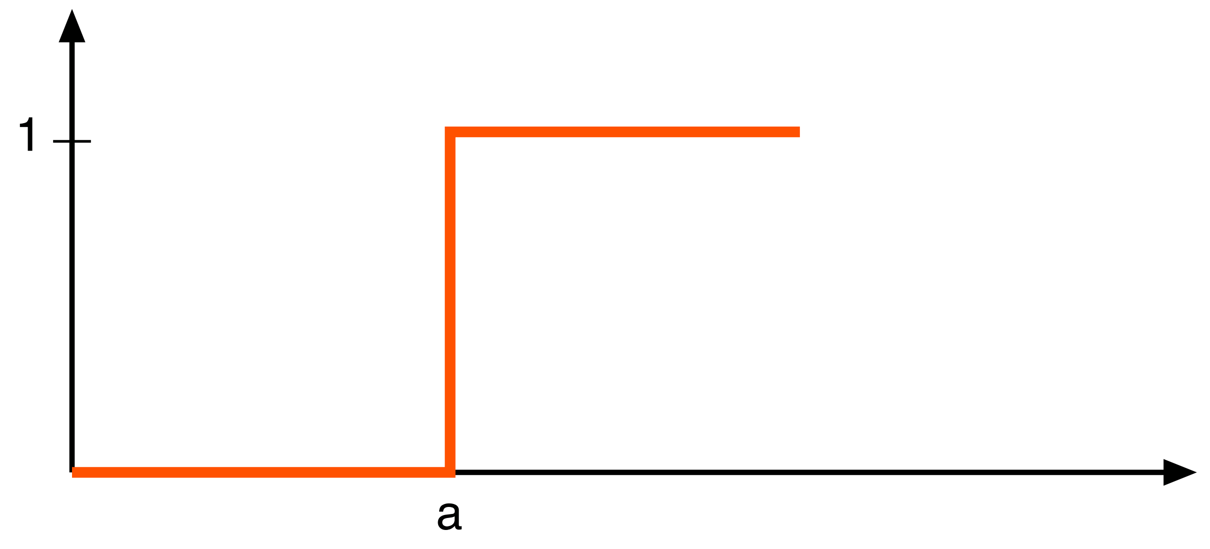
Use functions like step()

More important for if-statements with long multiple branches.

Step (Heaviside) revisited

A simple 0-1 transistion

```
float step(float a, float x)
{
   return (float) x >= a;
}
```



Frequency response infinite! Has ringing to infinitely high frequencies!

Anti-aliasing of shapes

Thus, step() isn't good either!

Pixel-wise generated shapes have problems with aliasing.

Special function for reducing this: aastep()

Variant of smoothstep for anti-aliasing edges.

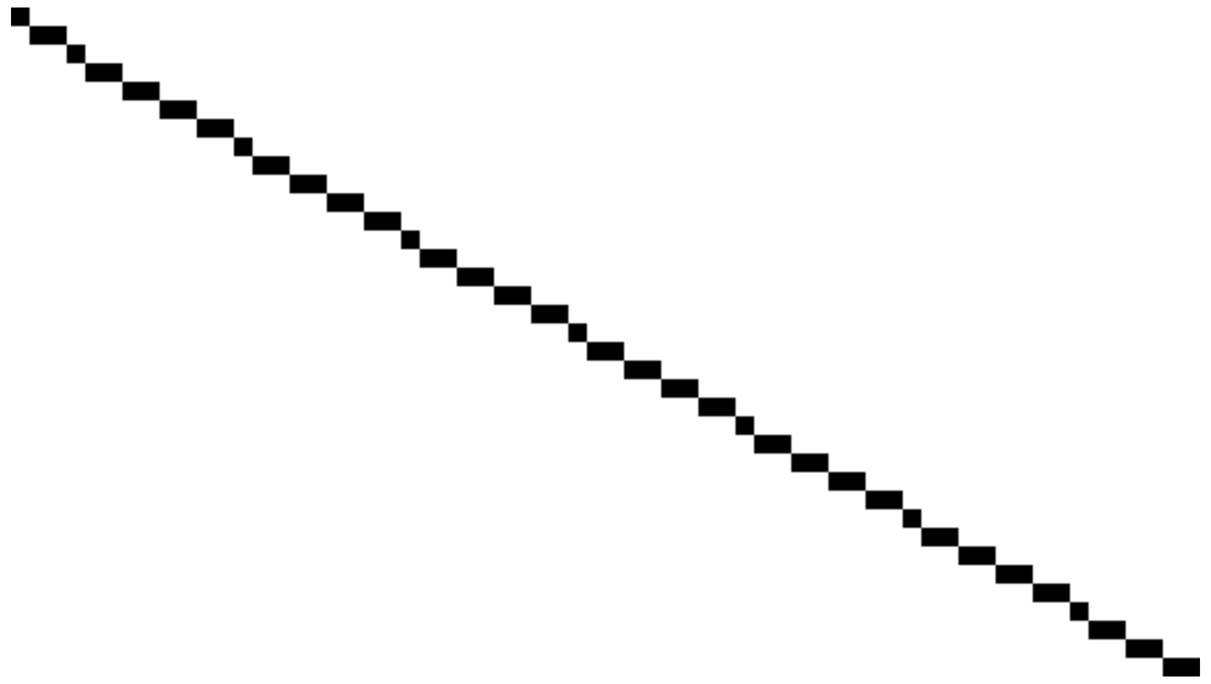
Conventional Anti-aliasing

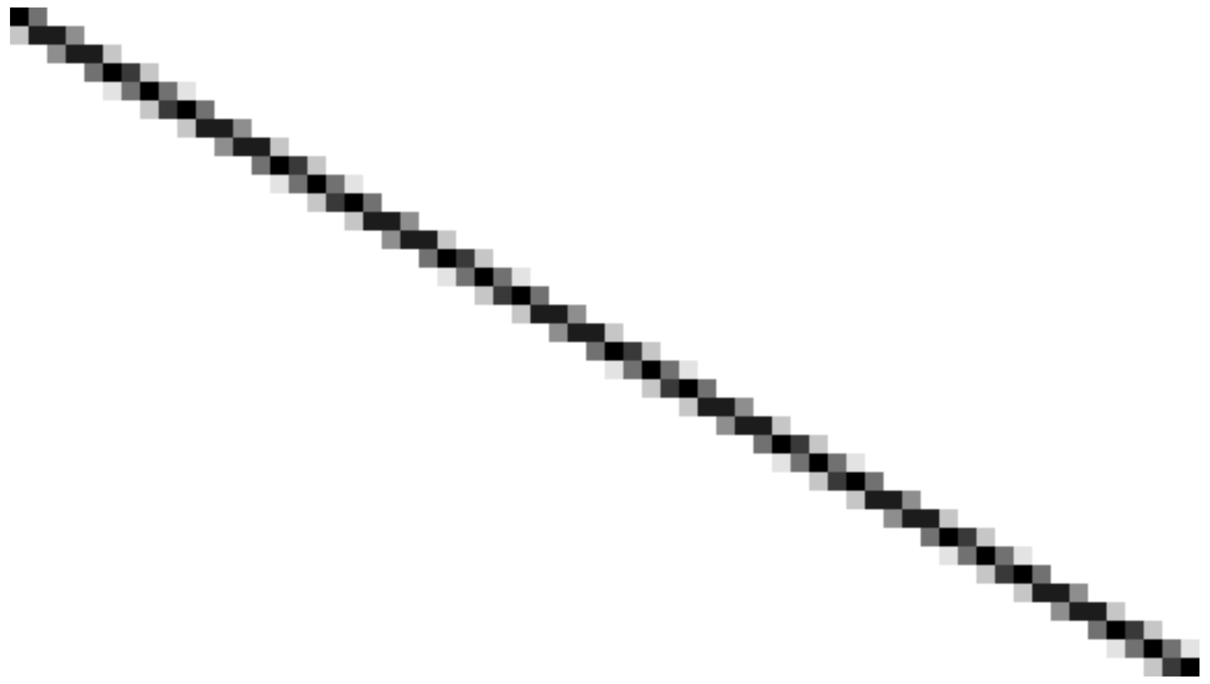
Supersamping

Multisampling

Post-processing of edges

Anti-aliasing





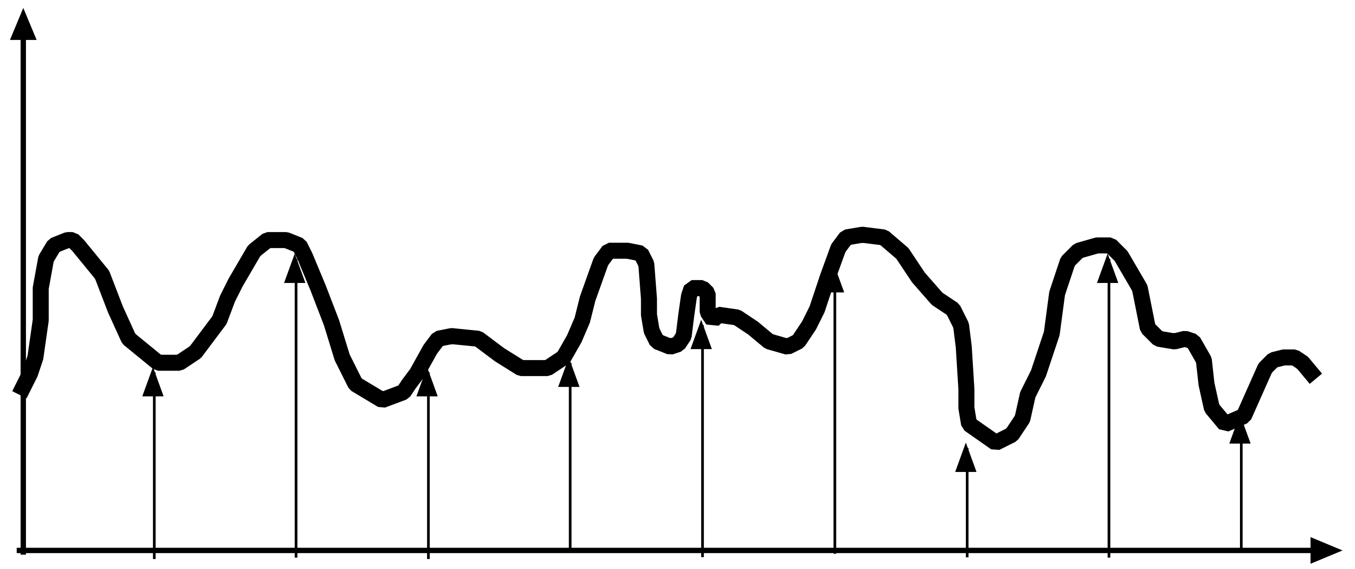
Frequency space

Any signal can be decomposed into sinus waves

The amplitudes of all sinus waves form a new space - frequency space

This space exists in any dimension. An image is a 2D signal and has a 2D frequency space.

Sampling



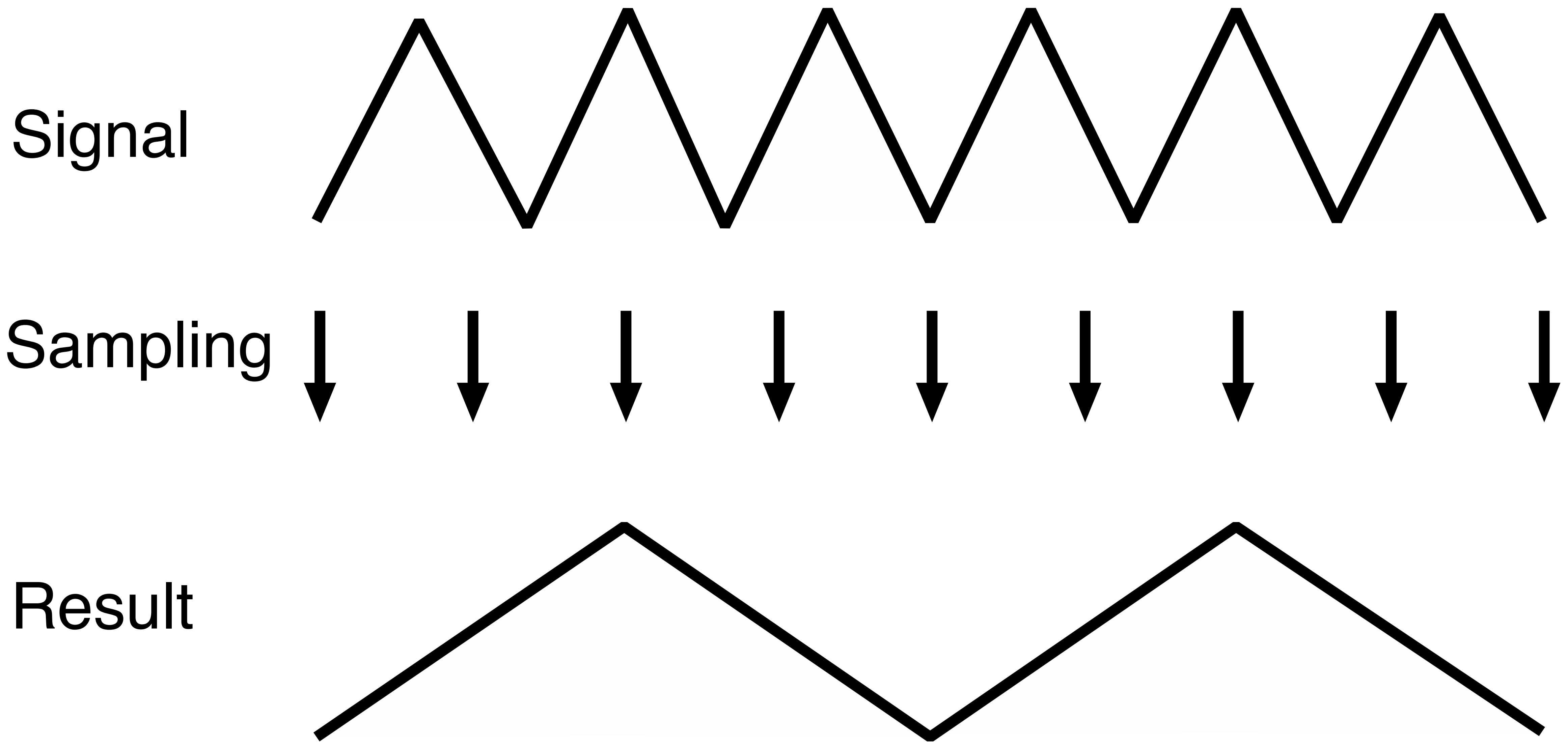
A digital image is a sampled version of an underlying analog 2D signal.

Sampling

When presenting the digital signal, it is reconstructed to an analog signal.

A signal can be decomposed into different frequency bands.

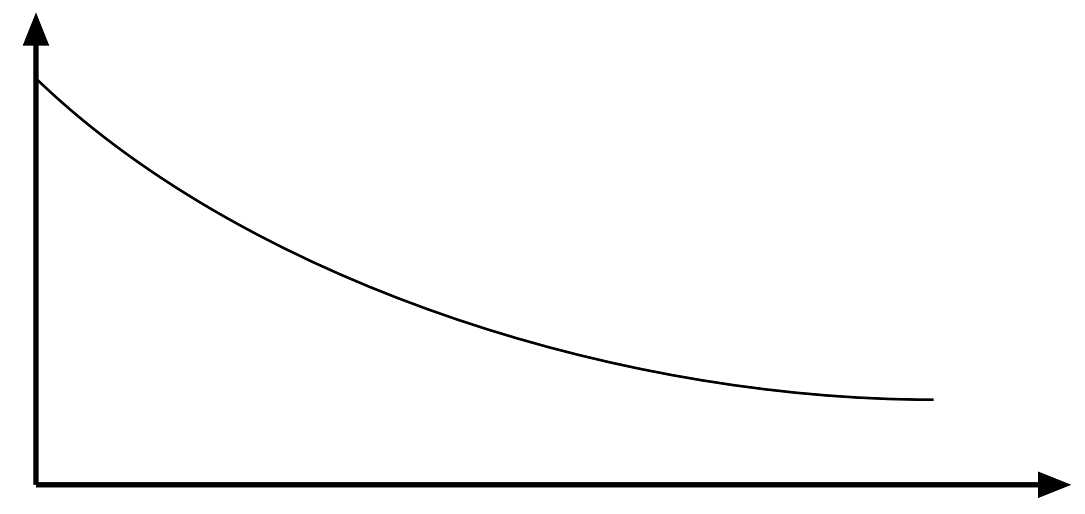
What parts of the original signal that are accurately reconstructed depend on the frequencies.

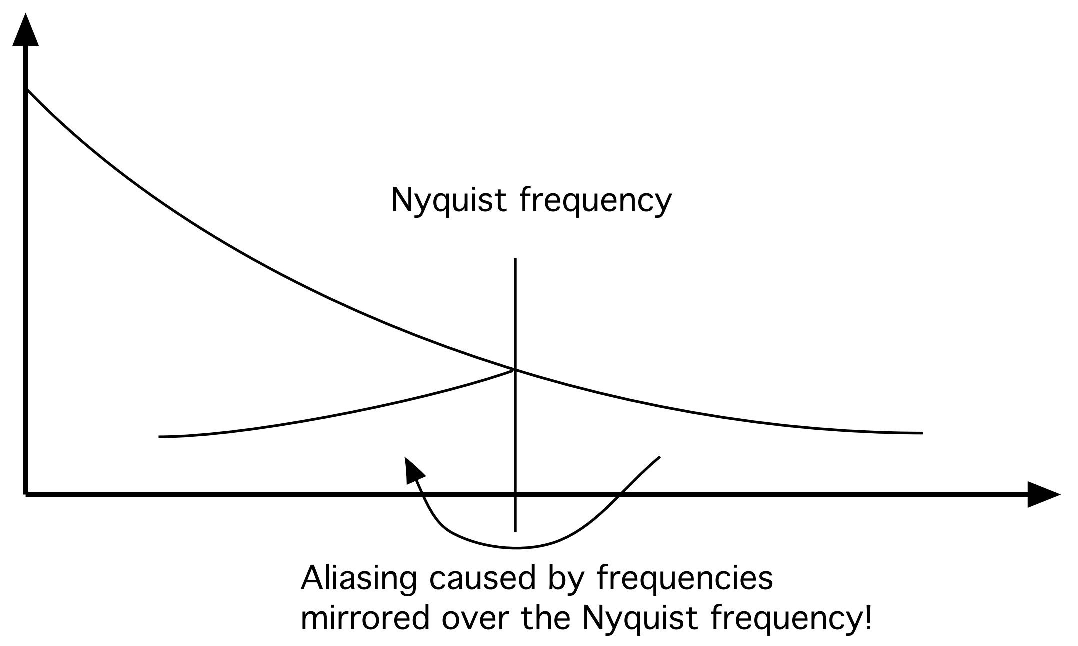

Signal
Sampling
Result

Think in frequency space!

Which frequencies are preserved and which cause problems?

Amplitudes usually lower for higher frequencies!

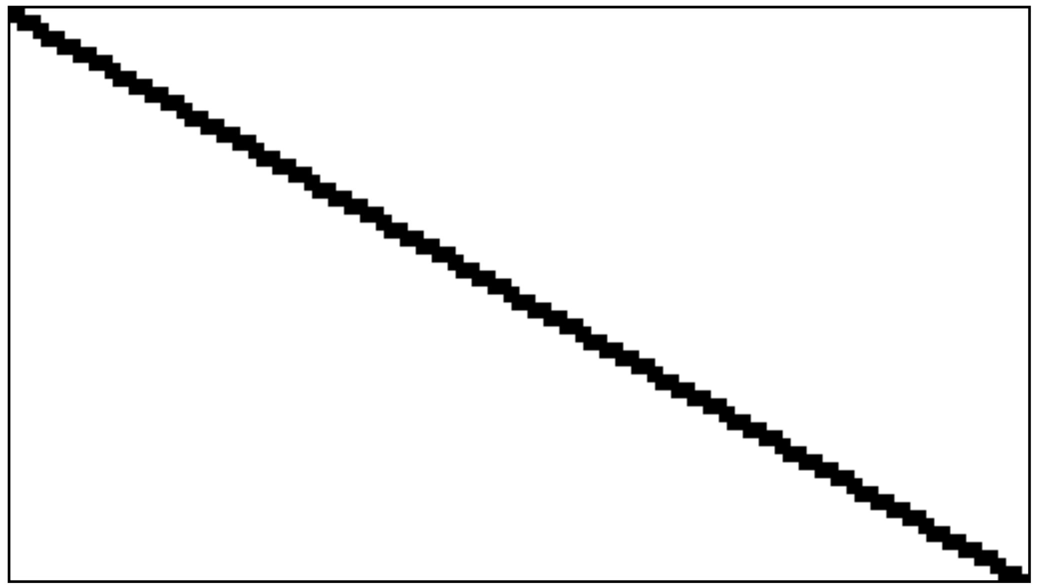




Anti-aliasing methods

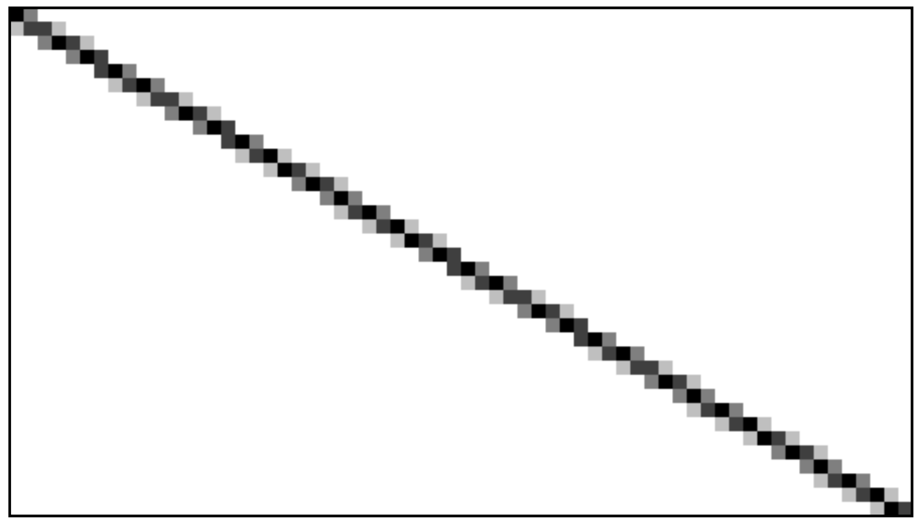
- Linear post-filtering. Bad.
- Super-sampling: Split pixels in sub-pixels. Average the result to the destination pixel.
- Area sampling: Calculate covered areas of each pixel.
- Non-linear post-filtering.

Supersampling



Simple but slow and memory-demanding

Sample down to destination buffer



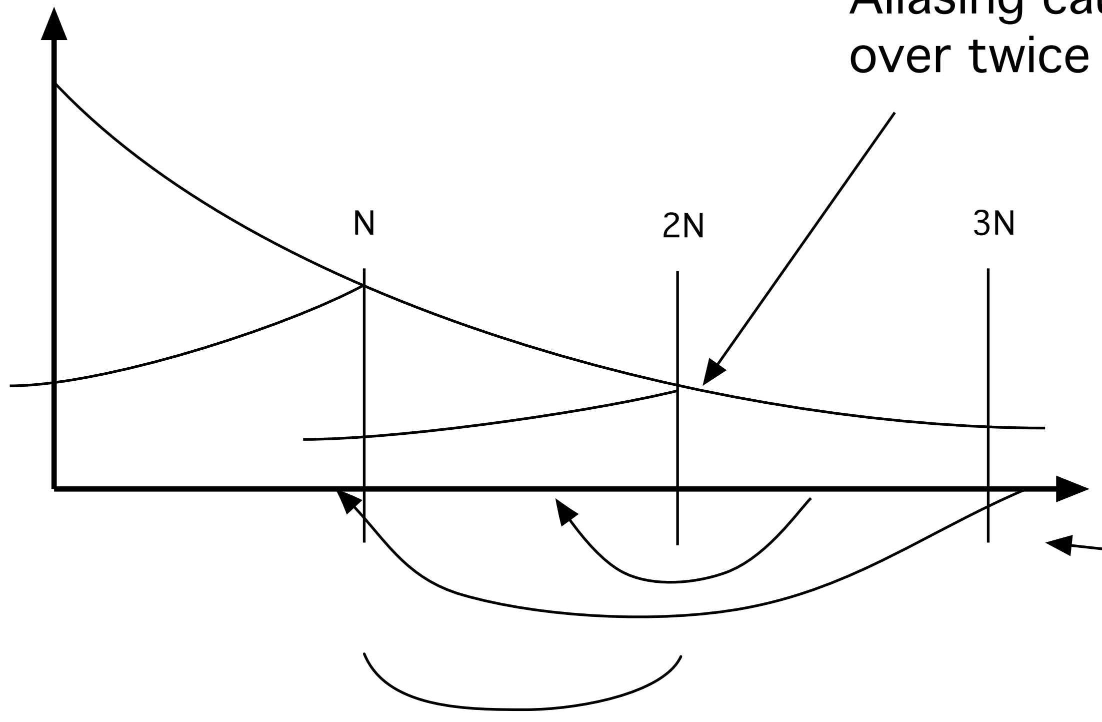
Draw in a high-res image buffer

Efficiency of supersampling

Assume amplitude variation by frequency 1/f

"The 1/f rule"

2x supersampling:



Aliasing caused by frequencies mirrored over twice Nyquist frequency

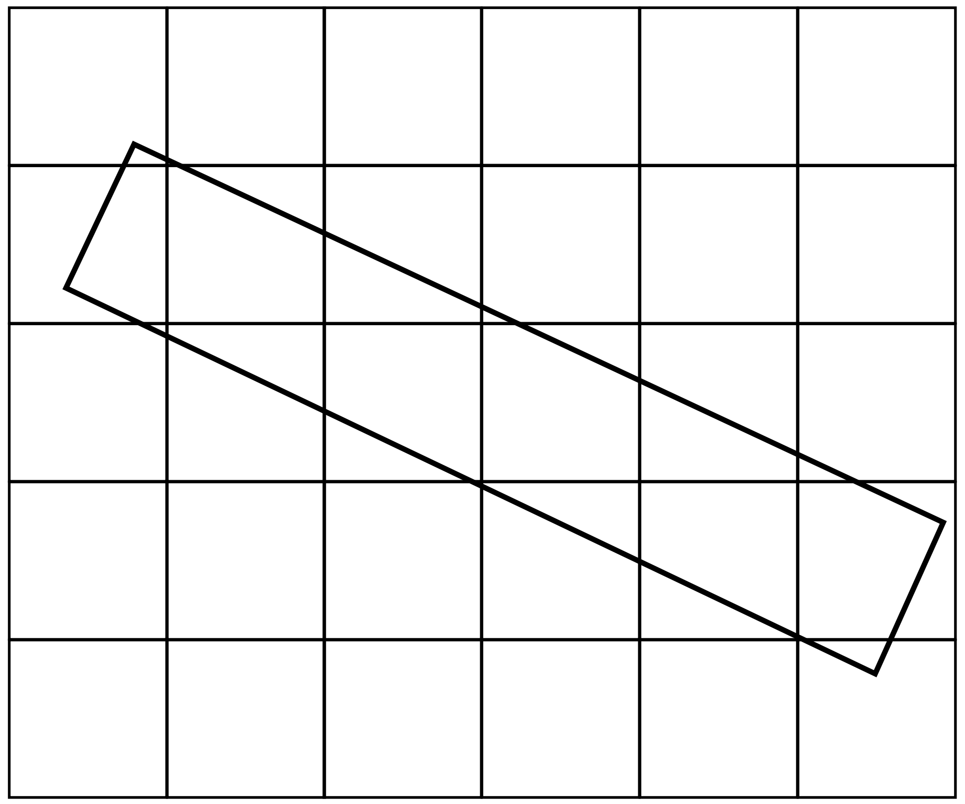
What happens for 3x supersampling?

Frequencies over 3N remain as errors! Approximately 1/3!

This is filtered away!

Area sampling

Determine how much of each pixel
that is covered by the shape



Requires knowledge of the
location of the border.

Can not anti-alias textures

Usually impractical... but for us?

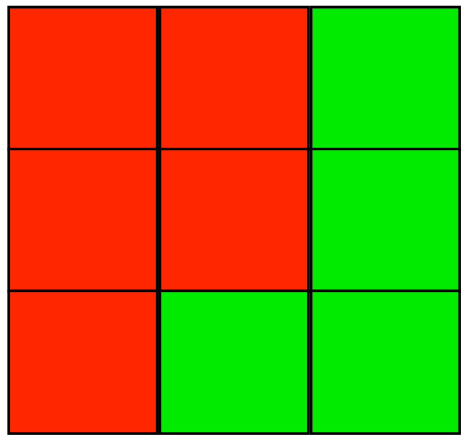
Multisampling

Variant of supersampling

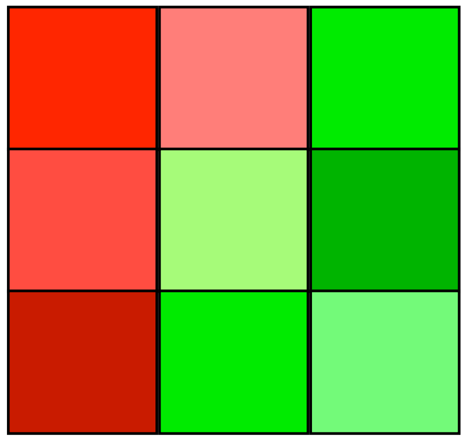
More efficient

Only improves edges

Multisampling



Multisampling: Only one execution of fragment shader for all samples from the same geometry



Supersampling: One execution of fragment shader for each sample

Multisampling

Less fragment processing

Fewer texture accesses

Same number of memory writes and same post-processing

FXAA = Fast approXimative AA

Post-processing

No higher resolution image

Non-linear filter

Don’t filter patterns

Several recent methods of this kind

Anti-aliasing in OpenGL

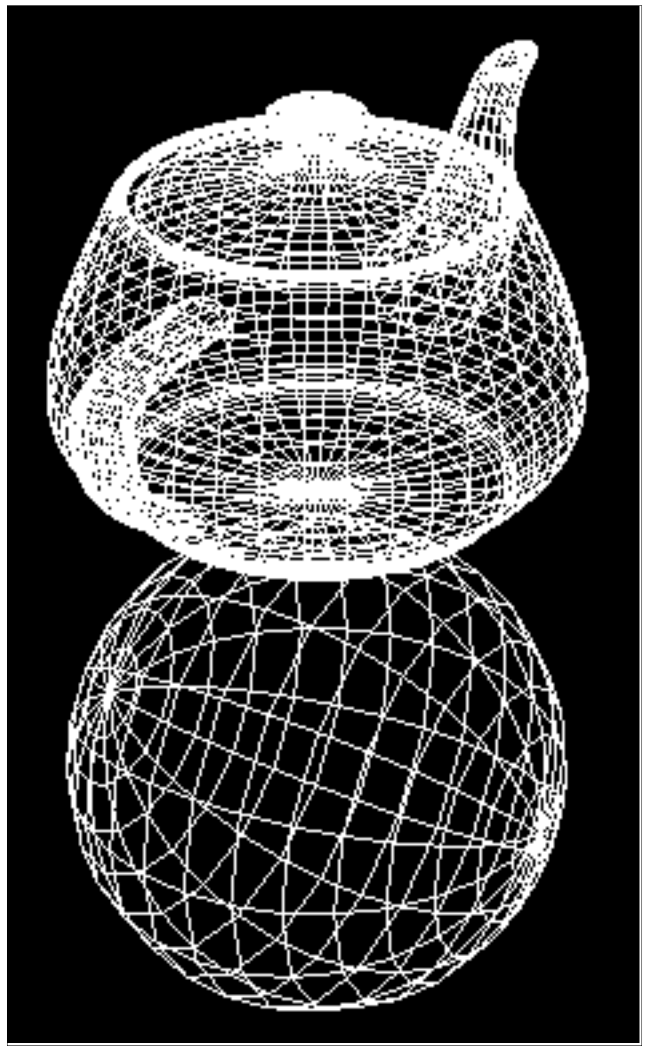
GL_POLYGON_SMOOTH - Old built-in, avoid

Accumulation buffer tricks: Obsolete, avoid

glEnable(GL_MULTISAMPLE); Preferred!

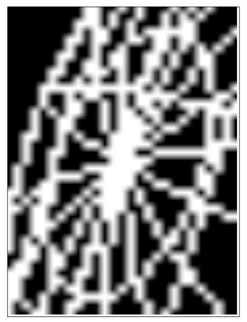
Can also be done by shaders. Usually unnecessary.

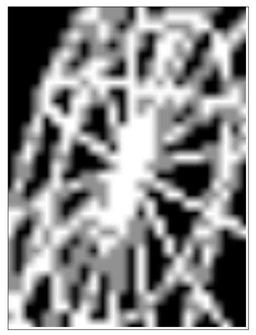
FSAA example

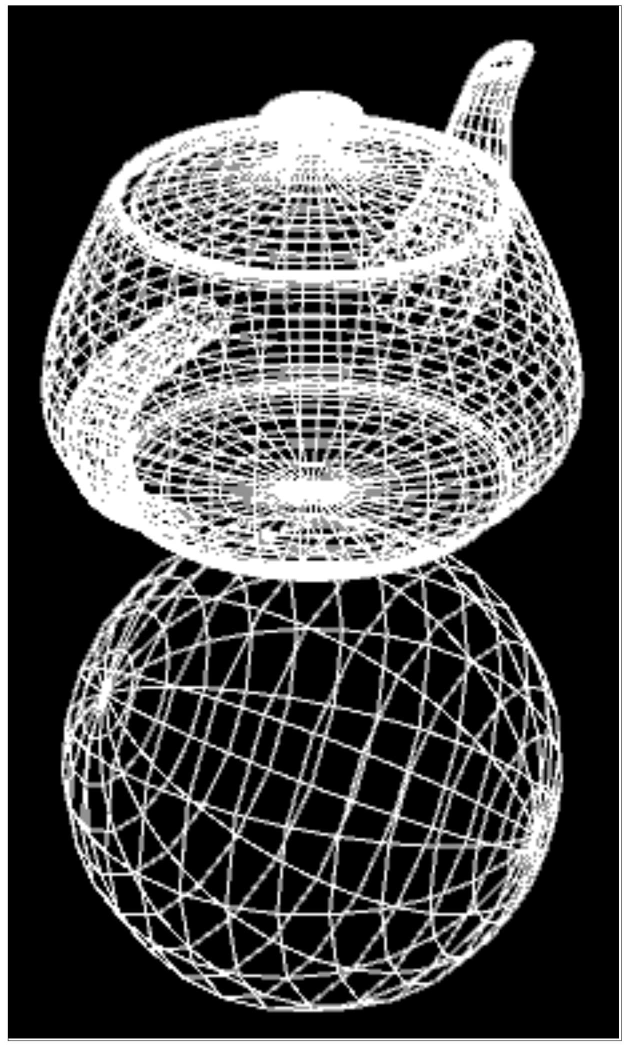


No AA

AA







Anti-aliasing for procedural shapes

New situation!

Big difference: We know where the edge is located!

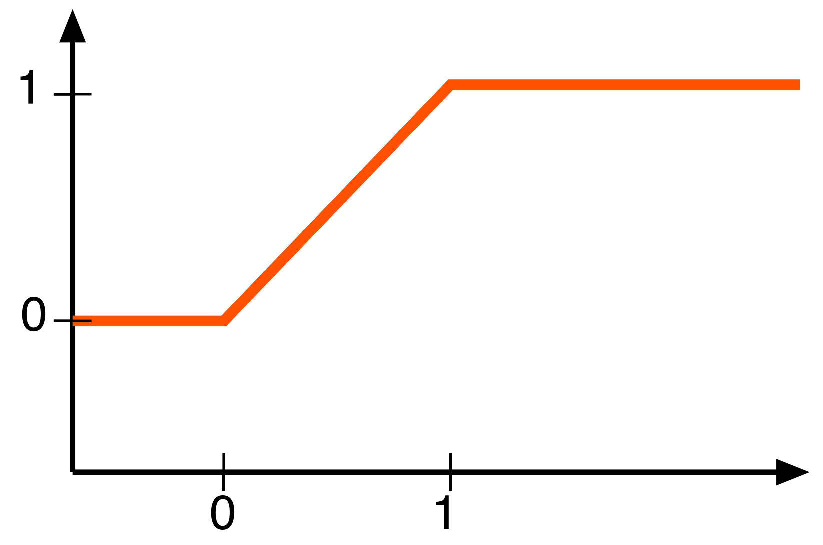
Similar to area sampling

We can make a transition at the edge

Clamp

variant clamping from 0 to 1

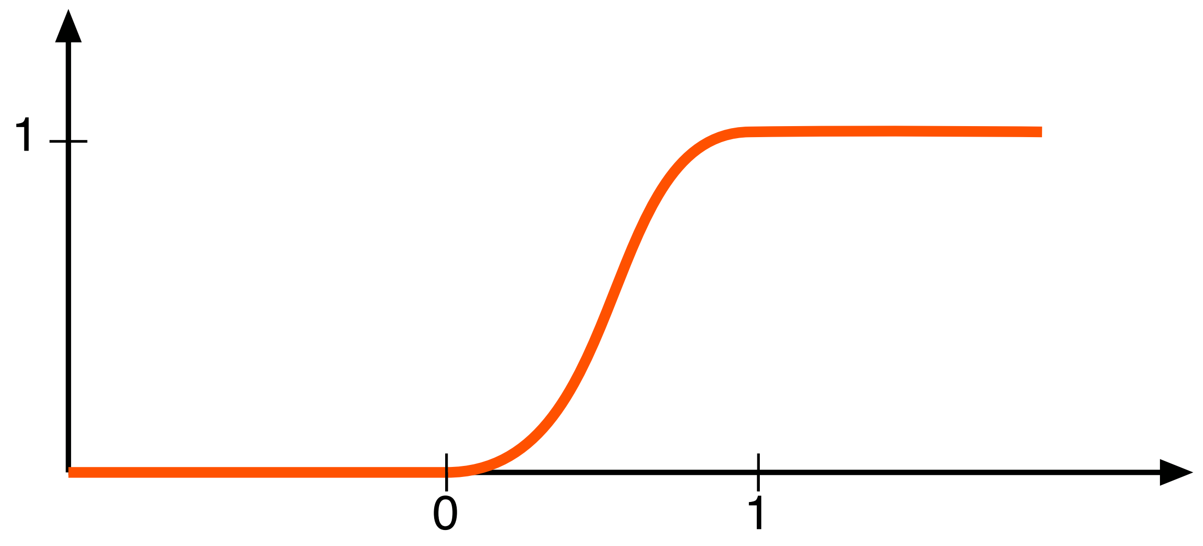
Linear anti-aliasing



Smoothstep

Variant from 0 to 1

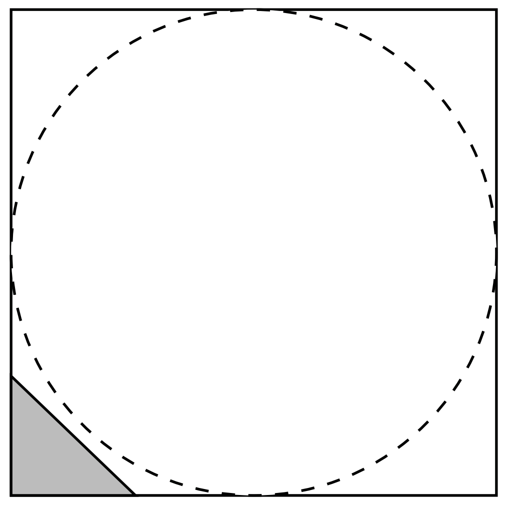
Non-linear anti-aliasing



Direction dependency

Just going from 0 to one on the distance is not perfect.

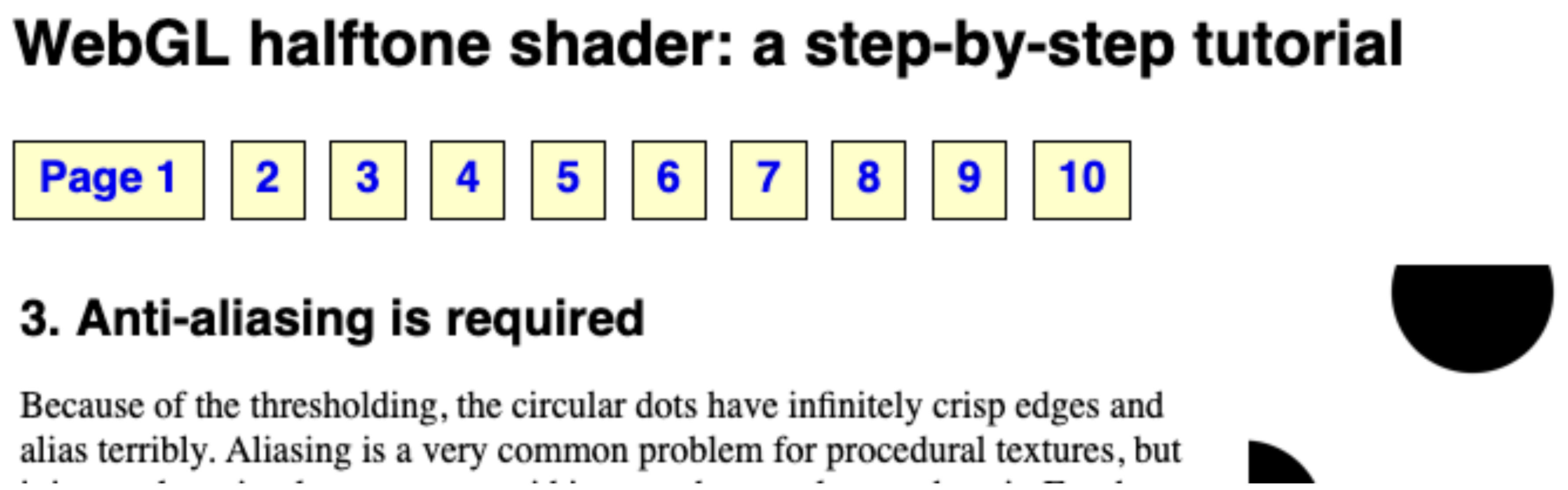
Example:



The edge passes through the pixel on a distance > 1!

More on anti-aliasing in Stefan's halftoning tutorial (esp page 3):

https://weber.itn.liu.se/~stegu/webglshadertutorial/shadertutorial.html



WebGL halftone shader: a step-by-step tutorial

Page 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

3. Anti-aliasing is required

Because of the thresholding, the circular dots have infinitely crisp edges and alias terribly. Aliasing is a very common problem for procedural textures, but

Texture anti-aliasing

Aliasing in textures are reduced by two methods:

Linear filtering

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
```

Mip-mapping

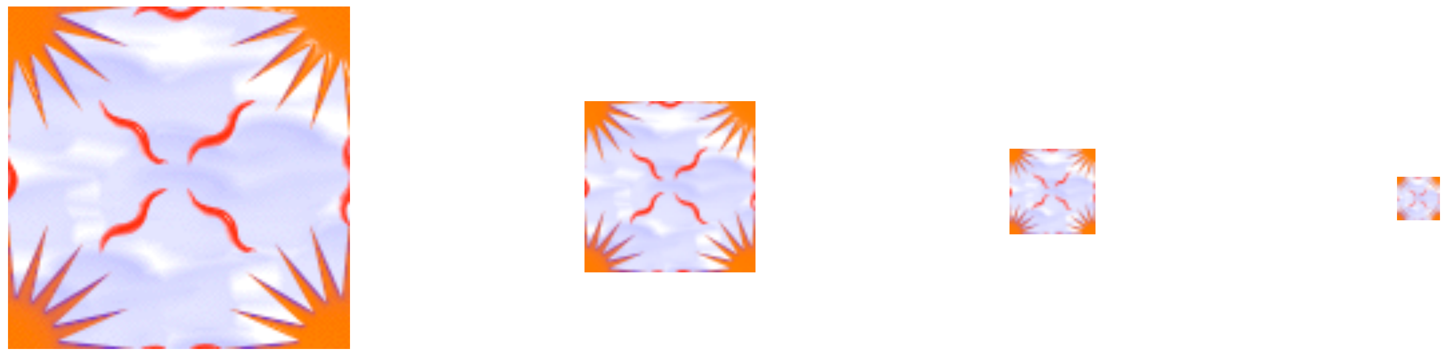
```
glGenerateMipmap();

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
```

MIP mapping

Texture mapping with anti-aliasing.

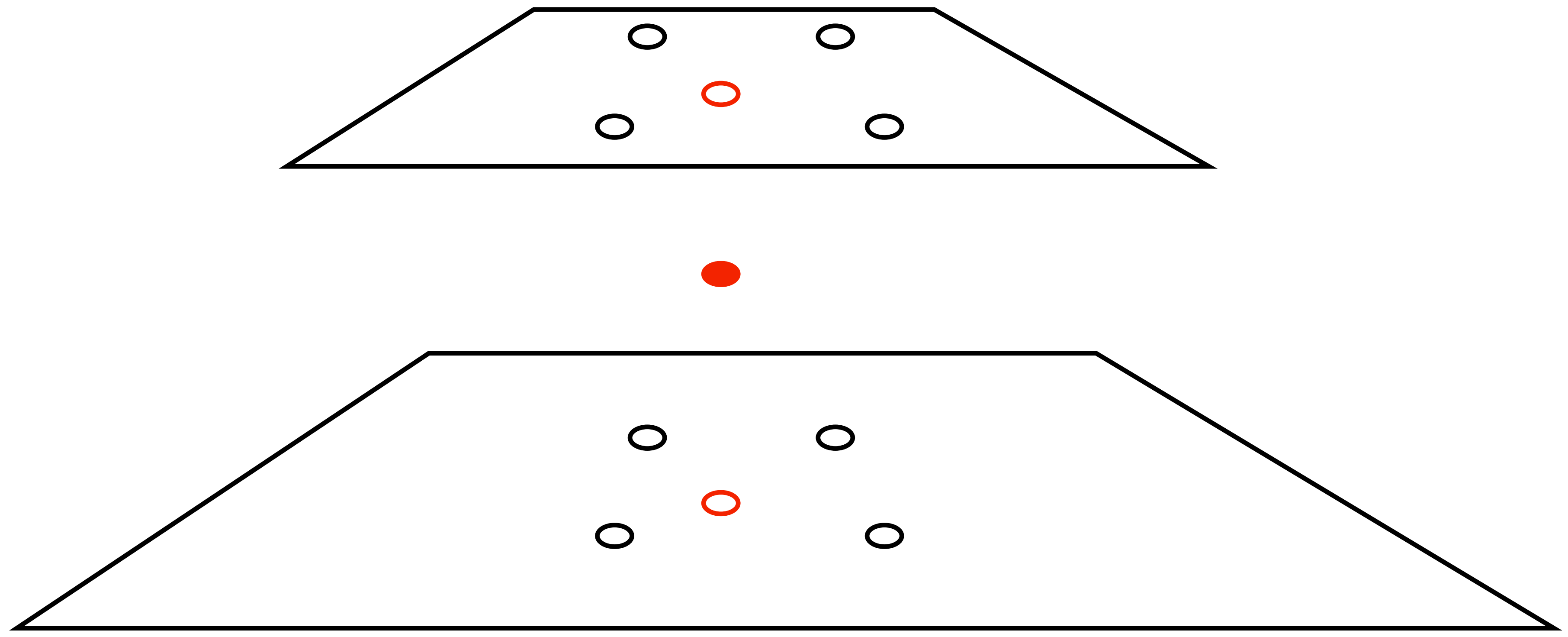
A *resolution pyramid* is built from every texture.

Memory cost: 33% more. Cheap!



MIP mapping filtering

Tri-linear filtering: Both within a level and between!



MIP mapping filtering

GL_NEAREST
GL_LINEAR
GL_NEAREST_MIPMAP_NEAREST
GL_LINEAR_MIPMAP_NEAREST
GL_NEAREST_MIPMAP_LINEAR
GL_LINEAR_MIPMAP_LINEAR

Preferred:
GL_LINEAR for magnification
GL_LINEAR_MIPMAP_LINEAR for minification

MIP mapping

Gives anti-aliasing of textures at a very low cost.

Good results in most situations.

Aliasing problems remain at steep angles.

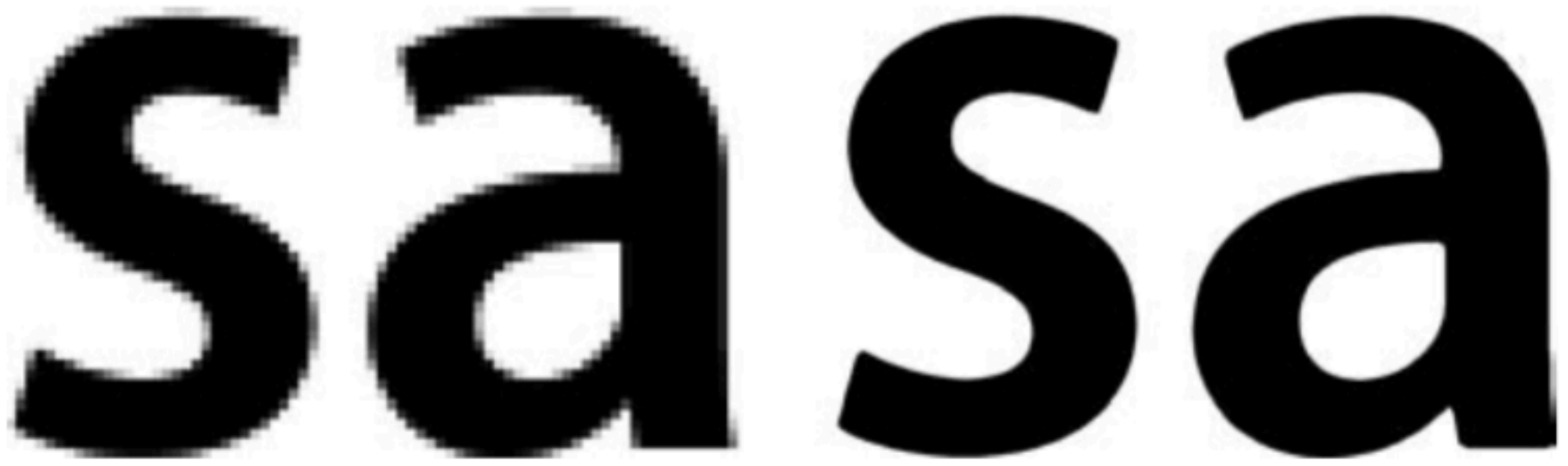
Distance map based anti-aliasing

Stefan's AAEDT has been used for high quality rescaling with minimal aliasing.

Uses an anti-aliased image of low resolution, and reconstructs it to a sharp anti-aliased image in high resolution

Principle: Imagine circles placed along the edge.

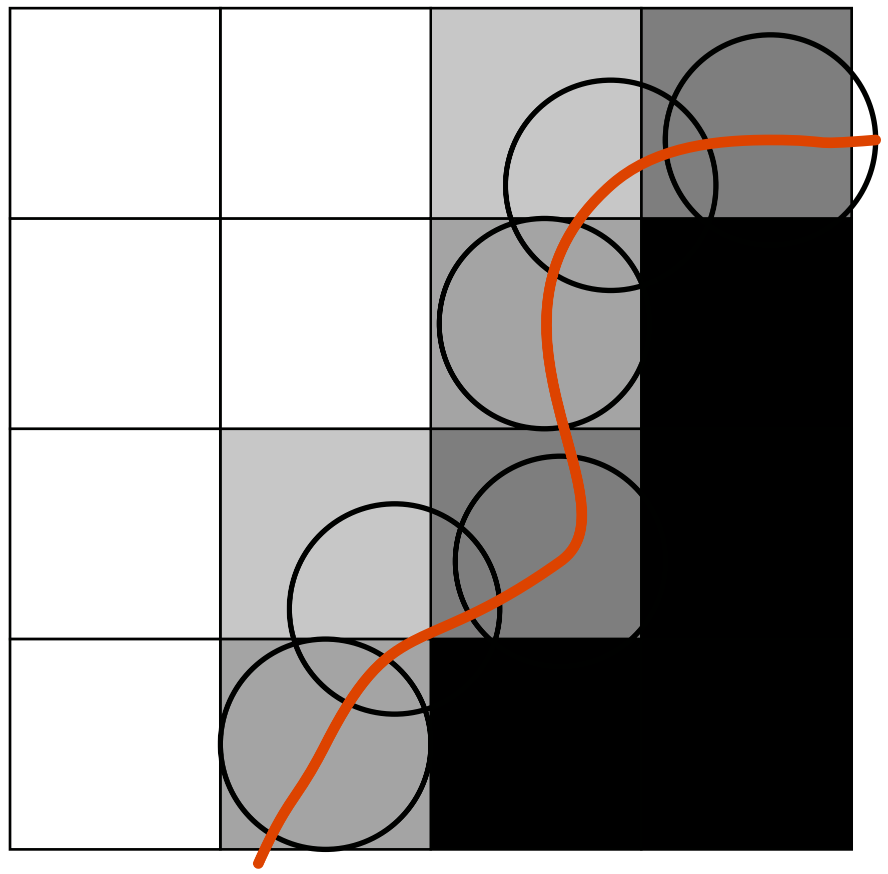
Example from "OpenGL insights"



The right image is created from the left using the AAEDT

Simpler data than a spline-based representation!

Principle: A smooth edge is reconstructed from distance information + smoothstep



Domain specific anti-aliasing

AAEDT: Uses existing anti-aliased data and refines it

Ray-tracing: Uses randomness/noise to hide aliasing

FBM: A later lecture

Summary

Supersampling: Computationally heavy. Excellent results. Popular method in ray-tracing

Multisampling: Simplified supersampling. Only improves edges

Area sampling/Smoothstep edges: Efficient improvement for edges

Mip-mapping: Anti-aliasing for textures