Lecture 23
Even More Shaders

Last Lecture: Shaders

Today: More Shaders?

This Week:

How to use the hardware to do cool stuff

Shader Techniques and Tricks

Thinking About Shaders

The same little program run over different data

Historically, SIMD (single-instruction, multiple data)

Each processor runs in lock-step (same program counter)

```
if (x>.5) {
    // something big
} else {
    // something small
}
```

```
if (x>.5) {
    // something big
} else {
    // something small
}
```

```
if (x>.5) {
    // something big
} else {
    // something small
}
```

```
if (x>.5) {
    // something big
} else {
    // something small
}
```

All pixels execute both paths!

It doesn't work this way anymore - but it motivates how we think about shaders

Use Built-Ins, not Conditionals

Functions for common tests:

- clamp
- step
- sign
- min
- max

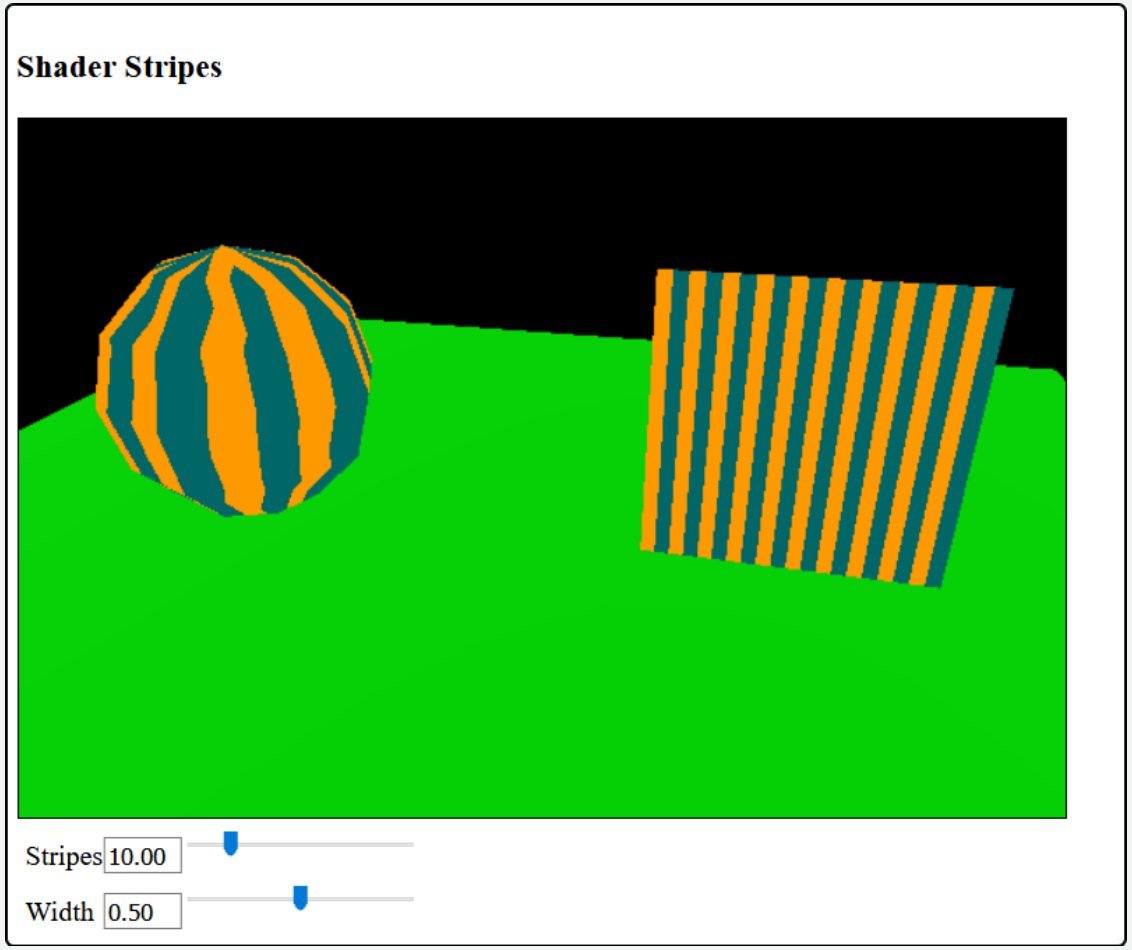
Designed to be more efficient

Convenient

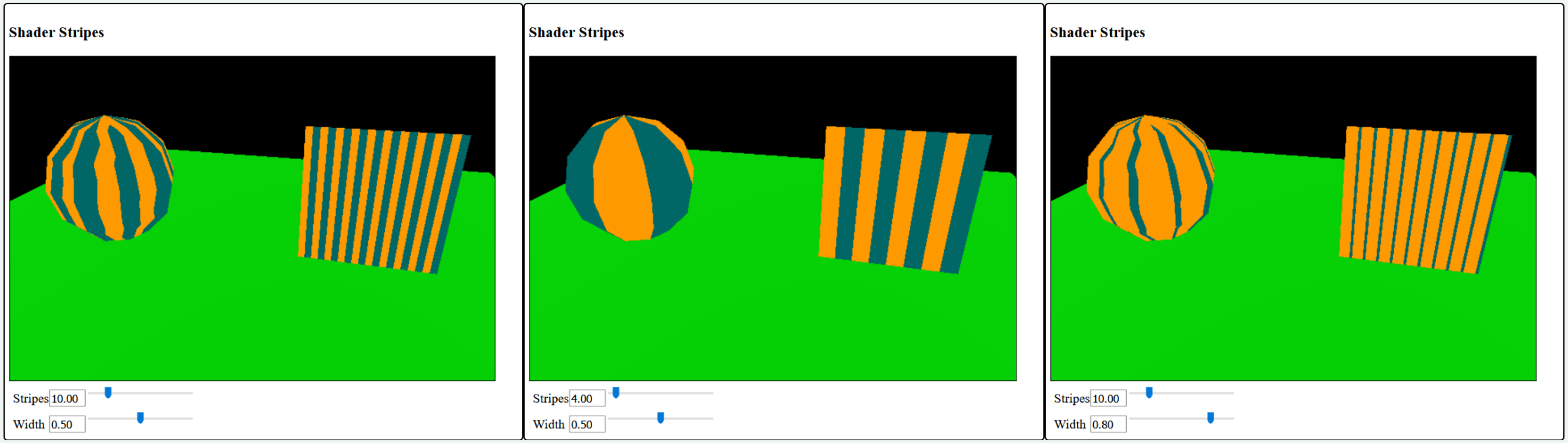
Use Built-Ins, not Expressions

- mix

A Shader... Stripes



Parameters - change as needed



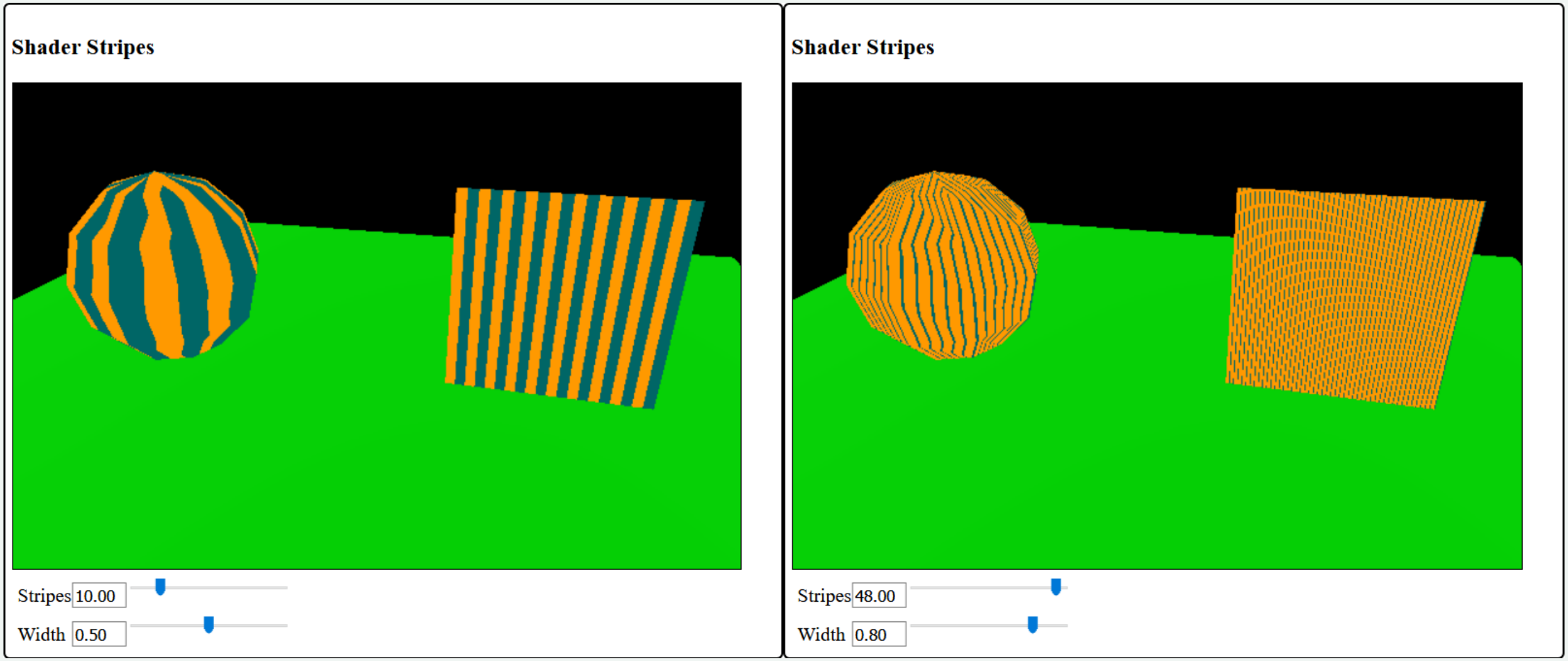
Or change the colors...

```
varying vec2 v_uv;

uniform vec3 color1;
uniform vec3 color2;
uniform float sw;
uniform float stripes;

void main()
{
    // broken into multiple lines to be easier to read
    float su = fract(v_uv.x * stripes);
    float st = step(sw,su);
    vec3 color = mix(color1, color2, st);
    gl_FragColor = vec4(color,1);
}
```

Note the jaggies



Anti-Aliasing in Shaders

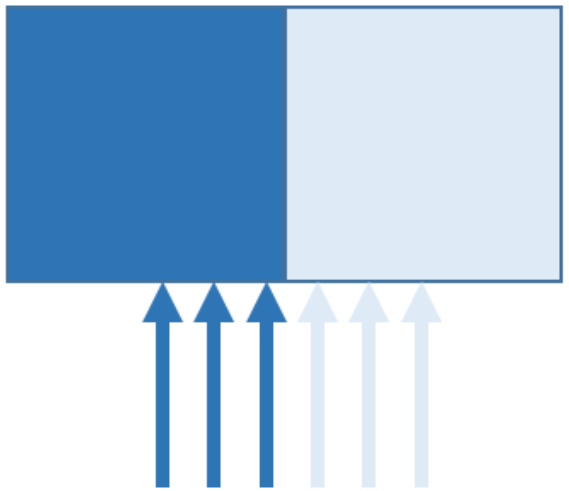
The problem - we are measuring at a specific place for each fragment

Intuition: Sharp Edges are bad

Sharp Edge:

Small change in position

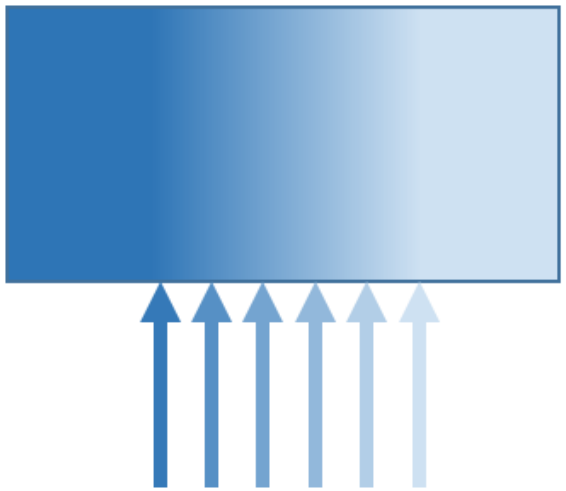
Big change in value



Smoother “Edge:”

Small change in position

Doesn’t matter (that much)



Where do sharp edges come from?

Step vs. SmoothStep

step

```
float st = step(sw,su);
```

smoothstep

```
uniform float blur;
float st = smoothstep(sw-blur,sw+blur,su);
```

Warning - this is one sided!

Step is only for the 0-1 transition

The 1-0 transition happens at the repeat

Need to deal with it separately

Handling 2 sides

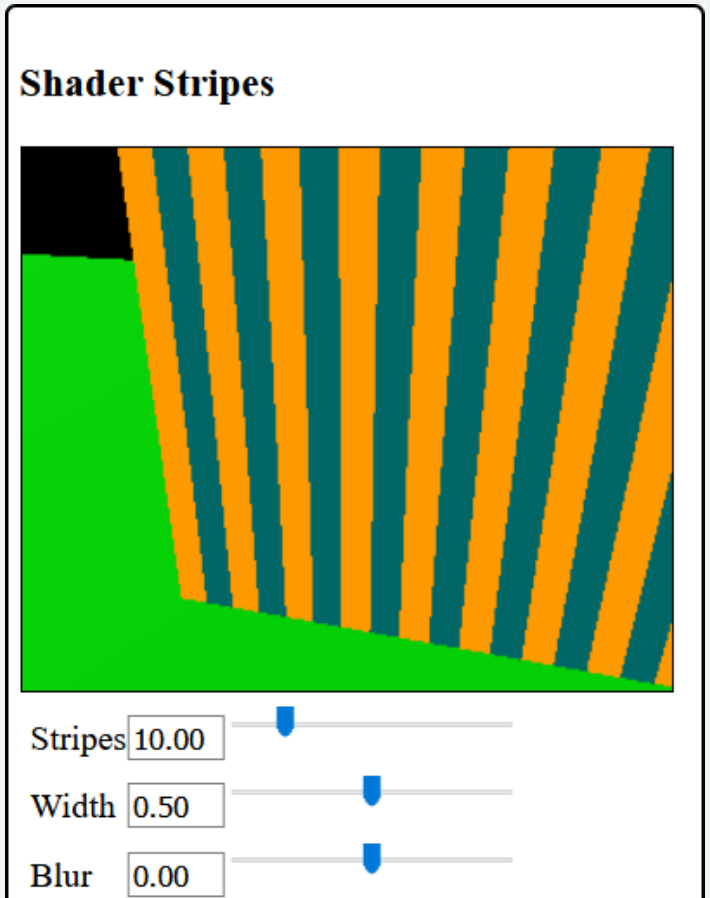
This is a little tricky, since the "edge" is in two places

Easy way: put the string in the center of the "tile"

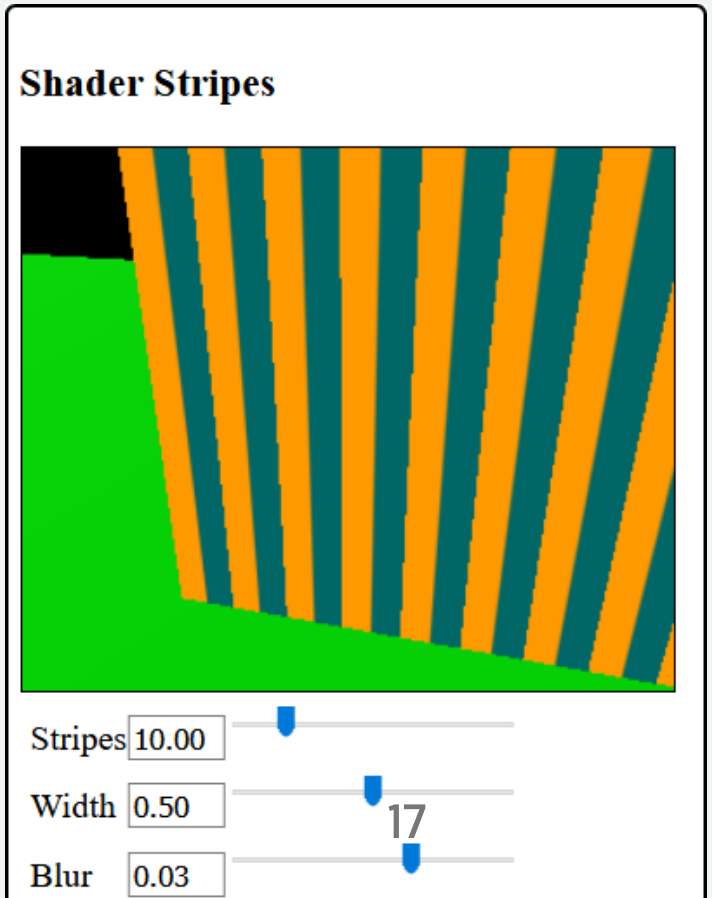
```
float st = smoothstep((sw/2)-blur,(sw/2)+blur,abs(su-.5));
```

Does this help?

Width 0 (no blur)

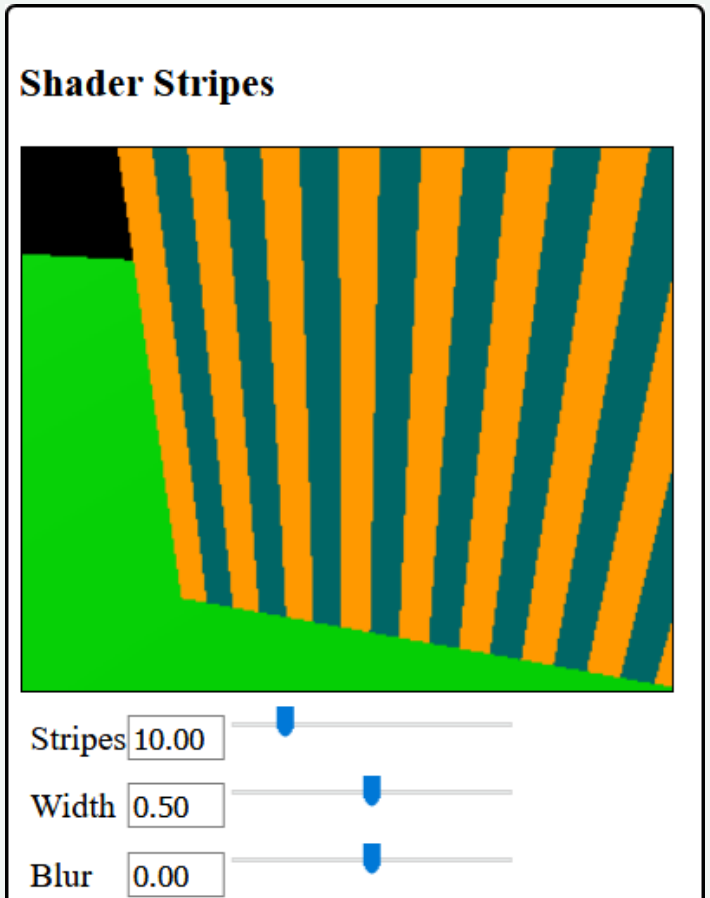


Width 3 (blur)

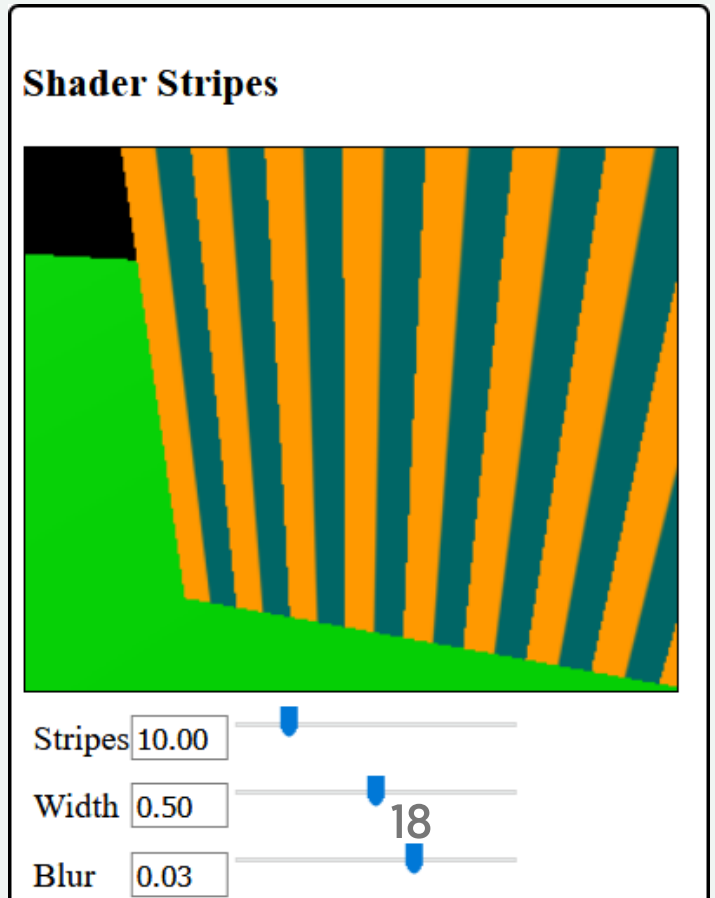


Too much of a good thing?

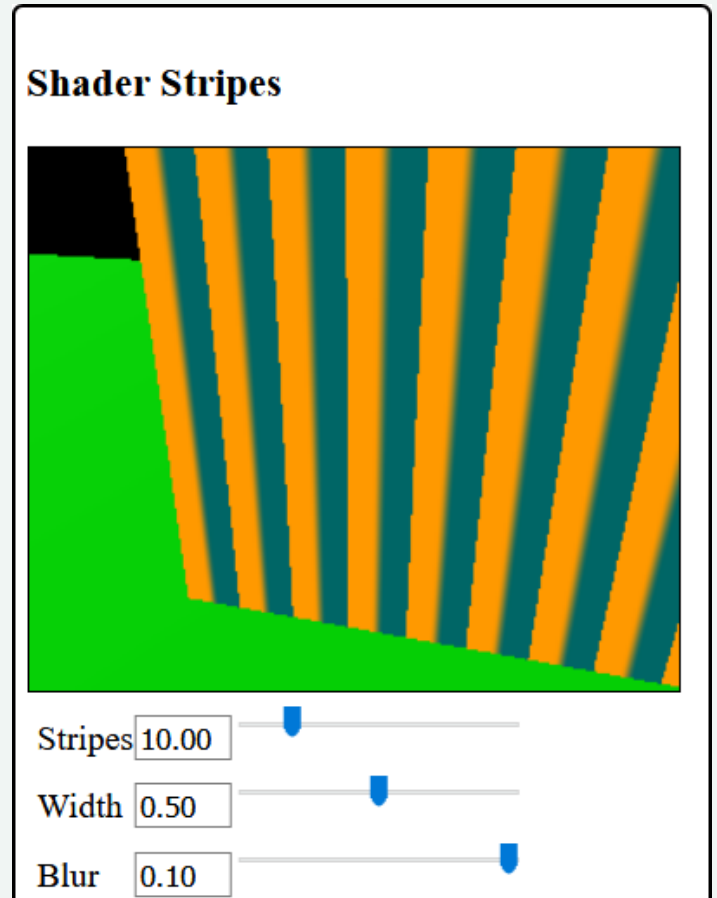
Width 0 (no blur)



Width 3 (blur)



Width 10 (blurry)



What value for the width?

Want the blur to be "one pixel wide"

But what is that in u values?

GLSL will figure it out for us!

- `dFdx` - derivative of function with respect to x
- `fwidth` - derivative of function with respect to x and y

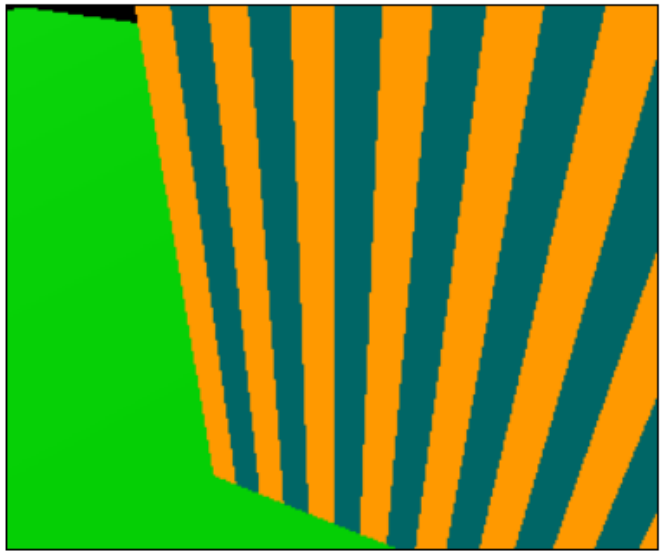
These are **extensions** to GLSL-ES (but three loads them for us)

- we need to enable them in the shader

fwidth: just right!

No Blur

Shader Stripes



Stripes 10.00

Width 0.50

Blur 0.00

FWidth Blur

Shader Stripes

Stripes 10.00

Width 0.50

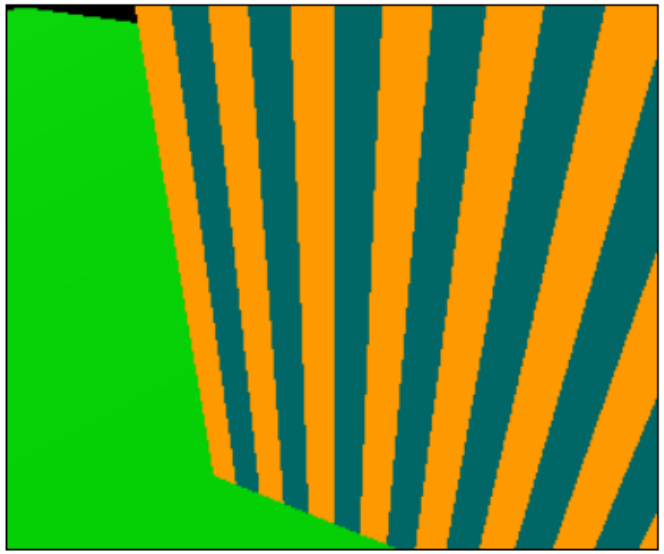
20

Blur -0.10

fwidth: just right!

No Blur

Shader Stripes



Stripes 10.00

Width 0.50

Blur 0.00

FWidth Blur

Shader Stripes

Stripes 10.00

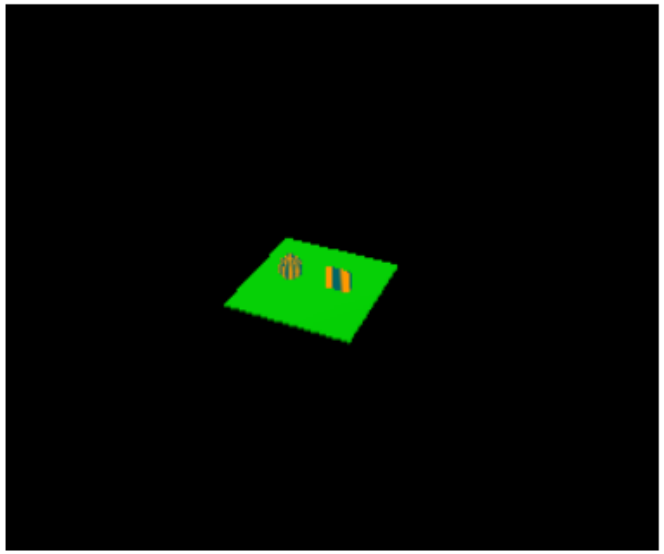
Width 0.50

21

Blur -0.10

Zoom out?

Shader Stripes



Stripes 10.00

Width 0.50

Blur -0.10

Smoothstep handles texture Magnification

Not texture Minification

We only compute 1 color (texture coordinate) per fragment

- even if this fragment covers a large area!

We still need some way to do filtering

Hard to do in a general way for procedures

- over-sampling

Much easier to do for images

The Model In Review

Each vertex is independent

Each fragment is independent

We compute each thing separately

- it's OK, parallel so it's fast!

What about randomness?

Regular patterns look too boring

We don't want the patterns to be obvious...

We can't really use randomness

- Each frame independent
  - random would cause it to change each time
- Each fragment independent
  - no way to have structure between fragments

Noise

1. Psuedo-random
    - pattern too complex to see
    - but still is controlled / deterministic
2. Structured
    - control properties that we care about

Noise - A Simple Method

Start with a 1D pattern (since it's simpler than 2D)

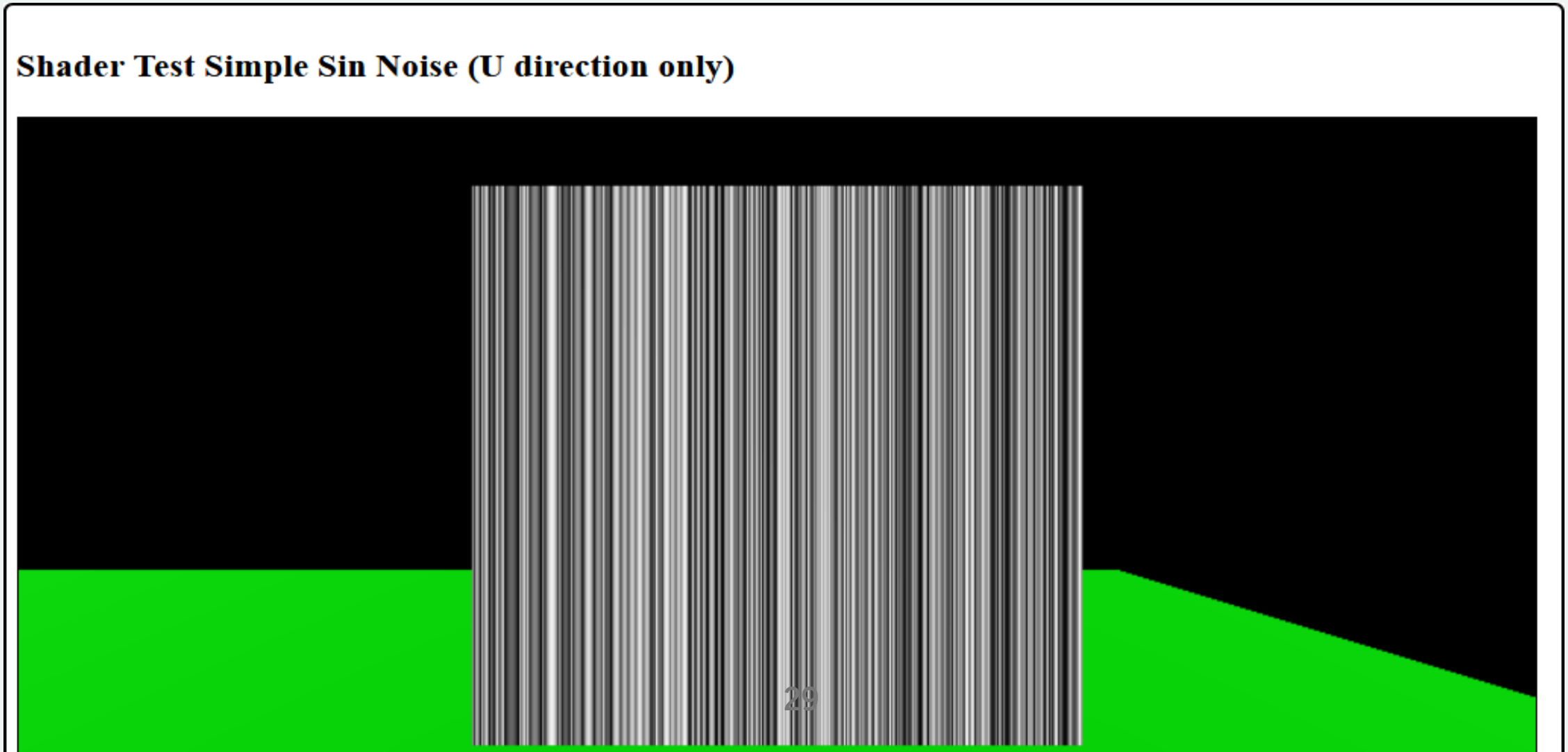
color = f(u) (like stripes)

Simple Psuedo-Random

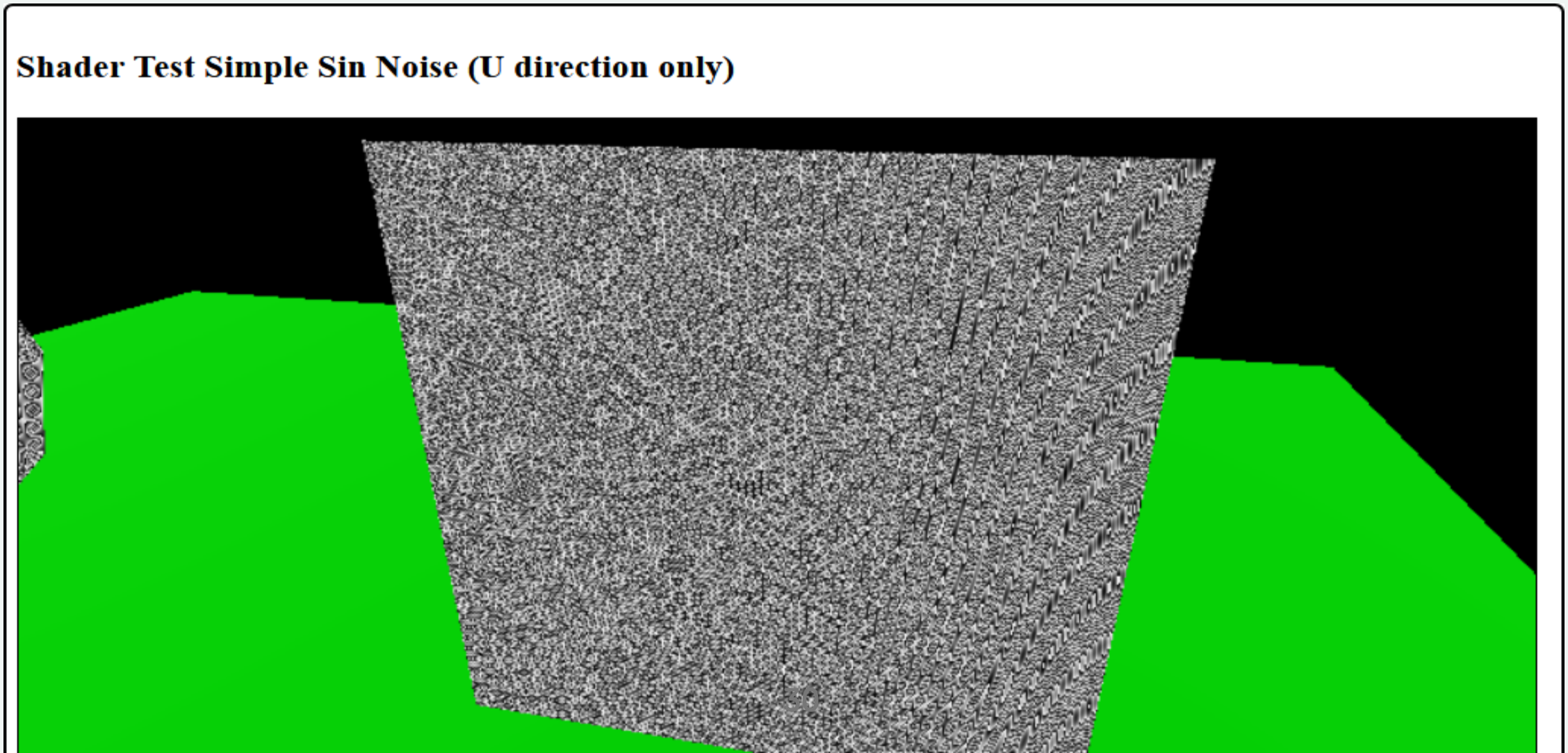
(many better choices)

```
r = fract( sin(u) * 100000. )
```

What does this look like?



A problem



What's happening?

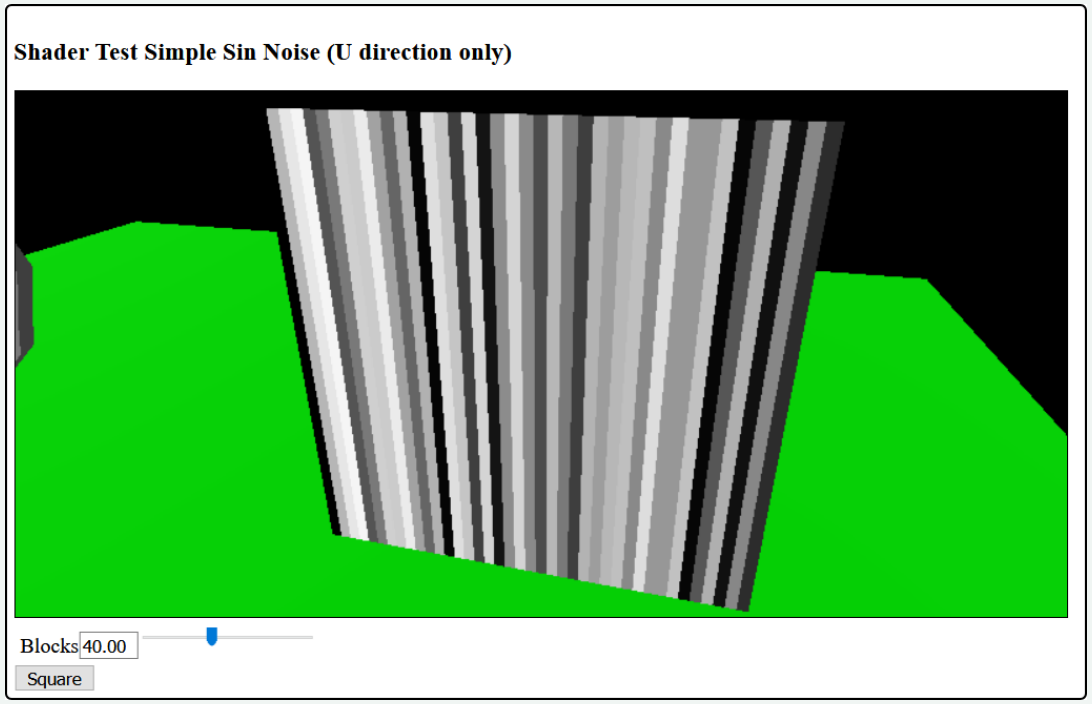
Structure: change slowly

Sample (points along line)

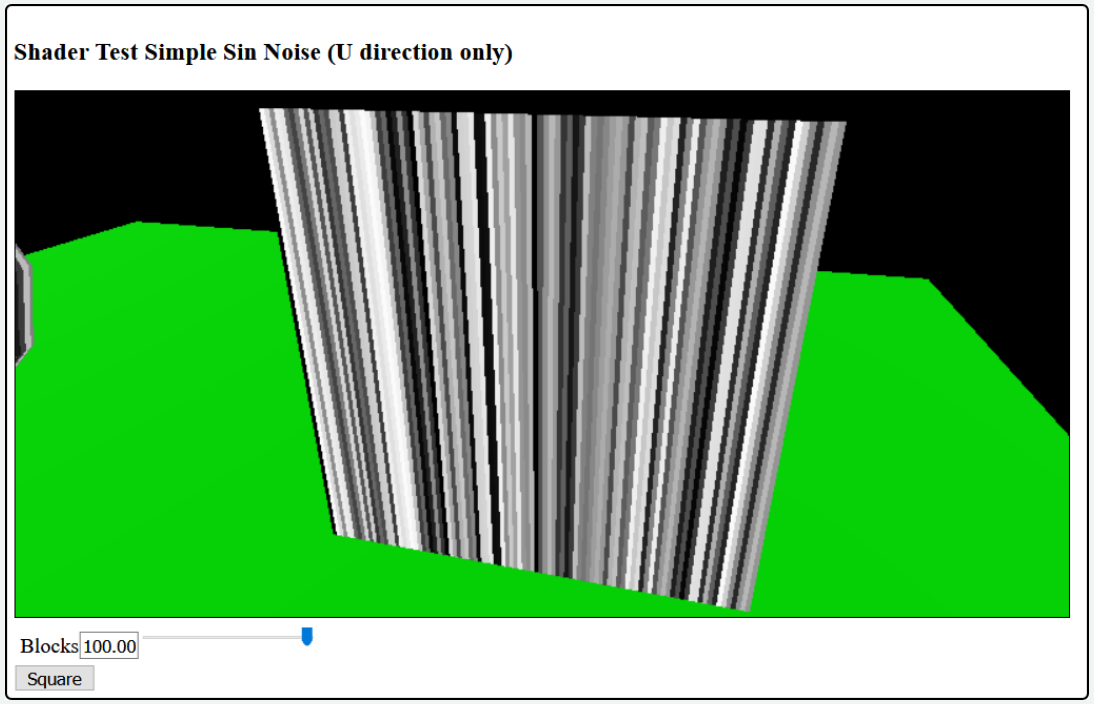
Aliasing - but adds to randomness

Using Simple Randomness

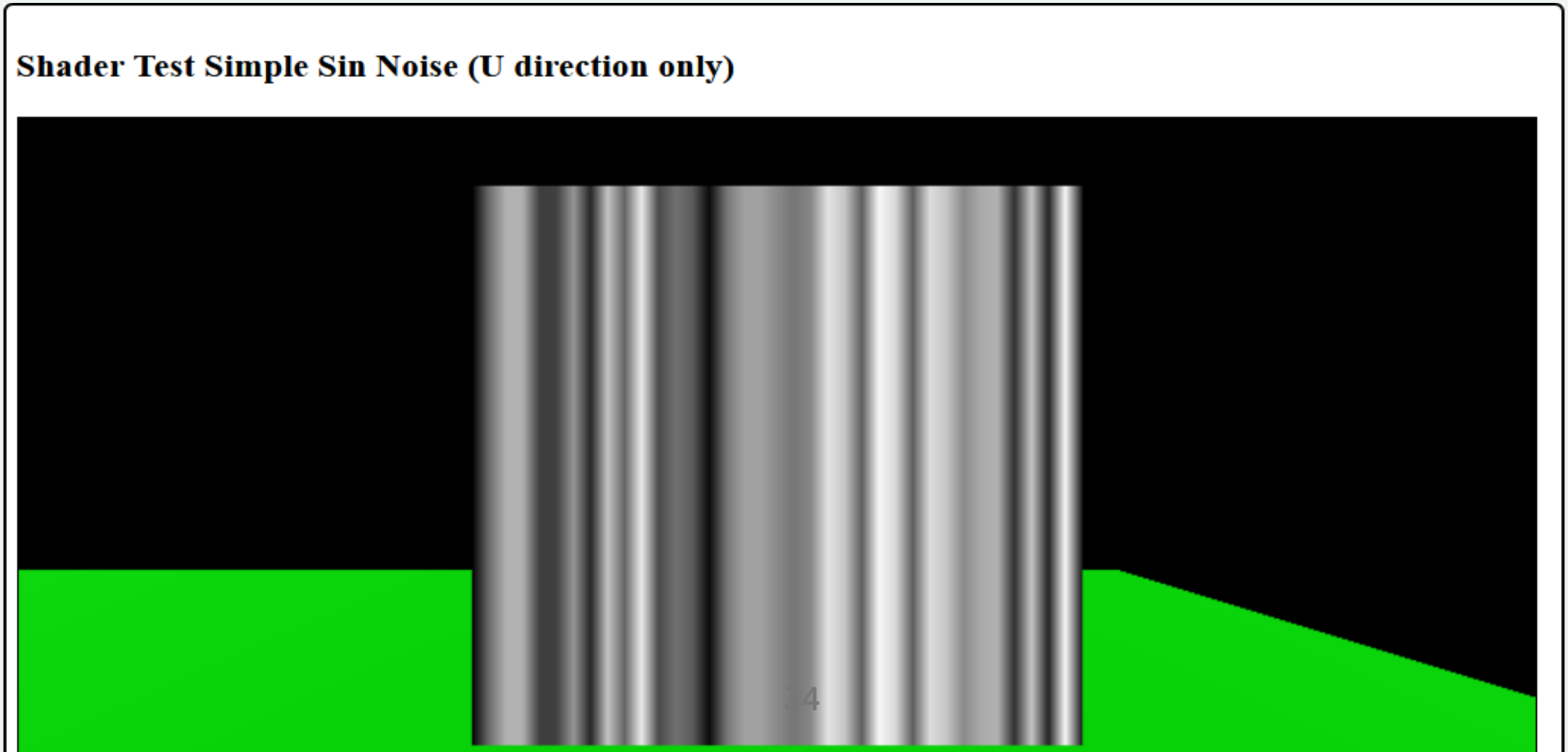
40 points



100 points



Interpolate to make smooth



You can do a lot better...

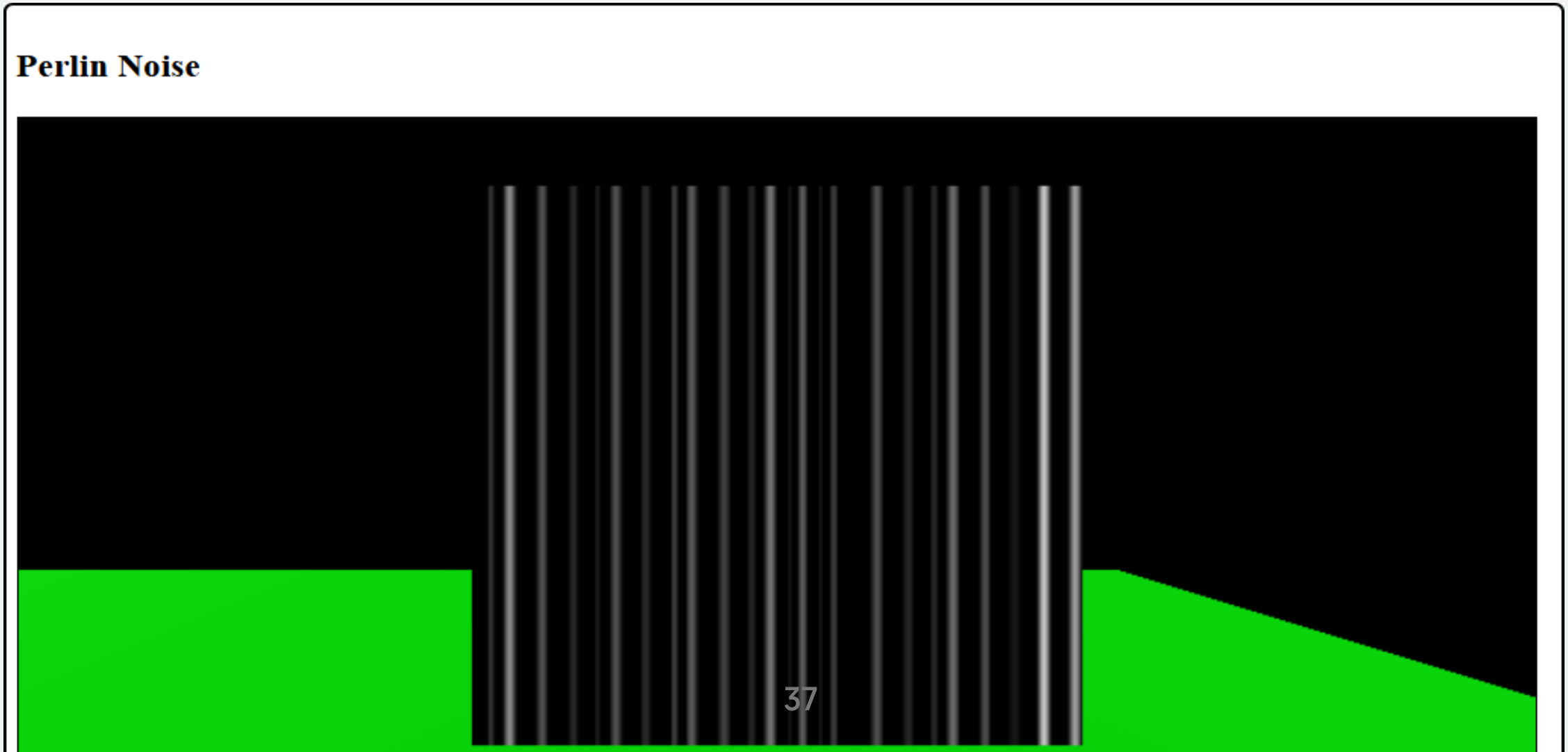
- Better Psuedo-random functions
- Efficient in multi-dimensions (2D, 3D)
- Tileable
- Better interpolation
- Multiple frequencies

Perlin Noise

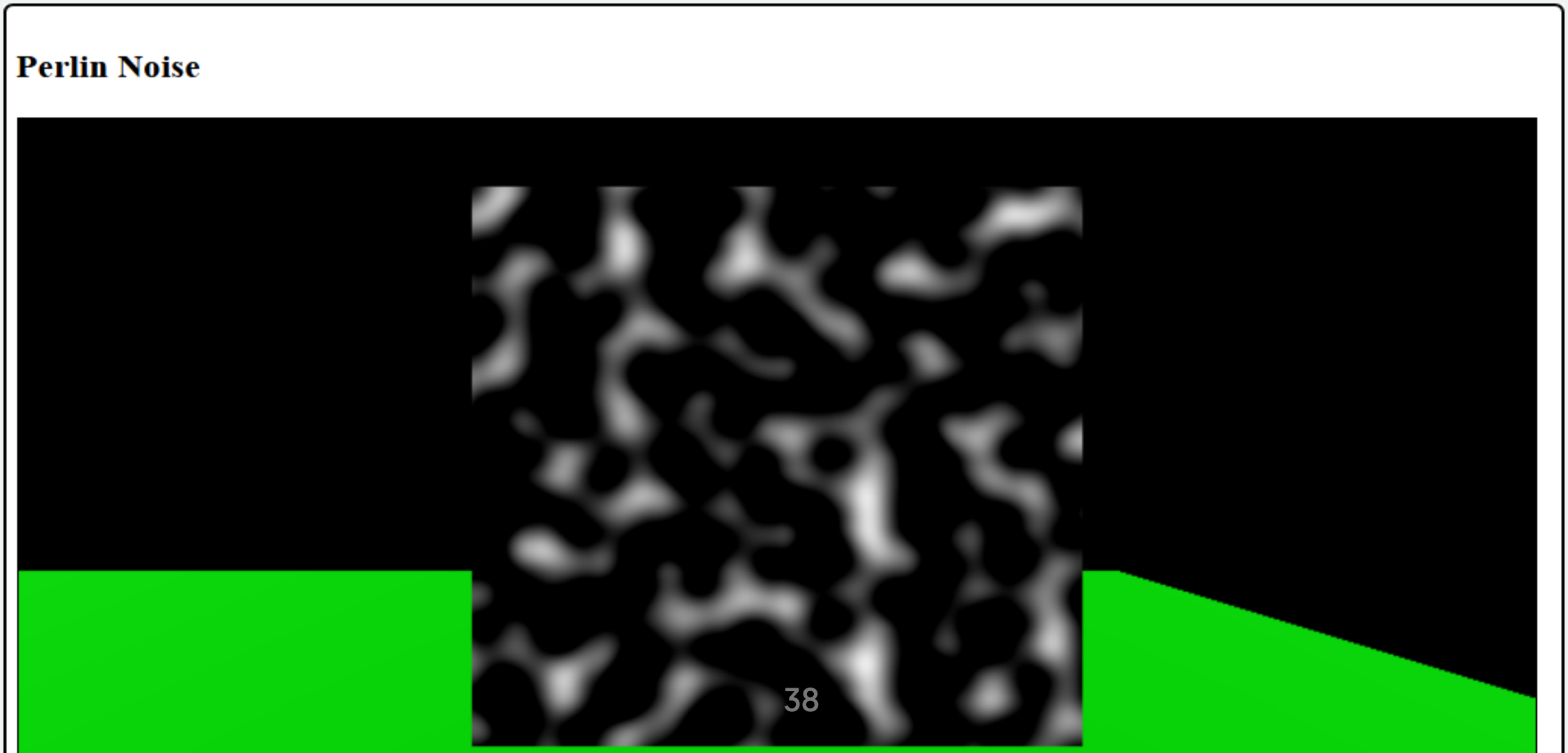
The classic noise function

- newer variants are more efficient on GPUs
- even better psuedo-randomness
    - noise - controlled psuedo-randomness
        - Perlin noise
            - coherence at different frequencies
            - demo (1D)
            - demo (2D)

Perlin Noise in 1D

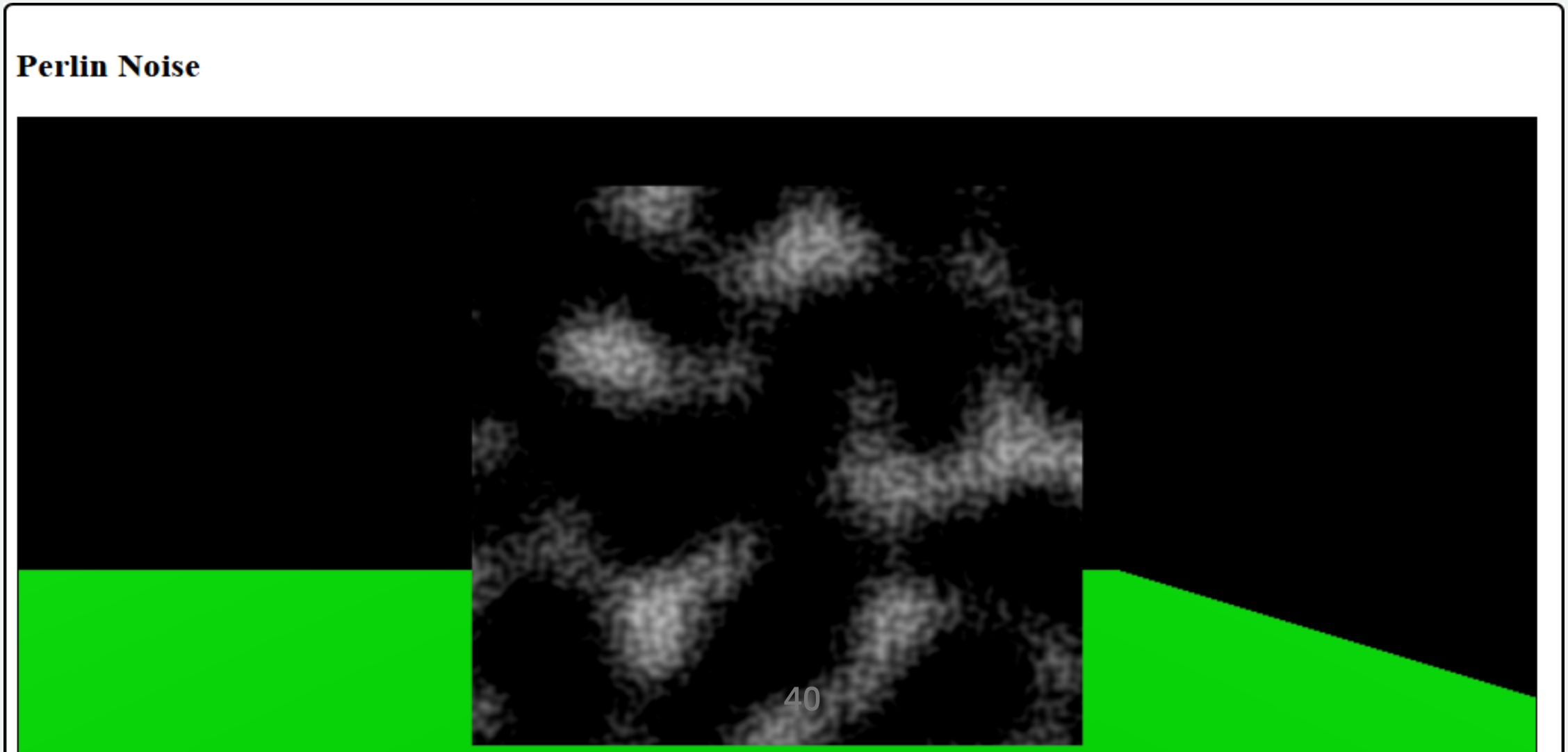


Perlin Noise in 2D



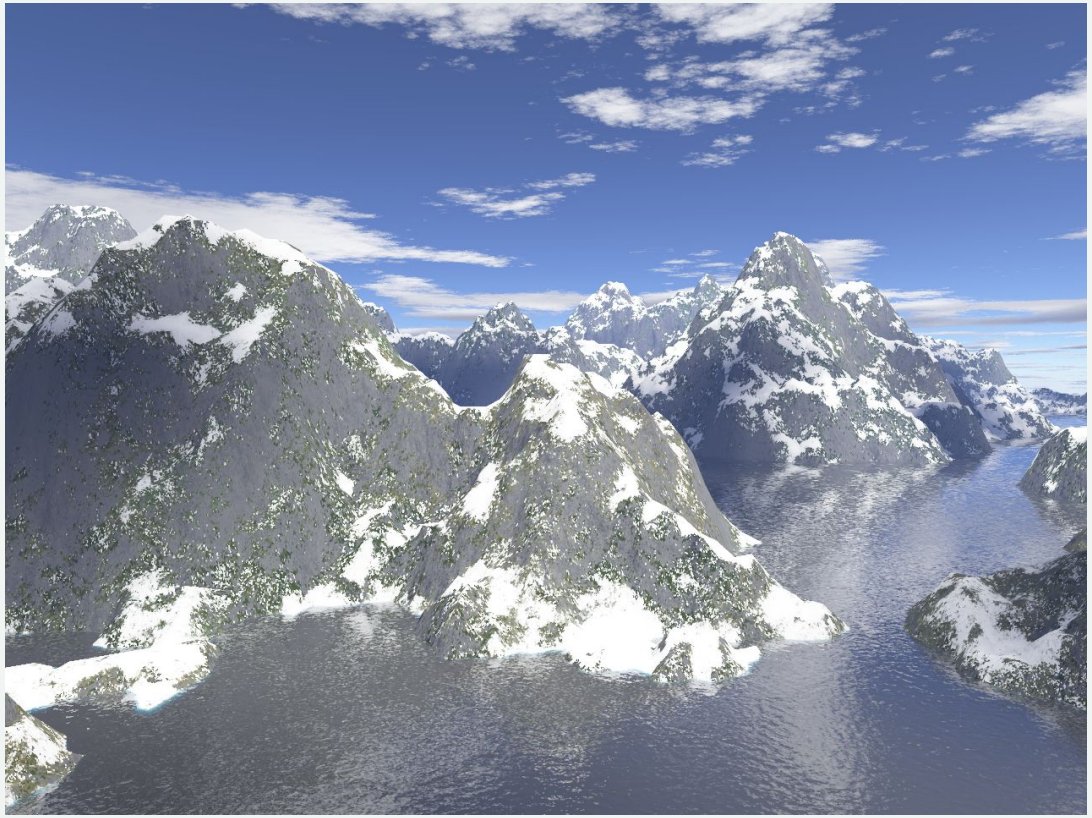
Perlin Noise in 2D - High Frequency

Multi-Frequency: Low + High

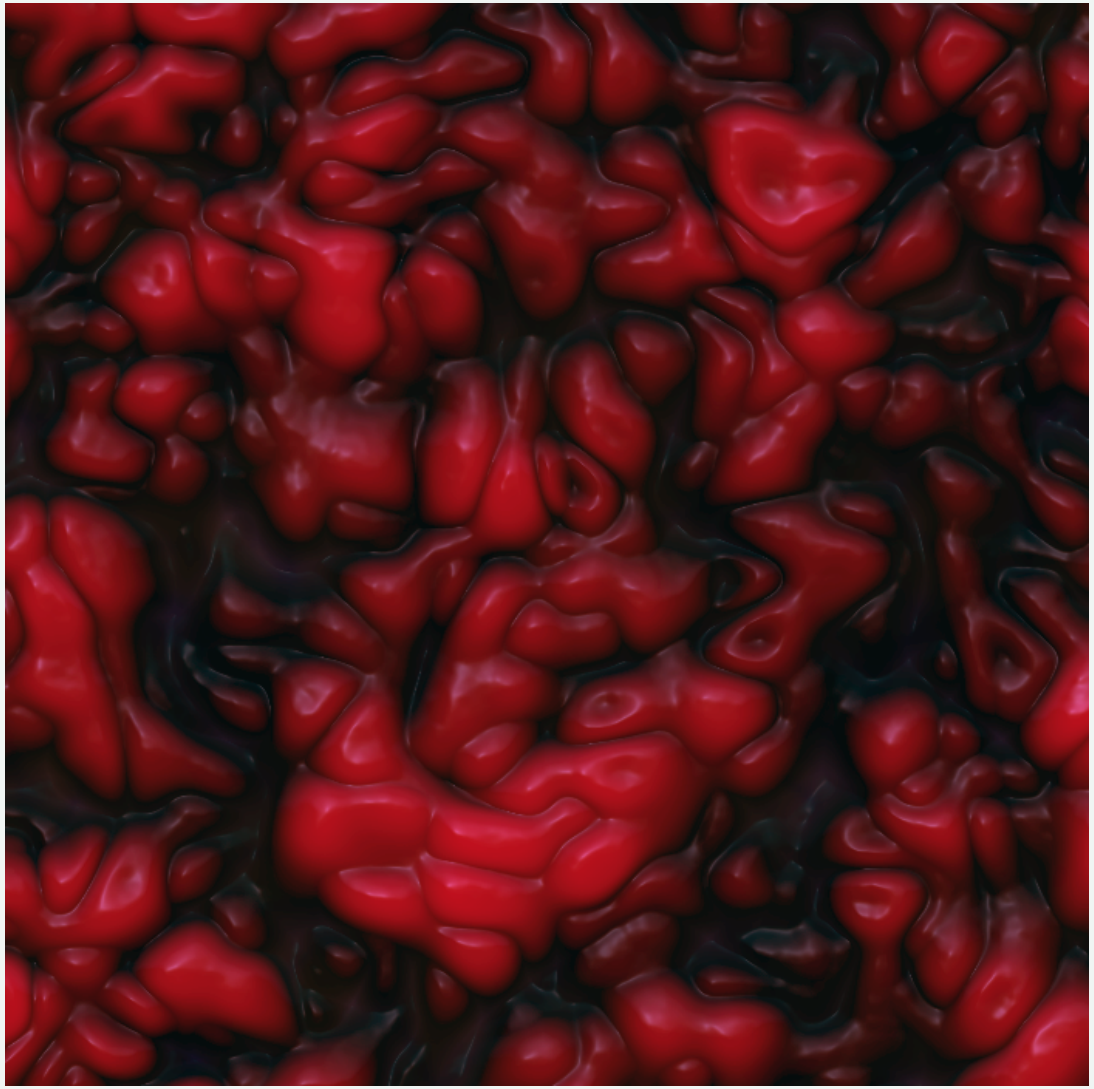


How do you use this?

- Find an implementation on the web
- Mix different frequencies to get desired effects
- Add noise to make things less "perfect"
- It's an art



By Stevo-88 - self-made, used Adobe Photoshop for Perlin noise creation and Terragen for rendering., Public Domain, https://commons.wikimedia.org/w/index.php?curid=2208011



By Simon Strandgaard from Kastrup, Danmark - pink/red liquid using perlin noise + bump + coloring, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=76348609

Changing Shape

Can we change a sphere into a Cornavirus?

Summary: Shaders

1. Understand the model
2. Understand the shading language (GLSL)
3. Understand the mechanics (connect GLSL to JavaScript/THREE)
4. Use shading for Lighting
5. Use shading with Image-Based Textures
6. Use common shading idioms
7. Use Anti-Aliasing
8. Use Noise to create randomness

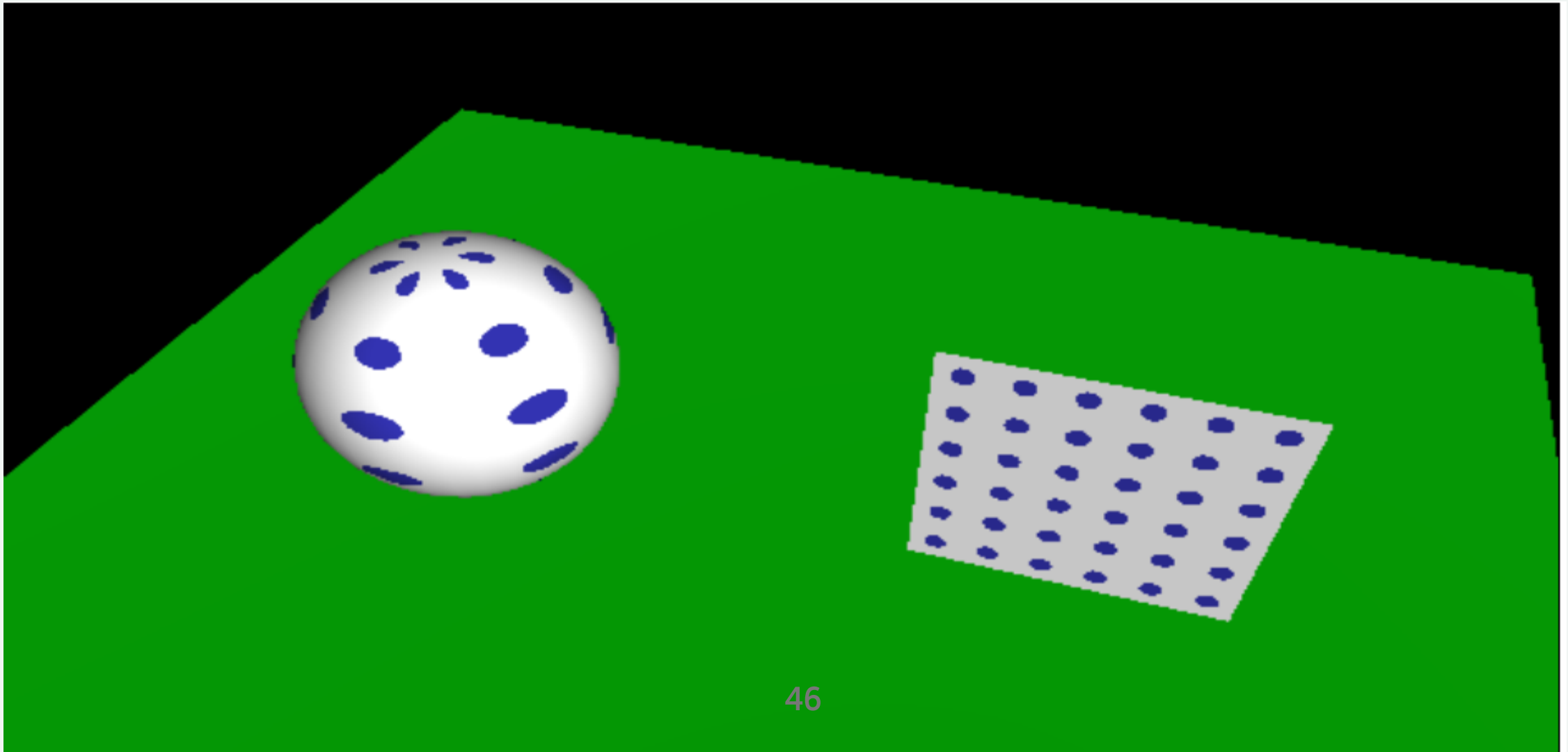
Fake Shape

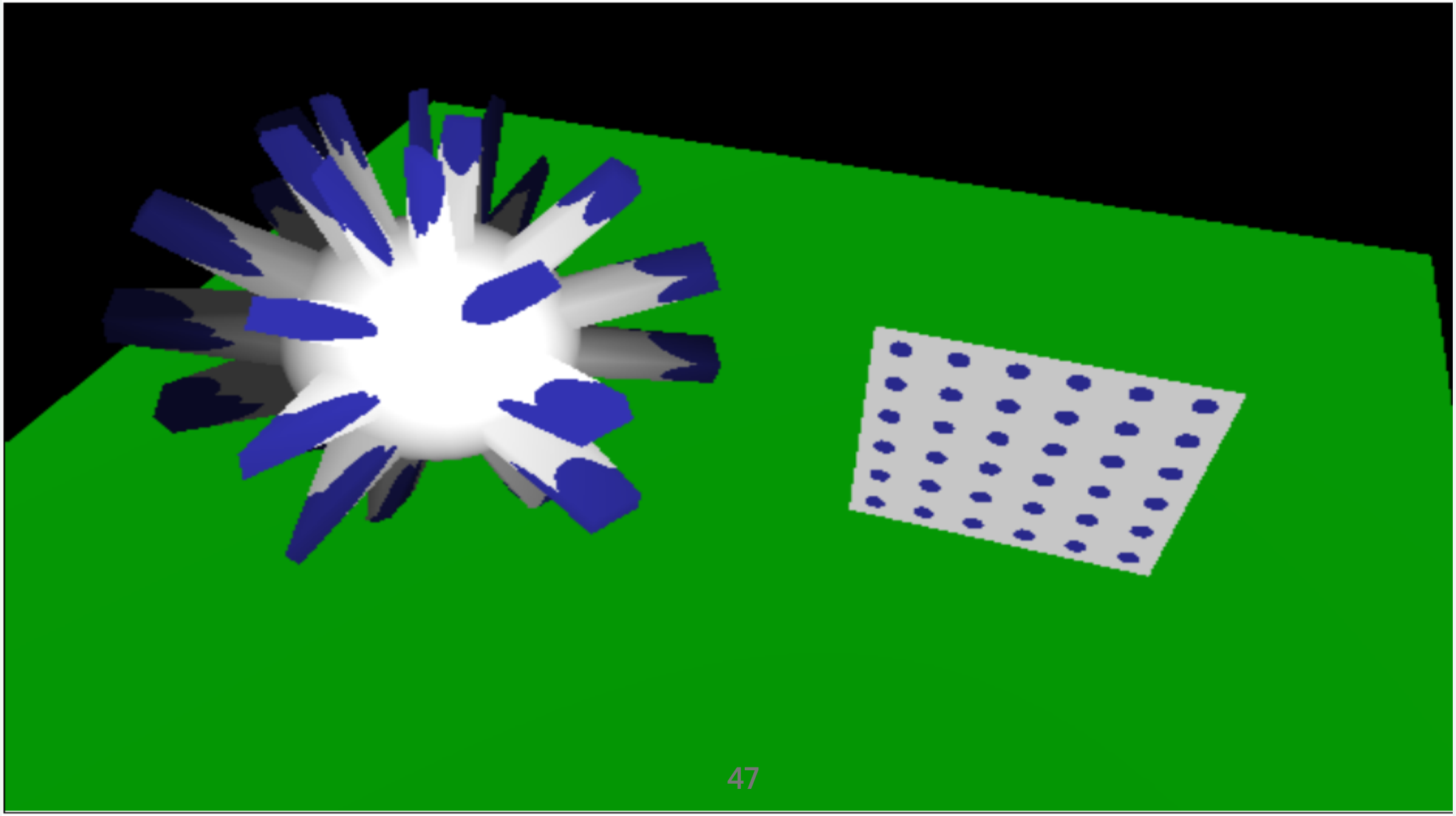
Using a **vertex shader** we can change geometry

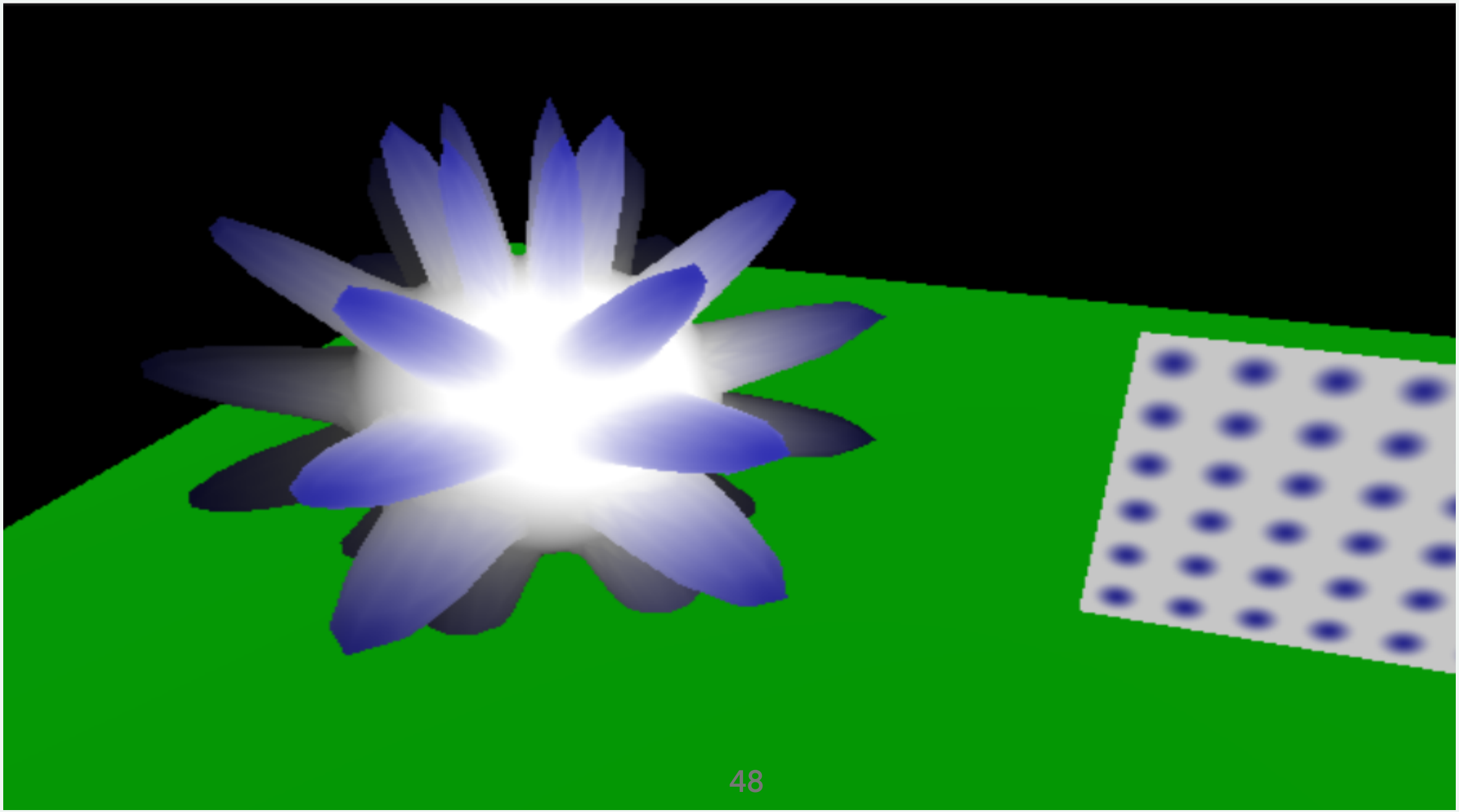
Changing geometry with a texture is called **displacement mapping**

The texture is used to **displace** points

Coronavirus?







How did we do that?

Use dots to move things in the normal direction

Notice:

- only move vertices (plane doesn't change)
- didn't change lighting
  - possible, but tricky
- spacing depends on mapping
- some triangles get really stretched
- we really changed the shape

```
uniform float disp;
varying vec2 v_uv;
varying vec3 l_normal;
varying vec3 v_world_position;

void main() {
    v_uv = uv;

    float d = fdot(uv);

    vec4 world_pos = (modelMatrix * vec4(position + disp * d * normal,1.0));
    v_world_position = world_pos.xyz;

    // the main output of the shader (the vertex position)
    gl_Position = projectionMatrix * viewMatrix * world_pos;

    // compute the normal and pass it to fragment
    // note - this is in world space, but uses a hack that
    // assumes the model matrix is its own adjoint
    // (which is true, sometimes)
    l_normal = (modelMatrix * vec4(normal,0)).xyz;
}
```

Displacement Maps

- could have done this with an image texture

Why not?

- depends on meshing (needs small triangles)
- may want to make more triangles
- tiny triangles cause problems
- messes up the mesh
- need to get things aligned with vertex samples

Fake Lighting

Can we give the appearance of something more complex?

Can we make fancier lighting effects?

How do we get things to not appear flat?

1. Make lots of triangles
2. Fake it with texture

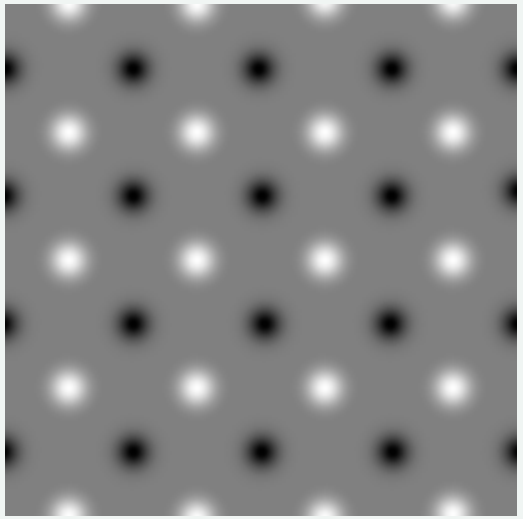
The Real Normal to a Triangle
The Fake Normal to a Triangle

Review (this was part of Advanced Texturing)

Per Pixel/Fragment Normals

- Normal Maps (look up XYZ in the map)
- Bump Maps (look up height in the map)

Bump Map

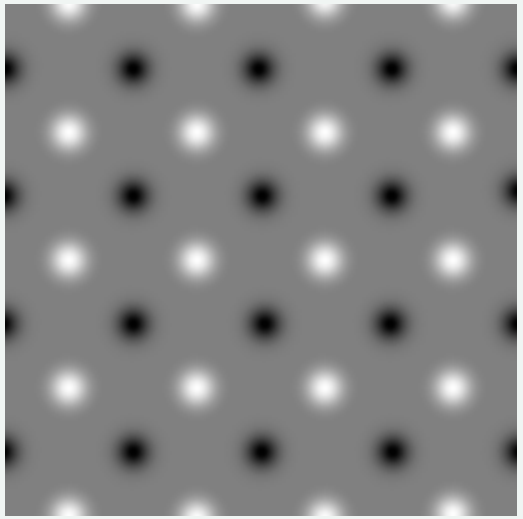


Gray = middle, black=down,

white=up

(it's all relative)

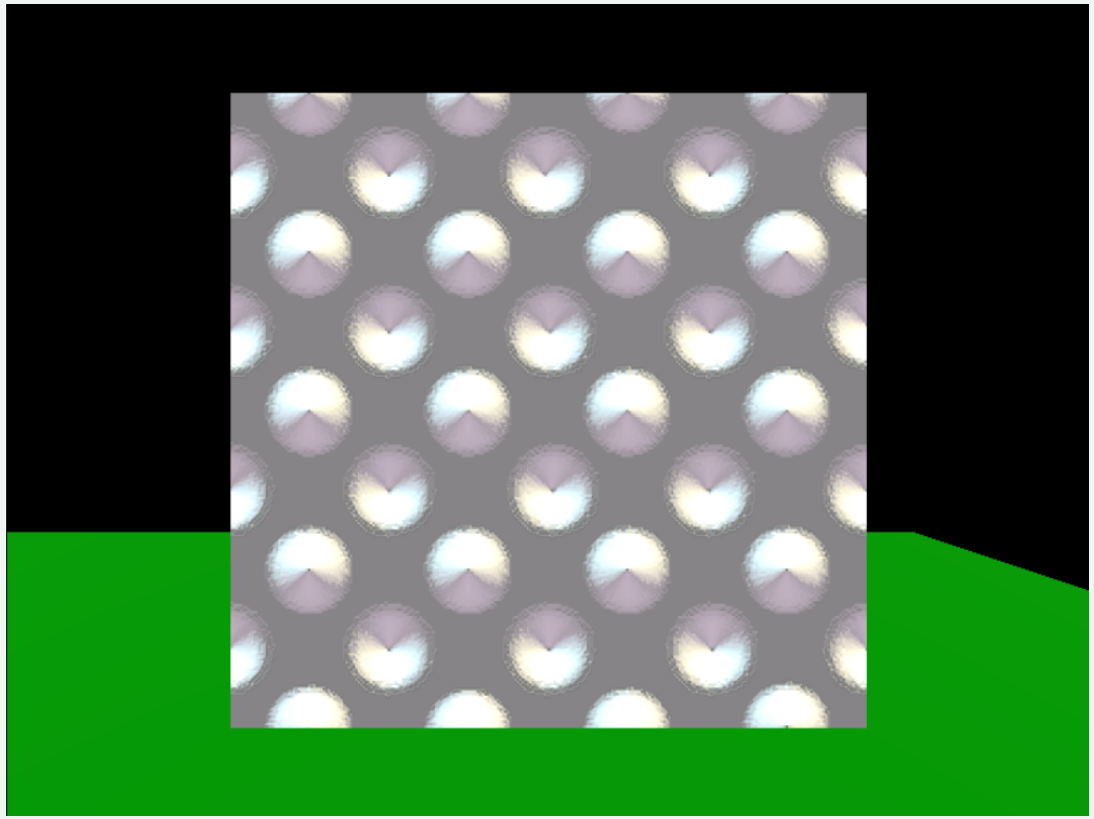
A more realistic example



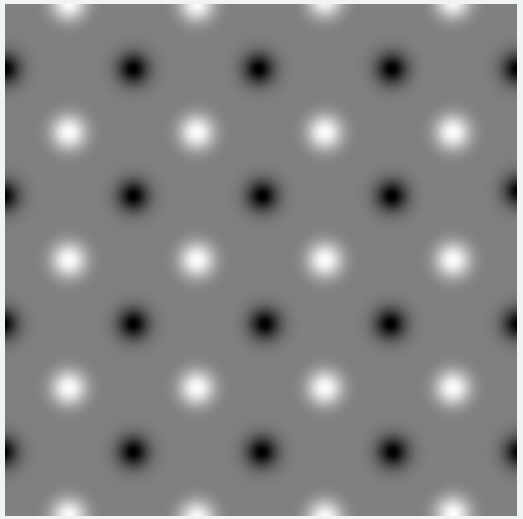
Bright from above

Dark from Below

Lighting not reflection!



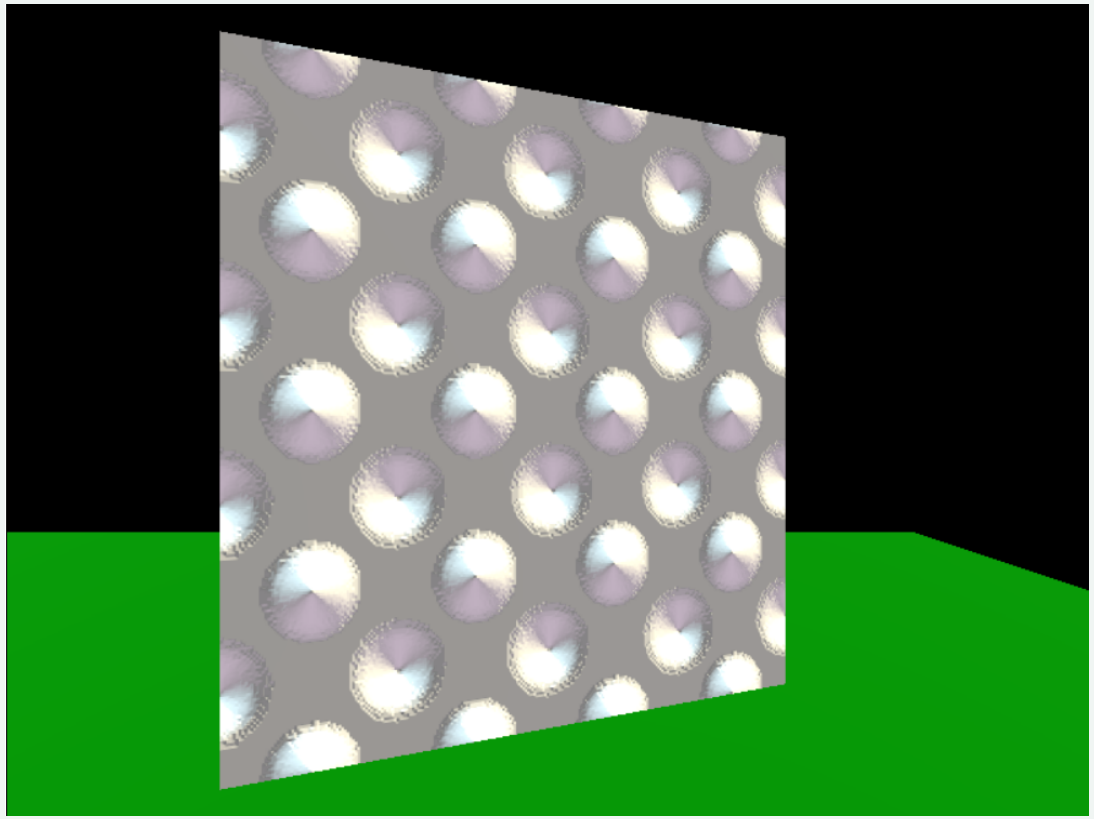
A more realistic example



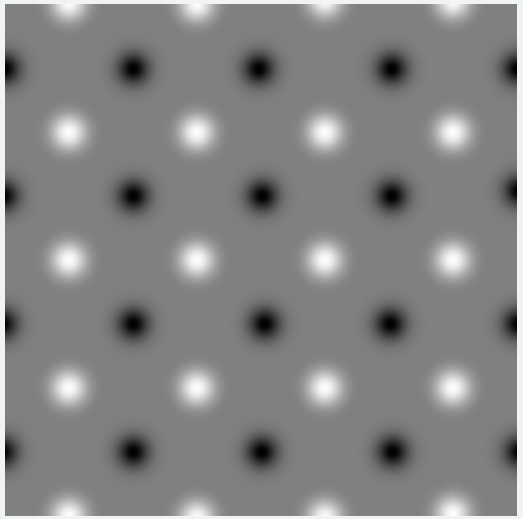
Bright from above

Dark from Below

Lighting not reflection!

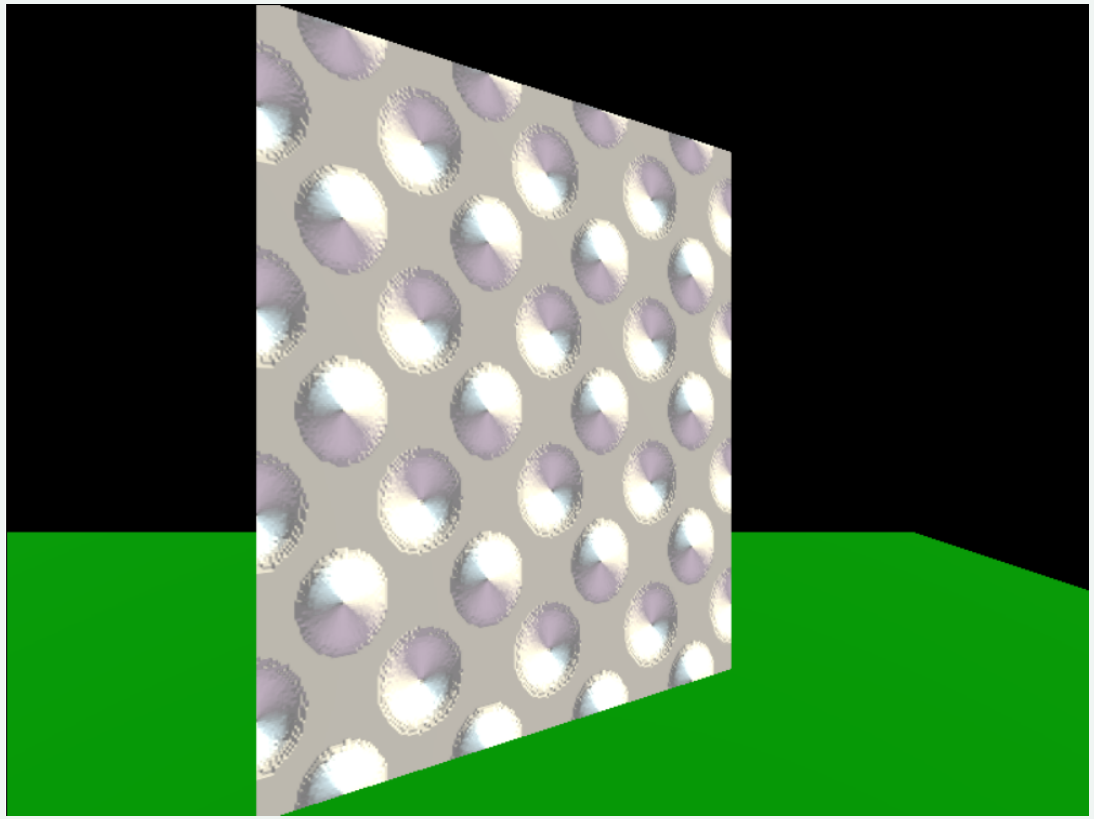


A more realistic example

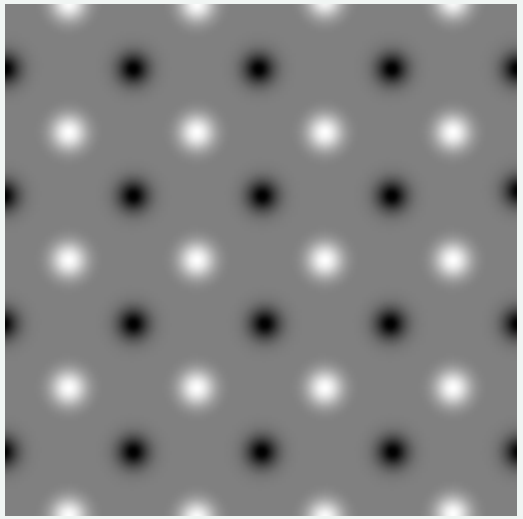


In motion, it can be convincing

(if you don't look too close)



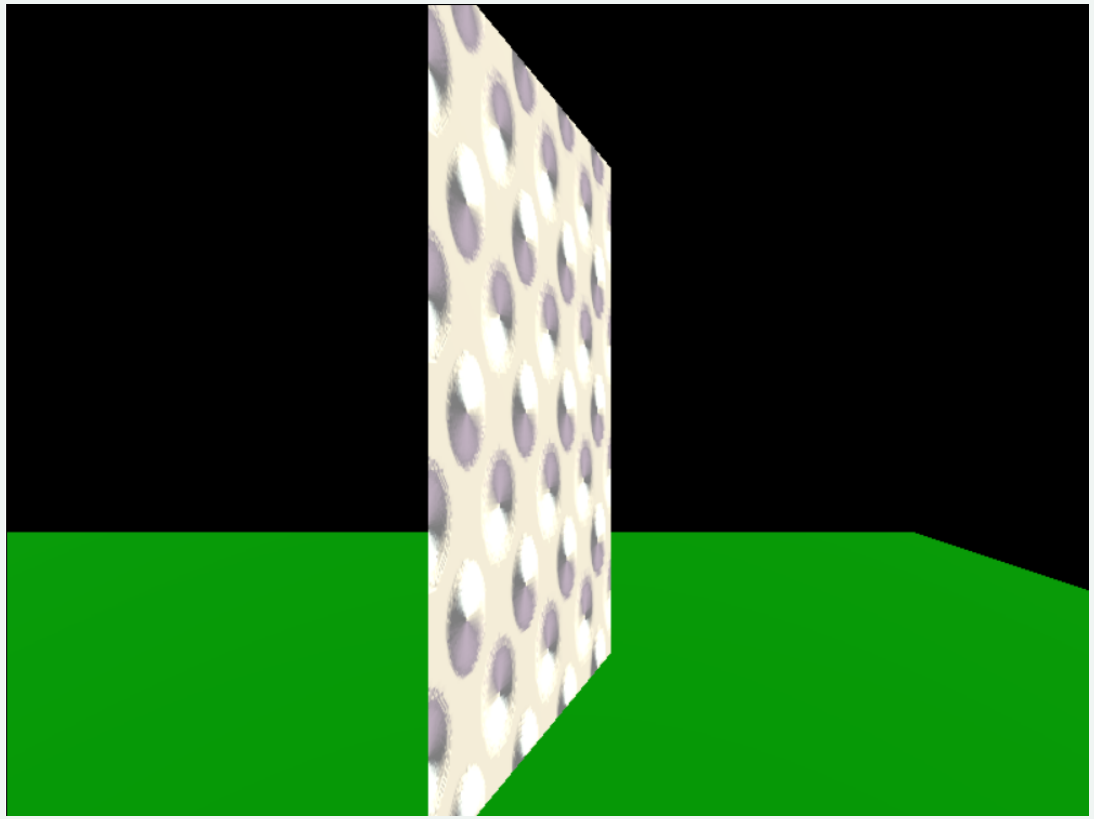
Where it breaks



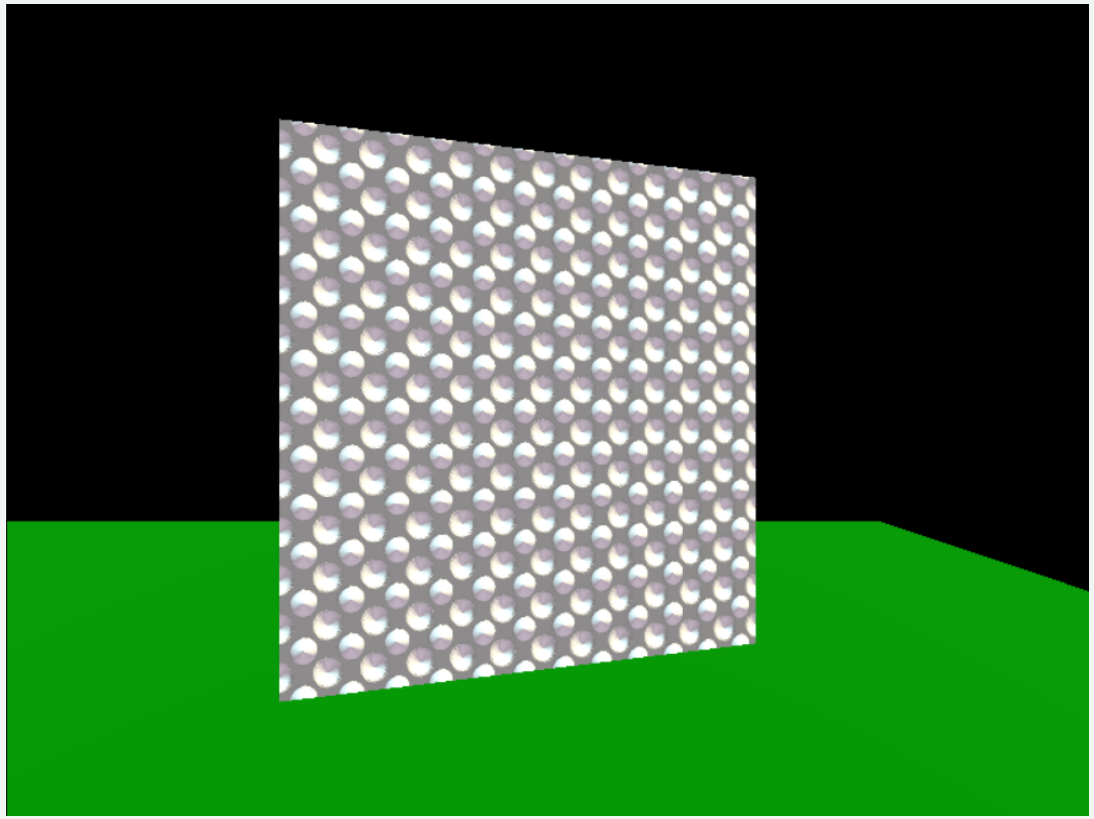
Doesn't change the shape!

It's still flat

Doesn't change the silhouette



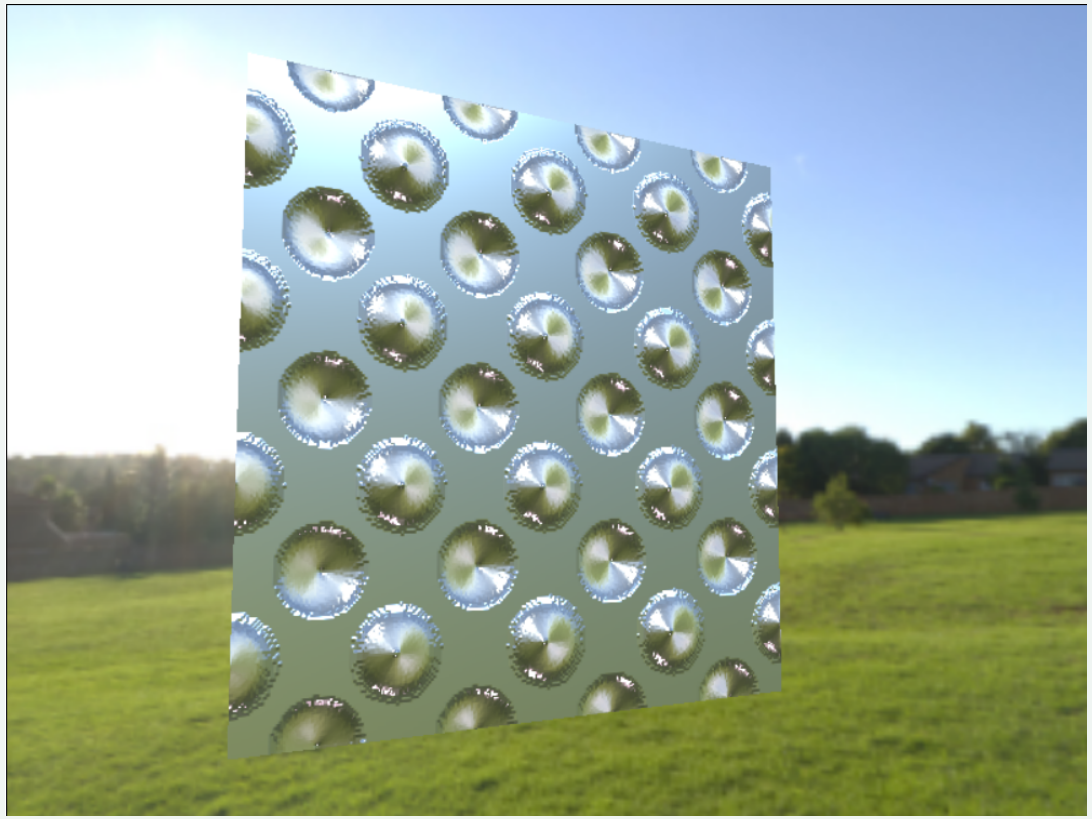
Works better if subtle/small/moving

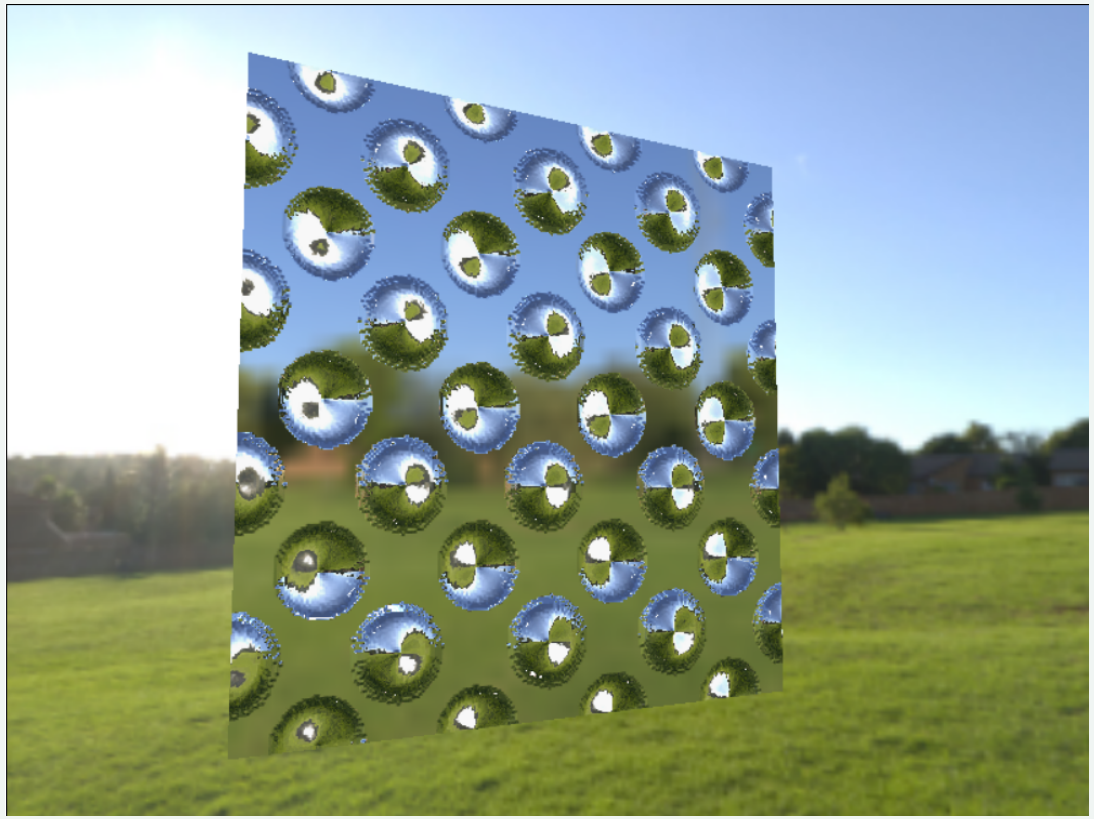


Easy in THREE.js!

```
let bumps = new T.TextureLoader().load("dots-bump.png");
let mat = new T.MeshStandardMaterial({bumpMap:bumps});
```

Combine with Environment Maps ...





Normal Maps and Bump Maps

Good

Easy to specify surface details

Doesn't actually change shape

Gets basic lighting effects

Works with lighting

Works well with Environment Maps

Easy in THREE

Bad

Doesn't change side view

Doesn't cause occlusions

Doesn't work for big effects

Doesn't cause shadows

Why not Displacement Maps?

Much harder to implement

- actually moves pixels (can't do per-pixel)
- may cause gaps
- doesn't fit hardware model

Summary

- Fake Normals to get Smooth Surfaces
- Fake Normals to get Bumpy Surfaces
  - Bump Maps
  - Normal Maps
- Normal/Bump Maps are not Displacement Maps