Lecture 22
Shaders 2

Last Time... SHADERS!

- Draw Objects in our program
  - uniform values over the object
  - each vertex has attributes
- Process each vertex independently
  - vertex shaders!
- Rasterizer converts to fragments
  - each fragment depends on 3 vertices
- Process each fragment
  - fragment shader determines color

But the shaders were BORING

Vertex Shader

```
uniform mat4 modelViewMatrix;
attribute vec3 position;
attribute vec3 color;

varying vec3 vcolor;

void main() {
   gl_Position = modelViewMatrix*
        vec4(position,1.0);
   vcolor = color;
}
```

Fragment Shader

```
varying vec3 vcolor;

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

Now that we can we can write shaders...

What can we do that is more intersting?

- better coloring (lighting and texture)
- procedural patterns (computed colors)
- other tricks

Lighting

THREE implements lighting very well (nice shaders)

- fancy lighting models
- correct works with different kinds of lights
- supports all kinds of maps (environment, shadow, light, ...)

Generally, we don't need to implement it ourselves

- if we want to understand lighting
- if we want to do something non-standard
- if we want to combine it with some other shading

Where to implement lighting?

Vertex Shader

Compute lighting at every vertex

Interpolate colors across triangle

Per-Vertex Lighting

Gouraud Shading

Used in the old days

Fragment Shader

Interpolate normals and surface colors

Compute lighting at every pixel

Per-pixel (fragment) shading

Hardware is fast enough to do it

What Lighting to Implement?

We did actually talk about lighting (Lecture 17)

Local Lighting

1. consider each point individually
2. assume light comes from sources (not other objects)
3. only consider how material reflects light

this means no global effects:

- no shadows
- no reflections
- no spill / bleed

we used texture hacks

"The" Lighting Model (Historic)

- Emission (things give off light)
- Ambient (light from nowhere)
- Specular (direct reflection)
- Diffuse (rough reflection)

Less Historic

- ditch the hack pieces (emission, ambient)
- use less idealized models for specular and diffuse

Diffuse Reflection

$$r_{\text{diffuse}} = \hat{\mathbf{n}} \cdot \hat{\mathbf{l}}$$

where:

- r_{diffuse} = amount of diffuse reflection
- $\hat{\mathbf{n}}$ - unit surface normal
- $\hat{\mathbf{l}}$ - unit vector to light source

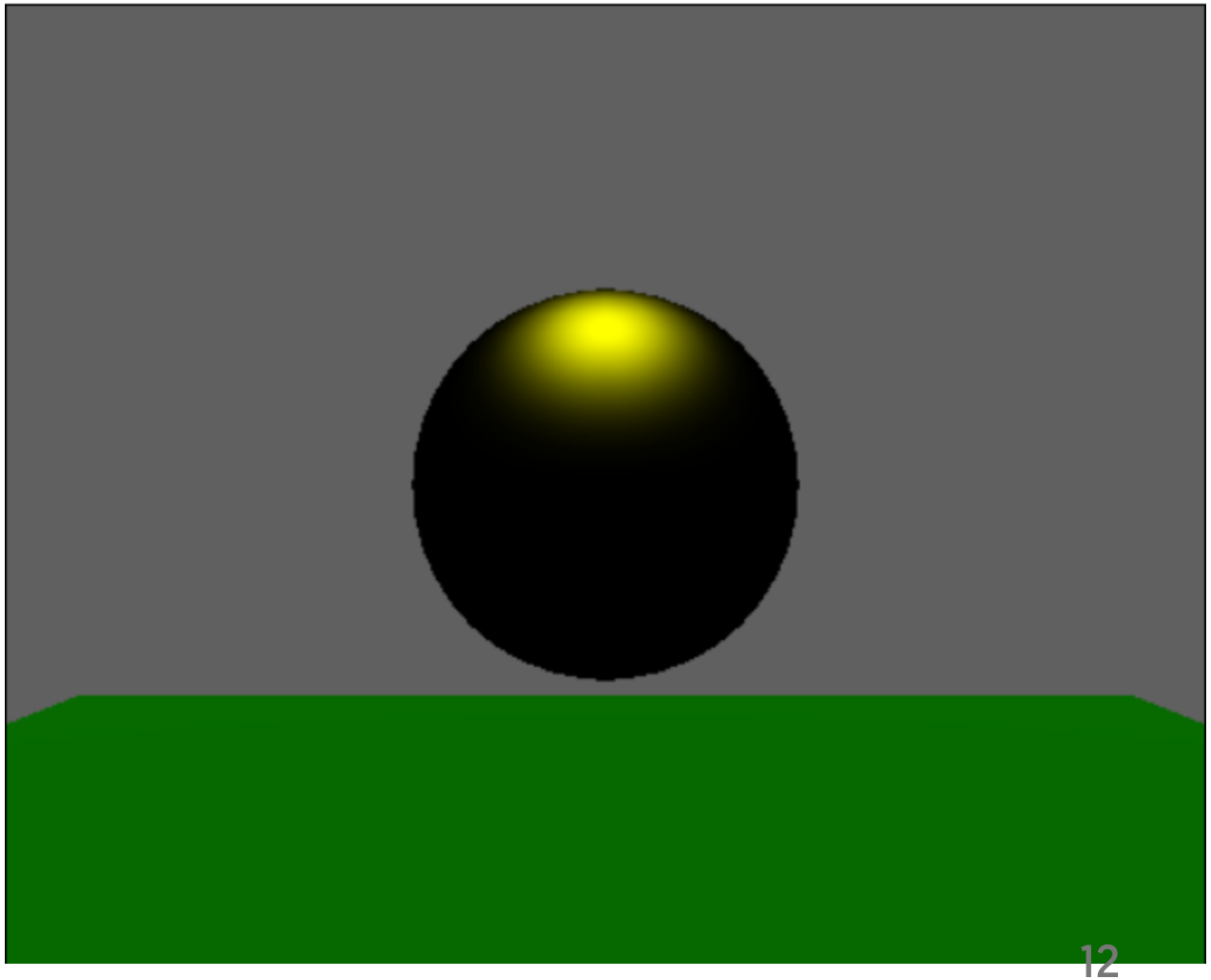
Phong Specular Model (see the book)

$r_{specular} = (\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^p$

where

- $\hat{\mathbf{n}}$ is the normal vector
- $\hat{\mathbf{h}}$ is the half-way vector (between $\hat{\mathbf{l}}$ and $\hat{\mathbf{e}}$)
- p is the "shininess" (material property), phong exponent
- $r_{specular}$ is the amount of specular reflection

Specular



The Phong Lighting Model

$$\mathbf{color} = \mathbf{c_e} + \mathbf{c_a}\ \mathbf{l_a} + \sum_{l \in lights} \left((\hat{\mathbf{n}} \cdot \hat{\mathbf{l}})\ \mathbf{c_{light}}\ \mathbf{c_d} + (\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^p\ \mathbf{c_{light}}\ \mathbf{c_s} \right)$$

which uses:

- the surface geometry $\hat{\mathbf{n}}$
- the light direction $\hat{\mathbf{l}}$
- the eye direction (combined with light direction into the half-vector) ($\hat{\mathbf{h}}$)
- the light color (instensity) $\mathbf{l_a}$) and $\mathbf{c_{light}}$
- the surface properties $\mathbf{c_e}$, $\mathbf{c_a}$, $\mathbf{c_d}$, $\mathbf{c_s}$ and p

Diffuse

$$\mathbf{color} = (\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}) \quad \mathbf{c}_{\mathbf{light}} \quad \mathbf{c}_{\mathbf{d}}$$

where

- $\hat{\mathbf{n}}$ - unit surface normal
- $\hat{\mathbf{l}}$ - unit vector to light source
- $\mathbf{c}_{\mathbf{light}}$ - color/intensity of light
- $\mathbf{c}_{\mathbf{d}}$ - color of the material (diffuse reflectance)

Diffuse and Ambient

$$\mathbf{color} = \left((\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}) \ \mathbf{c}_{\mathbf{light}} \ + \mathbf{c}_{\mathbf{ambient}} \right) \ \mathbf{c}_{\mathbf{d}}$$

where

- $\hat{\mathbf{n}}$ - unit surface normal
- $\hat{\mathbf{l}}$ - unit vector to light source
- $\mathbf{c}_{\mathbf{light}}$ - color/intensity of light
- $\mathbf{c}_{\mathbf{ambient}}$ - color/intensity of ambient light
- $\mathbf{c}_{\mathbf{d}}$ - color of the material (diffuse reflectance)

Where do those values come from?

$\hat{\mathbf{n}}$ - unit surface normal

$\hat{\mathbf{l}}$ - unit vector to light source

$\mathbf{c}_{\mathbf{light}}$ - color/intensity of light

$\mathbf{c}_{\mathbf{ambient}}$ - color of ambient light

$\mathbf{c}_{\mathbf{d}}$ - **color of material**

Surface color

uniform for material?

varying (copy from attribute)

read from texture

Where do those values come from?

$\hat{\mathbf{n}}$ - unit surface normal

$\hat{\mathbf{l}}$ - unit vector to light source

$\mathbf{c}_{\text{light}}$ - **color/intensity of light**

$\mathbf{c}_{\text{ambient}}$ - **color of ambient light**

$\mathbf{c}_{\text{d}}$ - color of material

Light Properties

Constants?

Uniforms?

Get from THREE's objects (tricky)

The Geometric Properties

$\hat{\mathbf{n}}$ - **unit surface normal**

$\hat{\mathbf{l}}$ - **unit vector to light source**

$\mathbf{c}_{\mathbf{light}}$ - color/intensity of light

$\mathbf{c}_{\mathbf{ambient}}$ - color of ambient light

$\mathbf{c}_{\mathbf{d}}$ - color of material

What coordinate system?

Transform vertices

Interpolate normal vectors

The Vertex Shader

```
attribute vec3 position;          // these are provided by THREE
attribute vec3 normal;            // you don't need to declare them
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
  gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);

  fNormal = normalize(normalMatrix * normal);
  fPosition = modelViewMatrix * vec4(position, 1.0);
}
```

Transformations of Normals

Transforming an object changes the normals

Normals are transformed by the inverse transpose (adjoint)

What coordinate system?

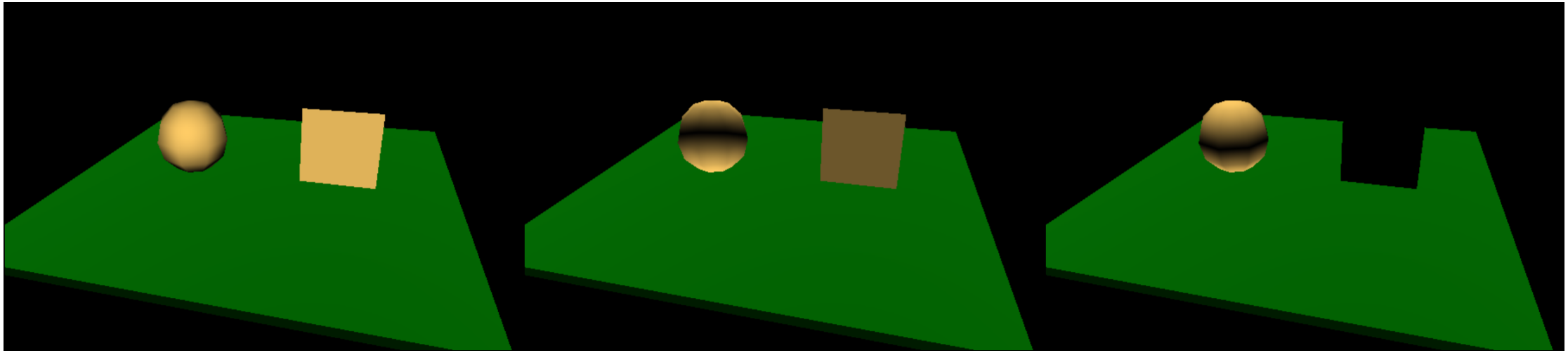
World coordinate system?

Camera coordinate system?

Typically: lighting in camera coordinates

Does it make a difference?

Z-Axis Camera Y-Axis Camera Y-Axis World



What does THREE do?

from the documentation...

```
// = inverse transpose of modelViewMatrix
uniform mat3 normalMatrix;
```

THREE's lights are transformed into view coordinates

Light Geometry: Directional Light

```
const vec3 lhat = vec3(0,1,0);
```

Y Axis in same coordinate system as the normals (view)

Light Geometry 2: Point Light

```
const vec3 lightPos = vec3(10,10,0);
varying vec3 fPosition;

main() {
    ...
    vec3 lhat = normalize(fPosition - lightPos);
```

but light position must be in camera coordinates

Light Geometry2b: Point Light

```
uniform vec3 cameraPosition;
const vec3 lightPos = vec3(10,10,0);
varying vec3 fPosition;

main() {
    ...
    vec3 lhat = normalize(wPosition - lightPos);
```

if we had `wPosition` as a varying as:

```
wPosition = modelMatrix * vec4(position, 1.0);
```

Diffuse Lighting Shader...

```
varying vec3 v_normal;
const vec3 lightDir = vec3(0,0,1);
const vec3 baseColor = vec3(1,.8,.4);
const vec3 lightColor = vec3(.5,.5,.5);
const vec3 ambientColor = vec3(.1,.1,.2);

void main()
{
    /* we need to renormalize the normal since it was interpolated */
    vec3 nhat = normalize(v_normal);
    /* deal with two sided lighting */
    float light = abs(dot(nhat, lightDir));
    // or not...
    float light = clamp(dot(nhat, lightDir),0.0,1.0);

    /* brighten the base color */
    gl_FragColor = vec4((light*lightColor + ambientColor) * baseColor,1);
}
```

Beyond Diffuse

Implement the equations of the model...

What does THREE actually do?

1. `ShaderMaterial` - whatever we tell it to do
2. `MeshPhongMaterial` - basically the model we discussed

   see `lights_phong_pars_fragment.glsl`
3. `MeshStandardMaterial` - less clear what it does

   might be Phong with improved workflow
4. `MeshPhysicalMaterial` - a more complex model

What if we want to use THREE's lights

We need to get them from THREE

- uniforms lib

We need to loop over however many lights there are

We need to do all the different kinds of lights

This is tricky - and not that instructive

Best bet: use the code from THREE's shaders

Textures

1. Image-Based Textures
2. Procedural Textures

Image-Based Textures in Shaders

Texture Maps are part of "Texture Objects"

- texure map (image)
- mip-map (images of different sizes)
- parameters (filter type, wrapping, ...)

GLSL Type `sampler2D` (for a 2D image)

Getting the color from a texture

```
 varying vec2 v_uv;

 // get the texture from the program
 uniform sampler2D texture;

void main()
{
    gl_FragColor = texture2D(texture, v_uv);
}
```

How does it know how to filter?

Mip-Mapping set up in JavaScript (THREE's default)

Vertex Shader

It doesn't know how to sample - uses the texture itself

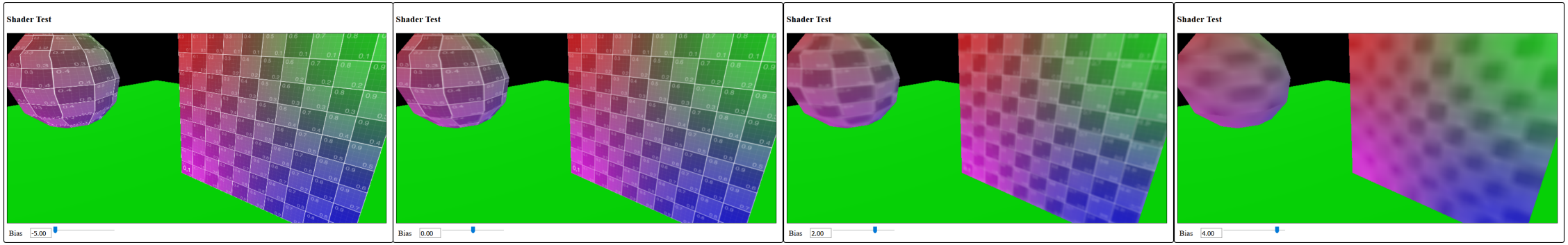
Fragment Shader

It knows how far the next pixel is and uses that to estimate

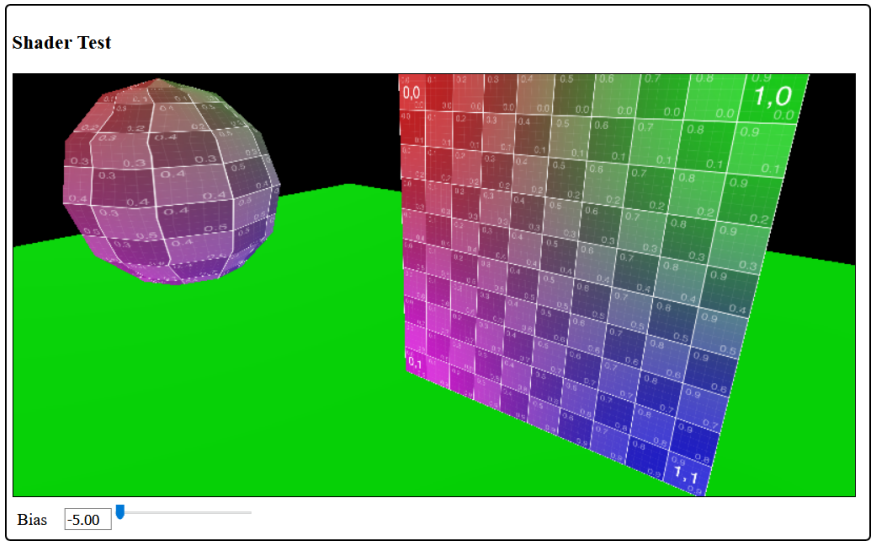
Adjusting Filtering

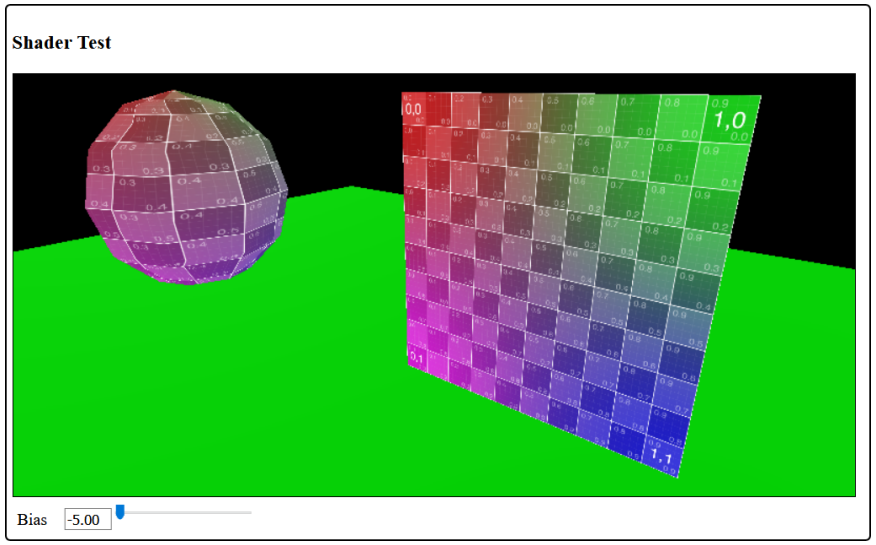
we can **bias** the mip-map level (amount of blur)

```
gl_FragColor = texture2D(texture, v_uv, bias);
```

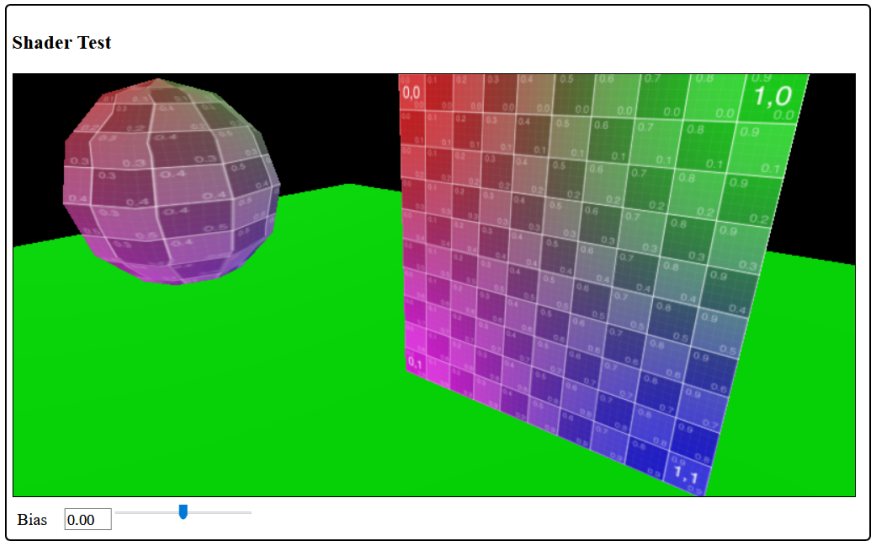


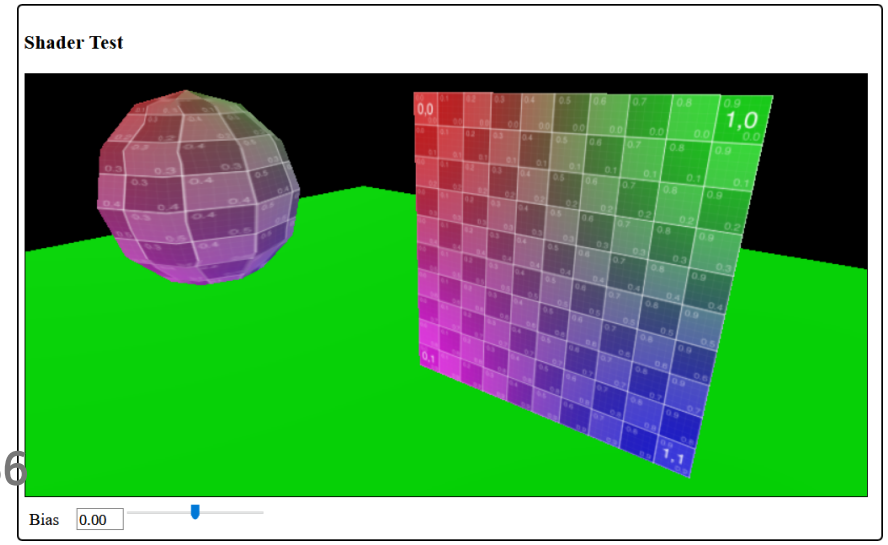
no filtering





filtering





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

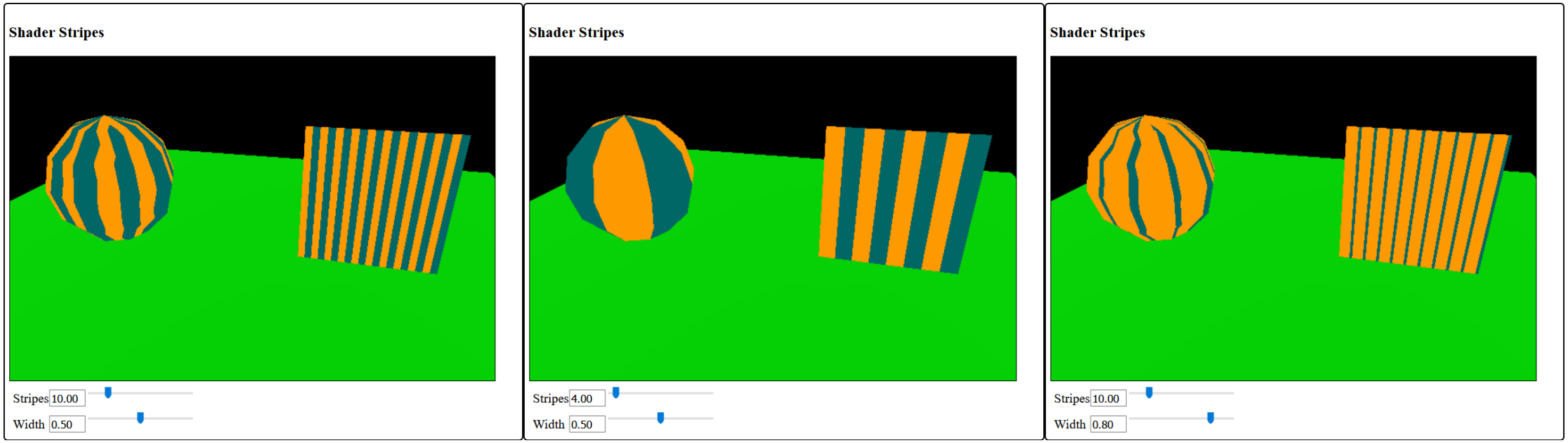
Convenient

Use Built-Ins, not Expressions

- mix

A Shader... Stripes

Parameters - change as needed

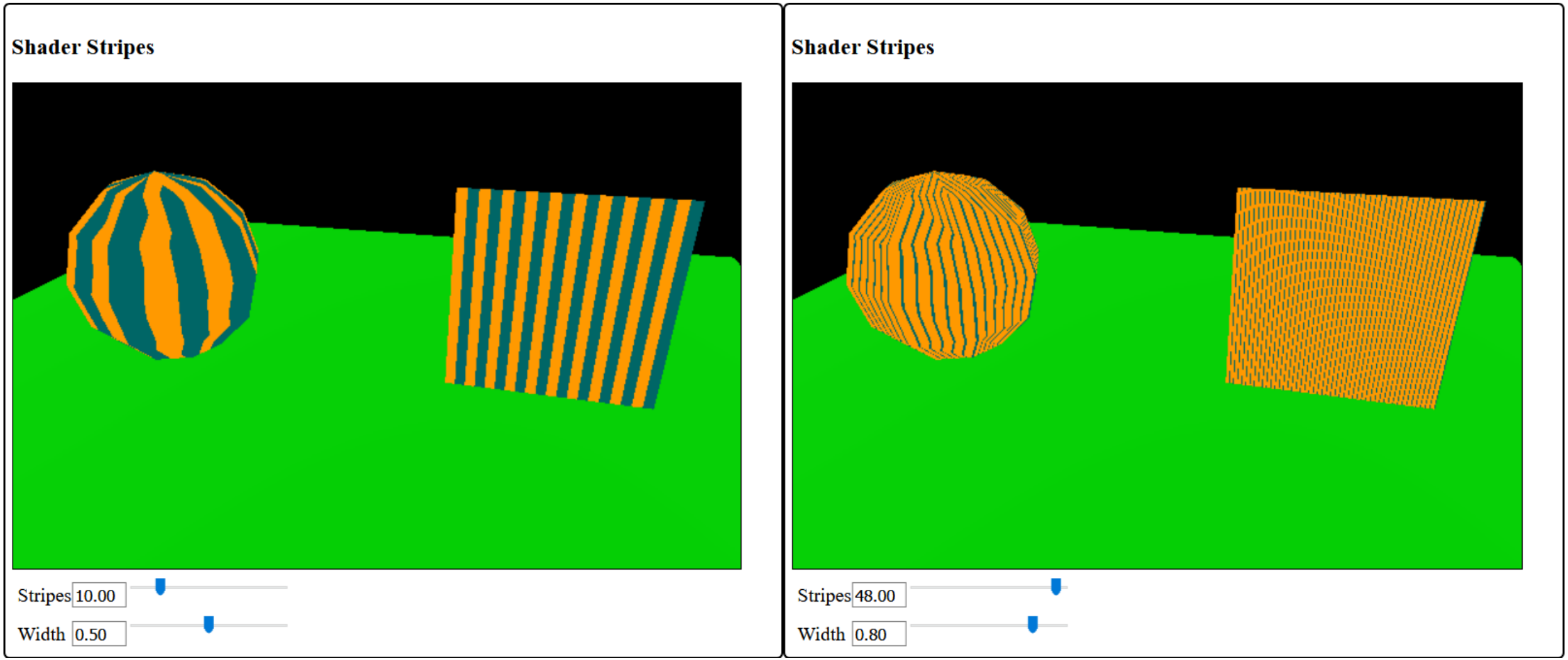


Parameters for the number of stripes and the width

Or change other things (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);
}
```

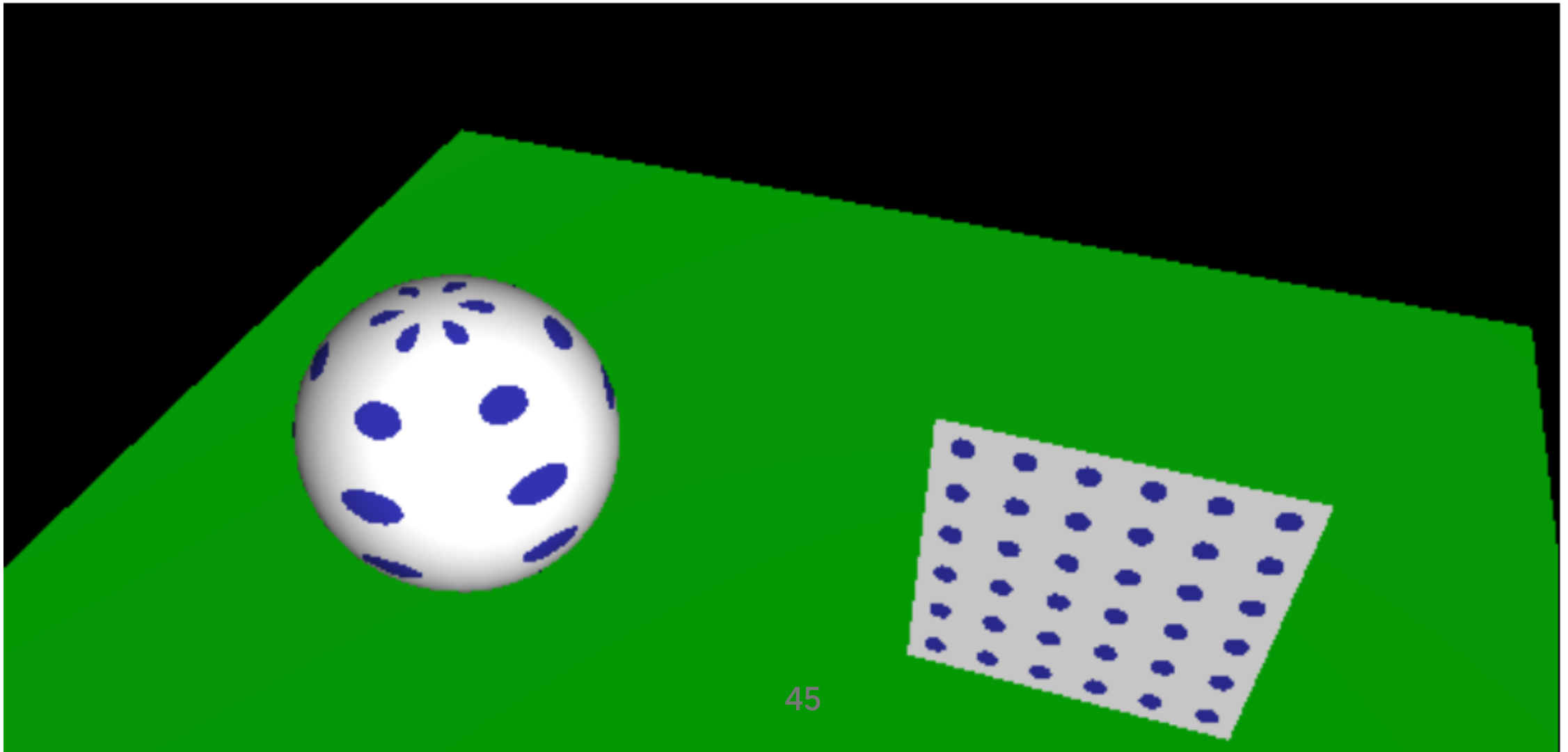
Note the jaggies



Aliasing and Anti-Aliasing

We'll talk about it next week...

Another Shader... Dots

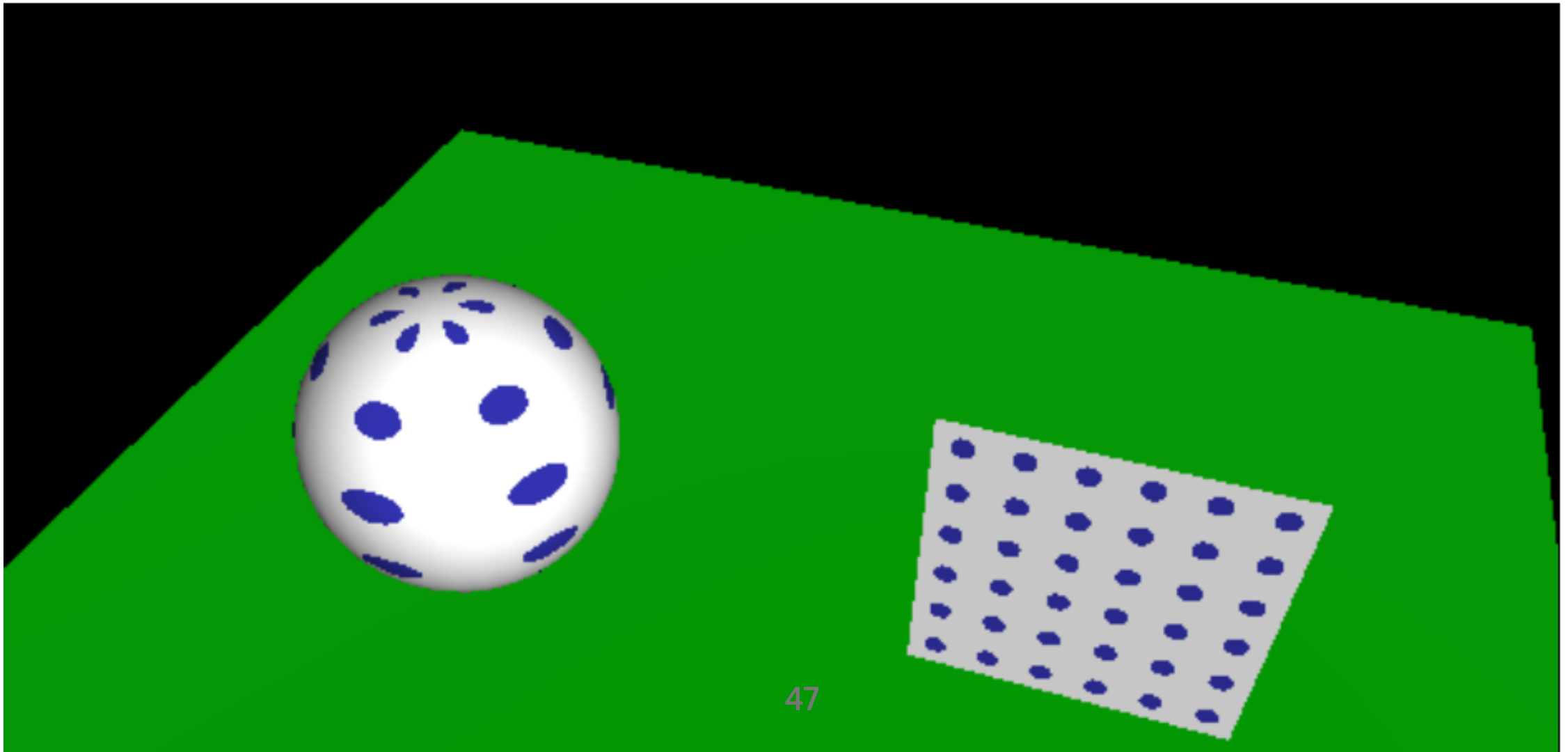


Functions (GLSL)

```
varying vec2 v_uv;
varying vec3 l_normal;

uniform vec3 light;
uniform vec3 dark;
uniform float dots;
uniform float radius;

const float ambient = .2;
const vec3 lightDir = normalize(vec3(1,1,1));

void main()
{
    float dc = fdot(v_uv);
    vec3 baseColor = mix(light,dark,dc);

    // simple diffuse lighting
    vec3 nhat = normalize(l_normal);
    float bright = clamp(dot(nhat, lightDir),0.0,1.0);
    // add ambient
    bright = clamp(bright+ambient,0.0,1.0);

    // brighten the base color
    gl_FragColor = vec4(bright * baseColor,1);
}
```

```
float fdot(vec2 uv) {
    float x = uv.x * dots;
    float y = uv.y * dots;

    float xc = floor(x);
    float yc = floor(y);

    float dx = x-xc-.5;
    float dy = y-yc-.5;

    float d = sqrt(dx*dx + dy*dy);

    dc = 1.0-step(radius,d);

    return dc;
}
```

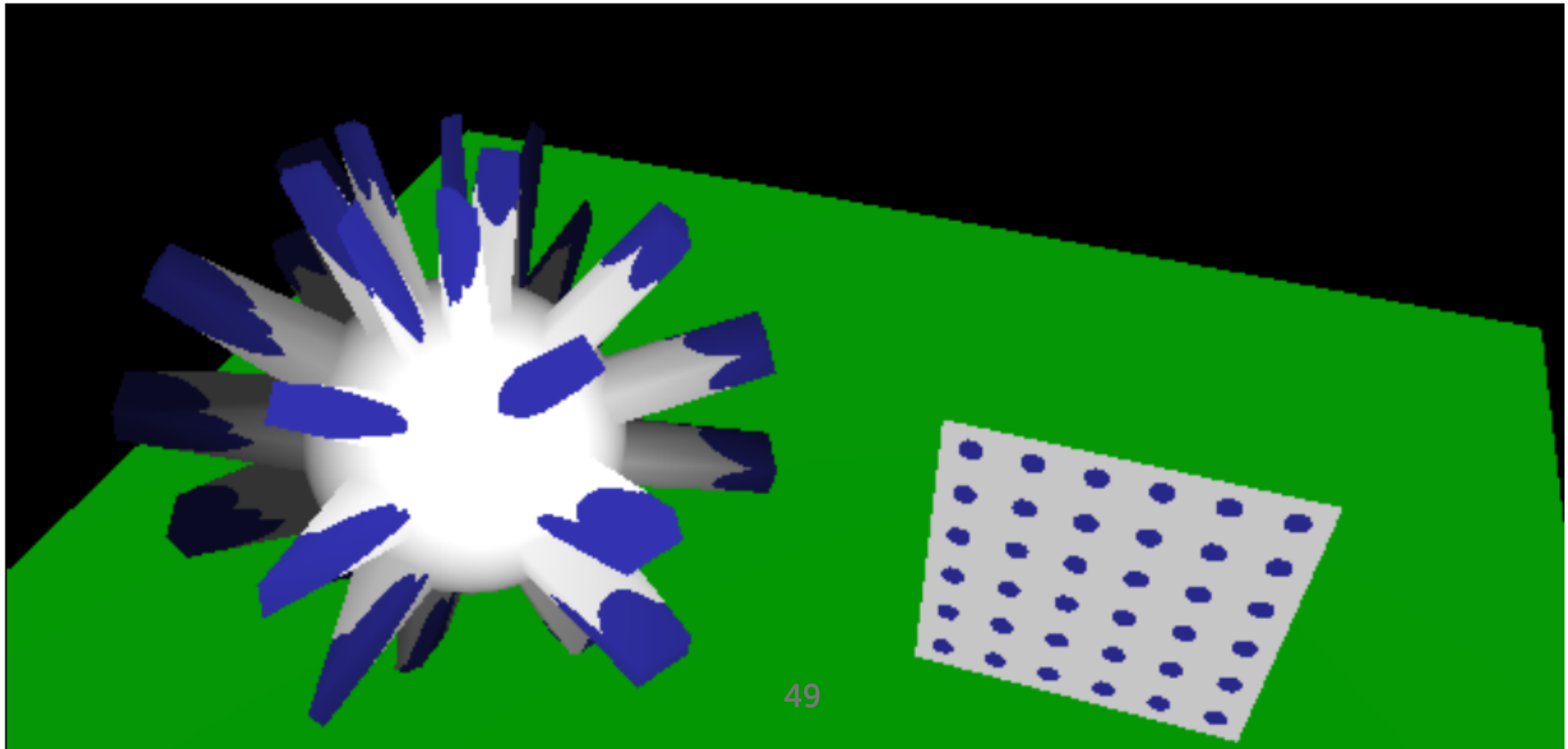
Coronavirus?



Displacement Map - move vertices!

```
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;

    // transformed normal (for lighting)
    l_normal = (modelMatrix * vec4(normal,0)).xyz;
}
```

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

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

There's more...

- Other Shader Programming Tricks
- Understanding Aliasing (for more than shaders)
- Things beyond shaders