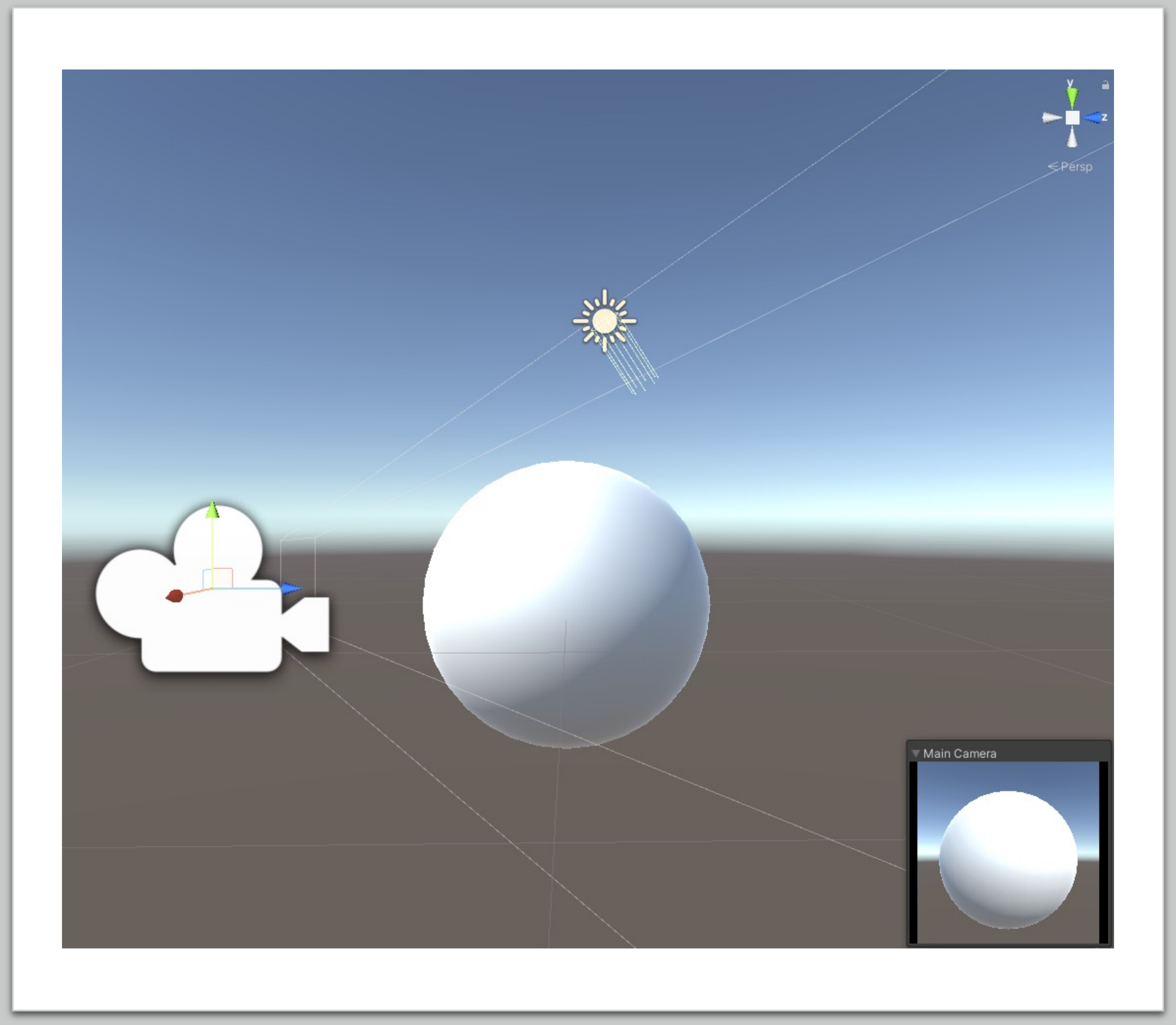
Phong Shader

Dr. Hamish Carr Dr. Rafael Kuffner dos Anjos

Phong Shading

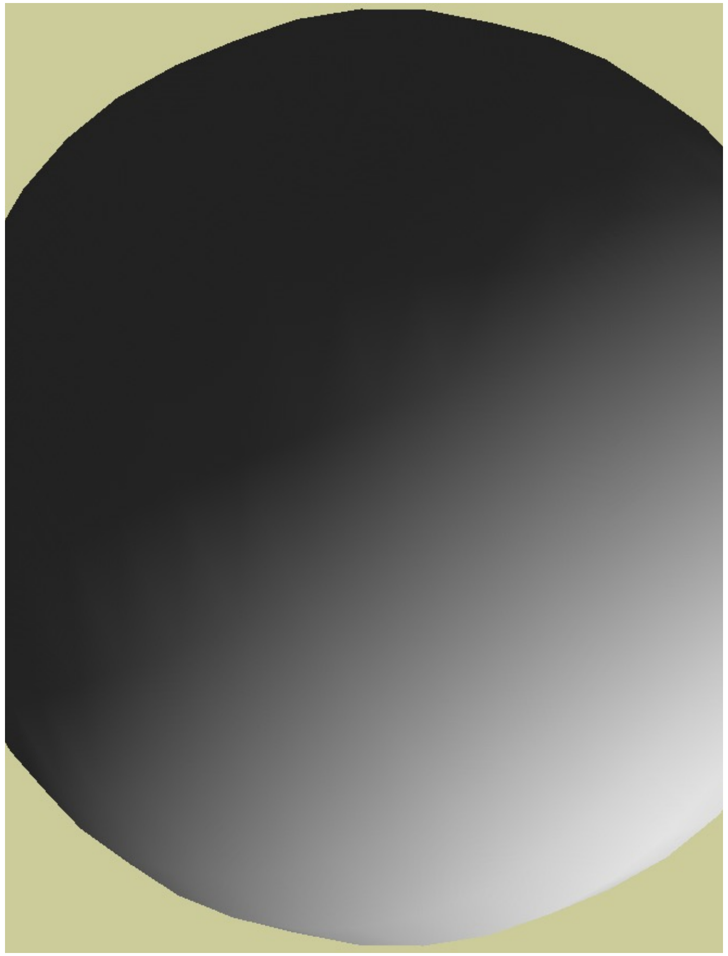
- The first real shader people write
- Per-pixel **Blinn-Phong** illumination
- Just "Phong" is rarely used anymore. So when we say Phong we normally mean Blinn-Phong
- Pass all material properties as vertex attributes
- Interpolate the normal vector
  - In view/world coordinates
- Properties of the light are uniform
  - Since they're the same everywhere

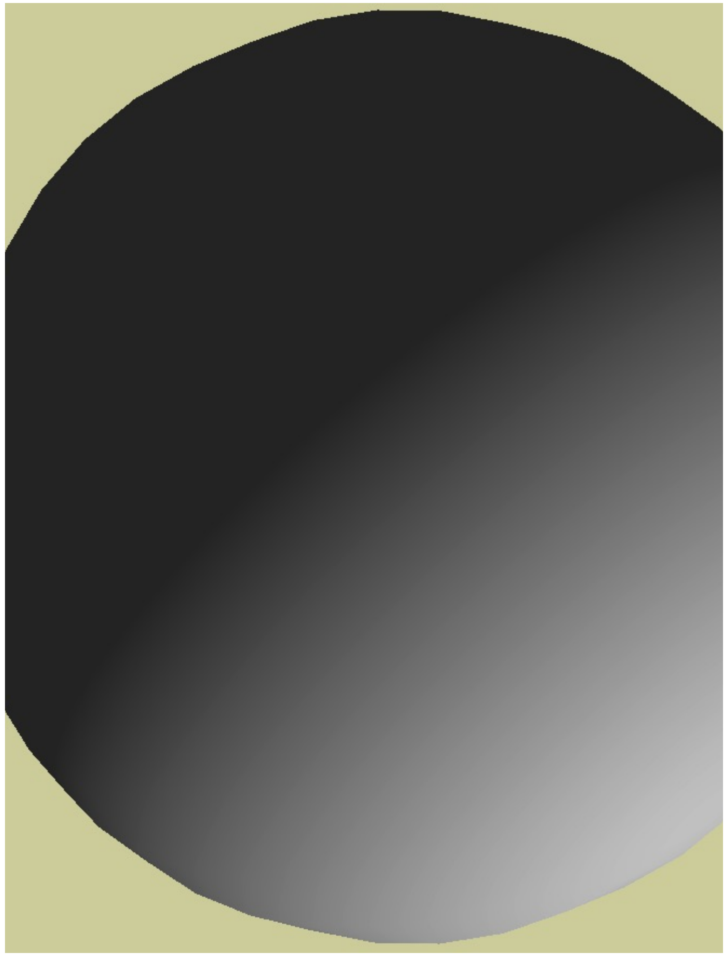


Fixed Function Pipeline

- Vertex stage transforms vertices
- Performs lighting calculations
- Passes colour to rasteriser
- Rasteriser interpolates colour
  - This is called Gouraud shading
- Fragment stage combines with texture
- And stores in frame buffer

Gouraud vs. Phong Shading





- Phong shading is the ideal
  - Every pixel computes its own lighting
  - But this is (was) expensive
- Gouraud shading is a hack
  - Interpolate colour over each triangle
  - Much cheaper
  - But many fragments => many computations

Recap: Blinn-Phong model

$$I_{total}(p) = I_{specular}(p) + I_{diffuse}(p) + I_{ambient}(p) + I_{emitted}(p)$$

$$= l * r_{specular} \left(\frac{\vec{n} \cdot \vec{v}_b}{\|\vec{n}\| \|\vec{v}_b\|}\right)^{h_{specular}}$$

$$+ l * r_{diffuse} \frac{\vec{n} \cdot \vec{v}_l}{\|\vec{n}\| \|\vec{v}_l\|}$$

$$+ l * r_{ambient}$$

$$+ l_{emitted}$$

Goal

- Blinn-Phong shading in fragment shader
- Needs as input:
  - Fragment (position implicit)
  - Normal (in WCS or VCS)
  - Light vector (ditto)
  - Material properties
  - So vertex shader has to pass them through

Vertex Shader Inputs

Vertex Buffer:

Vertex position, normal

Vertex texture coordinates

Vertex Color

Uniform Buffer:

Light position & properties

Transformation Matrices

Vertex Shader Outputs

- Vertex positions in NDCS
- Normals in VCS
- Light vectors in VCS
- Vertex colors
- Texture coordinates

Vertex Shader I/O

```
// Vertex Shader code
// layout ESSENTIALLY means struct
// location / binding means which element in the buffer
// (according to our layout specified above)
// in / out specify read/write access
// uniform specifies data shared between instances

// this assumes we've set the vertex buffer up for this
// and assumes the material color is the same for ambient, diffuse, specular

layout (location = 0) in vec4 inPosition;
layout (location = 1) in vec3 inColor;
layout (location = 2) in vec2 inTexCoord;
layout (location = 3) in vec4 inNormal;

layout(location = 0) out vec3 fragColor;
layout(location = 1) out vec2 fragTexCoord;
layout(location = 2) out vec4 fragLightVector;
layout(location = 3) out vec4 fragEyeVector;
layout(location = 4) out vec3 fragNormal;
```

Vertex Shader Uniforms

```glsl
// This first line is the declaration with qualifiers
// UniformBufferObject is the type name
// ubo is the local "variable name"

layout(binding = 0) uniform UniformBufferObject
{
    mat4 model;     // as above, the model matrix
    mat4 view;      // the view matrix
    mat4 proj;      // the projection matrix
}ubo;

//Information about our light
 layout(binding = 1) uniform LightInformation {
        vec4 lightPosition;
        vec4 lightColor;
 }lighting;

//We dont need access to the material here
```


UNIVERSITY OF LEEDS

Vertex Shader Main()

```
void main()
{
    //Apply the matrices to the vertex position
    vec4 VCS_position = ubo.view * ubo.model * inPosition;

    //We want to retain the rest in VCS (projection distorts lighting)
    fragNormal = ubo.view * ubo.model * inNormal

    //Assume lighting is not at infinity (point light)
    fragLightVector = ubo.view * lighting.lightPosition - VCS_position

    //Eye is at (0,0,0), so the eye vector is easy
    fragEyeVector = - VCS_position;

    //now compute the fully transformed position
    gl_Position = ubo.proj * VCS_position;

    //Pass along the fragment colour & uv coords
    fragColor = inColor;
    fragTexCoord = inTexCoord;
}
```

Raster Stage

Rasterises from vertex positions (NDCS)

Interpolates

Normals (VCS)	Light vectors (VCS)	Lighting colour (RGB)	Material properties (if desired)	Texture coordinates (UV)

Fragment Shader I/O and Uniforms

```
//Fragment shader inputs match vertex shader outputs

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTexCoord;
layout(location = 2) in vec4 fragLightVector;
layout(location = 3) in vec4 fragEyeVector;
layout(location = 4) in vec3 fragNormal;

layout(location = 0) out vec4 outColor;
```

```
//Information about our light
 layout(binding = 1) uniform LightInformation {
        vec4 lightPosition;
        vec4 lightColor;
 }lighting;

// a second use of a uniform buffer for material
layout(binding = 2) uniform MaterialConstants
{
    //the three principal Phong components
    vec4 Ambient;
    vec4 Diffuse;
    vec4 Specular;
    float SpecularExponent;
}material;

//declare the texture sampler
layout (binding = 3) uniform sampler2D texSampler;
```

Fragment Shader Main(), I

```
//Fragment shader main routine
void main()
{
    //I've decided to ignore emissive lighting
    //So we compute the other three

    //Ambient is the easy one - it's just a constant
    vec4 I_ambient = lighting.lightColor * material.Ambient * fragColor;

    //Because of interpolation, we are not guaranteed unit vectors, so normalise
    vec4 normEyeVector = normalise(fragEyeVector);
    vec4 normLightVector = normalise(fragLightVector);
    vec4 normNormal = normalise(fragNormal);

    //Diffuse lighting uses the dot product
    float diffuseDotProduct = dot(normLightVector,normNormal);
    vec4 I_diffuse = lighting.lightColor*material.Diffuse * fragColor * diffuseDotProduct;
```

Fragment Shader Main(), II

```
    //Specular lighting uses the half-angle
    vec4 halfAngleVector = normalise((normEyeVector + normLightVector)/2.0);
    float specularDotProduct = dot(halfAngleVector,normNormal);
    float specularPower = pow(specularDotProduct, material.SpecularExponent);
    vec4 I_specular = lighting.lightColor * material.Specular * fragColor * specularPower;

    //tompute total lit colour
    vec4 I_total = I_ambient + I_diffuse + I_specular;

    //now we need to deal with textures
    vec4 textureColor = texture(TexSampler, fragTexCoord);

    //finally, we combine them (i.e we are modulating texture & lighting)
    outColor = I_total * textureColor;
}
```

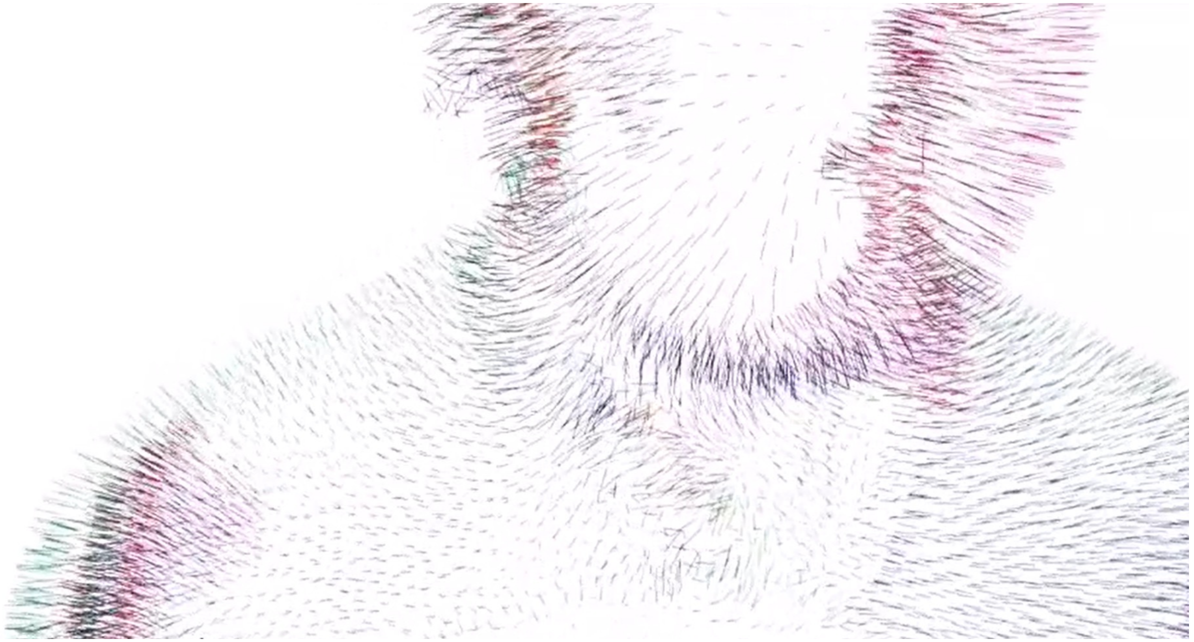
Blinn-Phong shader

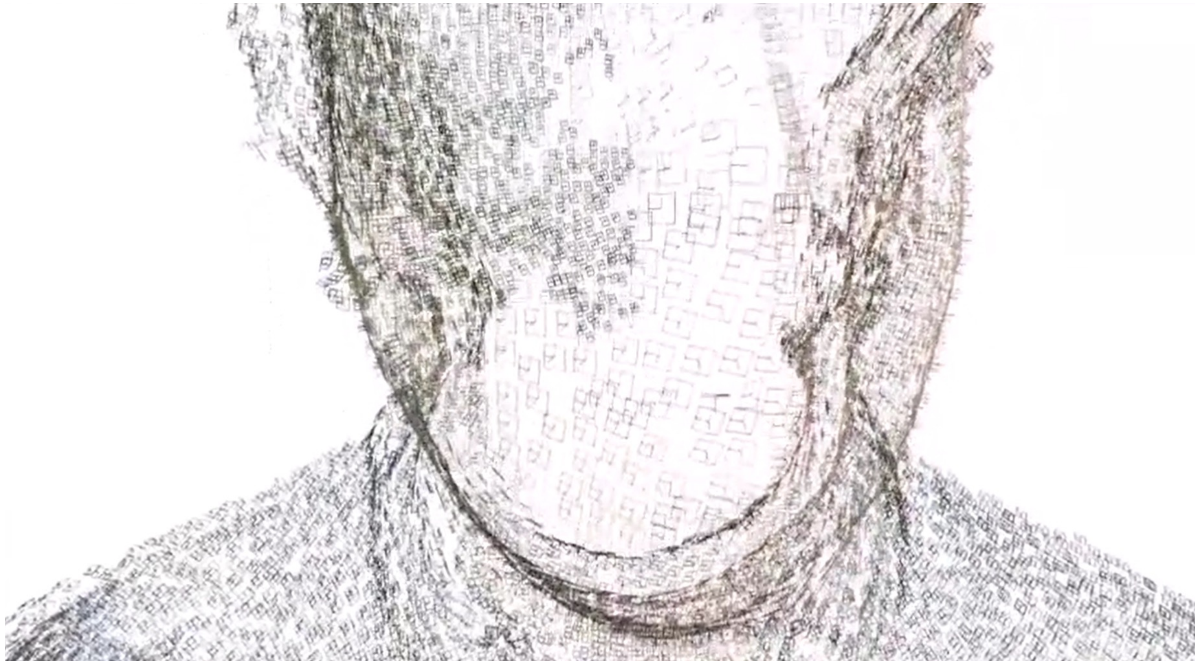
- There are variants of this that are equally OK.
- Per-vertex material.
- Lighting with colour per each component
- Non modulating with texture
- Texture mapping material properties
- Etc.

Extra exemple: Geometry Shader

- As this lecture was quick, lets look at a different example:
- Geometry shader for point-cloud rendering
- Input: Points and normals.
- Normals calculated offline by PCA of a neighborhood of points
- Output: quads oriented towards normals.
- Simple version just has normals pointing at the camera: Billboards!

Surface-aligned splats





- Given the normal and point
- Find 2 orthogonal vectors in that plane (several possible methods)
- Generate a quad of size s
- Filter it in fragment shader

Simple version: Camera aligned

```
uniform mat4 camera;
uniform mat4 model;
//We get points in
layout(points) in;
//And we put out a trianglestrip of 4 vertices
layout(triangle_strip, max_vertices = 4) out;

uniform float size;

in Vertex
{
    vec4 color;

} vertex[];

out vec4 VertexColor;

void main() {
    mat4 V = camera;
    vec4 c = vertex[0].color;

    //Camera right and up vectors
    vec3 right = vec3(V[0][0],V[1][0],V[2][0]);
    vec3 up = vec3(V[0][1], V[1][1], V[2][1]);
    vec3 P = gl_in[0].gl_Position.xyz;
```

```
    //Generate 4 vertices
    vec3 va = P - (right + up) * size;
    gl_Position =  camera * model *vec4(va, 1.0);
    VertexColor = c;
    EmitVertex();

    vec3 vb = P - (right - up) * size;
    gl_Position = camera * model *vec4(vb, 1.0);
    VertexColor = c;
    EmitVertex();

    vec3 vd = P + (right - up) * size;
    gl_Position = camera * model *vec4(vd, 1.0);
    VertexColor = c;
    EmitVertex();

    vec3 vc = P + (right + up) * size;
    gl_Position =camera * model * vec4(vc, 1.0);
    VertexColor = c;
    EmitVertex();

    EndPrimitive();
}
```


UNIVERSITY OF LEEDS

Surface-aligned

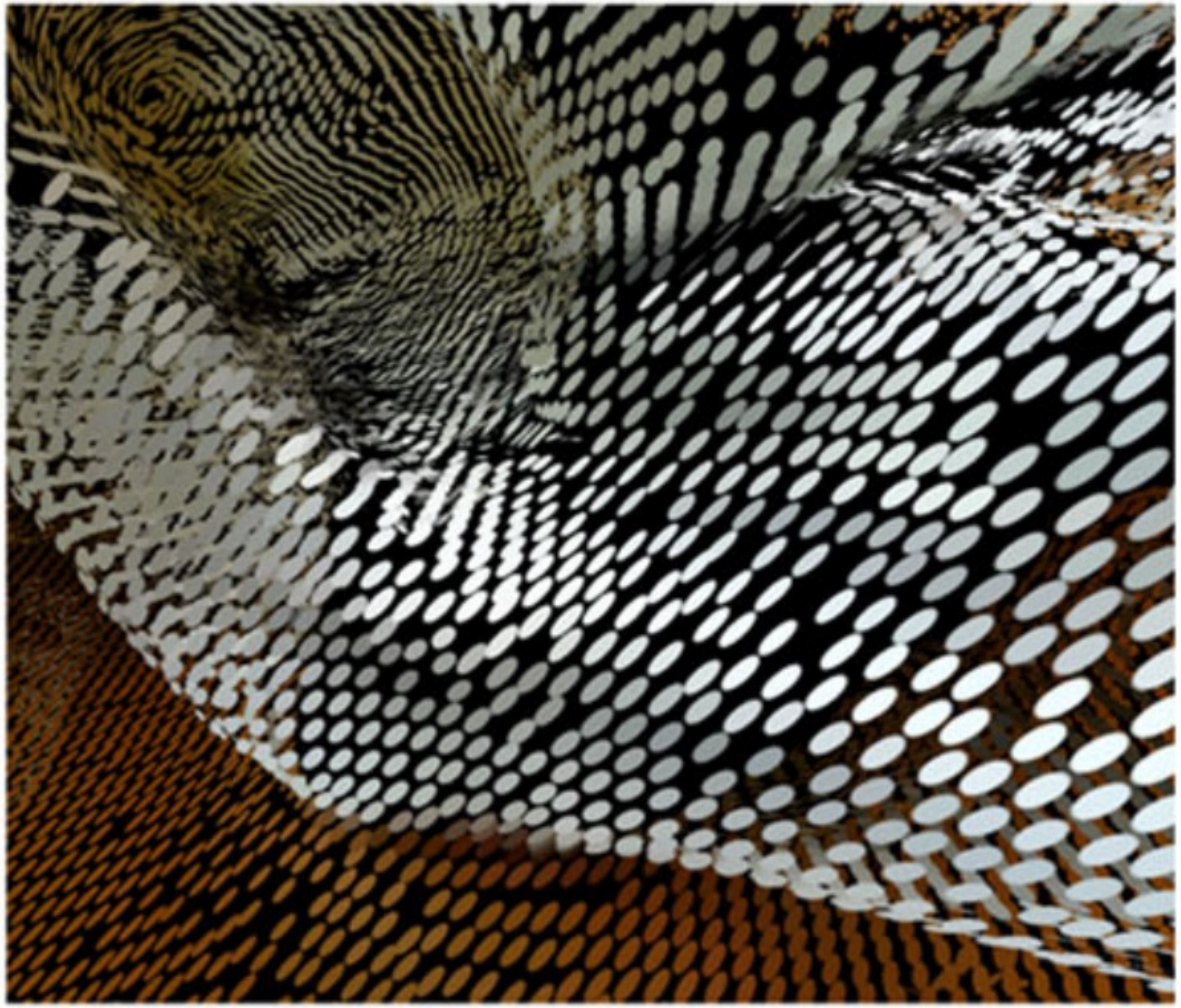
- Small changes:
- Add tangents as parameters
  - 2 float3 per point: more bandwidth
  - Calculate in real-time

```
in Vertex
{
    vec4 color;
    vec4 normal;
    vec4 right;
    vec4 up;
} vertex[];
```

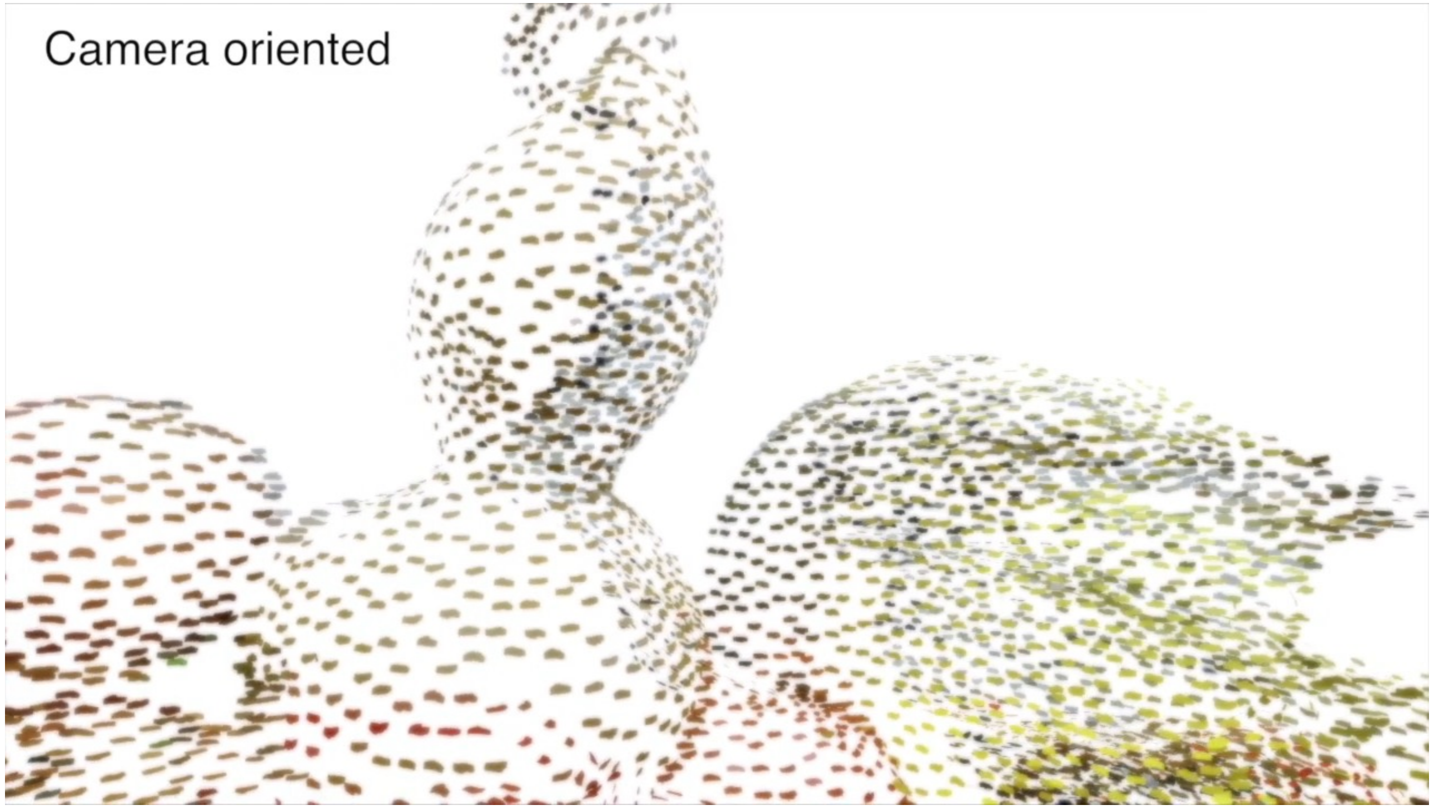
```
//Householder formula for tangent vectors
float n = sqrt(pow(nx,2) + pow(ny,2) + pow(nz,2));
float h1 = max( nx - n , nx + n );
float h2 = ny;
float h3 = nz;
float h = sqrt(pow(h1,2) + pow(h2,2) + pow(h3,2));
right = vec3(-2*h1*h2/pow(h,2), 1 - 2*pow(h2,2)/pow(h,2), -2*h2*h3/pow(h,2));
up = vec3(-2*h1*h3/pow(h,2), -2*h2*h3/pow(h,2), 1 - 2*pow(h3,2)/pow(h,2));
```

Fragment shader

- If you add UV to each emitted vertex, it gets interpolated over quad.
- Useful to render circles instead of squares.
- Or any other shape you like.



Example of different tangent vectors



UNIVERSITY OF LEEDS