Noise

in OpenGL

Introduction

Noise

Introduction

- For realistic-looking objects, we need to simulate the imperfections of real surfaces. That includes things such as scratches, rust, dents, and erosion.
- And sometimes want to represent natural surfaces such as wood grain or natural phenomena such as clouds to be as realistic as possible without giving the impression of them being synthetic or exhibiting a repetitive pattern or structure

Introduction

- Most effects or patterns in nature exhibit a certain degree of randomness and non-linearity. Therefore, you might imagine that we could generate them by simply using random data.
- However, random data such as the kind that is generated from a pseudo-random number generator is not very useful in computer graphics.
- There are two main reasons:

Introduction

- First, we need data that is repeatable, so that the object will render in the same way during each frame of the animation. (We could achieve this by using an appropriate seed value for each frame, but that only solves half of the problem.)
- Second, in order to model most of these natural phenomena, we actually need data that is continuous, but still gives the appearance of randomness. Continuous data more accurately represents many of these natural materials and phenomena. Purely random data does not have this continuity property. Each value has no dependence on the previous value.

Introduction

- The ground breaking work of Ken Perlin,gave us the concept of **noise** (as it applies to computer graphics). His work defined noise as a function that has certain qualities such as the following:
  - It is a continuous function
  - It is repeatable (generates the same output from the same input)
  - It can be defined for any number of dimensions
  - It does not have any regular patterns and gives the appearance of randomness
- **Perlin noise** is the noise function originally defined by Ken Perlin (see http://mrl.nyu.edu/~perlin/doc/oscar.html) .

Introduction

- To use Perlin noise within a shader, we have the following three main choices:
  - We can use the built-in GLSL noise functions
  - We can create our own GLSL noise functions
  - We can use a texture map to store pre-computed noise data

Noise texture

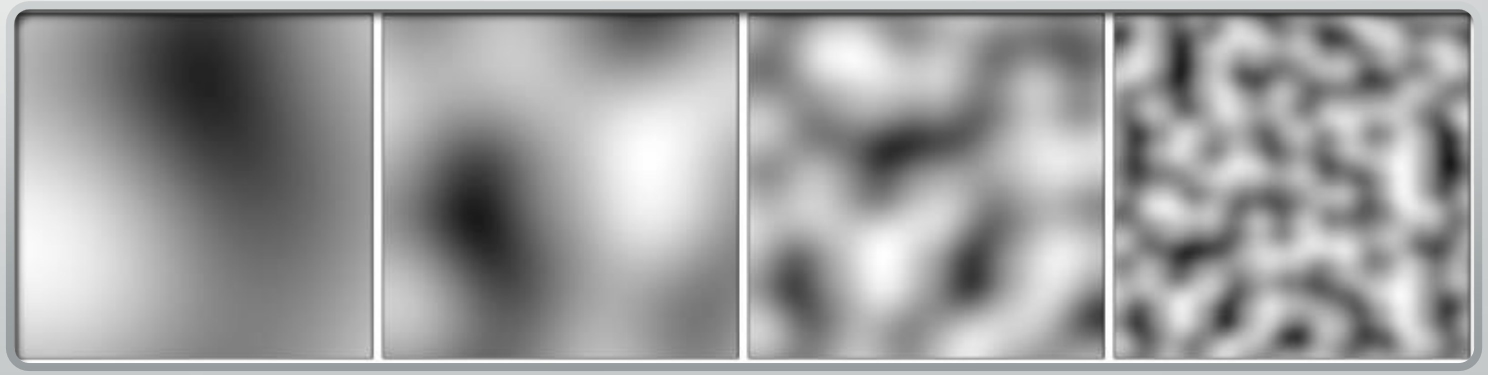
using GLM

Noise texture

- We'll use GLM to generate a 2D texture of noise values created using a **Perlin noise** generator. GLM can generate 2D, 3D, and 4D Perlin noise via the glm::perlin function.
- It is a common practice to use Perlin noise by summing the values of the noise function with increasing frequencies and decreasing amplitudes. Each frequency is commonly referred to as an **octave** (double the frequency).

Noise texture

- The results of the 2D Perlin noise function sampled at four different octaves. The sampling frequencies increase from left to right.
- The leftmost image in the following image is the function sampled at our base frequency, and each image to the right shows the function sampled at twice the frequency of the one to its left:



Noise texture

- In mathematical terms, for our 2D noise function P(x,y), then the previous images can be generated with the formula:

$$n_i(x, y) = P(2^i x, 2^i y)$$

i = 0, 1, 2 and 3 (from left to right)

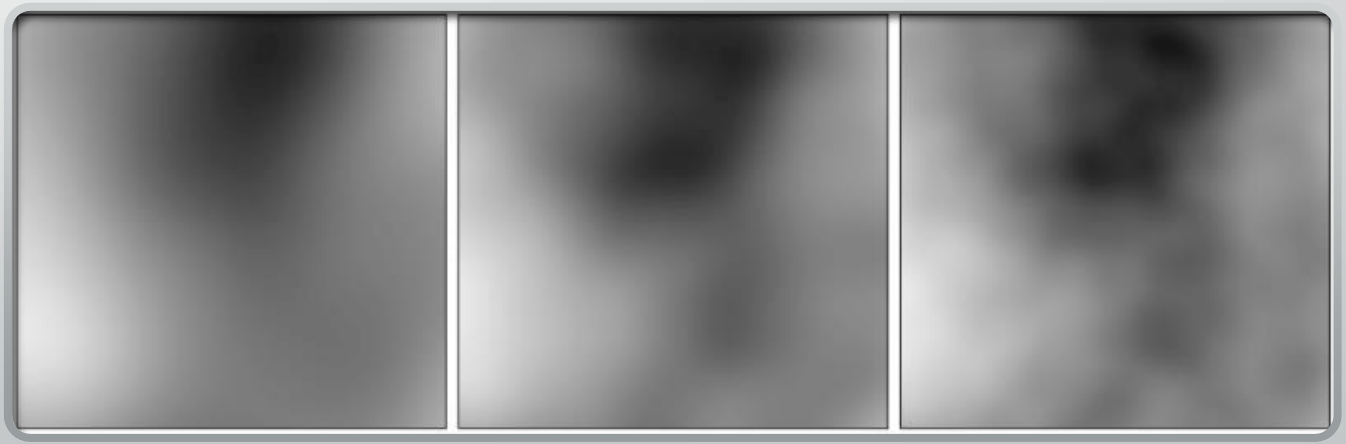
Noise texture

- The common practice is to sum octaves together to get the final result. We add each octave to the previous equation, scaling the amplitude down by some factor. So, for **N** octaves, we have the following sum :

$$n_i(x, y) = \sum_{i=0}^{N-1} \frac{P(2^i ax, 2^i ay)}{b^i}$$

a and b are tuneable constants

This shows the sum of 2, 3, and 4 octaves (left to right) with **a** = 1 and **b** = 2:



Noise texture

- Summed noise involving higher octaves will have more high-frequency variation than noise involving only lower octaves.
- However, it is possible to quickly reach frequencies that exceed the resolution of the buffer used to store the noise data, so care must be taken not to do unnecessary computation.
- In practice, it is both an art and a science. Feel free to experiment with the formula
- We'll store four noise values in a single 2D texture. We'll store Perlin noise with **one octave** in the first component (**red channel**), **two octaves** in the **green channel**, **three octaves** in the **blue channel**, and **four octaves** in the **alpha channel**.

Noise texture

- Generating the noise data (part1):

```
GLubyte *data = new GLubyte[ width * height * 4 ];

float xFactor = 1.0f / (width - 1);
float yFactor = 1.0f / (height - 1);

for( int row = 0; row < height; row++ ) {
 for( int col = 0 ; col < width; col++ ) {
  float x = xFactor * col;
  float y = yFactor * row;
  float sum = 0.0f;
  float freq = a;
  float scale = b;
```

Noise texture

- Generating the noise data (part2):

```
// Compute the sum for each octave
  for( int oct = 0; oct < 4; oct++ ) {
   glm::vec2 p(x * freq, y * freq);
   float val = glm::perlin(p) / scale;
   sum += val;
   float result = (sum + 1.0f)/ 2.0f;

   // Store in texture buffer
   data[((row * width + col) * 4) + oct] =
           (GLubyte) ( result * 255.0f );
   freq *= 2.0f;  // Double the frequency
   scale *= b;    // Next power of b
  }
 }
}
```

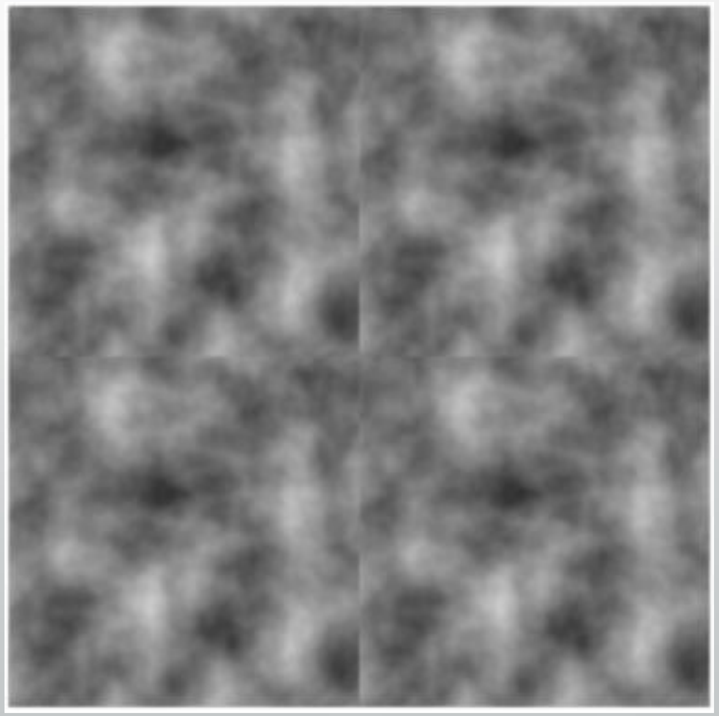
Noise texture

- Load data into an OpenGL texture:

```
GLuint texID;
glGenTextures(1, &texID);

glBindTexture(GL_TEXTURE_2D, texID);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, width,
height);
glTexSubImage2D(GL_TEXTURE_2D,0,0,0,width,height,
  GL_RGBA,GL_UNSIGNED_BYTE,data);

delete [] data;
```

Noise texture

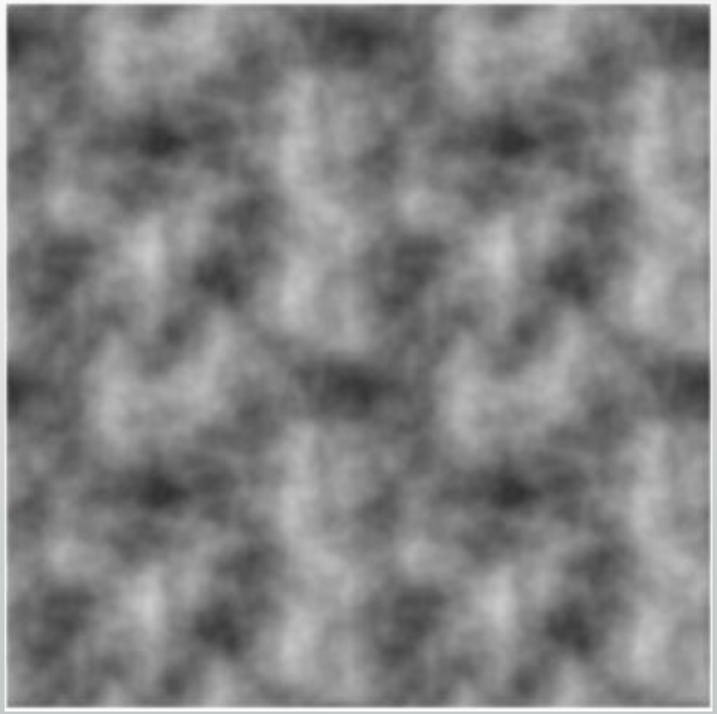
- It can be particularly useful to have a noise texture that tiles well. If we simply create a noise texture as a finite slice of noise values, then the values will not wrap smoothly across the boundaries of the texture.
- This can cause hard edges (seams) to appear in the rendered surface if the texture coordinates extend outside of the range of zero to one.



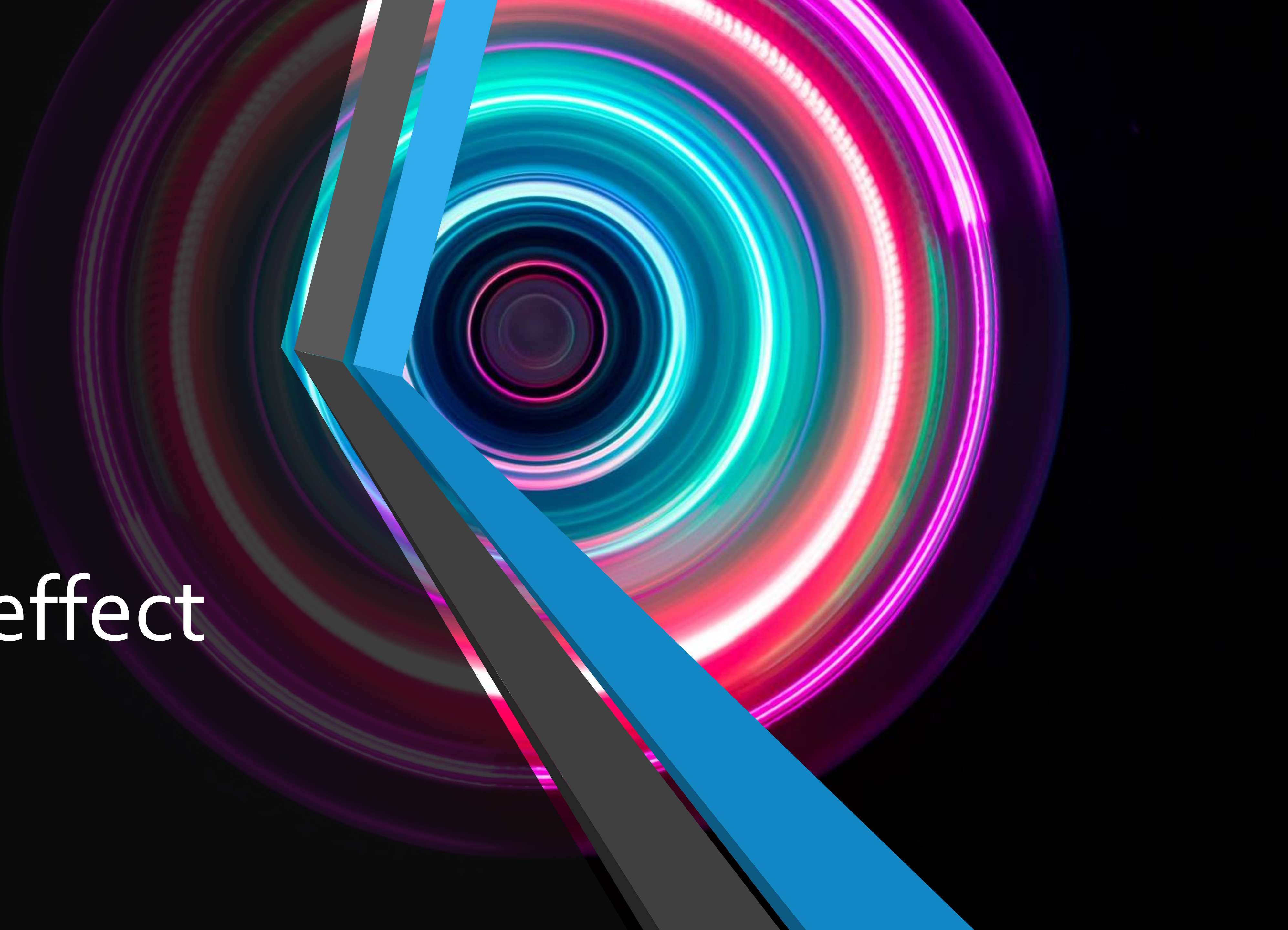
Noise texture

- GLM provides a periodic variant of Perlin noise that can be used to create a seamless noise texture.
- Within the innermost loop, instead of calling glm::perlin, we call an overloaded function that provides periodic Perlin noise:

```
val = glm::perlin(p, glm::vec2(freq)) / scale;
```



Doodle here…



Cloud effect

with Perlin noise

Cloud effect

- To create a texture that resembles a sky with clouds, we can use the noise values as a blending factor between the sky colour and the cloud colour.
- As clouds usually have large-scale structure, it makes sense to use low-octave noise. However, the large-scale structure often has higher frequency variations, so some contribution from higher octave noise may be desired.
- To create this effect, we take the cosine of the noise value and use the result as the blending factor between the cloud colour.

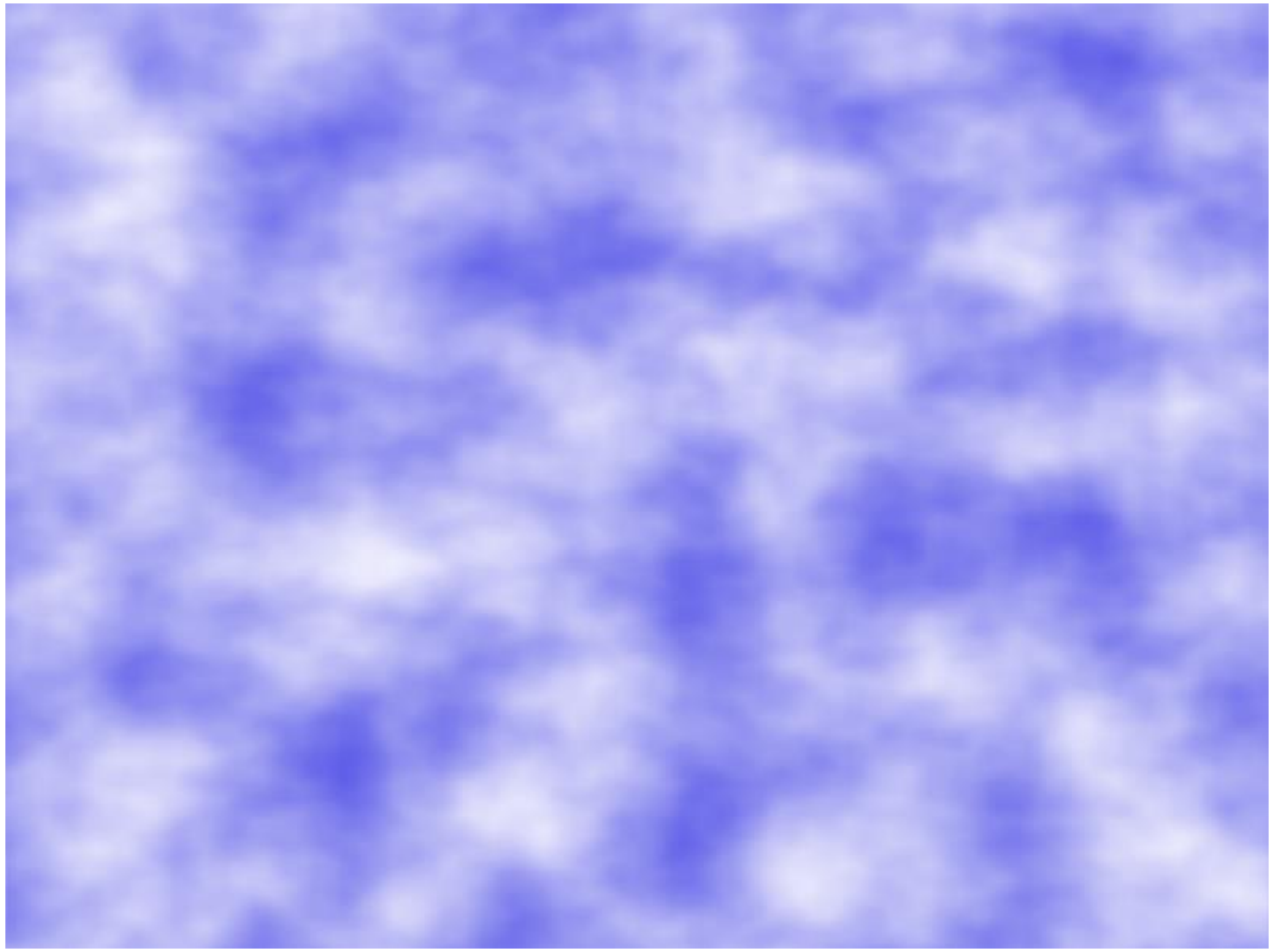
Cloud effect

- Set up your program to generate a seamless noise texture and make it available to the shaders through the NoiseTex uniform sampler variable.
- There are two uniforms in the fragment shader that can be assigned from the OpenGL program:
  - SkyColour: The background sky colour
  - Cloud colour: The colour of the clouds
- Set up your vertex shader to pass the texture coordinates to the fragment shader via the TexCoord variable

Cloud effect

- In the fragment shader:

```
#define PI 3.14159265

layout( binding=0 ) uniform sampler2D NoiseTex;

uniform vec4 SkyColor = vec4( 0.3, 0.3, 0.9, 1.0 );
uniform vec4 CloudColor = vec4( 1.0, 1.0, 1.0, 1.0 );

in vec2 TexCoord;

layout ( location = 0 ) out vec4 FragColor;

void main() {
  vec4 noise = texture(NoiseTex, TexCoord);
  float t = (cos( noise.g * PI ) + 1.0) / 2.0;
  vec4 color = mix( SkyColor, CloudColor, t );
  FragColor = vec4( color.rgb , 1.0 );
}
```



Cloud effect

Wood grain effect

with Perlin noise

Wood grain effect

- To create the look of wood, we can start by creating a virtual "log" with perfectly cylindrical growth rings. Then, we'll take a slice of the log and perturb the growth rings using noise from our noise texture.
- The virtual log is aligned with the **y** axis, and extends infinitely in all directions. The growth rings are aligned with integer distances from the **y** axis.
- Each ring is given a darker colour, with a lighter colour in-between rings. Each growth ring spans a narrow distance around the integer distances

Wood grain effect

- To take a "slice," we'll simply define a 2D region of the log's space based on the texture coordinates.
- Initially, the texture coordinates define a square region, with coordinates ranging from zero to one. We'll assume that the region is aligned with the x-y plane, so that the **s** coordinate corresponds to **x**, the **t** coordinate corresponds to **y**, and the value of **z** is zero.
- We can then transform this region, to create an arbitrary 2D slice. After defining the slice, we'll determine the colour based on the distance from the **y** axis. However, before doing so, we'll perturb that distance based on a value from the noise texture. The result has a general look that is similar to real wood.

Wood grain effect implementation

- Set up your program to generate a noise texture and make it available to the shaders through the uniform variable NoiseTex.
- There are three uniforms in the fragment shader that can be assigned from the OpenGL program:
  - LightWoodColour: The lightest wood colour
  - DarkWoodColour: The darkest wood colour
  - Slice: A matrix that defines the slice of the virtual "log" and transforms the default region defined by the texture coordinates to some other arbitrary rectangular region

Wood grain effect implementation

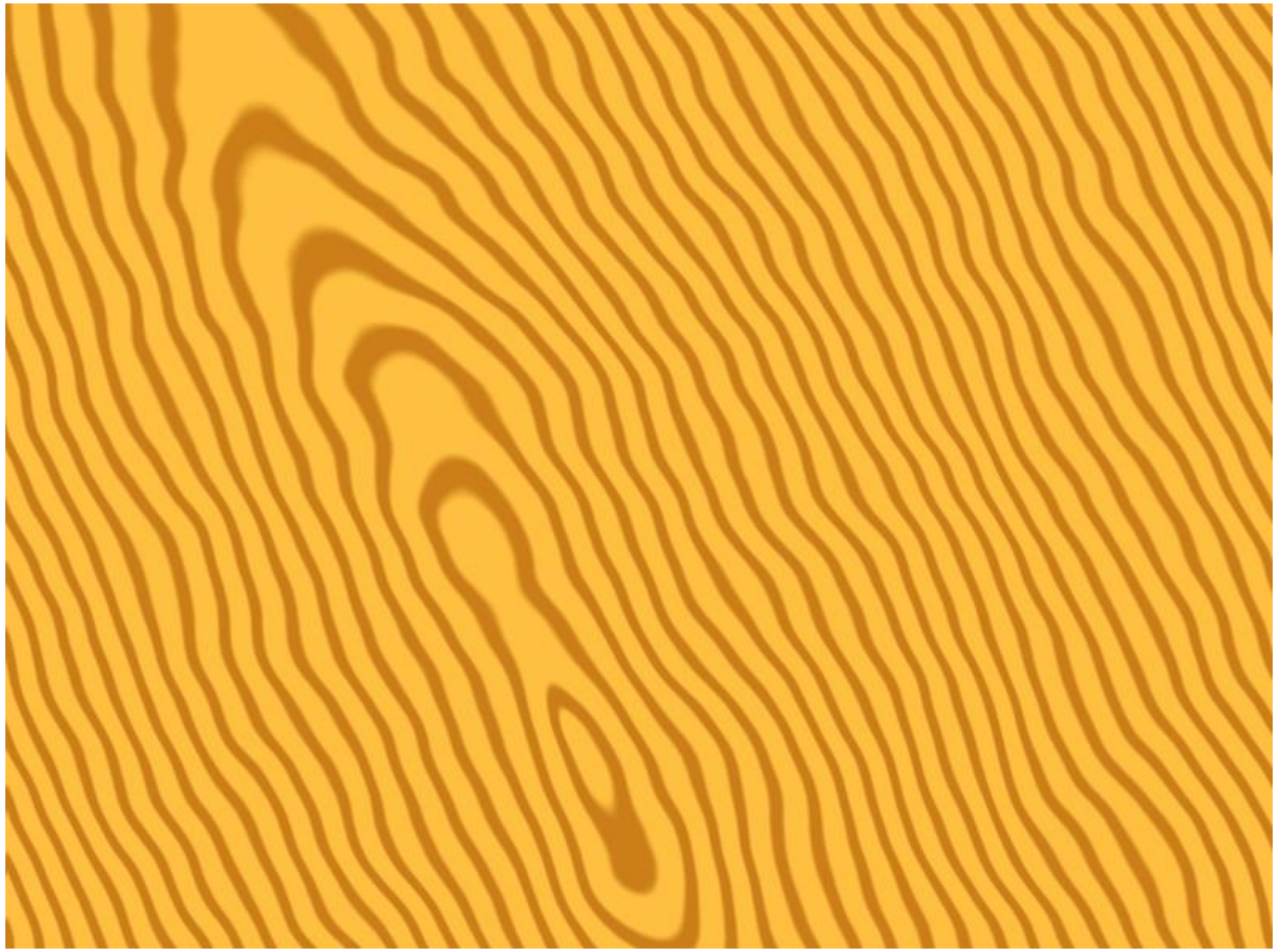
- Fragment shader (part1):

```
layout(binding=0) uniform sampler2D NoiseTex;

uniform vec4 DarkWoodColor = vec4( 0.8, 0.5, 0.1, 1.0 );
uniform vec4 LightWoodColor = vec4( 1.0, 0.75, 0.25, 1.0 );
uniform mat4 Slice;

in vec2 TexCoord;

layout ( location = 0 ) out vec4 FragColor;

void main() {
  // Transform the texture coordinates to define the
  // "slice" of the log.
  vec4 cyl = Slice * vec4( TexCoord.st, 0.0, 1.0 );
```

Wood grain effect implementation

- Fragment shader (part2):

```
//The distance from the log's y axis.
  float dist = length(cyl.xz);


  // Perturb the distance using the noise texture
  vec4 noise = texture(NoiseTex, TexCoord);
  dist += noise.b;


  // Determine the color as a mixture of the light and
  // dark wood colors.
  float t = 1.0 - abs( fract( dist ) * 2.0 - 1.0 );
  t = smoothstep( 0.2, 0.5, t );
  vec4 color = mix( DarkWoodColor, LightWoodColor, t );


  FragColor = vec4( color.rgb , 1.0 );
}
```



Wood grain effect

Disintegration effect

using Perlin noise

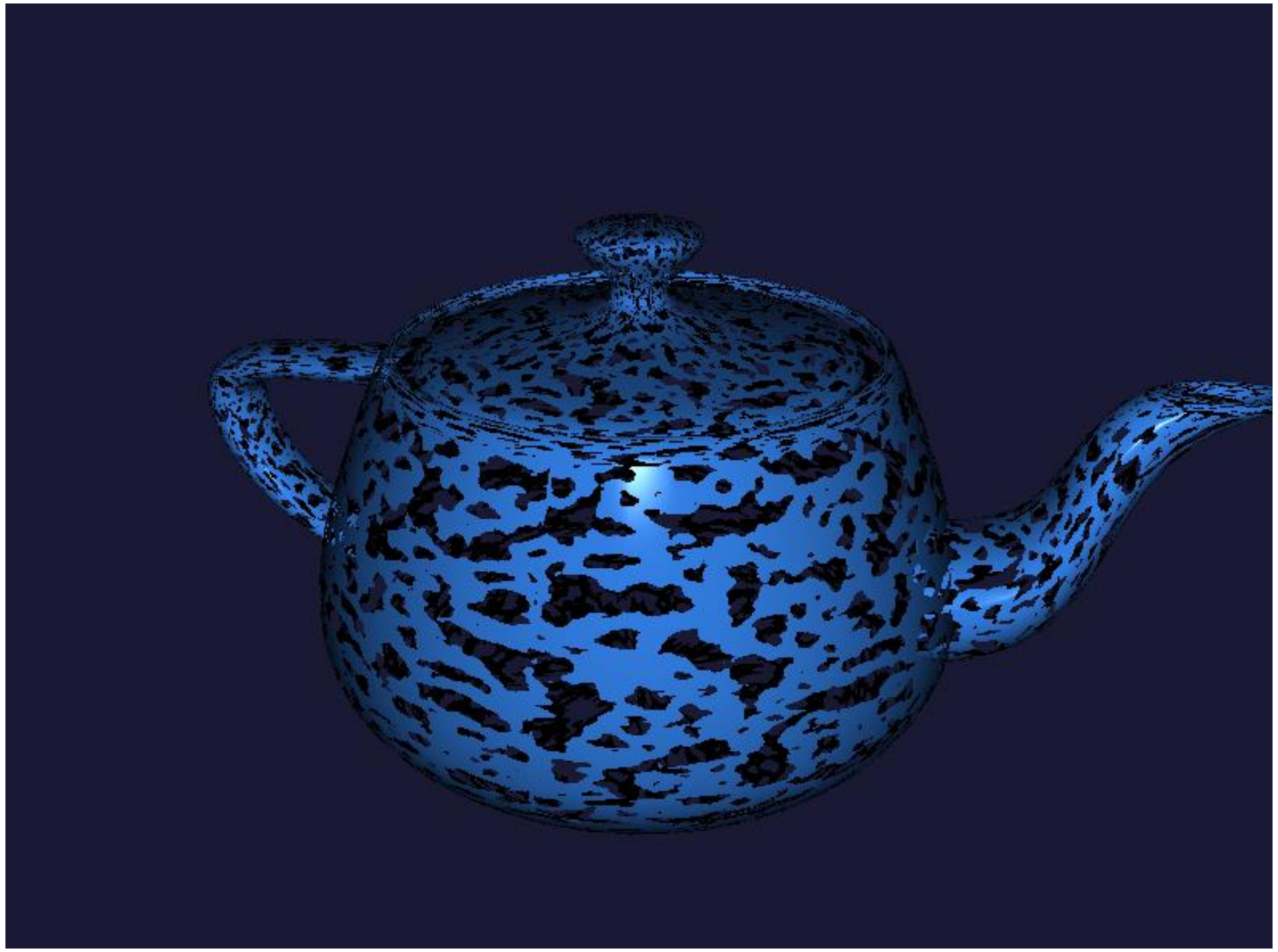
Disintegration effect

- It is straightforward to use the GLSL discard keyword in combination with noise to simulate erosion or decay. We can simply discard fragments that correspond to a noise value that is above or below a certain threshold.
- Set up your OpenGL program to provide position, normal, and texture coordinates to the shader. Make sure that you pass the texture coordinate along to the fragment shader.
- Create a seamless noise texture and place it in the appropriate texture channel.
- Declare the following uniforms:
  - NoiseTex: The noise texture
  - LowThreshold: Fragments are discarded if the noise value is below this value
  - HighThreshold: Fragments are discarded if the noise value is above this value

Disintegration effect

- Fragment shader:

```
layout(binding=0) uniform sampler2D NoiseTex;
in vec4 Position;
in vec3 Normal;
in vec2 TexCoord;
uniform float LowThreshold;
uniform float HighThreshold;
layout ( location = 0 ) out vec4 FragColor;

vec3 phongModel() {
 // Compute Phong shading model...
}
void main()
{
 // Get the noise value at TexCoord
 vec4 noise = texture( NoiseTex, TexCoord );

 // If the value is outside the threshold, discard
 if( noise.a < LowThreshold || noise.a > HighThreshold)
  discard;

 // Color the fragment using the shading model
 vec3 color = phongModel();
 FragColor = vec4( color , 1.0 );
}
```



Disintegration effect

Paint spatter effect

using Perlin noise

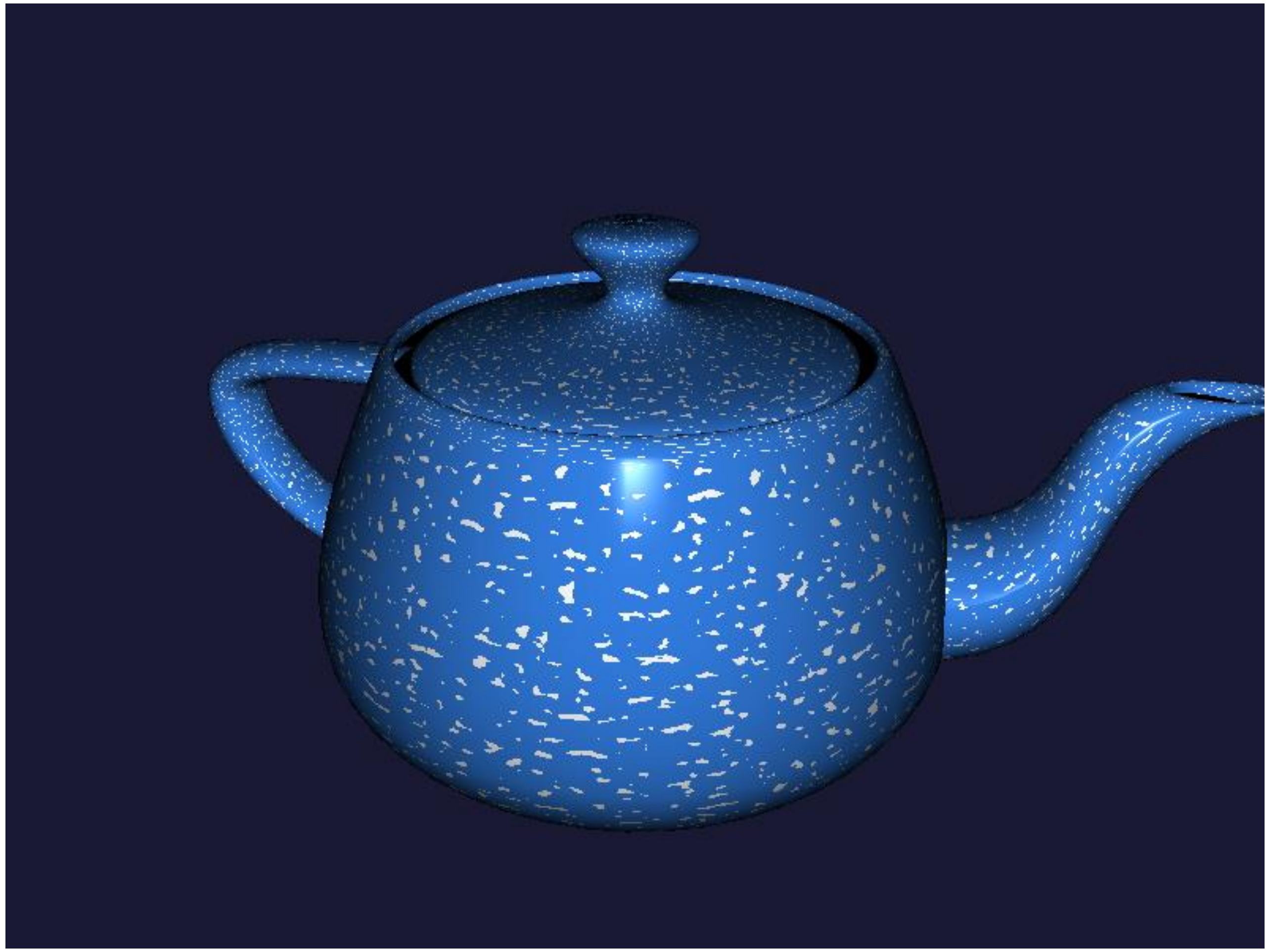
Paint spatter effect

- Using high-frequency noise, it is easy to create the effect of random spatters of paint on the surface of an object.
- We'll use either the base colour or paint colour as the diffuse reflectivity of the shading model. If the noise value is above a certain threshold, we'll use the paint colour; otherwise, we'll use the base colour of the object.
- Create a couple of uniforms variables:
  - PaintColour: The colour of the paint spatters
  - Threshold: The minimum noise value where a spatter will appear
- Create a noise texture with high-frequency noise.

Paint spatter effect

- Fragment shader:

```
// Uniforms for the Phong shading model
...
// The noise texture
layout(binding=0) uniform sampler2D NoiseTex;
// Input from the vertex shader
in vec4 Position;
in vec3 Normal;
in vec2 TexCoord;

// The paint-spatter uniforms
uniform vec3 PaintColor = vec3(1.0);
uniform float Threshold = 0.65;

layout ( location = 0 ) out vec4 FragColor;

vec3 phongModel(vec3 kd) {
  // Evaluate the Phong shading model
}

void main() {
  vec4 noise = texture( NoiseTex, TexCoord );
  vec3 color = Material.Kd;
  if( noise.g> Threshold ) color = PaintColor;
  FragColor = vec4( phongModel(color) , 1.0 );
}
```



Paint spatter effect

Night vision effect

using Perlin noise

Night vision effect

- Noise can be useful to simulate static or other kinds of electronic interference effects.
- We'll create the look of night-vision goggles with some noise thrown in to simulate some random static in the signal. We’ll also outline the scene in the classic **binocular** view.
- We'll apply the night-vision effect as a second pass to the rendered scene. The first pass will render the scene to a texture and the second pass will apply the night-vision effect.

Night vision effect

- Create a **framebuffer object** (**FBO**) for the first pass. Attach a texture to the first color attachment of the FBO.
- Create a set of uniforms:
  - Width: The width of the viewport in pixels
  - Height: The height of the viewport in pixels
  - Radius: The radius of each circle in the **binocular** effect (in pixels)
  - RenderTex: The texture containing the render from the first pass
  - NoiseTex: The noise texture
  - RenderPass: The subroutine uniform used to select the functionality for each pass

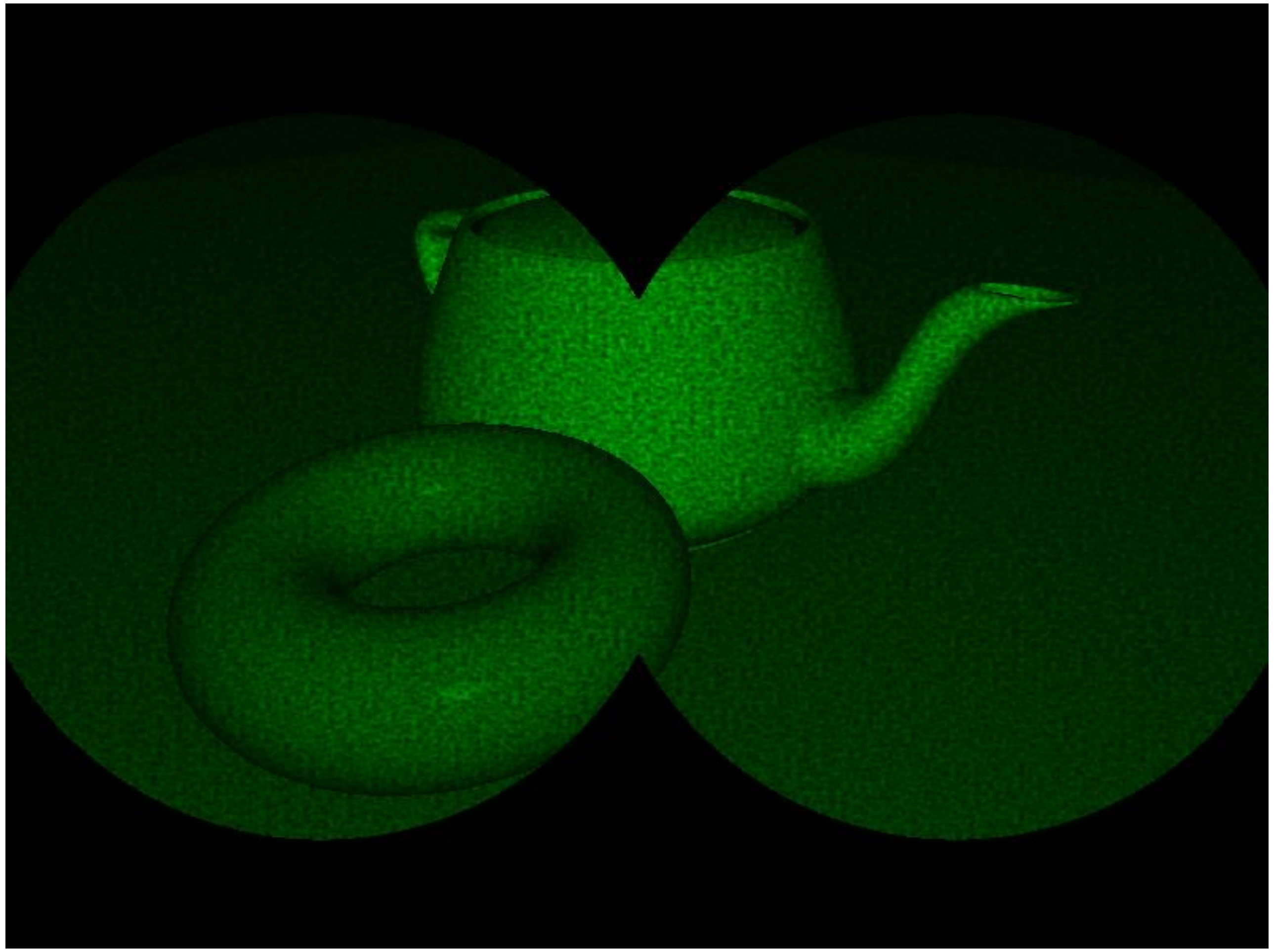
Night vision effect

- Fragment shader:

```
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;

uniform int Width;
uniform int Height;
uniform float Radius;
layout(binding=0) uniform sampler2D RenderTex;
layout(binding=1) uniform sampler2D NoiseTex;

subroutine vec4 RenderPassType();
subroutine uniform RenderPassType RenderPass;

// Define any uniforms needed for the shading model.

layout( location = 0 ) out vec4 FragColor;

vec3 phongModel( vec3 pos, vec3 norm ) {
  // Compute the Phong shading model
}
// Returns the relative luminance of the colour value
float luminance( vec3 colour ) {
  return dot( colour.rgb, vec3(0.2126, 0.7152, 0.0722) );
}
```

Night vision effect

```
subroutine (RenderPassType)
vec4 pass1() {
 return vec4(phongModel( Position, Normal ),1.0);
}

subroutine( RenderPassType )
vec4 pass2() {
 vec4 noise = texture(NoiseTex, TexCoord);
 vec4 color = texture(RenderTex, TexCoord);
 float green = luminance( color.rgb );

 float dist1 = length(gl_FragCoord.xy -
    vec2(Width*0.25, Height*0.5));
    float dist2 = length(gl_FragCoord.xy -
    vec2(3.0*Width*0.25, Height*0.5));
 if( dist1 > Radius && dist2 > Radius ) green = 0.0;

 return vec4(0.0, green * clamp(noise.a + 0.25, 0.0, 1.0),
    0.0 ,1.0);
}

void main() {
 // This will call either pass1() or pass2()
 FragColor = RenderPass();
}
```



Night vision effect

Useful links

- Noise and turbulence: https://mrl.cs.nyu.edu/~perlin/doc/oscar.html
- The book of shaders, noise: https://thebookofshaders.com/11/
- To read – Chapter 6 Thinking outside the box: Non stock shaders (OpenGL Superbible – see link on the DLE)
- To read: Using noise in shaders (OpenGL 4 Shading Language Cookbook).
- Noise (khronos.org): https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/noise.xhtml
- GLM: https://github.com/g-truc/glm
- Cloud generation: http://vterrain.org/Atmosphere/Clouds/
- Fract (Khronos): https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/fract.xhtml