Image Processing Techniques

in OpenGL

Introduction

Image Processing Techniques

Introduction

- The ability to render to a texture, combined with the power of the fragment shader, opens up a huge range of possibilities.
- We can implement image processing techniques such as **brightness**, **contrast**, **saturation**, and **sharpness** by applying an additional process in the fragment shader prior to output.
- We can apply **convolution filters** such as edge detection, smoothing (blur), or sharpening.
- **Deferred shading** involves rendering additional information to textures beyond the traditional colour information and then, in a subsequent pass, further processing that information to produce the final rendered image.

Introduction

- This techniques work directly with the pixels in a framebuffer and typically involve multiple passes
- An initial pass produces the pixel data and subsequent passes apply effects or further processes those pixels
- To implement this, we often make use of the ability provided in OpenGL for rendering directly to a texture, see the Lab session from last week where we rendered to a texture

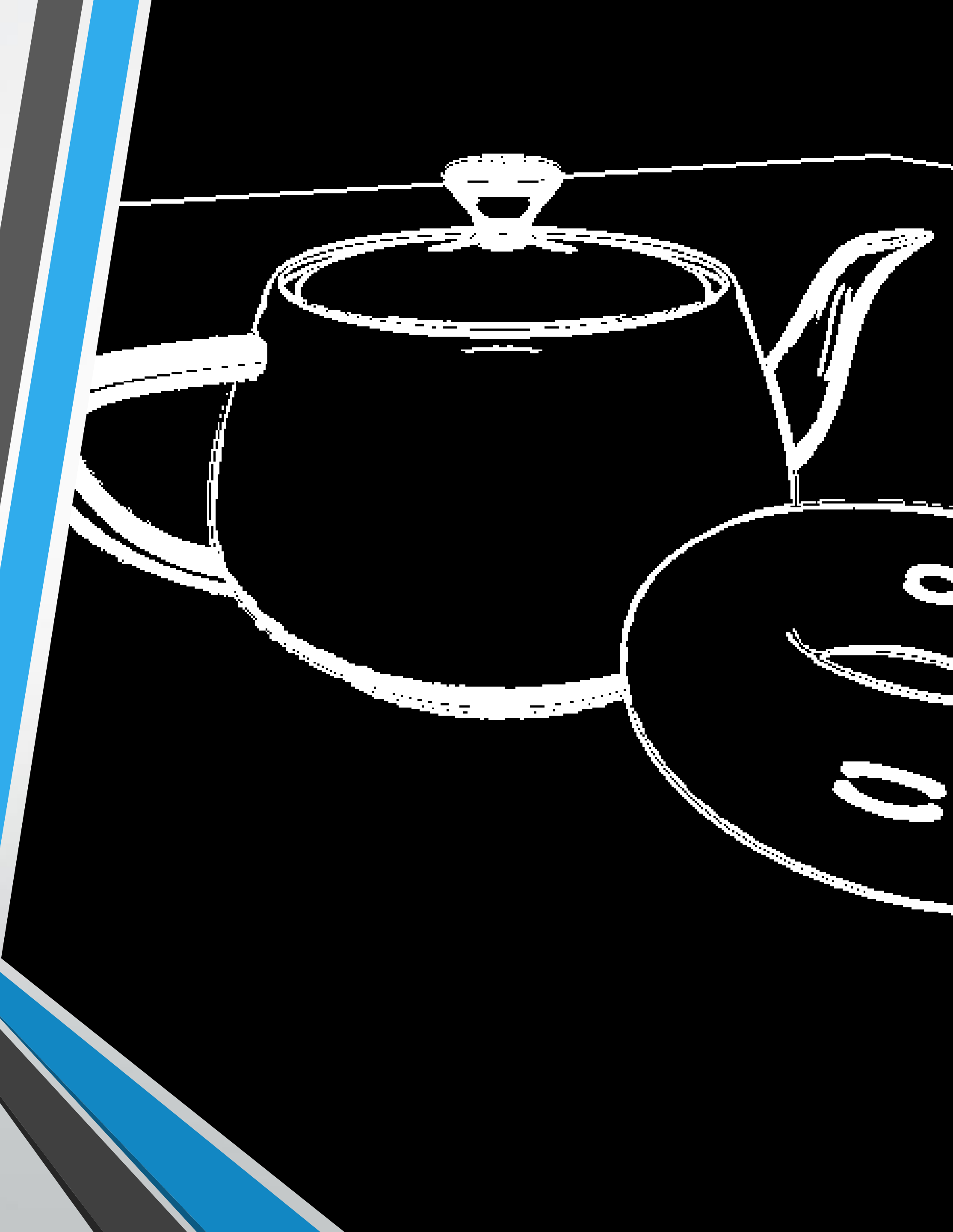
Edge detection filter

In OpenGL

Edge detection filter

Edge detection: is an image processing technique that identifies regions where there is a significant change in the brightness of the image.

- It provides a way to detect the boundaries of objects and changes in the topology of the surface.



Edge detection filter

- A **convolution filter** is a matrix that defines how to transform a pixel by replacing it with the sum of the products between the values of nearby pixels and a set of pre-determined weights.
- The values of the pixels could represent grayscale intensity or the value of one of the RGB components.
- We'll use a two-pass algorithm
  - In the first pass, we'll render the image to a texture
  - in the second pass, we'll apply the filter by reading from the texture and send the filtered results to the screen.

Edge detection filter

- One of the simplest technique for edge detection is **Sobel operator**.
- The Sobel operator is designed to approximate the gradient of the image intensity at each pixel. It does so by applying two 3 x 3 filters.
- The results of the two are the vertical and horizontal components of the gradient.
- We can then use the magnitude of the gradient as our edge trigger. When the magnitude of the gradient is above a certain threshold, we assume that the pixel is on an edge.

Edge detection filter

- The 3 x3 filter kernels used by **Sobel operator**:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$g = \sqrt{s_x^2 + s_y^2}$$

- If g is above a certain threshold, we consider the pixel to be an edge pixel and we highlight it in the resulting image.

Edge detection filter implementation

1. Set up a framebuffer object
2. Connect the FBO colour to a texture object
3. First pass
   - Render the scene to this texture
4. Second pass
   - Render a single quad that covers the entire screen
   - We retrieve the values of the eight neighbouring pixels of the texture and compute their brightness.

Edge detection filter implementation

- Set up a framebuffer object that has the same dimensions as the main window.
- Connect the first colour attachment of the FBO to a texture object in texture unit zero.
- During the first pass, we'll render directly to this texture.
- Make sure that the mag and min filters for this texture are set to GL_NEAREST.

Edge detection filter implementation

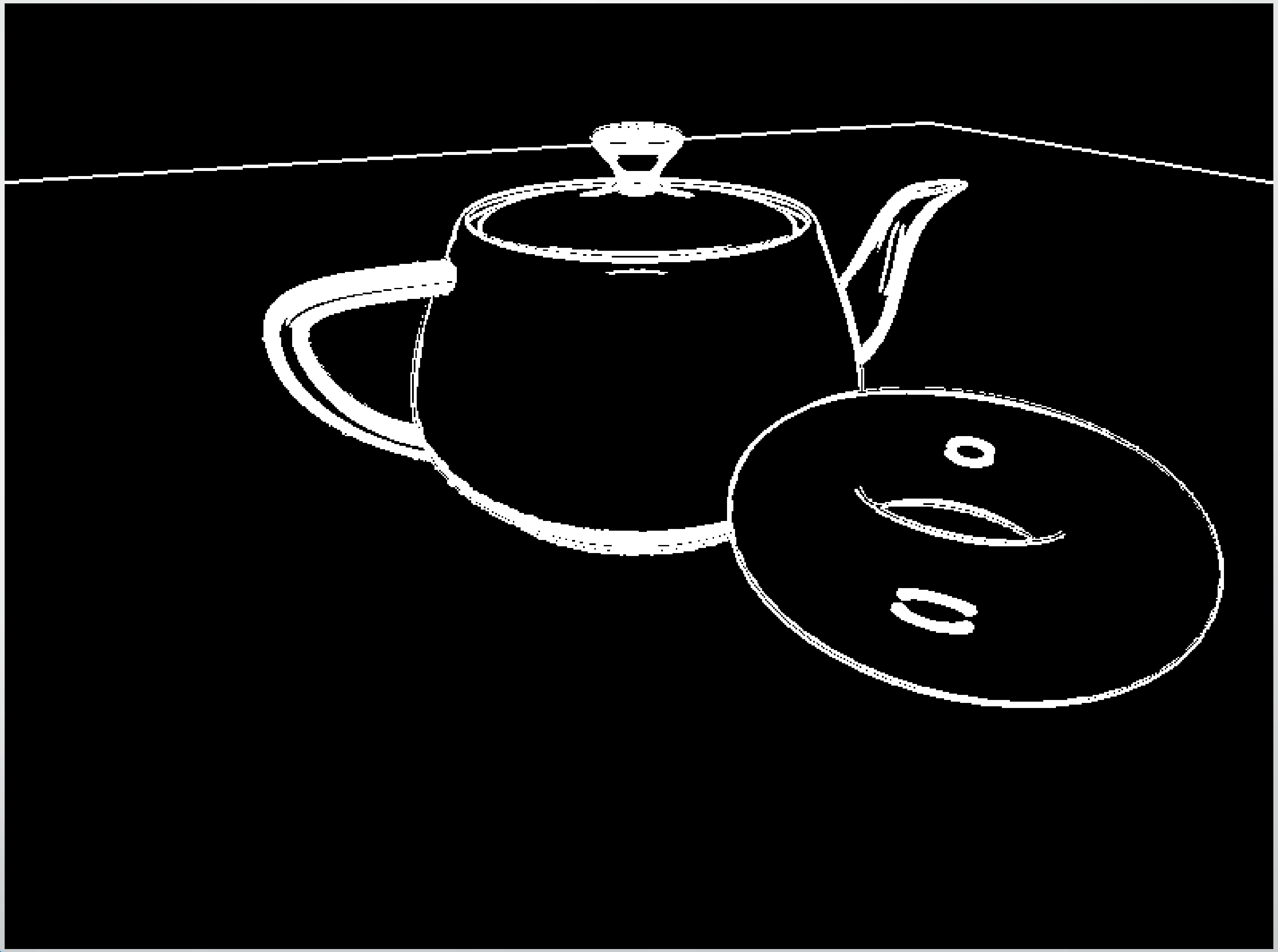
First pass renders all of the scene’s geometry sending the output to a texture:

- Select the FBO and clear the colour and depth buffers
- Set the pass uniform to 1
- Set up the model, view and projection matrices and draw the scene

Edge detection filter implementation

- Deselect the FBO (revert to the default framebuffer) and clear the color/depth buffer
- Set the pass uniform to 2
- Set the model, view, and projection matrices to the identity matrix
- Draw a single quad (or two triangles) that fills the screen (-1 to +1 in x and y), with texture coordinates that range from 0 to 1 in each dimension.

Edge detection filter implementation



Gaussian blur filter

In OpenGL

Gaussian blur filter

- Blur filter: mixes the colour of a pixel with that of nearby pixels using a weighted sum.
- The weights typically decrease with the distance from the pixel (in 2D screen space).

Gaussian blur

- uses the two-dimensional Gaussian function to weight the contributions of the nearby pixels
- Compute the weighted sum of all pixels in the image scaled by the value of the Gaussian function at that pixel

Gaussian blur filter

- A blur filter can be useful in many different situations, especially in games but one of its main goals is to reduce the amount of noise in the image
- The basic idea of any blur filter is to mix the colour of a pixel with that of nearby pixels using a weighted sum.
- The weights typically decrease with the distance from the pixel (in 2D screen space) so that pixels that are far away contribute less than those closer to the pixel being blurred.
- A **Gaussian blur** uses the two-dimensional Gaussian function to weight the contributions of the nearby pixels

Gaussian blur filter

- Two problems with this approach:
  - Too slow for real-time use
  - The weights must sum to one

- Compromise: using a 9 x 9 Gaussian blur filter
- For an image 800 x 600:

  800 * 600 * 81 = 38,880,000 (texture fetches)

Gaussian blur filter

- We can limit the number of pixels that we blur with a given pixel (instead of the entire image), and we can normalise the values of the Gaussian function.
- using a 9 x 9 Gaussian blur filter, we'll only compute the contributions of the 81 pixels in the neighbourhood of the pixel being blurred.
- 81 pixels texture look up, for each pixel that we want to blur which is quite a lot
- For an image 800 x 600:

  800 * 600 * 81 = 38,880,000 (texture fetches)

Gaussian blur filter

- We can improve this by doing 2 passes, decomposing a two-dimensional equation to a one dimension

$$G(x, y) = G(x) \cdot G(y)$$

- we can compute the sum over j (the vertical sum) and store the results in a temporary texture.
- In the second pass, we compute the sum over i (the horizontal sum) using the results from the previous pass.

$$k = \sum_{i=-4}^{4} G(i)$$

Gaussian blur filter implementation

- We implement this technique using 3 passes and 2 textures

1. First pass: render the entire scene to a texture.
2. Second pass: apply the vertical sum
   - a distance of four pixels in each vertical direction
3. Third pass: apply the horizontal sum
   - a distance of four pixels in each horizontal direction
   - incorporate the sums produced in the second pass into our overall weighted sum.

The output colour goes to the default framebuffer to make up the final result

Gaussian blur filter implementation

- Set up two framebuffer objects (refer to the previous implementation of blur) and two corresponding textures.
- The first FBO should have a depth buffer because it will be used for the first pass.
- The second FBO need not have a depth buffer because, in the second and third passes, we'll only render a single screen-filling quad in order to execute the fragment shader once for each pixel.

Gaussian blur filter implementation

First pass, renders the scene to a texture using the Blinn-Phong reflection model:

- Select the render framebuffer, enable depth test and clear the colour and depth buffers
- Set the pass uniform to 1
- Draw the scene

Gaussian blur filter implementation

Second pass, applies the weighted vertical sum of the Gaussian blur operation, and stores the results in yet another texture. We read pixels from the texture created in the first pass, offset in the vertical direction by the amounts in the PixelOffset array.

We sum in both directions at the same time, a distance of four pixels in each vertical direction:

- Select the intermediate framebuffer, disable the depth test, and clear the colour buffer
- Set the pass uniform to 2
- Set the view, projection, and model matrices to the identity matrix
- Bind the texture from pass #1 to texture unit zero
- Draw a full screen quad

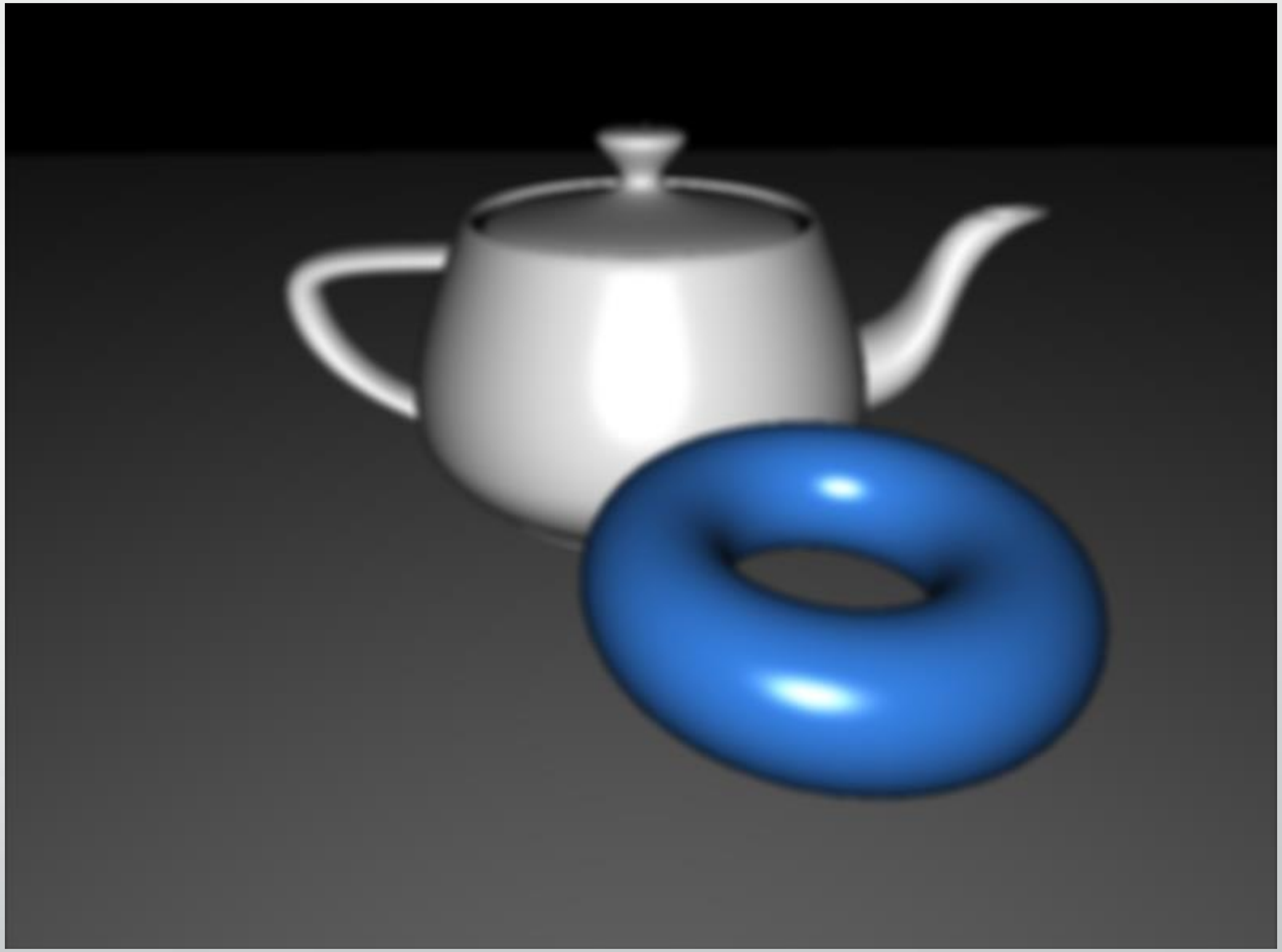
Gaussian blur filter implementation

Third pass, we accumulate the weighted, horizontal sum using the texture from the second pass. By doing so, we are incorporating the sums produced in the second pass into our overall weighted sum.

We are creating a sum over a 9 x 9 pixel area around the destination pixel, the output colour goes to the default framebuffer to make up the final result:

- Deselect the framebuffer (revert to the default), and clear the colour buffer
- Set the pass uniform to 3
- Bind the texture from pass #2 to texture unit zero
- Draw a full screen quad

Gaussian blur filter implementation



Doodle here…

HDR lighting

with tone mapping

HDR lighting

- OpenGL uses floating-point values for colour intensities, mapped to the floating-point range [0.0, 1.0].
- Potential solution: reduce the strength of light sources and ensure no area of fragments in the scene end up brighter than 1.
- Actual solution is to temporarily allow the values to exceed 1 and transform them back to the original range 0.0 and 1.0

HDR lighting

- OpenGL uses floating-point values for colour intensities, mapped to the floating-point range [0.0, 1.0].
- Real scenes, however, have a much wider range of luminance and anything that gets over 1 it gets clamped at 1.
- We end up with an image that is washed out and we're losing a significant amount of detail.
- One solution is to reduce the strength of light sources and ensure no area of fragments in the scene end up brighter than 1. This is really not a solution as we're forced to use unrealistic lighting parameters.
- The solution is to temporarily allow the values to exceed 1 and transform them back to the original range 0.0 and 1.0 as the final step without losing detail.

HDR lighting

- **High Dynamic Range rendering (HDR rendering) :** The process of computing the lighting/shading using a larger dynamic range

- **Tone mapping** is the process of taking a wide dynamic range of values and compressing them into a smaller range

- **Tone Mapping Operator (TMO)**: The mathematical function used to map from one dynamic range to a smaller range
  - **Local operator**: determines the new value for a given pixel by using its current value and perhaps the value of some nearby pixels.
  - **Global operator**: needs some information about the entire image in order to do its work.

HDR lighting

- We'll use the simple global operator described in the book Real Time Rendering. This operator uses the log-average luminance of all pixels in the image. The log-average is determined by taking the logarithm of the luminance and averaging those values, then converting back.

$$\bar{L}_w = e^{\frac{1}{N}\sum_{x,y}\ln(0.0001+L_w(x,y))}$$

- Lw(x, y) is the luminance of the pixel at (x, y).

- The 0.0001 term is included in order to avoid taking the logarithm of zero for black pixels

This log-average is then used as part of the tone mapping operator

$$L(x,y) = \frac{a}{\bar{L}_w} L_w(x,y)$$

HDR lighting

The log-average is then used as part of the tone mapping operator

$$L(x,y) = \frac{a}{\bar{L}_w} L_w(x,y)$$

a: is acting as the exposure level in a camera, with typical values of 0.18 to 0.72

In order to avoid compressing the dark and light values a bit too much, we'll use a modified version that has a maximum luminance (L_{white})

$$L_d(x,y) = \frac{L(x,y)\left(1 + \frac{L(x,y)}{L_{\text{white}}^2}\right)}{1 + L(x,y)}$$

HDR lighting

- We start with an RGB value, we can compute its luminance, but once we modify the luminance, it ill be difficult to modify the RGB components to reflect the new luminance without changing the hue or chromaticity (perceived colour independent of the brightness of the colour).
- We'll use a colour space called the **CIE XYZ** and a derived colour space called **CIE xyY space.** The Y component contains the luminance and the x and y components contain the chromaticity.

HDR lighting

The process involves converting from RGB to CIE XYZ, then converting to CIE xyY, modifying the luminance, and reversing the process to get back to RGB.

- Converting from RGB to CIE XYZ (and vice-versa) can be described as a transformation matrix .
- The conversion from XYZ to xyY, we'll use this:

$$x = \frac{X}{X+Y+Z} \qquad y = \frac{Y}{X+Y+Z}$$

- The conversion from xyY back to XYZ, we'll use this:

$$X = \frac{Y}{y} * x \qquad Z = \frac{Y}{y}(1 - x - y)$$

HDR lighting implementation

The general steps are the following:

- Render the scene to a high-resolution texture.
- Compute the log-average luminance (on the CPU).
- Render a screen-filling quad to execute the fragment shader for each screen pixel.

Fragment shader

- Read from the texture created in step 1
- Apply the tone mapping operator
- Send the results to the screen.

To get set up, create a high-res texture (using GL_RGB32F) attached to a framebuffer with a depth attachment. Set up your fragment shader with a uniform to select the pass. The vertex shader can simply pass through the position and normal in eye coordinates.

HDR lighting implementation

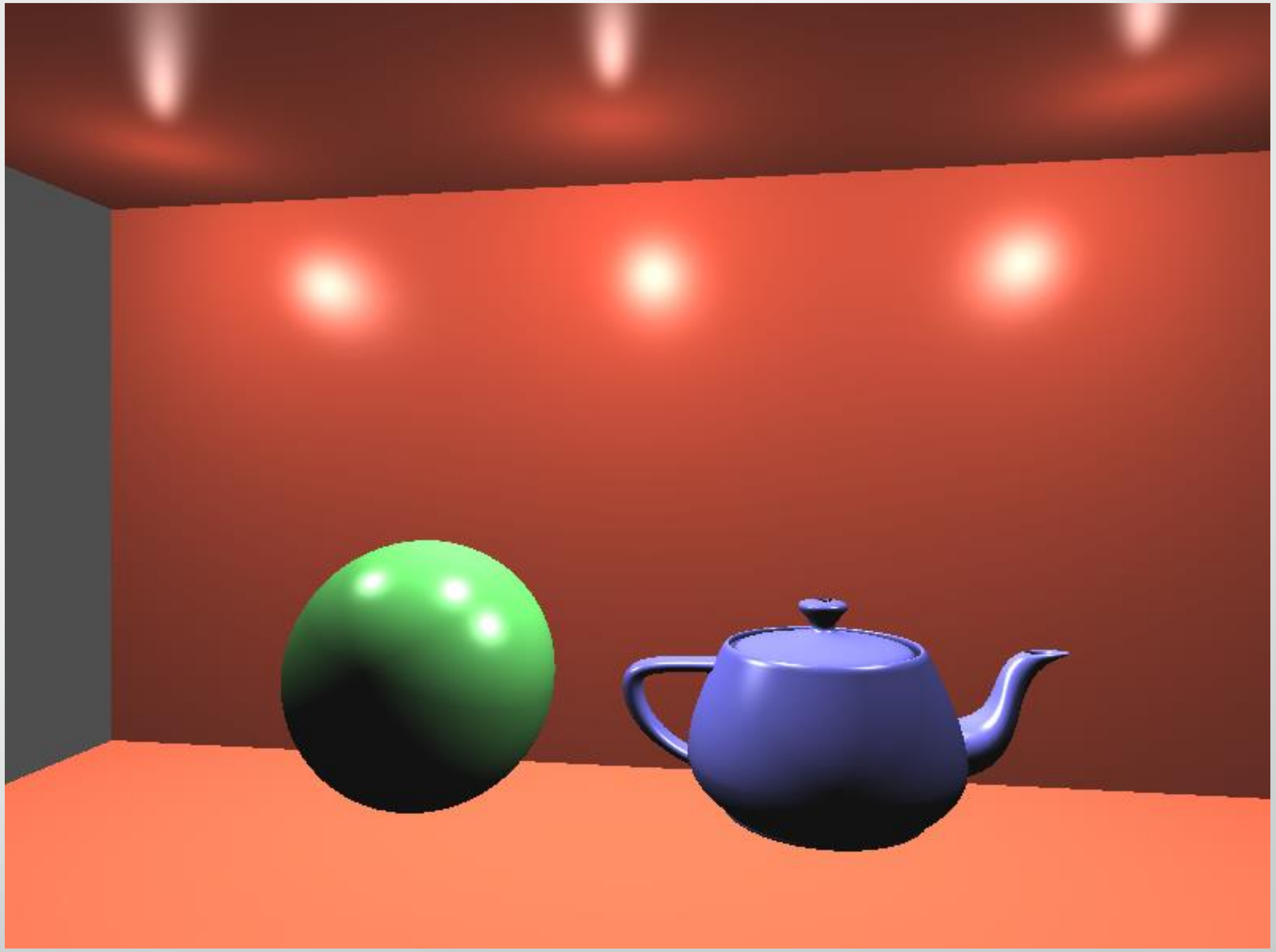
The general steps are the following:

- Render the scene to a high-resolution texture.
- Compute the log-average luminance (on the CPU).
- Render a screen-filling quad to execute the fragment shader for each screen pixel.

Fragment shader

- Read from the texture created in step 1
- Apply the tone mapping operator
- Send the results to the screen.

HDR lighting implementation



Bloom Effect

In OpenGL

Bloom effect

Bloom: visual effect where the bright parts of an image seem to have fringes that extend beyond the boundaries into the darker parts of the image.

- Sources of bright light bleed into other areas of the image due to the so-called **airy disc**

Process:

- determine which parts of the image are bright enough
- Extract the bright parts
- Blur
- Re-combining with the original image

Airy disc: diffraction pattern produced by light that passes through an aperture.

Bloom effect

Typically consists of five passes but can be summarised as four major steps:

1. In the first pass, we will render the scene to an **HDR texture**.
2. The second pass will extract the parts of the image that are brighter than a certain threshold value. We'll refer to this as the **bright-pass filter**.
3. The third and fourth passes will apply the **Gaussian blur** to the bright parts.
4. In the fifth pass, we'll apply **tone mapping** and add the tone-mapped result to the blurred bright-pass filter results.

Bloom effect implementation

- We'll use 2 framebuffer objects, each associated with a texture:
  - HdrTex – used for the original HDR render
  - BlurTex – used for the two passes of the Gaussian blur.
- A uniform variable LumThresh which represents the minimum luminance value used in the second pass. Any pixels greater than that value will be extracted and blurred in the following passes.
- Vertex shader make sure it passes through the position and normal in eye coordinates and the texture coordinates used for unpacking the textures.

Bloom effect implementation

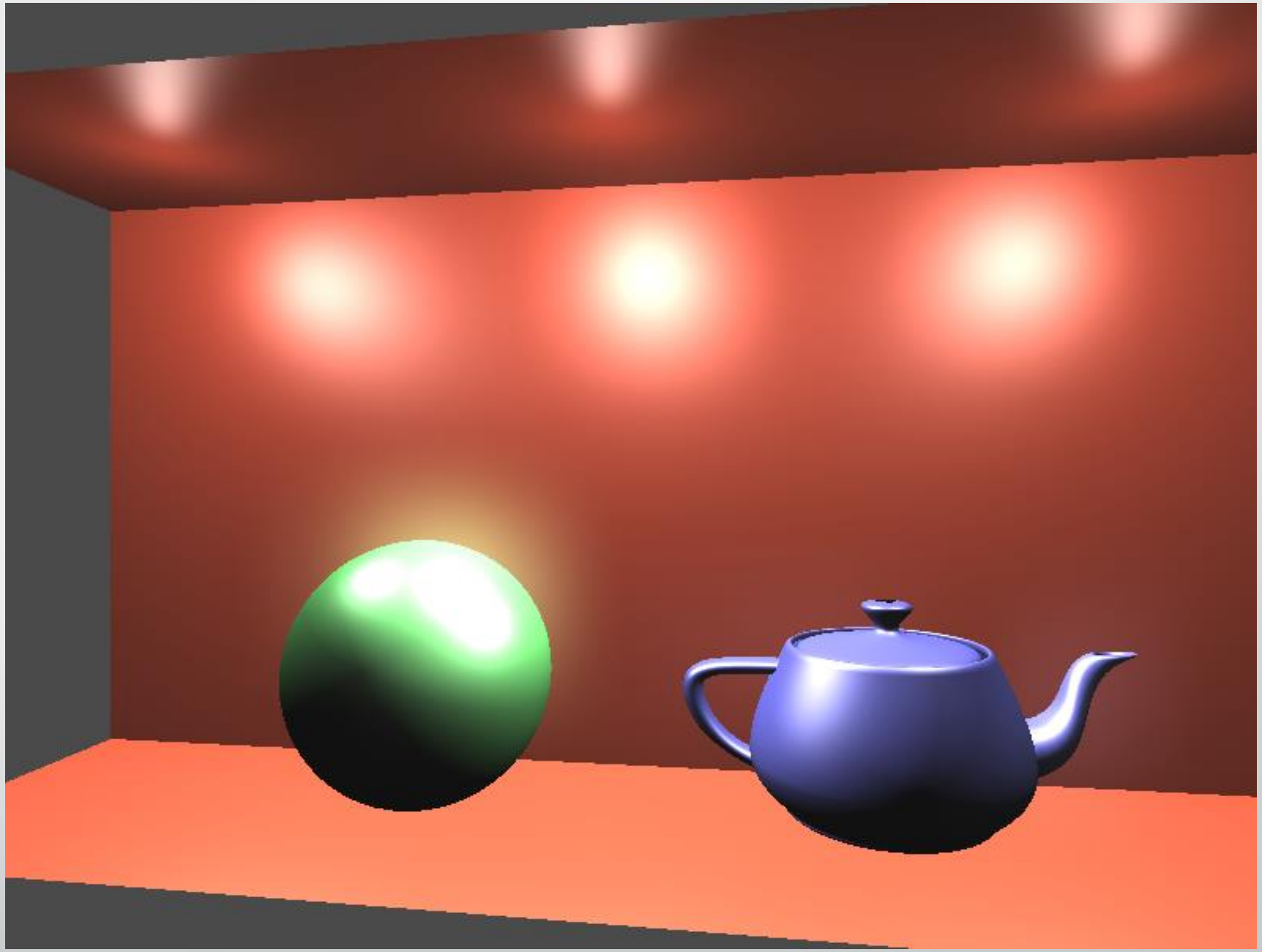
Implementation steps:

1. First pass:
   - Render the scene to the framebuffer with a high-res backing texture (look at **HDR rendering**).
2. Second pass:
   - Switch to a framebuffer containing a high-res texture that is smaller than the size of the full render (one-eighth the size).
   - Draw a fullscreen quad to initiate the fragment shader for each pixel in the fragment shader sample from the high-res texture
   - Write only those values that are larger than LumThresh, otherwise the pixel should be black (look at **edge detection** filter).
3. Third and fourth passes:
   - Apply the Gaussian blur to the results of the second pass. This can be done with a single framebuffer and two textures.
   - Ping-pong between them, reading from one and writing to the other. (see **Gaussian blur** implementation).

Bloom effect implementation

4. Fifth pass:
   - Switch to linear filtering from the texture that was produced in the fourth pass.
   - Switch to the default frame buffer (the screen).
   - Apply the tone-mapping operator from the Implementing **HDR lighting** with tone mapping recipe to the original image texture (HdrTex)
   - Combine the results with the blurred texture from step 3 (BlurTex). The linear filtering and magnification should provide an additional blur

Bloom effect implementation



Gamma correction

In OpenGL

Gamma correction

- There is a difference in the way we perceive pixel intensity as humans and how the monitors display that intensity
- There's a non linear mapping happening with monitors, but humans prefer to use a more linear mapping
- The idea is to apply an inverse of monitor's gamma value (typically 2.2) to the final output colour before displaying it.
- The inverse of the value is 1/22

Gamma correction

Taking a dark red colour (0.5, 0.0, 0.0)

We apply gamma correction:

$$(0.5, 0.0, 0.0)^{1/22} = (0.5, 0.0, 0.0)^{0.45} = (0.73, 0.0, 0.0)$$

The corrected colours are sent to the monitor and the resulting colour is:

$$(0.73, 0.0, 0.0)^{2.2} = (0.5, 0.0, 0.0)$$

The monitor displays the intended colour.

Gamma correction implementation

In the fragment shader:

```
…

vec3 colour = lightingModel( … );

float Gamma = 2.2f;
FragColor = vec4( pow( colour, vec3(1.0/Gamma) ), 1.0 );
```

Deferred shading

In OpenGL

Deferred shading

Deferred shading: technique that involves postponing (or deferring) the lighting/shading step to a second pass.

- **First pass**: we render the scene, but instead of evaluating the reflection model to determine a fragment colour, we simply store all of the geometry information (position, normal, texture coordinate, reflectivity, and so on) in an intermediate set of buffers, collectively called the **g-buffe**r (g for geometry).
- Second pass: we simply read from the g-buffer, evaluate the reflection model, and produce a final colour for each pixel.

Deferred shading

- Consider a pixel located in an area where two polygons overlap.
- The fragment shader may be executed once for each polygon that covers that pixel; however, the resulting colour of only one of the two executions will end up being the final colour for that pixel (assuming that blending is not enabled).
- The cycles spent in evaluating the reflection model for one of the two fragments are effectively wasted. With **deferred shading**, the evaluation of the reflection model is postponed until all the geometry has been processed, and the visible geometry is known at each pixel location.
- The reflection model is evaluated only once for each pixel on the screen. This allows us to do lighting in a more efficient fashion. For example, we could use even hundreds of light sources because we are only evaluating the lighting once per screen pixel.

Deferred shading implementation

- In the first pass we'll store the position, normal and diffuse colour (reflectivity) in the g-buffer
  - g-buffer will contain 3 textures for storing the data: PositionTex, NormalTex and ColourTex
  - For position and normal texture make sure you use GL_RGB32F internal format for high resolution capture
  - For reflectivity, you can use a GL_RGB8, as we don't need extra resolution
- In the second pass we'll evaluate the diffuse lighting model using the data stored in the g-buffer

Vertex shader will be similar to the other examples, passing the through the position and normal in eye coordinates and the texture coordinates used for unpacking the textures

Deferred shading implementation

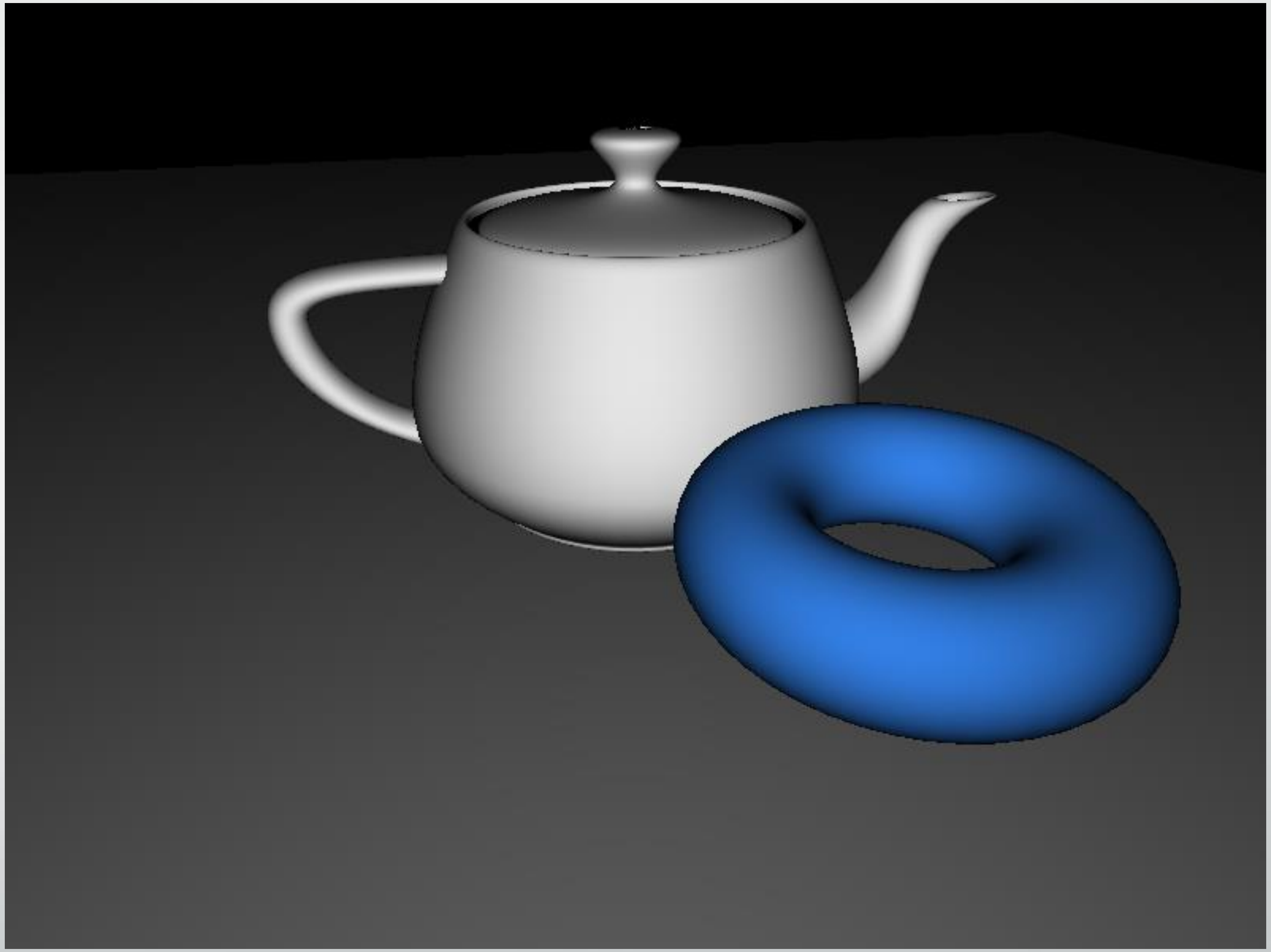
- To create the framebuffer object use this code (see more details in the Lab sheet):

```
void createGBufTex(GLenum texUnit, GLenum format,  GLuint &texid )
{
        glActiveTexture(texUnit);
        glGenTextures(1, &texid);
        glBindTexture(GL_TEXTURE_2D, texid);
        glTexStorage2D(GL_TEXTURE_2D,1,format,width,height);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
        GL_NEAREST);
}
```

Deferred shading implementation

- In render(), just call pass1() and immediately pass2()
- pass1() steps are the following:
  - Bind the framebuffer object deferredFBO
  - Clear the colour/depth buffers
  - Set pass to 1 and enable the depth test
  - Render the scene normally
- pass2() steps are the following:
  - Revert to the default FBO
  - Clear the colour/depth buffers
  - Set pass to 2 and disable the depth test
  - Render a screen-filling quad with texture coordinates

Deferred shading implementation



Useful links

- To read - Chapter 9 Advance buffers beyond the basics (OpenGL Superbible – see link on the DLE)
- To read - HDR: https://learnopengl.com/Advanced-Lighting/HDR
- Book: High dynamic range imaging, Reinhard et. al, (for further reference on HDR) : https://www.sciencedirect.com/book/9780125852630/high-dynamic-range-imaging
- To read – Bloom: https://learnopengl.com/Advanced-Lighting/Bloom
- To read: Image processing and screen space techniques in OpenGL 4 Shading Language Cookbook
- Beginner's guide to (CIE) Colorimetry: https://medium.com/hipster-color-science/a-beginners-guide-to-colorimetry-401f1830b65a
- To read - Gamma correction: https://learnopengl.com/Advanced-Lighting/Gamma-Correction