Textures

in shaders

Introduction

textures

Introduction

- **Textures** – colour information, depth information, shading parameters, displacement maps, normal vectors, etc
- **Immutable storage textures**: refers to textures storage that can't be changed but the texture itself can be changed. The allocation in memory and its format is immutable.
- Immutable storage textures have been introduce in OpenGL 4.2 and we normally use glTexStorage to allocate memory
- Immutable storage textures are useful because many runtime check for consistency can be avoided

Applying a texture

In OpenGL

Applying a texture

- Applying a texture to a surface involves accessing texture memory to retrieve a colour associated with a texture coordinate, and then applying that colour to the output fragment
- We are using a **sampler variable** which is a handle to a texture unit.
- A **sampler variable** is typically declared as a uniform variable within the shader and initialized within the main OpenGL application to point to the appropriate texture unit

Implementing a texture

```
GLuint Texture::loadTexture(  const std::string & fName  )
{
        int width, height;
        unsigned char * data = Texture::loadPixels(fName,  width, height);
        GLuint tex = 0;
        if( data != nullptr )
        {
                glGenTextures(1,  &tex);
                glBindTexture(GL_TEXTURE_2D,  tex);
                glTexStorage2D(GL_TEXTURE_2D,  1, GL_RGBA8,  width, height);
                glTexSubImage2D(GL_TEXTURE_2D,  0, 0, 0, width, height, GL_RGBA,
                                GL_UNSIGNED_BYTE,  data);

                glTexParameteri(GL_TEXTURE_2D,  GL_TEXTURE_MAG_FILTER,  GL_LINEAR);
                glTexParameteri(GL_TEXTURE_2D,  GL_TEXTURE_MIN_FILTER,  GL_NEAREST);

                Texture::deletePixels(data);
        }

        return tex;
}
```

Load and bind a texture

//load a texture

GLuint texID = Texture::loadTexture("../Project_Template/media/texture/brick1.jpg");

//bind a texture

glActiveTexture(GL_TEXTURE0);

glBindTexture(GL_TEXTURE_2D, texID);

//set the location of the texture variable in the shader

//with key word “binding”

layout(binding=0) uniform sampler2D Tex1;

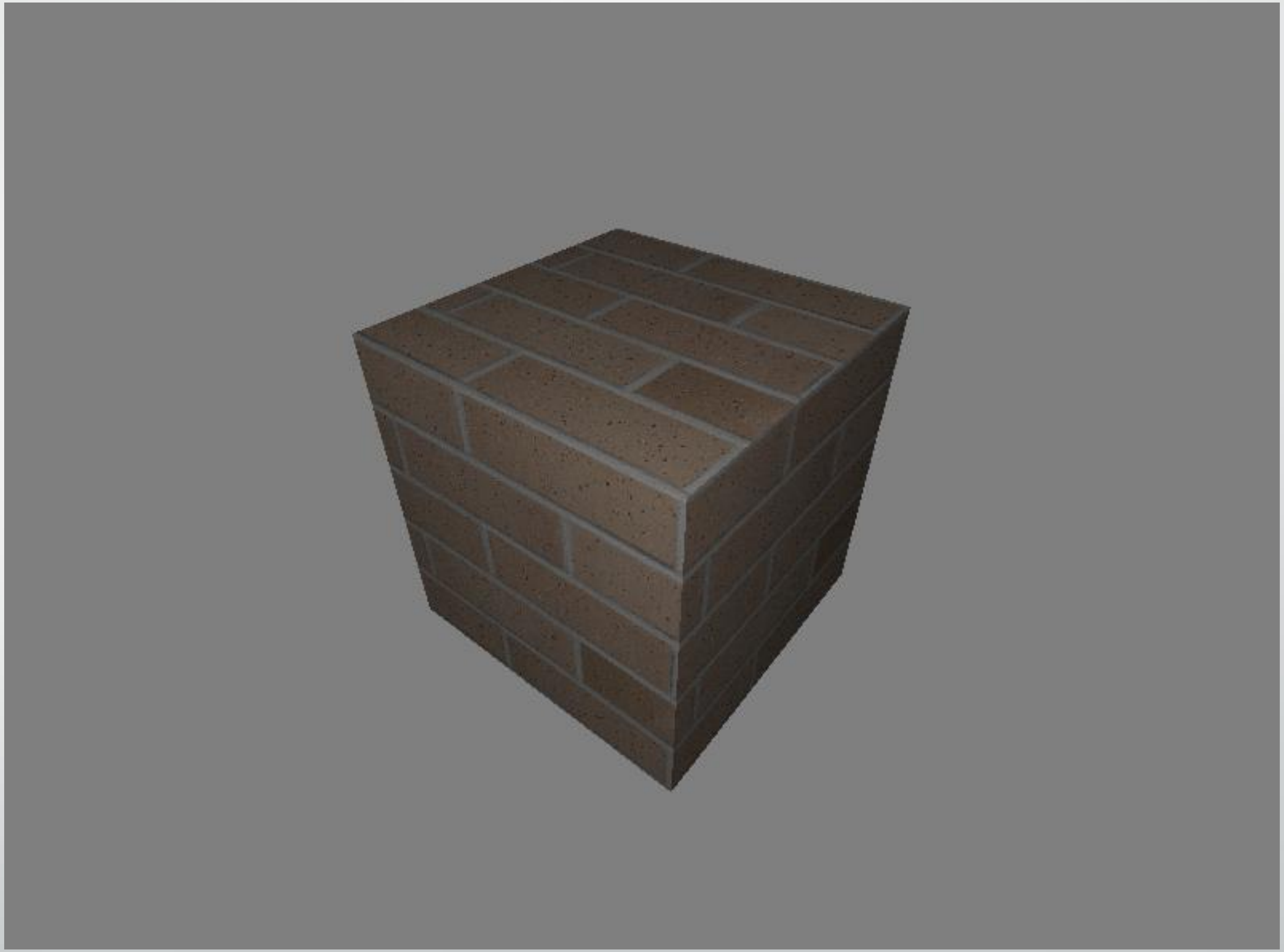
Texture Coordinates in the vertex shader

```
//in the vertex shader we pass the
//vertex texture coordinates
...
layout (location = 2) in vec2 VertexTexCoord;
...
out vec2 TexCoord;
...
void main()
{
    TexCoord = VertexTexCoord;
    ...
}
```

Look up Texture value in the fragment shader

```
//in the fragment shader we look up the texture value
…
in vec2 TexCoord;
…
// The texture sampler object
uniform sampler2D Tex1;
…
vec3 blinnPhong(vec3 pos, vec3 n)
{
        vec3 texColor = texture(Tex1, TexCoord).rgb;
        vec3 ambient = Light.La * texColor;
        …
}
```

Applying a texture



Texture

wrapping

Texture wrapping

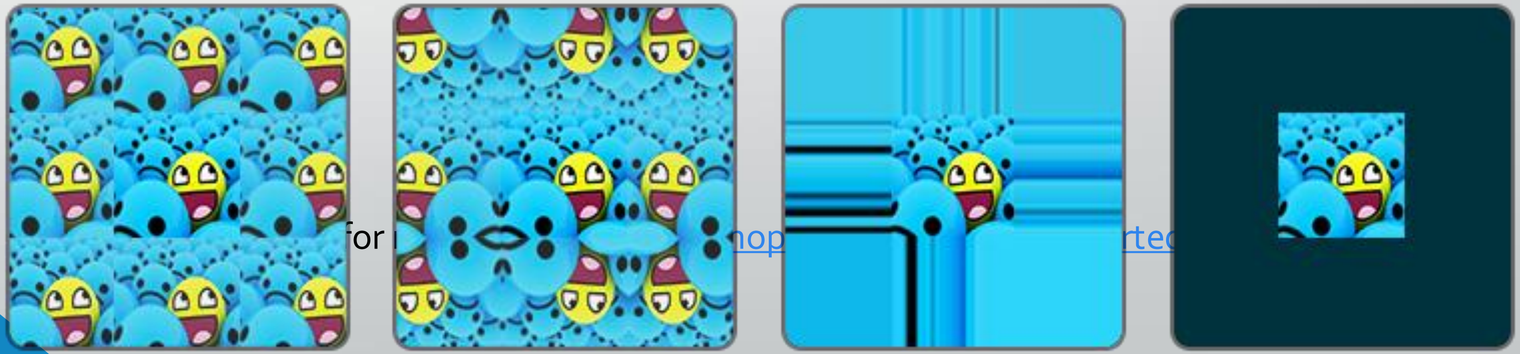
- Texture coordinates usually range from (0,0) to (1,1), going outside this range will cause the texture to repeat

- Some of the options for glTexParameter are:
  - GL_REPEAT
  - GL_MIRROR_REPEAT
  - GL_CLAMP_TO_EDGE
  - GL_CLAMP_TO_BORDER

For more information on glTexParameter, see this link: https://www.khronos.org/opengl/wiki/GLAPI/glTexParameter

Texture wrapping

- **GL_REPEAT**: default behaviour for textures, repeats the texture image
- **GL_MIRRORED_REPEAT**: Same as GL_REPEAT but mirrors the image with each repeat.
- **GL_CLAMP_TO_EDGE**: Clamps the coordinates between 0 and 1. The result is that higher coordinates become clamped to the edge, resulting in a stretched edge pattern
- **GL_CLAMP_TO_BORDER**: Coordinates outside the range are now given a user-specified border colour.



GL_REPEAT GL_MIRRORED_REPEAT GL_CLAMP_TO_EDGE GL_CLAMP_TO_BORDER

Texture

filtering

GL_NEAREST

GL_LINEAR

Texture filtering

- Texture coordinates don't depend on the resolution, so OpenGL needs to know which texture pixel to map to texture coordinates.
- A texture pixel is also know as a **texel**.
- Mapping is really important, especially on a very large object and low resolution texture

- **GL_NEAREST** (nearest neighbour or point filtering): selects the texel that centre is closest to the texture coordinates
- **GL_LINEAR** (bilinear filtering): takes an interpolated value from the texture coordinate's neighbouring texels, approximating a colour between the texels. The smaller the distance to a texel's centre the more that texels contributes to the sampled colour.

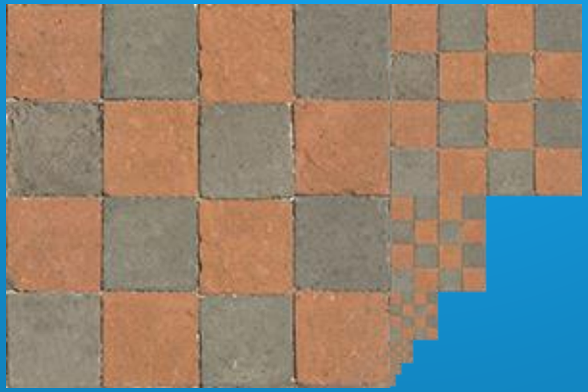
- Texture filtering can be set for scaling up or down of a texture (magnifying or minifying)

See images here for reference: https://learnopengl.com/Getting-started/Textures

Texture

mipmaps

Texture mipmaps



- Texture mipmaps a collection of texture images where each subsequent texture is twice as small compared to the previous one.
- After a certain distance threshold from the viewer, OpenGl will use a different mipmap texture that best suits the distance to the object.
- We can use glGenerateMipmaps to create a collection mipmapped textures
- Switching between mipmaps can cause some visual artifacts, so we can use filtering

See images here for reference: https://learnopengl.com/Getting-started/Textures

Texture mipmaps

- GL_NEAREST_MIPMAP_NEAREST: takes the nearest mipmap to match the pixel size and uses nearest neighbour interpolation for texture sampling.
- GL_LINEAR_MIPMAP_NEAREST: takes the nearest mipmap level and samples that level using linear interpolation.
- GL_NEAREST_MIPMAP_LINEAR: linearly interpolates between the two mipmaps that most closely match the size of a pixel and samples the interpolated level via nearest neighbor interpolation.
- GL_LINEAR_MIPMAP_LINEAR: linearly interpolates between the two closest mipmaps and samples the interpolated level via linear interpolation

You cannot use mipmaps filtering options for magnification filter, as mipmaps are typically used for downscaling.

See images here for reference: https://learnopengl.com/Getting-started/Textures

Multiple Texture

application

Multiple textures

- Multi texture application to a surface is typically used to create a variety of effects. We start with a base layer (clean surface) and additional layer can provide additional details:
  - Shadow
  - Blemishes
  - Roughness
  - Damage

Typically the light maps are applied as a second texture on top of the original texture.

Pre-backed lighting. They add additional information as light exposure, producing shadows and shading without the need to calculate the reflection model at runtime.

Multiple textures implementation

- We load the textures in the init()

```
GLuint brick =
Texture::loadTexture("../Project_Template/media/
texture/brick1.jpg");

GLuint moss =
Texture::loadTexture("../Project_Template/media/
texture/moss.png");


//load texture into channel 0
glActiveTexture(GL_TEXTURE0);

glBindTexture(GL_TEXTURE_2D, brick);


//load texture into channel 1
glActiveTexture(GL_TEXTURE1);

glBindTexture(GL_TEXTURE_2D, moss);
```

Multiple textures implementation

- Vertex shader we just pass the texture coordinates similarly to a single texture shader (see previous slides):

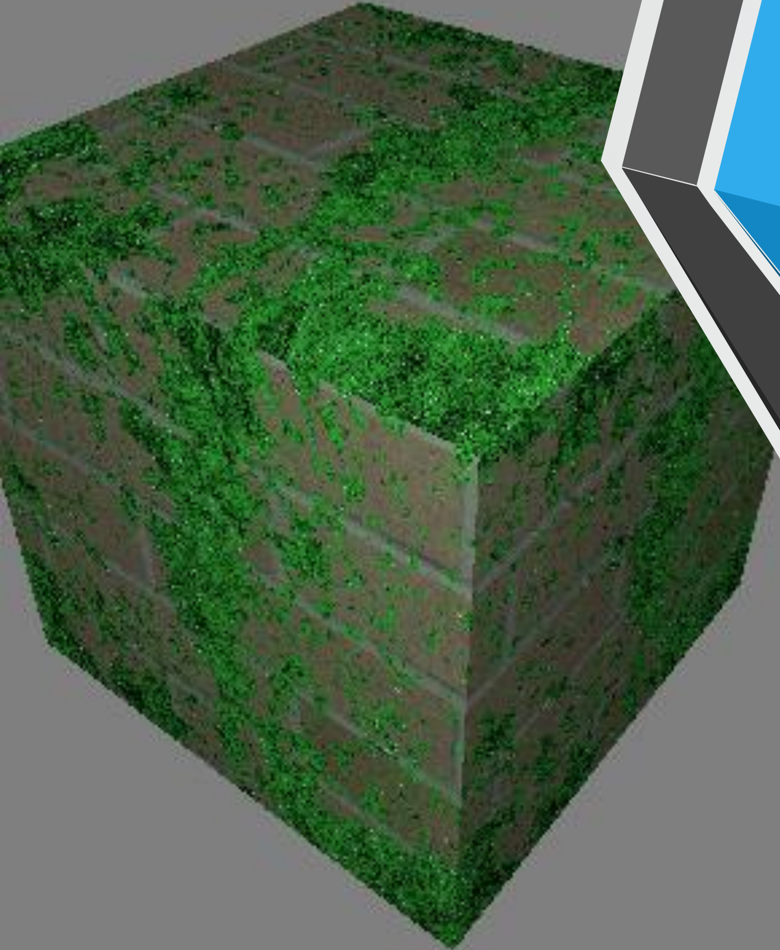
```
layout (location = 2) in vec2 VertexTexCoord;
...
```

- In fragment shader we bind the textures:

```
layout(binding=0) uniform sampler2D BrickTex;
layout(binding=1) uniform sampler2D MossTex;
```

In blinnPhong:

```
vec4 brickTexColor = texture( BrickTex, TexCoord );
vec4 mossTexColor = texture( MossTex, TexCoord );
vec3 col = mix(brickTexColor.rgb, mossTexColor.rgb, mossTexColor.a);
```



Multiple textures implementation

Discarding pixels

with alpha maps

Discarding pixels

- To create an effect of an object that has holes we can use a texture with an alpha channel that contains information about the transparent parts of the object.
- We can use the keyword **discard** to completely discards fragments when alpha value is bellow a certain value.
- Using the key word discard, we don’t need to depth sort our polygons because there is no blending.
- We might need to use two-sided lighting when rendering the object

Discarding pixels

- Vertex shader is similar to the other examples
- In fragment shader we bind 2 textures, base texture and alpha texture:

  layout(binding=0) uniform sampler2D BaseTex;

  layout(binding=1) uniform sampler2D AlphaTex;

- In main(), we use alpha value to discard fragments and execute blinnPhong based on the frontFacing or backFacing :

```
void main()
{
        vec4 alphaMap = texture( AlphaTex, TexCoord );

        if(alphaMap.a < 0.15 )
                discard;
        else
                if( gl_FrontFacing )
                        FragColor = vec4(blinnPhong(Position,normalize(Normal)), 1.0);
                else
                        FragColor = vec4( blinnPhong(Position,normalize(-Normal)), 1.0);
}
```

Discarding pixels

Normal map

texture

Normal maps

- **Normal mapping** is a technique for "faking" variations in a surface that doesn't really exist in the geometry of the surface.
- It is useful for producing surfaces that have bumps, dents, roughness, or wrinkles without actually providing enough position information (vertices) to fully define those deformations.
- The underlying surface is actually smooth, but is made to appear rough by varying the normal vectors using a texture (the normal map).

Normal maps

- A normal map is a texture in which the data stored within the texture is interpreted as normal vectors instead of colours.
- The normal vectors are typically encoded into the RGB information of the normal map so that the red channel contains the **x** coordinate, the green channel contains the **y** coordinate, and the blue channel contains the **z** coordinate.
- The normal map can then be used as a **texture** in the sense that the texture values affect the normal vector used in the reflection model rather than the colour of the surface.

Normal maps

- Normal maps are interpreted as vectors in a **tangent space** (also called the **object local coordinate system**).
- The origin is located at the surface point and the normal to the surface is aligned with the **z** axis (0, 0, 1), therefore the **x** and **y** axes are at a tangent to the surface.
- The normal vectors stored within the normal map can be treated as perturbations to the true normal, and are independent of the object coordinate system.

Normal maps

- We transform the vectors used in our reflection model (light direction and view direction into tangent space in the vertex shader, and then pass them along to the fragment shader where the reflection model will be evaluated.
- We need 3 normalised vectors in order to produce a matrix used for transformation:
  - n: **normal vector** used for z axis
  - t: **tangent vector** used for x axis
  - b: **binormal vector** used for y axis

Normal maps

$$\begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} = \begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

S: point in tangent space

P: point in camera coordinates

The matrix is formed of the t, b and n vectors.

- t and n vectors are usually provided
- we just need to calculate the b vector as the cross product between normal and tangent vector

Normal maps

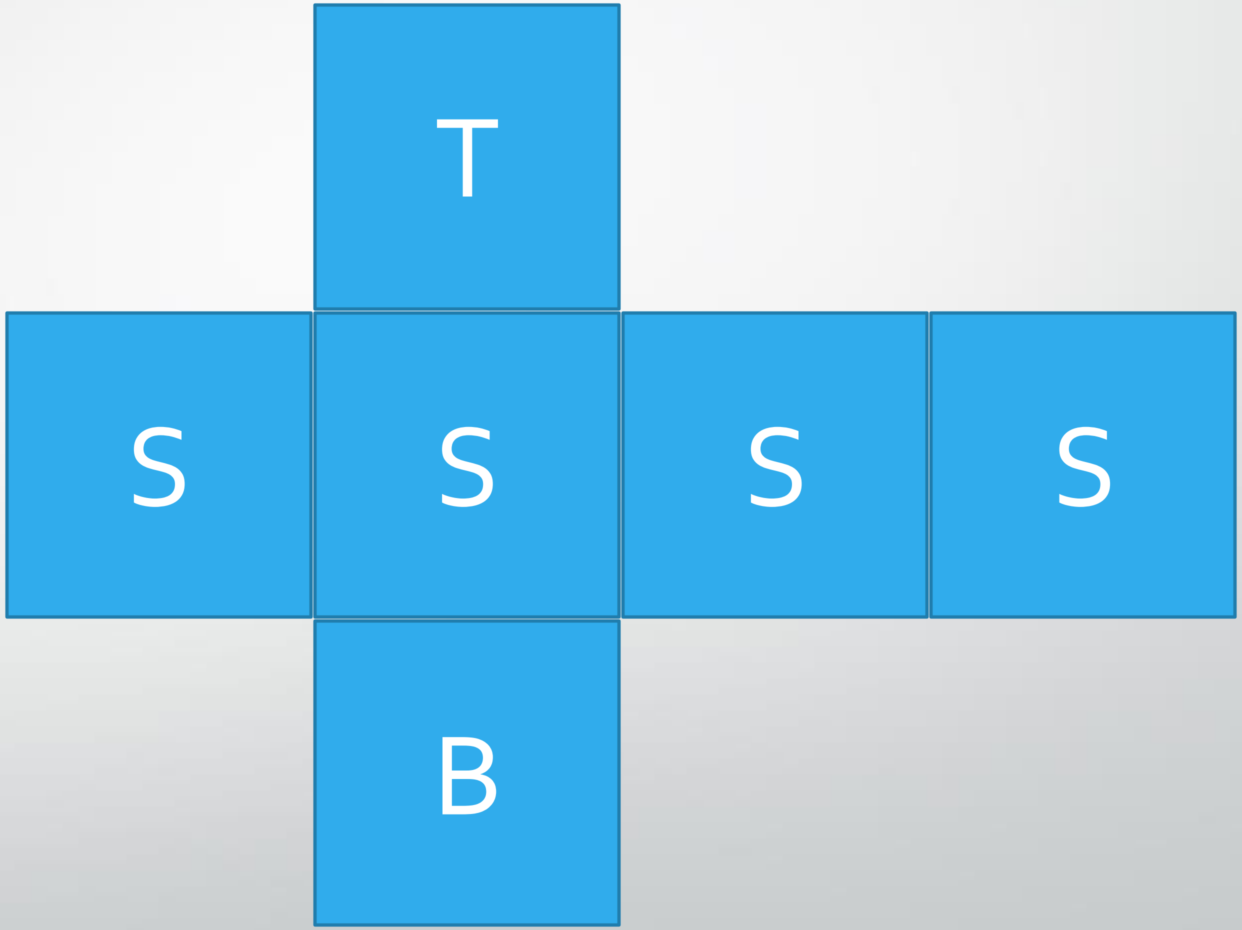
Skybox

texture

Skybox

- A **cube map** is one of the more common varieties of textures used in environment mapping.
- A cube map is a set of six separate images that represent the environment projected onto each of the six faces of a cube.
- The six images represent a view of the environment from the point of view of a viewer located at the centre of the cube.

Skybox – unravelled cube



Skybox

- OpenGL provides built-in support for cube map textures (using the GL_TEXTURE_CUBE_MAP target
- The texture is accessed using a three-dimensional texture coordinate (s, t, r). The texture coordinate is interpreted as a direction vector from the centre of the cube.
- The line defined by the vector and the centre of the cube is extended to intersect one of the faces of the cube. The image that corresponds to that face is then accessed at the location of the intersection.
- All images need to be square (preferably with dimensions that are a power of two), and that they are all the same size.

Projective texture mapping

technique

Projective texture mapping

- We apply a texture to the objects in a scene as if the texture was a projection from an imaginary "projector" located somewhere within the scene.
- To project a texture onto a surface, all we need do is determine the texture coordinates based on the relative position of the surface location and the source of the projection (the projector).
- An easy way to do this is to think of the projector as a camera located somewhere within the scene.

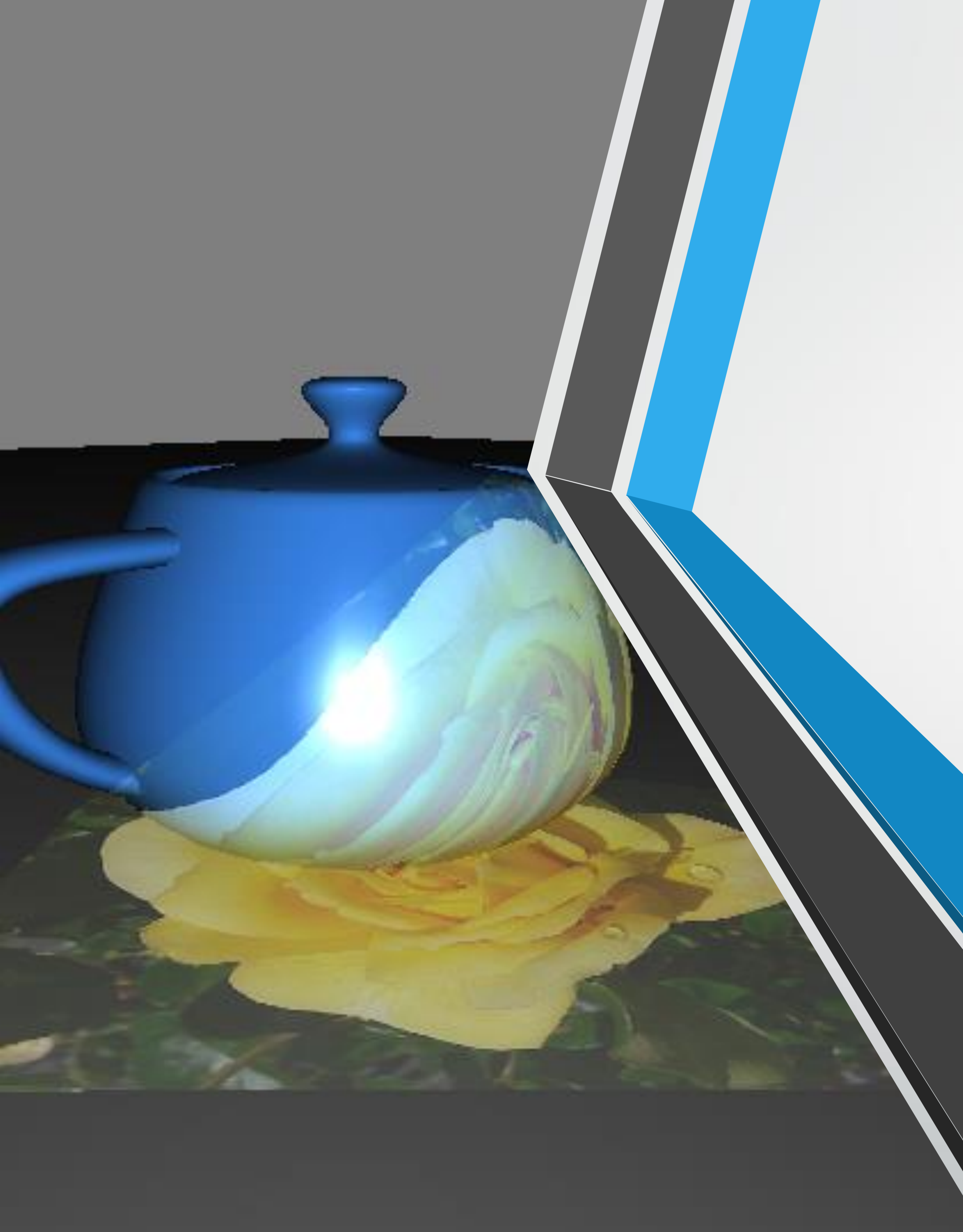
Projective texture mapping

- We define a coordinate system centred at the projector's location, and a **view matrix** (**V**) that converts coordinates to the projector's coordinate system.
- Next, we'll define a perspective **projection matrix** (**P**) that converts the view frustum (in the projector's coordinate system) into a cubic volume of size two, centred at the origin.
- We add an additional matrix for re-scaling and translating the volume to a volume of size one shifted so that the volume is centred at (0.5, 0.5, 0.5)

Projective texture mapping

$$M = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} PV$$

The matrix will convert world coordinates that fall within the view frustum of the projector to a range between 0 and 1 (homogeneous), which can then be used to access the texture.



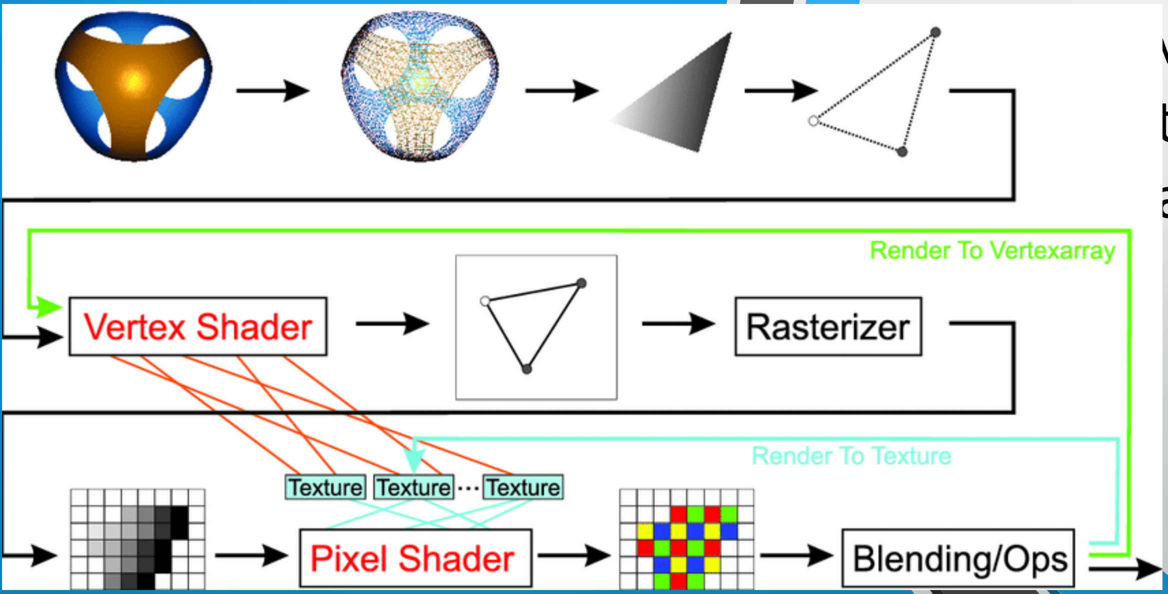
Projective texture mapping

Render to a texture

technique

Render to a texture

- Rendering to a texture is a useful technique, it could be a pattern that is generated from some internal algorithm (a so-called **procedural texture**)
- or it could be that the texture is meant to represent another portion of the scene like a video screen where one can see another part of the game world, perhaps via a security camera in another room.



Render to a texture

- In OpenGL is easy to render to a texture thanks to the introduction of **framebuffer objects** (**FBOs**).

- The steps involved are first we initialise:
  - Setup the FBO
- At rendering:
  - Bind to the FBO
  - Render the texture
  - Unbind the FBO (back to the default framebuffer)
  - Render the scene using the texture

Render to a texture

Setup the FBO:

```
void SceneBasic_Uniform::setupFBO() {
   // Generate and bind the framebuffer
   glGenFramebuffers(1, &fboHandle);
   glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);

   // Create the texture object
   GLuint renderTex;
   glGenTextures(1, &renderTex);
   glActiveTexture(GL_TEXTURE0);  // Use texture unit 0
   glBindTexture(GL_TEXTURE_2D, renderTex);

   glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, 512, 512);

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

   // Bind the texture to the FBO
   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, renderTex, 0);
....
```

Render to a texture

Setup the FBO:

```
// Create the depth buffer
  GLuint depthBuf;
  glGenRenderbuffers(1, &depthBuf);
  glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
  glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, 512, 512);

  // Bind the depth buffer to the FBO
  glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_RENDERBUFFER, depthBuf);

  // Set the targets for the fragment output variables
  GLenum drawBuffers[] = { GL_COLOR_ATTACHMENT0 };
  glDrawBuffers(1, drawBuffers);

  GLenum result = glCheckFramebufferStatus(GL_FRAMEBUFFER);
  if (result == GL_FRAMEBUFFER_COMPLETE) {
    cout << "Framebuffer is complete" << endl;
  }
  else {
    cout << "Framebuffer error: " << result << endl;
  }

  // Unbind the framebuffer, and revert to default framebuffer
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Render to a texture

Render:

```
void SceneBasic_Uniform::render()
{
    //bind the buffer
    glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);

    //render to texture
    renderToTexture();

    //flush the buffer
    glFlush();

    //unbind the write buffer and bind the default buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);

    //render the scene using the newly written texture
    renderScene();
}
```

Render to a texture

Render to texture:

```
void SceneBasic_Uniform::renderToTexture() {
    prog.setUniform("RenderTex", 1);
    glViewport(0, 0, 512, 512);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    view = glm::lookAt(vec3(0.0f, 0.0f, 2.5f), vec3(0.0f, 0.0f, 0.0f),
vec3(0.0f, 1.0f, 0.0f));
    projection = glm::perspective(glm::radians(50.0f), 1.0f, 0.3f, 100.0f);

    prog.setUniform("Light.Position", glm::vec4(0.0f, 0.0f, 0.0f, 1.0f));
    prog.setUniform("Material.Ks", 0.95f, 0.95f, 0.95f);
    prog.setUniform("Material.Shininess", 100.0f);

    model = mat4(1.0f);
    model = glm::rotate(model, angle, vec3(0.0f, 1.0f, 0.0f));
    setMatrices();
    spot->render();
}
```

Render to a texture

Render scene:

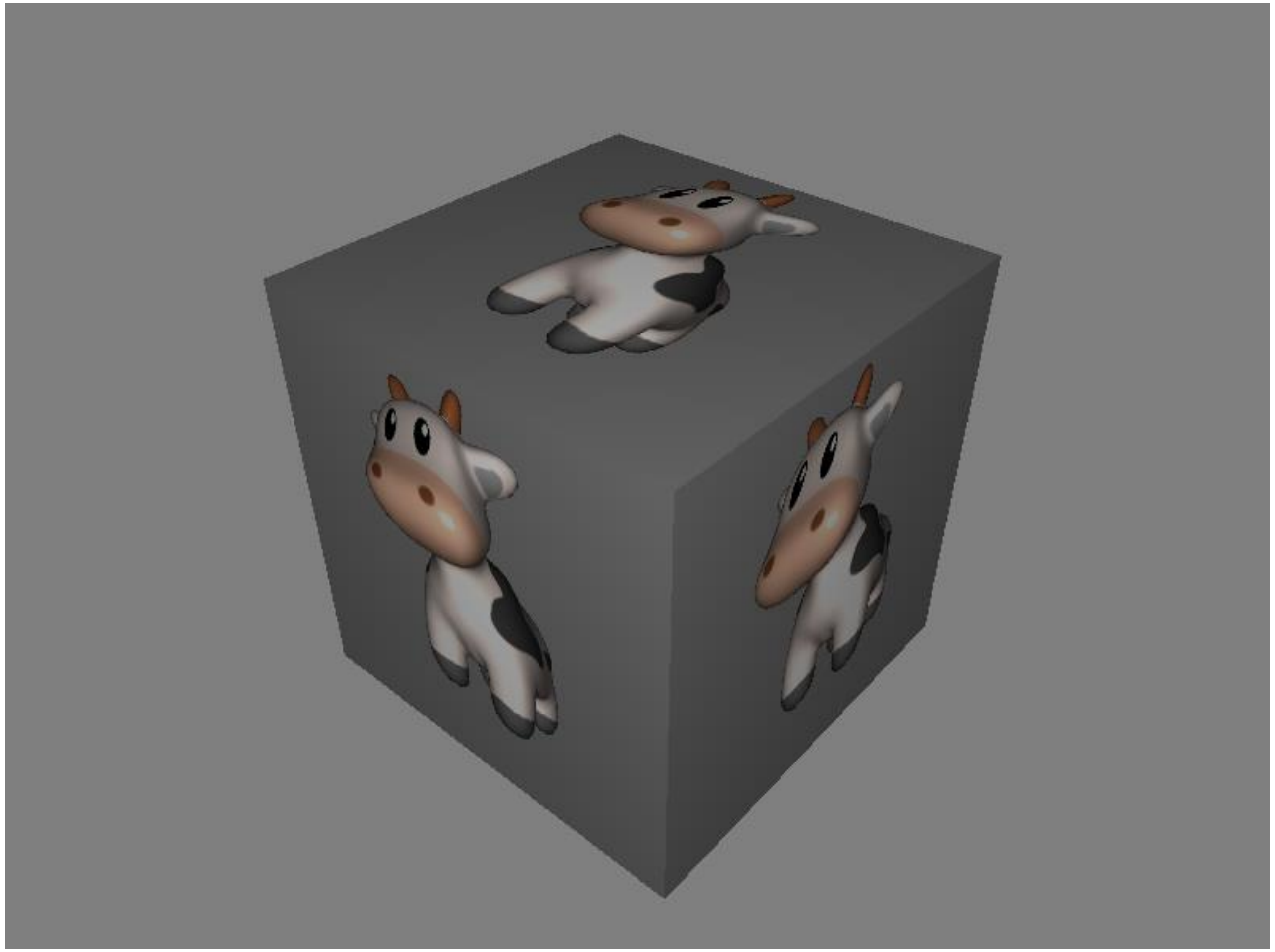
```
void SceneBasic_Uniform::renderScene() {
    prog.setUniform("RenderTex", 0);
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    vec3 cameraPos = vec3(2.0f * cos(angle), 1.5f, 2.0f * sin(angle));
    view = glm::lookAt(cameraPos, vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 1.0f, 0.0f));

    projection = glm::perspective(glm::radians(45.0f), (float)width / height, 0.3f,
100.0f);

    prog.setUniform("Light.Position", glm::vec4(0.0f, 0.0f, 0.0f, 1.0f));
    prog.setUniform("Material.Ks", 0.0f, 0.0f, 0.0f);
    prog.setUniform("Material.Shininess", 1.0f);

    model = mat4(1.0f);
    setMatrices();
    cube.render();
}
```



Render to a texture

Useful links

- To read - Chapter 5 Basic texturing (OpenGL Superbible – see link on the DLE)
- To read - Chapter 7 More advanced texture topics (OpenGL Superbible – see link on the DLE)
- STB image loader: https://github.com/nothings/stb/blob/master/stb_image.h
- Sean Barret: https://github.com/nothings
- To read - Textures: https://learnopengl.com/Getting-started/Textures
- To read: Using textures in OpenGL 4 Shading Language Cookbook
- Binding textures: https://www.khronos.org/opengl/wiki/Sampler_(GLSL)
- NVIDIA texture tool exporter: https://developer.nvidia.com/nvidia-texture-tools-exporter
- Frame buffers: https://learnopengl.com/Advanced-OpenGL/Framebuffers