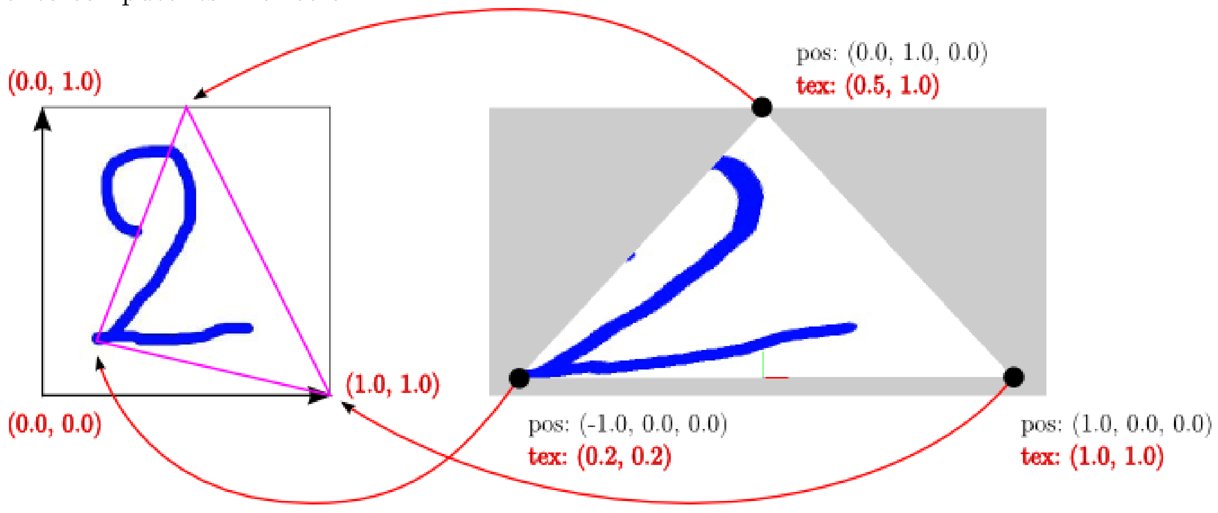
Computer Graphics
Practical 7: Texturing

INSA Fourth Year - 2023/2024
Maud Marchal, Glenn Kerbiriou

1 About textures

You might have heard about texturing as a technique that could increase the realism of a rendering, by mapping parts of an image onto triangles. In fact, texturing is a wider concept: it is the sampling of a function f at a real valued position p. The coordinates of p are all in the range $[0, 1]$. The function f is known at regularly spaced discrete positions, i.e. on a 1-, 2-, or 3-dimensional grid, depending on the case. The values of f on this grid is referred to as the **texture** and the real valued position p is known as the **texture coordinate**.

In this practical, all textures will be 2D RGBA images, i.e. functions that associate a RGBA color to a 2-dimensional coordinate. The texture coordinates are generally written (u, v) in that case. Such coordinates are given for each vertex of a triangle (see the image below to see where is the coordinates origin in the **OpenGL convention**). Texture coordinates are vertex attributes, thus fragments will have interpolated vertex coordinates. The texture will be then **sampled** at those coordinates, giving **texels** (texture elements) and a fragment will use its texel to compute its final color.



You will not code much in this practical. We have chosen to let you observe the code and experiment the effects of different option values, in order to understand both the mechanism and the possibilities of texturing. When you get the concept, you will then add texturing to the renderables of your project.

Texture of other dimensions. Textures can have other dimensions and store different values. For example:

- a two dimensional texture storing real values that represent the height of the ground relatively to a known maximum height. The texture is what we call an elevation map.
- a one dimensional texture storing RGB colors. This is typically used as a lookup table to associate a color to a scalar value, in order to visualize 1-D values defined on a surface.
- a three dimensional texture storing real values in order to visualize medical images (CT scan, IRM, ...).

Defining texture coordinates. The definition of texture coordinates for 2D surfaces may be a tough problem. The surface needs to be embedded in a two dimensional space. Even for 3D meshes, some methods exists but they are beyond the scope of this course. Thus, for the project, try to keep things simple or find a mesh with existing texture coordinates relatively to an existing image (see exercise 1).

2 About the provided code

The provided code illustrates different basic texturing techniques:

- CPU code is encapsulated into C++ classes that are gathered in a texturing subdirectory.
- GPU code is gathered in the shaders subdirectory.

We will now give more details about important parts of the provided code.

2.1 CPU code for texturing

Set up a texture Like any memory place in OpenGL, a texture is managed thanks to an ID. This ID is first generated then bound to a binding point before transferring the data to the GPU. Apart from the texture options defined by the `glTexParameteri` function (studied in the following exercises), this operation is very similar to what you are now used to do on vertex buffers. After this operation is done (only once), the texture can be used for rendering.

C++ code to setup a texture

```
// Create a new texture id, then bind it
glGenTextures(1, &m_texId);
glBindTexture(GL_TEXTURE_2D, m_texId);

// Texture options
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);

// Load an image (here using the sfml library)
sf::Image image;
image.loadFromFile(filename);
// sfml inverts the v axis...
// Hence, flip it to put the image in OpenGL convention:
    lower left corner is (0,0)
image.flipVertically();

// Transfer the image to the texture on the GPU
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, image.getSize().x,
     image.getSize().y, 0, GL_RGBA, GL_UNSIGNED_BYTE, (const
    GLvoid*)image.getPixelsPtr());

// Release the texture id
glBindTexture(GL_TEXTURE_2D, 0);
```

Define texture coordinates. As we saw in the introduction, texture coordinates are vertex attributes. As such, they are loaded in a vertex buffer. This is not something new, but you should remember to define and load the texture coordinate attribute.

C++ code to load texture coordinates

```
// Example with a single triangle
m_positions.push_back(glm::vec3(-1.0, 0.0, 0.0));
m_positions.push_back(glm::vec3( 1.0, 0.0, 0.0));
m_positions.push_back(glm::vec3( 0.0, 1.0, 0.0));

m_texCoords.push_back(glm::vec2(0.2, 0.2)); //close to the
    texture's lower left corner
m_texCoords.push_back(glm::vec2(1.0, 0.0)); //lower right
    corner of the texture
m_texCoords.push_back(glm::vec2(0.5, 1.0)); //middle of the
    top line of the texture

// Generate buffers and send data to the GPU, as usual
...
glGenBuffers(1, &m_tBuffer);
glBindBuffer(GL_ARRAY_BUFFER, m_tBuffer);
glBufferData(GL_ARRAY_BUFFER, m_texCoords.size()*sizeof(glm::
    vec2), m_texCoords.data(), GL_STATIC_DRAW);
```

Using the texture in the shader program. This step is actually different than for other type of vertex attributes. The texture should first be bound to a binding point the shader program can use. This binding point is named a **texture unit**, and in this case we will use `GL_TEXTURE0`. In the fragment shader, we will use a special uniform of type `sampler2D` named `texSampler`. This uniform will sample a texel at the given texture coordinate. In order for the sampler to know which texture it should sample, this uniform should be initialized with the texture unit the texture is bound to.

C++ code to use a texture in the shader program

```
// Send positions , normals , ...  as usual
...

// Get ids from the shader program
int texcoordLocation = m_shaderProgram->getAttributeLocation(
   "vTexCoord");
int texsamplerLocation = m_shaderProgram->getUniformLocation(
   "texSampler");

// Bind texture on the texture unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D , m_texId);

// Tells the sampler to use the texture unit 0
glUniform1i(texsamplerLocation , 0);

// Populate the vertex attribute with the texture coordinates
    buffer
glEnableVertexAttribArray(texcoordLocation);
glBindBuffer(GL_ARRAY_BUFFER , m_tBuffer);
glVertexAttribPointer(texcoordLocation , 2, GL_FLOAT , GL_FALSE
   , 0, (void*)0);

// Once you're done , release the texture
glBindTexture(GL_TEXTURE_2D , 0);
```

2.2 GPU code for texturing

Vertex shader. In the vertex shader, there is not much to do. You just have to transmit the texture coordinates to the fragment buffer in order to have interpolated texture coordinates for each fragment.

GLSL vertex shader

```
#version 400
uniform mat4 projMat , viewMat , modelMat;

in vec3 vPosition;
in vec2 vTexCoord;
out vec2 surfel_texCoord;

void main()
{
    gl_Position = projMat*viewMat*modelMat*vec4(vPosition ,
       1.0f);
    // simply pass the texture coordinate to the fragment
    surfel_texCoord = vTexCoord;
}
```

Fragment shader. The fragment is more interesting since this is the place where a texel is fetched and used to compute the color of a fragment. The sampling of the texture at the given texture coordinate is done by the sampler (a `sampler2D` in our case as the texture is two-dimensional). This operation is performed by applying the function `texture()` with the sampler and the texture coordinate.

GLSL fragment shader

```
#version 400
uniform sampler2D texSampler;

in vec2 surfel_texCoord;
out vec4 outColor;

void main()
{
    // here, the color is simply the texel
    outColor = texture(texSampler, surfel_texCoord);
}
```

In the above example, the final color of a fragment is the sampled texel. We can use the texel in other ways. For example, it can be combined with the local illumination color of a surfel (as done in the previous practical).

GLSL fragment shader with local illumination

```
#version 400
...

void main()
{
    // compute the Phong color based on material and lights
    illuminationColor = ...

    // get the texel
    textureColor = texture(texSampler, surfel_texCoord);

    // combine the two, using custom operations (here a
       simple *) or the GLSL funtion ‘mix‘
    outColor = textureColor * vec4(illuminationColor, 1.0);
}
```

3 Exercise 1: textured bunny

Look at the files `texturing/TexturedMeshRenderable.[cpp|hpp]`. Texturing was added to the mesh class of previous practical. Identify the different steps of texturing, presented in the above section.

4 Exercise 2: wrapping options

First, let's focus on the ground plane of the scene. A grass texture is directly mapped on it: the coordinates textures of each corner are respectively (0,0), (1,0), (1,1) and (0,1).

Ok, the plane looks like your garden. But what happens if you zoom on it? You should clearly see the (square) pixels of the image, which is not really nice. Actually this is due to the image resolution (here 512x512 pixels): it looks good only if mapped on a not too large plane, or if the camera is far enough. One could use a larger image, which would however consumes more resources. The usual solution is to use **texture tiling**, which means an image will be repeated to fully cover the polygon to texture, while limiting its zooming factor.

Several wrapping options could be defined, with the command `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, ...);`. Press the F6 key to change the wrapping options for the ground:

- the texture coordinates will not be in $[0,1]^2$ anymore, but in $[-5,5]^2$.
- the wrapping value will cycle in this order :
  - `GL_CLAMP_TO_EDGE`
  - `GL_REPEAT`
  - `GL_MIRRORED_REPEAT`
  - `GL_CLAMP_TO_BORDER`

Questions:

- What is the behavior with these different values?
- What characteristic must be fulfilled by the image when using texture tiling ?

5 Exercise 3: filtering options

The F7 key is assigned to interactively toggle the texture **interpolation** mode between :

- `GL_LINEAR`
- `GL_NEAREST`
- `GL_MIPMAP`, but we will study this one in the next exercice

This is again done with the `glTexParameteri` function (see how key events are handled in `TexturedMeshRenderable.cpp`). Observe and understand the differences. At what rendering stage is this filtering operation performed?

6 Exercise 4: multi-resolution (mipmapping)

Texture resolution plays an important role in the quality of the scene rendering. However, with a higher resolution comes a higher price in terms of resource consumption (storage) and bandwidth (texture loading and access). Also, a polygon rendered far away doesn't need to be rendered with high resolution. In fact, it would run into the aliasing problem linked to discrete sampling of the texture.

The **mipmapping** technique is a way to address these issues by using several pre-filtered resolution levels of the same texture image. Depending on the distance between each fragment and the camera, the rendering process will automatically switch to the most adequate resolution of the texture.

The cube on the left (an instance of the `TexturedCubeRenderable` class) is textured without mipmapping. However, mipmapping is used for the cube in the middle (a `MipMapCubeRenderable`).

- Look at the `MipMapCubeRenderable` class: a single texture is created, however several images are loaded with different resolutions (here 256x256, 128x128, ...). You can observe these images directly with a standard viewer: `textures/mipmap1.png`. A different figure enables to clearly identify which image is currently used to render a fragment.
- Play with the camera to zoom in and out, and visualize the scene with different angles. On the left cube, you should clearly observe the aliasing problem. On the middle one, notice how the rendered image is swapped from one view point to another.
- In mipmapping as well, several filtering options are available:
  - `GL_NEAREST_MIPMAP_NEAREST`
  - `GL_LINEAR_MIPMAP_NEAREST`
  - `GL_NEAREST_MIPMAP_LINEAR`
  - `GL_LINEAR_MIPMAP_LINEAR`

  Toggle the F8 key to test these filtering options.
- Can you explain the differences? What parameter provides the best and smoothest rendering quality? At what cost?

In a real application, you generally will not create several images at different resolutions. Instead, one should load the desired image then use the `glGenerateTextureMipmap` function once to generate the sub-resolutions images. This is done in `TexturedMeshRenderable` when you press two times F7.

7 Exercise 5: multi-texturing

The cube on the right (of class `MultiTexturedCubeRenderable`) is textured with two textures. This means that:

- two textures are created and loaded
- two texture units are used
- a vertex may have two texture coordinates attributes (not the case here)
- the fragment shader uses two samplers.

The cube on the left is drawn using `multiTexture<Vertex/Fragment>.glsl`. Look at the fragment shader.

- The two texel colors are multiplied together
- Think of another way of mixing the colors. (you can search for GLSL `mix()` function)
- Make the blending vary over time (press F4 to start the animation)

The cube on the right is an example of secondary texture usage : normal mapping (see `multiTextureNormalFragment.glsl`). So far, if we were to display a geometrically very detailed surface, we would use a lot of triangles. Normal mapping consists in storing subtle normal deviations in a texture called normal map. Then, the shader samples this texture, reconstructs the tilted normal and uses it instead of the surfel normal for lighting computations. As a result, the viewer is led to perceive depth where there is only a flat surface. Secondary textures can be used to store of other features : 3D displacements, ADS components, shininess, ambient occlusion ($\approx$self shadows) and more. These features could be passed as vertex attributes but then we would need high polygon count to reach the same resolution. Also, image processing techniques can be used is the texture domain.

Secondary textures are extensively used in video games and any 3D application involving realtime realistic rendering.

8 Exercise 6: cubemap

Cubemaps allow us to display a detailed environment all around the scene. The goal is to render the inside of a textured cube to trick the viewer to see a vast landscape. Uncomment the last block of the `.cpp` to see it in action. Even if an easier tecnique is to use a panorama (360 degrees image) we often prefer using cubemaps. Can you guess why ?

One may also want to illuminate the objects in the scene from the environment. `demo_environment_lighting.cpp` shows a way to do that.

9 Exercise 7: project related

Texturing is the last feature you will see in these practicals. We have seen the wrapping and filtering options of the textures and how to play with them for other purposes such as normal mapping or cubemaps.

You can now add textures to the renderables of your project such as your boat and some surrounding elements. If you do not want to spend hours to build a texture and a set of texture coordinates, find repeating textures. To avoid texture aliasing, don't forget to use adequate filtering.