Computer Graphics
– Textures

J.-Prof. Dr. habil. Kai Lawonn

Introduction

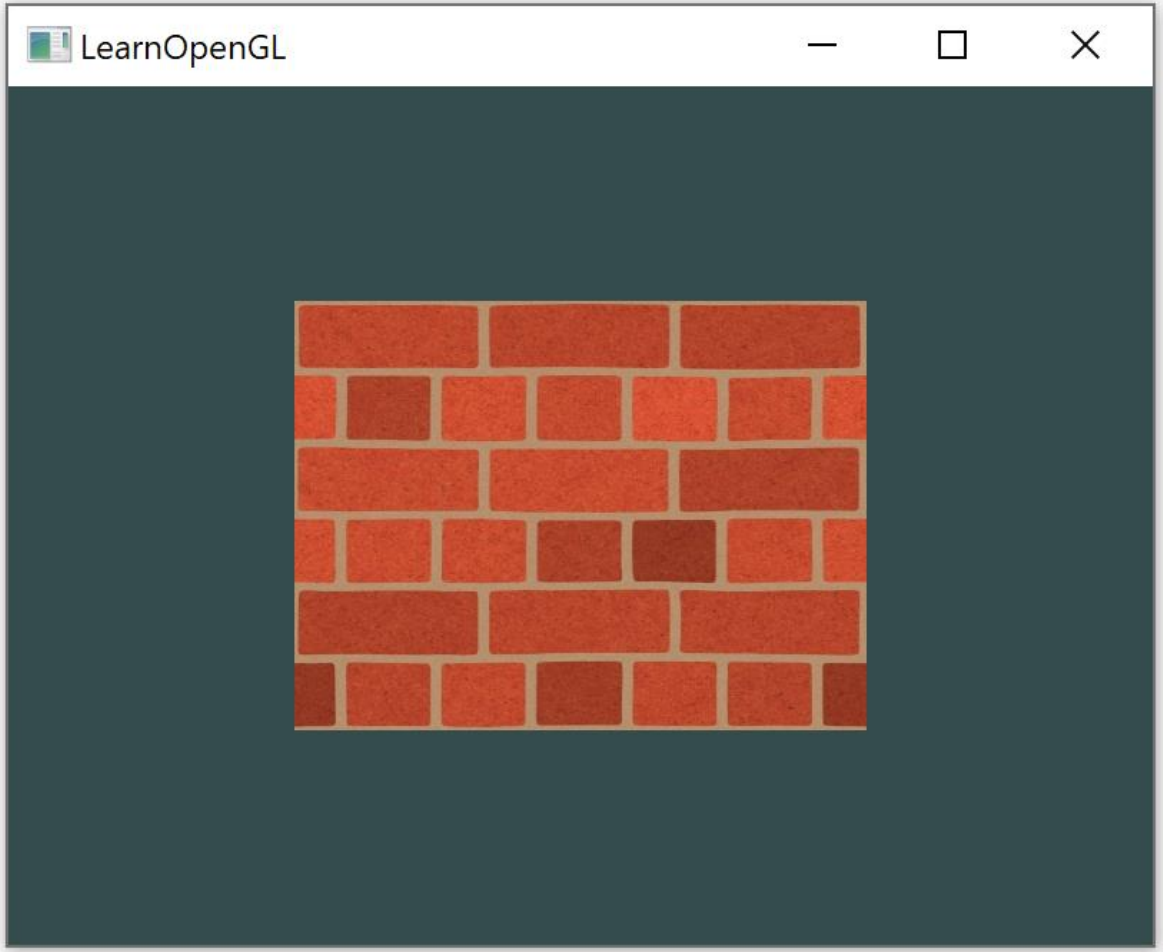
- Objects can be colored for each vertex to create some interesting images
- To add realism we have to add many vertices so we could specify a lot of colors → considerable amount of extra overhead (every object would need more vertices)

Introduction

- Artists and programmers generally prefer to use a texture
- A texture is a 2D image (even 1D and 3D textures exist) used to add detail to an object, e.g., think of brick texture on the object
- Can insert a lot of detail in a single image, without having to specify extra vertices

Introduction

- A rectangle with a brick texture



Introduction

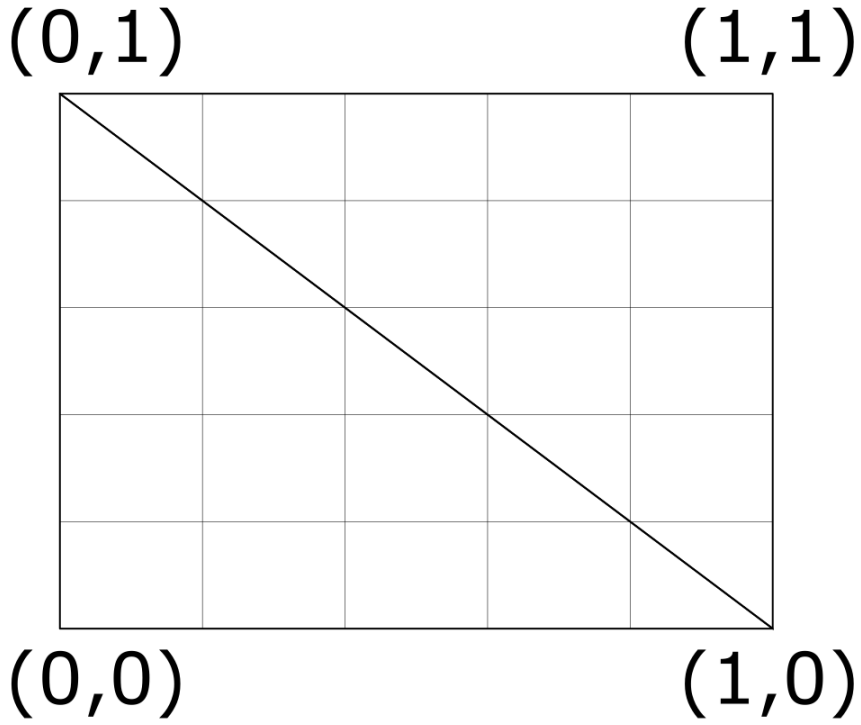
Aside from images, textures can also be used to store a large collection of data to send to the shaders.

Introduction

- To map a texture to the triangles, texture coordinate (TC) are needed for the vertices
- Fragment interpolation then does the rest for the other fragments.
- TC range from 0 to 1 in the x and y axis
- Retrieving the texture color using TC is called sampling
- TC start at (0,0) for the lower left corner of a texture image to (1,1) for the upper right corner of a texture image

Introduction

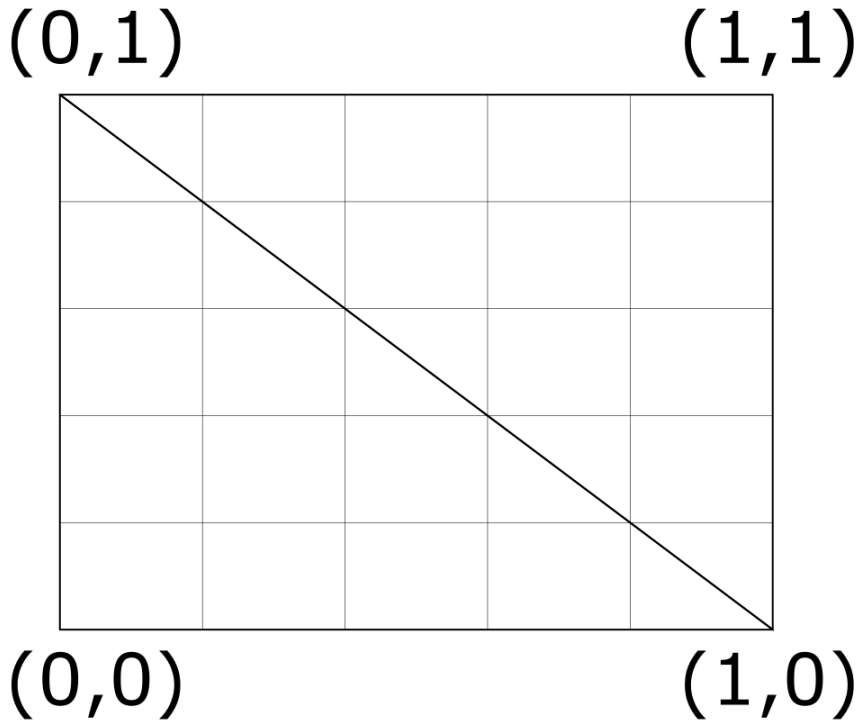
- The following image shows how we map texture coordinates to the triangle:



Introduction

- The following image shows how we map texture coordinates to the triangle:

```
float vertices[] = {
     // positions         // colors          // texture coords
     0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f,
     0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f
    };
    unsigned int indices[] = {
     0, 1, 3, // first triangle
     1, 2, 3  // second triangle
    };
```



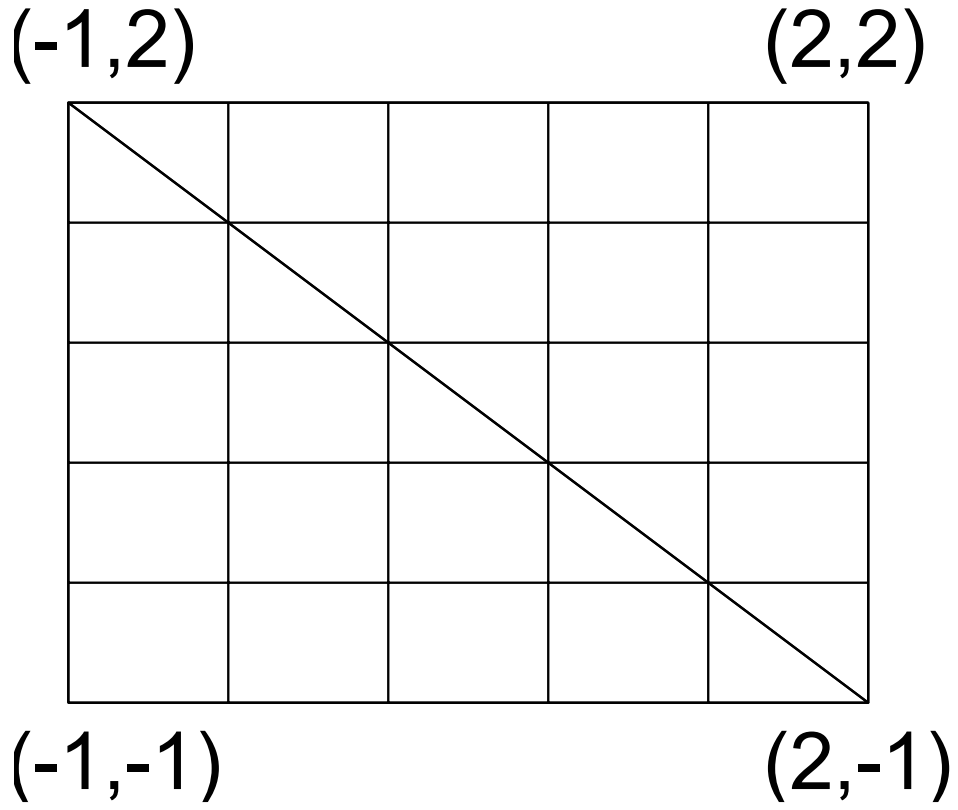
Texture Wrapping

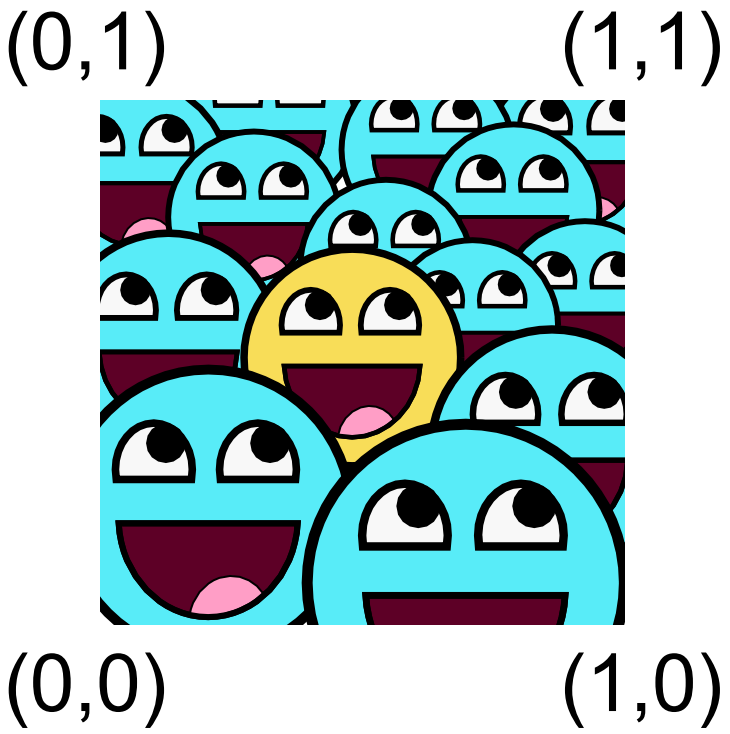
Texture Wrapping

- TC usually range from (0,0) to (1,1), what if coordinates outside this range
- Default behavior of OpenGL: repeat the texture images, but there are more options OpenGL offers:
  - GL_REPEAT: The default behavior 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 (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 color

Texture Wrapping

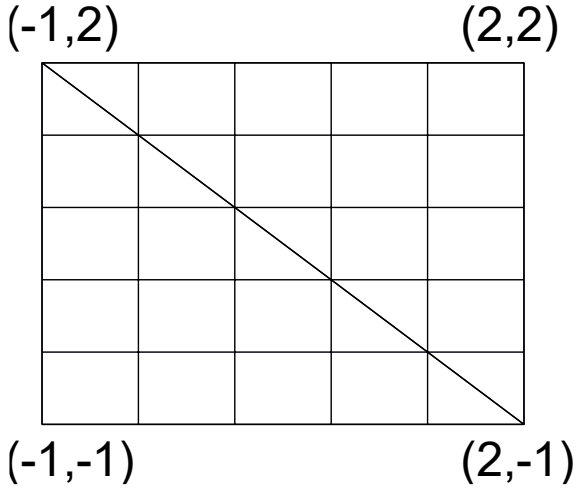
- Left TC and right the image to be mapped

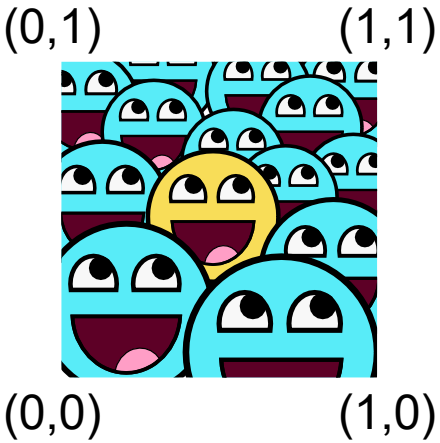


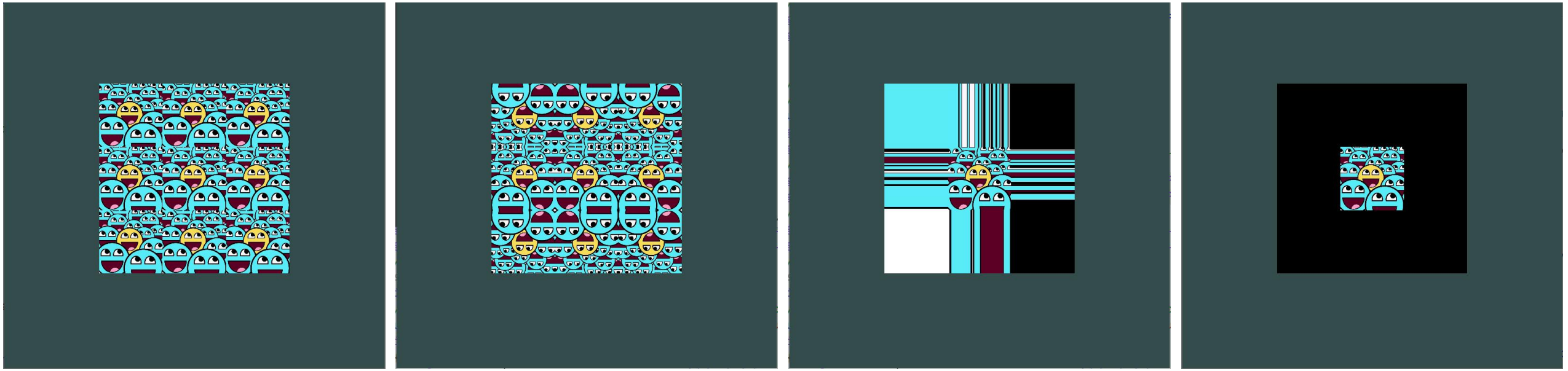


Texture Wrapping

- Different settings







GL_REPEAT GL_MIRRORED_REPEAT GL_CLAMP_TO_EDGE GL_CLAMP_TO_BORDER

Texture Wrapping

- Options can be set per coordinate axis (s, t (and r for 3D textures) equivalent to x,y,z) with the glTexParameter* function:

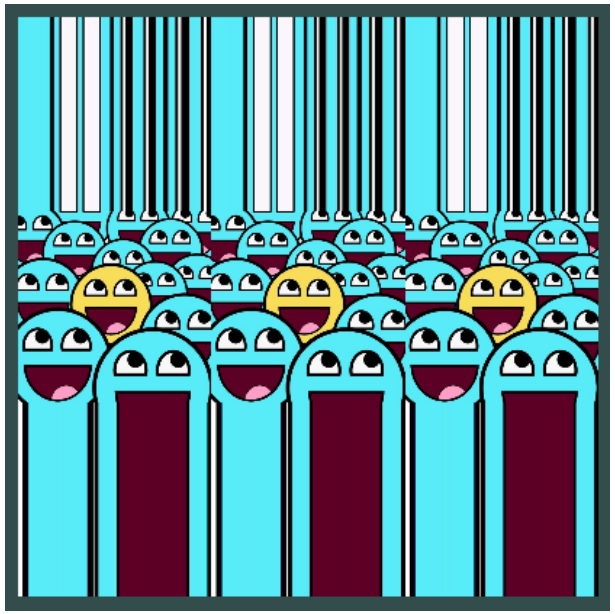
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

- First argument: specifies the texture target (2D textures → GL_TEXTURE_2D)
- Second argument: option we want to set and for which texture axis: configure the WRAP option and specify it for both the S and T axis.
- Last argument: texture wrapping mode

Texture Wrapping

- Set different modes:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```



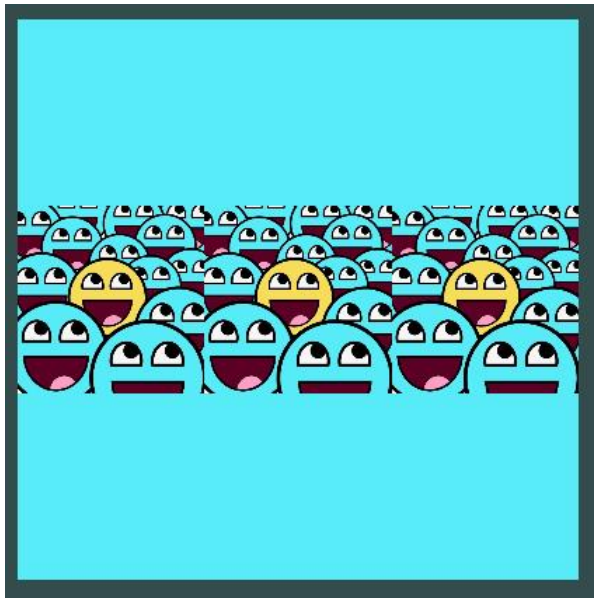
Texture Wrapping

- If GL_CLAMP_TO_BORDER option is used, a border color should be defined:
- This is done using the fv equivalent of the glTexParameter function with GL_TEXTURE_BORDER_COLOR:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);

float borderColor[] = { 88 / 255.0f, 236 / 255.0f, 248 / 255.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

Texture Wrapping

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);

float borderColor[] = { 88 / 255.0f, 236 / 255.0f, 248 / 255.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

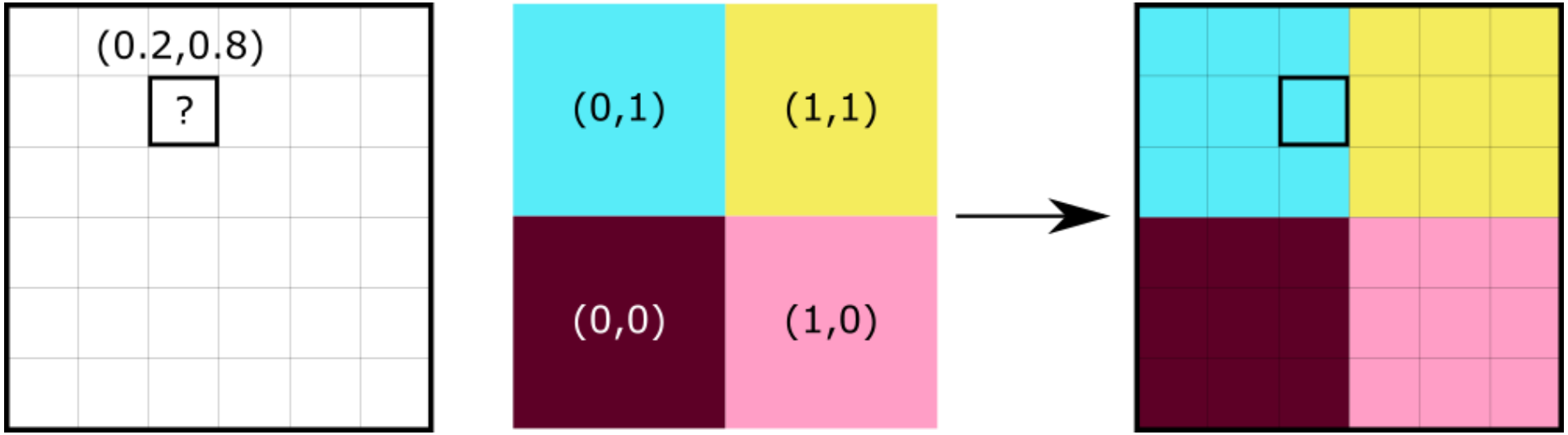


Texture Filtering

- TC do not depend on resolution, but can be any floating point value
- OpenGL has to figure out which texture pixel (also known as a texel) to map the texture coordinate to
- Especially important if you have a very large object and a low resolution texture
- OpenGL has options for this texture filtering
- The most important options: GL_NEAREST and GL_LINEAR

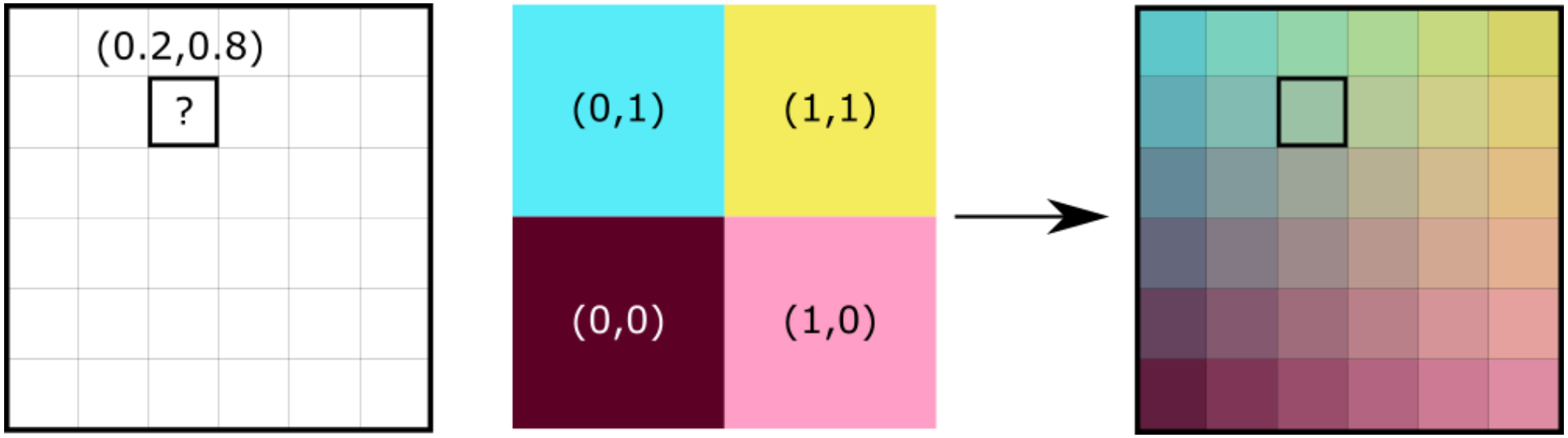
Texture Filtering

- **GL_NEAREST** (nearest neighbor filtering):
  - Default texture filtering method
  - Selects the pixel which center is closest to the TC



Texture Filtering

- **GL_LINEAR** ((bi)linear filtering):
  - Interpolated value from the TC's neighboring texels, approximating a color between the texels



Texture Filtering

GL_NEAREST

GL_LINEAR

Texture Filtering

- GL_NEAREST results in blocked patterns
- GL_LINEAR produces a smoother pattern where the individual pixels are less visible
- GL_LINEAR produces a more realistic output, but some developers prefer a pixel look and pick the GL_NEAREST option

GL_NEAREST

GL_LINEAR

Texture Wrapping

- Texture filtering can be set for magnifying and minifying operations (when scaling up or downwards)
- For example use nearest neighbor filtering when textures are scaled downwards and linear filtering for upscaled textures
- Specify the filtering method for both options via glTexParameter*:

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

Mipmaps

Introduction

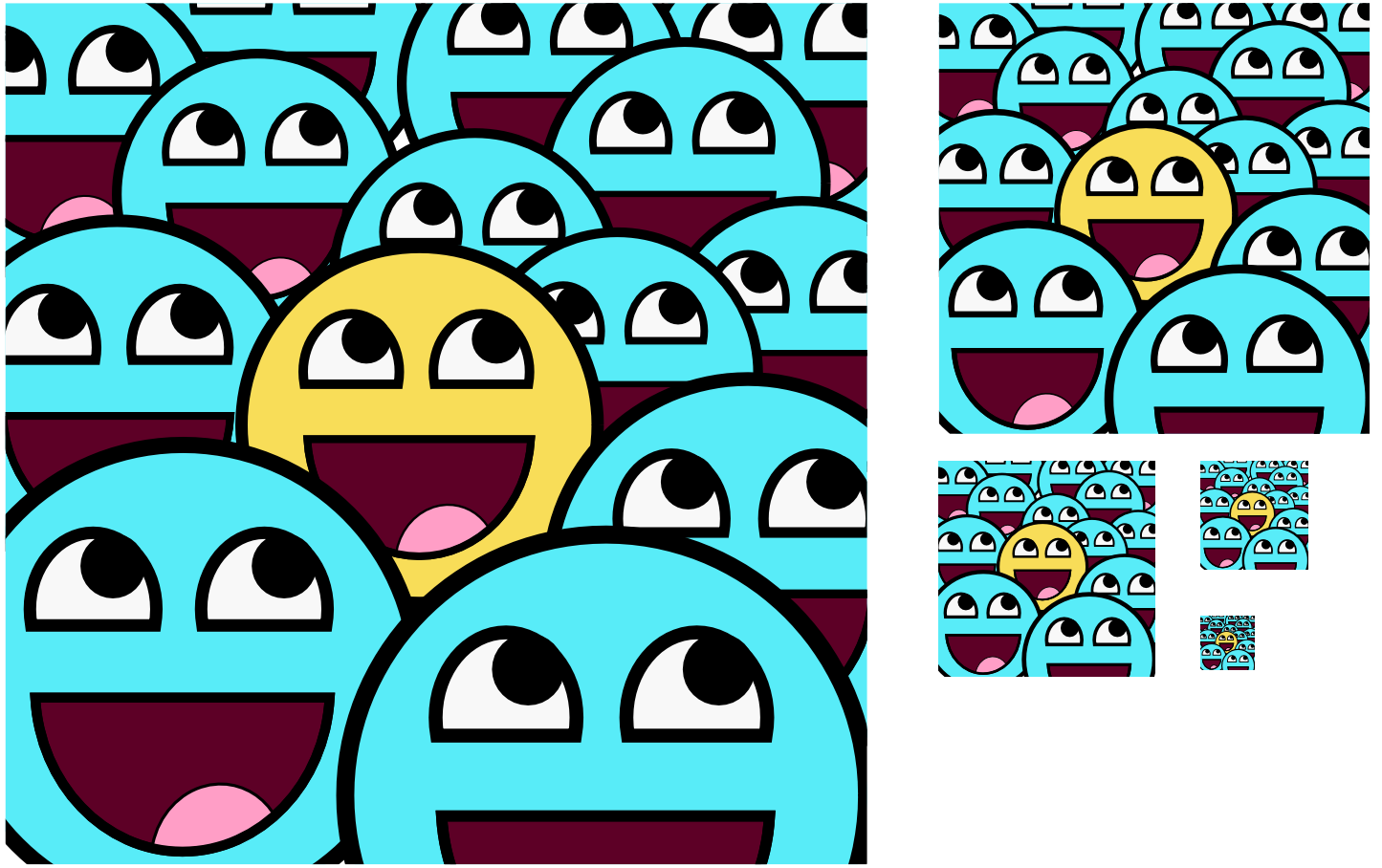
- Imagine we render a large room with thousands of textured objects
- Some objects are far away, but have the same high resolution texture attached as the objects close to the viewer
- The objects, which are far away produce a few fragments and OpenGL has difficulties retrieving the right color from the high resolution texture
- A fragment spans a large part of the texture → produce visible artifacts on small objects
- Furthermore, it is a waste of memory to use high resolution textures on small objects

Introduction

- To solve this, OpenGL uses mipmaps: a collection of texture images where each subsequent texture is twice as small
- Idea: after a certain distance threshold from the viewer, OpenGL will use a different mipmap texture that best suits the distance to the object
- The far away the object, the smaller the resolution (not noticeable to the user)
- Mipmaps are good for performance

Introduction

- Creating a collection of mipmapped textures for each texture image is cumbersome
- Luckily OpenGL is able to do it via glGenerateMipmaps after the created texture (more on this later)



Introduction

- Some artifacts may show up, when switching between two mipmap levels (like sharp edges)
- Like normal texture filtering, it is also possible to filter between mipmap levels using NEAREST and LINEAR filtering for switching between mipmap levels

Introduction

- To specify the filtering method between mipmap levels, four options are available:
  - GL_NEAREST_MIPMAP_NEAREST: takes the nearest mipmap to match the pixel size and uses nearest neighbor interpolation for texture sampling
  - GL_LINEAR_MIPMAP_NEAREST: takes the nearest mipmap level and samples using linear interpolation
  - GL_NEAREST_MIPMAP_LINEAR: linearly interpolates between the two mipmaps that most closely match the size of a pixel and samples via nearest neighbor interpolation
  - GL_LINEAR_MIPMAP_LINEAR: linearly interpolates between the two closest mipmaps and samples the texture via linear interpolation

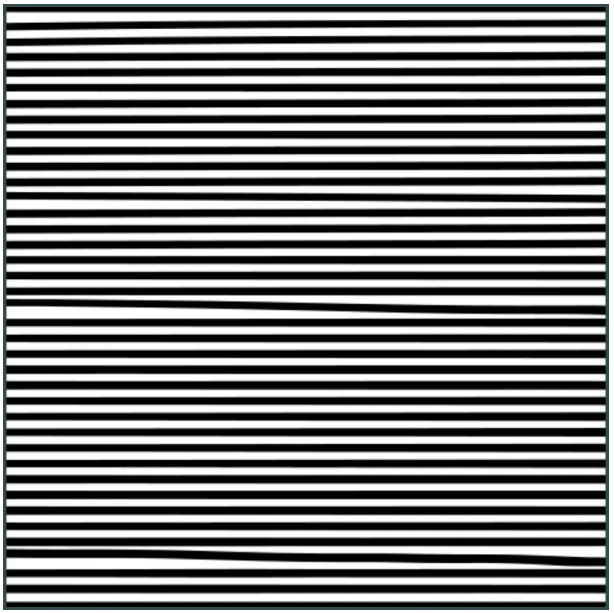
Texture Wrapping

- Similar to texture filtering, set filtering method to one of the 4 aforementioned methods using glTexParameteri:

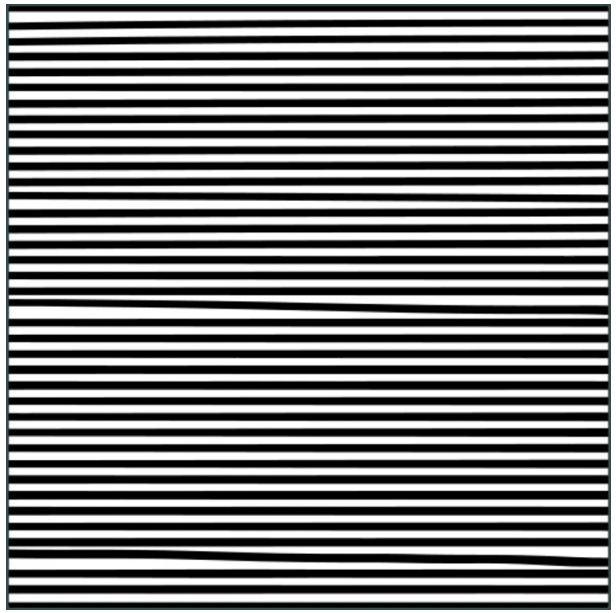
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

- A common mistake is to set one of the mipmap filtering options as the magnification filter (doesn't have any effect since mipmaps are primarily used for when textures get downscaled)
- Texture magnification doesn't use mipmaps and giving it a mipmap filtering option will generate an OpenGL GL_INVALID_ENUM error code

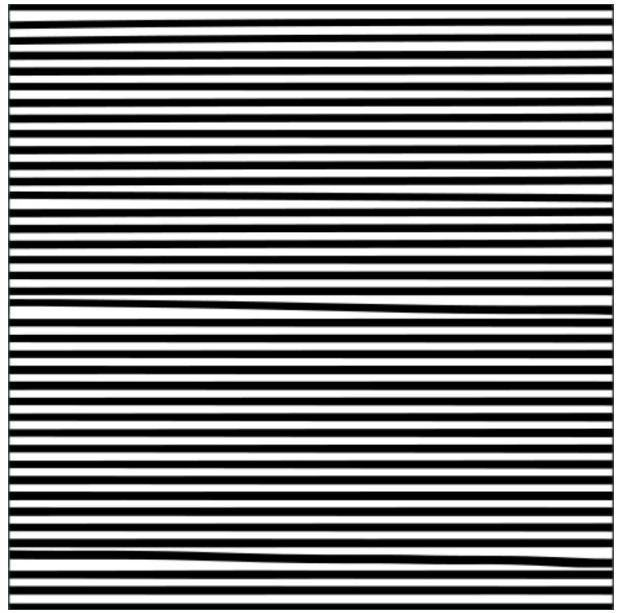
Comparison



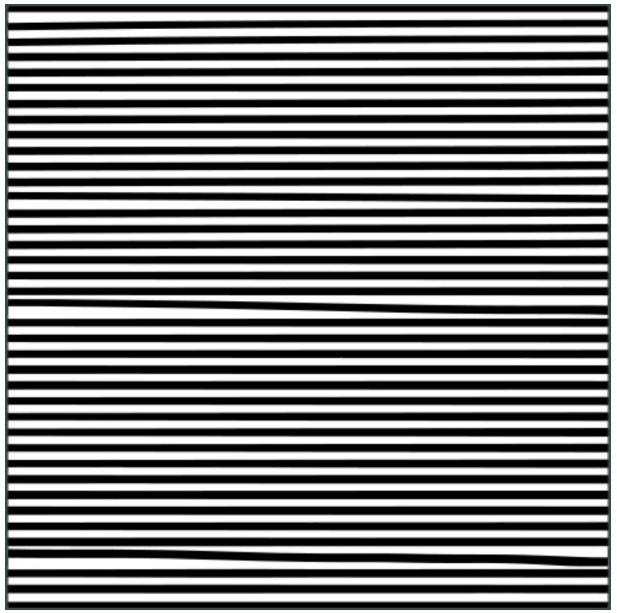
GL_NEAREST_MIPMAP_NEAREST



GL_LINEAR_MIPMAP_NEAREST



GL_NEAREST_MIPMAP_LINEAR



GL_LINEAR_MIPMAP_LINEAR

Loading and Creating Textures

Loading and Creating Textures

- The first thing to do: load textures into the application
- Texture images can be stored in dozens of file formats, each with their own structure and ordering of data
- To load them, one solution is to choose a file format, e.g., PNG, and write an image loader
- Not very hard, but cumbersome (what about other formats?)
- Then write an image loader for each format

- → We use the library `stb_image.h`

stb_image.h

- stb_image.h is a popular single header image loading library
- Able to load most popular file formats
- Easy to integrate in your project(s)
- Download the single header file, add it to your project as stb_image.h and create an additional C++ file with the following code:

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

https://github.com/nothings/stb/blob/master/stb_image.h

stb_image.h

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

- #define STB_IMAGE_IMPLEMENTATION: preprocessor modifies the header file such that it only contains the relevant definition source code, effectively turning the header file into a .cpp file
- Then include stb_image.h
- To load an image using stb_image.h, use its stbi_load function:

```
int width, height, nrChannels;
stbi_set_flip_vertically_on_load(true); // flip loaded texture's on the y-axis.
unsigned char* data = stbi_load("texture.jpg", &width, &height, &nrChannels, 0);
```

stb_image.h

```
int width, height, nrChannels;
stbi_set_flip_vertically_on_load(true); // flip loaded texture's on the y-axis.
unsigned char* data = stbi_load("texture.jpg", &width, &height, &nrChannels, 0);
```

- Need to flip the y-axis
- First argument location of an image file
- Then three ints: width, height and number of color channels of the image
- Last argument forces number of channels (set to 0 to keep nrChannels)

Generating a Texture

- Textures are referenced with an ID (Like any previous objects)
- glGenTextures: first input number of texture names to be generated
- stores, second stores them in a unsigned int array
- Finally, binding so any subsequent texture commands will configure the currently bound texture:

```
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
```

Generating a Texture

- After texture bound, generate a texture using the previously loaded image data
- Textures are generated with glTexImage2D:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D);
```

Generating a Texture

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
```

- 1. specifies texture target; GL_TEXTURE_2D → generate a texture on the currently bound texture object at the same target (other textures bound to targets GL_TEXTURE_1D or GL_TEXTURE_3D will not be affected)
- 2. specifies the mipmap level 0 means base (interesting manual mipmap levels)
- 3. tells format of the texture (image has only RGB values → GL_RGB)
- 4./5. sets the width and height of the resulting texture
- 6. should always be 0 (some legacy stuff) (khronos states the same)
- 7./8. specify the format and datatype of the source image (loaded the image with RGB values and stored them as chars (bytes))
- 9. actual image data

Generating a Texture

- Note: OpenGL assembles textures automatically into an RGBA
- E.g., using GL_RED: “GL converts it to floating point and assembles it into an RGBA element by attaching 0 for green and blue, and 1 for alpha. Each component is clamped to the range [0,1].”

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml

Generating a Texture

```
glGenerateMipmap(GL_TEXTURE_2D);
```

- After glTexImage2D is called, the currently bound texture object now has the texture image attached to it
- It only has the base-level of the texture image loaded, for mipmaps either set it manually or call glGenerateMipmap after generating the texture
- Automatically generate all the required mipmaps for the currently bound texture

Generating a Texture

- Finally, it is good practice to free the image memory:

```
stbi_image_free(data);
```

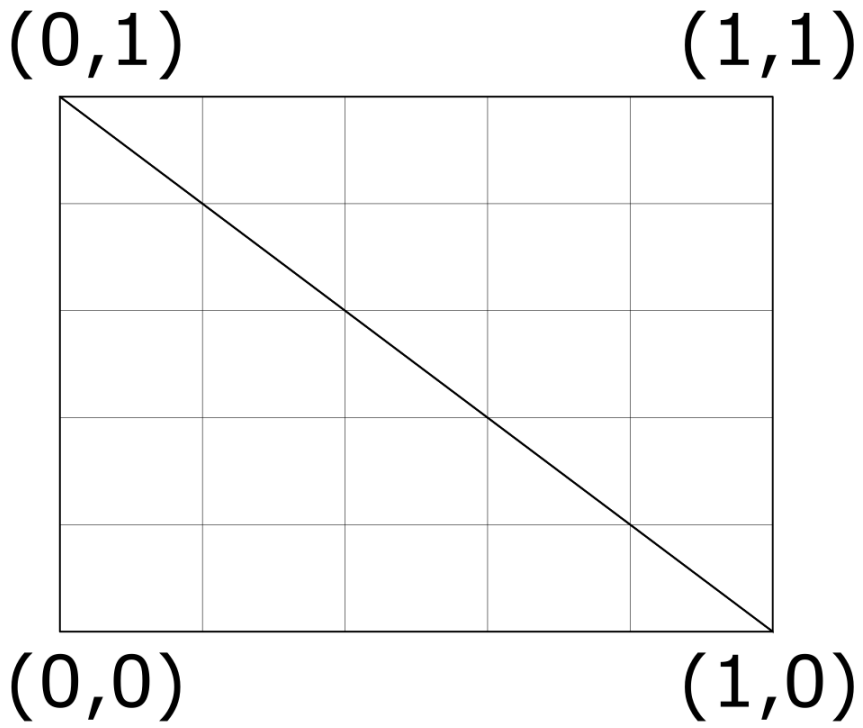
All together

```
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
// set texture filtering parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// load image, create texture and generate mipmaps
int width, height, nrChannels;
stbi_set_flip_vertically_on_load(true); // flip loaded texture's on the y-axis.
unsigned char *data = stbi_load("texture.jpg", &width, &height, &nrChannels, 0);
if (data)
 {
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
   GL_UNSIGNED_BYTE, data);
   glGenerateMipmap(GL_TEXTURE_2D);
 }
else
 {
   std::cout << "Failed to load texture" << std::endl;
 }
stbi_image_free(data);
```

Applying Textures

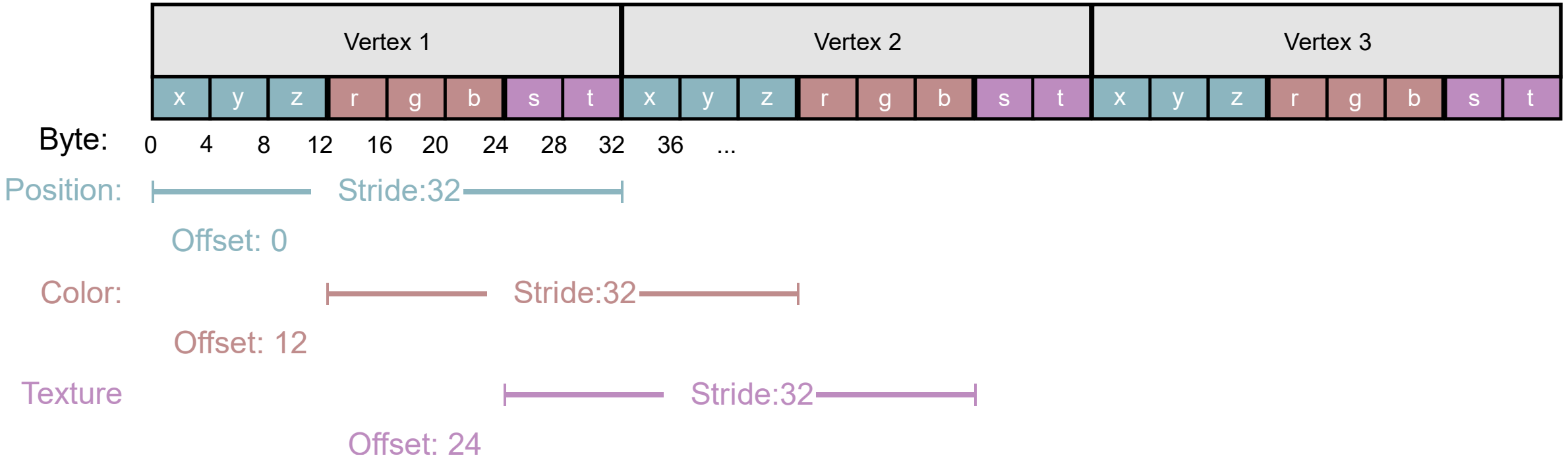
- We will use the rectangle shape drawn with glDrawElements
- Again:

```
float vertices[] = {
     // positions        // colors          // texture coords
     0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f,
     0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f
    };
    unsigned int indices[] = {
     0, 1, 3, // first triangle
     1, 2, 3  // second triangle
    };
```

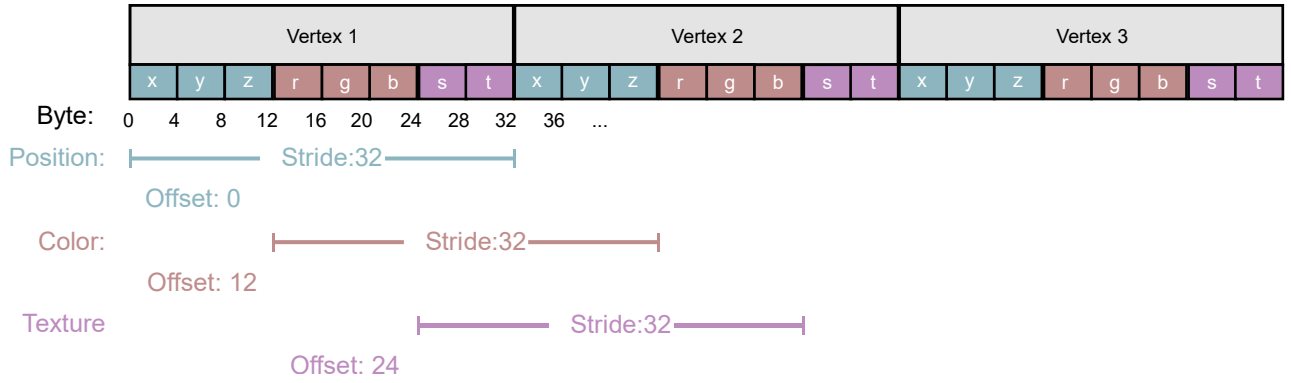


Applying Textures

- New vertex format:



Applying Textures



- This results in:

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3
* sizeof(float)));
glEnableVertexAttribArray(1);
// texture coord attribute
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6
* sizeof(float)));
glEnableVertexAttribArray(2);
```

Applying Textures

- Change the vertex shader:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;

out vec3 ourColor;
out vec2 TexCoord;

void main()
{
   gl_Position = vec4(aPos, 1.0);
   ourColor = aColor;
   TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

Applying Textures

- GLSL has a built-in data-type for texture objects: sampler

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

// texture sampler
uniform sampler2D texture1;

void main()
{
    FragColor = texture(texture1, TexCoord);
}
```

Applying Textures

```
FragColor = texture(texture1, TexCoord);
```

- To sample the color of a texture, use the texture function:
- 1. argument: a texture sampler
- 2. argument: the corresponding texture coordinate.
- Texture function samples the corresponding color value using the texture parameters we set earlier
- Output of this fragment shader is then the (filtered) color of the texture at the (interpolated) texture coordinate

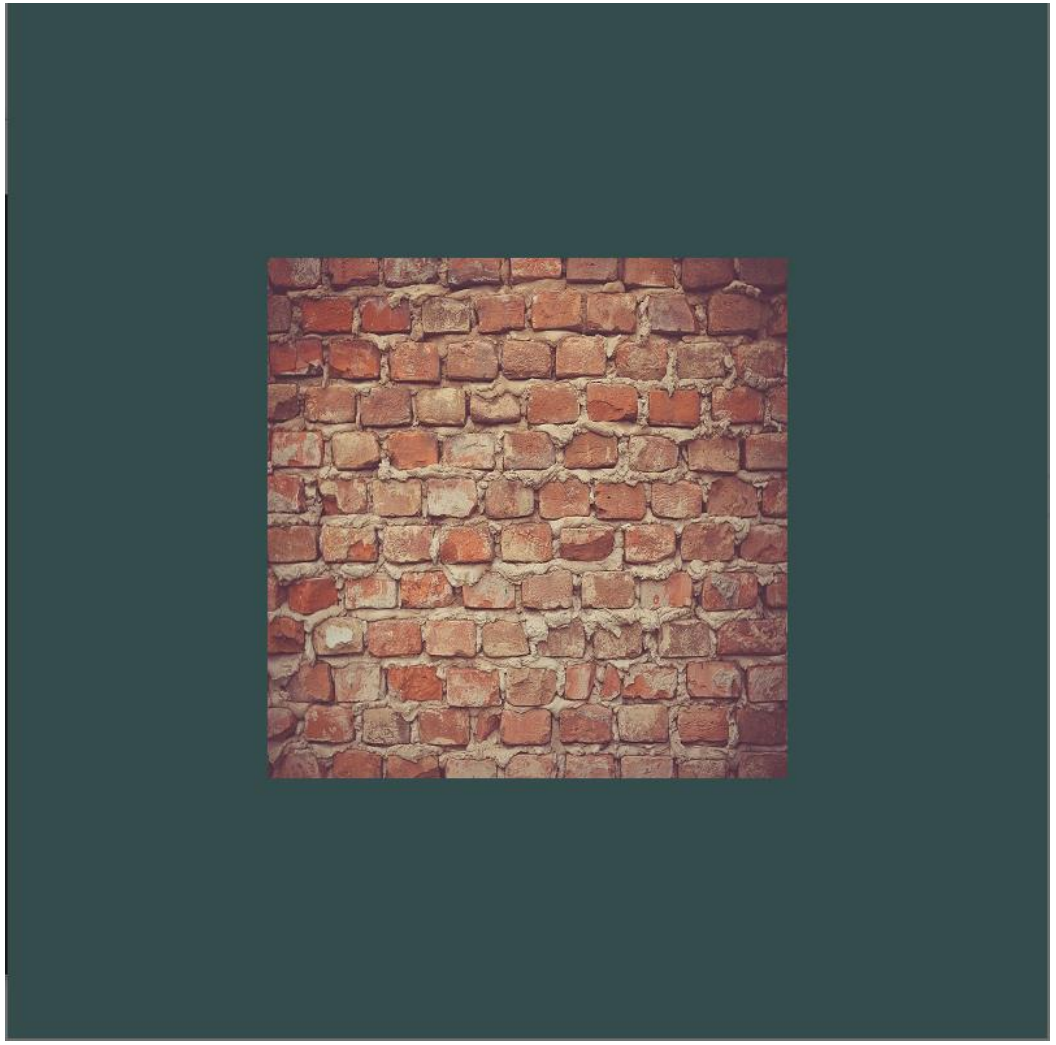
Applying Textures

- Last, bind the texture before calling the glDrawElements and it will then automatically assign the texture to the fragment shader's sampler:

```
glBindTexture(GL_TEXTURE_2D, texture);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

F5…

… a rectangle with a texture!



Add Colors

- Now, we want to use the color as well (fragment shader):

```
FragColor = texture(texture, TexCoord)*vec4(ourColor, 1.0);
```

Texture Units

Texture Units

- Why is the sampler2D variable a uniform?
- Didn’t even assign it some value with glUniform
- With glUniform1i a location value can be assigned to the texture sampler: multiple textures can be set at once in a fragment shader
- This location of a texture is known as a texture unit
- The default texture unit for a texture is 0 (default active texture unit)
- Note: not all graphics drivers assign a default texture unit → previous section might not have rendered

Texture Units

- Texture units allow to use more than 1 texture in shaders
- By assigning texture units to the samplers, multiple textures can be bind at once as long as they activate the correspondin texture unit first
- Like glBindTexture, activate texture units using glActiveTexture:

```
// bind textures on corresponding texture units
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture0);
```

- After activating a texture unit, a subsequent glBindTexture call will bind that texture to the currently active texture unit
- (GL_TEXTURE0 is always by default activated)

Texture Units

OpenGL should have a at least a minimum of 16 texture units to use. Can be activated using GL_TEXTURE0 to GL_TEXTURE15.

They are defined in order so we could also get GL_TEXTURE8 via GL_TEXTURE0 + 8 (useful for loop over several texture units).

Texture Units

- Need to edit the fragment shader to accept another sampler:

```
#version 330 core
...
uniform sampler2D ourTexture1;
uniform sampler2D ourTexture2;
void main()
{
FragColor = mix(texture(ourTexture1, TexCoord), texture(ourTexture2, TexCoord), 0.2);
}
```

Texture Units

- Final output color is now the combination of two texture lookups
- GLSL's built-in mix function: takes two values as input and linearly interpolates between them based on its third argument:

$$mix(x, y, a) = x \cdot (1 - a) + y \cdot a$$

Texture Units

- Now, create, load and generate another texture using glTexImage2D
- To use the second texture (and the first texture), change the rendering procedure a bit by binding both textures to the corresponding texture unit:

```
// bind textures on corresponding texture units
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture2);

glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Texture Units

- Tell OpenGL to which texture unit each shader sampler belongs to by setting each sampler using glUniform1i
- Set this once, so can do this before the render loop:

```
ourShader.use();
// either set it manually like so:
glUniform1i(glGetUniformLocation(ourShader.ID, "texture1"), 0);
// or set it via the texture class
ourShader.setInt("texture2", 1);

while (…)
{
    […]
}
```

F5…

… faces on brick!

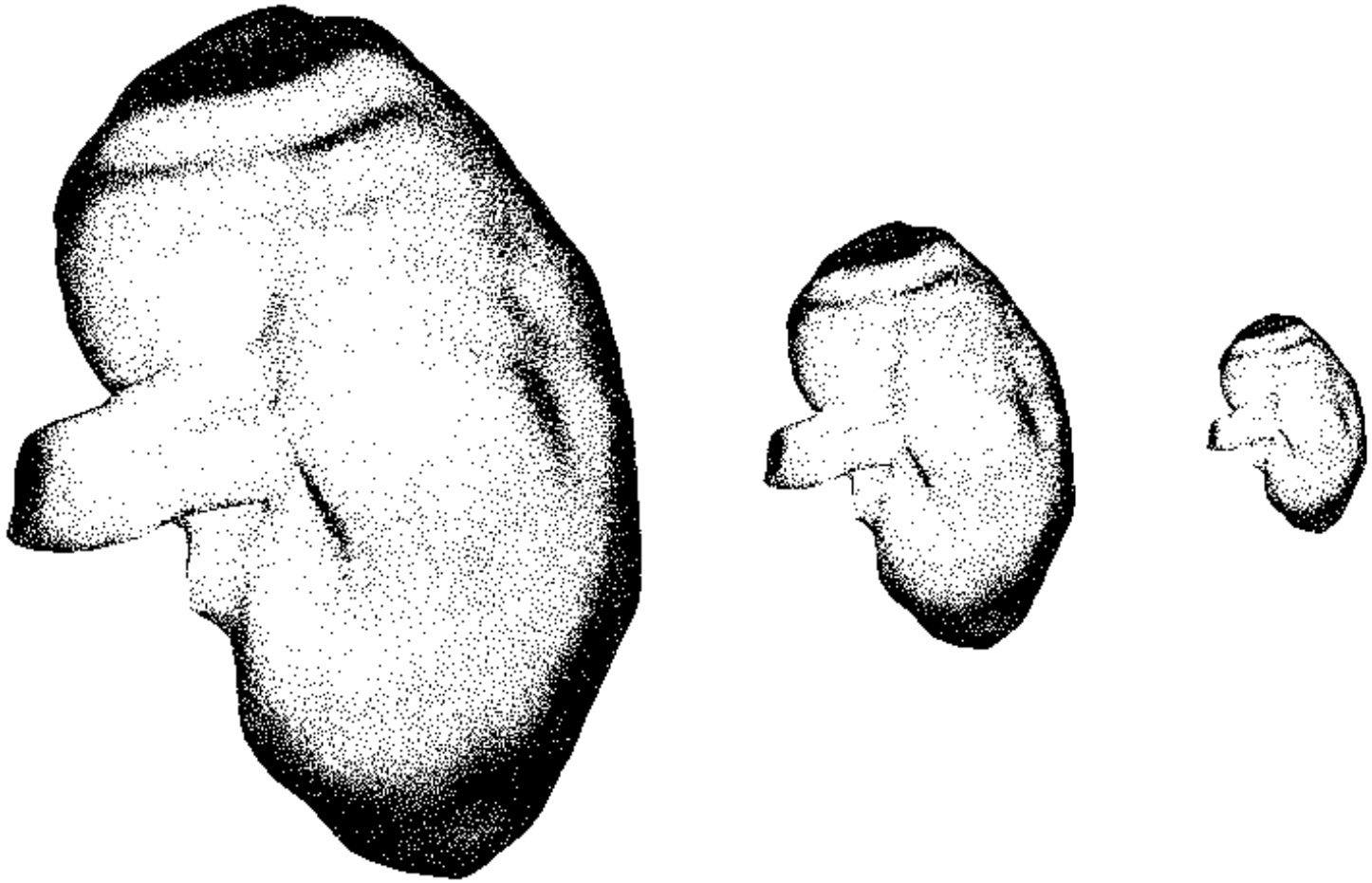
Mipmaps II*

Introduction

- Sometimes it is necessary to generate the mipmaps manually
- Automatically generated mipmaps may fail for certain textures

Example

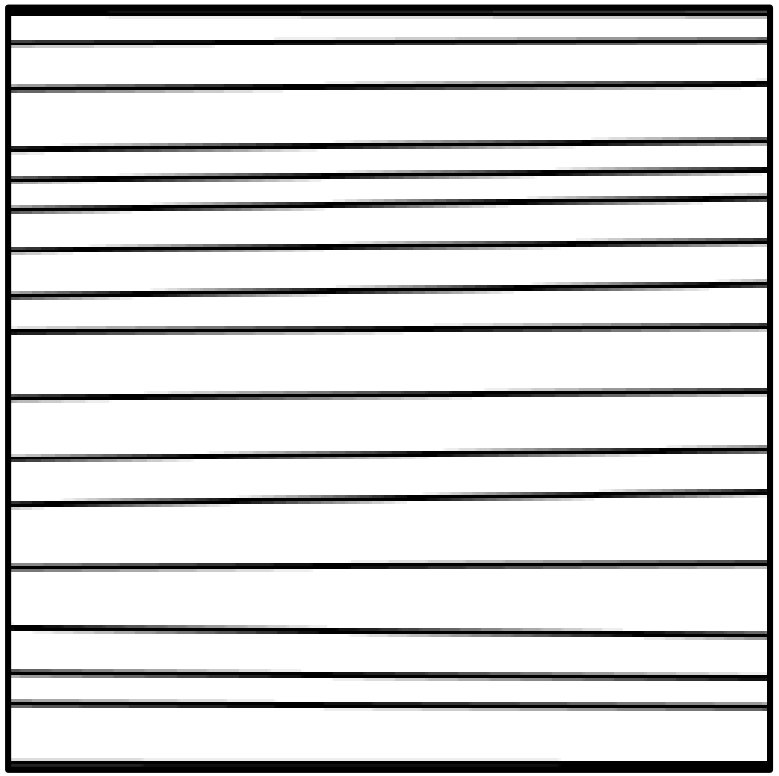
- Manually generated mipmap textures

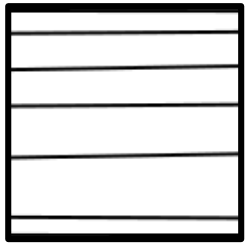


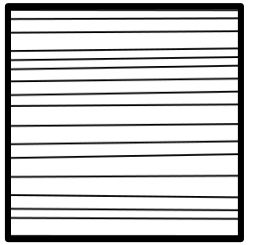
Baer et. al / Hardware-accelerated Stippling of Surfaces derived from Medical Volume Data

Example

- Top – Original
- Left – Manually generated
- Right – GL generated

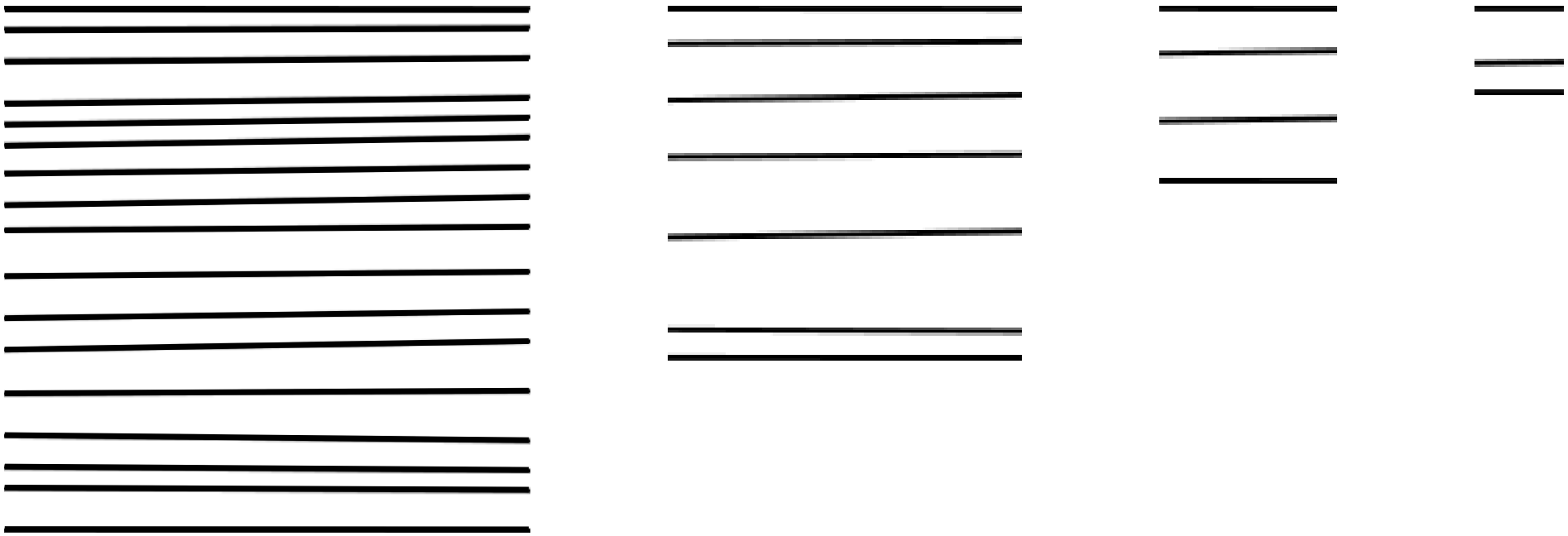






Introduction

- Generate an image pyramid (256x256, 128x128, 64x64, 32x32 pxs):



Set Up

- Set the max level of the mipmap pyramid, the base layer is 0:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// set pyramid level (no. of images - 1)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 3);
```

Set Up

- Load the images (we set the width and height manually later, do not worry about over writing):

```
unsigned char* data  = stbi_load("ln1.jpg", &width, &height, &nrChannels, 0);
unsigned char* data2 = stbi_load("ln2.jpg", &width, &height, &nrChannels, 0);
unsigned char* data3 = stbi_load("ln3.jpg", &width, &height, &nrChannels, 0);
unsigned char* data4 = stbi_load("ln4.jpg", &width, &height, &nrChannels, 0);
```

Set Up

- Specify the 2D textures and set the mipmap level (2nd argument):

```
if (data && data2 && data3 && data4)
{
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE,
data);
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGB, 128, 128, 0, GL_RGB, GL_UNSIGNED_BYTE,
data2);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGB, 64, 64, 0, GL_RGB, GL_UNSIGNED_BYTE,
data3);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGB, 32, 32, 0, GL_RGB, GL_UNSIGNED_BYTE,
data4);
//glGenerateMipmap(GL_TEXTURE_2D);
}
```

Make an Animation

- We upload a uniform (in the render loop):

```
float time = glfwGetTime();
ourShader.use();
ourShader.setFloat("time", time);
```

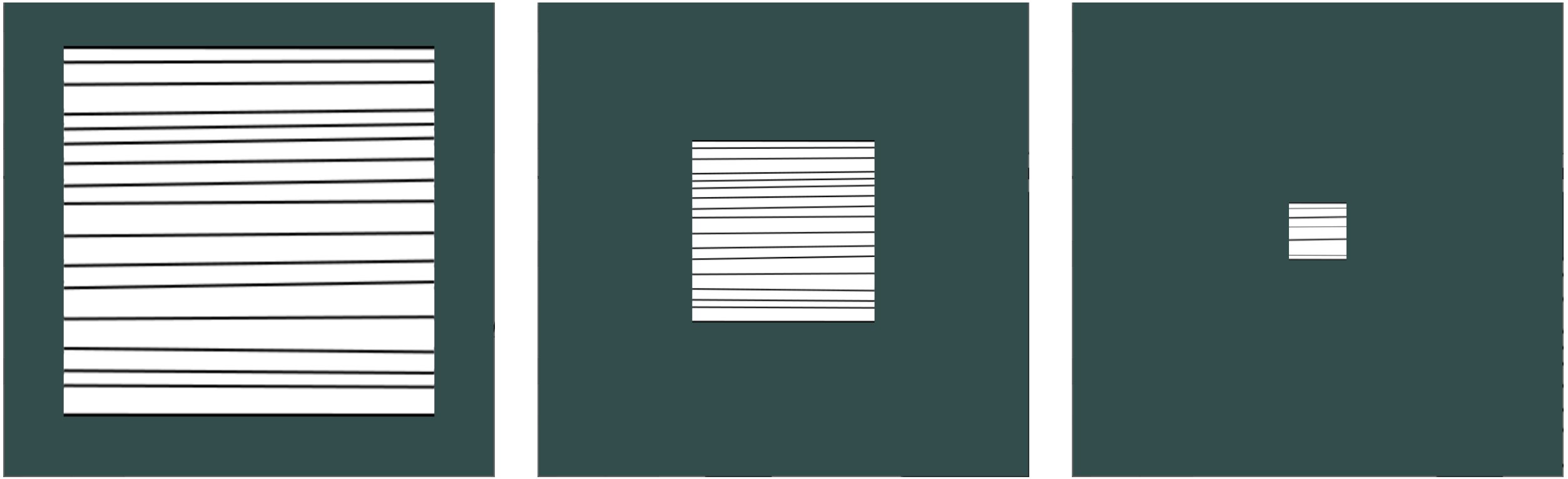
Make an Animation

- Vertex shader:

```
#version 330 core
layout (location = 0) in vec3 aPos;
…

uniform float time;

void main()
{
	float t=sin(time/3)*0.5+0.5;
	vec2 dir=vec2(0)-aPos.xy;

	gl_Position.xy=aPos.xy+t*dir;
…
}
```

F5…

… we get an animation with manually generated mipmaps



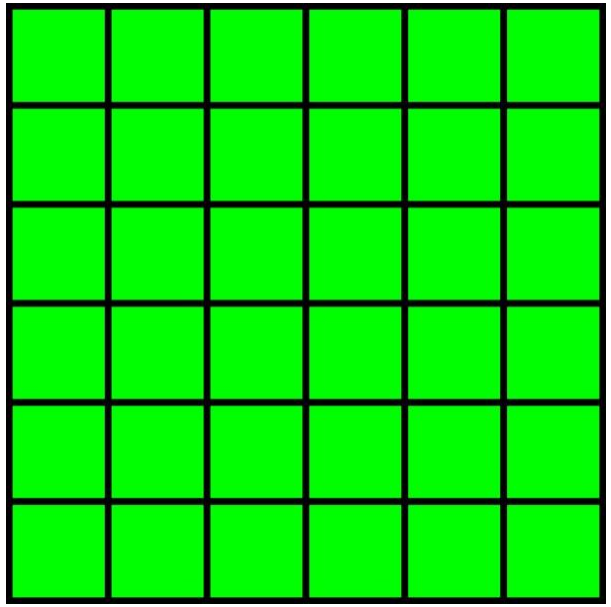
Discard*

Introduction

- Sometimes, we do not want to draw stuff in the fragment shader
- What if we are interested in objects behind a certain object, e.g., if we load a window texture, we do not want the glass texture drawn over the stuff behind it

Introduction

- Let’s assume we have the following texture
- We only want to draw the grid, but we want to omit the green part



Discard

- First, create a Boolean vector:

```
vec4 col = texture(texture1, TexCoord);
bvec3 greenColor = bvec3(col.r < 0.1, col.g>0.9, col.b<0.1);
```

- Bvec3 is a three dimensional vector
- Because we deal with a jpg image, we use thresholds for the green color otherwise we could ask if the color is identical to (0,1,0)

Discard

- If all components of the bvec3 are true, we discard the fragment otherwise, we assign it to red:

```
if(all(greenColor))
   discard;
 else
   FragColor.rgb=vec3(1,0,0);
```

- The function all(.) returns true if all components are true, too

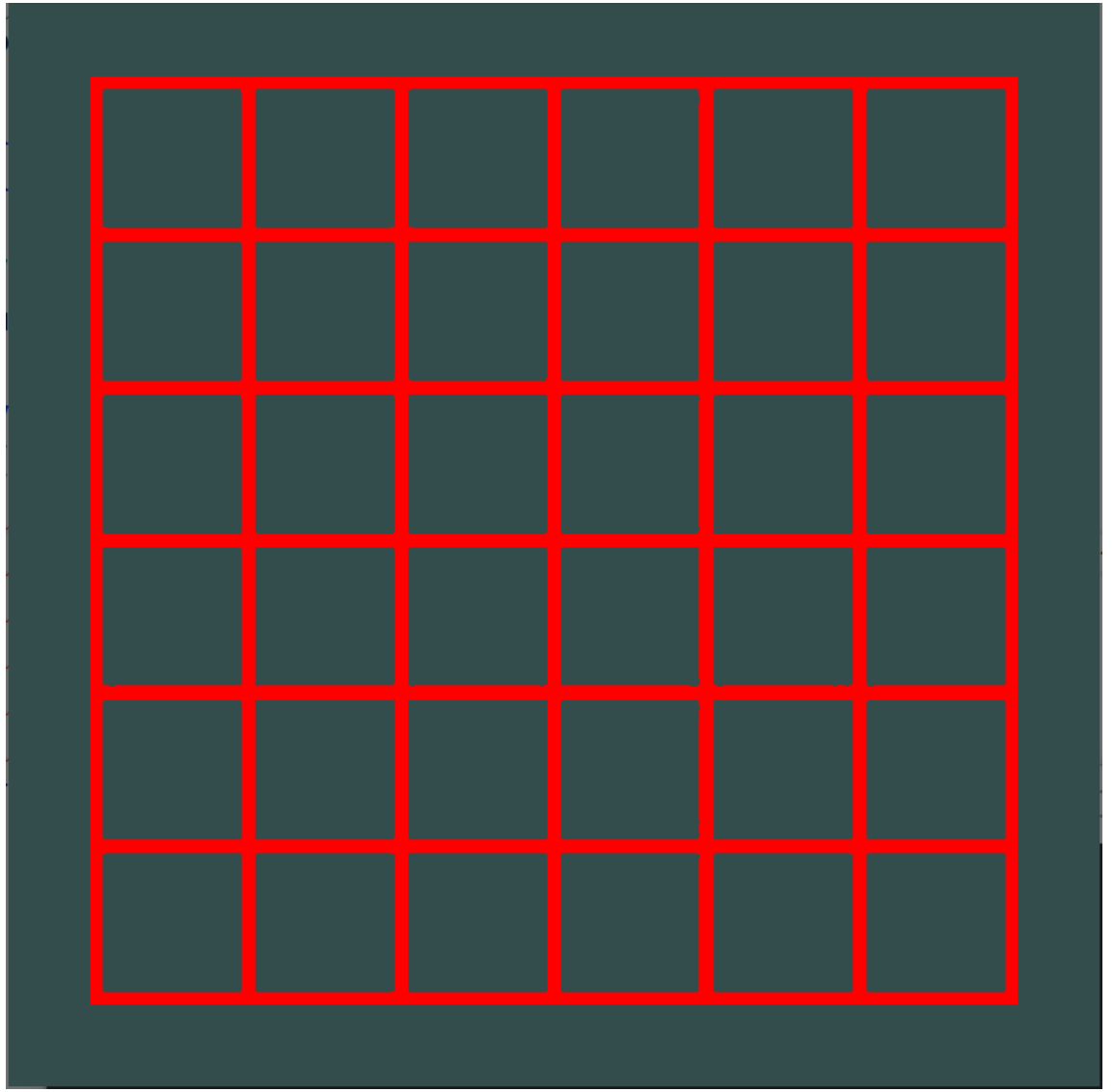
Discard

- Final fragment shader:

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoord;
uniform sampler2D texture1;

void main()
{
  vec4 col = texture(texture1, TexCoord);
  bvec3 greenColor = bvec3(col.r < 0.1, col.g>0.9, col.b<0.1);
  if(all(greenColor))
    discard;
  else
    FragColor.rgb=vec3(1,0,0);
}
```

F5…

… we get a red grid



Take a Screenshot*

Introduction

- A basic feature for a graphics tool is to take a screenshot
- We want to press a key, e.g., ‘s’ and this results in a screenshot, which will be stored on the hard drive

Write PNG

- We already met the stb_image.h header to load images
- But, we also need a library that saves images: `stb_image_write.h`
- Download the library and include it in the project
- Do not forget #define… as this is necessary for the usage

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include <stb_image_write.h>
```

https://github.com/nothings/stb/blob/master/stb_image_write.h

Press ‚S‘

- We extent the processInput function to check whether the key ‘S’ was pressed:

```
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {…}
}
```

Press ‚S‘

- This code captures the window and saves the image as ‘screenshot.png’

```
if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {
        int width, height;
        glfwGetWindowSize(window, &width, &height);

        GLubyte* screen = new GLubyte[width * height * 4];

        glReadPixels(0, 0, width, height, GL_RGBA, GL_UNSIGNED_BYTE, screen);
        stbi_write_png("screenshot.png", width, height, 4, screen, 0);
        delete[] screen;
    }
```

Press ‚S‘

```
glReadPixels(0, 0, width, height, GL_RGBA, GL_UNSIGNED_BYTE, screen);
```

- glReadPixels (arguments):
  - 1./2.: window coordinates of the first pixel (location is the lower left corner)
  - 3./4.: dimensions of the window (bottom right)
  - 5.: format of the pixel data
  - 6.: data type
  - 7.: returns the pixel data

Press ‚S‘

```
stbi_write_png("screenshot.png", width, height, 4, screen, 0);
```

- stbi_write_png (arguments):
  - 1.: filename
  - 2./3.: width, height of the screenshot
  - 4.: components (4 → RGBA)
  - 5. pixel data
  - 6. stride

~~F5…~~ S…

…and we get a nice screenshot saved on the hard drive

Questions???