(4) Textures

GPU Programming

Thorsten Grosch

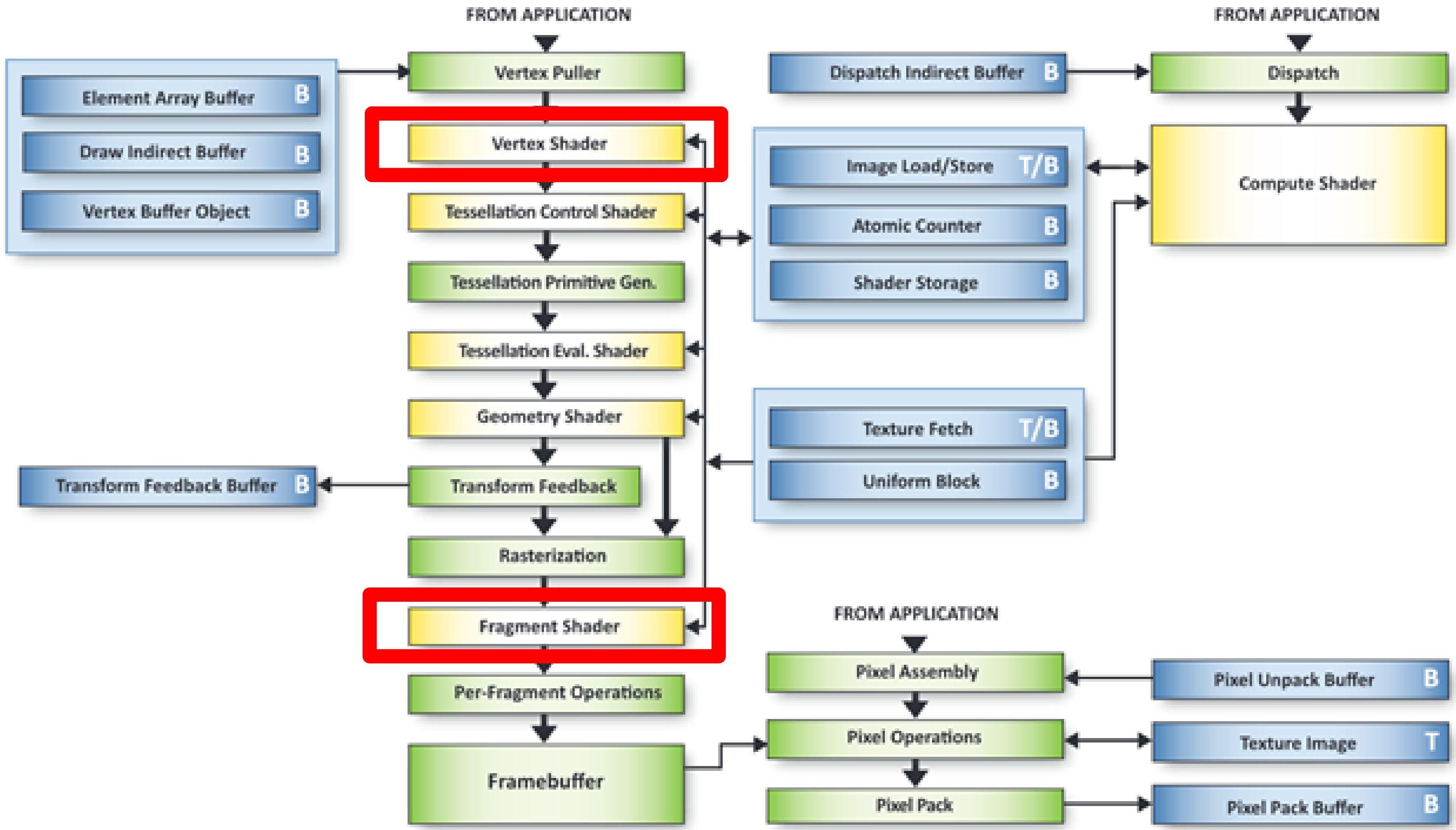
TU Clausthal

Today

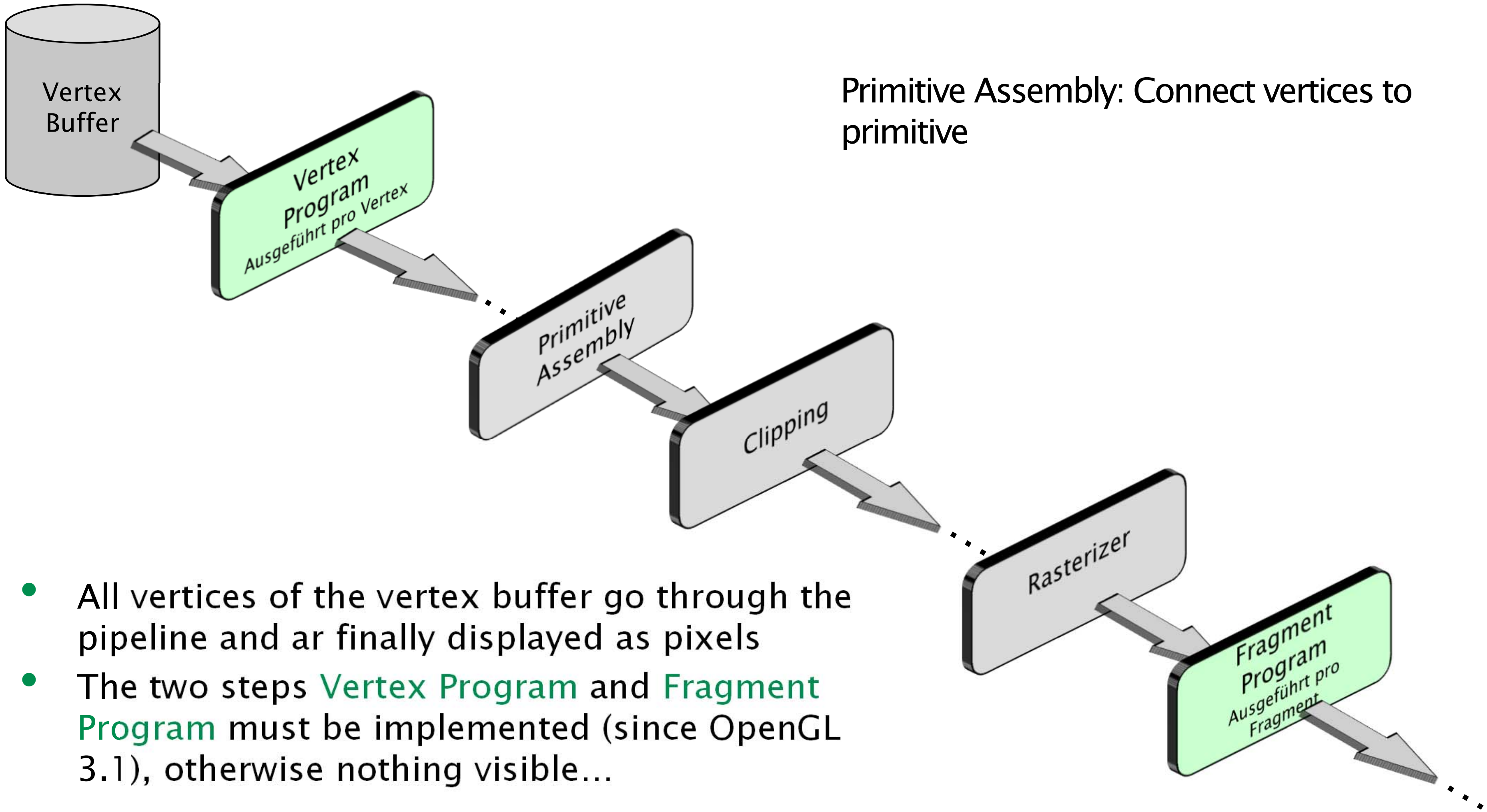
- Have a look at textures and sampler objects

- Repeat the basics about GLSL

OpenGL 4.4


FROM APPLICATION
Vertex Puller
Element Array Buffer B
Draw Indirect Buffer B
Vertex Buffer Object B
Vertex Shader
Tessellation Control Shader
Tessellation Primitive Gen.
Tessellation Eval. Shader
Geometry Shader
Transform Feedback Buffer B
Transform Feedback
Rasterization
Fragment Shader
Per-Fragment Operations
Framebuffer
Dispatch Indirect Buffer B
FROM APPLICATION
Dispatch
Image Load/Store T/B
Atomic Counter B
Shader Storage B
Compute Shader
Texture Fetch T/B
Uniform Block B
FROM APPLICATION
Pixel Assembly
Pixel Unpack Buffer B
Pixel Operations
Texture Image T
Pixel Pack
Pixel Pack Buffer B

Simplified OpenGL-Pipeline



Primitive Assembly: Connect vertices to primitive

- All vertices of the vertex buffer go through the pipeline and ar finally displayed as pixels
- The two steps Vertex Program and Fragment Program must be implemented (since OpenGL 3.1), otherwise nothing visible...

Generation of Shader programs

- Generate a Shader Object first

```
GLuint shader = glCreateShader(GL_VERTEX_SHADER);
```

- Insert the shader code as a simple C-string `source` into the shader object:

```
void glShaderSource(GLuint shader, GLsizei count,
GLchar **source, GLint length);
```

- We set `count = 1` and `length = NULL`

Compile a Shader

- Like a C program a shader must be compiled

```
glCompileShader(shader);
```

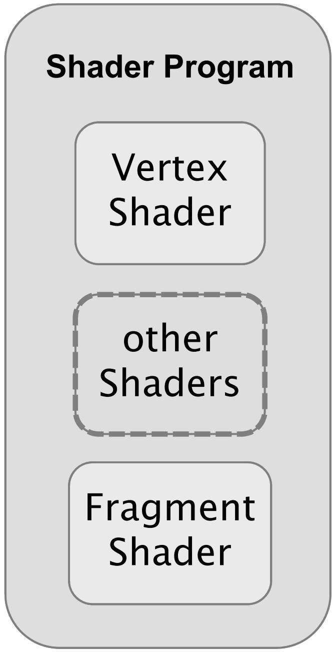
- Possible compiler errors can be requested by

```
glGetShaderInfoLog(GLuint shader, GLsizei
bufSize, GLsizei *length, char *infoLog);
```

- **`infoLog`** contains a string of length **`length`** (max. length **`bufSize`**) with error messages
- Always check this, in case of compiler errors, the shader does not work…

Assemble Shader Programs

- In GLSL we do not activate single Shaders, but complete **Shader Programs**
  - Vertex- and Fragment Shader are always activated **together** (possibly with other shaders)



- Therefore we create a Shader Program

```
GLuint program = glCreateProgram(void);
```

- Each Shader is then attached with

```
glAttachShader(program, shader);
```

  to the Shader Program

Linking Shaders

- Similar to the compilation of multiple C-Programs to one executable file, all the shaders must be linked (Object Files → Executable)

```
void glLinkProgram(GLuint program);
```

- We can check for errors here
  - A linker error can occur, e.g. if the output of the Vertex Shader does not fit to the input of the Fragment Shader

```
void glGetProgramInfoLog(GLuint program, GLsizei
bufSize, GLsizei *length, char *infoLog);
```

Activate the Shader Program

- All the attached shaders in a shader program can be activated with

```
void glUseProgram(GLuint program)
```

When drawing geometry afterwards, all the vertices go through the vertex shader and the resulting pixels go through the fragment shader.

Vertex Shader Example

```
// Create empty shader object (vertex shader)
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);

// vertex shader source
const char* sourcePtr = {
      "#version 440 core\n"
      "layout(location = 0) in vec4 vPos;"
      "void main() {"
      "       gl_Position = vPos;"
      "}"
};

// Attach shader code
glShaderSource(vertexShader, 1, &sourcePtr, NULL);
// Compile
glCompileShader(vertexShader);
```

Fragment Shader Example

```
GLuint fragmentShader =glCreateShader(GL_FRAGMENT_SHADER);

// fragment shader source
const char* sourcePtr2 = {
            "#version 440 core\n"
            "out vec4 color;"
            "void main() { color = vec4(1.0); }"
     };

// Attach shader code
glShaderSource(fragmentShader, 1, &sourcePtr2, NULL);

// Compile
glCompileShader(fragmentShader);
```

Link & Activate Example

```
GLuint shaderProgram = glCreateProgram();

// Attach shader
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);

// Link program
glLinkProgram(shaderProgram);

// activate Shader Program
glUseProgram(shaderProgram);
```

Class for Shaders

- Since the loading and activation of shaders always requires the same commands, we use a **Shader class** in the exercises
- Here you can load all types of shaders, link them, check for errors, …

```
try {
   // Loading...
   Shader simpleVert(Shader::Type::VERTEX, "shaders/simple.vert");
   Shader shadingFrag(Shader::Type::FRAGMENT, "shaders/shading.frag");
   Program standardShader(simpleVert, shadingFrag);

   Pipeline drawPipe;
   drawPipe.shader = &standardShader;

    // Using...
    context.setState(drawPipe);
    ...
} catch(...) {
   // Link and compile errors will result in an exception
}
```

GLSL Overview

- OpenGL Shading Language
  - Integrated in OpenGL since version 2.0 (2004)
  - „GLSlang“
  - GLSL has a similar syntax like C/C++

- The main function is

```
void main(void)
{
  // Shadercode
}
```

GLSL Data types

Type	Description
`float`	IEEE 32-Bit Floating Point
`double`	IEEE 64-Bit Floating Point
`int`	32 Bit Integer
`uint`	32 Bit Unsigned Integer
`bool`	Boolean (8 Bit)

- A direct initialization of variables is possible like in C++

```
float g = 9.81;
int i = 1;
```

Conversion

Target type	Can be converted from
`uint`	`int`
`float`	`int, uint`
`double`	`int, uint, float`

- In most cases, type casting must be done manually
  - GLSL is very picky here…

```
float f = 10.0;          // OK
int ten = f;             // Error
int ten2 = int(f);       // OK
```

- the type cast syntax differs from C, here it would be `(int)(f)`

Mathematical Functions

- sin(), cos(), tan(), asin(), acos(), atan(), …
- sqrt(), log(), exp(), pow(), …
- abs(), floor(), clamp(), round(), …
- min(), max(), step(), smoothstep(), …
- mix() : linear interpolation
- …

Vector Types

- GLSL has predefined types for vectors (and matrices)
- These types are identical to the known types from **glm**
- On old GPUs, all four vector operations run in parallel (**SIMD**, Single Instruction Multiple Data)
  - Modern GPUs run even more computations in parallel

Base type	2D	3D	4D
`float`	`vec2`	`vec3`	`vec4`
`double`	`dvec2`	`dvec3`	`dvec4`
`int`	`ivec2`	`ivec3`	`ivec4`
`uint`	`uvec2`	`uvec3`	`uvec4`
`bool`	`bvec2`	`bvec3`	`bvec4`

Examples for Vectors

```
vec2 velocity = vec2(1.0, 2.0);
vec3 red = vec3(1.0, 0.0, 0.0);
vec3 green = vec3(0.0, 1.0, 0.0);
vec3 yellow = red + green;
vec3 white = vec3(1.0);   // sets all three components to 1.0
vec3 v = yellow + velocity;   // Error, types to not fit
vec3 v2 = yellow + vec3(velocity, 0.0);    // OK
vec2 v3 = yellow.xy + velocity;          // OK
```

Vector operations

- Similar to glm we have some useful vector operations
  - `dot(u, v)`
  - `cross(u, v)`
  - `length()`
  - `distance(p, q)`
  - …
- Examples

```
float s = dot(u, v);              // dot product
vec3 crossPod = cross(u, v);  // cross product
vec3 compMult = red * green;  // multiply each component
```

Access Vector Components

- There are different possibilities to access the single components of a vector
- The following **appendix** is possible
  - (x,y,z,w) : Interpret as position/direction
  - (r,g,b,a) : Interpret as color
  - (s,t,p,q) : Interpret as texture coordinate (later)

```
vec3 color = vec3(0.5, 0.7, 0.9);
float redComponent = color.r;
float greenComponent = color.y; // also possible
float blueComponent = color[2]; // access by []
```

Swizzle and Masking

- When assigning vectors, we can use a **Swizzle-Operation** on the right side of the assignment to change the **order** of the components

```
vec3 u,v;
v = u.zxy;  // vx = uz, vy = ux, vz = uy
v = u.xxx;  // vx = ux, vy = ux, vz = ux
```

- On the left side of the assignment this appendix is a **masking** of the selected components

```
vec2 u;
vec3 v;
v = u;     // Error
v.xy = u; // OK, v.z remains unchanged
```

Structs

- Like in C, we can assemble multiple variables in a struct

```
struct Particle {
     float lifetime;
     vec3 position;
     vec3 velocity;
}

Particle p = Particle(10.0, pos, vel); // pos and vel are vec3
p.lifetime = 100.0;
```

Arrays

- Similar to C: Access by []-Operator
- Different from C: Negative indices and indices above the maximum size are not allowed
- integrated **`length()`** function

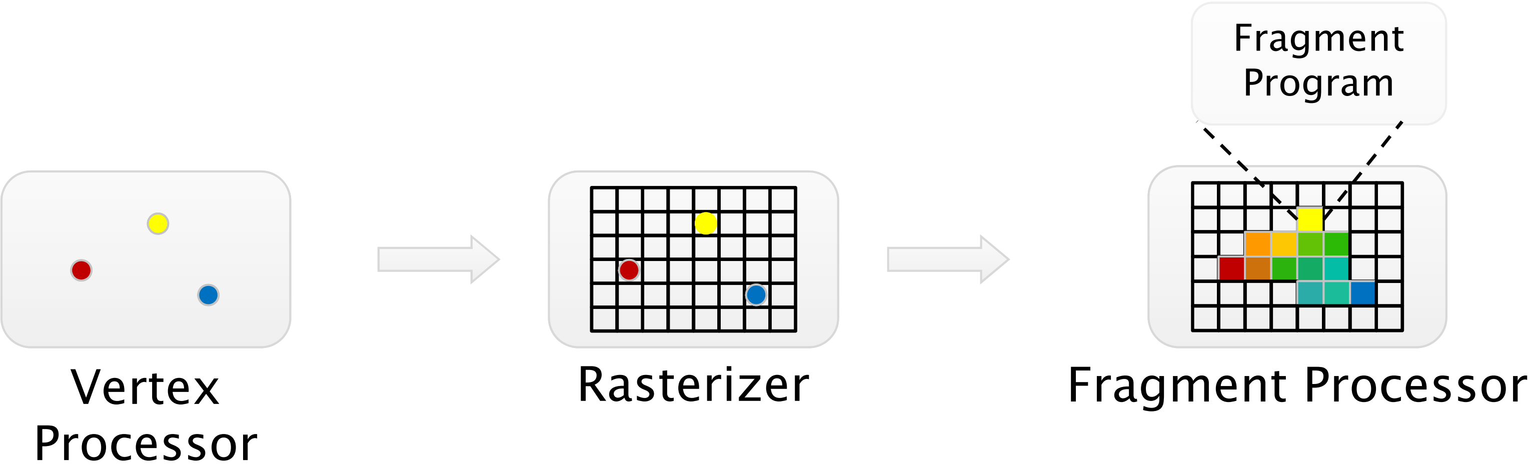
```
int myArray[5];

for (int i = 0 ;  i < myArray.length() ; i++)
    myArray[i] = i;
```

Functions

- Syntax like in C
- Main limitation:
  - no recursion
- Parameter modifiers:
  - **`in`** (read only)
  - **`inout`** (call-by-reference)
  - **`out`** (call-by-reference, write only)

Input/Output for Shader Programs

- A Shader can pass values to next shader stage by using so-called **Varying Variables**
- For example, the Vertex Shader can define an output which is transmitted to the Fragment Shader as input
  - The transmitted values (e.g. vertex colors) are **interpolated automatically** (bilinear interpolation)



Input/Output for Shader Programs

- Varying Variables are marked with **in** and **out**
  - Old Syntax: **varying**

- **Name** and **type** of the variable must be identical in both shaders

- or

  `layout(location = x)` can be used

Input/Output for Shader Programs

```
#version …
// Vertex Shader

layout(location = 0) in vec4 vPos;
out float myParam;

void main()
{
    gl_Position = vPos;
    myParam = …;
}
```

```
#version …
// Fragment Shader

in float myParam;
out vec4 pixelColor;

void main()
{
  pixelColor = vec4(myParam);
}
```

- In the example, the Vertex Shader defines an output variable
- This is set to a (different) value per vertex
- The Fragment Shader gets the interpolated value per pixel

VAOs with multiple Attributes

- A possible application for varying variables: VAOs with multiple attributes
  - e.g. Position and Color
- The Vertex Shader gets the VAO data as input

```
layout(location = …) in …;
```

- This data must be passed as „out“ to the Fragment Shader where it is collected as „in“

Shader Program for VAOs with Vertex Colors

```
#version …
// Vertex Shader

layout(location = 0) in vec2 vPos;
layout(location = 1) in vec3 vCol;
out vec3 myColor;

void main()
{
    gl_Position = vec4(vPos, 0, 1);
    myColor = vCol;
}
```

```
#version …
// Fragment Shader

in vec3 myColor;
out vec3 pixelColor;

void main()
{
    pixelColor = myColor;
}
```

- In the example, the Vertex Shader uses attribute 0 for the vertex position and attribute 1 for the vertex color
- The vertex color is passed as varying
- The Fragment Shader gets the interpolated color

Shader Program for VAOs with Vertex Colors

```
#version …
// Vertex Shader

layout(location = 0) in vec2 vPos;
layout(location = 1) in vec3 vCol;
layout(location = 0) out vec3 outColor;

void main()
{
    gl_Position = vec4(vPos, 0, 1);
    outColor = vCol;
}
```

```
#version …
// Fragment Shader

layout(location = 0) in vec3 inColor;
out vec3 pixelColor;

void main()
{
   pixelColor = inColor;
}
```

- Alternative matching of varying variables
    - Names can be different

Integrated Variables

- Vertex Shader
  - **`gl_Position`** : output position
  - **`gl_VertexID`** : vertex number (read)
  - …
- Fragment Shader
  - **`gl_FragCoord`** : xy position of current pixel (read)
    - **`gl_FragCoord.z`** is the pixel depth value (read)
  - **`gl_FragDepth`** : pixel depth value (write)
  - …

- In older Versions of GLSL, there are additional varyings, like **`gl_Normal, gl_FrontColor`**, …
  - in modern OpenGL we only have attributes

Preprocessor

- Preprocessor like in C

```
#define
#undef
#if
#ifdef
#ifndef
#else
#elif
#endif
```

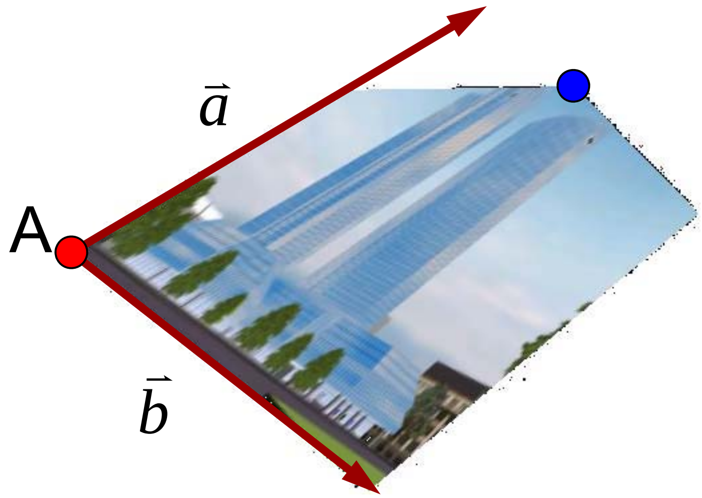
Quick Reference Guide

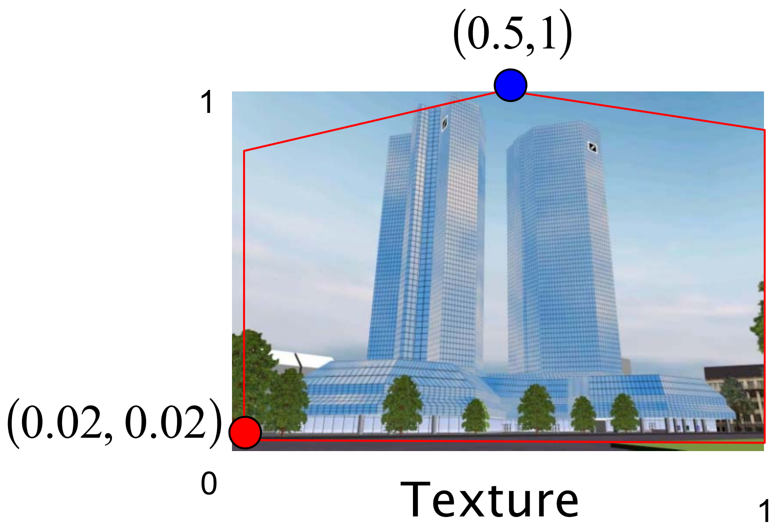
- https://www.khronos.org/files/opengl44-quick-reference-card.pdf
- Good overview about GLSL and OpenGL

Textures

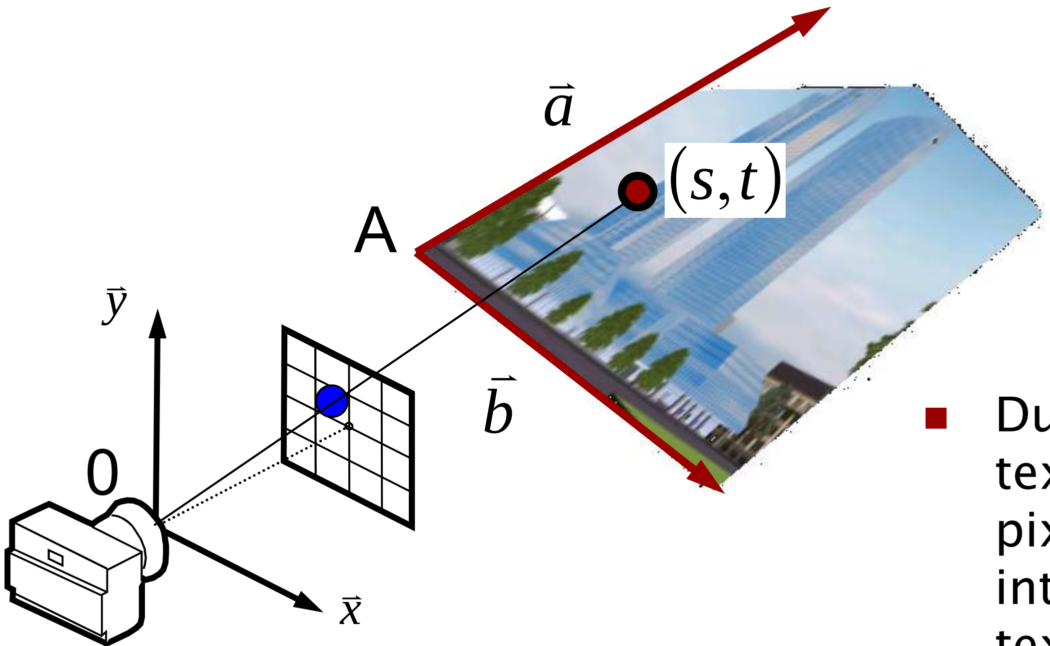
Texturing

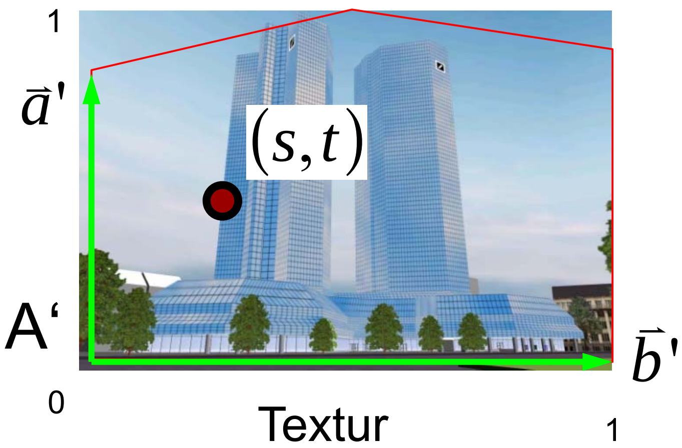
- During the rasterisation the texture is mapped onto the polygon.
- For each vertex, texture coordintaes (s,t) in the range [0..1] must be defined, that describe which part of the texture is mapped onto the polygon.
- Texture coordinates can be defined as vertex attribute in a VAO





Texture Coordinates





- Using texture coordinates, we define a coordinate system in the texture/image space
- During rasterization, we get (s,t) texture coordinates for each pixel which are bilinearly interpolated from the vertex texture coordinates
- For each rasterized pixel, we read the texture pixel (texel) using these interpolated texture coordinates and store them in the pixel
- *Texel*: Texture Element

Texture Checklist

- To use a texture in OpenGL, do the following:

- Create a texture object (**`glGenTexture`**, **`glBindTexture`**)
- Assign texture storage and the pixel data (**`glTexStorage`**, **`glTexSubImage`**)
- Associate a sampler for the texture (**`glGenSampler, glBindSampler, glSamplerParameter`**)
- Include texture coordinates as an attribute with the vertices
- Use a vertex shader to pass through the texture coordinates and retrieve the texel values through a sampler in the fragment shader (**`texture`**)

Create a Texture Object

- Generate a texture ID

```
GLint textureID;
glGenTextures(1, &textureID)
```

- This can then be bound as current texture

```
glBindTexture(GL_TEXTURE_2D, textureID);
```

- Afterwards we define texture data, filters etc.
  - They are stored in a texture object
  - Later we can restore all settings with `glBindTexture(…)`

Texture Storage

- The recommended way to define a texture is to first define the **texture storage** and afterwards the pixel data
- The texture storage defines only the **format** (width, height, rgb format) of the texture without the actual pixel data
- This format is then fixed (**immutable** storage), the pixel data can still be changed

Texture Storage

```
glTexStorage{1,2,3}D( target, levels, internalFormat,
                      width, [height, depth] );
```

target: GL_TEXTURE_2D (other values later)

levels: number of MipMap levels (later)

internal format: What is stored internally (GPU) per Texel (GL_RGBA32F, GL_R32F, GL_RGB16I, ...)

width: texture width in pixels

height: texture height in pixels (2D Texture)

depth: texture depth (3D Texture)

Define the Pixel Data

```
glTexSubImage{1,2,3}D( target, level, xoffset, [yoffset,
zoffset], width, [height, depth], format, type, data );
```

- **`target`**: GL_TEXTURE_2D (other values later)
- **`level`**: select the MipMap level (later)
- **`xoffset`**: start position of texture block in x direction
- **`yoffset`**: start position of texture block in y direction
- **`width`**: texture width in pixels
- **`height`**: texture height in pixels (2D Texture)
- **`depth`**: texture depth (3D Texture)
- **`format`**: which pixel format is used in CPU memory: GL_RGB, GL_BGR, GL_RGBA, ...
- **`type`**: data type per color channel in CPU memory: GL_UNSIGNED_BYTE, GL_FLOAT, …
- **`data`**: pixel data adress in CPU memory

Texture Usage

- Using `glTexSubImage(…)` the texture data is (usually) copied from CPU RAM into texture memory on the GPU (and usually remains there)
  - Use this command only once, afterwards use `glBindTexture()` to re-activate the texture
- Each vertex needs a texture coordinate:
  - We need an additional VBO with texture coordinates as additional vertex attribute
  - Texture coordinates are usually in the range [0,1]x[0,1]
    - In most cases we only need the first two components (called s and t), the 3rd (p) is required for 3D-Textures, the 4th (q) – like homogeneous coordinates – for projective texturing
- There is no integrated image loader in OpenGL
  - Additional library required, e.g. stb_image or DevIL

Texture Units

- Each graphics card has multiple so-called **texture units**
  - Each texture can be assigned to a texture unit using **`glActiveTexture`** to activate the texture unit and **`glBindTexture`** to assign the texture to this texture unit
  - By assigning textures to texture units, multiple textures can be used in a shader
  - Example for two textures on Texture Units 0 and 1:

```
// use Texture Unit 0 for Texture texID
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texID);

// use Texture Unit 1 for Texture otherTexID
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, otherTexID);
```

Sampler

- For each texture we need a so-called **Sampler** to define how we „read“ the pixel values from given texture coordinates
- This includes
  - Clamp- and Repeat modes for texture coordinates outside the range [0,1]
  - Filter Modes for minification and magnification (interpolating between multiple pixel values)

- OpenGL uses a default sampler, but it is a good practice to define a sampler for each texture unit

Create a Sampler Object

- Similar to a texture ID we can generate a Sampler ID

```
GLint samplerID;
glGenSampler(1, &samplerID)
```

- This can then be bound to a texture unit as current sampler

```
glBindSampler(textureUnit, samplerID);
```

Texture Coordinate Wrapping

- For the sampler, then use the general function:

```
glSamplerParameter{if}( GL_TEXTURE_{12}D, name,
                        value );
```

- What happens with texture coordinates outside the range [0,1]x[0,1] ?
  - controlled by **`name=GL_TEXTURE_WRAP_{ST}`**

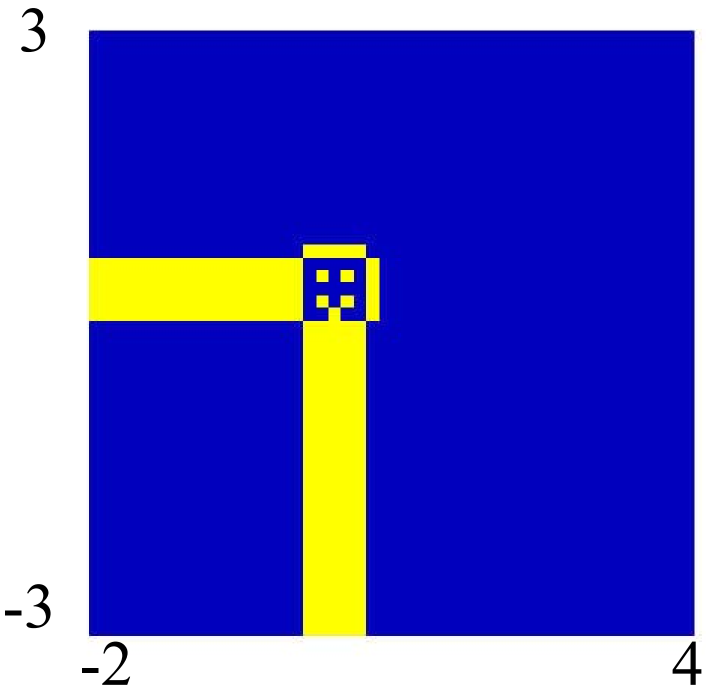
    `value=GL_CLAMP`: values <0 are set to 0, values > 1 are set to 1

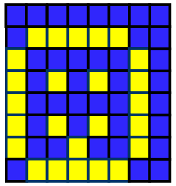
    `value=GL_CLAMP_TO_EDGE`: Similar to **`GL_CLAMP`**, but half a texel inside (linear texture filter stays inside)

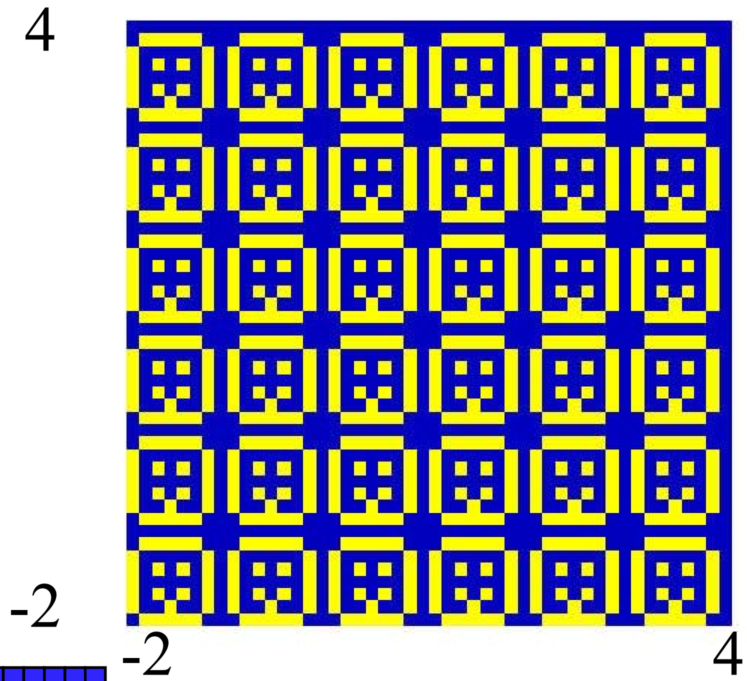
    `value=GL_CLAMP_TO_BORDER` : Similar to **`GL_CLAMP`**, but half a texel outside (linear texture filter stays outside)

    `value=GL_REPEAT`: use only the fractional part (tiling)

Coordinate Wrapping







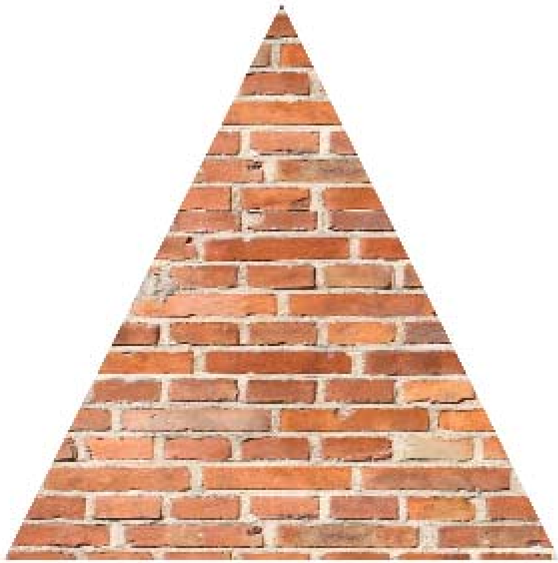
GL_CLAMP

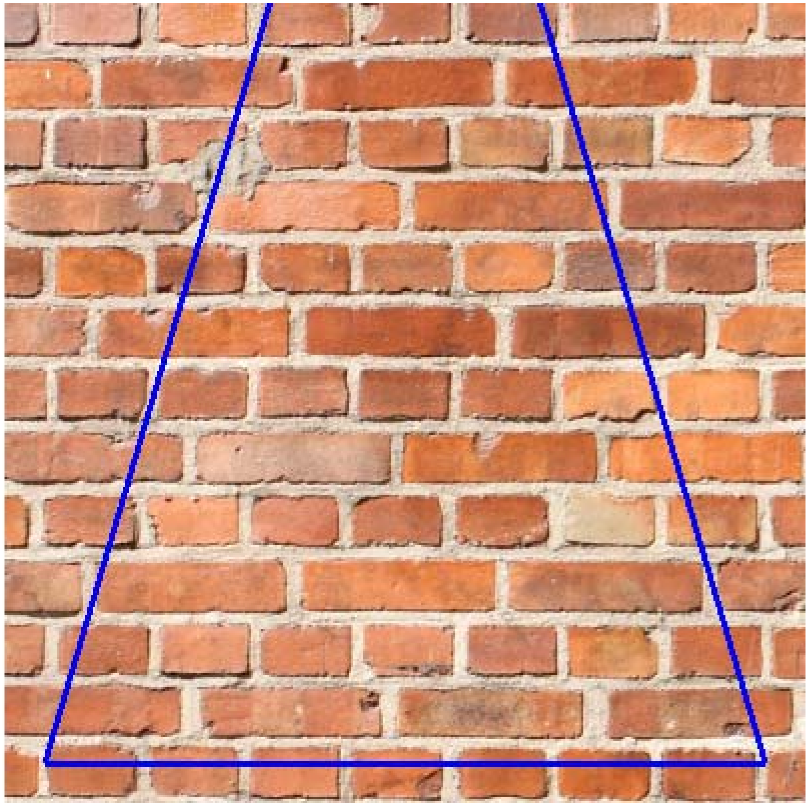
„Picture on the Wall“
Texture coordinates <= 0 are set to 0
Texture coordinates >= 1 are set to 1

GL_REPEAT

„Tiles on the Wall“
Use only the fractional part
of the texture coordinate

Wrapping Parameter Example

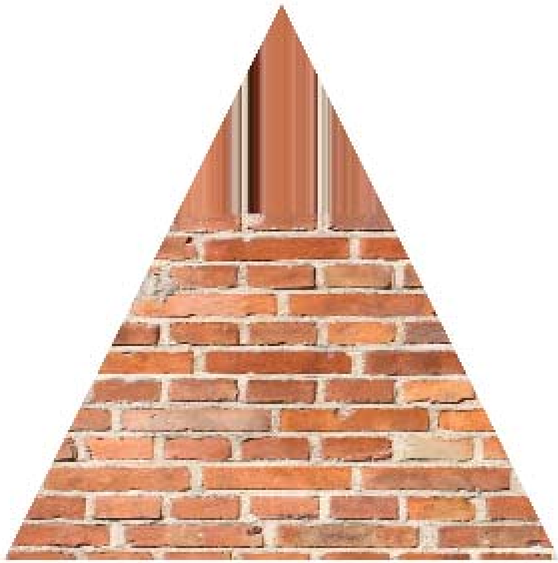


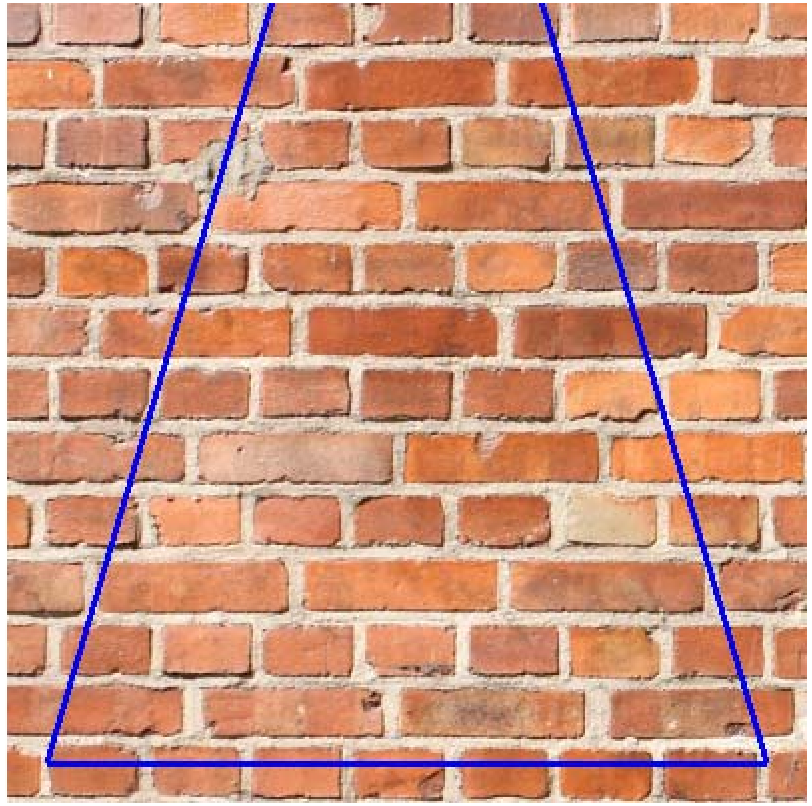


GL_REPEAT

Wrapping Parameter Example

Clamp To Edge:
Clamp to 0 + „half Texel“
or 1– „half Texel“



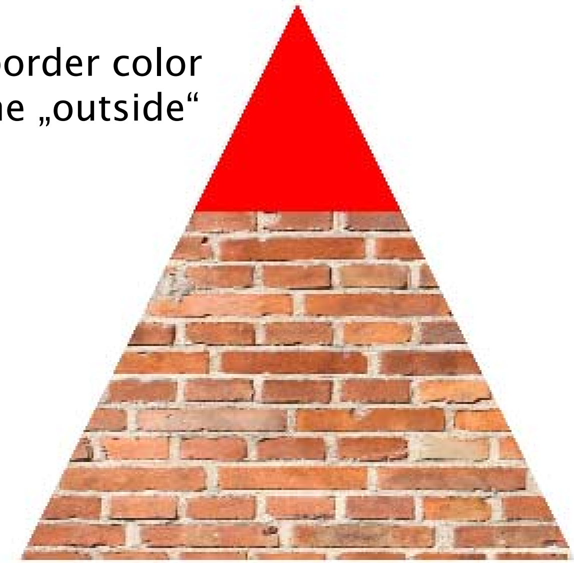


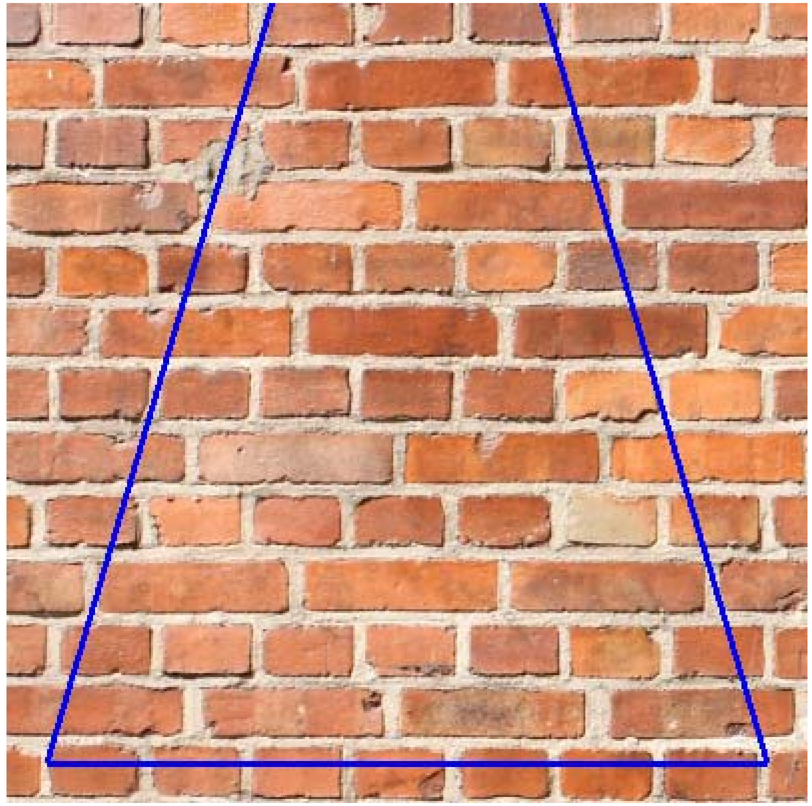
`GL_CLAMP_TO_EDGE`

Wrapping Parameter Example

Clamp To Border:
Clamp to 0 – „half Texel“
or 1+ „half Texel“

Use border color
for the „outside“

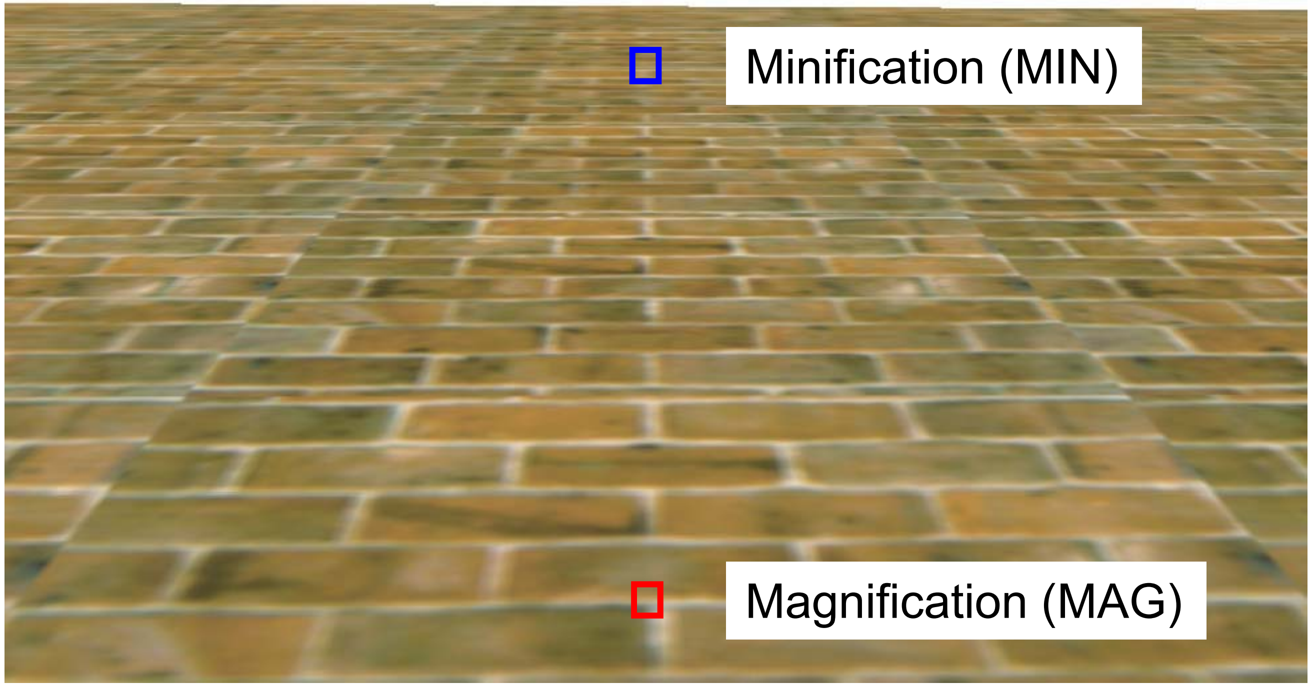




GL_CLAMP_TO_BORDER

Texture Filter

- Texture Magnification
  - One texel is mapped to multiple pixels
- Texture Minification
  - Multiple texels are mapped to one pixel



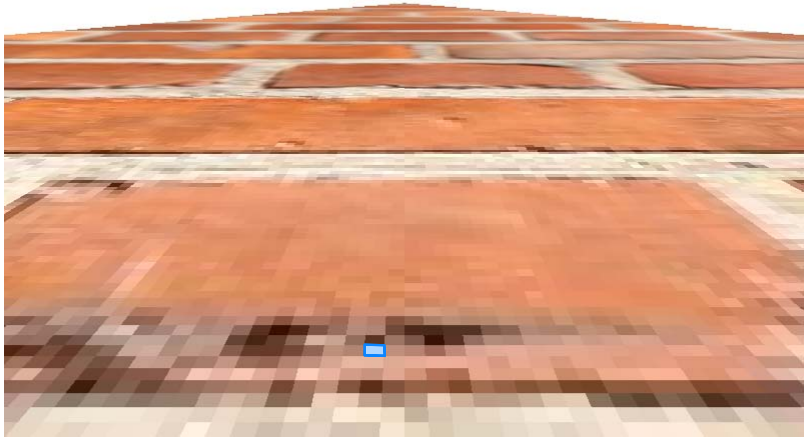
Filter Specification

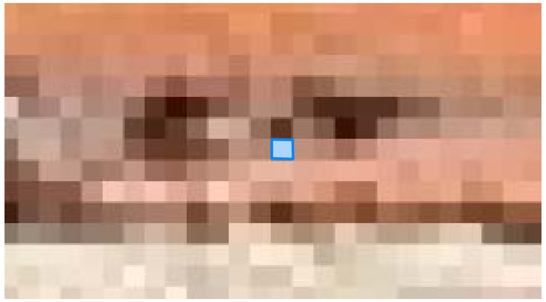
- using **name=GL_TEXTURE_{MIN,MAG}_FILTER** as parameter of **glSamplerParameter{if}[v].**

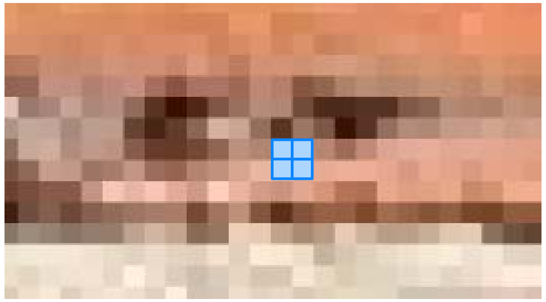
- for ***_MAG_*** we have
  - **GL_NEAREST**: point filter
  - **GL_LINEAR**: linear filter

- for ***_MIN_*** additionally
  - **GL_{NEAREST,LINEAR}_MIPMAP_{NEAREST,LINEAR}**
  - Linear interpolation within a level and between two levels (tri-linear interpolation)

Texture Magnification







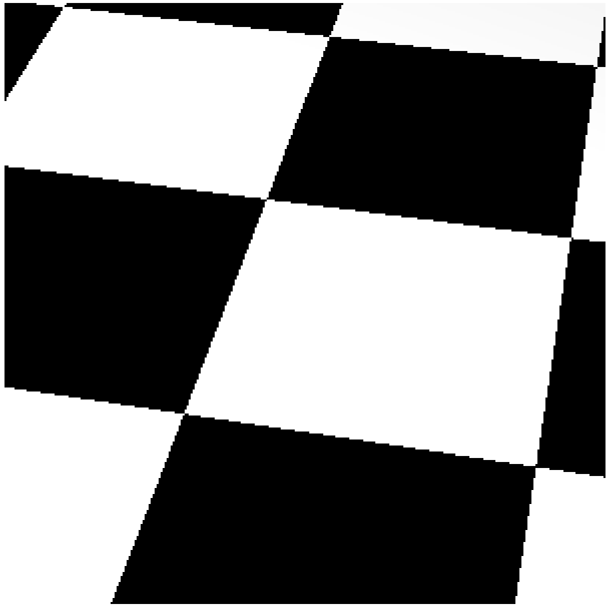
```
glSamplerParameter(GL_TEXTURE_2D,
                   GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
```

Use only a single texel

```
glSamplerParameter(GL_TEXTURE_2D,
                   GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR);
```

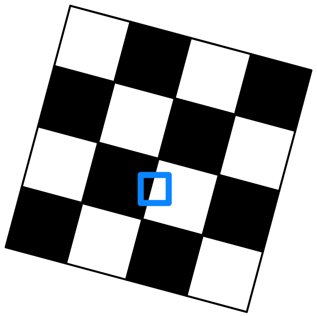
Bilinear interpolation from 4 neighbor texels

Texture Magnification



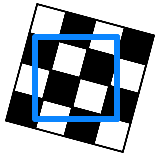
Nearest

Linear



Texture Minification

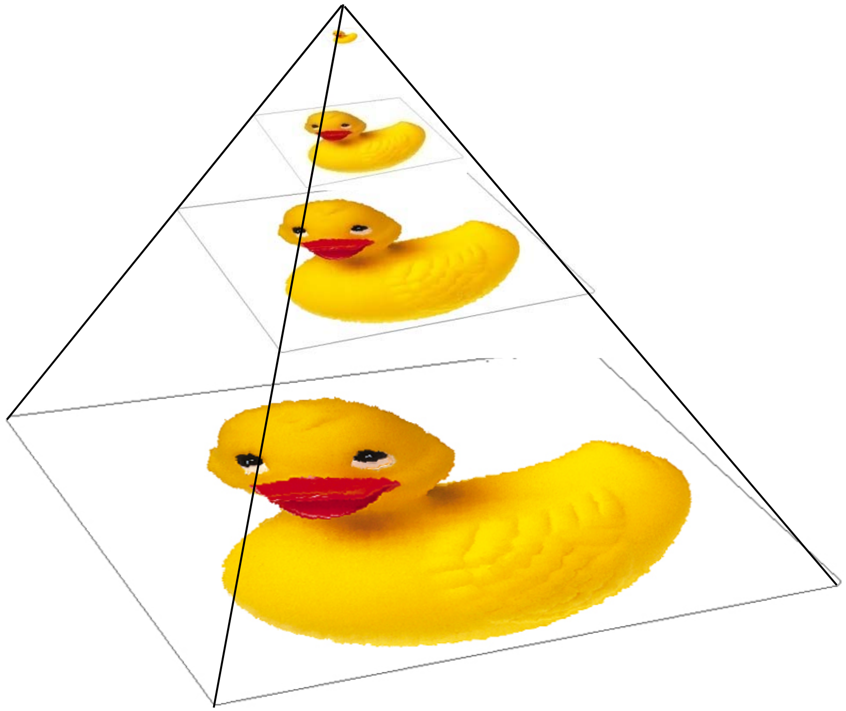
- In the distant region (MIN), many texture pixels are inside of one screen pixel
  - In case of a strong minification we have to compute the average all these texels since they are all projected onto the same screen pixel.
  - This is too time-consuming for a real-time application.



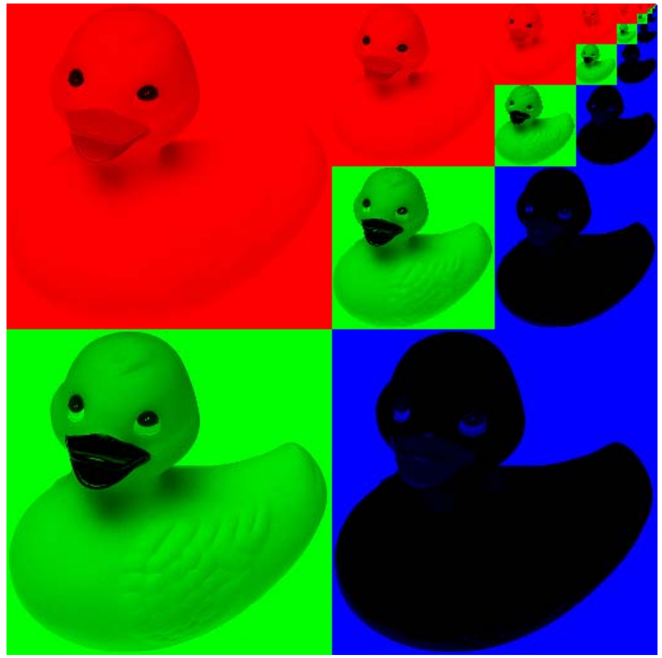
- Solution: Before starting, we create smaller versions of the texture by repeated averaging of 2x2 texels. If many texels are mapped to the same screen pixel, we select the best-fitting minified texture.

- The resulting textures are called **MipMaps** (lat. „Multum in Parvo“)

MipMaps



We have a texture in different resolution levels



Internally, the 2^n-image can be extended to a 2^{n+1}-image.

Automatic MipMap Generation

- the **`level`** Parameter of **`glTexSubImage{12}D`** decides which MipMap level should be set
  - 0 is the largest map, each following map has half of the size until 1x1

- Helper function:

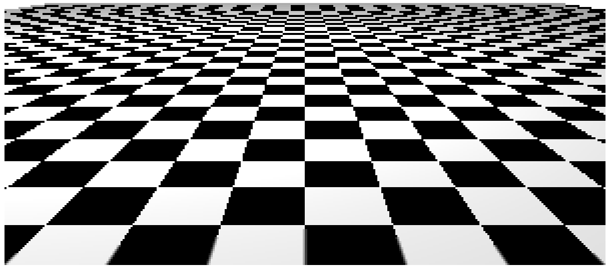
  `glGenerateMipmap(GL_TEXTURE_2D)`

  generates all MipMap-Levels for the currently bound texture

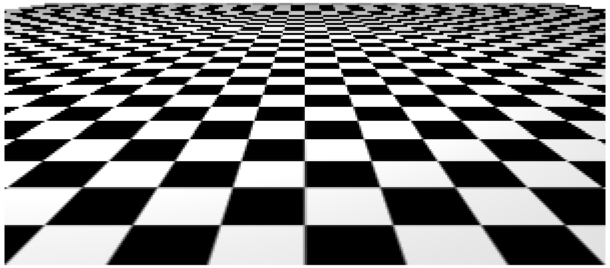
Selection of the MipMap Level

- Depending on the distance from viewer to object, the MipMap level is selected
- Question:
  - How many texture pixels are inside a current screen pixel ?
- Reformulated question:
  - How do the texture coordinates change, if I move one screen pixel ?
    - derivative dFdx, dFdy → see Red Book
  - Often the largest polygon edge in screen space is used
    - MipMap-Filter is sometimes too large (in one direction)
    - Optimization: Anisotropic Filter (not in OpenGL standard)

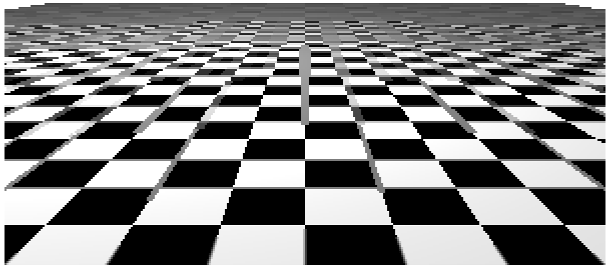
Different Settings for Texture Minification



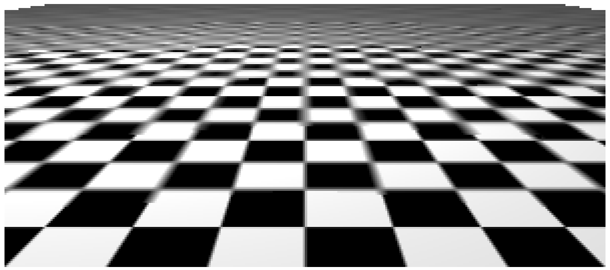
GL_NEAREST



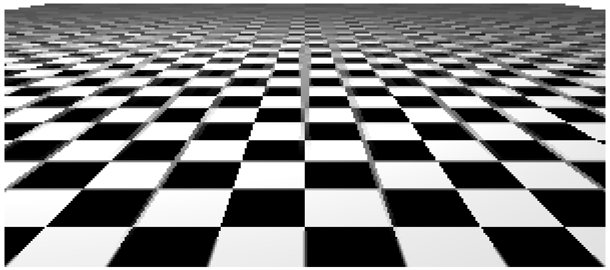
GL_LINEAR



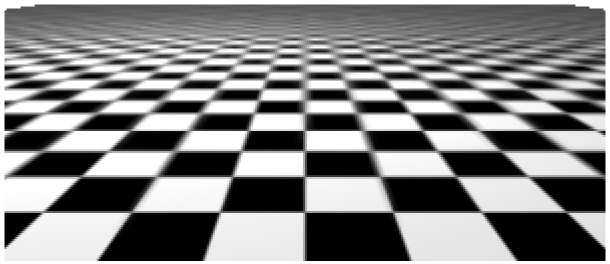
GL_NEAREST_MIPMAP_NEAREST



GL_LINEAR_MIPMAP_NEAREST



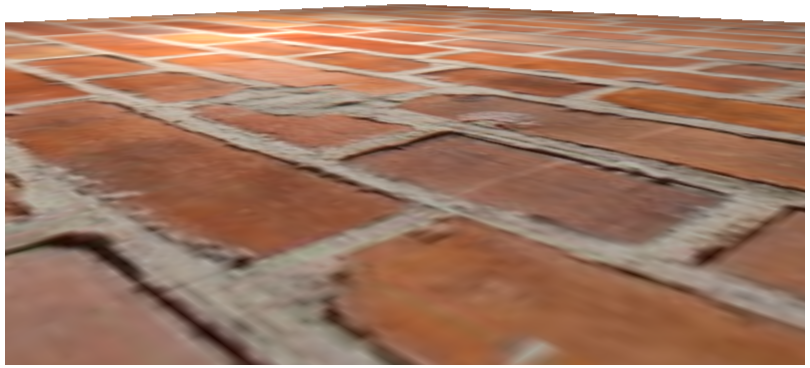
GL_NEAREST_MIPMAP_LINEAR

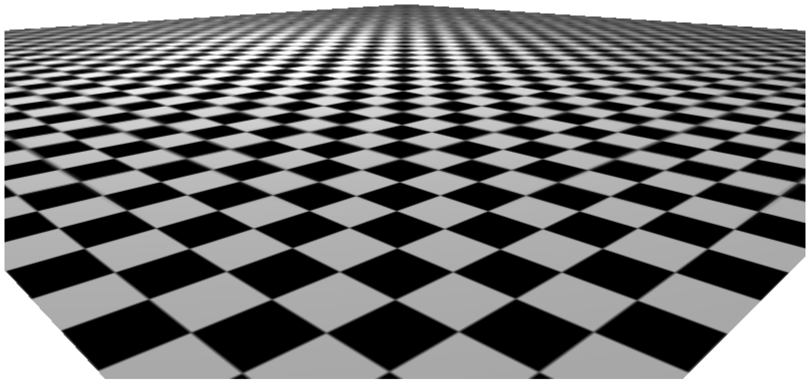


GL_LINEAR_MIPMAP_LINEAR

Texture Minification: Mipmaps

- The MipMap Filter is especially useful if the camera/object is moving → see exercise





Components for Texturing

- Create texture
  - **`glGenTextures(), glBindTexture(), glTexStorage2D() glTexSubImage2D(), …`**
- Set sampler parameters
  - **`glGenSampler(), glBindSampler(), glSamplerParameter(…)`**
- Create VAO with texture coordinates as vertex attribute
- **Vertex Shader** passes texture coordinates (s,t) through (varying)
  - Linear interpolation of (s,t) for the Fragment Shader
- **Fragment Shader** reads the texture pixel using the interpolated (s,t)-coordinates and writes it as pixel color
  - For accessing textures we can use a **Sampler** (uniform parameter)

Example: VAOs with Texture Coordinates

```
GLuint vao, vboPos, vboTexCoord;
GLfloat vertices[6][4] = { {2.0f, 3.0f, 4.0f, 1.0f}, … };
GLfloat texcoords[6][2] = { {0.0f, 1.0f}, … };
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glGenBuffers(1, &vboPos);
glBindBuffer(GL_ARRAY_BUFFER, vboPos);
glBufferStorage(GL_ARRAY_BUFFER, sizeof(vertices), vertices, 0);
glVertexAttribFormat(0, 4, GL_FLOAT, GL_FALSE, 0);
glVertexAttribBinding(0, 0);
glEnableVertexAttribArray(0);

glGenBuffers(1, &vboTexCoord);
glBindBuffer(GL_ARRAY_BUFFER, vboTexCoord);
glBufferStorage(GL_ARRAY_BUFFER, sizeof(texcoords), texcoords, 0);
glVertexAttribFormat(3, 2, GL_FLOAT, GL_FALSE, 0);
glVertexAttribBinding(3, 1);
glEnableVertexAttribArray(3);
```

Here: Attribute 3 are texture coordinates which are connected to binding index 1

We use the default texture unit 0

Example: Draw VBO with TexCoords

```
void display()
{
  glClear(…);
  glBindVertexArray(vao);
  glBindVertexBuffer(0, vboPos, 0, 16);
  glBindVertexBuffer(1, vboTexCoord, 0, 8);
  glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Shader for Textures

```
#version …
// Vertex Shader

layout(location = 0) in vec4 vPos;
layout(location = 3) in vec2 vTexCoord;
out vec2 myTexCoord;

void main()
{
    gl_Position = vPos;
    myTexCoord = vTexCoord;
}
```

```
#version …
// Fragment Shader

layout (binding = 0) uniform sampler2D image;
in vec2 myTexCoord;
out vec4 pixelColor;

void main()
{
  pixelColor = texture(image, myTexCoord);
}
```

- In the example, the Vertex Shader gets the texture coordinate as attribute 3
- We write the texture coordinate as varying „out“
- The Fragment Shader gets the interpolated texture coordinate as varying input „in“
- Using these texture coordinates, we take a `sampler2D` to read from the image with `texture()`
- Here texture unit 0 is selected using layout (binding = 0)
- Finally, the selected texel is written as pixel color

Sampler Types

- For different textures there are different sampler types
  - 1D/2D/3D textures
    - **`sampler1D, sampler2D, sampler3D`**
  - Cube Maps
    - **`samplerCube`**
  - Shadow Maps
    - **`sampler2DShadow`**
  - Integer / unsigned integer pixel colors
    - **`isampler… usampler…`**
  - …
  - complete List: see Red Book

Example: Texture Definition

```
glGenTextures(1, &textureID);
glBindTexture( GL_TEXTURE_2D, textureID );

glTexStorage2D(GL_TEXTURE_2D, 8, GL_RGBA32F, width, height);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_RGBA, GL_FLOAT, data );
glGenerateMipmap(GL_TEXTURE_2D);  // optional

glGenSampler(1, &samplerID);
glBindSampler(0, samplerID );  // assign sampler to (default) texture unit 0

glSamplerParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glSamplerParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glSamplerParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glSamplerParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
```

Attention: The default value for `GL_TEXTURE_MIN_FILTER` is `GL_NEAREST_MIPMAP_LINEAR` !
If no MipMaps are generated (e.g. using `glTexImage`), this can result in **black pixels !**
The Min Filter should therefore be set to `GL_NEAREST` or `GL_LINEAR` **!**

That´s all for today

- Next week: Advanced OpenGL