(5) Advanced OpenGL

GPU Programming

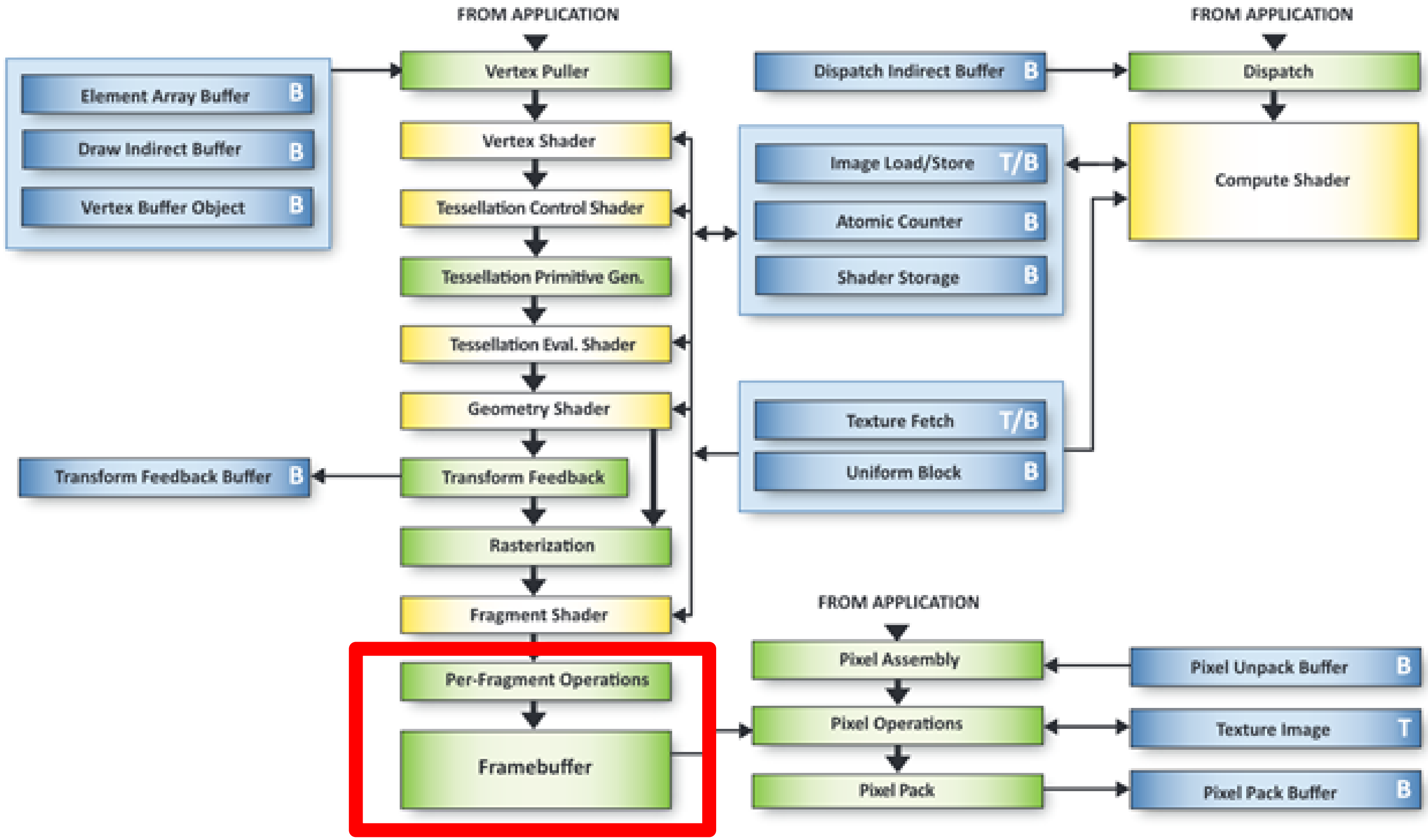
Thorsten Grosch

TU Clausthal

Overview

- OpenGL Pixel Pipeline
  - Blending
  - Logic Operations
- Display Buffers
  - Stencil Buffer
- Fast Rendering
  - Geometry Buffer

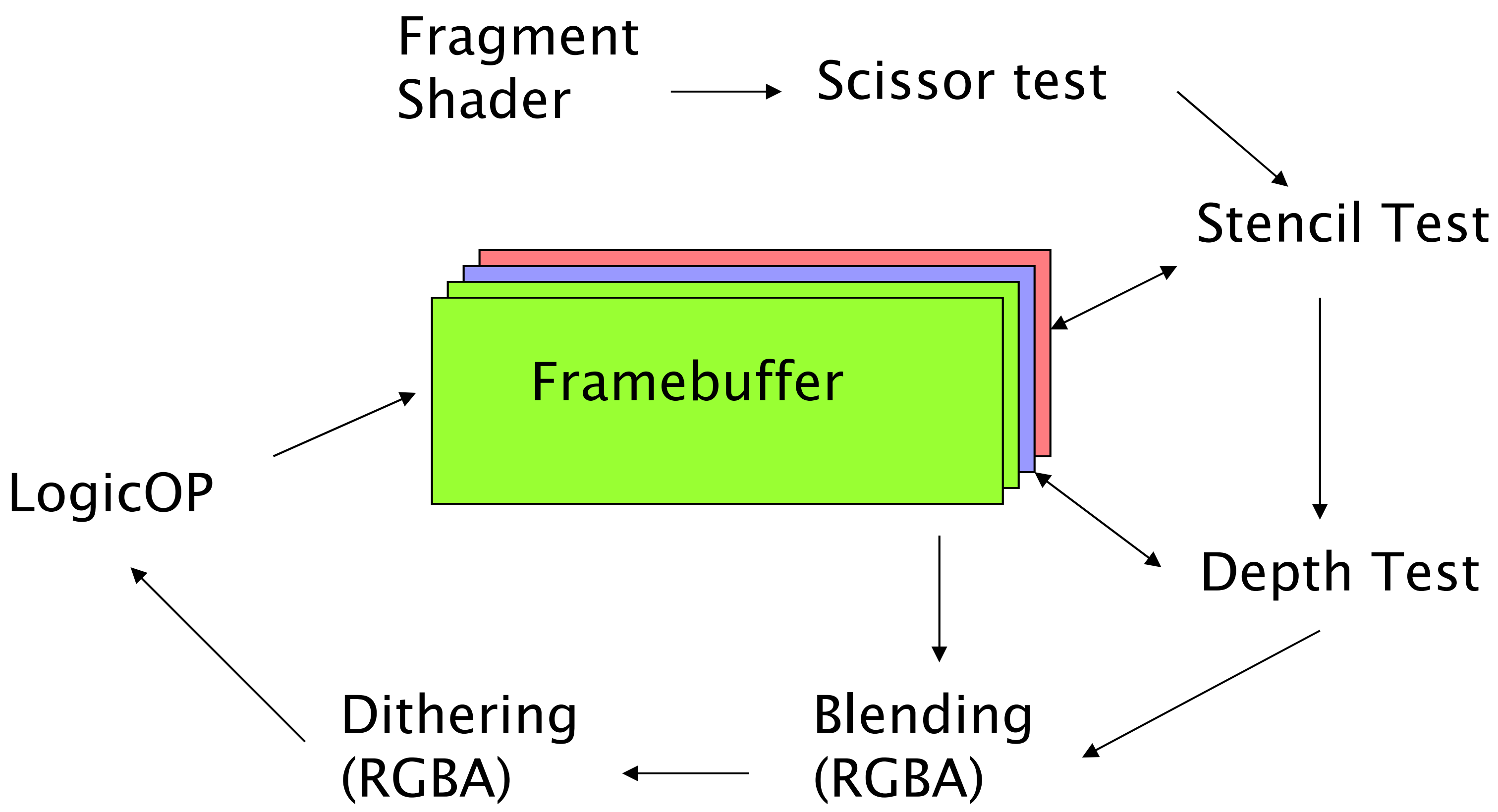
OpenGL 4.4



FROM APPLICATION
Element Array Buffer B
Draw Indirect Buffer B
Vertex Buffer Object B
Vertex Puller
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
Compute Shader
Image Load/Store T/B
Atomic Counter B
Shader Storage B
Texture Fetch T/B
Uniform Block B
FROM APPLICATION
Pixel Assembly
Pixel Operations
Pixel Pack
Pixel Unpack Buffer B
Texture Image T
Pixel Pack Buffer B

Pixel Pipeline

Several tests and operations are performed before the final pixel color is written into the color buffer…



Hint: Modern GPUs have an **early z** option: Scissor, Stencil and Depth test can be performed before the Fragment Program. This can not be controlled manually.
Some operations do not work in combination, e.g. Blending and LogicOp

Blending

Alpha Values and Blending

- Colors are 4D values
  - R, G, B, α
- Up to now we ignored α
- This value has an effect only if Blending is enabled

```
glEnable(GL_BLEND);
```

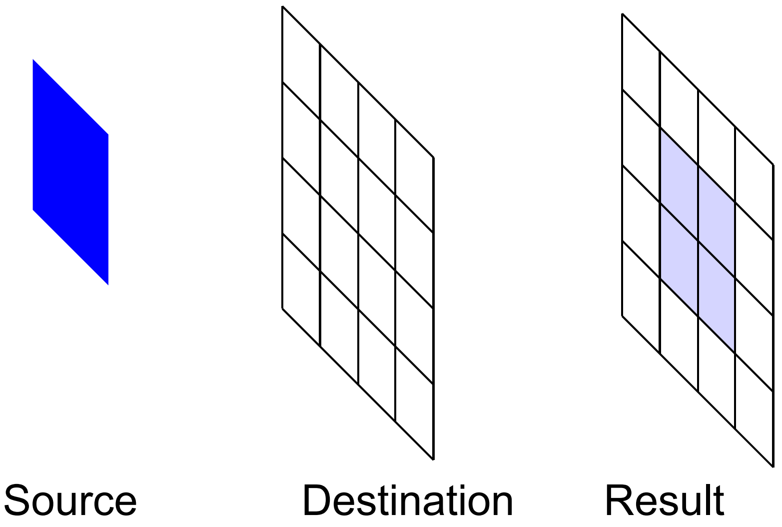
- In OpenGL the α value describes **opacity** by default (opposite of transparency)
  - $\alpha=0$: transparent
  - $\alpha=1$: opaque

Alpha Values and Blending

- Usually the color of a new pixel simply overwrites the existing color in the color buffer
- When Blending is enabled we get a combination of
  - The color of the new pixel (*Source*)
  - The color of the old pixel stored in the color buffer (*Destination*)
- Select the Blending function

```
glBlendFunc( Glenum
sfactor, Glenum dfactor );
```

Typical combination:
Transparent Objects



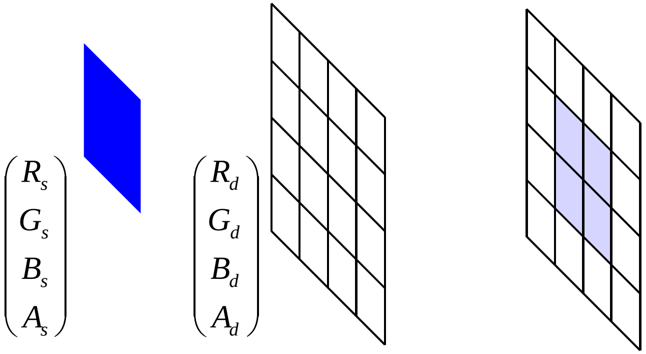
$$\alpha_s \cdot \begin{pmatrix} R_s \\ G_s \\ B_s \\ \alpha_s \end{pmatrix} + (1-\alpha_s) \cdot \begin{pmatrix} R_d \\ G_d \\ B_d \\ \alpha_d \end{pmatrix} =$$

Blending

General Blending

(sfactor and dfactor are 4D vectors)

$$\begin{pmatrix} R_S \cdot S_r \\ G_S \cdot S_g \\ B_S \cdot S_b \\ A_S \cdot S_a \end{pmatrix} \oplus \begin{pmatrix} R_d \cdot D_r \\ G_d \cdot D_g \\ B_d \cdot D_b \\ A_d \cdot D_a \end{pmatrix}$$



Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	destination	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	source	$(1, 1, 1, 1)-(R_d, G_d, B_d, A_d)$
GL_ONE_MINUS_SRC_COLOR	destination	$(1, 1, 1, 1)-(R_s, G_s, B_s, A_s)$
GL_SRC_ALPHA	source or destination	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	source or destination	$(1, 1, 1, 1)-(A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	source or destination	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	source or destination	$(1, 1, 1, 1)-(A_d, A_d, A_d, A_d)$
GL_SRC_ALPHA_SATURATE	source	$(f, f, f, 1)$; $f = \min(A_s, 1-A_d)$
GL_CONSTANT_COLOR	source or destination	(R_c, G_c, B_c, A_c)
GL_ONE_MINUS_CONSTANT_COLOR	source or destination	$(1, 1, 1, 1)-(R_c, G_c, B_c, A_c)$
GL_CONSTANT_ALPHA	source or destination	(A_c, A_c, A_c, A_c)
GL_ONE_MINUS_CONSTANT_ALPHA	source or destination	$(1, 1, 1, 1)-(A_c, A_c, A_c, A_c)$

Table 6-1 Source and Destination Blending Factors

Example: Transparency

- The difference between:
  - `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`

$$\alpha_s \cdot \begin{pmatrix} R_s \\ G_s \\ B_s \\ \alpha_s \end{pmatrix} + (1-\alpha_s) \cdot \begin{pmatrix} R_d \\ G_d \\ B_d \\ \alpha_d \end{pmatrix}$$

if $\alpha_s = 1$, we set the new pixel color. α is interpreted as opacity.
(this is the „standard“ blending)

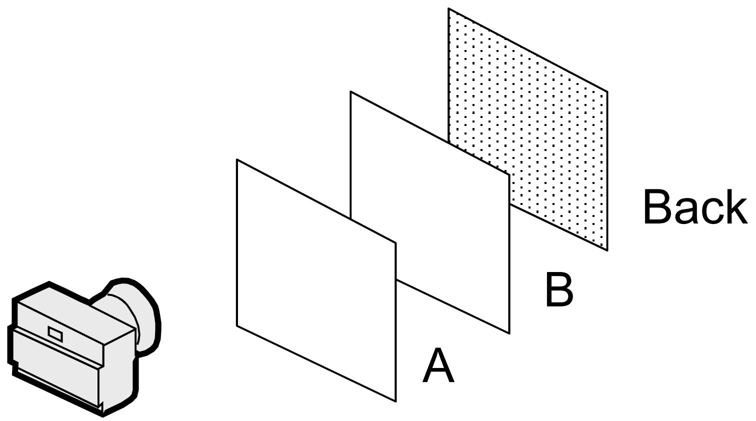
  - `glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);`

$$(1-\alpha_s) \cdot \begin{pmatrix} R_s \\ G_s \\ B_s \\ \alpha_s \end{pmatrix} + \alpha_s \cdot \begin{pmatrix} R_d \\ G_d \\ B_d \\ \alpha_d \end{pmatrix}$$

if $\alpha_s = 1$, we use the old pixel color. α is interpreted as transparency.

Problem: Transparencies

- A problem occurs if we render multiple transparent objects behind each other
- This works only if we render the transparent objects from **back to front**

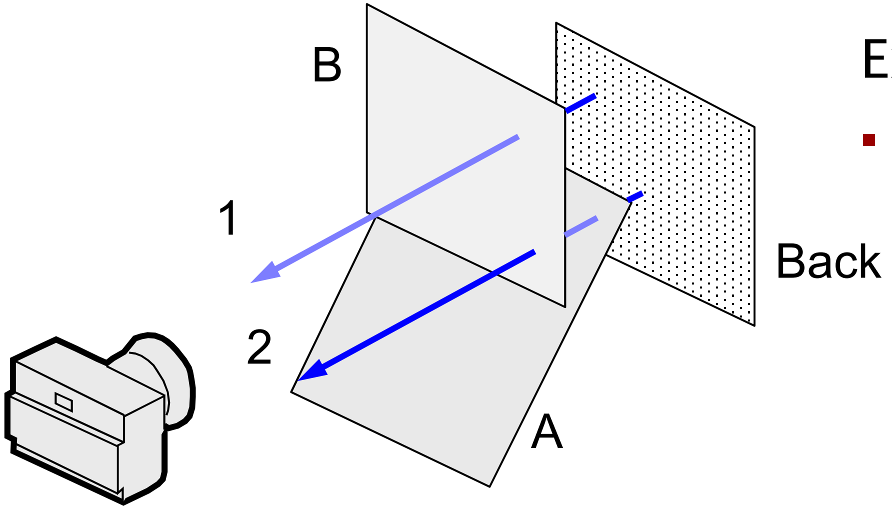


- Suppose each face has 20% transparency
- Blending only works if we render the background first, then B and then A
  - Result: 4% of the background color
- Drawing A first will also fill the z Buffer → B (and background) are never drawn
  - Blending is computed after the depth test

Solution:

- In case of multiple transparent objects behind each other:
  - Draw all non-transparent objects first (Depth-Buffer on)
  - Sort all transparent objects along the viewing direction
  - Set Depth-Buffer to Read-Only
    - Perfom a depth test, but do not overwrite the z values in the depth buffer
  - Render all transparent objects from back to front

Why do we set the Depth Buffer to „Read-Only“ ?



- Ray 1 is decreased by B
- Ray 2 is decreased by A, then B

- Sorting the polygons by their center point can lead to errors (e.g. A is sometimes in front of B, no perfect solution)

Example: B is drawn first

- With an activated z buffer, the upper part of A would never be drawn
  - Depth test is performed before blending
  - Only one transmission along ray 2
- Without filling the z buffer
  - B is drawn first (Blending with background)
  - Then A is drawn (blending with B)
  - Wrong drawing order, but results are often visually OK (sometimes even correct)

Blending Functions

```
GL_ZERO              GL_ONE
GL_DST_COLOR         GL_ONE_MINUS_DST_COLOR
GL_SRC_COLOR         GL_ONE_MINUS_SRC_COLOR
GL_DST_ALPHA         GL_ONE_MINUS_DST_ALPHA
GL_SRC_ALPHA         GL_ONE_MINUS_SRC_ALPHA
```

- Not all combinations make sense
- The source α value is often used
- Often multiple solutions for a problem
  - E.g. front-to-back blending uses the destination alpha

Example 1: Blending Functions

- Mixing of two images, each with 50%

```
glBlendFunc(GL_ONE, GL_ZERO);  // or: Blending off
// draw first image
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
// draw second image (with a=0.5)
```

- How do we get 75% of the first image and 25% of the second image ?

```
// draw second image (with a=0.25)
```

- Alternative Blending using a constant alpha

```
glBlendFunc(GL_CONSTANT_ALPHA, GL_ONE_MINUS_CONSTANT_ALPHA);
```

Example 2: Blending Functions

- Mixing of three images, each with 1/3
- Is this a correct solution ?

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
// draw all images with alpha=0.33
```

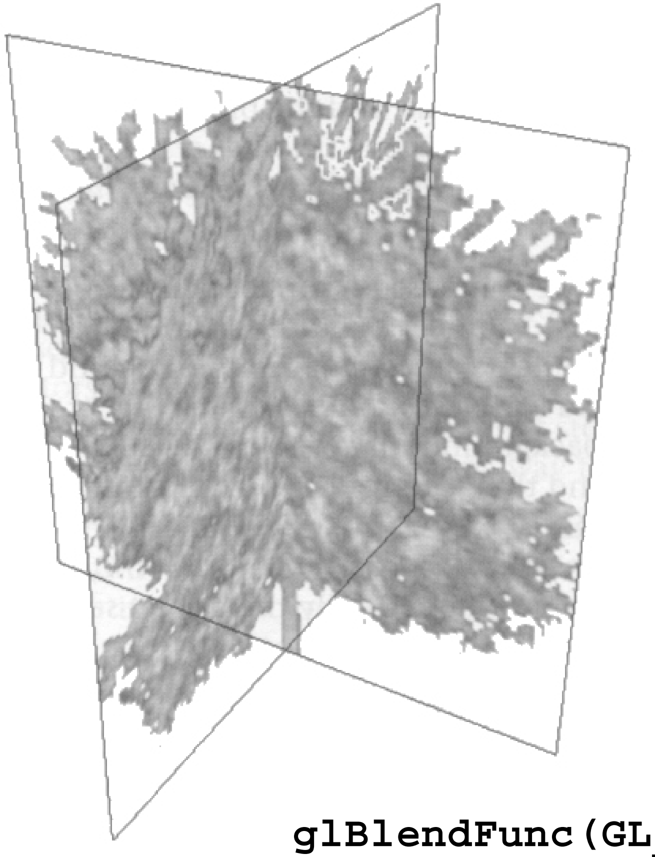
- Solution: correct if we start with a black screen
- Alternative:

```
// draw image one
glEnable(GL_BLEND)
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
// draw image two with alpha=0.5
// draw image three with alpha=0.33
```

Example 3: Blending Functions

- Air brush for a paint program
  - `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`
  - Define a circle for the brush and increase the α value in the center

Example 4: Transparent Textures



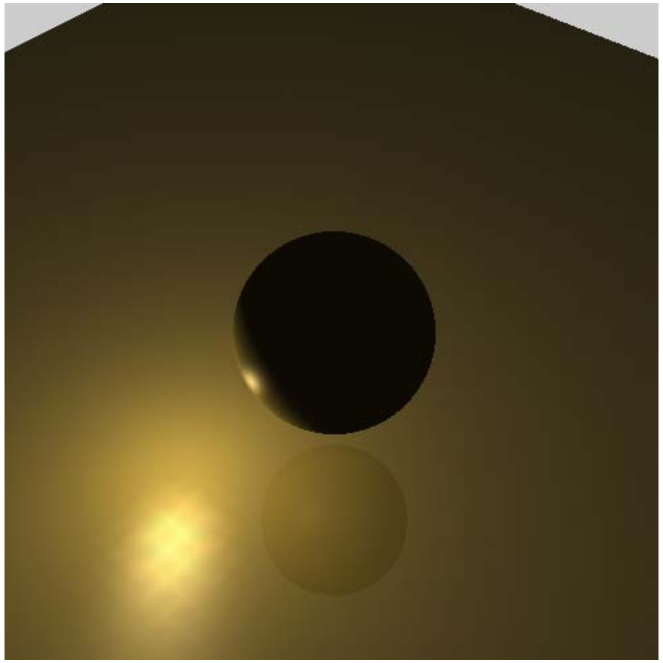
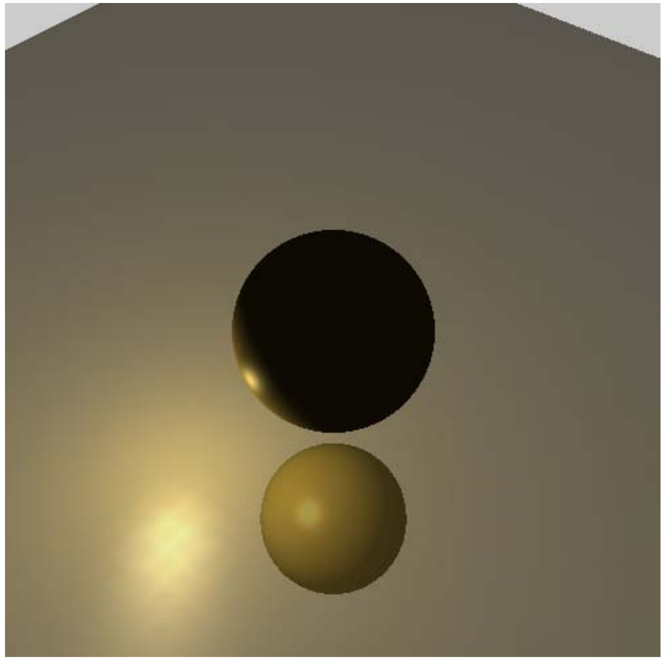
```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Tree: α=1, otherwise α=0

```
glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
```

Tree: α=0, otherwise α=1

Example 5: Fake Mirror

A second sphere is drawn below the polygon surface.
The ground is drawn with transparency, which looks like a mirror.

Blending Extensions

Change the Blending Function **`glBlendEquation();`**

- **`GL_FUNC_ADD, GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT, GL_MIN, GL_MAX`**

Treat Color and Alpha separately

- **`glBlendEquationSeparate (…);`**
- **`glBlendFuncSeparate(…);`**

Logic Operations

TU Clausthal

Color Buffer: Logical Operations

- Logical operations can be computed in the color buffer when pixels are drawn
- Defined by

```
glLogicOp( Glenum opcode );
```

- Activated by `glEnable( GL_COLOR_LOGIC_OP );`
- Now each new pixel is combined (bit by bit) with the existing pixel in the color buffer before it is drawn into the color buffer

Color Buffer: Logical Operations

- Operations (n: new value, b: existing color):

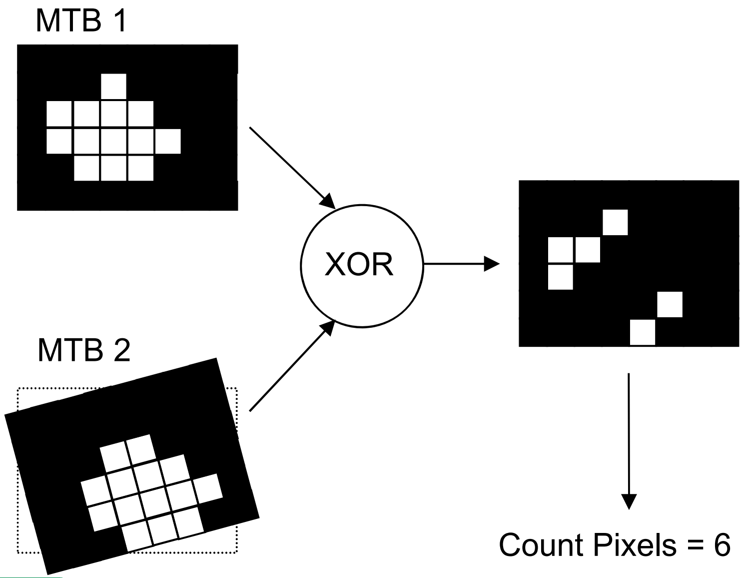
GL_CLEAR	0	GL_SET	1
GL_COPY	n	GL_COPY_INVERTED	~n
GL_NOOP	b	GL_INVERT	~b
GL_AND	n & b	GL_NAND	~(n & b)
GL_OR	n | b	GL_NOR	~(n | b)
GL_XOR	n ^ b	GL_EQUIV	~(n ^ b)
GL_AND_REVERSE	n & ~b	GL_AND_INVERTED	~n & b
GL_OR_REVERSE	n | ~b	GL_OR_INVERTED	~n | b

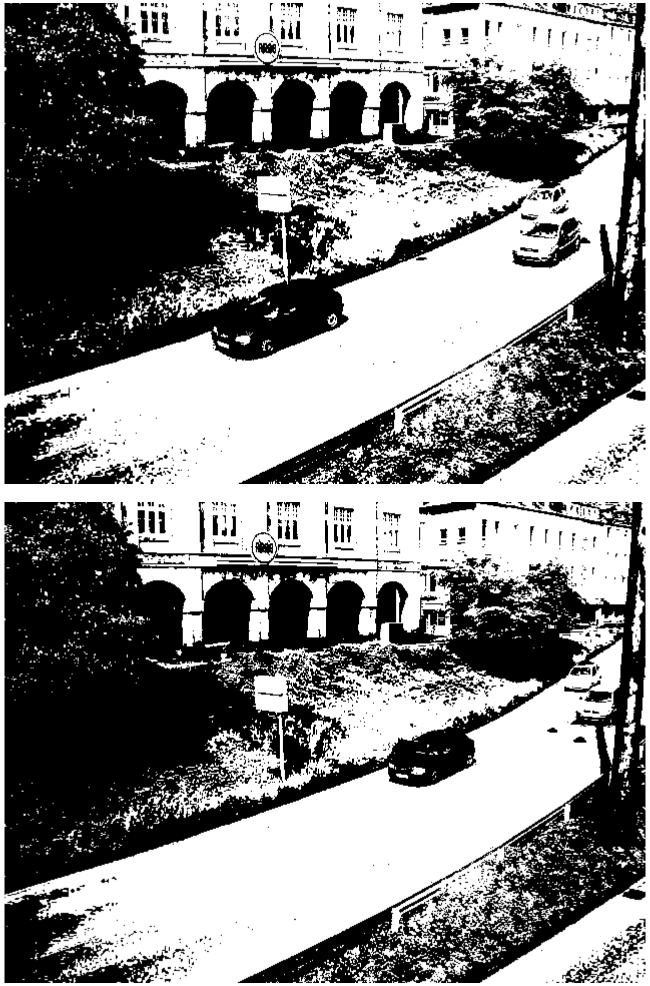
Application Example: HDR Image Alignment

- Several photographs with different shutter time
  - Combine to High Dynamic Range (HDR) image
- No tripod used for the camera, therefore slight translation and rotation
- Combination
  - blurry
  - ghosts

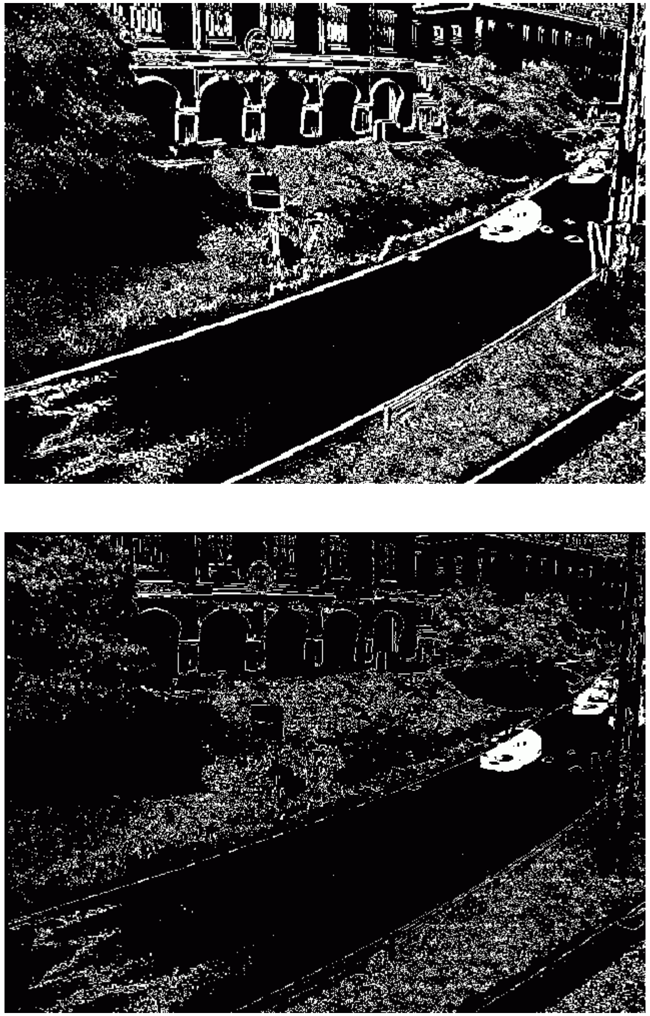
HDR Alignment with MTBs

- Alignment of the original images is difficult due to the different brightness
- Conversion to bitmaps (so-called Median Threshold Bitmaps [Ward 2003])
- Search for translation / rotation, such that the XOR combination creates an image with the lowest number of white pixels

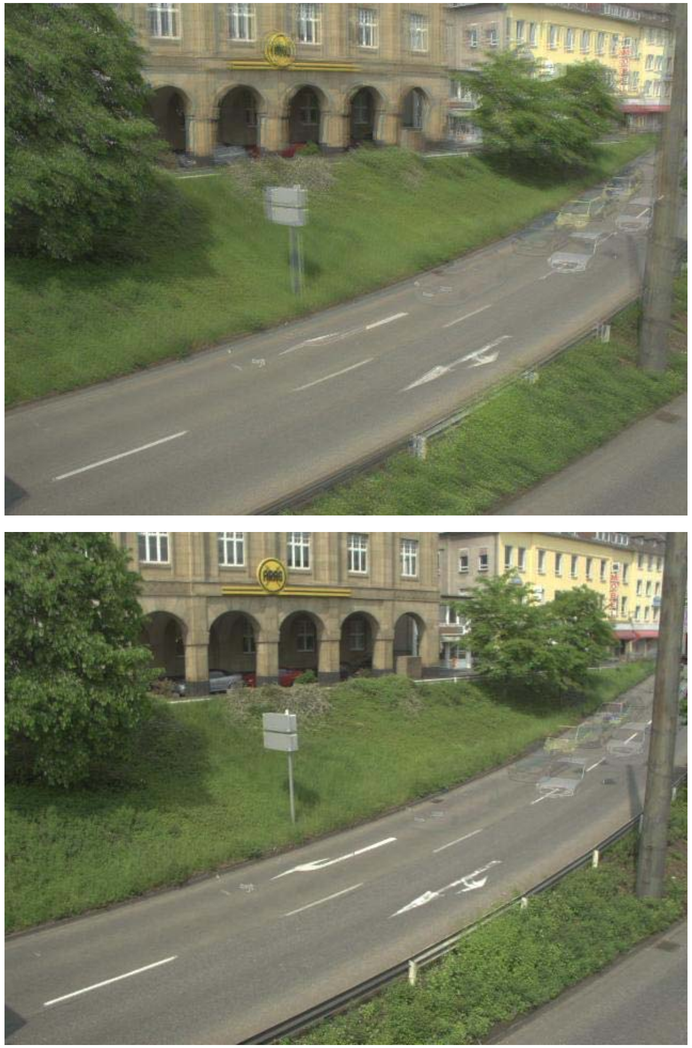




Optimization on the GPU



Downhill Simplex, 41 Iterations



approx. 5 x faster than the CPU

Details see [Grosch 2006]

Stencil Buffer

OpenGL Display Buffers

- Color Buffer
- Depth Buffer
  - Depth comparison per pixel
- Stencil Buffer
  - Render only in a section
  - Operations per pixel, e.g. counting

- Multiple Display Buffers exist
  - Double Buffer
    - Synchronization with display device, no flickering
  - Left/Right Buffer
    - Stereo Rendering

Stencil Buffer

- When using the Stencil Buffer, we have the possibility to select if a pixel is drawn into the color buffer or not (masking)

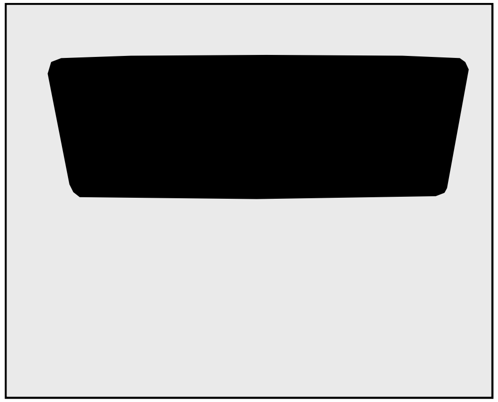
- In addition, we can perform calculations inside the stencil buffer (counting)

- The usage of the stencil buffer is a bit tricky… there are two commands
  - `glStencilFunc (…)` decides when and how we write into the Color Buffer ("Stencil-Test")
  - `glStencilOp (…)` decides how we write into the Stencil Buffer itself

Typical Example: Masking

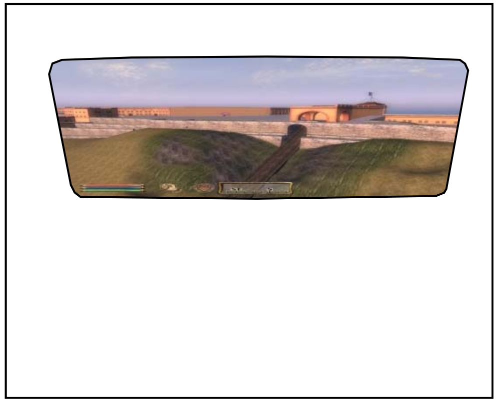
Color Buffer

Stencil Buffer



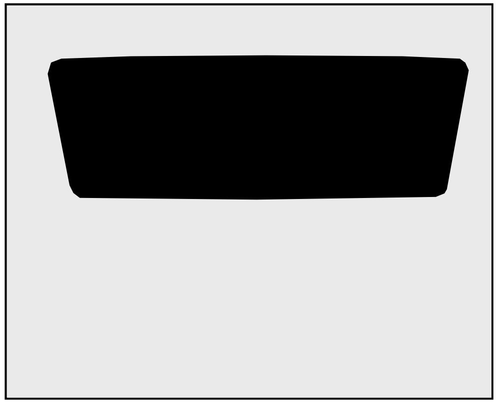
Clear the Stencil buffer and draw object in Stencil buffer, the Color buffer is not changed (glColorMask,read-only).

Color Buffer



Draw object in Color buffer, but only if the corresponding stencil value is set; the Stencil buffer is not changed.

Stencil Buffer



Stencil Function

Describe how the color buffer is modified

- `glStencilFunc(comparison, ref, mask);`

The Stencil test is performed per pixel:

(pixel & mask) comparison (ref & mask)

- Per pixel: Compare the content of the Stencil Buffer by a *comparison* function with a reference value *ref*
- We only write into the Color Buffer if this comparison was successful

Comparison can be: {`GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS`}

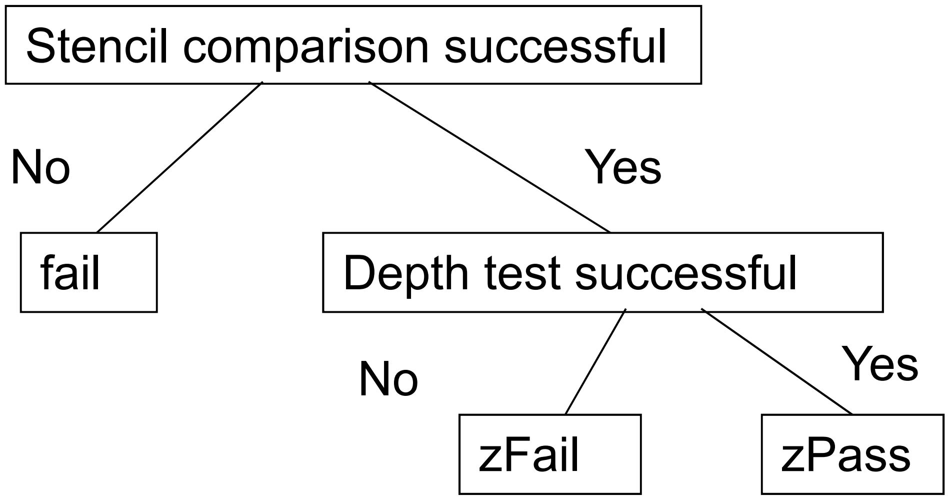
Example: `glStencilFunc(GL_EQUAL, 1, 0xff):` only write into the Color Buffer, if the corresponding pixel in the Stencil Buffer == 1

Stencil Operation

- What happens in the stencil buffer if we draw something ?
  - Generate a mask: write into Stencil Buffer
  - Use the mask: leave Stencil Buffer unchanged
- Stencil Buffer modification is controlled by

```
glStencilOp(fail, zFail, zPass)
```

  - 3 different cases



Stencil Operation

- **`glStencilOp(fail, zFail, zPass)`**
  - The possible operations are:
  - **`GL_KEEP`**: keep pixel in Stencil Buffer unchanged
  - **`GL_ZERO`**: clear Stencil pixel
  - **`GL_REPLACE`**: write reference value (see StencilFunc)
  - **`GL_INCR`**: increase stencil pixel by one
  - **`GL_DECR`**: decrease stencil pixel by one
  - **`GL_INVERT`**: invert stencil pixel

- Example: **`glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE)`**
  - **`glStencilFunc(GL_ALWAYS, 1, 0xff)`**
  - For all visible pixels (zPass), the content in the Stencil Buffer is replaced by the reference value (1)

Stencil Buffer

- Must be activated (and cleared)

```
glEnable( GL_STENCIL_TEST );
glClear( GL_STENCIL_BUFFER_BIT );
```

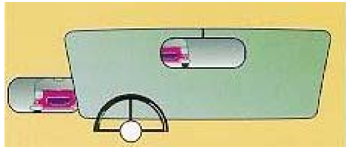
- The default window in GLFW contains a Stencil Buffer
- Applications:
  - Arbitrary masks
    - **`glStencilOp`** with **`GL_REPLACE`** : write reference value
    - **`glStencilFunc`** with **`GL_EQUAL`** : draw only in regions with the reference value
  - Numerical operations as Stencil Operation
    - use **`GL_INCR`** to count how often a pixel is drawn

Stencil Buffer

- There is no special function to write into the stencil buffer
- Example for mask generation and usage:

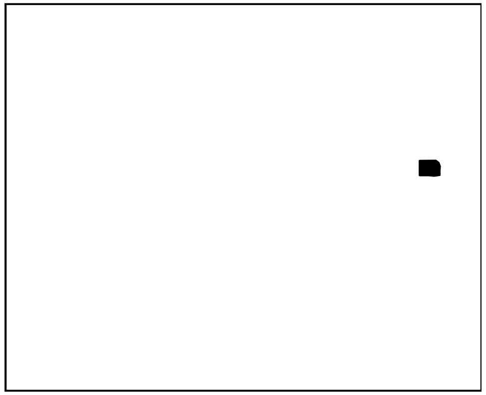
```
glEnable(GL_STENCIL_TEST);
glClear( GL_STENCIL_BUFFER_BIT );
glStencilFunc(GL_ALWAYS,1,0xff);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
/* draw the mask ( = 1 ) here */
glStencilFunc(GL_EQUAL,1,0xff);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glClear( GL_COLOR_BUFFER_BIT );
/* draw the image (only if Stencil == 1) */
```

Example: Mirror

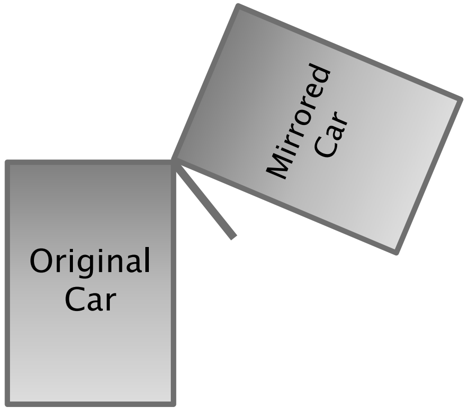


Method

- Clear stencil buffer
- Draw scene
- Draw mirror and fill Stencil buffer (create mask)
- Draw mirrored scene
  - Scale with −1 along the mirror normal
- Use the mask: write into Color buffer only if mask in Stencil buffer is set
- See exercise

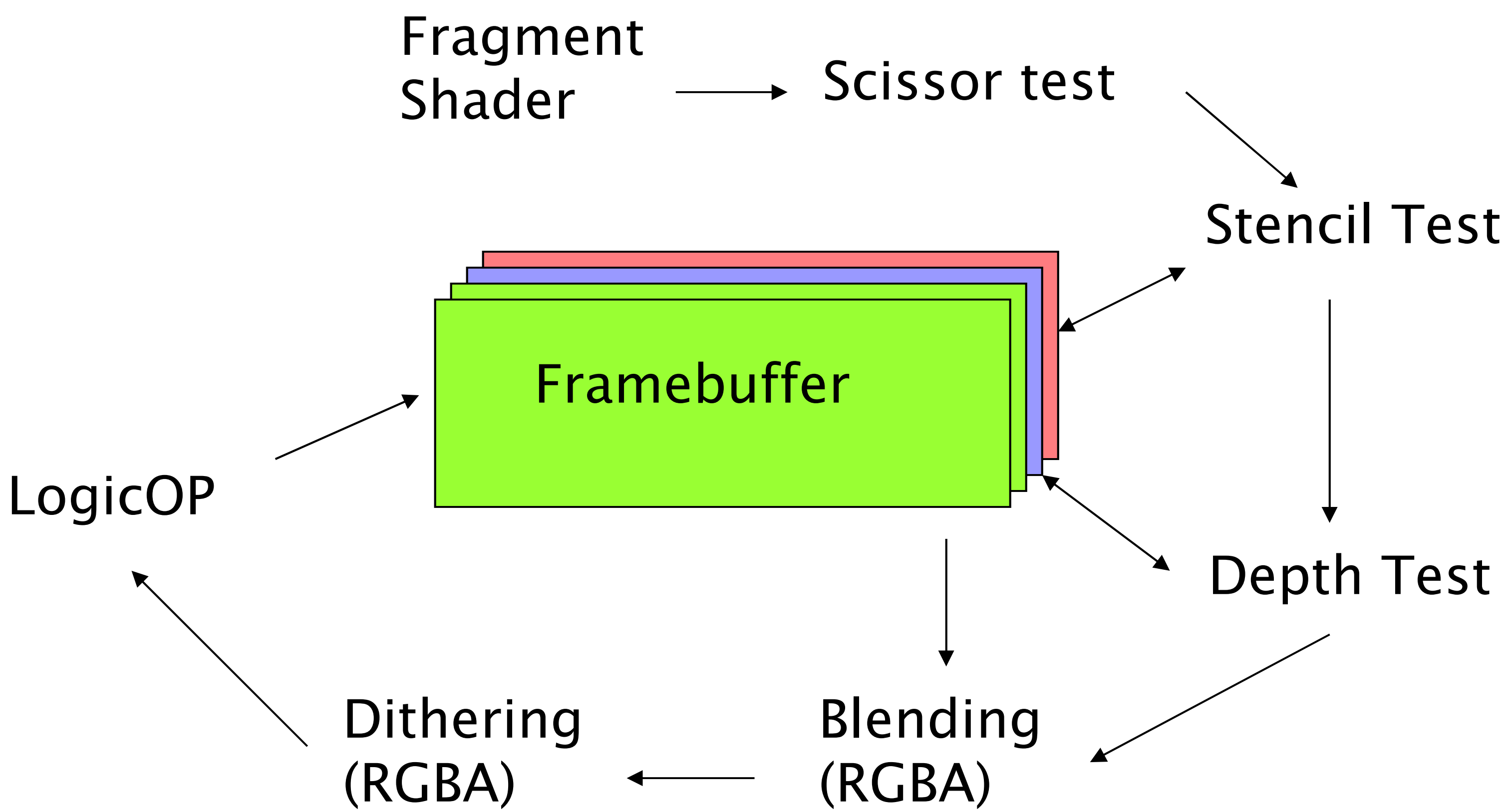


Stencil buffer



Pixel Pipeline

Several tests and operations are performed before the final pixel color is written into the color buffer…



Hint: Modern GPUs have an **early z** option: Scissor, Stencil and Depth test can be performed before the Fragment Program. This can not be controlled manually.
Some operations do not work in combination, e.g. Blending and LogicOp

Geometry Buffer

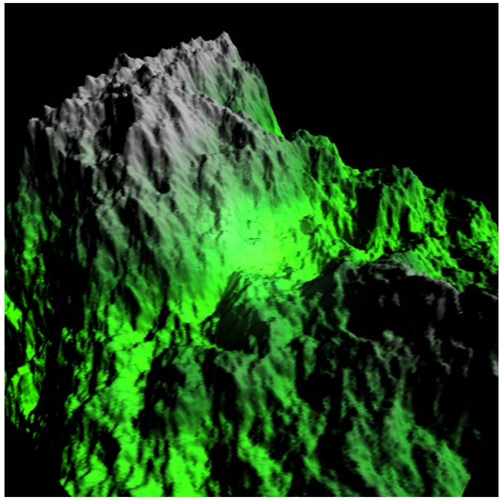
TU Clausthal

Screen-space Computations

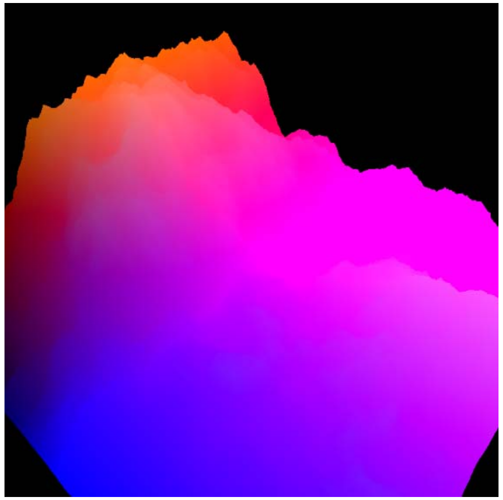
- In real-time rendering, we often work in **screen-space**
  - Here we have information about the neighbor pixels
  - A (complex) fragment shader is executed exactly once per pixel
- Idea
  - Render the complex scene with simple shaders that generate several textures which contain the information that we need per pixel (**Geometry Buffer**)
    - Typical information per pixel: Position, normal, color
  - Afterwards, draw a **screen-filling quad**
    - Now activate the complex fragment shader
    - Read the required information from the textures

Geometry Buffer Example

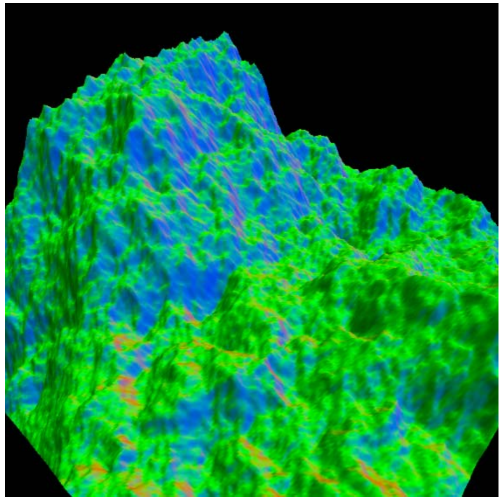
Terrain (Computer Graphics 1 exercise)



Illuminated Terrain



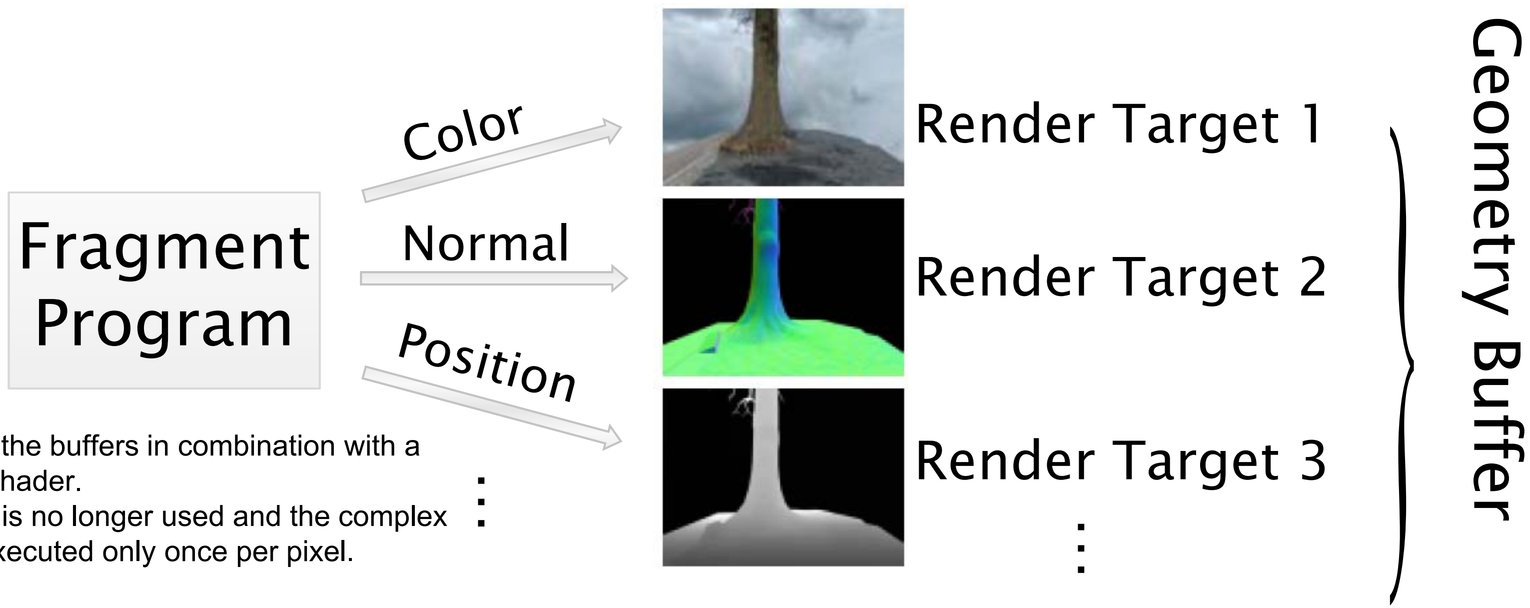
XYZ Position
mapped to RGB



XYZ Normal
mapped to RGB

Multiple Render Targets (MRTs)

- To render directly into a texture with OpenGL we a need a so-called **Framebuffer Object (FBO)**
- We then assign a **Render Target** to this FBO
  - A Render Target is basically the texture we write into from our Fragment Program
- We can also assign **Multiple Render Targets (MRT)**
  - Now only **one render pass** of the (complex) geometry is required that writes into multiple textures



Geometry Buffer: Use the buffers in combination with a (complex) Fragment Shader.
The original geometry is no longer used and the complex Fragment Shader is executed only once per pixel.

Multiple Render Targets

```
// generate texture Ids and assign to texture units
…
// MRT FBO for position/normal/color
glGenFramebuffers(1, &mrtFB);
glBindFramebuffer(GL_FRAMEBUFFER, mrtFB);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, positionTextureId, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
                       GL_TEXTURE_2D, normalTextureId, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2,
                       GL_TEXTURE_2D, colorTextureId, 0);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_TEXTURE_2D, depthTextureId, 0);

GLenum buffers[3] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
    GL_COLOR_ATTACHMENT2};
glDrawBuffers(3, buffers);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Generate a Framebuffer Object

Attach three color textures

Attach a depth texture → z Buffer

Define order of the buffers we draw into

Hint: When activating (binding) a Framebuffer Object, nothing is drawn in the normal Framebuffer !

Shaders for MRT Generation

```
#version 440
// MRT Vertex Shader

layout (location = 0) in vec4 vPosition;
layout (location = 1) in vec3 vNormal;
layout (location = 2) in vec4 vColor;

out vec4 position;
out vec3 normal;
out vec4 color;

uniform mat4 modelviewProjection;

void main()
{
    position = vPosition;
    normal = vNormal;
    color = vColor;

    gl_Position = modelviewProjection *
    vPosition;
}
```

```
#version 440
// MRT Fragment Shader

in vec4 position;
in vec3 normal;
in vec4 color;

layout (location = 0) out vec4 fPosition;
layout (location = 1) out vec3 fNormal;
layout (location = 2) out vec4 fColor;


void main()
{
    fPosition = position;
    fNormal = normal;
    fColor = color;
}
```

The vertex shader passes through all the information we need in the geometry buffer

The fragment shader writes the information into several render targets (textures)

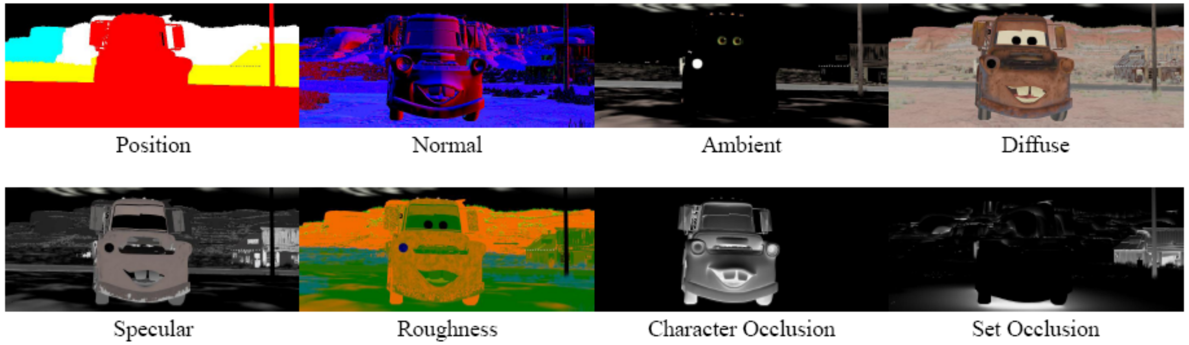
MRT Usage

- Draw a screen-filling quad
- Read information from the textures at current pixel position
  - ensure that a possibly complex fragment shader is executed only once per pixel

```
layout (binding = 0) uniform sampler2D positionTexture;  // we use texture units 0..2
layout (binding = 1) uniform sampler2D normalTexture;
layout (binding = 2) uniform sampler2D colorTexture;

out vec4 color;

void main()
{
   ivec2 pixelCoords = ivec2(gl_FragCoord.xy);               // check where we are

   vec4 pos = texelFetch(positionTexture, pixelCoords, 0);   // read the required infos
   vec4 normal = texelFetch(normalTexture, pixelCoords, 0);
   vec4 color = texelFetch(colorTexture, pixelCoords, 0);

   // make any calculation with the information of the buffers (e.g. illumination)…

  color = ….
}
```

Geometry Buffer Examples

Illuminate a 3D scene only with the information from the position/normal/…textures
By reading the neighbor pixel information, (ambient) occlusion effects can be rendered



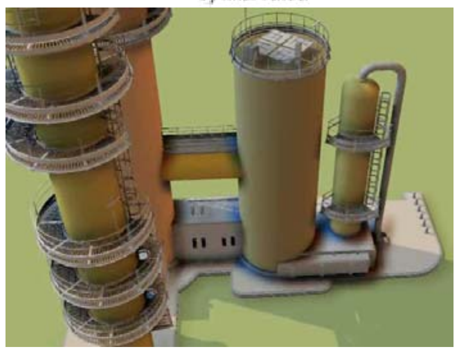

a) lpics render


b) final render

Lpics (Pellacini et al. 2005):
Preview 0.1 seconds, Final Render several minutes


Starcraft (Fillion et al. 2008)


CryTek (Mittring et al. 2007)


SSDO (Ritschel et al. 2009)

The Render Buffer

- Optionally, a so-called Render Buffer can be used in OpenGL for each texture of a Framebuffer Object
  - **`glGenRenderbuffer, glBindRenderbuffer, glRenderbufferStorage`**
- The Render Buffer can then be attached to the currently bound Framebuffer Object
  - **`glFramebufferRenderbuffer(target, attachment, renderbuffertarget, renderbuffer)`**
    - **`target = GL_FRAMEBUFFER`**
    - **`attachment = GL_COLOR_ATTACHMENT, GL_DEPTH_ATTACHMENT, ..`**
    - **`renderbuffertarget = GL_RENDERBUFFER`**
    - **`renderbuffer = 0`**
- This is an alternative to directly assigning textures to the framebuffer object

That´s all for today

- Next week: Geometry Shader