Framebuffers

Offscreen Rendering and Post Processing

Francesco Andreussi

Bauhaus–Universität Weimar

6 December 2018

Bauhaus-Universität Weimar

Faculty of Media

The Framebuffer

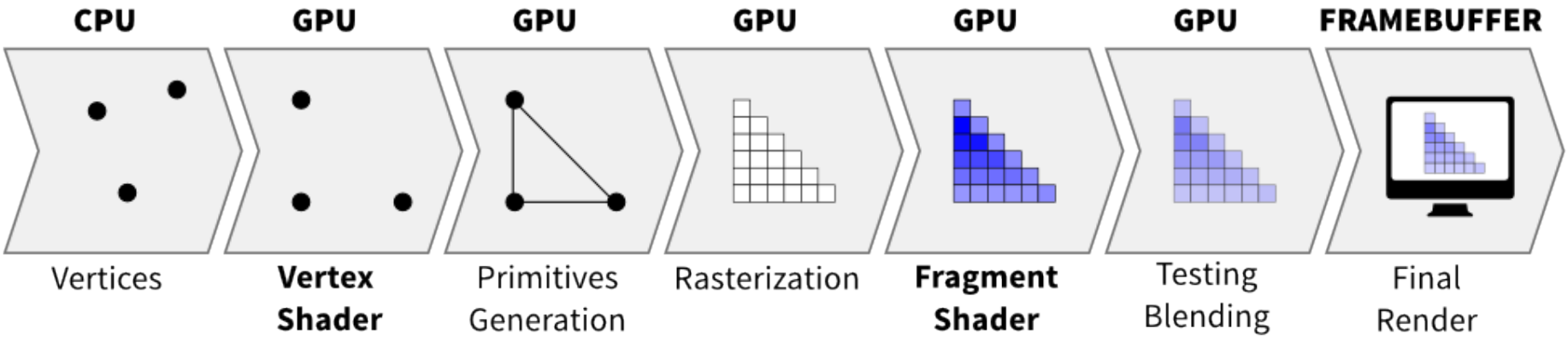

Fig. 1: The Rendering Pipeline (again (again))

The Framebuffer is the **final result** of the Rendering Pipeline.
It is a memory buffer where the colours of every pixel of the screen are saved, when the frame is ready to be rendered. It is sometimes called screen/video/regen(eration)/off–screen buffer and its dimension depends on the amount of pixels to be drawn (monitor resolution) and color depth (1/4/8/16/24 bits).

Fragment Operations & Tests (1)

Just before writing the Framebuffer, the GPU performs few Tests in order to be sure of rendering only what the programmer wants.

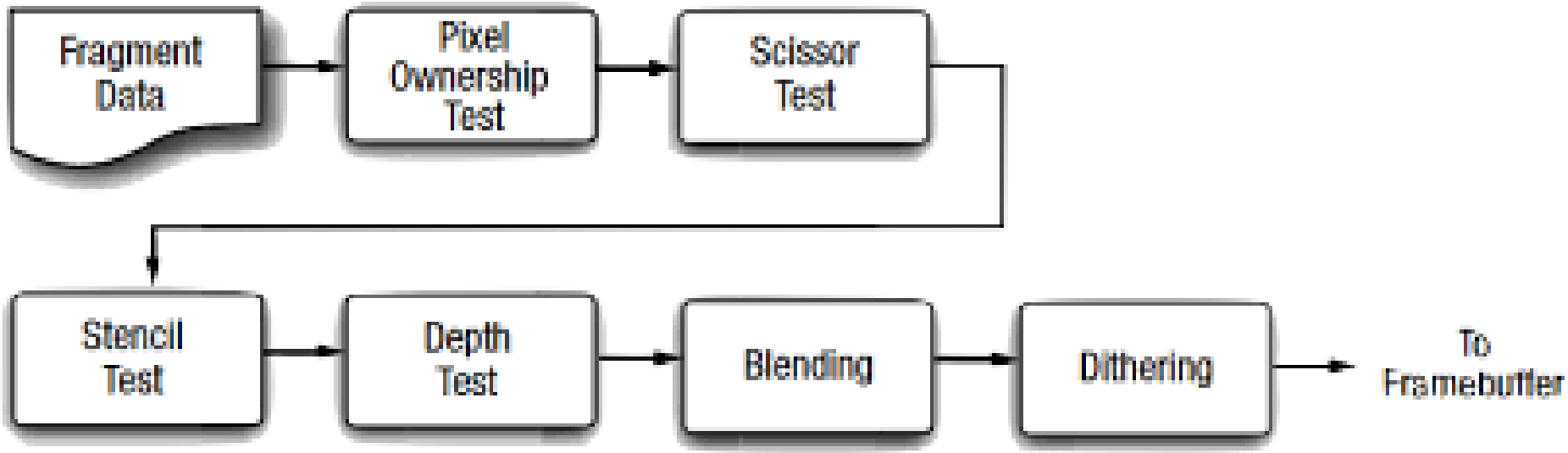

Fig. 2: The Final Fragment Operations (again (yes, this one too))

Fragment Operations & Tests (2)

- **Pixel Ownership Test** (mandatory): discards data for pixels of the window that are covered by other elements on the display.
- **Scissor Test** (optional): discards data for pixels lying outside a certain *scissor rectangle* defined by the programmer.
- **Alpha Test** (optional): discards data of fragments that have a transparency value not compliant with a dev–defined threshold.
- **Stencil Test** (optional): discards or modifies a fragment if a specific relationship with a dev–defined value (*stencil buffer*) is not satisfied.
- **Depth Test** (optional): discards a fragment if its depth value does not satisfy an dev–specified relationship with a value (*depth buffer*).
- **Blending** (optional): not a test but an z–dependent operation, that overwrites the colour in a certain position in the Framebuffer, its combination with the colour of a fragment in the same position.
- **Dithering** (optional): improves the colour appearance, especially for low–quality output devices. [link]

Framebuffer

A Framebuffer can contain three types of buffers (*attachments*): **color buffer** stores the `glColor` data output by the Fragment Shader, the **depth** and the **stencil buffer** are matrices of values defined in the Fragment Shader as well with the same dimensions of the screen, they are used in the per–fragment operations.

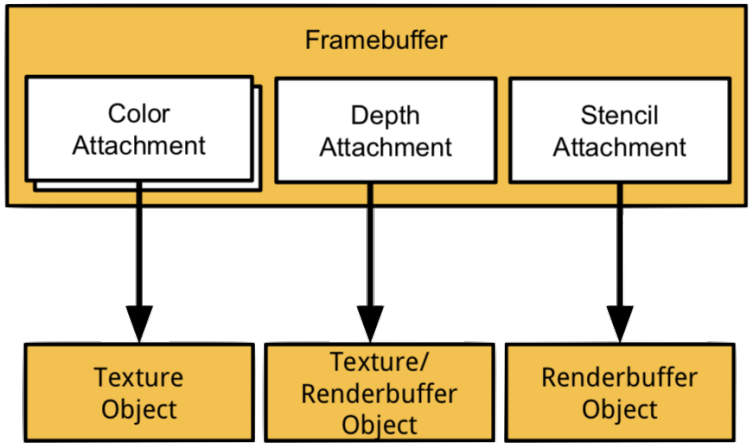

Fig. 3: Framebuffer Configuration

OpenGl allows the presence of custom Framebuffer Objects, in addition to the Default Framebuffer and, obviously, allows only one Framebuffer to be active at a time.

It is possible to clear all the buffers of a Framebuffer with the command `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT)`.

Renderbuffer

Renderbuffers are OpenGL objects similar to textures used for depth and stencil buffers.

They can only be used as render target and when they are bound to a Framebuffer (using `GL_RENDERBUFFER`).

```
glGenRenderbuffer(1, &rb_handle)
glBindRenderbuffer(GL_RENDERBUFFER, rb_handle)
glRenderbufferStorage(GL_RENDERBUFFER, format, width,
height)
```

Default Framebuffer

The Default Framebuffer is created with the OpenGL Context, configured during the Context initialisation and bound by default with index 0.
Its content is what is displayed on screen, and its dimensions changes automatically accordingly to the window size.

It contains a depth buffer, a stencil buffer and 4 colour buffers: `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, `GL_BACK_RIGHT`. Those are used in order to enable **stereoscopic rendering** (left/right) and **double rendering** (front/back).

In double buffering, the front buffers are displayed while the back are written. Then, the buffers the data in the back ones are passed to the front and the process starts again.

Framebuffer Objects

Framebuffer Objects are OpenGL Objects created, configured by the user at runtime. They can contain up to `GL_MAX_COLOR_ATTACHMENTS` Colour Attachments, one Depth and one Stencil Attachment.

It important to remember that **all** the held buffers **must have the same dimensions** and they (can) be written with the varying output of the Fragment Shader.

Framebuffer Definition and Usage

Define Framebuffer

```
glGenFramebuffers(1, &fbo_handle)
glBindFramebuffers(GL_FRAMEBUFFER, fbo_handle)
```

Define Attachments (one call for each attachment to be defined)

```
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENTi /
GL_DEPTH_ATTACHMENT, tex_handle, mipmap_level)
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT / GL_STENCIL_ATTACHMENT,
GL_RENDERBUFFER, rb_handle)
```

Define which Buffers to Write

```
GLenum draw_buffers[n] = {GL_COLOR_ATTACHMENT0, ...}
glDrawBuffers(n, draw_buffers)
glDrawBuffers(1, GL_DEPTH_ATTACHMENT/GL_STENCIL_ATTACHMENT)
```

Check that the Framebuffer can be written; hence, that...

```
glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
GL_FRAMEBUFFER_COMPLETE
```

Offscreen Rendering

Theory

Since **only** the Default Framebuffer content is actually displayed, the content of a Framebuffer Object must be transferred to the Default.

Direct data transfer via **blitting** (special copy–pasting operation for fragment values between Framebuffers) is slow and not so flexible.

A better technique is defining a quad in front of the "Main Camera", feeding (through the Rendering Pipeline) the Default Framebuffer.

This method is commonly used for picture–in–picture, dynamic and high–detailed mirroring effects and post–processing.

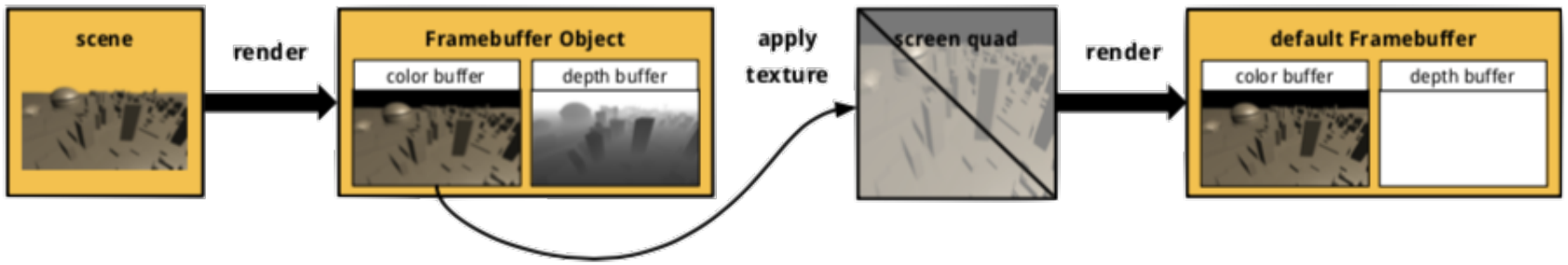

Fig. 4: Offscreen Rendering Scheme

Offscreen Rendering

Screen Quad

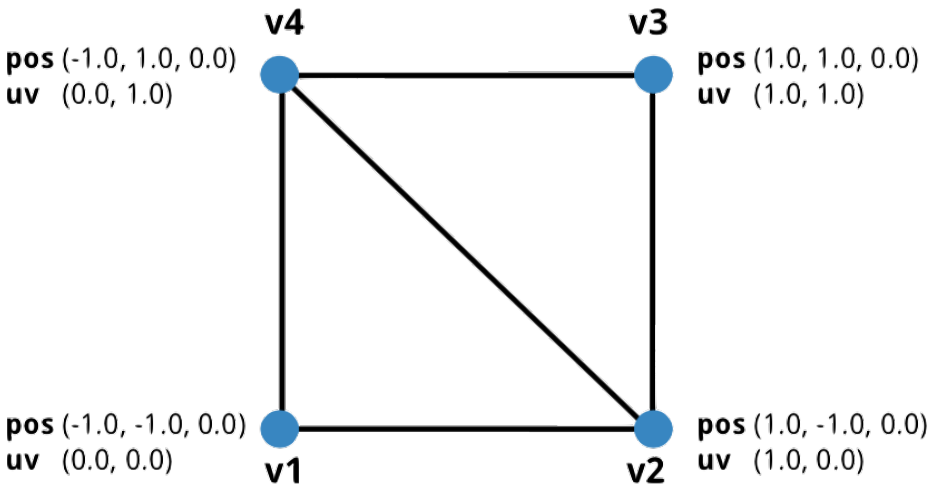

Fig. 5: The Screen Quad

No transformation needed because it is already defined in Normalised Device Coordinates.
It is possible to define the vertices in the order `v1`, `v2`, `v3`, `v4`, and render them as a `GL_TRIANGLE_STRIP`.
In addition, there are needed a Vertex Shader, calculating only Texture Coordinates, and a Fragment Shader, for sampling and mapping the texture passed from the Framebuffer Object and output the correct color to the Default Framebuffer.

Post–Processing Effects (1)

The Post–Processing capabilities are limited only by your imagination! It is possible to find interesting effects as "Filters" in tons of mobile apps. Here, three only three effects will be analysed:

- **Grayscale**: computing a value taking in account how much light is emitted by a pixel; however, the magnitude of the RGB `vec3` is not accurate enough because the perceived luminance is different for the three channels. Hence, the values should be weighted in this way: $luminance = 0.2126 * R + 0.7152 * G + 0.0722 * B$.
- **Mirroring**: inverting the texture coordinates around the desired axes.
- **Blur**: overwriting the colour of the target pixel with the weighted sum of the colours of the neighbouring pixels. It requires to know the pixel size and, hence, the offset for reading the colours of the neighbours.

Post–Processing Effects (1)

- **Blur (continued**: It can be computed from the relation `pass_TexCoord = pixel_size · gl_FragCoord`, where `pass_TexCoord` is the position of the texel in **texture** space and `gl_FragCoord` is its position in **window** space. The weights for the computation are taken from a Gaussian Kernel: a square matrix (the bigger it is, the less blurred is the effect and the more is hard for the GPU) looking like the one here.

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

Fig. 6: A 3x3 Gaussian Kernel

Thanks for the Attention!