OpenGL
Framebuffer Objects

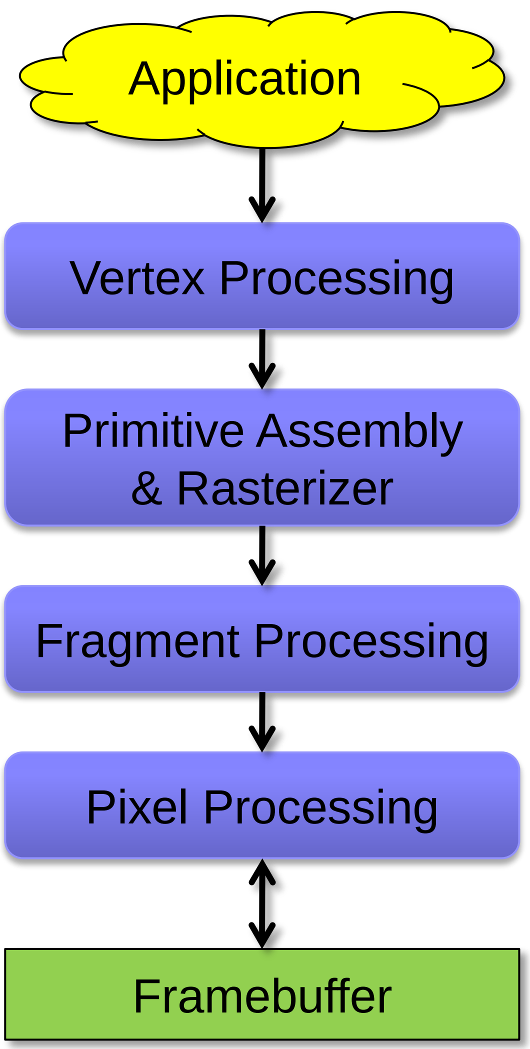
Marco Di Benedetto

Visual Computing Laboratory – ISTI – CNR, Italy

OpenGL Roadmap

- 1.0 - Jan 1992 - First Version
- 1.1 - Jan 1997 - Vertex Arrays, Texture Objects
- 1.2 - Mar 1998 - 3D Texturing, Separate Specular Color, Vertex Array draw element range
- 1.2.1 - Oct 1998 - Multi-Texturing
- 1.3 - Aug 2001 - Compressed Textures, Cube Maps, Multi-Sampling
- 1.4 – Jul 2002 – Depth Textures, HW Shadowing, Separate Blend, Extended Texture Addressing
- 1.5 – Jul 2003 – Vertex Buffer Objects, Occlusion Queries, Extended Shadow Functions
- 2.0 – Sep 2004 – Vertex and Fragment Shaders, Multiple Render Targets, Separate Stencil
- 2.1 – Jul 2006 – Pixel Buffer Objects, sRGB
- 3.0 – Jul 2008 – Framebuffer Objects, HW Instancing, Vertex Array Objects
- 3.1 – Mar 2009 – Texture and Uniform Buffer Objects, Integer Textures, Fast Buffer Copy (OpenCL)
- 3.2 – Aug 2009 – Geometry Shaders, Multisampled Textures, Synch and Fence Objects
- 3.3 – Mar 2010 – Sampler Objects, Profiles Introduction
- 4.0 – Mar 2010 - Tessellation Shaders, Per-Sample Fragment Shaders, Shader Subroutines, Double Precision

The Abstract Graphics Pipeline



1. The application specifies vertices & connectivity.
2. The VP transforms vertices and compute attributes.
3. Geometric primitives are assembled and rasterized, attributes are interpolated. Culling occurs here.
4. The FP computes final “pixel” color.
5. The PP (output merger) writes pixels onto the FB after stencil/depth test, color blending.

Introductory Example

- Scenario:
  - We must draw a control room with a live action security camera TV
  - The camera is recording a scene simulated by our system

- → We must be able to draw the TV while it is showing a dynamic, computer-generated scene

- In practice:
  - Render the scene as seen by the security camera
  - Use the result of the rendering as a texture mapped on the TV screen

Use the Color Buffer as a Texture

- We want to be able to use the framebuffer content as a texture
- OpenGL can do this since v1.0
  - glCopyTexImage2D() : copy the content of the color buffer to a texture
- Problems
  - *Pixel Ownership Test*: if the window we are rendering to is partially occluded by something (e.g. other overlapping windows), the occluded pixels will not be written
    → *holes* in the texture
  - We must pay a copy operation

Render-to-Texture (RTT)

- Ideally, we want to draw *directly* on the texture to avoid windows-manager issues and memory copy operations

- Early solution: PBuffers
  - Framebuffers that can be used as textures
  - PBuffers have their own OpenGL context → data must be shared
  - GL context switches → slow synchronization → slow slow slow ...
  - ... but the idea was good!

- Modern solution: Framebuffer Objects
  - Draw *directly* into a texture

The Framebuffer

- A set of *ancillary* buffers: Color - Depth - Stencil

- Double Buffering
  - If we render several objects, one at a time, directly on the memory region used by the screen, we may experience *flickering* in the image
  - To avoid flickering, rendering is done on a ***back*** (frame)buffer (which is not visible), while the screen shows the ***front*** (frame)buffer
  - When rendering is done, buffers are swapped (or flipped)
  - → the screen can present the completely composed image at once

- OpenGL defines:
  - *Main* front/back buffers
  - *Left* and *Right* front/back buffers (for stereo rendering)
  - *Auxiliary* buffers (how many is implementation dependent)

Framebuffer Object (FBO)

- Simply put, the result of a rendering is written in a memory region
- Historically, the front and back buffers are the regions of memory chips written by the graphics hardware accelerator and read by the screen interface

- Texture images are regions of the graphics memory

- With OpenGL FBOs we can tell the hardware:
  - “this is your framebuffer (color, depth, stencil) memory pointer, write there”
  - We can render-to-texture!

FBO

- A FBO is enhough flexible to hold just the ancillary buffers it needs
  - Any combination of color/depth/stencil
  - Some actual implementation have depth-stencil buffers tied
- Render Targets: what a FBO can contain
  - Textures
  - Renderbuffers (for texture formats that are not writable)
- Steps:
  - Create the FBO
  - Attach textures or renderbuffers to attachment points
  - Bind as the target framebuffer

FBO

```
// construction at application init
GLuint fbo = 0;
glGenFramebuffers(1, &fbo);

glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,  GL_TEXTURE_2D, depth_tex, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, color_tex, 0);
const Glenum draw_buffers[] = { GL_COLOR_ATTACHMENT0 };
glDrawBuffers(sizeof(draw_buffers)/ sizeof(draw_buffers[0]), draw_buffers));
glBindFramebuffer(GL_FRAMEBUFFER, 0); // rebind main framebuffer (screen)

// usage
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
draw_texture_content();
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// render scene and use color_tex
render_scene();

// destruction at application exit
glDeleteFramebuffers(1, &fbo);
fbo = 0;
```

Multiple Render Targets (MRT)

- FBOs can contain multiple color buffers
- The actual composition is
  - A set of N color buffers (N is implementation dependent)
  - Zero or one depth buffer
  - Zero or one stencil buffer

- What it means
  - We have one depth/stencil buffer
  - We can output *simultaneously* N different *color* values

- Fragment shader:
  - gl_FragData[***i***] outputs to ***i***th color target
  - gl_FragColor is an alias for gl_FragData[0]

FBO - MRT

```
// construction at application init
GLuint fbo = 0;
glGenFramebuffers(1, &fbo);

glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,  GL_TEXTURE_2D, depth_tex,  0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, color_tex0, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, color_tex1, 0);
const Glenum draw_buffers[] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers(sizeof(draw_buffers)/ sizeof(draw_buffers[0]), draw_buffers));
glBindFramebuffer(GL_FRAMEBUFFER, 0); // rebind main framebuffer (screen)

// usage
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
draw_textures_content();
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// render scene and use color_tex0 and color_tex1
render_scene();

// destruction at application exit
glDeleteFramebuffers(1, &fbo);
fbo = 0;
```

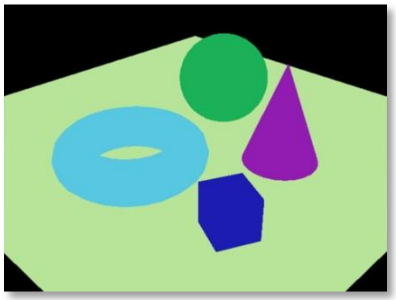
FBO Usage

- Introductory Example
  - The target color texture will be used as the TV texture
- Deferred Shading:
  - Pixels are written several times in a complex scene (depth complexity)
  - The closest one contributes to the final image
  - What if we perform complex calculations (e.g. lighting) just on visible pixels?

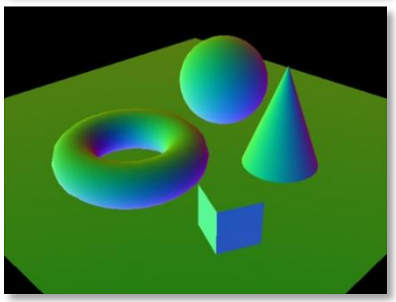
Deferred Shading

- Multipass rendering algorithm
- For each pixel, store in a set of textures all the values we need for our calculations
  - Normals - Vertex position - Base color
  - The set of textures we render to is called G-Buffer
- For each light source and for each pixel (full screen pass) fetch data from the G-Buffer and use them for lighting calculations

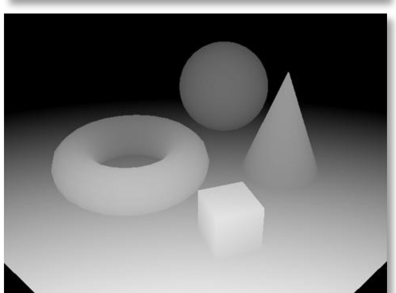
Deferred Shading



Base Color

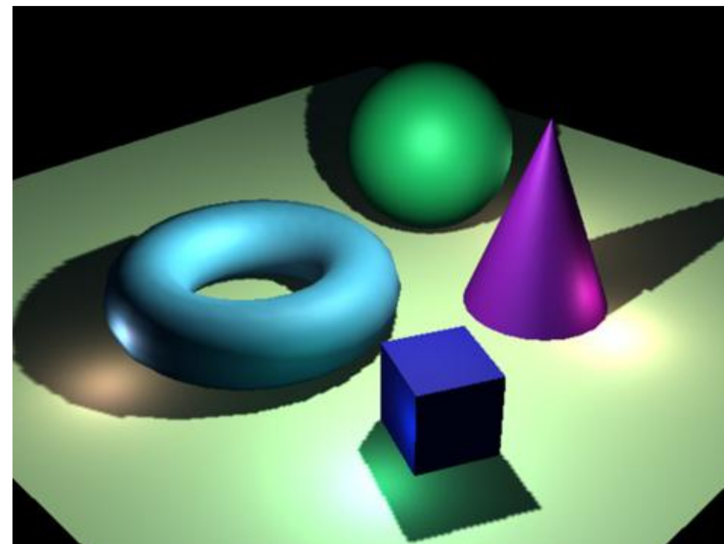


Normal



Depth

G-Buffer



Final Composed Image

Other Usages

- Post processing
  - Draw into a texture, then use image processing techniques
- Apply edge detection to perform selective antialiasing
- Procedural textures
- Screen Space Ambient Occlusion
- …

Render to Texture Arrays

- GL_TEXTURE_2D_ARRAY: a stack of 2D textures
- The active layer can be selected in the Geometry Shader

```
// when creating the FBO
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, color_tex_array, 0);
```

```
// geometry shader
// . . .
gl_Layer = 0;
gl_Position = …;
EmitVertex();
EndPrimitive();

gl_Layer = 1;
gl_Position = …;
EmitVertex();
EndPrimitive();
```

EOF