DigiPen
INSTITUTE OF TECHNOLOGY

CS300
Render to Texture

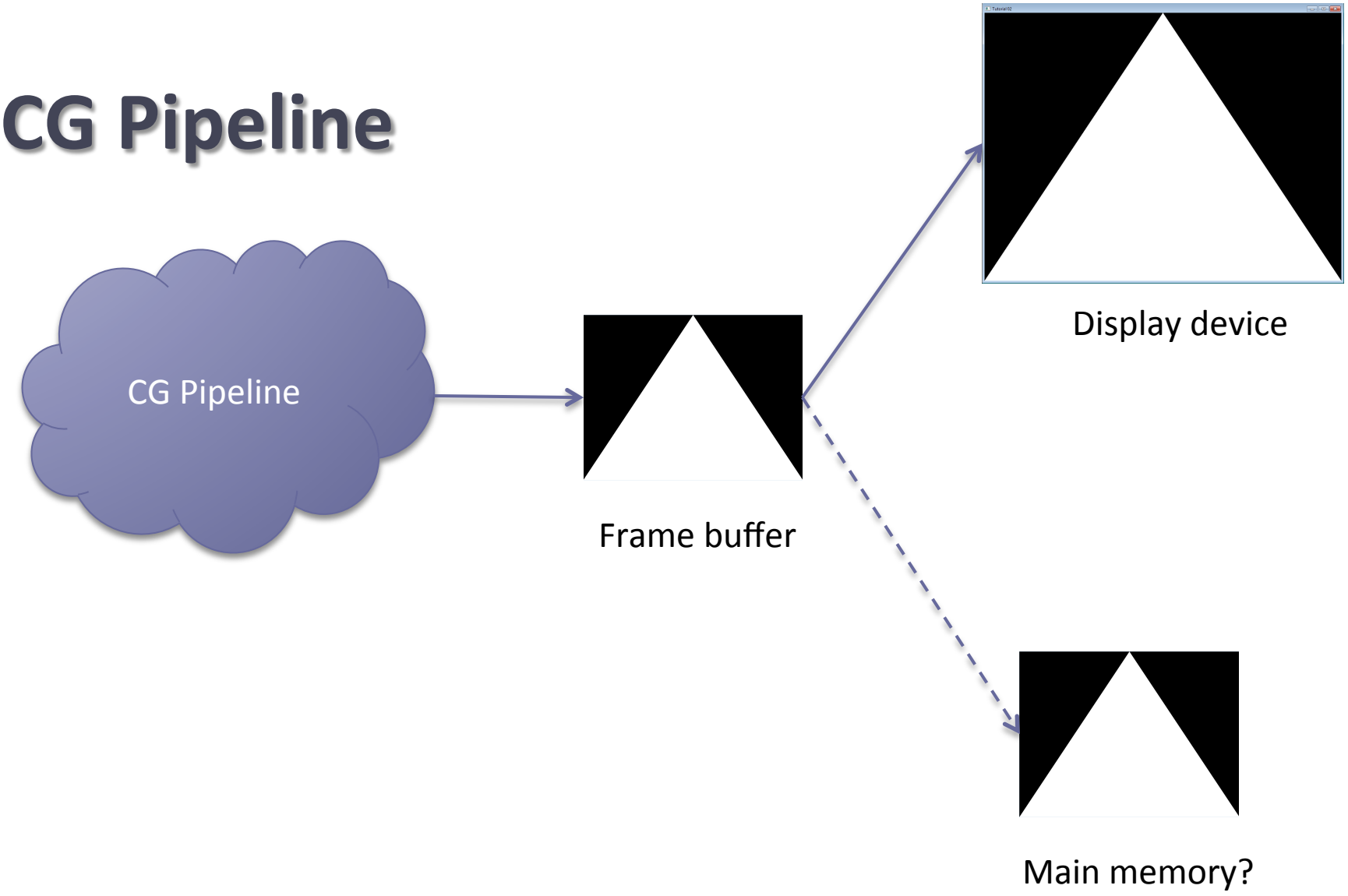
A swiss-army knife for advanced rendering

CG Rendering pipeline

- Aim
  - Create a 2D representation of a 3D scene
- Final output
  - Special area in video memory – “Frame Buffer”

- Frame Buffer – A block of memory that is polled by hardware for updates
  - Transparent operation as far as the programmer is concerned

CG Pipeline

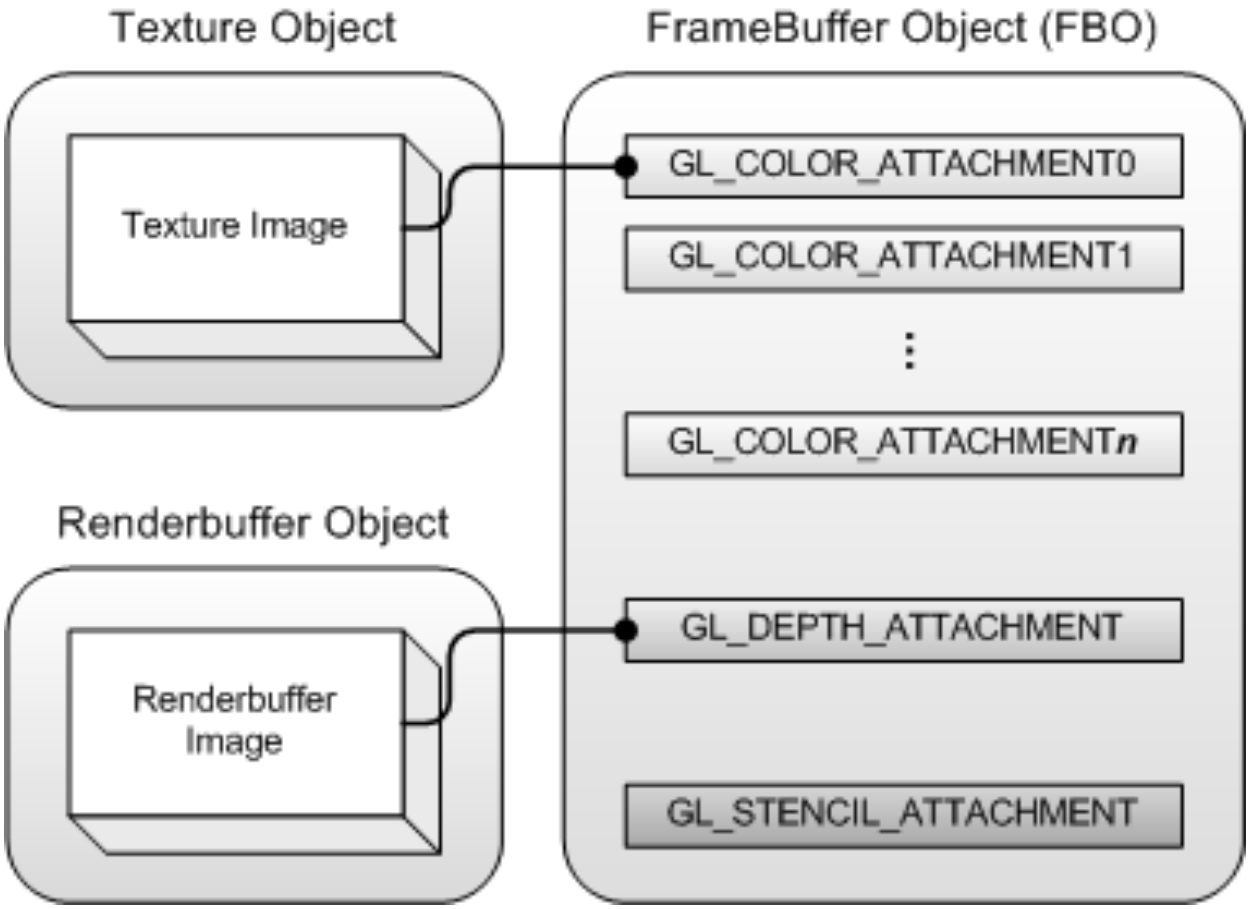


Using custom frame buffer

- The FB is not passed on to the display device, but stored in-core (CPU / GPU)
- Can be treated as
  - buffer of data (unstructured array)
  - stream of information (logically aligned data records)
  - visual output (image/texture)

- If the output is interpreted as an image and used in further processing as input → Render-to-texture.
- If output is used as renderbuffer → Offscreen rendering

Overall schematic



Frame Buffer Object

- OpenGL data structure that encapsulates what information is passed from OpenGL to underlying windowing system
- Color, stencil and depth values
- Bypass the window-system pipeline
  - Application should provide all the data storage for applicable slots

Step 1: Get handle to the FB Object

- FrameBuffer – output of our CG pipeline
- No “pointers” on the GPU
  - so create a handle to the FB object

```
GLuint FramebufferName = 0;                    // handle to framebuffer
glGenFramebuffers(1, &FramebufferName);  //  create the FB object handle
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName); // assign the handle to an
                                                    // object of type GL_FRAMEBUFFER,
                                                    // GL_DRAW_FRAMEBUFFER, or
                                                    // GL_READ_FRAMEBUFFER
```

FBO

- Binding to current handle overwrites any previous binding
  - To revert back to the default FB, use '0'
    - glBindFrameBuffer(GL_FRAMEBUFFER, 0);
- Immediately after binding
  - GL_COLOR_ATTACHMENT0 = GL_NONE
  - GL_DEPTH_ATTACHMENT = GL_NONE
  - GL_STENCIL_ATTACHMENT = GL_NONE
  - No Accumulation Buffer provided

Step 2: Create a texture

- Feedback loop
  - Output of step 1 will be used as input in subsequent passes
- Create standard OpenGL texture
  - width, height = viewport width and height
  - no actual data pointer
    - last argument = '0'
  - Data format – GL_RGB
    - Can customize!
    - Use floating point for HDR Rendering

```
// The texture we're going to render to
GLuint renderedTexture;
glGenTextures(1, &renderedTexture);

// "Bind" the newly created texture
// all future texture functions will modify this texture
glBindTexture(GL_TEXTURE_2D, renderedTexture);

// Give an empty image to OpenGL
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB, 1024, 768, 0,GL_RGB, GL_UNSIGNED_BYTE, 0);
```

Step 2.5 : Add Depth buffer

- Generate the handle to depth buffer
- Bind it as an object of type GL_RENDERBUFFER

- Set memory storage for width and height of the FB
- Choose data type of this buffer

- Crucial step
  - "Attach" the render buffer as a "depth buffer" to receive depth values from Opengl

```
// The depth buffer
GLuint depthrenderbuffer;

glGenRenderbuffers(1,
&depthrenderbuffer);

glBindRenderbuffer(GL_RENDERBUFFER,
depthrenderbuffer);

glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT, 1024, 768);

glFramebufferRenderbuffer(
    GL_FRAMEBUFFER,
    GL_DEPTH_ATTACHMENT,
    GL_RENDERBUFFER,
    depthrenderbuffer);
```

Step 3: Now add the “render target”

- Texture pointed to by “renderedTexture” (Step 2) is attached as output of the OpenGL pipeline

```
// Set "renderedTexture" as our colour attachement #0
glFramebufferTexture(
    GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0,
    renderedTexture,
    0);

// Set the list of draw buffers.
GLenum DrawBuffers[1] = {GL_COLOR_ATTACHMENT0};

Gluint numBuffers = 1;
glDrawBuffers(numBuffers, DrawBuffers);
```

Important Issues to consider

- Size and format of FBO is completely controlled by the OpenGL application
- FBOs are not affected by system events
  - window resizing, display mode changes
- The FBOs always return a valid pixel ownership test
- No concept of front and back buffers
  - Have to explicitly set GL_BACK after reverting to default FBO
- No multisample buffer

Shader code

- Use layout specification on out variable

```
layout(location=0) out vec3 color
```

location = n, where n is the index of the attached buffer in the glDrawBuffers call

If we add the depth attachment as a render-target, then we can also add the following code:

```
Glenum DrBuffers={GL_COLOR_ATTACHMENT0,
GL_DEPTH_ATTACHMENT, GL_COLOR_ATTACHMENT1}

glDrawBuffers( numBuffers, DrBuffers );
```

Shader code (contd.)

CPU Code

```
Glenum DrBuffers={GL_COLOR_ATTACHMENT0,
GL_DEPTH_ATTACHMENT, GL_COLOR_ATTACHMENT1}
```

GPU Code

```
layout(location = 0) out vec3 color;
layout(location = 1) out float depth;
layout(location = 2) out vec3 normal;
```

Note: The index is the location in the array, NOT the index referred in GL_COLOR_ATTACHMENT*i*

Switching FBOs

- Very expensive – avoid it
- Switch attachment objects on same FBO
  - glFrameBufferTexture2D (for render-to-textures)
  - glFrameBufferRenderbuffer (for renderbuffer objects)

Resources

- http://www.opengl-tutorial.org/
- www.khronos.org, OpenGL Documentation