Intermediate Graphics & Animation Programming

GPR-300
Daniel S. Buckstein

Multi-Pass & Post-Processing Pipelines
Weeks 5 – 6

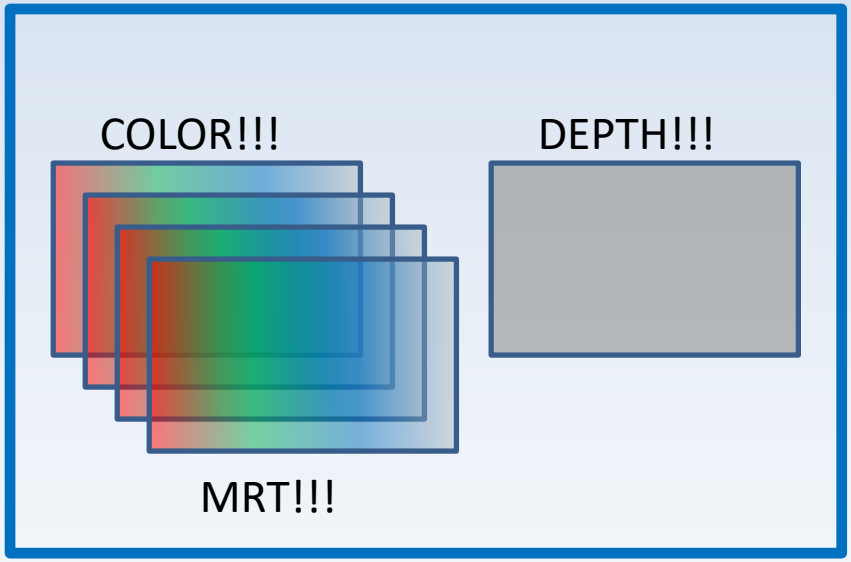
License

Multi-Pass & Post-Processing

- Review of framebuffers
- How framebuffer objects work
- Overview of pipelines and data flow
  - Key terms and helpful metaphors
- Pass diagrams, shader network diagrams

Frame Buffer Objects & Off-Screen Rendering

- “On-screen rendering”: targeting the default buffers



Default buffers: on-screen target
→OpenGL-managed
→minimal control over data

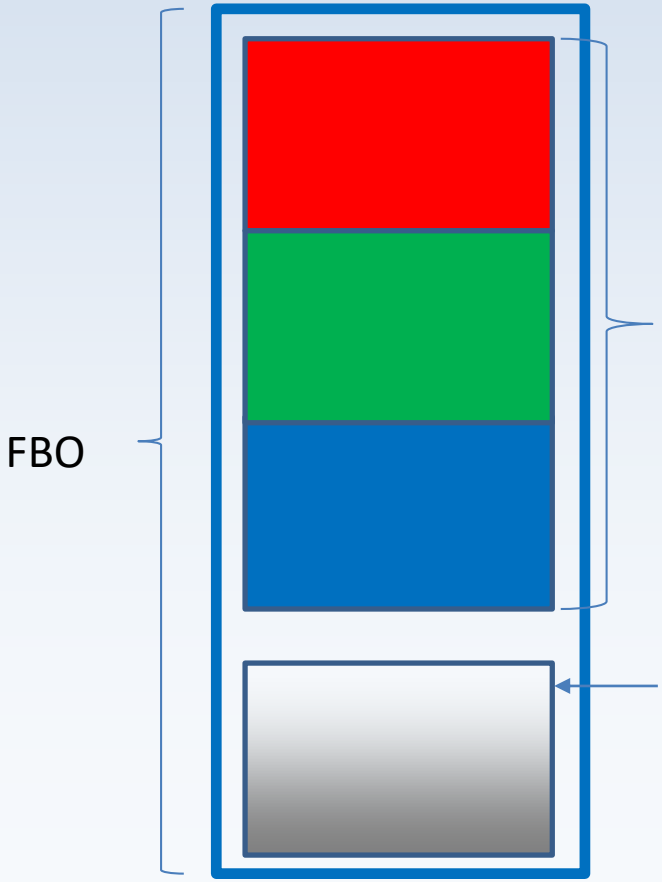
FBO: off-screen target(s)
→User-managed:
→fully accessible

Frame Buffer Objects & Off-Screen Rendering

- ***Multiple Render Targets*** (MRT):
- A single frame buffer has several components:
- Color, depth, stencil
- ***One*** target for depth OR depth/stencil combo
- … ***color*** can have *many targets*
- More than one color target: MRT
- I.e. fragments painted in more than one way

Frame Buffer Objects & Off-Screen Rendering

- Think of your FBO as a *layered set of images* all packaged together:



Render targets: *one or more textures* that store color (MRT if more than 1)

Depth buffer and/or stencil buffer: *one texture* that stores depth and/or stencil value (still a *texture*, just not a *target*)

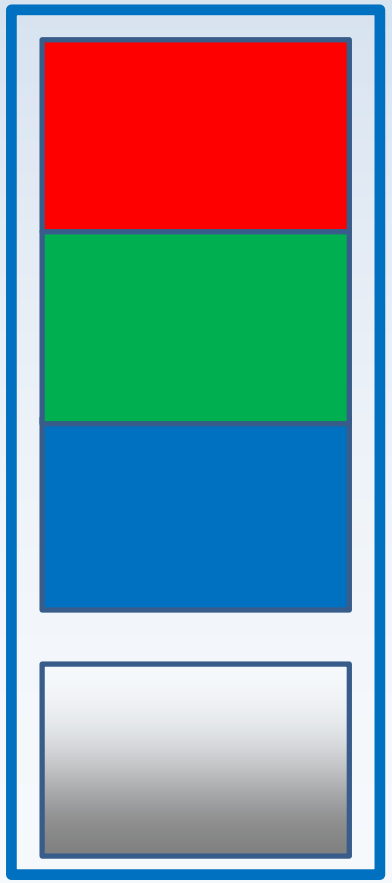
Frame Buffer Objects & Off-Screen Rendering

- How do framebuffers get *written to*?
- First, bind for writing (activate):

```
glBindFramebuffer(
    GL_FRAMEBUFFER, handle);
```

- All rendering is now off-screen until unbound or another FBO is bound:

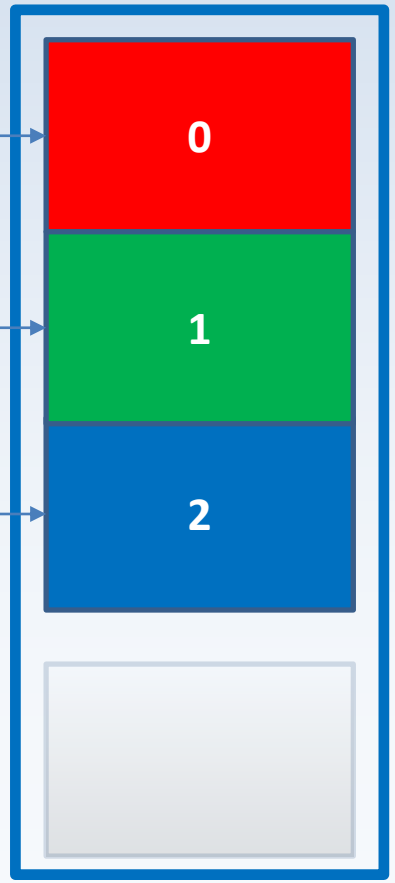
```
glBindFramebuffer(
    GL_FRAMEBUFFER, 0);
```



Frame Buffer Objects & Off-Screen Rendering

- How do framebuffers get their *color*?
- ***Fragment shader*** writes to whatever framebuffer is bound
- Render targets “directed” therein:

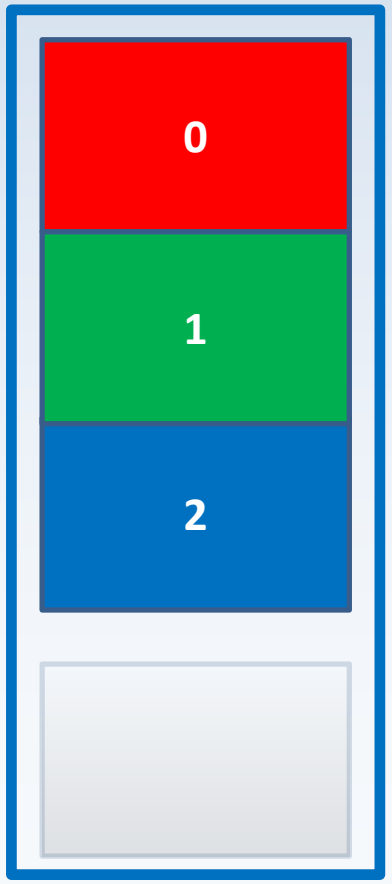
```
layout (location = 0) out vec4 rt0;
layout (location = 1) out vec4 rt1;
layout (location = 2) out vec4 rt2;
```



Frame Buffer Objects & Off-Screen Rendering

- How do framebuffers get their *color*?
- Example: FS that outputs R, G and B

```
layout (location = 0) out vec4 rtRED;
layout (location = 1) out vec4 rtGREEN;
layout (location = 2) out vec4 rtBLUE;
void main()
{
    rtRED   = vec4(1.0, 0.0, 0.0, 1.0);
    rtGREEN = vec4(0.0, 1.0, 0.0, 1.0);
    rtBLUE  = vec4(0.0, 0.0, 1.0, 1.0);
}
```

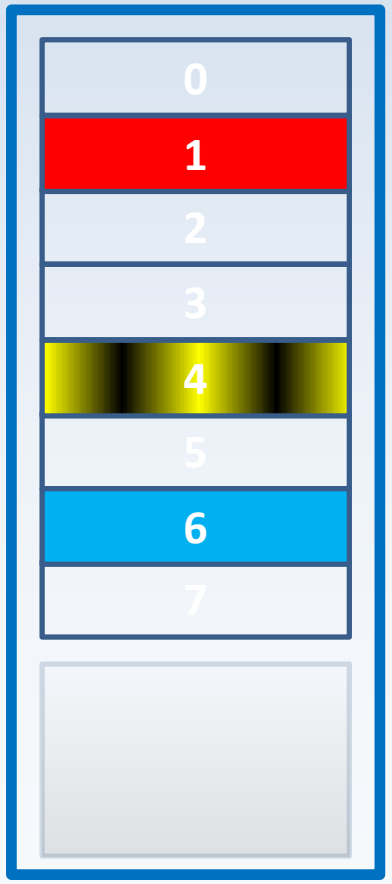


Frame Buffer Objects & Off-Screen Rendering

- How do framebuffers get their *color*?
- Render targets are *user-defined in FS*, in any order, or not used at all! E.g:

```
layout (location = 6) out vec4 rtCYAN;
layout (location = 1) out vec4 rtRED;
layout (location = 4) out vec4 rtOMGBEES;
```

- The only requirement is that your FBO supports the requested targets!



Frame Buffer Objects & Off-Screen Rendering

- How do framebuffers get their *depth value*?
- ***Raster position*** is passed to FS:

```
in vec4 gl_FragCoord;
```

- Depth value is also writeable in FS:

```
out float gl_FragDepth;
```

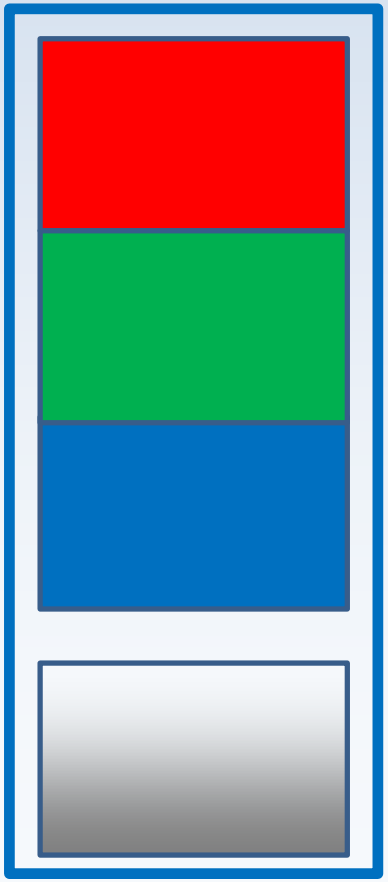
- Automatically set and tested after FS if user does not set it manually:

```
gl_FragDepth = gl_FragCoord.z;
```

- If depth test passes, fragment stored!

Frame Buffer Objects & Off-Screen Rendering

- Well, that explains how framebuffers are *written to*… how do we *read from* them?
- Remember: color targets and depth buffer are simply ***textures***
- Can be bound and sampled just like any other texture! So…
- Render offscreen → activate FBO
- Use results → bind FBO texture

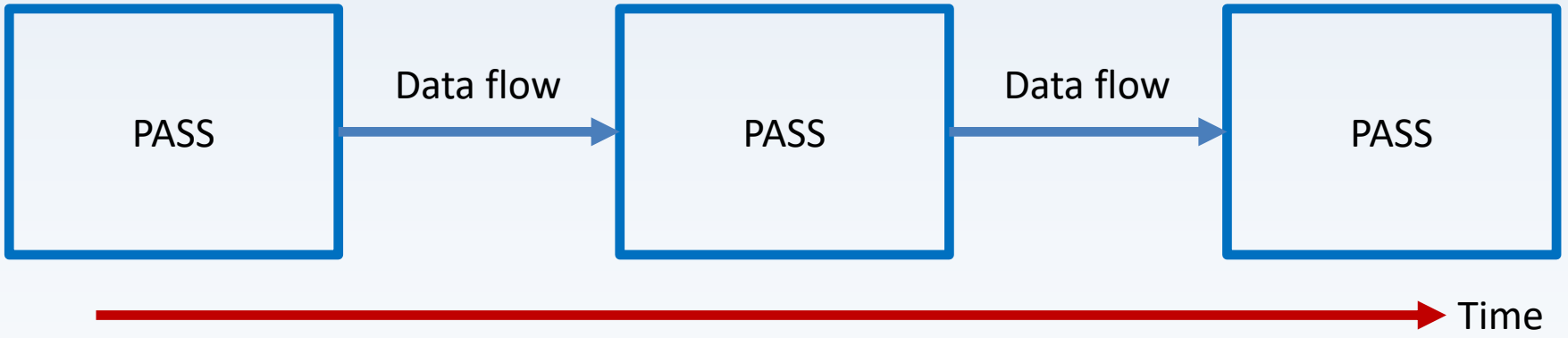


Multi-Pass & Post-Processing

- ***Rendering pipeline***: A set of rendering stages that result in an image on-screen
- ***Render pass***: A single stage in the *pipeline*
- ***Framebuffer***: Stores results of a *pass*
- So, FBOs are directly associated with passes?
- YES.

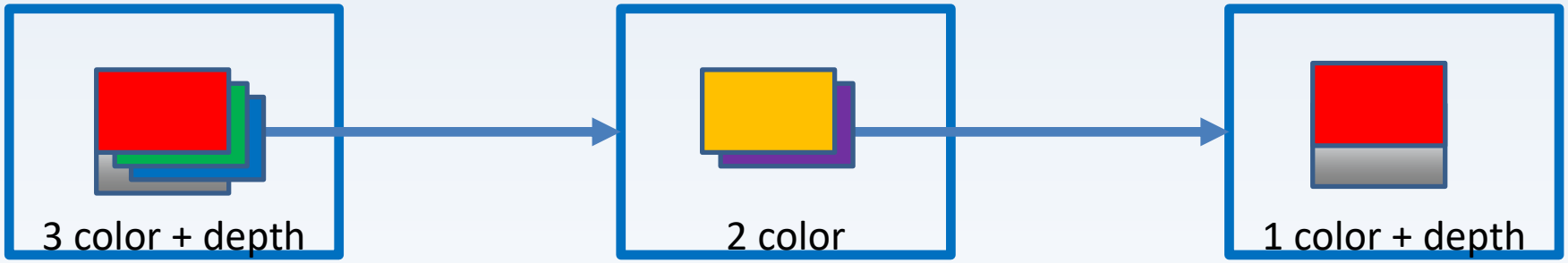
Multi-Pass & Post-Processing

- ***Rendering pipeline***: Think of it as an actual pipeline: data flows from pass to pass
- Changes happen over time
- Modelled ***horizontally*** like this:



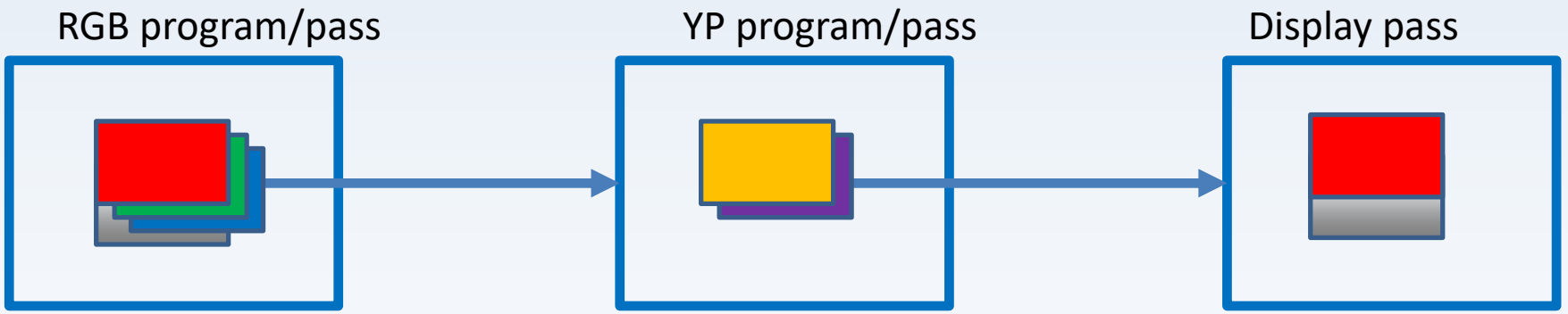
Multi-Pass & Post-Processing

- If each pass has a framebuffer associated with it, then think of the framebuffer as a ***vertical, layered slice*** of the pipeline
- Some may have more layers, some may only have one type of layer (color/depth)



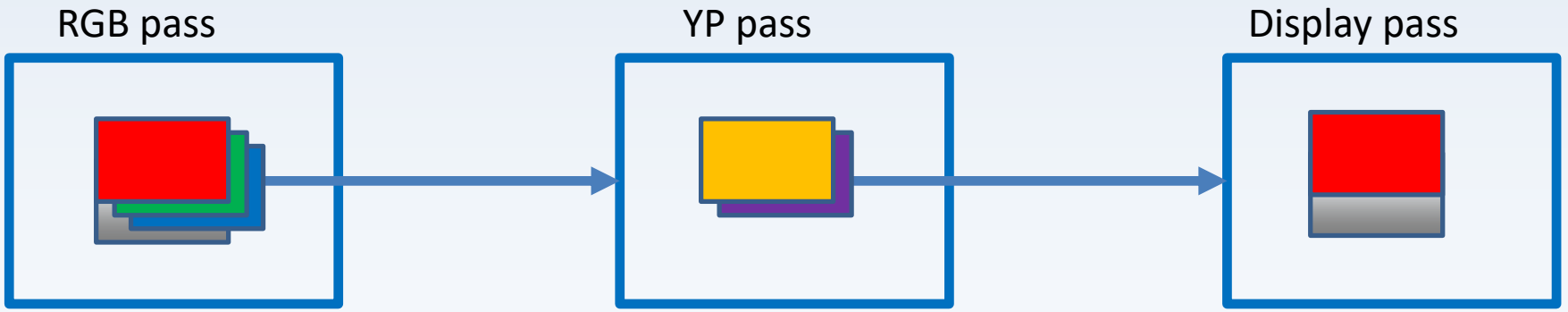
Multi-Pass & Post-Processing

- The actual “transfer” of data is handled by…
- ***THE SHADER PROGRAMS***
- They describe how to read and write the data!
- Shader name is also typically the pass name



Multi-Pass & Post-Processing

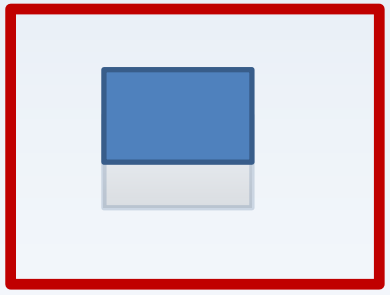
- These labelled pipeline diagrams are called ***shader network diagrams*** or ***pass diagrams***
- They are a visual representation of your custom graphics pipeline!



Multi-Pass & Post-Processing

- ***Shader network diagram example***:
- Ultra simple example: render scene directly to back buffer
- Single-pass with 1 color and depth

Scene pass (e.g. Phong shading… description of pass here)



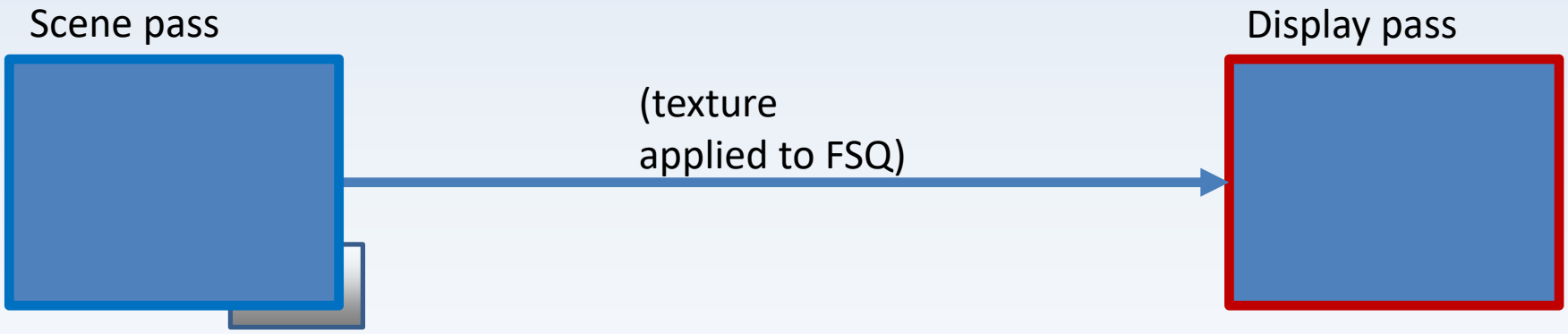
Red box: we are drawing to final destination: back buffer

Blue box: the color output, will be displayed

Translucent box: the inaccessible depth buffer

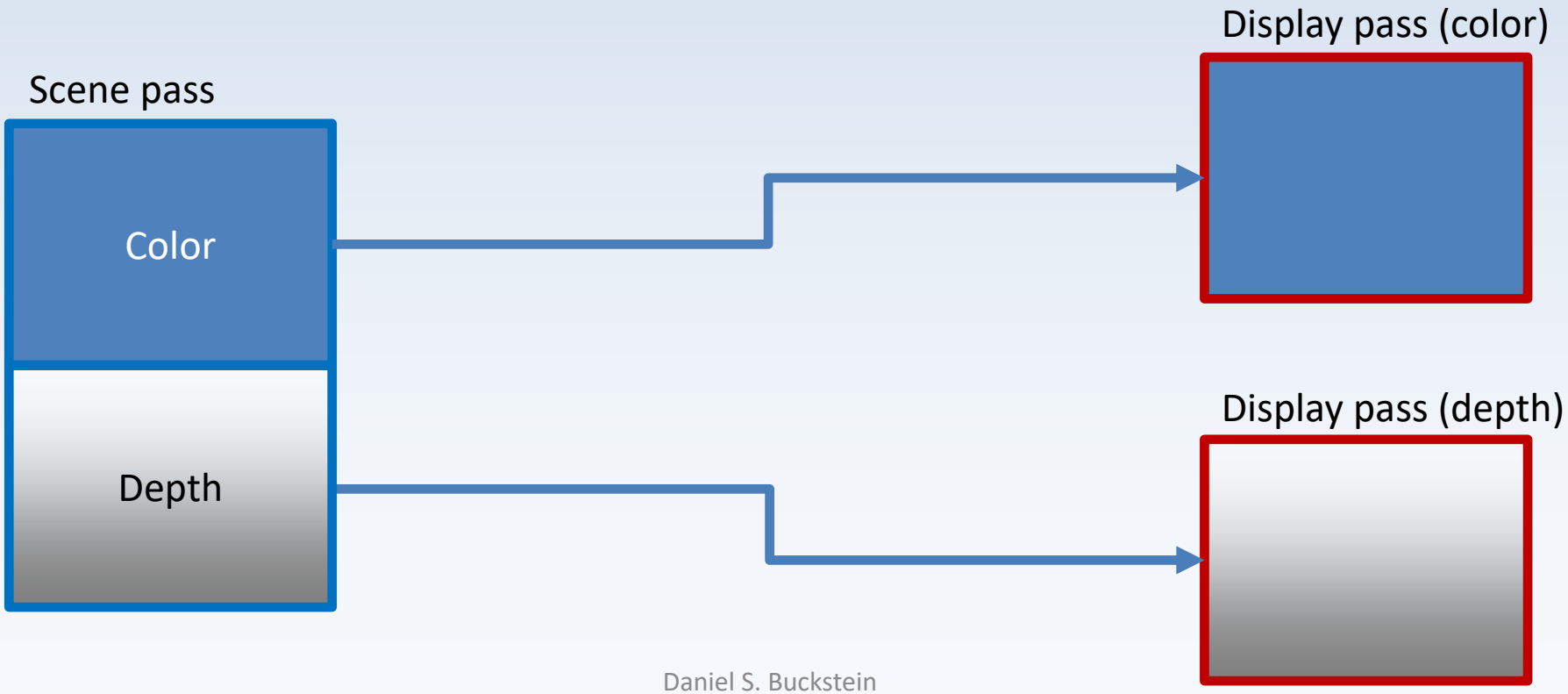
Multi-Pass & Post-Processing

- ***Shader network diagram example***:
- Simple example: render scene off-screen, display on back buffer using FSQ
- We can “hide” unused things



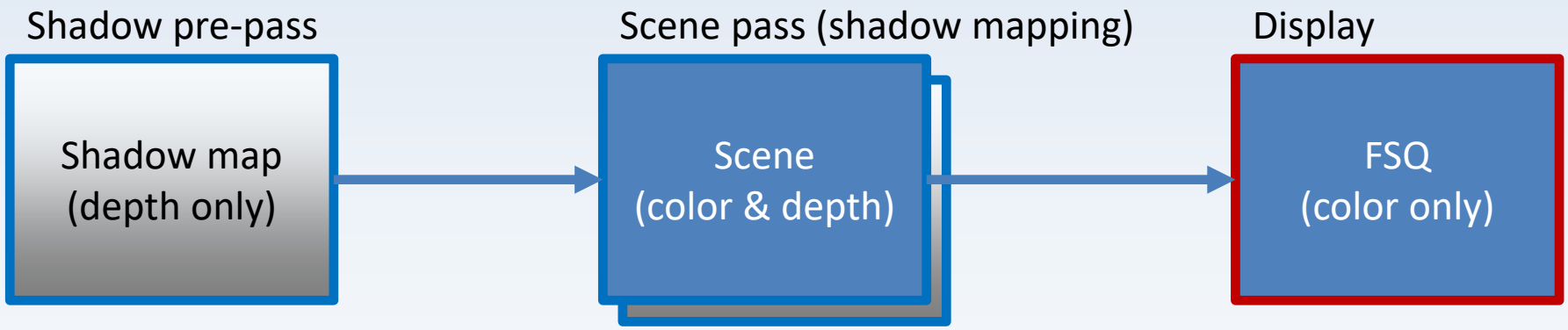
Multi-Pass & Post-Processing

- ***Shader network diagram example***:
- What if we have multiple display modes?



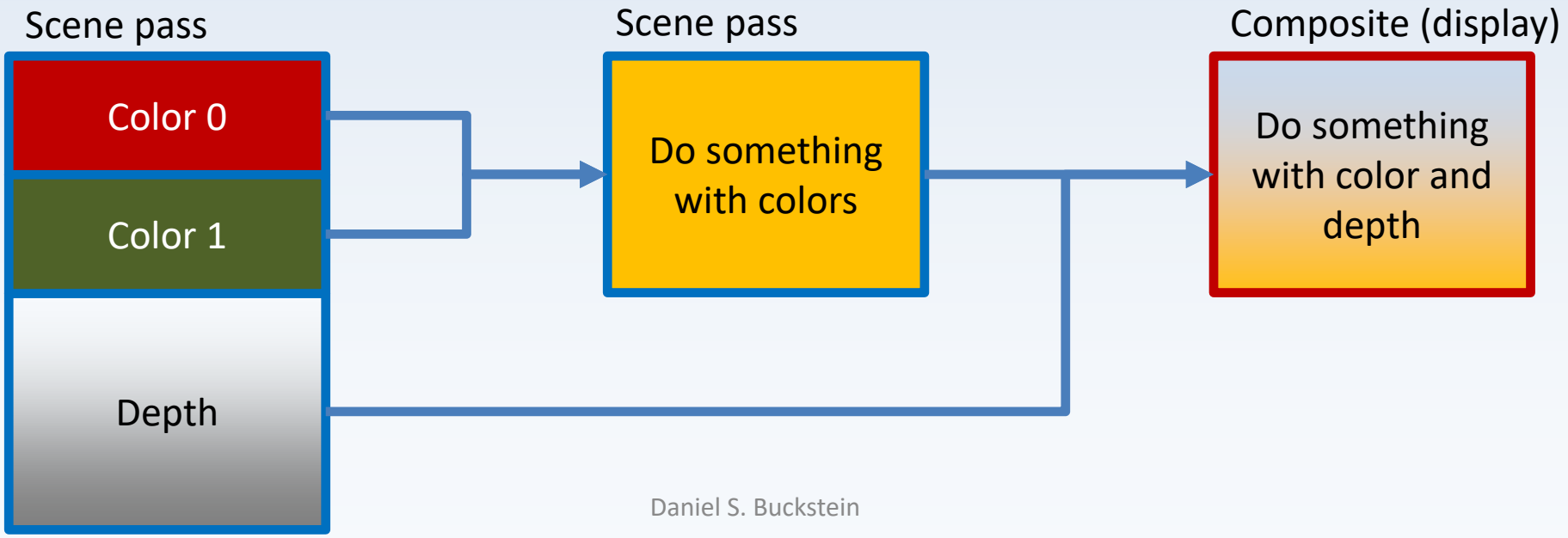
Multi-Pass & Post-Processing

- ***Shader network diagram example***:
- Here's one for shadow mapping with display pass added at the end:



Multi-Pass & Post-Processing

- ***Shader network diagram example***:
- Here's one for processing and compositing; the final image comes together as the FSQ is rendered to the back buffer…



Multi-Pass & Post-Processing

- Use pipeline and/or shader network diagrams to help you better understand the flow of data
- FBOs can be considered a ***vertical slice*** of a ***horizontal pipeline***
- Stages in pipeline, passes, are represented by an active framebuffer
- Framebuffers are great
- ☺

The end.

- Questions? Comments? Concerns?