Framebuffer Techniques

CS 385 - Class 24
21 April 2022

Framebuffers

Remember a long time ago?

- The terminus of the graphics pipeline is a *framebuffer*

- Unless you do something special, you'll use the *default framebuffer*
  - it's the one that can be displayed on the screen
- But it's not the only framebuffer available, and
- It's not the only *surface*† that you can render into
  - In WebGL, you can draw into *renderbuffers*, and *textures*

† That's the modern term for a place filled with pixels you can draw into

Types of Surfaces

- There are three types of framebuffers used in computer graphics:

Buffer Type	Data Type	Possible Data Values
color	vec4 (vector of four floating-point values)	colors, normals, texture coordinates, etc.
depth	single floating-point value (in a special range)	depth values
stencil	integer value	stencil values

- You can create a renderbuffers or textures in these formats and use them for whatever purpose you might need

Framebuffer Objects (FBOs)

- WebGL collects the current rendering targets in a *framebuffer object*
- FBOs have *attachments*, which are the places where you attach a surface to the FBO

Attachment Name	Surface Type
`gl.COLOR_ATTACHMENTn`	Color (vec4) buffer
`gl.DEPTH_ATTACHMENT`	Depth buffer
`gl.STENCIL_ATTACHMENT`	Stencil buffer

Aside: Getting Pixels out of a Framebuffer

- You can save the pixel values from any framebuffer

```
gl.ReadPixels( x, y, width, height, format, type, pixels );
```

- `(x,y)`, and `(width,height)` define the rectangle of pixels to read
- `format` specifies the type of pixels you want to read: `gl.ALPHA`, `gl.RGB`, or `gl.RGBA`
- `type` specifies the datatype for storing the retrieved pixel values: `gl.UNSIGNED_BYTE`, `gl.FLOAT`, etc.
- `pixels` is a typed array (e.g., `UInt8Array` — array of unsigned bytes) of the appropriate size
  - `width` x `height` x numPixels(`format`)

Reading Pixels

- Call `gl.readPixels` at the very end of your `render` function
- Write stored pixels to an image file (e.g., JPEG)

JavaScript

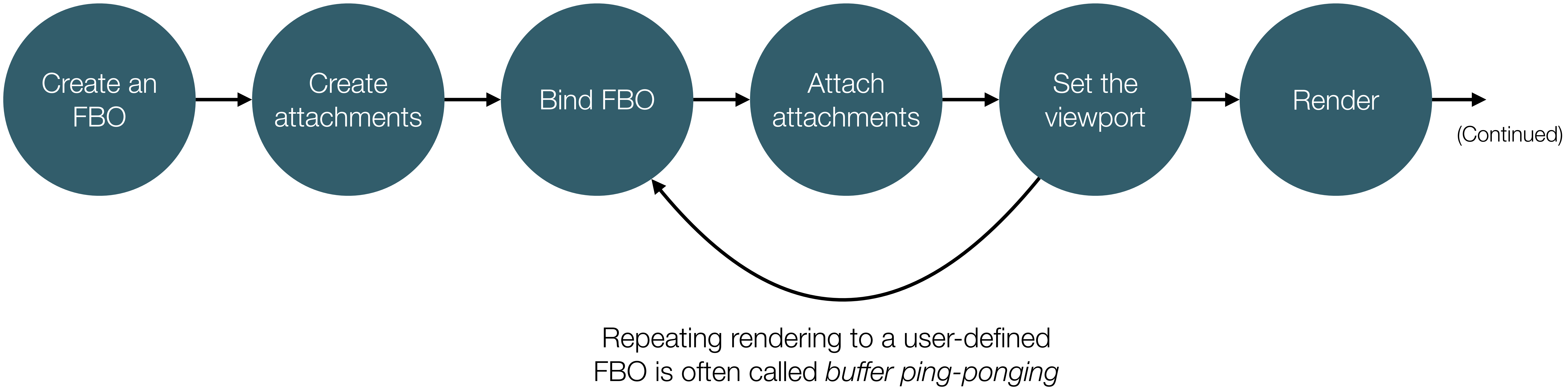
```
function render() {
   // render your scene
  sphere.render();
  ...

  var w = gl.drawingBufferWidth;
  var h = gl.drawingBufferHeight;

  // allocate array to store pixel values
  var nPixels = w * h * 4; // RGBA ubyte
  var pixels = new Uint8Array(nPixels);

  // read the framebuffer
  gl.readPixels(0, 0, w, h, gl.RGBA,
    gl.UNSIGNED_BYTE, pixels);
};
```

Why use FBOs?

- FBOs provide several capabilities:
  - *offscreen rendering* – use WebGL to create images without displaying them
    - generating images on demand on a web server
    - image processing
  - *generating temporary textures* – generate/update a texture map that you sample later in the same frame
    - update a texture
      - add "damage" to the surface texture of a vehicle in a war game
    - synthesize a texture to be applied to other objects later in the scene
  - *deferred techniques* – generate various results into buffers to be accumulated later
    - reflections
    - deferred lighting

Using FBOs in Applications

Steps for Using FBOs in WebGL Applications

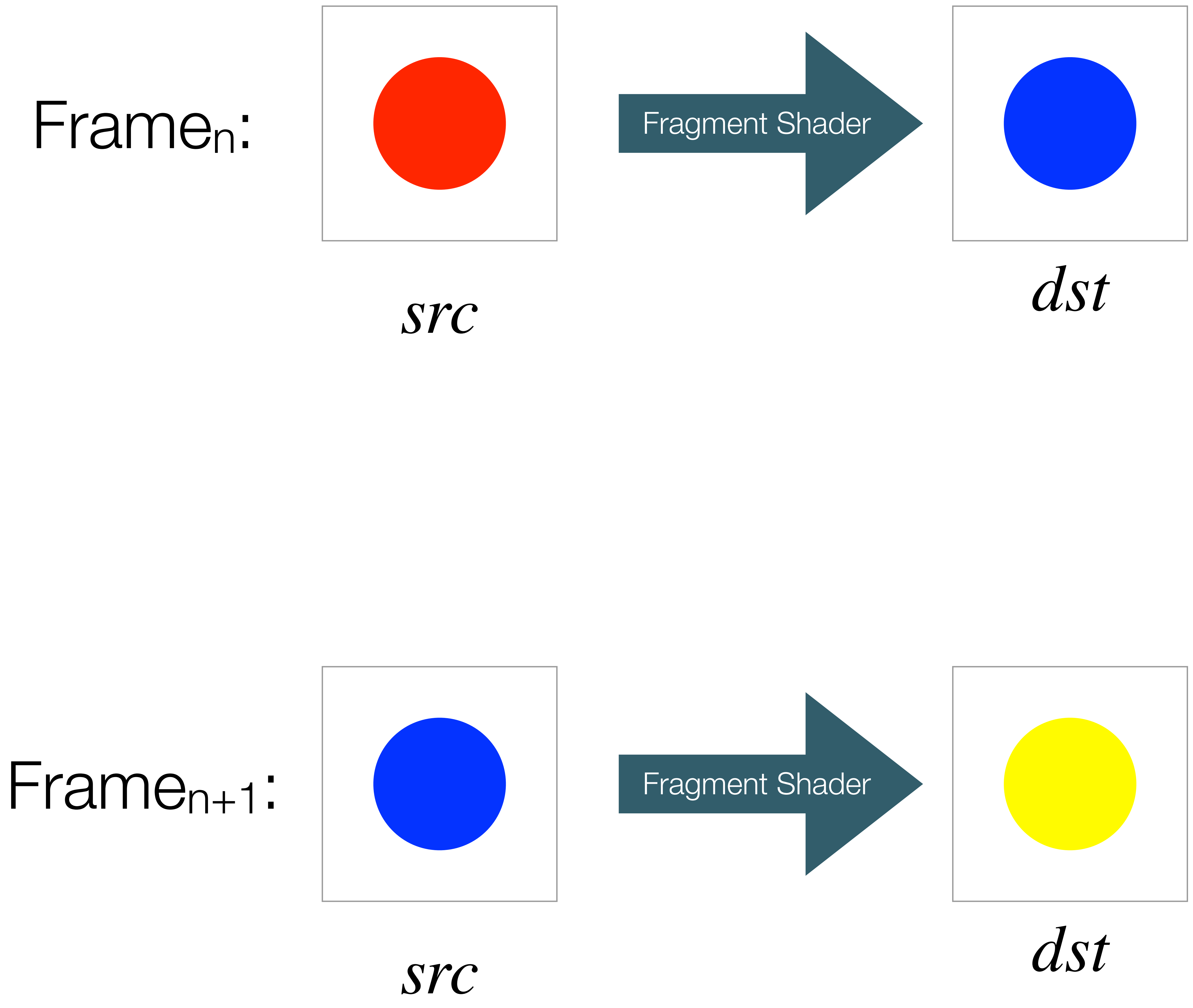


Framebuffer Ping-Ponging

- Standard method for doing *incremental* updates to an image
- How do I know when to use one?
  - if frame_{n+1} relies on *state* from frame_n
  - Mathematically, think of

$$x_{n+1} = f(x_n)$$

- For the next frame (frame_{n+1}), we read the texture written in frame_n

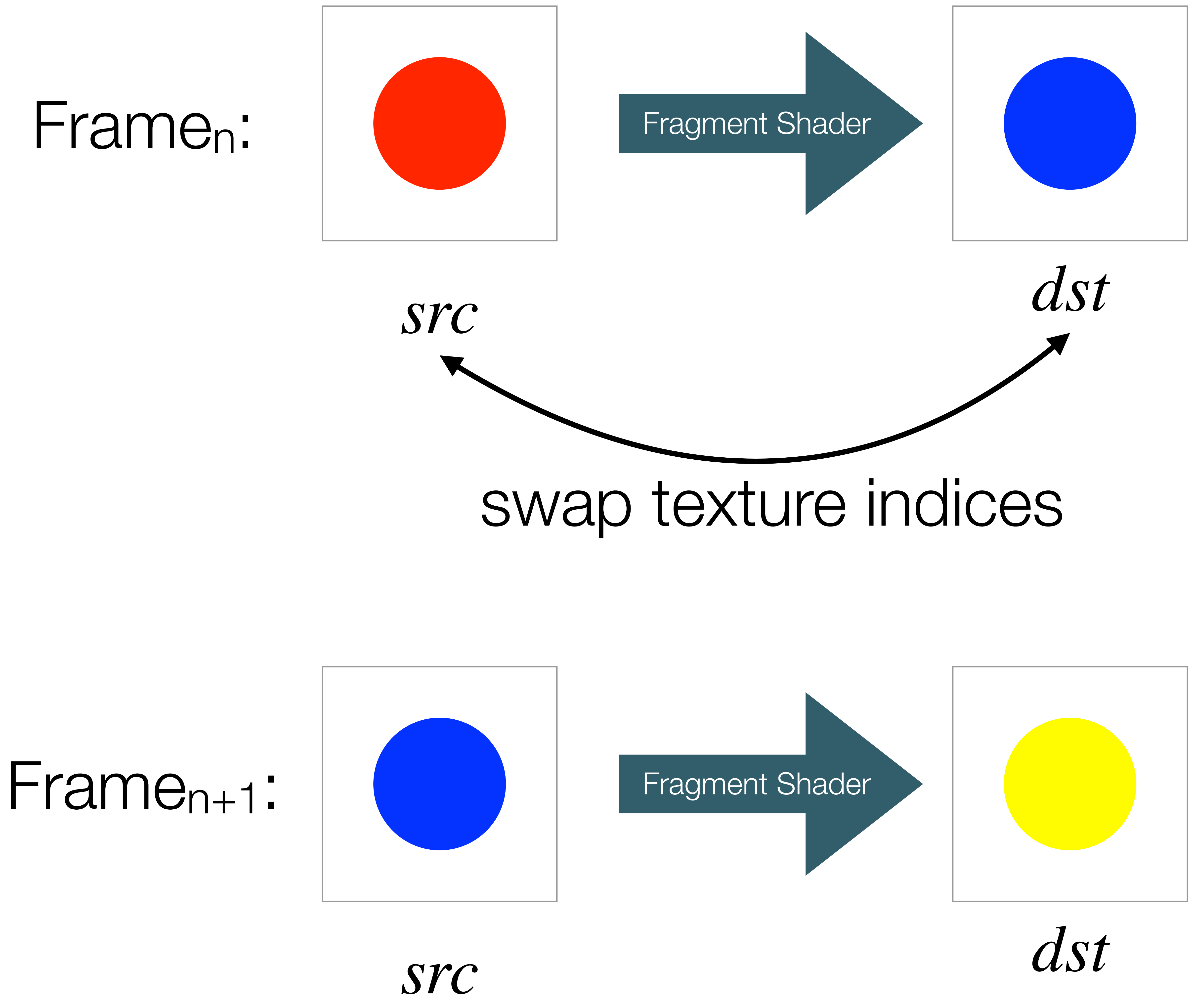


Framebuffer Ping-Ponging

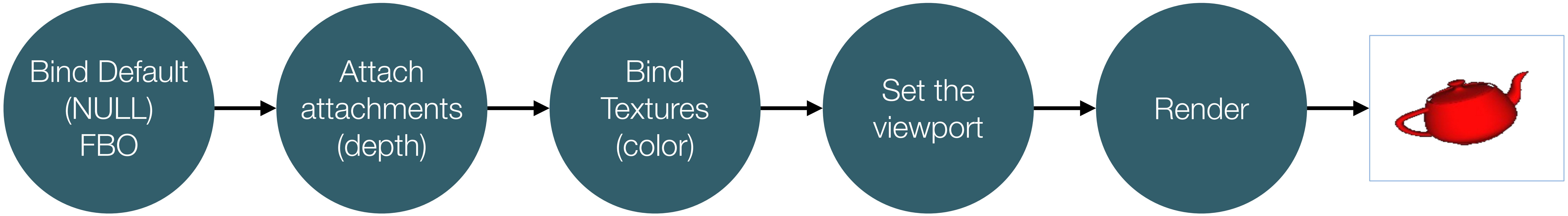
- Standard method for doing *incremental* updates to an image
- How do I know when to use one?
  - if frame$_{n+1}$ relies on *state* from frame$_n$
  - Mathematically, think of

$$x_{n+1} = f(x_n)$$

- For the next frame (frame$_{n+1}$), we read the texture written in frame$_n$
- At some point in rendering, exchange the two textures



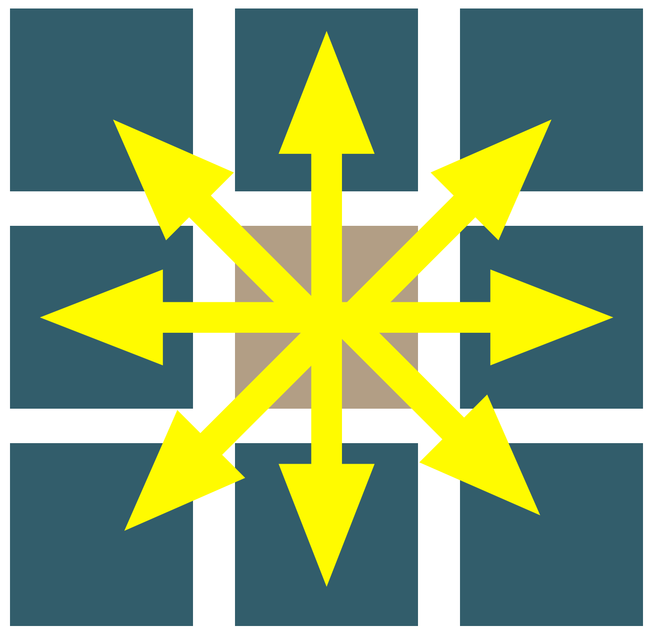
Rendering from an FBO for Display



Example: Game of Life

The Game of Life

- A cellular automaton example developed by John Horton Conway
- Each *cell*'s lifetime is determined by its neighbor's state



Neighbor Condition	Current Cell Condition	Cell Outcome
less than 2	—	cell dies
exactly 2 or 3	alive	cell lives
greater than 3	alive	cell dies
exactly 3	dead	cell reborn

- State of frame$_{n+1}$ relies on the state of frame$_n$
  - perfect use of buffer *ping-ponging*

Storing state in a Texture

- For the Game of Life, we'll store a color in each pixel of a texture
  - white — alive; black — deceased
- We'll do the life evaluation in the fragment shader
  - just texture samples and a bit of logic
  - write our new life state to a texture bound to an FBO
- Ping-pong buffers to the next frame

Framebuffer Object Setup

1. Create our framebuffer object (FBO)
2. Create a pair of textures that we'll ping-pong between
   - we specify `null` for the texels — WebGL will create an empty image
   - since we only want to sample individual texels, we use `gl.NEAREST` filtering
   - and we set the wrap modes to `gl.REPEAT` so when we go off one edge of the texture, we wrap around
     - we're simulating life, so pretend it's a planet; there aren't any edge to fall off
3. Bind the FBO
4. Attach attachments
5. Set the rendering viewport
6. Initialize the system's state by rendering into the texture attached to our FBO

JavaScript

```
var fbo = null;
var textures = [];

var src = 0;  //indices for ping-ponging buffers
var dst = 1;
var NumTextures = 2;

function init() {

 ❶  fbo = gl.createFramebuffer();

 ❷  for ( var i = 0; i < NumTextures; ++i ) {
        textures[i] = gl.createTexture();
        gl.bindTexture( gl.TEXTURE_2D, textures[i] );
        gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB, texWidth, texHeight, 0,
            gl.RGB, gl.UNSIGNED_BYTE, null );
        gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST );
        gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST );
        gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT );
        gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT );
    }

 ❸  gl.bindFramebuffer( gl.FRAMEBUFFER, fbo );

 ❹  gl.framebufferTexture2D( gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
        gl.TEXTURE_2D, textures[src], 0 );

 ❺  gl.viewport( 0, 0, texWidth, texHeight);

 ❻  gl.clearColor( 0.0, 0.0, 0.0, 1.0);
    gl.clear( gl.COLOR_BUFFER_BIT );
    initialState.render();
}
```

Rendering

1. Update our ping-pong indices
   - this is basically a *ring buffer* where the indices loop around
   - since we only have two buffers, it merely swap indices
2. Bind the FBO, attaching the destination texture to receive the updated values
   - note we update the viewport to match the size of the texture
3. Bind the source texture, which contains the state for the current frame
4. Update the system by rendering a full-screen (viewport) quad with a fancy fragment shader

JavaScript

```
function render() {

❶  src = (src + 1) % NumTextures;
   dst = (src + 1) % NumTextures;

❷  gl.bindFramebuffer( gl.FRAMEBUFFER, fbo );
   gl.framebufferTexture2D( gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
       gl.TEXTURE_2D, textures[dst], 0 );
   gl.viewport( 0, 0, texWidth, texHeight );

❸  gl.bindTexture( gl.TEXTURE_2D, textures[src] );

❹  step.render();

❺  gl.bindFramebuffer( gl.FRAMEBUFFER, null );
   gl.viewport( 0, 0, w, h );

❻  gl.bindTexture( gl.TEXTURE_2D, textures[dst] );

❼  display.render();

   window.setTimeout( render, 100 );
}
```

Rendering (cont'd)

5. Bind to the *default framebuffer* (the one that can be displayed)
    - again, we update the viewport, this time matching the canvas' size
6. Bind the updated texture which we'll render to the viewport
7. Render another full-screen (viewport) quad, which merely copies samples the texture and copies the value to the corresponding pixel

JavaScript

```
function render() {

❶  src = (src + 1) % NumTextures;
   dst = (src + 1) % NumTextures;

❷  gl.bindFramebuffer( gl.FRAMEBUFFER, fbo );
   gl.framebufferTexture2D( gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
       gl.TEXTURE_2D, textures[dst], 0 );
   gl.viewport( 0, 0, texWidth, texHeight );

❸  gl.bindTexture( gl.TEXTURE_2D, textures[src] );

❹  step.render();

❺  gl.bindFramebuffer( gl.FRAMEBUFFER, null );
   gl.viewport( 0, 0, w, h );

❻  gl.bindTexture( gl.TEXTURE_2D, textures[dst] );

❼  display.render();

   window.setTimeout( render, 100 );
}
```

Game of Life
Fragment Shader

1. We use a special GLSL variable that returns which fragment we are in the viewport: `gl_FragCoord`
   - `.xy` returns the fragment location
   - `.z` returns the depth value
   - dividing this value by the viewport will give us values in the range $[0,\ 1]$
     - just what we need for texture coordinates
2. Collect the state of the cell, and its neighbors
   - we do that with a lot of texture samples
   - we use the fragment coordinate and adjust up/down/left/right one fragment
   - Note that we only look at the red component

Fragment Shader

```
uniform sampler2D texture;
uniform vec2 viewportSize;

out vec4 fColor;

void main() {
 ❶  vec2 c = gl_FragCoord.xy;

 ❷ float me = texture( texture, (c + vec2(  0,  0 )) / viewportSize ).r;
   float up = texture( texture, (c + vec2(  0,  1 )) / viewportSize ).r;
   float ul = texture( texture, (c + vec2( -1,  1 )) / viewportSize ).r;
   float lf = texture( texture, (c + vec2( -1,  0 )) / viewportSize ).r;
   float ll = texture( texture, (c + vec2( -1, -1 )) / viewportSize ).r;
   float dn = texture( texture, (c + vec2(  0, -1 )) / viewportSize ).r;
   float lr = texture( texture, (c + vec2(  1, -1 )) / viewportSize ).r;
   float rt = texture( texture, (c + vec2(  1,  0 )) / viewportSize ).r;
   float ur = texture( texture, (c + vec2(  1,  1 )) / viewportSize ).r;

 ❸  int count = 0;
   count += int( up > 0.0 );
   count += int( ul > 0.0 );
   count += int( lf > 0.0 );
   count += int( ll > 0.0 );
   count += int( dn > 0.0 );
   count += int( lr > 0.0 );
   count += int( rt > 0.0 );
   count += int( ur > 0.0 );

 ❹  bool cellAlive = bool( me > 0.0 );
   bool live = (cellAlive && count == 2) || count == 3;

 ❺  fColor = float(live) * vec4( 1.0, 1.0, 1.0, 1.0 );
}
```

Game of Life
Fragment Shader (cont'd)

3. Sum up the number of "live" cells
4. Determine our state based on:
   - whether the cell is alive
   - state of neighbors

Recall the rules:

- if 2 or 3 neighbors are alive with my being alive
- or, exactly 3 neighbors alive while I'm dead

I keep (or return to the) living ...

5. Set the updated state of the cell

Fragment Shader

```
uniform sampler2D texture;
uniform vec2 viewportSize;

out vec4 fColor;

void main() {
❶   vec2 c = gl_FragCoord.xy;

❷  float me = texture( texture, (c + vec2(  0,  0 )) / viewportSize ).r;
    float up = texture( texture, (c + vec2(  0,  1 )) / viewportSize ).r;
    float ul = texture( texture, (c + vec2( -1,  1 )) / viewportSize ).r;
    float lf = texture( texture, (c + vec2( -1,  0 )) / viewportSize ).r;
    float ll = texture( texture, (c + vec2( -1, -1 )) / viewportSize ).r;
    float dn = texture( texture, (c + vec2(  0, -1 )) / viewportSize ).r;
    float lr = texture( texture, (c + vec2(  1, -1 )) / viewportSize ).r;
    float rt = texture( texture, (c + vec2(  1,  0 )) / viewportSize ).r;
    float ur = texture( texture, (c + vec2(  1,  1 )) / viewportSize ).r;

❸   int count = 0;
    count += int( up > 0.0 );
    count += int( ul > 0.0 );
    count += int( lf > 0.0 );
    count += int( ll > 0.0 );
    count += int( dn > 0.0 );
    count += int( lr > 0.0 );
    count += int( rt > 0.0 );
    count += int( ur > 0.0 );

❹   bool cellAlive = bool( me > 0.0 );
    bool live = (cellAlive && count == 2) || count == 3;

❺   fColor = float(live) * vec4( 1.0, 1.0, 1.0, 1.0 );
}
```