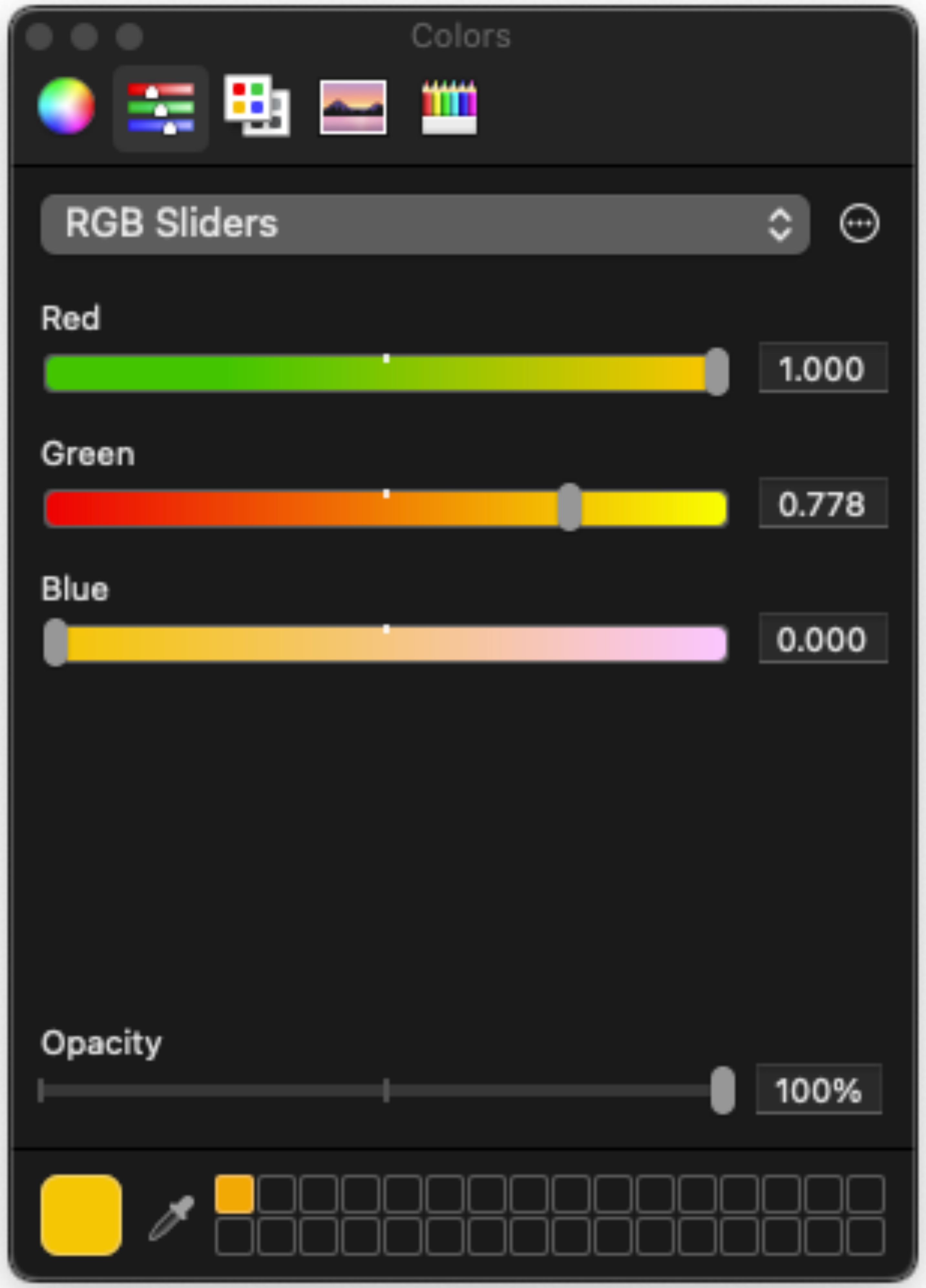
Color & Blending

CS 385 - Class 15
15 March 2022

Color Spaces (Revisited)

Representing Color

- The colors we see are the light *reflected* from a surface
- We use an *additive* color model
  - create colors by adding the *primary colors:* red, green, and blue
  - Those match our biology
    - rods, cones, and all that ...
- Colors in the graphics pipeline are represented as floating-point values in the range [0, 1]
  - *high-dynamic range (HDR)* allows values outside of that range
- Colors are *quantized* for storage in frame buffers
  - discussion in Class 2



Representing Color

- There are many other color spaces to represent colors
  - HSV (HSB): hue, saturation, value (brightness)
    - *hue* is the angle around the color wheel
    - *saturation* is the distance from the center
    - *value* is how much color
  - CMYK : cyan, magenta, yellow, black
    - mostly used for printing
    - colors *absorb* wavelengths of color, not reflect them

Colors in WebGL

- Colors in WebGL have four values:
  - RGBA: red, green, blue, and alpha
  - represented as a `vec4` with four floating-point values

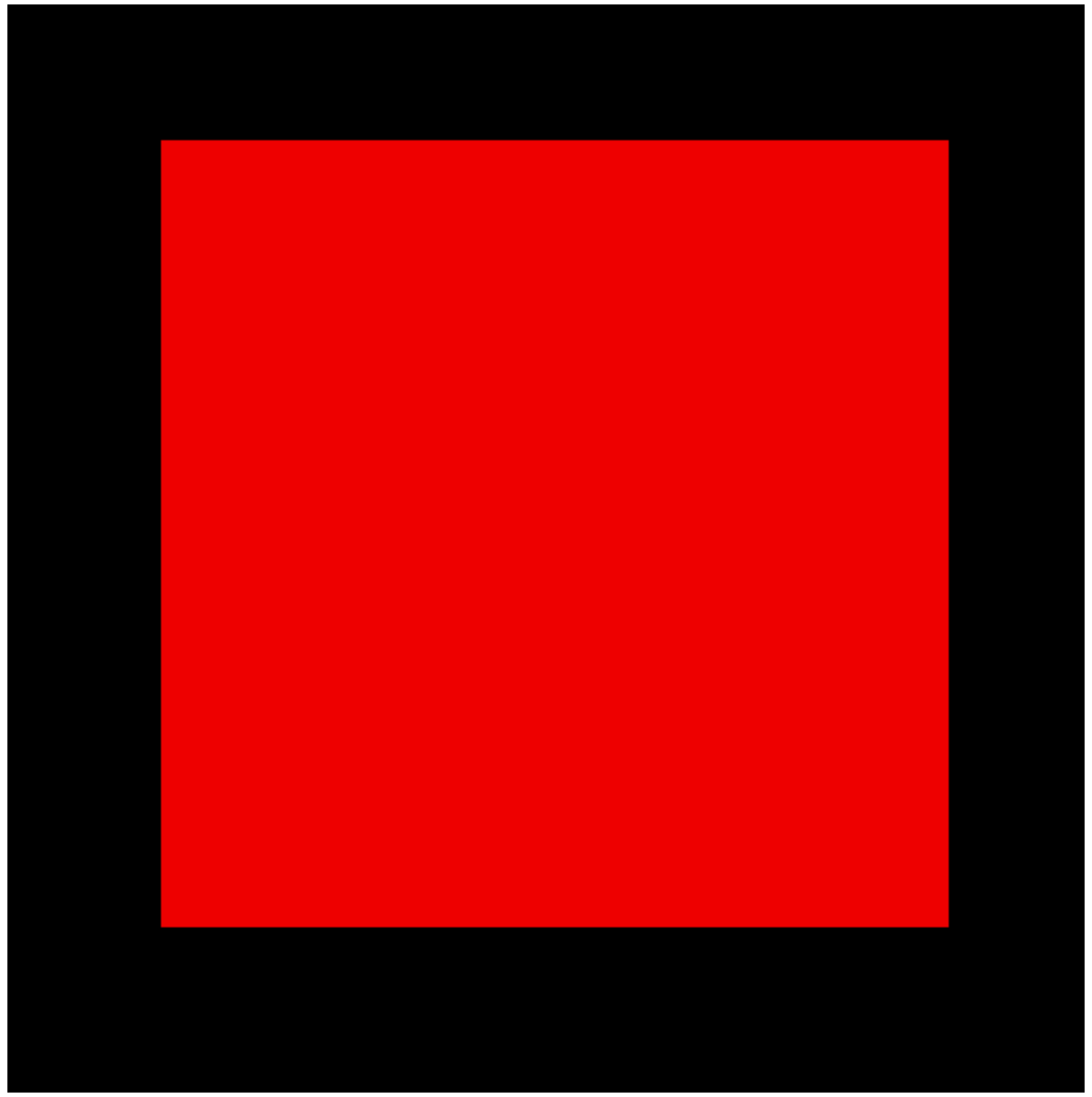
Fragment Shader

```
out vec4 fColor;

void main()
{
    fColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Shading Models

Flat Shading (hard-coded color)

- Constant color across the entire primitive



Fragment Shader

```
out vec4 fColor;

void main()
{
    fColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Flat Shading (`uniform` color)

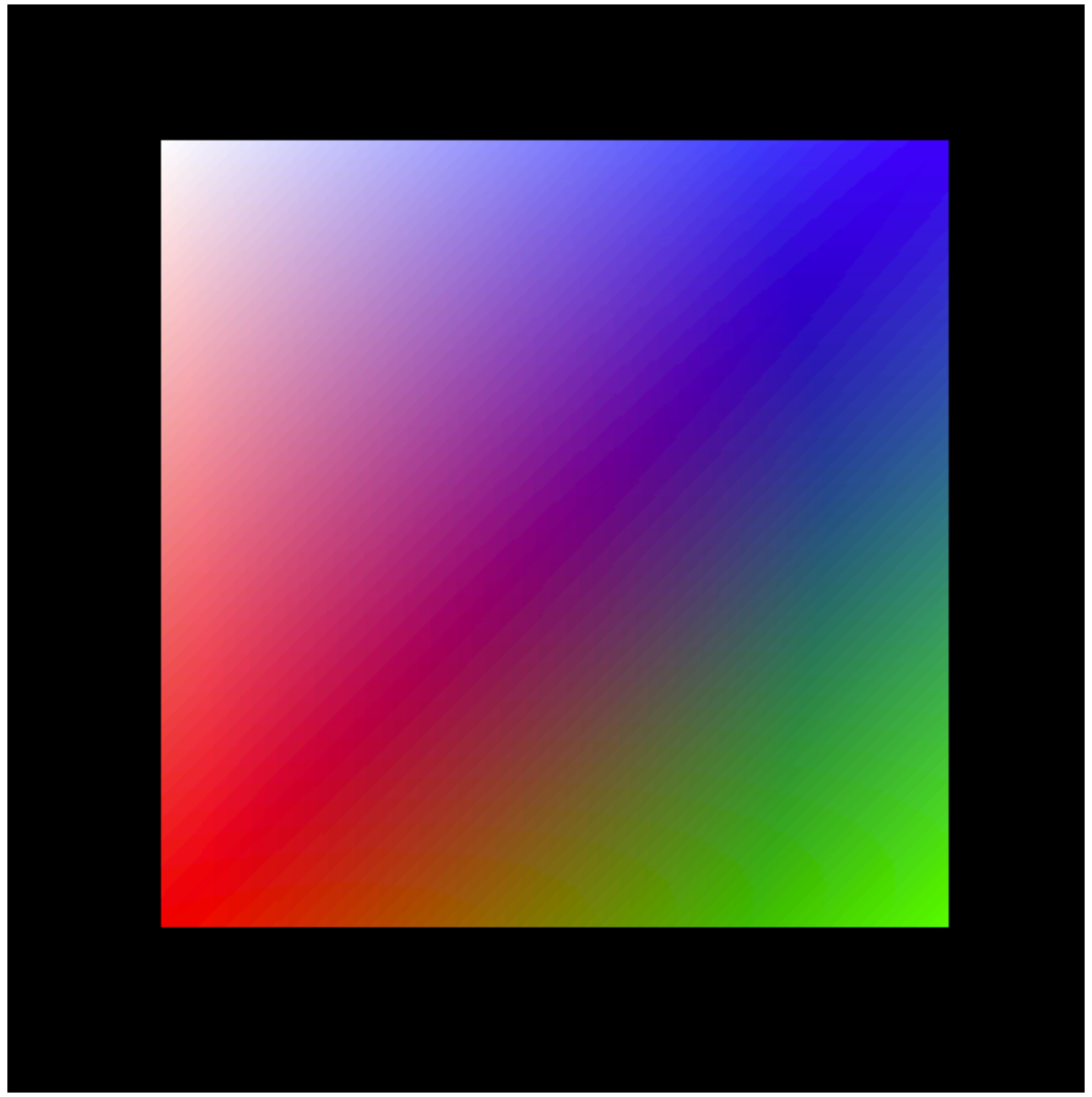
- Again, constant color across the entire primitive
- Application can set the color through a `uniform` variable
  - this is how the Sphere object in assignment 4 works

Fragment Shader

```
out vec4 fColor;

uniform vec4 color;

void main()
{
    fColor = color;
}
```

Gouraud Shading

- Colors interpolated across the primitive
  - color gradient by using different colors
- Color specified as an *attribute* for each vertex



Fragment Shader

```
// vertex color - interpolated by the rasterizer
in  vec4 vColor;
out vec4 fColor;

void main()
{
    fColor = vColor;
}
```

Gouraud Shading

- Application specifies a color per vertex
- Copy input application color to vertex color
  - tells the rasterizer to *interpolate* the color

Vertex Shader

```
in  vec4  aPosition;
in  vec4  aColor;  // application-provided color
out vec4  vColor;  // vertex color for rasterizer

uniform mat4 P;
uniform mat4 MV;

void main()
{
    vColor = aColor;
    gl_Position = P * MV * aPosition;
}
```

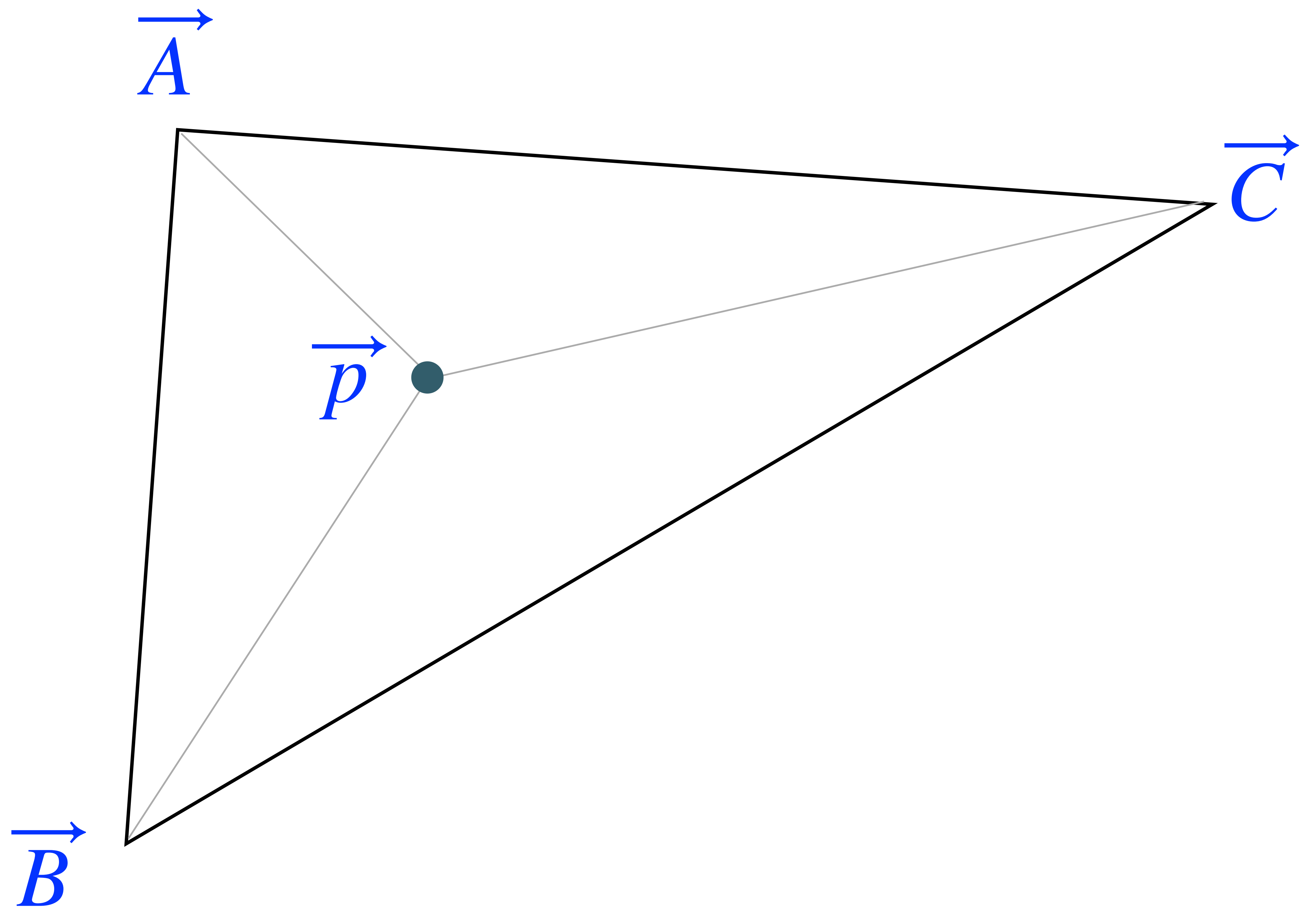
Color Interpolation

Barycentric Coordinates

- An area-based interpolation scheme
- Any point inside of the triangle can be described using a combination of the vertex positions (e.g., $\vec{A}$)

$$\vec{p} = a\vec{A} + b\vec{B} + c\vec{C}$$

- For each coefficient, compute the ratio of two triangles
  - for example, for the a coefficient, compute
    - $\Delta\ pBC$ the area of the triangle bounded by $\vec{p}$, $\vec{B}$, and $\vec{C}$
    - similarly, compute the area for $\Delta\ ABC$, which is the area of the entire triangle



$$a = \frac{\Delta\ pBC}{\Delta\ ABC}$$

Specifying Multiple Vertex Attributes

Multiple Vertex Buffer Approach

Vertex Colors

- An *attribute* of a vertex
  - just like position
- Create another array of values
  - specifying the correct number of components

JavaScript

```
function Square() {
  // Normal initialization / shader program
  positions = [
    0.0, 0.0,   // Vertex 0
    1.0, 0.0,   // Vertex 1
    1.0, 1.0,   // Vertex 2
    0.0, 1.0    // Vertex 3
  ];
  positions.numComponents = 2;  // (x, y)

  colors = [
    1.0, 0.0, 0.0, // Vertex 0
    0.0, 1.0, 0.0, // Vertex 1
    0.0, 0.0, 1.0  // Vertex 2
    1.0, 1.0, 1.0, // Vertex 3
  ];
  colors.numComponents = 3;  // RGB

  // continued
```

Vertex Colors (cont'd)

- Repeat steps just like for positions

JavaScript

```
function Square() {
  ...
  positions.buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, positions.buffer);
  gl.bufferData(gl.ARRAY_BUFFER,
    new Float32Array(positions), gl.STATIC_DRAW);

  colors.buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, colors.buffer);
  gl.bufferData(gl.ARRAY_BUFFER,
    new Float32Array(colors), gl.STATIC_DRAW);
```

Vertex Colors (cont'd)

- Find shader variable, just like for positions

JavaScript

```
function Square() {
   ...
   aPosition = gl.getAttribLocation(program, "aPosition");
  gl.enableVertexAttribArray(aPosition);

  aColor = gl.getAttribLocation(program, "aColor");
  gl.enableVertexAttribArray(aColor);
```

Vertex Colors (cont'd)

- Bind buffer and specify vertex attribute parameters, just like for position

JavaScript

```
function Square() {

  this.render = function () {
    gl.useProgram(program);

    gl.bindBuffer(gl.ARRAY_BUFFER, positions.buffer);
    gl.vertexAttribPointer(aPosition,
      positions.numComponents, gl.FLOAT,
      false, 0, 0);

    gl.bindBuffer(gl.ARRAY_BUFFER, colors.buffer);
    gl.vertexAttribPointer(aColor,
      colors.numComponents, gl.FLOAT,
      false, 0, 0);

    gl.drawArrays(...);
  };
```

Specifying Multiple Vertex Attributes

Single Vertex Buffer Approach

Combined Vertex Attributes

- All of the attributes for vertices can be stored in a single buffer
  - this is kind of a performance hack

JavaScript

```
function Square() {
  // Normal initialization / shader program
  let attributes = [
    0.0, 0.0, 1.0, 0.0, 0.0, // x, y, R, G, B
    1.0, 0.0, 0.0, 1.0, 0.0,
    1.0, 1.0, 0.0, 0.0, 1.0,
    0.0, 1.0, 1.0, 1.0, 1.0
  ];
```

Combined Vertex Attributes

- Set up some hints to decode attribute data
  - need to know:
    - number of attribute components
    - starting offset between successive vertex attributes (in bytes)
    - size of all attributes for a vertex (in bytes)

JavaScript

```
function Square() {
  // Normal initialization / shader program
  attributes = [
    0.0, 0.0, 1.0, 0.0, 0.0, // 5 : x, y, R, G, B
    1.0, 0.0, 0.0, 1.0, 0.0,
    1.0, 1.0, 0.0, 0.0, 1.0,
    0.0, 1.0, 1.0, 1.0, 1.0
  ];

  positions = {
    numComponents : 2,  // (x, y)
    stride : 5 * 4 /* sizeof(float) */,
    offset : 0
  };
  colors = {
    numComponents : 3, // RGB
    stride : 5 * 4 /* sizeof(float) */,
    offset : positions.numComponents * 4
  };
```

Vertex Colors (cont'd)

- Create a single buffer holding all of the vertex attributes

JavaScript

```
function Square() {
  ...
  attributes.buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, attributes.buffer);
  gl.bufferData(gl.ARRAY_BUFFER,
    new Float32Array(attributes), gl.STATIC_DRAW);
```

Vertex Colors (cont'd)

- Find shader variables, just like before

JavaScript

```
function Square() {
  ...
  aPosition = gl.getAttribLocation(program, "aPosition");
  gl.enableVertexAttribArray(aPosition);

  aColor = gl.getAttribLocation(program, "aColor");
  gl.enableVertexAttribArray(aColor);
```

Vertex Colors (cont'd)

- Bind buffer and specify vertex attribute parameters
- Now, we only have one buffer
  - only one bind call
- Still have two attributes
  - one call to set each one up
  - now *stride*, and *offset* are relevant

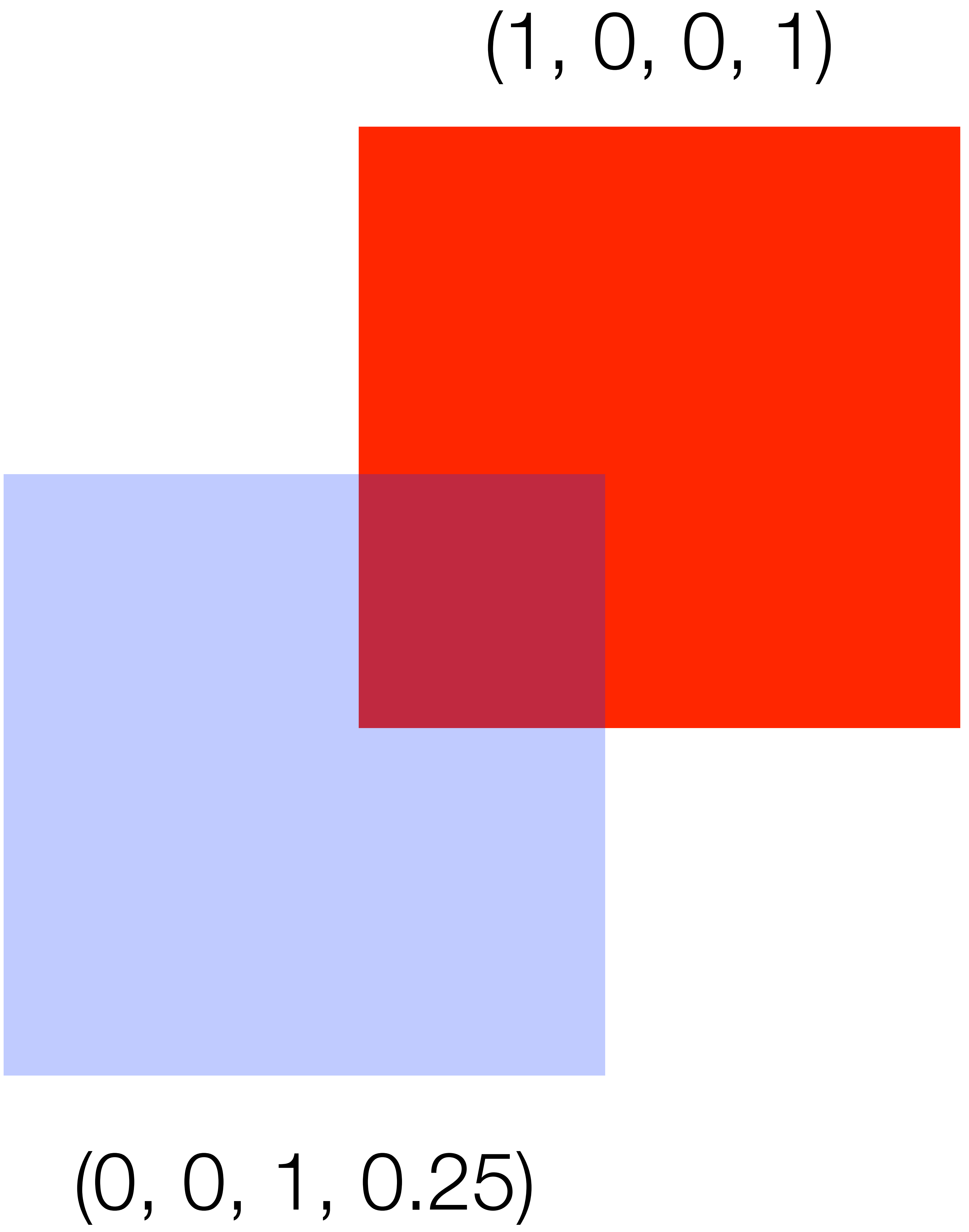
JavaScript

```
function Square() {

  this.render = function () {
    gl.useProgram(program);

    gl.bindBuffer(gl.ARRAY_BUFFER, attributes.buffer);
    gl.vertexAttribPointer(aPosition,
      positions.numComponents, gl.FLOAT,
      false, positions.stride, positions.offset);
    gl.vertexAttribPointer(aColor,
      colors.numComponents, gl.FLOAT,
      false, colors.stride, colors.offset);

    gl.drawArrays(...);
  };
```

Blending

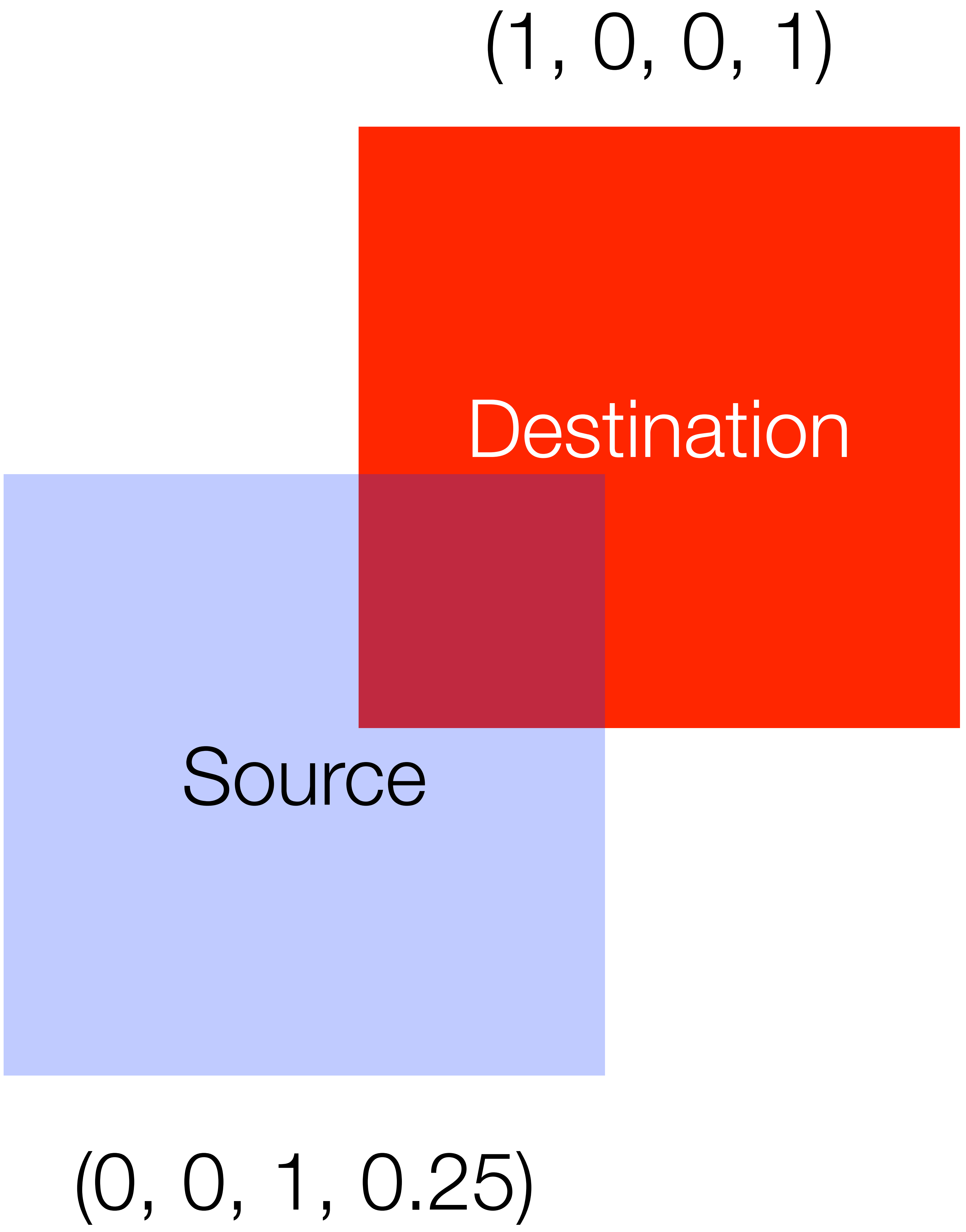
Simulating Translucency

- Up to now, all the objects we're rendered have been *opaque*
- To simulate things like glass, we need to somehow model some position of light being transmitted through the medium
- This is what the *alpha* color component is for
  - it measures translucency
    - 1.0 - totally opaque
    - 0.0 - totally transparent



Alpha Blending

- Incoming fragments are *combined* with the existing color for that pixel location
- Some terminology:
  - *source* color is the newly computed color
  - *destination* color is the existing color in the framebuffer

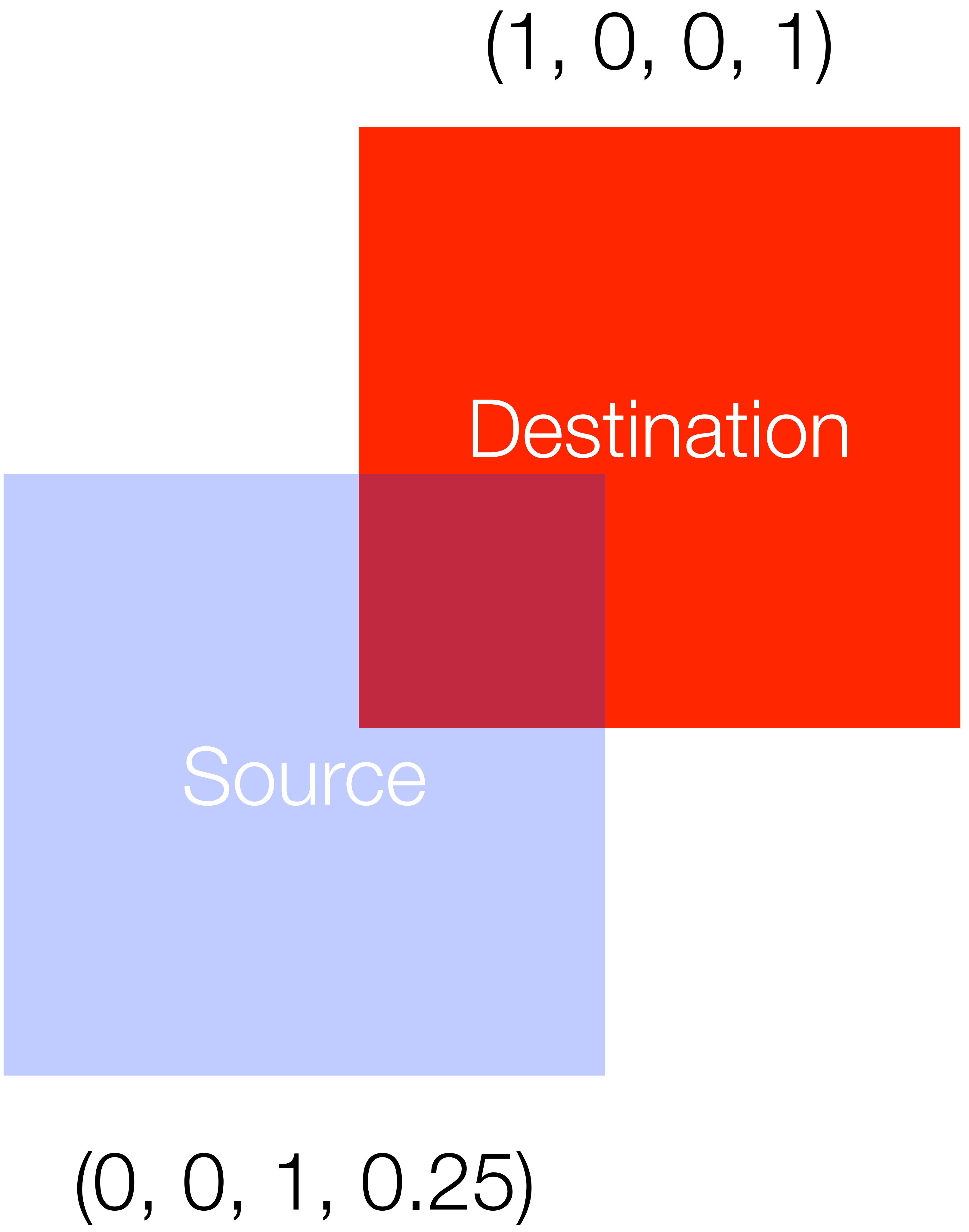


The (Default) Blending Equation

- The default combining function is

$$C = \alpha S + (1 - \alpha)D$$

Source Alpha



Enabling Blending

- Enable (and disable) blending for particular objects in your scene
  - usually not enabled for all of the `render()` routine

JavaScript

```
function render() {
  gl.clear(...);

  gl.enable(gl.BLEND);
  // draw something
  gl.disable(gl.BLEND);
};
```

Blending Factors

- The default equation is really

$$C = f_S\, S \ \mathrm{op}\ f_D\, D$$

- By default, the blending equation uses the *source fragment*'s alpha value

Term	Default	Enum
f_S	α	`gl.SRC_ALPHA`
op	$+$	`gl.FUNC_ADD`
f_D	$1-\alpha$	`gl.ONE_MINUS_SRC_ALPHA`

JavaScript

```
function render() {
  gl.clear(...);

  gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
  gl.blendEquation(gl.FUNC_ADD);
  gl.enable(gl.BLEND);
  // draw something
  gl.disable(gl.BLEND);
};
```

Blending Hints

- Blending works at the *fragment* level
  - no concept of objects
  - depth buffering will affect which fragments are operated on
- General advice:
  - render all opaque objects writing to the depth buffer
  - render translucent objects using depth testing without writing
- There's no magic for rendering order - you need to control that in your application

Depth & Blending

- Fragments from objects behind opaque objects shouldn't affect the pixel's color
  - reject them using depth testing
- However, translucent fragments shouldn't occlude other objects
  - disable them modifying the depth buffer
  - `gl.depthMask()` controls writing to the depth buffer

JavaScript

```javascript
function init() {
  ...
  gl.enable(gl.DEPTH_TEST);
  ...
}

function render() {
  gl.clear(...);

  gl.depthFunc(gl.LESS);  // default setting
  gl.depthMask(true);
  // draw opaque objects
  gl.depthMask(false);
  gl.enable(gl.BLEND);
  // draw translucent objects
  gl.disable(gl.BLEND);
};
```