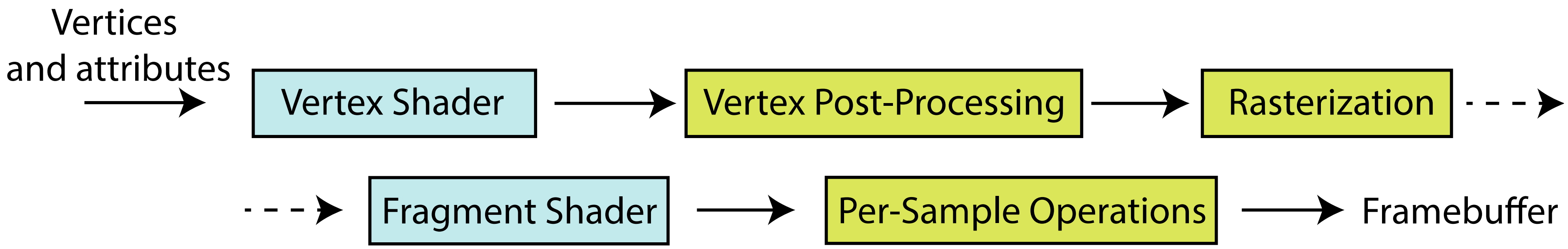
CPSC 427
Video Game Programming

Advanced OpenGL

Helge Rhodin

Coordinate transformations

World Coordinates | Camera Coordinates | Window Coordinates | Pixel-wise attributes*



*usually multiple fragments for every pixel (fragment != pixel)

Recap: GLSL Vertex shader

The OpenGL Shading Language (GLSL)

- Syntax similar to the C programming language
- Build-in vector operations
  - functionality as the GLM library our assignment template uses

```
uniform mat3 transform;
uniform mat3 projection;
in vec3 in_pos;
void main() {
    // Transforming The Vertex
    vec3 out_pos = projection * transform * vec3(in_pos.xy, 1.0);
    gl_Position = vec4(out_pos.xy, in_pos.z, 1.0);
}
```

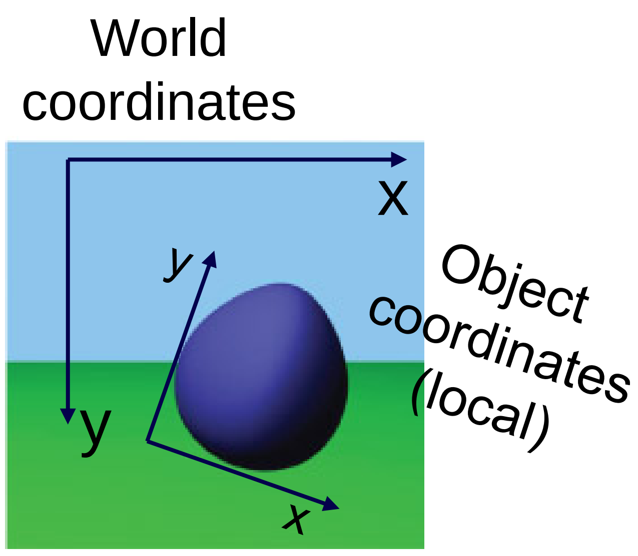
world -> camera

object -> world

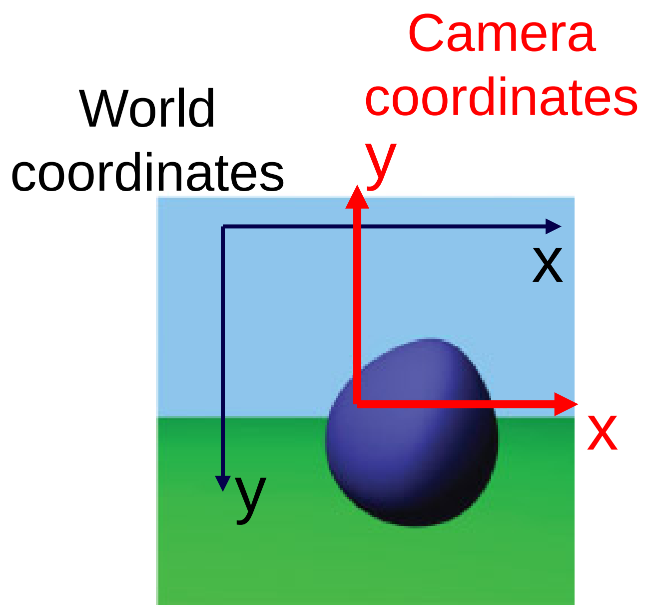
vertex-specific input position

mandatory to set

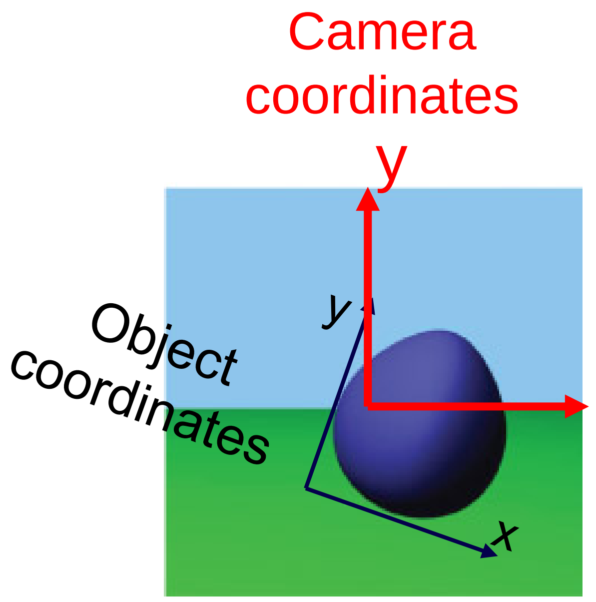
Recap: From local object to camera coordinates



object -> world



world -> camera



object -> camera

`transform`

`projection`

`projection * transform`

Variable Types

Uniform

- same for all vertices

Out/In (varying)

- computed per vertex, automatically interpolated for fragments

In (attribute)

- values per vertex
- available only in Vertex Shader

Setting (Vertex) Shader Variables

Uniform variable

```
mat3 projection_2D{ { sx, 0.f, 0.f },{ 0.f, sy, 0.f },{ tx, ty, 1.f } }; // affine transformation as introduced in the prev. lecture
GLint projection_uloc = glGetUniformLocation(texmesh.effect.program, "projection");
glUniformMatrix3fv(projection_uloc, 1, GL_FALSE, (float*)&projection);
```

In variable (attribute for every vertex)

```
// assuming vbo contains vertex position information already
GLint vpositionLoc = glGetAttribLocation(program, “in_position");
glEnableVertexAttribArray(vpositionLoc);
glVertexAttribPointer(vpositionLoc, 3, GL_FLOAT, GL_FALSE, sizeof(vec3), (void*)0);
```

Salmon Vertex shader

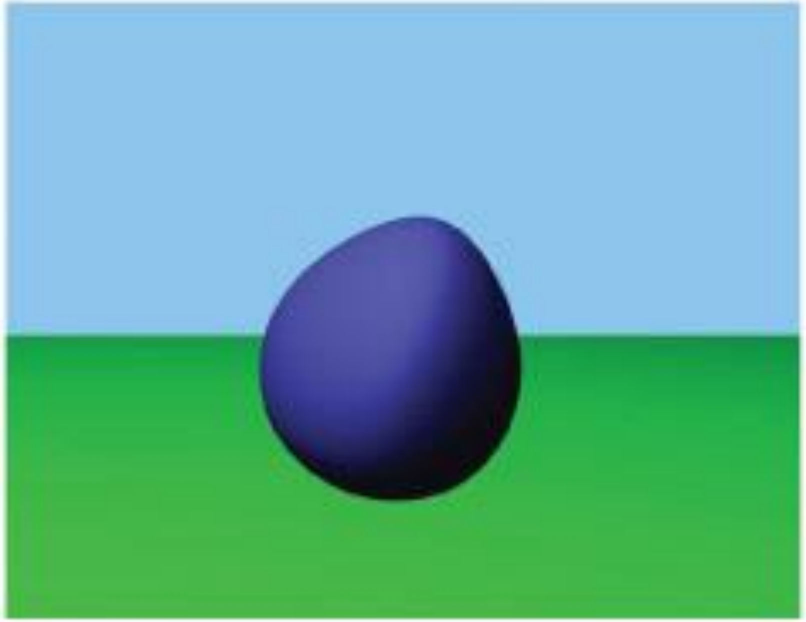
```
#version 330
// Input attributes
in vec3 in_position;
in vec3 in_color;

out vec3 vcolor;
out vec2 vpos;

// Application data
uniform mat3 transform;
uniform mat3 projection;

void main() {
        vpos = in_position.xy; // local coordinated before transform
        vcolor = in_color;
        vec3 pos = projection * transform * vec3(in_position.xy, 1.0);
        gl_Position = vec4(pos.xy, in_position.z, 1.0);
}
```

pass on color and position in object coordinates

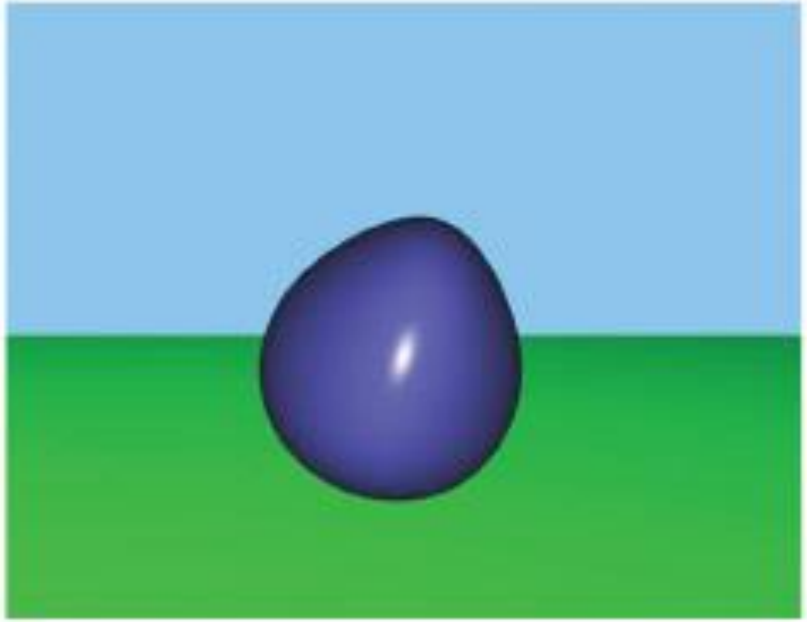
as before

Recap: Fragment shader examples

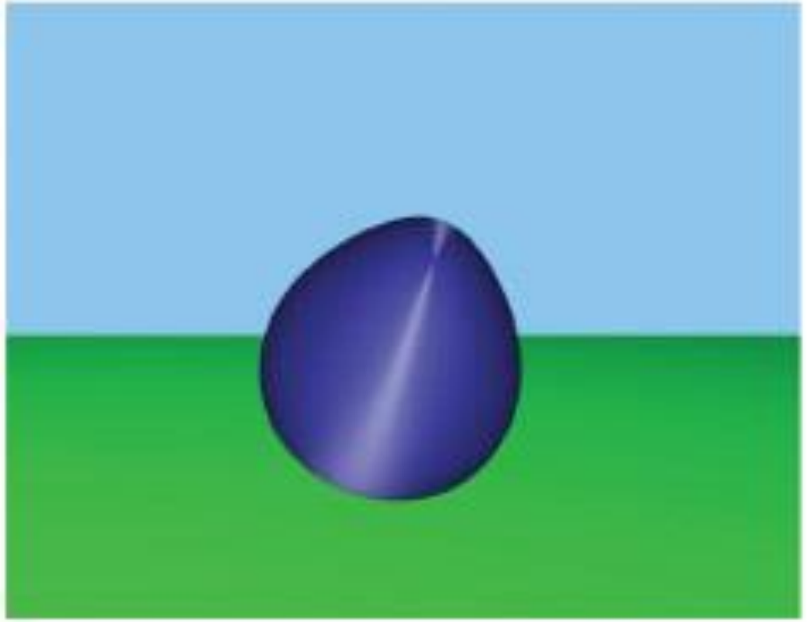
- ***simulates materials and lights***
- ***can read from textures***



Diffuse

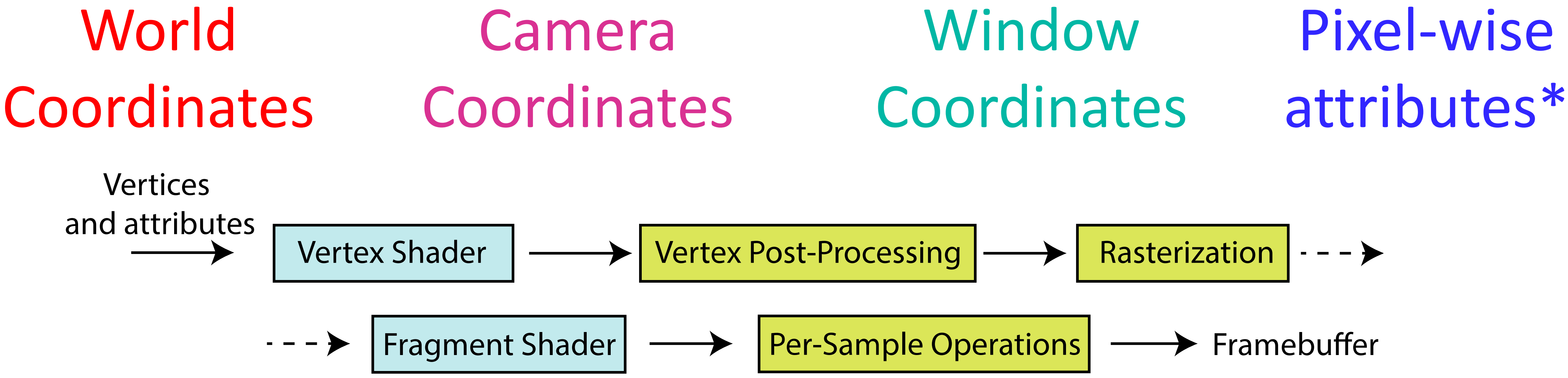


Specular



Directional

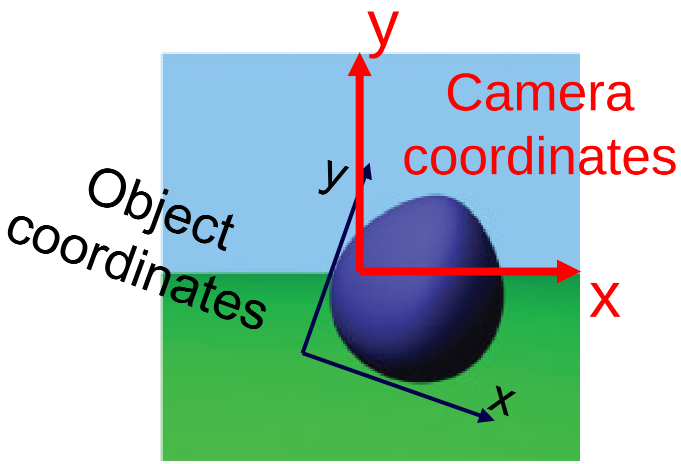
Coordinate transformations



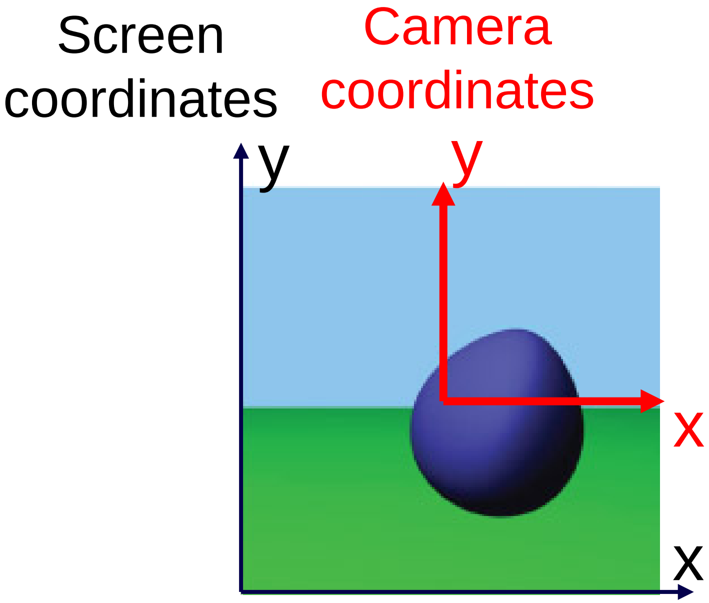
*usually multiple fragments for every pixel (fragment != pixel)

(Hidden) Vertex Post-Processing

- Viewport transform: camera coordinates to screen/window coordinates
  - set with `glViewport(0, 0, w, h);`



object -> camera

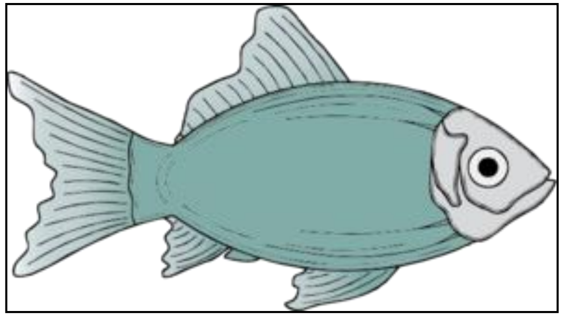


camera -> screen

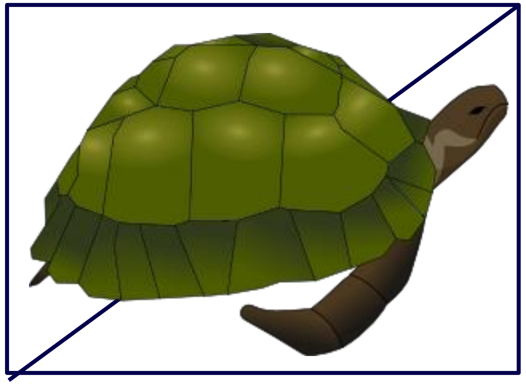
- Clipping: Removing invisible geometry (outside view frame)

SPRITES: Faking 2D Geometry

- Creating geometry is hard
- Creating texture is “easy”
- In 2D it is hard to see the difference

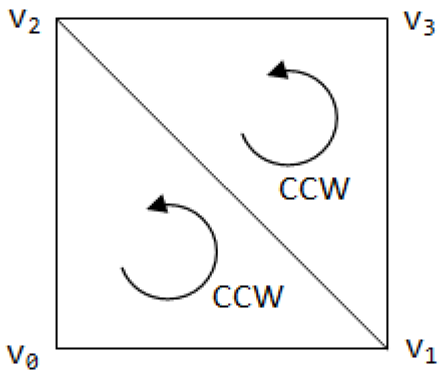


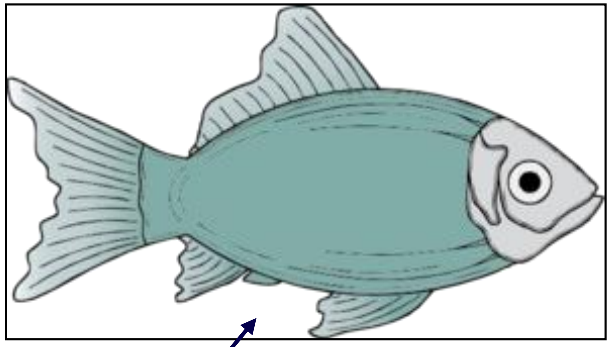
- SPRITE:
  - *Use basic geometry (rectangle = 2 triangles)*
  - *Texture the geometry (transparent background)*
  - *Use blending (more later) for color effects*

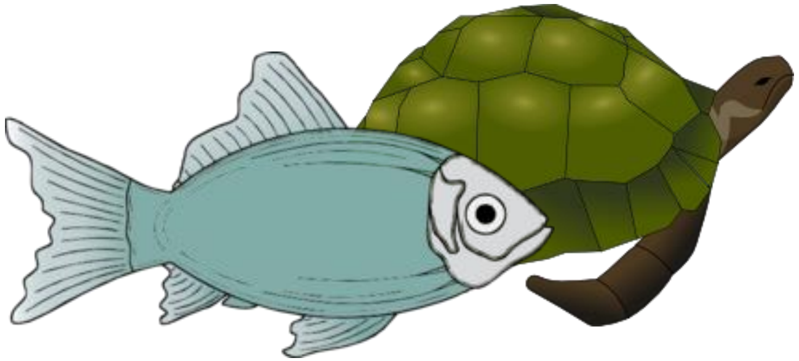


Sprite basics

A textured quad looks like fine-grained 2D geometry







Transparent with alpha = 0
e.g., color_RGBA = {1,1,1,0}

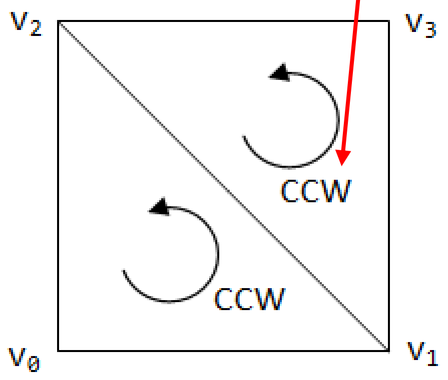
Proper occlusion despite simple geometry

SPRITES: Creation

OpenGL initialization (once):

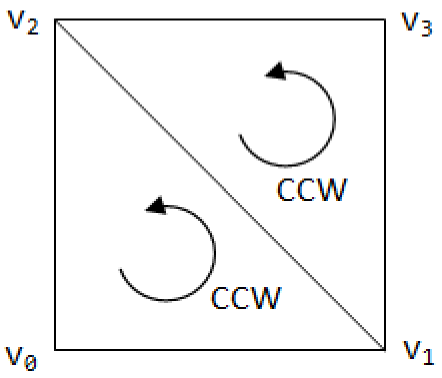
Counter-clockwise winding (CCW)

Create Quad Vertex Buffer




```
vec3 vertices[] = { v0, v1, v2, v3 };

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, vertices_size, vertices,
GL_STATIC_DRAW);
```

SPRITES: Creation

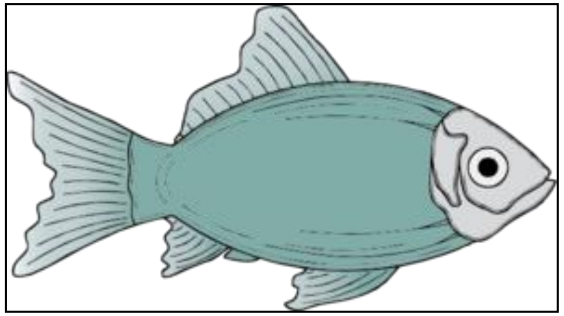


OpenGL initialization (once):

Create Quad Index Buffer

```
uint16_t indices[] = { 0, 1, 2, 1, 3, 2 };
Gluint ibo;
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices_size, indices,
GL_STATIC_DRAW);
```

Load Texture



```
Gluint tex_id;
glGenTextures(1, &tex_id);
glBindTexture(GL_TEXTURE_2D, tex_id);
glTexImage2D(GL_TEXTURE_2D, GL_RGBA, width, height, .., tex_data);
```

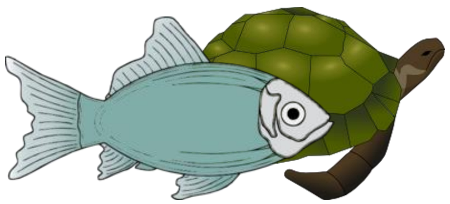
SPRITES: Rendering

OpenGL rendering (every frame):

Bind Buffers

```
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
```

Enable Alpha Blending



```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
// Alpha Channel Interpolation
// RGB_o = RGB_src * ALPHA_src + RGB_dst * (1 - ALPHA_src)
```

SPRITES: Rendering

Bind Texture

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texmesh.texture.texture_id);
```

Draw

```
glDrawElements(GL_TRIANGLES, 6, ..); // 6 is the number of indices
```

Color and Texture Mapping

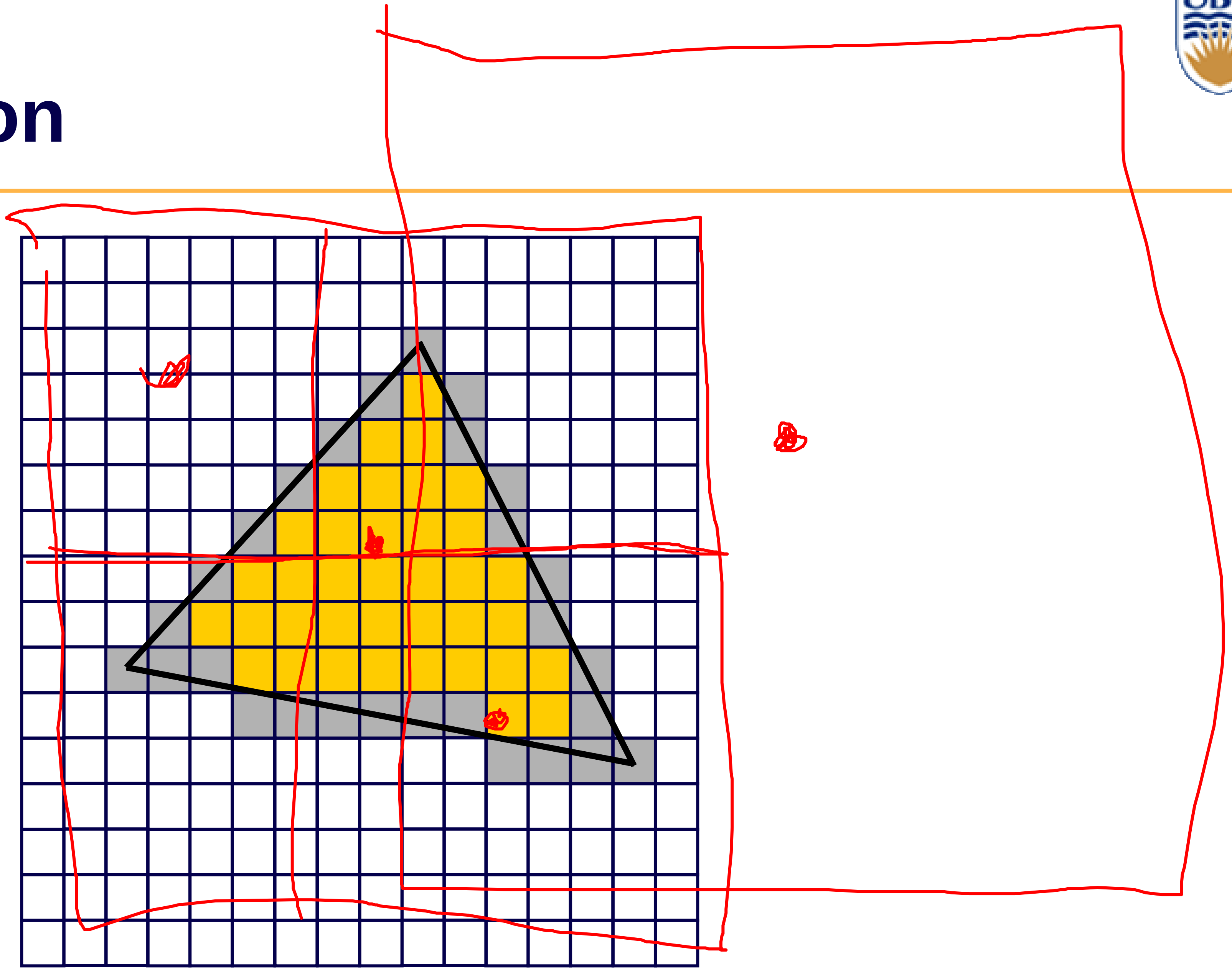
- ***How to map from a 2D texture to a 3D object that is projected onto a 2D scene?***

Scan Conversion/Rasterization

- Convert continuous 2D geometry to discrete
- Raster display – discrete grid of elements
- Terminology

  - ***Screen Space:*** *Discrete 2D Cartesian coordinate system of the screen pixels*

Scan Conversion

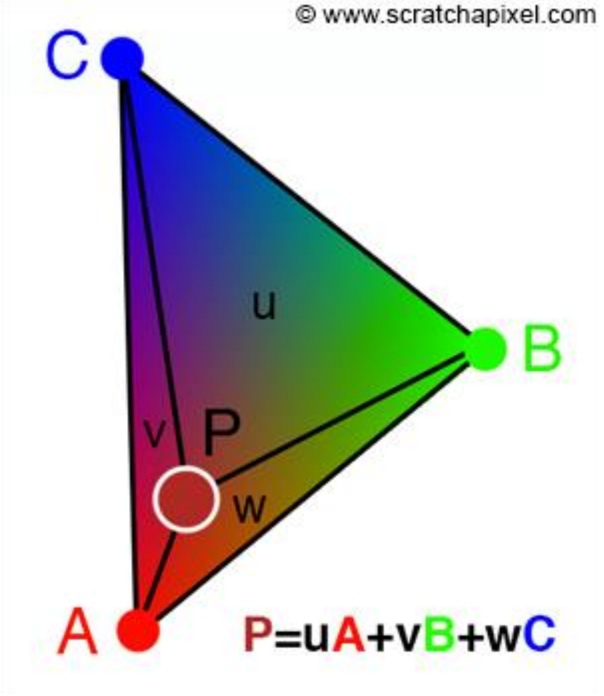


Self study:

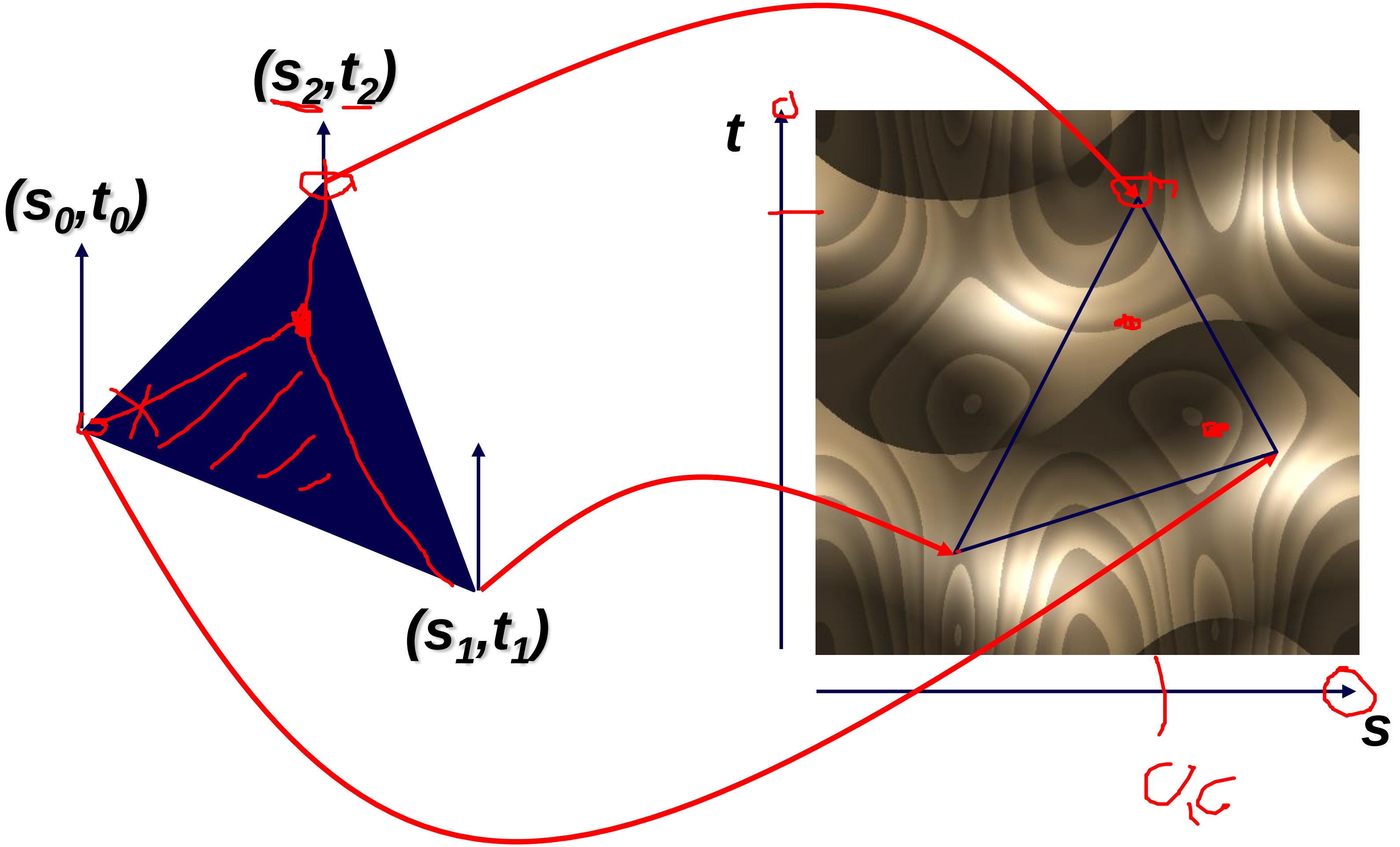
Interpolation with barycentric coordinates

- *linear combination of vertex properties*
  - *e.g., color, texture coordinate, surface normal/direction, …*

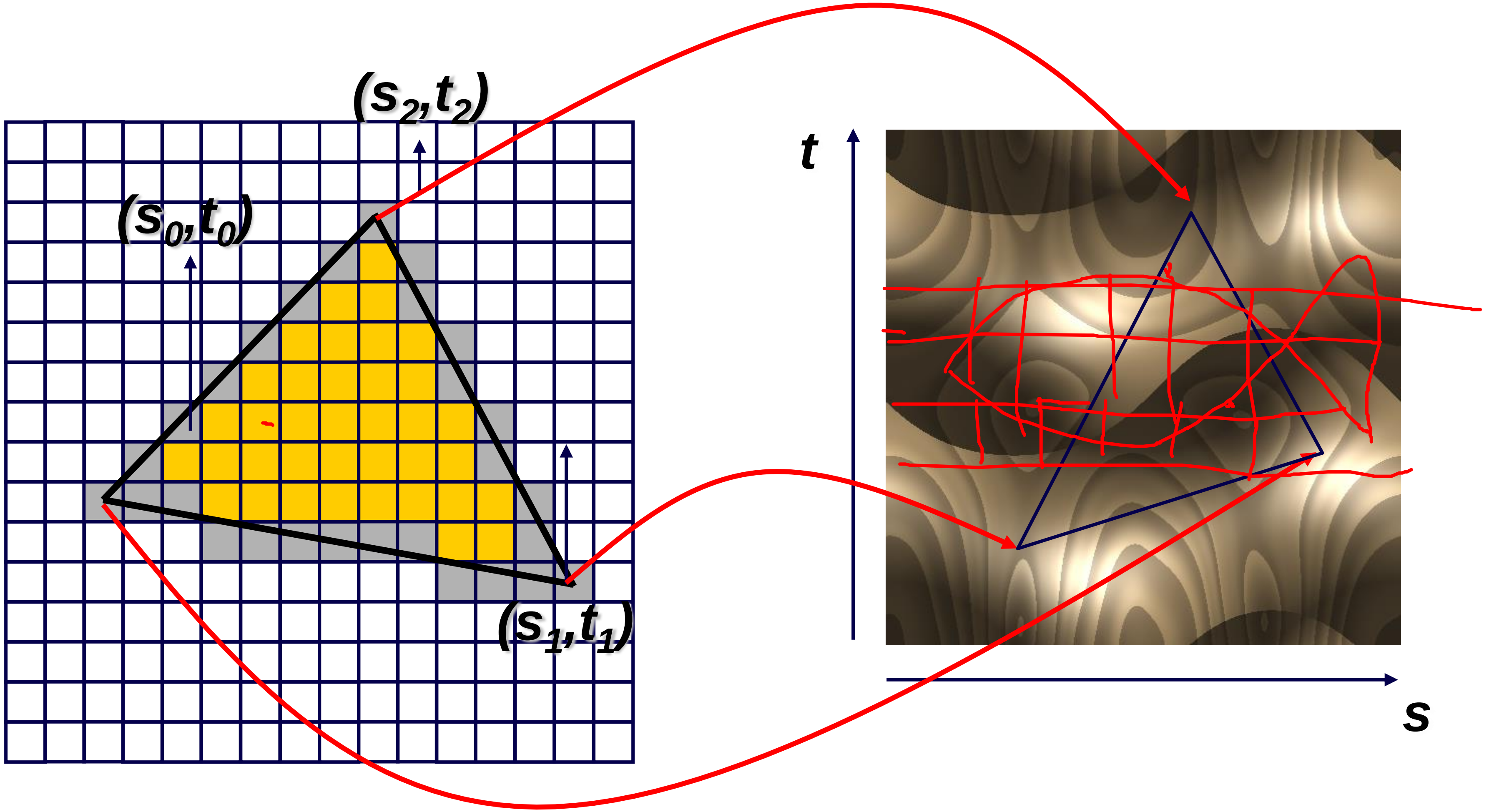
- *weights are proportional to the areas spanned by the sides to query point P*



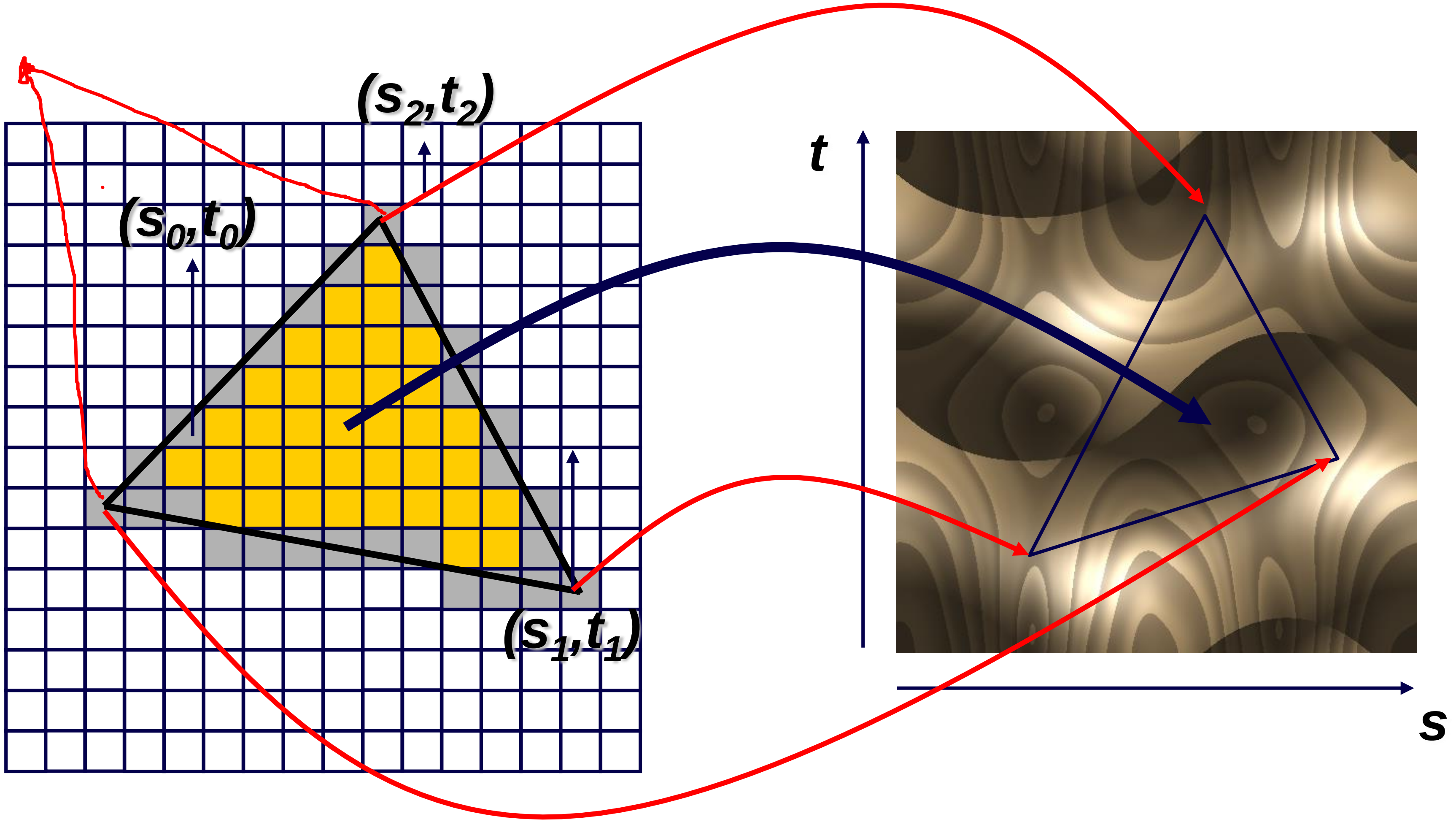
Texture mapping



Texture mapping



Texture mapping



Blending

Blending:

- Fragments -> Pixels
- Draw from farthest to nearest
- No blending – replace previous color
- Blending: combine new & old values with some arithmetic operations
  - *Achieve transparency effects*

Frame Buffer : video memory on graphics board that holds resulting image & used to display it

Depth Test / Hidden Surface Removal

Remove occluded geometry

- Parts that are hidden behind other geometry
- For 2D (view parallel) shapes – use depth order
  - *draw objects back to front*
    - sort objects: furthest object first, closest object last

Self study: Alternative to ordering

Depth buffer with transparent sprites

- ***Fragment shader writes depth to the depth buffer***
  - *discard fragment if depth larger than current depth buffer (occluded)*
  - *alleviates the ordering of objects*
- ***Issue, depth buffer written for fragments with alpha = 0***
- ***Solution:***
  explicitly discard fragments with alpha < 0.5
  - *note, texture sample interpolation leads to non-binary values even if texture is either 0 or 1.*

```
#version 330
in vec2 texCoord;
out vec4 outColor;
uniform sampler2D theTexture;

void main() {
  vec4 texel = texture(theTexture, texCoord);
  if(texel.a < 0.5)
    discard;
  outColor = texel;
}
```

CPSC 427
Video Game Programming

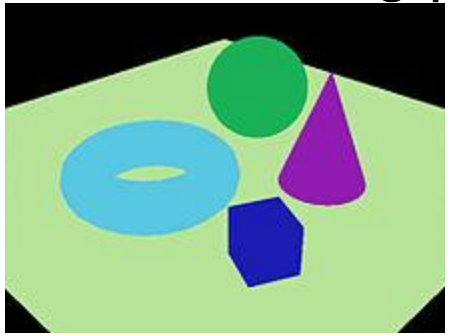
Advanced OpenGL

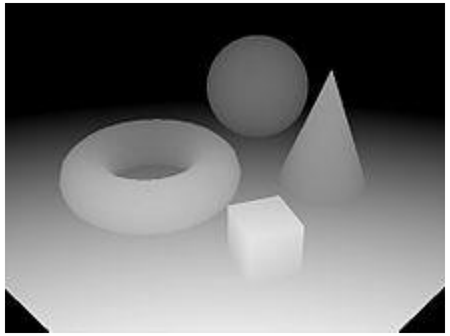
Helge Rhodin

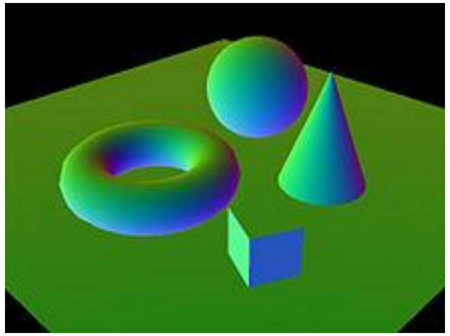
Motivation

- ***Deferred shading (a form of screen-space rendering)***

First rendering pass

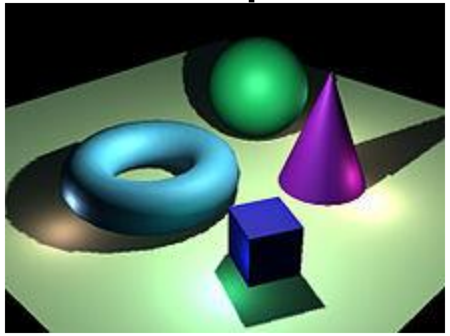






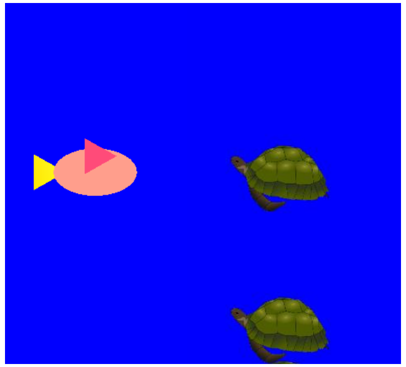
Input ➡

Second pass

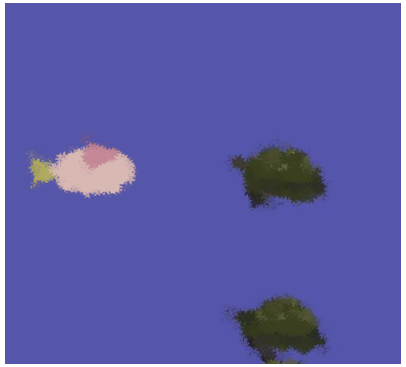


- ***or water effects***

First pass



Second pass



Shaders - GLSL

- Each stage is expected to produce a certain output:
  - Vertex Shader Output: Vertex clip-space position
  - Fragment Shader Output: Pixel color
- Input data comes from:
  - Attributes: Geometry or previous stage's output
  - Uniforms: Variables, Arrays, Textures, ..
- Extensive built-in library
- Stages have to have matching input/outputs

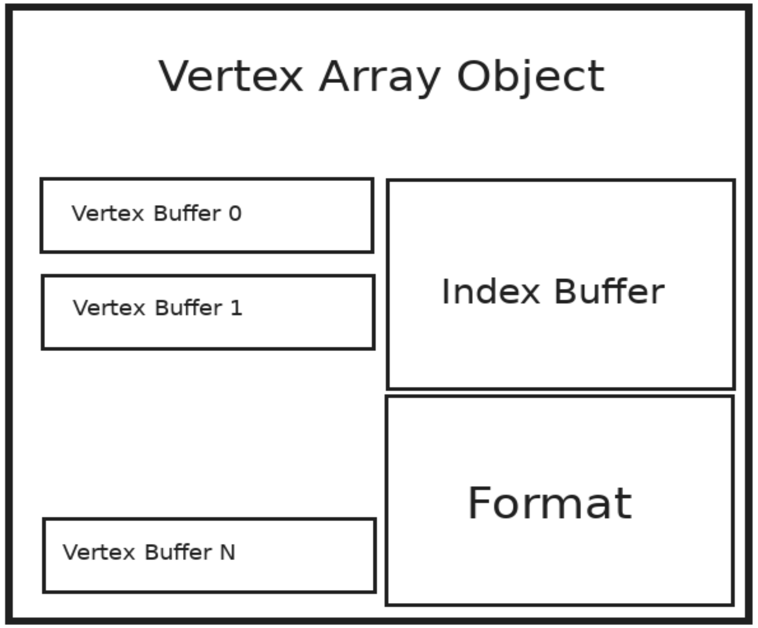
Let’s start from resources

- Reside in GPU memory
- Standard lifecycle (`glGen*` `glBind*` `glDelete*`)
- Require to be bound to be used `glBind*` (State-machine OpenGL)
- Different types:
  - *Buffers*
  - *Textures (& Samplers)*
  - *Shaders*
  - *Framebuffers*
  - ..

Geometry

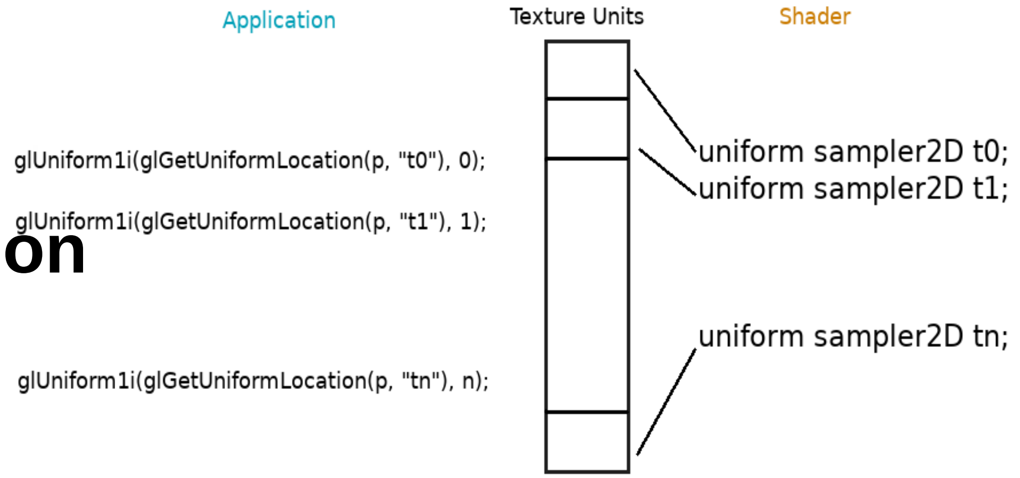
- **Explicit representation as a set of vertices organized in primitives**
- **Vertices and indices are contained in Buffers**
- **Submitted through Vertex Array Objects (VAO)**
- **VAOs are containers for:**

  Vertex Data (VBOs)
  Index Data (IBOs)
  Format (glVertexAttribPointer)



Textures & Samplers

- Conceptually similar to 2D (or 3D) buffers
- Used(sampled) by Shader Samplers
- Filtering options set by the application
- Binding done through Texture Units

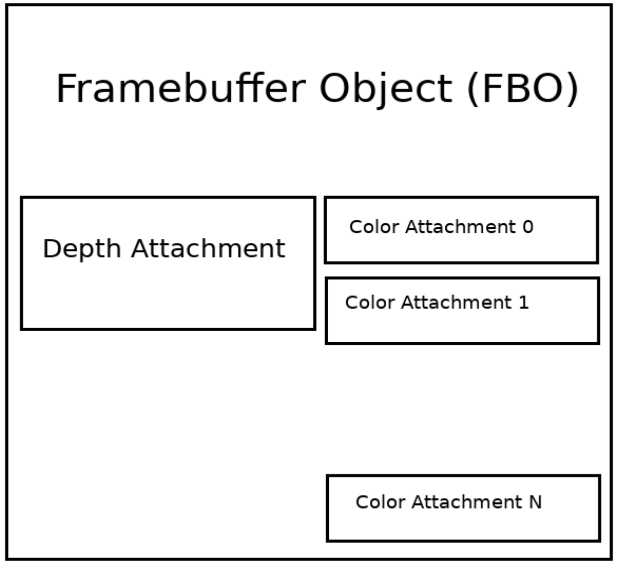


Sampler(Shader): Bound to texture units using `glUniform1i()`

Textures(App): Bound to to texture units using `glActiveTexture()`

Framebuffers

- **The output of the rendering pipeline is written to Texture(s)**
- **Framebuffers are containers for such Textures**
- **They allow for two types of attachment**
  - ***Color(s): Fragment shader outputs***
  - ***Depth/Stencil: Depth buffer***
- **Framebuffer 0 (default) writes to the window's buff**
- **Contained Textures can be reused in later stages (Render to Texture)**

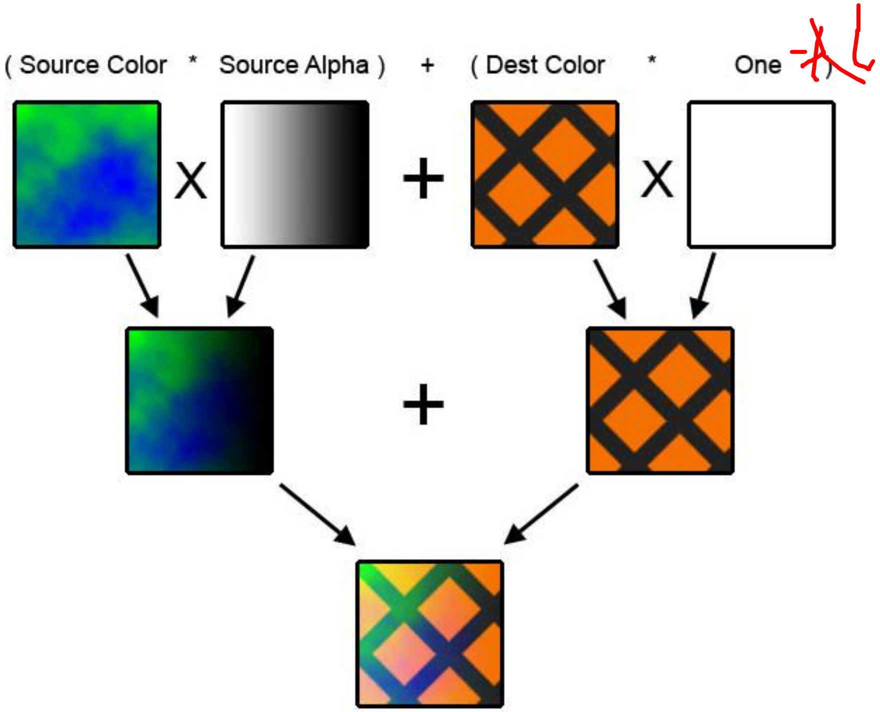


A few advanced examples

- Blending
- Sprite Sheets
- Render to Texture
- Post-processing Effects: Bloom

Blending

- Controls how pixel color is blended into the FBO's Color Attachment
- Control on factors and operation of the equation
- RGB and Alpha are controllabe separately



$$RGB_o = RGB_{src} * F_{src} \; [+ \; - \; / \; *] \; RGB_{dst} * F_{dst}$$

Cloud (source) on top of grid (dest)

Blending: Example Presets

- **Additive Blending**

```
//---
// RGB_o = RGB_src + RGB_dst
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
```

- **Alpha Blending**

```
//---
// RGB_o = RGB_src * ALPHA_src + RGB_dst * (1 - ALPHA_src)
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Sprite Sheets

- Compact (and fast) approach for 2D animations
- Every frame only a region of the original Texture is rendered
- Texture Coordinates are updated as clock ticks
- Does not require dynamic VBOs

Time

Animation type

Sprite Sheets: Example

```
// APPLICATION
void load() {
    // ANIMATION_FRAME_[W|H] is in texture coordinates range [0, 1]
    vertices[0].texcoord = (0, 0);
    vertices[1].texcoord = (ANIMATION_FRAME_W, 0);
    vertices[2].texcoord = (ANIMATION_FRAME_W, ANIMATION_FRAME_H);
    vertices[3].texcoord = (0, ANIMATION_FRAME_H);
}

void update(float ms) {
    elapsed_time += ms;
    if (elapsed_time > ANIMATION_SPEED)
        frame = (frame+1)%NUM_ANIMATION_FRAMES;
}

void render() {
    glUniform1i(shader_program, &frame);
    ..
}
```

```
// SHADER
uniform vec2 texcoord_in; // Attribute coming from geometry (.texcoord)

void main() {
    texcoord = texcoord_in;

    // Sliding coordinates along X direction
    texcoord.x += ANIMATION_FRAME_W * frame;
}
```

Render To Texture

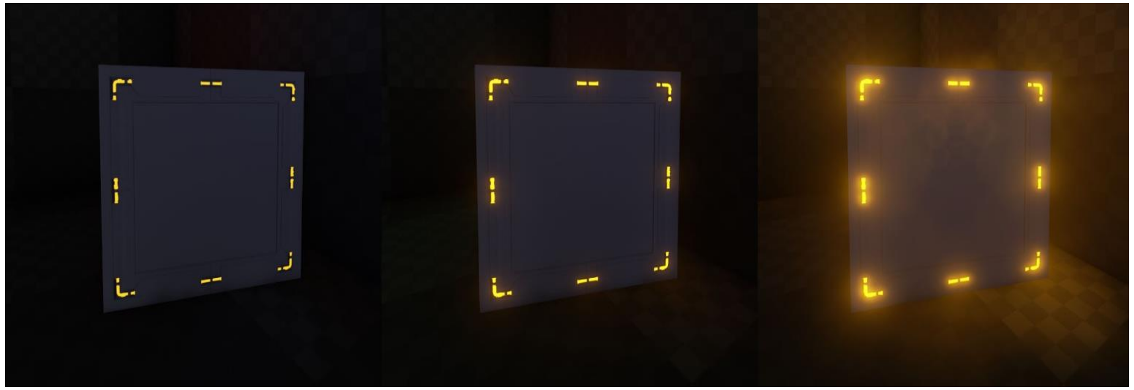
- **Building block of any multipass pipeline**
- **Just putting two concepts together..**

- First Pass: Pixel colors are written to the FBO's Color Attachment
- Second Pass: The same Texture can be bound and used by Samplers

```
// When Loading
render_target = create_texture(screen_resolution)  // Create texture (Usually with same screen resolution)
fbo = create_fbo(render_target) // Bind <render_target> as <fbo>'s color attachment

// First pass
bind_fbo(fbo)
draw_first_pass()

// Second pass
bind_fbo(0) // Reset to default FBO (Window)
bind_texture(render_target) // You can use <render_target> as you would you any other texture
draw_second_pass()
```

Post-processing: Bloom

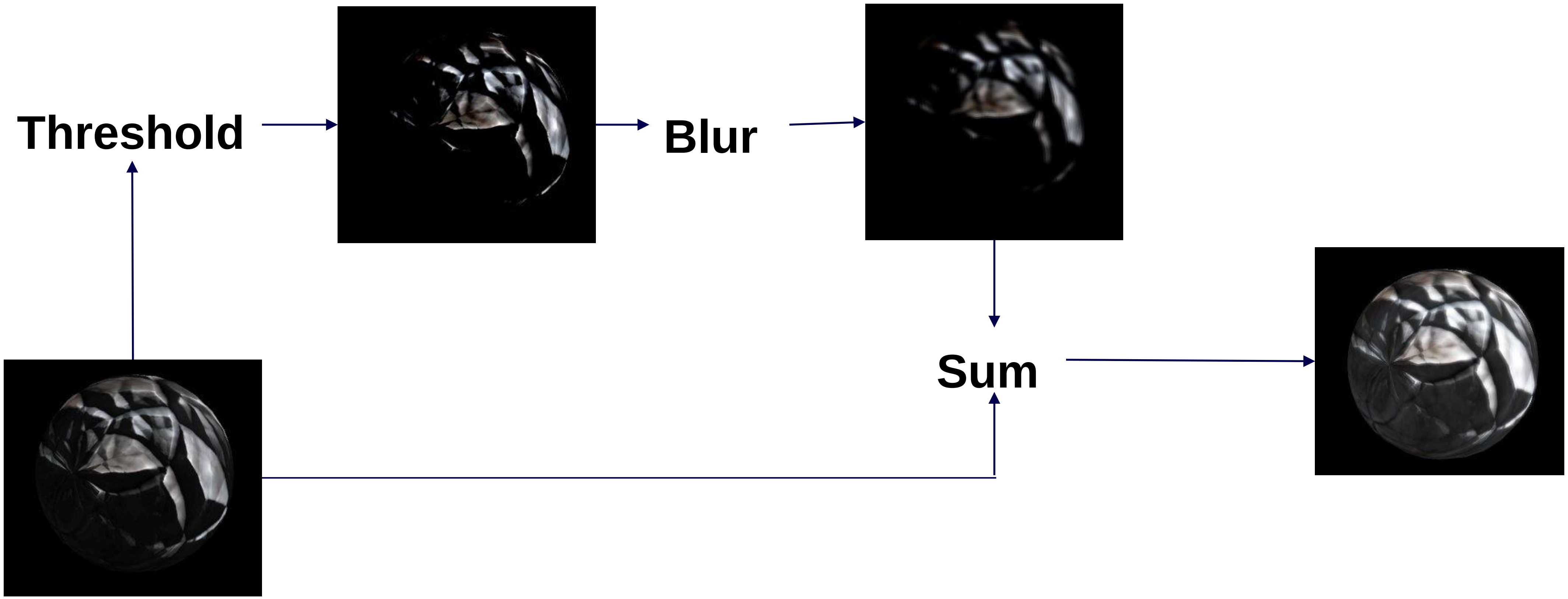
- Fullscreen Effect to highlight bright areas of the picture
- Post-processing: Operates on Images after the scene has been rendered



- High level overview:

1. Render scene to texture
2. Extract bright regions by thresholding
3. Gaussian blur pass on the bright regions
4. Combine original texture and highlights texture with additive blending

Post-processing: Bloom



Post-processing: Bloom

```
GLuint original_rt, bright_rt, blur_rt;

bind_fbo(original_rt);
render_scene(original_rt);

bind_fbo(bright_rt);
threshold(original_rt); // Only keep pixels brighter than threshold

bind_fb(blur_rt);
gaussian_blur(bright_rt); // Blur bright regions

bind_fbo(0); // Writing to window's framebuffer
add(blur_rt, original_rt);
```

Self study:
Post-processing: Bloom

As many details have been skipped, here are a couple of hints:

- A fullscreen effect is achieved by rendering a textured quad with the same dimensions as the screen. No need for any camera or projection matrix as you already know that you want the vertices to correspond to the corners of the screen.

- Thresholding bright areas can be achieved in the fragment shader with something as simple as: `return Intensity > Threshold ? Color : 0.0;`
- Where `Intensity` is some function of the pixel's RGB values. You can start from max component, average, or explore other color space.

- Regarding Gaussian Blur (or Bloom altogether) there are lots of online resources of various quality.
- A suggested place to start for tutorials is https://learnopengl.com.
- The standard reference book for real-time rendering is "Real-Time Rendering" (http://www.realtimerendering.com/)

Shaders: Example (A1)

```
// COMPILATION
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vs_src, &vs_len);

GLuint compile_shader(int type, const char* src) {
    // CREATION
    GLuint shader = glCreateShader(type);
    glShaderSource(shader, 1, src, strlen(src));

    // COMPILE
    glCompileShader(shader);
    GLint success = 0;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (success == GL_FALSE)
        // ERROR

    return shader;
}

GLuint vs = compile_shader(GL_VERTEX_SHADER, "...");
GLuint fs = compile_shader(GL_PIXEL_SHADER, "...");
// LINKING
GLuint program = glCreateProgram();
glAttachShader(program, vs);
glAttachShader(program, fs);
glLinkProgram(program);
```

Shaders - GLSL: Example

```
//---
// vertex_shader.glsl
// Input attributes as provided by Vertex Array Object (VAO)
in vec3 position_in;
in vec2 texcoord_in;

// Output attributes passed to the fragment shader
out vec2 texcoord;

// Uniform data passed from the application
uniform mat4 MVP;

void main() {
    texcoord = texcoord_in;

    // The vertex shader expects a gl_Position to be written to.
    gl_Position = MVP * vec4(position_in, 1.0);
}

//---
// fragment_shader.glsl
// Every 'out' in the vertex shader should have an equivalent 'in' with the same name.
// If you want to use different names, you need to explictly set the layout location.
in vec2 texcoord;

// Uniform data passed from the application
uniform sampler2D color_map;

void main() {
    // texture2D is a built-in. For more, see https://www.khronos.org/registry/OpenGL-Refpages/gl4/index.php
    return texture2D(color_map, texcoord);
}
```

Framebuffers: Example

```
//---
// When loading
// Creating a texture to store the color output
GLuint render_target;
glGenTextures(1, &render_target);
glBindTexture(GL_TEXTURE_2D, render_target); // Subsequent operations will affect <render_target>

// Reserves the space for a WxH texture. NULL indicates that we don't want to upload
// any initial values
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, W, H, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);

// Creating a texture to be used as depth buffer
GLuint depth_buffer;
glGenTextures(1, &depth_buffer);
glBindTexture(GL_TEXTURE_2D, depth_buffer);
// Similar to the color texture, we create an empty WxH texture. The only difference is in
// the format: instead of RGB, we just need to store 1 single value which occupies all 32bits
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32, W, H, 0, GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE, NULL);

// Creating out Framebuffer Object (FBO)
GLuint fbo;
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

// When <fbo> is bound, it will use <render_target> to write the color and <depth_buffer> to write the depth.
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, render_target, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depth_buffer, 0);

// ---
// When rendering
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```