CPSC 436D

Hands-on OpenGL

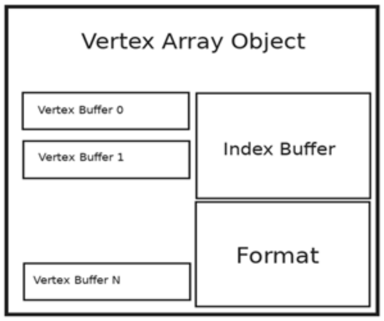
Let’s start from resources

- 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)

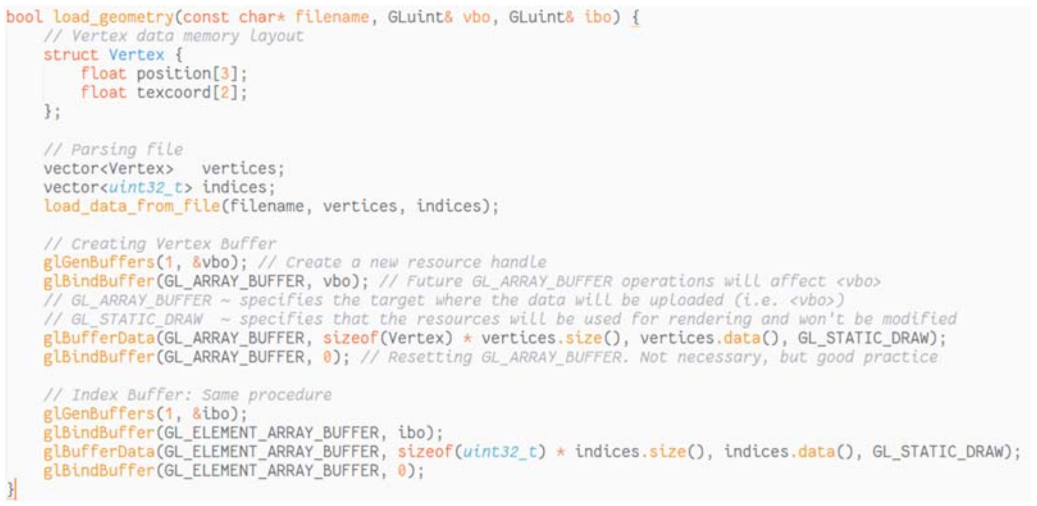


Dominici

Geometry: Example

```
// ---
// When loading..
GLuint vbo, ibo;
load_geometry("salmon.mesh", vbo, ibo);

GLuint vao;
glGenVertexArrays(vao); // Generate container
glBindBuffer(GL_ARRAY_BUFFER, vbo); // <vao> now references <vbo>'s vertices
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); // <vao> now references <ibo>'s indices

// POSITION_LOC and TEXCOORD_LOC are the locations the vertex shaders expects the
// attributes to be bound. By default correspond to order of declaration.
// in vec3 in_pos;      <== Location 0
// in vec2 in_texcoord; <== Location 1
// The remaining arguments specify how the data is laid out in <vbo>. In this example
// the data is interleaved: |x1|y1|z1|u1|v1|x2|y2|z2|.... |vn|
glVertexAttribPointer(POSITION_LOC, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0)
glVertexAttribPointer(TEXCOORD_LOC, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), sizeof(float) * 3);
glEnableVertexAttribArray(POSITION_LOC);
glEnableVertexAttribArray(TEXCOORD_LOC);

// ---
// When Rendering..
glBindVertexArray(vao);
```

Geometry: Example

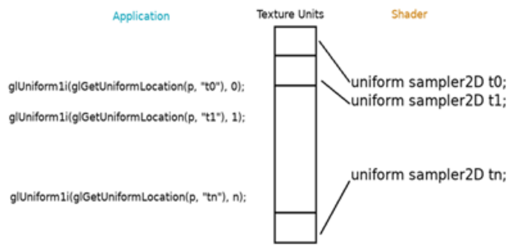


```
bool load_geometry(const char* filename, GLuint& vbo, GLuint& ibo) {
    // Vertex data memory layout
    struct Vertex {
        float position[3];
        float texcoord[2];
    };

    // Parsing file
    vector<Vertex>   vertices;
    vector<uint32_t> indices;
    load_data_from_file(filename, vertices, indices);

    // Creating Vertex Buffer
    glGenBuffers(1, &vbo); // Create a new resource handle
    glBindBuffer(GL_ARRAY_BUFFER, vbo); // Future GL_ARRAY_BUFFER operations will affect <vbo>
    // GL_ARRAY_BUFFER ~ specifies the target where the data will be uploaded (i.e. <vbo>)
    // GL_STATIC_DRAW  ~ specifies that the resources will be used for rendering and won't be modified
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * vertices.size(), vertices.data(), GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0); // Resetting GL_ARRAY_BUFFER. Not necessary, but good practice

    // Index Buffer: Same procedure
    glGenBuffers(1, &ibo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(uint32_t) * indices.size(), indices.data(), GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
```

© Edoardo A. Dominici

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**

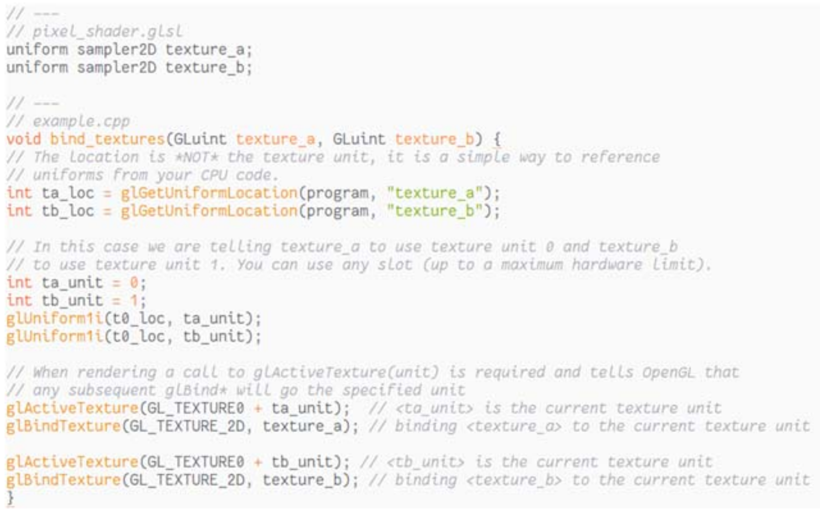

Application
Texture Units
Shader
glUniform1i(glGetUniformLocation(p, "t0"), 0);
glUniform1i(glGetUniformLocation(p, "t1"), 1);
glUniform1i(glGetUniformLocation(p, "tn"), n);
uniform sampler2D t0;
uniform sampler2D t1;
uniform sampler2D tn;

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

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

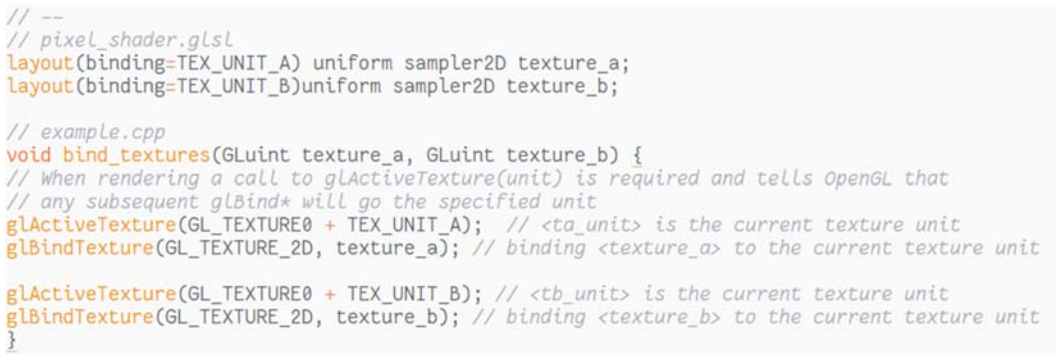
© Edoardo A. Dominici

Textures & Samplers: Example



```
// ---
// pixel_shader.glsl
uniform sampler2D texture_a;
uniform sampler2D texture_b;

// ---
// example.cpp
void bind_textures(GLuint texture_a, GLuint texture_b) {
// The location is *NOT* the texture unit, it is a simple way to reference
// uniforms from your CPU code.
int ta_loc = glGetUniformLocation(program, "texture_a");
int tb_loc = glGetUniformLocation(program, "texture_b");

// In this case we are telling texture_a to use texture unit 0 and texture_b
// to use texture unit 1. You can use any slot (up to a maximum hardware limit).
int ta_unit = 0;
int tb_unit = 1;
glUniform1i(t0_loc, ta_unit);
glUniform1i(t0_loc, tb_unit);

// When rendering a call to glActiveTexture(unit) is required and tells OpenGL that
// any subsequent glBind* will go the specified unit
glActiveTexture(GL_TEXTURE0 + ta_unit);  // <ta_unit> is the current texture unit
glBindTexture(GL_TEXTURE_2D, texture_a); // binding <texture_a> to the current texture unit

glActiveTexture(GL_TEXTURE0 + tb_unit); // <tb_unit> is the current texture unit
glBindTexture(GL_TEXTURE_2D, texture_b); // binding <texture_b> to the current texture unit
}
```

Textures: Example (Modern 4.2+)

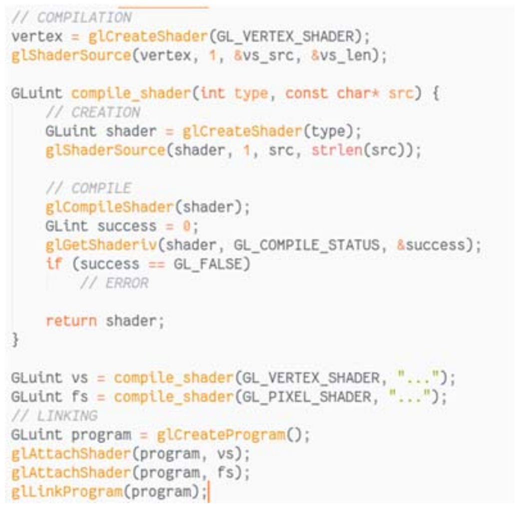


```
// --
// pixel_shader.glsl
layout(binding=TEX_UNIT_A) uniform sampler2D texture_a;
layout(binding=TEX_UNIT_B)uniform sampler2D texture_b;

// example.cpp
void bind_textures(GLuint texture_a, GLuint texture_b) {
// When rendering a call to glActiveTexture(unit) is required and tells OpenGL that
// any subsequent glBind* will go the specified unit
glActiveTexture(GL_TEXTURE0 + TEX_UNIT_A);  // <ta_unit> is the current texture unit
glBindTexture(GL_TEXTURE_2D, texture_a); // binding <texture_a> to the current texture unit

glActiveTexture(GL_TEXTURE0 + TEX_UNIT_B); // <tb_unit> is the current texture unit
glBindTexture(GL_TEXTURE_2D, texture_b); // binding <texture_b> to the current texture unit
}
```

Shaders

- Custom code which runs on the GPU at different stages
- Requires compilation and linking
- Linked into a single Program (**Vertex Shader** + .. + **Pixel Shader**)
- Standard lifecycle `glGenShaders`() `glDeleteShaders`()

Shaders: Example (A1)

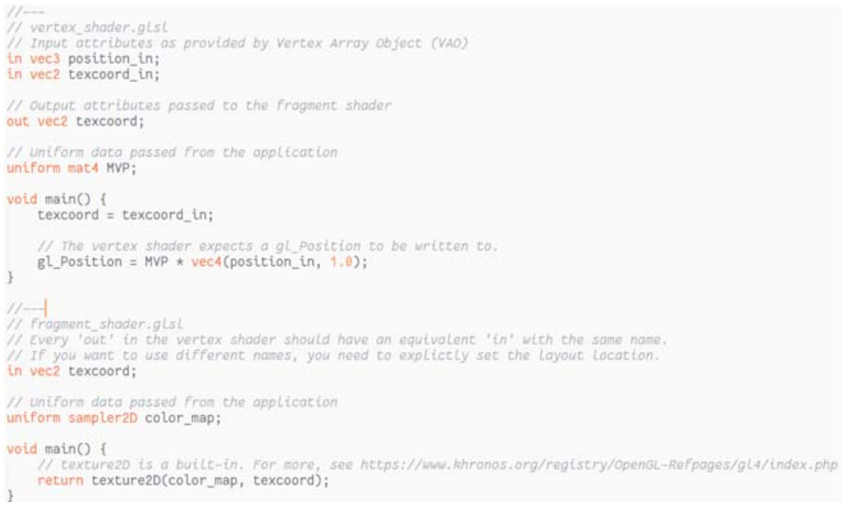


```
// COMPILATION
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vs_src, &vs_len);

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

- 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

Shaders - GLSL: Example

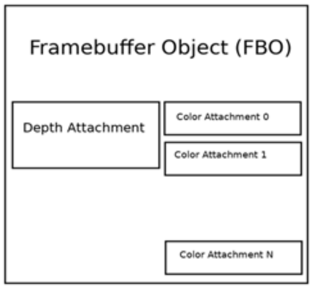


```
//---
// vertex_shader.glsl
// Input attributes as provided by Vertex Array Object (VAO)
in vec3 position_in;
in vec2 texcoord_in;

// 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);
}
```

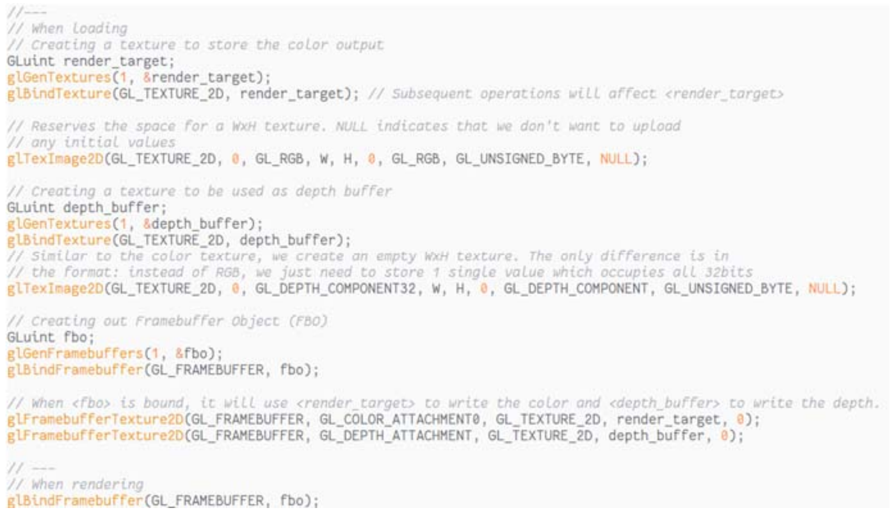
Framebuffers

- 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 buffer
- Contained **Textures** can be reused in later stages (Render to Texture)



Framebuffers: Example

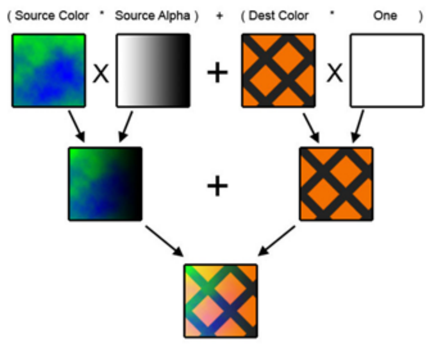


```
//---
// When loading
// Creating a texture to store the color output
GLuint render_target;
glGenTextures(1, &render_target);
glBindTexture(GL_TEXTURE_2D, render_target); // Subsequent operations will affect <render_target>

// 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);
```

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

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



Blending: Example Presets

- **Additive Blending**

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

- **Alpha Blending**

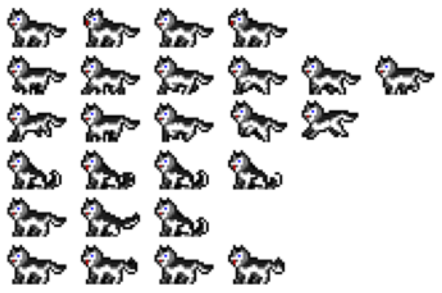
```
//---
// RGB_o = RGB_src * ALPHA_src + RGB_dst * (1 - ALPHA_src)
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

A few examples

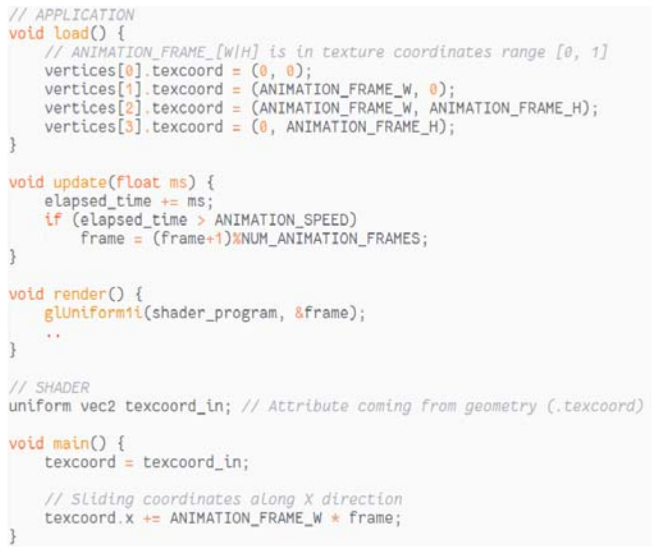
- Sprite Sheets
- Render to Texture
- Particle Systems
- Post-processing Effects: Bloom

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**



Sprite Sheets: Example

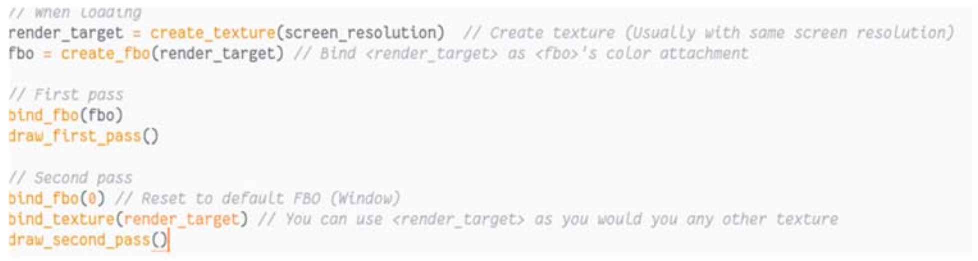


```
// APPLICATION
void load() {
    // ANIMATION_FRAME_[W|H] is in texture coordinates range [0, 1]
    vertices[0].texcoord = (0, 0);
    vertices[1].texcoord = (ANIMATION_FRAME_W, 0);
    vertices[2].texcoord = (ANIMATION_FRAME_W, ANIMATION_FRAME_H);
    vertices[3].texcoord = (0, ANIMATION_FRAME_H);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

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**

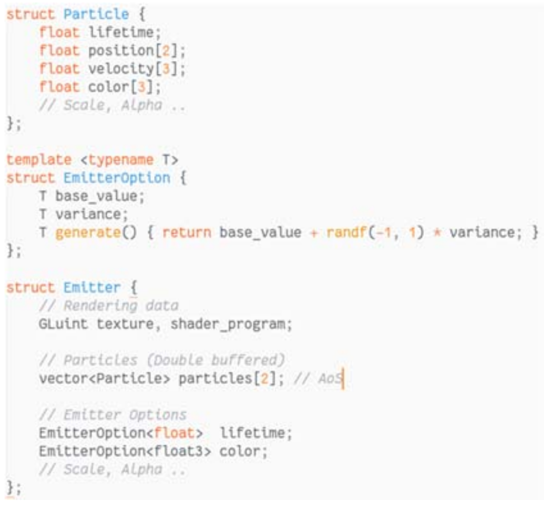


```
// When loading
render_target = create_texture(screen_resolution)  // Create texture (Usually with same screen resolution)
fbo = create_fbo(render_target) // Bind <render_target> as <fbo>'s color attachment

// 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()
```

Basic Particle System

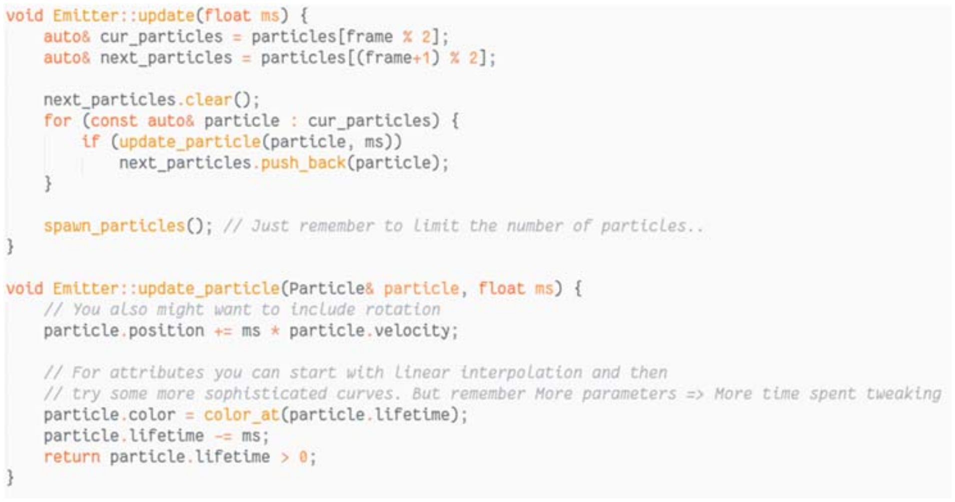
- (Physics-based) simulation of tiny individual particles
- Comp.osed of one or more **Emitters**
- Particles are usually rendered as **Textured Quads**, but any geometry can be used
- Requires lots of tweaking to look right (good)
- Minimum requirements:
  - *"Some" Scalability*
  - *Smart rendering*
  - *Correct blending*

Basic Particle System: Example



```
struct Particle {
    float lifetime;
    float position[2];
    float velocity[3];
    float color[3];
    // Scale, Alpha ..
};

template <typename T>
struct EmitterOption {
    T base_value;
    T variance;
    T generate() { return base_value + randf(-1, 1) * variance; }
};

struct Emitter {
    // Rendering data
    GLuint texture, shader_program;

    // Particles (Double buffered)
    vector<Particle> particles[2]; // AoS

    // Emitter Options
    EmitterOption<float>  lifetime;
    EmitterOption<float3> color;
    // Scale, Alpha ..
};
```

Basic Particle System: Example



```
void Emitter::update(float ms) {
    auto& cur_particles = particles[frame % 2];
    auto& next_particles = particles[(frame+1) % 2];

    next_particles.clear();
    for (const auto& particle : cur_particles) {
        if (update_particle(particle, ms))
            next_particles.push_back(particle);
    }

    spawn_particles(); // Just remember to limit the number of particles..
}

void Emitter::update_particle(Particle& particle, float ms) {
    // You also might want to include rotation
    particle.position += ms * particle.velocity;

    // For attributes you can start with linear interpolation and then
    // try some more sophisticated curves. But remember More parameters => More time spent tweaking
    particle.color = color_at(particle.lifetime);
    particle.lifetime -= ms;
    return particle.lifetime > 0;
}
```

Basic Particle System: Rendering

- **On Load**:

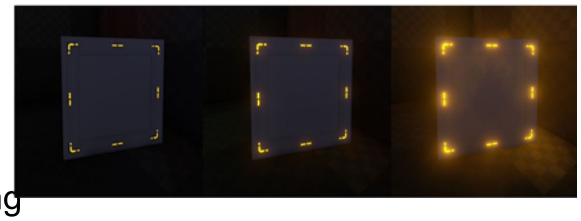
Create a **VBO** with as many entries as the maximum number of particles:
Vertex Attributes: Position, Texcoord, (Index), ..

- **On Render**:

Bind Texture(s) used by the emitter
Update Uniforms (Scale, Color, ..):
    Option: Update material for every particle
    Option: Single buffer for all particle materials
glDrawElements(GL_TRIANGLES, 6 * num_alive_particles, ..)

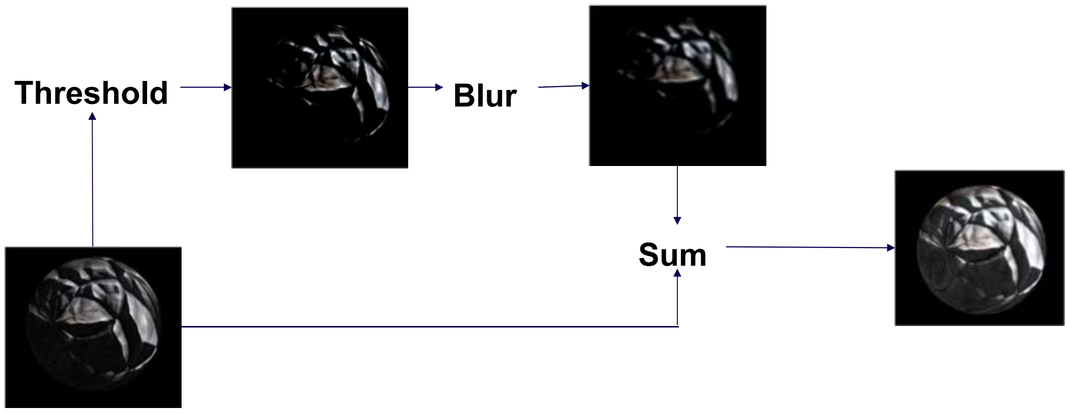
Post-processing: Bloom

- **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);
```