Tutorial 04: Writing a Post-Processing Pass

> 💡 **Tip:** It's recommended using the 04_postprocessing.html version of this tutorial as copying code works best there regarding padding and formatting.

Welcome to the post-processing tutorial! In this guide, you'll learn how to create a render pass that applies screen-space effects to rendered images. You'll implement the **vignette post-processing effect** — a simple but effective screen-edge darkening technique — and integrate it into the engine's rendering pipeline.

What you'll learn:

- Engine render pass architecture and lifecycle
- Multi-pass rendering (scene → post-process → composite)
- Shader registry and pipeline manager patterns
- Bind group caching for efficient resource reuse
- Fullscreen triangle rendering technique
- Integrating new passes into the renderer pipeline

Important Note: This tutorial implements **one hardcoded post-processing effect (vignette)** added to the engine. This is NOT a general-purpose configurable post-processing system. To add more effects (bloom, tone mapping, color grading), you would need to create additional shader and pass classes. A future tutorial could cover building a flexible post-processing framework.

What you'll build:

- Implement `PostProcessingPass` methods one by one
- Integrate into `Renderer::renderToTexture()` to apply vignette after debug rendering
- Add proper initialization and cleanup in `Renderer`

What's provided:

- `PostProcessingPass.h` - Complete header with all method signatures (already implemented)
- `PostProcessingPass.cpp` - Skeleton ready for your implementation
- `postprocess_vignette.wgsl` - Vignette shader (in `resources/`)
- Shader registration - Already set up in `ShaderRegistry.cpp`
- Renderer integration points - Already marked with tutorial comments

Prerequisites:

- Completed Tutorial 01 (shader basics and WebGPU fundamentals)
- Understanding of bind groups (Tutorial 02)
- Familiarity with RenderPass pattern (look at `MeshPass`, `ShadowPass`, `CompositePass`)

Understanding the Architecture

Steps 1-4 (Shadow, Mesh, Debug, PostProcessing) are **per-camera operations**. If multiple cameras, each one:

- Its own shadow maps

- Its own scene render
- Its own debug overlays
- Its own post-processing

Frame Rendering Pipeline:

```
Frame Start
    ↓
PER-CAMERA LOOP (for each camera):
    ├─ [1] Shadow Pass        → Render shadow maps from light perspective
    ├─ [2] Mesh Pass          → Render 3D scene to HDR texture (with
lighting/shadows)
    ├─ [3] Debug Pass         → Render debug overlays (wireframes, gizmos)
    └─ [4] POST-PROCESSING    → Apply vignette effect (darkens edges) ← THIS
TUTORIAL
    ↓
[5] Composite Pass   → Combine all camera render targets into surface texture
    ↓
[6] UI Pass          → Render ImGui on top
    ↓
Present to screen
```

Then **Composite Pass** (step 5) combines all camera results together and renders to the final surface.

Why Post-Process After Debug Pass?

Post-processing should affect the entire rendered image for that camera. If we placed it before debug pass, wireframes and gizmos would not receive the vignette effect. By placing it after debug pass, all rendered content (scene + debug overlays) gets processed together.

Pattern Overview:

All render passes in this engine follow the same lifecycle:

```
// Step 1: Create pass
auto pass = std::make_unique<PostProcessingPass>(context);

// Step 2: Initialize (one-time setup)
pass->initialize();  // Get shader, create sampler, create pipeline

// Step 3: Per-frame render
pass->setInputTexture(inputTex);            // Configure input
pass->setRenderPassContext(context);        // Configure output
pass->render(frameCache);                   // Execute on GPU

// Step 4: Cleanup (on shutdown/resize)
pass->cleanup();
```

Step 1: PostProcessingPass::initialize()

This method performs one-time setup: loading the shader and creating the rendering pipeline.

Overview:

- Get the vignette shader from the shader registry
- Retrieve a sampler from the sampler factory
- Lazy pipeline creation (happens in render())

What to understand:

1. **Shader Registry Pattern** - Instead of loading shaders directly, we use `shaderRegistry().getShader()`. This allows:
   - Centralized shader management
   - Hot-reloading support (shaders can be updated without recompiling)
   - Bind group layout information already parsed from shader
2. **Lazy Pipeline Creation** - The pipeline is created in `render()`, not `initialize()`. This allows:
   - Different output formats for different render targets
   - Pipeline recreation if shader reloads
   - Pattern used by `MeshPass` and `CompositePass`
3. **Sampler Reuse** - We get a pre-made sampler (`getClampLinearSampler()`) instead of creating one. This is more efficient and reuses GPU resources.

Your Task:

Open `src/engine/rendering/PostProcessingPass.cpp` and implement the `initialize()` method:

```
bool PostProcessingPass::initialize()
{
    spdlog::info("Initializing PostProcessingPass");

    // Tutorial 4 - Step 1: Get vignette shader from registry
    // The shader registry was populated in ShaderRegistry.cpp with
createVignetteShader()
    // The shader contains:
    // - Vertex shader (vs_main): Generates fullscreen triangle
    // - Fragment shader (fs_main): Applies vignette darkening
    // - Bind Group 0: Sampler + input texture
    m_shaderInfo = m_context-
>shaderRegistry().getShader(shader::defaults::VIGNETTE);
    if (!m_shaderInfo || !m_shaderInfo->isValid())
    {
        spdlog::error("Vignette shader not found in registry");
        return false;
    }
    // Get a sampler for texture filtering (linear interpolation, clamp-to-edge)
    // This is a pre-made sampler shared across the engine
    m_sampler = m_context->samplerFactory().getClampLinearSampler();

    spdlog::info("PostProcessingPass initialized successfully");
    return true;
}
```

Key Points:

- `shader::defaults::VIGNETTE` is a constant defined in `ShaderRegistry.h` with value `"Vignette_Shader"`
- `m_shaderInfo` contains the shader module AND the bind group layout (parsed from `@group(0)` in WGSL)
- `m_sampler` is used in `getOrCreateBindGroup()` later
- The actual pipeline is created in `render()` method (lazy initialization)

Step 2: PostProcessingPass::setInputTexture()

This method stores which texture to read post-processing input from.

Your Task:

```
void PostProcessingPass::setInputTexture(const
std::shared_ptr<webgpu::WebGPUTexture> &texture)
{
    // Tutorial 4 - Step 2: Store the texture to post-process
    // This is the output of MeshPass/DebugPass (the rendered scene)
    // Called before render() to specify what we're processing
    m_inputTexture = texture;
}
```

Why This Matters:

- Each render target (camera) has its own texture
- Before rendering, we need to tell the pass which texture to sample from
- The texture contains the scene after mesh and debug passes

Step 3: PostProcessingPass::setRenderPassContext()

This method stores where post-processing output should be written.

Your Task:

```
void PostProcessingPass::setRenderPassContext(const
std::shared_ptr<webgpu::WebGPURenderPassContext> &renderPassContext)
```

```
{
    // Tutorial 4 - Step 3: Store where to render output
    // The render pass context defines:
    // - Color attachment (where to draw)
    // - Clear flags (do we clear before rendering?)
    // - Load/store operations
    m_renderPassContext = renderPassContext;
}
```

How This Works:

The render pass context is created in `Renderer::renderToTexture()` with a descriptor specifying:

- Output texture
- Whether to clear
- Load/store operations

Step 4: PostProcessingPass::render() - Part A (Validation & Setup)

This is the main method called each frame. We'll implement it in two parts.

Part A: Validate inputs and create pipeline

```
void PostProcessingPass::render(FrameCache &frameCache)
{
    // Tutorial 4 - Step 4A: Validate inputs
    if (!m_inputTexture || !m_renderPassContext)
    {
        spdlog::warn("PostProcessingPass: Missing input texture or render pass 
context");
        return;
    }

    // Tutorial 4 - Step 4B: Get or create pipeline
    // The pipeline is created lazily (on first use) because:
    // 1. Output format depends on render pass context (determined at runtime)
    // 2. Shader might be hot-reloaded, needing a new pipeline
    // 3. Follows the pattern used by MeshPass
    auto pipeline = m_pipeline.lock();  // Try to get weak_ptr
    if (!pipeline)
    {
        // Create new pipeline with:
        // - Shader info (loaded in initialize())
        // - Output format from render pass (the texture we're writing to)
        // - No depth buffer needed
        // - Triangle list topology (for fullscreen triangle)
        // - No culling (both sides of triangle must be visible)
        m_pipeline = m_context->pipelineManager().getOrCreatePipeline(
            m_shaderInfo,
            m_renderPassContext->getColorTexture(0)->getFormat(),  // Match output
format
            wgpu::TextureFormat::Undefined,  // No depth needed for post-
processing
            Topology::Triangles,
            wgpu::CullMode::None,
            1  // Sample count
        );
        pipeline = m_pipeline.lock();

        if (!pipeline || !pipeline->isValid())
        {
            spdlog::error("PostProcessingPass: Failed to create pipeline");
            return;
        }
    }
```

Key Concepts:

1. **Lazy Pipeline Creation** - The pipeline is created on first use, not in initialize()
2. **Weak Pointer** - We use `std::weak_ptr<WebGPUPipeline>` to avoid circular references
3. **Pipeline Manager** - Handles pipeline caching and hot-reload
4. **Output Format** - We match the output texture's format (from render pass context)

Step 5: PostProcessingPass::render() - Part B (Bind Groups & Drawing)

Part B: Create bind group and draw

```
    // Tutorial 4 - Step 4C: Get or create bind group for input texture
    // Bind groups are cached by texture pointer to avoid recreation each frame
    auto bindGroup = getOrCreateBindGroup(m_inputTexture);
    if (!bindGroup)
    {
        spdlog::warn("PostProcessingPass: Failed to create bind group");
        return;
    }

    // Tutorial 4 - Step 4D: Record and submit draw commands
    // Create command encoder (WebGPU's command recording object)
    auto encoder = m_context->createCommandEncoder("PostProcessingPass Encoder");

    // Begin render pass (starts recording commands into this pass)
    auto renderPass = m_renderPassContext->begin(encoder);

    // Set the pipeline (tells GPU which shader to use)
    renderPass.setPipeline(pipeline->getPipeline());

    // Bind the group 0 (texture + sampler for shader to use)
    renderPass.setBindGroup(0, bindGroup->getBindGroup(), 0, nullptr);

    // Draw 3 vertices (fullscreen triangle)
    // Vertices are generated in shader's vs_main using vertex_index
    renderPass.draw(3, 1, 0, 0);

    // End render pass
    m_renderPassContext->end(renderPass);

    // Submit commands to GPU
    m_context->submitCommandEncoder(encoder, "PostProcessingPass Commands");
}
```

Understanding the Draw Call:

```
renderPass.draw(3, 1, 0, 0);
//              ↓  ↓ ↓ ↓
//              │  │ │ └─ First instance
//              │  │ └──── First vertex index
//              │  └────── Instance count (1 instance)
//              └───────── Vertex count (3 vertices for triangle)
```

The shader generates coordinates based on `@builtin(vertex_index)`:

- Vertex 0 → (-1, -1) bottom-left
- Vertex 1 → (3, -1) bottom-right (off-screen)
- Vertex 2 → (-1, 3) top-left (off-screen)

These form a large triangle covering the entire screen.

Step 6: PostProcessingPass::getOrCreateBindGroup() - Part A (Cache & Layout)

This method creates or reuses bind groups that bind the texture + sampler to shader bindings.

Part A: Check cache and get layout

```
std::shared_ptr<webgpu::WebGPUBindGroup> PostProcessingPass::getOrCreateBindGroup(
    const std::shared_ptr<webgpu::WebGPUTexture> &texture,
    int layerIndex
)
{
    // Tutorial 4 - Step 5A: Validate input
    if (!texture)
        return nullptr;

    // Tutorial 4 - Step 5B: Create cache key
    // We cache bind groups by texture pointer + layer index
    // This avoids recreating the same bind group multiple times per frame
    auto cacheKey = std::make_pair(reinterpret_cast<uint64_t>(texture.get()),
layerIndex);
```

```
    // Tutorial 4 - Step 5C: Check if bind group already cached
    auto it = m_bindGroupCache.find(cacheKey);
    if (it != m_bindGroupCache.end())
        return it->second;  // Reuse cached bind group

    // Tutorial 4 - Step 5D: Get bind group layout from shader
    // The shader info was loaded in initialize()
    // The layout describes what resources Group 0 expects:
    // - Binding 0: sampler (for texture filtering)
    // - Binding 1: texture (the input scene)
    auto bindGroupLayout = m_shaderInfo->getBindGroupLayout(0);
    if (!bindGroupLayout)
        return nullptr;
```

Why Caching?

Without caching, we'd create a new bind group every frame for the same texture, wasting GPU memory and CPU time. By caching, we only create it once per unique texture.

Step 7: PostProcessingPass::getOrCreateBindGroup() - Part B (Create & Cache)

Part B: Create new bind group

```
    // Tutorial 4 - Step 5E: Create bind group entries matching shader layout
    // The shader defines:
    // @group(0) @binding(0) var inputSampler: sampler;
    // @group(0) @binding(1) var inputTexture: texture_2d<f32>;

    // We create entries in the same order:
    std::vector<wgpu::BindGroupEntry> entries = {
        WGPUBindGroupEntry{nullptr, 0, nullptr, 0, 0, m_sampler, nullptr},
        // binding=0, sampler resource

        WGPUBindGroupEntry{nullptr, 1, nullptr, 0, 0, nullptr, texture-
>getTextureView(layerIndex)},
        // binding=1, texture resource
    };

    // Tutorial 4 - Step 5F: Create WebGPU bind group descriptor
    wgpu::BindGroupDescriptor desc = {};
    desc.layout = bindGroupLayout->getLayout();
    desc.entryCount = static_cast<uint32_t>(entries.size());
    desc.entries = entries.data();

    // Tutorial 4 - Step 5G: Create raw WebGPU bind group
    auto bindGroupRaw = m_context->getDevice().createBindGroup(desc);

    // Tutorial 4 - Step 5H: Wrap in engine's bind group object
    // WebGPUBindGroup is a wrapper that tracks resources and provides utilities
    auto bindGroup = std::make_shared<webgpu::WebGPUBindGroup>(
        bindGroupRaw,
        bindGroupLayout,
        std::vector<std::shared_ptr<webgpu::WebGPUBuffer>>{}  // No buffers in
this bind group
    );

    // Tutorial 4 - Step 5I: Cache for future use
    m_bindGroupCache[cacheKey] = bindGroup;
    return bindGroup;
}
```

Binding Details:

The WGPUBindGroupEntry structure is initialized as:

```
WGPUBindGroupEntry{
    nullptr,           // nextInChain (for extensions)
    binding_index,     // which @binding(N) this is for
    nullptr,           // buffer (null since this is for sampler/texture)
    0,                 // offset (unused for sampler/texture)
    0,                 // size (unused for sampler/texture)
    sampler_or_null,   // sampler resource (if this binding is a sampler)
    textureview_or_null// texture resource (if this binding is a texture)
}
```

Step 8: PostProcessingPass::cleanup()

This method releases cached resources.

Your Task:

```
void PostProcessingPass::cleanup()
{
    // Tutorial 4 - Step 6: Release bind group cache
    // Called on shutdown or window resize
    // The pipeline and sampler are managed by pipeline manager and sampler
factory
    // We only need to clear bind group cache
    m_bindGroupCache.clear();
}
```

Why Only Clear Cache?

- `m_shaderInfo` - Managed by shader registry
- `m_sampler` - Managed by sampler factory (shared resource)

- `m_pipeline` - Managed by pipeline manager (weak_ptr, auto-cleans)
- `m_inputTexture` - Managed by caller (Renderer)
- `m_renderPassContext` - Managed by caller (Renderer)
- `m_bindGroupCache` - **We** own this, so we must clean it

Step 9: Renderer::initialize() - Add PostProcessingPass Setup

Now integrate the pass into the renderer.

Your Task:

Open `src/engine/rendering/Renderer.cpp` and find the initialize() method. Look for the comment:

```
// Tutorial 4 - Step xx: Initialize PostProcessingPass here
```

Add this code:

```
    // Tutorial 4 - Step 9: Initialize PostProcessingPass
    m_postProcessingPass = std::make_unique<PostProcessingPass>(m_context);
    if (!m_postProcessingPass->initialize())
    {
        spdlog::error("Failed to initialize PostProcessingPass");
        return false;
    }
```

What's Happening:

During initialization, we create the PostProcessingPass object and call its `initialize()` method. This one-time setup:

- Loads the vignette shader from the shader registry
- Gets a pre-made sampler for texture filtering
- Prepares the pass for rendering (actual rendering happens in the render phase, not initialization)

Step 10: Prepare Post-Processing Texture

Before we can render to a post-process texture, we need to create it.

Location: In `Renderer::renderToTexture()`, find the comment:

```
// STEP 2: Prepare Depth Buffer and Post-Processing Texture
```

This code should already exist (the engine creates it for you):

```
    // Post-processing texture is an intermediate render target for effects like
bloom, tone mapping, etc.
    if (!m_postProcessTextures[renderTargetId])
    {
        m_postProcessTextures[renderTargetId] = m_context-
>textureFactory().createRenderTarget(
            -renderTargetId - 1, // Use negative ID to differentiate post-process
textures from main render targets
            renderFromTexture->getWidth(),
            renderFromTexture->getHeight(),
            renderFromTexture->getFormat() // match format of main render target
        );
    }
```

What This Does:

- Creates an intermediate texture with the same dimensions and format as the main render target
- Uses negative ID (`-renderTargetId - 1`) to distinguish post-process textures from main render targets
- Only creates the texture once; reuses it for subsequent frames

---

Step 11: Renderer::renderToTexture() - Call PostProcessingPass

This is where post-processing actually executes each frame.

Location: Find this comment in `renderToTexture()`:

```
// Tutorial 4 - Step 10: Apply vignette effect
```

Your Task:

Uncomment and implement this code after the Debug Pass section:

```
    // ========================================
    // STEP 7: Post-Processing Pass
    // ========================================
    // Tutorial 4 - Step 10: Apply vignette effect
    // Texture swapping: MeshPass/DebugPass output → input for post-processing
    // Output: Post-processed image (stored in m_postProcessTextures for
CompositePass)

    renderFromTexture = renderToTexture; // Post-processing reads from the main
render target
    renderToTexture = m_postProcessTextures[renderTargetId];
    auto postProcessingContext = m_context->renderPassFactory().create(
        renderToTexture,  // Color attachment (post-process output)
        nullptr,          // No depth attachment (use existing depth)
```

```
            ClearFlags::None, // Don't clear anything
            renderTarget.backgroundColor
        );

        m_postProcessingPass->setCameraId(renderTargetId);
        m_postProcessingPass->setInputTexture(renderFromTexture);
        m_postProcessingPass->setRenderPassContext(postProcessingContext);
        m_postProcessingPass->render(m_frameCache);
```

How It Works:

1. **Texture Swapping:**
   - `renderFromTexture = renderToTexture` - Save the current render target (scene + debug)
   - `renderToTexture = m_postProcessTextures[renderTargetId]` - Switch output to an intermediate post-process texture
2. **Rendering:**
   - Create a render pass that outputs to the post-process texture
   - `setInputTexture(renderFromTexture)` - Tell post-processing to READ from the scene texture
   - `setRenderPassContext(postProcessingContext)` - Tell post-processing to WRITE to the intermediate texture
   - `render()` - Execute the vignette shader
3. **Result:**
   - Input: Scene + debug overlays (from MeshPass + DebugPass)
   - Processing: Vignette shader darkens the edges
   - Output: Post-processed image in `m_postProcessTextures[renderTargetId]`
   - Next step: CompositePass will use this post-processed texture

Why Separate Textures?

Using intermediate textures allows:

- Read and write to different textures (required by WebGPU)
- Chain multiple post-processing effects
- Keep original scene data for debugging
- Proper texture synchronization between passes

Step 12: Renderer::onResize() - Handle Post-Processing Texture Resize

When the window is resized, all textures need to be updated to match the new dimensions.

Location: In `Renderer::onResize()`, this code should already exist:

```
void Renderer::onResize(uint32_t width, uint32_t height)
{
    for (auto &[id, target] : m_renderTargets)
    {
        // ... existing resize code ...

        // Resize post-processing texture
        auto postProcessingTexture = m_postProcessTextures[id];
        if (postProcessingTexture)
            postProcessingTexture->resize(*m_context, viewPortWidth,
viewPortHeight);
    }

    // ... cleanup passes ...

    if (m_postProcessingPass)
        m_postProcessingPass->cleanup();  // Clear bind group cache and reset

    spdlog::info("Renderer resized to {}x{}", width, height);
}
```

What This Does:

- Resizes the post-processing texture to match the new window dimensions
- Calls `cleanup()` on PostProcessingPass to clear the bind group cache
    - The cached bind groups are tied to the old texture dimensions
    - Next frame, new bind groups will be created with the correct dimensions

---

Summary: Complete Post-Processing Pipeline

After implementing all steps, here's the complete flow each frame:

```
Frame Setup
  └─ For each camera:
       └─ renderToTexture(camera)
            ├─ STEP 1: Create/update main render target texture
            │
            ├─ STEP 2: Create/update depth buffer
            │          Create/update post-process texture  ← Step 10
            │
            ├─ STEP 3-6: Culling, GPU prep, Mesh Pass, Debug Pass
            │
            ├─ STEP 7: Post-Processing Pass  ← Step 11
            │   ├─ setInputTexture(scene + debug output)
            │   ├─ setRenderPassContext(post-process texture)
            │   └─ render() → Apply vignette effect
            │
            ├─ STEP 8: Store final texture for compositing
            │
            └─ Window Resize: onResize() → Resize all textures  ← Step 12
```

No action needed - this is automatic.

Summary: Complete Flow

Here's what happens each frame:

```
// Frame setup (Renderer::renderFrame)
  └─ For each camera:
       └─ Renderer::renderToTexture(camera)
            ├─ MeshPass::render()          // Renders 3D scene
            │   └─ Output: renderTarget.gpuTexture with lit scene
            │
            ├─ DebugPass::render()         // Renders wireframes, gizmos
            │   └─ Output: Same texture, with debug overlays added
            │
            ├─ PostProcessingPass::render() // ← YOU ADDED THIS!
            │   ├─ setInputTexture(gpuTexture)
            │   ├─ setRenderPassContext(renderPassContext)
            │   └─ render() {
            │       - Get pipeline
            │       - Create bind group (texture + sampler)
            │       - Draw 3 vertices (fullscreen triangle)
            │       - Vignette shader darkens edges
            │     }
            │   └─ Output: Same texture, but with vignette effect
            │
            └─ CompositePass::render()     // Copies to surface
                └─ Output: Final image on screen
```

Building and Testing

Build the Engine

```
# Windows
scripts\build.bat Debug WGPU

# Linux
bash scripts/build.sh Debug WGPU
```

Run a Tutorial Example

Any example that renders a 3D scene will show the vignette effect:

```
examples/build/main_demo/Windows/Debug/MainDemo.exe
examples/build/tutorial/Windows/Debug/Tutorial.exe
```

Expected Result

You should see: ✅ **Vignette Effect** - Screen edges are darker, center is brighter ✅ **Smooth Falloff** - Gradual transition from center to edges ✅ **Applied to Everything** - Both 3D scene and debug overlays affected ✅ **Screen-Space** - Effect doesn't rotate with camera movement

Visual Test

- **Move camera around** - Vignette stays screen-aligned (doesn't follow camera)
- **Look at bright areas** - Center remains visible despite vignette
- **Look at edges** - Edges are noticeably darker

Understanding the Vignette Shader

The vignette effect happens in `resources/postprocess_vignette.wgsl`:

Vertex Shader (`vs_main`):

```
// Generate fullscreen triangle without vertex buffers
let x = f32((vertexIndex << 1u) & 2u);  // 0, 2, 0
let y = f32(vertexIndex & 2u);          // 0, 0, 2
// Converts to NDC: (-1,-1), (3,-1), (-1,3)
// This covers entire screen
```

Fragment Shader (`fs_main`):

```
// Calculate distance from center
let dist = distance(input.texCoord, vec2f(0.5, 0.5));

// Smooth falloff: 1.0 at center, 0.0 at edges
let vignette = 1.0 - smoothstep(0.0, 1.0, dist * vignetteFalloff);

// Mix between darkened edges (0.5) and normal brightness (1.0)
let vignetteFactor = mix(1.0 - vignetteIntensity, 1.0, vignette);

// Apply effect by multiplying color
finalColor = sceneColor * vignetteFactor;
```

Key Concepts Learned

✅ **Render Pass Lifecycle** - initialize() → render() → cleanup() ✅ **Shader Registry** - Centralized shader management with hot-reload support ✅ **Pipeline Manager** - Lazy pipeline creation with caching ✅ **Bind Group Caching** - Efficient resource reuse across frames ✅ **Fullscreen Triangle** - Procedural vertex generation in shader ✅ **Multi-Pass Rendering** - Reading from previous pass output ✅ **Screen-Space Effects** - Operations in normalized screen coordinates

---

Common Issues & Debugging

Black screen or no effect visible:

- Check `postprocess_vignette.wgsl` exists in `resources/`
- Verify shader registry has `createVignetteShader()` called
- Check render pass context output format matches

Vignette too strong/weak:

- Adjust parameters in shader:

  ```
  let vignetteIntensity = 0.5;  // Darker edges (0.0=none, 1.0=black)
  let vignetteFalloff = 2.0;    // Transition speed (higher=sharper)
  ```

Compilation errors:

- Verify all includes are correct
- Check WebGPU API usage matches your version
- Look at CompositePass/MeshPass for pattern reference

What's Next?

You've implemented a **hardcoded post-processing effect** that's baked into the engine. To extend this:

1. **Add more effects** - Create new shader + new PostProcessingPass-like class for each effect
2. **Multiple passes** - Chain effects (vignette → bloom → tone mapping)
3. **Effect selection** - Allow runtime effect switching via configuration
4. **Framework design** - Build a flexible system supporting arbitrary effects

Future tutorials could cover these advanced topics!

Reference

- Shader source: `resources/postprocess_vignette.wgsl`
- Pass header: `include/engine/rendering/PostProcessingPass.h`
- Pass implementation: `src/engine/rendering/PostProcessingPass.cpp`
- Renderer integration: `src/engine/rendering/Renderer.cpp`
- Shader registration: `src/engine/rendering/ShaderRegistry.cpp`
- Similar passes: `CompositePass.cpp`, `MeshPass.cpp`, `ShadowPass.cpp`