Tutorial 01: Writing a Custom Unlit Shader

Welcome to your first shader tutorial! In this guide, you'll learn how to write a custom WGSL shader from scratch for the Vienna WebGPU Engine. We'll create a simple unlit shader that displays textured geometry.

What you'll learn:

- Engine bind group requirements (Frame, Object, Material)
- WGSL vertex and fragment shader structure
- Matrix transformations (model → world → view → clip space)
- Texture sampling in shaders
- Material assignment to model submeshes

What's provided:

- Complete C++ setup in `examples/tutorial/main.cpp` (scene, models, lighting)
- Empty shader file at `examples/tutorial/assets/shaders/unlit.wgsl`
- Tutorial project ready to build and run

Step 1: Understanding the Project Structure

File: `examples/tutorial/main.cpp`

The C++ code handles scene setup, shader registration, and material creation. The only missing piece is the material assignment (commented out until the shader is complete).

CPU vs GPU: Who Does What?

This tutorial teaches both sides of rendering:

- **CPU Side (C++)**: Creates resources, records commands, manages data
- **GPU Side (WGSL Shader)**: Executes commands in parallel, transforms geometry, colors pixels

The C++ code prepares everything the GPU needs, but the actual drawing happens on the GPU running your shader code.

Why is material assignment commented out? Without a complete shader file, the engine can't create a **render pipeline** (the GPU program combining your shaders with rendering state). We'll uncomment it in Step 10.

File: `examples/tutorial/assets/shaders/unlit.wgsl`

Currently empty - you'll write this! The engine expects:

- Vertex shader: `vs_main`
- Fragment shader: `fs_main`
- Three bind groups (Frame, Object, Material)

Building the Project

You can build the tutorial at any time to check for errors. Before starting, try building once:

```
# Windows
scripts\build-example.bat tutorial Debug WGPU

# Linux
bash scripts/build-example.sh tutorial Debug WGPU
```

VS Code shortcuts:

- **Build**: Press `Ctrl+Shift+B` → select **"Build Example: Tutorial (Debug)"**
- **Run**: Press `F5` or use the **Run and Debug** panel (`Ctrl+Shift+D`) → select **"Tutorial (Debug)"**

Since the shader file is empty, the build will succeed but the the floor won't render yet (only magenta as its the fallback color) until we assign the material in a later step.

Step 2: Define Vertex Input Structure

Before we write shaders, we need to understand what data flows through them. Let's start with what the vertex shader receives from the mesh.

```
struct VertexInput {
    @location(0) position: vec3f,
    @location(1) normal: vec3f,
    @location(2) texCoord: vec2f,
}
```

What it contains:

- `position` - 3D vertex position in model space (where the vertex is in the model)
- `normal` - Surface normal vector (which way the surface faces, used for lighting)
- `texCoord` - UV coordinates (where on the texture to sample, range 0-1)

Why these attributes? The engine's mesh loader provides these three attributes for every vertex. The `@location(N)` numbers map to the vertex buffer layout defined in C++. While the engine's mesh format includes other properties (tangents, vertex colors), you only need to declare the attributes your shader actually uses. We'll cover shader reflection in detail in Tutorial 03.

Step 3: Define Vertex Output Structure

Now define what the vertex shader outputs (and what the fragment shader receives):

```
struct VertexOutput {
    @builtin(position) position: vec4f,
```

```
    @location(0) texCoord: vec2f,
}
```

What it contains:

- `@builtin(position)` - **Required!** Final clip-space position (tells GPU where to draw)
- `texCoord` - UV coordinates passed through to fragment shader

Why vec4f for position? The GPU needs 4D homogeneous coordinates (x, y, z, w) for perspective-correct rendering. The w component handles perspective division.

Data interpolation: Values with `@location` are automatically interpolated across the triangle. If a triangle has UV (0,0) at one corner and (1,1) at another, pixels in between get smoothly interpolated values.

The Rendering Pipeline Flow:

Your shader participates in WebGPU's three-stage pipeline:

1. **Vertex Stage** → Your `vs_main()` runs once per vertex
2. **Rasterization** → GPU's fixed-function hardware converts triangles to pixels
3. **Fragment Stage** → Your `fs_main()` runs once per pixel

Data flows: Vertex shader outputs → Rasterizer interpolates → Fragment shader inputs. This is why `VertexOutput` has `@location` attributes - they're the bridge between stages.

The Rendering Pipeline Flow:

Your shader participates in WebGPU's three-stage pipeline:

1. **Vertex Stage** → Your `vs_main()` runs once per vertex
2. **Rasterization** → GPU's fixed-function hardware converts triangles to pixels
3. **Fragment Stage** → Your `fs_main()` runs once per pixel

Data flows: Vertex shader outputs → Rasterizer interpolates → Fragment shader inputs. This is why `VertexOutput` has `@location` attributes - they're the bridge between stages.

Understanding Bind Groups

Now let's understand the external data shaders receive via bind groups.

What are bind groups? Bind groups are "data packages" from CPU to GPU containing uniforms, textures, and samplers.

Why three separate groups? Each has different update frequency:

- **Group 0 (Frame)** - Once per frame (camera, time)
- **Group 1 (Object)** - Per object (transform)
- **Group 2 (Material)** - Per material (textures, properties)

This separation enables efficient reuse. When rendering 100 objects with the same material, only Group 1 changes.

⚠ CRITICAL: FrameUniforms MUST be at @group(0)

The engine **requires** `FrameUniforms` at bind group 0. This is not optional - the rendering system expects camera data there. However, if your shader doesn't use any frame uniforms (rare cases like post-processing or utility shaders), you can omit the bind group entirely. Most gameplay shaders need camera transforms, so you'll typically declare this in every shader you write.

Step 4: Declare Bind Group 0 - Frame Uniforms

Required at @group(0) - This must always be bind group 0 in all engine shaders.

Open `examples/tutorial/assets/shaders/unlit.wgsl` and add:

```
struct FrameUniforms {
    viewMatrix: mat4x4<f32>,
    projectionMatrix: mat4x4<f32>,
    viewProjectionMatrix: mat4x4<f32>,
    cameraPosition: vec3<f32>,
    time: f32,
}

@group(0) @binding(0)
var<uniform> frameUniforms: FrameUniforms;
```

What it contains:

- `viewMatrix` - Camera's view matrix (transforms world space to camera space)
- `projectionMatrix` - Projection matrix (applies perspective/orthographic)
- `cameraPosition` - Camera world position (useful for effects like reflections, fog)
- `time` - Time since engine start in seconds (useful for animations)

Why it's needed: Every vertex needs to be transformed from world space to screen space. The Frame bind group provides the camera matrices to do this. Without it, the GPU wouldn't know where to draw objects on screen.

Step 5: Declare Bind Group 1 - Object Uniforms

```
struct ObjectUniforms {
    modelMatrix: mat4x4f,
    normalMatrix: mat4x4f,
}

@group(1) @binding(0)
var<uniform> objectUniforms: ObjectUniforms;
```

What it contains:

- `modelMatrix` - Transforms vertices from model space to world space (position, rotation, scale)
- `normalMatrix` - Correctly transforms normals (handles non-uniform scaling)

Why it's needed: Each object has a different position, rotation, and scale in the scene. This bind group provides the object's transform so vertices can be placed correctly in the world.

Step 6: Declare Bind Group 2 - Material Uniforms

```
struct UnlitMaterialUniforms {
    color: vec4f,
}

@group(2) @binding(0)
var<uniform> unlitMaterialUniforms: UnlitMaterialUniforms;

@group(2) @binding(1)
var textureSampler: sampler;

@group(2) @binding(2)
var baseColorTexture: texture_2d<f32>;
```

What it contains:

- `UnlitMaterialUniforms` - Material properties (color tint in this case)
- `textureSampler` - How to sample the texture (filtering: linear/nearest, wrapping: repeat/clamp)
- `baseColorTexture` - The actual texture image data

Why it's needed: Different materials have different appearances. This bind group provides the material-specific data (textures, colors, properties) that determine how the object looks.

Step 7: Write the Vertex Shader

```
@vertex
fn vs_main(input: VertexInput) -> VertexOutput {
    var output: VertexOutput;

    let worldPos = objectUniforms.modelMatrix * vec4f(input.position, 1.0);
    let viewPos = frameUniforms.viewMatrix * worldPos;
    output.position = frameUniforms.projectionMatrix * viewPos;
    output.texCoord = input.texCoord;

    return output;
}
```

This transforms the vertex position through three matrices (model, view, projection) to get screen coordinates. The `1.0` in `vec4f(input.position, 1.0)` indicates this is a position (not a direction).

What happens when this shader runs:

When you uncomment the material assignment in Step 9, the engine creates a **render pipeline** - a compiled GPU program containing:

- Your vertex and fragment shaders
- Vertex buffer layout (position, normal, UV)
- Depth testing settings (enabled)
- Blend mode (opaque)
- Face culling (back faces)

This pipeline is created once and reused every frame. It's WebGPU's way of "freezing" all rendering state into a single object for performance.

Step 8: Write the Fragment Shader

```
@fragment
fn fs_main(input: VertexOutput) -> @location(0) vec4f {
    let textureColor = textureSample(baseColorTexture, textureSampler,
input.texCoord);
    let finalColor = textureColor * unlitMaterialUniforms.color;
    return finalColor;
}
```

This samples the texture at the UV coordinates and multiplies by the tint color. No lighting calculations - that's why it's "unlit".

Step 9: Assign Material to the Floor

In `examples/tutorial/main.cpp` (line ~127), uncomment:

```
floorModel->getSubmeshes()[0].material = floorMaterial->getHandle();
```

Why `[0]`? The plane.obj has only one submesh. Models with multiple parts (like a car) would have multiple submeshes that can each have different materials.

Note: The shader won't be loaded and validated until runtime. The C++ build only compiles the application - shader errors will appear in the console when you run the program.

Rebuild and Run

```
# Rebuild and run
scripts\build-example.bat tutorial Debug WGPU
examples/build/tutorial/Windows/Debug/Tutorial.exe
```

VS Code shortcuts:

- Press `F5` to build and run with debugger
- Or open **Run and Debug** panel (`Ctrl+Shift+D`) → select **"Tutorial (Debug)"** → click green play button

Expected Result

Now you should see:

- ☑ **Floor** with cobblestone texture (unlit, flat appearance)
- ☑ **Fourareen object** above the floor (uses default PBR shader, has lighting)
- ☑ **Dark background** (background color set to dark gray)

Visual comparison:

- **Floor (your unlit shader)**: Texture appears flat, no shadows or highlights
- **Fourareen (PBR shader)**: Has realistic lighting, shadows, and material response

---

How Your Shader Gets Executed

Before understanding what we built, let's see how your shader actually runs on the GPU:

The Command Recording Model:

WebGPU doesn't execute commands immediately. Instead:

1. **CPU Records Commands** - The engine uses two encoder types:

```
// Step 1: Create command encoder (organizes work into command buffer)
CommandEncoder commandEncoder = device.createCommandEncoder();

// Step 2: Begin render pass (creates RenderPassEncoder for drawing)
RenderPassEncoder renderPass = commandEncoder.beginRenderPass(colorTexture,
depthTexture);

// Step 3: Record drawing commands using RenderPassEncoder
renderPass.setPipeline(yourShaderPipeline);
renderPass.setBindGroup(0, frameBindGroup);  // Camera data
renderPass.setBindGroup(1, objectBindGroup);  // Transform
renderPass.setBindGroup(2, materialBindGroup); // Textures
renderPass.draw(vertexCount);                  // "Draw this mesh!"

// Step 4: End render pass
renderPass.end();

// Step 5: Finish encoding to get command buffer
CommandBuffer commandBuffer = commandEncoder.finish();
```

Key distinction:

- `CommandEncoder` - Top-level container for all GPU work (can create multiple render passes, compute passes, copy operations)
- `RenderPassEncoder` - Specific to drawing operations (setPipeline, setBindGroup, draw)

2. **CPU Submits to GPU** - Command buffer sent to GPU queue:

```
queue.submit(commandBuffer);
```

3. **GPU Executes Asynchronously** - Your shaders run in parallel:

- `vs_main()` runs for all 4 floor vertices simultaneously
- Rasterizer generates ~10,000 pixels for the floor
- `fs_main()` runs for all pixels simultaneously

Render Pass = Drawing Phase:

A render pass defines **what you're drawing to**:

- **Color Attachment**: Texture receiving pixel colors (your screen or a render target)
- **Depth Attachment**: Texture storing depth values (for depth testing)
- **Load/Store Operations**: Clear before drawing? Save result after?

Your fragment shader's `@location(0) vec4f` output goes directly to the color attachment.

Where to see this: Check Renderer.cpp:renderToTexture() - line 250+ shows the full command recording sequence.

Understanding What We Built

You created a complete WebGPU rendering shader with three main components:

1. Data Structures

- `VertexInput` - Maps to vertex buffer attributes via `@location` decorators. WebGPU uses these to know which buffer data goes where (location 0 = position from buffer slot 0, etc.)
- `VertexOutput` - Data passed from vertex to fragment stage. `@builtin(position)` is WebGPU's required clip-space output, `@location` attributes are interpolated by the rasterizer
- Uniform structs - Declared as `var<uniform>` to indicate read-only GPU memory. WebGPU enforces memory alignment rules (vec3f followed by f32 requires padding)

2. Bind Groups (WebGPU's Resource Binding Model)

- `@group(0)` - Frame data: WebGPU lets you update bind groups independently. Group 0 changes once per frame
- `@group(1)` - Object data: Rebound for each object, but Groups 0 stays bound
- `@group(2)` - Material data: Contains uniform buffer (`@binding(0)`), sampler (`@binding(1)`), texture (`@binding(2)`)

This is WebGPU's optimization strategy: group resources by update frequency. When you change an object, only Group 1 rebinds—the others stay cached in GPU state.

3. Shader Pipeline Stages

- `@vertex fn vs_main()` - Vertex stage: Transforms positions to clip-space. WebGPU's render pipeline invokes this once per vertex
- `@fragment fn fs_main()` - Fragment stage: Determines pixel color. WebGPU's rasterizer interpolates vertex outputs across primitives, then invokes this per fragment

Between stages, WebGPU's fixed-function rasterizer converts clip-space triangles to screen-space fragments, performing depth testing and face culling.

How a Draw Call Works:

When the engine issues `encoder.draw(vertexCount)`, the GPU:

1. **Fetches Vertices**: Reads position/normal/UV from vertex buffer using the pipeline's vertex layout
2. **Runs Vertex Shader**: Executes `vs_main()` in parallel for all vertices (4 for our plane)
3. **Assembles Primitives**: Groups vertices into triangles (2 triangles for our plane)
4. **Rasterizes**: Converts triangles to fragments (pixels), discarding anything outside screen or behind other geometry
5. **Runs Fragment Shader**: Executes `fs_main()` in parallel for all visible pixels
6. **Outputs to Attachments**: Writes colors to render target, depth values to depth buffer

All bind groups must be set before the draw call - that's why Groups 0, 1, 2 are bound in Renderer.cpp before calling `draw()`.

Performance Insight:

The floor has:

- 4 vertices \u2192 `vs_main()` runs 4 times
- ~10,000 visible pixels \u2192 `fs_main()` runs ~10,000 times

This is why fragment shaders are performance-critical - they run far more often than vertex shaders!

WebGPU Concepts:

- **Render Pipeline** - Combines shaders, vertex layout, blend state, depth state into a single GPU program
- **Bind Groups** - Resource descriptors (buffers, textures, samplers) bound to `@group(N)` slots in shaders
- **Command Encoding** - CPU records GPU commands (set pipeline, bind groups, draw), then submits batch to GPU queue

Where to Find These Concepts in the Engine:

If you want to understand how the engine implements these WebGPU features:

- **Bind Group Creation** → WebGPUBindGroupFactory.cpp - See how `@group` and `@binding` map to WebGPU resources
- **Pipeline Building** → WebGPUPipelineFactory.cpp - See how shaders, vertex layouts, and state combine into pipelines

- **Vertex Layout Definition** → WebGPUPipelineManager.cpp - Look for `VertexBufferLayout` with attribute formats and offsets
- **Shader Compilation** → WebGPUShaderFactory.cpp - See WGSL → shader module → bind group layout extraction
- **Bind Group Binding Logic** → BindGroupBinder.cpp - See reuse policies and automatic rebinding
- **Frame Rendering Loop** → Renderer.cpp - See the complete flow from `renderFrame()` through all passes

Experiments to Try

Now that your shader works, try these quick modifications:

1. Change the tint color - In `main.cpp`:

```
unlitProperties.color = glm::vec4(1.0f, 0.5f, 0.5f, 1.0f);  // Red tint
```

**2. Texture Tiling - In `unlit.wgsl` fragment shader:

```
let tiledUV = input.texCoord * 4.0; // Tile 4x4
let textureColor = textureSample(baseColorTexture, textureSampler, tiledUV);
```

Will result in smaller stone tiles.

2. Animate with time - In `unlit.wgsl` fragment shader:

```
let scrolledUV = input.texCoord + vec2f(frameUniforms.time * 0.1, 0.0);
let textureColor = textureSample(baseColorTexture, textureSampler, scrolledUV);
```

Troubleshooting

Shader compilation failed

- Check semicolons, struct syntax, and WGSL types
- Verify bind group numbers are 0, 1, 2
- Check entry point names match `vs_main` and `fs_main`

Floor appears black/white

- Ensure texture file exists in `resources/`
- Check material assignment line is uncommented

Floor doesn't render

- Verify material assignment is uncommented (Step 10)
- Check shader name matches in registration and material creation

Key Takeaways

- ☑ **Bind Groups** - Engine provides Frame (0), Object (1), Material (2)
- ☑ **Vertex Shader** - Transforms vertices through matrix pipeline
- ☑ **Fragment Shader** - Samples textures and outputs color
- ☑ **Material Assignment** - Connect materials to model submeshes
- ☑ **WGSL Syntax** - Strict typing with `@group/@binding/@location`

What's Next?

In **Tutorial 02**, you'll learn:

- Creating custom bind groups beyond the standard Frame/Object/Material
- Registering custom bind groups in shader reflection
- Implementing `preRender()` to provide per-object data
- Using `BindGroupDataProvider` to send custom data to GPU
- Extending `ModelRenderNode` to add custom shader parameters
- Practical example: Texture tiling and offset control per object

Next Tutorial: 02_custom_bindgroup.md / 02_custom_bindgroup.pdf

Further Reading

- WebGPU WGSL Specification
- Engine Bind Group System
- Getting Started Guide
- LearnWebGPU Tutorial