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).

Why is material assignment commented out? Without a complete shader file, the engine can't create a render pipeline. 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.

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: mat4x4f,
    projectionMatrix: mat4x4f,
    cameraPosition: vec3f,
    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).

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

---

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. 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:

- Writing custom shaders with advanced features
- Using time-based animations in shaders
- Working with multiple textures
- Creating visual effects (scrolling, tiling, distortion)

Next Tutorial: 02_custom_bindgroup.md

Further Reading

- WebGPU WGSL Specification
- Engine Bind Group System
- Getting Started Guide
- LearnWebGPU Tutorial modelMatrix: mat4x4f, // Model-to-world transform normalMatrix: mat4x4f, // Normal transformation matrix }

@group(1) @binding(0) var objectUniforms: ObjectUniforms;

```
**Usage:**
- Transform vertices from model space to world space
- Transform normals correctly (using normalMatrix)

### Bind Group 2: MaterialUniforms (Provided by material system)

The engine's material system provides textures and properties:
```

```wgsl
@group(2) @binding(0) var materialTexture: texture_2d<f32>;
@group(2) @binding(1) var materialSampler: sampler;
```

Note: For this tutorial, we'll use the default material's diffuse texture.

Step 1: Write the Vertex Shader

Open `examples/tutorial/assets/unlit.wgsl` and let's start with the vertex shader.

Define Bind Groups

First, declare the bind groups we'll use:

```
// ============================================================================
// Bind Group 0: Frame Uniforms (Engine-provided, required)
// ============================================================================
struct FrameUniforms {
    viewMatrix: mat4x4f,
    projectionMatrix: mat4x4f,
    cameraPosition: vec3f,
    time: f32,
}

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

// ============================================================================
// Bind Group 1: Object Uniforms (Engine-provided for model rendering)
// ============================================================================
struct ObjectUniforms {
    modelMatrix: mat4x4f,
    normalMatrix: mat4x4f,
}

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

// ============================================================================
// Bind Group 2: Material Texture (Engine material system)
// ============================================================================
@group(2) @binding(0)
var materialTexture: texture_2d<f32>;

@group(2) @binding(1)
var materialSampler: sampler;
```

Define Vertex Input/Output

The engine provides vertex data in a specific format:

```
// ============================================================================
// Vertex Shader Input/Output
// ============================================================================
struct VertexInput {
    @location(0) position: vec3f,    // Vertex position in model space
    @location(1) normal: vec3f,      // Vertex normal in model space
    @location(2) texCoord: vec2f,    // UV texture coordinates
}

struct VertexOutput {
    @builtin(position) position: vec4f,  // Clip-space position (required)
    @location(0) worldPosition: vec3f,   // World-space position (for fragment)
    @location(1) texCoord: vec2f,        // UV coordinates (for fragment)
}
```

Key Points:

- `@location(N)` maps to vertex buffer attributes
- `@builtin(position)` is required - tells GPU where to draw the vertex
- Output locations pass data to fragment shader

Implement Vertex Shader

Now write the transformation logic:

```
// ============================================================================
// Vertex Shader
// ============================================================================
@vertex
fn vertexMain(input: VertexInput) -> VertexOutput {
    var output: VertexOutput;

    // Transform vertex position: Model Space → World Space → View Space → Clip
Space
    let worldPos = objectUniforms.modelMatrix * vec4f(input.position, 1.0);
    output.worldPosition = worldPos.xyz;

    // Apply camera transformation
    let viewPos = frameUniforms.viewMatrix * worldPos;
    output.position = frameUniforms.projectionMatrix * viewPos;

    // Pass through texture coordinates
    output.texCoord = input.texCoord;

    return output;
}
```

Transformation Pipeline:

1. **Model → World:** `modelMatrix * position` transforms from model's local space to world space
2. **World → View:** `viewMatrix * worldPos` transforms relative to camera
3. **View → Clip:** `projectionMatrix * viewPos` applies perspective and prepares for screen mapping
4. **Pass Data:** UV coordinates are passed unchanged to fragment shader

---

Step 2: Write the Fragment Shader

The fragment shader determines the color of each pixel.

Implement Unlit Fragment Shader

```
// ============================================================================
// Fragment Shader
// ============================================================================
@fragment
fn fragmentMain(input: VertexOutput) -> @location(0) vec4f {
    // Sample the diffuse texture at the given UV coordinates
    let textureColor = textureSample(materialTexture, materialSampler,
input.texCoord);

    // Return the texture color directly (unlit - no lighting calculations)
    return textureColor;
}
```

What's happening:

- `textureSample()` reads color from texture at UV coordinates
- We return the color directly without any lighting calculations
- This creates an "unlit" look - the object shows its texture but doesn't respond to lights

---

Complete Shader Code

Here's the complete `unlit.wgsl` shader:

```
// ============================================================================
// Tutorial 01: Unlit Shader
// A simple shader that displays textured geometry without lighting
// ============================================================================

// ============================================================================
// Bind Group 0: Frame Uniforms (Engine-provided, required)
// ============================================================================
struct FrameUniforms {
    viewMatrix: mat4x4f,
    projectionMatrix: mat4x4f,
```

```
    cameraPosition: vec3f,
    time: f32,
}

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

// ================================================================================
// Bind Group 1: Object Uniforms (Engine-provided for model rendering)
// ================================================================================
struct ObjectUniforms {
    modelMatrix: mat4x4f,
    normalMatrix: mat4x4f,
}

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

// ================================================================================
// Bind Group 2: Material Texture (Engine material system)
// ================================================================================
@group(2) @binding(0)
var materialTexture: texture_2d<f32>;

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

// ================================================================================
// Vertex Shader Input/Output
// ================================================================================
struct VertexInput {
    @location(0) position: vec3f,
    @location(1) normal: vec3f,
    @location(2) texCoord: vec2f,
}

struct VertexOutput {
    @builtin(position) position: vec4f,
    @location(0) worldPosition: vec3f,
    @location(1) texCoord: vec2f,
}

// ================================================================================
// Vertex Shader
// ================================================================================
@vertex
fn vertexMain(input: VertexInput) -> VertexOutput {
    var output: VertexOutput;

    // Transform vertex position: Model Space → World Space → View Space → Clip
Space
    let worldPos = objectUniforms.modelMatrix * vec4f(input.position, 1.0);
    output.worldPosition = worldPos.xyz;
```

```
    // Apply camera transformation
    let viewPos = frameUniforms.viewMatrix * worldPos;
    output.position = frameUniforms.projectionMatrix * viewPos;

    // Pass through texture coordinates
    output.texCoord = input.texCoord;

    return output;
}

// ================================================================================
// Fragment Shader
// ================================================================================
@fragment
fn fragmentMain(input: VertexOutput) -> @location(0) vec4f {
    // Sample the diffuse texture at the given UV coordinates
    let textureColor = textureSample(materialTexture, materialSampler,
input.texCoord);

    // Return the texture color directly (unlit - no lighting calculations)
    return textureColor;
}
```

Step 3: Understanding What We Built

Let's break down the data flow:

Vertex Processing

```
Mesh Data (CPU)
    ↓
Vertex Input (position, normal, texCoord)
    ↓
Model Matrix (ObjectUniforms) → World Space
    ↓
View Matrix (FrameUniforms) → Camera Space
    ↓
Projection Matrix (FrameUniforms) → Clip Space
    ↓
Vertex Output (position, worldPosition, texCoord)
```

Fragment Processing

```
Vertex Output (interpolated)
    ↓
Sample Texture at UV coordinates
    ↓
Fragment Color (RGBA)
```

```
    ↓
Screen Pixel
```

Why "Unlit"?

Compare with a lit shader:

```
// Lit shader (simplified)
@fragment
fn fragmentMain(input: VertexOutput) -> @location(0) vec4f {
    let textureColor = textureSample(materialTexture, materialSampler,
input.texCoord);

    // Calculate lighting (diffuse, specular, etc.)
    let normal = normalize(input.worldNormal);
    let lightDir = normalize(lightPosition - input.worldPosition);
    let diffuse = max(dot(normal, lightDir), 0.0);

    // Apply lighting to texture color
    return textureColor * diffuse * lightColor;
}
```

Our unlit shader skips all lighting calculations - it just shows the raw texture color.

Next Steps

In the next tutorial, we'll learn how to:

1. **Register this shader** with the engine's shader system
2. **Create a custom pipeline** that uses our shader
3. **Use shader reflection** to automatically discover bind groups
4. **Assign the shader** to specific render nodes

For now, the shader is complete and ready to use!

Common Patterns

Texture Tiling

Multiply UV coordinates to repeat the texture:

```
let tiledUV = input.texCoord * 2.0;  // Tile 2x2
let textureColor = textureSample(materialTexture, materialSampler, tiledUV);
```

Texture Scrolling

Animate texture using time:

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

Vertex Color Tinting

Add color variation per vertex:

```
// In VertexInput, add:
@location(3) color: vec3f,

// In fragment shader:
let textureColor = textureSample(materialTexture, materialSampler,
input.texCoord);
return vec4f(textureColor.rgb * input.color, textureColor.a);
```

Key Takeaways

- ☑ **Bind Group 0 (FrameUniforms)** - Always required, provides camera and time
- ☑ **Bind Group 1 (ObjectUniforms)** - Required for 3D models, provides transforms
- ☑ **Bind Group 2 (MaterialUniforms)** - Provided by material system, contains textures
- ☑ **Vertex Shader** - Transforms vertices through model → world → view → clip spaces
- ☑ **Fragment Shader** - Determines pixel color (texture sampling, lighting, effects)
- ☑ **Unlit Rendering** - No lighting calculations, just raw texture/color output

---

Troubleshooting

Shader Compilation Errors

"unknown identifier 'frameUniforms'"

- Check that bind group is declared with `@group(0) @binding(0)`
- Ensure struct and variable names match exactly

"'textureSample' requires sampler to be filtered"

- Verify materialSampler is declared with correct type
- Check binding numbers match (texture=0, sampler=1)

Visual Issues

Black screen

- Verify bind group indices (0, 1, 2)
- Check texture is loaded and bound correctly

- Ensure matrices are not zero/identity incorrectly

Stretched/distorted textures

- Check UV coordinates are being passed correctly
- Verify model has valid UV data (check in Blender/modeling tool)

Model not visible

- Verify transformation matrices are correct
- Check camera position and look direction
- Ensure model is within camera's view frustum

Further Reading

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

Next Tutorial: 02_shader_registration.md - Register and use your custom shader in the engine