Tutorial 02: Custom Bind Groups for Per-Object Data

> 💡 **Tip:** It's recommended using the 02_custom_bindgroup.html version of this tutorial as copying code works best there regarding padding and formatting.

> ⚠ **Build issues?** See Troubleshooting at the end of this tutorial for help reading build errors from the terminal.

In Tutorial 01, you learned the three standard bind groups (Frame, Object, Material) that the engine provides. Now you'll learn how to add your own custom bind groups to pass additional data to shaders.

What you'll learn:

- Creating custom bind groups beyond Frame/Object/Material
- Registering custom bind groups in shader reflection
- Implementing `preRender()` to provide per-frame data
- Using `BindGroupDataProvider` to send data to GPU
- Texture tiling and offset manipulation in shaders

What you'll build: A tiled floor with controllable tiling and offset, perfect for scrolling textures or adjusting texture scale per object.

What's provided:

- Working unlit shader from Tutorial 01
- Tutorial project with scene setup
- Empty `CustomRenderNode.h` class to implement

Understanding Custom Bind Groups

Why add custom bind groups?

The engine's standard bind groups cover common cases:

- Group 0: Camera data (same for all objects)
- Group 1: Transform data (unique per object)
- Group 2: Material data (shared by objects with same material)

But what if you want per-object data that's NOT a transform or material? Examples:

- Scrolling water at different speeds
- Individual object animations
- Per-object effects or parameters

Custom bind groups solve this: They let you add Group 3, 4, 5... with your own data structures.

Step 1: Understanding the Project Structure

This tutorial builds directly on Tutorial 01. You should have completed the unlit shader first.

Files You'll Modify:

1. `examples/tutorial/assets/shaders/unlit_custom.wgsl` - Copy unlit.wgsl and extend with TileUniforms
2. `examples/tutorial/main.cpp` - Register new shader with custom bind group
3. `examples/tutorial/CustomRenderNode.h` - Implement preRender() to provide data

Starting Point:

The `main.cpp` has one commented line (similar to Tutorial 01) if it was not modified previously:

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

You'll uncomment this at the end once everything is ready.

Step 2: Open the Shader File

Open `examples/tutorial/assets/shaders/unlit_custom.wgsl`.

This file already contains the complete unlit shader from Tutorial 01:

- VertexInput and VertexOutput structs
- Frame, Object, and Material bind groups (Groups 0-2)
- Vertex shader (vs_main) and fragment shader (fs_main)

You'll extend this by adding a custom bind group (Group 3) for tiling parameters.

Step 3: Add TileUniforms Struct to Shader

Open `examples/tutorial/assets/shaders/unlit_custom.wgsl` and add after `UnlitMaterialUniforms`:

```
struct TileUniforms
{
    tileOffset: vec2f,
    tileSize: vec2f,
}
```

What it contains:

- `tileOffset` - UV offset (for scrolling or shifting texture)
- `tileSize` - UV scale (how many times to repeat texture)

WebGPU alignment: Each `vec2f` is 8 bytes, so total is 16 bytes - perfect for uniform buffer alignment (must be multiple of 16).

Step 4: Declare Custom Bind Group

Add the bind group declaration after Group 2 (Material):

```
@group(3) @binding(0)
var<uniform> tileUniforms: TileUniforms;
```

Important: Group numbers don't have to be sequential - the engine finds bind groups by name, not by order. The only hard requirement is **Group 0 must be FrameUniforms** if used. You could use `@group(5)` or `@group(10)` for custom bind groups; the number just needs to match between shader and registration.

WebGPU bind group slots:

- Group 0: **Must** be FrameUniforms (if your shader needs camera data)
- Groups 1-2: Typically Object, Light, Material, something like this
- Groups 3+: Custom bind groups (any index you choose)

Each group can have multiple bindings (0, 1, 2...) for different resources within that group.

Step 5: Modify Fragment Shader to Use TileUniforms

Update the fragment shader to apply tiling before texture sampling:

```
@fragment
fn fs_main(input: VertexOutput) -> @location(0) vec4f {
    // Apply tiling and offset to UV coordinates
    let tiledUV = input.texCoord * tileUniforms.tileSize +
tileUniforms.tileOffset;

    // Sample texture with modified UVs
    let textureColor = textureSample(baseColorTexture, textureSampler, tiledUV);
    let finalColor = textureColor * unlitMaterialUniforms.color;
    return finalColor;
}
```

What this does:

1. `input.texCoord` - Original UV from mesh (0-1 range)
2. `* tileUniforms.tileSize` - Scale UV (e.g., ×4 = repeat texture 4 times)
3. `+ tileUniforms.tileOffset` - Shift UV (e.g., +0.5 = scroll halfway)
4. Sample texture at modified coordinates

Example values:

- `tileSize = (2, 2)` - Texture repeats 2×2 times across surface
- `tileOffset = (0.5, 0)` - Texture shifted 50% to the right

Step 6: Register Shader with Custom Bind Group

Now open `examples/tutorial/main.cpp` and find the shader registration section (around line 70).

⚠ **Important:** You don't need to copy the entire registration code. Just:

1. Change the shader **file path** to use the new shader
2. Add the custom bind group code at the very end

Change the Shader Path Only:

In the `.begin()` call, change ONE line:

```
PathProvider::getShaders("unlit_custom.wgsl"),  // Change "unlit.wgsl" →
"unlit_custom.wgsl"
```

Everything else in the `.begin()` call stays the same (frame, object bind groups, material, sampler, texture remain).

Add Custom Bind Group at the End:

After the `.addMaterialTexture()` call and **before** `.build()`, add:

```
        .addBindGroup(
            "TileUniforms",                                    // Bind group name
            engine::rendering::BindGroupReuse::PerObject,      // Cache per object
            engine::rendering::BindGroupType::Custom           // Mark as custom
        )
        .addCustomUniform(
            "tileUniforms",                                    // Variable name in
shader
            sizeof(demo::CustomRenderNode::TileUniforms),      // 16 bytes (2 ×
vec2f)
            WGPUShaderStage_Fragment                           // Used in fragment
shader
        )
        .build();

shaderRegistry.registerShader(shaderInfo);
```

What you're telling the engine:

- Group name: `"TileUniforms"` (matches `@group(3)` in shader)
- Uniform struct size: 16 bytes
- Used in fragment shader stage

How the Binding Happens:

> **Note:** The complex binding logic happens in the `build()` method. When you call `.addBindGroup()` and `.addCustomUniform()`, you're specifying exactly how bind groups should be created. The engine then uses this specification to create bind group layouts, allocate GPU buffers, and manage resource binding.
>
> **Advanced Readers:** Check out WebGPUBindGroupFactory.cpp to see how the factories handle bind group creation and resource allocation. The `WebGPUShaderFactory::build()` method orchestrates the whole process.

Why explicit registration?

- **No WGSL Reflection**: WebGPU doesn't provide official shader reflection APIs, and there are no mature third-party tools for WGSL parsing
- **Practical Necessity**: Without automatic reflection, you must explicitly tell the engine what bind groups and uniforms exist
- **Educational**: You see exactly what bind groups exist and their purposes
- **Control**: Fine-grained control over resource allocation and caching
- **Debugging**: Easy to see which bind groups are used by which shaders

---

Step 7: Implement CustomRenderNode

Open `examples/tutorial/CustomRenderNode.h`. The struct is already defined:

```
struct TileUniforms
{
    glm::vec2 tileOffset;
    glm::vec2 tileSize;
};
```

Now implement the `preRender()` method:

```
virtual void preRender(std::vector<engine::rendering::BindGroupDataProvider>
&outProviders) override
{
    auto dataProvider = engine::rendering::BindGroupDataProvider::create(
        "unlit",      // Shader name (must match registration)
        "TileUniforms",      // Bind group name (must match shader
.addBindGroup())
        tileUniforms,        // Uniform data (struct instance)
        engine::rendering::BindGroupReuse::PerObject,  // Cache per object
        getId()              // Instance ID (node's unique ID)
    );
    outProviders.push_back(dataProvider);
}
```

Understanding preRender():

This method is called by the scene graph before rendering each frame. It's your opportunity to provide updated data to the GPU.

BindGroupDataProvider::create() parameters:

1. **Shader name** - Which shader needs this data
2. **Bind group name** - Which bind group in that shader (from `.addBindGroup()`)
3. **Data** - Actual uniform data (will be copied to GPU)
4. **Reuse policy** - When to rebind (PerObject means cache per unique object)
5. **Instance ID** - Unique identifier for caching (use node ID for per-object)

Why PerObject reuse? If you have 10 floor tiles with different tiling, each gets its own cached bind group. The renderer only rebinds when switching between different objects.

Step 8: Create CustomRenderNode Instance

The `main.cpp` already has the floor setup (around line 155). Update it to use CustomRenderNode:

```
// Change from ModelRenderNode to CustomRenderNode
auto floorNode = std::make_shared<demo::CustomRenderNode>(floorModel);
floorNode->tileUniforms.tileOffset = glm::vec2(0.2f, 0.0f);
floorNode->tileUniforms.tileSize = glm::vec2(4.0f, 4.0f);
floorNode->getTransform().setLocalScale(glm::vec3(10.0f, 1.0f, 10.0f));
rootNode->addChild(floorNode);
```

What this does:

- Creates custom node with floor model
- Sets tileOffset to shift texture 20% to the right

- Sets tileSize to 4×4 (repeat texture 16 times total)
- Scales floor physically to 10×10 units

Step 9: Uncomment Material Assignment

Finally, uncomment the material assignment line (around line 150):

```
// Uncomment this line:
floorModel->getSubmeshes()[0].material = floorMaterial->getHandle();
```

Why uncomment now? Now that the shader is complete and registered, the engine can create the render pipeline successfully.

Step 10: Build and Run

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

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

VS Code: Press `F5` to build and run.

Expected Result

You should see:

- ☑ **Floor** with tiled cobblestone (4×4 repetitions)
- ☑ **Texture slightly offset** to the right (20% shift)
- ☑ **Fourareen object** with PBR lighting (unchanged)

Compare to Tutorial 01:

- Tutorial 01: Texture stretched across entire floor (1×1)
- Tutorial 02: Texture repeated 4×4 times with smaller tiles

Understanding the Data Flow

Lifecycle of custom bind group data:

```
1. Scene graph traversal:
   └─ Scene calls preRender() on each node

2. CustomRenderNode::preRender():
   ├─ Reads tileUniforms member variable
   ├─ Creates BindGroupDataProvider with data
   └─ Adds provider to outProviders vector

3. Scene collects all providers:
   └─ Passes them to Renderer::renderFrame()

4. Renderer processes providers:
   ├─ FrameCache::processBindGroupProviders() creates GPU buffers
   ├─ Caches bind group by (shaderName, bindGroupName, instanceId)
   └─ Uploads data to GPU via writeBuffer()

5. During rendering:
   ├─ BindGroupBinder::bind() checks cache
   ├─ Finds cached bind group by instanceId
   └─ Binds to correct (in this case @group(3)) in shader

6. Shader execution:
   └─ Fragment shader reads tileUniforms.tileOffset and tileSize
```

Key insight: You provide data in `preRender()`, the engine handles caching and GPU upload automatically.

Why preRender() Works - The Timing:

Remember WebGPU's command recording model:

1. **Scene Traversal**: Scene calls `preRender()` on all nodes \u2192 Collects bind group providers
2. **GPU Upload**: Engine processes providers \u2192 Creates buffers \u2192 Writes data with `queue.writeBuffer()`
3. **Command Recording**: Engine records draw commands \u2192 Binds groups \u2192 Issues `draw()`
4. **Submission**: Complete command buffer sent to GPU queue
5. **GPU Execution**: Commands execute, reading the fresh data you provided

This is why updating `tileUniforms` in `preRender()` works - the data reaches the GPU before the draw command executes!

Understanding BindGroupReuse Policies

Why PerObject for TileUniforms?

The reuse policy determines when bind groups are cached and rebound:

Policy	When it Changes	Example Use Case
Global	Never (constant for whole frame)	Default textures, global settings
PerFrame	Once per frame	Camera, time, global lighting
PerObject	Per object	Object transforms, per-object parameters
PerMaterial	Per material	Material textures, material properties

For custom bind groups:

- Use `PerObject` when data is unique per object instance (like our tiling)
- Use `PerFrame` if data updates every frame but is shared (like a global animation time)
- Use `PerMaterial` if data is shared by objects with same material

Performance impact:

- `PerObject` with unique data = creates many bind groups (one per object)
- `PerObject` with shared data = reuses cached bind groups automatically

The engine's `BindGroupBinder` handles all caching - you just specify the policy.

Experiments to Try

1. Animate the offset - In `CustomRenderNode`, add an `update()` method:

Implement the `UpdateNode` behavior.

```
class CustomRenderNode : public engine::scene::nodes::ModelRenderNode, public
engine::scene::nodes::UpdateNode
```

Override the default `update(float deltaTime)` method.

```
virtual void update(float deltaTime) override {
    tileUniforms.tileOffset.x += deltaTime * 0.1f; // Scroll right
}
```

2. Different tiling per object - Create multiple floor nodes:

```
auto floor1 = std::make_shared<demo::CustomRenderNode>(floorModel);
floor1->tileUniforms.tileSize = glm::vec2(2.0f, 2.0f);
floor1->getTransform().setLocalPosition(glm::vec3(-5.0f, 0.0f, 0.0f));
floor1->getTransform().setLocalScale(glm::vec3(5.0f, 1.0f, 5.0f));
rootNode->addChild(floor1);

auto floor2 = std::make_shared<demo::CustomRenderNode>(floorModel);
floor2->tileUniforms.tileSize = glm::vec2(8.0f, 8.0f);
floor2->getTransform().setLocalPosition(glm::vec3(5.0f, 0.0f, 0.0f));
floor2->getTransform().setLocalScale(glm::vec3(5.0f, 1.0f, 5.0f));
rootNode->addChild(floor2);
```

3. Add rotation - Extend TileUniforms:

Adjust C++ Struct

```
// CustomRenderNode.h
struct TileUniforms {
    glm::vec2 tileOffset;  // Offset of the tile
    glm::vec2 tileSize;    // Scale/size of the tile
    glm::vec4 rotation;    // Rotation in .x, padding in .yzw for 16-byte
alignment
};
```

Adjust WGSL Struct

```
struct TileUniforms {
    tileOffset: vec2f,   // Offset
    tileSize: vec2f,     // Scale
    rotation: vec4f,     // rotation.x = angle in radians, yzw = padding
}
```

Define WGSL Helper Function

```
fn rotate2D(uv: vec2f, angle: f32) -> vec2f {
    let cosA = cos(angle);
    let sinA = sin(angle);
    return vec2f(
        uv.x * cosA - uv.y * sinA,
        uv.x * sinA + uv.y * cosA
    );
}
```

Update Fragment Shader

```
let rotatedUV = rotate2D(input.texCoord - 0.5, tileUniforms.rotation.x) + 0.5;
let tiledUV = rotatedUV * tileUniforms.tileSize + tileUniforms.tileOffset;
```

Understanding WebGPU Custom Resources

WebGPU bind group creation:

When you call `BindGroupDataProvider::create()`, the engine internally:

1. **Creates GPU buffer** - Allocates uniform buffer with `wgpu::BufferUsage::Uniform | CopyDst`
2. **Writes data** - Copies your struct to GPU with `queue.writeBuffer()`
3. **Creates bind group** - Links buffer to bind group layout via `device.createBindGroup()`
4. **Caches by key** - Stores in `FrameCache::customBindGroupCache[key]` where key = (shader, bindGroup, instanceId)

Next frame:

- Same object? Reuse cached bind group, just update buffer if data changed
- Different object? Create new bind group or fetch from cache by different instanceId

Memory management:

- Bind groups live in `FrameCache` and are recreated each frame
- GPU buffers are pooled and reused when possible
- Old bind groups are automatically cleaned up at frame end

Troubleshooting

Build Failures - Reading Terminal Output

⚠ **Important:** When using `scripts/build.bat`, the task system may report success even if the build actually failed. You **MUST check the terminal output** to see the real result.

What to look for in terminal:

1. Scroll to the **very end** of the terminal output
2. Look for `[SUCCESS] Build completed successfully!` - if this appears, build succeeded
3. If you see `[ERROR] Build failed.` - the build failed regardless of task status

Common build issues:

- **Missing semicolons** - WGSL requires `;` at end of statements
- **Struct size mismatch** - `tileUniforms` size must be 16 bytes (2 × vec2f)
- **Bind group naming** - `"TileUniforms"` in registration must match shader `@group(3)`
- **CMake cache** - Run `rm -r build` (or delete `build/` folder) then rebuild clean

Bind Group Issues

"Unable to find bind group 'TileUniforms'"

- Check shader name matches in `preRender()` ("unlit") and registration as well as material assignment
- Ensure bind group name matches exactly (case-sensitive)
- Verify `@group(3)` in shader doesn't conflict with other bind groups

Floor appears stretched/wrong

- Verify TileUniforms struct size matches shader (16 bytes = 2 × vec2f)
- Check reuse policy is `PerObject` in both registration and `preRender()`

Data doesn't update

- Ensure `preRender()` is called (check node is enabled and in scene graph)
- Verify you're modifying the member variable, not a local copy
- Check instance ID is correct in `BindGroupDataProvider::create()`

Shader compilation error

- Check `@group(3)` doesn't conflict with other bind groups
- Ensure TileUniforms struct is defined before use
- Verify binding index `@binding(0)` matches registration
- Verify there are no missing `;`

Debug Strategy

If errors are unclear:

1. Open `MeshPass.cpp` in your editor
2. Add a breakpoint in the `render()` method
3. Press `F5` to start debugging with VS Code
4. Check the **Terminal Output** panel - shader errors will be printed there

---

Key Takeaways

- ☑ **Custom Bind Groups** - Extend shader capabilities with Group 3+ for custom data
- ☑ **BindGroupDataProvider** - Simple API to pass CPU data to GPU shaders
- ☑ **preRender() Lifecycle** - Called before rendering, perfect for per-frame updates
- ☑ **Reuse Policies** - Control caching behavior (PerFrame, PerObject, PerMaterial)
- ☑ **Node Customization** - Extend ModelRenderNode to add custom shader data

WebGPU Concepts Learned:

- Bind group indices beyond standard groups (3, 4, 5...)
- Custom uniform buffer creation and caching
- Alignment requirements for uniform structs (16-byte boundaries)
- Efficient resource reuse via caching policies

What's Next?

In **Tutorial 03**, you'll learn:

- Shadow mapping with directional lights
- Sampling depth textures in shaders
- Transform coordinates from world space to light space
- Percentage Closer Filtering (PCF) for soft shadows

Next Tutorial: 03_shadow_mapping.md / 03_shadow_mapping.pdf / 03_shadow_mapping.html

---

Further Reading

- Bind Group System Documentation
- Node System Guide
- WebGPU Bind Group Spec
- Tutorial 01: Unlit Shader