Gigi Shader Documentation

This documents how to author shaders in Gigi.

Gigi shaders use special markup tokens for certain things and the reason for that is twofold:

The reason for this is twofold:

1. Different platforms have different requirements for things such as how resources are declared, or the name of the function to call to trace a ray.
2. Tokens allow hints to Gigi from within the shader. For instance, you can use a Gigi variable in a shader, which Gigi will create a constant buffer for and manage automatically. You can also declare that you want to read a texture on disk, and Gigi will handle loading and shuttling the resource to your shader.

Tokens

All tokens are of the form /*$(***Token:Param1:Param2:…:ParamN***)*/.

Listed below are the token strings supported in shaders, as well as an explanation of their parameters.

In the case of multiple parameters, the parameters are positional, and are separated by colons (:).

- **Variable:name** – give the name of a Gigi variable and that variable will appear in the shader. The variable will be sent down through a constant buffer, unless the variable is marked "const" in which case, the default value will be put in as a literal instead. See the Built In Variables section for a list of built in variables that can be accessed, in addition to the variables defined in the render graph.
- **VariableAlias:name** – shaders let you specify variable aliases where when nodes use those shaders they specify which variable should actually be used for that node. This mimics the idea of push constants by letting you specify different variables for the same constant buffer fields, for different nodes. This tag specifies where the value of the actual variable chosen should go, in the shader.
- **ShaderResources** – This should be at the top of every shader file. This is where resource declarations are put, sampler definitions, as well as enum and struct definitions, and any include files the target platform wants shaders to include.
- **RayTraceFn** – Replaced by the target platform's "TraceRay" function. Some platforms abstract this function, and using this token allows them to handle that correctly. The token is optional however, and you can call TraceRay directly if you don't need the portability.
- **_compute:entryPoint** – This declares a "compute" shader entry point. Optional token that helps portability, but also will put the numthreads defined in the shader in the CSNumThreads field. You are expected to put the parameter list after this token.
- **_amplification:entryPoint** – This declares an "amplification" shader entry point. Optional token that helps portability, but also will put the numthreads defined in the shader in the CSNumThreads field. You are expected to put the parameter list after this token.

- **_mesh:entryPoint:topology** – This declares a “mesh” shader entry point. Optional token that helps portability, but also will put the numthreads defined in the shader in the CSNumThreads field. You are expected to put the parameter list after this token. Topology is optional, and if not given, defaults to “triangle”. The other option is “line”.
- **_raygeneration:entryPoint** – This declares a “ray gen” shader entry point. Optional token that helps portability.
- **_miss:entryPoint** – same as above but defines a miss shader.
- **_anyhit:entryPoint:attribStruct** – same as above but defines an any hit shader. The attribStruct parameter is optional and allows you to define the attribute struct the function takes in. “BuiltInTriangleIntersectionAttributes” is the default for DX12, but if you are using an intersection shader, you likely want to use your own struct instead to be able to communicate information between the intersection shader and the other shader types.
- **_intersection:entryPoint** – same as above but defines an intersection shader.
- **_closesthit:entryPoint:attribStruct** – same as above, but defines a closest hit shader. The attribStruct parameter is optional and allows you to define the attribute struct the function takes in. “BuiltInTriangleIntersectionAttributes” is the default for DX12, but if you are using an intersection shader, you likely want to use your own struct instead to be able to communicate information between the intersection shader and the other shader types.
- **Image2D:fileName:format:viewType:sRGB:makeMips** – This allows you to specify that you want to read a 2d texture in your shader. Gigi will load the texture and get it to your shader as an SRV. Image can be used instead of Image2D. fileName is a relative path from the .gg file location. makeMips is optional and defaults to false. Example from skybox.hlsl within the skybox technique:

```
    float3 texel = float3(0.0f, 0.0f, 0.0f);
    switch(/*$(Variable:skybox)*/)
    {
        case Skybox::Arches: texel =
/*$(Image:skyboxes/Arches_E_PineTree_3k.hdr:RGBA32_Float:float4:false)*/.SampleLe
vel(texSampler, uv, 0).rgb; break;
        case Skybox::SataraNightHdr: texel =
/*$(Image:skyboxes/satara_night_4k.hdr:RGBA32_Float:float4:false)*/.SampleLevel(t
exSampler, uv, 0).rgb; break;
        case Skybox::SataraNightExr: texel =
/*$(Image:skyboxes/satara_night_4k.exr:RGBA32_Float:float4:false)*/.SampleLevel(t
exSampler, uv, 0).rgb; break;
    }
```

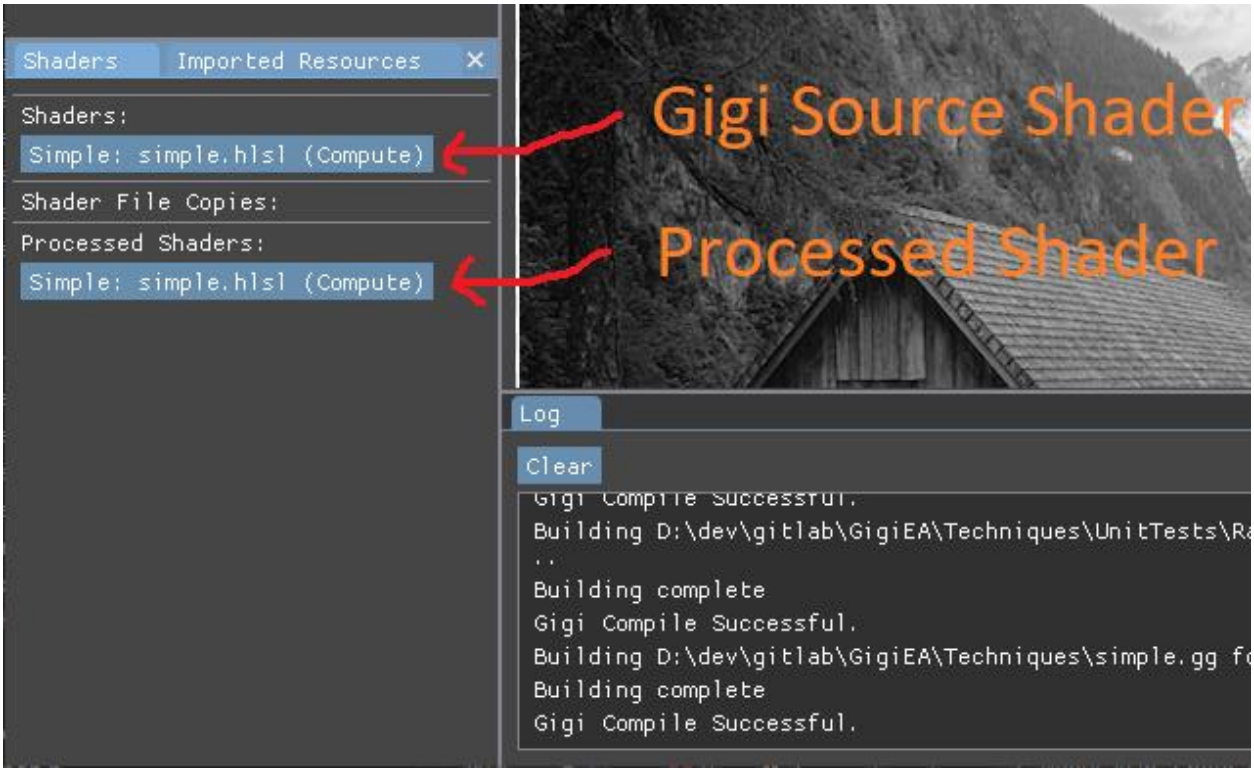
- **Image2DArray:fileName:format:viewType:sRGB:makeMips** – Same as Image2D except that it can load multiple textures into a 2D Array texture. To load multiple textures, put a %i in the file name and Gigi will iterate from 0 to N, loading all the textures found into the array.
- **Image3D:fileName:format:viewType:sRGB:makeMips** – The same behavior as Image2DArray, except it loads it into a 3D texture.

- **ImageCube:fileName:format:viewType:sRGB:makeMips** – the same behavior as above, but it loads it into a cube map. The indices are +x, -x, +y, -y, +z, -z. You can alternately use %s instead of %i, and it will fill in the words “Right”, “Left”, “Up”, “Down”, “Front”, “Back”.
- **RTHitGroupIndex:name** – This is replaced with the index of a hit group in the render graph hit group list, which is also the index of the hit group in the shader table. Used by raytracing shaders.
- **RTHitGroupCount** – This is replaced with the number of hit groups in the render graph. Used by raytracing shaders.
- **RTMissIndex:name** – This is replaced with the index of the miss shader within the miss shader table. Used by raytracing shaders.
- **NumThreads** – This is replaced with a uint3 holding the number of threads are ran per dispatch for this shader, as specified on the shader.
- **DispatchMultiply, DispatchDivide, DispatchPreAdd, DispatchPostAdd** – These are replaced with uint3s holding these dispatch calculation values as specified on the node.
- **Assert:Condition[,fmt][…]** – Within compute shaders, vertex shaders, and pixel shaders, you can write an assert that if failed, will write a message to the log window of the viewer. Viewer only, disappears in generated code. Example: **/*$(Assert: 1>4, "v0:%.1f v1:%.1f v2:%.1f v3:%.1f v4:%.1f v5:%.1f", v0, v1, v2, v3, v4, v5)*/**. Where v0 through v5 are shader variables. Up to 6 parameters supported max. All parameters must be floats. The condition can use shader variables too, of course. Asserts only fire once for each time the file is loaded.
- **RWTextureR:name** – This declares read access to a read/write texture and will be replaced by the name of the texture resource. This is optional to use, but helps webgpu get around the limited R/W texture format issues, by allowing it to split the access into a read only resource and write only resource.
- **RWTextureW:name** – The same as RWTextureR, but declares write access.

Debugging Processed Shaders

Shader compilation errors give information about the processed shader files, which can be confusing or difficult to debug. To help this, the viewer allows you to view the processed shader files under the “Shaders” tab. Click the button to view either the Gigi source shader, or the processed shader output.

The processed shaders can also be seen in the generated code packages.



More ShaderResources Token Details

Besides resource declarations and similar, the ShaderResources token is where enums and structs are defined, so they can be used by the shader.

Enums are defined as structs with static const int variables for each value. If a technique has an enum named “ViewMode”, with three values “Result”, “Error”, “AbsError”, this is inserted into the ShaderResources section:

```
struct ViewMode
{
    static const int Result = 0;
    static const int Error = 1;
    static const int AbsError = 2;
};
```

That allows enums to be used like this:

```
    switch(/*$(Variable:viewMode)*/)
    {
        case ViewMode::Result: result = color; break;
        case ViewMode::Error: result = (colorGT - color) *
/*$(Variable:errorScale)*/; break;
        case ViewMode::AbsError: result = abs(colorGT - color) *
/*$(Variable:errorScale)*/; break;
```

```
        default:
        {
            // To show when there's an unhandled case
            result = float3(0.0f, color.g, 0.0f);
            break;
        }
    }
```

Shader Headers

If you want to make a shader header file that can be included by other Gigi files, you can do so using the steps below. Shader header files do not undergo processing though, and are not scanned for variable use, texture loading, or similar.

1. In your shader, do an include with a relative path from the current shader, to the included shader.
2. Add this file as a file copy of type “Shader” in the editor. This will cause this file to be copied into the shaders folder, preserving any relative directory paths.