WEBGPU SHADERS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/configuring-effects/webgpu-shaders

Effect addons that support WebGPU must provide a shader written in WebGPU's shading language WGSL. This section provides information specific to WebGPU shaders.

Providing a WGSL shader variant

To provide a WGSL shader variant for WebGPU, the `"supported-renderers"` property in addon.json must specify `"webgpu"`, e.g.:

```
"supported-renderers": ["webgl", "webgpu"]
```

> *Note that the supported renderers can also include `"webgl2"` if your shader also uses a WebGL 2 shader variant.*

This tells Construct that the effect also supports WebGPU, and it will look for a WGSL shader file with the name *effect.wgsl*.

Writing WGSL shaders

WGSL is a substantially different shader language to GLSL. This documentation does not cover the full details of the shader language. However there are two significant differences to GLSL shaders to note when writing WGSL shaders in Construct:

1. As WebGPU is a lower-level API than WebGL, WGSL shaders tend to be more verbose than GLSL, and also need to explicitly specify engine-specific details like binding and group numbers. To avoid hard-coding details that may change in future in to WGSL shaders, Construct provides a simple preprocessor based on tokens of the form `%%NAME%%`. These are not part of the WGSL language but are Construct-specific placeholders that Construct will replace with WGSL attributes and code. A full list of the supported placeholders is included below.
2. Effect parameters are stored in a struct and have no name associated with them in WGSL (they are referenced by a byte offset). Therefore they ignore the *uniform* property for the parameter set in addon.json. Instead just list all effect parameters in the `ShaderParams` struct in the order they are defined and with the appropriate type, and Construct will automatically calculate their byte offsets and update them accordingly.

Construct-specific placeholders

Here is a list of placeholders of the form `%%NAME%%` that Construct will replace in WGSL shaders.

%%SAMPLERFRONT_BINDING%%

%%TEXTUREFRONT_BINDING%%

Replaced with the `@binding` and `@group` attributes for the foreground sampler and texture. Example usage:

```
%%SAMPLERFRONT_BINDING%% var samplerFront : sampler;
%%TEXTUREFRONT_BINDING%% var textureFront : texture_2d<f32>;
```

%%SAMPLERBACK_BINDING%%

%%TEXTUREBACK_BINDING%%

Replaced with the `@binding` and `@group` attributes for the background sampler and texture. Example usage:

```
%%SAMPLERBACK_BINDING%% var samplerBack : sampler;
%%TEXTUREBACK_BINDING%% var textureBack : texture_2d<f32>;
```

%%SAMPLERDEPTH_BINDING%%

%%TEXTUREDEPTH_BINDING%%

Replaced with the `@binding` and `@group` attributes for the depth sampler and texture for depth effects like fog. Example usage:

```
%%SAMPLERDEPTH_BINDING%% var samplerDepth : sampler;
%%TEXTUREDEPTH_BINDING%% var textureDepth : texture_depth_2d;
```

%%FRAGMENTINPUT_STRUCT%%

Defines the `FragmentInput` structure used as input to the fragment shader method. This structure defines `fragUV : vec2<f32>` as the current fragment texture co-ordinates (equivalent to `vTex` in GLSL shaders). It also defines `@builtin(position) fragPos : vec4<f32>` and two utility methods that use it (see below).

%%FRAGMENTOUTPUT_STRUCT%%

Defines the `FragmentOutput` structure returned from the fragment shader method. This structure defines `color : vec4<f32>` which is used to write the output color from the shader (equivalent to writing to `gl_FragColor` in WebGL 1 shaders).

%%SHADERPARAMS_BINDING%%

Replaced with the `@binding` and `@group` attributes for the structure containing custom effect parameters. This structure must be defined by your shader matching the effect parameters in the same order. It can be omitted if the shader does not use any custom parameters. Example usage from the 'Set color' sample shader:

```
struct ShaderParams {
        setColor : vec3<f32>
```

```
};
%%SHADERPARAMS_BINDING%% var<uniform> shaderParams : ShaderParams;
```

%%C3PARAMS_STRUCT%%

Defines a structure named `c3Params` which contains members that correspond to the Construct-provided uniforms for WebGL shaders, as well as a set of utility methods. The members of the structure currently include:

```
        srcStart                    : vec2<f32>,
        srcEnd                      : vec2<f32>,
        srcOriginStart          : vec2<f32>,
        srcOriginEnd            : vec2<f32>,
        layoutStart                 : vec2<f32>,
        layoutEnd                   : vec2<f32>,
        destStart                   : vec2<f32>,
        destEnd                     : vec2<f32>,
        devicePixelRatio        : f32,
        layerScale                  : f32,
        layerAngle                  : f32,
        seconds                     : f32,
        zNear                       : f32,
        zFar                        : f32,
        isSrcTexRotated         : u32
```

%%C3_UTILITY_FUNCTIONS%%

Defines a set of utility functions that are useful for many kinds of effects (see below)

Utility functions

Some placeholders also include definitions for useful helper functions that perform common tasks in shaders. The available functions are documented below.

Provided by %%FRAGMENTINPUT_STRUCT%%

fn c3_getBackUV(fragPos : vec2<f32>, texBack : texture_2d<f32>) -> vec2<f32>

Helper function to calculate the texture co-ordinates to sample the background texture at for background blending effects. Example: `c3_getBackUV(input.fragPos.xy, textureBack)`

fn c3_getDepthUV(fragPos : vec2<f32>, texDepth : texture_depth_2d) -> vec2<f32>

Helper function to calculate the texture co-ordinates to sample the depth texture at for depth-processing effects. Example: `c3_getDepthUV(input.fragPos.xy, textureDepth)`

Provided by %%C3PARAMS_STRUCT%%

fn c3_srcToNorm(p : vec2<f32>) -> vec2<f32>

fn c3_normToSrc(p : vec2<f32>) -> vec2<f32>

fn c3_srcOriginToNorm(p : vec2<f32>) -> vec2<f32>

fn c3_normToSrcOrigin(p : vec2<f32>) -> vec2<f32>

Pass `input.fragUV` to `c3_srcToNorm()` to return a position normalized in the range [0, 1] relative to the box `srcStart` to `srcEnd`. The `c3_normToSrc()` function performs the reverse calculation. The `srcOrigin` variants work relative to the box `srcOriginStart` to `srcOriginEnd` instead.

---

fn c3_clampToSrc(p : vec2<f32>) -> vec2<f32>

fn c3_clampToSrcOrigin(p : vec2<f32>) -> vec2<f32>

Clamps a given position to the box `srcStart` to `srcEnd` or `srcOriginStart` to `srcOriginEnd`.

---

fn c3_getLayoutPos(p : vec2<f32>) -> vec2<f32>

Pass `input.fragUV` to calculate the current corresponding position in layout co-ordinates.

---

fn c3_srcToDest(p : vec2<f32>) -> vec2<f32>

Maps a texture co-ordinate in the `srcStart` to `srcEnd` rectangle to the corresponding position in the `destStart` to `destEnd` rectangle.

---

fn c3_clampToDest(p : vec2<f32>) -> vec2<f32>

Clamps a texture co-ordinate to the `destStart` to `destEnd` rectangle.

---

fn c3_linearizeDepth(depthSample : f32) -> f32

Linearize a sample from the depth texture to a Z distance. Depth texture samples are usually in a normalized range [0, 1]; this method returns a Z distance based on the near and far planes, which is a more useful number for things like fog effects.

Provided by %%C3_UTILITY_FUNCTIONS%%

---

fn c3_premultiply(c : vec4<f32>) -> vec4<f32>

fn c3_unpremultiply(c : vec4<f32>) -> vec4<f32>

Premultiplies the RGB components by the A component in a color, and the reverse operation.

---

fn c3_grayscale(rgb : vec3<f32>) -> f32

Convert RGB colors to a corresponding grayscale component.

fn c3_getPixelSize(t : texture_2d<f32>) -> vec2<f32>

Returns the size of a pixel in texture co-ordinates on the given texture.

> *This uses the `textureDimensions()` WGSL built-in, and can be used as a replacement for the `pixelSize` uniform in the WebGL renderer. It is further also capable of determining the pixel size for any given texture.*

fn c3_RGBtoHSL(color : vec3<f32>) -> vec3<f32>

fn c3_HSLtoRGB(hsl : vec3<f32>) -> vec3<f32>

Converts RGB values to the equivalent in HSL, and the reverse operation.

Useful shader calculations

Some common calculations done in WGSL shaders are listed below.

To sample the foreground pixel:

```
var front : vec4<f32> = textureSample(textureFront, samplerFront, input.fragUV);
```

To sample an adjacent pixel, offset by the pixel size:

```
// get width of a pixel in texture co-ordinates
var pixelWidth : f32 = c3_getPixelSize(textureFront).x;

// sample next pixel to the right
var next : vec4<f32> = textureSample(textureFront, samplerFront, input.fragUV + vec2<f32>(pixelW
```

To calculate the position to sample the background, use the `c3_getBackUV()` helper function:

```
var back : vec4<f32> = textureSample(textureBack, samplerBack, c3_getBackUV(input.fragPos.xy, te
```

Sampling the depth buffer works similarly to sampling the background, but using the `c3_getDepthUV()` helper function on the depth texture and sampler. It's commonly useful to then linearize the resulting depth sample to a Z distance based on the near and far planes, which the `c3_linearizeDepth()` helper function does.

```
// sample depth buffer
var depthSample : f32 = textureSample(textureDepth, samplerDepth, c3_getDepthUV(input.fragPos.xy

// linearize depth sample to Z distance
var zLinear : f32 = c3_linearizeDepth(depthSample);
```

To calculate the current texture co-ordinate relative to the object being rendered, without being affected by clipping at the edge of the viewport, use the `c3_srcOriginToNorm()` helper method:

```
var n : vec2<f32> = c3_srcOriginToNorm(input.fragUV);
```

To calculate the current layout co-ordinates being rendered, use the `c3_getLayoutPos()` helper method:

```
var l : vec2<f32> = c3_getLayoutPos(input.fragUV);
```

Construct renders using premultiplied alpha. Often it is convenient to modify the RGB components without premultiplication. To do this, use the `c3_unpremultiply()` and `c3_premultiply()` helper methods:

```
// sample front texture
var front : vec4<f32> = textureSample(textureFront, samplerFront, input.fragUV);

// unpremultiply
front = c3_unpremultiply(front);

// ...modify unpremultiplied front color...

// premultiply again
front = c3_premultiply(front);
```

Precision in WebGPU shaders

WebGL shaders allow the use of shader precision qualifiers such as `lowp` and `mediump`. WebGPU uses a different approach with explicit types such as `f32`. Some devices support a lower-precision `f16` type if they support the `shader-f16` feature. To help make it easy to use the `f16` type, Construct requests to use the `shader-f16` feature where supported, and defines a type in WebGPU shaders named `f16or32`, which is `f16` when `shader-f16` is supported, otherwise it is `f32`.

Currently all of Construct's built-in inputs, outputs and library functions use the `f32` type exclusively for broadest compatibility. However shaders can make use of the `f16or32` type for their internal calculations, converting to `f32` where necessary, to help improve shader performance, especially as `f32` is high precision with a higher performance cost compared to `lowp` or `mediump` precision in GLSL.

Compatibility differences with WebGL shaders

Due to API differences between WebGL and WebGPU, the WebGL src/srcOrigin/dest uniforms use an inverted Y direction. This means instead of ranging from 0-1 for top-to-bottom, they range from 1-0.

Sometimes this does not have any impact on the effect. However in some cases it does, depending on the kinds of calculation done in the shader. When porting a GLSL shader to WGSL, you may need to emulate the inverted Y direction in WGSL to achieve the same effect. For example the Lens2 effect uses the following code pattern in WGSL to emulate the inverted Y direction:

```
// At start of shader: get normalized source co-ordinates
// and then invert Y direction to match WebGL
var tex : vec2<f32> = c3_srcToNorm(input.fragUV);
tex.y = 1.0 - tex.y;

// ... rest of effect ...

// At end of shader: invert Y direction again and then
// calculate background sampling position
p.y = 1.0 - p.y;

var output : FragmentOutput;
output.color = textureSample(textureBack, samplerBack, mix(c3Params.destStart, c3Params.destEnd,
```

If you are writing a new effect, consider writing the WebGPU shader first, and then if necessary applying the Y inversion in the WebGL shader instead. As WebGPU is the newer technology, in the long term the WebGL renderer may eventually be retired, in which case it is better to have a natural code style in the WGSL shader.