# WEBGL SHADERS

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

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

## Adding a WebGL 2 shader variant

All shaders written for WebGL 1 (using GLSL ES 1.0) are compatible with both WebGL 1 and WebGL 2. There is no need to write a WebGL 2 variant of a shader unless you need specific features only available with WebGL 2 (using GLSL ES 3.0).

If you do write a WebGL 2 shader, we strongly recommend still providing a WebGL 1 shader. Do whatever you can to support WebGL 1, perhaps by using WebGL 1 extensions (see the following section for more details), or use a fallback like a low quality version, a glitchy version, or even just output transparency so it doesn't render. If you don't provide a WebGL 1 shader at all, then any project using your shader will cause an error on devices that still only support WebGL 1, with the project failing to load and just displaying a blank screen.

### Providing a WebGL 2 shader variant

To provide a WebGL 2 shader variant, ensure `"webgl2"` is listed in the `"supported-renderers"` property of addon.json, e.g.:

```
"supported-renderers": ["webgl", "webgl2"]
```

This tells Construct to look for both a WebGL 1 and WebGL 2 shader for your addon.

The WebGL 1 shader is still in the file `effect.fx` as before. If enabled then the file `effect.webgl2.fx` specifies the shader to load for WebGL 2. A sample of an effect using both WebGL 1 and WebGL 2 shaders is provided in the effect SDK download.

### Writing WebGL 2 shaders

WebGL 2 shaders are written using GLSL ES 3.0, as opposed to GLSL ES 1.0 for WebGL 1 shaders. This documentation does not cover the full details of how to write WebGL shaders - there are lots of other resources across the web covering that. However some key points to note when writing a WebGL 2 shader are:

- A WebGL 2 shader MUST start with the line `#version 300 es`. This must be the first line - no comments or other lines are allowed before it.
- Change `varying` to `in` for the `vTex` declaration.

- `gl_FragColor` is not used in WebGL 2 shaders. Instead declare `out lowp vec4 outColor;` at the top level and assign the result color to that.
- The `texture2D()` function for sampling a texture is now just `texture()` with WebGL 2.

Once adapted you can then make use of WebGL 2 shader features, such as `dFdx()`, `dFdy()` and `textureGrad()`.

### Using WebGL 1 extensions

When only WebGL 1 is supported, Construct unconditionally activates the following extensions if supported:

- EXT_frag_depth
- OES_standard_derivatives
- EXT_shader_texture_lod

If your WebGL 2 shader uses equivalent features, this means you can sometimes support WebGL 1 too by activating them in your WebGL 1 shader, e.g.:

```
#extension GL_EXT_frag_depth : enable
#extension GL_EXT_shader_texture_lod : enable
#extension GL_OES_standard_derivatives : enable

// now you can use gl_FragDepthEXT, dFdx, dFdy, texture2DGradEXT etc.
```

Note Construct currently doesn't support any way to provide an alternative WebGL 1 shader when these extensions are not supported. However this approach lets you support more devices as instead of requiring WebGL 2, your shader can work with WebGL 1 as well when the necessary extensions are available.

> *WebGL 2 does not support these extensions as they are built-in features with WebGL 2. You cannot write just a WebGL 1 shader using those extensions, as it won't work with WebGL 2. Instead you must write a WebGL 2 shader variant.*

### Testing

The Construct editor provides a setting to force the editor and preview to run with WebGL 1. This can help you test your shader variants with both WebGL 1 and WebGL 2 (assuming your device supports WebGL 2). Note this option exists for shader testing only - exported projects will continue to use WebGL 2 when available regardless of the editor setting.

## Shader uniforms

Shaders are written in a GLSL (OpenGL Shading Language) ES 1.0 fragment shader and interpreted by the browser's WebGL implementation. As with normal fragment shaders, the output is written to the special `gl_FragColor` variable. A WebGL 2 shader variant can be provided which must be written in GLSL ES 3.0 which has a number of differences; see the previous section on adding a WebGL 2 shader variant for more details.

The current foreground texture co-ordinate is provided in the special variable `vTex`. This is normally used to read the foreground texture, but it is actually optional (in case you want to write a shader that generates all of its output without reference to the foreground texture at all). All other uniforms are optional, and are documented below. The full uniform declaration is included with the recommended precision.

---

**uniform lowp sampler2D samplerFront;**

The foreground texture sampler, to be sampled at `vTex`.

---

**uniform mediump vec2 srcStart;**

**uniform mediump vec2 srcEnd;**

The current foreground rectangle being rendered, in texture co-ordinates. Note this is clamped as the object reaches the edge of the viewport. These are mainly useful for calculating the background sampling position.

---

**uniform mediump vec2 srcOriginStart;**

**uniform mediump vec2 srcOriginEnd;**

The current foreground source rectangle being rendered, in texture co-ordinates. This is not clamped, so can cover a rectangle that leaves the viewport. These are mainly useful for calculating the current sampling position relative to the object being rendered, without changing as the object clips against the viewport.

---

**uniform mediump vec2 layoutStart;**

**uniform mediump vec2 layoutEnd;**

The current foreground source rectangle being rendered, in layout co-ordinates. This allows the current fragment's position in the layout to be calculated.

---

**uniform lowp sampler2D samplerBack;**

The background texture sampler used for background-blending effects. The `blends-background` property in addon.json should also be set to `true` before using this. For the correct way to sample the background, see the next section.

---

**uniform lowp sampler2D samplerDepth;**

The depth texture sampler used for depth-based effects. The `uses-depth` property in addon.json should also be set to `true` before using this. The depth texture is the same size as the background texture, so this is sampled similarly to `samplerBack`. See the next section for more details.

---

**uniform mediump vec2 destStart;**

**uniform mediump vec2 destEnd;**

The current background rectangle being rendered to, in texture co-ordinates, for background-blending effects. For the correct way to sample the background, see the next section.

---

**uniform highp float seconds;**

The time in seconds since the runtime started. This can be used for animated effects. The `animated` property in addon.json should be set to `true`.

> *Note `highp` can only be used on certain platforms. To work across all systems, check `#ifdef GL_FRAGMENT_PRECISION_HIGH` to see if `highp` is supported, else fall back to using `mediump`.*

---

**uniform mediump vec2 pixelSize;**

The size of a texel in the foreground texture in texture co-ordinates. This allows calculating distances in pixels rather than texture co-ordinates.

---

**uniform mediump float layerScale;**

The current layer scale as a factor (i.e. 1 is unscaled). This is useful to ensure effects scale according to zoom.

---

**uniform mediump float layerAngle;**

The current layer angle in radians.

---

**uniform mediump float devicePixelRatio;**

The value of devicePixelRatio in the browser, which is the number of device pixels per CSS pixel. This may be necessary in some effects to handle high-DPI displays.

---

**uniform mediump float zNear;**

**uniform mediump float zFar;**

The values of the project properties *Near distance* and *Far distance*, which represent the distance of the near and far planes from the camera position.

## Useful shader calculations

Some common calculations done with the available uniforms are listed below.

To sample the foreground pixel:

```
lowp vec4 front = texture2D(samplerFront, vTex);
```

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

```
// sample next pixel to the right
lowp vec4 next = texture2D(samplerFront, vTex + vec2(pixelSize.x, 0.0));
```

To calculate the position to sample the background, find the normalised position `n` of `vTex` in the foreground rectangle, and apply that to the background rectangle:

```
mediump vec2 n = (vTex - srcStart) / (srcEnd - srcStart);
lowp vec4 back = texture2D(samplerBack, mix(destStart, destEnd, n));
```

Sampling the depth buffer works similarly to sampling the background, but only provides one component, so just read the `r` value. Note that the value in the depth buffer is normalized (0-1 range) and does not linearly correspond to distance. To get a linearized Z value for a depth sample, use the calculation below, which uses the `zNear` and `zFar` uniforms.

```
mediump vec2 n = (vTex - srcStart) / (srcEnd - srcStart);
mediump float depthSample = texture2D(samplerDepth, mix(destStart, destEnd, n)).r;
mediump float zLinear = zNear * zFar / (zFar + depthSample * (zNear - zFar));
```

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 original source rectangle:

```
mediump vec2 srcOriginSize = srcOriginEnd - srcOriginStart;
mediump vec2 n = ((vTex - srcOriginStart) / srcOriginSize);
```

To calculate the current layout co-ordinates being rendered, add an extra step to interpolate `n` across the layout rectangle:

```
mediump vec2 srcOriginSize = srcOriginEnd - srcOriginStart;
mediump vec2 n = ((vTex - srcOriginStart) / srcOriginSize);
mediump vec2 l = mix(layoutStart, layoutEnd, n);
```

Construct renders using premultiplied alpha. Often it is convenient to modify the RGB components without premultiplication. To do this, divide by alpha to unpremultiply the color, but be sure not to divide by zero.

```
lowp vec4 front = texture2D(samplerFront, vTex);
lowp float a = front.a;

// unpremultiply
if (a != 0.0)
        front.rgb /= a;

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

// premultiply again
front.rgb *= a;
```