# IRENDERER SCRIPT INTERFACE

**View online:** https://www.construct.net/en/make-games/manuals/construct-3/scripting/scripting-reference/graphics-interfaces/irenderer-interface

The `IRenderer` script interface provides access to Construct's renderer in the runtime. The same interface can be used regardless of the underlying rendering technology (e.g. WebGL or WebGPU). The interface's methods provide high-level drawing commands implemented by Construct, so you don't need to handle low-level concerns like vertex buffers.

`IRenderer` can be used both in project code, such as with the ILayer events `"beforedraw"` and `"afterdraw"`, and in the addon SDK for drawing plugins.

## Renderer state

`IRenderer` uses a persistent rendering state. Therefore to correctly render something, all the intended state must be specified, otherwise it will use an undefined previous state. `IRenderer` simplifies the renderer state to:

1. A blend mode. Typically a normal alpha blend mode is used.
2. A fill mode (internally, the current fragment shader). The fill modes can be *color fill* (draw a solid color), *texture fill* (draw a texture), and *smooth line fill* (for drawing smooth lines).
3. A color set by `setColor()` or `setColorRgba()`. The alpha component of the color is used as the opacity in texture fill mode.
4. A texture set by `setTexture()`. This is only used in texture fill mode.

There are two other states that are more applicable to 3D content:

- The face culling mode, set by `setCullFaceMode()`
- The front face winding that is also used by the face culling mode, set by `setFrontFaceWinding()`

A drawing method should begin by at least specifying the blend mode, the fill mode, the color, and the texture (if texture fill mode is used), before continuing to draw. The renderer efficiently discards redundant calls, so if the state does not actually change then these calls have minimal performance overhead.

Once all state is set up, quads can be issued using one of the `rect()` or `quad()` method overloads. These methods draw using the currently set state.

## Texture options

Some texture methods accept the same options objects to specify the texture parameters. To avoid repeating them, the common options that can be specified are documented below.

- `wrapX` : the texture horizontal wrap mode: one of `"clamp-to-edge"` , `"repeat"` , `"mirror-repeat"`
- `wrapY` : as with `wrapX` but for the vertical wrap mode
- `sampling` : the texture sampling mode, one of `"nearest"` , `"bilinear"` or `"trilinear"` (default)
- `mipMap` : boolean indicating if mipmaps should be used for this texture, default true

## Methods

#### setAlphaBlendMode()

Set the blend mode to a premultiplied alpha blending mode.

#### setBlendMode(blendMode)

Set the blend mode by a string which must be one of `"normal"` , `"additive"` , `"copy"` , `"destination-over"` , `"source-in"` , `"destination-in"` , `"source-out"` , `"destination-out"` , `"source-atop"` , `"destination-atop"` , `"lighten"` , `"darken"` , `"multiply"` , `"screen"` . Passing `"normal"` is equivalent to calling `setAlphaBlendMode()` .

#### setColorFillMode()

Set the fill mode to draw a solid color, specified by the current color.

#### setTextureFillMode()

Set the fill mode to draw a texture, specified by the current texture, and using the alpha component of the current color as the opacity.

#### setSmoothLineFillMode()

Set the fill mode to draw smooth lines using the current color.

#### setColor(color)

Set the current color from a four-element array representing the RGBA components in [0, 1] range, e.g. `[1, 0, 0, 1]` for opaque red.

#### setColorRgba(r, g, b, a)

Set the current color by directly passing the RGBA components. in [0, 1] range.

**setOpacity(o)**

Set only the alpha component of the current color in [0, 1] range. Note this does not affect the RGB components.

**resetColor()**

Set the current color to opaque white (1, 1, 1, 1).

**setCullFaceMode(mode)**

**getCullFaceMode()**

Set or get the face culling mode, which may be one of the following strings:

- `"none"` : all faces are rendered
- `"back"` : back faces are culled
- `"front"` : front faces are culled

Whether a face counts as front or back depends on the front face winding (see `setFrontFaceWinding()` ). The default mode is `"none"` .

*Note that mirrored or flipped sprites are in fact showing a back face, which is why Construct defaults to* `"none"` .

**setFrontFaceWinding(mode)**

**getFrontFaceWinding()**

Set or get the front face winding, which is used to determine whether a triangle is front facing or back facing depending on the order ("winding") of the vertices. This is used by the cull face mode (see `setCullFaceMode()` ). The mode may be either the string `"cw"` for clockwise winding, or `"ccw"` for counter-clockwise winding. The default mode is `"cw"` , because most content that Construct renders itself uses clockwise winding.

**setCurrentZ(z)**

**getCurrentZ()**

Set and get the current Z component used for all 2D drawing commands that don't specify Z components, such as the `rect2()` and `quad3()` .

**rect(rect)**

Draw a rectangle given by an DOMRect.

**rect2(left, top, right, bottom)**

Draw a rectangle by directly passing the left, top, right and bottom positions.

**quad(quad)**

Draw a quad given by a DOMQuad.

**quad2(tlx, tly, trx, try_, brx, bry, blx, bly)**

Draw a quad by directly passing the positions of each of the four points in the quad.

**quad3(quad, rect)**

Draw a quad given by a DOMQuad, using a DOMRect for the source texture co-ordinates to draw from.

**quad4(quad, texQuad)**

Draw a quad given by an DOMQuad, using another `DOMQuad` for the source texture co-ordinates to draw from.

**quad5(quad, texQuad, colorArr)**

As with `quad4`, but adds a `colorArr` parameter for per-vertex colors. This must be a Float32Array with 16 elements in the order r, g, b, a for the top-left, top-right, bottom-right and bottom-left vertices, in that order.

**quad3D(tlx, tly, tlz, trx, try_, trz, brx, bry, brz, blx, bly, blz, rect)**

**quad3D2(tlx, tly, tlz, trx, try_, trz, brx, bry, brz, blx, bly, blz, texQuad)**

**quad3D3(tlx, tly, tlz, trx, try_, trz, brx, bry, brz, blx, bly, blz, texQuad, colorArr)**

Draw a 3D quad, specifying all four points of the quad with X, Y and Z co-ordinates. The first method accepts texture co-ordinates via a DOMRect *rect*. The second method accepts texture co-ordinates via a DOMQuad *texQuad*. The third is the same as the second, but adds a `colorArr` parameter for per-vertex colors. This must be a Float32Array with 16 elements in the order r, g, b, a for the top-left, top-right, bottom-right and bottom-left vertices, in that order.

**drawMesh(posArr, uvArr, indexArr, colorArr)**

Draw an array of textured triangles based on the given position, texture co-ordinate and index arrays, and an optional per-vertex color array. The `posArr` parameter must be a Float32Array of vertex positions in the sequence x, y, z (and therefore its size must be a multiple of 3). The `uvArr` parameter must be a Float32Array of texture co-ordinates in the sequence u, v (and therefore its size must be a multiple of 2). The `indexArr` parameter must be a Uint16Array of indices of vertices and texture co-ordinates, in the sequence i, j, k with each set defining a single triangle to be drawn (and therefore its size must be a multiple of 3). Note that indices refer to the index of a vertex, rather than a direct index in to either array, e.g. a position array with elements x1, y1, z1, x2, y2, z2 has six elements but defines two vertices, and so index 1 refers to the second vertex. If `colorArr` is specified, it must be a Float32Array of colors in the sequence r, g, b, a (and therefore its size must be a multiple of 4). These colors override the renderer's current color. If `colorArr` is not specified, it uses the renderer's current color for every vertex. For a code sample, see the section *Drawing meshes* below.

> *Note that for performance reasons, this method does not take in to account the layer Z elevation - all Z co-ordinates are used exactly as-is. If you wish to apply the layer Z elevation, you must offset all the position Z components yourself.*

---

**convexPoly(pointsArray)**

Draw a convex polygon using the given array of points, in alternating X, Y order. Therefore the size of the array must be even, and must contain at least six elements (to define three points).

---

**line(x1, y1, x2, y2)**

Draws a quad from the point (x1, y1) to (x2, y2) with the current line width.

---

**texturedLine(x1, y1, x2, y2, u, v)**

Draws a quad from the point (x1, y1) to (x2, y2) with the current line width, and using (u, 0) as the texture co-ordinates at the start, and (v, 0) as the texture co-ordinates at the end.

---

**lineRect(left, top, right, bottom)**

Draws four lines along the edges of a given rectangle.

---

**lineRect2(rect)**

Draws four lines along the edges of a given DOMRect.

---

**lineQuad(quad)**

Draws four lines along the edges of a given DOMQuad.

---

**pushLineWidth(w)**

**popLineWidth()**

Set the current line width for line-drawing calls. This must be followed by a `popLineWidth()` call when finished to restore the previous line width.

---

**pushLineCap(lineCap)**

**popLineCap()**

Set the current line cap for line-drawing calls. This must be followed by a `popLineCap()` call when finished to restore the previous line cap. The available line caps are `"butt"` and `"square"`.

---

**setTexture(texture)**

Set the current texture to a given ITexture.

---

**createStaticTexture(data, opts)**

Create an ITexture with the content specified by `data`. This method is asynchronous and so returns a Promise that resolves with the created `ITexture`. The `data` parameter may be one of `HTMLImageElement`, `HTMLCanvasElement`, `OffscreenCanvas` or `ImageBitmap`. (Note that in worker mode, only `OffscreenCanvas` and `ImageBitmap` are available.) `opts` specifies options for the texture - see the section *Texture options* above for more details.

> *Static textures do not support changing their content, and are optimized accordingly. If you want to be able to change the content of a texture, use `createDynamicTexture()`.*

---

**createDynamicTexture(width, height, opts)**

Create a new empty ITexture for dynamic use, i.e. expecting the texture content to be replaced using `updateTexture()`. The size of the texture is given by `width` and `height` which must be positive integers. `opts` specifies options for the texture - see the section *Texture options* above for more details.

---

**updateTexture(data, texture, opts)**

Upload *data* as the new texture contents for the ITexture *texture*. This can only be used for textures created with `createDynamicTexture()` and managed by your addon.
*data* can be one of the following types: `HTMLImageElement`, `HTMLVideoElement`, `HTMLCanvasElement`, `ImageBitmap`, `OffscreenCanvas` or `ImageData`. Note in worker mode the DOM types cannot be used (`HTMLImageElement`, `HTMLVideoElement`, `HTMLCanvasElement`); in this case use `ImageBitmap` or `OffscreenCanvas` instead. This method cannot resize an existing texture, so the data must match the size the texture was created with; if the size needs to change, destroy and re-create the texture.
*opts* specifies options for the texture upload which is an object that can include the following properties:

- `premultiplyAlpha` : a boolean indicating whether to premultiply alpha of the image content specified by *data* (default true). Construct always renders using premultiplied alpha so this is normally necessary; however if the data is known to already be premultiplied, set this to false.

**deleteTexture(texture)**

Delete an ITexture, releasing its resources. This can only be used for textures created with `createDynamicTexture()` and managed by your addon. Do not attempt to delete textures managed by the Construct engine.

**async loadTextureForImageInfo(imageInfo, opts)**

For use with the addon SDK. Load a texture for a given IImageInfo. Returns a promise that resolves with the loaded ITexture. `opts` specifies options for the texture - see the section *Texture options* above for more details.

**releaseTextureForImageInfo(imageInfo)**

For use with the addon SDK. Release a texture for a given IImageInfo that was previously loaded with `loadTextureForImageInfo()` .

**getTextureForImageInfo(imageInfo)**

For use with the addon SDK. Returns the existing ITexture for a given IImageInfo that was previously loaded with `loadTextureForImageInfo()` , or returns `null` if no texture is loaded (or the texture is still asynchronously loading).

**createRendererText()**

Return a new IRendererText interface. This manages text wrapping, drawing text, and uploading the results to a WebGL texture.

**setDeviceTransform()**

Set the co-ordinate system to be in device transform mode, which is in units of device pixels and relative to the screen. This can be useful for achieving pixel-perfect rendering.

**setLayerTransform(layer)**

Set the co-ordinate system to match the given ILayer. This is the default mode - this method is normally called after `setDeviceTransform()` to restore normal rendering.

## Drawing meshes

The `drawMesh()` method allows passing typed arrays with vertex, texture co-ordinate, index, and optionally color data, for efficient rendering of entire meshes. Note that the mesh drawn with this method uses the same renderer state (blend mode, fill mode, color and texture) for all triangles. To draw parts of the mesh with different renderer state, you will need to make multiple calls to `drawMesh()` with other calls to change state in between. You can create typed arrays over different ranges of the same ArrayBuffer to draw sections of a mesh - see the MDN guide on TypedArray for more details.

Here is some sample code for using `drawMesh()`. This draws four triangles arranged as two quads with the single call to `drawMesh()`. This code assumes `quad` is a DOMQuad of the first quad to draw, and the second quad is drawn shifted 200px to the right; and that `rcTex` is the texture co-ordinates to use (which is repeated for both quads).

```
// Vertex positions as sequence of x, y, z
const posArr = new Float32Array([
        quad.p1.x, quad.p1.y, 0,
        quad.p2.x, quad.p2.y, 0,
        quad.p3.x, quad.p3.y, 0,
        quad.p4.x, quad.p4.y, 0,

        quad.p1.x + 200, quad.p1.y, 0,
        quad.p2.x + 200, quad.p2.y, 0,
        quad.p3.x + 200, quad.p3.y, 0,
        quad.p4.x + 200, quad.p4.y, 0
]);

// Texture co-ordinates as sequence of u, v
const uvArr = new Float32Array([
        rcTex.left, rcTex.top,
        rcTex.right, rcTex.top,
        rcTex.right, rcTex.bottom,
        rcTex.left, rcTex.bottom,

        rcTex.left, rcTex.top,
        rcTex.right, rcTex.top,
        rcTex.right, rcTex.bottom,
        rcTex.left, rcTex.bottom
]);

// Indices of vertices in sequence i, j, k
const indexArr = new Uint16Array([
        0, 1, 2,
        0, 2, 3,

        4, 5, 6,
        4, 6, 7,
]);

renderer.drawMesh(posArr, uvArr, indexArr);
```

Also note that the usual drawing methods (e.g. quad methods) automatically apply the current layer's Z elevation. For performance reasons, drawing meshes does not do this automatically, and will just use the Z components in the positions array as absolute values. If you want the content to move with layer Z elevation, you'll need to offset the Z components in the positions array yourself.