# orxSHADER structure

## Summary

```
[ShaderTemplate]
Code = "// Shader code
  void main()
  {
   // Do stuff
  }"
KeepInCache    = <bool>
ParamList      = ParamFloat # ParamTexture # ParamVector
ParamFloat     = <float>
ParamVector    = <vector>
ParamTexture   = path/to/TextureFile|screen
UseCustomParam = <bool>
```

## Details

Here's a list of the available properties for an `orxSHADER` structure:

- `Code`: This block [1] contains the code that will be executed. It needs to be provided and be valid GLSL fragment shader code.
- `KeepInCache`: Defines if the shader code should be kept in memory even if no shader of this type is currently in use. This saves time (reading from disk + compiling) but costs memory. Its default value is `false`.
- ParamList: This defines the list of parameters needed and used by the shader's code. Every defined parameter must have a default value that will help orx guess their type. If none is provided, then its type will be assumed to be a texture. Available types are <float>, <vector> and texture (if a path to a texture file or the keyword `screen` is provided). If an invalid path is provided for a parameter, or the parameter isn't defined at all, the owner's texture will be used [2]. **If an explicit list is provided for any parameter, the shader variable will be an array of this parameter type (instead of a regular variable) and its size will be the number of items in the list.**
- `UseCustomParam`: Defines if parameters can have their value overridden at runtime (ie. interactive). Its default value is `false` which means only the default values will be used.

Here's a simple example of a non-interactive shader as seen in the spawner/shader tutorial.

```
[Decompose]
Code = "void main()
{
  float fRed, fGreen, fBlue;

  // Computes positions with offsets
  vec2 vRedPos    = vec2(gl_TexCoord[0].x + offset.x, gl_TexCoord[0].y +
offset.y);
  vec2 vGreenPos  = vec2(gl_TexCoord[0].x, gl_TexCoord[0].y);
```

```
  vec2 vBluePos   = vec2(gl_TexCoord[0].x - offset.x, gl_TexCoord[0].y -
offset.y);

  // Red pixel inside texture?
  if((vRedPos.x >= 0.0) && (vRedPos.x <= 1.0) && (vRedPos.y >= 0.0) &&
(vRedPos.y <= 1.0))
  {
    // Gets its value
    fRed = texture2D(texture, vRedPos).r;
  }

  // Green  pixel inside texture?
  if((vGreenPos.x >= 0.0) && (vGreenPos.x <= 1.0) && (vGreenPos.y >= 0.0) &&
(vGreenPos.y <= 1.0))
  {
    // Gets its value
    fGreen = texture2D(texture, vGreenPos).g;
  }

  // Blue pixel inside texture?
  if((vBluePos.x >= 0.0) && (vBluePos.x <= 1.0) && (vBluePos.y >= 0.0) &&
(vBluePos.y <= 1.0))
  {
    // Gets its value
    fBlue = texture2D(texture, vBluePos).b;
  }

  // Outputs the final decomposed pixel
  gl_FragColor = vec4(fRed, fGreen, fBlue, 1.0);
}"
ParamList = texture # offset
offset    = (-0.05, -0.05, 0.0) ~ (0.05, 0.05, 0.0); <= Let's take some
random offset
```

Please see the spawner/shader tutorial for more information.

### Overriding Parameters at Runtime with UseCustomParam

Shader parameters can be defined on the fly if

```
UseCustomParam = true
```

is set in your shader. An event of type `orxEVENT_TYPE_SHADER` and ID `orxSHADER_EVENT_SET_PARAM` will be fired for all parameters and its payload will contain the name of the param and its default value. Event handler can then modify that value if need be, and it'll get used by the shader.

However, when UseCustomParam is defined to true, those objects can't be batched at rendering, making the rendering phase more expensive. The severity of the processing penalty depends on how many affected objects are displayed. See the test/playground code, orxBounce, for an example on how to set those shader parameters on the fly.

## Shader Execution Environment

### Using built-in 'time' keyword as parameter argument

“time” is a keyword recognized by orx: the parameter value will be the object's “age”, in seconds. Example:

```
ParamList = fTime
fTime = time
```

### Using the internal 'pixel' texture

There is an internal texture called `pixel`. It can be used to specify an image of arbitrary size when used with the Scale property of the object:

```
[Object]
Graphic = MyTexture
Scale = (16, 16, 1)

[MyTexture]
Texture = pixel
```

In the example above, an Object has a Graphic that will span over 16×16 pixels.

### Coordinate System

Shaders contain implicit parameters containing owner's texture coordinates. For example:

```
[GameObject]
Graphic = @
Texture = ObjectTexture.png
TextureOrigin = (16, 16, 0)
TextureSize = (8, 8, 0)
ShaderList = Shader

[Shader]
ParamList = MyTexture # ...
Code = ...
```

Then orx will generate extra parameters behind the scene regarding the texture MyTexture. The names follow the pattern:

```
<NameOfTexture>_top, <NameOfTexture>_left,
<NameOfTexture>_bottom, <NameOfTexture>_right
```

In the above example, the names will then be:

```
MyTexture_top, MyTexture_left, MyTexture_bottom, MyTexture_right
```

If MyTexture's dimensions are 32×32, we'd then get:

```
MyTexture_top = (32 - 16) / 32 => 0.5
MyTexture_left = 16 / 32 => 0.5
MyTexture_bottom = (32 - (16 + 8)) / 32 => 0.25 (it has been automatically
inverted for the user)
MyTexture_right = (16 + 8) / 32 => 0.75
```

1)

delimited by double quotes (“) as seen in the syntax page

2)

if the owner is a viewport, it will be its associated texture; if it's an object, it's current graphic/animation key's texture will be used

From:
https://www.orx-project.org/wiki/ - **Orx Learning**

Permanent link:
**https://www.orx-project.org/wiki/en/orx/config/settings_structure/orxshader?rev=1368258990**

Last update: **2025/09/30 17:26 (14 months ago)**