MXwendler
Fragment Shader Development Reference Version 1.0

This document describes the MXwendler fragmentshader interface. You will learn how to write shaders using the GLSL language standards and the MXW runtime data exchange interface. This document does not describe the development of GLSL shaders in general.

06.05/Hendrik Wendler

1.0 What are fragment shaders?

Vertex and pixel (or fragment) shaders are shaders that run on a graphics card, executed once for every vertex or pixel in a specified 3D mesh. They operate in the context of interactively rendering a 3D scene, usually using either the Direct3D or OpenGL API. Flexible shaders were initially used for non-interactive rendering applications like Pixar's RenderMan, which was designed to render film-quality images.

Fragment shaders operate after the geometry pipeline and before final rasterization. They operate in parallel with multitexturing to produce a final pixel color and z-value for the final, rasterization step in the graphics pipeline (where alpha blending and depth testing occur). (taken from Wikipedia)

Fragment shaders and pixel shaders are basically the same. A fragment is a block of pixels calculated at once, because they share the same vertex boundaries, that is the only difference.

MXwendler uses fragment shader to apply effects to footage and to combine the resulting visuals in various ways. Using this fragment shader has several advantages like impressive performance and quality. A disadvantage may be the resulting inflexibility. A resulting visual is available in video memory but not in system memory, which may be undesireable in case of stacked effect chains, time-based effects or motion analysis.

1.1 General fragment shader aspects

Fragment shaders perform color operations at a remarkable high speed. This performance is achieved through – among other concepts – a simple and well known concept: massive parallel computing. Recent graphic accelerators offer 12 to 24 pixel pipelines. This means, they can compute 12 to 24 pixels at once. To achieve this speed, each pixel color has to be calculated completely independent. It is obvious, that the 24th pixel processor cannot wait for the results of the first pixel processor. So you should be aware of certain fragment shader development aspects:

a pixel process cannot retrieve the result of another pixel process
a pixel process cannot retrieve the value of a previous frame
a pixel process cannot store a value for the next frame
a fragment shader is length limited[1]
a fragment shader operates with a limited instruction set[2]
a fragment shader cannot access values except the passed uniforms

1.2 What do i need to write a MXwendler fragment shader?

All you need is an ascii text editor, like notepad, wordpad or ultra-edit. You need a GLSL (Open**GL S**hader **L**anguage) command reference[3]. Since GLSL is a programming language, there are many helpful tools. There is a GLSL syntax checker available from 3DLabs[4]. You will find a copy of this tool in your Program/effects folder. With this tool, you can open a fragment shader, and check it for correct syntax. Helpful may be a collection of keywords for the highlighting capabilities of your editor. An example for Ultra Edit can be found here[5].

1.3 MXwendler fragment shader integration concept

On startup, MXW checks for a folder called „effects“ relative to the working directory. All files ending with „.mxf“ will be loaded. Loading of .mxf files beginning with „_“ will be skipped. An easy way to disable a specific shader is appending „_“ to its name. Loading itself does not compile the shaders. A shader will be compiled once it is activated. Any erroneous shader will be disabled, and the user will receive a message that „shader link failed“.

A MXW shader is a fully GLSL compliant fragment shader. It should compile in any OpenGL application. MXW shaders are backward compatible to OpenGL.

MXW is a video performance application – the applied effects need GUI accessible parameters. On startup, MXW loads the shaders, and creates widgets for every uniform variable following this naming scheme:

```
uniform float mxw_vertslider_description_0x0_1x0_0x5_mxw;
```

This variable name will be parsed into the following segments:

`mxw[_]`	-	a mxw fragment shader uniform variable
`vertslider[_]`	-	create a vertical slider
`description[_]`	-	descriptive name of parameter
`0x0[_]`	-	minimum value 0.0
`1x0[_]`	-	maximum value 1.0
`0x5[_]`	-	initial widget value 0.5
`mxw`	-	end with mxw

Whenever an effect containing this uniform variable is activated, the related clip will show up a vertical slider with the predefined initial value. On each render cycle the fragment shader uniform variable will receive the interpolated value

```
(max-min)*widgetvalue + min
```

This calculation is done in the host application to save fragment shader instructions. A variable

```
uniform float mxw_vertslider_description_20x50_40x0_0x5;
```

would receive initially the value 30.25 [(40-20.5)*0.5 +20.5]. Please note that the initializing value is normalized.

2.0 What would be a very simple fragment shader?

The simplest possible shader would be a shader creating a red image:

```
void main(void)
{
     vec4 color = vec4(0.0);
     color.r = 1.0;
     color.a = 1.0;
     gl_FragColor = color;
}
```

The first line after the declaration creates a valid 4-component (RGBA) color variable. The next two lines set the red (`color.r`) and alpha (color.a) components to 1.0 (maximum). A fragment shader must have a main() function which assignes a vec4 color vector to a special variable named gl_FragColor. The main() function is – as known from the C language – the entry point for the fragment shader. The final assignment to gl_FragColor is the result of the fragment shader which will be passed to the following OpenGL pipeline stages (Fog, Depth Cueing etc.). The resulting image of the sample shader is plain red.

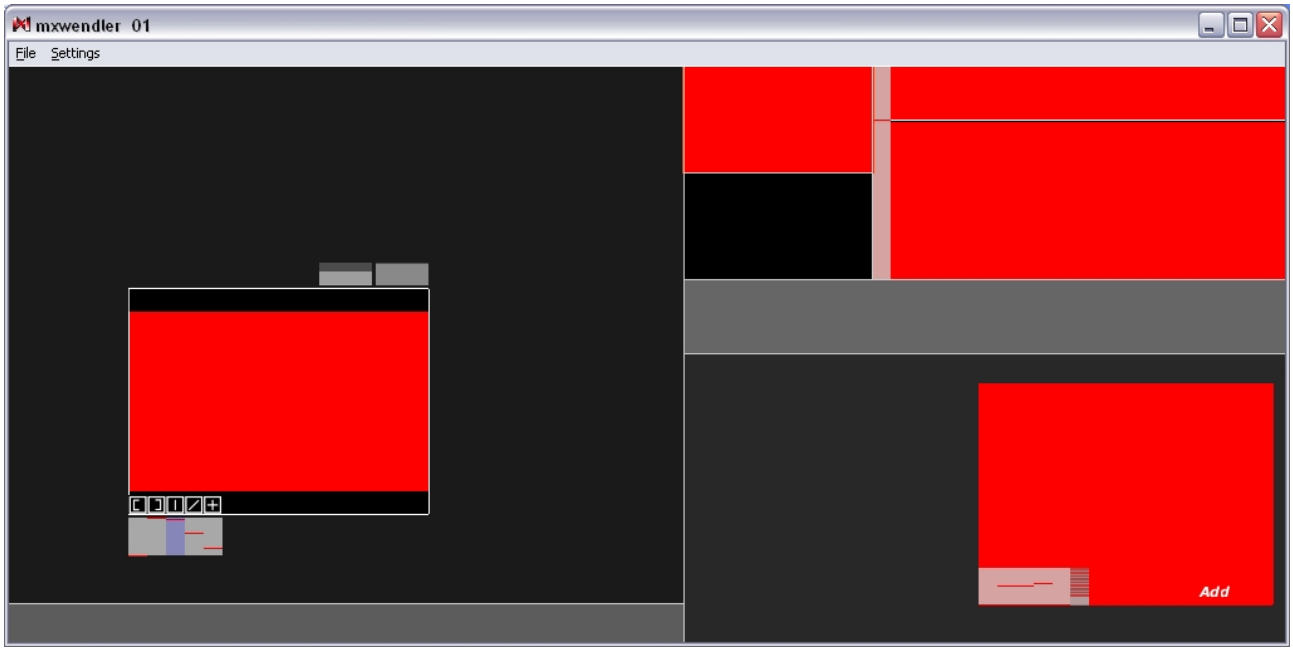

Fig 1: Simple shader creating red pixels

Slightly more complicated is the integration of texture data. Texture handles are always delivered through uniform variables called samplers:

```
uniform sampler2D texUnit;
void main(void)
{
   vec2 texCoord = gl_TexCoord[0].xy;
   vec4 c = texture2D(texUnit, texCoord);
   gl_FragColor = c;
}
```

The line before the declaration of the main function declares a texture handle named texUnit. MXwendler will assign the footage texture to this texture unit.

The line beginning with „vec2“ creates a two-value vector containing the current texture coordinate using a special GLSL variable called gl_TexCoord. The term glTexCoord[0].xy means: give the x and x value (OpenGL knows 1D and 3D – textures as well) of the first texture unit (by now MXW uses only one texture unit) and put it into the 2-component vector texCoord. The values of texCoord will change with every pixel.

The line beginning with „vec4“ retrieves a RGBA color value in form of a 4-component vector by doing a texture lookup at the texture position specified by texCoord.

The line beginning with „gl_FragColor“ assigns the texture lookup value to the fragment shader / fragment processor output. The result of this shader is a simple reproduction of the footage image data:

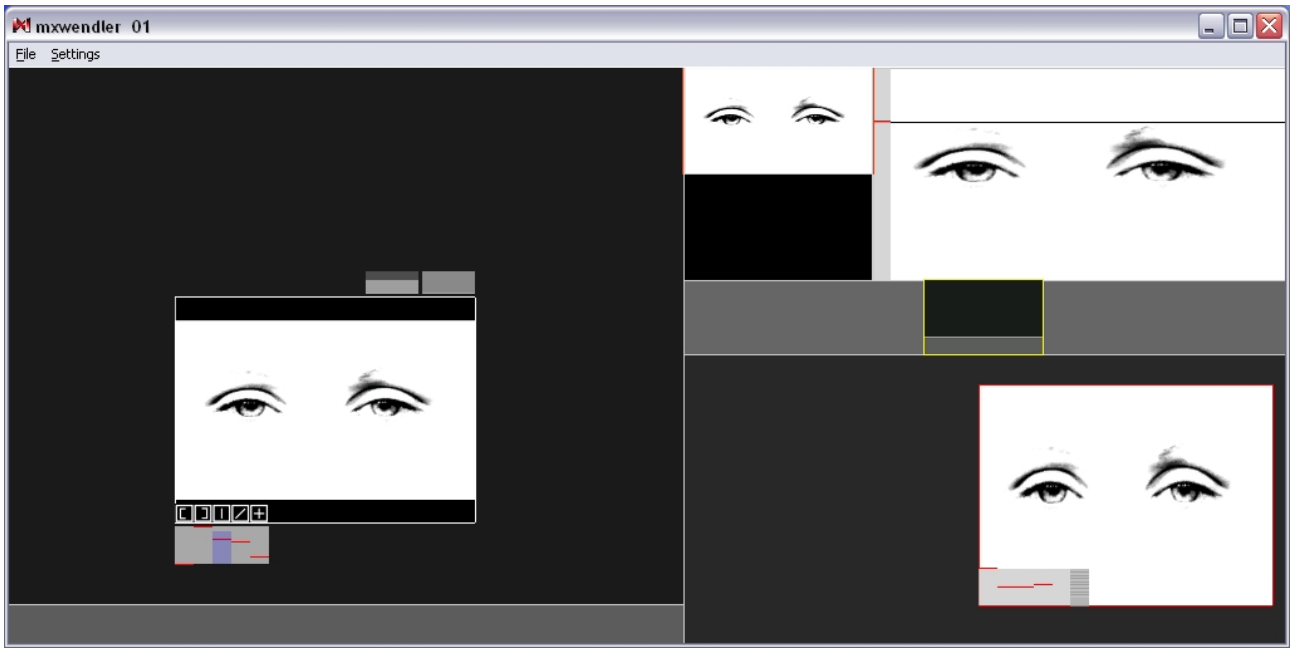

Fig 2: Simple shader II: show movie texture color

2.1 How do i pass values to the fragment shaders?

There are several reasons to pass values to the fragment shaders. First of all, you want to pass your parameter values to the shader. If you develop a eg. contrast shader, you want to control the amount of contra st enhancement your shader does. There are more resons: maybe you want to create a time-based shader, and you want to know the simulation time. Or you want to know the maximum texture coordinates. MXW automatically parses the fragment shaders for the following uniforms and passes the related values on each render cycle:

`mxw_millis`	– the amount of milliseconds since application start
`mxw_maxV`	- the maximum V coordinate
`mxw_maxU`	- the maximum U coordinate
`mxw_texSize`	- the POT size of the texture

To use one of these values, simply declare a uniform value the following way:

```
uniform float mxw_millis;
```

This uniform variable will receive its value on each render cycle. To create, control and receive custom values, declare uniform variables as described in 1.3.

2.2 How do i create an interface to the fragment shader?

If you want to control the behaviour of your fragment shader, you have to declare a set of variables as described above. For each uniform variable as described in 1.3 a vertical slider will appear.

For example, the fragment shader code of the histogram shader is:

```
uniform sampler2D texUnit;
uniform float mxw_vertslider_b_0x0_1x0_0x0_mxw;
uniform float mxw_vertslider_m_0x2_5x0_0x2_mxw;
uniform float mxw_vertslider_t_0x0_1x0_1x0_mxw;

void main(void)
{
     /*
     convert to lesser tedious vars
     */
     float b = mxw_vertslider_b_0x0_1x0_0x0_mxw;
     float m = mxw_vertslider_m_0x2_5x0_0x2_mxw;
     float t = mxw_vertslider_t_0x0_1x0_1x0_mxw;

     /*
     lookup
     */
     vec4 col = texture2D(texUnit, gl_TexCoord[0].xy);

     /*
     now do correction:
     - scale color space up to top-bottom ~ 1.0
     - subtract bottom
     - then apply weight (simple pow)
     */
     vec3 m3 = vec3(m);

     col.rgb -= b;
     col.rgb *= t>b?(1.0/t-b):1.0;
     col.rgb = pow(col.rgb,m3);

     gl_FragColor = col;

}
```

The four uniform variables declare:

a texture unit
a vertical slider for the bottom, ranging 0..1 with init 0.0
a vertical slider for the weight, ranging 0.2..5.0 with init 1.16 (widget 0.2)
a vertical slider for the top, ranging 0..1 with init 1.0

At first a texture lookup is performed. Then a series of calculations scales the results to the limits defined by the sliders 1 and 3. A third calculation performs a component-wise pow() to apply a weight to the gradient.

When you activate this shader, you will see three slider on top of the related clip:

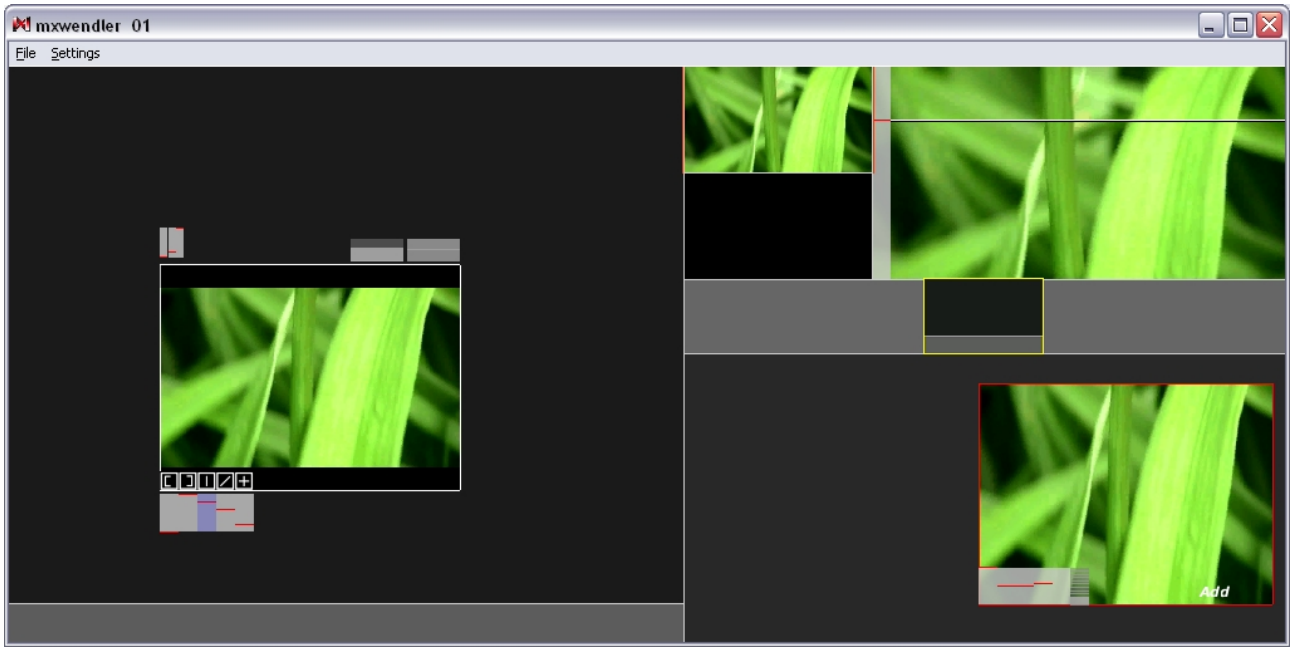

Fig 3: Shader with three vertical sliders to control the behaviour

With these three sliders you can control the complete behaviour of the histogram shader. It is possible to create more or less than three control sliders for a shader.

3.0 Shader development guidelines

Fragment shaders run in a highly optimized but limited hardware environment. There are some characteristics you should consider when you develop fragment shaders, especially for MXWendler.

The most important development guideline by now (Q1/2005) is:

Keep Shaders As Short As Possible

Current consumer hardware up to Nvidia's Geforce 6X-line and ATI's support Shader Model 2.0. This means – among other restrictions – the maximum fragment shader length is limited to 128 to 160 instructions. If you combine multiple layers with long fragment shaders, the overall length of the shaders will exceed the hardware limit, and the renderer will fall back to software rendering with a performance loss of 95%. As a rule of thumb, you should check if *three of your shaders* combined in the layer manager with the „add mode“ run at full hardware speed.

Another very important guideline is:

Check Your Implementation

Current implementations of the OpenGL 1.5 standard have partially very poor support for the functionality announced in the standard specification. Do not rely on hardware supported control flow. Sometimes loops are expanded by the compiler if the loop count is determinable, but once the loop condition depends on runtime conditions the whole shader is executed in software.

To keep the shader sets stable:

Check shader combinations

Sometimes shader do not behave very well in combination with others. Always combine your shaders with as much shaders from different origins as possible. If an effect does not behave well, do not distribute it.

To optimize performance:

Check for approximations

Most of the times you do not need expensive trigonometric functions for simple manipulations. Eg. atan(x) can be approximated by a variant of y = 1.0/(exp(-x)+1.0), a continous function that runs several faster on a recent GPU[6].

To optimize rendering quality:

Check for image quality

Check your shaders against dark and light footage, high and low resolution footage, fast and slow motions. Check for unwanted moiree and other patterns in different output resolutions. Check for color gradient continuity especially in very dark and light cases.

Its cross-platform:

Use ASCII and unix line-endings (LF only)

mxf shaders are designed to work on Windows, MacOSX and Linux. MXwendler can read any type of lineendings, but your system tools can not. To avoid confusion during creation, editing and serving shaders across platforms, please use the Unix lineendings. On Linux and OSX you can use any editor (use LF natively), on Windows use a free editor like crimson. Furthermore GLSLang compilers accross all platforms only accept ASCII code (no UNICODE, UTF8 ...).

Claim your credits:

Embed your initials

Apart from the preview, the descriptive shader name is the only way for a user to keep track of his active shaders. Prepending your initials will allow a user to quickly detect shader sets and their origin. If shader combinations (see above) tend to fail, it is much easier for a user to track down misbehaving shaders. Nevertheless, this is an opportunity to associate good shaders with your initials: „blur.mxf“ -> „bb_blur.mxf“

1. Shader Model 2.0 allows about 160 instructions, Shader Model 3.0 allows up to 65536 instructions in the fragment shader.
2. Fragment shaders are limited to the GLSL command set. It is not possible to link against external libraries.
3. http://opengl.org/documentation/specs/version2.0/glspec20.pdf
4. http://developer.3dlabs.com/downloads/glslvalidate/
5. http://www.mindcontrol.org/~hplus/ultra-edit-glsl-wordfile.txt
6. A nice (german) site to visualize function graphs: http://www.arndt-bruenner.de/mathe/java/plotter.htm