

# The AnyFX Effects Language specification, version 1.3

Gustav Sterbrant

# Contents

|          |                                    |           |
|----------|------------------------------------|-----------|
| <b>1</b> | <b>Compiler</b>                    | <b>3</b>  |
| 1.1      | Keywords . . . . .                 | 3         |
| 1.2      | Expressions . . . . .              | 3         |
| 1.3      | Profile . . . . .                  | 4         |
| 1.4      | Variables . . . . .                | 4         |
| 1.4.1    | Initialization . . . . .           | 5         |
| 1.4.2    | Arrays . . . . .                   | 5         |
| 1.4.3    | Compute shader variables . . . . . | 7         |
| 1.5      | Constants . . . . .                | 7         |
| 1.6      | Render states . . . . .            | 9         |
| 1.7      | Functions . . . . .                | 10        |
| 1.8      | Subroutines . . . . .              | 14        |
| 1.9      | Varblocks . . . . .                | 15        |
| 1.10     | Varbuffers . . . . .               | 16        |
| 1.11     | Programs . . . . .                 | 17        |
| 1.12     | Structures . . . . .               | 19        |
| 1.13     | Samplers . . . . .                 | 20        |
| 1.14     | Annotations . . . . .              | 21        |
| 1.15     | Usage . . . . .                    | 21        |
| <b>2</b> | <b>API</b>                         | <b>23</b> |
| 2.1      | Initialization . . . . .           | 23        |
| 2.2      | Effect . . . . .                   | 23        |
| 2.2.1    | EffectProgram . . . . .            | 23        |
| 2.2.2    | EffectShader . . . . .             | 23        |
| 2.2.3    | EffectRenderState . . . . .        | 23        |
| 2.2.4    | EffectVariable . . . . .           | 23        |
| 2.2.5    | EffectVarblock . . . . .           | 24        |
| 2.2.6    | EffectVarbuffer . . . . .          | 24        |
| 2.2.7    | EffectSubroutine . . . . .         | 24        |
| 2.2.8    | Annotations . . . . .              | 24        |

# 1 Compiler

This section describes the grammar, keywords and structure which complies to the AnyFX language. For future reference, syntax definition will look as such:

**Mandatory** *Optional* Keyword

## 1.1 Keywords

AnyFX reserves some keywords, these keywords can be found in Table 1.

Table 1: Keywords

|              |
|--------------|
| const        |
| true         |
| false        |
| in           |
| out          |
| inout        |
| program      |
| samplerstate |
| state        |
| struct       |
| varblock     |
| patch        |

## 1.2 Expressions

The compiler can perform static expression evaluation. This may come in handy when using preprocessor macros which may alter the result of the effect during compile time. Expressions can use values in any of the default types, that is *float*, *bool* and *int*. An example of expression usage could be:

```
#define NUMINSTANCES 5

[inputprimitive] = triangles
[outputprimitive] = triangle_strip
[maxvertexcount] = 3 * NUMINSTANCES

shader
void
gsStatic(in vec3 Position[], in vec2 inUV[], out vec2 outUV, out vec3 outTriDistance)
{
    ...
}
```

For the mathematical types *float* and *int*, the binary operators  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$ ,  $==$  are supported. The compiler will statically evaluate and implicitly convert between float, double and int. Unary operator  $-$  is also supported for these types.

Booleans have a smaller subset of operators, namely  $\neq$  and  $==$ . The only unary operator supported for booleans is  $!$  which gets the conjugate of the boolean.

### 1.3 Profile

Since AnyFX doesn't analyze the executing code by itself, it needs to be provided a target language to which AnyFX code can be translated into. Table 2 shows the currently implemented translation targets.

Table 2: Profiles

| Status          | Target | Version       |
|-----------------|--------|---------------|
| Implemented     | gl44   | OpenGL 4.4    |
| Implemented     | gl43   | OpenGL 4.3    |
| Implemented     | gl42   | OpenGL 4.2    |
| Implemented     | gl41   | OpenGL 4.1    |
| Implemented     | gl40   | OpenGL 4.0    |
| Not implemented | gl33   | OpenGL 3.3    |
| Not implemented | gl32   | OpenGL 3.2    |
| Not implemented | gl31   | OpenGL 3.1    |
| Not implemented | gl30   | OpenGL 3.0    |
| Not implemented | gl20   | OpenGL 2.0    |
| Not implemented | gl21   | OpenGL 2.1    |
| Not implemented | gl15   | OpenGL 1.5    |
| Not implemented | gl14   | OpenGL 1.4    |
| Not implemented | gl13   | OpenGL 1.3    |
| Not implemented | gl12   | OpenGL 1.2    |
| Not implemented | gl121  | OpenGL 1.2.1  |
| Not implemented | gl11   | OpenGL 1.1    |
| Not implemented | gl10   | OpenGL 1.0    |
| Not implemented | dx11   | DirectX 11    |
| Not implemented | dx10   | DirectX 10    |
| Not implemented | dx9    | DirectX 9     |
| Not implemented | ps4    | Playstation 4 |
| Not implemented | ps3    | Playstation 3 |
| Not implemented | wiiu   | Wii U         |
| Not implemented | wii    | Wii           |

### 1.4 Variables

Variables are declared using the syntax:

```
qualifier0 ... qualifierN type name [ Annotation ] ;
```

Here, **type** can be any of the entries in Table 3. For future reference, whenever the **type** keyword is encountered, it is assumed to be any value in this table. Variables supports qualifiers, and the two currently existing qualifiers are image format qualifiers and data access qualifiers.

Note that we have two sets of definitions at the moment, one for OpenGL and one for DirectX. AnyFX doesn't care about which 'style' you use, although it will throw a warning if compiled with a mismatching profile, so as to inform the user that they are using a perhaps confusing variable syntax. An example of this would be:

```
float4 color;
...
vec4 func()
{
    return color;
}
```

In which we combine DirectX style variables (**float4**) with OpenGL style functions (**vec4**). This will cause a warning.

### 1.4.1 Initialization

Variables can also be initialized with a default value, for which the syntax is:

```
type name [ Annotation ] = type(expressions) ;
```

Here, *expressions* is a linear list of data, an example of an initialization list would be:

```
vec4 color = vec4(1.0f, 0.0f, 0.0f, 1.0f);
```

Which gives us a float vector of size 4 where:

$$\begin{pmatrix} x = 1.0 \\ y = 0.0 \\ z = 0.0 \\ w = 1.0 \end{pmatrix}$$

We can also apply the initializer shorthand, which only works for the default types *float*, *double*, *int*, *bool*, *short*, *uint*. It follows this syntax

```
type name [ Annotation ] = expression;
```

Here, *expression* is assumed to be the same as *type*. If this is not the case, a compilation error will be thrown. Example code could be:

```
float f = 1.0f;
```

Do also note that whenever a variable gets set from the API, the default value gets replaced and cannot be retrieved, and thus only exists as a convenient way to represent a 'default' state of the variable.

### 1.4.2 Arrays

Variables can also be of array type, for which the syntax is:

```
type name [ Annotation ] [ size expression ];
```

Table 3: Variable types

| AnyFX Type                                    | OpenGL                      | DirectX                         |
|---|-----------------------------|---------------------------------|
| float   | float                       | float                           |
| float[2]                                      | vec2                        | float2                          |
| float[3]                                      | vec3                        | float3                          |
| float[4]                                      | vec4                        | float4                          |
| double  | double                      | double                          |
| double[2]                                     | dvec2                       | double2                         |
| double[3]                                     | dvec3                       | double3                         |
| double[4]                                     | dvec4                       | double4                         |
| int   | int                         | int                             |
| int[2]  | ivec2                       | int2                            |
| int[3]  | ivec3                       | int3                            |
| int[4]  | ivec4                       | int4                            |
| unsigned int                                  | uint                        | uint                            |
| unsigned int[2]                               | uvec2                       | uint2                           |
| unsigned int[3]                               | uvec3                       | uint3                           |
| unsigned int[4]                               | uvec4                       | uint4                           |
| short   | short                       | short                           |
| short[2]                                      | svec2                       | short2                          |
| short[3]                                      | svec3                       | short3                          |
| short[4]                                      | svec4                       | short4                          |
| bool  | bool                        | bool                            |
| bool[2]                                       | bvec2                       | bool2                           |
| bool[3]                                       | bvec3                       | bool3                           |
| bool[4]                                       | bvec4                       | bool4                           |
| matrix 2x2 (float[4])                         | mat2x2 (alternatively mat2) | float2x2                        |
| matrix 2x3 (float[6])                         | mat2x3                      | float2x3                        |
| matrix 2x4 (float[8])                         | mat2x4                      | float2x4                        |
| matrix 3x2 (float[6])                         | mat3x2                      | float3x2                        |
| matrix 3x3 (float[9])                         | mat3x3 (alternatively mat3) | float3x3                        |
| matrix 3x4 (float[12])                        | mat3x4                      | float3x4                        |
| matrix 4x2 (float[8])                         | mat4x2                      | float4x2                        |
| matrix 4x3 (float[12])                        | mat4x3                      | float4x3                        |
| matrix 4x4 (float[16])                        | mat4x4 (alternatively mat4) | float4x4                        |
| 1D Texture object                             | sampler1D                   | Texture1D                       |
| 1D Texture object array                       | sampler1DArray              | Texture1DArray                  |
| 2D Texture object                             | sampler2D                   | Texture2D                       |
| 2D Texture object array                       | sampler2DArray              | Texture2DArray                  |
| 2D Texture object multisampled                | sampler2DMS                 | Texture2DMS                     |
| 2D Texture object multisampled array          | sampler2DMSArray            | Texture2DMSArray                |
| 3D Texture object                             | sampler3D                   | Texture3D                       |
| Cube Texture object                           | samplerCube                 | TextureCube                     |
| Cube Texture object array                     | samplerCubeArray            | TextureCubeArray                |
| 1D Read-Write access image                    | image1D                     | RWTexture1D                     |
| 1D Read-Write access image array              | image1DArray                | RWTexture1DArray                |
| 2D Read-Write access image                    | image2D                     | RWTexture2D                     |
| 2D Read-Write access image array              | image2DArray                | RWTexture2DArray                |
| 2D Read-Write access image multisampled       | image2DMS                   | <i>not available in DirectX</i> |
| 2D Read-Write access image multisampled array | image2DMSArray              | <i>not available in DirectX</i> |
| 3D Read-Write access image                    | image3D                     | RWTexture3D                     |
| 3D Read-Write access image array              | image3DArray                | <i>not available in DirectX</i> |
| Cube Read-Write access image                  | imageCube                   | <i>not available in DirectX</i> |
| Cube Read-Write access image array            | imageCubeArray              | <i>not available in DirectX</i> |
| Compute shader atomic counter                 | atomic_uint                 | <i>not available in DirectX</i> |

Note that arrays can also be initialized using this syntax:

```
type name [ Annotation ] [ size expression ] = { type(expression1), type(expression2), type(expression3) } ;
```

We can also initialize variable lists with basic type, for which we don't need to specify which type each element is. The syntax is:

```
type name [ Annotation ] [ size expression ] = { expression1, expression2, expression3 } ;
```

However, this is only valid for the basic types *int*, *bool* and *float*. These array initializers use the verbose syntax, meaning that the size expression must be equal to the number of qualifiers. We can also initialize arrays by omitting the *size* expression, which tells AnyFX to create an array with a size equal to the number of initializers. An example could be:

```
float verboseArray [4] = { 1.0f, 2.0f, 3.0f, 4.0f };  
float lazyArray []      = { 1.0f, 2.0f, 3.0f, 4.0f };
```

These two arrays are equal in size and data, however *verboseArray* explicitly defines the size, while *lazyArray* implicitly defines it.

### 1.4.3 Compute shader variables

Some variable types, namely the image types are considered specialized for compute shaders. An image variable denotes a buffer which is treated as a texture, however it can be accessed and written to using special methods. As such, an image denotes a texture-backed data buffer, which can be used for data storage and reads from a compute shader.

Every compute shader variable (as of version 1.0, this only applies to images) has a special qualifier called an *access qualifier*. The access qualifier can be any of the values in Table 4

Table 4: Access modes

| Syntax    | Functionality                        |
|-----------|--------------------------------------|
| read      | Image can only be read from          |
| write     | Image can only be written to         |
| readwrite | Image supports both reads and writes |

Also, image variables have yet another special qualifier called an *image format qualifier*. They are used by the shaders to explain how each pixel should be interpreted by the compute shader. Valid values are shown in Table 5.

An example declaration of an image variable may look like this:

```
write rgba32f image2D img;
```

## 1.5 Constants

Sometimes, we might want to declare constant values in our shaders which lie outside the scope of any function or shader body. The syntax for this is very similar to the C standard:

```
const type name = type(expression);
```

Table 5: Image formats

| Syntax      | Functionality                                     |
|-------------|---|
| rgba32f     | 32-bit RGBA floating point texture                |
| rgba16f     | 16-bit RGBA floating point texture                |
| rg32f       | 32-bit RG floating point texture                  |
| rg16f       | 16-bit RGBA floating point texture                |
| r11g11b10f  | 11-bit R 11-bit G 10-bit B floating point texture |
| r32f        | 32-bit R floating point texture                   |
| r16f        | 16-bit R floating point texture                   |
| rgba16      | 16-bit RGBA untyped texture                       |
| rgb10a2     | 10-bit RGB 2-bit A untyped texture                |
| rg16        | 16-bit RG untyped texture                         |
| rg8         | 8-bit RG untyped texture                          |
| r16         | 16-bit R untyped texture                          |
| r8          | 8-bit R untyped texture                           |
| rgba16snorm | 16-bit RGBA untyped normalized texture            |
| rgba8snorm  | 8-bit RGBA untyped normalized texture             |
| rg16snorm   | 16-bit RG untyped normalized texture              |
| rg8snorm    | 8-bit RG untyped normalized texture               |
| r16snorm    | 16-bit R untyped normalized texture               |
| r8snorm     | 8-bit R untyped normalized texture                |
| rgba32i     | 32-bit RGBA integer texture                       |
| rgba16i     | 16-bit RGBA integer texture                       |
| rgba8i      | 8-bit RGBA integer texture                        |
| rg32i       | 32-bit RG integer texture                         |
| rg16i       | 16-bit RG integer texture                         |
| rg8i        | 8-bit RG integer texture                          |
| r32i        | 32-bit R integer texture                          |
| r16i        | 16-bit R integer texture                          |
| r8i         | 8-bit R integer texture                           |
| rgba32ui    | 32-bit RGBA unsigned integer texture              |
| rgba16ui    | 16-bit RGBA unsigned integer texture              |
| rgb10a2ui   | 10-bit RGB 2-bit a unsigned integer texture       |
| rgba8ui     | 8-bit RGBA unsigned integer texture               |
| rg32ui      | 32-bit RG unsigned integer texture                |
| rg16ui      | 16-bit RG unsigned integer texture                |
| rg8ui       | 8-bit RG unsigned integer texture                 |
| r32ui       | 32-bit R unsigned integer texture                 |
| r16ui       | 16-bit R unsigned integer texture                 |
| r8ui        | 8-bit R unsigned integer texture                  |



Constant values can be initiated simpler for single-member types, namely *float*, *int* and *bool* like so:

```
const type name = expression;
```

For array constants, use the following syntax:

```
const type name[size expression] = { type(expression1), type(expression2), type(expression3), ... };
```

This syntax demands that we initialize every element, that is the number of values in the initializer list be equal to *size*.

Or more simplified for the basic types *int*, *bool* and *float* etc:

```
const type name[size expression] = expression1, expression2, expression3, ...;
```

AnyFX will throw an error if the constant isn't completely initialized, if any value is of incorrect type. We can also chose to avoid the verbosity with constant arrays as demonstrated with variable arrays. If we omit the *size expression*, AnyFX will automatically evaluate the size of the array based on the amount of initializers.

## 1.6 Render states

A render state describes an object encapsulating a render state for the graphics card. As such, it's essential to define a render state if we want to perform features like alpha-blending, stencil-testing and such. There are two different definitions for render states, the first being:

```
state name [ Annotation ];
```

This creates a default render state. The default values are explained in Table 6.

The other syntax is:

```
state name [ Annotation ]  
{  
    flag = value;  
    ...  
    flag = value;  
};
```

A *state* object can have any number of flags, although it is highly recommended to assign a value to a flag only once. If a flag is set more than once, the previous value of the flag will be overwritten by the latter. The available flags and their values can be seen in Table 6.

The semantic describes two allowed syntaxes, either the one with double camel case, and the other with all lower case. You may notice that for some settings we have an array-like accessor, for example *BlendEnabled[i]*. Here, *i* stands for the render target index to which we apply the setting. If such a state is not defined explicitly, the default value is chosen for that render target. The different special types, such as *Comparison Function*, *Rasterizer Mode*, *Cull Mode*, *Stencil Op*, *Blend Mode* and *Blend Op* can take on special values. Values for *Comparison Function* can be found in Table 7. Values for *Rasterizer Mode* can be found in Table 8. Values for *Cull Mode* can be found in Table 9. Values for *Stencil Op* can be found in Table 10. Values for *Blend Mode* can be found in Table 11. Values for *Blend Op* can be found in Table 12.

Table 6: Render State Flags

| Flag                   | Default Value | Type                |
|------------------------|---------------|---------------------|
| DepthEnabled           | true          | bool                |
| DepthWrite             | true          | bool                |
| DepthClamp             | true          | bool                |
| SeparateBlend          | false         | bool                |
| StencilEnabled         | false         | bool                |
| StencilReadMask        | 0             | int                 |
| StencilWriteMask       | 0             | int                 |
| AlphaToCoverageEnabled | false         | bool                |
| MultisampleEnabled     | false         | bool                |
| DepthFunc              | Less          | Comparison Function |
| RasterizerMode         | Fill          | Rasterizer Mode     |
| CullMode               | Back          | Cull Mode           |
| StencilFrontFunc       | Always        | Comparison Function |
| StencilBackFunc        | Always        | Comparison Function |
| StencilFrontFailOp     | Keep          | Stencil Op          |
| StencilBackFailOp      | Keep          | Stencil Op          |
| StencilFrontPassOp     | Keep          | Stencil Op          |
| StencilBackPassOp      | Keep          | Stencil Op          |
| StencilFrontRef        | 0             | int                 |
| StencilBackRef         | 0             | int                 |
| BlendEnabled[i]        | false         | bool                |
| SrcBlend[i]            | One           | Blend Mode          |
| DstBlend[i]            | Zero          | Blend Mode          |
| BlendOp[i]             | Add           | Blend Op            |
| SrcBlendAlpha[i]       | One           | Blend Mode          |
| DstBlendAlpha[i]       | Zero          | Blend Mode          |
| BlendOpAlpha[i]        | Add           | Blend Op            |

**IMPORTANT:** The render state name *placeholder* is reserved for AnyFX internally, and is therefore not allowed to use.

## 1.7 Functions

Functions are treated specially. Although they follow the standard C syntax, they have some additions which allows for shader linkage and system variable access. As per the C standard, a function is defined as:

```
[attribute]
shader
type name(parameters)
{
    body
}
```

Here, *type* defines the return value of the function, *name* describes an identifier, and *parameters* defines a list of parameters. We will explain parameters later, right now we will focus on the *body* part of the function. The *qualifier* can be any value defined in Table 13, although some qualifiers are only applicable on function bodies which will be used as a shader stage, and only some types of shaders support some types of qualifiers.

Table 7: Comparison Function

| Semantic | Meaning                  |
|----------|--------------------------|
| Never    | Always fails             |
| Less     | Pass if less             |
| LEqual   | Pass if less or equal    |
| Greater  | Pass if greater          |
| GEqual   | Pass if greater or equal |
| Equal    | Pass if equal            |
| NEqual   | Pass if not equal        |
| Always   | Always pass              |

Table 8: Rasterizer Mode

| Semantic | Meaning                         |
|----------|---------------------------------|
| Fill     | Fills polygons                  |
| Line     | Draws polygons as line segments |
| Point    | Draws polygons as points        |

The *shader* qualifier tells AnyFX this function should be bound as a shader. This is required if this function is to be bound to a program object. This is required because in OpenGL, only the shader main functions have access to the built in GLSL variables.

The possible values for *Topology* can be found in Table 14, values for *Winding* in Table 15, values for *Partitioning* in Table 16, *InputPrimitive* in Table 17 and lastly *OutputPrimitive* in Table 18.

The body of the function is written in target-language specific code. This means that if we want a OpenGL compatible shader, we have to implement the function bodies in OpenGL. An example could be:

```
vec4
func(vec4 color)
{
    return color + vec4(1,1,1,1);
}
```

Now, if we were to compile with DirectX as our target, this would obviously fail.

System variables are accessed in different ways depending on the implementation target. We can always declare parameter attributes, although in OpenGL we have to directly implement them as such:

```
mat4 model;
mat4 view;
mat4 projection;

shader
void
vsMain(in vec4 position)
{
    gl_Position = projection * view * model * position;
}
```

In DirectX we must supply function attributes to tell AnyFX and DirectX what usage our parameter should have. Since DirectX has no global variables such as *gl\_Position*, we must declare how DirectX should bind this specific parameter during compilation. A parameter to a function follows this syntax:

..., [attribute] qualifier0 ... qualifierN type **name**, ...

Table 9: Cull Mode

| Semantic | Meaning                    |
|----------|----------------------------|
| Back     | Culls faces facing away    |
| Front    | Culls faces facing towards |
| None     | Disables culling           |

Table 10: Stencil Op

| Semantic     | Meaning                                    |
|--------------|--|
| Keep         | Keep current stencil value                 |
| Zero         | Write 0 to stencil buffer                  |
| Replace      | Replace stencil value with reference value |
| Increase     | Increase stencil value                     |
| IncreaseWrap | Increase stencil value and clamp           |
| Decrease     | Decrease stencil value                     |
| DecreaseWrap | Decrease stencil value and wrap            |
| Invert       | Invert stencil data                        |

The qualifiers which can be provided to a parameter can be either *const*, an input/output qualifier or *patch*. The input/output qualifier can be one of the following: *in*, *out*, *inout*. The input/output qualifier explains how the variable should be treated, where *in* means input, *out* means output, and *inout* means both input and output. Be careful with the use of *inout* since it won't be possible to apply on variables which requires different qualifiers. The *attribute* can be any of the values defined in Table 19. The *patch* qualifier is optional, and is only available in hull/control as outputs and in domain/evaluation shaders as inputs. The *patch* qualifier tells the GPU that the variable should be per-patch instead of per-vertex. The *const* qualifier is only available for helper functions, and tells ensured that the parameter is never changed within the context of the function.

Whenever parameters needs to be inputs in forms of arrays, which is required for example in a Hull/Control shader, Domain/Evaluation shader and/or Geometry shader, the name of the parameter should include '[]'. Example syntax:

```
//-----
/**
 * Simple hull shader with application defined tessellation levels
 */
[inputvertices] = 3
[outputvertices] = 3

shader
void
hsStatic(in vec3 inPosition[], in vec2 inUV[], out vec3 outPosition[], out vec2 outUV[])
{
    outPosition[gl_InvocationID] = inPosition[gl_InvocationID];
    outUV[gl_InvocationID] = inUV[gl_InvocationID];
    gl_TessLevelInner[0] = TessFactorInner;
    gl_TessLevelOuter[0] = TessFactorOuter;
    gl_TessLevelOuter[1] = TessFactorOuter;
    gl_TessLevelOuter[2] = TessFactorOuter;
}
```

The size of the input arrays is determined for Hull/Control and Domain/Evaluation shaders by the '[inputvertices]' attribute. The size of the output arrays is determined by the '[outputvertices]' attribute, however this attribute is only legit for the Hull/Control shader. If any linking is attempted between a Hull/Control shader and a Domain/Evaluation shader where the *outputvertices* from the Hull/Control shader is mismatched with the *inputvertices* value of the Domain/Evaluation shader, then linking will fail. AnyFX will automatically

Table 11: Blend Mode

| Semantic              | Meaning  |
|-----------------------|--|
| Zero                  | Multiplies blend value with 0                    |
| One                   | Multiplies blend value with 1                    |
| SrcColor              | Uses source color (shader result)                |
| OneMinusSrcColor      | Uses 1 - source color (shader result)            |
| DstColor              | Uses destination color (render target value)     |
| OneMinusDstColor      | Uses 1 - destination color (render target value) |
| SrcAlpha              | Uses source alpha                                |
| OneMinusSrcAlpha      | Uses 1 - source alpha                            |
| DstAlpha              | Uses destination alpha                           |
| OneMinusDstAlpha      | Uses 1 - destination alpha                       |
| SrcAlphaSaturate      | Uses saturated source alpha                      |
| ConstantColor         | Uses constant color                              |
| OneMinusConstantColor | Uses 1 - constant color                          |
| ConstantAlpha         | Uses constant alpha                              |
| OneMinusConstantAlpha | Uses 1 - constant alpha                          |

Table 12: Blend Op

| Semantic | Meaning                   |
|----------|---------------------------|
| Add      | Adds colors               |
| Sub      | Subtracts colors          |
| InvSub   | Subtracts by 1 - SrcColor |
| Min      | Selects minimum value     |
| Max      | Selects maximum value     |

resolve the size of the input variable arrays (if necessary), depending on either the *[outputvertices]* or *[inputvertices]* attribute or the geometry shader input primitive type.

A parameter to a function may have one or none of these qualifiers. An example of a function utilizing all of the above features may look like:

```
sampler2D DiffuseTexture;

[earlydepth]

shader
void
psMain(in vec2 uv, [color0] out vec4 Color)
{
    Color = texture(DiffuseTexture, uv);
}
```

This function can be bound as a pixel shader, which performs an early depth-test predication before executing the actual shader. It samples a texture from a texture resource and writes it to the 0'th render target.

If the target is OpenGL, then most of these parameter attributes can be omitted, since most system-variables can be accessed through the 'gl.<systemvar>' syntax. Since we AnyFX doesn't bother with code analysis, it is left to the programmer to handle system variables. However it is still allowed to assign a attribute to a variable, however the result will be that the attribute will be ignored unless it serves some functionality in AnyFX. Although, if a pixel/fragment shader has any parameter which doesn't supply the *[colorX]* attribute, an error will be thrown since binding color outputs to render targets must be done explicitly.

Table 13: Function Attributes

| Semantic                                   | Functionality                                     | Compatibility                                      |
|--|---|--|
| [topology] = <b>Topology</b>               | Defines tessellation input topology               | OpenGL and DirectX ( <b>ds</b> )                   |
| [winding] = <b>Winding</b>                 | Defines tessellation geometry winding order       | OpenGL and DirectX ( <b>ds</b> )                   |
| [partition] = <b>Partitioning</b>          | Defines tessellation partitioning method          | OpenGL and DirectX ( <b>ds</b> )                   |
| [maxvertexcount] = int                     | Defines geometry shader maximum vertex outputs    | OpenGL and DirectX ( <b>gs</b> )                   |
| [inputprimitive] = <b>InputPrimitive</b>   | Defines geometry shader input primitive           | OpenGL and DirectX ( <b>gs</b> )                   |
| [outputprimitive] = <b>OutputPrimitive</b> | Defines geometry shader output primitive          | OpenGL and DirectX ( <b>gs</b> )                   |
| [maxtess] = int/float                      | Defines maximum tessellation amount               | optional in DirectX ( <b>hs</b> )                  |
| [patchfunction] = string                   | Defines constant patch function                   | DirectX ( <b>hs</b> )                              |
| [inputvertices] = int                      | Defines tessellation input control points         | OpenGL and DirectX ( <b>hs</b> ) and ( <b>ds</b> ) |
| [outputvertices] = int                     | Defines tessellation output control points        | OpenGL and DirectX ( <b>hs</b> )                   |
| [instances] = int                          | Defines geometry shader instance invocations      | OpenGL and DirectX ( <b>gs</b> )                   |
| [earlydepth]                               | Enables pixel/fragment shader early depth testing | OpenGL and DirectX ( <b>ps</b> )                   |
| [localsize <sub>x</sub> ] = int            | Sets compute shader local dimension in x          | OpenGL and DirectX ( <b>cs</b> )                   |
| [localsize <sub>y</sub> ] = int            | Sets compute shader local dimension in y          | OpenGL and DirectX ( <b>cs</b> )                   |
| [localsize <sub>z</sub> ] = int            | Sets compute shader local dimension in z          | OpenGL and DirectX ( <b>cs</b> )                   |

Table 14: Topology

| Semantic | Functionality                |
|----------|------------------------------|
| triangle | Treats geometry as triangles |
| quad     | Treats geometry as quads     |
| line     | Treats geometry as lines     |
| point    | Treats geometry as points    |

## 1.8 Subroutines

Subroutines is a feature introduced in OpenGL 4.0 and DirectX 11. They work pretty much like function pointers in the sense that they allow you to exchange the function without having to switch shaders. This can allow you to avoid unnecessary shader switches and only perform an incremental update of the shader. Remember that subroutines only exchange functions, and can as such not change the input/outputs between shaders, meaning subroutine bindings takes place after linking is done. The syntax for subroutines is twofold, first a declaration and then a definition.

```
prototype type name ( arg0, ..., argN );
```

This declares a subroutine prototype which can be implemented using the following syntax:

```
subroutine ( prototype ) type name ( arg0, ..., argN )
{
...
}
```

This declares a possible implementation of a prototype. Here, the prototype argument must match a previously declared prototype. When this is done, we can simply declare a subroutine variable as such:

```
type name;
```

Which follows the standard variable declaration syntax, however **type** must be a previously defined prototype. In order to assign an implementation to a subroutine variable, we put it inside the parenthesis declared in

Table 15: Winding Order

| Semantic | Functionality  |
|----------|--|
| cw       | Treats geometry winding order to be in clockwise order         |
| ccw      | Treats geometry winding order to be in counter clockwise order |

Table 16: Partitioning Method

| Semantic | Functionality  |
|----------|--|
| integer  | Performs tessellation through integer division pattern             |
| even     | Performs tessellation through even floating point division pattern |
| odd      | Performs tessellation through odd floating point division pattern  |
| pow      | Performs tessellation through power function pattern               |

each program shader. It follows the following syntax:

```

prototype vec4 SomePrototype();
subroutine (SomePrototype) SomeSubroutine()
{
    return vec4(0);
}

SomePrototype someSubroutineVariable;
program SubroutineProgram
{
    ...
    VertexShader = vsStatic(someSubroutineVariable = SomeSubroutineImplementation);
    ...
}

```

## 1.9 Varblocks

Variable blocks lets us group variables into buffers, which allows for efficient updating of variables through chunks. The syntax for a varblock is:

```

shared buffers = X varblock name [ Annotation ]
{
    variable
    ...
    variable
};

```

In this definition, *variable* is a variable as described earlier in this chapter. *name* is the name of the varblock. In the varblock, we can define as many variables as we wish. This variable declaration works just like the ordinary one explained earlier in the chapter, meaning each variable can also be initialized with a default value. The number of variables defined in a varblock is arbitrary, although it's good practice to group variables in the order and frequency they need be updated. It is highly recommended to use varblocks for object-independent variable assignments such as camera view and projection matrices, focus depth, etc. For example, if we render a series of object using the same program, it's somewhat unnecessary to set the view and projection matrices per each render. Instead, it's much simpler to just set the view and projection matrices before the render, and then just update the model matrix and material variables. This could be

Table 17: Input Primitive

| Semantic           | Functionality   |
|--------------------|---|
| points             | Treats input geometry as a series of points             |
| lines              | Treats input geometry as a series of lines              |
| lines_adjacent     | Treats input geometry as a series of adjacent lines     |
| triangles          | Treats input geometry as a series of triangles          |
| triangles_adjacent | Treats input geometry as a series of adjacent triangles |

Table 18: Output Primitive

| Semantic       | Functionality                            |
|----------------|--|
| points         | Outputs geometry as a series of points   |
| line_strip     | Outputs geometry as a strip of lines     |
| triangle_strip | Outputs geometry as a strip of triangles |

done using two varblocks, a per-pass varblock and a per-model varblock.

Varblocks can be internally buffered by a defined amount of buffers. The more buffers in use, the less the application has to wait for each buffer to be successfully updated before continuing rendering, so more buffers results in a slower per-object latency. This amount is denoted by the *buffers = X* qualifier, in which *X* represents the amount of backing buffers. Note that when the amount of buffers increase, so does also the memory it occupies. Varblocks also need to declare the qualifier *shared* if the user wants AnyFX to only allocate one actual varblock which can be shared by all others in the application. This allows for updating a single varblock instead of many, and can result in simplicity of programming since one doesn't have to redundantly set variables for each shader, but can rather apply them once and have them retain over the course of a pass.

## 1.10 Varbuffers

Varbuffers work very similar to varblocks, however they provide an interface which lets us leave the buffer size undefined in the shader. Instead, we can tell a varbuffer to allocate the internal buffer with a certain size. This allows us to modify the size in the application. We can also retrieve the buffer handle so we may use it outside of AnyFX, for example as an input to some other shader. Varbuffers are equal to shader storage blocks in OpenGL, or RWBuffers in DirectX, and allows the shader to read and **write** to the buffer. The syntax is also similar to a varblock, except variables may be of an unsized array type. It follows the following syntax:

```
varbuffer name [ Annotation ]
{
    variable
    ...
    variable
};
```

We can as aforementioned also do something like this:



Table 19: Variable Attributes

| Semantic          | Functionality  |
|-------------------|--|
| drawinstanceID    | CPU invoked instance ID for instanced rendering          |
| vertexID          | Vertex ID of current vertex                              |
| primitiveID       | Primitive ID of current geometry shaded primitive        |
| invocationID      | Invocation ID of current geometry shader invocation      |
| viewportID        | Viewport ID of current geometry shader invocation        |
| rendertargetID    | Render target ID of current geometry shader invocation   |
| innertessellation | Inner tessellation factor in hull shader                 |
| outertessellation | Outer tessellation factor in hull shader                 |
| position          | Vertex position from vertex shader                       |
| pointsize         | CPU assigned point size                                  |
| clipdistance      | Rasterizer clip distance                                 |
| frontface         | Holds if the current pixel is facing front               |
| coordinate        | Screen space pixel coordinate                            |
| depth             | Depth buffer write output                                |
| color0            | Render target 0 write output                             |
| color1            | Render target 1 write output                             |
| color2            | Render target 2 write output                             |
| color3            | Render target 3 write output                             |
| color4            | Render target 4 write output                             |
| color5            | Render target 5 write output                             |
| color6            | Render target 6 write output                             |
| color7            | Render target 7 write output                             |
| workgroupID       | Workgroup execution ID for compute shaders               |
| numgroups         | Number of groups in current invocation of compute shader |
| localID           | Local ID of invocation in compute shader                 |
| localIndex        | Local index of invocation in compute shader              |
| globalID          | Global ID of invocation in compute shader                |

```
varbuffer name [ Annotation ]
{
    variable
    ...
    variable[]
};
```

In which case we can allocate dynamic amount of size by calling the varbuffer in the API and have it allocate a set amount of space. Note that this amount is not in bytes, but in number of elements, so if we for example have an array of a struct, we will allocate said amount of structs in the varbuffer.

## 1.11 Programs

The last structure you can define in an AnyFX file is a program. Programs follow the following syntax:

Table 20: Shader Dependencies

| Shader                                 | Dependency(ies)  |
|--|--|
| Vertex shader ( <b>vs</b> )            | None   |
| Pixel/Fragment shader ( <b>ps</b> )    | Vertex shader ( <b>vs</b> ) and optionally Geometry shader ( <b>gs</b> ) and/or Hull-Domain shaders ( <b>hs - ds</b> ) |
| Hull/Control shader ( <b>hs</b> )      | Vertex shader ( <b>vs</b> ) and Domain/Evaluation shader ( <b>ds</b> )   |
| Domain/Evaluation shader ( <b>ds</b> ) | Vertex shader ( <b>vs</b> ) and Hull/Control shader ( <b>hs</b> )  |
| Geometry shader ( <b>gs</b> )          | Vertex shader ( <b>vs</b> ) and optionally Hull-Domain shaders ( <b>hs - ds</b> )                                      |
| Compute shader ( <b>cs</b> )           | None   |

```

program name [ Annotation ]
{
    VertexShader = func();
    PixelShader = func();
    HullShader = func();
    DomainShader = func();
    GeometryShader = func();
    ComputeShader = func();
    RenderState = state;
}

```

The program object has 7 'slots'. Each slot except for *RenderState* are used for shader function assignments. Here, the symbolic name *func()* is the name of the function we want to attach to the given shader slot. This converts the function to a shader entry point, which also removes it from the list of helper functions, making it inaccessible to call from another shader program. The *state* flag is the identifying name of a render state object as described earlier. Although all shader slots are optional, some of them are dependent on others. The dependencies shown in Table 20 declare what prerequisites each shader has. The shaders are only pair-wise dependent and as such are oblivious to whatever shader is attached further down the pipeline. If the *RenderState* field is not declared explicitly, a default render state will be used for the program.

Subroutines are bound to a certain shader stage by putting the assignment in the argument list in *func()*. See section 1.8 for a complete syntax explanation.

An example program, with a render state and a pixel shader and vertex shader could look something like this:

```
state AlphaState
{
    DepthEnabled = false;
    BlendEnabled[0] = true;
    SrcBlend[0] = One;
    DstBlend[0] = OneMinusSrcAlpha;
    CullMode = None;
};

//-----
/**
 */
shader
void
vsStatic(in vec3 position, in vec2 uv, out vec2 UV)
{
    gl_Position = Projection * View * Model * vec4(position, 1.0f);
    UV = uv;
}

//-----
/**
 */
[earlydepth]

shader
void
psStatic(in vec2 uv, [color0] out vec4 Color)
{
    vec4 color = texture(DiffuseTexture, uv) * vec4(2.0f, 1, 1, 1);
    Color = color;
}

//-----
/**
 */
program Alpha
{
    vs = vsStatic();
    ps = psStatic();
    state = AlphaState;
};
```

## 1.12 Structures

Structures can be defined as described with the C-standard. It follows the following syntax.

```
struct name
{
    type member name;
    ...
    type member name;
};
```

Structures can be used as inputs, outputs and function local objects. The GLSL shader storage buffer functionality is currently unsupported. Example of a structure can look as such:

```

struct Data
{
    float multiplier;
    int divider;
};

```

## 1.13 Samplers

Samplers serves as objects which controls sampling from textures. The point of a sampler is to enable sampling a single texture using several sampling methods. Since samplers has multiple forms of definition and implementation, some functionality is not necessary for some languages. In OpenGL, samplers are purely CPU-based, meaning a C object is responsible for modifying the sampler state. In HLSL version 4 and up, a sampler state object is used together with a texture object in the HLSL code, meaning the sampler has to be declared and used in the shader code as an individual object. AnyFX supports letting a single sampler handling several textures, this is done by assigning more than one texture to a sampler state. An example could be:

```

sampler2D DiffuseMap;
sampler2D SpecularMap;
sampler2D EmissiveMap;

samplerstate DefaultSampler
{
    Samplers = { DiffuseMap, SpecularMap, EmissiveMap };
    BorderColor = { 1.0f, 0.0f, 0.0f, 1.0f };
    AddressU = Border;
    AddressV = Border;
};

```

This feature is of course only available and useful on platforms where samplers are not separate shader objects and has to be assigned. If a sampler is assigned to a texture which was already assigned by another sampler, the last defined sampler will take precedence.

A sampler can use the fields declared in Table 21. The *Texture* field is only necessary for implementations where the sampler is purely CPU-located. In HLSL, this field is still valid, although it serves no purpose since the texture-sampler coupling is done when sampling the texture rather than before rendering occurs. However, for GLSL and other languages in which a sampler needs a texture to have any use at all, the *Texture* field has to be defined. The *AddressU/V/W* field can be provided with any value in the Table 22. The *Filter* field can be provided with any value in Table 23. All filter modes which begins with *Comparison* turns the sampler into a comparison sampler, meaning it will produce no result unless it is used as a comparison sampler in the shader code. This can be used for hardware-accelerated shadow mapping, for example. The values in Table 21 in bold font represent enumeration values defined in the tables listed above.

The filter mode *Anisotropic* is equal to *MinMagMipLinear*

The syntax for the string list and 4-component float list looks as follows:

```
{ expression0, ..., expressionN } or { identifier0, ..., identifierN }
```

An example of how this looks can be seen in the above code example. In the case for **BorderColor**, *element* are expressions. In the case with **Samplers**, *element* are sampler identifiers.

Table 21: Sampler settings

| Argument   | Functionality                                     | Data type                  | Default value              |
|------------|---|----------------------------|----------------------------|
| Samplers   | Assigns sampler to sampler or list of samplers    | string list                | {}                         |
| Filter     | Determines filtering method                       | <b>Filter Mode</b>         | MinMagMipLinear            |
| AddressU   | Address mode in U-dimension                       | <b>Address Mode</b>        | Wrap                       |
| AddressV   | Address mode in V-dimension                       | <b>Address Mode</b>        | Wrap                       |
| AddressW   | Address mode in W-dimension                       | <b>Address Mode</b>        | Wrap                       |
| Comparison | Sets if the sampler should be a comparison sample | bool                       | false                      |
| CompFunc   | Comparison function                               | <b>Comparison Function</b> | Less                       |
| LodBias    | LOD bias for mip-maps                             | float                      | 0.0f                       |
| MinLod     | Minimum LOD level for mip-maps                    | float                      | -FLT_MAX                   |
| MaxLod     | Maximum LOD level for mip-maps                    | float                      | FLT_MAX                    |
| Border     | Border color                                      | 4-component float list     | { 0.0f, 0.0f, 0.0f, 0.0f } |

Table 22: Address mode

| Value  | Functionality            |
|--------|--------------------------|
| Wrap   | Repeats texture          |
| Mirror | Repeats texture mirrored |
| Clamp  | Clamps border values     |
| Border | Applies border color     |

## 1.14 Annotations

Variables, Varblocks, Programs and RenderStates may have an alternative annotation field. Note that these are optional, and therefore doesn't have to be declared. They also have no impact on target code, but exists solemnly to attach arbitrary data to identifiers so that they can be read in the API. Annotations works by assigning arbitrary data with a name using the following syntax:

```
[..., type name = expression; ...]
```

The type field can be *string*, *int*, *bool*, *float*, *double*. The name can be any name not already defined in this annotation field. A real application of an annotation can look like this:

```
[ string Mask = "Opaque"; ]
```

Annotations must be declared directly after the object name as explained in the subsections for each AnyFX type. An example of a program with an annotation may look like the following:

```
program Solid [ string Mask = "Opaque"; ]
{
    vs = vsStatic();
    ps = psStatic();
    state = OpaqueState;
};
```

## 1.15 Usage

The compiler can be invoked with the command line arguments found in Table 24.

Note that the absence of any of the required argument will cause a compiler error, and will abort before type checking, code generation and compilation takes place. The -D arguments can be more than one, providing

Table 23: Filter mode

| Value                | Functionality   |
|----------------------|---|
| MinMagMipPoint       | Performs full point sampling (no interpolation)                 |
| MinMagMipLinear      | Performs full linear sampling                                   |
| MinMagPointMipLinear | Point sampling on minification and mag, linear on mip           |
| MinMipPointMagLinear | Point sampling on minification and mip, linear on magnification |
| MinPointMipMagLinear | Point sampling on minification, linear on mip and magnification |
| MinLinearMipMagPoint | Linear on minification, point on mip and magnification          |
| MinMipLinearMagPoint | Linear on minification and mip, point on magnification          |
| MinMagLinearMipPoint | Linear on minification and magnification, point on mip          |
| Anisotropic          | Performs anisotropic filtering                                  |
| Linear               | Performs linear filtering without mipmapping                    |
| Point                | Performs point filtering without mipmapping                     |

Table 24: Shader Dependencies

| Argument                            | Functionality                      | Required |
|-------------------------------------|------------------------------------|----------|
| -f <file name>                      | Input raw AnyFX-formatted fx-file  | Yes      |
| -o <output path>                    | Outputs compiled AnyFX binary blob | Yes      |
| -target <supported target language> | AnyFX target language              | Yes      |
| -D <name>                           | mcpp preprocessor definition       | No       |

each as a definition for preprocessing.

As of version 1.0, the compiler may generate the same error multiple times. This is because beneath the hood, AnyFX simply copies all variables and helper functions into every shader it generates. As such, if there is an error outside a shader function, this error will appear for each shader, causing a plethora of errors. Unfortunately, we don't perform any type of code analysis, so this problem will remain as long as AnyFX doesn't support it.

## 2 API

### 2.1 Initialization

To initialize AnyFX, you need to create an `EffectFactory`. To get access to it, simply include the `anyfxapi.h` provided in the AnyFX source. Then, it's a simple matter of loading an effect, either from file or from a memory blob. It can look something like this:

```
AnyFX::EffectFactory* factory = new AnyFX::EffectFactory;
AnyFX::Effect* effect = factory->CreateEffectFromFile("C:/demo/effect.afx");
```

Done!

### 2.2 Effect

The effect object contains all the information declared in an effect. The interfaces one can access from the API are *EffectProgram*, *EffectShader*, *EffectRenderState*, *EffectVariable* and *EffectVarblock*. Each and every of these interfaces can be fetched using their identifying name in the AnyFX code, or through indexing.

#### 2.2.1 EffectProgram

The *EffectProgram* interface holds a program. The main functions for this are **Apply()** and **Commit()**. The **Apply()** function applies the *EffectProgram*, meaning it sets the GPU render state and shaders in order to prepare it for rendering. The **Commit()** function flushes all data currently accumulated in the varblocks parented by the *EffectProgram*, as such, it updates the variables in a chunk like manner. Note that **Commit()** must be called before rendering if one wishes to apply the currently assigned shader variables.

#### 2.2.2 EffectShader

The *EffectShader* interface supplies basic access to shader name, native shader code and, if any, the shader compilation error. If for some reason there is a compilation error, then this is a bug in the compiler, since the compiler shouldn't let any compilation errors pass through.

#### 2.2.3 EffectRenderState

If one wants to change the behaviour of a render state during runtime, the *EffectRenderState* supplies this functionality. Also note that the *EffectRenderState* can be applied without the need to apply an *EffectProgram*, meaning we can force apply a render state anytime we want. We can also modify the render state itself by setting any of the render state flags, stencil reference values etc.

#### 2.2.4 EffectVariable

The *EffectVariable* supplies an interface to a variable. Note that setting a variable has no direct effect in the back-end, and will thus not be set in the shader until **Commit()** gets called on the variable, or the program. Calling **Commit()** on the program causes all variables currently set to be 'flushed' to GPU memory, meaning that for each variable with a handle in the currently applied program, the variable will be properly set.

### 2.2.5 EffectVarblock

Varblocks using modern graphics APIs implements a method for mapping a buffer to CPU memory, and have it be synced only in the very instant it is required to render. Therefore, some buffers may not be updated until some other API call takes place, which makes the updating of the buffer hard to determine. In order to disable this functionality, one must explicitly get the varblock in question, and call **SetFlushManually(true)**. Then, whenever the buffer is supposed to be flushed, one has to call **FlushBuffer()**.

*EffectVariable* instances which are written in the shader code as a member of a *EffectVarblock* are automatically assigned to its block. As such, setting a variable inside an *EffectVarblock* doesn't require any manipulation of said object.

### 2.2.6 EffectVarbuffer

The *EffectVarbuffer* represents an AnyFX varbuffer shader object. The *EffectVarbuffer* class can allocate, reallocate and deallocate the back-end storage buffer used in the shader. It sports three functions, **Allocate()** which takes a set amount of elements to allocate, **Reallocate()** which takes the same arguments as **Allocate()**, except it allocates a new buffer and copies the new data over. The **Deallocate()** function is used to completely nullify the buffer, meaning it cannot be used in any useful fashion, so it's recommended this is executed whenever the varbuffer variable is no longer needed.

### 2.2.7 EffectSubroutine

This class only contains information about a subroutine, but doesn't provide anything useful or assignable. Subroutines are set from inside the *EffectProgram* object. This class should mainly be used for shader debugging.

### 2.2.8 Annotations

*EffectProgram*, *EffectRenderState* and *EffectVariable* has annotations, which can be get through *GetAnnotationX* where X is the type of annotation data requested. The type of data here can be *Int*, *Bool*, *Float*, *Double*, *String*. All annotation getter methods must be provided with a string which should be the name of the string declared in the original source file. If a type is mismatched with the function, the behavior is undefined so be cautious with this.