The Shadow Framework Pipeline Programming Language

1) **General Language Informations**
2) **Structure and Grids**
3) **Shader Components**
4) **Writing a Shader Component**

The Shadow Framework Pipeline Programming Language

The SF Pipeline Programming Language is used for write code for Shadow Framework Shading Components.

1) **General Language Informations**

Basic Information

- General Format
  - To do anything in this language you use this simple line format:
    - **@command_name data**
- Inclusion
  - You can include any other file commands with the include command
    - **@include filename**
- Comments:
  - You can add comments by starting your lines with //
- **Register**:
  - Registers are a really important element of this language. A Register is a predefined language variable whose value can be read or written according to the pipeline stage they are used. Refer to each Stage to know what registers can be read and what can be written

Variables Types

All Variables and Registers has a size which is defined by its type:

- <>: Generic, which stays for a not-assigned type
- **float**: scalar value (size is 1)
- **float2**: a 2 components vector
- **float3**: a 3 components vector
- **float4**: a 4 components vector

Writing Expressions

Expression can be written by combining variables and register name with expressions operation. Each expression support all variable dimesion; whenever an operator is called on variables with differen size, there are rules to determine the result. Usually, the rules is that all operands are aligned to the higher size and the result will have that size.

- **Sum**: expression_1 + expression_2 [+ expression_i]
  - evaluates the sum of more expression. The result is a vector of the same size of the operands. If the expressions haven't the same size for all operands, the higher size is used for the result. Undefined components for vectors having shorter size are assigned the value 0. Can be combined

with Subtraction.

- **Multiplication**: expression_1 * expression_2 [*expression_i]
  - evaluates the sum of more expression. The result is a vector of the same size of the operands, where multiplication is performed for each component. If the expressions haven't the same size for all operands, the higher size is used for the result. Undefined components for vectors having shorter size are assigned the value 1.
- **Divide**: expression_1/ expression_2
  - evaluates the division between expression_1 and expression_2. Expression_1 can be of any size, expression_2 must be scalar. The resultant expression will have expression_1 size
- **Subtraction**: expression_1 - expression_2 [- expression_i]
  - evaluates the subtraction of more expression. The result is a vector of the same size of the operands. If the expressions haven't the same size for all operands, the higher size is used for the result. Undefined components for vectors having shorter size are assigned the value 0. Can be combined with Sum.
- **Composition**: value_1,value_2(,value_3(,value_4))
  - construct a value of type float2, flaot3 or float 4 according to the number of values combined.
- **Texture Evaluation** : `texture%texCoord0`
  - Evaluates a texture, texture must be a valid **texture Register**, texCoord0 must be a **float2** value.
- **Dot Product**: expression_1 ° expression_2
  - Evalutaes the dot Product of two expression, which must be of the same size. The result is always a scalar value.

Operations Order:

- Composition
- TextureEvaluation, DotProduct
- Divide
- Multiplication
- Sum, Subtraction

2) Structure and Grids

//TODO (see data/pipeline/structure for some example)

3) Shader Component

A **Shader Component** is a program which can be assigned to one of the ShadowFramework Pipeline Stages. There are 4 different stages with 4 different Shader Components Types:

1. Tesselator
2. Primitive
3. Material
4. Light Step

> **Note**: during **SF2.0** development there was also a transform stage, which actually has been changed to fixed functionality.

A Complete Shadow Program is required to make the Shadow Framework Pipeline work. A Complete Program is made up of:

- 1 Tesselletor
- More Primitives.
- More Materials
- 1 Light Step.

While lights steps depends upon the **Rendering Algorithm**, Tessellator, Primitives and Material are part of a scene description.

Tessellation Step

TODO

Primitive Step

TODO

Material Step

TODO

Light Step

Available input Registers.

- **texture0**: texture, usually used more by materials
- **texture1**: texture, usually used more by materials
- **texture2**: texture, usually used more by materials
- **texture3**: texture, usually used more by materials
- **normal**: normal value
- **position**: the world position of the point getting enlightened
- **duVector**: first component of the tangent plane
- **dvVector**: second component of the tangent plane
- **color**: a color value assigned from material
- **modelview**: the modelview matrix used for transforms
- **projection**: the projection matrix used for transforms

Available output Registers.

- **fcolor** : final fragment color.
- **fcolor0**: when using RenderedTexture, the final fragment color to be assigned to the first texture
- **fcolor1**: when using RenderedTexture, the final fragment color to be assigned to the second texture
- **fcolor2**: when using RenderedTexture, the final fragment color to be assigned to the third texture
- **fcolor3**: when using RenderedTexture, the final fragment color to be assigned to the fourth texture

4) Writing a Shader Component

Shader Component can be loaded from text files. Here some simple syntax rules to write them.

Each program will be described in this way:

```
@begin ShaderType ShaderName
[inputs]
[commands]
@end
```

where `@begin` and `@end` are keywords, and `ShaderType` is one of **Tesselator/Primitive/Material/LightStep**. ShaderName will be the name used to identify the program in the pipeline once loaded, and must be unique. `[inputs]` and `[commands]` are a sequence of instructions which describe what the component does.

Available commands for ***input*** are :

- **use**: tells that this component will use one of the **input Register.**
  - Syntax : @use registerName
  - Example: `@use color` is used in the LightStep to tell that the light step will use the Register color, whose value is usally generated by Material steps.
- **param**: define a set of parameters which is not part of the Default pipeline Registers, but which is ad hoc for the Shader Component declaring them. This parameters can be assigned before each rendering process, but will work like a constant within the same rendering process (and like GLSL uniforms). Each parameter must be instance of some structure.
  - Syntax: `@param name1[,namei] as StructureName`
  - Example: suppose that a structure as been defined like this:

    ```
    @begin Structure PLight01
          @float3 intensity
          @float3 position
    @end Structure
    ```

    then the following instruction

    ```
    @param intensity,lPosition as PLight01
    ```

    defines two parameters intensity and lPosition with the same structure as Plight01, so intensity will be **float3** and lPosition will be a **float3**.
- **grid**: define a set of parameters which is not part of the Default pipeline Registers, but which is ad hoc for the Shader Component declaring them, and which is structured according to a grid. All the parameters in the same grid has the same type, but that type is not defined in the component and can be assigned at **Program-Build Time**.
  - Syntax: `@grid name1[,namei] as GridName:<>`
  - Example: suppose that the grid.

    ```
    @begin Grid Bezier2Grid
    @vertex A
    @vertex B
    @vertex C
    @edge A AB B
    @edge B BC C
    @edge C CA A
    @path AB BC CA AB
    ```

```
@end
```
has been defined. The following command
```
@grid A,B,C,D,E,F as Bezier2Grid:<>
```
declare 6 parameters organized in a grid structure like Bezier2Grid. Type for such parameters is undefined and need to be assigned at **Program-Build Time**.

Available commands for ***commands*** are :

- **define**: define a temporary variable to be used within the ShaderComponent.
  - Syntax: `@define variableName:size expression`
    - **variableName**: the Name of the Variable to be declared
    - **size**: the Size of the Variable, should be in the range [1,4]. 1: scalar value, and 2-3-4 stays for a Vertex/Colour with 2-3-4 coordinates.
    - **expression**: a valid expression
  - Examples:
    - `@define bright:3 0.4,0.4,0.4` defines a temporary variable called bright has a vector with 3 components and assigns this component with values (0.4,0.4,0.4)
    - `@define tmp1:1 (position.z+1)*0.5` defines a temporary scalar variable called tmp1 and assigns to it 0.5*(position.z+1) where position.z is the z-component of SFPipeline Register called position.
- **write**: write one of the output Register, assigning it a value. You do not need to assign all output Register.
  - Syntax: `@write registerName expression`
    - registerName: the register Name
    - expression: a valid expression, its value is assigned to the Register
  - Examples
    - `@write fColor color` assigns the value of 'color' to the Register fColor.