University of Toronto:
CSC467F Compilers and Interpreters, Fall 2009

ARB Fragment Program

ARB fragment program is an openGL extension that allows the programmer to program the fragment processor/shader. It exposes a set of GPU instruction that allow the user to manipulate fragments. The programmer writes a program in ARB fragment program assembly like language, and loads it onto the GPU using the openGL extension. While the instruction set is assembly like, it is important to keep in mind that ARB fragment program is not a typical assembly language. It does not guarantee a 1 to 1 mapping of instructions to actual GPU instructions. Furthermore, it allows some syntax that is not common is other assembly languages. This document will describe most of the things that you will need to know in order successfully complete the lab. A full (and rather unreadable) documentation of ARB fragment program can be found at http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt

General Syntax

The rest of the document will describe the syntax in more detail. Generally speaking, ARB fragment program consists of a list of instructions and variable declaration. Each instruction and declaration must be followed with a semicolon (';'). Comments start with the '#' character, and end at the end of the line.

Register Set

ARB fragment program operates on 4 different types of register. All registers are 4 component floating point registers. As all registers are of a floating point type, integer and boolean types are going to have to be implemented using the floating point registers. It is up to you to figure out how to implement that.
Color is by default represented in an RGBA format(reb,green,blue,alpha).

It is possible to access the individual components of a register. The 4 components of each register can be indexed using the letters xyzw. For example
regname.x *# access the first component of the vector.*
regname.y # access the second component of the vector.
regname.w *# access the last(4th) component of the register.*
Regname.xyz *# access the first 3 components of the vector.*

The register types are:

Constant Registers:
These are read only registers that are used to pass data from openGL to the fragment shader. There are 3 types of registers.

- Program Environment Parameters: Read only registers, can be accessed as program.env[index]. Used to pass environment data from openGL to the fragment shader.
- Program Local Parameters: Read only registers, can be accessed as program.local[index]. Are used to pass local data from openGL to the fragment data.
- Custom registers: These are registers that are used to store user defined constants. When an ARB fragment program gets executed on the GPU all constants must be stored in registers (the ARB fragment program instruction set does not support immediate operands). Constants may be bound to registers in 2 ways:
  - Explicitly: A constant variables can be declared using the keyword PARAM, for example:

    PARAM var = {1.0,2.0,3.0,4.0}; *#{1.0,2.0,3.0,4.0} is saved in some register. var now points to the regsiter*

  - Implicitly: Any use of a constant will essentially result in the constant being assigned to a register, for example:

    ADD a,b, {1.0,2.0,3.0,4.0}; *# {1.0,2.0,3.0,4.0} is saved in some register.*

  Constants with less than 4 components can be used. For example the following statements are legal:

  PARAM a = {1.0,2.0,3.0};
  PARAM a = 1.0;

  Furthermore, constants can be bound to openGL state constants. For example:

  PARAM ambient = state.material.ambient.;

Also, a reference to the program parameter register can be created using the PARAM key work. For example:
PARAM var = program.local[8]; *#var points to the register program.local[8]*
PARAM var1 = program.env[1];

NOTE: This section can be a little bit overwhelming. Most of it is given as background knowledge . For the purpose of this course, all you have to know is that you can create constants using the PARAM key word, and that you can use constants explicitly in your code.

Input Registers:

- Fragment Attributes: Read only registers, can be accessed as fragment.*regname*. Are used to pass data from the previous stage in the GPU pipe line to the fragment shader.

Output Registers:

- Results : Write only registers, can be accessed as result.*regname*. Are used to pass data from the fragment shader to the next stage in the pipe line. The result.color register must be written for a pixel to be displayed on the

screen. If the result.color register is not written, the pixel will essentially be transparent.

Temporary Registers:

- Temporary: Read/Write registers, can be declared by the user. Are used to store temporary results.
  Temporary variables can be declared using the TEMP key word. For example:
  TEMP tmp, tempVar; *# declares 2 temp variables. Each variable points to a uniqe temporary register*

An ARB fragment program reads data from the input and constant registers, performs some math operations, and writes the results to the output result registers.

For further details on the purpose of each register please refer to the shader programming presentation.

For the purpose of this course, we are going to use only a limited number of constant, input and output registers. The registers are mapped to the MiniGLSL custom variables as follows:

```
gl_FragColor -> result.color
gl_FragDepth -> result.depth
gl_FragCoord -> fragment.position
gl_TexCoord -> fragment.texcoord
gl_Color -> fragment.color
gl_Secondary -> fragment.color.secondary
gl_FogFragCoord -> fragment.fogcoord
gl_Light_Half -> state.light[0].half
gl_Light_Ambient -> state.lightmodel.ambient
gl_Material_Shininess -> state.material.shininess
env1 -> program.env[1]
env2 -> program.env[2]
env3 -> program.env[3]
```

The exact number of registers of each type is hardware depended. While even low ends GPUs have enough register to support most shader programs, you should be careful when you are creating temporary register. Temporary registers have to be created to store intermediate results of instructions. For example,
a = 7 + b + a, will requires you to store a + b in some temporary register. If you create a new register for every intermediate result, you will run out of registers pretty fast.

Identifier names

Registers are identified using identifiers. Identifiers follow the following rules

- An identifier can consist of any sequence of one or more
  - letters (A to Z, a to z),
  - digits (“0” to ”9”)
  - underscores (“_”)
  - dollar signs “$”
- First character may not be a digit
- Case sensitive
- Legal: A, b, _ab, $_ab, a$b, $_
- Not Legal:9A, ADDRESS, TEMP (other reserved words)

Ideally, each MiniGLSL variable should be mapped to a unique ARB fragment program assembly identifier. However, to simplify the implementation, you are allowed to use the variable name from MiniGLSL as an identifier in your assembly code. For example:
Float myVar1; , will be declared in the ARB fragment assembly as:
TEMP myVar1;

Any temporary registers that you create in order to store intermediate values must be name tempVar1, tempVar2, etc.. This will improve the readability of your assembly code, and it will ensure that your intermediate temporary registers don’t have the same identifier as some MiniGLSL variable (tempVar* will not be used as a variable name for any of the test benchmarks).-

Instruction set

ARB fragment program supports the following instruction set:

Instruction	Inputs	Output	Description
ABS	v	v	absolute value
ADD	v,v	v	add
CMP	v,v,v	v	compare
COS	s	ssss	cosine with reduction to [-PI,PI]
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DPH	v,v	ssss	homogeneous dot product
DST	v,v	v	distance vector
EX2	s	ssss	exponential base 2
FLR	v	v	floor
FRC	v	v	fraction
KIL	v	v	kill fragment
LG2	s	ssss	logarithm base 2
LIT	v	v	compute light coefficients
LRP	v,v,v	v	linear interpolation
MAD	v,v,v	v	multiply and add
MAX	v,v	v	maximum
MIN	v,v	v	minimum
MOV	v	v	move

MUL	v,v	v	multiply
POW	s,s	ssss	exponentiate
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
SCS	s	ss--	sine/cosine without reduction
SGE	v,v	v	set on greater than or equal
SIN	s	ssss	sine with reduction to [-PI,PI]
SLT	v,v	v	set on less than
SUB	v,v	v	subtract
SWZ	v	v	extended swizzle
TEX	v,u,t	v	texture sample
TXB	v,u,t	v	texture sample with bias
TXP	v,u,t	v	texture sample with projection
XPD	v,v	v	cross product

Where "v" indicates a floating-point vector input or output, "s" indicates a floating-point scalar input, "ssss" indicates a scalar output replicated across a 4-component result vector, "ss--" indicates two scalar outputs in the first two components, "u" indicates a texture image unit identifier, and "t" indicates a texture target.

Not all of the instructions are needed to successfully complete the lab. Use only the instructions that are required to implement the functionality of MiniGLSL.

Implementing MiniGLSL functions:

MiniGLSL defines 3 functions. These functions can essentially be implemented using ARB fragment program instructions. The functions map to instructions as follows.
dp3 -> dp3
rsq -> rsq
lit -> lit

Branch instructions:

As you might have noticed, the instruction set does not have any branch or jump instructions. Therefore, if else statements are going to have to be implemented using the CMP instruction. It is up to you to figure out how to do that. It is also up to you to figure out how to properly implement the boolean data type.