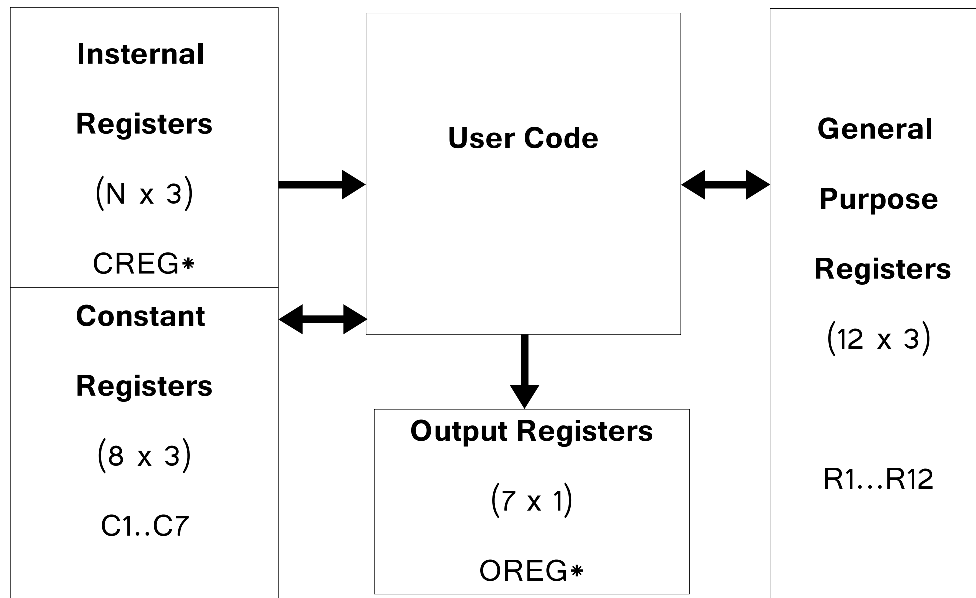
THEIA Simple Shader Simulation Tutorial

Thank you very much for your interest in the project!

In this tutorial, you will set up THEIA to simulate a simple 3D scene consisting of 2 texturized triangles. You will write a series of simple pixel shaders to get familiar with the shanding system.

First a little bit of background

THEIA's pixels shaders are written using the THEIA programming language. This is an assembly programming language consisting of a series of instructions that operate on a series of registers. There is no stack.
The following diagram illustrates THEIA's programming model.



Registers: THEAI has 4 types of registers.

- **Internal Registers**:
  Accessibility: Read-Only.
  Scope: The values of these registers can change from one pixel iteration to the next.
  Purpose: They are special registers used by the internal GPU routines. Sometimes you may want to read from these registers. Examples: `CREG_CAMERA_POSITION`, `CREG_RESOLUTION`, `CREG_PIXEL_2D_POSITION`, etc.

- **General purpose Registers**:
  Accessibility: Read-Write.

Scope: The values of these registers can change from one pixel iteration to the next. This means that even though you can write any value you want, you are **not** guaranteed that this value will be persistent from one pixel iteration to the next.
Purpose: You can basically store whatever you want in these registers, but again, the values are only kept for the duration of your shader.

- **Constant Registers:**
  Accessibility: Read-Write.
  Scope: The values of these registers are kept for the entire duration of the GPU execution. This means that if you store a value, this value is kept until you replace it with a different value or reset the GPU.
  Purpose: Although you can use these registers as you like, the idea is to store constants during a special execution stage, so that you can reuse these values for each pixel iteration.

- **Output Registers:**
  Accessibility: Write-Only.
  Scope: The values of these registers can change from one pixel iteration to the next.
  Purpose: These are special registers that indicate an output from a particular stage of THEIA's execution. Examples: `OREG_PIXEL_COLOR`, `OREG_TEX_COORD1`, `OREG_TEXWEIGHT1`, etc.

Instructions:

THEIA has arithmetic, logic and flow control instructions. There are two flavors of instructions:

Type 1: **OPERATION** DESTINATION SOURCE_REGISTER1 SOURCE_REGISTER2

Type 2: **OPERATION** DESTINATION IMMEDIATE_VALUE

There is a full list of the instructions in the Documentation section. But for now, let's just see some simple shaders to get a taste of how this works.

Step 1 – Download and the Simulation environment

1.1 Download the 'theia_gpu_latest.tar.gz' file from Download section of the project web page.
1.2 Extract the tarball.
1.3 Please refer to the 'readme.txt' document under the 'test_bench' folder for instructions on how to setup the XILINX ISE project.
1.4 Use the following paths:
SIM_DIR: project directory created by ISE Project Navigator, this is where all the Verilog files are located.
SRC_DIR: folder that gets created after you unzip 'theia_gpu_latest.tar.gz'. Contains a copy of the sources, doc,examples, etc.

Step 2 – Copy the input files

2.1 Copy the input files (*.mem) from the SRC_DIR/examples/shaders/example1 to SIM_DIR/
Creg.mem: has control register options (more on this in the documentation).
Params.mem: Scene configuration such as camera origen, light, Axis Aligned Bounding Box position, etc. (more on this in the documentation).
Textures.mem: binary representation of the texture memory (more on this in the documentation).
Vertex.mem: Binary representation of the triangle primitives (more on this in the documentation).

Example 1: Default Shader

Our first example will render 2 texturized triangles. It will use the same shader as the default one in ROM. Let's see how it looks:

```
ENTRYPOINT_ADRR_PIXELSHADER:

 COPY OREG_PIXEL_COLOR CREG_TEXTURE_COLOR VOID
 RETURN  RT_TRUE
```

So, the first thing to notice is the label "**ENTRYPOINT_ADRR_PIXELSHADER**". This is a special label. It tells the compiler that code that follows should be placed under the "User pixel shader section" in the user memory inside the GPU.

Next you see the

```
COPY OREG_PIXEL_COLOR CREG_TEXTURE_COLOR VOID
```

This copies the value stored in internal register "CREG_TEXTURE_COLOR" into the output register "OREG_PIXEL_COLOR". VOID is a special constant; it just stands for a NULL placeholder for the compiler.

CREG_TEXTURE_COLOR stores the color components (R,G,B) from the texture memory for this particular pixel.

OREG_PIXEL_COLOR stores the final color components (R,G,B) for the current pixel.

Next we see the instruction

```
RETURN RT_TRUE
```

This tells the GPU to stop the current subroutine and set the immediate value RT_TRUE as the return value. There are 2 possible return values from subroutines: RT_TRUE or RT_FALSE. It is not really important what value you choose to return for this examples.

In summary, we are just telling the GPU to copy the texture color to the final output color (no lights).

To compile and run the shader do:

>perl $SRC_DIR\scripts**theia_compile $**SRC_DIR\examples\shaders\example1\shader1.shdr

This will re-create the file "Instructions.mem". Copy this file under your SIM_DIR. And re-run the RTL simulation to look the final result.

Example 2: Modify a single color component

So now let's write another shader that increases a single color channel. In this example we will multiply the Blue component for each pixel by 2.

```
ENTRYPOINT_ADRR_USERCONSTANTS:
   SETX C1 32'h20000    //1
   SETY C1 32'h20000    //1
   SETZ C1 32'h40000    //2
   RETURN RT_TRUE


ENTRYPOINT_ADRR_PIXELSHADER:
   MUL  OREG_PIXEL_COLOR CREG_TEXTURE_COLOR C1
   RETURN  RT_TRUE
```

So, the first new thing to notice is the label "**ENTRYPOINT_ADRR_USERCONSTANTS**". This is another special label. It indicates the compiler that the code that follows should be placed under the "User constant section" in the user memory inside the GPU. This section of code is executed once, before the GPU starts traversing the pixels.
Next we see a series of Set instructions. Notice the use of "//" to indicate comments in the code. The SET* family of instructions loads a single immediate value into one of the X, Y or Z components of a register.

```
SETX C1 32'h20000    //1
SETY C1 32'h20000    //1
SETZ C1 32'h40000    //2
```

Here we are loading the values 1,1 and 2 to the X,Y and Z components of the C1 register. Notice that THEIA currently uses fixed point arithmetic with 17 bits of scale, ie. $Q_{15.17}$. This means 15 bits for the integer part and 17 bits for the decimal portion of the number. Therefore the number 1 in $Q_{15.17}$ becomes:

$1 * 2^{17}$ to hexadecimal = 0x20000

and the number 2 becomes:

$2 * 2^{17}$ to hexadecimal = 0x40000

Finally, we return from this subroutine as usual:

```
RETURN RT_TRUE
```

Now that we have these constants defined, we simply multiply the current texture color by this constants and store the result into the output register "OREG_PIXEL_COLOR".
For this multiplication we use the "MUL" operation, it performs 3 simultaneous RADIX-N multiplications for the X,Y and Z components of the operands and write the results back into the destination register.

```
MUL  OREG_PIXEL_COLOR CREG_TEXTURE_COLOR C1
RETURN RT_TRUE
```

The final result is that we multiplied the BLUE color component of the texture by 2.

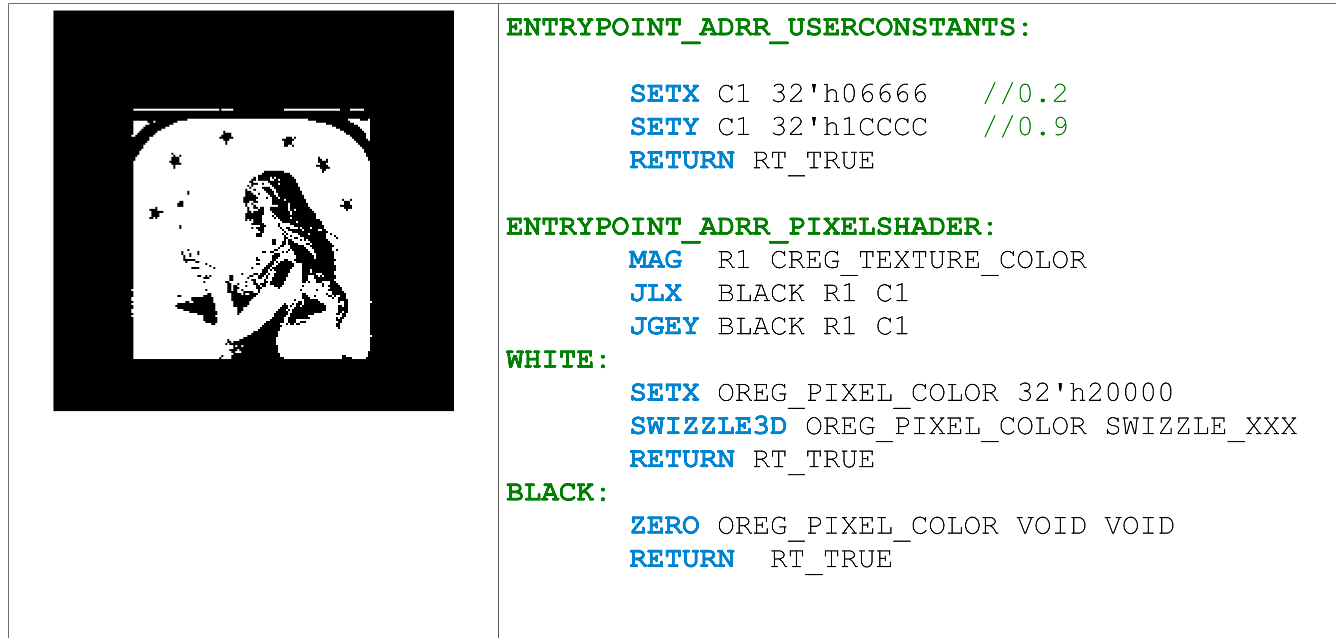
To compile and run the shader do:

>perl $SRC_DIR\scripts**theia_compile $**SRC_DIR\examples\shaders\example1\shader2.shdr

This will re-create the file "Instructions.mem". Copy this file under your SIM_DIR. And re-run the RTL simulation to look the final result.

Example 3: More operations and flow control.

The final output from this example doesn't look very artistic, however it helps to illustrate the use of flow control for the shaders.



```
ENTRYPOINT_ADRR_USERCONSTANTS:

        SETX C1 32'h06666   //0.2
        SETY C1 32'h1CCCC   //0.9
        RETURN RT_TRUE

ENTRYPOINT_ADRR_PIXELSHADER:
        MAG  R1 CREG_TEXTURE_COLOR
        JLX  BLACK R1 C1
        JGEY BLACK R1 C1
WHITE:
        SETX OREG_PIXEL_COLOR 32'h20000
        SWIZZLE3D OREG_PIXEL_COLOR SWIZZLE_XXX
        RETURN RT_TRUE
BLACK:
        ZERO OREG_PIXEL_COLOR VOID VOID
        RETURN  RT_TRUE
```

First we declare 2 constants under the “User constant section” as we learned from the previous example.
Next we see a new operation “MAG”, this calculates the magnitude of a vector.
MAG([x y z]) = SQUAREROOT(x^2 +y^2 +z^2)

```
MAG  R1 CREG_TEXTURE_COLOR
```

Notice we are the using the general purpose register R1 as the destination for the MAG operation. Once you return from this Shader, there is no guarantee that R1 will still hold this value, however you can keep the value if you use a C* register instead.

Next, we see a jump type instruction, JLX.

```
JLX  BLACK R1 C1
```

This will position the instruction pointer into the position marked by the label “BLACK”, only if the condition R1.x < C1.x gets satisfied.
Notice we declared some labels: “WHITE” and “BLACK”. You can use any string for your labels as long as you don't start the label with a special pattern such as ENTRYPOINT*.

Then we have a similar instruction, JGEY. This will position the instruction pointer into the position marked by the label “BLACK”, only if the condition R1.y >= C1.y gets satisfied.

Next we have an instruction called SWZZLE3D. SWZZLE3D lets you re-arrange the X,Y and Z components of a register. In this example the X, Y and Z components of the

"OREG_PIXEL_COLOR", they all become the X component.

Finally we have the instruction ZERO.

```
ZERO OREG_PIXEL_COLOR VOID VOID
```

ZERO just sets the x,y and z component of the destination register to zero.

To compile and run the shader do:

>perl $SRC_DIR\scripts**theia_compile $**SRC_DIR\examples\shaders\example1\shader3.shdr

This will re-create the file "Instructions.mem". Copy this file under your SIM_DIR. And re-run the RTL simulation to look the final result.

That's all for now, please refer to a complete list of the instruction set and registers under the doc section of the project.

Have a nice one!.