RenderMan for Everyone!

Basic Programming Concepts

by Rudy Cortes

This lesson is for those who have never done any programming. It will teach you basic programming concepts that relate to writing shaders in RSL. This lesson wont be the most thrilling and it might be a lot of information to absorb at once, but its contents are very important, so please read along or come back to this page as reference whenever you need it. If you have done programming or scripting before then you can just move on to **RSL 101**.

Variables

Virtually every programming language has variables. A variable is nothing but a placeholder for a value, an alias, a nickname that makes code a little easier to read and write. Variables can be global or local. When writing a shader you will use a good amount of variables to manipulate your data. To create a new variable you simple use the following format

```
Type VariableName = initial value;
```

Where type can be any of float, string, color, vector, normal, point, array, matrix. We will cover these in detail on future lessons. A variable will keep its value until changed by an assignment operator. To exemplify how variables are used lets look at some simple code

```
float a = 10;
float b = 2
float c = a * b;
a = c * b;
```

So, the initial value for a is 10 and for b is 2, c's initial value is 20 (a * b = 10 * 2). Then a receives the value of 40. At this point the original value of a is lost forever since you overwrote a with c*b. The only way to get access to the original value would be to create another variable and store the value of a there before overwriting a, for example

```
float atemp = a;
a = c * b;
Float d = atemp * b;
```

'atemp' holds the original value of a so the initial value of 'd' becomes 20. With time you will become quite good at keeping track of the values of your variables. Variables can also be parameters to a shader or function. A parameter is usually a variable that you have access to once the shader is already compiled. All other variables are said to be 'hard coded' into the shader. Parameters are the controls you give the user to change the appearance of the shader.

Assignment Operators

To assign a value to a variable you can type the name of the variable followed by the = sign and then the value you want to assign. There are other assignment cases we should go over. For example lets assume you want to do something like this

```
float a = 10;
f loat b = 5;
a = a + b;
```

You would save yourself a little typing by writing "`a += b;`" Which means exactly the same as what we originally typed. You can do this with +=,-=, *=, /=. It wont save that much time, but I am lazy and I take every shortcut I can.

Functions

Functions are very simple to understand. They are chunks of code neatly packed to be used over and over with ease. We will learn to write our own functions later on. Most functions require parameters to be passed into it. If you read your RiSpec you will see what kind of parameters a function requires. For example the shading functions 'ambient' and 'diffuse' are defined as:

```
color ambient();
color diffuse(normal N);
```

The first line returns (outputs) a color and requires no parameters to be passed, while the diffuse function also returns a color but requires a normal to passed to it. Functions in RSL have the following format:

```
type functionName (parameters){
some code……
Return value}
```

A very simple example of a function would be

```
float multiply ( float a; float b)
{
return a * b;
}
```

This function multiplies the values of the two passed parameters. To use this function you could write something like:

```
float foo = 10;
float bar = 5;
float foobar = multiply(foo,bar);
```

foobar will receive the returned value of the multiply functions. Note that in RSL all functions need to be declared before their first use. You can do this by putting them before you start the actual shader, inside the shader before you use it for the first time, or in a header file.

Control Structures

RSL code is interpreted by machines from top to bottom. The machine interprets the commands you give it and performs the proper action then moves on the next line. It is very common however for a program to include some code that only executes if certain conditions are met. Or sometimes you might want the same chunk of code to repeat itself over and over until another condition is met. These are called control structures and RSL supports some that are very useful.

'if' and 'if then'

The most basic control structure is the 'if' command. Its very easy to implement and understand. All it does is restrict the execution of some code based on whether the 'test argument' is true. For example

```
float foo = 10;

if ( foo == 8){
foo = 20;}
```

These lines initialize the variable foo to be 10. Then the 'if' command tells us "if foo is equal to 8, execute the code between the

braces". Note that to compare the value of foo to 8 we use 2 '=' signs. This example could have also been written without the curly braces that enclose 'foo = 20'. The braces are only needed if you need to execute more than one line of code. Since foo is not equal to 8 then the code between the braces will not be executed and foo will retain its value of ten. If we changed the value of foo to 8 then by the end of our mini-program foo would have a value of 20.

Sometimes you want to perform an action whether a condition is met or not. This is where you would use the 'if else' commands. They are used very similar to the plain 'if' but its extended a little bit. Using the previous code we have:

```
float foo = 10;

if ( foo == 8){
foo = 20;
}else{
foo = 30;}
```

Here no matter what value food was originally it WILL be changed, more than likely to 30, unless foo's original value is 8. When using the if command you will have to pass a conditional argument for it to test. These arguments are

== equal
=! Not equal
< less than
> greater than
<= less or equal to
>= greater or equal to

You can pass any number of conditions to the if statement. You do this if you want any of several conditions to execute the contained code, or if you want the code to execute only if all the conditions are met.

```
if (( foo == 8) || (foo == 12))…

if (( foo == 8) && (foo == 12))…
```

The first line will execute the code if either one of the two conditions is true. The second line will only execute if both conditions are met. I'm sure you figured out that || means or and && means and. If you didn't, don't worry, it took me a while to figure it out. A very popular use of 'if' statements is on the use of texture files. You usually want a texture file to be used only if it is provided, if its not provided you can pass a default color or a warning color to the renderer.

For – while

Sometimes you want to execute the same lines of code over and over until you tell it to stop. This can be done with conditional loops. They are very useful for generating turbulence patterns or if you want to generate patterns that have repeating features. Here is how they are defined

```
for ( initial expression; test expression; new expression){
some code…

}
```

If the test expression is true then the new expression is executed and the code between the braces is run once. Then the test expression is run again. If true it does the same as before. The loop will continue to run until the test expression is false.

```
float foo;
float bar = 0;

for (foo = 1;foo <= 5; foo +=1){
bar += foo;
```

```
}
```

Here the loop will execute five times and by the time it ends bar will have a value of 15 since the value of foo is 1 the first time the loop runs, 2 the second, 3 the third and continuing all the way to 5. The for loop will only be activated if the condition returns true, if not it will skip it completely. Another variation of the for loop is the 'while' statement. The while statement is defined as

```
while (test expression){
some code}
```

As you see the main difference is that while doesn't have an initialization value or an incremental expression.

Header Files

As you begin to code more shaders you will come up with some custom functions that are going to be very handy to keep at hand. Instead of always typing them into your shader source file or copying it from another file, you can simply have a separate file and you can have the compiler load that file while its compiling your shader. These are called header files and are very, very handy. With the use of header files you can reduce the time it takes to write a shader dramatically.
A header file usually has a '*.h' extension and are loaded into your file by using the include command at the top of your shader source file with either of the following commands

```
include <myheader.h> or
include "myheader.h"
```

It is very common to include pre-processor macros and defines in header files, actually you can include them anywhere you wish. A pre-processor is a part of the compiler that runs before compiling actually takes place. It loads into the file any header files and initializes any macros or defines. There are several important pre-processor tags that you will use on writing shaders. I will list them as an example and then explain their meaning

```
#define HEIGHT 5.10

#ifndef MYHEADER_H
#define MYHEADER_H
#endif

#ifdef MYHEADER_H
```

The first line defines a constant called Height, this constant will have the same value everywhere in the shader and can not be changed at anytime. '*#ifndef*' means "if not defined", this is similar to the 'if' **control structure** but at pre-processor level. In this particular case if the file 'myheader.h' has not been included anywhere, the preprocessor moves through the next commands until it hits '*#endif*'. This command lets the pre-processor know that here is where the conditional arguments stop. The '*#ifdef*' command means "if defined" This macro can be useful to avoid name conflicts in functions and header files. We will look at how header files are written in further chapters

Back to the Main Hall

Home