R4 Programming Tutorial

Jussi Härkönen

Version 1.0

Table of Contents

1 Introduction

R4 is a standalone OpenGL accelerated music visualization program aiming to produce stunning 3D graphics in real time. It is designed to be highly customizable and contains a high speed scripting engine. R4 also has a built-in web server that allows the user to control the visuals from any other network-connected computer without disrupting the visuals. R4 and its precursor, R2/Extreme, were created by Gordon Williams.

For users who want to create their own R4 scenes, R4 features sophisticated tools for 2D and 3D graphics. The tools include numerous highly configurable and textured 3D object structures, morphing objects, music reactivity, support for self-repeating structures and even a capability to incorporate live video feeds.

This tutorial aims to guide you to create your own custom scenes with R4. Unfortunately, a little experience in *some* programming language is *almost* necessary if you want to learn R4 programming. However, if you are familiar with C programming, learning R4 will be relatively easy. Knowing basics in OpenGL programming will also help you.

This tutorial is not a complete documentation of R4. However, it should give you the basic knowledge on how R4 scenes work and help you to continue exploring R4 on your own.

Some files associated with this tutorial can be found in the subfolders of the directory where you extracted the tutorial archive. The files are discussed later in this tutorial.

If you find errors or mistakes in this tutorial, want to send me feedback or if you want to contribute to developing this tutorial, please send me email to the address **Violet at VioletIndustries dot com**.

1.1 Setting up the R4 Development Environment

In this section, instructions for installing the necessary tools for R4 programming are given. If you have not done it yet, go to **http://www.rabidhamster.org/R4/** to download and install the newest version of R4. The default installation directory is **C:\Program Files\R4**. Installation instructions for Crimson Editor are given below. Crimson Editor is a useful free programming editor but any other text editor can be used as well.

Go to **http://www.crimsoneditor.com/**, download Crimson Editor and install it (the default folder is **C:\Program Files\Crimson Editor**). Then extract the contents of **r4_syntax.zip** found in **R4 Syntax for Crimson Editor** directory to the folder you installed Crimson Editor. This will install colour highlighting for the R4 programming language.

To add a macro that runs R4, run Crimson Editor and select *Conf. User Tools…* in the *Tools* menu. Fill the fields according to the values in Table 1.1, check the *Save before execute* checkbox and select *OK*. Now you can edit R4 custom scene files in R4, execute them by simply pressing F5 or selecting *Run R4* from the *Tools* menu. The R4 file you are editing must be saved to **R4\data\predefine** directory. It is a good idea to create a subfolder **R4\data\predefine\disable** and move all **.r4** files from **predefine** directory to **predefine\disabled** scenes. Save only the file you are editing to the predefine directory. Consequently, it will be loaded when you run R4.

Table 1.1 A user-defined macro configuration for launching R4.

Field	**Value**
Menu Text	Run R4
Command	R4.exe in R4 installation directory (default is **C:\Program Files\R4\R4.exe**)
Argument	(Empty)
Initial Dir	R4 installation directory (default is **C:\Program Files\R4**)
Hot Key	F5 (or any other key)

You can also configure a key in R4 to reload a scene. Run R4 in Crimson Editor by pressing F5. Press ESC to bring up the menu and select *Settings, Keys, Assign a key, F5, <<< scene keys >>>, Reload Scene (BETA)*. Now you can keep R4 running while you work with Crimson Editor and reload the scene to R4 by activating the R4 window and pressing F5.

1.2 R4 Subdirectories

It is essential to know what can be find in R4 subdirectories. The contents are described in Table 1.2. The most important directory is **R4\data\predefine** where you should save your custom scene file. If you use custom textures, water morphs or point morphs, they should be saved to the corresponding directories **R4\data\tex**, **R4\data\watermorph** and **r4\data\pointmorph**, respectively.

Table 1.2 R4 subdirectories and their contents.

Directory	**File(s)**	**Description**
R4\dll\	***.r4**	Definitions of modules available in R4
R4\docs	**brander.html**	Brander wizard instructions
	readme.html	Readme file containing general information about R4
	script.html	Available script commands
	shader.txt	Shader commands
R4\utils\brander	**R4Brander.exe**	R4 brander wizard
R4\data\model	***.raw, *.asc, *.md2**	3D Model files
R4\data\pointmorph	***.PointMorph**	Point morph files
R4\data\predefine	***.r4**	Load directory for R4 scene files
R4\data\tex	***.jpg, *.png, etc.**	Textures used for scenes
R4\data\watermorph	***. watermorph**	Watermorph files

2 R4 Programming

In this chapter, R4 programming elements and syntax are discussed. Because the R4 programming language is basically a stripped-down version of C, learning R4 programming is much easier if you are familiar with C/C++ or Java.

A custom scene (simply referred as *scene*[1]) defines what R4 should draw onto the screen. Basically, a custom scene is a program that is interpreted by the R4 scripting engine. Scenes are stored in **.r4** files.

In general, all R4 identifiers are *case insensitive*, that is, the lower and upper case letters are used interchangeably. All rows, excluding conditional expressions, should end with semicolon ';'.

2.1 Overview to Custom Scene File Structure

An R4 custom scene file consists of a *header section*, variable definitions and function definitions. The modules to be included to the scene are defined in the header. In addition, the header includes scene name and author fields. A header looks like this:

```
scene (
    "name" = "Scene Name";
    "author" = "Author Name";
    MODULETYPE1 moduleName1(moduleParameters1);
    MODULETYPE2 moduleName2(moduleParameters2);
    // ...more module definitions
)
```

This defines a scene containing module object instances named `moduleName1` and `moduleName2` of type `MODULETYPE1` and `MODULETYPE2`, respectively. For example, defining

```
scene (
    "name" = "Example Scene";
    "author" = "Jussi Härkönen";
    SOLID background();
    TEXTURE texFish();
    MEDUSA fish(background, texFish);
)
```

creates a `MEDUSA` module called `fish` that is drawn on `background` and that uses the texture `texFish`.

In addition to the header, you should define *init, render* and *reset* functions according to the syntax

```
void init() {
    // Initialization
}

void reset() {
    // Reset
}

void render() {
    // Rendering
}
```

The code inside the curly braces after `void init()` will be executed when R4 loads the scene. `reset()` is called just before R4 starts drawing the scene and `render()` is called every time before a frame is drawn. Consequently, you should place your initialization code to `init()`, rendering and scene updating code to `render()` and reset code to `reset()`.

User-defined variables are defined according to the syntax

[1] In Finnish, scene should be pronounced with 'k' as 'skene'.

```
vartype varName;
const vartype varName = constValue;
```

`Vartype` specifies the variable type and **`varName`** the variable name. You can define a variable with a constant value by using the **`const`** keyword. For example,

```
int numFishes;
const int MAX_NUM_FISHES = 10;
```

defines an integer variable **`numFishes`** and an integer constant **`MAX_NUM_FISHES`**. You cannot change the value of a variable defined with the keyword **`const`** after its definition or you will get a compile error when you try to run the scene. In addition, if you want to initialize your not **`const`**-defined variables, you have to do it in the **`init()`** function. Note that when you define a floating point constant of type const float, *you have to specify at least one decimal* place or you will get a compiler error.

2.2 Number Data Types – int and float

There are two different classes of data types – *modules* that are used in the scene header specification, and the **`int`** and **`float`** data types used for custom variables. Here, **`float`** and **`int`** data types are discussed.

The values in all variables and arrays is undefined scene start-up. If you want to initialize your variables, you should do the initialization in the **`init()`** or **`reset()`** function.

You can define several variables on a same row by writing

```
varType var1, var2, var3;
```

However, this syntax cannot be used with the **`const`** specifier. You can set the value of a variable with the equality operator **`=`**. Furthermore, several variables can be given the same value by writing

```
var1 = var2 = var3 = varValue;
```

Note that R4 defines the data type of a number based on the existence of a decimal point. Consequently, if **`var1`** is an integer and **`var2`** a floating point number, the expression

```
var2 = var1 / 5;
```

would result in integer division returning an integer to **`var2`**. However, writing

```
var2 = var1 / 5.0;
```

would result in a floating point number.

2.2.1 The `const` Specifier

You can use the **`const`** specifier with **`float`** and **`int`** types. However, you cannot use other constants or carry out mathematical operations when you define constants. For example, the definitions

```
const int a = 4 + 5; // error
```

or

```
const int a = 4; // ok
const int b = a; // error
```

would generate an error.

Note that when you define a floating-point constant, you *have to define at least one decimal*, even if it is zero. Otherwise you will get a “Syntax error – wanted <floating point>” compile error. Furthermore, *constants must be nonnegative*.

It is recommended to use constants defined with the **`const`** keyword instead of magic numbers in your code. This is very helpful if you want to change the value of a constant used in several places in the code. It will also make the code more readable for you and for others.

2.2.2 Arrays

Furthermore, you can define arrays of types **`float`** and **`int`**. Arrays are specified according to the syntax

```
vartype arrayName[numElements];
```

where **`numElements`** specifies the number of elements in the array. You cannot use the **`const`** specifier with arrays. The elements should be indexed from **`0`** to **`numElements-1`**. Using larger indices than specified as the array size may not cause an error, but will overwrite the data in other scenes and generally make R4 unstable. You should therefore always use indices from **`0`** to **`numElements-1`**.

2.2.3 Strings

Strings are represented as integer arrays in R4. If you define an integer array

```
int string[20];
```

You can copy a text string into **`string`** with the command

```
strcpy(string, "String to be copied");
```

A pointer to the **`string`** array is denoted with the '**`*`**' character, for example **`*string`**. This is used in some native R4 function definitions.

2.3 Conditionality

A *boolean* variable can have one of the two values true and false. As in almost every programming language, true is represented by 1 and false by 0. Logical expressions for comparing numbers can be constructed using *relational operators*. The available operators are listed in Table 2.3.

Table 2.3 Available operators in logical expressions.

Operator	Description
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to

Assuming you have defined the integers **`i`** and **`retVal`**, you can, for example, write

```
i = 5;
retVal = i == 5;
```

Because the expression **`i == 5`** is true, this will insert 1 (that is, true) to **`retVal`**. Writing

```
i = 0;
retVal = i > 0;
```

inserts 0 into **`retVal`**, as the expression **`i > 0`** is false. Furthermore, you can combine logical expressions with the 'and' operator **`&`** and the 'or' operator **`|`**. Writing

```
i = 1;
retVal1 = (1 < i) & (i < 10);
retVal2 = (i == 0) | (i == 1);
```

will insert 0 to **`retVal1`** (as 1 is not in the open interval]1, 10[and **`(1 < i)`** is false) and 1 to **`retVal2`** (as **`i == 1`** is true).

2.3.1 If and Else

You can use the logical expressions discussed above to control program execution flow. You can do this with the **`if`** statement defined according to the syntax

```
if ( <conditional expression 1> ) {
    // Code executed if the expression 1 is true
}
else if ( <conditional expression 2> ) {
    // Code executed if expr 2 is true and expr1 is false
}
else {
    // Executed if none of the if/else if statements is true
}
```

If the conditional code consists only of one line, you can leave out the curly braces. For example,

```
if (i < 0)
    numFishes = 0;
else if (i <= 10) {
    // Curly braces are needed
    numFishes = i;
    fishAdded();
}
else
    numFishes = 10;
```

will set **`numFishes`** to **`i`** if **`i`** is between 0 and 10. Note that if the one of the conditional expressions is true, the following **`else if`** expressions are not evaluated even if they would be true.

2.4 Loops

You can use **`for`**, **`while`** and **`do-while`** loops. The syntaxes for the different loops are

```
for (loopVar = initVal; <bool expression>; <update loopVar>) {
    <loop code>
}

while ( <boolean expression> ) {
    <loop code>
}

do {
    <loop code>
} while ( <boolean expression> )
```

For example, assuming **`int i`** has been defined, you can write

```
for (i = 0; i < numFishes; i=i+1)
    fishAge[i] = fishAge[i] + 1; // Increase fish age

while (fishFood < numFishes) {
    numFishes = numFishes - 1;
    killFishes = killFishes + 1;
}

do {
    killFish();
    killFishes = killFishes - 1;
} while (killFishes > 0)
```

As for **`if`** expressions, you can leave out the curly braces if the loop code only contains one row of code. You should note that unlike C and Java, R4 contains no ++ or -- operators.

2.5 Functions

In addition to the functions **`init()`**, **`reset()`** and **`render()`**, you can define your own functions. Functions are defined according to the syntax

```
    returnType functionName(<parameter list>) {
        <function code>
        result = returnValue;
    }
```

You can, for example, define a function

```
    float randNum(float minVal, float maxVal) {
        // Return a random number between minVal and maxVal
        result = minVal + ((maxVal - minVal) * rand());
    }
```

This function will return a random number between **`minVal`** and **`maxVal`**. The return value is assigned to a variable labelled **`result`**. You can use the function to assign the return value to a variable **`var`** by writing

```
    val = randNum(-1, 1);
```

If the return type of a function is void, the function returns no value. It is impossible to use arrays as return values or function parameters. The functions must be defined in the scene file before they can be called.

2.5.1 The Functions `init()`, `reset()` and `render()`

When R4 loads the scene for the first time, the **`init()`** function is called. Also **`reset()`** is called *after* the init call at first start-up. Furthermore, **`reset()`** is called just before R4 decides to change to another scene. It is noteworthy that the variables of a scene are retained during the whole R4 session. Consequently, *you should place the initialization of the variables that are altered in* **`render()`** *to the* **`reset()`** *function*. Otherwise, when the scene is loaded for second time, the variable values from the previous scene load are used. This may produce unwanted results.

2.6 Native Functions and Variables

The native functions and variables available in R4 are defined in the file **native.r4** located in **R4\data** directory. As most of them are quite self-explanatory, only the sound interaction and time variables are discussed here.

Note, however, that trying to use the **`pi`** constant will generate a compile error. Instead you can use the **`pi()`** function or explicitly define **`pi`** as a constant in your code.

2.6.1 Interaction with Music

R4 provides some variables that contain information about the music currently playing. These variables are listed in Table 2.4.

`soundA`, **`soundB`** and **`soundC`** are suitable to be used in, for example, gain calculation for different purposes to make your scene react to music. The **`waveLeft`** and

`waveRight` arrays can be used to draw the signal curve, that is, a waveform. **`specLeft`** and **`specRight`** provide the Fourier transform of the signal. The elements in the beginning of the arrays describe the magnitude of low frequencies, whereas the elements towards the end of the arrays describe the magnitude of high frequencies.

Table 2.4 Variables for music interaction.

Type	Variable name	Description
`float`	`soundA, soundB soundC`	Change in bass, middle and treble frequencies, respectively.
`float array`	`waveLeft[512], waveRight[512]`	The signal values.
`float array`	`specLeft[512], specRight[512]`	The frequency content of the signal (the Fourier transform).

2.6.2 Time Variables

There are two available variables indicating time. The variable **`time`** gives the time elapsed since the scene start-up and the variable **`timePass`** gives the time elapsed since the previous **`render()`** call. These variables are essential to make a scene change as a function of time.

2.7 Modules

The modules used in a scene must be defined in the scene header. *A background module* and a varying number of *input modules* must be defined for most of the modules.

Modules are defined according to the syntax

```
MODULETYPE1 moduleName1(background,input1,input2,...) xz, yz;
```

where **`background`** is used as background for the module and the following input modules are usually used as textures. The optional parameters **`xz`** and **`yz`** specify the size of the module output. **`xz`** and **`yz`** should be greater than or equal to 16 and be of type 2^n, where n is an integer.

Modules usually draw textured objects onto the background. An arbitrary number of modules can be defined, and the last defined module will be drawn onto the screen. The module used as background or input for other modules must have been defined earlier. Consequently, the first module must be a module that does not need a background or input module[1]. It is usually a **`TEXTURE`** or a **`SOLID`** module.

2.7.1 Accessing Module Member Variables

Most of the modules contain member variables that define the internal activity of the module. You can access the variables with the point operator '**`.`**' after the module name. If a **`SOLID`** object labelled red has been defined, its **`col`** member variable can be set by writing

```
red.col = rgb(1, 0, 0);
```

This will set the colour of the **`SOLID`** object to red. The **`rgb`** function converts the red, green and blue floating point colour components into a 32-bit integer representation of the same colour.

[1] Faders and overlays make an exception to this rule as they can access external surfaces. Faders and overlays are discussed in sections 2.9 and 0.

2.8 Shaders

Many of the available modules include a string member variable labelled **`shader`**. *Shaders* define how the modules or textures are drawn onto the screen. Most of the shader commands are shorthand for OpenGL functions that will be called.

You can set the shader string of a module with the command

```
strcpy(moduleName.shader, "ShaderString");
```

This is usually done in the **`init()`** function. If the module contains multiple skins, you must specify the skin pointer as one in a shader array, that is,

```
strcpy(moduleName.shader[skinIndex], "ShaderString");
```

where **`skinIndex`** gives the number (starting from 0) of the skin. The **`TUNNEL`** module, for example, supports up to 4 skins that are numbered from 0 to 3.

Every command in a shader is represented by a single character, followed by more characters representing parameters. All commands are separated by a semicolon '**`;`**'. Furthermore, every shader must end with a semicolon. The available shader commands and their descriptions are given in Table 2.5.

The available *blending modes* used by the blending mode command **`B`** are listed in Table 2.6. The blending mode defines how the resulting pixel colour is calculated using source and destination texture pixels. If the source colour is denoted with c_S, the destination colour with c_D and the source and destination blending coefficients with B_S and B_D, the result of the rendering operation is given by the equation

[illegible] **(1)**

Some common blending modes are given in Table 2.7. Another interesting blending mode is **`Bcc`**. According to Table 2.6, **`Bcc`** means that $B_S = 1 - c_D$ and $B_D = 1 - c_S$. Inserting these to equation (1) gives

$$\begin{aligned} c &= (1 - c_D)c_S + (1 - c_S)c_D \\ &= c_S + c_D - 2c_S c_D \end{aligned}$$

Inserting different values to the equation reveals that the result is large if either c_S or c_D is large. If they both are small or large, the resulting colour value is small. If the source texture is black and white, this corresponds to using the source as inverting mask, inverting the colour for $c_S = 1$ and leaving the colour unchanged for $c_S = 0$.

Table 2.5 Available shader commands.

Letter	**Format**
`T`	`Tx;`
Parameters – `x`: Number of texture starting from 0. **Description:** Specifies the number of texture to be used. You can only use one texture for each skin of a module.	
`B`	`Bxy;`
Parameters – `x/y`: Source/destination colour multiplier. **Description:** Specifies the blending mode. See Table 2.6 for available modes.	
`W`	`Wx;`
Parameters – `x`: A floating point wireframe line width. Although this is not required. **Description:** The vertices are drawn using wireframes instead of textures	
`D`	`D;`
Parameters: None. Description: Enables depth testing and clears the depth buffer.	
`d`	`d;`
Parameters: None. **Description:** Enables depth testing but does not clear the depth buffer. This can be very useful with transparent textures on, for example, a `TUNNEL` module using several skins with different radii. The first skin with largest radius uses `D` to fill the depth buffer and the other skins use `d` and, consequently, the same depth buffer.	
`F`	`Fx;`
Parameters – `x`: `B` (cull Back) or `F` (cull Front). **Description:** Specifies the culling mode. If cull back is used, the front side of the surfaces drawn are invisible and the back side is visible. If cull front is used, the front side of the surfaces drawn are invisible.	
`C`	`Cr,g,b,a;`
Parameters – `r,g,b,a`: Red, green, blue and alpha values in the interval [0,1]. **Description:** Multiplies texture colours with the given colour coefficients.	
`G`	`Gxy;`
Parameters – `x`: `S,T,R` or `Q` specifying texture coordinate. `S` and `T` specify the x and y coordinates, whereas `R` and `Q` are rarely used in R4. **Parameters** – `y`: `O` (object linear), `E` (eye linear) or `S` (sphere map) **Description:** Specifies how the texture coordinates are generated. Using `GSS;GTS;` generates the x and y coordinates using sphere mapping making the texture look like a reflection. The texture is mapped like the object would be perfectly reflective and surrounded by an infinitely large sphere. See the liquid metal robot in *Terminator 2* for an illustration of *environment mapping* and the OpenGL documentation for more information on texture generation.	
`P`	`Pabx,y,z;`

Parameters – `a`: `S,T,R` or `Q` specifying texture coordinate. `S` and `T` specify the x and y coordinates.
Parameters – `b`: `E` (eye plane), `O` (object plane).
Parameters – `x,y,z`: A floating point vector.
Description: Specifies a texture generation plane. The texture coordinate c specified by `a` for a vertex (v_x, v_y, v_z) is $c = (v_x, v_y, v_z) \cdot (x, y, z) = xv_x + yv_y + zv_z$. If, for example, [illegible], the texture coordinate is $c = 0.1 \cdot v_x + 0 \cdot v_y + 0 \cdot v_z = 0.1 v_x$, that is, the signed distance of the vertex from the yz-plane multiplied by 0.1. The `b` parameter specifies if the vertex is given in view or world coordinates. The command `GSO;GTO;PSO0.0005,0,0;PTO0,0.0005,0;` generates the x and y coordinates as distances from the yz and xz planes, respectively.

`L`	`Lx;`

Parameters – `x`: A floating point value specifying the amount of blur.
Description: A higher value of `x` will increase the blur of the texture. 0 means no blur. This only works for static textures that were loaded from a file.

Table 2.6 Available blending mode flags.

Source colour coefficient		**Destination colour coefficient**	
Character	Coefficient	Character	Coefficient
`0`	0	`0`	0
`1`	1	`1`	1
`C`	Destination colour	`C`	Source colour
`c`	1 - destination colour	`c`	1 - source colour
`A`	Source alpha	`A`	Source alpha
`a`	1 - source alpha	`a`	1 - source alpha
`D`	Destination alpha	`D`	Destination alpha
`d`	1 - destination alpha	`d`	1 - destination alpha
`S`	Alpha saturate		

Table 2.7 Common blending modes

Command	Description
`B10;`	Draw using source colour
`BAa;`	Draw the source colour with transparency
`B11;`	Additive drawing
`BA1;`	Additive drawing using source alpha

2.9 Overlays

Overlays are basically custom scenes that are applied to the currently running scene. One difference compared to custom scenes is that you have to start the header module with the word **`OVERLAY`** instead of **`SCENE`**. Furthermore, you can access the currently running scene as a module called **`scenefrom`**. The **`scenefrom`** module acts exactly as any user-defined module. You can, for example, write in the overlay header

```
OVERLAY (
    SOLID black();
    WARP w(black, scenefrom);
```

```
	)
```

This will make the **WARP** module to be applied to the current scene output.

2.10 Faders

Faders are used in transition from one scene to another. Like overlays, faders are special type of scenes. In the header section of the fader, you can access the **scenefrom** (the previous scene) and **sceneto** (the target scene) modules. In the **render()** function, you can access a variable labelled **finished**. When **finished** is set to 1, the fader will exit and the target scene is rendered to the screen. Note that **finished** should be reset to 0 in the **reset()** function in order to make the fader work more than once during an R4 session.

2.11 Writing Comprehensible Code

There are some simple ways to make the source code more readable and easier to understand. You can write comments in the code after two backslashes '**//**' (but '/*' like Java and C doesn't work). Remember that writing comments is free! When you are writing a custom scene, you should take the time to write some comments explaining what the code does. This will make the code more readable (or, usually, less incomprehensible) – for others and for yourself.

It is recommended to use tabulator indentation for code segments outlined by curly braces. Code can be further structured with empty lines separating logical sections of code. Using empty space in equations make them easier for eye to read.

Although R4 is case insensitive, it is recommended to use cases to make the code more readable. In general, constants and module types are written with upper case, whereas variable names and functions are written in lower case. If a variable name consists of several words, the first letter of every word after the first one should be upper case. This makes the name easier to read.

The variable names should tell what the variable does. Avoid using magic variables with meaningless names like **a**, **b** or **temp2**. Furthermore, avoid magic numbers in your code and define constants instead.

Here is a small example encapsulating the above guidelines.

```
void fishFarm() { // Farm starts with capital
	// Indentation
	// Increase fish count
	numFishes = numFishes + 1; // Fish starts with capital

	// Empty line before if expression
	if ( (numFishes == MAX_FISHES) & (fisherman == true) )
		// Note empty spaces in the logical expression
		// Indentation
		// The fisherman kills all the fish
		numFishes = 0;
} // End of fishFarm
```

3 Scene Walkthrough – The Caribbean Sea

In this chapter, we are going to construct a simple scene called Caribbean Sea Example. Because it primarily aims to illustrate some most common modules, it is not the most impressive one. However, it features some essential R4 modules and techniques. The example can be found in the file **Caribbean Sea Example.r4** in the tutorial subfolder **Caribbean Sea**.

3.1 Waveforms and Flowfields

The **`waveLeft`** and **`waveRight`** arrays are used in order to draw *a waveform*, a segment of the sound signal, with the **`GL`** module. The Caribbean Sea also features *a flowfield* consisting of **`BUFLOAD`**, **`BUFSAVE`** and **`MAP`** modules.

The **`MAP`** module consists of a 32-by-24 grid of points. For every point, the user can specify how the grid points are mapped from the source surface to the destination surface. In other words, you can specify how much every point is offset in the horizontal and vertical directions. Furthermore, you can specify a target colour for red, green and blue colour components to which the colours are slowly attenuated. The map destination surface is saved using the **`BUFSAVE`** module and it is loaded from the **`BUFLOAD`** module during the next **`render()`** call. Thus, the mapping specified by **`MAP`** is recursively applied. If **`MAP`** is initialized properly and something is drawn to the **`BUFLOAD`** surface, an impression of flowing colours is obtained. Flowfields are the main components in, for example, the MilkDrop and G-Force visualizations.

3.1.1 Header Section

```
SCENE (
    "name" = "The Caribbean Sea Example";
    "author" = "Jussi Härkönen";
    SOLID black();
    BUFLOAD bl();
    MAP map(black, bl);
    GL gl(map);
    BUFSAVE bs(gl);
)
```

The header definition contains some string fields above the module. The name of the scene is specified by the **`"name"`** field and the author by the **`"author"`** field as defined above.

A **`SOLID`** module **`black`** is created to be used as a background for the **`MAP`** module[1] **`map`**. A **`BUFLOAD`** module **`bl`** is used to load the screen output from previous **`render()`**. Furthermore, it is used as input module for the **`map`** module. The **`GL`** module **`gl`** is used to draw the waveform onto **`map`**. Finally, the **`BUFSAVE`** module **`bs`** is used to save the **`gl`** module to be loaded from **`bl`** in next **`render()`**. As **`bs`** (that equals **`gl`**) is the last module defined, it is shown on the screen.

3.1.2 Variable Definitions

```
// Loop index
int i;
// Temporary x coordinate value
float x;
// Loop indices for the MAP module
int xix, yix;
```

[1] Actually **`black`** is not necessarily needed and **`bl`** could be used instead as the background for **`map`**. **`bl`** is always initialized to black when the scene is loaded.

```
    // Coefficient for map flow magnitude
    const float MAP_FLOW = 0.1;
    // Wave height scale
    const float WAVE_SCALE = 0.2;
```

The integer variable **i** will be used as a **for** loop index for a loop drawing the waveform. The floating point number **x** is used in the same loop to store the x coordinate value. The integers **xix** and **yix** (x index and y index) will be used in the two **for** loops that initialize the **map** module.

The constant **MAP_FLOW** defines the speed of flow of the flowfield. **WAVE_SCALE** defines the amplitude of the waveform.

3.1.3 Initialization

```
    void init() {
        // Do not correct for aspect ratio
        gl.aspect = false;
        // Set shader
        strcpy(gl.shader,"W3.5;");

        // Initialize map
        for (yix = 0; yix < 24; yix = yix + 1) {
            for (xix = 0; xix < 32; xix = xix + 1) {
                // Set map flow speed in vetical direction
                map.y[yix][xix] = -MAP_FLOW * sign(yix - 11.1);

                // Set color attenuation
                if (yix < 12) {
                    // The lower half
                    map.r[yix][xix] = 0; // Filter all red
                    // Filter green progressively
                    map.g[yix][xix] = yix/12.0;
                    map.b[yix][xix] = 1; // Do not filter blue
                }
                else { // (yix >= 12)
                    // The upper half
                    // Filter red progressively
                    map.r[yix][xix] = (24 - yix)/12.0;
                    map.g[yix][xix] = 0; // Filter all green
                    map.b[yix][xix] = 1; // Do not filter blue
                }
            }
        }
    }
```

First, we set the **aspect** data member of **gl** to **false**. This causes **gl** to not correct for the aspect ratio. If it was set to **true**, **gl** would transform the **bl** input module into a rectangular shape. Then we set the shader string of **gl**. The **W** character makes **gl** draw wireframes, that is, objects only consisting of lines. As we are only going to draw lines, this sounds pointless. However, this allows us to define the line width for the waveform as the number after **W**.

The **map** module is initialized in two loops inside each other – one for x and one for y coordinate. As **map** contains a 32-by-24 grid, the loop indices **yix** and **xix** go from 0 to 23 and 31, respectively.

The two-dimensional data member array **map.y[24][32]** defines how much the map points are offset in vertical direction. A positive offset value results in movement downwards and a negative offset upwards. The constant **MAP_FLOW**[1] defines the magnitude of offset, whereas the expression **sign(yix - 11.1)** is -1 for **yix** ≤ 11 and 1 for **yix** > 11. Note that if 11.0 would be subtracted from **yix**, the result would be 0 for **yix** = 11 and **sign(yix - 11)** would return 0. This would cause the corresponding map points to not move at all.

In the following if-else structure, the **map** target colours are specified. The colour values specify the colour to which the objects drawn to map are slowly attenuated to. The values are stored in 24-by-32 member arrays **r**, **g** and **b**. For **yix** ≤ 11, red is set to 0 and blue to 1. Green is a linear function of **yix**. Because the point (0,0) is the lower left corner and (32,24) is the upper right corner in map coordinates, this makes the green colour to be attenuated less towards the vertical centre of the screen, that is, the line **yix** = 11. For **yix** > 11, red is a linear function of **yix** being 1 at the centre and 0 at **yix** = 23.

3.1.4 Rendering

```
void render() {
   // Set load buffer handle
   bl.handle = bs.handle;

   // Clear the gl command queue
   gl.clear();
   // Translate so that x and y are in the interval [-1,1]
   gl.glTranslate(0, 0, -2.414);
   // Begin drawing lines
   gl.glBegin(GL_LINE_STRIP);

   for (i = 0; i < 512; i = i + 1) {
      // Calculate the x coordinate that corresponds to i
      x = 2 * i / 511.0 - 1;
      // Draw next point in the line strip
      gl.glVertex(x, WAVE_SCALE*(waveLeft[i]+waveRight[i]),0);
   }

   // Finished drawing lines
   gl.glEnd();
}
```

First, the **handle** member variable of the save buffer is saved into the load buffer **handle**. This must be separately done for every **render()** call. After that, the command queue of the **gl** module is first emptied by calling **clear()**. Then, the **gl** coordinates are translated with the **glTranslate** member function so that the **gl** screen x and y coordinates are in the interval [-1,1]. The **glBegin** function with the parameter **GL_LINE_STRIP** prepares drawing a line strip. The vertices specified by the **glVertex** function between **glBegin** and **glEnd** will consequently be connected with lines. The line width was earlier specified in **gl.shader**.

[1] Recall that negative constants cannot be specified. Consequently, **MAP_FLOW** is preceded by a minus sign.

The vertices of the waveform are drawn inside the loop following **glBegin**. First, the x coordinate in the interval [-1,1] corresponding to the loop index **i** in the interval [0,511] is calculated. Then the next vertex is drawn to a point specified by **x** and the waveform value in the **i**th **waveLeft** and **waveRight** array cells. After the loop, the line strip is finished by calling **glEnd**.

3.2 Using Textures

To illustrate how textures can be loaded, we will change the scene to draw the waveform out of textured particles. The modified scene can be found in the file **Caribbean Sea Example with Textures.r4** in the tutorial subfolder **Caribbean Sea**.

Firstly, the module list in the header must be changed to

```
SOLID black();
TEXTURE tex();
BUFLOAD bl();
MAP map(black, bl);
GL gl(map, tex);
BUFSAVE bs(gl);
```

A texture module **tex** is added in order to load a texture file. **tex** is given to **gl** as input module so that **gl** can use it for drawing.

The constant

```
// Ring texture size
const float RING_SIZE = 0.05;
```

is added to the constant definition section. **RING_SIZE** specifies the size of the textured particles.

In the **init()** function, the initialization of gl shader is changed to

```
strcpy(gl.shader,"T0;B11;");
```

This makes the texture to be drawn additively. Also the lines

```
// Set texture file name
strcpy(tex.filename,"particle_ring.png");
```

specifying the texture file name are added to the **init()** function.

The **render()** function using textures is listed below.

```
void render() {
// Set load buffer handle
bl.handle = bs.handle;
// Clear the gl command queue
gl.clear();
// Set particle scale + make them face the viewer
gl.glParticleOrient(RING_SIZE);
// Translate so that x and y are in the interval [-1,1]
gl.glTranslate(0, 0, -2.414);
// Begin drawing lines
gl.glBegin(GL_QUADS);

for (i = 0; i < 512; i = i + 16) {
    // Calculate the x coordinate that corresponds to i
    x = 2 * i / 511.0 - 1;
    // Draw next point in the line strip
  gl.glParticle(x,-WAVE_SCALE*(waveLeft[i]+waveRight[i]),0,1);
}
// Finished drawing lines
gl.glEnd();
}
```

The particle size is set by the **`glParticleOrient`** function and instead of **`GL_LINE_STRIP`**, the glBegin function uses **`GL_QUADS`**. This is due to the fact that texture particles consist of four points specifying the corners of the rectangular particles.

In the **`for`** loop, the loop index **`i`** is increased by steps of 16 at a time, as it is not reasonable to draw 512 rings. The texture particles are drawn with the **`glParticle`** function.

In general, texture dimensions should be of type 2^n, where n is an integer. Also textures of arbitrary dimension can be used, but with decreased performance.

3.2.1 **Modifying the Scene**

Feel free to modify the scene file. You can modify the constant values or shader string, or add some new features to see how the scene works. You might want to see the **gl.r4** file in **R4\dll** to see other available **gl** commands. If you want to draw textured objects with **`glVertex`**, you have to use **`glTexCoord`** to separately specify the texture coordinates before every **`glVertex`** call.

3.3 Further Reading

When you are working with a new module, the first thing to do is to open the corresponding definition file found in the **R4\dll** directory. The definition file lists all the variables of the module and usually also contain some comments on what the variables do. Also, included to the default R4 scenes are several scenes named **simple_<module>.r4** where **<module>** is a module name. These files illustrate how the corresponding modules are used. You might find it interesting to look at and modify the files to familiarize with the modules.

The **`GL`** module is one of the most useful ones. Basically, almost everything that cannot be done with other modules can be done with **`GL`**. Knowledge in OpenGL or other 3D programming interface is an advantage when using **`GL`**. See **gl.r4** and the OpenGL documentation for further information.

In Appendix I you can find some common compile errors and possible causes for them. Hopefully it will help you if you cannot find the cause of an error.

If you are looking for examples of how to use a certain module or parameter, you can use the *Find in Files…* function in Crimson Editor. Select *Find in Files…* in the *Search* menu and type the name of the module or parameter to the *Find what* field. Write the folder name where you have the default custom scene files to the *Folder* field (assumingly **C:\Program Files\R4\data\predefine\disable**). If you are looking for a certain module, you can check the *Match case* box and search with an upper case module name to only search for module definitions.

You can find interesting information and discussions at the official R4 forum at **http://www.rabidhamster.org/phpBB2/**. You can also find the scenes written by other users. If you make a nice scene, please contribute to the community by sharing it at the forum!

Appendix I – Compile errors

When an error is caught in a scene file, you will get a popup window describing the error. The error type is printed after the text "Exception Caught : ", followed by the path and name of the file that caused the error and two numbers separated by a colon ':'. The row in the scene file where the error occurred is given by the second number. Some of the compile errors are listed below with a short explanation.

Syntax error – wanted <floating point> – You have defined a floating-point constant but you have not defined any decimals. Define at least one decimal for the constant.

ID not found – You have probably tried to use a variable but no variable with the name has been defined. You might have typed a wrong variable name or by mistake removed the definition.

Cannot convert types (to/from) "float" "void"

Cannot convert types (to/from) "int" "void" – You assumingly are trying to insert the return value of a void function into a variable.

Rendering Error, current = <scene file path>, next = null, fade = <fade path> – You have assumingly not specified sufficiently many input modules for some module. Include more input modules.

Syntax Error – wanted [':' Character] – a parameter might be missing in a function call.

Appendix II – Version History

Version	Changes
v1.0	• Added the version history appendix. • Corrected an error in overlay section - header must be defined as **`OVERLAY`** instead of **`SCENE`**. • Added a comment about division with a constant. • Added some comments about the **dll** directory to further reading section.