FunkGL

Language Reference Manual

John Gallagher <jmg2016@columbia.edu>
Mack Lu <yl2194@columbia.edu>
Oren Sivan <os2109@columbia.edu>
Christos Savvopoulos <cs2467@columbia.edu>

Introduction

FunkGL is a language designed for the purpose of 3D graphics manipulation. The language offers a simple interface to specify an environment, create objects within the environment, and algorithmically manipulate the objects. It is important to remember that FunkGL operates within the "update-render cycle" paradigm of real-time graphics. The update part of the cycle determines what should be drawn and how, and the render part of the cycle actually draws the objects onto the screen. FunkGL creates a large state machine that keeps track of the state of the environment and all objects. The state of each object is updated through functional programming and rendered by OpenGL libraries every cycle. This simple paradigm allows us to harness the power of OpenGL in a flexible way to create 3D graphics.

Lexical Conventions

A program consists of one or more series of definitions. There are two types of definitions: *Declarative* and *Functional*. The former specifies 3D objects and their attributes, whereas the latter specifies functions used to update the attributes of objects.

Tokens

There are six classes of tokens: identifiers, keywords, floating constants, operators, string constants and other separators. White space is only meaningful in the context of separating otherwise adjacent tokens and is otherwise ignored.

Comments

The characters "/*" introduce a multi-line comment, which terminates with "*/". Multi-line comments do not nest. The characters "//" introduce a single-line comment, which terminates with the next new line character. LF and CR by themselves represent a single new line whereas CRLF represents two new lines.

Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter. Upper and lower case letters are treated as different letters. Underscores are considered letters.

Keywords

The following identifiers are reserved for keywords, and may not be used otherwise:

`mesh`	`material`
`light`	`camera`
`float`	`list`
`vector`	`print`
`time`	`file`
`mouse`	`keyboard`

Constants

There are four types of constants: floating,vector, list, and string constant.

<constant>:
    <floating_constant> | <vector_constant> | <list_constant> | <string_constant>

Floating Constants

Floating constants are the computer equivalent of scientific notation and serve as atomic elements of lists and vectors. The exact number of significant figures and the range of possible exponents are machine-specific; FunkGL always uses the C data type `double` to store floating-point numbers. Floating-point constants are defined in a very similar way to C, that is:

> "A floating constant consists of an integer part, a decimal part, a fraction part, an `e` or `E`, an optionally signed integer exponent"… "the integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing." – The C Programming Language

Vector Constants

A vector constant is a structure of three floating-point constants, representing magnitudes in the spatial x, y and z dimensions. Vectors are used to represent many object attributes, such as position, velocity, or even color. A vector is strictly of the form "(x,y,z)" where x,y,z represent three floating-point constants.

Lists Constants

Lists are implemented as singly linked lists to allow for maximum flexibility. Linked lists are simple to modify and iterate through. A list may contain a floating-point constant, a vector constant or another list constant as an element. An empty list is declared as “List <identifier> = []”.

String Constant

A string constant is a sequence of characters surrounded by double quotes. String constants are only used to specify the pathname (relative to the directory of the executable program) of the file that represents an object. As such, they only appear in *declarative* definitions. Specifically, they always appear in the following format:

string file=<*string constant*>;

Basic Data Types

<data_type>:

float | vector | list | string

There are four data types: floating-point numbers, vectors, lists, and strings whose data type labels are 'float', 'vector', 'list', and 'string' respectively. A data type label indicates that the data representation of the memory held by the identifier should be interpreted as the type specified when placed in front of an attribute or function parameter. It indicates that the data representation of the result of a function should be interpreted as the type specified when placed in front of a function definition.

Objects

Objects are the basic building blocks of rendered scenes. Each object has attributes describing various characteristics of the object. Objects are different from data types because they cannot be used as arguments. An important distinction between objects in FunkGL and objects in traditional Object-Oriented languages such as Java is that FunkGL objects are statically created based on declarative definitions. Thus, they cannot be dynamically created with functional definitions. There are five types of objects: meshes, materials, lights, cameras, and prints. See section below on declarative definitions for more information on specific objects.

Definitions

Declarative Definitions

Declarative definitions are used to define objects and their attributes. An attribute may be linked to a functional definition through the assignment symbol =. The function that the attribute is attached to updates the value of the attribute on every update cycle. The following are templates for the declarative definitions of all five object types. Note that each object type has some special attributes that, when declared, have unique and predefined consequences on the rendering of that object (such as the `position` of certain objects). If a special attribute is not declared, OpenGL’s default values will be used. All other attributes are extra attributes and are used exclusively to that particular object.

Mesh Objects

A mesh is a collection of triangles. Special attributes: `file` (string constant), `position` (vector), `rotation` (vector), `scale` (vector), and `material` (identifier for a material object).

```
mesh:<name>
{
        string <file>=<path>;
        <data_type> <attribute> = <expression>;
        <data type>  ~<attribute> = <constant>;
        material = <material identifier>

}
```

Where:

- <name> is a unique identifier not used by any other object
- <path> is a string constant representing pathname to file describing mesh
- <attribute> is an identifier. To use an actual attribute, use the reserved identifier for it
- '~' is used to denote an initial value statement for an attribute that has an initial value returned by <constant>
- Note that one can declare as many <attributes> in the way described above as desired
- <material identifier> is an optional identifier for a material object describing the mesh

Material Objects

A material is used as an attribute for a mesh and specifies the mesh's appearance. Special attributes: `ambient` (vector), `diffuse` (vector), `specular` (vector), and `shininess` (float).

```
material:<name>
{
       string <file>=<path>;
       <data_type> <attribute> = <expression>;
      <data type> ~<attribute> = <constant>;
}
```

Where:

- <name> is a unique identifier not used by any other object
- <path> is a string constant representing pathname to file describing material
- <attribute> is an identifier. To use an actual attribute, use the reserved identifier for it
- '~' is used to denote an initial value statement for an attribute that has an initial value returned by <constant>
- Note that one can declare as many <attributes> in the way described above as desired

Light Objects

A light is necessary for meshes to appear three-dimensional. Without lighting, and the gradients light creates, it is impossible for humans to perceive depth. Special attributes: `ambient` (vector), `diffuse` (vector), `specular` (vector), `position` (vector), `attenuation` (vector), and `w` (float).

```
light:<name>
{
        <data_type> <attribute> = <expression>;
        <data_type> ~<attribute> = <constant>;
}
```

Where:

- <name> is a unique identifier not used by any other object
- <attribute> is an identifier. To use an actual attribute, use the reserved identifier for it
- '~' is used to denote an initial value statement for an attribute that has an initial value returned by <constant>

Camera Objects

A camera is defined once. It allows the user to move through the scene. Special attributes: `position` (vector), `direction` (vector), and `up` (vector).

```
camera:<name>
{
        <data_type> <attribute> = <expression>;
        <data_type>  ~<attribute> = <constant>;
}
```

Where:

- <name> is a unique identifier not used by any other object
- <attribute> is an identifier. To use an actual attribute, use the reserved identifier for it
- '~' is used to denote an initial value statement for an attribute that has an initial value returned by <constant>

Print Objects

A print is a special kind of object used by the programmer to print attribute values to the command-line.

```
print:<name>
{
        <data_type> <attribute> = <expression>;
        <data_type>  ~<attribute> = <constant>;
```

}

Where:

- <name> is a unique identifier not used by any other object
- <attribute> is an identifier
- '~' is used to denote an initial value statement for an attribute that has an initial value returned by <constant>

Implicit Global Objects

Implicit global objects allow for user to probe relevant information about the environment in which they run. The `time,mouse,and keyboard` objects each have attributes that can be inspected just like the attributes are any other object. The the objects themselves and their attributes' update functions are predefined. Were they to be written explicitly, they would have this appearance.

```
global:time
{
	float dt = /*the actual elapsed time between this frame and the last*/
}

global:mouse
{
	float x = /* origin-relative x position of the mouse pointer */
	float y = /* origin-relative y position of the mouse pointer */
	float LeftUp = /* left mouse button released on this frame (0|1)*/
	float LeftPressed =/* left mouse button depressed during this frame (0|1)*/
	float LeftDown = /* left mouse button depressed on this frame (0|1)*/
	float RightUp = /* right mouse button released on this frame (0|1)*/
	float RightPressed=/*right mouse button depressed during this frame (0|1)*/
	float RightDown = /* right mouse button depressed on this frame (0|1)*/
}

global:keyboard
{
	float ?Up = /* '?' key released on this frame (0|1) */
	float ?Pressed = /*'?' key depressed during this frame (0|1)*/
	float ?Down = /* '?' key button depressed on this frame (0|1)*/
	/*where ? is any physical key on a standard US qwerty keyboard*/
}
```

Examples of use:

```
vector updatepos(vector pos, vector vel):pos*vel*time.dt;
vector updatepos(vector pos):(keboard.LeftCtrlDown & keyboard.SpaceDown)?2*pos:pos;
```

Functional Definitions

Functional definitions define the operations that update the attributes of each object. As the name suggests, these definitions are purely functional. Statements cannot be sequenced. Functions are evaluated concurrently based on the current state, and changes will only be manifest after the next update-render cycle.

Functional definitions are of the form:

<function_declaration> :

```
        <data_type> <function_name>(<data_type> <parameter1>, <data_type>
<parameter2>, ...) ':' <expression>;
```

Where type specifies the return type, function specifies the function name, and parameter specifies the parameter name. Expression is any expression that evaluates to a data type consistent with the return type of the function and will be the return value of the function.

Expressions

Expressions are constructed of floating point values, vectors, lists, or function calls, combined using conditional or mathematical operators which ultimately return the data type specified by the function definition which precedes it. Expressions types are listed here in order of precedence (greatest to least).

Floats, vectors, and lists have already been discussed above under constants.

```
<expression>:
        <conditional_expr>
<conditional_expr>:
        <logical_or><_expr> ? <expression> ':' <conditional_expr> | <logical_or_expr>
<logical_or_expr>:
        <logical_or><_expr> '|' <logical_and_expr> | <logical_and_expr>
<logical_and_expr>:
        <logical_and_expr> & <relational_expr> | <relational_expr>
<relational_expr>:
        <relational_expr> '<' < additive_expr> | <relational_expr> '<=' <additive_expr> |
        <relational_expr> == <additive_expr> | <relational_expr> '>=' <additive_expr> |
        <relational_expr> '>' <additive_expr> | <relational_expr> != <additive_expr> | <additive_expr>
<additive_expr>:
        <additive_expr> + <multiplicative_expr>| <additive_expr> - <multiplicative_expr>|
        <multiplicative_expr>
<multiplicative_expr>:
        <multiplicative_expr> * <unary_expr>|<multiplicative_expr> / <unary_expr>|<unary_expr>
<unary_expr>:
        !<multiplicative_expr>|<postfix_expr>
<postfix_expr>:
        <postfix_expr>.identifier | <postfix_expr>(<argument_expr_list>) | <primary_expr>
<argument_expr_list>:
        <expression>, <argument_expr_list> | <expression>
<primary_expr>:
        <identifier> | <constant> | (<expression>)
```

Operators

Our language provides several operators for mathematical, logical, and conditional operations. Their specific function depends on the data types on which they operate.

Access

Operators: .

The dot operator is used to either access an attribute of a specific object (“objectname.attributename”), to access an element of a vector (“vectorname.x” or “vectorname.y” or “vectorname.z”), or to access the first element of a list (“listname.first”) or the sub-list following the first element (“listname.rest”).

Function Calls

Operators: function_name(argument_expression_list)

Calls to functions are made by naming naming the function to be called, and passing an expression for each of the defined parameters of the function whose evaluation matches the data type specified for that parameter. For simplicity, evaluation is always strict. That is, f(g(x)) will always substitute the result of g(x) before evaluating f. This means that expressions passed as parameters must each be fully evaluated to a floating point number, a constant vector, or constant list before the function is actually called. A function call is a postfix expression, whose arguments are specified as an argument expression list.

Logical Inversion

Operators: !

Unary operator, written before a floating point number, that returns 1.0 if the floating point number is 0.0, and returns 0.0 otherwise.

Multiplicative Operators

Operators: * /

The * and / operators are grouped from left-to-right. A vector multiplied by a vector results in a floating-point number representing the dot product. If a vector is multiplied or divided by a scalar, the vector is element-wise multiplied by the scalar or its inverse, respectively. Vector-to-vector division, as well as any mathematical operation on lists, is illegal.

Additive Operators

Operators: + -

The + and – operators are grouped from left-to-right. Addition and subtraction between floating numbers and between floating point numbers and vectors follow the standard rules of mathematics and are used in infix notation. That is, floating point values can be added and subtracted as expected. Vectors can be added or subtracted with other vectors according to vector addition rules. Mathematical operations can be grouped using parentheses.

Relational Operators

Operators: `< <= > >= == !=`

These return 1.0 on evaluating true and 0.0 otherwise and are used in infix notation. Floating point numbers can be compared as expected. Vectors and lists can be element-wise compared for equality and inequality. Due to the imprecision associated with floating-point numbers, care should be taken when testing the equality of two floating-point numbers.

Logical Operators

Operators: `& |`

These return 1.0 on evaluating true and 0.0 otherwise and are used in infix notation. The OR '|' operator returns 1.0 if either of its operands are unequal to 0.0. The AND '&' operator returns 1.0 if both of its operands are unequal to 0.0. AND takes precedence over OR.

Conditional

Operators: `<expression 1>? <expression 2> : <expression 3>`

The first expression is evaluated, including all side effects. If the result is true (1.0), the result is the value of the second expression, otherwise that of the third expression. Only one of the second and third operands is evaluated. Note that because there are only floating-point numbers in FunkGL, Boolean expressions should only be produced by either logical operators or built-in functions to ensure correct/exact “true” and “false” values.

Operator Precedence

These are the precedence levels. They read up down from left to right with & having the highest precedence and – having the lowest. If the precedence of two operators is the same, the left-most one is evaluated first. This is a translated version of the expressions CFG above.

Operator	**Description**	**Association**
() .	Parentheses for nested arithmetic expressions accessor Selection of inner element	left-to-right
!	Logical inversion	N/A
* /	Multiplication/division	left-to-right
+ -	Addition/subtraction	left-to-right
< <= > >=	Logical less than/less than or equal to Logical greater than/greater than or equal to	left-to-right
== !=	Logical is equal to/is not equal to	left-to-right
&	Logical AND	left-to-right
|	Logical OR	left-to-right
?:	Conditional	right-to-left
=	Assignment	right-to-left

Built-in Functions

The following built-in functions provide some extra-lingual abilities.

List Functions

append(list1, list2) – Appends list2 to the end of list1.
cons(element, list) – Adds element to front of list

Predicate Functions

These functions test the type of a variable and return true (1.0) or false (0.0):
islist(L) – test for list
isvec(V) – test for vector
isnum(F) – test for floating-point number

Control Functions

exit() - breaks out of the otherwise infinite update-render loop that the program runs in.