FunkGL

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.

FunkGL follows the "update-render cycle" model 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. In designing our language, we aimed to abstract away redundancies and similarities in updating and rendering. 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.

Motivation

Our main motivation for creating FunkGL is to be able to simplify basic 3D object manipulation and interactions, in the process reducing the amount of code for rendering procedures. Programming in OpenGL requires a lot of janitorial work to setup and manipulate the objects and the camera that views them into scene that the user can understand. FunkGL will take care of unnecessary configurations and let the code go straight to work.

Rendering objects is also cumbersome. To draw a blue triangle, you would have to call three functions: one to change the current position of the desired triangle object, one to change the color attribute of the triangle object to blue, and one to render the triangle. The triangle is the basic working unit of OpenGL but for complex objects, it is hard to work with and error prone.

Program Structure

Since real-time graphics require constant rendering, every FunkGL program will have a basic update-render loop. Thus, the underlying structure of every FunkGL program will resemble:

```
init();
while(active)
{
  update();
  render();
}
```

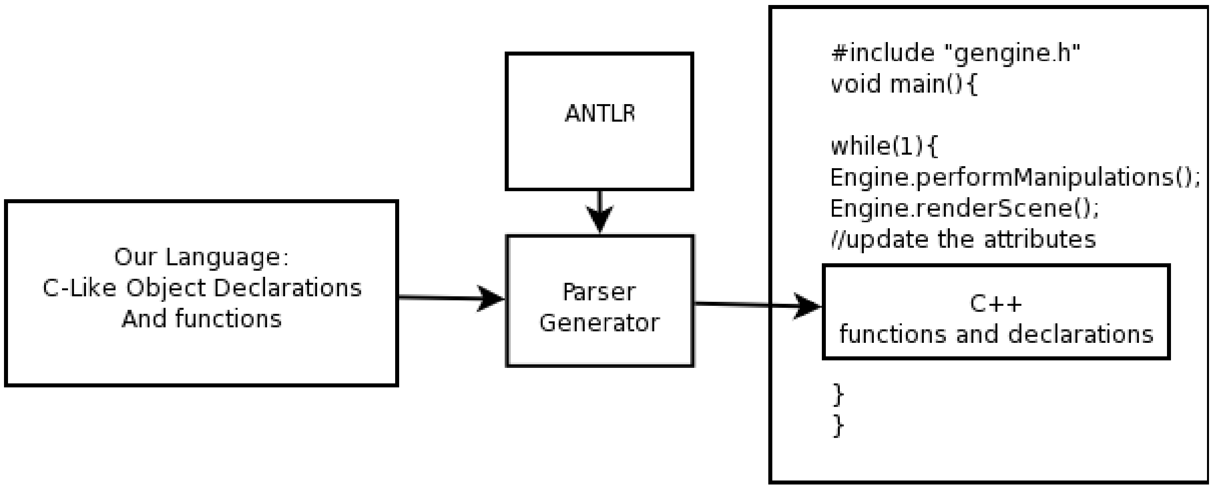
The init() function will be responsible for initially enabling OpenGL and loading all of the declared objects into a pre-existing OpenGL engine.

The update() function executes this sort of sequence every time:

```
set the camera
set the lights
loop through all objects
  calculate new state of object
  modify object to reflect new state
```

The render() function will wrap OpenGL rendering functions to render the objects onto the screen.

Diagram 1: Compilation and execution flow



Language Structure

Programs written in our language will consist of a series of definitions that describe the objects comprising a 3D scene and how they move. There are two types of definitions: *Declarative* and *Functional*.

Declarative Definition

Declarative definitions specify and initialize objects. An object is either a 3D mesh, a light, a camera, or a material. Objects have various attributes. In the case of a mesh for example, we have position. Each function can be attached to an attribute. Then, in every cycle through the main loop, the value of this attribute gets updated based on the value returned from the function attached to it. This provides an efficient way to update the state of an object.

The following is a template for a declarative definition:

```
object_type:name
{
        file="path";
        data_type attribute=function(parameter1, parameter2, ...);
        data_type attribute=initial_value;
}
```

Where:

- Object is one of the words: mesh, material, light, camera, print.
- Name is an identifier
- File is a special attribute that is mandatory for some objects
- Type is one of: float, vector, list
- Attribute is an identifier
- Function is an identifier of a function defined somewhere in the code
- Parameter is an attribute from any object, the format is: name.attribute (where name is the name of an object - can be omitted if referring to the declared object).

Functional definitions

For each attribute we can define a function, or an initial value, or both. When we omit the initial value, the compiler assumes it's 0, (0,0,0), or [], depending on the type. While the attribute name can be arbitrary, some of them have a special meaning, e.g. position. The ones that don't have any predefined meaning, are for internal use, e.g. velocity(arbitrary) versus position(predefined).

Functional definitions define the operations to 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 will only affect the *next* state.

Functional definitions are of the form:

```
type function(data_type parameter1, data_type parameter2, ...):expression;
```

Where type, function, and parameter are defined as above, and expression is:

- floating point number
- vector: (float,float,float) where float can either be a constant float or an expression that evaluates to one
- lists: [expression, expression, ...]
- function call: function(expression, expression, ...) where the expressions have to match the types of the function
  conditional: Conditionals are identical to the shortened ones in C: condition?iftrue:iffalse (condition, iftrue and iffalse are expressions)
- operations: syntax: expression operator expression (again, exactly like C)
- variable: can either be one of the parameters to a function or a global variable, like dt (delta of time)

- dot(.) operator: can be used to refer to specific parts of something, either to access the attribute of any object (object.attribute), or to access certain attributes of data types (e.g vector.x or list.rest)

Misc
Note that we don't have any looping instructions. This is done using recursive function calls, which our language allows. C++ style comments are available. /* */ and / /. We consider both LF and CR a new line. CRLF for simplicity is *two* newlines.

Data Types

Floating Point Numbers
Floating point numbers are the atomic elements of lists and vectors.

Vectors
A vector is a structure of three floating point numbers, representing magnitudes in the spatial x, y and z dimensions. Though vectors can be thought of as specialized or limited lists, the domain area of our language provides sufficient motivation to make the manipulation of vectors, particularly through mathematical operators, as easy and natural as possible. Vectors are used to represent many object attributes, such as position, velocity, or even color.

Lists
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 number, a vector or another list as an element. Another incentive for using lists is that there does not appear to be a very strong need for random access in the domain of our language. The one exception is vector components, and this case has been handled by providing a vector data type.

This language is strongly typed to enhance the clarity and readability of code. All of the basic mathematical and conditional operations are "overloaded" to do the correct thing given particular types.

Objects
Objects are not strictly data types in that they cannot be used in their entirety in calculations. An important distinction between objects in FunkGL and traditional objects in Java is that FunkGL objects are statically created based on declarative definitions. They *cannot* be dynamically created with functional definitions.

Types of objects:

- mesh - a collection of triangles. Reserved attributes: file, position, rotation, scale, and material.
- material - used as attributes to meshes and they specify their appearance. For example, a mesh can have a very specular appearance, it can be green or it can even have a texture.
- light - necessary for meshes to appear three dimensional. Without lighting, and the gradients it

creates, it is impossible for humans to conceive depth. Reserved attributes: ambient, diffuse, specular, position.

- camera - defined once. It allows the user to move through the scene. Reserved attributes: position, direction, and up.
- print - special kind of object. It is useful to the programmer, as it can evaluate functions and print the result on the terminal.

Globals

We have a set of global variables that help make the products of our language interactive. They are all reserved keywords. One of them is dt. dt is the time difference between the start of the current cycle (i.e. before update) and the start of the previous one.

We also have a set of global variables that allow the programmer to probe keyboard and mouse. Their format is:

- mouse{Left, Right}{Up, Down, Pressed}
- mouse{X, Y}
- keyName{Up, Down, Pressed}

Note that the "Up" and "Down" variables stay true only for the frame in which the event occurred (i.e. the frame in which the user released a button).

Operators

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

Mathematical

Operators: `+ - * /`
Operations between floating numbers and between floats and vectors follow the standard rules of mathematics. That is, floating point values can be added, subtracted, multiplied, and divided as expected. Vectors can be added or subtracted with other vectors according to vector addition rules. They can also be multiplied with other vectors, and the result is a floating point number, the dot product. If multiplied or divided by a scalar, the vector is element-wise multiplied by the scalar or its inverse, respectively. Vector-to-vector division, and any operation on lists is illegal.

Comparison

Operators: `< <= > >= == !=`
Floating point numbers can be compared as expected. Vectors and lists can be element-wise compared for equality and inequality. Note that the == and != operators should be used very carefully due to the imprecision of floats. The result of a comparison is 0.0 on false and 1.0 on true.

Logical

Operators: `& |`

These return 1.0 on true and 0.0 otherwise. The logical operators should be used only with comparisons, due to the imprecision of floats. These values are false: the floating point number zero, the zero vector (0,0,0) and the empty list [].

Precedence
These are classes of precedence. If the precedence of two operators is the same, the left-most one comes first.

- & |
- < >   = > >= == !=
- * /
- + -

Functions

Our language comes with a few built-in functions which provide some extra-lingual abilities, that is, things you could not program yourself in this language. These built-in functions are coded in the underlying c++ in to which this language is translated.

Lists:
append(list1, list2)
cons(element, list)

Predicates:
These functions are useful for knowing the type:
islist(L)
isvec(v)
isnum(f)

Control functions
exit() breaks out of the otherwise infinite update-render while loop in which the compiled code runs.

Syntactic Example

The following code is a snippet of code from our language. The first two definitions are declarative and describe all the objects (a sphere and a plane) that will exist in the graphics engine. The third definition is functional and describes the function that determines the velocity of the sphere. It basically describes the velocity of a ball subject to gravity and bouncing on a floor with perfectly elastic collisions. The final definition is also functional and describes the function that determines the position of this ball.

```
mesh:sphere
{
        file="data/sphere.obj";
        vector position=bounce_pos(position,vel);
        vector position=(0,0,0);
        vector vel=bounce_vel(position,vel);
}

mesh:plane
{
        file="data/plane.obj";
        vector position=(0,0,0);
}

vector bounce_vel(vector pos, vector vel):
        vel.y<0 && pos.y<0?
                (0,0,0)-vel+g*dt
                :vel+g*dt;

vector bounce_pos(vector pos, vector vel):
        pos+vel*dt;
```