ELMO Language Reference Manual

Jeffrey Cua
jmc2108@columbia.edu

Stephen Lee
sl2285@columbia.edu

Erik Peterson
edp2002@columbia.edu

John Waugh
jrw2005@columbia.edu

October 21, 2004

Chapter 1

Introduction

ELMO Loves Manipulating Objects (ELMO) is a language for building graphical 3D scenery. It is designed to be easy to read and requires very little previous programming experience. It should also appeal to individuals who understand basic linear algebra, especially vectors and affine transformations.

The ELMO language follows much of the style of C/C++/Java, but it removes (or abstracts) many of the lower level details, especially pointers and memory management in general.

1.1 Motivating Example

```
/*
 * OpenGL example
 * - scales vector v by lambda and stores results in vector u
 */
vec3 & mult(vec3 & u, const vec3& v, const nv_scalar& lambda)
{
    u.x = v.x*lambda;
    u.y = v.y*lambda;
    u.z = v.z*lambda;
    return u;
}

multi( u, v, lambda );

/* ELMO example */
u = v * lambda;
```

Chapter 2

Lexical Conventions

There are five kinds of tokens: identifiers, keywords, constants, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

2.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`. A single line that starts with optional whitespace followed by `//` also denotes a comment line.

2.2 Identifiers (Names)

An identifier is a sequence ofletters and digits; the first character must be alphabetic. The underscore counts as alphabetic. Upper and lowercase letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

```
PI
number, vector, group, object
rotate, move, scale, stamp, attach, remove
around along by degrees radians
if, else, for, switch, foreach, in, default, break, void
```

2.4 Constants

2.4.1 Vector Constants

A vector constant is a `<` character followed by a `>` with number constants or variables in between separated by commas. Vectors only support three dimensions. (e.g. `<0, 1.2, a>`)

mention something about "global" coordinate axes?

2.4.2 Numerical Constants

A number constant is a sequence of digits. The digits can be interrupted by one decimal only. These will be implicitly assigned and used just as a number variable. (e.g. `512`, `0`, `0.50`, `35.56`, or `PI`)

mention something about degrees/radians?

2.4.3 Object Constants

An object constant is the name of either a OBJ file or an ELMO. If a file does not exist, a run-time error will occur and the ELMO program that referenced it will halt.
`<args>` takes the same format as for user-defined functions

```
"tree.obj"
```

object constant loaded from OBJ file tree.obj

```
"myprogram.elmo"(foo=5 ,bar=3)
```

object constant loaded from ELMO program `myprogram.elmo` using given arguments

Chapter 3

Meaning of Identifiers

3.1 Types

- *number*
- *vector*
- *group*
- *object* (every object is also a group)

3.1.1 number Variables

A number variable is explicitly assigned by another number variable or a number constant. Numbers are treated as a float in Java unless it reaches a whole number at which time it will be stored as an int.

```
number i = 5;
```

`i` is a declared `number` variable
`5` is an implicit `number` constant

Implicit numbers can be given with random-number syntax as well

```
number i = [-1..3] //i is a random number between -1 and 3
number j = [4] //j is a number between 0 and 4
```

3.1.2 vector Variables

A vector variable is explicitly assigned by another vector variable or a vector constant. Vectors are simply a collection of three number variables locally associated as `x`, `y`, and `z`. These are accessed with vectorname.x, vectorname.y, vectorname.z respectively.

3.1.3 group Variables

Groups are variables that contain objects and can have transformations applied to them. A group is defined by three vectors (axis) `x`, `y`, and `z`. These are accessed with `groupname.x`, `groupname.y`, `groupname.z` respectively. All of the vectors are **READ ONLY**.

3.1.4 object Variables

An object variable is a data structure that represents a 3D object. These objects are either primitives or imported OBJ or ELMO files. Objects have their own implicit group that allows a user to directly apply transformations. An object has three attributes (axis) `x`, `y`, and `z`. These vectors are accessed with `objectname.x`, `objectname.y`, `objectname.z` respectively. All of the vectors are **READ ONLY**.

```
object o = "tree.obj"
```

`o` is assigned to the value of an object constant, `"tree.obj"`

Chapter 4

Expressions

4.1 Primary Expressions

primary-expression:
    identifier
    constant
    `(` *expression* `)`
expression:
    assignment-expression
    expression `,` *assignment-expression*

For the most part, the operators in ELMO work have the same precedence and associativity as those in the C Language Specification. However, ELMO does not have the exact same concept as pointers as C. Instead, ELMO uses references transparently, like Java. Moreover, ELMO lacks bitwise and shift operators.

4.2 Postfix Expressions

postfix-expression:
    primary-expression
    postfix-expression `(` (*argument-expression-list*)? `)`
    postfix-expression `.` *field*
    postfix-expression `++`
    postfix-expression `--`
argument-expression-list:
    assignment-expression
    argument-expression-list `,` *assignment-expression*
field:
    `x`

y
z

The only addition

4.3 Operators

4.3.1 ELMO Operators

vector-expression:
`move` *identifier* `along` *expression* (`by` *expression*)?
`rotate` *identifier* `around` *expression* (`by` *expression units-expression*)?
`scale` *identifier* `around` *expression* (`by` *expression*)?
`stamp` *expression*
`attach` *expression* `to` *identifier*
`remove` *identifier* `from` *identifier* (`inherit`)?
`clone` *primary-expression* `as` *identifier*

This are really the main operators in the ELMO language. None of these operators will return anything. They will solely operate on the contents of the first argument. The following are some examples of these operators in action.

```
move myobject along myobject.x by 3.2;     //ok - every object is also a group
move mygroup.x along <1,2,3>;              //ILLEGAL - myobject.x is READ-ONLY!
move myvector along <1,2,3>;               //ok
rotate mygroup around mygroup.x by PI rad;
rotate myvector myOtherVector 30;
scale tree 5;                              //make a big tree!
scale "tree.obj" by 3.2;                   //ILLEGAL - "tree.obj" is not a
                                           //declared group
stamp "mytree.obj";
stamp myTree;                              //output myTree wherever it happens
                                           //to be right now it will forever
                                           //have that visual info in the output
                                           //file even if we move myTree later.

remove myTree from myForest;               //might make myTree jump back to the
                                           //origin, if that's where it started

remove eyeball head inherit;               //keeps the eyeball where it used to
                                           //be, just not further affected by
                                           //head's transformations

clone thisVector as thatVector;
clone thisGroup as thatGroup;
```

4.3.2 Units

units-expression:
    `deg` (degrees)
    `rad` (radians)

4.3.3 Unary Operators

unary-expression:
    postfix-expression
    `++` *unary-expression*
    `--` *unary-expression*
    `|` *unary-expression* `|`

4.3.4 Multiplicative Operators

multiplicative-expression:
    unary-expression
    multiplicative-expression `*` *unary-expression*
    multiplicative-expression `/` *unary-expression*
    multiplicative-expression `%` *unary-expression*

4.3.5 Additive Operators

additive-expression:
    multiplicative-expression
    additive-expression `+` *multiplicative-expression*
    additive-expression `-` *multiplicative-expression*

4.3.6 Relational Operators

relational-expression:
    additive-expression
    relational-expression `<` *additive-expression*
    relational-expression `>` *additive-expression*
    relational-expression `<=` *additive-expression*
    relational-expression `>=` *additive-expression*

4.3.7 Equality Operators

equality-expression:
    relational-expression

equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

4.3.8 Assignment Expressions

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
`= =& *= /= += -=`

The basic `=` will create a copy of the right side and assign it to a variable named on the left side. The `=&` will create a reference of the right side and assign it to the variable named on the left side.

Chapter 5

Statements

A sequence of statements is executed one after another, resolving the current one before beginning the next. Each statement has an effect on the current data and/or generates a value but has no value itself. Statements can be subdivided into multiple categories.

statement:
expression-statement
compound-statement
selection-statement
iteration-statement

5.1 Expression Statement

expression-statement:
(*expression*)?

This generic statement type included assignments and function calls.

5.2 Compound Statement

compound-statement:
(*declaration-list statement-list*)?
declaration-list:
declaration
declaration-list declaration
statement-list:
statement
statement-list statement

Otherwise known as a block statement, a compound statement allows for the declaration of local variables and the execution of a sequence of statements.

5.3 Selection Statement

selection-statement:
`if (` *expression* `)` *statement*
`if (` *expression* `)` *statement* `else` *statement*

These statements allow for the conditional execution of a sequence of statements. The expression is first evaluated, and the output determines which statement is executed. In the top production, a result of true or non-zero result leads to the execution of the following statements. In the other production, a true or non-zero result leads to the execution of the immediately following sequence of statements whereas a false or zero result leads to the execution of the statement sequence following the "else" keyword.

5.4 Iterator Statement

iteration-statement:
`for (` (*expression*)? `;` (*expression*)? `;` (*expression*)? `)` *statement*
`foreach (` *identifier* `in` *identifier* `)` *statement*

These statements allow for the repeated execution of a sequence of statements. The first production takes three expressions which are executed once at the beginning, once before each loop, and once after each loop, respectively. The second expression's evaluation as true or false determines whether the statements are executed or the loop exited. The typical format is to initialize some sentinel value, test that it is within some range, and then increment or decrement its value for each iteration of the loop.

The second production serves as a short-cut for executing the same statements on each highest-level element in the specified group. "foreach" loops do not recurse to groups within groups. The current element is stored to the identifier specified. For recursive descent into nested groups, a "foreach" loop can be nested within another and executed using a selection statement to first test whether the element is a group or an individual object.

Chapter 6

External Declarations

6.1 Function Defintion

Function declarations are similar to those in C, but with thorough support of default variables (even more so than C++). Example:

```
vector myfunc(vector v1=<0,0,0>, vector v2=<1,1,1>)
{
    return v1 + v2
}
```

The above function, if called with no parameters, would return `<0,0,0> + <1,1,1>` (in other words, `<1,1,1>`) Example:

```
vector v = myfunc();    //v now equals <1,1,1>
```

The caller specifies arguments like so:

```
function_name([input1=value1] [,input2=value2] ... )
```

So for our example, we could have the following code segment:

```
vector v = myfunc(v1=<2,2,2>)       //leave v2 as its default <1,1,1>
                                    //- so v is <3,3,3> now
vector q = myfunc(v2=v)         //now q = <0,0,0> + v, which is just v
```

Note that since each input is referred to as a key=value pair, some C++ issues regarding default parameters do not arise. For example, in C++ if one wrote:

```
int myfunc(int foo=0, int bar=1)
{
    return foo+bar;
}
```

There would be no way for the caller to specify a value for bar but leave foo default

```
int q = myfunc(5);      //5 is assigned to foo -
                        //there is no way to make the 5 "bind" to bar
```

Of course, defaults are not required in ELMO. The following function is valid:

```
vector myfunc(vector v1=<0,0,0>, vector v2)
{
    return v1 - v2;
}
```

In such a case, v2 MUST be specified by the caller.

```
vector v = myfunc(v1=<2,2,2>) //ERROR - v2 not specified and has no default
```

Note also that, unlike with C++, default values can be assigned to any input, not just those at the end of the list of inputs.

One potential disadvantage to ELMO's scheme of defaults is that there could be some confusion with variable names. For example:

```
vector v1 = <9,9,9>
vector v = myfunc(v1=v1) //v = <9,9,9> + <1,1,1> = <10,10,10>
```

The `v1=v1` input to myfunc is not ambiguous, but it is perhaps not entirely intuitive. The name on the left of `=` is in the function's scope, while the name on the right is in the caller's scope.

Using `input=value`, the value, input will be a copy of value. This means that the original variable's value cannot be changed by the function. Note that while in a language such as Java, one can pass an Object by value, and then alter the contents of that Object using the "dot" syntax (internally dereferencing a pointer). This is not how ELMO behaves. If an a complex data type (such as an object or vector) is passed by value, the function can not alter the contents of the original input.

If by-reference functionality is desired, use the `=&` operator. example:

```
void myfunc(vector vec)     // note: no default value for vec -
                            // compiler will yell at you if
                            // you don't supply a value for vec
{
    vec = <1,1,1>
}
//elsewhere in the program...
vector v = <3,3,3>
myfunc(vec=&v)      //now v has value <1,1,1> since the original
                    //was altered by the function
```

The `=&` operator can be used for any type of inpus, such as numbers.