SIGL Language Reference Manual

Phong Pham　　Abelardo Gutierrez　　Alketa Aliaj

Programming Languages and Translators
Spring 2007

1 Introduction

Simple Image Generation Language (SIGL) is a simple drawing language based on Virtual Reality Modeling Language (VRML), widely used for specifying 3D models. It deploys the familiar image specification methodology and syntax of VRML while providing more control and flexibility with conditional branching and loop similar available in any other modern programming languages. Our designing goal for SIGL is to provide users with a simple, natural and yet flexible way to draw 2D images. This document acts as a complete reference manual for the language.

2 Lexicon

SIGL uses a standard grammar and character set. The specific elements that comprise this grammar and character set are described in the following sections:

- Character set (Section 2.1)
- Rules for identifiers (Section 2.2)
- Use of comments in a program (Section 2.3)
- Keywords (Section 2.4)
- Operators (Section 2.5)
- Interpretation of constant values (Section 2.6)

SIGL compiler treats source code as a stream of characters. These characters are grouped into tokens, which can be punctuators, operators, identifiers, keywords, or constants. Tokens are the smallest lexical element of the language. The compiler forms the longest token possible from a given string of characters; the token ends when white space is encountered, or when the next character could not possibly be part of the token.

White space can be a space character, new-line character, tab character, form-feed character, or vertical tab character. Comments are also considered white space. Section 2.1 lists all the white space characters. White space is used only as a token separator, but is otherwise ignored in the character stream, and is used mainly for human readability.

In order to simplify the complexity of implementing the language, we choose not to support strings in SIGL. Being a drawing language, this limitation is acceptable in most cases.

2.1 Character set

SIGL supports only a small subset of ASCII characters as follows:

- The 26 lowercase Roman characters
  `a b c d e f g h i j k l m n o p q r s t u v w x y z`
- The 26 uppercase Roman characters
  `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
- The 10 decimal digits
  `0 1 2 3 4 5 6 7 8 9`
- The 23 graphic characters
  `!   % & * ( ) - _ = + ; / | \ { } [ ] , .   < >`
- White spaces: space, horizontal tab, new line, carriage return

2.2 Identifiers

An *identifier* in SIGL is a sequence of characters consisting of one or more uppercase or lowercase alphabetic characters, the digits from 0 to 9, and the underscore character (_). Identifiers must not starts with a digit. SIGL identifiers are case-sensitive, i.e. test1, TEST1, Test1 are 3 different identifiers. In general, any sequence of characters satisfying the previous rules can be used as identifier, except for some reserved words known as *keywords*(see Section 2.4). An identifier represents a name for variables and functions.

2.3 Comments

SIGL supports 2 commonly used comment styles:

- Conventional C multi-line comments: the `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This syntax is the same as ANSI C. Comments of this type are not allowed to be nested.
- C++ single-line comments: the // (two slashes) characters, followed by any sequence of characters. A new line not terminates this form of comment.

2.4 Keywords

Keywords are special sequences of characters reserved by compiler, and are not allowed to be used as identifiers. Table 1 shows all the keywords used in SIGL

break	else	return	ellipse	rotate	color
continue	for	while	line	scale	
do	if	polygon	translate	fun	

Table 1: SIGL keywords

2.5 Operators

SIGL supports almost all unary, binary, and ternary operators in C, both arithmetic and logical operators. Summary of all operators, their precedences and associativity are shown in Table 2.

Precedence	Operator	Description	Examples	Associativity
1	()	Grouping operator	(a + b) / 4	left to right
	[]	Array access	a[i]	
2	!	Logical negation	if (!done)	right to left
3	*	Multiplication	i = 2 * 4	left to right
	/	Floating point division	i = 9 / 4	
		Integer division	i = 9 2	
	%	Modulo	i = 9 % 2	
4	+	Addition	i = 3 + 4	left to right
	-	Subtraction	i = 4 - 3	
5	<	Comparison less-than	if (i < 42)	left to right
	>	Comparison greater-than	if (i > 42)	
	<=	Comparison less-than-or-equal	if (i <= 42)	
	>=	Comparison greater-than-or-equal	if (i >= 42)	
6	<	Comparison equal-to	if (i == 42)	left to right
	<	Comparison not-equal-to	if (i != 42)	
7	&&	Logical AND	if (a && b)	left to right
8	||	Logical OR	if (a || b)	left to right

Table 2: Summary of SIGL operators

2.6 Constants

There are 2 types of constants in SIGL, integer constants (such as $63, 0, 42$), and floating-point constants (such as $1.2, 0.00, 77E + 2$).

Integer constants are used to represent whole numbers. To specify a decimal integer constant, use a sequence of decimal digits $0 \ldots 9$. Value of a decimal constant is computed in base 10. Integer constant values are always nonnegative; a preceding minus sign is interpreted as a unary operator, not as part of the constant.

A floating-point constant has a fractional or exponential part. Floating-point constants are always interpreted in decimal radix (base 10). Floating-point constants can be expressed with decimal point notation, signed exponent notation, or both. A decimal point without a preceding or following digit is not allowed (for example, .E1 is illegal).

The significand part of the floating-point constant (the whole number part, the decimal point, and the fractional part) may be followed by an exponent part, such as 32.45E2 . The exponent part (in the previous example, E2) indicates the power of 10 by which the significand part is to be scaled.

3 Declaration

3.1 Variables

SIGL does not require explicit variable declaration. A variable is declared when its value is assigned for the first time. SIGL deploys a dynamic type system, variable types are determined at runtime.

3.2 Functions

The syntax for declaring a function is as follows

```
fun function-name (parameter-list)
{
    statement*
}
```

The function will take parameter list as its parameters (this can be empty). The type of the function depends on the type of the return value yielded from executing function body. Recursive function is also supported. Functions are not allowed to be nested.

4 Program execution

Program execution is from top to bottom. Function definitions are not part of the execution. However, a function must be defined before it can be called.

5 Expressions

Primary expressions

5.0.1 identifier

An identifier is a primary expression.

constant

An integer or floating-point constant is a primary expression. Its type is `int` in the former case, and is `double` in the latter.

(expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

primary-expression [expression]

A primary expression followed by an expression in square brackets is a primary expression. This is used to refer to an element of an associative array. Its type is the type of the object stored in the associative array.

primary-expression (expression-list)

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in SIGL is strictly by value. A function may change the values of its formal parameters, but these changes cannot possibly affect the values of the actual parameters. Recursive calls to any function are permissible. The type of the function call is the type of its return value.

5.1 Unary operators

– expression

The result is the negative of the expression, and has the same type. The type of the expression must be int, or double.

!expression

The result of the logical negation operator ! is `true` if the value of the `expression` is false, and `false` if the value of the `expression` is true. The type of the result is `boolean`. Expression must also have type `boolean`.

5.2 Multiplicative operators

expression * expression

The binary * operator indicates multiplication. If both operands are int, the result is int; if one is double, the other is converted to double, and the result is double; if both are double, the result is double. No other combination is allowed.

expression / expression

The binary / operator indicates floating-point division. Both operands, if not already of type double, are converted to double. The result is double.

expression \ expression

The binary \ operator indicates integer division. Both operands must be of type int. The result is of type int.

expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int, and the result is int. In the current implementation, the remainder has the same sign as the dividend.

5.3 Additive operators

expression + expression

The result is the sum of the expressions. If both operands are int, the result is int. If both are double, the result is double. If one is int, it is converted to double and the result is double. No other type combinations are allowed.

expression - expression

The result is the difference of the operands. If both operands are int, or double, the same type considerations as for + apply.

5.4 Relational operators

expression < expression

expression > expression

expression <= expression

expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield false if the specified relation is not valid and true if it is satisfied. Both operands must be of boolean type.

5.5 Equality operators

expression == expression

expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence (Thus `a < b == c < d` is true whenever a < b and c < d have the same truth value). However, they can be used to compare any 2 objects of same type, both must be boolean, or both must be numeric. Comparison between int and double or double and double might yield unexpected result.

5.6 expression && expression

The && operator returns `true` if both its operands are `true`, `false` otherwise. The second operand is not evaluated if the first operand is `false`. Both operands must have boolean type.

5.7 expression || expression

The || operator returns `true` if either of its operands is `true`, and `false` otherwise. The second operand is not evaluated if the value of the first operand is `true`. Both operands must have boolean type.

5.8 Assignment operator lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. The type of the expression is the type of expression. The type of lvalue is also changed to type of expression after the assignment.

6 Statements

Except as indicated, statements are executed in sequence

6.1 Expression statement

Most statements are expression statements, which have the form

```
expression ;
```

Usually expression statements are assignments or function calls.

6.2 Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

```
compoundstatement:
    { statement* }
```

6.3 Conditional statement

The two forms of the conditional statement are

```
if ( expression ) statement
if ( expression ) statement else statement
```

In both cases the `expression` is evaluated and if it is true, the first substatement is executed. In the second case the second substatement is executed if the expression is false. As usual the `else` ambiguity is resolved by connecting an else with the last encountered elseless `if`.

6.4 While statement

The while statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains true. The test takes place before each execution of the statement.

6.5 For statement

The for statement has the form

```
for ( expression1 ; expression2 ; expression3 ) statement
```

This statement is equivalent to

```
expression1;
while (expression2) {
    statement
    expression3;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes false; the third expression typically specifies an incrementation which is performed after each iteration. Any or all of the expressions may be dropped. A missing `expression2` makes the implied while clause equivalent to `while(true)`; other missing expressions are simply dropped from the expansion above.

6.6 Break statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while` or `for` statement; control passes to the statement following the terminated statement.

6.7 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop continuation portion of the smallest enclosing `while`, or `for` statement; that is to the end of the loop.

6.8 Return statement

A function returns to its caller by means of the return statement, which has one of the forms

```
return ;
return ( expression ) ;
```

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. Flowing off the end of a function is equivalent to a return with no returned value.

6.9 Translate statement

Creates an translation environment

```
translate (expression1, expression2)
{
    statement*
}
```

all drawing statements in this environment would be executed under effect of the translation. The amount of translation is determined by `expression1` and `expression2`.

6.10 Rotation statement

Creates an rotation environment

```
rotate (expression)
{
    statement*
}
```

all drawing statements in this environment would be executed under effect of the rotation. The amount of rotation is determined by `expression`. The rotation center is at the origin.

6.11 Scale statement

Creates an rotation environment

```
scale (expression1 , expression2)
{
    statement*
}
```

all drawing statements in this environment would be executed under effect of scaling. The scaling factor is determined by `expression1` and `expression2`.

6.12 Color statement

Specify the color for the current drawing environment

```
color (r-expression , g-expression , b-expression);
```

Notice that the color is similar to variable, the effect of the color is only valid in the scope it is defined.

7 Scoping

A variable is declared when it is first used. The scope of a variable is the context within which it is defined. Any variable declared within a function is only local to that function. The scope of a parameter of a function definition begins at the start of the function block and persists through that function. The programmer can start a new scope any time they create a statement block with { }, such as in iterative statements or if/else statements. Moreover, identifiers declared within curly bracket are not accessible outside of the brackets.

SIGL is a static scope language. The environment of a function is the environment at the time it is defined, not the time it is called.

8 Sample SIGL programs

8.1 Example 1

```
/* Draw a simple image */
polygon{               // we would like to draw a polygon
    {0,0},             // list of vertices of the polygon
    {0,1},             // here the vertices will form unit square
    {1,1},
    {1,0}
}
translate(0.5,0.5){    // draw a circle inside the square
```

```
    circle(0.5);
}
// we'll draw the same square here
// except that this square would be rotated and translated
translate(2,0){         // translate it in horizontal direction by 2 units
    rotate(45){         // rotate it counter-clockwise by 45 degree
        polygon{        // the same drawing instructions as above
            {0,0},
            {0,1},
            {1,1},
            {1,0}
        }
        translate(0.5,0.5){
            circle(0.5);
        }
    }
}
```

8.2 Example 2

```
/* Draw many squares with increasing size and rotating angles */
fun square (size)
    scale(size){           // we use scale to get the size we want
        polygon{
            {0,0},
            {0,1},
            {1,1},
            {1,0}
        }
    }
}
for (i = 0;i < 36;i = i + 10)
{
    rotate(i * 10)
    {
        square(i);
    }
}
```