PhysEx

Language Reference Manual

Joshua Nuez (jn2548)
David Pu (sp3396)
Steven Ulahannan (su2206)
Justin Pugliese (jp3571)

October, 2016

Contents

1 Introduction

Physics engines are useful simulation tools used to observe behaviors of objects in physical systems. They facilitate discovery of patterns which can assist in making future predictions. Unfortunately, traditional programming languages are not optimized for these types of applications. PhysEx is a high-level programming language focused on easing the creation of Physics Engines. An implicit loop simulates the passage of time and applies a stimulus, essentially a function, to each object in the environment on each iteration. This document serves as a reference manual for the programming language.

2 Lexical Syntax

2.1 Identifiers

Identifiers are character sequences which combine together to describe a PhysEx program. They can be comprised of letters, digits, and the following special characters:

```
$ _ -
```

The first character of an identifier cannot be a digit or a dash.

PhysEx is a case-sensitive language, `foo` and `Foo` are unique identifiers.

2.2 Keywords

Keywords are identifiers with special meaning and cannot be declared for general use. The following identifiers that are reserved included (but are not limited to):

```
int char string function bool blob
return null
```

These keywords will be picked up by the parser and stored as tokens.

3 Primitive Data Types

PhysEx is statically typed and provides the following primitive types:

- `char` - An 8-bit signed data type used for storing ASCII characters and escape sequences.
- `string` - Used for storing `char` sets of arbitrary length. Values are wrapped in a pair of quotes.
- `int` - A 32-bit data type used for storing integer values.
- `float` - A single-precision floating point value.
- `bool` - A boolean value which can be indicated by the literals [0, 1, true, false].
- `blob` - Represents a physical object which can have forces applied to it

Primitive data types can be directly assigned by using the following template:

```
primitiveType variableName = value;
```

If the value assigned to a variable does not match the variable's assigned type an error will be thrown.

4 Arrays

Any array is a data structure which allows storage of one or more elements consecutively in memory. It can contain any type of elements but each element must be of the same type. Arrays are indexed at position zero.

4.1 Declaration and Instantiation

Arrays are declared by creating a variable with an element data type, its name, and setting it equal to a pair of brackets. For example:

```
int[] myArray = [];
```

Arrays are initialized at declaration and their length is dynamically allocated as elements are inserted and deleted. Optionally, arrays can be initialized with values by adding a list of comma separated values between the brackets. These elements must be the same type as array. For example:

```
int[] myArray = [1, 2, 3];
```

4.2 Array Access

Array elements can be accessed by specifying the array name, an open bracket, the index position, and a close bracket. The element will allow read and write capabilities. The following statement updates the initial element in the array initialized in the previous section to the value of 9.

```
myArray[0] = 9;
```

If a read event occurs on an uninitialized index `null` will be returned.

4.3 Sparse Arrays

Sparse arrays are not supported by PhysEx. When an index is written to the languages ensure all preceding indices have been initialized. If an index has not yet been initialized, it will be initialized to a 0 value.

5 Comments

Only single line, block comments are supported. Comments begin with a pair of forward slashes and end with the first newline character the scanner encounters, anything between will not be passed to the parser. For example:

```
// This sentence will not be passed read by the parser.
```

6 Operators

PhysEx supports most standard operators as well as some which aid in Physics centric calculations. The tables below are listed from highest precedence to the lowest and all operators are left-associative unless specified.

6.1 Arithmetic

Symbol	*Operator*	*Meaning*
()	Parenthesis	Overrides associativity of the other operators. Everything between them will have precedence over everything outside of them.
e	Exponential Notation	Multiplies the left operand by 10^n, where n is some integer.
^	Exponent	Multiplies the left operand by itself a number of times equal to the right operand's value. If that value is negative, then another step will occur, where it produces 1/(the new product).
*	Multiplication	Multiplies the two operands and produces the product.
%	Modulus	Performs division and produces the remainder.
/	Division	Divides the left operand by the right one, and produces the quotient.
+	Addition	Adds the operands before and after it and produces a sum.
-	Subtraction	Subtracts the left operand by the right and produces the difference.

6.2 Relational

Symbol	*Operator*	*Meaning*
!=	Not Equal	Returns true if lvalue and rvalue are not equal to each other. Otherwise, returns false.
==	Equal	Returns true if lvalue and rvalue are equal to each other. Otherwise, returns false.
<	Less than	Returns true if the left operand is less in value than the right operand. Otherwise, false.
<=	Less than or equal to	Returns true if the left operand is less in value than the right operand or equal in value. Otherwise, false.
>	Greater than	Returns true if the left operand is greater in value than the right operand. Otherwise, false.
>=	Great than or equal to	Returns true if the left operand is greater in value than the right operand or equal in value. Otherwise, false.

6.3 Logical

Symbol	*Operator*	*Meaning*
||	OR	If either of the two operands are non-zero, it returns true. Otherwise, false.
&&	AND	If both of the two operands are non-zero, it returns true. Otherwise, false.
!	NOT	Reverse the boolean value of the right operand.

6.4 Assignment

The following operators are right-associative.

Symbol	*Operator*	*Meaning*
=	Assignment	Takes the value of the right operand, and stores it into the left operands (assumes it is an identifier).
*=	Multi-signment	Multiplies the left operand with the right and assigns the product to the left operand.
/=	Divi-signment	Divides the left operand with the right and assigns the quotient to the left operand.
+=	Plus Assignment	Adds the left operand with the right and assigns the sum to the left operand.
-=	Sub Assignment	Subtracts the left operand with the right and assigns the difference to the left operand.

6.5 Overall Precedence

Category	*Operator*	*Associativity*
Postfix	() ++ --	Left to Right
Unary	!	Right to Left
Multiplicative	e ^ * / %	Left to Right
Additive	+ -	Left to Right
Relational	< <= > >=	Left to Right
Equality	== !=	Left to Right
Logical AND	&&	Left to Right
Logical OR	||	Left to Right
Assignment	= += -= *= /=	Right to Left

7 Statements and Blocks

Every statement must be terminated with a semicolon. Blocks of code (e.g. code that executes in part of a control flow) must be contained between a set of braces. It is possible to nest blocks within other blocks. The compiler will go through each line of code sequentially.

8 Control Flow

PhysEx supports `if` statements, `while` loops, and `for` loops. From a syntax perspective, all parentheses and braces are mandatory.

8.1 `if` Statements

An `if` statement is used to execute a particular block of code based on the result of a boolean condition. In the following example, if *condition* is a true statement then *if-block* will be executed and *else-block* will not be executed. In the case where *condition* is a false statement, the opposite code block will be executed.

```
if (condition) {
      if-block
} else {
      else-block
}
```

Additionally, `else if` statements can be used to add additional conditions to a decision making flow. If the initial *if* condition is false, the *else-if* condition is then evaluated and works similarly to an `if` statement. For example:

```
if (x == 2) {
      if-block
else if (x == 17) {
      else-if-block
} else {
      else-block
}
```

8.2 `while` Loops

The condition of `while` loop is evaluated prior to each block execution, if the condition is true the *while-block* is executed. After execution the condition is again tested and, if true, the *while-block* is executed again. This series of events is repeated until the condition is false and the application continues onto the next statement. Example syntax:

```
while (condition) {
      while-block
}
```

8.3 `for` Loops

The `for` loop is similar to the `while` loop in that the *for-block* is executed as long as the condition evaluates to true. The primary difference is that a variable can be initialized in prior to the first evaluation of the condition statement and the step expression is executed before each evaluation of the condition statement. Once the condition evaluates to false, the code precedes with the subsequent code block. Example syntax:

```
for (initialize; condition; step) {
      for-block
}
```

9 Functions

Functions separate code blocks into re-usable, distinct subprocedures.

9.1 Declaration

Functions are declared using the `function` keyword, the name, an open parenthesis, a comma separated list of parameters, and a closing parenthesis. The general form is show below:

```
function functionName (parameter1, parameter2, …, parameterN)
```

PhysEx does not require a return type to explicitly set, all functions return `null` or a value specified in the definition. Similarly, the language does not require all parameters to have values provided, they are all optional. Any parameter defined in the declaration

but not provided a value when called will be `null`. An example declaration is shown below:

```
function fooBar (x, y);
```

9.2 Definition

The definition section is the blocks of code which are executed when the function is called. On each execution the parameters are updated to match the values passed along with the function call.

```
function fooBar (x, y) {
      code-block
}
```

In the above example, the braces are required in order to correctly define the contained code.

10 Stimulus

Stimulus' are similar to a global function except that they are called directly by the program during runtime at regular intervals instead of directly from the application code.

10.1 Declaration

Declaring a stimulus follows the same conventions as functions from the prior section. The general form is as shown below:

```
stimulus stimulusName (optDelay, [optBlobs]);
```

There are two optional parameters which can be provided during declaration. The first parameter is a time offset which tells the application when to begin applying the stimulus. The second optional parameter is a list of blobs the stimulus should affect. If not provided, the stimulus will be applied to all blobs declared globally.

11 Parser and Scanner

Parser:

```
%{ open Ast %}

%token LBR LPR RBR RPR SEMICOLON
%token PLUS MINUS TIMES DIVIDE EOF ASN COMMA
%token MULTASN DIVASN PLUSASN SUBASN NE EQ LT LTE GT GTE
%token OR AND NOT NEG IF ELSE ELIF WHILE FOR FUNC STIM INT CHAR
%token STR FLT BOOL BLOB

% <int> LITERAL
% <int> VARIABLE

%right ASN
%left OR
%left AND
%left EQ NEQ
%left LT GT LTE GTE
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG

%start expr
%type < Ast.expr > expr

%%

expr:
        expr   PLUS   expr   { 0 }
|       expr   MINUS        expr   { 0 }
|       expr   TIMES expr   { 0 }
|       expr   DIVIDEexpr   { 0 }
|       expr   COMMA        expr   { 0 }
|       NEG             expr          { 0 }
|       asn
|       loop
|       LITERAL                              { 0 }
```

```
|       VARIABLE                        { 0 }



asn:
        tpe             VARIABLE    ASN     expr    { 0 }
|       VARIABLE        MULTASN     expr    { 0 }
|       VARIABLE        DIVASN              expr    { 0 }
|       VARIABLE        PLUSASN     expr    { 0 }
|       VARIABLE        SUBASN              expr    { 0 }
|       tpe             VARIABLE    ASN       l_expr        { 0 }


l_expr:
        NOT             l_expr              { 0 }
|       l_expr          AND         l_expr  { 0 }
|       l_expr          OR          l_expr  { 0 }
|       BOOL

loop:
        IF      LPR l_expr      RPR LBR expr RBR { 0 }
|       FOR LPR asn SEMICOLON l_expr SEMICOLON ASN RPR LBR expr RBR { 0 }
|       WHILE LPR l_expr RPR LBR expr RBR { 0 }
|       IF      LPR l_expr      RPR LBR expr RBR ELSE LBR expr RBR { 0 }
|       IF      LPR l_expr      RPR LBR expr RBR ELIF LPR l_expr RPRLBR expr RBR { 0 }

program: decls EOF { 0 }

decls:  { 0 }
| decls vdecl { 0 }
| decls fdecl { 0 }
| decls expr { 0 }

fdecl:

FUNC VARIABLE LPR formal_list RPR LBR vdecl_list stmt_list RBR { 0 }

formal_list:
typ VARIABLE { 0 }
| formal_list COMMA typ VARIABLE { 0 }

typ: INT { 0 }
| CHAR { 0 }
```

```
| STR { 0 }
| BLOB { 0 }
| FLT
| BOOL { 0 }
| NULL { 0 }

vdecl_list:  { 0 }
| vdecl_list vdecl { 0 }

vdecl: typ VARIABLE SEMICOLON { 0 }

stmt_list: {0}
| expr {0}
```

++
++

Scanner:

```
{ open Parser }

rule token = parse
[' ' '\t' '\r' '\n'] { token lexbuf }
| '+' { PLUS }                                  | "&&" { AND }
| '-' { MINUS }                                 | '!' { NOT }
| '*' { TIMES }
| '/' { DIVIDE }                                | "if" { IF }
| '%' { MOD }                                   | "else" { ELSE }
                                                | "else if" { ELIF }
| ['0'-'9']+ as lit { LITERAL(int_of_string lit)
}                                               | "while" { WHILE }
| eof { EOF }                                   | "for" { FOR }
| '$'['0'-'9'] as lit { VARIABLE(int_of_char
lit.[1] - 48) }                                 | "function" { FUNC }
| ',' { COMMA }                                 | "stimulus" { STIM }

| '=' { ASN }                                   | "int" { INT }
| "*=" { MULTASN }                              | "char" { CHAR }
| "/=" { DIVASN }                               | "string" { STR }
| "+=" { PLUSASN }                              | "float" { FLT }
| "-=" { SUBASN }                               | "bool" { BOOL }
                                                | "blob" { BLOB }
| "!=" { NE }                                   | "null" { NULL }
| "==" { EQ }
| "<" { LT }                                    | '(' { LPR }
| "<=" { LTE }                                  | ')' { RPR }
| ">" { GT }                                    | '{' { LBR }
| ">=" { GTE }                                  | '}' { RBR }

| "||" { OR }                                   | ';' { SEMICOLON }
```