SEAMScript

Language Reference Manual

Sean Inouye (si2281)
Edmund Qiu (ejq2106)
Akira Baruah (akb2158)
Maclyn Brandwein (mgb2163)

October 2015

Contents

1 Introduction **2**

2 Fundamental Types **2**

3 Arrays **2**

4 Comments **2**

5 Literals **3**

6 Variables **3**
6.1 Names 3
6.2 Declaration 4
6.3 Access 4
6.4 Assignment 4

7 Operators **4**

8 Statements and Blocks **5**

9 Control Flow **5**
9.1 If/Else Statement 5
9.2 While Loop 6
9.3 For Loop 6

10 Entities **6**

11 Built-In Entities **7**
11.1 screen 8
11.2 keyboard 8
11.3 loader 8

12 Built-In Functions **8**

13 Layout **9**

14 File Structure **10**

15 Function Definitions **10**

1 Introduction

SEAMScript is a simple high-level language that focuses on entity-based applications. Applications, primarily simulations and games, benefit from a built in system for handling running events periodically, and from built in functionality to simplify the typical I/O expected from these sorts of apps. Simple games, such as Breakout! or Snake, can be prototyped much more rapidly than in other languages. Other simulations, like cars interacting at an intersection, can also be written fairly quickly. Compared to real-life, the accuracy of a SEAMScript program is low due to concerns left to developers such as buffering and interpolating events between time deltas, but the native support for 'steps' saves developers from the hassle of manually starting/stopping entities.

Throughout this document, "...[a comment]..." will be used to indicate places where code of the type described in the comment is omitted for brevity but assumed present by the compiler.

2 Fundamental Types

SEAMScript is statically typed and supports the following primitive types:

- `int` - Signed integers with architecture-specific size.
- `string` - ASCII-based strings of arbitrary length and enclosed by a pair of double quotations.
- `float` - 64-bit IEEE floating point numbers.
- `bool` - Boolean data.
- `texture` - Stored image primitives.
- `entity` - Agent primitives. See section 10.

3 Arrays

Every fundamental type can be used with arrays, and array operations. Arrays are indicated by adding `[<# of items>]` after the identifier in a given variable declaration. For example, to declare an array of 10 entities, one would write:

```
entity blocks[10]
```

To assign an entire array, start with a `{` token, write the expected number of valid literals (see literals) for the array type delimited by `,` tokens, and end with a `}` token. entity and texture array types cannot be assigned this way. A trailing comma after the last element in the array is not permitted. For some examples:

```
int int_array[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } # Valid #
float float_array[5] = { .50 }                        # Invalid; only 1 of 5 floats def #
string string_array[2] = { "hello", "world" }         # Valid #
boolean bool_array[3] = { true, true, false, }        # Invalid; trailing comma #
```

4 Comments

Only block comments are supported. They are started with the token `#` and ended with another `#`, and may not be nested. Anything in between the comments will not be read by the parser. Comments may not be nested. For example, `# This is a comment #` is a comment. `# Malformed # comment ##` is a malformed comment (once the parser hits "comment", a syntax error is indicated).

5 Literals

Literals represent fixed values of ints, strings, floats, and booleans. These values are used in assignment or often calculation operations. The format and semantics of literals for each type are as follows:

- `int` - Integers are declared with either a sequence of one or more digits from 0-9, potentially prefixed with a `-` to indicate negative numbers. You may have integers of any length, although numeric overflow may result if you exceed the representable length of an int on your hardware.

  Examples:

  ```
  int a_number = -35 # Valid #
  int num = 24. # Invalid #
  int a_positive = +5 # Invalid; \+" is assumed #
  ```

- `float` - Floating point values are declared with an optional prefix of `-` to indicate negative numbers, a sequence of zero or more digits from 0-9, a mandatory `.`, and one or more digits from 0-9. Like integers, you may write out numbers unrepresentable on your hardware, but numeric overflow will occur. Floating point values will lose a minute amount of precision once run on hardware.

  Examples:

  ```
  float a_float = -35.1 # Valid #
  float another_float = -.334 # Valid #
  float not_valid = 34. # Invalid; need a decimal portion. #
  float also_wrong = . # Invalid #
  ```

- `string` - String literals are defined by ASCII characters within quotes. To include a quotation mark within a string literal, you must first escape it with a `\`. String literals may be empty, and there is no limit to their length.

  Example:

  ```
  string string_beans = \String beans" # Valid #
  string a_quote = \Quoth the Raven, \"Hello\"" # Valid #
  string bad_quote = \He dictated \here is a dictate"" # Invalid #
  ```

- `boolean` - Boolean literals are defined by `true` and `false`.

  Example:

  ```
  boolean bool_type = true # Valid #
  boolean wrong = FALSE # Invalid; wrong case #
  ```

- `entity` and `texture` - These types do not have associated literals.

6 Variables

6.1 Names

Variable names are a combination of lowercase letters, uppercase letters, and underscores. They must begin and end with a letter (either uppercase or lowercase). For example, `Hello`, `hi_there`, and `variable_` would be supported, but `_variable`, and `hello2` would not be.

6.2 Declaration

Variables are declared in the format:

`<type> <identifier><optional array specification>`

You may optionally assign the newly declared variable a value upon creation, but you must heed the standard variable assignment rules (see below). Re-declaring variables with any reused name from any scope is unsupported, except within the scope of the entity. Two entities may have member variables with same identifier (and often will, in fact), and they may reuse identifiers in the global scope. You may declare variables in the global scope, within functions, in the body of an entity, and in entity member functions.

6.3 Access

Variables are considered "accessed" when their identifier is used outside of their initial declaration or assignment. For example, `string catdog = cat + dog` would access the values stored at `cat` and `dog`, but not `catdog` because it is being declared. Global variables may be accessed from any portion of code after they're declared. Local variables – variables found in function arguments or declared within a given block – may be accessed within the same or a deeper level of block indentation, but not within shallower indentation. For example,

```
for(int i = 0; i < 2; i++):
    screen.log(int_to_string(i))
screen.log(int_to_string(i))
```

would be an invalid block of code because the second `screen.log(int_to_string(i))` references `i`, which is already out of scope (variables declared in a for loop may not be used outside of the indented code in the loop.

6.4 Assignment

Variables are assigned to literals, other variables, or the results of built in operators with the `=` token.

7 Operators

The supported operators are shown in the below table. Note that promotion is not supported – you may not divide a float by and int, or add a float and an int, and so on and so forth. See built-in functions for functions that deal provide conversions to get around these sorts of issues.

Operator	Meaning	Supported Types (LHS/RHS)
`+`	Add the LHS value and the RHS value and return the result	any pair of `int`, `float`, `string` (concatenation)
`-`	Subtract the RHS value from the LHS value and return the result	any pair of `int`, `float`
`*`	multiply the RHS and the LHS and return the result	any pair of `int`, `float`
`/`	divide the RHS and the LHS and return the result	any pair of `int`, `float`
`==`	Compare the LHS with the RHS for equality (true if equal, otherwise false)	any pair of `int`, `float`, `string`, `entity`, `bool`
`!=`	Compare the LHS with the RHS for inequality (true if not equal, otherwise false)	any pair of `int`, `float`, `string`, `entity`, `bool`
`&&`	If both the LHS and RHS are true, return true, otherwise return false	boolean expressions
`||`	If either the LHS or RHS are true, return true, otherwise return false	boolean expressions
`%`	Divide the LHS by RHS and return the remainder	pair of `int`

8 Statements and Blocks

Statements are terminated by a newline character. Blocks of code (e.g. what follows control flow or function declaration) are marked by increasing the level of indentation by one tab. Tabs alone are supported – tabbing done with other forms of whitespace will not be recognized and will generate syntax errors.

9 Control Flow

Control flow is supported with if/else statements, while loops, and for loops.

9.1 If/Else Statement

If statements start with an `if`, are followed with a left paren, an expression, a right paren, a colon, an indented block, optionally all followed by an `else`, a colon, and another indented block . The indented blocks must contain code other than comments. For example:

```
if(score > 100):
    score = score + 50
else:
    score = score + 100
```

would be accepted as valid. However,

```
if(score > 150):
    else:
        score = 100
```

or

```
if(score > 200):
        score = 250
    else:
    score = score + 10
```

would be considered invalid.

9.2 While Loop

While loops start with a `while`, are followed with a left paren, an expression that evaluates to true or false, a right paren, a colon, and an indented block. The indented block must contain code other than comments. For example:

```
int i = 5
while(i < 10):
    i = i + 1
```

is considered valid. However,

```
int i = 5
while(true):
i = i + 5
```

is considered invalid.

9.3 For Loop

For loops start with a `for`, are followed with a left paren, a declaration of any supported variable, a semicolon, a boolean expression, a semicolon, an operation that runs every time after the for loop completes, a right paren, a colon, and an indented block. The indented block may be empty. For example:

```
for(int i = 0; i < 5; i = i + 1):
    # i would be 5, but it's now out of scope #
```

is considered valid. However,

```
int i = 10
for(; i < 5; i = i - 1):
```

is considered invalid because there's no variable declaration. The variable declaration may simply be assignment, however, so:

```
int i = 10
for(i = 0; i < 5; i++):
```

is considered valid.

10 Entities

Entities are collections of variables and methods, with special methods that are invoked by SEAMScript at various times if they exist. Conceptually, entities are very close to objects in other object-oriented languages, although entities lack certain features of objects and possess a bit of extra functionality. Entities may contain methods, they may contain any number of variables (including other entities).

Entities are started with the keyword `spawn`, followed by a left parenthesis, an entity's name, the arguments it takes, and then a right parenthesis. For example,

```
entity car = spawn(Car, 50, 100) # Car's start function takes in two integers #
```

As soon as an entity is spawned, it is considered 'staged' to have its step and render functions called in the pipeline. Entities are stopped with the keyword `kill`. After an entity is 'killed', it may no longer be used. For example,

```
kill car # Stops the car
```

Entities are declared with `entity`, an identifier that must start with a capital letter, a colon, and followed by an indented block containing (in order):

- Variable declarations for any variables accessible throughout the entity and to other portions of code with a reference to the entity. Note assignment with declaration is not allowed here.
- Any user-defined functions. The format for these is the same as other function declarations. Other code can directly call these functions.
- Functions the language uses. These functions, all of which are optional, but if used must have at least one statement of executable code, are:
  - `start` - This function is called when an entity is created. start's arguments are user-defined, but all must be provided to the language keyword `spawn` that starts the entity. `start` can be considered a sort of constructor.
  - `stop` - This function is called when an entity is destroyed with `kill`. `stop` is considered like a destructor.
  - `step` - Step is called 24 times a second on any entities that have 'start'ed.
  - `render` - Render is also called 24 times a second, but is called on each entity after every entity with a step function has had step called (i.e. in a program with 2 entities, SEAMScript will call `step` on both first, and then call `render` on both. Render is highly recommended not to modify any variable value, and should just be used for drawing/output work, but it is to use render as a general-purpose function.

Entities `step` and `render` functions are called in the order the entities are 'spawned'. When an entity is removed with `kill`, the order in which `step` and `render` functions are called is not modified, except to remove the 'dead' entity from the list. Neither step nor render should contain infinite loops; this will prevent the program from running. Some examples of entity definition and use are:

```
entity Player:
    int score
    string name

    int reverse_direction(int direction):
       ...a user defined function...

    function start(int start_score):
       ...initialization code...

    function stop():
        ...stop code...

    function step():
        ...step code...

    function render():
        ...draw code...
```

11 Built-In Entities

To facilitate rapid development of certain types of applications, SEAMScript contains a few built-in objects that behave like entities. These built-ins are:

11.1 screen

- Properties
  - `width` - The width of the display screen. (int)
  - `height` - The height of the display screen. (int)
- Methods
  - `draw_sprite(texture tex, int x, int y)` - returns 0 - Draw a texture 'tex' to the screen at `x`, `y`.
  - `draw_rect(int color, int x, int y, int width, int height)` - returns 0 - Draw a filled rectangle with no border with the color `color`, width `width`, height `height`, x-position `x`, and y-position `y`.
  - `log(string to_log)` - returns 0 - Logs the string `to_log` to `stdout`.

11.2 keyboard

- Properties
  - `{left,right,up,down,space}.pressed` - returns boolean - Whether one of the listed keys has been pressed. Once checked, subsequent checks will return false until a complete key up/key down event has been performed again. For example,

    ```
    if(keyboard.left.pressed == true):
        screen.log(\Left pressed!")
    ```

    would be a valid use of this property.
- Methods
  - (NONE)

11.3 loader

- Properties
  - (NONE)
- Methods
  - `load_tex(string filename, int desired_width, int desired_height)` - returns a texture - The only way to load a texture, `load_tex` takes in a filename, width, and height, and generates a texture of those parameters. If the given file (expected to be in a directory relative to the executable) is not found, a runtime error is created and the program will crash.

12 Built-In Functions

Built-in functions provide conversion facilities.

- `int_to_string(int i)` - Convert `i` to its string representation.
- `int_to_float(int i)` - Convert `i` to its floating-point representation. This may result in a slight loss of precision.
- `int_to_boolean(int i)` - Convert `i` to its boolean representation. `0` will be converted to `false`, while everything else will be converted to `true`.

- `float_to_int(float f)` - Convert `f` to its integer representation. If the floating-point value exceeds what is representable in integers, or has a decimal portion, a loss of precision will result.
- `float_to_string(float f)` - Convert `f` to its string representation. Up to 4 decimals places will be printed.
- `boolean_to_int(boolean b)` - Convert `b` to its integer representation. `false` will be converted to `0`, while true will be converted to `1`.
- `boolean_to_string(boolean b)` - Convert `b` to its string representation. false will be converted to `false`, and true will be converted to `true`.

13 Layout

The layout of a SEAMScript program is as follows (in order):

- File includes (optional); see file structure
- Global variable declarations (optional); assignment here is not supported
- Function definitions (optional)
- Entity definitions (optional, but necessary for any real work)
- Main (required) - A function named `main` that's the entry point of the program. Main is responsible for initially staging all entities. If a program needs to restage entities regularly, developers should consider creating an entity that does staging in its step function depending on various state values stored in global variables (e.g. a couple variables called `level` and `is_done`, and an entity of type `level` would be a canonical way to do it). The main function is called once the program starts and is never called again. Although the compiler doesn't check, the main function should not contain an infinite loop. Since the functions for `step` and `render` on entities are not called on a regular basis until after main concludes, infinite loops will prevent the program from running.

An example layout would be as follows:

```
include \tilemap.seam"
...more includes...

int level
...more global variables...

int get_current_terrain(int x, int y):
    ...function definition...
...more user functions...

entity Player:
    ...player definition...
...more entity declarations...

function main():
    player = spawn(Player, 50, 50)
    ...more init code...
```

14 File Structure

SEAMScript does not have a robust system of library supports, but it is possible to approximate libraries by including other files in your program so long as there are no namespace conflicts. To other file in your program, whose mains will be called before the main of your program, and in the order they are included, using the following syntax:

```
include "filename.seam"
```

15 Function Definitions

Functions must be defined before they are called in SEAMScript. A function declaration must adhere to the following format:

```
<return type OR "function" keyword> <identifier> (<argument list>):
    ...block of statements...
```

If a function definition begins with the `function` keyword, it is implied that the function does not return a value (similar to `void` in C-like languages). The argument list consists of 0 or more identifiers separated by commas. The block of statements describing the function's behavior must be one indentation level past that of the function declaration itself. If a return type is specified (i.e. the declaration begins with a primitive type rather than `function`), the function block must contain a `return` statement, which returns control to the calling function and returns the value of the expression following the `return` keyword.