Simple 2D Game Language Documentation

Author: William Luo
Created On: November 22, 2019
Last Modified: November 25, 2019

Contents

1 Overview

Simple 2D Game Language (*SGL2D*) is an external domain specific language aimed towards new or novice programmers interested in created 2D computer games. With *SGL2D*, users can expect to create tile-based or grid-based computer games with basic features such as player movement, object (sprite) interactions, and simple color customization. One key feature of *SGL2D* is that it simplifies the programming aspect for the user, as a result the user avoids the intricacies and complexities of the data structures and frameworks that are necessary for 2D games and only needs to work with *SGL2D's* basic intuitive syntax. Through working with *SGL2D*, users can learn about basic game development which can serve as an entry point into more advanced game development and gain a basic understanding of writing code.

The current version (1.2) of *SGL2D* is an early version of *SGL2D* and has limited functionality and basic error checking which may be expanded on and improved in future versions.

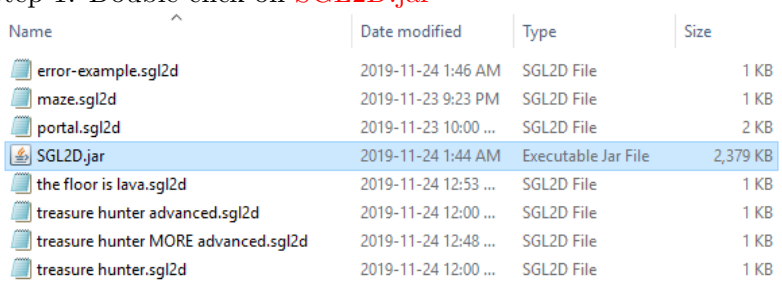
2 Getting Started

SGL2D is written in it's own unique syntax and is parsed and interpreted into the Java programming language. As a result Java is required to use *SGL2D*. *SGL2D* has been tested to support Version 8, alternative versions have not been tested. Java can be downloaded and installed at https://www.java.com/en/download/.

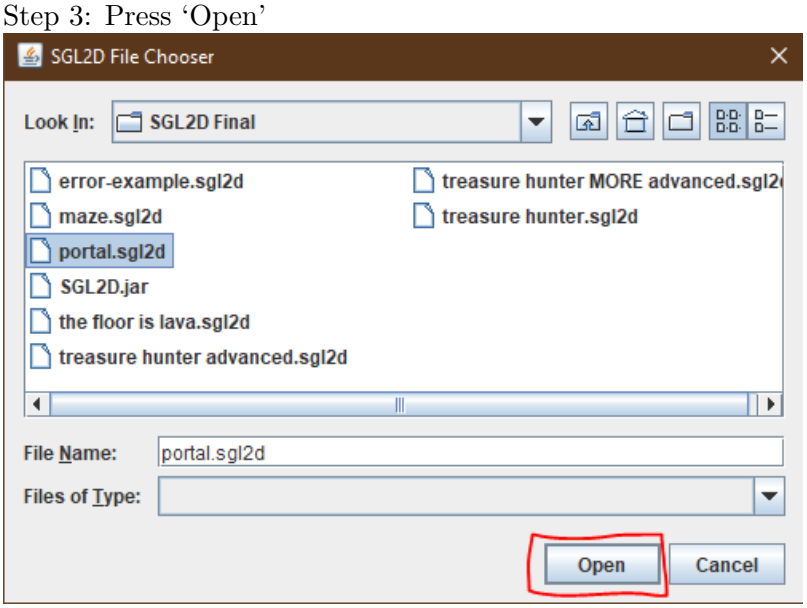
SGL2D can be downloaded HERE as an executable jar file.
Alternatively, the source code can be found HERE and compiled manually.

The file extension for *SGL2D* games is '.sgl2d' which helps with organization and is for compatibility with future versions. However, in the current version of *SGL2D*, plain-text files, '.txt' will work.

To use SGL2D.jar:
Step 1: Double-click on SGL2D.jar



Step 2: Select the file containing the game code
Step 3: Press 'Open'



3 Language Syntax and Semantics

3.1 Introduction to the Syntax

```
expression
    :
    | environment
    | goal
    | player
    | sprite
    | counter
    | event
    | set
    | newline
    ;
```

```
action
    : incCounter1
    | decCounter1
    | incCounter2
    | decCounter2
    | incCounter3
    | decCounter3
    | incSpriteCounter
    | decSpriteCounter
    | setSpriteCounter
    | moveToRandomOnZero
    | transformOnZeroCounter
    | gameOverOnZeroCounter
    | winOnZeroCounter
    | moveToRandom
    | transformToSprite
    | movePlayerTo
    | playerIncCounter1
    | playerIncCounter2
    | playerIncCounter3
    | playerDecCounter1
    | playerDecCounter2
    | playerDecCounter3
    | playerIncSpriteCounter
    | playerDecSpriteCounter
    | gameOver
    | win
    ;
```

The simplified Extended Backus–Naur form is show above which dictates what language, syntax, or code is allowed for *SGL2D*. *SGL2D* consists of 7 base functions which are used to control all the objects in the environment. The environment is the space in which all objects, called 'sprites' are visualized. The goal is the cell on the environment in which the player must go to to win the game, this is non-essential for making a *SGL2D* game. The player is the controllable player in the game. Sprites are the objects in the game that aren't the player or goal and can take on a large variety of events or actions which will cause a change in the environment when the condition is met. Set is for placing the sprites onto the environment at specific locations.

More information on the specific syntax and usage can be found in **Section 5: Keyword Index**.

3.2 Language Features

SGL2D has some language features that are useful for beginner programmers such as named keywords, duplicate arguments, default settings, newline separation, automatic whitespace and newline detection, non-case-sensitive, typo-handling, and error feedback.

1. Named Keywords
   Named keywords appear in Python as well as in Cascading-Style-Sheets (CSS) and are also used in *SGL2D*. The advantage of named keywords is that the user explicitly defines what value is for which argument. This makes it easier to interpret what each argument to a function is and it removes the need for the user finding and/or memorizing the order of arguments

to functions. For example:

```
Without named keywords: setSize > 5 5
With named keywords: setSize > x=5 y=5 (or setSize > y=5 x=5)
```

For named keywords, the keyword identifier specifies which variable is being modified or set followed by a '=' and a value. In the above example, the 'x' and 'y' would be the keyword identifier and '5' would be the value to set each of them as.

2. Duplicate Arguments
   Duplicate arguments allows the passing of the same arguments to a function. This feature is useful because it prevents accidentally duplicating arguments as an error and also for allows users to quickly test modifications without modifying the original value(s). In *SGL2D*, only the last argument to a function is applied unless otherwise specified. For the following example, the color to set 'player' to will be 'green', the 'player' x-position and y-position will be set as '1' and '2' respectively:

```
Player > x=5 y=6 color=red color=blue color=green x=1 y=2
```

3. Default Settings
   Most of the functions in *SGL2D* (excluding events) has a default value so if it is not specified by the user, the default values will be used. This features allows users to not have to set common values such as the 'goal' sprite having a 'yellow' color and avoids errors in the case that the user forgets to pass an argument to a function. For the following example, the 'goal' sprite will be positioned at (0,0) and have the color 'yellow'.

```
Goal >
```

4. Newline Separation In *SGL2D*, newlines are used to separate each function call rather than the typical semi-colons in most programming languages. The benefit of this is that code becomes easier to read as newlines prevent multiple functions from being performed on one line and also avoids the common error of forgetting to place semi-colons.

5. Automatic Whitespace and Newline Detection
   In *SGL2D*, multiple whitespaces between keywords and multiple newlines between functions are automatically detected by the parser and ignored. The benefit is that the user can use newlines to section code for easier management and any accidental whitespace between arguments are ignored which lessens the strictness of the syntax. For the following example, both of the two function will run perfectly fine:

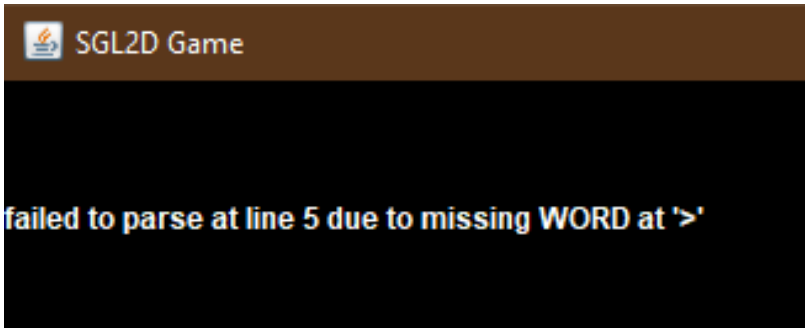
```
Player > x=5 y=5 color=blue
Goal > x=7 y=7 color=yellow
----------------------------------------
Player    > x=5    y=5         color=blue


Goal   >        x=7 y=7   color=yellow
```

6. Non-Case-Sensitive
   SGL2D is not case-sensitive so for example if the name of a sprite is called 'Tree', it will be interpreted the same as if it were 'tree' or 'TREE'; this applies to keywords as well. The benefit is that if proper capitalization is not matched, it will still function as expected which can be a common programming error among beginners. This will also make the language easier to read as the user will not have to distinguish between similarly spelled words with different capitalization.

7. Typo-Handling
   SGL2D will detect typos in the syntax and if there is a typo within the keyword arguments, the parser will end parsing prior to the typo. If the typos are at the end of the keyword argument, it will ignore it when parsing the correct keyword argument and end parsing after that. For the first example, the parsing stops after '>' and the color of 'goal' will be set to the default 'yellow' and in the second example, the counter is set to 10 and parsing will stop after that argument:

```
Goal > color=gren
Counter counter1 > 10p true
```

8. Error feedback
   SGL2D can provide users with error feedback when their code produces some problems with parsing which is invaluable when it comes to debugging code. The error message indicates the line number where the problem occurred, the character(s) that are causing the problem, and what the expected value(s) was supposed to be. In the following example, the error is on line 5, the character(s) causing the problems is before '>', and the parser expected a 'WORD':



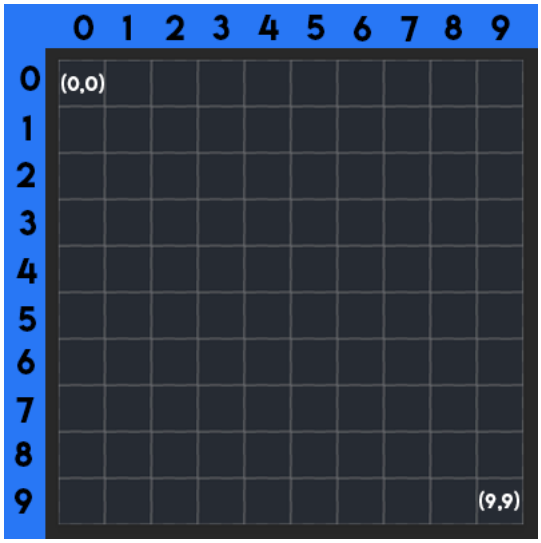
error-example.sgl2d - Notepad

File Edit Format View Help

```
environment >
player > x=5 y=5 color=blue
sprite low > color=red
sprite med > color=yellow
sprite > color=green
sprite floor > color=grey
counter counter1 > true 0
counter counter2 > true 50 gameover
counter counter3 > true 55
event player > playerdeccounter2=1
event low > movetorandom=floor incc
event med > movetorandom=floor incc
event high > movetorandom=floor inc
set floor > (x=0 y=0, x=9 y=9)
set low > x=1 y=5
set med > x=5 y=1
set high > x=9 y=5
set floor > (x=0 y=0, x=9 y=9)
```

4 Basics

4.1 The Environment Grid



The environment in the current version of *SGL2D* is represented by a 10-by-10 grid using 0-based indexing with the top left corner being represented by (0,0) and the bottom right corner being represented by (9.9). Each sprite can take up one cell in this grid and they are placed onto it with the Set function. The player and goal sprites are placed on the upper-most level so if a sprite and a player or goal are on top of each other, the player or goal would appear in the environment. A default sprite is initially placed in every grid cell which has the color black and is not solid (traversable). All sprites and player movements are also automatically bounded by the environment so they cannot leave the environment.

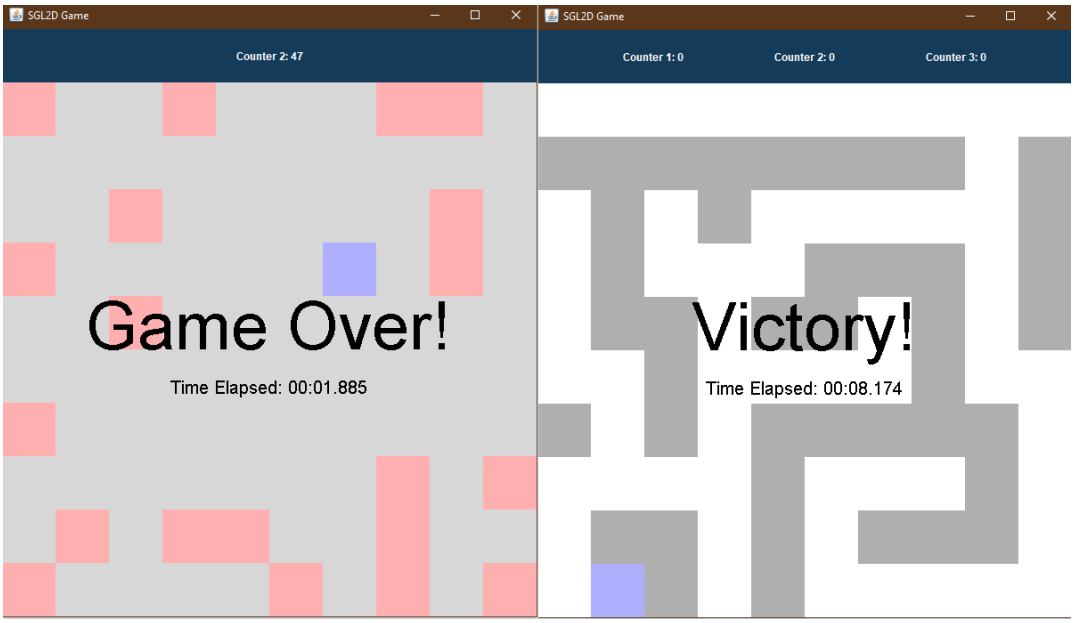
4.2 Updating the Game

Most, if not all *SGL2D* games will require a player to be instantiated which can be done with the player function. In *SGL2D*, all updates to the environment are triggered by player movements. All sprites are initially static and will not respond to player movements however several events can be applied to the sprites to give them certain effects, some of which will allow them to change the environment depending on whether the player moves on top of them, or if the player makes any moves at all. More details can be found in Section 5.6: Event.

4.3 Player Controls

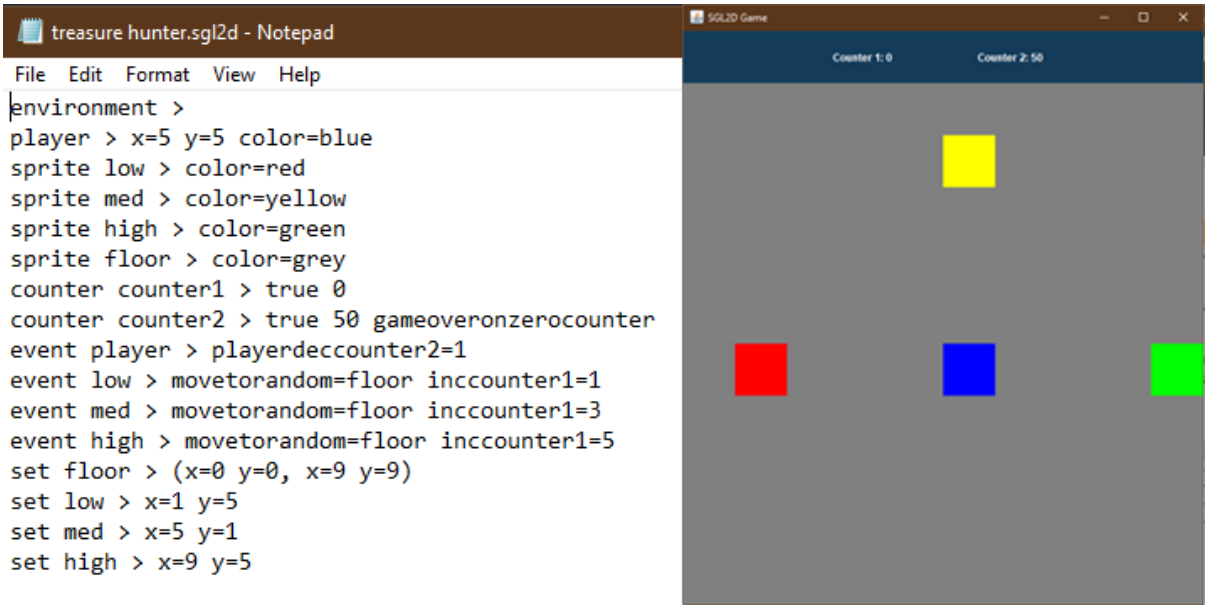
Currently there are only 5 keyboard inputs to a *SGL2D* game. The player is moved with the arrow keys UP, DOWN, LEFT, RIGHT and ESC can be used to close the game. Player movement is locked when the game ends. This will be expanded on in future versions of *SGL2D* to allow more interactability as well as a replay input.

4.4 Game Over and Victory States



There are two ways to properly end a game in *SGL2D*, the 'game over' state and the 'win' state. At the end of a game, the total time elapsed will be displayed. There are several ways in *SGL2D* to trigger the end state such as having the player reach the goal sprite, having a sprite's internal counter reach zero with the flag GameOverOnZeroCounter or WinOnZeroCounter enabled, touching a sprite with the flag GameOver or Win, or having one of the three counters reach zero with the the flags GameOverOnZero or WinOnZero enabled. Once the game ends, all player movement will be locked. There is currently no way to reset the game without closing and reopening the game and counter values and the time elapsed is not saved. This will be added in future implementations of *SGL2D*. For more information on the flags see Section 5.6: Event.

4.5 Writing the Code



SGL2D 'reads' the code from the first line to the last line unless an error occurs in which case it stops parsing at the error. Therefore, in order to use any kind of sprites, they must be defined prior to usage. The recommended ordering of writing the code for any *SGL2D* game is:

1. Environment
2. Player, Goal, Sprites, Counters
3. Events
4. Set

The first thing to define in any *SGL2D* game is the environment which is where all sprites will exist. This corresponds to the first line in the sample game 'Treasure Hunter'. The following 7 lines correspond to the definition of the player, specifying it's position and color as well as the 'low', 'med', 'high' sprites, and counters used in the game. Afterwards, the next 4 lines corresponds to events that are added to the player and sprites to give them interactability. Finally the last 4 lines sets all the sprites positioning on the environment. In *SGL2D*, each grid cell only is allowed to hold one sprite excluding the player and goal sprites so there cannot be any overlap in sprites. To use this to our advantage, the environment should be set from the background to the foreground. In the sample code, the floor first covers the entire grid and sprites corresponding to the treasure objects are added later which replaces the floor sprite. This allows less code to be written as we no don't need to use multiple ranges to place the floor around specific grid cells. For more examples and sample games, see **Section 6: Sample SGL2D Games**.

5 Keyword Index

5.1 Environment

```
environment : ENVIRONMENT ARROW (XINT | YINT)*;
```

The environment keyword initializes and defines the environment to be of a fixed size given by 'x=*columns' and 'y*=rows'. When no corresponding arguments are given, the default settings are 'x=10' and 'y=10'.
*Note: The current version of *SGL2D* does not support customizable grid sizes so this is not required for a functional *SGL2D* game. It is however recommended to be added for future compatibility.

```
Environment >
Environment > x=12 y=12
```

5.2 Goal

```
goal : GOAL ARROW (XINT | YINT | COLOR)*;
```

The goal keyword initializes and defines a sprite which is a goal in the environment. The goal implicitly has the win event applied to it. When no corresponding arguments are given, the default settings are 'x=0', 'y=0', and 'color=yellow'.

XINT: x-position
YINT: y-position
COLOR: color, see **Section 5.8.1 : Color**

```
Goal >
Goal > color=green
Goal > x=1 y=2 color=red
```

5.3 Player

```
player : PLAYER ARROW (XINT | YINT | COLOR)*;
```

The player keyword initializes and defines a sprite which is the player in the environment. The player is controllable via arrow keys and can trigger events with sprites. When no corresponding arguments are given, the default settings are 'x=0', 'y=0', and 'color=blue'.

XINT: x-position
YINT: y-position
COLOR: color, see **Section 5.8.1 : Color**

```
Player >
Player > color=green
Player > x=1 y=2 color=red
```

5.4 Sprite

```
sprite : SPRITE WORD ARROW (COLOR | SOLID)*;
```

The sprite keyword initializes and defines a sprites named as the given 'WORD'. The sprites has settings for whether it is solid or not and can be set to a give color. If the sprites is set to be solid, the player will not be able to move onto the sprites. When no corresponding arguments are given, the default settings are 'color=yellow' and 'solid=false'.

COLOR: color, see **Section 5.8.1 : Color**
SOLID: whether the sprite is traversable or not

```
Sprite Snow >
Sprite Grass > color=green
Sprite Wall > color=black solid=true
```

5.5 Counter

```
counter : COUNTER COUNTERINDEX ARROW (BOOLEAN | INT | WINONZERO | GAMEOVERONZERO)*;
```

The counter keyword initializes and defines one of the three counters in the Counter Panel. Counter 1 is located at the top left, Counter 2 is located in the middle and Counter 3 is located at the top right. If only one counter is enabled, it will be centered in the Counter Panel. Additional flags are allowed to be given to trigger certain end game states when the counter reaches exactly 0. The counter value is able to go negative. When no corresponding arguments are given the default settings are 'false' and '0'.

COUNTERINDEX: specifies the counter to modify (1, 2, or 3) BOOLEAN: whether the counter is enabled or disabled
INT: value to set the initial counter value to
WINONZERO: sets the game to a win state when the specified counter reaches 0
GAMEOVERONZERO: sets the game to a game over state when the specified counter reaches 0

```
Counter 1 >
Counter 2 > false 10
Counter 3 > true 30 WinOnZero
```

5.6 Event

```
event : EVENT (PLAYER | WORD) ARROW action*;
```

Applies the given event listener(s) to a initialized sprite or goal. For the list of available **actions**, see below.

5.6.1 IncCounter[1,2,3]

```
incCounter# : INCCOUNTER# (INT)*;
```

The IncCounter# keyword sets the flag for an event on a sprite which increments the corresponding counter based on the given value when a player moves onto the sprite. All arguments must be specified, there is no default value.

INCCOUNTER#: specifies the counter to be incremented when a player moves onto the sprite (ex. IncCounter1)
INT: the value which to increment the counter by when a player moves onto the redsprite

```
Event Treasure > IncCounter1=10
```

5.6.2 DecCounter[1,2,3]

```
decCounter# : DECCOUNTER# (INT)*;
```

The DecCounter# keyword sets the flag for an event on a sprite which decrements the corresponding counter based on the given value when a player moves onto the sprite. All arguments must be specified, there is no default value.

DECCOUNTER#: specifies the counter to be decremented when a player moves onto the sprite (ex. DecCounter1)
INT: the value which to decrement the counter by when a player moves onto the sprite

```
Event TaxCollector > DecCounter1=100
```

5.6.3 IncSpriteCounter

```
incSpriteCounter : INCSPRITECOUNTER (INT)*;
```

The IncSpriteCounter keyword sets the flag for an event on a sprite which increments its internal counter based on the given value when a player moves onto the sprite. All arguments must be specified, there is no default value.

INCSPRITECOUNTER: default action keyword
INT: the value which to increment the sprite's internal counter by when a player moves onto the redsprite

```
Event Treasure > IncSpriteCounter=5
```

5.6.4 DecSpriteCounter

```
decSpriteCounter : DECSPRITECOUNTER (INT)*;
```

The DecSpriteCounter keyword sets the flag for an event on a sprite which decrements its internal counter based on the given value when a player moves onto the sprite. All arguments must be specified, there is no default value.

DECSPRITECOUNTER: default action keyword
INT: the value which to decrement the sprite's internal counter by when a player moves onto the redsprite

```
Event Treasure > DecSpriteCounter=5
```

5.6.5 SetSpriteCounter

```
setSpriteCounter : SETSPRITECOUNTER (INT)*;
```

The SetSpriteCounter keyword sets the sprite's internal counter to the given value. All arguments must be specified, there is no default value.

SETSPRITECOUNTER: default action keyword
INT: the value which to set the sprite's internal counter to

```
Event FoodSupply > SetSpriteCounter=10
```

5.6.6 MoveToRandomOnZero

```
moveToRandomOnZero : MOVETORANDOMONZERO;
```

The MoveToRandomOnZero keyword sets the flag for an event on a sprite to move to a random location when its internal counter reaches 0. All arguments must be specified, there is no default value.

MOVETORANDOMONZERO: default action keyword, takes the name of the sprite to set the old location to after moving

```
Event Magician > MoveToRandomOnZero=Floor
```

5.6.7 TransformOnZeroCounter

```
transformOnZeroCounter : TRANSFORMONZERO;
```

The TransformOnZeroCounter keyword sets the flag for an event on a sprite to transform it into the given sprite when its internal counter reaches 0. All arguments must be specified, there is no default value.

TRANSFORMONZERO: default action keyword, takes the name of the sprite to set the old location to after moving

```
Event Bomb > TransforOnZeroCounter=Fire
```

5.6.8 GameOverOnZeroCounter

```
gameOverOnZeroCounter : GAMEOVERONZERO;
```

The GameOverOnZeroCounter keyword sets the flag for an event on a sprite to set the game to 'game over' when its internal counter reaches 0. There are no arguments for it.

GAMEOVERONZERO: default action keyword

```
Event Money > GameOverOnZeroCounter
```

5.6.9 WinOnZeroCounter

```
winOnZeroCounter : WINONZERO;
```

The WinOnZeroCounter keyword sets the flag for an event on a sprite to set the game to 'victory' when its internal counter reaches 0. There are no arguments for it.

WINONZERO: default action keyword

```
Event Debt > WinOnZeroCounter
```

5.6.10 MoveToRandom

```
moveToRandom : MOVETORANDOM;
```

The MoveToRandom keyword sets the flag for an event on a sprite to move it to a random location when a player moves onto the sprite. The old location is replaced by the give sprite. All arguments must be specified, there is no default value.

MOVETORANDOM: default action keyword, takes the name of the sprite to set the old location to after moving

```
Event Wizard > MoveToRandom=Grass
```

5.6.11 TransformToSprite

```
transformToSprite : TRANSFORMTOSPRITE;
```

The TransformToSprite keyword sets the flag for an event on a sprite to transform it to a given sprite when a player moves onto it. All arguments must be specified, there is no default value.

TRANSFORMTOSPRITE: default action keyword, takes the name of the sprite to set sprite to when a player moves onto it

```
Event Flowers > TransformToSprite=Dirt
```

5.6.12 MovePlayerTo

```
movePlayerTo : MOVEPLAYERTO (XINT | YINT)*;
```

The MovePlayerTo keyword sets the flag for an event on a sprite which will teleport the player to the given location when a player moves onto it. All arguments must be specified, there is no default value.

MOVEPLAYERTO: default action keyword
XINT: x-position to move player to
YINT: y-position to move player to

```
Event Portal > MovePlayerTo= x=5 y=4
```

5.6.13 PlayerIncCounter[1,2,3]

```
playerIncCounter# : PLAYERINCCOUNTER# (INT)*;
```

The PlayerIncCounter# keyword sets the flag for an event on the player which increments the corresponding counter based on the given value when a player moves. All arguments must be specified, there is no default value.

PLAYERINCCOUNTER#: specifies the counter to be incremented when a player moves (ex. PlayerIncCounter1)
INT: the value which to increment the counter by when the player moves

```
Event Player > PlayerIncCounter1=10
```

5.6.14 PlayerDecCounter[1,2,3]

```
playerDecCounter# : PLAYERDECCOUNTER# (INT)*;
```

The PlayerDecCounter# keyword sets the flag for an event on the player which decrements the corresponding counter based on the given value when a player moves. All arguments must be specified, there is no default value.

PLAYERDECCOUNTER#: specifies the counter to be decremented when a player moves (ex. PlayerDecCounter1)
INT: the value which to decrement the counter by when the player moves

```
Event Player > PlayerDecCounter1=10
```

5.6.15 PlayerIncSpriteCounter

```
playerIncSpriteCounter : PLAYERINCSPRITECOUNTER (INT)*;
```

The PlayerIncSpriteCounter keyword sets the flag for an event on a sprite which increments its internal counter by the given value when a player moves. All arguments must be specified, there is no default value.

PLAYERINCSPRITECOUNTER: default action keyword
INT: the value which to increment the sprite's internal counter by when the player moves

```
Event Grass > PlayerIncSpriteCounter=1
```

5.6.16 PlayerDecSpriteCounter

```
playerDecSpriteCounter : PLAYERDECSPRITECOUNTER (INT)*;
```

The PlayerDecSpriteCounter keyword sets the flag for an event on a sprite which decrements its internal counter by the given value when a player moves. All arguments must be specified, there is no default value.

PLAYERDECSPRITECOUNTER: default action keyword
INT: the value which to decrement the sprite's internal counter by when the player moves

```
Event Icecube > PlayerDecSpriteCounter=1
```

5.6.17 GameOver

```
gameOver : GAMEOVER;
```

The GameOver keyword sets the flag for an event on a sprite sets the game to 'game over' when a player moves onto it. There are no arguments for it.

GAMEOVER: default action keyword

```
Event Hole > GameOver
```

5.6.18 Win

```
win : WIN;
```

The Win keyword sets the flag for an event on a sprite sets the game to 'victory' when a player moves onto it. There are no arguments for it.

WIN: default action keyword

```
Event Home > Win
```

5.7 Set

```
set : SET WORD ARROW (position | range | NEWLINE)*;
```

The set keyword is used to specify and place sprites onto the grid in the environment. Unlike every other function mentioned so far, set allows newline separation between positions and ranges to help with better organization of the code. Multiple positions and ranges can be specified on a single line, however it is recommended to use newlines when lines get too long for readability. If no arguments are specified, nothing happens.

POSITION: see **Section 5.8.2 : Position**
RANGE: see **Section 5.8.3 : Range**
NEWLINE: a newline

```
Set Tree >
Set Tree > x=5 y=5
Set Tree > (x=5 y=5, x=9 y=9)
Set Tree > x=5 y=5 x=1 y=1 (x=1 y=1, x=3 y=3) x=2 y=2
Set Tree > x=5 y=5
x=1 y=1
(x=1 y=1, x=3 y=3)
x=2 y=2
```

5.8 Attributes

5.8.1 Color

```
fragment ROYGBIV    : (R E D |
                       O R A N G E |
                       Y E L L O W |
                       G R E E N |
                       B L U E |
                       I N D I G O |
                       V I O L E T |
                       B L A C K |
                       W H I T E |
                       G R E Y |
                       G R A Y);
COLOR : C O L O R EQUALS ROYGBIV;
```

The Color keyword sets the color for all types of sprites. *SGL2D* uses the **ROYGBIV** color scheme which allows the colors ‘red’, ‘orange’, ‘yellow’, ‘green’, ‘blue’, ‘indigo’, ‘violet’, ‘black’, ‘white’, and ‘grey’ or ‘grey’.

```
Player > color=red color=orange color=yellow color=green color=blue
Goal > color=indigo color=violet color=black color=white color=grey
```

5.8.2 Position

```
position
    : XINT YINT
    | YINT XINT
    ;
```

The Position keyword sets the position for sprites. Unlike the usual 'XINT' and 'YINT' arguments which are optionally specified, the position keyword requires exactly one of each where order doesn't matter.

```
Set Floor > x=5 y=5
Set Wall > y=3 x=3
```

5.8.3 Range

```
range : '('position ',' position')';
```

The Range keyword sets the range of positions for sprites. The first position is the top left starting grid position and the second position is the bottom right ending grid position.

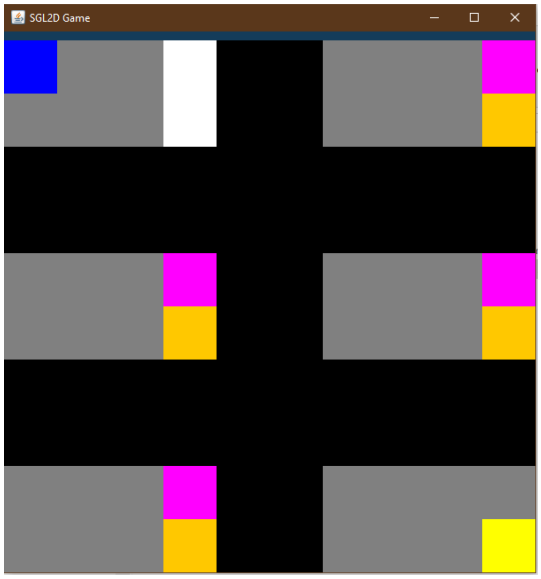
```
Set Floor > (x=0 y=0, x=9 y=9)
Set Wall > (x=0 y=3, x=8 y=3)
```

6 Sample SGL2D Games

6.1 Maze

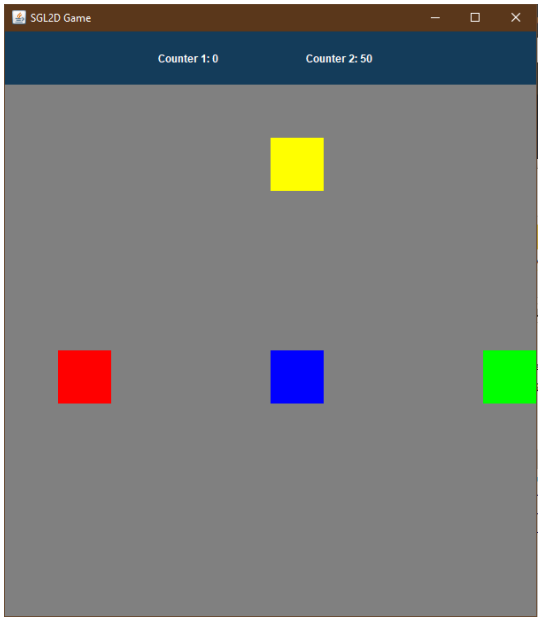
Maze is a typical maze game where you, the player, must get to the goal cell.
The *SGL2D* code for **Maze** can be found HERE.

6.2 Portal



Portal is a teleporting puzzle game where you, the player, must get to the goal cell by travelling through multiple portals where some may take you backwards.
The *SGL2D* code for **Portal** can be found HERE.

6.3 Treasure Hunter



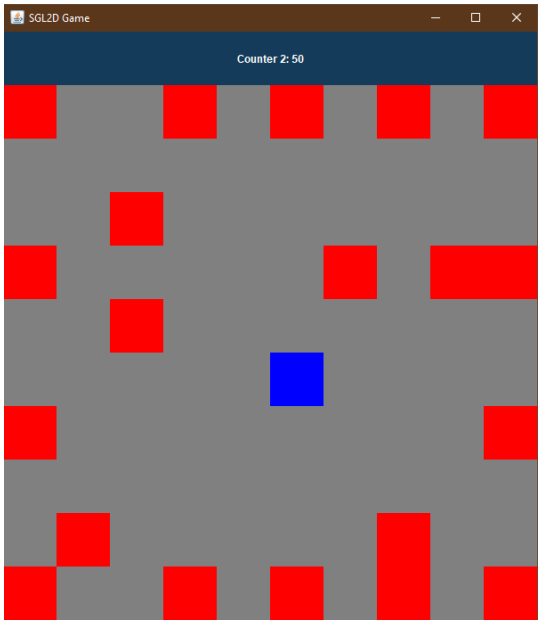
Treasure Hunter is a point-based game where you, the player, must earn points by collecting the colored blocks. The goal is to collect as many points as possible within the set moves. Red blocks are worth 1 point, yellow blocks are worth 3 points and green blocks are worth 5 points.
The *SGL2D* code for **Treasure Hunter** can be found HERE.

Alternative and more difficult versions (Included above):

Treasure Hunter Advanced is modified so that the blocks will move after the player makes a certain number of moves. Red blocks move after 15 moves, yellow blocks move after 10 moves, and green blocks move after 5 moves.

Treasure Hunter MORE Advanced is further modified where now red blocks are worth -3 points. Yellow blocks are still worth 3 points and will move after 12 moves. Green blocks are still worth 5 points and will move after 8 moves. The number of moves is increased to 75.

6.4 The Floor Is Lava



The Floor Is Lava is a survival game where you, the player, must make 50 moves without touching the lava. The two lava cells will move after the player make a move and the goal is to make all 50 moves as fast as possible.
The *SGL2D* code for **The Floor Is Lava** can be found HERE.