Ballr: A 2D Game Generator

players gonna play

Language Reference Manual

Noah Zweben (njz2104)
Jessica Vandebon (jav2162)
Rochelle Jackson (rsj2115)
Frederick Kellison-Linn (fjk2119)

I. Lexical elements:

1. Identifiers

Identifiers are for naming entities, gameboards, colors, and other data types. They are defined as at least one lowercase letter followed by any combination of letters, numbers, and underscores.

2. Reserved Keywords & Symbols

The following case sensitive keywords are reserved for specific purposes and cannot be used as identifiers:

entity	func	**time**	**restart**	**load**
gameboard	**click**	**keypress**	**add**	**remove**
float	int	bool	**><**	**->**
event	clr	pos	mov	if
while	self	color	vec	size
click_pos	init	frame	include	

3. Literals

1. **Integer literals**
   Sequences of one or more digits (ex. 253)

2. **Float Literals**

Sequences of one or more digits containing a '.' with at least one digit before the '.' and at least one digit after the '.' (ex. 1.23)

4. Operators

*, /, %	multiplication, division, modulo
+, -	add, subtract
>, >=, <, <=	inequality operators
==, !=	equal, not equal
=	assignment
!, &&, ||	not, and, or
.	access

5. Delimiters

1. **Parentheses**
   Parentheses are used to enclose arguments in function calls as well as to force precedence orders in expression evaluations.

2. **Commas**
   Commas are used to separate arguments in function calls and to separate values in color and vector data types.

3. **Semicolons**
   Used to terminate a statement.

4. **Curly braces**
   Used to enclose a series of statements in conditional blocks, loops and entity, gameboard and function definition blocks

6. Whitespace

Whitespace is only used to separate tokens.

7. Comments

Only single line comments are allowed and are started with //

II. Data Types

1. Primitive Data Types

Integers: Most numbers will be declared as type int (all numbers relating to position of entities within a gameboard will be implicitly rounded to an int)
Floats: Floating point numbers will be declared as type float, and can be used in mathematical expressions

Booleans: Boolean values will be declared as type bool and can be True or False

2. Non-Primitive Data Types

Color
A `color` is a built in datatype composed of a red (r), green (g), and blue (b) value. Many predefined colors available in the Standard Library _____. The values of the red, green, and blue components are integers between 0 and 255 and are enclosed in () and separated by commas. Color's r,g,b components can be accessed or set using . notation.

Syntax	Example

`color <name> = (r,g,b);`
`//set entity color equal to new color`
`entity.clr = <name>`
`//change color`
`<name>.<r/g/b> = <val>;`
	`color blue = (0,0,255);`
	`//set entity color equal to new color`
	`player.clr = blue;`
	
	`//unnamed color allowed`
	`player.clr = (255,0,255);`
	
	`//change color`
	`player.clr.g = 255;`

Vector
a `vec` is a built in datatype composed of an x and a y value. The values in the `vec` may be constructed with or ints, but everything will be rounded to int. The values of x and y are enclosed in () and separated by a comma. A vector's x and y component can be accused or set using . notation.

Syntax	Example

`vec <name> = (x,y);`
`//move player to vector`
`<entity>.pos = <name>`
 	
`//change vec`
`<name>.<x/y> = <val>;`
	`vec teleport = (100,100);`
	`//move player to vector`
	`player.pos = teleport;`
	
	`//unnamed vector allowed`
	`player.pos = (100,100);`

	`//change vec` `teleport.x = 42;`

Vector Operations

Available operators for vectors are as follows. All operations are component-wise:

Symbol	Name	Definition	Example
+	Vector Addition	vec1 + vec2 = (vec1.x+vec2.x,vec1.y+vec2.y)	vec new = (100,50)+(25,30); //(125,80)
-	Vector Subtraction	vec1 - vec2 = (vec1.x - vec2.x,vec1.y - vec2.y)	vec new = (100,50)-(25,30); //(75,20)
*	Vector Multiplication	vec1 * vec2 = (vec1.x * vec2.x,vec1.y * vec2.y)	vec new = (100,50) * (2,1); //(200,50)
/	Vector Division	vec1 + vec2 = (vec1.x / vec2.x,vec1.y / vec2.y)	vec new

Gameboard

`gameboard` is the data type that represents the current window that a user is playing in and interacting with. Only one gameboard is allowed per file and the name must match the filename.

gameboard.size

The size represents the size of the window the user will interact with and is of type `vector`. The vector encodes information about the playable window's width and height.

gameboard.init

The **init** event is a reserved event keyword that is triggered at the very beginning of the game (see section ? for description of events). Behavior needed to set up the game such as adding players is contained in the gameboard's init event.

Syntax	Example

`gameboard <name> {`
`  size = (width,height);`
`  clr = (r,g,b);`
`  init -> { setup code }`
`}`
	`gameboard <name> {`
	`  size = (200,100);`
	`  clr = (255,255,255);`
	`  init -> add(player(),(100,50));`
	`}`

Entity

An `entity` is a rectangular component within a gameboard. An entity is composed of a color and a size. Users may also define additional variables to be associated with the entity. Event driven behavior can be added to entities. Events are described in the following section. In addition, users may define movement for an entity to set up automatic motion.

Entity.size (required)

All entities must possess a size variable of type `vec`. The size encodes the dimensions of the entity as width and height. It can be both read and set.

Entity.clr (required)

All entities must posses a clr variable of type color. This variable sets the color of the entity when it appears on the gameboard. It can be both read and set.

Entity.pos

Once entities are placed on a gameboard they contain a variable called `pos` of type `vector` encoding the x and y position of the entity center. `entity.pos` can be set or read from.

Entity.init_pos

Entities possess a member called init_pos of type vector which contains the position at which they were initially placed. `entity.init_pos` member is read only, and is useful for setting up parametric motion around a point (see example)

Entity.frame(optional)

Entities frame event is triggered every frame. This allows you to apply frame by frame changes to color or position.

Syntax	Example

`entity <name> {`
`//required`
`  size = (width,height);`
`  clr = (R,G,B);`
`//optional`
`  frame -> <change>`
`//adding non-required variables`
`  <type> <name> = <init. val>;`
`}`
	`entity player {`
	`  size = (10,10);`
	`  clr = (255,255,0);`
	`//parametrically sets up circular motion`
	`  frame -> self.pos = self.init_pos+(10*cos(time),10*sin(time));`
	
	`//adding non-required variables`
	`  int score = 10;`
	`}`

To create an instance of an entity you call it's name using the syntax entity(). For example
obstacle obs = obstacle();
add(obs, (50,30));

To refer to an entity within its own body, use self.

When referring to entities from within functions, you pass the entity name. Then the function checks against every game piece of that type. For example,

player >< obstacle -> remove(obstacle)

III. Functions

1. **Built-in Functions**
   add(entity, vec) - adds an entity at specified position
   remove(entity) - removes from gameboard
   load(gameboard) - loads a different gameboard file, useful for switching between levels
   restart() - restarts gameboard from init

2. **User-Defined Functions**
   Defining a named function: func function_name(args) return return_type
   To call a named function: function_name(args)

IV. Events

1. **<event trigger> -> <behavior> (Event Operator)**
   An `event trigger` is a boolean expression which appears on the left hand side of the '->' symbol. This can be a collision between two objects, keyboard input, and win conditions.

   Any statement to the left of a -> operator is interpreted as an event trigger, and must evaluate to True or False. Named events are useful for composing more complicated events, but not all events must be named. Event triggers get checked every frame in an unspecified order.

   Event driven behavior specific to an entity is defined within the entity block. For example:

```
entity main_character{
      size = (10,10);
      clr = (255,0,0);
      int damage = 0;
      keypress(KEY_UP) -> pos.x+10
```

```
    keypress(KEY_DOWN) -> pos.x-10
    onclick -> add(self,click_pos)
    (self.damage > 100) -> remove(self)
}
```

Built-in Events
keypress(int key): triggered when the user presses a key
self >< <entity>: triggered when the object collides with an entity of type <entity>
onclick: triggered when the user clicks in the screen (with global click_pos storing the location)
frame: triggered once every frame for each object that defines a behavior

V. Control Flow Statements

1. **Conditional Statements:**
   if (<bool>) {
   <expr>
   } elsif (<bool>) {
   <expr>
   } else {
   <expr>
   }

2. **While loops:**
   while (<bool>) {
   <expr>
   }

VI. Program Structure and Scope

All code for a program is to be contained within a single file. This file must have one `gameboard` with the same name as the file.

Within a program file, the globally scoped values are functions and gameboards, and global values defined by the standard library (KEY_UP, etc.) or user. Names of entities can also be referenced globally within the file.

To link in global libraries use the syntax
include <library name>

Within blocks (entities, functions, etc), scope is determined by curly braces (variable shadowing not allowed). Behavior blocks and event expressions declarations can also reference a special local variable ‘self’ which refers to the current entity.

VII. Sample Program

```
entity obstacle {
      clr = blue;
      size = (20,20);
      mov =  myMov;
}

func pos myMov(ent){
      return(ent.pos.x, ent.pos.y + sin(time));
}

entity player{
      size = (20,20);
      clr = red;
      bool has_pickup = false;

      keypress(KEY_RT) -> pos.x++;
      keypress(KEY_LF) -> pos.x--;
      keypress(KEY_UP) -> pos.y++;
      keypress(KEY_DN)-> pos.y--;

      >< obstacle -> restart();
      >< pickup -> {
            self.has_pickup = true;
      }
      >< endzone -> {
            if (self.has_pickup) {
                  load game2;
            }
      }
}

entity pickup {
      clr = yellow;
      size = (20,20);
      >< player {
            remove(self);
      }
}
```

```
entity endzone {
      size = (30,50);
      clr = green;
}

gameboard game1 {
      size = (200,100)
      init {
            int i = 0;
            while (i < 6) {
                  obstacle obs = obstacle();
                  if (i%2) {obs.mov = obs.mov * (1,-1); }
                  add(obs,(20+20*i,50));
                  i += 1;
            }
            add(player(), (10,50));
            add(pickup(), (100,50));
            add(endzone(), (175,50));
      }
};
```