# Ballr: A 2D Game Generator

*players gonna play*

*by*
Noah Zweben (njz2140)
Jessica Vandebon (jav2162)
Rochelle Jackson (rsj2115)
Frederick Kellison-Linn (fjk2119)

# Language Description

The language will be used to build simple 2D games with user-defined environment (bounds, obstacles, tokens), rules and player control. The programmer will be able to place moving or stationary rectangular obstacles and tokens throughout a bounded space which will be displayed in a GUI. They will be able to define how to control player tokens (mouse-clicks, arrows keys), and the rules associated with collisions between players and obstacles, boundaries, and/or special tokens. This will also be useful for creating simple drawing/animation applications.

Ballr abstracts away the event loop that is necessary when writing a game in, say, C++. Instead, the programmer is free to focus on events that may occur during the course of the game: collisions, key presses, and lifetime events being primary examples. This allows the structure of the game to be clearer from a simple glance at the code.

# Language Features

Gameboard blocks:

- Define a particular board for the game (like a single level)
- Include an init function to populate the board with entities (will be run every time the board is loaded).

Entity blocks:

- Define features of all the entities on the gameboard - players, obstacles, special tokens etc
- Required members:
    - scale: (x, y) dimensions
    - clr: color
- Optional members

        - mov: can define automatic x and y motion
        - Entity specific events, such as left_key, right_key

Events & events blocks:

- Describe things that can happen during the game
    - Collisions with other entities
    - Triggers such as mouse clicks, arrow keys
    - Win conditions
- One “events” block per gameboard is required to describe behavior on the occurrence of different events
- Named events can be defined using the “event” keyword
- The events block can be thought of as a listing of the rules of the game. Such as what happens when two things collide, or something moves into a specific area . . .

Reserved Keywords:

**time**: represents the time that has passed since the current level began. Useful for setting up automatic sinusoidal/circular motion.
**restart**: useful for the concept of a “death” during a game. Restarts the gameboard using the init block.
**load <gameboard.name>**: starts a new level, running the init block of a gameboard specified by the gameboard keyword.
**destroy <entity>**: removes entity from gameboard
**entity**: Defines the various elements on the board.
**gameboard**: Defines a particular level in the game (as outlined above) and gives the starting state in the init function.
**event:** Constructor for events. see events & events block section
**add** <entity> **at** <position>: Adds the entity to the gameboard at position
**func**: Signifies a custom function definition.

Operators:

**>< -** collision operator, triggers collision event if the entities applied to have collided
**-> -** do operator attached to events. When an event occurs, such as the left key being pressed, the code to the right of the error gets executed. So in code this would look like:

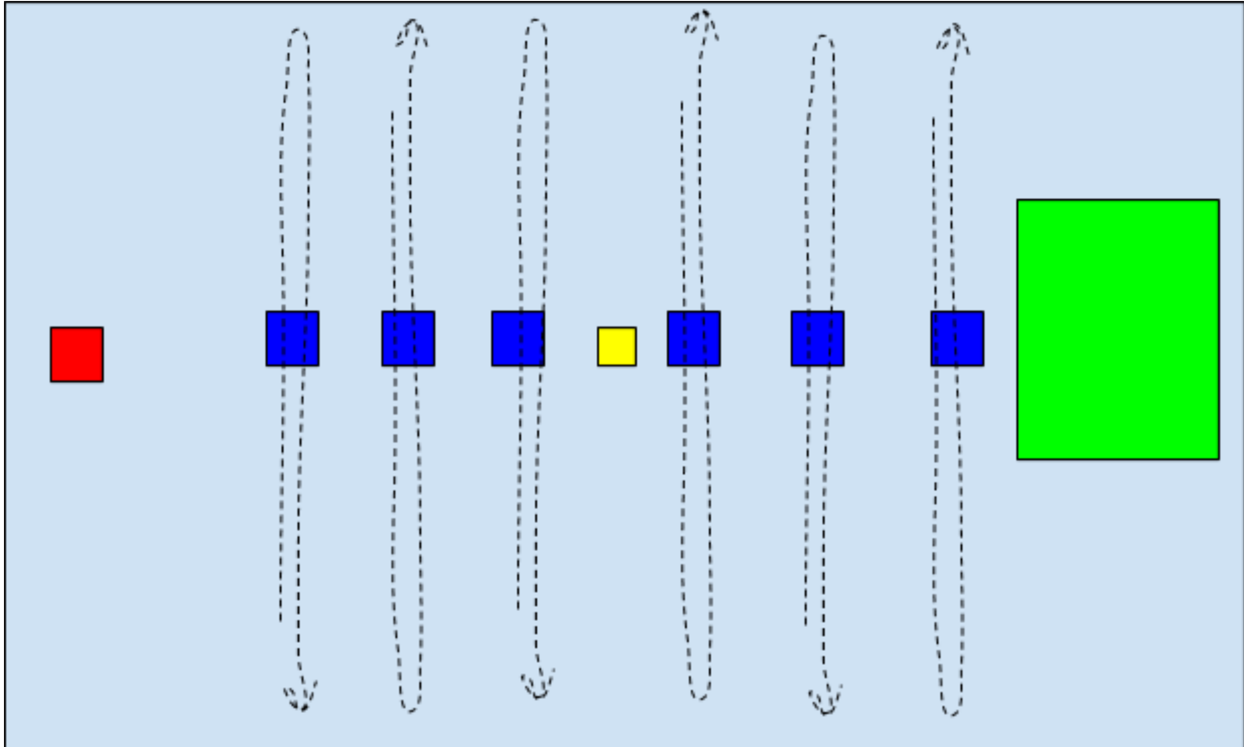
```
left_key -> player.x++
```

Functions:

Global functions can be defined at the top level of the code, and can be used for things such as calculating position, score, or other game elements.

# Sample Program

Our goal for this program is to mimic the game as demonstrated below. The red square (the player) is controlled by the arrow keys.  The blue squares are obstacles which oscillate up and down. The player must dodge the blue obstacles, pick up the yellow token in the center of the game, and then navigate to the greenzone in order to win. (See picture for details)



```
entity obstacle {
      clr = blue;
      scale = (1,1);
}

entity player{
      scale = (1,1);
      right_key -> pos.x++;
      left_key -> pos.x--;
      up_key -> pos.y++;
      down_key -> pos.y--;
      clr = red;
}
```

```
entity pickup {
	clr = yellow;
	scale = (1,1);

}

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

func my_sin(float t) return float {
	return sin(t);
}

gameboard game1{
	size = (200,100)
	init {
		for i in range(6){
			obstacle obs;
			if (i%2){obs.mov.y = my_sin(time);}
			else {obs.mov.y = -1*my_sin(time);}
			add obs at (20+20*i,50);
		}
		add player at (10,50);
		add pickup at (100,50);
		add endzone at (175,50);
	}
};

events {
	player >< obstacle -> restart;
	player >< pickup -> {
		player.has_pickup = true;
		destroy pickup;
	}
	win -> load game2; //loads next level
}

event win {
	(player >< endzone) and player.has_pickup
}
```