Pong and Breakout!

Part 1: *Submission to website:* Tuesday, May 3, 10pm

Part 2: *Submission to website:* Wednesday, May 11, 10pm

Checkoff by LA/TA: Thursday, May 12, 10pm

This lab assumes you have Python 2.7 installed on your machine. Please use the Chrome web browser.

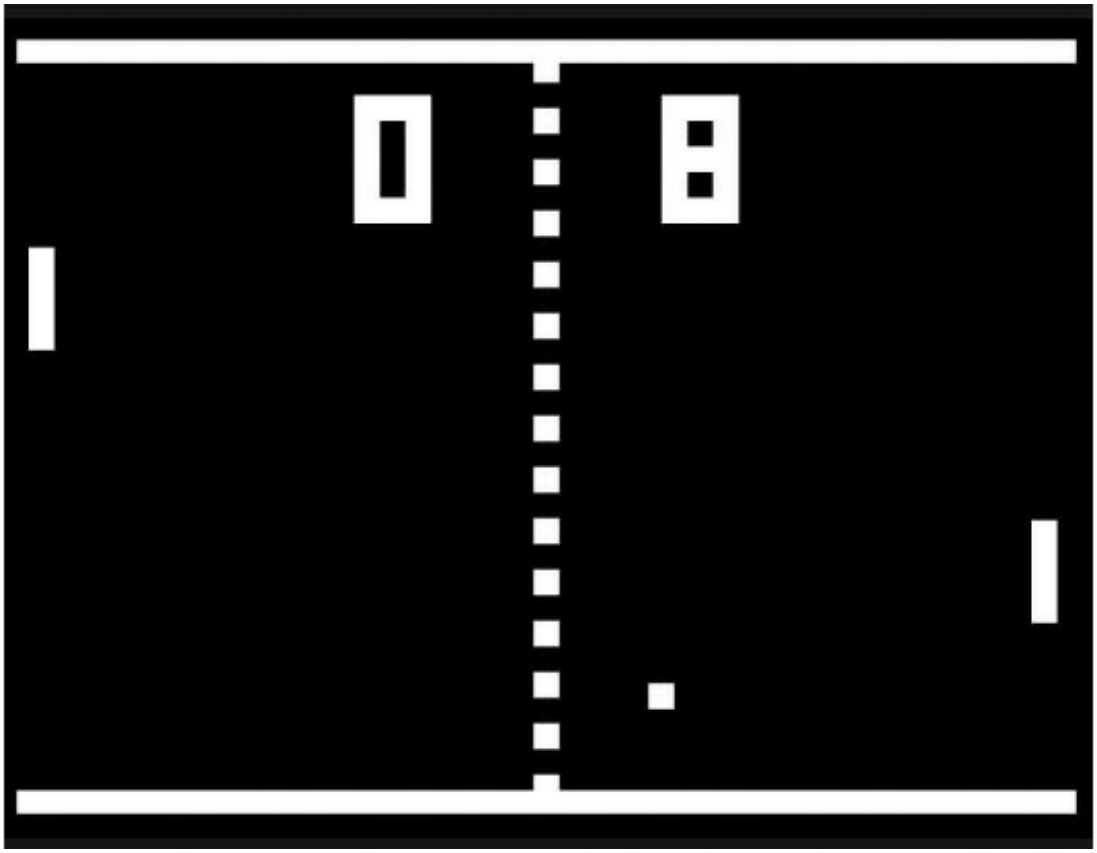
We strongly suggest that you read the entire specification document before writing any code.

Introduction

Video games are one of the most fun things you can do with a computer. Nowadays we have a lot of fancy 3D video games like Fallout 4, but back when we first started making video games, things had to be made a little simpler.

Pong is one of the first video games ever made – and now it's *your* turn to build it!

6.S04pong

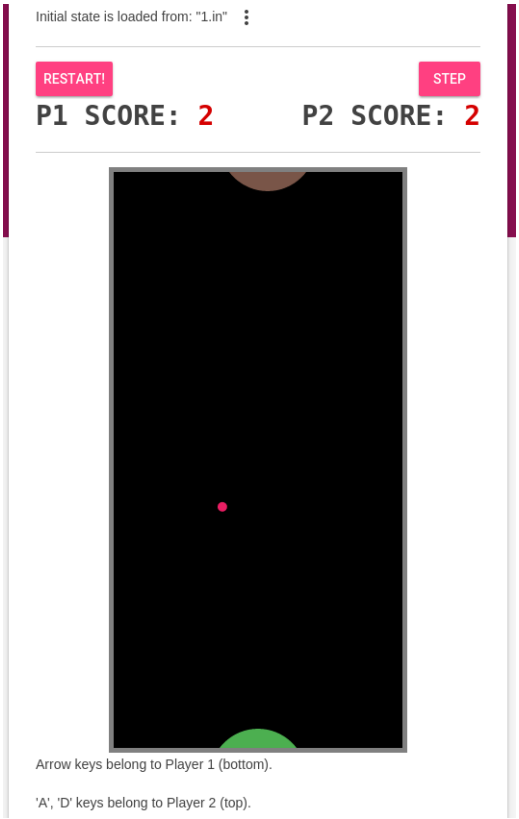


Pong is a simple computerized version of Ping Pong.

There are two players, who each control a paddle. Player 1 uses the left and right arrow keys, while player 2 uses the A and D keys. A ball moves on screen, bouncing off the walls and the paddles. Whenever a player misses the ball with her paddle, the other player gets a point.

In 6.S04pong, there is no way to lose, and the fun never ends! (Note: Alas, playing 6.S04pong is probably not a valid excuse for missing your final exams).

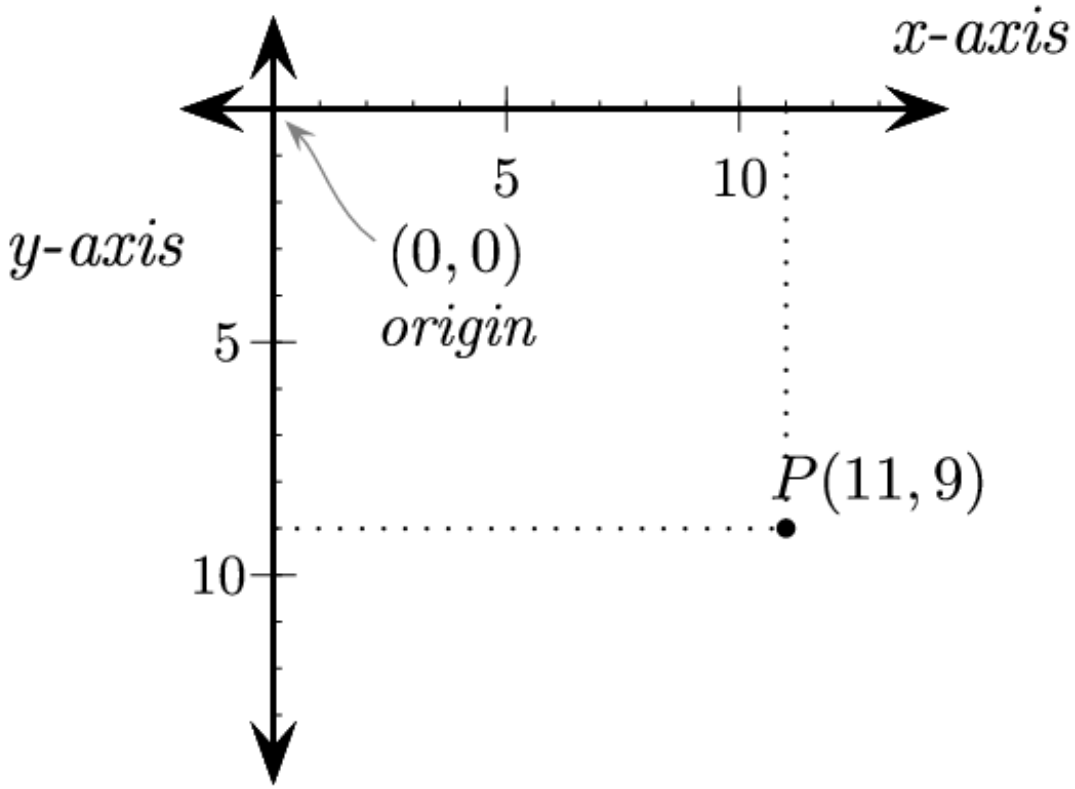
We also deviate from the classic game by arranging the players vertically, with player 1 on the bottom.



The rules are quite simple: - Whenever the ball collides with the top or bottom wall, a point is given to whichever player is *not* defending that edge. - Whenever the ball hits the left or right wall, the ball bounces, as described in the “Collisions” section. - Whenever the ball hits one of the circular paddles, the ball bounces (again, see the “Collisions” section).

The On-Screen Coordinate System

When dealing with computer screens, we define the following coordinate system, with the origin at the top left corner of the image, and the Y values increasing as we move down.



For example, in the grid shown above, point P corresponds to `[11,9]`.

Required Functions

You are responsible for implementing 6.S04pong, which exposes three functions to the user interface:

`init(width, height, ball_position, ball_velocity, blocks)`

`init` initializes any persistent game state you choose to implement and returns `None`. `width` and `height` describe (in pixels) the size of the playing field, while `ball_position` and `ball_velocity` are 2D vectors (of the form `[x,y]`, where `x` and `y` are floats such as `12.3`). `ball_position` gives the location of the ball, while `ball_velocity` explains in which direction it is moving. The last argument, `blocks` is always `[]` for this milestone, and will be used for the second half of this lab.

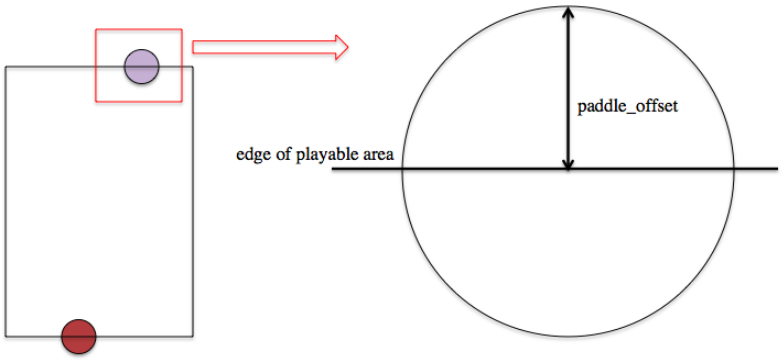
(If you are impatient, and want to start early, `blocks` is of the form `[{'color':'#123456', 'center': [50,60], 'width': 10, 'height': 10}, ...]`.)

NOTE: `init` should completely reset the game, meaning that calling `init` twice should reset the game to the state provided by the latest call to `init`.

`step(time, paddle_1_x, paddle_2_x, paddle_offset,`

`paddle_radius)`

`step` advances the game state by `time` units, handling collisions and other game logic, and returns an integer in `{-1,0,1}`. A return value of `-1` indicates a point is awarded to player 2 (collision with bottom wall), and `1` awards a point to player 1 (collision with top wall). `0` means no one got a point. As arguments, you are given the center `x`-coordinate positions of both player paddles via `paddle_1_x` and `paddle_2_x`. The paddle centers are positioned above and below the playable area by `paddle_offset` pixels, and are circles of radius `paddle_radius`. The top paddle is moved up and the bottom paddle is moved down, as in the picture below.



You should check to see if a collision will happen within the time specified. You will need to implement two types of collisions: a collision between the ball (modelled as a moving **point**) and a line, as well as a collision between the ball and a circle (the players' paddles). If at any time the ball collided with anything (the wall, the paddle, etc.), you should **immediately stop simulating**, then appropriately reflect the ball's velocity, and finally move the ball's position forwards by **.001** times its new velocity, and return.

To avoid any ambiguity, check collisions against the left wall, then the right wall, then the top wall, then the bottom wall, then check for collisions with blocks (not applicable in this milestone), and finally check for collisions with the paddles.

`draw(canvas, paddle_1_x, paddle_2_x, paddle_offset, paddle_radius)`

`draw` returns `None`, and uses `canvas` to communicate a list of shapes to be drawn to the screen. The arguments are equivalent to arguments for the `step` function, except `canvas` (which is used for drawing, as described later in this document).

Vector Math

In this lab you will be asked to work with points and vectors.

Points represent position. A 2D point consists of two coordinates, an `x` coordinate and a `y` coordinate.

Vectors are used to represent velocity. A 2D vector also consists of two coordinates `x` and `y`. Vectors have both a direction and a magnitude. See here for a simple explanation of vector math.

We will be using a few operations on vectors in our calculations for this lab.

Dot Product

`dot(a,b)` denotes the vector dot product between `a` and `b`. For a 2-dimensional vector, it will return a scalar equal to:

```
dot(a,b) = a_x*b_x + a_y*b_y
```

where `a_x, a_y` denote the `x` and `y` components of vector `a`, respectively.

Cross Product

`cross(a,b)` denotes the cross product between `a` and `b`. For a 3-dimensional vector, this will return a vector equal to:

```
cross(a,b) = (a_y*b_z - a_z*b_y, a_z*b_x - a_x*b_z, a_x*b_y - a_y*b_x)
```

...where the tuple notation `()` denotes a vector with 3 components.

For this lab, you will treat your 2-dimensional vectors as 3-dimensional vectors with a z-component of 0. You will only be asked for the z-component of your cross product, that is:

```
cross_z(a,b) = a_x*b_y - a_y*b_x
```

For a more thorough explanation of these operations, please consult this resource.

Normals

A “normal” to a line or a circle is the vector that that goes perpendicular to it. To get the normal to a circle, we take the vector from the center to the point of intersection:

```
n = (p_x - c_x, p_y - c_y)
```

where `p` is the point of intersection with the circle and `c` is the center of the circle, and where the tuple notation `()` denotes a vector with 2 components. Normals are usually *normalized* (divided by their length) to make a direction vector of length 1.

Note that the way we represent points and vectors here is only for notational convenience, and should not necessarily be used in your implementation.

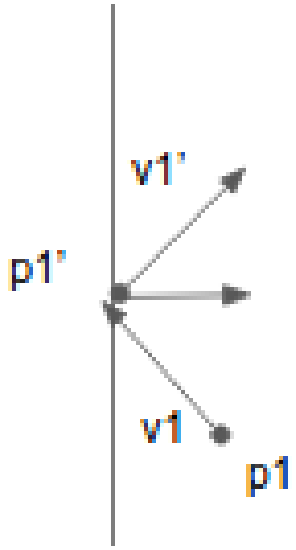
Reflection

What really makes the game interesting is how bouncing off of walls reflects part of the velocity.

In the straightforward case of a collision with a wall, the idea is simple.

If we hit a wall on the right or left, reverse the `x` component of the velocity vector. If we hit a wall on the top or bottom, reverse the `y` component of the velocity vector.

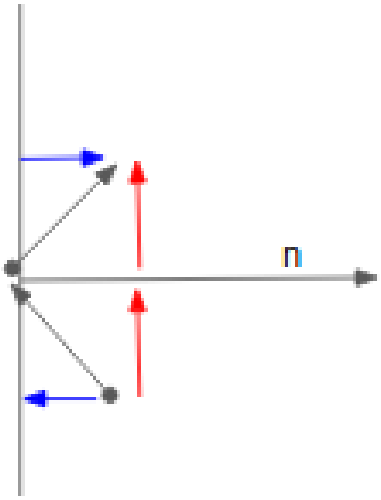
The result should look something like this:



But what about something more complicated, like bouncing off of a sphere?

First, let's generalize the reflection off a wall to work for a line of any orientation.

Suppose we wanted to bounce off a line with a normal `n` as follows. We see that we can achieve the desired reflection by reversing the component of the velocity in the same direction of the normal (shown in blue), leaving the rest of the velocity (shown in red) unchanged.



We will find the formula for the "projection" (the component of the velocity `v` in the direction of the normal `n`) useful:

```
proj(v,n) = dot(v,n) / dot(n,n) * n
```

Then our formula for the bounced velocity is as follows:

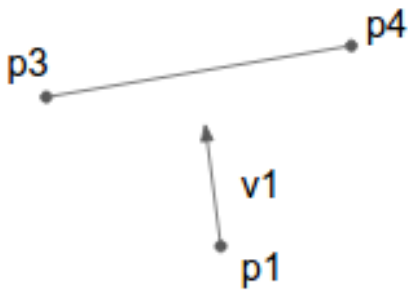
```
v' = ( v - proj(v,n) ) - ( proj(v,n) )
v' = v - 2 * proj(v,n)
```

It turns out that this formula works for reflections off of a circle as well! In the case of a circle, the normal is simply the vector from the center of the circle to the point of intersection.

Line Intersections

Now let's figure out how to intersect the ball with a line segment.

Imagine we had a line segment going from point `p3` to point `p4`, as depicted below. We have our ball at a point `p1`, moving at a velocity `v1`.



How do we determine whether `p1` will eventually intersect the line – and if so, at what time?

To figure it out, we have to use …. math!

We begin by representing both of these structures as lines:

```
v2 = p4 - p3
l1(t) = p1 + t*v
l2(u) = p3 + u*v2
```

An intersection means that the two points are the same, thus

```
l1(t) = l2(u)
p1 + t*v1 = p3 + u*v2
```

Now, we use some vector math to fix our equations

```
cross(v2, p1 + t*v1 ) = cross(v2, p3 + u*v2 )
cross(v2, p1 + t*v1 ) = cross(v2, p3)
t = cross( v2, p3 - p1) / cross( v2, v1 )
```

Likewise for `u`,

```
cross( v1, p1 + t*v1 ) = cross( v1, p3 + u*v2 )
cross( v1, p1 ) = cross(v1, p3 + u*v2 )
u = cross( v1, p1 - p3 ) / cross( v1, v2 )
```

Note that we need only the z component of the cross product in the division (and need not worry about taking the magnitude).

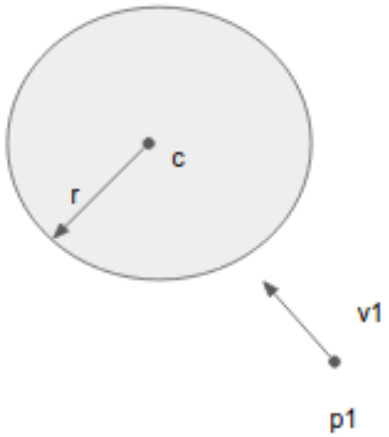
The point will collide at time `t` (calculated above) if `0 < u < 1`, or if `0 < t`. If the denominator of the expression for t (or u) equals zero, there is no collision.

Likewise, the point of intersection will be `p1 + t*v1`.

Circle Intersections

Now let's figure out how to intersect the ball with a circle (like the paddle).

Pretend we have a circle centered at point `c` of radius `r`. Our ball starts at a point `p1` traveling at velocity `v1`.



First we check to see if the ball is already inside of the circle. If it is, there is no collision.

In other words, if `|p1 - c| < r`, there is no collision.

We can now perform some vector analysis to find the intersections.

The following equations represent for the circle and line, respectively.

```
dot(x-c, x-c) = r*r
x(t) = p1 + t * v1
```

Substituting for `x`, we find

```
dot( (p1 -c )+ (t * v1) ) = r*r
dot(p1-c,p1-c) + 2*dot(p1-c, t * v1) + dot(t*v1,t*v1) = r*r
dot(v1,v1) t*t + 2 dot(v, p1 - c ) t + dot(p1 - c,p1 - c) - r*r = 0
```

Thus, defining `a`, `b`, `d` as below, we find two possible values for `t`.

```
a = dot(v1,v1)
```

`b = 2 dot(v1, p1 - c )`
`d = dot(p1 - c,p1 - c) - r*r`
`t = (-b +/- sqrt(b^2-4ad) ) / (2a)` If the argument to that square root is less than zero, there is no collision.

We then take the minimum positive time that solves the above equation. That is the time of intersection.

Likewise, the point of intersection is simply `p1 + t * v1`.

Drawing

Recall the `draw` method. The first argument, `canvas`, represents a blank canvas, and you may draw shapes on it via two methods:

- `canvas.draw_circle(center_x, center_y, radius, color)`
- `canvas.draw_rectangle(upper_left_x, upper_left_y, lower_right_x, lower_right_y, color)`

`color` is a web-safe RGB color string like `"#E91E63"`. The other arguments are self-explanatory. Many nice colors can be found here. You may choose any colors you like for the ball and paddles, but each brick (which will become relevant in the second half of the lab!) specifies a color. A color picker is useful for figuring out the strings for your favorite colors.

For example, `canvas.draw_circle(1, 2, 3, "#FF0000")` draws a red circle of radius 3, centered at `[1,2]`.

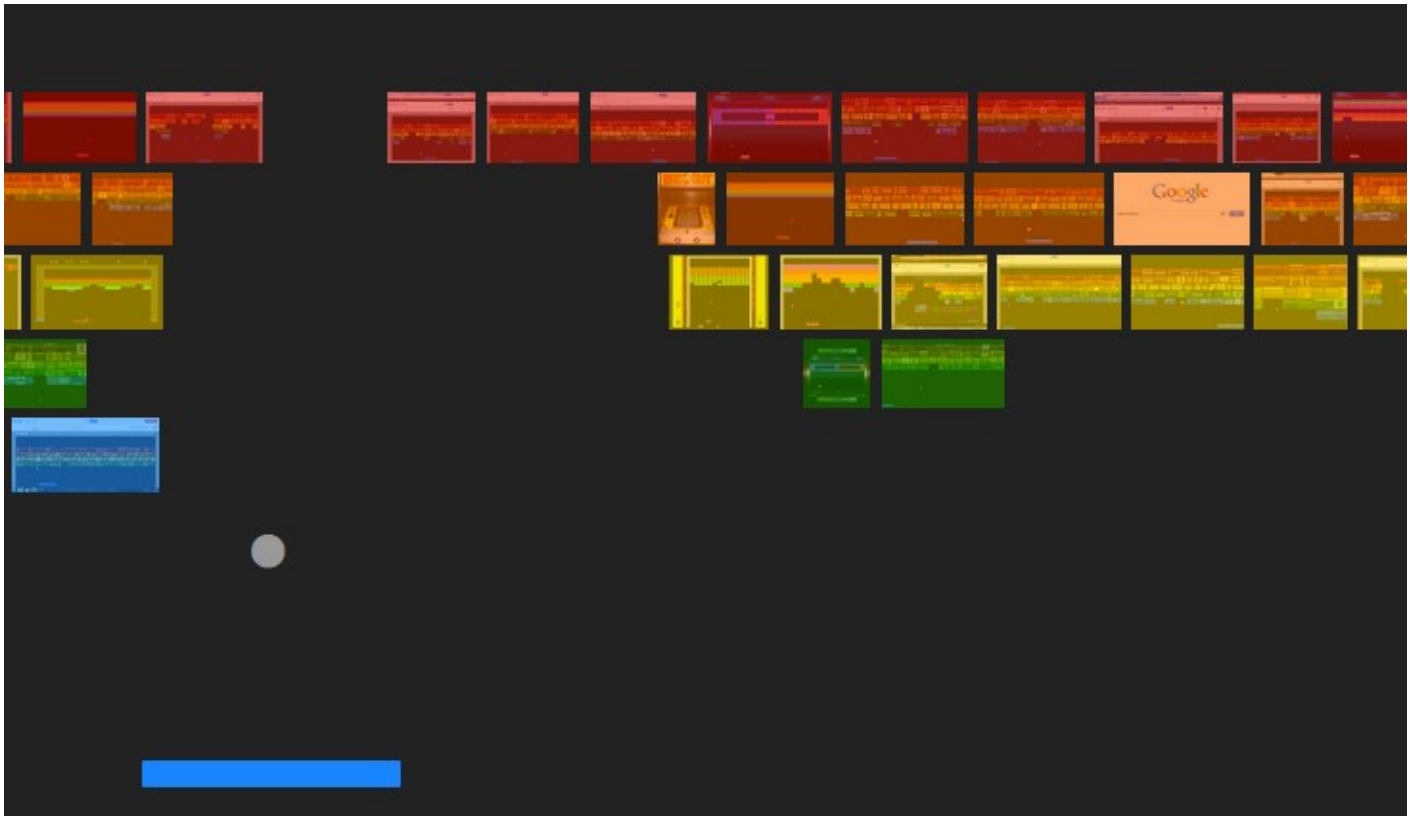
The ball's radius is always 5, and both paddles should have the same radius (`paddle_radius`, which is provided).

Putting Everything Together

Ready? Set? Go.

Part 2 – Breakout

Breakout is another classic (beautifully demonstrated by a Google easter egg).



The second part of this lab asks you to convert your very fine game of Pong to an even better game of Breakout! 6.S04breakout is essentially 6.S04pong with destructible bricks that reflect the ball much like walls do. A brick is destroyed immediately after a ball collides with it.

Recall the `init` method. The final argument `blocks` is a list of dictionaries, each of which encodes a block as follows: `{'color':'#FF0000', 'center':[50,60], 'width':10, 'height':30}`, which encodes a 10x30 red brick centered at `x=50`, `y=60`.

Collisions with a block are handled by collisions with any of the block's 4 edges. To avoid any ambiguity, check the left edge first, then the right edge, then the top edge, and finally the bottom edge.

How to play the game

As usual, you may play the game via `./server.py`, and use a browser (Chrome!) at localhost:8000. Note that you are invited to make your own games by creating your own inputs! Go forth and impress everyone.

Using the GUI to debug your test cases will make your life easier.

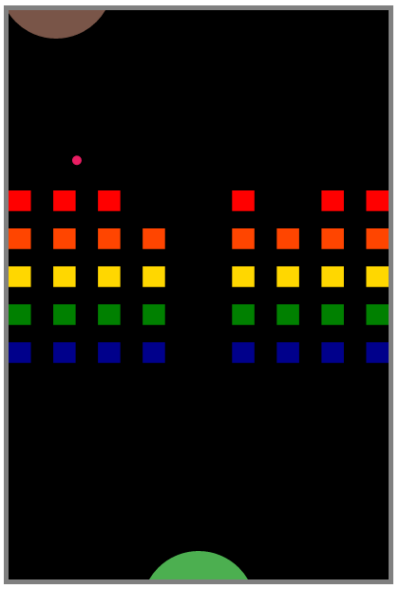
Initial state is loaded from: "11.in"

RESTART!

STEP

P1 SCORE: 1

P2 SCORE: 0



Arrow keys belong to Player 1 (bottom).

'A', 'D' keys belong to Player 2 (top).