

Hyper-literate programming environment with a unique database/causality model.

Flappy Eve

Setup

Draw the game world!

Game menus

Score calculation

Start a new game

Drawing

Player

Obstacles

Game Logic

Obstacles

Flapping the player

Scroll the world

Collision

Flappy Eve

When a player starts the game, we commit a `#world`, a `#player`, and some `#obstacles`. These will keep all of the essential state of the game. All of this information could have been stored on the world, but for clarity we break the important bits of state into objects that they effect.

- The `#world` tracks the distance the player has travelled, the current game screen, and the high score.
- The `#player` stores his current y position and (vertical) velocity.
- The `#obstacles` have their (horizontal) offset and gap widths. We put distance on the world and only keep two obstacles; rather than moving the player through the world, we keep the player stationary and move the world past the player. When an obstacle goes off screen, we will wrap it around, update the placement of its gap, and continue on.

Setup

Add a flappy eve and a world for it to flap in:

```

commit
[ #player #self name: "eve" x: 25 y: 50 velocity: 0 ]
[ #world screen: "menu" frame: 0 distance: 0 best: 0 gravity: -0.061 ]
[ #obstacle gap: 35 offset: 0 ]
[ #obstacle gap: 35 offset: -1 ]
        
```

Next we draw the backdrop of the world. The player and obstacle will be drawn later based on their current state. Throughout the app we use resources from @bhauman's flappy bird demo in clojure. Since none of these things change over time, we commit them once when the player starts the game.

Draw the game world!

```

search
world = [#world]

commit @browser
world <- [#div style: [user-select: "none" -webkit-user-select: "none" -moz-user-select: "none"] children:
[ #svg #game-window viewBox: "10 0 80 100", width: 480 children:
[ #rect x: 0 y: 0 width: 100 height: 53 fill: "rgb(112, 197, 206)" sort:
        
```

The screenshot shows the game's 'Game Over' screen. At the top, a blue hand icon is next to the text 'Game Over :('. Below that, the score 'Score 4' and 'Best 0' are displayed in a large, bold font. At the bottom, the text 'Click to play again!' is centered. The background features a stylized cityscape with green buildings and a blue sky with a few clouds. Two green pipes are visible on either side of the screen, and a green ground strip is at the bottom.

- **State database.** All state is in a database of records. Program fragments read and write to this database, which also encapsulates errors. In this fashion, complex programs are build by composing simple processes that read/write to the database. (Analogous, in some ways, to a spreadsheet.)
- **Causality tracking.** Database enables system to track causality—what led to what. (This design brings Realtalk to mind.)
- **Inspect output** (e.g. HTML) to see what might have created it.
- **Hyperliterate programming.** Code is not just embedded in prose, but a complex program takes the form of navigable hierarchical prose (i.e. a book). Code view is dynamic: select which parts of the program you want to see.
- **Inline errors**
- **Inline data.** Easy inline data visualization (notebook style), with some simple widgets, like bar graphs.