Game Engine

Version 7.3

May 29, 2019

This is a game engine built for rapid prototyping and education.

In this engine, a game is a collection of entities. This means that ultimatley, you're going to create a game by running a command like this:

```
(start-game entity1
            entity2
            entity3)
```

Naturally, you'll need to build (or import) those entities. This package provides a DSL for specifying the components that define those entities.

Here's a full example:

```
(require game-engine
         game-engine-demos-common)

(define WIDTH  640)
(define HEIGHT 480)

(define bg-entity
  (sprite->entity (rectangle WIDTH HEIGHT "solid" "black")
                  #:name     "bg"
                  #:position (posn 0 0)))

(define spaceship-entity
  (sprite->entity (circle 20 "solid" "red")
                  #:name       "ship"
                  #:position   (posn 100 100)
                  #:components (key-movement 5)))
```

```
(start-game (instructions WIDTH HEIGHT "Use arrow keys to move")
            spaceship-entity
            bg-entity)
```

Conceptually speaking, I like to think of entities as a combination of two things: art and game logic. The art is specified by the sprite; the game logic is specified by the collection of components.

1 Sprites

The sprite->entity function is the powertool for converting from an image to an entity, allowing you to specify components in the process.

```
(sprite->entity imgs
                #:name name
                #:position position
                #:components components ...) → entity?
  imgs : (or/c image? (listof image?))
  name : string?
  position : posn?
  components : component?
```

Converts *imgs* (which may be a single image or a list of images) into an entity. You may also specify a name, position, and any number of components

```
(change-sprite sprite-or-func) → func?
  sprite-or-func : (or image? func?)
```

This is a function that allows the sprite of an entity to be changed. If sprite-or-func is a image, then the entity's sprite will be changed to the image. If sprite-or-func is an function, then the entity's sprite will be changed to what is returned by the function.

1.1 Sheets

A sheet is an image made up of multiple images, where each image within the sheet can be used for animating the sprite.

```
(sheet->sprite sheet
               #:row r
               #:column c
               #:row-number rnum
               #:speed delay)    → sprite?
  sheet : image?
  r : integer?
  c : integer?
  rnum : integer?
  delay : integer?
```

Divides the sheet into r rows and c columns. This function will then return a sprite that will display each of the rnum row's images with a delay between each image of delay ticks.

```
(sheet->rainbow-hue-sheet sheet) → sheet?
  sheet : image?
(sheet->rainbow-tint-sheet sheet) → sheet?
  sheet : image?
```

Converts a 1 row animation or single image to 12 rows with shifted hue or tint

2 Components

```
(component? x) → boolean?
  x : any/c
```

There is no data structure called "component". A component is anything that has been previously registered with the engine by calling `new-component`.

Conceptually speaking, a component is 1) some struct, and 2) an update function.

If you're just now embarking on your journey to learn about game design, I would recommend sticking to the components provided by others. This engine is structured so that you don't need to worry too much about how the components work or how to make new components.

NOTE: Components can also be added, removed, and updated during runtime.

That said, here's another function for creating new components (which you can also safely ignore if you're just beginning).

```
(new-component struct? update) → void?
  struct? : (-> any/c boolean?)
  update : (-> game? entity? component? entity?)
```

To register a new component, you must use this function. You must provide two functions. 1) a way of recognizing your component, 2) a handler function that the engine will call on every tick, for every entity that has this component.

This handler function returns an entity, which will replace the entity that has the component. The handler function also receives the game as an input. In effect, this means your handler function has read-only access to the entire game state. And it has "write" access to the entity possessing the component.

I would recommend looking at some of the existing components (i.e. `key-movement`) if you're going to make new components of your own.

2.1 Pre-built Components

These components are design to allow you to add behaviour to entities with as little effort as possible. This engine is designed for rapid prototyping – which means that you should be able to experiment with a lot of different game design ideas very quickly. You do that by using pre-built components as the fundamental building blocks for your game.

```
(key-movement speed) → component?
  speed : integer?
```

An entity with this component will move when you press the arrow keys. The speed parameter changes how quickly the entity moves.

Technically speaking, this component updates the entity's `posn` component whenever the arrow keys are pressed.

Example:

```
(define spaceship-entity
  (sprite->entity (circle 20 "solid" "red")
                  #:name       "ship"
                  #:position   (posn 100 100)
                  #:components (key-movement 5)))
```

```
(on-key key func) → func?
  key : symbol?
  func : func?
```

Will call a handler function whenever the specified key is pressed.

Example:

```
(define shooter-entity
  (sprite->entity (rectangle 20 40 "solid" "black")
                  #:name        "shooter"
                  #:position    (posn 100 100)
                  #:components  (on-key 'space (spawn bullet-
entity))))
```

```
(posn x y) → component?
  x : integer?
  y : integer?
```

Determines where the entity will render. If you're using `sprite->entity`, then you should specify this with the `#:name` keyword, not the `#:components` keyword

Example:

```
(define spaceship-entity
  (sprite->entity (circle 20 "solid" "red")
                  #:name       "ship"
                  #:position   (posn 100 100)
                  #:components (key-movement 5)))
```

```
(after-time ticks fun) → component?
  ticks : integer?
  fun : (-> game? entity? component? entity?)
```

Runs a handler function after the specified number of ticks and then all after-time components are removed from the entity

Example:

```
(define bullet
  (sprite->entity bullet-sprite
                  #:position   (posn 100 100)
                  #:name       "bullet"
                  #:components (every-tick (move-left #:speed 5))
                               (after-time 50     die)))
```

```
(spawner sprite amount) → component?
  sprite : entity?
  amount : integer?
```

Spawns a sprite every amount ticks. `spawner` uses the sprite's position component as the relative spawn location to the main entity

```
(on-collide name fun) → component?
  name : string?
  fun : (-> game? entity? component?)
```

Runs a handler function when this entity touches the named entity

Example:

```
(define spaceship-entity
  (sprite->entity spaceship-sprite
                  #:name       "ship"
                  #:position   (posn 100 100)
                  #:components (key-movement 5)
                               (on-collide "ore"    (change-speed-
by 1))
                               (on-collide "enemy"  die)
                               (on-collide "bullet" die)))
```

```
(every-tick func) → component?
  func : (-> game? entity? component?)
```

Runs a handler function every tick in game

Example:

```
(define bullet
  (sprite->entity bullet-sprite
                  #:position   (posn 100 100)
                  #:name       "bullet"
                  #:components (every-tick (move-left #:speed 5))
                               (after-time 50     die)))
```

```
(circle-collider radius) → component?
  radius : number?
```

Changes out the default rect-collider for a circle collider with the specified radius

Example:

```
(define circle-entity
  (sprite->entity (circle 20 "solid" "red")
                  #:name       "ship"
                  #:position   (posn 100 100)
                  #:components (key-movement 5)
                               (circle-collider 20)))
```

```
(health amount) → component?
  amount : integer?
```

Kills this entity when (if) the health reaches 0

Example:

```
(define spaceship-entity
  (sprite->entity spaceship-sprite
                  #:name       "ship"
                  #:position   (posn 100 100)
                  #:components (health 5)
                               (on-collide "bullet" (change-
health -1))))
```

```
(physical-collider) → component?
```

Two entities with physical colliders will not pass through each other. This overrides things like key-handler.

```
(detect-collide entity-name1
                entity-name2
                func)        → component?
  entity-name1 : string?
  entity-name2 : string?
  func : func?
```

Runs a handler function whenever the two named entities collide

Example:

```
(define score-entity
  (sprite->entity score-sprite
                  #:name        "score"
                  #:position    (posn 0 0)
                  #:components  (detect-collide "player" "coin" (add-
to-score))))
```

```
(on-edge edge #:offset off func) → component?
  edge : symbol?
  off : integer?
  func : func?
```

Runs a handler function whenever entity collides with specified edge. Possible values for edge are [’left, ’right, ’top, ’bottom].

Example:

```
(define player-entity
  (sprite->entity player-sprite
                  #:name       "player"
                  #:position   (posn 100 100)
                  #:components (on-edge 'right (win))))
```

```
(detect-edge name edge func) → component?
  name : string?
  edge : symbol?
  func : func?
```

Runs a handler function whenvever the named entity collides with specified edge. Possible values for edge are [’left, ’right, ’top, ’bottom].

Example:

```
(define bg-entity
  (sprite-entity background-sprite
                 #:name        "bg"
                 #:position    (posn 0 0)
                 #:components  (detect-edge "player" 'right (change-
background))))
```

(stop-on-edge *edges*) → component?
  edges : symbols?

Prevents the entity from moving off screen along specified edges. Possible values for edge are ['left, 'right, 'top, 'bottom]. There can be any number of specified edges. If no parameters are passed in, default will be all edges.

Example:

```
(define player-entity
  (sprite-entity player-sprite
                 #:name       "player"
                 #:position   (posn 100 100)
                 #:components (stop-on-edge)))
```

(wrap-around *mode*) → component?
  mode : symbol?

Moves entity to other edge when it moves off screen along specified modes. Possible values for mode are ['left-right, 'top-bottom]. There can be any number of modes. If no parameters are passed in, default will be both modes

Example:

```
(define player-entity
  (sprite->entity player-sprite
                  #:name       "player"
                  #:position   (posn 100 100)
                  #:components (wrap-around)))
```

(rotation-style *mode*) → component?
  mode : symbol?

Only usable with entites with (move). Will flip the sprite image/animation left-right when the entity has direction between 90-270. (rotation-style) assumes that the sprite is drawn facing right. Possible values for mode are 'left-right and 'face-direction. 'left-right

will flip the sprite horizontally whenever direction is between 90 and 270. 'face-direction will rotate the sprite according to the direction.

Example:

```
(define snake-entity
  (sprite->entity snake-sprite
                  #:name      "snake"
                  #:position  (posn 100 100)
                  #:component (speed 5)
                              (direction 180)
                              (every-tick (move))
                              (do-every 10 (random-
direction 0 360))
                              (rotation-style 'face-direction)))
```

]

```
(follow name interval) → component?
  name : string?
  interval : number?
```

Will update the direction component of the entity to point towards the named entity every interval ticks. If interval is not specified, the default will be to set it to 1. `(follow)` does not move the entity. Use `(move)` to move the entity

3 Functions

In addition to components, we also include various pre-built functions to allow for quick and easy prototyping.

```
(move) → func?
```

Should be used with components `(speed)` and `(direction)`. The direction component determines where the entity moves and the speed component determines how quickly the entity moves.

Example:

```
(define bullet
  (sprite->entity bullet-sprite
                  #:position   (posn 100 100)
                  #:name       "bullet"
                  #:components (speed 5)
                               (direction 180)
                               (every-tick (move))))
```

```
(move-right #:speed spd) → func?
  spd : integer?
(move-down #:speed spd) → func?
  spd : integer?
(move-left #:speed spd) → func?
  spd : integer?
(move-up #:speed spd) → func?
  spd : integer?
```

Will move the entity with a speed of spd and a direction of 0, 90, 180, 270 respectively. The speed and direction cannot be changed once set.

Example:

```
(define bullet
  (sprite->entity bullet-sprite
                  #:name       "bullet"
                  #:position   (posn 100 100)
                  #:component  (every-tick (move-
left #:speed 5))))
```

```
(move-random #:speed spd) → func?
  spd : integer?
```

Will randomly call `(move-left)`, `(move-right)`, `(move-up)`, or `(move-down)` and pass in the speed parameter to the chosen function call

Example:

```
(define random-bullet
  (sprite->entity bullet-sprite
                  #:name      "random"
                  #:position  (posn 100 100)
                  #:component (every-tick (move-
random #:speed 5))))
```

```
(move-up-and-down #:min small
                  #:max large
                  #:speed spd) → func?
  small : integer?
  large : integer?
  spd : integer?
```

Moves the entity up and down within a boundary set by small and large. The speed parameter determines how fast the entity moves

Example:

```
(define goalie-entity
  (sprite->entity goalie-sprite
                  #:name       "goalie"
                  #:position   (posn 500 300)
                  #:component  (every-tick (move-up-and-
down #:min 150 #:max 450 #:speed 5))))
```

```
(spin #:speed spd) → func?
  spd : integer?
```

Will rotate the entity, and spd will determine how quickly the entity spins

Example:

```
(define spinner-entity
  (sprite->entity spinner-sprite
                  #:name      "spinner"
                  #:position  (posn 300 300)
                  #:component (every-tick (spin #:speed 2))))
```

```
(go-to x y) → func?
  x : integer?
  y : integer?
```

Will change posn of entity to `(posn x y)`

Example:

```
(define teleporter-entity
  (sprite->entity teleporter-sprite
                  #:name      "teleporter"
                  #:position  (posn 500 500)
                  #:component (key-movement 5)
                              (do-every 30 (go-to 500 500))))
```

```
(go-to-random min-x max-x min-y max-y) → func?
  min-x : integer?
  max-x : integer?
  min-y : integer?
  max-y : integer?
```

Will change posn of entity to a random posn, with the bounds for the x-coordinate being min-x to max-y, and the bounds for the y-coordinate being min-y to max-y

Example:

```
(define coin-entity
  (sprite->entity coin-sprite
                  #:name      "coin"
                  #:position  (posn 300 300)
                  #:component (on-collide "player" (go-to-
random 0 600 0 600))))
```

```
(go-to-pos pos #:offset offset) → func?
  pos : symbol?
  offset : integer?
(go-to-pos-inside pos #:offset offset) → func?
  pos : symbol?
  offset : integer?
```

Will move the entity to somewhere along the edge of the screen. Possible values for pos: ['left, 'right, 'top, 'bottom, 'top-left, 'top-right, 'bottom-left, 'bottom-right, 'left-center, 'right-center, 'top-center, 'bottom-center, 'center]. `(go-to-pos)` will move the entity's

center to the edge, while `(go-to-pos-inside)` will keep the entity completely inside the screen.

offset will move the "edge" of the screen. A positive offset value will always move the edge to the right or down. A negative offset value will do the opposite. offset only applies for 'left, 'right, 'top, and 'bottom.

Example:

```
(define player-entity
  (sprite->entity player-sprite
                  #:name      "player"
                  #:position  (posn 100 100)
                  #:component (key-movement 5)
                              (on-collide "enemy" (go-to-
pos 'left))))
```

```
(respawn edge #:offset offset) → func?
  edge : symbol?
  offset : integer?
```

Will move the center of the entity to the specified edge of the screen. The entity will be placed somewhere along the edge randomly. Possible values for edge: ['left, 'right, 'top, 'bottom]

offset will move the "edge" of the screen. A positive offset value will always move the edge to the right or down. A negative offset value will do the opposite. offset only applies for 'left, 'right, 'top, and 'bottom.

Example:

```
(define enemy-entity
  (sprite->entity enemy-sprite
                  #:name      "enemy"
                  #:position  (posn 100 100)
                  #:component (speed 5)
                              (direction 180)
                              (every-tick (move))
                              (on-edge 'left (respawn 'right))))
```

```
(set-speed amount) → func?
  amount : integer?
(set-player-speed amount) → func?
  amount : integer?
(set-direction amount) → func?
```

```
  amount : integer?
(set-counter amount) → func?
  amount : integer?
```

Change the specified component of the entity to amount. `(set-speed)` should be used with entities not controlled by the user (should not have `key-movement`). `(set-player-speed)` should be used with entities controlled by the user (should have `key-movement`)

Example:

```
(define player-entity
  (sprite->entity player-entity
                  #:name      "player"
                  #:position  (posn 100 100)
                  #:component (key-movement 5)
                              (on-collide "speed-entity" (set-
player-speed 10))))
```

```
(random-direction min max) → func?
  min : integer?
  max : integer?
(random-speed min max) → func?
  min : integer?
  max : integer?
```

Change the speed or direction component of the entity to a randomly choosen value between min and max

Example:

```
(define wandering-entity
  (sprite->entity wandering-sprite
                  #:name      "wandering"
                  #:position  (posn 600 300)
                  #:component (speed 3)
                              (direction 180)
                              (every-tick (move))
                              (do-every 15 (random-
direction 90 270))))
```

```
(change-ai-speed-by inc) → func?
  inc : integer?
(change-speed-by inc) → func?
  inc : integer?
(change-direction-by inc) → func?
```

```
  inc : integer?
(change-counter-by inc) → func?
  inc : integer?
```

Increase the specified component of the entity by inc. The specified component can be lowered by having inc be negative. `(change-ai-speed-by)` should be used with entities not controlled by the user (should not have `key-movement`). `(change-speed-by)` should be used with entities controlled by the user (should have `key-movement`)

Example:

```
(define wolf-entity
  (sprite->entity wolf-sprite
                  #:name      "wolf"
                  #:position (posn 0 0)
                  #:component (key-movement 3)
                              (on-collide "sheep" (change-speed-
by 1))))
```

```
(spawn sprite) → func?
  sprite : entity?
```

Spawns a sprite at current position of the entity. The sprite's posn component will be used as the offset from the current position of the entity. `(spawn)` will automatically adjust for `rotation-style`

Example:

```
(define gun-entity
  (sprite->entity gun-sprite
                  #:name      "gun"
                  #:position  (posn 100 100)
                  #:component (on-key 'space (spawn bullet-
entity))))
```

```
(point-to name) → func?
  name : string?
```

Sets the direction component of the entity to point towards the named entity. `(point-to)` does not rotate the sprite of the entity

Example:

```
(define missile-entity
```

```
(sprite->entity missile-sprite
                #:name      "missile"
                #:position  (posn 0 0)
                #:component (do-every 20 (point-to "target"))))
```