HC Documentation

Release 0.1-1

Matthias Richter

Nov 19, 2017

Contents

HC is a Lua module to simplify one important aspect in computer games: Collision detection.

It can detect collisions between arbitrary positioned and rotated shapes. Built-in shapes are points, circles and polygons. Any non-intersecting polygons are supported, even concave ones. You can add other types of shapes if you need them.

The main interface is simple:

1. Set up your scene,
2. Check for collisions,
3. React to collisions.

CHAPTER 1

First steps

This is an example on how to use HC. One shape will stick to the mouse position, while the other will stay in the same place:

```
HC = require 'HC'

-- array to hold collision messages
local text = {}

function love.load()
    -- add a rectangle to the scene
    rect = HC.rectangle(200,400,400,20)

    -- add a circle to the scene
    mouse = HC.circle(400,300,20)
    mouse:moveTo(love.mouse.getPosition())
end

function love.update(dt)
    -- move circle to mouse position
    mouse:moveTo(love.mouse.getPosition())

    -- rotate rectangle
    rect:rotate(dt)

    -- check for collisions
    for shape, delta in pairs(HC.collisions(mouse)) do
        text[#text+1] = string.format("Colliding. Separating vector = (%s,%s)",
                                      delta.x, delta.y)
    end

    while #text > 40 do
        table.remove(text, 1)
    end
end
```

```
function love.draw()
    -- print messages
    for i = 1,#text do
        love.graphics.setColor(255,255,255, 255 - (i-1) * 6)
        love.graphics.print(text[#text - (i-1)], 10, i * 15)
    end

    -- shapes can be drawn to the screen
    love.graphics.setColor(255,255,255)
    rect:draw('fill')
    mouse:draw('fill')
end
```

CHAPTER 2

Get HC

You can download the latest packaged version as zip- or tar-archive directly from github.

You can also have a look at the sourcecode online here.

If you use the Git command line client, you can clone the repository by running:

```
git clone git://github.com/vrld/HC.git
```

Once done, you can check for updates by running:

```
git pull
```

from inside the directory.

CHAPTER 3

Read on

3.1 Reference

HC is composed of several parts. Most of the time, you will only have to deal with the main module and the *Shapes* sub-module, but the other modules are at your disposal if you need them.

3.1.1 Main Module

```
HC = require 'HC'
```

The purpose of the main modules is to connect shapes with the spatial hash – a data structure to quickly look up neighboring shapes – and to provide utilities to tell which shapes intersect (collide) with each other.

Most of the time, HC will be run as a singleton; you can, however, also create several instances, that each hold their own little worlds.

Initialization

HC**.new**([*cell_size* = *100*])

Arguments

- **cell_size** (*number*) – Resolution of the internal search structure (optional).

Returns Collider instance.

Creates a new collider instance that holds a separate spatial hash. Collider instances carry the same methods as the main module. The only difference is that function calls must use the colon-syntax (see below).

Useful if you want to maintain several collision layers or several separate game worlds.

The `cell_size` somewhat governs the performance of *HC.neighbors()* and *HC.collisions()*. How this parameter affects the performance depends on how many shapes there are, how big these shapes are and (somewhat) how the shapes are distributed. A rule of thumb would be to set `cell_size` to be about four times the size of the

average object. Or just leave it as is and worry about it only if you run into performance problems that can be traced back to the spatial hash.

Example:

```
collider = HC.new(150) -- create separate world

-- method calls with colon syntax
ball = collider:circle(100,100,20)
rect = collider:rectangle(110,90,20,100)

for shape, delta in pairs(collider:collisions(ball)) do
    shape:move(delta.x, delta.y)
end
```

HC.**resetHash**([*cell_size = 100*])

Arguments

- **cell_size** (*number*) – Resolution of the internal search structure (optional).

Reset the internal search structure, the spatial hash. This clears *all* shapes that were registered beforehand, meaning that HC will not be able to find any collisions with those shapes anymore.

Example:

```
function new_stage()
    actors = {} -- clear the stage on our side
    HC.resetHash() -- as well as on HC's side
end
```

Shapes

See also the *HC.shapes* sub-module.

Note: HC will only keep weak references to the shapes you add to the world. This means that if you don't store the shapes elsewhere, the garbage collector will eventually come around and remove these shapes.See also this issue on github.

HC.**rectangle**(*x*, *y*, *w*, *h*)

Arguments

- **x,y** (*numbers*) – Upper left corner of the rectangle.
- **w,h** (*numbers*) – Width and height of the rectangle.

Returns The rectangle *Shape()* added to the scene.

Add a rectangle shape to the scene.

Note: *Shape()* transformations, e.g. `Shape.moveTo()` and `Shape.rotate()` will be with respect to the *center, not* the upper left corner of the rectangle!

Example:

```
rect = HC.rectangle(100, 120, 200, 40)
rect:rotate(23)
```

HC.**polygon**(*x1*, *y1*, ..., *xn*, *yn*)

Arguments

- **x1,y1,...,xn,yn** (*numbers*) – The corners of the polygon. At least three corners that do not lie on a straight line are required.

Returns The polygon *Shape()* added to the scene.

Add a polygon to the scene. Any non-self-intersection polygon will work. The polygon will be closed; the first and the last point do not need to be the same.

Note: If three consecutive points lie on a line, the middle point will be discarded. This means you cannot construct polygon shapes that are lines.

Note: *Shape()* transformations, e.g. `Shape.moveTo()` and `Shape.rotate()` will be with respect to the center of the polygon.

Example:

```
shape = HC.polygon(10,10, 40,50, 70,10, 40,30)
shape:move(42, 5)
```

HC.**circle**(*cx*, *cy*, *radius*)

Arguments

- **cx,cy** (*numbers*) – Center of the circle.
- **radius** (*number*) – Radius of the circle.

Returns The circle *Shape()* added to the scene.

Add a circle shape to the scene.

Example:

```
circle = HC.circle(400, 300, 100)
```

HC.**point**(*x*, *y*)

Arguments

- **x, y** (*numbers*) – Position of the point.

Returns The point *Shape()* added to the scene.

Add a point shape to the scene.

Point shapes are most useful for bullets and such, because detecting collisions between a point and any other shape is a little faster than detecting collision between two non-point shapes. In case of a collision, the separating vector will not be valid.

Example:

```
bullets[#bulltes+1] = HC.point(player.pos.x, player.pos.y)
```

HC.**register**(*shape*)

Arguments

- **shape** (Shape) – The *Shape()* to add to the spatial hash.

Add a shape to the bookkeeping system. *HC.neighbors()* and Hc.collisions() works only with registered shapes. You don't need to (and should not) register any shapes created with the above functions.

Overwrites Shape.move(), Shape.rotate(), and Shape.scale() with versions that update the *HC.spatialhash*.

This function is mostly only useful if you provide a custom shape. See *Custom Shapes*.

HC.**remove**(*shape*)

Arguments

- **shape** (Shape) – The *Shape()* to remove from the spatial hash.

Remove a shape to the bookkeeping system.

Warning: This will also invalidate the functions Shape.move(), Shape.rotate(), and Shape.scale(). Make sure you delete the shape from your own actor list(s).

Example:

```
for i = #bullets,1,-1 do
    if bullets[i]:collidesWith(player)
        player:takeDamage()

        HC.remove(bullets[i]) -- remove bullet from HC
        table.remove(bullets, i) -- remove bullet from own actor list
    end
end
```

Collision Detection

HC.**collisions**(*shape*)

Arguments

- **shape** (Shape) – Query shape.

Returns Table of colliding shapes and separating vectors.

Get shapes that are colliding with shape and the vector to separate the shapes. The separating vector points away from shape.

The table is a *set*, meaning that the shapes are stored in *keys* of the table. The *values* are the separating vector. You can iterate over the shapes using pairs (see example).

Example:

```
local collisions = HC.collisions(shape)
for other, separating_vector in pairs(collisions)
    shape:move(-separating_vector.x/2, -separating_vector.y/2)
    other:move( separating_vector.x/2,  separating_vector.y/2)
end
```

HC.**neighbors**(*shape*)

Arguments

- **shape** (Shape) – Query shape.

Returns Table of neighboring shapes, where the keys of the table are the shape.

Get other shapes in that are close to `shape`. The table is a *set*, meaning that the shapes are stored in *keys* of the table. You can iterate over the shapes using `pairs` (see example).

Note: The result depends on the size and position of `shape` as well as the grid size of the spatial hash: *HC.neighbors()* returns the shapes that are in the same cell(s) as `shape`.

Example:

```
local candidates = HC.neighbors(shape)
for other in pairs(candidates)
    local collides, dx, dy = shape:collidesWith(other)
    if collides then
        other:move(dx, dy)
    end
end
```

HC.**hash**

Reference to the `SpatialHash()` instance.

3.1.2 HC.shapes

```
shapes = require 'HC.shapes'
```

Shape classes with collision detection methods.

This module defines methods to move, rotate and draw shapes created with the main module.

As each shape is at it's core a Lua table, you can attach values and add functions to it. Be careful not to use keys that name a function or start with an underscore, e.g. *move* or *_rotation*, since these are used internally. Everything else is fine.

If you don't want to use the full blown module, you can still use these classes to test for colliding shapes. This may be useful for scenes where the shapes don't move very much and only few collisions are of interest - for example graphical user interfaces.

Base Class

class Shape(*type*)

Arguments

- **type** (*any*) – Arbitrary type identifier of the shape's type.

Base class for all shapes. All shapes must conform to the interface defined below.

Shape:move(*dx*, *dy*)

Arguments

- **dx, dy** (*numbers*) – Coordinates to move the shape by.

Move the shape *by* some distance.

Example:

```
circle:move(10, 15) -- move the circle 10 units right and 15 units down.
```

Shape:moveTo(*x*, *y*)

Arguments

- **x,y** (*numbers*) – Coordinates to move the shape to.

Move the shape *to* some point. Most shapes will be *centered* on the point `(x,y)`.

Note: Equivalent to:

```
local cx,cy = shape:center()
shape:move(x-cx, y-cy)
```

Example:

```
local x,y = love.mouse.getPosition()
circle:moveTo(x, y) -- move the circle with the mouse
```

Shape:center()

Returns `x, y` - The center of the shape.

Get the shape's center.

Example:

```
local cx, cy = circle:center()
print("Circle at:", cx, cy)
```

Shape:rotate(*angle*[, *cx*, *cy*])

Arguments

- **angle** (*number*) – Amount of rotation in radians.
- **cx, cy** (*numbers*) – Rotation center; defaults to the shape's center if omitted (optional).

Rotate the shape *by* some angle. A rotation center can be specified. If no center is given, rotate around the center of the shape.

Example:

```
rectangle:rotate(math.pi/4)
```

Shape:setRotation(*angle*[, *cx*, *cy*])

Arguments

- **angle** (*number*) – Amount of rotation in radians.
- **cx, cy** (*numbers*) – Rotation center; defaults to the shape's center if omitted (optional).

Set the rotation of a shape. A rotation center can be specified. If no center is given, rotate around the center of the shape.

Note: Equivalent to:

```
shape:rotate(angle - shape:rotation(), cx,cy)
```

Example:

```
rectangle:setRotation(math.pi, 100,100)
```

Shape:rotation()

Returns The shape's rotation in radians.

Get the rotation of the shape in radians.

Shape:scale(*s*)

Arguments

- **s** (*number*) – Scale factor; must be > 0.

Scale the shape relative to it's center.

Note: There is no way to query the scale of a shape.

Example:

```
circle:scale(2) -- double the size
```

Shape:outcircle()

Returns `x, y, r` - Parameters of the outcircle.

Get parameters of a circle that fully encloses the shape.

Example:

```
if player:hasShield() then
    love.graphics.circle('line', player:outcircle())
end
```

Shape:bbox()

Returns `x1, y1, x2, y2` - Corners of the counding box.

Get axis aligned bounding box. `x1, y1` defines the upper left corner, while `x2, y2` define the lower right corner.

Example:

```
-- draw bounding box
local x1,y1, x2,y2 = shape:bbox()
love.graphics.rectangle('line', x1,y1, x2-x1,y2-y1)
```

Shape:draw(*mode*)

Arguments

- **mode** (*DrawMode*) – How to draw the shape. Either 'line' or 'fill'.

Draw the shape either filled or as outline. Mostly for debug-purposes.

Example:

```
circle:draw('fill')
```

Shape:support(*dx*, *dy*)

Arguments

- **dx, dy** (*numbers*) – Search direction.

Returns The furthest vertex in direction *dx, dy*.

Get furthest vertex of the shape with respect to the direction `dx, dy`.

Used in the collision detection algorithm, but may be useful for other things - e.g. lighting - too.

Example:

```
-- get vertices that produce a shadow volume
local x1,y1 = circle:support(lx, ly)
local x2,y2 = circle:support(-lx, -ly)
```

Shape:collidesWith(*other*)

Arguments

- **other** (Shape) – Test for collision with this shape.

Returns `collide, dx, dy` - Collision indicator and separating vector.

Test if two shapes collide.

The separating vector `dx, dy` will only be defined if `collide` is `true`. If defined, the separating vector will point in the direction of `other`, i.e. `dx, dy` is the direction and magnitude to move `other` so that the shapes do not collide anymore.

Example:

```
if circle:collidesWith(rectangle) then
    print("collision detected!")
end
```

Shape:contains(*x*, *y*)

Arguments

- **x, y** (*numbers*) – Point to test.

Returns `true` if `x,y` lies in the interior of the shape.

Test if the shape contains a given point.

Example:

```
if unit.shape:contains(love.mouse.getPosition) then
    unit:setHovered(true)
end
```

Shape:intersectionsWithRay(*x*, *y*, *dx*, *dy*)

Arguments

- **x, y** (*numbers*) – Starting point of the ray.
- **dx, dy** (*numbers*) – Direction of the ray.

Returns Table of ray parameters.

Test if the shape intersects the given ray. The ray parameters of the intersections are returned as a table. The position of the intersections can be computed as `(x,y) + ray_parameter * (dx, dy)`.

Example:

```
local ts = player:intersectionsWithRay(x,y, dx,dy)
for _, t in ipairs(t) do
    -- find point of intersection
    local vx,vy = vector.add(x, y, vector.mul(t, dx, dy))
    player:addMark(vx,vy)
end
```

Shape:intersectsRay(*x*, *y*, *dx*, *dy*)

Arguments

- **x, y** (*numbers*) – Starting point of the ray.
- **dx, dy** (*numbers*) – Direction of the ray.

Returns `intersects, ray_parameter` - intersection indicator and ray paremter.

Test if the shape intersects the given ray. If the shape intersects the ray, the point of intersection can be computed by `(x,y) + ray_parameter * (dx, dy)`.

Example:

```
local intersecting, t = player:intersectsRay(x,y, dx,dy)
if intersecting then
    -- find point of intersection
    local vx,vy = vector.add(x, y, vector.mul(t, dx, dy))
    player:addMark(vx,vy)
end
```

Custom Shapes

Custom shapes must implement at least the following methods (as defined above)

- *Shape:move()*
- *Shape:rotate()*
- *Shape:scale()*
- *Shape:bbox()*
- *Shape:collidesWith()*

Built-in Shapes

class ConcavePolygonShape()

class ConvexPolygonShape()

class CircleShape()

class PointShape()

newPolygonShape(...)

Arguments

- **...** (*numbers*) – Vertices of the *Polygon()*.

Returns *ConcavePolygonShape()* or *ConvexPolygonShape()*.

newCircleShape(*cx*, *cy*, *radius*)

Arguments

- **cx,cy** (*numbers*) – Center of the circle.
- **radius** (*number*) – Radius of the circle.

Returns *CircleShape()*.

newPointShape()

Arguments

- **x, y** (*numbers*) – Position of the point.

Returns *PointShape()*.

3.1.3 HC.polygon

```
polygon = require 'HC.polygon'
```

Polygon class with some handy algorithms. Does not provide collision detection - this functionality is provided by *newPolygonShape()* instead.

class Polygon(*x1*, *y1*, ..., *xn*, *yn*)

Arguments

- **x1,y1, ..., xn,yn** (*numbers*) – The corners of the polygon. At least three corners are needed.

Returns The polygon object.

Construct a polygon.

At least three points that are not collinear (i.e. not lying on a straight line) are needed to construct the polygon. If there are collinear points, these points will be removed. The shape of the polygon is not changed.

Note: The syntax depends on used class system. The shown syntax works when using the bundled hump.class or slither.

Example:

```
Polygon = require 'HC.polygon'
poly = Polygon(10,10, 40,50, 70,10, 40,30)
```

Polygon:unpack()

Returns x1,y1, ..., xn,yn - The vertices of the polygon.

Get the polygon's vertices. Useful for drawing with `love.graphics.polygon()`.

Example:

```
love.graphics.draw('line', poly:unpack())
```

Polygon:clone()

Returns A copy of the polygon.

Get a copy of the polygon.

Note: Since Lua uses references when simply assigning an existing polygon to a variable, unexpected things can happen when operating on the variable. Consider this code:

```
p1 = Polygon(10,10, 40,50, 70,10, 40,30)
p2 = p1          -- p2 is a reference
p3 = p1:clone() -- p3 is a clone
p2:rotate(math.pi) -- p1 will be rotated, too!
p3:rotate(-math.pi) -- only p3 will be rotated
```

Example:

```
copy = poly:clone()
copy:move(10,20)
```

`Polygon:bbox`()

Returns `x1, y1, x2, y2` - Corners of the counding box.

Get axis aligned bounding box. `x1, y1` defines the upper left corner, while `x2, y2` define the lower right corner.

Example:

```
x1,y1,x2,y2 = poly:bbox()
-- draw bounding box
love.graphics.rectangle('line', x1,y2, x2-x1, y2-y1)
```

`Polygon:isConvex`()

Returns `true` if the polygon is convex, `false` otherwise.

Test if a polygon is convex, i.e. a line line between any two points inside the polygon will lie in the interior of the polygon.

Example:

```
-- split into convex sub polygons
if not poly:isConvex() then
    list = poly:splitConvex()
else
    list = {poly:clone()}
end
```

`Polygon:move`(*x*, *y*)

Arguments

- **`x, y`** (*numbers*) – Coordinates of the direction to move.

Move a polygon in a direction..

Example:

```
poly:move(10,-5) -- move 10 units right and 5 units up
```

`Polygon:rotate`(*angle*[, *cx*, *cy*])

Arguments

- **angle** (*number*) – The angle to rotate in radians.
- **cx, cy** (*numbers*) – The rotation center (optional).

Rotate the polygon. You can define a rotation center. If it is omitted, the polygon will be rotated around it's centroid.

Example:

```
p1:rotate(math.pi/2)          -- rotate p1 by 90° around it's center
p2:rotate(math.pi/4, 100,100) -- rotate p2 by 45° around the point 100,100
```

Polygon:triangulate()

Returns table of Polygons: Triangles that the polygon is composed of.

Split the polygon into triangles.

Example:

```
triangles = poly:triangulate()
for i,triangle in ipairs(triangles) do
    triangles.move(math.random(5,10), math.random(5,10))
end
```

Polygon:splitConvex()

Returns table of Polygons: Convex polygons that form the original polygon.

Split the polygon into convex sub polygons.

Example:

```
convex = concave_polygon:splitConvex()
function love.draw()
    for i,poly in ipairs(convex) do
        love.graphics.polygon('fill', poly:unpack())
    end
end
```

Polygon:mergedWith(*other*)

Arguments

- **other** (Polygon) – The polygon to merge with.

Returns The merged polygon, or nil if the two polygons don't share an edge.

Create a merged polygon of two polygons **if, and only if** the two polygons share one complete edge. If the polygons share more than one edge, the result may be erroneous.

This function does not change either polygon, but rather creates a new one.

Example:

```
merged = p1:mergedWith(p2)
```

Polygon:contains(*x*, *y*)

Arguments

- **x, y** (*numbers*) – Point to test.

Returns true if x,y lies in the interior of the polygon.

Test if the polygon contains a given point.

Example:

```
if button:contains(love.mouse.getPosition()) then
    button:setHovered(true)
end
```

Polygon:intersectionsWithRay (*x*, *y*, *dx*, *dy*)

Arguments

- **x, y** (*numbers*) – Starting point of the ray.
- **dx, dy** (*numbers*) – Direction of the ray.

Returns Table of ray parameters.

Test if the polygon intersects the given ray. The ray parameters of the intersections are returned as a table. The position of the intersections can be computed as `(x,y) + ray_parameter * (dx, dy)`.

Polygon:intersectsRay (*x*, *y*, *dx*, *dy*)

Arguments

- **x, y** (*numbers*) – Starting point of the ray.
- **dx, dy** (*numbers*) – Direction of the ray.

Returns `intersects, ray_parameter` - intersection indicator and ray paremter.

Test if the polygon intersects a ray. If the shape intersects the ray, the point of intersection can be computed by `(x,y) + ray_parameter * (dx, dy)`.

Example:

```
if poly:intersectsRay(400,300, dx,dy) then
    love.graphics.setLine(2) -- highlight polygon
end
```

3.1.4 HC.spatialhash

```
spatialhash = require 'HC.spatialhash'
```

A spatial hash implementation that supports scenes of arbitrary size. The hash is sparse, which means that cells will only be created when needed.

class Spatialhash ([*cellsize = 100*])

Arguments

- **cellsize** (*number*) – Width and height of a cell (optional).

Create a new spatial hash with a given cell size.

Choosing a good cell size depends on your application. To get a decent speedup, the average cell should not contain too many objects, nor should a single object occupy too many cells. A good rule of thumb is to choose the cell size so that the average object will occupy only one cell.

Note: The syntax depends on used class system. The shown syntax works when using the bundled hump.class or slither.

Example:

```
Spatialhash = require 'hardoncollider.spatialhash'
hash = Spatialhash(150)
```

Spatialhash:cellCoords (*x*, *y*)

Arguments

- **x, y** (*numbers*) – The position to query.

Returns Coordinates of the cell which would contain `x,y`.

Get coordinates of a given value, i.e. the cell index in which a given point would be placed.

Example:

```
local mx,my = love.mouse.getPosition()
cx, cy = hash:cellCoords(mx, my)
```

Spatialhash:cell (*i*, *k*)

Arguments

- **i, k** (*numbers*) – The cell index.

Returns Set of objects contained in the cell.

Get the cell with given coordinates.

A cell is a table which's keys and value are the objects stored in the cell, i.e.:

```
cell = {
    [obj1] = obj1,
    [obj2] = obj2,
    ...
}
```

You can iterate over the objects in a cell using `pairs()`:

```
for object in pairs(cell) do stuff(object) end
```

Example:

```
local mx,my = love.mouse.getPosition()
cx, cy = hash:cellCoords(mx, my)
cell = hash:cell(cx, cy)
```

Spatialhash:cellAt (*x*, *y*)

Arguments

- **x, y** (*numbers*) – The position to query.

Returns Set of objects contained in the cell.

Get the cell that contains point x,y.

Same as `hash:cell(hash:cellCoords(x,y))`

Example:

```
local mx,my = love.mouse.getPosition()
cell = hash:cellAt(mx, my)
```

Spatialhash:shapes()

Returns Set of all shapes in the hash.

Get *all* shapes that are recorded in the hash.

Spatialhash:inSameCells(*x1*, *y1*, *x2*, *y2*)

Arguments

- **x1,y1** (*numbers*) – Upper left corner of the query bounding box.
- **x2,y2** (*numbers*) – Lower right corner of the query bounding box.

Returns Set of all shapes in the same cell as the bbox.

Get the shapes that are in the same cell as the defined bounding box.

Spatialhash:register(*obj*, *x1*, *y1*, *x2*, *y2*)

Arguments

- **obj** (*mixed*) – Object to place in the hash. It can be of any type except *nil*.
- **x1,y1** (*numbers*) – Upper left corner of the bounding box.
- **x2,y2** (*numbers*) – Lower right corner of the bounding box.

Insert an object into the hash using a given bounding box.

Example:

```
hash:register(shape, shape:bbox())
```

Spatialhash:remove(*obj*[, *x1*, *y1*[, *x2*, *y2*]])

Arguments

- **obj** (*mixed*) – The object to delete
- **x1,y1** (*numbers*) – Upper left corner of the bounding box (optional).
- **x2,y2** (*numbers*) – Lower right corner of the bounding box (optional).

Remove an object from the hash using a bounding box.

If no bounding box is given, search the whole hash to delete the object.

Example:

```
hash:remove(shape, shape:bbox())
hash:remove(object_with_unknown_position)
```

Spatialhash:update(*obj*, *x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*)

Arguments

- **obj** (*mixed*) – The object to be updated.
- **x1,y1** (*numbers*) – Upper left corner of the bounding box before the object was moved.
- **x2,y2** (*numbers*) – Lower right corner of the bounding box before the object was moved.
- **x3,y3** (*numbers*) – Upper left corner of the bounding box after the object was moved.
- **x4,y4** (*numbers*) – Lower right corner of the bounding box after the object was moved.

Update an objects position given the old bounding box and the new bounding box.

Example:

```
hash:update(shape, -100,-30, 0,60, -100,-70, 0,20)
```

Spatialhash:draw (*draw_mode*[, *show_empty* = *true*[, *print_key* = *false*]])

Arguments

- **draw_mode** (*string*) – Either 'fill' or 'line'. See the LÖVE wiki.
- **show_empty** (*boolean*) – Wether to draw empty cells (optional).
- **print_key** (*boolean*) – Wether to print cell coordinates (optional).

Draw hash cells on the screen, mostly for debug purposes

Example:

```
love.graphics.setColor(160,140,100,100)
hash:draw('line', true, true)
hash:draw('fill', false)
```

3.1.5 HC.vector

```
vector = require 'HC.vector'
```

See `hump.vector_light`.

3.1.6 HC.class

```
class = require 'HC.class'
```

See `hump.class`.

3.2 Tutorial

To be rewritten.

3.3 License

3.4 Indices and tables

- genindex
- modindex
- search

Index