6.905/6.945 Final Project Draft: Extensible Physics Engine

Due on 8 May 2017

Michael Chang, Abigail Choe, Leon Shen

1 Infrastructure

We used the object system from pset 4 to model physical objects in our simulated world, as well as the world itself. Having the ability to create a hierarchy of types allows us to derive new types of physical objects to be simulated from previously defined types (see section Object Types).

2 Arithmetic

We incorporated the generic arithmetic developed in pset 3. We extended the vector-arithmetic to support element-wise operations between scalars and vectors. We did this by coercing scalars to become vectors that contain the same value. For example, `(+ #(3 4) 1)` gets transformed into `(+ #(3 4) #(1 1))`.

This extension was made to allow our system to operate on any nth dimension vector, and therefore be applicable to any nth dimension world representation. For example, though the examples in this paper concern 2-dimensional worlds, the physics can easily be extended to 3-dimensinoal worlds. The default velocity of an object is set to 0 if it is not specified. Supporting arithmetic operations between a scalar and a vector enables our system to use a single default value 0 that is compatible with any vector instead of limiting our system to the dimension of the default zero vector specified.

The coercion method is

```
(define (coerce a size-reference)
  (if (vector? a)
    a
    (make-vector (vector-length size-reference) a)))
```

The rest of the infrastructure for generic procedures was adapted from pset 4, but pset 4's generic arithmetic is not compatible with that of pset 3. To fix this issue, we first loaded the relevant pset 3 dependencies, installed a generic arithmetic with vector operations, redefined arithmetic operations to be the ones from that generic arithmetic (such as with `(define * (access * arith-environment))`), and then loaded the pset 4 generic procedure infrastructure on top.

3 Physics Engine

3.1 Object Types

There are three types of objects in the system: `thing`, `interaction`, and `world`. `thing` refers to all physical objects in the system. There is a hierarchy of physical objects that have additional properties. For example, type `point-charge` has an additional property `charge` and experiences electric forces while type `ball` experiences only gravity. Each `thing` keeps a list of `interaction` that is used to update itself. `interaction` refers to forces exerted on `thing`s. `interaction` includes a procedure that calculates the force as well as `influences`, a list of `thing`s that contribute to that force. Similar to `thing`, subtypes of `interaction`s form a hierarchy. Lastly, `world` keeps a list of all `thing`s and an `update` call to the `world` calls on each `thing` to update itself.

3.2 Updating Physical Objects

There are mainly two ways positions of physical objects could be updated: 1) The evolver can take in a list of forces and objects and apply forces as appropriate 2) Each object can be registered with a set of forces applicable to it and the evolver can call each object to evolve itself.

Our system adopted the second model where each object updates itself. This model makes it easier to handle different types of forces exerted on specific types of object. Moreover, in the second model, the engine does not need to be notified and updated everytime a new object is added. Instead, update or addition of new `interaction`s is taken care of at the object level when it is added.

3.3 Output Display

Our engine updates each object every timestep and re-renders them on the screen:

```
(define (run-engine world steps)
  (reset-graphics)
  (if (> steps 0)
    (begin
      (for-each
        (lambda (thing)
            (newline)
            (display (cons (get-name thing) (get-position thing)))  ; for debugging
            (render thing))
        (get-world-all-things world))
      (update-world world)
    (run-engine world (- steps 1)))))
```

where the `render` procedure takes in a `thing`, checks what type of thing it is, and renders the object appropriately.

4 Demos

4.1 Gravity

We can easily extend our system with new forces and new objects. We begin by creating an gravity as an `interaction`:

```
(define (make-gravity thing all-things)
  (define (procedure thing influences)
    (sum (map (lambda (influence)
                (let* ((m1 (get-mass thing))
                       (m2 (get-mass influence))
                       (G 6.674e-11)
                       (v (- (get-position influence)  ; vector between influence and thing
                             (get-position thing)))
                       (r (magnitude v))               ; distance between influence and thing
                       (u (/ v r))                     ; unit vector
                       (gmag (* (* G m1 m2)            ; magnitude of gravity
                                (/ 1 (square r)))))
                  (* u gmag)))                         ; gravitational force vector
              influences)))
  (let ((influences (delq thing all-things)))
    (make-interaction gravity? 'gravity procedure influences)))
```

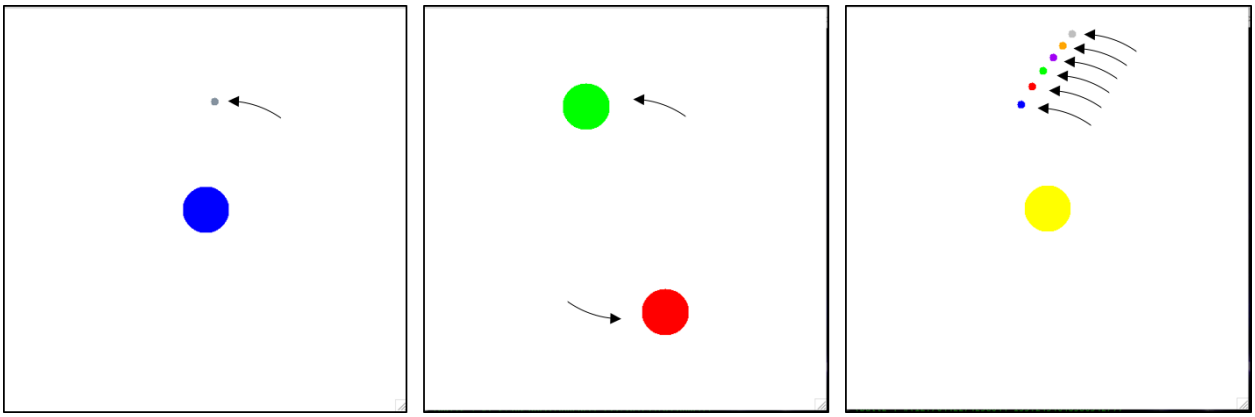

Figure 1: [*left*] `earth-moon`; [*center*] `binary-stars`; [*right*] `solar-system`

We've created several worlds that exhibit gravity (Fig. 1). Once an object is added to the world, any object that has mass will feel gravitational forces with other object that have mass.

This world contains an earth with a moon orbiting around it.

```
(define (earth-moon)
  (define w (make-world "world"))
  (define b1 (make-ball "earth" 30 1e15 #(0 0) #(0 0) "blue"))
```

```
  (define b2 (make-ball "moon" 5 1e5 #(100 100) #(-15.361 15.361) "#85929E"))

  (add-mass! b1 w)
  (add-mass! b2 w)
  w)
```

This world contains two equal masses orbiting a common center of mass.

```
(define (binary-stars)
  (define w (make-world "world"))
  (define b1 (make-ball "ball1" 5 1e15 #(-100 -100) #(9 -9) "red"))
  (define b2 (make-ball "ball2" 5 1e15 #(100 100) #(-9 9) "green"))

  (add-mass! b1 w)
  (add-mass! b2 w)
  w)
```

This world contains a massive sun and several planets orbitting around the sun.

```
(define (solar-system)
  (define w (make-world "world"))
  (define s (make-ball "sun" 30 1e15 #(0 0) #(0 0) "yellow"))
  (define b1 (make-ball "ball1" 5 1e5 #(100 100) #(-15.361 15.361) "blue"))
  (define b2 (make-ball "ball2" 5 1e5 #(110 110) #(-15.361 15.361) "red"))
  (define b3 (make-ball "ball3" 5 1e5 #(120 120) #(-15.361 15.361) "green"))
  (define b4 (make-ball "ball4" 5 1e5 #(130 130) #(-15.361 15.361) "purple"))
  (define b5 (make-ball "ball5" 5 1e5 #(140 140) #(-15.361 15.361) "orange"))
  (define b6 (make-ball "ball6" 5 1e5 #(150 150) #(-15.361 15.361) "gray"))

  (add-mass! s w)
  (add-mass! b1 w)
  (add-mass! b2 w)
  (add-mass! b3 w)
  (add-mass! b4 w)
  (add-mass! b5 w)
  (add-mass! b6 w)
  w)
```

4.2 Electromagentism

Let's add electric forces between point charges into the physics engine.

```
(define (make-electric-force point-charge all-point-charges)
  (define (procedure point-charge influences)
    (sum (map (lambda (influence)
                (let* ((q1 (get-point-charge-charge point-charge))
                    (q2 (get-point-charge-charge influence))
                    (epsilon0 8.854e-12) ; vacuum permittivity
                       ;; vector between influence and point charge
                    (v (- (get-position influence)
```

```
                    (get-position point-charge)))
               ;; distance between influence and point charge
             (r (magnitude v))
               ;; unit vector * -1 to account for opposite signs
               ;; attract & same signs repel
             (u (* -1 (/ v r)))
             (melec (/ (* q1 q2)  ; magnitude of electric force
                       (* 4 pi epsilon0 (square r)))))
         (* u melec))) ; multiply unit vector by magnitude of force
      influences)))
  (let ((influences (delq point-charge all-point-charges)))
    (make-interaction electric-force? 'electric-force procedure influences)))
```

This is a simple world with two point charges repelling each other.

```
(define (charges-1)
  (define w (make-world "world"))
  (define m1 (make-point-charge "charge1" 1e-3 1 #(-30 -30) #(0 0) "red"))
  (define m2 (make-point-charge "charge2" 1e-3 1 #(30 30) #(0 0) "red"))

  (add-point-charge! m1 w)
  (add-point-charge! m2 w)
  w)
```

This world contains two positively charged masses (blue) and two negatively charged masses (red).

```
(define (charges-2)
  (define w (make-world "world"))
  (define m1 (make-point-charge "charge1" 1e-3 1 #(-100 -100) #(0 0) "red"))
  (define m2 (make-point-charge "charge2" -1e-3 1 #(100 100) #(0 0) "blue"))
  (define m3 (make-point-charge "charge3" 1e-3 1 #(-100 100) #(0 0) "red"))
  (define m4 (make-point-charge "charge4" -1e-3 1 #(100 -100) #(0 0) "blue"))

  (add-point-charge! m1 w)
  (add-point-charge! m2 w)
  (add-point-charge! m3 w)
  (add-point-charge! m4 w)
  w)
```

We can also combine gravity and electric charge. Here, we created a solar-system-like world where each of the planets repel each other and the sun attracts the planets via gravity and electric force. As figure 2 (right) shows, this charged solar system exhibits wider and diverging orbits compared to figure 1 (right), the solar system with only gravity.

```
(define (charged-solar-system)
  (define w (make-world "world"))
  (define s (make-point-charge "sun" 3e-3 1e15 #(0 0) #(0 0) "blue"))
  (define b1 (make-point-charge "ball1" -1e-1 1e5 #(100 100) #(-15.361 15.361) "red"))
  (define b2 (make-point-charge "ball2" -1e-1 1e5 #(110 110) #(-15.361 15.361) "red"))
  (define b3 (make-point-charge "ball3" -1e-1 1e5 #(120 120) #(-15.361 15.361) "red"))
  (define b4 (make-point-charge "ball4" -1e-1 1e5 #(130 130) #(-15.361 15.361) "red"))
```

```
(define b5 (make-point-charge "ball5" -1e-1 1e5 #(140 140) #(-15.361 15.361) "red"))
(define b6 (make-point-charge "ball6" -1e-1 1e5 #(150 150) #(-15.361 15.361) "red"))
```

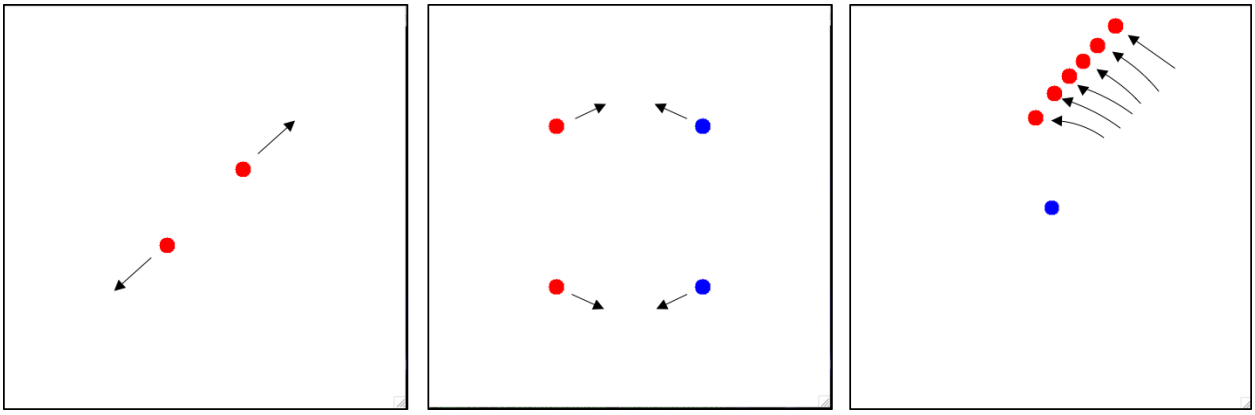

Figure 2: [*left*] `charges-1`; [*center*] `charges-2`; [*right*] `charged-solar-system`

5 Extensibility of the System

Future extensions include collisions between objects, constraints such as joints, simulating objects in three dimensions, and being able to combine objects into larger systems of objects. Collisions and other kinds of constraints could be implemented by adding an interaction that produces forces that enforce these constraints. Objects could be simulated in three dimensions by defining three dimensional object types and using the generic arithmetic to operate on higher dimensional vectors. Objects could be combined into larger systems using procedures, if there is a way to link them together, for example by constraints.