Dynapad Geometry and Physics

Dan Bauer
UCSD Cognitive Science Dept.
HCI & Distributed Cognition Lab

1/31/03

1 Actors

DrScheme is limited to single-inheritance– that is, object classes can inherit behavior from only one superclass lineage. But sometimes we want objects to have different behaviors, inherited from several unrelated classes.

This can be achieved by using *actors*. An actor is like a secretary or other assistant: it has specific talents and handles certain jobs on behalf of its "boss" object. The boss can hire any number of specialists and then delegate any job it can't handle to a qualified assistant; together the team can handle many more jobs than any individual expert. Likewise, an object can attach any number of actors, so that together this "team" has behaviors inherited from multiple super-classes.

Example

First, create a specialized actor class and an object to use the actor:

```
(define critter (make-object rect% dynapad '(0 0 20 10)))
(define greeting-actor%
  (class actor%
    (super-instantiate ())
    (define/public (greet)
     (let ((myclass (send (send this object) dynaclass)))
       (display (format "Hi, I'm a ~a.~%" myclass))
       this))))
```

Note that the actor uses (*send this object*) to refer to the object it's attached to, and that it returns itself.

Create an instance of the actor and attach it to the object:
(*send* (*make-object greeting-actor%*) *attach-to critter*)
Finally, send the *greet* method to the object, and the relevant actor responds:
(*send-actor critter* 'greet) $\Longrightarrow$ (*actor*) $\boxed{\text{"Hi, I'm a rect\%."}}$
Besides displaying a message, the call returns a list of the return values from all responding actors– in this case, the actor itself.

2 Named Actors

Actors are somewhat inefficient because send-actor messages must consider all of an object's actors. When we know an object has a certain type of actor, it's more efficient to send messages directly to that actor. This can be done with *named-actors*. A named-actor is bound to its object by one or more names, and multiple actors may share a name. Named actors are stored in the object's alist (by name), not actor-list, and addressed directly with (*send-actor-named obj name msg...*) instead of (*send obj msg...*).

```
(define blue-filler%
  (class named-actor%
    (super-instantiate ())
    (define/public (blueify)
      (send (send this object) fill "blue"))))

(define blue-liner%
  (class named-actor%
    (super-instantiate ())
    (define/public (blueify)
      (send (send this object) pen "blue"))))

(define red-liner%
  (class named-actor%
    (super-instantiate ())
    (define/public (redify)
      (send (send this object) pen "red"))))

(define square (make-object rect% dynapad '(0 0 1 1)))
(define blue-filler (make-object blue-filler%))
(define blue-liner (make-object blue-liner%))
(define red-liner  (make-object red-liner%))

(send blue-filler attach-to square 'blues)
(send blue-liner attach-to square 'blues)
(send blue-liner attach-to square 'liners)
(send red-liner  attach-to square 'liners)

(get-actors-named square 'blues) ⟹ (blue-filler blue-liner)
(send-actors-named square 'blues blueify) ⟹ [blue fill & pen]
```

Assuming a name includes only one actor, the singular forms (*get-actor-named...*) and (*send-actor-named...*) reach only the first actor of that name:

```
(get-actor-named  square 'blues) ⟹ blue-filler
(send-actor-named square 'blues blueify)  ⟹ [blue fill only]
```

3 Geometry

The library *geometry.ss* contains classes for geometric elements (lines, circles, polygons, etc) represented mathematically for analytic operations (intersection, area, etc.). Most of the objects use a "compute-on-demand" strategy – they always maintain the minimal necessary data, but when additional aspects are needed, the result is stored and reused next time. For example, a line segment has four independent parameters (x and y for each endpoint) but also remembers many dependent parameters (e.g. length, slope, angle, midpoint, y-intercept).

All geo-objects are named-actors (see *actors.ss*) which can be attached to dynaobjects' alists (normally under the key *geo-object-name*), or used independently. To create and attach a geo-object to a dynaobject, use (*geodize dynaobj*).

Each geo-object can draw itself with (*send obj draw* [*color*]) (although this is primarily for debugging; if a geo-object is to be visible it usually makes more sense to attach it to a normal dyna-object).

3.1 Hierarchy

```
geo-actor%
    geo-point%
    geo-angle%
        geo-vector%
    geo-line%
        geo-ray%
            geo-segment%
                geo-poly-segment%
                    geo-polygon-edge%
    geo-polyline%
    geo-polygon%
```

3.2 Class Methods

This section lists the methods available for each class and demonstrates the syntax for using each. Each syntax pattern resembles the following:

(send **object** message **object-parameter** *numeric-parameter...*) $\Longrightarrow$
returned-value or **returned-object**

Literal elements (i.e. send, message) are in normal type, numeric parameters are in *italics*, and objects are in **boldface**. The value returned (if any) is listed after $\Longrightarrow$.

Argument syntax is often flexible. In general, a two-dimensional object argument (e.g. point, vector) may be replaced by a list of two values, or by two separate arguments (unless the result is ambiguous).

By default, all methods of a class are inherited by all its subclasses. Exceptions and changes are noted, but otherwise inherited methods are not re-listed.

3.2.1 geo-actor%

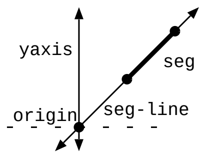
The following methods are inherited by all geo-actor%s.

(describe **obj**)
(send **obj** describe)
    Displays all fields of obj. See section 3.3 for details.

(send **obj** clone) $\Longrightarrow$ **obj2**
    Returns duplicate of obj.

(send **obj** copy-as-*superclass*) $\Longrightarrow$ **obj2**
    Although most object inherit all behaviors of their superclasses, sometimes you'll need to copy as one of its superclasses. For example:

    (**define** *seg* (*make-object geo-segment%* '(1 1) '(2 2)))
    (**define** *yaxis* (*make-object geo-line% infinity* 0))
    (*send seg intercept*) $\Longrightarrow$ 0
    (*send seg intersects yaxis*) $\Longrightarrow$ null
    (**define** *seg-line* (*send seg copy-as-line*))
    (*send seg-line intersects yaxis*) $\Longrightarrow$ (**origin**)



3.2.2 geo-point%

(make-object geo-point% *x y*) $\Longrightarrow$ **point**
(make-object geo-point% (*x y*)) $\Longrightarrow$ **point**

(send **point** x) $\Longrightarrow$ *x*
(send **point** y) $\Longrightarrow$ *y*
(send **point** xy) $\Longrightarrow$ *(x y)*

(send **point** x *val*)
(send **point** y *val*)
(send **point** xy *val val*)
(send **point** xy *(val val)*)

(send **point** slide [**vector** | *(dx dy)* | *dx dy*])
    Slides the object by dx, dy.

(send **point** slide-copy [**vector** | *(dx dy)* | *dx dy*]) $\Longrightarrow$ **point2**
   Copies point and slides copy by dx, dy.

(send **point** vector-to [**point2** | *(x y)*]) $\Longrightarrow$ **vector**
   Returns a vector between point and point2

(send **point** segment-to [**point2** | *(x y)*]) $\Longrightarrow$ **segment**
   Returns a segment between point and point2

3.2.3 geo-angle%

Angles are non-trivial because they represent both a *direction* and a *sweep* (angular difference between directions). Two angles may be the same in the former but different in the latter (for example, π and 3π). Counter-clockwise angles are positive.

Because it specifies a direction. a geo-angle% can also be treated as a unit vector (with undefined length). You can find its dot- or cross-product with other angles and vectors.

(make-object geo-angle% θ) $\Longrightarrow$ **angle**

(send **obj** copy-as-angle) $\Longrightarrow$ **angle**

(send **angle** theta θ)
   Sets **angle** to θ radians.

(send **angle** theta) $\Longrightarrow$ θ (sweep)

(send **angle** net-theta) $\Longrightarrow$ $-\pi < \theta < \pi$ (direction)

(send **angle** pos-theta) $\Longrightarrow$ $0 < \theta < 2\pi$ (direction)

(send **angle** cycles) $\Longrightarrow$ *fraction of circle* (sweep)

(send **angle** dxy *dx dy*)
   Set **angle** to direction of unit vector dx, dy (may both be negative). Note that this is equivalent to (*send angle theta* (*arctan dy dx*)).

(send **angle** dxy) $\Longrightarrow$ *(dx dy)*
   Returns units of a vector (length undetermined) parallel to **angle**.

(send **angle** slope) $\Longrightarrow$ *dy/dx*
   Note that opposite angles have the same slope.

(send **angle** l-normal) $\Longrightarrow$ **angle2**
(send **angle** r-normal) $\Longrightarrow$ **angle2**
Returns angle/vector normal to **angle** toward the left/right.

(send **angle** upward?) $\Longrightarrow$ *boolean*
(send **angle** downward?) $\Longrightarrow$ *boolean*
(send **angle** rightward?) $\Longrightarrow$ *boolean*
(send **angle** leftward?) $\Longrightarrow$ *boolean*
True if **angle** has any component toward specified side of horizontal/vertical.

(send **angle** plus-angle [**angle2** | θ]) $\Longrightarrow$ **angle3**
Returns angle in direction **angle** rotated ccw by **angle2**.

(send **angle** minus-angle [**angle2** | θ]) $\Longrightarrow$ **angle3**
Returns angle in direction **angle** rotated cw by **angle2**.

(send **angle** sweep-to-angle [**angle2** | θ]) $\Longrightarrow$ **angle3**
Returns angle that **angle** needs to add to reach **angle2**. Equal to (*send angle2 minus-angle angle*).

(send **angle** same-dir? **angle2**) $\Longrightarrow$ *boolean*
True if and **angle** and **angle2** have exactly (careful of rounding!) same direction.

(send **angle** parallel? **angle2**) $\Longrightarrow$ *boolean*
True if and **angle** and **angle2** are in same or exactly opposite direction.

(send **angle** between? **angle2 angle3**) $\Longrightarrow$ *boolean*
True if counter-clockwise rotation from **angle2** to **angle3** (inclusive) passes angle.

(send **angle** within? **angle2 angle3**) $\Longrightarrow$ *boolean*
True if angle lies (inclusive) in smaller of **angle2**$\rightarrow$**angle3**, **angle3**$\rightarrow$**angle2**.

(send **angle** splits? **angle2 angle3**) $\Longrightarrow$ *boolean*
True if **angle2**, **angle3** on opposite sides of **angle**'s line (i.e. **angle** or its opposite is within? **angle2**, **angle3**).

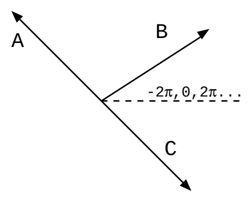
(send **angle** unit-cross [**angle2** | *(dx dy)* | *dx dy*]) $\Longrightarrow$ -1$\geq$*val*$\geq$1
Treats **angle** as if unit vector, and returns magnitude of cross-product with another unit vector. Equal to the sine of the angle between them, i.e. (*sin* (*send* (*send angle diff-from-angle angle2*) *theta*))
If angles are already defined using *dx, dy* instead of θ, unit-cross is considerably faster than (sin...).

(send **angle** unit-dot [**angle2** | *(dx dy)* | *dx dy*]) $\Longrightarrow$ -1$\geq$*val*$\geq$1
Similar to unit-cross, but equal to cosine between angles.

Examples:

(**define** *A* (*make-object geo-angle%* 0))
(*send A dxy* −1 1)
(**define** *B* (*make-object geo-angle%* (∗ *pi* 2.2)))
(**define** *C* (*make-object geo-angle%* (∗ *pi* −4.25)))

(*send A theta*) $\Longrightarrow$ .75π
(*send C theta*) $\Longrightarrow$ -4.25π
(*send C net-theta*) $\Longrightarrow$ -.25π
(*send C pos-theta*) $\Longrightarrow$ 1.75π
(*send B cycles*) $\Longrightarrow$ 1.1
(*equal?* (*send A slope*) (*send C slope*)) $\Longrightarrow$ #t
(*send A between? B C*) $\Longrightarrow$ #t
(*send B between? A C*) $\Longrightarrow$ #f
(*send B between? C A*) $\Longrightarrow$ #t
(*send A within? B C*) $\Longrightarrow$ #f
(*send B within? A C*) $\Longrightarrow$ #t (for any B!)
(*send B splits? A C*) $\Longrightarrow$ #t
(*send C splits? A B*) $\Longrightarrow$ #t
(*send A unit-cross C*) $\Longrightarrow$ 0
(*send A unit-dot C*) $\Longrightarrow$ -1



3.2.4 geo-vector%

A vector is a direction (i.e. angle) and a length. A geo-vector% is a "free" vector which has no location, and therefore does not uniquely specify a line and cannot (slide...).

(send **vector** length *len*)
Sets **vector** length to *len*, and scales *dx, dy* accordingly.

(send **vector** length) $\Longrightarrow$ *len*
Returns length of **vector**, according to *dx, dy*.

(send **vector** scale! *factor*)
Sets **vector** length to *factor * current length*, and scales *dx, dy* accordingly. If *factor* is negative, *dx* and *dy* change sign.

Unlike with geo-angle%, *dx* and *dy* may be accessed independently:

(send **vector** dx *dx*)
Set **vector**'s dx.

(send **vector** dx) $\Longrightarrow$ *dx*
Get **vector**'s dy.

(send **vector** dy *dy*)
Set **vector**'s dy.

(send **vector** dy) $\Longrightarrow$ *dy*
Get **vector**'s dy.

(send **vector** dxy [*(dx dy)* | *dx dy*])
Set **vector**'s dx and dy.

(send **vector** dxy) $\Longrightarrow$ *(dx dy)*
Get **vector**'s dx and dy.

(send **vector** add **vector2**) $\Longrightarrow$ **vector-sum**
Returns the vector sum of two vectors. Both must be proper geo-vectors%, not geo-angles%.

(send **vector** add! **vector2**)
Sets **vector** to sum with **vector2**.

(send **vector** unit-cross [**vector2** | *(dx dy)* | *dx dy*]) $\Longrightarrow$ -1$\geq$*val*$\geq$1
(send **vector** unit-dot [**vector2** | *(dx dy)* | *dx dy*]) $\Longrightarrow$ -1$\geq$*val*$\geq$1
Same as for geo-angle%; takes cross/dot product, ignoring vector lengths, and returns sin/cos of the angle between.

(send **vector** cross [**vector2** | *(dx dy)* | *dx dy*]) $\Longrightarrow$ -1$\geq$*val*$\geq$1
(send **vector** dot [**vector2** | *(dx dy)* | *dx dy*]) $\Longrightarrow$ -1$\geq$*val*$\geq$1
Magnitude of actual cross/dot product of vector, vector2. Equals
(∗ ([*sin*|*cos*] (*send* (*send vector diff-from-angle vector2*) *theta*))
(*send vector length*) (*send vector2 length*))

(send **vector** convert-to-basis **vector2**) $\Longrightarrow$ **vector3**
Treats **vector** as horizonal unit in new coordinate system, and expresses **vector** in terms of that system– i.e. returned **vector3** is direction **vector** relative to **vector2**.

(send **vector** convert-from-basis **vector2**) $\Longrightarrow$ **vector3**
Converts a vector from an alternate basis (via *convert-to-basis* above) back into standard coordinates. I.e. (*send* (*send x convert-to-basis v*) *convert-from-basis v*) $\Longrightarrow$ *x*' where x' is identical to x.

3.2.5 geo-line%

Although dynapad has a line% class, these are really bounded line-segments. A geo-line% is the mathematical abstaction: infinite in both directions, defined by a slope and y-intercept.

(make-object geo-line% *slope intercept*) $\Longrightarrow$ **line**

(send **obj** copy-as-line) $\Longrightarrow$ **line**

(send **line** slope *m*)
(send **line** intercept *b*)
(send **line** slope-intercept *(m b)*)
  Sets slope, y-intercept, or both.

(send **line** slope) $\Longrightarrow$ *m*
  Returns line's slope.

(send **line** normal-slope) $\Longrightarrow$ *m*
  Returns slope orthogonal to line.

(send **line** vertical?) $\Longrightarrow$ *boolean*

(send **line** intercept) $\Longrightarrow$ *b*

(send **line** slope-intercept) $\Longrightarrow$ *(m b)*
  Returns both slope and y-intercept of line.

(send **line** go-thru-point [**point** | *(x y)* | *x y*)
  Moves **line**, keeping slope, to pass through point.

(send **line** slide [**vector** | *(dx dy)* | *dx dy*])
  Moves **line** by dragging a point on it along **vector**. The effect is greatest when **vector** is normal to line, and changes nothing when parallel.

(send **line** vertical?) $\Longrightarrow$ *boolean*

(send **line** parallel? **line2**) $\Longrightarrow$ *boolean*
  Returns true iff **line** and **line2** are parallel.

(send **line** normal? **line2**) $\Longrightarrow$ *boolean*
  Returns true iff **line** and **line2** are normal.

(send **line** equal? **line2**) $\Longrightarrow$ *boolean*
  Returns true iff **line** and **line2** are the same.

(send **line** y-at-x *x*) $\Longrightarrow$ *y*
  Returns the y-value of the point on **line** at x, or *infinity* if vertical.

(send **line** x-at-y *y*) $\Longrightarrow$ *x*
  Returns the x-value of the point on **line** at y, or *infinity* if horizontal.

(send **line** intersects **obj**) $\Longrightarrow$ () or (**pt0 pt1...**)
  Returns a list of all points of intersection of line and obj, or the null list if no intersection. Maximum possible intersections depends on type of object.

(send **line** closest-pt-to [**point**| *(x y)*]) $\Longrightarrow$ *point2*
  Returns the point on **line** which is closest to **point**. If **line** is a geo-ray% of geo-segment%, **point2** may be an endpoint of **line**.

3.2.6 geo-ray%

A ray is a half-line with an origin and direction. Although a ray has a slope, the simple ratio dy/dx leaves direction ambiguous. Instead, direction may be specified with an angle, vector, or second point.

(make-object geo-ray% [**origin** | *(x y)*]
[**angle** | θ | **thru-point** | *(x y)* | **vector** | *dx dy*])

Exceptions to inherited methods:

(send **ray** slope)
(send **ray** intercept)
These values can be retrieved but not set, since neither is adequate to specify direction.

(send **ray** go-thru-point [**point** | *(x y)* | *x y*])
Redirects ray to pass through point, keeping origin but changing direction.

New methods:

(send **obj** copy-as-ray) $\Longrightarrow$ **ray**

(send **ray** origin) $\Longrightarrow$ **point**

(send **ray** origin [**point** | *(x y)* | *x y*])
Set **ray**'s origin to **point**.

(send **ray** slide [**vector** | *(dx dy)* | *dx dy*])
Slides **ray**'s origin by *dx, dy.*

(send **ray** angle) $\Longrightarrow$ **angle**

(send **ray** angle [**angle** | θ])

(send **ray** dxy) $\Longrightarrow$ *(dx dy)*
Although **ray** is infinite, dx and dy specify the unit vector in **ray**'s direction (as with geo-angle%).

(send **ray** dxy [**vector** | *(dx dy)* | *dx dy*])

3.2.7 geo-segment%

A geo-segment is a ray with finite length, or a "fixed" vector. Like a geo-ray%, a geo-segment% uniquely specifies a geo-line%.

(make-object geo-segment% [**origin** | *(x y)*]
[**endpoint** | *(x y)* | **vector** | *dx dy*])

Exceptions to inherited methods:

(send ... go-thru-point...) is eliminated; replaced by (send ... endpoint...)

New methods:

(send **obj** copy-as-segment) $\Longrightarrow$ **segment**

(send **segment** anchor-endpt?) $\Longrightarrow$ *boolean*
(send **segment** anchor-endpt? *boolean*)
If true, anchors **segment**'s endpoint instead of origin for following operations: *set dx/dy/dxy, set length, set angle, slide-to* (see below).

(send **segment** dx) $\Longrightarrow$ *dx*
(send **segment** dx *dx*)
(send **segment** dy) $\Longrightarrow$ *dy*
(send **segment** dy *dy*)
(send **segment** dxy) $\Longrightarrow$ *(dx dy)*
(send **segment** dxy [**vector** | *(dx dy)* | *dx dy*])
When setting dxy, leaves anchored either origin or endpoint (depending on *anchor-endpt?*) and moves opposite end to new dxy offset.

(send **segment** length) $\Longrightarrow$ *length*
(send **segment** length *length*)
When setting length, leaves anchored either origin or endpoint (depending on *anchor-endpt?*) and moves opposite end to new length.

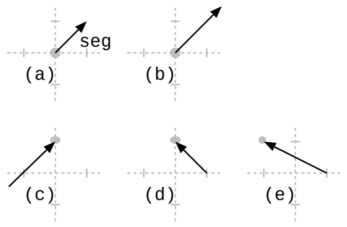
(send **segment** [angle | vector]) $\Longrightarrow$ **vector**
(send **segment** [angle | vector] [**angle** | **vector** | θ | *(dx dy)* | *dx dy*])
The messages *angle* and *vector* are interchangeable, since a segment's vector is also an angle. When setting segment direction, either the origin or endpoint remains anchored (depending on *anchor-endpt?*) and the opposite end moves.

(send **segment** slide-to [**point** | *(x y)* | *x y*])
Maintains direction and length, but slides origin (or endpoint, if anchor-endpt? is true) to point.

Examples:

(**define** *seg* (*make-object geo-segment%* '(0 0) '(1 1))) $\Longrightarrow$ **(a)**

(*send seg length* 2) $\Longrightarrow$ **(b)**
(*send seg anchor-endpt?* #t)
(*send seg slide-to* 0 1) $\Longrightarrow$ **(c)**
(*send seg vector* -1 1) $\Longrightarrow$ **(d)**
(*send seg slope*) $\Longrightarrow$ -1
(*send* (*send seg vector*) *theta*) $\Longrightarrow$ $.75\pi$
(*send seg endpoint* '(-1 1)) $\Longrightarrow$ **(e)**



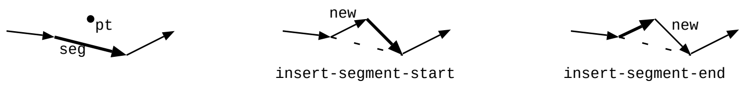
3.2.8 geo-poly-segment%

A geo-poly-segment% is a component segment of a geo-polygon% or geo-polyline% and should not be constructed independently or changed (unless you know what you're doing...) Inherited retrieval functions may be used safely, as well as the following:

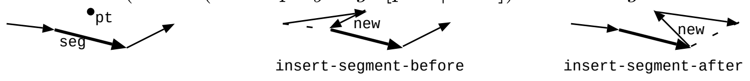
(send **poly-seg** next) $\Longrightarrow$ **poly-seg**
(send **poly-seg** prev) $\Longrightarrow$ **poly-seg**
Return the next/previous segment in their polyline/polygon.

(send **poly-seg** cut-origin)
(send **poly-seg** cut-endpoint)
Cut the origin/endpoint of this segment from the polyline/polygon.

(send **poly-seg** insert-segment-start [**point** | *(x y)*])
(send **poly-seg** insert-segment-end [**point** | *(x y)*])
Inserts new segment at start/end of poly-seg, leaving previous and next segments unchanged.



(send **poly-seg** insert-segment-before [**point** | *(x y)*])
(send **poly-seg** insert-segment-after [**point** | *(x y)*])
Inserts new segment before/after poly-seg, adjusting previous/next segment. Equivalent to (*send* (*send poly-seg* [*prev*|*next*]) *insert-segment-*[*end*|*start*]).



3.2.9 geo-polygon-edge%

A geo-polygon-edge% is an edge of a polygon; like a geo-poly-segment%, it should not be constructed or changed directly.

3.2.10 geo-polyline%

A polyline is a sequence of joined segments.

(make-object geo-polyline% [(**vtx0 vtx1**...) | (*(x0 y0) (x1 y1)...*)])

(send **polyline** clone) $\Longrightarrow$ **polyline**

(send **polyline** slide [**vector** | *dx dy* | *dx dy*])

(send **polyline** vertices) $\Longrightarrow$ (**vtx0 vtx1 ...**)

(send **polyline** segments) $\Longrightarrow$ (**seg0 seg1 ...**)

(send **polyline** vertex-n *n*) $\Longrightarrow$ **vtx-n**

(send **polyline** segment-n *n*) $\Longrightarrow$ **seg-n**

(send **polyline** coords) $\Longrightarrow$ (*x0 y0 x1 y1...*)

(send **line** intersects **obj**) $\Longrightarrow$ () or (**pt0 pt1...**)

To add or remove a segment, send a *cut...* or *insert...* to the relevant segment.

3.2.11 geo-polygon%

(make-object geo-polygon% [(**vtx0 vtx1**...) | (*(x0 y0) (x1 y1)...*)])

(send **polygon** clone) $\Longrightarrow$ **polygon**

(send **polygon** slide [**vector** | *dx dy* | *dx dy*])

(send **polygon** vertices) $\Longrightarrow$ (**vtx0 vtx1 ...**)

(send **polygon** edges) $\Longrightarrow$ (**seg0 seg1 ...**)

(send **polygon** vertex-n *n*) $\Longrightarrow$ **point**

(send **polygon** edge-n *n*) $\Longrightarrow$ **segment**

(send **polygon** spoke-n *n*) $\Longrightarrow$ **vector** (send **polygon** coords) $\Longrightarrow$ (*x0 y0 x1 y1...*)

(send **polygon** orientation) $\Longrightarrow$ 1 or -1

(send **polygon** invert)

(send **line** intersects **obj**) $\Longrightarrow$ () or (**pt0 pt1...**)

(send **polygon** anchor) $\Longrightarrow$ **point**

(send **polygon** anchor [**point** | *(x y)*])

(send **polygon** point-centroid) $\Longrightarrow$ **point**

(send **polygon** area-centroid) $\Longrightarrow$ **point**

(send **polygon** area) $\Longrightarrow$ *area*

(send **polygon** perimeter) $\Longrightarrow$ *perimeter*

(send **polygon** bbox) $\Longrightarrow$ (*lo-x lo-y hi-x hi-y*)

(send **polygon** width) $\Longrightarrow$ *width*

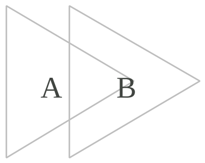
(send **polygon** height) $\Longrightarrow$ *height*

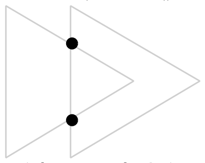
(send **polygon** contains-pt? [**point** | *(x y)*]) $\Longrightarrow$ *boolean*

(send **polygon** margin *width*) $\Longrightarrow$ **polygon2**
Creates a margin around **polygon** at distance *width* from all edges, and returns it as **polygon2**. If *width* is negative, margin is interior (which can cause problems if (abs width) is too great).

3.2.12 Polygon Combinations

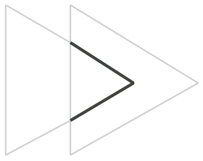
(EdgeIntersectEdge **polygon-A polygon-B**) $\Longrightarrow$ () or (**pt0 pt1...**)



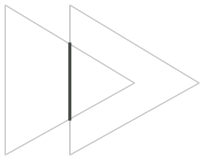


(EdgeIntersectEdge **A B**)
(EdgeIntersectEdge **B A**)

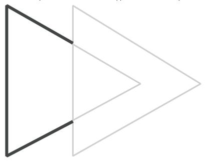
(EdgeIntersectArea **polygon-A polygon-B**) $\Longrightarrow$ () or (**polyline0...**)
(EdgeSubtractArea **polygon-A polygon-B**) $\Longrightarrow$ () or (**polyline0...**)



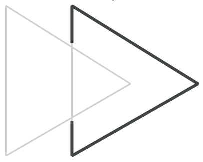
(EdgeIntersectArea **A B**)



(EdgeIntersectArea **B A**)

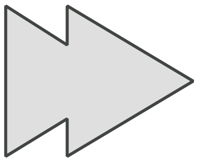


(EdgeSubtractArea **A B**)

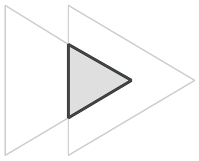


(EdgeSubtractArea **B A**)

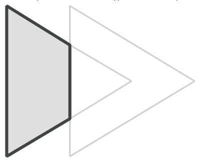
(AreaIntersectArea **polygon-A polygon-B**) $\Longrightarrow$ () or (**polygon0...**)



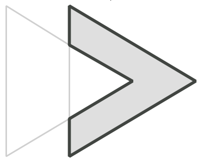
(AreaUnionArea **A B**)
(AreaUnionArea **B A**)



(AreaIntersectArea **A B**)
(AreaIntersectArea **B A**)



(AreaSubtractArea **A B**)



(AreaSubtractArea **B A**)

3.3 (describe...)

When something goes wrong, you can use the *describe* function to see all of an object's fields. For example:

(**define** *my-seg* (*make-object geo-segment%* '(0 0) 1 2))
(*describe my-seg*)

```
Object <0>:
geo-line%: slope:stale intercept:stale
geo-vector%: dx:1 dy:2 length:stale angle:stale
geo-segment%: origin:<1> endpoint:stale
```

The description shows values of fields at 3 different class levels. As a *geo-segment%*, *my-seg* has an origin and endpoint, but as a *geo-vector%*, it also has a length, angle, etc. Values which are not up-to-date are listed as *stale*, but notice that they refresh when accessed:

(*send my-seg slope*) $\Longrightarrow$ 2
(*describe my-seg*)

```
Object <0>:
geo-line%: slope:2 intercept:stale
geo-vector%: dx:1 dy:2 length:stale angle:stale
geo-segment%:origin:<1> endpoint:stale
```

Each object described is given a unique ID number, assigned sequentially from 0. In this case, *my-seg* is the first object described and has ID=0. It also contains another object, its origin point, which is shown only as ID=1. You can use any ID mentioned to describe that object:

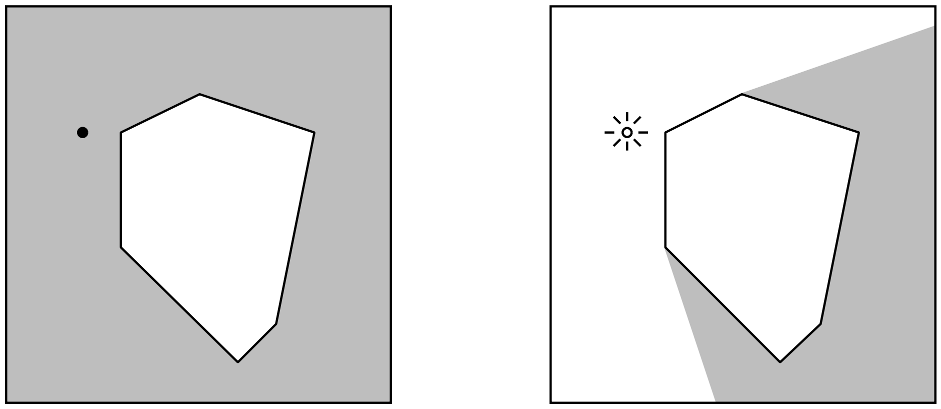
(*describe* 1)

```
Object <1>:
geo-point%: x:0 y:0
```

To assign a variable to a numbered object, use *grab*:

(**define** *my-point* (*grab* 1))
(*eq?* (*send my-seg origin*) *my-point*) $\Longrightarrow$ #t

3.4 Exercise



You're given three pad objects: a point, a polygon, and a rectangle. The point is outside the polygon, and both are enclosed by the rectangle. To keep it simple, assume the polygon is strictly convex. Imagine that they represent a light bulb (point) and an opaque object (polygon), both in a room (rectangle). Using the geometry library, shade the area of the room in the object's shadow.

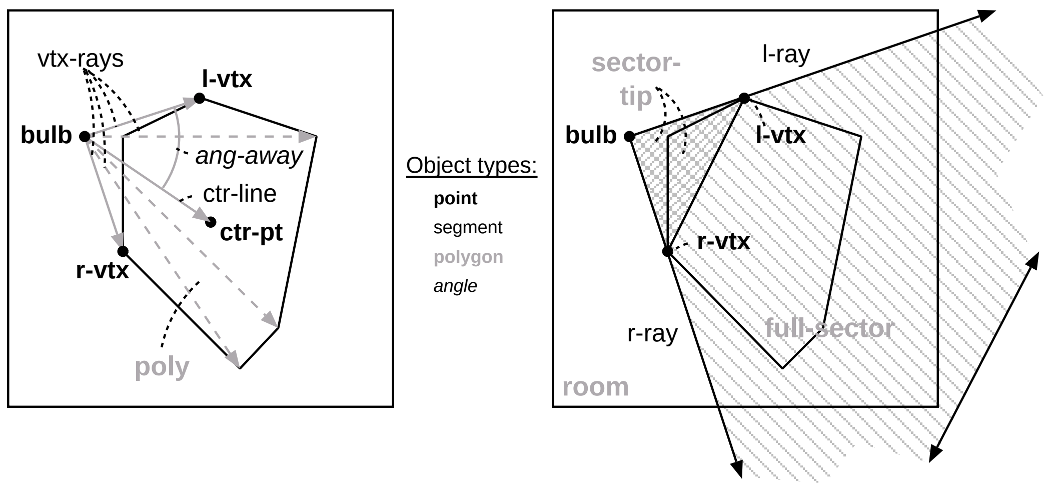
3.4.1 One Solution

Given dynapad objects: *BULB*, *POLY*, *ROOM*

To begin, make geo-actors for these objects:

(**define** *bulb* (*geodize BULB*))
(**define** *poly* (*geodize POLY*))
(**define** *room* (*geodize ROOM*))

The problem can be solved in two phases, illustrated in the figures below. First, find the limits of the polygon's shadow, where the light rays pass just tangent to it on two sides. Since it has only flat edges, these points must be vertices of the polygon. So simply test the rays from the bulb to each vertex, and see which have the maximal leftward and rightward angles from the polygon's "center". This center point can be anywhere inside the polygon– for instance, the centroid. (Even faster would be to use the midpoint of any segment.)



```
(define vtxs (send poly vertices))
(define vtx-rays (map (lambda (vtx) (send bulb vector-to vtx))
                      vtxs))
(define ctr-pt (send poly point-centroid))
(define ctr-line (send bulb vector-to ctr-pt))
(define angs (map (lambda (vtx-ray)
                    (let ((ang-away (send ctr-line sweep-to-angle vtx-ray)))[1]
                      (send ang-away net-theta)))
                  vtx-rays))
(define max-lft (apply max angs))  ;leftward angles are positive
(define max-rgt (apply min angs))  ;rightward angles are negative
(define lft-index (list-position angs max-lft))[2]
(define rgt-index (list-position angs max-rgt))

(define l-vtx (list-ref vtxs lft-index))
(define r-vtx (list-ref vtxs rgt-index))
```

[2]Since we only need the *relative* angle magnitude, it would be faster to use the unit cross product (i.e. sine of the angle) instead:
(send ctr-line unit-cross vtx-ray)

[2]Note: it's possible that two different vertices will have the same maximal leftward or rightward angle. In this case, that boundary light ray is tangent to the face of the polygon

Now, with these two vertices, we can determine two boundary rays which form a triangular sector enclosing the shadow. In principle, this region would be infinite, but we only need it big enough to reach beyond the maximum extent of the room– its diagonal. (For simplicity, we can use height + width, which is even bigger.) This triangle (*full-sector*) is a polygon, so we can use polygon operations to chop it down to just the shadow.

```
(define max-range (+ (send room width) (send room height)))
(define l-ray (make-object geo-segment% bulb l-vtx))
(send l-ray length max-range)
(define r-ray (make-object geo-segment% bulb r-vtx))
(send r-ray length max-range)
(define full-sector (make-object geo-polygon%
                                 (list bulb
                                 (send r-ray endpoint)
                                 (send l-ray endpoint))))[3]
(define sector-tip (make-object geo-polygon%
                                 (list bulb r-vtx l-vtx)))
(define fat-sector-tip (send sector-tip margin
                                 (/ (send sector-tip width) 10000)))[4]
(define shadow (car (AreaSubtractArea
                         (car (AreaIntersectArea room full-sector))
                         (car (AreaUnionArea fat-sector-tip poly)))))
```

Finally, display the shadow in dynapad:

```
(define SHADOW (make-object polygon% dynapad (send shadow coords)))
(send SHADOW fill "grey")
(send SHADOW pen "none")
```

4 Physics

4.1 motion-actor

4.2 lookout-actor

4.3 repulsor

5 Regions

Regions can be used to give special behavior to objects in a certain spatial area– for example, a region might automatically organize objects within it into a tidy layout, distributing them evenly in the space and preventing overlap.

between those two vertices, and it doesn't matter which we use. If there are duplicates, *list-position* will match the first.

[4]Note that polygon vertices should be listed in counter-clockwise order.

[4]Currently, polygon operations go haywire if edges line up exactly. We can use fat-sector-tip to nudge the vertices/edges out of alignment.

Any dynaobject can be turned into a region, and regions can be customized to have any kind of behavior.

Currently there are two region classes:

A *simple-region%* detects and applies an action to any objects contained within it (including other regions), but does not care *where* within it the objects are located. A simple-region instance must provide two functions: and enter-action (applied to objects when they first enter the region) and an exit-action (applied when they leave the region).

A subclass, the *field-region%*, also applies actions to contained objects, but the action can vary with the objects' position. In addition to the enter- and exit-actions required of a simple-region%, a field-region% must also provide an update-action for when object are moved within the region.

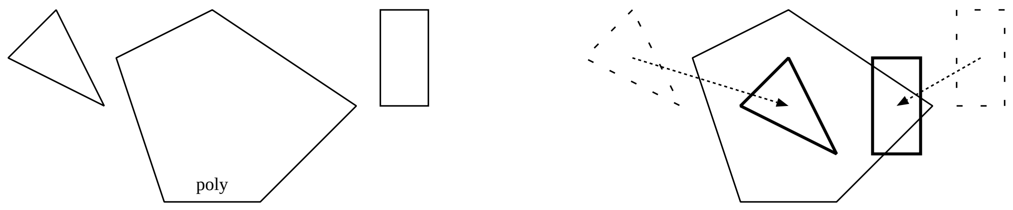
The turn on object into a region, just (*regionize...*) it and provide its region type:
(regionize **dynaobj region-class**)

Currently, the definition of "containment" in a region:
1) includes those objects whose centroid likes within the region, and
2) excludes objects whose bounding box engulfs the region's bounding box (thus preventing objects from dwarfing their "container").

Example 1

The *thickness-region%* below doubles the border-width of any object it contains:

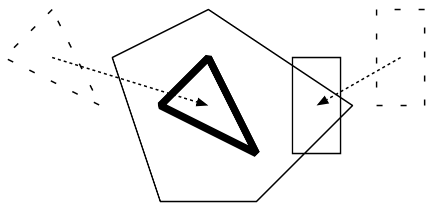
```
(define thickness-region%
  (class simple-region%
    (init _obj)
    (super-instantiate (_obj))

     (send this enter-action
               (lambda (obj)
                 (if (not (is-a? obj text%))
                    (let ((old-width (send obj penwidth)))
                      (send obj penwidth (* 2 old-width))))))
     (send this leave-action
               (lambda (obj)
                 (if (not (is-a? obj text%))
                    (let ((old-width (send obj penwidth)))
                      (send obj penwidth (/ 2 old-width))))))

     (send this refresh-contents)))

(regionize poly thickness-region%)
```



In this example, the new region subclass does not introduce any new methods of its own, but simply assigns functions to its enter- and exit- actions.

Example 2

Let's augment this region class to be sensitive to the position of objects within the region. The *thickness-lens%* region will adjust the thickness according to how deep within the region's boundary the object is placed. Since object position matters, use a field-region%:



```
(define thickness-lens%
  (class field-region%
    (init _obj)
    (super-instantiate (_obj))

     (define (thickness-factor obj)
       (let* ((obj-pt (make-object geo-point% (send obj xy)))
              (boundary-pt (send-actor-named (send this object) geo-actor-name
                                                'closest-pt-to obj-pt))
              (dist (send (send obj-pt vector-to boundary-pt) length))
              (rgn-size (maxdim (send (send this object) bbox))))
         (* 20 (/ dist rgn-size))))

     (define (adjust-thickness obj)
       (if (not (is-a? obj text%))
           (send obj penwidth (thickness-factor obj))))

     (define (save-thickness obj)
       (replace-else-push-onto-alist!
            assq 'old-pen-width (list (send obj penwidth)) obj alist))

     (define (restore-thickness obj)
       (if (not (is-a? obj text%))
           (send obj penwidth
               (cadr (get-and-rem-from-alist!
                       assq remv 'old-pen-width obj alist)))))

     (send this enter-action
           (lambda (obj)
             (save-thickness obj)
             (adjust-thickness obj)))

     (send this update-action adjust-thickness)

     (send this leave-action
           (lambda (obj) (restore-thickness obj)))

     (send this refresh-contents)
))

(regionize poly thickness-lens%)
```

Exercise

Using your shadow-finding algorithm from the exercise in section 3.4 to create a region class which behaves as a shadow-producing room. One solution is in the file *shadows.ss*.