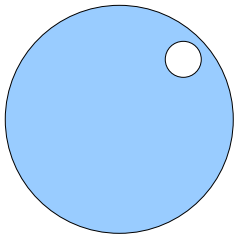
# *Machina Speculatrix* Project
# Part 1

In this project, the robot called *Machina Speculatrix* by W. Grey Walter in his book *The Living Brain* will be simulated. Building this simulation will be divided into at least two parts. In this part, the basic components of the simulation will be built and tested. In the next part, these components will be put together and programmed to give the lifelike behavior that *Machina Speculatrix* exhibited. The simulation will be made using procedures to represent objects. These procedures respond to messages using the message passing paradigm. For more details about representing objects, see Chapter 3 of the book and also the fox-rabbit-grass project that will be given to the regular class for their second project. Time will be simulated with the 'tick message. The robots (the only things that change in the simulation) will be kept in a list. The top level of the simulation will be a loop that repeatedly goes down this list and sends the message 'tick to each of the robots. When the robot receives the 'tick message, it will move a short distance and do whatever it has to do in a very short interval of time and remember its state so that when the next 'tick message arrives, it can continue its behavior for the next short interval of time. The short interval of time here is a small fraction of a second.

## The robot

The robot has several components that will participate in producing its behavior, though not all will be visible in the graphic representation of the robot. When the robot receives the message 'draw, it will draw itself in the graphics window. (See the project statements for the two CISC280-010 projects for details on opening and drawing lines and shapes in the graphics window; There is a lab assignment that does this too.) I suggest that the body of the robot be drawn as a solid-colored circle, and the photo-cell (and pilot light) be drawn as a small circle inside the body circle near its edge. The robot might look something like this:



For simplicity, let's refer to the small circle as the eye of the robot, since it shows where the robot's photo-cell is located. It is also where the pilot light is located (below the photo-cell on the real robot) and will be white when the pilot light is on, and black when the light is off. The eye is at the front of the robot.

The real *Machina Speculatrix* had two wheels at the back, like the wheels of a wagon and they were not powered. The front wheel was turned by a drive motor, which always turned in one direction, either at full speed, half speed or no speed (off). The wheel was mounted at the bottom end of a vertical steering column on top of which was mounted the photo-cell, which always pointed in the direction that the wheel would roll when the drive motor was running. The steering column is rotated, always in the same direction by a motor called the scanning motor. This motor also ran at either full speed, half speed or no

speed, We don't have to model all of this physical arrangement in great detail, but we have to model it enough to simulate the robot's motion.

Here is a suggested way to compute the motion of the robot during one tick of the simulation. The representation for the robot has within it several variables that determine it state. Among these are its location in the graphics window (x y coordinates), its orientation (represented by the angle $\alpha$ in radians between the line from the center of the robot to the center of the eye and the x axis), the orientation (represented by an angle $\beta$) of the eye (and the front wheel) relative to the orientation of the robot's body, the speed $d\beta$ of the scanning motor and the speed *ds* of the drive motor. We will think of $d\beta$ as the angle through which the steering column will turn and of *ds* as the distance that the front wheel will roll during one tick of the simulation. Thus, to figure out where to redraw the robot after one tick of motion, the steps would be something like this:

1. Change $\beta$ to $\beta + d\beta$. (You will need the Scheme function `set!` to do this. See how this is done in the procedure representation for grass created by the `make-grass` function in the fox-rabbit-grass project. Notice that the parameters to the `make-grass` function become the internal variables representing the state of the piece of grass being represented.)
2. Imagine a vector with orientation $\beta$ and length *ds*. This represents the approximate motion of the eye (and front wheel) of the robot. Compute the *dx* and *dy* components of this vector. This is done relative to the graphics window coordinates and we are momentarily ignoring $\alpha$. The numerical quantity *dx* will be the distance that the robot moves forward, and the numerical quantity *dy* will be the distance that the eye will move in a sideways direction.
3. Translate the body of the robot (move its center point) in the $\alpha$ direction by the distance *dx*.
4. Let $\gamma$ satisfy the equation $\tan(\gamma) = dy/(2*r)$ where *r* is the radius of the robot body. Rotate the body of the robot around its back-most point (where the rear wheels are) by angle $\gamma$. (Compute where the back-most point on the robot is, based on the orientation angle α, then recompute where the center of the robot will be assuming that the mack-most point does not move, but the robot's orientation angle is $\alpha + \gamma$.) This will effectively move the eye position to the side by a distance of *dy*. (Rotate around the back-most point, not the middle point. That is why *dy* is divided by the robot diameter *2*r*.)
5. Set $\alpha$ to $\alpha + \gamma$. Now draw the robot at its new position with orientation $\alpha$ and the eye at the front of the robot.

If this is done in fairly small steps, it should approximate the cycloidal motion seen in Grey Walter's robots.

## The other objects

There are three other kinds of objects that will be simulated. These are opaque barriers, reflective barriers, and light sources.

Opaque barriers are represented by finite line segments and they can be drawn as a straight line between the two end points. These will be immovable, so the moving routine for the robot will have to be modified by detecting when the motion would cause the robot to move through a barrier. The moving routine would then do something that would not let it move into the barrier. Perhaps the simplest strategy would be to just not move at all when the predicted movement will cause the robot to bump a barrier. Or even simpler, have a global parameter *b* and a procedure that determines whether the robot is within a distance *b* of any barrier. This procedure will be executed before the move calculations outlined above are done. Those calculations will be skipped when the procedure detects a barrier. Instead, the robot will change to a new state that will change its motion parameters, but let's not worry about that right now. Just

figure out a way to detect when the robot is too close to a barrier.

I suggest that you make a global function `contact?` such that if the center of a robot is located at point <*x,y*>, the call `(contact? x y)` will send to each barrier `b` a message by calling `(b 'contact? x y)`. Barrier `b` then compares the point <*x,y*> with its own end points to determine whether the robot is making contact with it. The barrier returns `#t` if the robot is making contact with it and `#f` otherwise. If any barrier returns `#t`, so does `contact?`. Otherwise, `contact?` returns `#f`.

Here is a way for a barrier to determine whether a robot is in contact with it. Let **A** and **B** be vectors representing the end points of the barrier. Then the line that goes through those two points can be represented by the formula **A** + *t* (**B** - **A**) where *t* is a real-valued parameter. Note that the values between 0 and 1 for *t* map to the line segment between point **A** and point **B**. Let the robot's center point be represented by the vector **C**. The point on the line through **A** and **B** that is nearest to **C** corresponds to the point where $t = -(\mathbf{E} \circ \mathbf{F})/(\mathbf{F} \circ \mathbf{F})$ where **E** = **A** - **C** and **F** = **B** - **A** and ◦ is the scalar product operator for vectors. Once that point is found, the distance between it and **C** can be computed and compared to the robot's radius if *t* is between 0 and 1. If *t* is not between 0 and 1, however, the nearest point on the barrier to the robot is one of the end points **A** and **B** and that distance must be computed. If the distance from the nearest point on the barrier to the robot is about equal to the robot's radius, `#t` is returned, else `#f` is returned.

A reflective barrier is like an opaque barrier and will be represented by a finite line segment too. It will be drawn with a different color than an opaque barrier. This barrier is a mirror. For simplicity, we'll only use a mirror with one robot to get it to respond to its own reflection (actually the reflection of its own pilot light). It will only be necessary to determine when a robot is in front of the mirror so that it will see the reflection of its own pilot light. It will not see the reflections of other light sources in the mirror.

A light source will be represented by a small circle with white interior. It probably should be the same size as the eye of the robot. In fact, the robot might be thought of having two parts, the body and the pilot light that we have been calling the eye (since the photo-cell will have the same horizontal position as the pilot light). The eye is the only light source that will go on and off. The thing that needs to be calculated here is how much light reaches the eye, either directly from another light source (including the eye of another robot) or from the reflection of a light source in a reflective barrier. Assume that barriers are not very high so that all lights can be seen regardless of where they are. (This allows the behavior where a robot is attracted to a light but runs into a barrier when it goes towards it.) The basic physics that has to be remembered here is that light intensity from a point source is proportional to $1/(d^2)$ where *d* is the distance that the light has traveled from the source. The intensity also varies in proportion to $\cos(\delta)$ where $\delta$ is the angle between where the eye is pointing and the direction that the light is coming from. Furthermore, if this *cosine* is negative, the eye can't see the light. (The eye can only see ahead, not behind.) You will need to write a procedure that will check all the light sources in the simulated world (including the eyes of other robots and even the robot's own eye if it is reflected in a reflective barrier) and determine which will produce the brightest light at the robot's eye. All that is needed to be returned by the procedure is the intensity of the light and not the light's location or direction.

# Testing

You will have to define a constructor `make-robot` to create the procedure representing your robot. The definition will look something like this:

```
(define (make-robot x y alpha beta dbeta ds)
```

```
  (lambda (msg)
     (cond ((eq? msg 'tick)
            ???)
            ???)))
```

The parameters will become the internal variables for the robot and will be updated as needed. (We will probably add more parameters when we start to model more complex behaviors.) A robot is created with an expression like

```
(define robot1 (make-robot 10 20 0 0 0.1 2))
```

To run this one robot for *n* ticks, define a `run` function, perhaps like this:

```
(define (run n)
  (define (rerun t)
     (robot1 'tick)
      (robot1 'draw)
      (if (= t 0)
          'done
          (rerun (- t 1))))
  ((clear-viewport w))
  (rerun n))
```

Now the command `(run 10)` will run the robot simulation for ten cycles (ticks).

I suggest that you first make the robot so that it moves reasonably well in a world where nothing else exists. You will have to experiment for suitable values for `dbeta` and `ds` so that you can get the various zig-zag behaviors shown in the photos of *Machina Speculatrix* in the articles I pointed you to in my recent email.

After this is working, add the opaque barriers and write the procedure that detects when the robot is too close to a barrier. Modify the robot so that on each tick it first determines whether it is too close to a barrier. If so, it does nothing. If not, it does the movements it did before.

Once that is working, add light sources. Write the procedure for figuring how much light is activating the eye from the light sources in the simulated world. On each tick, the robot will run this procedure and simply display the light intensity using the `display` and `newline` functions. The robot's motion will not be affected by this procedure at this time.

Next, add reflective barriers. These will stop the robot if it gets too close, just as opaque barriers do, but will make some light sources visible by reflection that were not visible before.

## Results

I will not set a definite deadline for doing the above, but I would like to see it done by the end of March so that we can work on adding the behaviors where the robot responds to moderately bright light and gets dazzled by bright light, and works its way around barriers. Hopefully we can get a robot to dance in front of a mirror and several robots to move around as a group.

Keep the TA and me updated on your progress through the stages outlined above by sending us your

working code along with any comments about how to run it and problems you are having with it. Don't hesitate to come to me with problems you encounter, and if it is a sufficiently difficult problem, I'll help solve it and share the solution with the rest of the class.

## Drawing circles

Note that the natural coordinate system in the graphics window is in terms of rows and column, so the origin *<0,0>* is in the upper left-hand corner and point *<x,y>* corresponds to row *y* column *x*. To draw a circle, one has to draw an ellipse inside of a bounding square. There are two functions for drawing ellipses. Here are sample function calls that will draw circles.

`(define ellipse (draw-ellipse w))` `w` is the viewport for the graphics window

`(ellipse (make-posn x y) d d)` `x,y` is the upper left-hand corner of the bounding square, `d` is the length of the side of the bounding square. Draws black border of a circle.

`(define solid-ellipse (draw-solid-ellipse w))`

`(solid-ellipse (make-posn x y) d d color)` Draws a solid circle filled in with the specified `color`. See project 2 for CISC280-010 for examples of defining colors using `make-rgb`.