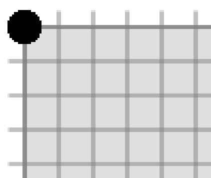
Drawings as Values

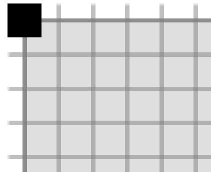
Summary: We consider a very different representation of images, in which we think of an image as a drawing that can be built from other drawings.

Introduction

Instead of thinking about images in terms of how we make them, we might also think of images in terms of the kinds of things that we are able to draw. For example, we might agree that a unit circle (a circle with diameter 1) is something we can draw, as is a unit square (a square with edge length 1). Why start with those two things? We have to start somewhere.

It might help to visualize these two simple drawings.





In these images, you see the unit circle and unit square superimposed upon a simple grid. The greyish area of the grid represents the part of the coordinate system we normally use (non-negative columns, non-negative rows). The grid and greyish area are there only to provide context, they are not part off the “drawing”.

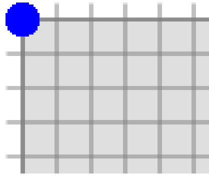
You can certainly imagine a few other simple things that one might draw.

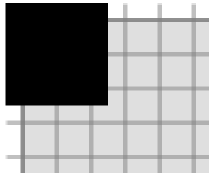
Suppose we have something that everyone agrees can be drawn, such as the unit circle. Could we draw the same thing in a different color? Certainly. Can we draw a bigger or smaller version of the same thing? It seems like we should be able to do so.

To help us discuss these things, let’s call something that can be drawn a “drawing”. So far, we know that

- A black unit circle is a drawing.
- A black unit square is a drawing.
- A recolored drawing is still a drawing.
- An scaled drawing is still a drawing.

For example, a blue unit circle is drawing, as is a unit square scaled by a factor of 5. (The rendering cuts off part of the scaled square, because it is so far outside of the actual image.)





Is a red circle of diameter 4 centered at (0,0) a drawing? Yes. We can start with a unit circle centered at (0,0), scale it by 4, and recolor it red.

If we rely on the four definitions above, is a a purple circle of diameter 4 centered at (2,3) a drawing? How about a green oval centered at (0,0)? In each case of these cases, we probably have to say “No”, because we can’t build either starting with the basic shapes (unit circle, unit square) and using the two operations mentioned above (recolor, scale).

At this point, you may be asking yourself “Why are we going into such detail about these simple drawings?” You may also be asking yourself “Why aren’t things that are obviously easy to draw, like the purple circle and green oval, considered drawings?”

We are going into this detail because it’s a common practice in computer science (and in many other related disciplines, such as mathematics) to formally specify the kinds of data you work with. That is, in formalizing a group of values, we often specify some basic values (such as squares and circles) and then ways they modify those basic values to create new values (recolor, scale). My Mathematician friends often describe the natural numbers (the non-negative integers) in a similar way: “Zero is a natural number. If you add one to a natural number, you get a natural number.”

There are many benefits to such formal definitions, some of which we will explore in this course. One natural benefit is it helps to ensure that everyone talking about a kind of data agrees about the form of those data.

So, why are purple circles (not centered at (0,0)) and green ovals not drawings, even though they are obviously easy to draw? Because our definition does not currently admit them. The inability to specify these natural drawings may suggest a flaw in our definition. Of course, it may also suggest a difference of opinion on what is easy to draw. (Many people can draw squares and circles on graph paper, particularly if given a straight edge and a compass. Fewer can draw ellipses and ovals.)

Since our definition is still relatively simple, let’s think of ways to extend it. Our preliminary of those questions dealt with a single drawing. Suppose we had two (or more) drawings and superimposed them. Would the result be a drawing? (Looking at it another way, if someone can draw each of two separate things, can they draw one and then the other on the same sheet of paper?) Yes, it seems so.

- If D1 is a drawing and D2 is a drawing, then the superimposition of D1 over D1 is a drawing.

You may be able to think of other ways we can decide whether we can draw something. For example, if we can draw something centered at one position, we can probably draw it at another position.

However, Computer Scientists (and some Mathematicians) often find it useful to describe things in this way.

In the remainder of this reading, we'll consider how we might describe drawings using a similar technique, using Scheme to formalize the values and operations. That is, we'll start with some basic drawing values and provide operations that build new drawings from old. In particular, we'll design a `drawing` data type. (For now, the implementation of the procedures on this data type will be concealed. As the semester progresses, we'll think about ways to implement those procedures.)

Basic Values and Basic Operations

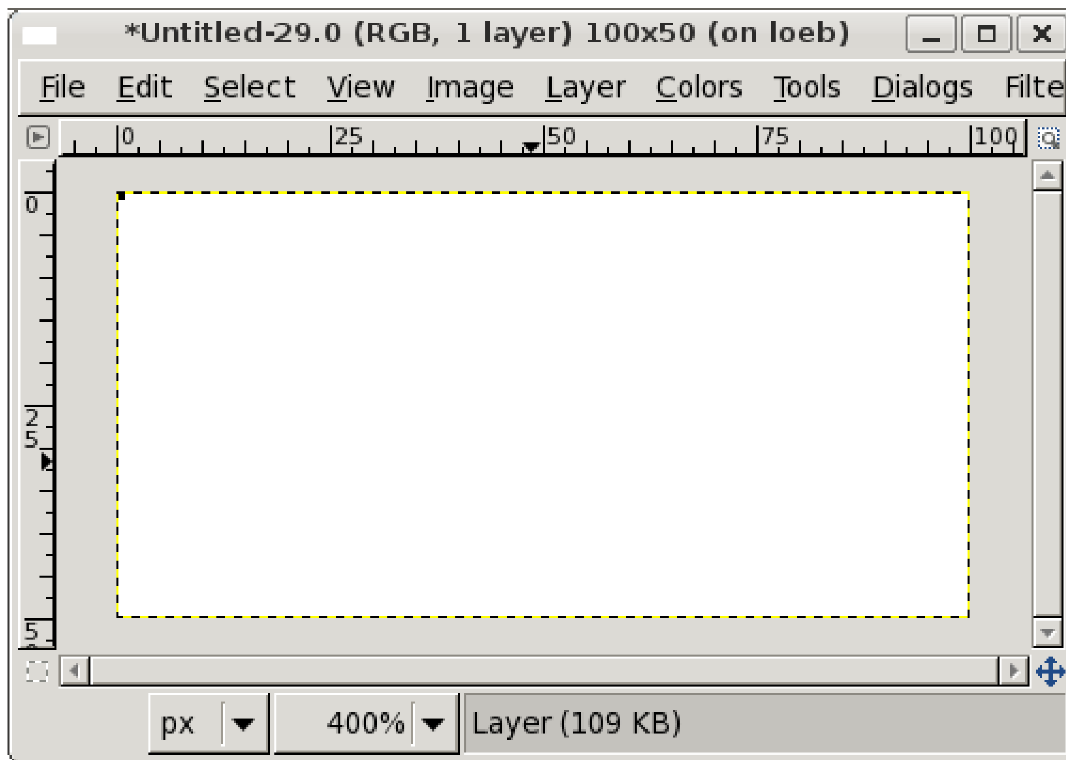
For our initial exploration of drawings as values, we'll start with two simple drawing values: the unit circle and the unit square, both centered at (0,0), and drawn in black. Why start with these two values? They're easy to think about, easy to draw, and relatively straightforward to think about.

In Scheme, we'll represent these two values as `drawing-unit-circle` and `drawing-unit-square`. Note, these are *values*, not procedures. That is, you will not place them immediately after an open paren. Rather, you will use them in various procedures that take drawings as parameters.

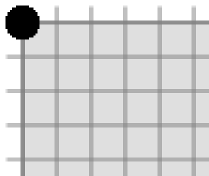
What procedures will take drawings as parameters? In the subsequent sections, we'll consider a number of procedures that build new drawings from old. But there's one other very important one: `(drawing->image` *`drawing width height`*`)` will convert a drawing to an image that we can display. For example, we could show the simplest circle drawing on a 100x50 canvas with

```
> (image-show (drawing->image drawing-unit-circle 100 50))
```

Of course, the unit circle is relatively small (diameter 1), so its rendering on the 100x50 canvas is essentially invisible, even when we zoom in.



Nonetheless, we can envision the actual drawing that this image represents.

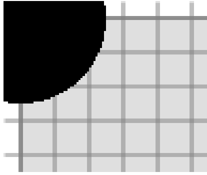


Could we have other basic values in addition to the unit circle and the unit square? Certainly. We might want lines, other kinds of polygons, more complex curves, and so on and so forth. However, for now, our exercises in design will involve only variations of these two values.

Scaling Drawings

If all that we can draw are the unit circle and unit square, and only draw them in one position, we're in a bit of trouble. So let's think of how we might vary them. Well ... both are fairly small, so we might want to *scale* them. Hence, we should provide a procedure that scales drawings. (It will scale the two basic drawings, but as we come up with other drawings, it will scale those, too.) That procedure will be called (`drawing-scale` *`drawing factor`*), and, given a drawing, will return a scaled version of the drawing.

When we scale the unit circle by 3, we can envision getting something like



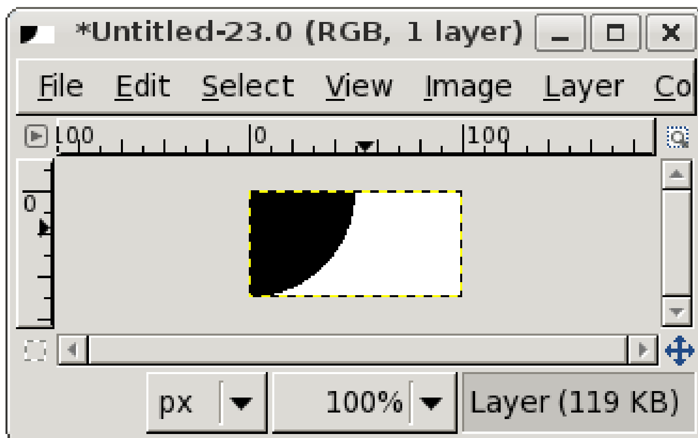
Once again, when rendered on a real image, that will be small enough to be difficult to see. Hence, in practice, we are more likely to make a bigger circle (e.g., with radius 50) with

```
> (define big-circle (drawing-scale drawing-unit-circle 100))
```

And we could build an image for that circle, too.

```
> (image-show (drawing->image big-circle 100 50))
```

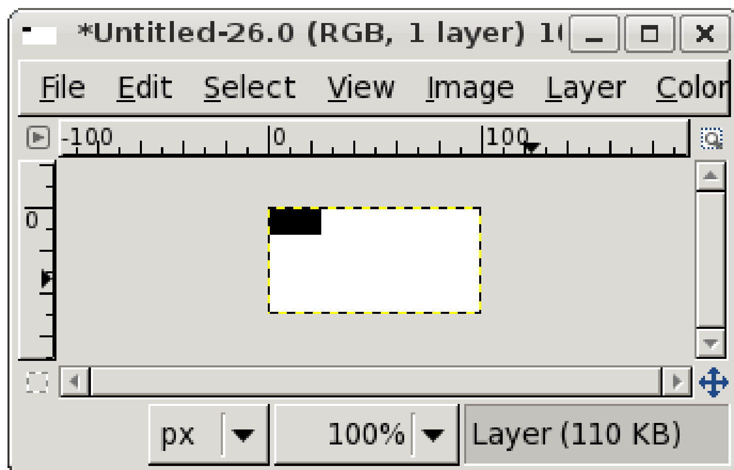
In the GIMP, we would then see something like the following.



Note that `image-scale` does not change the underlying drawing. Rather, it builds a new one, just like those machines they advertise on television.

To mix things up a bit, let's say that in addition to scaling drawings in both directions (horizontally *and* vertically), we can also scale them in a single direction (horizontally *or* vertically). We'll call the procedures to do so `drawing-hscale` and `drawing-vscale`. So, to make a rectangle that is 50 wide and 25 high, we might write

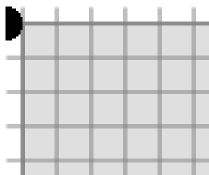
```
> (define rect (drawing-vscale (drawing-hscale drawing-unit-square 50) 25))
```

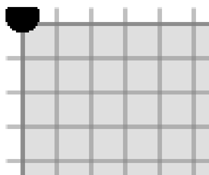


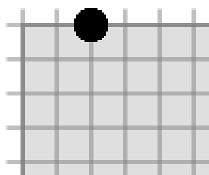
Shifting Drawings

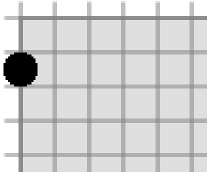
But we don't just want our images centered at (0,0). Hence, we should also be able to *shift* them around. We'll use `(drawing-hshift` *`drawing amt`*`)` to shift the drawing to the right (or to the left, if *`amt`* is negative) and `(drawing-vshift` *`drawing amt`*`)` to shift the drawing downward (or upward, if *`amt`* is negative). Once again, these don't really shift the original drawing. Rather, they make new copies of the drawing.

At a small scale, we can envision shifting our unit circle left one-half unit, up one-quarter unit, right two units, or down one-and-a-half units.







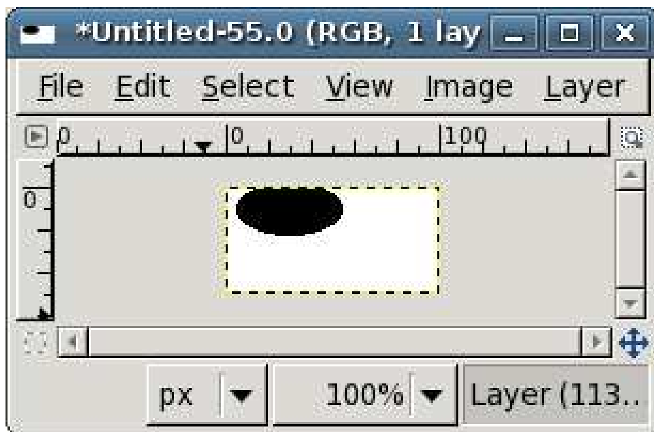


We can also work with previously-created drawings. In the following, `low-circle` will be centered at (0,25) and `right-circle` will be centered at (60,0). That is, the call to `drawing-vshift` on `big-circle` does not affect the position of `big-circle`. It is forever centered at (0,0)

```
> (define low-circle (drawing-vshift big-circle 25))
> (define right-circle (drawing-hshift big-circle 60))
```

We can now describe drawings that consists of a single black square, a single black rectangle, a single black circle, or a single black ellipse that falls anywhere on the canvas. For example, a black ellipse of width 50 and height 25, centered at (30,10) can be built from the unit circle as follows:

```
> (define sample-ellipse
    (drawing-vshift
      (drawing-hshift
        (drawing-vscale
          (drawing-hscale
            drawing-unit-circle
            50)
          25)
        30)
       10))
```

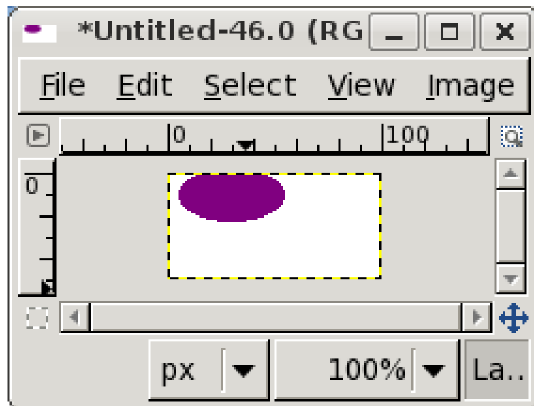


Yes, that code is a bit long. However, one of the intellectual challenges of algorithm design (and of programming) is finding creative ways to use the basic operations you've been given. A consequence is that you'll sometimes write long code. And, as you'll see in a few days, if you find you're regularly writing similar long code, there are elegant ways to combine a group of small operations into a single big operation.

Making Drawings More Lively: Adding Color and Outlines

But the drawings we can describe are still black. Let's make them a bit more lively. In particular, let's add a procedure that recolors drawings. (That is, makes a copy of the drawing in another color.) We'll call that procedure `(drawing-recolor drawing newcolor)`.

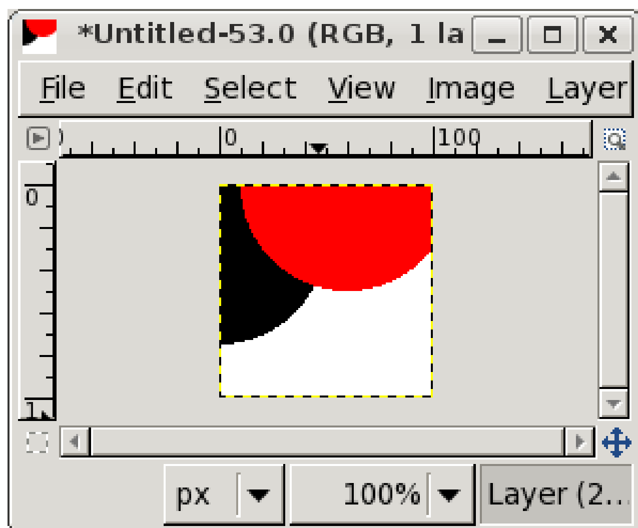
Our ellipse, recolored in purple, might then be rendered as follows.



Combining Drawings

Our repertoire for describing drawings has expanded significantly. We've can now describe circles, squares, rectangles, and ellipses in a variety of colors and at any size and any location. But a single shape rarely makes a compelling drawing. Hence, we'll want to combine drawings into new drawings. The procedure `(drawing-group drawing1 drawing2)` combines drawings for us. We might render the two circles we created above with

```
> (image-show (drawing->image (drawing-group low-circle (image-recolor right-circle "red")) 100 100))
```



Now, here's the really fun part. If we agree that a group of superimposed drawings is also a drawing, then we can do anything to that group that we did to the simpler drawings: We can recolor it, scale it, shift it, and combine it with other groups. Together, those different features will allow us to create some fairly interesting drawings. For example, once we've described a single face, we might make new copies of the face in different colors, at different locations, stretched in various ways, and so on and so forth.

Detour: Pure Operations

As you might guess, one of the reasons we've looked at drawings in terms of operations that build new drawings from old is to encourage you to think in this new way about data. (Just as our Mathematician friends think of integers in terms of a basic value and the operation that creates new integers, you can now think of drawings as basic values and operations that create new drawings.)

However, there's another reason that we like this way of describing drawings: All of the operations are *pure*: They do not modify the underlying value they are applied to. Rather they build new values. Contrast this to the various image operations you've learned. If you tell GIMP to switch the foreground color, GIMP's "state" changes, and the old foreground color is lost. In contrast, if you recolor a drawing, the original drawing is still in the old color. There are both advantages and disadvantages to working with pure operations, but, like working with a small set of operations, it's a useful intellectual challenge.

By the way, here's one great advantage of working with pure operations: Since the original values are still around somewhere, you never need to undo an action.

Drawings as Scheme Values, Revisited

At the beginning of this reading, we noted that we often specify data types by providing the basic data values and operations that build new values from other values. Let's review the ones that we've come up with.

We have two basic drawing values.

- `drawing-unit-circle` is a drawing. It represents the unit circle (diameter 1, centered at (0,0)).
- `drawing-unit-square` is a drawing. It represents the unit square (edge length 1, centered at (0,0)).

We have a variety of procedures that operated on drawings.

- If *`d`* is a drawing and *`factor`* is a real number, then `(drawing-scale d factor)` is a drawing that represents a scaled version of *`d`*.
- If *`d`* is a drawing and *`factor`* is a real number, then `(drawing-hscale d factor)` is a drawing that represents a horizontally scaled version of *`d`*.
- If *`d`* is a drawing and *`factor`* is a real number, then `(drawing-vscale d factor)` is a drawing that represents a vertically scaled version of *`d`*.
- If *`d`* is a drawing and *`offset`* is a real number, then `(drawing-hshift d offset)` is a drawing that represents *`d`* shifted right by *`offset`*.
- If *`d`* is a drawing and *`offset`* is a real number, then `(drawing-vshift d offset)` is a

drawing that represents `d` shifted down by `offset`.

- If `d` is a drawing and `c` is a color, then `(drawing-recolor d c)` is a drawing that represents `c` redrawn in color `c`.
- If `d1` is a drawing and `d2` is a drawing, then `(drawing-group d1 d2)` is a drawing that represents the overlay of `d2` over `d1`.

Copyright (c) 2007-2009 Janet Davis, Matthew Kluber, Samuel A. Rebelsky, and Jerod Weinman. (Selected materials copyright by John David Stone and Henry Walker and used by permission.)

This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/2.5/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.