The Design of a Functional Image Library

Ian Barland
Radford University
ibarland@radford.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Abstract

We report on experience implementing a functional image library designed for use in an introductory programming course. Designing the library revealed subtle aspects of image manipulation, and led to some interesting design decisions. Our new library improves on the earlier Racket library by adding rotation, mirroring, curves, new pen shapes, some new polygonal shapes, as well as having a significantly faster implementation of `equal?`.

Keywords functional image library, image equality

1. Introduction

This paper reports on Racket's (Flatt and PLT June 7, 2010) latest functional image library, `2htdp/image`. The library supports 2D images as values with a number of basic image-building functions for various shapes like rectangles and ellipses, as well as combinations like images overlaid on each other, rotated images, and scaled images. The image library is designed to be used with *How to Design Programs*, starting from the very first exercises, while still being rich enough to create sophisticated applications.

An earlier, unpublished version of Racket's image library had a number of weaknesses that this library overcomes.

- Image equality testing was far too slow.
- Overlaying images off-center was sometimes unintuitive to beginners.
- Rotation and reflection were not supported.

When images are regarded as immutable values (rather than a side-effect of drawing routines), then unit tests are easier to create, and the question of equality plays a particularly prominent role. For example, when writing a video game (using the `2htdp/universe` library (Felleisen et al. 2009)) one might write a function `draw-world : world → image` and create unit tests similar to:

```
(check-expect
 (draw-world (move-left initial-world))
 (overlay/xy player -5 0 initial-image))
```

For beginners using DrRacket, any poor performance of image equality in unit tests becomes apparent, since the test cases are included in the source code and are evaluated with each update to their code. One teacher reported that a student's test-cases for a tic-tac-toe game took approximately 90 seconds with the previous version of the library. Improvements to the library helped considerably, achieving (in that particular case) approximately a five-hundred-fold speedup.

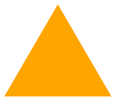
2. The `2htdp/image` Library API

The `2htdp/image` library's primary functions consist of:

- constructors for basic images:

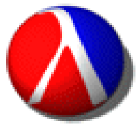
```
> (rectangle 60 30 "solid" "blue")
```

```
> (triangle 50 "solid" "orange")
```



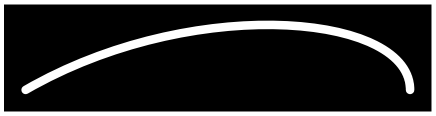
```
> (text "Hello World" 18 "forest green")
```

```
> (bitmap icons/plt-small-shield.gif)
```



- operations for adding lines and curves onto images:

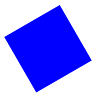
```
> (add-curve
   (rectangle 200 50 "solid" "black")
   10 40 30 1/2
   190 40 -90 1/5
   (make-pen "white" 4
             "solid" "round" "round"))
```



(Lines are specified by end points; curves are specified by end points each augmented with an angle to control the initial direction of the curve at that point, and, intuitively, a "pull" to control how long the curve heads in that direction before turning towards the other point. More precisely, the angle and pull denote a vector: the difference between the endpoint and its adjacent control point for a standard Bezier curve.)

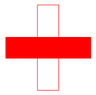
- an operation for rotating shapes:

```
> (rotate 30 (square 30 "solid" "blue"))
```

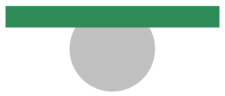


- operations for overlaying shapes relative to their bounding boxes:

```
> (overlay
   (rectangle 40 10 "solid" "red")
   (rectangle 10 40 "outline" "red"))
```

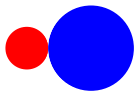


```
> (overlay/align
   "middle" "top"
   (rectangle 100 10
              "solid" "seagreen")
   (circle 20 "solid" "silver"))
```

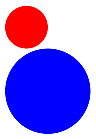


- putting images above or beside each other:

```
> (beside (circle 10 "solid" "red")
          (circle 20 "solid" "blue"))
```

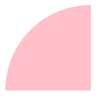


```
> (above/align "left"
               (circle 10 "solid" "red")
               (circle 20 "solid" "blue"))
```



- cropping shapes to a rectangle:

```
> (crop 0 0 40 40
        (circle 40 "solid" "pink"))
```



- flipping and scaling images:

```
> (above
   (star 30 "solid" "firebrick")
   (scale/xy
    1 1/2
    (flip-vertical
     (star 30 "solid" "gray"))))
```

- and equality testing:

```
> (equal?
   (rectangle 40 20 "outline" "red")
   (rotate
    90
    (rectangle 20 40 "outline" "red")))
#t
```

The library includes many additional, related functions for dealing with pen styles, colors, framing images, width, height, and (for drawing text) baseline of images, as well as a number of different kinds of polygons (triangles, regular polygons, star polygons, rhombuses, etc). The full `2htdp/image` API is a part of the Racket documentation (The PLT Team 2010).

3. From `htdp/image` to `2htdp/image`

For those familiar with the earlier library `htdp/image` of Racket (formerly PLT Scheme), this section gives a brief overview of the conceptual changes and a rationale for them. The new version can largely be seen as simply adding features: a few more primitive shapes, as well as some more combinators such as `overlay/align`, `rotate`, functions for scaling and flipping. However, the original library did include two concepts which the new version has jettisoned: pinholes, and scenes. Also, the new library changes the semantics of `overlay`.

3.1 No More Pinholes

An image's *pinhole* is a designated point used by the original library to align images when overlaying them. Imagine sticking a push-pin through the images, with the pin passing through each pinhole. The pinhole can be interpreted as each image's local origin. The primitive image constructors (mostly) created images whose pinhole was at their center, so the original `(overlay img1 img2)` tended to act as the new version's `(overlay/align img1 "center" "center" img2)`.

Sometimes this default method of overlaying was intuitive to students (e.g. when overlaying concentric circles or concentric rectangles), but sometimes it wasn't (e.g. when trying to place images next to each other, or aligned at an edge). While this was a teaching moment for how to calculate offsets, in practice these calculations were cluttered; many calls to `overlay/xy` would either include verbose expressions like `(- (/ (image-height img1) 2) (/ (image-height img2) 2))`, repeated again for the width, or more likely the student would just include a hard-coded approximation of the correct offset. While functions to retrieve and move an image's pinhole were provided by the library, most students found these less intuitive than calling `overlay/xy`.

Pinholes are not included in our new library, although they might make a comeback in a future version as an optional attribute, so that beginners could ignore them entirely.

3.2 No More Scenes

The original library had attempted to patch over the pinhole difficulties by providing the notion of a *scene*—an image whose pinhole is at its northwest corner. The library had one constructor for scenes, `empty-scene`; the overlay function `place-image` required its second image to be a scene, and itself returned a scene. This often led to confusing errors for students who weren't attuned to the subtle distinction between scenes and ordinary images (and thought that `place-image` and `overlay/xy` were interchangeable). Part of the confusion stemmed from the fact that an image's pinhole was invisible state. The new library dispenses with notion of scenes, and includes `overlay/align` to make image placement natural for many common cases.

3.3 Changes to `overlay`

In `htdp/image`, the arguments to `overlay` were interpreted as "the first image is *overlaid with* the second." Students were repeatedly confused by this, taking it to mean "the first image is overlaid onto the second;" we changed `overlay` to match the latter interpretation, and provided a new function `underlay` for the natural task of placing images onto a background (see Section 4).

4. Other API Considerations

We discuss the rationale for other decisions made about what to (not) include in the API, several involving issues in overlaying images.

- **coordinates for `overlay/xy`** There are several ways to combine two images in `2htdp/image`, including:
  - `overlay/align` lets the caller to specify how the images are aligned, and is sufficient for several common cases.
  - The default version `overlay` uses the center of each image.
  - When more precise placement is required, `overlay/xy` specifies how to align the images using image-relative coordinates.

  There was discussion of whether `overlay/xy` should consider each image's origin to be the northwest corner with increasing *y* coordinates moving down, (consistent with most computer graphics libraries), or the center of each image with increasing *y* coordinates moving up (avoiding a privileged corner, consistent with `overlay`'s default assumption, and arguably in closer harmony with mathematics).

  The final decision was to have indeed have `overlay/xy` use the northwest corner. Even in a pedagogic setting where we strive to strengthen the connection between math and programming, it was felt we also have some duty to teach the conventions ubiquitous in computing, such as this coordinate system.

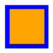
  Note that another related function, `place-image`, is also provided; it differs from `overlay/xy` in two ways: `(place-image img1 dx dy img2)` first places the *center* of `img1` offset from the `img2`'s northwest corner by `dx`,`dy`. Second, it crops the result so that the resulting bounding box is the same as `img2`'s. (See Figure 1 below.) The function `place-image` is intended for the common case where the second image argument is regarded as a background or a window for an object of interest. This asymmetry of purpose is reflected in the asymmetry of the alignment conventions.

- **`underlay` vs. `overlay`** The new library includes both `underlay` and `overlay` functions, which do the same thing but take their arguments in different order: `(overlay img1 img2)` is equivalent to `(underlay img2 img1)`.

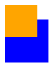
  Providing both `overlay` and its complement `underlay` initially seems a bit redundant; after all the library provides `above` and `beside` yet no complements such as `below` or `beside/right` (which would only differ in swapping the order of their arguments). The reason `underlay` is included is that `(overlay/xy img1 dx dy img2)` (which overlays `img1`'s coordinate `dx`,`dy` on top of `img2`'s origin), would require negative coordinates for the common task of "place `img1`'s origin on top of `img2`'s coordinate (`dx`,`dy`)," in addition to swapping the order of its arguments. (See Figure 1.) This situation was deemed common enough that it was decided to provide both versions.

- **`rotate` needs no center of rotation** It was suggested by several contributors and helpers that `rotate` must specify the point of rotation. However, this doesn't actually fit the model of images as values: images are images without any enclosing frame-of-reference; rotating about the lower-left is the same as rotating about the center. (Of course, when the implementation is rotating a composed image, we rotate each sub-part and then worry about how to re-compose them.)

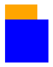
```
> (overlay (square 15 "solid" "orange")
           (square 20 "solid" "blue"))
```



```
> (overlay/xy (square 15 "solid" "orange")
              0 7
              (square 20 "solid" "blue"))
```



```
> (underlay/xy (square 15 "solid" "orange")
               0 7
               (square 20 "solid" "blue"))
```



```
> (place-image (square 15 "solid" "orange")
               0 7
               (square 20 "solid" "blue"))
```

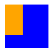


Figure 1: `overlay/xy`, and a motivation for `underlay/xy`

5. Implementation

We discuss the implementation of the image library, focusing on unexpected difficulties and issues, as well as the rationale for certain choices. We present the data representations used, then the algorithm for implementing equality, and finish with assorted issues involving rotation and cropping.

5.1 Internal Representation

Internally, an image is represented by a pair of a bounding box and a shape. A shape is a tree where the leaf nodes are the various basic shapes and the interior nodes are overlaid shapes, translated shapes, scaled shapes, and cropped shapes. In the notation of Typed Scheme (Tobin-Hochstadt 2010; Tobin-Hochstadt and Felleisen 2008) (where "`U`" denotes a union of types, and "`Rec`" introduces a name for a recursive type), this is the type for images:

```
(image
 (bounding-box width height baseline)
 (Rec Shape
      (U Atomic-Shape  ; includes ellipses,
                       ; text, bitmaps, etc
         Polygon-Shape ; includes rectangles,
                       ; lines, curves, etc
         (overlay Shape Shape)
         (translate dx dy Shape)
         (scale sx sy Shape)
         (crop (Listof point) Shape))))
```

where the various record constructors (`image`, `bounding-box`, `overlay`, `point`, etc) are not shown. The `crop`'s `(Listof point)` always form a rectangle.

5.2 Defining Equality

Checking whether two shapes are equal initially seems straightforward: just check whether they are the same type of shape, and have the same arguments. However, upon reflection, there are many cases where differently-constructed shapes should still be considered equal. Recognizing and implementing these was a significant source of effort and revision.

Intuitively, two images should be equal when no operations on the images can produce images that behave differently. That is, the two images should be observationally equivalent. In our case, this means that the images should draw the same way after any amount of rotation or scaling, or after any amount of overlays with equal images.

A natural technique for implementing this form of equality is to represent the shapes as (say) a tree of constructor calls (or perhaps a sequence of translated, rotated primitive shapes), and implement equality as a recursive traversal of the shapes. However, there were quite a few difficulties encountered with this simple-seeming approach.

For polygons, there are a number of different ways to represent the same shape. For example, these four images should be equal:

- `(rectangle 10 20 "outline" "blue")`

-
  ```
  (rotate 90
          (rectangle 20 10 "outline" "blue"))
  ```
- a polygon connecting (0,0), (10,0), (10,20), (0,20)
- four entirely disjoint line segments rotated and placed above or beside each other to achieve the same rectangle.

One could take this as an indication that all polygon shapes should be represented as a collection of line segments where ordering is only relevant if the line segments overlap (and are different colors).

Worse, our image library supports shapes that can have a zero width or a zero height. One might imagine that the image equality can simply ignore such images but, in the general case, they can contribute to the bounding box of the overall image. For example, consider a 10×10 square with a 20×0 rectangle next to it. The bounding box of this shape is 20×10 and thus the overlay operations behave differently for this shape than they do for just the 10×10 square alone.

Even worse, consider a 10×10 black square overlayed onto the left half of a 20×10 red rectangle, as opposed to a 10×10 red square overlayed onto the right half of a 20×10 black rectangle. Or, overlaying a small green figure top of a larger green figure in such a way that the small green figure makes no contribution to the overall drawn shape.

One might conclude from these examples that the overlay operation should remove the intersections of any overlapping shapes. We did briefly consider adding a new operation to pull apart a compound shape into its constituent shapes, thereby adding a new sort of "observation" with which to declare two shapes as different, under the notion of observational equivalence.

Yet even worse, the ability to crop an ellipse and to crop a curve means that we must be able to compute equality on some fairly strange shapes. It is not at all obvious whether or not two given curves are equal to one curve that has been cropped in such a way as to appear to be two separate curves.

While these obstacles all seem possible to overcome with a sufficient amount of work, we eventually realized that the students will have a difficult time understanding why two shapes are not equal when they do draw the same way at some fixed scale. Specifically, students designing test cases may write down two expressions that evaluate to images that appear identical when drawn as they are, but are different due to the reasons above. The right, pedagogically motivated choice is to define equality based on how the two images draw as they are, and abandon the idea of fully observationally equivalent images.[1]

There are still two other, subtle points regarding image equality where images that look very similar are not `equal?`. The first has to do with arithmetic. When shapes can be rotated, the computations of the verticies typically requires real numbers which, of course, are approximated by IEEE floating point numbers in Racket. This means that rotating a polygon by 30 degrees 3 times is not always the same as rotating it by 45 degrees twice. To accomodate this problem, the the image library also supports an approximate comparison where students can specify a tolerance and images are considered the same if corresponding points in the normalized shapes are all within the tolerance of each other.

The second issue related to equality is the difference between empty space in the image and space that is occupied but drawn in white. For example, a 20×10 white rectangle looks the same as a 20×0 rectangle next to a 10×10 white rectangle when drawn on a white background, but not on any other color. We decided not to consider those images equal, so the equality comparison first draws the two images on a red background and then draws the two images on a green background. If they look different on either background, they are considered different.

5.3 Implementing Equality

Unfortunately, defining equality via drawing the images means that equality is an expensive operation, since it has to first render the images to bitmaps and then compare those bitmaps, which takes time proportional to the square of the size of the image (and has a large constant factor when compared to a structural comparison).

Since students used this library for writing video games, unit-testing their functions could easily involve screen-sized bitmaps; this slow performance was noticeable enough that it discouraged students from writing unit tests. Slow performance was especially painful for students who have a single source file which includes their unit tests, since the tests are re-interpreted on each change to their program, even if the change does not affect many of the tested functions.

Ultimately, we settled on a hybrid solution. Internally, we normalize shapes so that they are represented as `Normalized-Shape`s, according to this type definition ("CN" for "cropped, normalized"):

```
(Rec Normalized-Shape
     (U (overlay Normalized-Shape CN-Shape)
        CN-Shape))
(Rec CN-Shape
     (U (crop (Listof point)
              Normalized-Shape)
        (translate num num Atomic-Shape)
        Polygon-Shape))
```

Note that the overlay of two other overlays is "linearized" so that the second shape is not an (immediate) overlay[2]. A non-translated `Atomic-Shape` is represented with a translation of *(0,0)*. This normalization happens lazily, before drawing or checking equality (not at construction, or else we wouldn't have constant-time `overlay`, etc).

Once the shapes are normalized, the equality checker first tries a "fast path" check to see if the two shapes have the same normalized form. If they do, then they must draw the same way so we do not have to actually do the drawing. If they do not, then the equality test creates bitmaps and compares them. While this only guarantees

[1] A related point has to do with fonts, specifically ligatures. A sophisticated user might expect the letters "fi", when drawn together, to look different than an image of the letter "f" placed beside an image of the letter "i", due to the ligature. Since we expect this would confuse students, should they stumble across it, we use the simpler conceptual model, and break the text into its constitutent letters and draw them one at a time, defeating the underlying GUI platform's ligatures (and possibly the font's kerning). If this ends up surprising the sophisticated, it would not be difficult to add a new text-constructing operation that does not do this, and thus would have proper ligatures (and kerning).

[2] At first blush, it seems that if two overlaid shapes don't actually overlap, it shouldn't matter which order they are stored in, internally. Surprisingly this is not the case, for our definition of observationally equivalent: If the entire image is scaled down to a single pixel, then the color of one of the two shapes might be considered to "win" to determine the color of that pixel.

Library	Time	Speedup
Original library	9346 msec	
`2htdp/image` library, without fast path	440 msec	21x
`2htdp/image` library, with fast path	18 msec	509x

Figure 2: Timing a student's final submission, run on a Mac Pro 3.2 GHz machine running Mac OS X 10.6.5, Racket v5.0.0.1

equality at the particular scale the bitmap is created, using the normalized form means that simple equalities are discovered quickly. For example, two shapes that are overlaid and then rotated will be equal to the two shapes that are rotated individually and then overlaid.

Overall, the image equality comparison in `2htdp/image` is significantly faster than in the previous version of the library, for two reasons. First, it offloads drawing of the bitmaps to the graphics card (via the underlying OS) instead of doing computations on the main cpu via Racket, and second the fast-path case of checking the normalized forms frequently allows for a quick result in many situations (specifically, when the majority of a student's test cases build the expected-result in the same way that their code builds the actual result, and the test succeeds). Figure 2 shows the speedup for a program from a student of Guillaume Marceau's when he was at the Indian Institute of Information Technology and Management in Kerala (where their machines appear to have been significantly slower than the machine the timing tests were run on so these optimizations would be even more appreciated). Those timings, involving image-equality tests for drawing a tic-tac-toe board, are not representative of a general benchmark (since they don't involve any user bitmaps), but do illustrate a real-world case that motivated part of the library re-design.

5.4 Implementing Scaling and Rotation

When scaling or rotating most types of atomic shapes, the appropriate transformations are applied to the shape's defining vertices, and a new shape is returned.

However, scaling and rotation of bitmaps and ellipses are handled differently from other atomic shapes: if a bitmap is repeatedly re-sampled for repeated rotation or scaling, significant artifacts easily accrue. Instead, we just store the original bitmap with its "cumulative" scale factor and rotation (implementing, in essence, the bookkeeping sometimes done by a client's program in some side-effecting graphics libraries). Each time the bitmap is actually rendered, one rotation and scaling is computed, and cached. This approach avoids accumulating error associated with re-sampling a bitmap, at the cost of doubling the memory (storing the original *and* rotated bitmap).

5.5 `rotate`'s Time Complexity is Linear, Not Constant

While developing the library, one goal was to keep operations running in constant time. This is easy for `overlay`, `scale`, and `crop` that just build a new internal node in the shape tree. We do not know, however, how to `rotate` in constant time[3].

In particular, consider constructing a shape involving n alternating `rotate`s and `overlay`s: The `overlay` functions require knowing a bounding box of each child shape, but to rotate a compound shape we re-compute the bounding box of each sub-shape, which recursively walks the entire (tree) data structure, taking linear time. As an example, see Figure 3, where a sequence of calls to `rotate` and `above` gives a figure whose bounding box is difficult to determine.

[3] Even disregarding the time to rotate a bitmaps, where it is reasonable to require time proportional to its area.

```
> (define r (rectangle 20 10 "solid" "red"))
> (define (rot-above p)
    (above (rotate 30 p) r))
> (rot-above
   (rot-above
    (rot-above
     (rot-above
      (rot-above
       r)))))
```

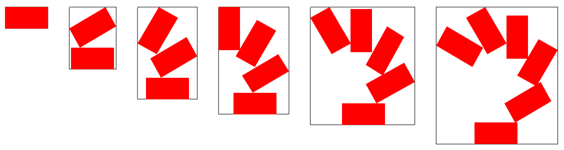


Figure 3: A difficult bounding box to compute, since each `above` wants to know the bounding box of each of its sub-shapes to find the relative (horizontal) centers. (Note that each new rectangle is added on the bottom.)

5.6 Don't Push Cropping to the Leaves

Scaling or rotating a compound shape involves pushing the scale/rotation to each of the children shapes. As seen in the definition of normalized shapes above (Section 5.3), overlays and crops are left as interior nodes in the shape (whose coordinates get scaled and rotated).

For a while during development, cropping was handled like rotating and scaling: When cropping an overlaid-shape, the crop was pushed down to each primitive shape. Thus, a shape was essentially a *list* of overlaid primitive shapes (each of which possibly rotated, scaled, or cropped). However, since two successive crops can't be composed into a single crop operation (unlike rotations and scales), repeatedly cropping a list of shapes would end up replicating the crops in each leaf of the tree. For example, normalizing

```
(crop
 r1
 (crop
  r2
  (crop
   r3
   (overlay s1 s2))))
```

resulted in

```
(overlay
 (crop r1
       (crop r2
             (crop r3 s1)))
 (crop r1
       (crop r2
             (crop r3 s2))))
```

To remove the redundancy, we modified the data definition of a normalized shape so that it is now a tree where the leaves are still primitive shapes but the nodes are overlay *or* crop operations.

5.7 Pixels, Coordinates, and Cropping

Coordinates do not live on pixels, but instead live in the infinitesimally small space between between pixels. For example, consider the (enlarged) grid of pixels show in Figure 4 and imagine building a 3×3 square. Since the coordinates are between the pixels, and we want to draw 9 pixels, we should make a polygon that has the verticies (0,0), (0,3), (3,3), and (3,0). Despite the apparent off-by-one error in those verticies, these coordinates do enclose precisely 9 pixels. Using these coordinates means that scaling the square is a simple matter of multiplying the scale factor by the verticies. If

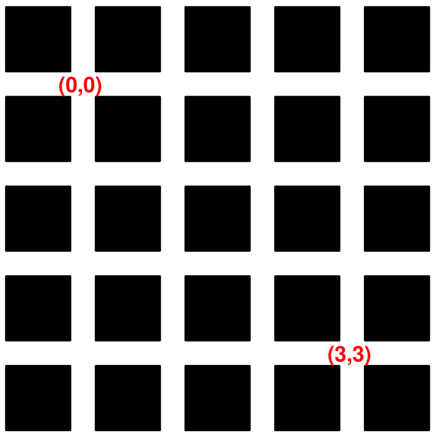


Figure 4: Pixels and coordinates

we had counted pixels instead of the edges between them, then we might have had the the polygon (0,0), (0,2), (2,2), and (2,0), which means we have to do add one before we can scale (and then subtract one after scaling) and, even worse, rotation is significantly more difficult, if not impossible (assuming we use a simple list-of-verticies representation for polygons).

While this convention for pixel-locations works well for solid shapes, drawing outlines becomes a bit more problematic. Specifically, if we want to draw a line around a rectangle, we have to actually pick particular pixels to color, as we cannot color between the pixels. We opted to round forward by 1/2 and then draw with a 1-pixel wide pen, meaning that the upper row and left-most row of the filled square are colored, as well as a line of pixels to the right and below the shape.

5.8 Bitmap Rotations: a Disappearing Pixel

Rotating a bitmap was tricky at the edges. The general approach, when creating the new bitmap, is to calculate where the new pixel "came from" (its pre-image – presumably not an exact grid point), and taking the bilinear interpolation from the original. At the borders, this includes points which are outside the original's bounding box, in which case it was treated as a transparent pixel ($\alpha = 0$).

However, although large bitmaps seemed to rotate okay, there was a bug: a 1x1 bitmap would disappear when rotated 90 degrees. The reason stemmed from treating a pixel as a sample at a grid-point rather than the center of a square. The grid-point (0,0) of the new bitmap originates from (0,-1) of the original bitmap, which is transparent. The solution we used was to treat pixels as not as a sample at a grid point *(x,y)* (as advised in (Smith 1995), and as done in most of the image library), but rather as a sample from the center of the grid square, *(x+0.5, y+0.5)*.

6. Related Work

There are a large number of image libraries that build up images functionally, including at least Functional Pictures (Henderson 1982), PIC (Kernighan 1991), MLGraph (Chailloux and Cousineau 1992), CLIM (Son-Bell et al. 1992), Functional PostScript (Sae-Tan and Shivers 1996), FPIC (Kamin and Hyatt Oct 1997), Pictures (Finne and Peyton Jones July 1995), and Functional Images (Elliot 2003). These libraries have operators similar to our `2htdp/image` library, but to the best of our knowledge they are not designed for teaching in an introductory programming course, and they do not support an equality operation.

SICP (Abelson and Sussman 1996)'s picture language (Soegaard 2007) is designed for an introductory computer science course, but does not support image equality (since test cases and unit testing do not seem to be a significant emphasis).

Stephen Bloch's extension of `htdp/image` (Bloch 2007) inspired our exploration into adding rotation to this library. Since his library is based on `htdp/image`, the rotation operator is bitmap-based, meaning it is does not produce images that are as clear.

Acknowledgments

We are deeply indebted to Jean-Paul Roy for discussions and beta testing of the API, and for helping discover the $O(n^2)$ problem with repeated cropping. We also wish to thank Guillaume Marceau for providing real-world student test-cases and general discussions on testing equality. Carl Eastlund suggested keeping pinholes as optional, advanced state for images (a feature we plan to add to a future version of this library). Thanks to the anonymous reviewers for their plentiful, detailed suggestions. Finally, a special thanks to Matthias Felleisen for frequent, thoughtful input.

Bibliography

Harold Abelson and Gerald Jay Sussman. Structure and Interpretation of Computer Programs. Second Edition edition. MIT Press, 1996.

Stephen Bloch. Tiles Teachpack. 2007. `http://planet.racket-lang.org/users/sbloch/tiles.plt`

Emmanuel Chailloux and Guy Cousineau. The MLgraph Primer. Ecole Normale Superior, LIENS - 92 - 15, 1992.

Conal Elliot. Functional Images. Palgrave Macmillan Ltd., 2003.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A Functional I/O System, or, Fun For Freshman Kids. In *Proc. Proceedings of the International Conference on Functional Programming*, 2009.

Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Proc. Proc. Glasgow Functional Programming Workshop.*, July 1995.

Matthew Flatt and PLT. Reference: Racket. June 7, 2010. `http://www.racket-lang.org/tr1/`

Peter Henderson. Functional geometry. In *Proc. Proc. ACM Conference on Lisp and Functional Programming*, 1982.

Samual N. Kamin and David Hyatt. A special-purpose language for picture-drawing. In *Proc. Proc. USENIX Conference on Domain-Specific Languages.*, Oct 1997.

Brian W. Kernighan. PIC a graphics language for typesetting, user manual. Computer science technical report. AT&T Bell Laboratories., CSTR-116., 1991.

Wendy Sae-Tan and Olin Shivers. Functional PostScript. 1996. `http://www.scsh.net/resources/fps.html`

Alvy Ray Smith. A Pixel Is not A Little Square, A Pixel Is not A Little Square, A Pixel Is not A Little Square! (And a Voxel is not A Little Cube). Microsoft, Alvy Ray Microsoft Tech Memo 6, 1995. `http://alvyray.com/Memos/6_pixel.pdf`

Jens Axel Soegaard. SICP Picture Language. 2007. `http://planet.racket-lang.org/users/soegaard/sicp.plt`

Mark Son-Bell, Bob Laddaga, Ken Sinclair, Rick Karash, Mark Graffam, Jim Vetch, and Hanoch Eiron. Common Lisp Interface Manager. 1992. `http://www.mikemac.com/mikemac/clim/regions.html#3`

The PLT Team. HtDP/2e Teachpacks: image.ss. 2010. `http://docs.racket-lang.org/teachpack/2htdpimage.html`

Sam Tobin-Hochstadt. Typed Scheme. 2010. `http://docs.racket-lang.org/ts-guide/`

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. Proceedings of Symposium on Principles of Programming Languages*, 2008.