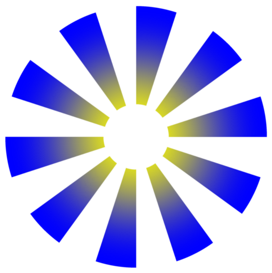


Functional Image Synthesis

Conal Elliott
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract

Software used to produce visual beauty is usually created with imperative programming languages and is typically unbeautiful itself. One fundamental reason for this situation is that these languages reflect the underlying discreteness and sequentiality of the computers that run them. The essential nature of *what an image is* becomes muddled with details of *how to display it* on a computer. We can, however, generate beautiful images with beautiful programs, by making a shift of programming paradigm, from *doing* to *being*. This claim is illustrated by many examples expressed in *Pan*, an image synthesis language with a freely available optimizing compiler.

1 Introduction

Visual artwork and program source code typically have very different properties, appealing to different modes of perception and comprehension. Art is usually percieved first as a whole and then dissected into parts for more detailed examination. Programs, on the other hand, are usually perceived in pieces from the beginning, as sequences of small operations, acting on one datum at a time. It is only after patient and repeated examination, internalization, and creative reconstruction that the whole emerges. (This difference has been noted for images vs language in general and written language in particular. See especially [16].)

Why point out this contrast? Is it of any harm that reading a program is so unlike viewing an artwork? After all, programs are meant to be understood and acted on by unfeeling, sequential machines, right? Not entirely. They also exist for humans to express ideas with clarity, to illuminate and inspire, and to be combined with other such idea expressions for forming richer results. To serve these human purposes, programs should be expressed as close as possible to the essence of the underlying ideas, stripping away incidental artifacts of the machines on which they are intended to execute.

In contrast with the prevailing tendencies of art and programs, some art forms embrace sequentiality (comics [15]), and some programming embraces the whole, in which even the smallest pieces of programs produce and transform the large and even the infinite. In this paper, I will demonstrate some of this latter form of programming in its use to create images.

In college math, we discover that spaces can have many dimensions, even infinite, and even uncountably many dimensions. We learn to talk about spaces of functions. Every "point" in such a space is an infinite thing, and yet, once accustomed, we can play with several of them at once, measuring distance between them, etc. This is a beautiful and awe-inspiring idea, even a brush with the Divine. It is an invitation to "hold Infinity in the palm of your hand," in the words of William Blake.

When conventional programming is applied to image production, the beauty is almost all in the display, very little in the program. I think that the causes for this situation are deeply rooted, and are as described by John Backus in his 1977 Turing Award lecture:

> Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor–the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs. [2]

Backus then went on to explore the harmful properties of "von Neumann" programming languages (i.e., ones that closely follow the principles of the sequential computer, pioneered by John von Neumann), and to offer an alternative, *functional programming*, drawing a compelling contrast between the two styles. Although his Turing lecture was given nearly 25 years ago, the practice of programming continues to be dominated by the "primitive word-at-a-time" style that Backus decried. I hope to persuade you that this need not be the case in image synthesis, but rather that beautiful images may be formed with beautiful programs.

This paper presents and illustrates an approach to the programming of image synthesis and transformation that is far removed from the sequential nature of the underlying computer architecture. Unlike the conventional approach, ideal images are described and manipulated directly. Even the smallest subexpressions create and manipulate entire infinite images. The task of turning the described ideal images into finite and discrete output for computer screen or paper is entirely automated, and so does not compromise the simplicity, clarity, and perhaps most importantly, the composability of the image programs.

The language used in this paper is implemented as a freely available optimizing compiler whose workings are described in [6]. It produces not just static pictures as shown here, but fast *interactive* images. I encourage you to visit the web site[1], peruse the image gallery, download and play with precompiled interactive images, and then use the language and compiler to create your own examples.

Concretely, the contributions of this work are as follows:

- A strikingly simple but precise model for resolution-independent images of any type, fitting neatly into modern typed functional languages.
- Within this model, precise and simple definitions for a library of useful image building blocks.
- Demonstration that the simple model is capable of producing a wide range of visually interesting images.

2 What is an "Image"?

When languages are given meaning precisely, it is often by being based on a precise mathematical model. A good model strips away incidental details, leaving the essence behind. A first question to consider then is

[1]`http://research.microsoft.com/~conal/pan`

what is a good model for images, that is, *what is the essence of "image"*?

One natural temptation is to confuse the essential notion with a familiar concrete embodiment. For instance, an artist or art lover might consider "image" to mean a painting or drawing, rectangular in shape, preferably framed. A photographer might come to think of images as photographic prints, or wall-projected photographic slides. A computer user or programmer might say that an image is the contents of a file using any of a number of formats, e.g., JPEG or PNG.

All of these candidates for the notion of an image carry excess baggage that comes from particular technologies of image production, storage or use. In extracting an essential notion of image, let's try to strip away these incidental artifacts.[2]

- Drop the requirement of rectangularity, allowing images to have any shape at all. Circular or oval matting is already a hint that many pictures feel uncomfortable inside rectangular frames.
- Drop the assumption of finiteness. Most representational pictures are finitely-cropped versions of a much larger scene, which the artist invites the viewer to imagine. This larger scene is an image as well, whether or not it is ever rendered physically. That larger scene may itself be part of a still larger one, and so on. Consider a piece of fruit, in a bowl, on a table, in the kitchen, in a house, on a hill, at the foot of a mountain range, in a far-away country, and so on.
- Drop the discreteness inherent in most computer-based and photographic image representations. Those *discrete* images are generally mere approximations of an underlying *continuous* image. (In other words, essential images are infinite in resolution.)

Saying that rectangularity, finiteness, and discreteness are all technological artifacts, rather than essential properties of images, is not to judge them unimportant. Artists, photographers and graphic designers are craftspeople enough not only to cope with these artifacts, but also to exploit them in creative ways, such as when the cropping of a picture imparts a feeling of confinement or of openness.

I propose the following as an essential notion of "image": an assignment of a color to every point in infinite, continuous, two-dimensional, Euclidean space. We will want to compose images out of simpler ones, so it will be useful to accommodate the desire for finite, though usually non-rectangular, images. A simple trick suffices: allow colors to have the property of partial opacity, ranging from fully transparent to fully opaque. A "finite image" then is simply one that is fully transparent outside of a finitely bounded region of space. Images may then be composed in layers, as illustrated in many examples throughout this paper. Partially transparent colors allow lower layers to show through, mixing with upper layers.

Infinity of extent and of resolution support two dual desires of an image's observer. Infinity of extent allows the observer to "zoom out", or step back from an image, getting more and more of the big picture, while losing the details. Infinity of resolution allows the observer to zoom in, or step closer, getting more and more detail while losing sight altogether of areas outside of the area of focus.

3 Expressing functions

I said that an image is an "assignment" of colors to every point in (infinite) two-dimensional ("2D") space. Fortunately, mathematics offers a precise and well-studied notion of such assignments, under the name of

[2]I do not want to be dogmatic here, arguing that there is one true essence of "image". Rather, removing these "incidental" aspects allows what I find to be a simpler semantic foundation on which to build a language. The rest of the article is offered as evidence. Note that this practice of removing inessentials may have no ultimate stopping place short of oblivion.

"function". One might express the definition of images as follows:[3]

$$\mathbf{type}\ \mathit{Image} = \ \mathit{Point}\ \rightarrow\ \mathit{Color} \quad \text{— first try}$$

where

$$\mathbf{type}\ \mathit{Point} = \ (\mathit{Float},\ \mathit{Float}) \quad \text{— Cartesian coords}$$

It is useful, however, to generalize the semantic model of images so that the range of an image is not necessarily Color, but an arbitrary type. For this reason, Image is really a type *constructor*, parameterized by an arbitrary type c:

$$\mathbf{type}\ \mathit{Image}\ c = \ \mathit{Point}\ \rightarrow\ c$$

It can also be useful to generalize the domain of images, from points in 2D space to other types (such as 3D space or points with integer coordinates), but we shall not exploit that generality in this paper.

Boolean-valued "images" are useful for representing arbitrarily complex spatial regions (or "point sets") for complex image masking. This interpretation is just the usual identification between sets and characteristic functions:

$$\mathbf{type}\ \mathit{Region}\ =\ \mathit{Image}\ \mathit{Bool}$$

As a first example, Figure 1 shows an infinitely tall vertical strip of unit width, vstrip, as defined below.[4]

$$\begin{aligned}&\mathit{vstrip}\ ::\ \mathit{Region}\\&\mathit{vstrip}\ (x,y)\ =\ |x|\ \leq\ 1/2\end{aligned}$$

For a slightly more complex example, consider the checkered region shown in Figure 2. The trick is to take the floor of the pixel coordinates and test whether the sum is even or odd. Whenever x or y passes an integer value, the parity of $\lfloor x\rfloor + \lfloor y\rfloor$ changes.

$$\begin{aligned}&\mathit{checker}\ ::\ \mathit{Region}\\&\mathit{checker}\ (x,y)\ =\ \mathit{even}\ (\lfloor x\rfloor\ +\ \lfloor y\rfloor)\end{aligned}$$

Images need not have straight edges and right angles. Figure 3 shows a collection of concentric black&white rings. The definition is similar to $\mathit{checker}$, but uses the distance from the origin to a given point, as computed by distO.

$$\mathit{altRings}\ p\ =\ \mathit{even}\ \lfloor \mathit{distO}\ p\rfloor$$

[3]All definitions in this paper are expressed in Haskell [13]. Comments follow " — ". We take some small liberties with notation. As described in [6], our implementation really uses "expression types" with names like FloatE instead of Float, in order to optimize and compile Pan programs into efficient machine code. Operators and functions are overloaded to work on expression types where necessary, but a few require special names, such as "$==*$" and "notE". The definitions used in this paper could, however, be used directly as a valid but less efficient implementation.

[4] Each figure shows an origin-centered finite window onto an infinite image and is annotated with the width of the window in logical coordinates. For instance, this figure shows the window $[-7/2, 7/2] \times [-7/2, 7/2]$ onto the infinite vstrip image.

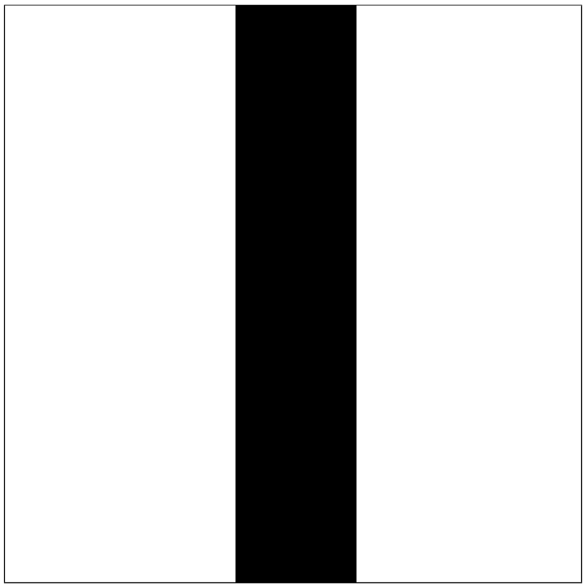

$[width = 7]$

Figure 1 *vstrip*

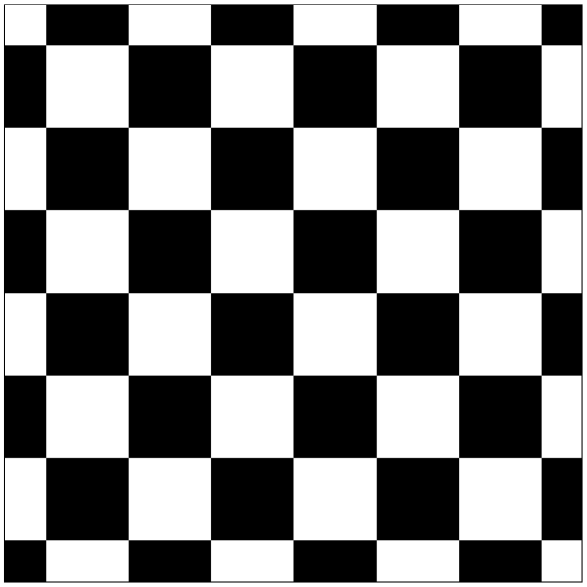

$[width = 7]$

Figure 2 *checker*

The distance-to-origin function is also easy to define:

$$distO\ (x, y)\ =\ \sqrt{x^2 + y^2}$$

It is often more convenient to define images using polar coordinates ($\rho,\ \theta$) rather than rectangular coordinates (x, y). The following definitions are helpful.

```
$type\ PolarPoint\ =\ (Float,\ Float)$

$fromPolar :: Point \rightarrow PolarPoint$
$toPolar\ \ \ \ :: PolarPoint \rightarrow Point$
$fromPolar(\rho, \theta)\ =\ (\rho \cdot cos\,\theta,\ \rho \cdot sin\,\theta)$
$toPolar\ \ \ \ (x, y) =\ (distO\ (x, y),\ atan2\ y\ x)$
```

Figure 4 shows a "polar checkerboard", defined using polar coordinates. The integer parameter n determines the number of alternations, and hence is twice the number of slices.[5] (We will see a simpler definition of *polarChecker* in Section 8.)

```
$polarChecker :: Int \rightarrow Region$
$polarChecker\ n\ =\ checker \circ sc \circ toPolar$
  **where**
    $sc\ (\rho, \theta)\ =\ (\rho,\ \theta \cdot fromInt\ n/\pi)$
```

For grey-scale images, we can use as "pixel" values in the range in the real interval $[0, 1]$. This constraint is not expressible in the type system of our language, but as a reminder, we introduce the type synonym *Frac*:

[5] The "$\circ$" operator is function composition; and *fromInt* turns an integer into some other type, here *Float*.

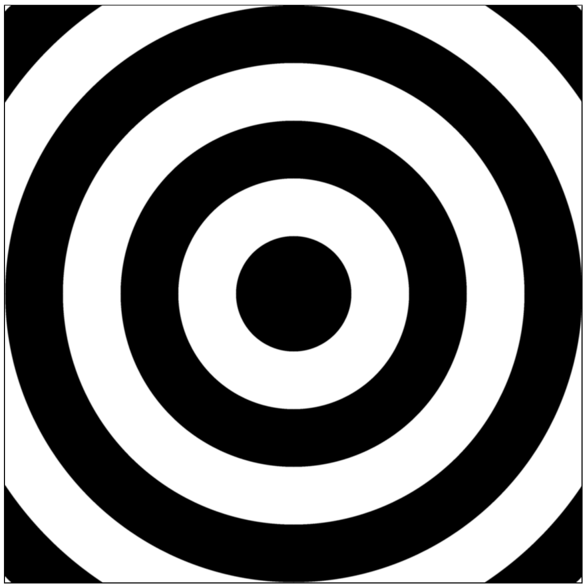


Figure 3 *altRings*

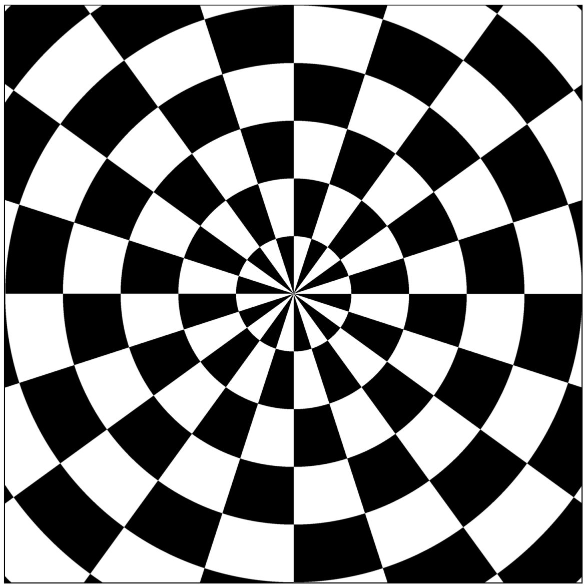


Figure 4 *polarChecker* 10

$\textbf{type}\ \mathit{Frac} = \mathit{Float}$ — In $[0, 1]$

Figure 5 shows a wavy grey-scale image that shifts smoothly between white (zero) and black (one) in concentric rings.

$$\begin{aligned}&\mathit{wavDist} :: \mathit{Image\ Frac}\\&\mathit{wavDist}\ p = (1 + \mathit{cos}\ (\pi \cdot \mathit{distO}\ p))\ /\ 2\end{aligned}$$

4 Colors

Pan colors are quadruples of real numbers in $[0, 1]$, with the first three components for blue, green, and red (BGR) components, and the last for transparency ("alpha"):

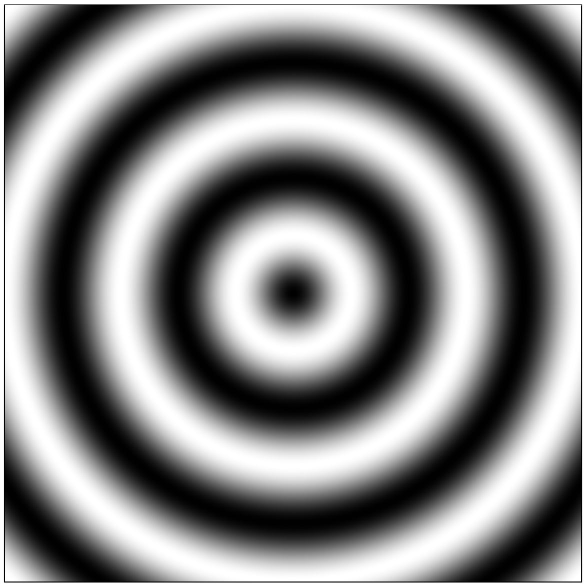
$\textbf{type}\ \mathit{Color} = (\mathit{Frac}, \mathit{Frac}, \mathit{Frac}, \mathit{Frac})$ — BGRA

The blue, green, and red components will have alpha multiplied in already, and so must less than or equal to alpha (i.e., we are using "pre-multiplied alpha" [18]). Given this constraint, there is exactly one fully transparent color:

$\mathit{invisible} = (0, 0, 0, 0)$

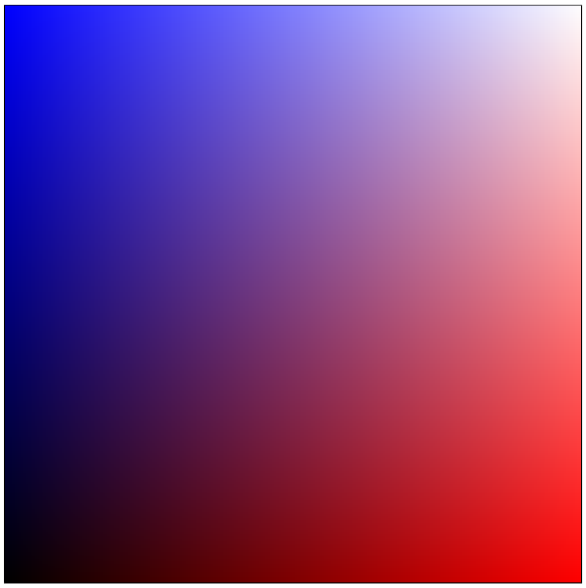
We are now in a position to define some familiar (completely opaque) colors:

$\mathit{red} = (0, 0, 1, 1)$



$[width = 10]$

Figure 5 $wavDist$



$[width = 1]$

Figure 6 $bilerpBRBW$

$$
\begin{aligned}
&green = (0,\ 1,\ 0,\ 1)\\
&\ldots
\end{aligned}
$$

It is often useful to interpolate (“lerp”) between colors, to create a smooth transition through space or time. This is the purpose of $lerpC\ w\ c_1\ c_2$. The first parameter w is a fraction, indicating the relative weight of the color c_1. The weight assigned to the second color c_2 is $1 - w$:

```
$lerpC\ ::\ Frac \rightarrow Color \rightarrow Color \rightarrow Color$
$lerpC\ w\ (b_1,\ g_1,\ r_1,\ a_1)\ (b_2,\ g_2,\ r_2,\ a_2) = (h\ b_1\ b_2,\ h\ g_1\ g_2,\ h\ r_1\ r_2,\ h\ a_1\ a_2)$
  **where**
    $h\ x_1\ x_2 = w \cdot x_1 + (1 - w) \cdot x_2$
```

With $lerpC$, we can define other useful functions, e.g.,

```
$lighten,\ darken\ ::\ Fraction \rightarrow Color \rightarrow Color$
$lighten\ x\ c = lerpC\ x\ c\ white$
$darken\ x\ c = lerpC\ x\ c\ black$
```

It is also easy to extend color interpolation to two dimensions, by making three applications of linear interpolation–two horizontal and one vertical. Figure 6 illustrates this operation, and is centered at $(1/2, 1/2)$ rather than the origin.

```
$bilerpBRBW = bilerpC\ black\ red\ blue\ white$

$bilerpC\ ::Color \rightarrow Color \rightarrow Color \rightarrow Color \rightarrow (Frac, Frac) \rightarrow Color$
$bilerpC\ ll\ lr\ ul\ ur\ (dx, dy) = lerpC\ dy\ (lerpC\ dx\ ll\ lr)\ (lerpC\ dx\ ul\ ur)$
```

Because of the type invariant on colors, this definition only makes sense if dx and dy fall in the interval $[0, 1]$.

A operation similar to $lerpC$ is color overlay, which will be used in the next section to define image overlay. The result is a blend of the two colors, depending on the opacity of the top (first) color. A full discussion of this definition can be found in [18]:

$$\begin{aligned}
&(b_1,\ g_1,\ r_1,\ a_1)\ \text{`}overC\text{`}\ (b_2,\ g_2,\ r_2,\ a_2)\ =\ (h\ b_1\ b_2,\ h\ g_1\ g_2,\ h\ r_1\ r_2,\ h\ a_1\ a_2)\\
&\quad \mathbf{where}\\
&\qquad h\ x_1\ x_2\ =\ x_1\ +\ (1\ -\ a_1)\ \cdot\ x_2
\end{aligned}$$

(Note the use of backquotes to turn the name $overC$ into an infix operator.) Not surprisingly, color-valued images are of particular interest, so we'll use a convenient abbreviation:

$$\mathbf{type}\ ImageC\ =\ Image\ Color$$

5 Pointwise lifting

Many image operations result from pointwise application of operations on one or more values. For example, the overlay of one image on top of another can be defined in terms of $overC$:

$$\begin{aligned}
&over\ ::\ ImageC\ \rightarrow\ ImageC\ \rightarrow\ ImageC\\
&(top\ \text{`}over\text{`}\ bot)\ p\ =\ top\ p\ \text{`}overC\text{`}\ bot\ p
\end{aligned}$$

This commonly arising pattern is supported by a family of "lifting" functionals:[6]

$$\begin{aligned}
&lift_1 :: (a \rightarrow b) \rightarrow (p \rightarrow a) \rightarrow (p \rightarrow b)\\
&lift_2 :: (a \rightarrow b \rightarrow c) \rightarrow (p \rightarrow a) \rightarrow (p \rightarrow b) \rightarrow (p \rightarrow c)\\
&lift_3 :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow (p \rightarrow a) \rightarrow (p \rightarrow b) \rightarrow (p \rightarrow c) \rightarrow (p \rightarrow d)\\
&\cdots
\end{aligned}$$

$$\begin{aligned}
&lift_1\ h\ f_1\ p &&= h\ (f_1\ p)\\
&lift_2\ h\ f_1\ f_2\ p &&= h\ (f_1\ p)\ (f_2\ p)\\
&lift_3\ h\ f_1\ f_2\ f_3\ p &&= h\ (f_1\ p)\ (f_2\ p)\ (f_3\ p)\\
&\cdots
\end{aligned}$$

Then $over\ =\ lift_2\ overC$.

Other examples of pointwise lifting includes selection ($cond$) and interpolation ($lerpI$) between two images:[7]

$$\begin{aligned}
&cond\ ::\ Image\ Bool\ \rightarrow\ Image\ c\ \rightarrow\ Image\ c\ \rightarrow\ Image\ c\\
&cond\ =\ lift_3\ (\lambda\ a\ b\ c\ \rightarrow\ if\ a\ then\ b\ else\ c)
\end{aligned}$$

$$\begin{aligned}
&lerpI :: Image\ Frac\ \rightarrow\ ImageC\ \rightarrow\ ImageC\ \rightarrow\ ImageC\\
&lerpI\ = lift_3\ lerpC
\end{aligned}$$

[6]For intuition, think of p as $Point$, so that $p \rightarrow a = Image\ a$ and similarly for b, c, d.

[7]In a call-by-value language, $cond$ would need to be defined differently, in order to avoid unnecessary evaluation.

Zero-ary lifting is already provided by Haskell's const function:

$$\begin{aligned}&\mathit{const} :: a \rightarrow (p \rightarrow a)\\&\mathit{const}\ a\ p = a\end{aligned}$$

Given const, we can define the empty image and give convenient names to several opaque, constant-color images:

$$\begin{aligned}&\mathit{empty} &&= \mathit{const\ invisible}\\&\mathit{whiteI} &&= \mathit{const\ white}\\&\mathit{blackI} &&= \mathit{const\ black}\\&\mathit{redI} &&= \mathit{const\ red}\\&\ldots\end{aligned}$$

Note that *all* pointwise-lifted functions are polymorphic over the domain type (not necessarily Point), and so could work for 1D images (e.g., interpreted as sound), 3D images (sometimes called "solid textures"), or ones over discrete or abstract domains as well.

Figure 7 shows a simple example of image interpolation based on the examples in Figures 5, 2, and 4. Since lerpI works on color images, we must first color the region arguments.

$$\mathit{bwIm}, \mathit{byIm} :: \mathit{Region} \rightarrow \mathit{ImageC}$$

$$\begin{aligned}&\mathit{bwIm\ reg} = \mathit{cond\ reg\ blackI\ whiteI}\\&\mathit{byIm\ reg} = \mathit{cond\ reg\ blueI\ yellowI}\end{aligned}$$

As a simpler example, Figure 8 interpolates between blue and yellow, and will be useful in several later examples.

$$\mathit{ybRings} = \mathit{lerpI\ wavDist\ blueI\ yellowI}$$

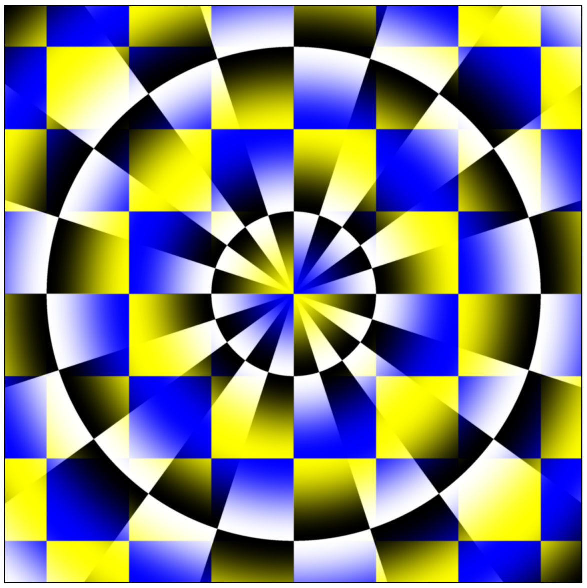
6 Spatial transforms

In computer graphics, spatial transforms are commonly represented by matrices, and hence are restricted to special classes like linear, affine, or projective. Application of transformations is implemented as a matrix/vector multiplication, and composition as matrix/matrix multiplication. In fact, this representation is so common that transforms are often thought of as *being* matrices. A simpler and more general point of view, however, is that transforms are simply space-to-space functions.

$$\mathbf{type}\ \mathit{Transform} = \mathit{Point} \rightarrow \mathit{Point}$$

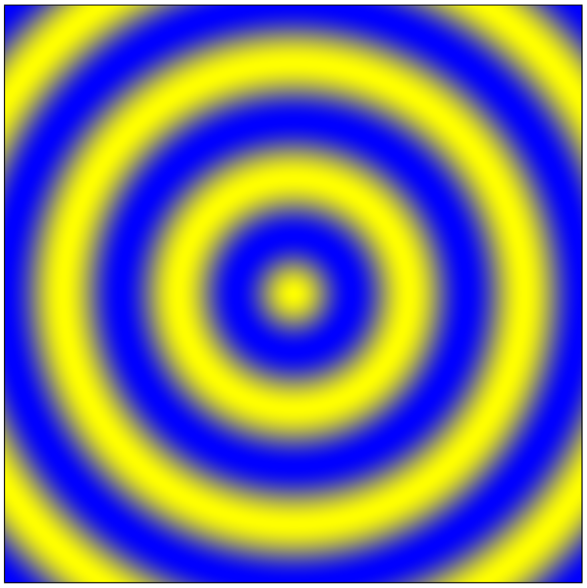
It is then easy to define the familiar affine transforms:

$$\begin{aligned}&\mathbf{type}\ \mathit{Vector} = (\mathit{Float}, \mathit{Float})\\&\mathit{translateP} :: \mathit{Vector} \rightarrow \mathit{Transform}\\&\mathit{translateP}\ (dx,\ dy)\ (x, y) = (x + dx,\ y + dy)\end{aligned}$$



$[width = 7]$

Figure 7 $lerpI\ wavDist(bwIm\ (polarChecker\ 10))$
$(byIm\ checker)$



$[width = 10]$

Figure 8 $ybRings$

$$\begin{aligned}
&scaleP\ ::\ Vector \rightarrow Transform\\
&scaleP\ (sx,\ sy)\ (x, y)\ =\ (sx \cdot x,\ sy \cdot y)
\end{aligned}$$

$$\begin{aligned}
&uscaleP\ ::\ Float \rightarrow Transform \quad \text{— uniform}\\
&uscaleP\ s\ =\ scaleP\ (s, s)
\end{aligned}$$

$$\begin{aligned}
&rotateP\ ::\ Float \rightarrow Transform\\
&rotateP\ \theta\ (x, y)\ =\ (x \cdot cos\ \theta - y \cdot sin\ \theta\ ,\ y \cdot cos\ \theta + x \cdot sin\ \theta)
\end{aligned}$$

By definition, transforms map points to points. Can we “apply” them, in some sense, to map images into transformed images?

$$applyTrans\ ::\ Transform \rightarrow Image\ c \rightarrow Image\ c$$

A look at the definitions of the *Image* and *Transform* types suggests the following simple definition:

$$applyTrans\ xf\ im\ \overset{?}{=}\ im \circ xf \quad \text{— } \textbf{wrong}$$

Figures 9 and 10 show a unit disk *udisk* and the result of $udisk \circ uscaleP\ 2$, where

$$\begin{aligned}
&udisk\ ::\ Region\\
&udisk\ p\ =\ distO\ p\ <\ 1
\end{aligned}$$

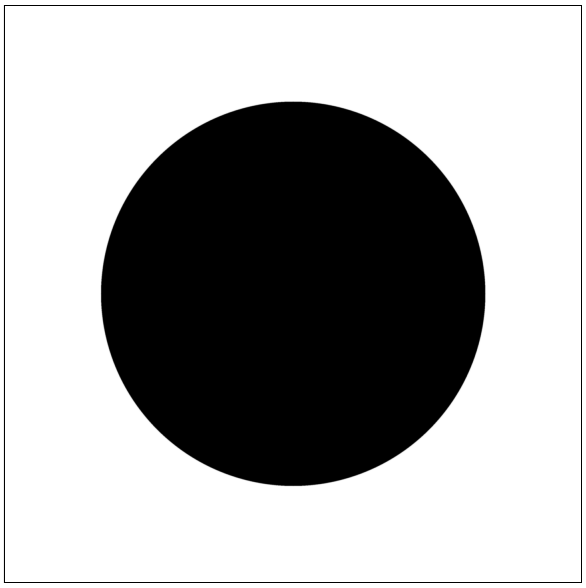

[$width = 3$]

Figure 9 $udisk$

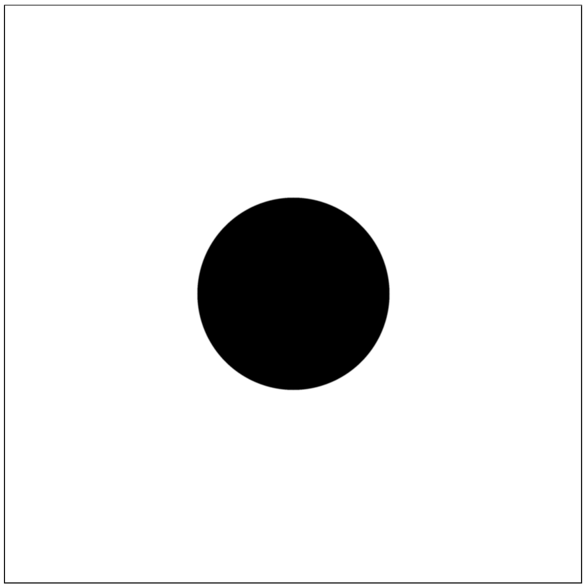

[$width = 3$]

Figure 10 $udisk \circ uscaleP\ 2$

Notice that the $uscaleP$-composed $udisk$ is *half* rather than twice the size of $udisk$. (Similarly, $udisk \circ translateP\ (1, 0)$ moves $udisk$ to the *left* rather than right.) The reason is that $uscaleP\ 2$ maps input points to be twice as far from the origin, so points have to start out within $1/2$ unit of the origin in order for their scaled counterparts to be within 1 unit.

In general, to transform an image, we must *inversely* transform sample points before feeding them to the image being transformed:

$$applyTrans\ xf\ im = im \circ xf^{-1}$$

While this definition is simple and general, it has the serious problem of requiring inversion of arbitrary spatial mappings. Not only is it sometimes difficult to construct inverses, but also some interesting mappings are many-to-one and hence not invertible. In fact, from an image-centric point-of-view, we *only* need the inverses and not the transforms themselves. For these reasons, we simply construct the transforms in inverted form, and do not use $applyTrans$.[8]

Because it can be mentally cumbersome always to think of transforms as functions and transform-application as composition, Pan provides a friendly vocabulary of image-transforming functions:

$$\textbf{type}\ Filter\ c = Image\ c \rightarrow Image\ c$$

$$\begin{aligned}
&translate,\ scale :: Vector \rightarrow Filter\ c\\
&uscale,\ rotate\ \ :: Float\ \ \rightarrow Filter\ c
\end{aligned}$$

$$\begin{array}{llcll}
translate\ (dx, dy) & im & = & im \circ translateP & (-dx,\ -dy)\\
scale\ \ (sx, sy) & im & = & im \circ scaleP & (1/sx,\ 1/sy)\\
uscale\ \ s & im & = & im \circ uscaleP & (1/s)\\
rotate\ \ \theta & im & = & im \circ rotateP & (-\theta)
\end{array}$$

[8]Easy invertibility is one of the benefits of restricting transforms to be affine and representing them as matrices.

In addition to these familiar affine transforms, one can define any other kind of space-to-space function, limited only by one's imagination. For instance, here is a "swirling" transform. It takes each point p and rotates it about the origin by an amount that depends on the distance from p to the origin. For predictability, this transform takes a parameter r that gives the distance at which a point is rotated through a complete circle (2π radians):

$$\begin{aligned}
&\mathit{swirlP} :: \mathit{Float} \to \mathit{Transform}\\
&\mathit{swirlP}\ r\ p = \mathit{rotate}\ (\mathit{distO}\ p \cdot 2\,\pi\ /\ r)\ p
\end{aligned}$$

$$\begin{aligned}
&\mathit{swirl} :: \mathit{Float} \to \mathit{Filter}\ c \quad \text{— Image version}\\
&\mathit{swirl}\ r\ \mathit{im} = \mathit{im} \circ \mathit{swirlP}\ (-r)
\end{aligned}$$

Applying the *swirl* effect to *vstrip* (Figure 1) defined earlier results in an infinite spiral whose arms thin out away from the origin (Figure 11).

It will be useful to have compact names for transformations of color images:

$$\textbf{type}\ \mathit{FilterC} = \mathit{Filter}\ \mathit{Color}$$

7 Animation

Just as an image is a function of space, an animation is a function of continuous time. This model is adopted from Fran [7, 5], and leads to temporal resolution independence, which allows animations to be transformed in time, as easily as images are transformed in space.

As a simple animation example, Figure 12 shows what *swirl* does to the half plane *xPos* given by $x > 0$. We square time to emphasize small and large values of the *swirl* parameter.

$$\begin{aligned}
&\mathit{xPos} :: \mathit{Region}\\
&\mathit{xPos}\ (x, y) = x > 0
\end{aligned}$$

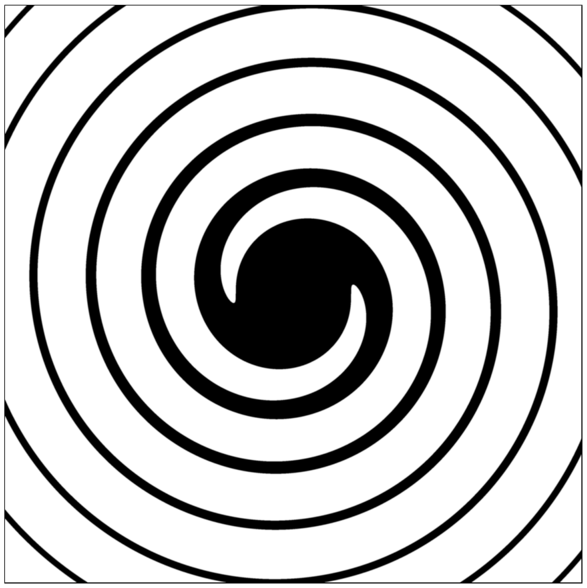
8 Region algebra

Boolean images are useful in many situations, and can be thought of as "regions" of space. This interpretation is just the usual identification between sets and characteristic functions:

$$\textbf{type}\ \mathit{Region} = \mathit{Image}\ \mathit{Bool}$$

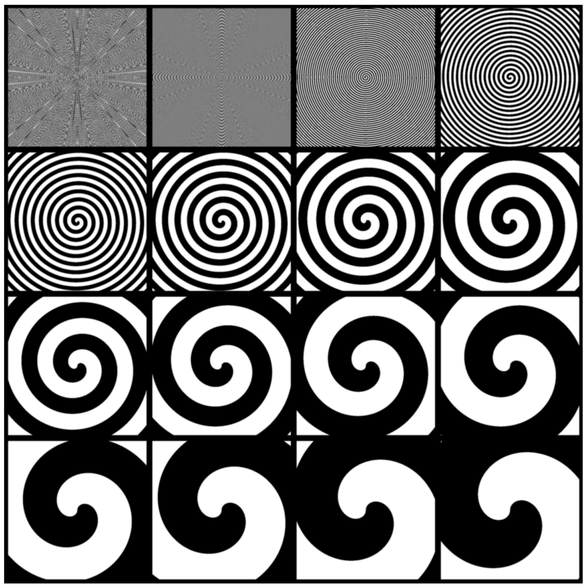
Set operations are useful and easy to define:

$$\begin{aligned}
&(\cap),\ (\cup),\ \mathit{xorR},\ (\backslash) &&:: \mathit{Region} \to \mathit{Region} \to \mathit{Region}\\
&\mathit{compR} &&:: \mathit{Region} \to \mathit{Region}\\
&\mathit{universeR},\ \mathit{emptyR} &&:: \mathit{Region}
\end{aligned}$$



$[width = 5]$

Figure 11 $swirl\ 1\ vstrip$



$[duration = 2, width = 5]$

Figure 12 $\lambda t \rightarrow swirl\ (t^2)\ xPos$

$$\begin{aligned}
(\cap) &= lift_2\ and\\
(\cup) &= lift_2\ or\\
xorR &= lift_1\ xor\\
compR &= lift_1\ not\\
universeR &= const\ True\\
emptyR &= const\ False\\
r \setminus r' &= r \cap compR\ r'
\end{aligned}$$

Let's see what we can do with these region operators. First, build an annulus out of our unit disk, given an inner radius, by subtracting one disk from another:

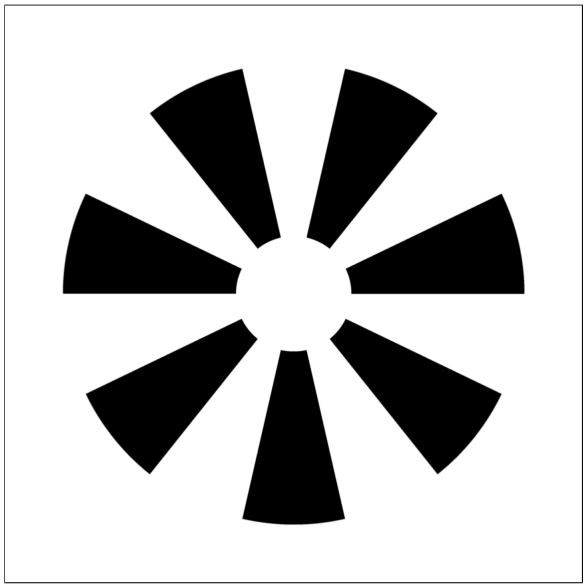
$$\begin{aligned}
&annulus :: Frac \rightarrow Region\\
&annulus\ inner = udisk \setminus uscale\ inner\ udisk
\end{aligned}$$

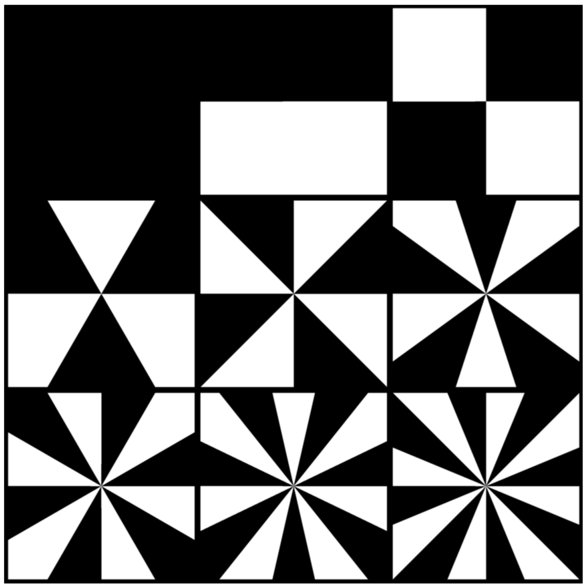
Next, make a region consisting of alternating infinite pie wedges (Figure 14), which is a simplification of Figure 4.

```
$radReg :: Int \rightarrow Region$
$radReg\ n = test \circ toPolar$
  where
    $test\ (r, a) = even\ \lfloor a \cdot fromInt\ n\ /\ \pi \rfloor$
```

Putting these two together, we get Figure 13.

$$\begin{aligned}
&wedgeAnnulus :: Float \rightarrow Int \rightarrow Region\\
&wedgeAnnulus\ inner\ n = annulus\ inner \cap radReg\ n
\end{aligned}$$


$[width = 2.5]$
Figure 13 $wedgeAnnulus\ 0.25\ 7$


$[duration = 8, width = 1]$
Figure 14 $radReg\ n$ for $n = 0, \ldots, 8$

The $xorR$ operator is useful for creating op art. For instance, Figure 15 is made from two copies of $altRings$ (Figure 3), shifted in opposite directions and combined with $xorR$.

```
$shiftXor :: Float \rightarrow Filter\ Bool$
$shiftXor\ r\ reg\ =\ reg'\ r\ `xorR`\ reg'\ (-r)$
  where
    $reg'\ d\ =\ translate\ (d, 0)\ reg$
```

Why stop at two copies of $altRings$? For any given n, the following definition distributes n copies of $altRings$ around a circle of radius r and xors them all together (Figure 16).

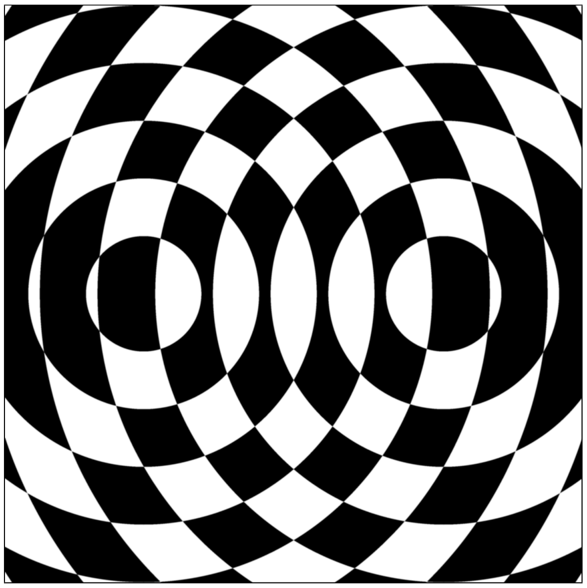
```
$xorgon :: Int \rightarrow Float \rightarrow Region \rightarrow Region$
$xorgon\ n\ r\ =\ xorRs\ (map\ rf\ [0\ ..\ n-1])$
  where
    $rf\ i\ =\ translate\ (fromPolar\ (r, a))\ altRings$
        where
          $a\ =\ fromInt\ i\ \cdot\ 2 \cdot \pi\ /\ fromInt\ n$
```

The function $xorRs$ does for a list of regions what $xorR$ does for two.

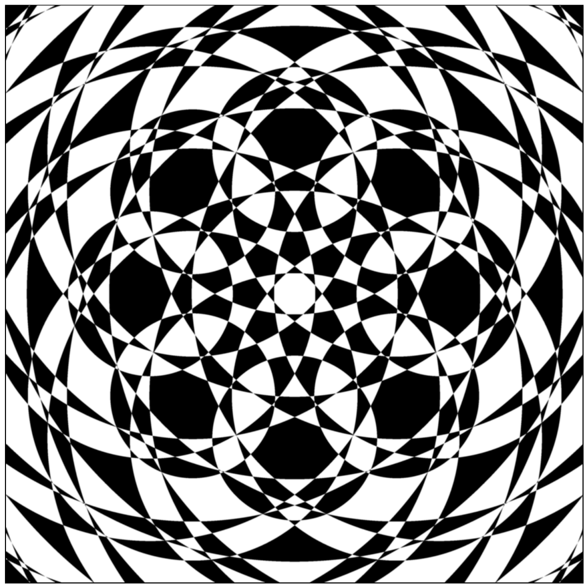
```
$xorRs :: [Region] \rightarrow Region$
$xorRs\ =\ foldr\ xorR\ emptyR$
```

Note also that $polarChecker$ (Figure 4) can be redefined very simply by applying $xorR$ to $altRings$ (Figure 3) and $radReg$ (Figure 14):

```
$polarChecker\ n\ =\ altRings\ `xorR`\ radReg\ n$
```



$[width = 10]$ **Figure 15** $shiftXor$ 2.6 $altRings$



$[width = 7]$ **Figure 16** $xorgon$ 8 $(7/4)$ $altRings$

Similarly, one could use $xorR$ and a coordinate-swapping filter to redefine $checker$ (Figure 2) in terms of a region with alternating horizontal or vertical slabs,

One use for regions is to crop a color-valued image:

$$\begin{array}{l} crop :: Region \rightarrow FilterC \\ crop\ reg\ im = cond\ reg\ im\ empty \end{array}$$

For instance, the title figures come from cropping $ybRings$ (Figure 8) with regions produced from $wedgeAnnulus$ (left of title) or a swirled $wedgeAnnulus$ (right).

9 Some polar transforms

The $swirlP$ function (from Section 6 and used to define $swirl$) can be somewhat simplified by considering points in polar rather than rectangular coordinates.[9]

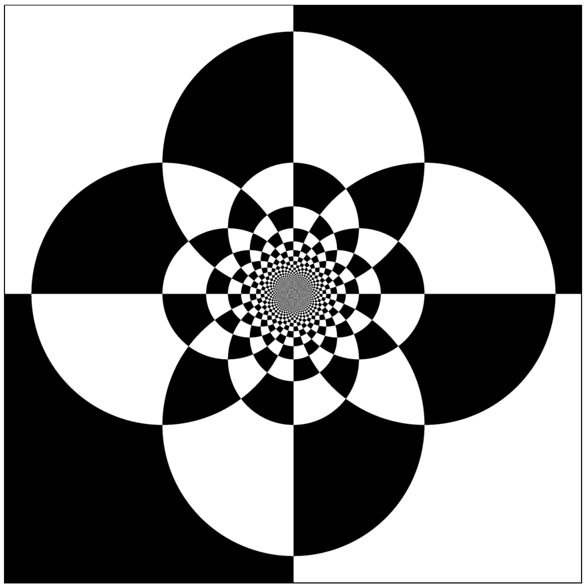
$$swirlP\ r = polarXf\ (\lambda\,(\rho, \theta) \rightarrow (\rho,\ \theta + \rho \cdot 2\pi / r))$$

Note that θ changes but ρ does not.

The useful function $polarXf$ is defined very simply:

$$\begin{array}{l} polarXf :: Transform \rightarrow Transform \\ polarXf\ xf = fromPolar \circ xf \circ toPolar \end{array}$$

[9]In polar coordinates, a point p is identified by a pair (ρ, θ), where ρ is the distance from the origin and θ is the angle between the positive X axis and the ray emanating fm the origin and passing through p.



$[width = 2.2]$
Figure 17 *radInvert checker*

9.1 Turning things inside out.

Next, let's consider a polar transform that changes ρ but not θ. Simply multiplying ρ by a constant is equivalent to uniform scaling ($uscale$). However, *inverting* ρ has a striking effect (Figure 17):

$$\begin{aligned}&radInvertP :: Transform\\&radInvertP = polarXf\ (\lambda\,(\rho,\theta) \rightarrow (1/\rho,\theta))\\[1em]&radInvert :: Image\ c \rightarrow Image\ c\\&radInvert\ im = im \circ radInvertP\end{aligned}$$

9.2 Radial ripples.

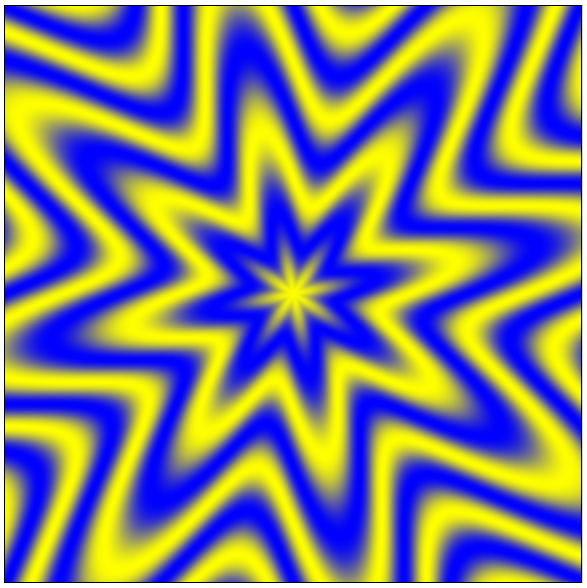
As another radial (ρ) transformation, we can multiply ρ by an amount that oscillates around 1 with a given magnitude s, having a given number n of periods as θ varies from 0 to 2π. As usual, define an image-transforming version as well:[10]

$$\begin{aligned}&rippleRadP :: Int \rightarrow Float \rightarrow Transform\\&rippleRadP\ n\ s = polarXf\ \$\ \lambda\,(\rho,\theta) \rightarrow (\rho \cdot (1 + s \cdot sin\,(fromInt\ n \cdot \theta)),\ \theta)\\&rippleRad :: Int \rightarrow Float \rightarrow Image\ c \rightarrow Image\ c\\&rippleRad\ n\ s\ im = im \circ rippleRadP\ n\ (-s)\end{aligned}$$

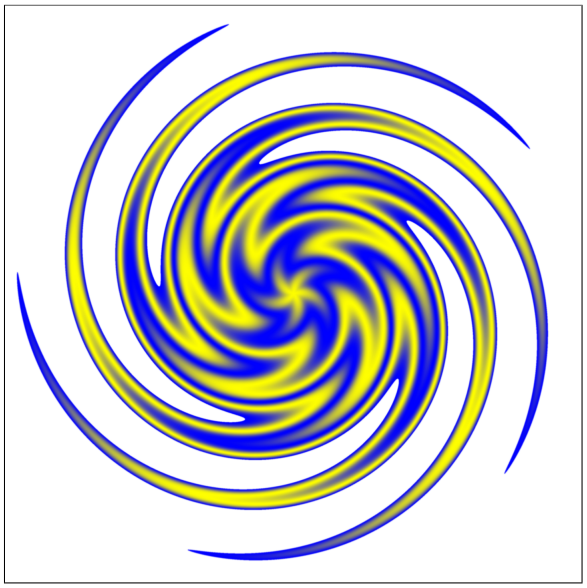
In order to visualize the effect of $rippleRad$, apply it to $ybRings$ (Figure 8). The result is Figure 18.

The examples so far have been infinite in size. We can also make finite ones by cropping against a region. As a convenience, define $cropRad$ as a function that crops an image to a disk-shaped region of a given radius:

[10] The "\$" operator is infix, right-associative, low-precedence function application. It often reduces the need for parentheses.



$[width = 10]$
Figure 18 $\mathit{rippleRad}\ 8\ 0.3\ \mathit{ybRings}$



$[width = 10]$
Figure 19 $\mathit{swirl}\ 8\ \$\ \mathit{cropRad}\ 5\$\ \mathit{rippleRad}\ 5\ 0.3\$\ \mathit{ybRings}$

$$\mathit{cropRad} :: \mathit{Float} \to \mathit{FilterC}$$
$$\mathit{cropRad}\ r = \mathit{crop}\ (\mathit{uscale}\ r\ \mathit{udisk})$$

To make the picture more interesting, let's crop, ripple, then swirl (Figure 19).

10 Related Work

Peter Henderson began the game of functional geometry for image synthesis [9]. Since then there have been several other such libraries, including [14, 19, 3, 1, 8, 7]. None have addressed the general notion of images.

Gerard Holzmann developed a system called "Pico", which consisted of an editor, a simple language for image transformations, and a machine-code generator for fast display. His delightful book shows many examples, using photos of Bell Labs employees [10]. Pico's model of images was the discrete rectangular array of bytes, which could be interpreted as grey-scale values or other scalar fields. The host language appears to have been very primitive, with essentially no abstraction mechanisms.

John Maeda's "Design by Numbers" (DBN) is another language aimed at simplifying image synthesis, sharing with Pan the goals of simplicity and encouragement of creative exploration [4]. In contrast, the DBN language is squarely in the imperative style (*doing* rather than *being*). Its programs are lists of commands for outputting dots or line segments and changing internal state, with an image emerging as the cumulative result. Like Pico, DBN presents a discrete notion of space, partitioned into a finite array of square pixels.

The Haskell "region server" [12] used characteristic functions to represent regions, in essentially the same formulation as Pan (Section 8). Those regions were not used for visualization, nor were they generalized to range types other than Boolean. Paul Hudak also used regions for graphics [11]. There an algebraic data-type represents regions, but an interpretation (semantics) is given by translating to a function from 2D space to Booleans.

In his work on evolution for computer graphics, Karl Sims represented images as Lisp expressions over variables with names x, y, and t (adding z for solid textures) [17]. He did not exploit Lisp's support for higher-order functional programming for composing image functions.

11 Conclusions

For the purpose of image synthesis, imperative programming languages (including most object-oriented ones) are unpleasantly "low-level" in the sense of Alan Perlis: "A programming language is low level when its programs require attention to the irrelevant."

This paper presents a simple model and high-level *functional* programming language for images, as functions from continuous 2D space to colors, and then tests the expressiveness of this model by means of several examples. These examples represent just a hint at what can be done, and are far from exhaustive, or even necessarily representative. I hope that readers are inspired to apply their own creativity to generate images and animations that look very different from the examples in this paper.

12 Acknowledgements

Sigbjørn Finne and Oege de Moor collaborated on the implementation and provided fruitful discussions. Brian Guenter suggested the project of an image manipulation language and optimizing compiler and has provided helpful expertise. Koen Classen provided helpful comments on a related paper.

References

[1] Kavi Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, January 1994.

[2] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[3] Joel F. Bartlett. Don't fidget with widgets, draw! Technical Report 6, DEC Western Digital Laboratory, 250 University Avenue, Palo Alto, California 94301, US, May 1991.

[4] John Maeda. Foreword by Paola Antonelli. *Design By Numbers*. MIT Press, May 1999. `http://www.maedastudio.com/dbn`.

[5] Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May/June 1999. Special Section: Domain-Specific Languages (DSL). `http://research.microsoft.com/~conal/papers/tse-modeled-animation`.

[6] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. In Walid Taha, editor, *Semantics, Applications and Implementation of Program Generation (SAIG)*. ICFP, September 2000. `http://research.microsoft.com/~conal/papers/saig00`.

[7] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, Amsterdam, The Netherlands, 9–11 June 1997. `http://research.microsoft.com/~conal/papers/icfp97.ps`.

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

[9] Peter Henderson. Functional geometry. In *ACM Symposium on LISP and Functional Programming*, pages 179–187, 1982.

[10] Gerard J. Holzmann. *Beyond Photography — the Digital Darkroom*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988. (Out of print).

[11] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[12] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs Awk vs ... an experiment in software prototyping productivity. Technical report, Yale, 1994.

[13] SL Peyton Jones, RJM Hughes, L Augustsson, D Barton, B Boutel, W Burton, J Fasel, K Hammond, R Hinze, P Hudak, T Johnsson, MP Jones, J Launchbury, E Meijer, J Peterson, A Reid, C Runciman, and PL Wadler. Haskell 98: A non-strict, purely functional language. `http://haskell.org/definition`, February 1999.

[14] Peter Lucas and Stephen N. Zilles. Graphics in an applicative context. Technical report, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, July 8 1987.

[15] Scott McCloud. *Understanding Comics*. Kitchen Sink Press, 1994.

[16] Leonard Shlain. *The Alphabet versus the Goddess – The Conflict Between Word and Image*. Penguin/Arkana, 1998.

[17] Karl Sims. Artificial evolution for computer graphics. *ACM Computer Graphics*, 25(4):319–328, July 1991. SIGGRAPH '91 Proceedings.

[18] Alvy Ray Smith. Image compositing fundamentals. Technical Report Technical Memo #4, Microsoft, July 1995. `http://www.alvyray.com/Memos`.

[19] S.N. Zilles, P. Lucas, T.M. Linden, J.B. Lotspiech, and A.R. Harbury. The Escher document imaging model. In *Proceedings of the ACM Conference on Document Processing Systems (Santa Fe, New Mexico)*, pages 159–168, December 5–9 1988.