

Programming, Abstraction and Modelling
Final Project 01

Graphics Description Language

Project Overview: Graphics Description Language

For this Final Project, students may implement a small graphics language – and demonstrate the most interesting results they can create with the language.

Aside from programming languages, there are other computer-based languages – such as *Postscript* and *SVG* for describing images. This project involves translating (*compiling*) a textual description of a picture into an image-format that can be displayed with a graphics-viewing program.

Project Requirements

Student-teams have a great deal of freedom to choose which particular graphics-manipulation instructions their program will support; however, they need to support the following.¹

- Colors (3-color model – RGB, for example)
- Some set of basic primitives, such as *circle*, *rectangle*, *line*, or the like (the actual set is to be determined by team-members)
- *Conditionals*, such as `if`
- *Variables*
- *Loops*

Using the implemented graphics-description language should result in viewable images in some standard graphics format. A suitable target graphics-format is the PPM-format, since it is quite easy to create (and debug). Furthermore, reading chapter 4 in SICP is recommended for learning how to implement an interpreter (i.e. the core “language” part of the project, consisting of the variables, conditionals, loops etc).

Possible Extensions

Here are also some suggestions for possible elaborations of such a project:

- Functions (for abstraction over different parts of images, such as for hierarchical modeling)
- Rotation
- Transparency (four-color model, such as RGBA)
- Color-patterns (example: a “checkerboard” color)
- *Time* (animations, for example)

¹Or some sets of features of equivalent scope.

Resources

You can find more about PPM and graphics languages here:

Specification of the PPM format:

<http://netpbm.sourceforge.net/doc/ppm.html>

Functional image synthesis:

<http://conal.net/papers/bridges2001/>

Bresenhams algorithm:

<http://www.netnam.vn/unescocourse/computervision/63.htm>
http://www.fact-index.com/b/br/bresenham_s_line_algorithm.html

Floodfill (also called seed fill):

http://www.fact-index.com/f/fl/flood_fill.html

SVG specification:

<http://www.w3.org/TR/SVG/>

Postscript:

<http://www.cs.indiana.edu/docproject/programming/postscript/postscript.html>

Example

To give some idea of what it would mean to *use* a graphics-language that meets some of the constraints, below is an example portion of a language that does not include *functions* or *abstraction*, but otherwise meets some of the requirements.

```
# set the size of the image to 117 * 108
size 120 108

# set the color to black
color 0 0 0

# define a pair of variables
def rectangle-size 24
def spacing 4
def hop-sideways ((rectangle-size / 2) + spacing)
def next rectangle-size + spacing
def ypos 0
def xpos 60 - (rectangle-size / 2)

rectangle xpos ypos rectangle-size rectangle-size

xpos = xpos - hop-sideways
ypos = ypos + next

rectangle xpos ypos rectangle-size rectangle-size
xpos = xpos + next
rectangle xpos ypos rectangle-size rectangle-size

xpos = xpos - next - hop-sideways
rectangle xpos ypos rectangle-size rectangle-size
xpos = xpos + next
rectangle xpos ypos rectangle-size rectangle-size
xpos = xpos + next
rectangle xpos ypos rectangle-size rectangle-size

xpos = xpos - (next * 2) - hop-sideways
rectangle xpos ypos rectangle-size rectangle-size
xpos = xpos + next
rectangle xpos ypos rectangle-size rectangle-size
xpos = xpos + next
rectangle xpos ypos rectangle-size rectangle-size
xpos = xpos + next
rectangle xpos ypos rectangle-size rectangle-size
```

And here is an example that does include *functions* and *abstraction*:

```

# set the size of the image to 117 * 108
size 117 108

# set the color to black
color 0 0 0

# define some variables
def rectangle-size 24
def spacing 4
def hop-sideways ((rectangle-size / 2) + spacing)
def next rectangle-size + spacing
def ypos 0
def xpos 60 - (rectangle-size / 2)

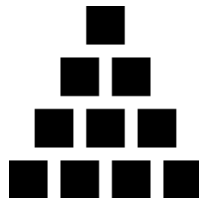
def draw-rectangle (x y)
  rectangle x y rectangle-size rectangle-size
enddef

def draw-n-rectangles (x y n)
  while n > 0 loop
    draw-rectangle x y
    x = x + next
  endloop
enddef

def i = 0
while i < 5 loop
  draw-n-rectangles(xpos, ypos, i)
  xpos = xpos - hop-sideways
  ypos = ypos + next
endloop

```

Both of the examples above result in the following image:



Tips

There are essentially two models of synthesising images, one based on *functional image synthesis* and the other one based on *imperative raster manipulation*. You have to choose which model you will base your project on. They have different pros and cons: different models make different kinds of image-manipulation easier/harder – and students can expect to gain different kinds of insights about language-design (and graphics), depending on the model chosen. Note: neither variation is “easier” than the other, and regardless of which method is chosen, teams are expected to put in the same amount of work. Here follows a quick overview of the two approaches.

Functional Image Synthesis

The main advantage of functional image synthesis is that it makes it easy to draw things – especially complex things. Groups choosing this model will have an easy time creating beautiful images

because everything we want to draw – from coordinates to colors – is described by a function. So for example, if we want to draw a black line from point 50 50 to point 100 75 we just use the equation for that line:

```
(define (make-line k m)
  (lambda (x y)
    (= y (+ (* k x) m))))

(define my-line (make-rect 0.5 25))
```

To draw this line we just iterate over every pixel of the image and check what value the procedure `my-line` returns for that pixel. We can also create interesting images by doing coordinate transformations (such as inverting the coordinates, rotating the coordinates in proportion to their distance to origin, etc.).

The disadvantage of this approach is that it is quite slow. To draw this line we have to draw every pixel of the image, even though the line only covers a very small part of the image.

The imperative approach

The main advantage of the imperative approach to image synthesis is that it results in substantially faster code. However the algorithms are harder to implement and understand. Instead of mapping coordinates to colors we here manipulate a raster (that is, setting the pixels of the image). If we want to draw the same line as above we could use Bresenham's line-drawing algorithm; if you look in the references you will see that it is not as simple as the functional example above. It is, however, this approach that is used most commonly in the “real world” and this approach will give you the opportunity to learn a lot more about low-level graphics (that is, a hardware perspective on how a line drawn on a computer screen).