# Project PixMix

September 26, 2017  
Programming Languages and Translators  
Stephen Edwards

## Team Members

<table>
<thead>
<tr>
<th>Name</th>
<th>Uni</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexandra Taylor</td>
<td>at3022</td>
<td>Tester</td>
</tr>
<tr>
<td>Christina Charles</td>
<td>cdc2192</td>
<td>Language Guru</td>
</tr>
<tr>
<td>Edvard Eriksson</td>
<td>ehe2107</td>
<td>Manager</td>
</tr>
<tr>
<td>Nathan Burgess</td>
<td>nab2180</td>
<td>System Architect</td>
</tr>
</tbody>
</table>
Objective

To create a language to allow images to be easily generated and manipulated programmatically.
Introduction

Currently, there are few, if any, languages designed specifically around the generation and manipulation of image data. This role is primarily left up to image editing software that an artist or designer must use.

Enter Pixel

Pixel is a simple, python-like programming language designed for efficient, programmatic generation and manipulation of images. The idea is to lower the user’s reliance on libraries by offering useful built-in data types and functions specific to image manipulation. These features will hopefully serve as intuitive building blocks that can be combined to create programs in an easier way than through the use of libraries. Pixel does not allow for image-processing only image generation and manipulation it eliminates many layers of processing and strives for readability, simplicity and speed. This narrow scope is useful to artists and designers who do not utilize image-processing functionality.

Pixel uses static typing and includes area-specific data types such as Image, Pixel and Color. It will also contain predefined common color keywords. This way a pixel can be assigned “blue” without having to type it out in hex. The keyword var can be used a la C# in loops to allow for optional implicit typing.

Overview of Notable Design Decisions (to laugh at and criticize)

- High-level image manipulation/generation language
- Python-like syntax
- Statically typed
- Built in data-types specific to image manipulation
- Built in functions specific to image manipulation
- Color keywords within a Color data type (red, blue, green, yellow, etc)
# Data Types

## Standard

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Basic 4-byte integer</td>
<td>Int varName = 0</td>
</tr>
<tr>
<td>Float</td>
<td>Standard single-precision 32-bit IEEE 754 value</td>
<td>Float varName = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Float varName = 1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Float varName = .2</td>
</tr>
<tr>
<td>Bool</td>
<td>Basic 1-bit boolean true, false, 0, or 1</td>
<td>Bool varName = true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bool varName = 0</td>
</tr>
<tr>
<td>Char</td>
<td>Basic 1-byte character from standard ASCII char.</td>
<td>Char varName = ‘c’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Char[] varName = “Hello”</td>
</tr>
<tr>
<td>Array</td>
<td>A standard array implementation</td>
<td>Array varName = [1, 2, 3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Array varName = []</td>
</tr>
</tbody>
</table>

## Unique to Pixel

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>The data type to hold all image data together in one variable.</td>
<td>Image im = Image.load(&quot;filename.bmp&quot;)</td>
</tr>
<tr>
<td>Pixel</td>
<td>A representation of a single pixel in an image.</td>
<td>Pixel pi = Color.lightblue</td>
</tr>
<tr>
<td>Color</td>
<td>A representation of an RGBA color. Various color names will be predefined in the Color namespace.</td>
<td>Color red = [255, 0, 0, 1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Color blue = [0, 255, 0]</td>
</tr>
</tbody>
</table>
Control Flow

Control is managed by indentation, following a Python-like approach.

Conditionals

if
elif
else
switch

Loops

Loop over components of an object (ex: Images iterate by Pixels)
for var i in Image:

Loop over a range
for var i in 2 to 10:

Loop until a condition becomes false
while var i < 10 or var x > 3:

Empty values

null
undefined
NaN
0
Empty string ""
Empty char ""

Logical operators

and
or
not
!
==
!=
>
<
\[ \geq \]
\[ \leq \]

**Arithmetic operators**

+ 
++ 
- 
-- 
* 
/ 
% 
+= 
-= 
*= 
/= 

**Comments**

// Line
/* */ Block

**Boolean variables**

true
false
Example Programs

Sample program 1: Hello World

// prints greeting according to time of day
Char[] out = “Good”

if Time.now < Time.get(1200):
    out += “ morning!”
elif Time.now < Time.get(“5 PM”):
    out += “afternoon!”
else
    out += “evening!”

Console.log(out) // Print the contents of “out” to stdout

Sample Program 2: Blacken any pixels that are “too” red, blur the image, then save as a new file

// Load an image from disk
Image img = Image.load(“sample.bmp”)

// Loop over every Pixel in the Image
for var p in img:
    // If red channel is more than 100, remove red
    if p.red > 100:
        p.red = 0

img.gaussianBlur(3) // Blur the image
img.saveAs(“sample-redMute-Blur.bmp”) // Save to a new file

Sample Program 3: Fibonacci

fun int fib(int i):
    if i == 0:
        return 0
    if i == 1:
        return 1

Console.read(int i) // Prompt user for input
Console.log(fib(i - 1) + fib(i - 2)) // Print result to stdout
Sample Program 4: Compositing Images

Image base = Image.load("base.bmp")
Image corner = Image.load("corner.bmp")
Image frame = Image.load("frame.bmp")

Int frameRepX = base.width / frame.width
Int frameRepY = base.height / frame.height

for var i in frameRepX:
    base.place(frame, 0, i)
    base.place(frame, base.height, i)

frame.rotate(90)
for var i in frameRepY:
    base.place(frame, i, 0)
    base.place(frame, i, base.width)

base.place(corner, 0, 0)
corner.rotate(90)
base.place(corner, base.width, 0)
corner.rotate(90)
base.place(corner, base.width, base.height)
corner.rotate(90)
base.place(corner, 0, base.height)

base.saveAs("SAEframed.bmp")