# PIXL

## A Pixel Processing Language

Justin Borczuk, Jacob Gold, Maxwell Hu, Shiv Sakhuja, Marco Starger

# A Preview of What's to Come

# Who did what?

Justin Borczuk - LLVM implementation guy, Windows user

Jacob Gold - refactoring for SAST, C functions, “go-to guy”

Max Hu - project manager, front-end guy

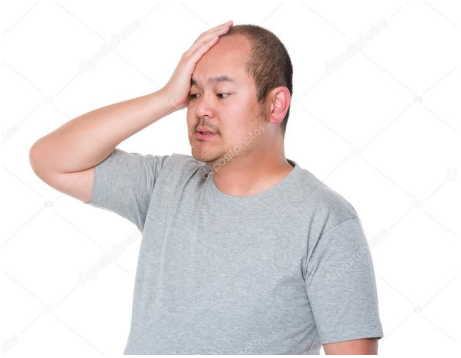
Shiv Sakhuja - semant construction guy (and user #0/ artist-in-residence)

Marco Starger - testing and standard library guy

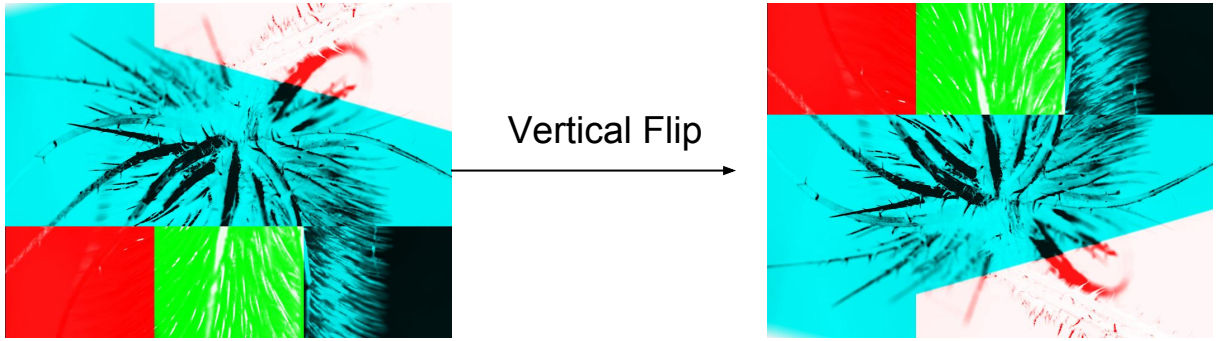
# Overview

What is PIXL?

# Why PIXL?



- Image processing can often be complicated
- PIXL presents simple syntax for lengthy image manipulations
- example: **|m** performs a vertical flip on the matrix m.



# Overview

## Basics of PIXL

- Compiles to LLVM with garbage collection
- C-like syntax and semantics
- Pixel and matrix types
- Image file I/O

## PIXL Features

- Large PIXL function library:
  - change opacity
  - change RGB
  - grayscale
  - subtraction
- Matrix Operators
  - crop
  - flip horizontal/vertical

# Syntax Basics

## Type

int, string, pixel, matrix, bool

## Control Flow

if, else, while, for, return

## Arithmetic Operators

\+ - * / ++ -- =

## Conditional Operators

== != < > <= >=

## Special Pixel/Matrix Operators

~ | + - << >> [ ] &&,

## Types and literals

```
int a;
int pixel p;
pixel matrix pm;

a = 1;
p = (42, 42, 42, 42);
pm = [p,p ; p,p];
```

# Some Library Functions

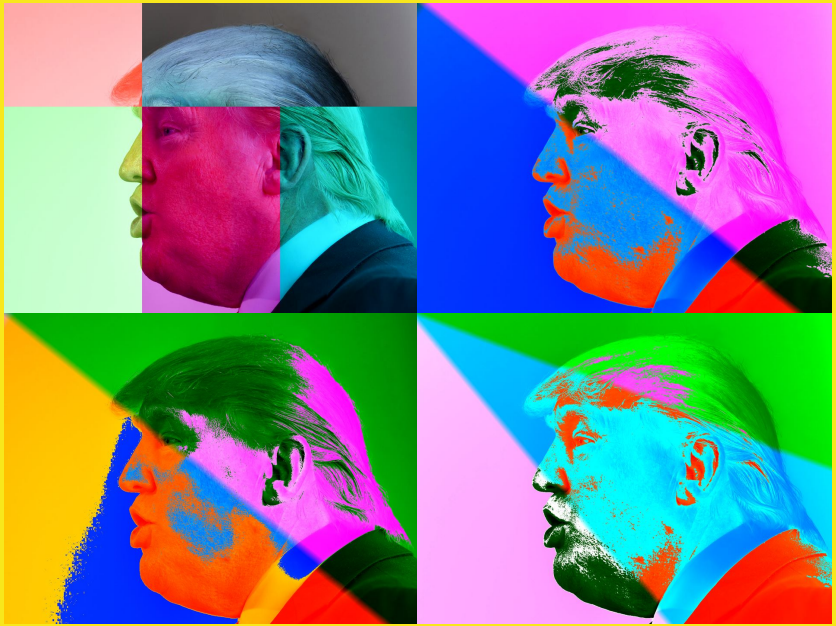
## Matrix Cropping

```
int matrix cropIntMatrix(int matrix m, int r1, int r2, int c1, int c2) {
    int i;
    int j;
    int a;
    int matrix m2;

    m2 = matrix(r2-r1, c2-c1, int);

    for (i = r1; i < r2; i=i+1)
    {
        for (j = c1; j < c2; j=j+1)
        {
            m2[i-r1][j-c1] = m[i][j];
        }
    }

    return m2;
}
```

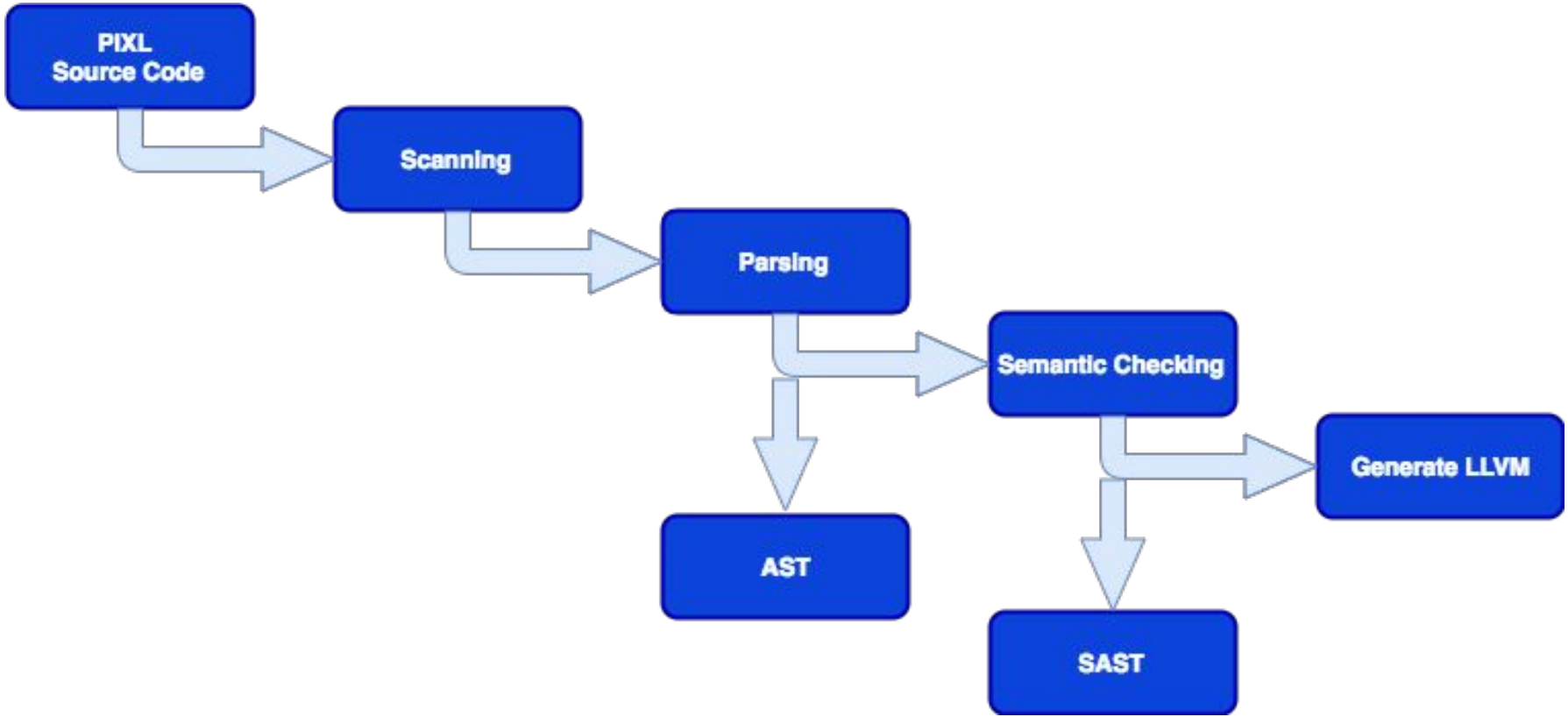
## Horizontal Image Flip

```
pixel matrix flipPixelMatrixH(pixel matrix pm) {
    int height;
    int width;
    int i;
    int j;
    pixel matrix pm2;
    height = pm.rows;
    width = pm.cols;

    pm2 = pm;

    for (i = 0; i < height; i=i+1)
    {
        for (j = 0; j < width; j=j+1)
        {
            pm2[i][j] = pm[i][width-1-j];
        }
    }

    return pm2;
}
```

# Implementation



# Architecture



# Testing PIXL

Automated test suite
testall.sh

```
# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to
PIXL="./pixl.native"
#MICROC="_build/microc.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.p files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
}
```

Output

```
-n testixelLit1.p...
OK
-n testixelReassignment.p...
OK
-n testrintInt.p...
OK
-n test-returnX...
-n test-rows...
OK
-n test-subtractIntMatrix...
OK
-n test-subtractPixel...
OK
-n fail-add...
OK
-n fail-assign...
OK
-n fail-cr.p...
OK
-n fail-cr2.p...
OK
-n fail-declare...
OK
-n fail-incArguments...
OK
-n fail-inconsistentMatrix...
OK
-n fail-matrixAssign...
OK
-n fail-matrixAssign1...
OK
-n failixelAssign.p...
OK
-n failrint.p...
OK
-n failrint1.p...
OK
-n fail-reassignment...
OK
-n fail-undeclared...
OK
-n fail-unrecognizedFunc...
OK
marcostarger@Marcos-MacBook-Pro:~/Dropbox/PLT/PIXL/pixl$
```

# ImageNet Classification with Deep Convolutional Neural Networks

Krizhevsky, Alex et al.

### 4.1 Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., [25, 4, 5]). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

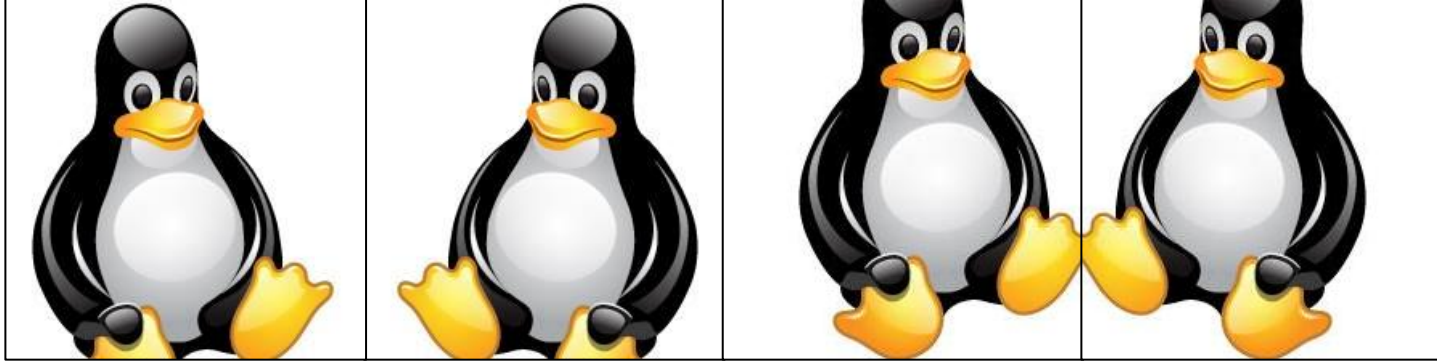
The first form of data augmentation consists of generating image translations and horizontal reflections. We do this by extracting random $224 \times 224$ patches (and their horizontal reflections) from the $256 \times 256$ images and training our network on these extracted patches[4]. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent. Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks. At test time, the network makes a prediction by extracting five $224 \times 224$ patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all), and averaging the predictions made by the network's softmax layer on the ten patches.

The second form of data augmentation consists of altering the intensities of the RGB channels in training images. Specifically, we perform PCA on the set of RGB pixel values throughout the ImageNet training set. To each training image, we add multiples of the found principal components, with magnitudes proportional to the corresponding eigenvalues times a random variable drawn from a Gaussian with mean zero and standard deviation 0.1. Therefore to each RGB image pixel $I_{xy} = [I^R_{xy}, I^G_{xy}, I^B_{xy}]^T$ we add the following quantity:

$$[\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3][\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3]^T$$

where $\mathbf{p}_i$ and $\lambda_i$ are $i$th eigenvector and eigenvalue of the $3 \times 3$ covariance matrix of RGB pixel values, respectively, and $\alpha_i$ is the aforementioned random variable. Each $\alpha_i$ is drawn only once for all the pixels of a particular training image until that image is used for training again, at which point it is re-drawn. This scheme approximately captures an important property of natural images, namely, that object identity is invariant to changes in the intensity and color of the illumination. This scheme reduces the top-1 error rate by over 1%.

# Penguins!



```
int main() {
    int i;
    int j;
    pixel matrix cropped;
    pixel matrix flipped;
    pixel matrix img;

    img = read("penguin.jpg");
    for (i = 0; i < 32; i=i+1) {
        for (j = 0; j < 32; j=j+1) {
            cropped = img<<i:i+224,j:j+224>>;
            write(cropped, "img" + str_of_int(i) + "_" + str_of_int(j), "jpg");
            flipped = ~cropped;
            write(flipped, "img" + str_of_int(i) + "_" + str_of_int(j) + "f", "jpg");
        }
    }
    return 0;
}
```

# DEMO

# Thank You!