Lab 4
John Munyi
18/09/2018

Table of Contents

1. Introduction

This assignment is divided into 2 parts, the first part deals with creating 2048 wide by 1024 high equirectangular image for 360 degrees viewing in the Oculus Go. The Image when complete makes the user feel like he or she is inside a big beach ball and the green color is always in front of him. The first part demonstrates Oculus Go image mapping in 360.
On the second part of the assignment, the goal is to perform flat projection of a 2D image in the Oculus Go. The goal here is to correct it such that when viewed in 2D the image doesn't look distorted, this demonstrates Oculus Go image mapping in 180 degrees.

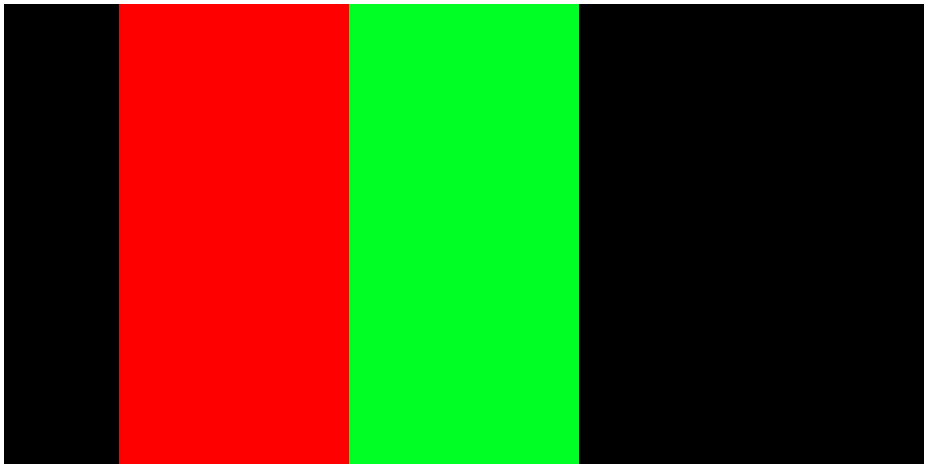
2. Part 1: Beach ball

In this section, the goal is to create a beach ball which has four different colors and Oculus Go image mapping in 360. The initial step is to create 2048 wide by 1024 high black image. The next process is to calculate and convert degrees to pixels so that we can easily map the existing pixel to an Oculus Go mapping in 360.
When we have the correct converted values, the process of painting the ball the 4 different colors starts. This is a step by step process where each color has specified polar coordinates. The final image is a colorful beach back which when viewed in the Oculus Go in 360 3D mode the user feel like they are inside the ball. The step by step resulting images are illustrated in the next section.

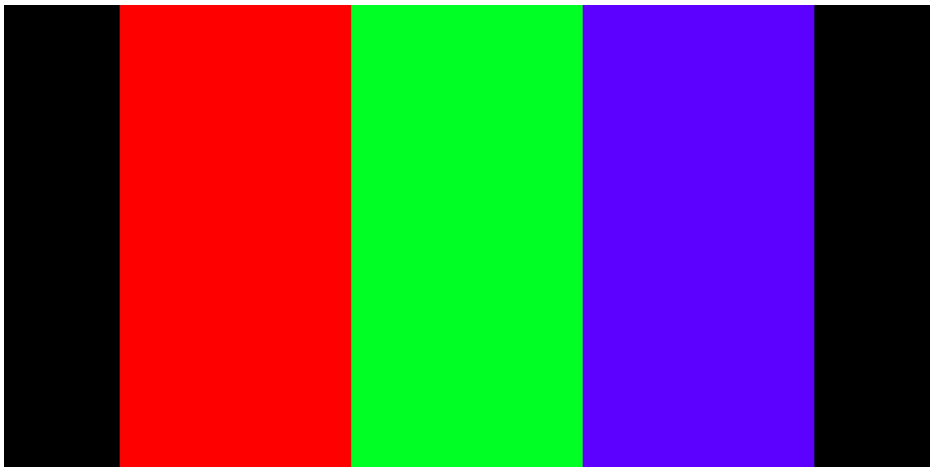
2.1. Results

Red color applied:

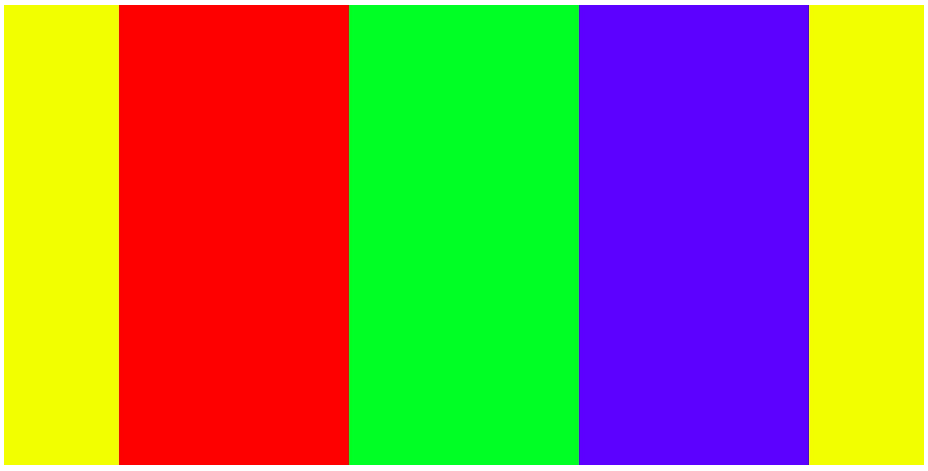
Green color applied:



Blue color applied:



Yellow color applied:



2.2 Discussion

The user feels immersed into the ball and he or she is right at the center of the sphere. The four colors meet at the poles of the sphere. Another interesting observation that every time I reset the focus of the Oculus Go, the green color is the one that always at the front.

3.Part 2: Flat Projection

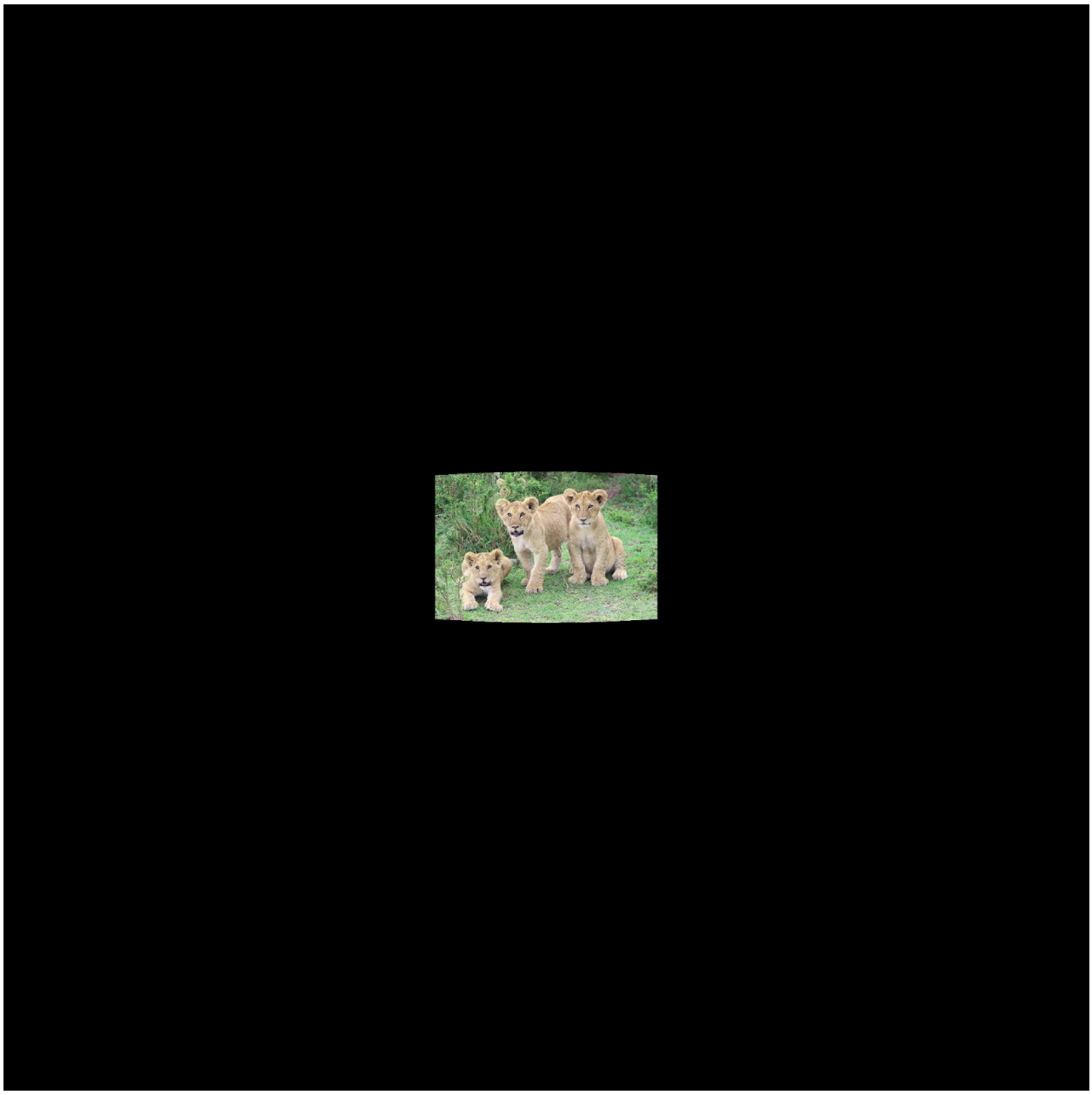
In this part, the goal of the exercise was to determine the largest dimension of the image, i.e., rows or columns and set the viewing distance equal to a value. The image is then mapped on a 180 degrees Oculus Go mapping, such that when viewed images in 2D -180 degrees the images don't look warped as it does when viewed normally on the computer.
The process to achieve the results involves first calculating the angles Theta and Phi which help in determining the widest “cone” angles from which the user is viewing from. This is followed by a bi-linear interpolation process as we map the images on to a new background, based on the 3 different distance that we have been provided. The resulting images are explained in the next section.

3.1 Results

Image at 0.5 distance is pasted above

Below shows the image at distance of 1.0

Below is the image at a distance of 1.5



3.2 Discussion

In this assignment it was very noticeable that the bigger the distance the further the images appears on the Oculus Go. Also in as much as the images looked warped a bit when viewed with a naked when viewed under 2D and 180 degrees the images look fine.

4. Conclusion

To conclude both python script show the inner working of how Oculus Go maps image on the presentation plane for both 2D and 3D images. The assignments also demonstrates the use of polar coordinates vs PICS and CICS.

5. Appendix

beachball.py

```
import numpy as np
import bmp_io_c
import math

cols = 2048
rows = 1024
layers = 3

# create a black image
black_image = np.zeros([3, rows, cols], np.uint8)
bmp_io_c.output_bmp_c("my_image.bmp", black_image)

# convert longitudes to pixels
def get_x(width, lng):
   return int(round(math.fmod((width * (180.0 + lng) / 360.0), (1.5 * width))))

# insert red
def insert_red(image):
   for i in range(cols):
       x1 = get_x(cols, -135)
       x2 = get_x(cols, -45)
       for lat in range(x1, x2):
           image[0,:,lat] = 255

   return image

new_image = insert_red(black_image)
bmp_io_c.output_bmp_c("image1.bmp", new_image)
```

```
# insert green
def insert_green(image):
   for i in range(cols):
       x1 = get_x(cols, -45)
       x2 = get_x(cols, 45)
       for lat in range(x1, x2):
           image[1,:,lat] = 255

   return image

new_image2 = insert_green(new_image)
bmp_io_c.output_bmp_c("image2.bmp", new_image2)

# insert blue
def insert_blue(image):
   for i in range(cols):
       x1 = get_x(cols, 45)
       x2 = get_x(cols, 135)
       for lat in range(x1, x2):
           image[2,:,lat] = 255

   return image

new_image3 = insert_blue(new_image2)
bmp_io_c.output_bmp_c("image3.bmp", new_image3)

# insert yellow
def insert_yellow(image):
   for i in range(cols):
       x1 = get_x(cols, -180)
       x2 = get_x(cols, -135)
       for lat in range(x1, x2):
           image[1,:,lat] = 255
           image[0,:,lat] = 255

   return image

new_image4 = insert_yellow(new_image3)
bmp_io_c.output_bmp_c("image4.bmp", new_image4)

# insert yellow
def insert_yellow2(image):
   for i in range(cols):
       x1 = get_x(cols, 135)
```

```
        x2 = get_x(cols, 180)
        for lat in range(x1, x2):
            image[1,:,lat] = 255
            image[0,:,lat] = 255

    return image

new_image5 = insert_yellow2(new_image4)
bmp_io_c.output_bmp_c("image5.bmp", new_image5)
```

flat_projection.py

```
import numpy as np
import bmp_io_c
import math
from numpy.linalg import inv

# read the sample image
rows, cols, pixels = bmp_io_c.input_bmp_c("image_test02.bmp")
print(rows, cols)
print(pixels)

# bi-linearly interp
def lininterp (lions_image, x, y):

    if x < 0 or x > (cols - 1):
        return 0, 0, 0

    if y < 0 or y > (rows - 1):
        return 0, 0, 0

    x1 = math.floor(x)
    x2 = math.ceil(x)
    y1 = math.floor(y)
    y2 = math.ceil(y)


    if x1 == x2 or y1 == y2:
        return lions_image[0, y1, x1]

       mat = np.asarray([[1, x1, y1, x1*y1], [1, x1, y2, x1*y2], [1, x2, y1,
x2*y1], [1, x2, y2, x2*y2]])
```

```
   mat = np.transpose(np.linalg.inv(mat))

   coordinates = np.dot(mat, np.asarray([[1], [x], [y], [x * y]]))

   R = (float(coordinates[0]) * lions_image[0, y1, x1]) +
(float(coordinates[1]) * lions_image[0, y1, x2]) \
   + (float(coordinates[2]) * lions_image[0, y2, x1]) +
(float(coordinates[3]) * lions_image[0,y2, x2])

   G = (float(coordinates[0]) * lions_image[1, y1, x1]) +
(float(coordinates[1]) * lions_image[1, y1, x2]) \
   + (float(coordinates[2]) * lions_image[1, y2, x1]) +
(float(coordinates[3]) * lions_image[1, y2, x2])

   B = (float(coordinates[0]) * lions_image[2, y1, x1]) +
(float(coordinates[1]) * lions_image[2, y1, x2]) \
   + (float(coordinates[2]) * lions_image[2, y2, x1]) +
(float(coordinates[3]) * lions_image[2, y2, x2])

   return R, G, B

# convert from CICS to PICS
def cicsToPics(x,y,r,c):
   return (x + (math.floor(c/2))), (y + math.floor((r/2)))

def compute_cone(d, rows, cols, lions_image, black_image):
   # theta = inv_of_tan(c/2d) and phi = inv_of_tan(R/2d)

   thetaM = np.arctan(cols/(2 * d))
   phiM = np.arctan(rows/(2 * d))

   deg_to_pix = 2048/180

   delta = np.pi/2048

   # Loop from -π/2 to π/2 in Longitude and -π/2 to π/2 in Latitude in
increments of Δ.
   for i in np.arange((-np.pi/2), (np.pi/2), delta):
       for j in np.arange((-np.pi/2), (np.pi/2), delta):

           if abs(i) > thetaM:
```

```
                continue

            if abs(j) > phiM:
                continue

            # for the formulas we know x = r * cos phi * sin theta and y = r
* sine phi and
            # z = r cos phi * cos theta and z = d so using the 3 d values
given, lets compute r
            # Let z = d, the viewing distance compute r, then x, then y

            z = d
            r = z / (np.cos(j) * np.cos(i))
            x = r * np.cos(j) * np.sin(i)
            y = r * np.sin(j)

            print(r, x, y)
            # (x, y) are in CICS. Convert to PICS
            pics_x, pics_y = cicsToPics(x, y, rows, cols)

            r,g,b = lininterp(lions_image, pics_x, pics_y)

            theta_deg = np.rad2deg(i)
            phi_deg = np.rad2deg(j)

            imageCX = theta_deg * deg_to_pix
            imageCY = phi_deg * deg_to_pix

            # print(imageCX,imageCY)

            imagePX, imagePY = cicsToPics(imageCX, imageCY, 2048, 2048)

            black_image[0, int(imagePY), int(imagePX)] = r
            black_image[1, int(imagePY), int(imagePX)] = g
            black_image[2, int(imagePY), int(imagePX)] = b

if cols > rows:
   d = cols
else:
   d = rows
```

```
dist_1 = 0.5 * d
dist_2 = 1.0 * d
dist_3 = 1.5 * d

myimage11 = np.zeros([3, 2048, 2048], np.uint8)
compute_cone(dist_1, rows, cols, pixels, myimage11)
bmp_io_c.output_bmp_c("myimage0.5.bmp", myimage11)

myimage22 = np.zeros([3, 2048, 2048], np.uint8)
compute_cone(dist_2, rows, cols, pixels, myimage22)
bmp_io_c.output_bmp_c("myimage1.0.bmp", myimage22)

myimage33 = np.zeros([3, 2048, 2048], np.uint8)
compute_cone(dist_3, rows, cols, pixels, myimage33)
bmp_io_c.output_bmp_c("myimage1.5.bmp", myimage33)
```