Demystifying AUGMENTED REALITY

Ramkumar Narayanan

I dedicate this document at the lotus feet of Sadguru Mata Amritanandamayi Devi. I also dedicate this book to my parents, brother and to all my teachers and friends.

Chapter 1

Introduction to Augmented Reality

What is Augmented Reality?
It is the merging of the real world with the 3D computer graphics world. In essence, you are overlaying 3D objects onto real world objects.

What does this book focus on?
This book primarily focuses on unraveling the basic concepts of Augmented Reality (AR). There are several building blocks in making a complete AR application. We will implement a rudimentary version each of these blocks. Of course, there are libraries that outperform ours. After we build a rudimentary version, we will download some of these libraries and integrate them with our code for enhanced efficiency. The motive of this book is to understand how AR technology works - to take a peak under the hood. Performance will not be the primary focus though at some parts we might discuss that as well. This book will provide a complete pipeline right from downloading and integrating libraries to creating a fully functional AR demo. In essence, this book will focus only on the crux of AR, other paraphernalia is beyond the scope.

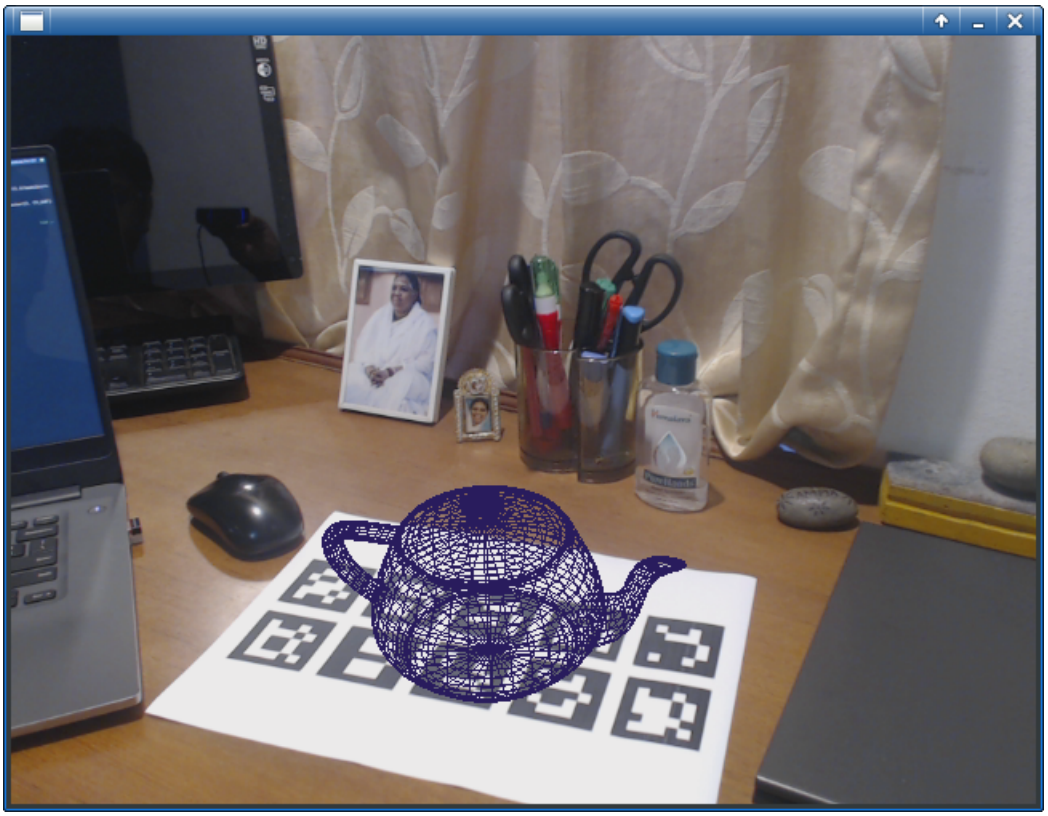


Figure 1.1: AR illustration - wireframe teapot on an ARUCO marker

What are the prerequisites for following this book?

I will assume that you have a decent background in linear algebra (if not, a great material to start with is Gilbert Strang's lecture on Linear Algebra - 18.06). A decent C++ programming experience is also necessary. Also, I will be extensively using python-2.7 to explain many of the concepts as it abstracts a lot of the implementation nuances, thus enabling us to focus on the concepts alone during the learning phase. Python is a very easy language to learn, so don't worry if you are not well versed with it; you can pick it up on the way. We will be developing all of our applications on Linux. Hence familiarity with Linux will come in handy.

What are the equipments/softwares I need before I start of with this?

1. A working computer with a decent RAM (4 GB is good) and atleast an i3 processor or equivalent hardware spec.

2. Linux environment - preferably Ubuntu/Arch (or similar flavours). I will presume most of you are using Ubuntu 16.04, and the installation instructions will be catered to that. Feel free to use any other flavor of linux as long as you obtain all the tools and packages that we will discuss as we move on - its not that hard. If you are using Windows or Mac, that is fine too as long as you can set all the packages up yourselves.

3. We are working with computer vision libraries, so get yourself a decent webcam. Built in webcams on the laptop will work just fine as you start.

4. And last and the most important, a curious mind :P

Chapter 2

Setting up the work environment

I hope at this point you have a working copy of linux installed on you machine. If not there are tons of youtube videos that can baby walk you through this process.
Before we dive head long into programming, let us setup the programming environment. This chapter will be solely dedicated to this. This is a bit of unavoidable drudge work. Unfortunately, most great projects start with this step. The first step is to setup an editor. This is a very important step as you are going to spend quite a bit of time programming. It is important to have an aesthetically pleasing environment that is comfortable for the eye. I prefer Vim (or Emacs) but Sublime or Atom are pretty good too. In the interest of providing one complete procedure to do things, let us install sublime text editor. Copy and paste the following commands on your Ubuntu terminal to install Sublime-text-editor.

```
sudo add-apt-repository ppa:webupd8team/sublime-text-3
sudo apt-get update
sudo apt-get install sublime-text-installer
```

Just like above, I will be using green text on black background to denote terminal commands henceforth.

We will proceed by setting up scripts for basic compilation, and installing certain libraries that we need to begin with. As we progress through several chapters, we will require more libraries and we will deal with them then. Sections 2.1 and 2.2 will help setup and run our first code in *C++* and *python*. Section 2.3 describes how to setup our first library-*OpenCV* in both cpp and python.

2.1 Compilation of first program in C++

After having set up the editor, let us first write a sample C++ code and compile it. We wont be doing anything fancy, just a simple *Makefile* system to compile the code.

Create a *helloworld.cpp* file in your working directory. Insert the following code into the file.
Note: I will be using the term *working directory* very often. By this, I simply mean a directory/folder that you choose to put your files in for the current topic. It could be located where ever you like.

```
#include <iostream>

int main()
{
   std::cout << "Hello World" << std::endl;
   return 0;
```

```
}
```

Create a document called *Makefile* in the same directory. Open the *Makefile* and insert the following code into the file.

```
CC = g++

INCLUDE =
LIBS =

all:  helloworld.cpp
   $(CC) $(INCLUDE) helloworld.cpp $(LIBS) -o helloworld

clean:
   rm ./helloworld
```

The first line of the *Makefile* states the compiler we will be using i.e. *g++*. The second and third line is where we add the path for *include* and *library* files necessary to compile and run the code. INCLUDE describes the location of all the header file viz,. *.h, .hpp etc.* LIBS point to the library files such as *.so, .a etc.* Since we are not using any external libraries, we will leave these fields blank for now. The fourth line describes the compilation command. The output of the code is named helloworld. The last line describes how to clean up the directory. In our case, we just have the executable helloworld. To compile the code, open the terminal pointing to the working directory where your code is and type

```
make
```

Notice in your working directory, an executable named *helloworld* would have appeared. To execute the program, type

```
./helloworld
```

You will see your terminal output *Hello World*. This is how we will be compiling and executing the code. If a new program is written, the appropriate filename and the output executable name is changed. Ofcourse, if external libraries are used, we will add some content to the INCLUDE and LIBS fields of the *Makefile*.

2.2 Running our first python code

Let us now try to write a python script and run it. Create a helloworld.py in your working directory and open it. Type the following code.

```
print "Hello World"
```

Let us try running it. On the terminal type

```
python helloworld.py
```

The terminal should output "Hello World".

2.3 Setup OpenCV

Let us now setup OpenCV. I will be using OpenCV 2.4.13 to work with throughout this book. You can install any version you like. OpenCV has several dependencies and to install them, type the following in your terminal.

```
sudo apt-get install libopencv-dev build-essential checkinstall cmake pkg-config yasm
  libtiff4-dev libjpeg-dev libjasper-dev libavcodec-dev libavformat-dev libswscale-dev
  libdc1394-22-dev libxine-dev libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev
  libv4l-dev python-dev python-numpy libtbb-dev libqt4-dev libgtk2.0-dev
```

Next download a copy of OpenCV 2.4.13 and extract the zip file. I'd generally prefer downloading and extracting all the libraries in the ∼/Downloads directory. However, its your choice.

```
wget https://github.com/Itseez/opencv/archive/2.4.13.zip
unzip opencv-2.4.13
```

Next navigate into the OpenCV folder. Enter the following commands in your terminal.

```
  cd opencv-2.4.13
  mkdir build
  cd build
  cmake -D WITH_XINE=ON -D WITH_OPENGL=ON -D WITH_TBB=ON -D BUILD_EXAMPLES=ON ..
  make -j4
  sudo make install
```

The entire process will take a while.

Finally, open /etc/ld.so.conf using the command (note I am using sublime text editor here, you can substitute it with your text editor)

```
  sudo subl /etc/ld.so.conf
```

Add the below line to the end of the file

```
/usr/local/lib
```

Let us now test if our OpenCV installation works properly. From this point onwards, you can find all the code on github (https://github.com/ramkalath/Augmented_Reality_Tutorials). To download (clone) the entire repository of code insert the following command on your terminal. Remember to insert the path to the directory where you want to download the code.

```
sudo apt-get install git
git clone https://github.com/ramkalath/Augmented_Reality_Tutorials <your path to work
   directory>
```

2.3.1 Test OpenCV with C++

Let us type a small code to open and display a *.jpg* file on a window. Find a *.jpg* file or download it. Create a *test_ opencv.cpp* file in your working directory and enter the following code.

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

int main()
{
   cv::Mat img;
   img = cv::imread("test.jpg");
   cv::imshow("my first opencv window", img);
   cv::waitKey();
   return 0;
}
```

The first few lines describe the header files required. The first code line in main(), describes the data type of the image i.e. Mat. Mat is a data container encapsulating several attributes about the image viz, the type, size, shape, etc etc. imread() method reads the data from our image. Incase the image is not in the working directory, entire path along with the image name has to be provided, *cv::imread("/path/image_ name.jpg").* *imshow("title of the widndow", image)* paints the data inside a window. The waitKey() method with no parameters waits for the user input before it closes the window. To compile the program, create a *Makefile* with the following contents. Remember we wrote a simple *Makefile*, we have to add the INCLUDES and LIBS.

```
CC=g++

INCLUDE = `pkg-config -cflags opencv`
LIBS = `pkg-config -libs opencv`

all:  test_opencv.cpp
   $(CC) $(INCLUDE) test_opencv.cpp $(LIBS) -o test_opencv

clean:
   rm ./test_opencv
```

Note in the Makefile, the ìn INCLUDE and LIBS is a backtick. Backtick is the key multiplexed with tilda, $\sim$ on standard keyboards. *pkg-config –cflags opencv* command returns the location of opencv headers and *pkg-config –libs opencv* command returns the location of opencv lib files. Compile using the *make* command and run using the *./test_ opencv* command. Terminal commands can be made to run

one after another using the && operator on the terminal. If this program runs without errors, you have configured OpenCV correctly.

2.3.2 Test OpenCV with Python

Now to obtain the same using python, type the following code in a new python file named *test_ opencv.py*.

```
import cv2
import numpy as np

image = cv2.imread("test.jpg")
cv2.imshow("my first opencv window", image);
cv2.waitKey()
```

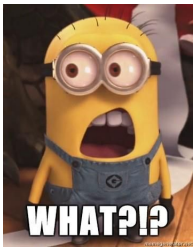
The function calls are exactly the same as that in the .cpp file. To run this type the following command in the terminal.

```
python test_opencv.py
```

You will be presented with an OpenCV window showing the image. Great, now we have setup our work environment, we can get started learning AR.

Chapter 3

Feature Detection and Matching



Relax! Before we begin delving into the concepts of AR, we are going to learn how to stitch two images. **Image Stitching** is a very rudimentary form of AR and it becomes a lot easier to learn AR once we master image stitching.

What is image stitching? It is the process of spatially inserting an image with common visual features into the other such that the output is a spatially collated image.

(a) (b)

Figure 3.1: images with common visual features

The above two images have common visual features. They can be stitched together to obtain the below image.

Figure 3.2: stitched image

The various functional blocks involved in stitching is described in figure 3.4.

In this chapter we will see how to perform Feature Detection and Matching. One of the main tasks in computer vision is finding out matching features between two images. For illustration, take a look at the side-by-side appended images as shown in figure 3.3. By feature matching, we have to find out certain points on the left image of figure 3.3 and its spatially matching corresponding points on the right image of figure 3.3. *Feature Detection* is the process of finding out such points and *feature matching* is the process of matching these points between images. The end result will be as shown in figure 3.3 where the points to be matched (a.k.a. *Keypoints*) are denoted as red dots and the matches are indicated in green.

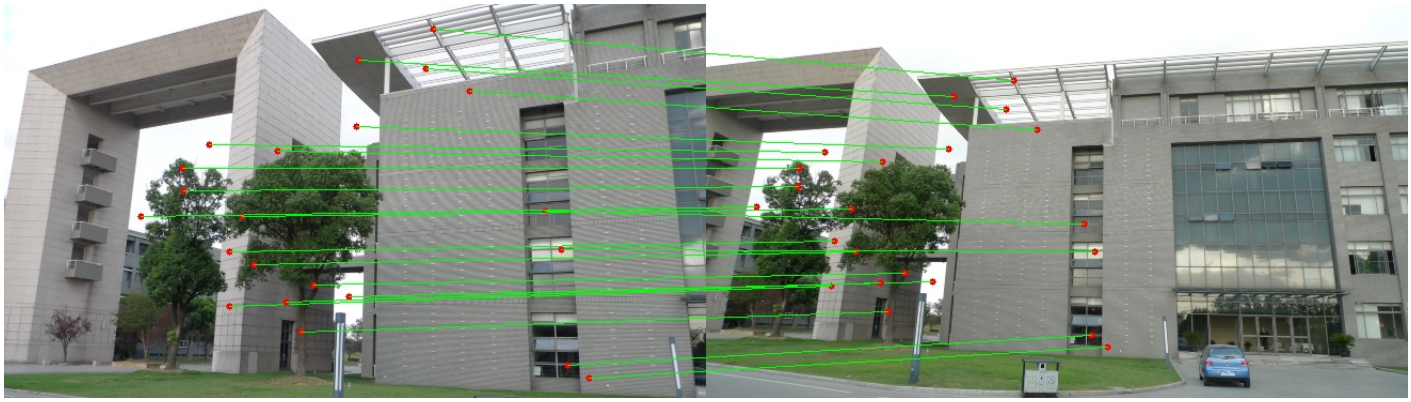


Figure 3.3: Feature Detection and Matching: Keypoints are indicated in red and the matches are denoted by green lines.

David Lowe's paper *Distinctive Image Features from Scale-Invariant Keypoints* discusses a method to extract stable keypoints and match them. Keypoints are red dots on the image above.
In a gist, the entire process involves construction of scale space where an image is analyzed over several magnifications. Certain extrema points that are repeating over several of these scales are chosen. Several orders of Laplacian of Gaussian(LOG) approximation is done. Keypoints are generated and bad ones are discarded. Each of these keypoints are associated with a set of descriptors that describes the area around the keypoint. All this is done separately for both the images. Keypoint matching is done by comparing the descriptors of one image with the descriptors of the other image. If a close correlation is found, then we have a possible match.
Don't worry if you did not understand anything from the above paragraph. David Lowe has created a binary file that gives us a list of all the keypoints and the associated descriptors of an image dumped into

a file. Open *http://www.cs.ubc.ca/~lowe/keypoints/* and click the *sift demo program*. This downloads the sift binaries zip file into your working directory. Extract the zip file. Since the binaries are 32 bit and it will not run if your operating system is 64 bit, you will have to run the following command on your terminal.

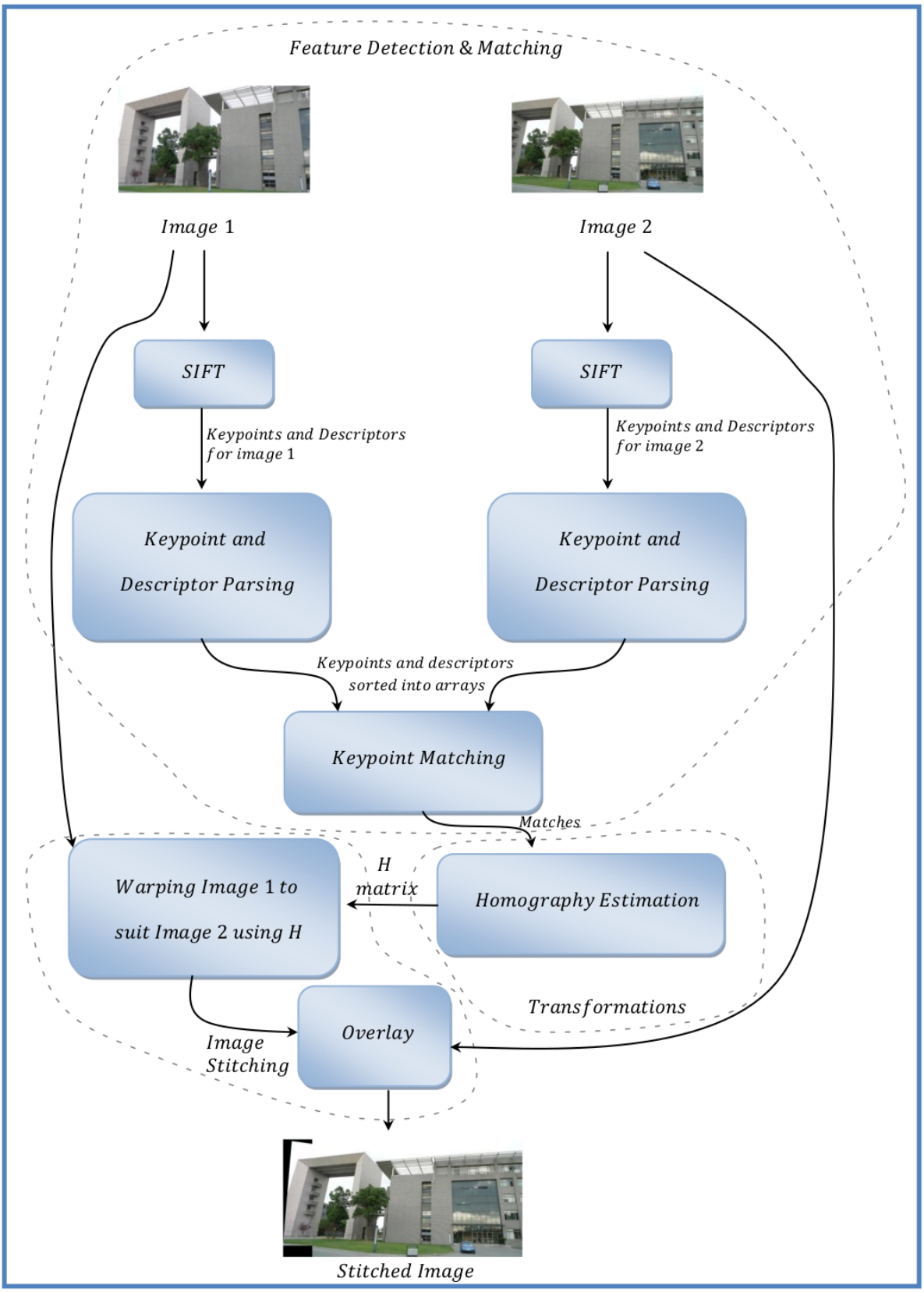


Figure 3.4: Flowchat illustrating the stitching operation

```
sudo dpkg -add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

Now run the sift binaries

```
./sift<book.pgm>book.key
```

This generates a key file in the working directory. Open the key file. It will look somewhat like this

```
882 128   <- 882 is the number of keypoints, 128 is the length of descriptor vector
138.42 44.74 17.49 0.159      <- (x, y, rotation_x, rotation_y) of keypoint number 1
7 0 0 0 0 0 0 0 116 1 0 0 3 4 13 66 13 0 0 11
82 62 27 31 2 2 1 7 41 43 16 7 38 2 0 0 0 0 0 0
150 14 3 2 14 9 1 43 44 4 7 35 150 81 3 19 14 17 7 17
44 33 4 2 76 1 0 0 0 0 0 3 150 42 21 24 11 8 1 25    <- these are the 1st set of
  128-length descriptors corresponding to keypoint 1
21 14 81 150 83 45 1 2 38 6 38 78 31 19 0 15 99 4 0 0
0 0 0 11 150 29 9 6 14 35 10 75 4 2 16 44 83 150 7 7
74 12 10 20 19 41 5 27
178.68 143.92 17.88 -0.088       <- 2nd Keypoint starts
12 4 1 12 76 68 9 5 24 28 9 15 17 21 14 3 88 60 4 6
22 8 1 4 1 1 0 1 124 48 1 1 26 15 38 93 46 73 7 1
33 8 34 90 45 32 15 3 124 25 3 11 13 14 9 55 9 0 0 1
124 67 7 11 42 9 16 38 26 124 28 9 33 6 17 36 35 80 55 4     <- these are the 2nd
  128-length set of descriptors corresponding to 2nd keypoint
124 119 15 10 19 9 6 19 10 23 15 23 124 13 1 2 39 43 82 60
15 37 10 4 1 4 41 113 55 25 7 0 25 124 22 27 12 0 0 0
12 59 16 35 80 0 0 0
.
.    there are 882 keypoints each with 128 descriptors
.
```

Okay thats enough theory! Lets get coding this stuff. We will be using David Lowe's prebuilt binary for sift. Alternatively, you can use the sift implementation in OpenCV.

To begin with obtain two images with enough details (by details I mean images with lots of visual features). The images ofcourse should have overlapping contents just like the image shown in figure 3.1. Our job is to find the matching keypoints of figure that corresponds to matching keypoints of figure. Let us write a python code to do the same. Since David Lowe's binary accepts only .pgm files, we have to convert the existing image (maybe with extension .jpg or .bmp etc) into .pgm. Then run the sift binary on the .pgm images to obtain the key files for both the images. We then parse the keypoints and descriptors from the *.key* into arrays using our python program before we test for matches. Let us use the python imaging library for conversion. There are certain other dependencies we would require; like numpy and scipy. Let us install them using the command,

```
sudo pip install Pillow
sudo apt-get install python-scipy
```

```
sudo pip install python-numpy
```

The basic algorithm we would be coding is as follows,

1. Checks if the image is a *pgm* file, if not converts the file into pgm.
2. Applies sift binary on the image to obtain a *.key* file that describes all the keypoints and descriptors.
3. Parses the *.key* file to obtain a list of keypoints and descriptors.
4. Matches the descriptors of both the images to obtain strong matches.
5. Keypoints corresponding to strong matching descriptors are dumped into a list.
6. Lines are drawn between matching keypoints after appending the images.

Let us do all the imports first. Then let us start with the main.

```
import numpy as np
import cv2
from PIL import Image as im
import sys
import os
import scipy.misc as sc
import time

if __name__ == "__main__":
   img_path = sys.argv # we pass the images to be matched as commandline arguments
   img1 = cv2.imread(img_path[1]) # image 1 -> command line argument 1
   img2 = cv2.imread(img_path[2]) # image 2 -> command line argument 2
```

Okay, now we have our images stored in variables *img1* and *img2*. The next step is to check if the images have the extension *.pgm*, if not convert the image to *.pgm*. Let us write a small function to do exactly that.

```
def imagecheck(img_path, n):
   if img_path[-3:] != "pgm":
      pgm_img_name = "tmp"+str(n)+".pgm"
      im.open(img_path).convert('L').save(pgm_img_name)
   return pgm_img_name

# The call statement should be written in main
img_path[1] = imagecheck(img_path[1], 1)
img_path[2] = imagecheck(img_path[2], 2)
```

The function *imagecheck* takes in the path of the image and a number variable as arguments. We use the number variable as image number index. Next we check if the last three characters of the image has the extension *pgm* , if not enter within the if block. We are going to convert the image to *.pgm* for processing. The pgm file is temporary and after processing we will discard it. The next line of code creates

a string named *tmp#.pgm* where # denotes image number. The next line converts the image to greyscale and saves it with the string name as define above.

Great, we have our *.pgm* image files. Now let us apply the sift binary on the image. The command for using the sift binary via the terminal is

```
./sift<image.pgm>output_file.txt
```

output_file.txt will contain the keypoints and descriptor information.

```
def sifti_fy(image,n):
    os.system("./sift <"+image+"> "+str(n)+".txt")
```

os.system() enables us to enter a terminal command and execute it from within the python script. We create the arguments for the function by combining the strings to form a meaningful command for our context.
The next part is to parse the key files to extract the keypoints and descriptors from the key files and dump them into numpy arrays. The package numpy in python exclusively handles arrays and enables us to manipulate the array in many ways.

```
def chunker(iterable, chunksize):
    return map(None,*[iter(iterable)]*chunksize)

def read_key(n):
    f = open(str(n)+".txt")
    lines = f.readlines()
    del lines[0]    #first line is not necessary
    #-------------------------------------
    # keypoints
    kp = []
    k = lines[0::8]
    del lines[0::8]
    for i in k:
        kp.append(map(round,map(float, i.split(" "))))
    #-------------------------------------
    #descriptor
    des = []
    desc = chunker(lines,7)
    for i in desc:
        l = "".join(list(i))
        des.append(map(int, l.strip().split(" ")))
    return np.array(kp).T[:-2].T, np.array(des)
```

The first line opens the key file and the second line parses all the lines in the file and dumps each line into a list of strings called *lines*. As we parse each string from *lines*, we delete it. The first line in the *key* file tells us the number of keypoints and the length of each of the descriptors. We can delete that. *kp = []* creates an empty list and we will append they keypoints as we keep reading them. Keypoints are found in every

8th line, we enumerate all of them into the variable k and delete them afterwards. Next lets move onto the line in the loop. The map function simultaneously operates on a list of numbers. This line performs a series of operations starting with splitting the list where ever it finds a single space " ". Then the map function is used to convert it to float, then roundoff. The result is appended to the variable kp.

Let us try the descriptors now. Next line we create an empty list called *des*. The chunker function reads through the remaining list of *lines* and groups them such that there are seven lines per group. If you notice the *key* file, each descriptor spans seven lines. Now we have to clean it up and join the descriptors. The first line in the for loop does exactly that. The second line splits the descriptors with single space as the delimiter and converts the descriptors to integers from strings. This is then appended to the empty list *des*. Finally, the *kp* and *des* is returned. Note that each row in *kp* has 4 values consisting of *(x, y, rotation_value, rotation_value)*. Only *x, y* values are of use to us. *np.array(kp).T[:-2].T* converts the list to a numpy array and transposes it next, then takes the list of two rows and trasposes it again. The *des* is converted to numpy array and returned.

Awesome! Now we have a list of keypoints and its corresponding list of descriptors that we parsed out of the *key* file. The next step is to match the descriptors of one image to the descriptors of the other. Each descriptor of image 1 is compared to each descriptor of image2. If a strong match is found, then the keypoints corresponding to the strong matches are appended into a list. Lets see how that is achieved.

```
def match(desc1,desc2):
    """ For each descriptor in the first image, select its match in the second image.
        input: desc1 (descriptors for the first image), desc2 (same for second image).
        """

    desc1 = np.array([d/np.linalg.norm(d) for d in desc1])
    desc2 = np.array([d/np.linalg.norm(d) for d in desc2])

    dist_ratio = 0.6
    desc1_size = desc1.shape

    matchscores = np.zeros((desc1_size[0],1),'int')
    for i in range(desc1_size[0]):
           dotprods = np.dot(desc1[i,:],desc2.T)
           dotprods = 0.9999*dotprods
           indx = np.argsort(np.arccos(dotprods))
           if np.arccos(dotprods)[indx[0]] < (dist_ratio *
               np.arccos(dotprods)[indx[1]]):
                   matchscores[i] = int(indx[0])
    return matchscores
```

First let us create a function called *match* and to the match function, the descriptors of image 1 and image 2 are passed. The first and the second lines normalizes the descriptors. We then set a difference treshold for strong matches as 0.6. Correlation (dot product) is the simple process of finding the descriptor match strength. *matchscores* is a numpy array to hold the list of match strengths. Let us compute the match strengths next. For each descriptor of image 1, we find the dot product with all the descriptors of image 2. I don't know why I multiplied the dot products by 0.9999, perhaps it threw an error and somewhere I read this as a fix, I think. Anyway, this does not change the strengths of our matches. $\cos^{-1}(dotprods)$ gives us the angle of deviation of the descriptor vectors. The matches are sorted and their index is returned. The *indx* list returns the indices of the sorted (decreasing) array of matches which describes the match strengths of descriptors of image 2 with the i^{th} descriptor of image 1. For a clear

winner to emerge, the difference match strength between the first and the runner up should be large i.e. if the match strengths between the first and second are close by then its an ambiguous match. If an ambiguous match occurs, discard the entire match and try another descriptor.

The next step is to find the list of keypoints corresponding to the matching descriptors.

```
def findkps(matchindx,kp1, kp2):
        matches = []
        locations = np.nonzero(matchindx)[0]
        for i in locations:
                matches.append([kp1[i], kp2[matchindx[i][0]]])
        return matches
```

That is easy, the above function takes in three arguments viz., match indices, keypoints 1 and keypoints 2 from images 1 and 2 respectively. *matchidx* contains a list of values, for example [0, 0, 0, 23, 0, 34, ...]. The 0's denote no proper matches while the positive numbers show index of descriptors of image 2 that matches with the descriptors of image 1, i.e, the third descriptor of image 1 matches with the 23rd descriptor of image 2. The *locations* variable output the non zero values from *matchidx*. *matches* creates a list of matching keypoints; it outputs the locations, e.g. $[x^k_{kp1}, y^k_{kp1}, x^l_{kp2}, y^l_{kp2}]$, where k^{th} keypoint of image 1 matches l^{th} keypoint of image 2.

If you have reached till this point, you have successfully found out how to find the matching features between two images.

Now let us try to draw this. First let us append these images side-by-side.

```
def append_images(img1, img2):
        h1 = len(img1)
        w1 = len(img1[0])
        h2 = len(img2)
        w2 = len(img2[0])
        appendedimage = np.zeros((max(h1,h2), w1+w2 ,3), np.uint8)
        appendedimage[:h1, :w1] = img1
        appendedimage[:h2, w1:w1+w2] = img2
        return appendedimage
```

The above function takes the two images as input. The first four lines find out the image heights and widths. *appendedimage* creates an empty template image, which sort of serves like a place holder for our new appended image; we fill it with zeros and its an 8bit image having 3 channels for b, g, r values and the height is the larger of the two images and the width is the sum of the widths of the images. The first second image data is copied to the *appendedimage* template next. *appendedimage* is then returned.

```
for i in matches:
       (x1, y1) = tuple(np.array([i[0][1],i[0][0]],int))
       (x2, y2) = tuple(np.array([i[1][1],i[1][0]],int))
       cv2.circle(appendedimage, (x1,y1), 3, (0,0,255), 0)
       cv2.circle(appendedimage, (x2+w,y2), 3, (0,0,255), 0)
```

```
    cv2.line(appendedimage, (x1, y1),(x2+w, y2), (0,255,0), 0)
```

This snippet of code draws a small circular dot and a line across the appended image. The first two lines yield the extracted keypoint values. The second line takes in 5 arguments viz., image you want to draw on, center point of the circle, size of the circle, color of the circle and shift value which we set to 0. The last line of the snippet draws a line between two points taking the arguments, image, point1, point2 (remember this has to be shifted by a width, w denoting image1 width), color and shift=0. Below is the function calls for all the functions written above.

```
if __name__ == "__main__":
    img_path = sys.argv
    img1 = cv2.imread(img_path[1])
    img2 = cv2.imread(img_path[2])
    img_path[1] = imagecheck(img_path[1],1)
    img_path[2] = imagecheck(img_path[2],2)
    sifti_fy(img_path[1],1)
    sifti_fy(img_path[2],2)
    kp1,des1 = read_key(1)
    kp2,des2 = read_key(2)
    matchindx = match(des1,des2)
    matches = findkps(matchindx,kp1,kp2)
    appendedimage = append_images(img1, img2)
    w = len(img1[0])
    print "number of matches = ", len(matches);
```

For the complete code checkout ch3/feature_detection_and_matching.py on github.
To run this code type the following on the terminal.

```
python feature_detection_and_matching.py image1.jpg image2.jpg
```

Awesome! This concludes the feature matching topic with SIFT. There are a lot of feature detectors and matchers such as Harris corner detector, SURF, ORB, FREAK etc. OpenCV implementation of the same is much easier but this gives you a better peek at the innards of the entire process. Henceforth, if you wish to use an opencv implementation of SIFT, you can find the code for that on ch3/feature_detection_and_matching_OpenCV.py directory on github. Ofcourse OpenCV abstracts a lot of implementation details; abstraction is not good for learning.

3.1 Exercise

1) Try other feature detectors such as SURF and ORB.

Chapter 4

Transformations

In the last chapter, we tried to match features between images. The next functional block is Image Transformations. **What is an image transformation?** To stitch two images we have to twist and yank one image such that it fits appropriately into the other. To do that we need some sort of transformation matrix that warps the image for image stitching. To begin with let us start with a simple transformation i.e., rotation.

4.1 Rotations

How do you rotate a point in the xy plane?
Consider a point, (x_1, y_1) at θ_1 degrees from the positive side of the x *axis*. The distance of (x_1, y_1) is r from the *origin*. Our aim is to rotate this point by $\Delta\theta$ such that after rotation, our new point has the coordinate, (x_2, y_2) with θ_2 angular displacement from the positive of the x *axis* as shown in figure 4.1.

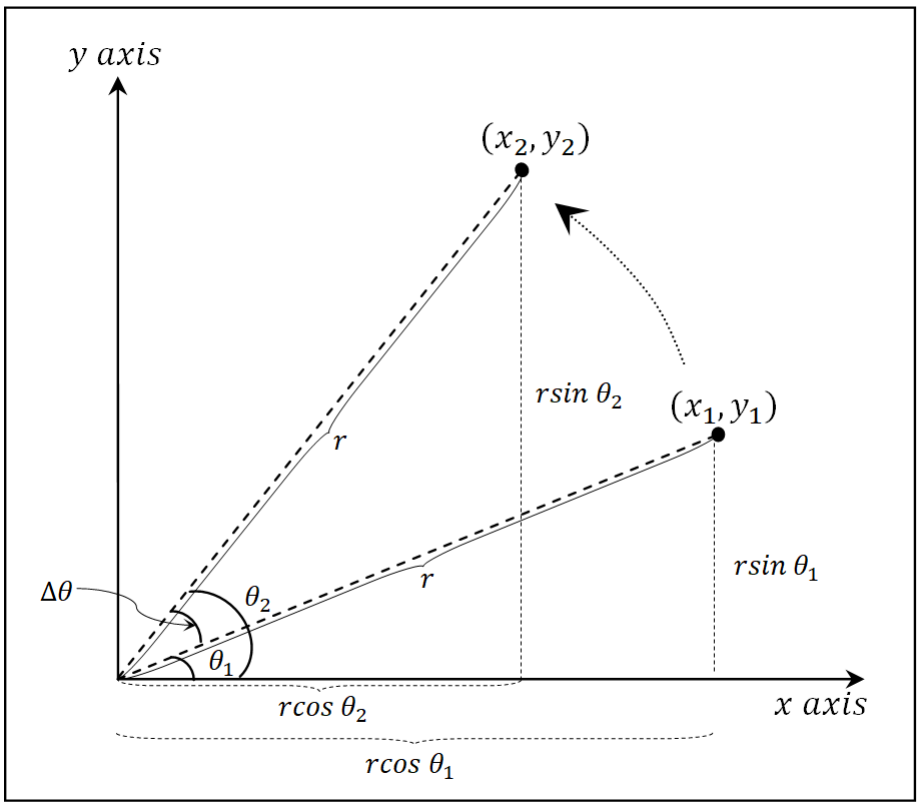


Figure 4.1: (x_1, y_1) rotated over $\Delta\theta$ to (x_2, y_2)

$$(x_1, y_1) = (r\cos\theta_1, r\sin\theta_1) \tag{4.1}$$
$$(x_2, y_2) = (r\cos\theta_2, r\sin\theta_2) \tag{4.2}$$
$$\theta_2 = \theta_1 + \Delta\theta$$
$$Therefore, (x_2, y_2) = (r\cos(\theta_1 + \Delta\theta), r\sin(\theta_1 + \Delta\theta))$$
$$(x_2, y_2) = (r\cos\theta_1\cos\Delta\theta - r\sin\theta_1\sin\Delta\theta), (r\sin\theta_1\cos\Delta\theta - r\cos\theta_1\sin\Delta\theta)$$

Substituting eq 4.1 and 4.2 in the above equation,

$$(x_2, y_2) = ((x_1\cos\Delta\theta - y_1\sin\Delta\theta), (y_1\cos\Delta\theta + x_1\sin\Delta\theta))$$

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\Delta\theta & -\sin\Delta\theta \\ \sin\Delta\theta & \cos\Delta\theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

Thus given a point (x_1, y_1), we have learnt to use the rotation matrix to rotate it over $\Delta\theta$ degrees to obtain its final destination coordinates, (x_2, y_2). An image is nothing but collection of a lot of pixels each with a unique color value and an index describing the position of that pixel on the image plane. To rotate an image by $\Delta\theta$, we list down all the indices one below the other i.e., we serialize the index values of the image as below. Then we multiply the indices with the rotation matrix to obtain the new index values.

$$\begin{bmatrix} x_f^0, y_f^0 \\ x_f^0, y_f^1 \\ x_f^0, y_f^2 \\ . \\ . \\ . \\ x_f^{nm}, y_f^{nm} \end{bmatrix}^T = \begin{bmatrix} \cos\Delta\theta & -\sin\Delta\theta \\ \sin\Delta\theta & \cos\Delta\theta \end{bmatrix} \begin{bmatrix} x_i^0, y_i^0 \\ x_i^0, y_i^1 \\ x_i^0, y_i^2 \\ . \\ . \\ . \\ x_i^{nm}, y_i^{nm} \end{bmatrix}^T$$

Note, in the above equation with each value in the form x_k^p, y_k^q, i and f describe the suffix for the initial and final image before and after the transformation. p and q describe the iterators over the index ranges of an nxm image. The color values of the initial image can now be substituted with the color values of the final image. Let us now code the same. One might argue that rotations are rudimentary and its unnecessary to implement them whilst learning AR. However, this exercise helps us get acquainted with the troubles we might face while performing transformations on images. Create a folder named rotations in your working directory and create a file named *(*rotations.py) in it. Find a *.jpg* file and put it in the working directory. Type the following code in the file.

```
# headers
import cv2
import numpy as np
import math

if __name__ == "__main__":
   frame = cv2.imread("image_name.jpg")
   height = len(frame)
   width = len(frame[0])
```

```
    theta = 45
```

The first line of reads the data from the image and dumps it into a numpy array. The second and third line of code reads the image height and width. Then we define an arbitary θ value equal to 45^0.

```
# the next step is to create a rotation matrix
rot_mat = np.array([[math.cos(np.radians(theta)), -math.sin(np.radians(theta))],
    [math.sin(np.radians(theta)), math.cos(np.radians(theta))]]);
# print rot_mat

# Let us try to create an index list of the image pixels starting from [[0, 0], [0,
    1].....[n, m]]---------------------
row1 = np.tile(np.arange(width), height)
row0 = np.repeat(np.arange(width), width)[:len(row1)]
indices_pre_rotation = np.vstack((row0, row1)).T
# print indices_pre_rotation
```

We then create a rotation matrix. Go ahead and try printing the rotation matrix. To rotate an image, we need to create an array of list of indices denoting the pixel positions of the image. That is a bit tricky. We use the tile and repeat methods to achieve the same. Then we stack and transpose it to obtain the list of indices. Try printing *indices_pre_rotation* to see how it looks like.

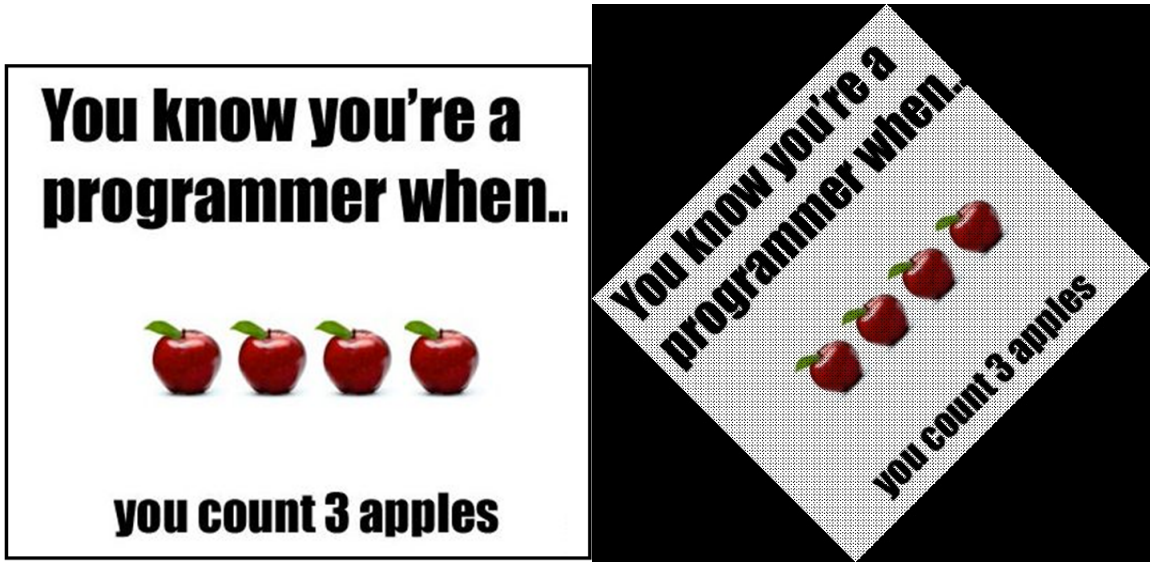
```
# Let us rotate the image ----------------------------------------------------------
indices_post_rotation = np.dot(rot_mat, indices_pre_rotation.T)
indices_post_rotation = np.around(indices_post_rotation.T)
# print indices_post_rotation
min_value_cols = min(indices_post_rotation.T[0])
max_value_cols = max(indices_post_rotation.T[0])
min_value_rows = min(indices_post_rotation.T[1])
max_value_rows = max(indices_post_rotation.T[1])

# lets create a new template image as a place holder for the rotated image
new_image = np.zeros((int(max_value_rows-min_value_rows+1),
    int(max_value_cols-min_value_cols+1), 3), np.uint8)
indices_post_rotation.T[0] = indices_post_rotation.T[0] - min_value_cols
```

Okay, once we have obtained the index list and the transformation matrix, let us try rotating the image. We multiply the rotation matrix with the indices list using the *np.dot()* method. The resultant *indices_post_rotation* gives us the results after rotation. The obtained values are floats and we round them off to quantize it to the nearest pixel value. The next step is finding the range of the rotated image on the *xy plane*. *min_value_cols, max_value_cols, min_value_rows and max_value_rows* give us the ranges. After that, we create a new template image which is going to host our rotated image. The last line of the code snippet shifts the negative indices to positive indices. In our case, we have negative indices only along the x axis.

```
frame = frame.reshape(width*height, 1, 3)
k = 0
for i in indices_post_rotation:
   new_image[int(i[0])][int(i[1])] = frame[k][0]
   k+=1

cv2.imshow("", new_image)
cv2.waitKey()
```

All that remains is to map the color values of the original image (*frame*) to the new template image (*new_image*) along the rotated index values (*indices_post_rotation*). The first line of this snippet reshapes the entire image into a long string so that we can use an iterator k to iterate over all the values of the image. Then, we substitute the *new_image* with the color values of *frame* along the *indices_post_rotation*. Finally we display the image using *imshow()* method. *waitKey()* method with no arguments waits indefinitely for a key press from the user to end the program, thus closing the window automatically.

Below figures show the before and after results of rotation. You can find the code in ch4/rotations/rotations.py.



(a) (b)

Figure 4.2: Before and after rotation. Notice the small aberrations in the rotated image.

Notice the rotated image has a lot of striations. This is because of the rounding off (*np.around*) method we performed which approximates the pixel values thereby leaving holes in the image. To remove this we need to do a post process called **interpolation**. There are several kinds of interpolation viz., bilinear (from performing linear interpolation twice), cubic, spline etc. Linear interpolation is the simplest of them, where the color value of the hole pixel is obtained by linearly interpolating and finding the mid value between the color values of the neighbouring pixels. Since the pixels are equidistant, this process resolves to the average of the pixel values of the neighbours being assigned to the hole pixel. Bilinear interpolation performs linear interpolation twice along the x and y axis. We wont be delving into those. Rotations on the *xy plane* or around the *z axis* represent one of the simplest transformations. What happens when you rotate an image outside the xy plane? Similar, to the images shown in the figure 4.3.

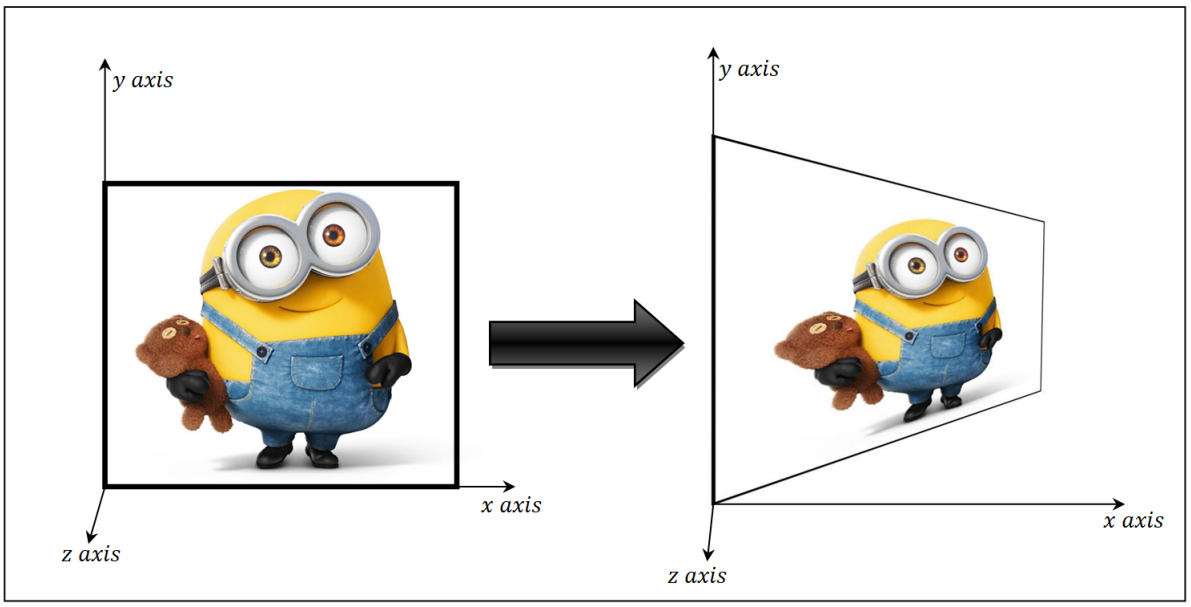


Figure 4.3: Perspective transformation of an image

Frame borders which were parallel lines before transformation start to converge after the transformation. This is called a perspective transformation. Distances between points further away seem smaller than the distances between points close by. To achieve that we need some sort of division by z component. z signifies the distance into or from the image plane. A concept called *Homogeneous Coordinate Systems* comes into play here. This branch of thought was first developed by by *August Ferdinand Mobius* in his 1827 work *Der barycentrische Calcul*. Using this coordinate system, we have a lot of advantages such as the ability to represent points at infinity as finite mathematically tangible points. Converting a 2 dimensional cartesian coordinate to homogeneous coordinate is simple. It is as easy as appending a 1 to every coordinate.

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \iff \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

The rotation transformation in the xy plane was represented by a 2x2 matrix. We will use a 3x3 transformation matrix to represent rotations over θ degrees.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For a series of transformations, the matrices can be cascaded on after the other to obtain the final transformation. For instance, if the transformation required us to rotate an image around the x axis by θ_1 degrees and around the y axis by θ_2 degrees afterwards, the cascaded transformation matrix will look like this,

$$\begin{bmatrix} x_0^f, y_0^f, w_{0,0} \\ x_0^f, y_1^f, w_{0,1} \\ x_0^f, y_2^f, w_{0,2} \\ . \\ . \\ . \\ . \\ x_{nm}^f, y_{nm}^f, w_{n,m} \end{bmatrix}^T = \begin{bmatrix} \cos\theta_2 & 0 & \sin\theta_2 \\ 0 & 1 & 0 \\ -\sin\theta_2 & 0 & \cos\theta_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_1 & -\sin\theta_1 \\ 0 & \sin\theta_1 & \cos\theta_1 \end{bmatrix} \begin{bmatrix} x_0^i, y_0^i, 1 \\ x_0^i, y_1^i, 1 \\ x_0^i, y_2^i, 1 \\ . \\ . \\ . \\ . \\ x_{nm}^i, y_{nm}^i, 1 \end{bmatrix}^T$$

The left side of the above equation gives a list of all the pixel values in the homogeneous coordinate system. To obtain the values in Cartesian coordinate system, simply divide the x's and y's by their corresponding w's.

4.2 Translations, Scaling and Shear

Similar to rotations, there are other transformations such as translations, scaling and shear and the transformation matrices for these can be derived easily. Below are the transformation matrices.

$$Translation\ matrix\ is\ given\ by \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$Scaling\ matrix\ is\ given\ by \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$Shear\ matrix\ is\ given\ by \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A combination of these transformations can be used to achieve what is called as perspective transformation.

In the previous chapter we learnt how to find correspondences between two images. Our goal in the remainder of this chapter is to calculate the transformation matrix given image correspondences. In our particular case, i.e., transforming images, the transformation matrix is also called *homography matrix*. We will be henceforth referring to it by this name. I recommend the readers to revisit the flow chart described in figure 3.4.

4.3 Estimation of Homography Matrix

What does a Homography matrix do? It transforms a list of pixel points from one perspective to another. From the output of chapter 3, we have a set of k matching image points, with i denoting *initial_image* and f denoting *final_image*, and c denoting a constant. They can be written as

$$\begin{bmatrix} c_0x_0^f & cy_0^f & c_0 \\ c_1x_1^f & c_1y_1^f & c_1 \\ c_2x_2^f & c_2y_2^f & c_2 \\ . & & \\ . & & \\ . & & \\ . & & \\ c_kx_k^f & c_ky_k^f & c_k \end{bmatrix}^T = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix}_{3x3} \begin{bmatrix} x_0^i & y_0^i & 1 \\ x_1^i & y_1^i & 1 \\ x_2^i & y_2^i & 1 \\ . & & \\ . & & \\ . & & \\ . & & \\ x_k^i & y_k^i & 1 \end{bmatrix}^T$$

We have to estimate the values of $h_{00}, h_{01}.....h_{22}$. A simple method to estimate this is *Direct Linear Transformation*(DLT). We convert them into a set of linear equations as follows. For the sake of illustration, let us take one point

$$\begin{bmatrix} cx^f \\ cy^f \\ c \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x^i \\ y^i \\ 1 \end{bmatrix}$$

This can be expanded to

$$h_{00}x^i + h_{01}y^i + h_{02} = cx^f \tag{4.3}$$

$$h_{10}x^i + h_{11}y^i + h_{12} = cy^f \tag{4.4}$$

$$h_{20}x^i + h_{21}y^i + h_{22} = c \tag{4.5}$$

Dividing equation *4.3* and *4.4* by *4.5* we get

$$h_{00}x^i + h_{01}y^i + h_{02} = x^f(h_{20}x^i + h_{21}y^i + h_{22}) \tag{4.6}$$

$$h_{10}x^i + h_{11}y^i + h_{12} = y^f(h_{20}x^i + h_{21}y^i + h_{22}) \tag{4.7}$$

Rearranging *4.6* and *4.7* we get,

$$h_{00}x^i + h_{01}y^i + h_{02} + h_{10}(0) + h_{11}(0) + h_{12}(0) + h_{20}(-x^fx^i) + h_{21}(-x^fy^i) + h_{22}(-x^f) = 0 \tag{4.8}$$

$$h_{00}(0) + h_{01}(0) + h_{02}(0) + h_{10}x^i + h_{11}y^i + h_{12} + h_{20}(-y^fx^i) + h_{21}(-y^fy^i) + h_{22}(-y^f) = 0 \tag{4.9}$$

This forms a set of homogeneous linear equations with $h_{i,j}$, as the unknowns. Each set of matching points between two images to be stitched gives us a pair of equations. Thus to estimate nine points we need atleast nine equations which can be only determined using atleast five points. However, in most cases h_{22} is 1. Thus, there are only eight unknowns now. A minimum of four matching points are required to perform image stitching.

$$\begin{bmatrix} x^i & y^i & 1 & 0 & 0 & 0 & -x^fx^i & -x^fy^i & -x^f \\ 0 & 0 & 0 & x^i & y^i & 1 & -y^fx^i & -y^fy^i & -y^f \\ x^i & y^i & 1 & 0 & 0 & 0 & -x^fx^i & -x^fy^i & -x^f \\ 0 & 0 & 0 & x^i & y^i & 1 & -y^fx^i & -y^fy^i & -y^f \\ x^i & y^i & 1 & 0 & 0 & 0 & -x^fx^i & -x^fy^i & -x^f \\ 0 & 0 & 0 & x^i & y^i & 1 & -y^fx^i & -y^fy^i & -y^f \\ & & & & & & \vdots & & \\ x^i & y^i & 1 & 0 & 0 & 0 & -x^fx^i & -x^fy^i & -x^f \\ 0 & 0 & 0 & x^i & y^i & 1 & -y^fx^i & -y^fy^i & -y^f \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

$$Ah = 0$$

Nullspace of the above equation gives us the solution for h. But, in cases when we obtain more number of matching points, which is usually the case, we obtain an over determined linear system. For more mathematically inclined readers, this means, we have more number of equations to solve a lesser number of unknowns. To solve an over determined system, we use methods such as least squares. The solution typically takes the form of a *cost bowl* function. The bowl function resembles the shape of a bowl with only one global minima. Let us implement DLT with four matching points so we need not worry about an over determined system. First we need to install a package called sympy which contains methods to calculate the nullspace of a matrix. To install sympy

```
sudo pip install sympy
```

Go ahead and create a file called DLT.py and insert the following code. We are going to code the following transformation as depicted by figure 4.4.

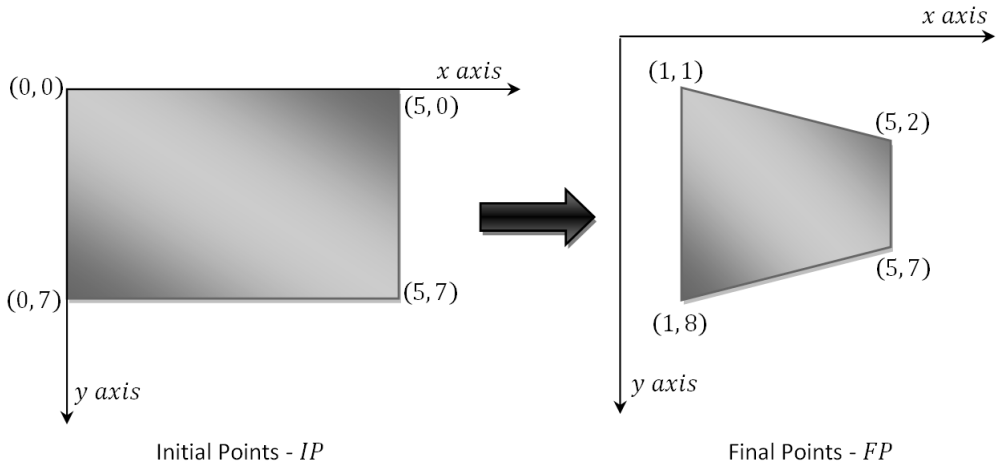


Figure 4.4: DLT transformation

```
import numpy as np
from sympy import Matrix

if __name__ == "__main__":
    npoints = 4    # since we have only 4 points
    ip = np.array([[0, 0], [5, 0], [5, 7], [0, 7]]) # initial points before transformation
    fp = np.array([[1, 1], [5, 2], [5, 7], [1, 8]]) # final points after transformation

    A = []

    # for each point append two rows of A
    for pointi in range(0, len(ip)):
        A.append([ip[pointi][0], ip[pointi][1], 1, 0, 0, 0, -fp[pointi][0]*ip[pointi][0],
            -fp[pointi][0]*ip[pointi][1], -fp[pointi][0]])
        A.append([0, 0, 0, ip[pointi][0], ip[pointi][1], 1, -fp[pointi][1]*ip[pointi][0],
            -fp[pointi][1]*ip[pointi][1], -fp[pointi][1]])
```

```
A = Matrix(A)  # creating a sympy matrix
h = np.array(A.nullspace()) # calculating the homography matrix
h = h.reshape(3, 3)   # reshaping h to form a 3x3 matrix
print h
```

You can find the code for the following in ch4/DLT/DLT.py. If there are more than 4 points, we can use the final column of v from $[u, s, v] = np.linalg.svd(A)$ to obtain the h matrix. However, for implementation sake we have a nice function called *findhomography* that abstracts the entire process.

```
h, status = cv2.findHomography(matchimg1,matchimg2,cv2.RANSAC,5.0)
```

The above function takes in the matching points as parameters and it uses a method called Random Sample Consensus, which does the same job as DLT but its much faster in estimating the homography matrix, h. We will be using this to stitch two images in the next chapter.

4.4 Exercises

Exercise 4.1: Try fiddling around with different combinations transformation matrices and observe that transformations are associative but not commutative.

Associative:

$$final\_points = (AB)C * initial\_points = A(BC) * initial\_points$$

Commutative:

$$A * B * points \neq B * A * points$$

Chapter 5

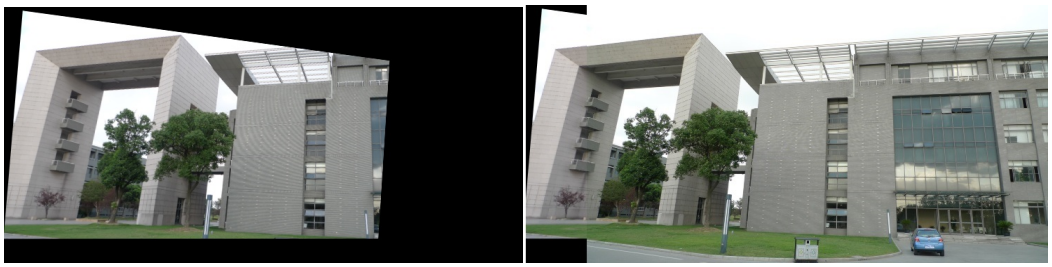
Image Stitching

In this chapter, we are going to finally deal with the last block that completes the entire operation of image stitching. Now, remember we performed the rotation operation on an image in *chapter 4* and the resultant image had holes in it. And thus we discussed that the transformation operation has to be succeeded by interpolation. This entire process is called **warping**. Warping here is carried out using OpenCV's *warpPerspective()* method. We also created a *template image* in chapter 4, that acts as a place holder for the rotated (transformed) image. We have to do the same here. Let us start by creating the template image.

```
def warpTwoImages(img1, img2, H):
   '''warp img2 to suit img1 with homography H'''
   h1,w1 = img1.shape[:2]
   h2,w2 = img2.shape[:2]
   pts1 = np.float32([[0,0],[0,h1],[w1,h1],[w1,0]]).reshape(-1,1,2)
   pts2 = np.float32([[0,0],[0,h2],[w2,h2],[w2,0]]).reshape(-1,1,2)
   pts2_ = cv2.perspectiveTransform(pts2, H)
   pts = np.concatenate((pts1, pts2_), axis=0)
   [xmin, ymin] = np.int32(pts.min(axis=0).ravel())
   [xmax, ymax] = np.int32(pts.max(axis=0).ravel())

   # print "xmax = ", xmax
   # print "xmin = ", xmin
   # print "ymax = ", ymax
   # print "ymin = ", ymin
   t = [-xmin,-ymin]
   Ht = np.array([[1,0,t[0]],[0,1,t[1]],[0,0,1]]) # translate

   result = cv2.warpPerspective(img2, Ht.dot(H), (xmax-xmin, ymax-ymin))
   result[t[1]:h1+t[1],t[0]:w1+t[0]] = img1
   return result
```

The first and the second line extracts the image shapes of the first and the second images to be stitched. *pts1 and pts2* carry the four corners of both the images. *perspectiveTransform()* function does a perspective transform on image 2 with the transformation matrix, *H*. Now, we obtain *pts2_* which depicts the extend to which *img2* would span after warping. This image would then be inserted into *img1*. Thus the total size of the image is the span distance of *pts1* and *pts2_*. Remember in chapter 4, we had to translate the negative indices after rotation to bring it all to positive? We do the same operation here except, we use translation

matrices (as discussed in section chapter 4.2). t takes the values for translation. The Ht matrix is the translation matrix. We then perform the warpPerspective operation. To the *warpPerspective()* method we pass the image to be transformed, the transformation matrix (note we multiply Ht and H) to obtain the compounded result; we can cascade matrix operations to obtain the compounded results. The warped image is then dumped into *result*. However, this resultant image carries only warped *img2*. We overlay *img1* to obtain the final stitched image. If you noticed, "stitching" is nothing but warping one image to suit the other. The output of warped image and the resultant stitched img are shown in figure 5.1.



(a) (b)

Figure 5.1: a) warped image just before stitching, b) stitched image

For the entire code visit ch5/sift.py.

Stitching is nothing but inserting one image into another. These images not necessarily be side by side. One can insert an image completely into another as shown in the figure below.

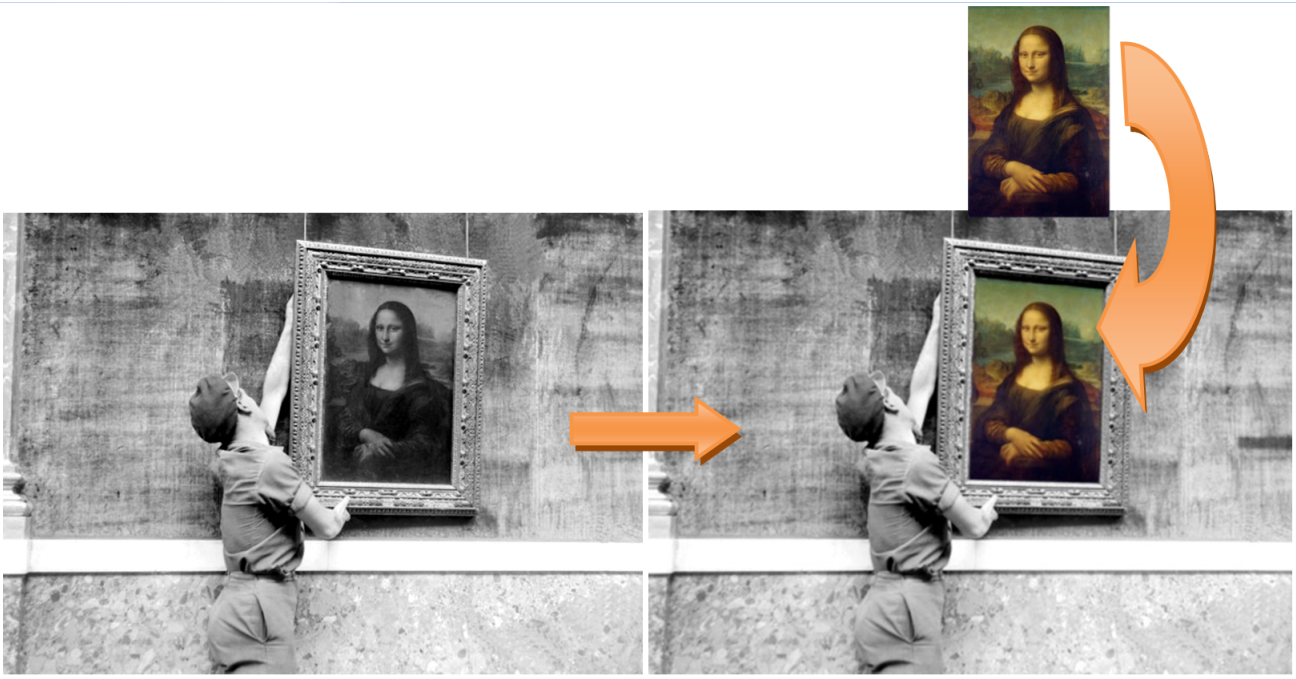


Figure 5.2: Inserting Monalisa color image into a photograph

Exercise 5.1: Attach a webcam to your computer and point it towards a *poster* on the wall (make sure that the camera does not move after the program starts running). Get a digital copy of the same poster (like the colored Monalisa image in figure 5.2) and perform the stitching operation such that, the *digital poster* is overlayed on the webcam frames exactly on the area of the *poster*. Here are the basic steps you need to follow

1. Once the camera is pointed towards the poster, grab one frame from the webcam containing the poster(*webcam frame*).

2. Next find the matching points between the *webcam frame* and the *digital poster*

3. Calculate the *Homography Matrix* between the *webcam frame* and *digital poster*

4. Grab subsequent frames from the webcam. Use the same homography matrix to warp the *digital poster* to suit the *webcam frames*

Exercise 5.2 OpenCV can read a *sample video* frame by frame. Instead of overlaying the *digital poster*, overlay the *sample video frames* on the poster.

If you have made it this far, congratulations, you have implemented a very rudimentary 2D AR system. Let's jump onto more adventurous stuff now.

Chapter 6

Realtime Marker Tracking

In the last chapter, we input the system with two images and one image was warped to suit the other image. The same can be extended to videos as in *Exercise 5.1*. But one major draw back of this is system is the camera has to be static. This is because *SIFT* takes two images and calculates the *homography* between the two images. To periceve continuous motion, about 25 frames have to be played back per second. This means, for the same scenario in *Exercise 5.1*, if the webcam is continuously moving, calculation of *homography matrix* and *warping* has to be performed under at the speed of atleast 1/25th of a second. SIFT however, is much slower and does not perform at this speed. Infact, *realtime - natural* marker detection is one of the major challenges in the field of Computer Vision. Here, we restrict ourselves to detection of *artificial markers*. *Artificial markers* are special patterns that a computer can easily detect thus reducing the time and computational requirement for marker detection by large orders of magnitude. In this chapter, we are going to create an artificial marker using SIFT and Lucas Kanade Optical Flow Algorithm.

6.1 Lucas Kanade Optical Flow

We will be focussing on the Optical Flow algorithm in this section. Optical Flow algorithm assumes that between subsequent frames of a video there is very little change. Say we have a video with frames being recorded at 30 frames per second, and we take a frame, f_t and the subsequent frame, $f_{t+\Delta t}$ will have very little change with respect to the previous frame, where Δt is the time between successive frames. This is especially true if the contents of the video are slow moving. This algorithm assumes,

$$f_t(x, y, t) = f_{t+\Delta t}(x + \Delta x, y + \Delta y, t + \Delta t)$$

i.e., the brightness of a pixel (x, y) of frame, f_t and the brightness of the displaced pixel, $(x + \Delta x, y + \Delta y, t + \Delta t)$ in the next frame, $f_{t+\Delta t}$ can be assumed to be constant. This is called *brightness constancy assumption*. With this assumption, the optical flow algorithm finds the new location of a pixel given the previous frame's pixel. This means we will be able to define some points on the image and track those points as long as the video is slow moving. Between each pair of frames we calculate the homography matrix knowing the matching points. This can then be used to overlay another image of our choosing. One might ask, what points should we track? It is usually easier for the computer to track points on an image with a high amount of detail. Images with a lot of intensity variations and striations (high frequency) work really well. OpenCV has a method called *goodFeaturesToTrack()* which outputs a list of points that are of high frequency. However this marker is far from perfect when compared to markers like "aruco" or "chilitags". But this exercise is still worth going through.

First obtain the hardcopy of a painting or a picture with a lot of sharp edges, lines and cuts. An image like in figure 6.1 will work well.

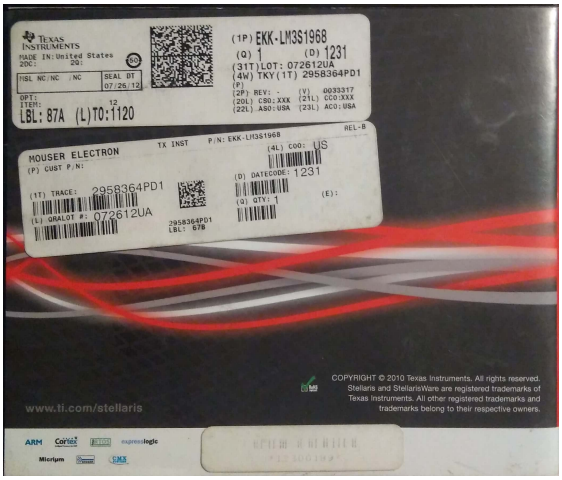


Figure 6.1

Lets code this up now in python. The first two lines does some imports. The next two lines sets some parameters for the algorithm.

```
import numpy as np
import cv2

# params for ShiTomasi corner detection
feature_params = dict( maxCorners = 100, qualityLevel = 0.3, minDistance = 7,
    blockSize = 7 )

# parameters for lucas kanade optical flow
lk_params = dict( winSize = (15,15), maxLevel = 2, criteria = (cv2.TERM_CRITERIA_EPS
    | cv2.TERM_CRITERIA_COUNT, 10, 0.03))
```

```
if __name__ == "__main__":
    cap = cv2.VideoCapture(0)
    # Create some random colors
    color = np.random.randint(0,255,(100,3))

    # Take first frame and find corners in it
    ret, old_frame = cap.read()
    old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
    p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

    # Create a mask image for drawing purposes
    mask = np.zeros_like(old_frame)
```

CV2.VideoCapture(device_number) set creates a camera device. *device_number* points to the camera device we want to use to capture. If you just have one camera connected to your laptop, its usually '0'. The second line describes the device from which we have to capture. If its the default webcam its 0 and any other camera device, its 1, 2, 3 etc. One can find all the list of camera (also other) devices attached to the computer by typing the following on the terminal.

```
ls /dev | grep video
```

To view if the camera is working or connected properly we can test it out using a small software called cheese. To install and run cheese, type

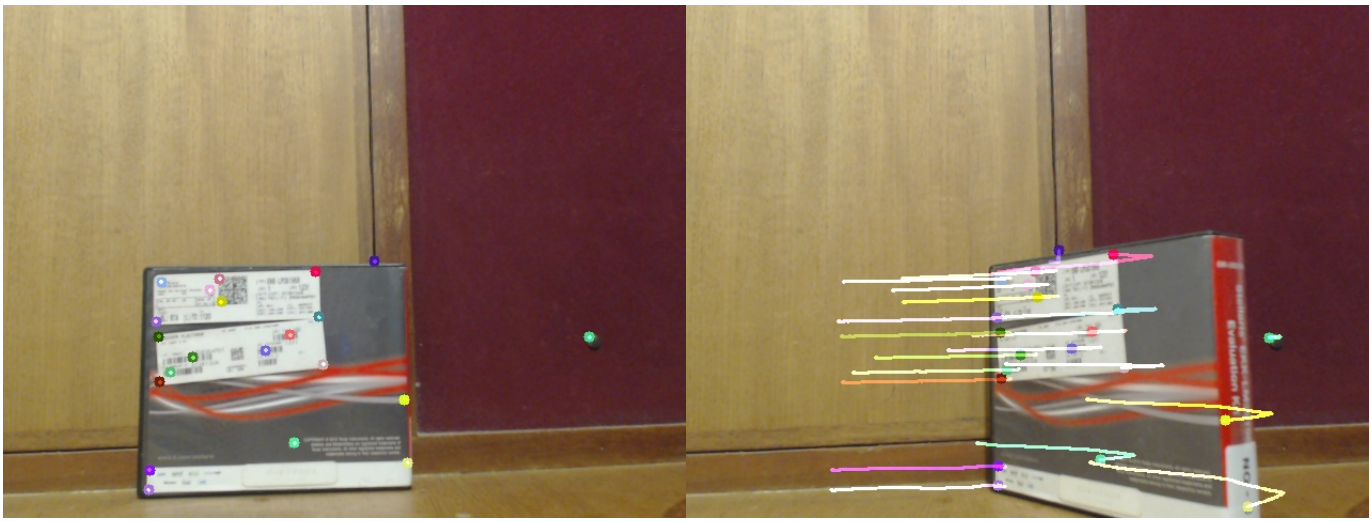
```
sudo apt-get install cheese
cheese /dev/video0
```

Now, back from the small digression; we are going to track some points. Each point can be indicated by a random color so that its distinctly visible. The third line reads from the camera device. The captured images are then converted to greyscale. *goodFeaturesToTrack* detects points that are of high frequency or gradient ensuring they are easier to track. *p0* are the points that are going to be tracked.

```
while(1):
    ret,frame = cap.read()
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # calculate optical flow
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)

    # Select good points
    good_new = p1[st==1]
    good_old = p0[st==1]

    # draw the tracks
    for i,(new,old) in enumerate(zip(good_new,good_old)):
        a,b = new.ravel()
        c,d = old.ravel()
        cv2.line(mask, (a,b), (c,d), color[i].tolist(), 2)
        cv2.circle(frame, (a,b), 5, color[i].tolist(), -1)

    img = cv2.add(frame,mask)

    cv2.imshow('frame', img)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

    # Now update the previous frame and previous points
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1, 1, 2)

cv2.destroyAllWindows()
cap.release()
```

Once we enter the while loop, the frames are read continuously and converted to greyscale. The *calcOpticalFlowPyrLK()* calculates the optical flow and gives us the new points *p1*. The *good_new* and the *good_old* points are the refined points that are tracked. The tracks are then drawn and added to the

image to be displayed. The rest of the code is self explanatory. Point the camera at the painting/picture you made/downloaded. Move the picture slowly. Watch the tracked points. Figure 6.2 shows the flow path of some points on an image.



(a) (b)

Figure 6.2: Lucas Kanade Optical Flow of some points on a box

6.2 Realtime Slow Motion Marker

In this section, we are going to build our very own realtime slow moving marker. We will be using sift and the optical flow algorithm to build this. The basic algorithm can be divided into two parts as depicted in figures 6.3 and 6.4. Let us look at the first flowchart (figure 6.3) where we start with obtaining a scanned copy of a picture. Let us call it the *Marker_ image*. Then, we need to obtain the first frame from the webcam pointed at the hardcopy of the marker image; let us call this f_1. We then perform SIFT and feature matching between *marker_ image* and f_1. Finally, we estimate the homography matrix, $H_{SIFT-LK}$ between the two images. We then use the optical flow algorithm on all the subsequent webcam frames, f_1, f_2...., f_k. The same keypoints obtained while computing SIFT are tracked by the LK-Optical Flow algorithm to obtain homographies - H_{LK1}, H_{LK2}, H_{LK3} ..... H_{LK4}. The second flowchart in figure 6.4 describes how to calculate the homography between the marker image and a frame-k. H_k is obtained by compounding all the homographies, $H_k = H_{SIFT-LK} * H_{LK1} * .....H_{LK(k-1)}$. For the sake of illustration, we are going to draw the boundaries around the tracked image where the marker image can be overlaid.

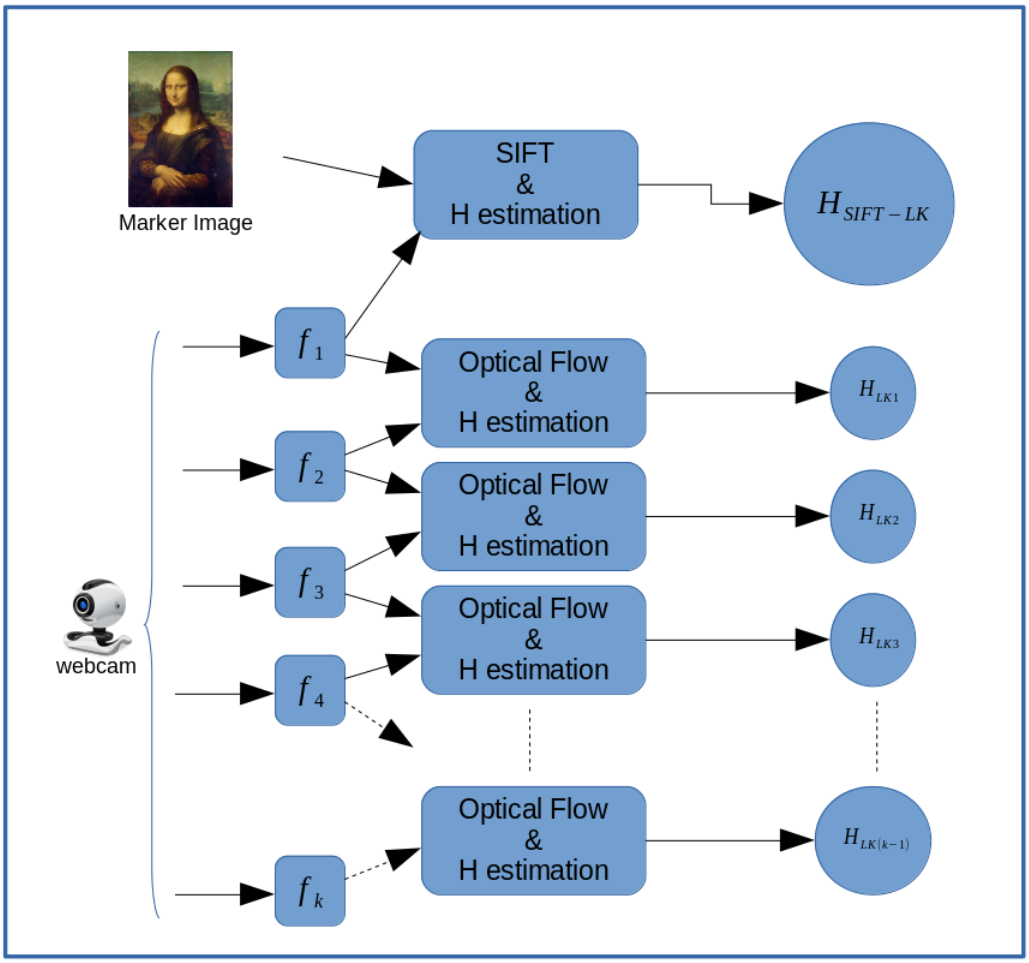


Figure 6.3: Estimating Homographies

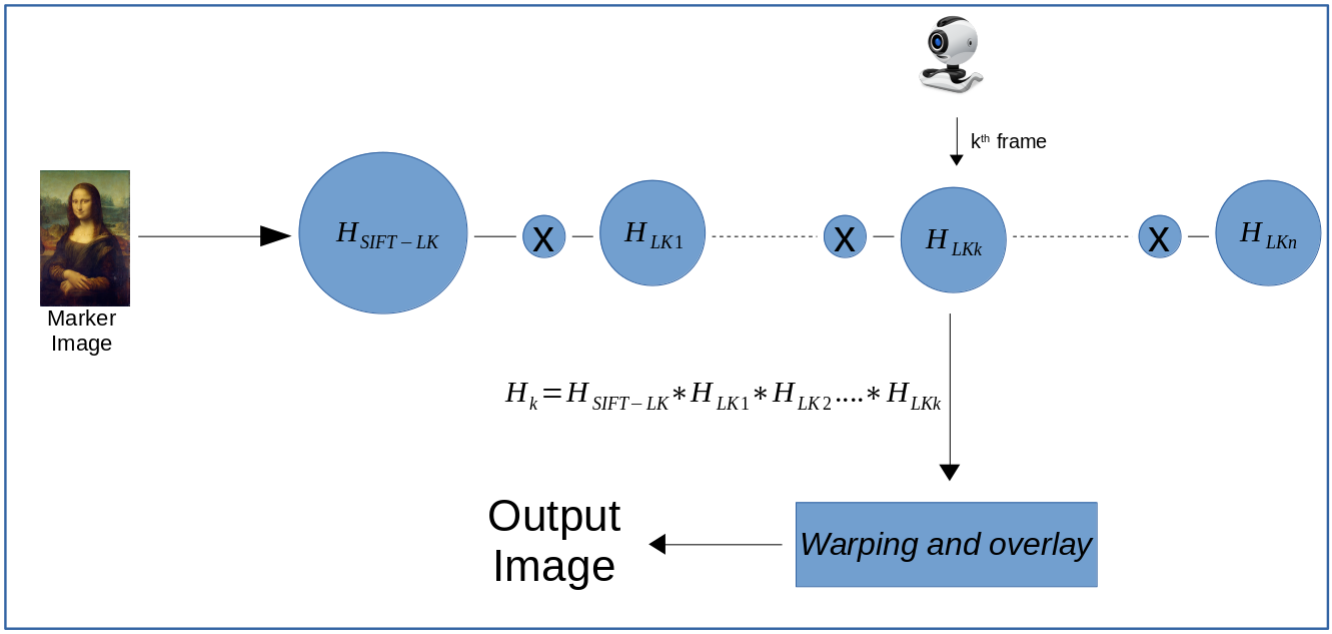


Figure 6.4: Computing compunded H for a sample frame k

Let us code this up. Create a file named realtime_marker_tracking.py.

```
import cv2
import numpy as np
import sys
import time

if __name__ == "__main__":
    img_path = sys.argv
    marker_img = cv2.imread(img_path[1])
    gray_marker_img = cv2.cvtColor(marker_img, cv2.COLOR_BGR2GRAY)

    print "keep the marker picture still in front of the camera for a short while"
    time.sleep(5)

    # capture from the webcam the first frame to compute the homography using SIFT. This
        will take some time
    cap = cv2.VideoCapture(0)
    k=0
    while k<80:
        flag, first = cap.read()
        k+=1

    gray_first = cv2.cvtColor(first, cv2.COLOR_BGR2GRAY)
    print "first frame from the webcam captured"

    # lets compute sift and perform feature matching between both the images
    print "computing sift between both the images"
    sift = cv2.SIFT()
    kp_marker, des_marker = sift.detectAndCompute(gray_marker_img, None)
    kp_first, des_first = sift.detectAndCompute(gray_first, None)
    print "sift computed"

    print "now matching"
    FLANN_INDEX_KDTREE = 0
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks = 50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des_marker, des_first, k=2)
    good = []
    for m, n in matches:
        if m.distance < 0.7*n.distance:
            good.append(m)

    # calculating homography
    if len(good) > 10:
        print "calculating homography"
        src_pts = np.float32([kp_marker[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp_first[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
        h, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

    else:
        print "could not find a matching marker"
```

```
    exit()
```

The above code has already been discussed in the previous chapters; it just deals with computing SIFT between the *marker_ image* and *first* frame of the webcam and estimating $H_{SIFT-LK}$.

```
# These are the points to be tracked to draw the frame around the tracked marker.
    Notice we are defining them for the marker image
points_on_marker_image = np.array([[0, 0, 1], [marker_img.shape[0], 0, 1],
    [marker_img.shape[0], marker_img.shape[1], 1], [0, marker_img.shape[1], 1]])

# p1 is the array of points to be tracked on the first image
p1 = dst_pts

while True:
    _, subsequent = cap.read() # marker and first image got computed earlier on, now
        the subsequent images
    gray_subsequent = cv2.cvtColor(subsequent, cv2.COLOR_BGR2GRAY)

    # tracking between first and subsequent frames using LK optical flow
    ps, st, err = cv2.calcOpticalFlowPyrLK(gray_first, gray_subsequent, p1, None,
        **lk_params)

    # lets compute homography between two images
    h_new, mask = cv2.findHomography(p1, ps, cv2.RANSAC, 5.0)
    h = np.dot(h, h_new) # compounded homography between images

    # lets track the frame over the subsequent points by multiplying the tracked
        marker with the compunded homography matrix
    points_on_webcam_image = np.dot(h, points_on_marker_image.T).T

    # to draw we need to reshape the points on the webcam image shape
    points_on_webcam_image_reshaped =
        np.array([[[points_on_webcam_image[0][0]/points_on_webcam_image[0][2],
        points_on_webcam_image[0][1]/points_on_webcam_image[0][2]]],
        [[points_on_webcam_image[1][0]/points_on_webcam_image[1][2],
        points_on_webcam_image[1][1]/points_on_webcam_image[1][2]]],
        [[points_on_webcam_image[2][0]/points_on_webcam_image[2][2],
        points_on_webcam_image[2][1]/points_on_webcam_image[2][2]]],
        [[points_on_webcam_image[3][0]/points_on_webcam_image[3][2],
        points_on_webcam_image[3][1]/points_on_webcam_image[3][2]]]]) # de-homogenized
        and reshaped

    # the display function in display.py draws the borders around the images
    subsequent = display(points_on_webcam_image_reshaped, subsequent)

    cv2.imshow("", subsequent)
    key = cv2.waitKey(20)
    if(key == 27):
        break
    if(key == 115):
```

```
            cv2.imwrite("tracked_image.jpg", subsequent)
        p1 = ps
        gray_first = gray_subsequent

cv2.destroyAllWindows()
cap.release()
```

In the above snippet of code, the first line creates the four points around the frame of the marker image. These are the points which are to be drawn around the tracked frames. The second line assigns $p1$ variable to be tracked by the LK-Optical flow algorithm. Inside the while loop, the first two lines read the image from webcam and turn it into grayscale images. The calcOpticalFlowPyrLK methods calculates the optical flow for the points p1 and outputs the tracked coordinates into ps. Then we calculate the homography between the two subsequent frames. h compounds the values of homography from the start. Then we calculate the new frame border coordinates by multiplying with the compunded h. Finally, we draw the points and display it. A sample image is output in figure 6.5. To find the code for the entire program, goto ch6 folder, inside realtime_marker_tracking/realtime_marker_tracking.py. A short video of the output can be found here - link

Figure 6.5: Marker tracking with SIFT and Lucas Kanade Optical Flow algorithm

6.2.1 Shortcomings of the above tracker

- Tracking is very slow and innacurate.
- Estimating homography involves tracking points. Lucas Kanade outputs a lot of errors which gets compunded as we calculate h.
- Once tracking is lost, the program has to be reset to make it work again. This is a pain.
- Overall, its totally not dependable.

Like I said before, this is a very rudimentary marker detection algorithm and it cannot be used for practical purposes. We will be looking into more robust algorithms in the next section.

6.3 Robust Fiducial Marker Detection

First we look at OpenCV's inbuilt chessboard corner detection. Though, this is strictly not built with realtime video tracking in mind, we will still explore this as we will be using it later for *camera calibration*. After that, we will be downloading and installing a robust marker detection library called *Aruco*.

6.3.1 OpenCV's Chessboard Detection

Go ahead and download the image of a chessboard and take a print out of it. After that create a file called *chessboard_ detection.py* in your work folder. Then insert the following code in the file

```
import numpy as np
import cv2

criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# this describes the shape of the chessboard corners we are tracking - outer corners are
    discarded
cols = 7
rows = 6

if __name__ == "__main__":
    cap = cv2.VideoCapture(0)
    count = 0
    while(count < 1000): # Here, 10 can be changed to whatever number you like to choose
        flag, frame = cap.read() # Capture frame-by-frame
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Find the chess board corners
        flag, corners = cv2.findChessboardCorners(gray_frame, (cols, rows), None) # the
            variable "corners" contains the pixel information of the tracked chessboard
            corners
        if flag == True:
            print count, ") track successful"

        # painting corners on the frame
        cv2.drawChessboardCorners(frame, (cols,rows), corners, flag)
        count += 1

        # display
        cv2.imshow("tracked chessboard", frame)
        key = cv2.waitKey(10)
        if(key==27):
            break

    # release the used resources
    cap.release()
    cv2.destroyAllWindows()
```

The first two lines of code does the usual imports. The third line describes termination conditions. Then we describe the shape of the chessboard image - the number of corners in the chessboard excluding

the outer squares. Inside main, we create a video device. And inside the while loop, we read the frames from the webcam, convert them to grey. Then we use *findChessboardCorners()* function to detect the corners on the captured frame. There is an inbuilt draw function called *drawChessboardCorners()* which draws the chessboard corners on the frame. This is then displayed. *count* variable simply counts the number of successful detections. Image below shows a frame of a successful detection. For video of the same, click this link.

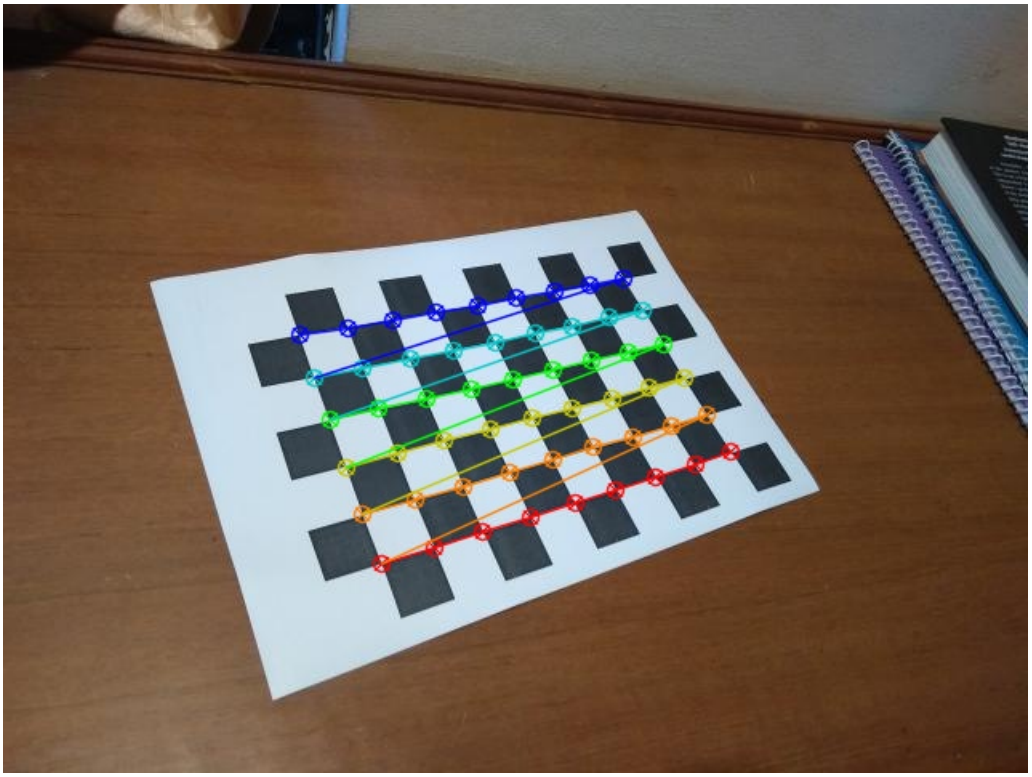


Figure 6.6: Corners detected on a chessboard

As you see the detection is way more robust. One disadvantage is its sensitive to occulsion as shown in the video.

6.3.2 Aruco Marker Detection

In this section, we are going to observe a more robust library called ARUCO, built by folks at University of Cordoba, Spain. The paper is titled "Automatic generation and detection of highly reliable fiducial markers under occlusion". Since we are using OpenCV 2.4.x version, we have to download and integrate the library ourselves. You can find a copy of aruco in the folder ch6/aruco_marker_detection/aruco-1.2.4.tar.gz and hit the "Download" link. Now type the following in the terminal.

```
cd /<path to aruco/>
tar -xvf aruco-1.2.4.tar.gz
cd aruco-1.2.4
mkdir build && cd build
cmake ..
make
sudo make install
```

If it compiles successfully then you have installed Aruco. Now lets start working with Aruco. In your

work directory, create a folder named *aruco_marker_detection*. Since, Aruco is a marker detection library, we have to create an Aruco marker. To do that type following commands in your terminal.

```
cd <path-to-aruco>/build/utils
./aruco_create_board 5:2 aruco_board.png board_config.yml
cp aruco_board.png <path-to-aruco_marker_detection-dir>
cp board_config.yml <path-to-aruco_marker_detection-dir>
```

For this part of the code we will be using C++ as this version of aruco does not have python bindings-AFAIK. Create a folder called aruco in your work directory. Create a file named *marker_detection.cpp* inside the aruco folder. Also create a file named *Makefile* in the *aruco_marker_detection* directory. Insert the following contents in your *Makefile*.

```
CC = g++

INCLUDE = `pkg-config --cflags opencv` -I/usr/local/include

LIBS = `pkg-config --libs opencv` -laruco

# enter filename without extension
FILENAME = marker_detection

all: $(FILENAME).cpp
   @$(CC) -g $(INCLUDE) $(FILENAME).cpp -o $(FILENAME) $(LIBS)

run:
   @./run.sh
```

INCLUDE and *LIBS* fields point to the respective libraries viz., OpenCV and ARUCO. Our "Makefile" is ready for compilation. We just need one more thing before we start writing the code.

To run aruco, we need to create a *camera_calibration.yml* file that describes the camera parameters. Don't worry about what it means now. We will be studying camera parameters in detail in the next chapter. On a gross level, camera parameters describe the properties of the camera like the focal length, distortions etc. To obtain the camera parameters, we will be using our chessboard. Type the following commands on the terminal,

```
cd <opencv-dir/build/bin>
./cpp-example-calibration -w 6 -h 7 -s 0.025
mv out_camera_data.yml <path-to-aruco_marker_detection-dir>
cd <path-to-aruco_marker_detection-dir> && mv out_camera_data.yml cameraConfig.yml
```

It will pop up a window which says press 'g to start'. The program captures 10 images and obtains the calibrations for the camera. The '-w' and '-h' fields in the above command indicates the number of inner corners on the chessboard, excluding the outer squares. '-s 0.025' describes the size of each square in meters measured with a ruler on the actual board. Mine happened to be 2.5 cms, measure yours and insert the figure correctly. The next line "mv out_camera_data.yml <path-to-aruco_marker_detection-dir>" moves the obtained calibration file to the *aruco_marker_detection* directory. The next command cd's into the directory and rename the file to *cameraConfig.yml*.

Here is the **list of files** you need in the *aruco_marker_detection* directory.

1. **marker_detection.cpp**
2. **Makefile**
3. **cameraConfig.yml**
4. **board_config.yml**
5. **aruco_board.png**

Now we are ready to insert our code into *marker_detection.cpp*. Following lines describe the code,

```
// all the imports
#include <iostream>
#include <fstream>
#include <sstream>
#include <opencv2/opencv.hpp>
#include <aruco/aruco.h> // include the aruco import
#include <strings.h>
#include <vector>

int main(int argc, char **argv)
{
   // some variables we will be using later
   int cam_device_no, size;
   double ThreshParam1 = 10, ThreshParam2 = 10;
   float probDetect, markersize;


   cv::Mat frame; // Mat is the data structure that holds an image in opencv-cpp
   aruco::BoardConfiguration boardconfig;
   aruco::CameraParameters camparams;
   aruco::BoardDetector boarddetect;

   if(argc < 3)
   {
      std::cerr << "Not enough arguments.\nUsage is : ./binary videodevice_number
          BoardConfig.yml CameraConfig.yml [size]" << std::endl;
      return 0;
   }

   // all the settings that aruco's datastructures/constructors need -------------
   cam_device_no = atoi(argv[1]);
   cv::VideoCapture cap(cam_device_no);
   boardconfig.readFromFile(argv[2]);
   cap >> frame;
   camparams.readFromXMLFile(argv[3]);
   camparams.resize(frame.size());
   markersize = atoi(argv[4]);
   boarddetect.setParams(boardconfig, camparams, markersize);
   boarddetect.getMarkerDetector().getThresholdParams(ThreshParam1, ThreshParam2);
```

```
// --------------------------------------------------------------------
```

After the imports, inside *main()* the first three lines of code defines some variables we will be using later. Just like the *numpy.array* data structure in python, the equivalent data structure is *cv::Mat* in C++. The next few lines also initialize some variables and initializes some constructors. The meat of the code is in the loop.

```
while(1)
{
   cap >> frame;
   probDetect = boarddetect.detect(frame);  // gives the strength of detection
   size = boarddetect.getDetectedMarkers().size(); // amongst the 10 markers on the
       board, only some might be detected at a particular time

   for (int i=0;i<size;i++)
      boarddetect.getDetectedMarkers()[i].draw(frame, cv::Scalar(0, 0, 255), 1); // For
          each detection draw the boundaries of the marker and the marker id

   cv::imshow("", frame);
   int key = cv::waitKey(10);
   if(key==27)
      break;
   else if(key==115)
      cv::imwrite("marker_with_data.jpg", frame);
}
```

The first line in the loop reads frames from the webcam. The next line detects the frames. The marker board consists of 10 individual markers. There is a good chance that not all markers are detected due to various reasons. 'size' variable gives the number of markers detected. The 'getDetectedMarkers().draw()' method draws a border around the detected marker. It also prints the marker id at the center. The rest of the code is simply the display and exit methods.

To run this code, several command line parameters have to be specified.

```
./marker_detection 0 board_config.yml cameraConfig.yml 0.025
```

Again 0.025 is the size of the aruco marker square on the board. In addition to the five in the **list of files** in *aruco_marker_detection* directory, we can also add a shell script named *run.sh* with the above terminal command for easy execution of the program. Hence to execute, simply type

```
make && ./run.sh
```

The complete source code can be found in ch6/aruco_marker_detection/code/marker_detection.cpp. Below figure shows the image of a detected marker.

Click the link to see the video of the same.

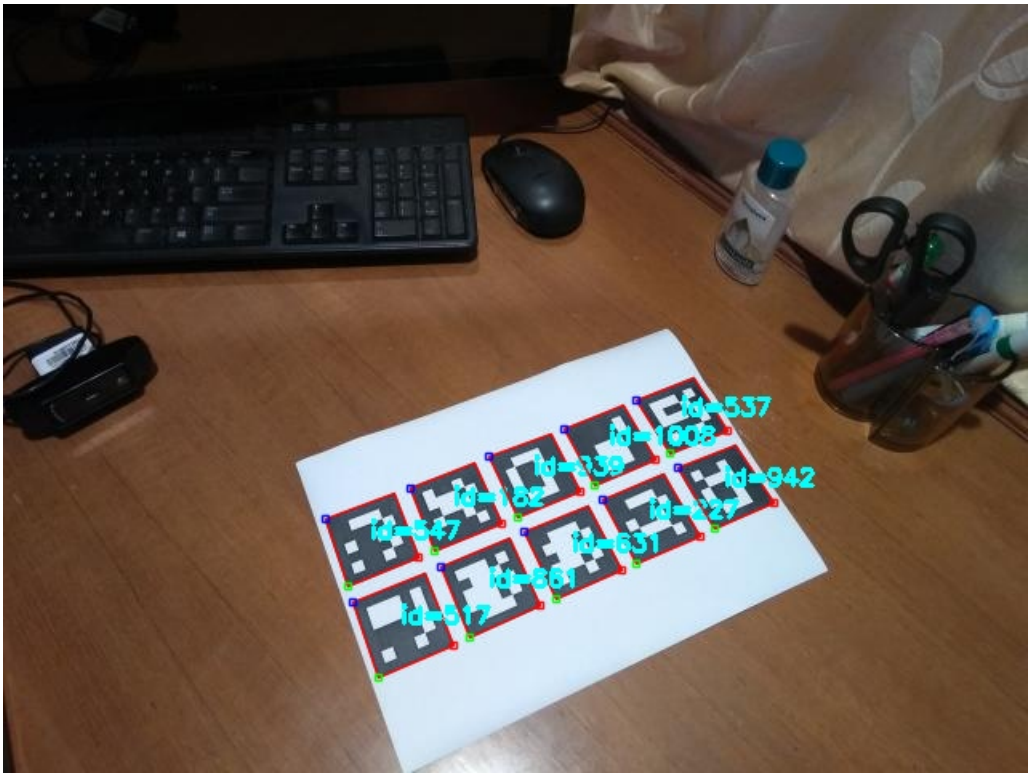


Figure 6.7: Corner detection with ARUCO

6.4 Exercises

1) With chessboard corners as the detected marker points, overlay a video on the marker.
2) We saw planar markers, a huge disadvantage of these are once, the camera that is pointing to the marker gets into an awkward angle they won't be able to identify the patterns. To solve this problem, we can go ahead and create a 3D marker as shown in figure 6.8. Go ahead and create your own 3D marker. Clue: *board_ config.yml* contains all the definitions for the layout of the board. Edit that file to construct your own 3D marker.

Figure 6.8: 3D marker

Chapter 7

Camera Calibration

In this chapter, we are going to study about camera calibration. ***What is camera calibration?***. It is the process of estimating camera parameters like focal length, center of an image, distortions etc. Focal length determines the zoom and the camera center determines the offset between the lens center and the image center. Camera lens and sensor can produce several aberrations in an image, for instance fish-eye/barrel effect or pin-cushion effect etc. These can be modeled into several distortion parameters. A good camera produces negligible distortions.

This section also covers the position and orientation of the camera with respect to the physical world around the camera. This physical world with all the contents around is known as the ***world space***. The previous statement automatically implies that we have a coordinate system representing the physical world around us. We will discuss that shortly.

7.1 Intrinsic Matrix

This mathematical construct deals with only the camera parameters and nothing outside it. Any simple camera (like a webcam) can be assumed to work much like a pin hole camera. Instead of the hole, a lens converges the light onto the image plane (camera sensor). Figure 7.1 depicts the relation.

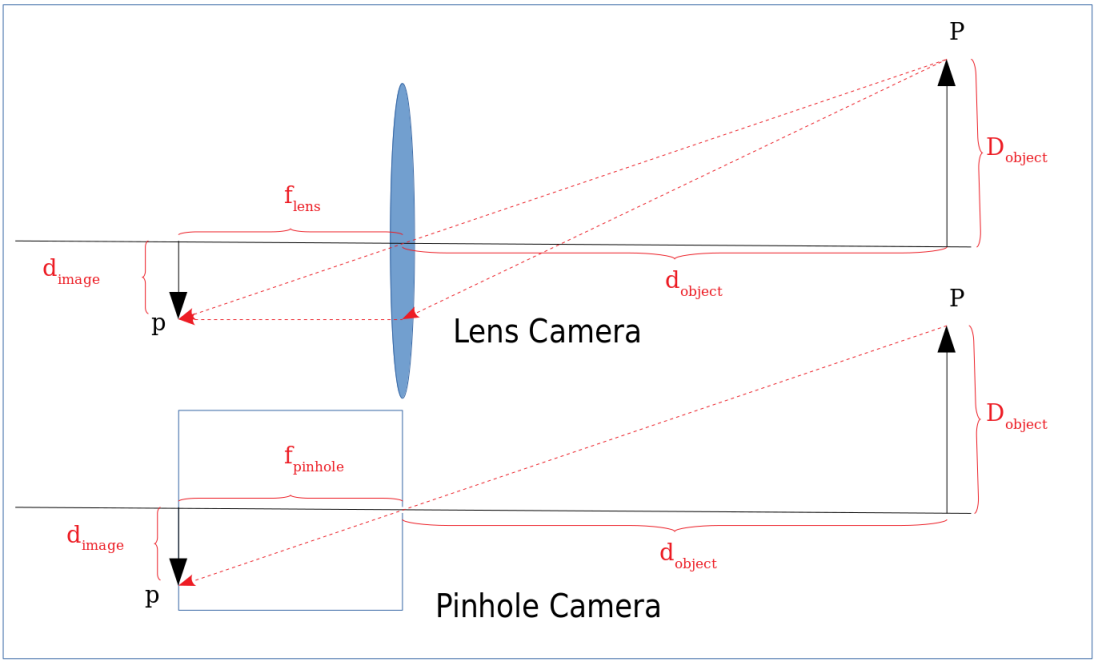


Figure 7.1: Similarity between lens and pinhole camera

In figure 7.2, given a point in the physical world, the intrinsic matrix describes where that point gets projected on the image plane. Point, P in world space gets projected to point, p on the image plane. Focal length of a lens is defined as the distance between the lens center and the image plane. In the y-z plane, we calculate this distance as f_y and in the x-z plane, the focal length is calculated as f_x. An ideal camera has $f_x = f_y$. However, there will be small variations in this. Let us calculate, f_y

$$By\ similar\ triangles,\ \frac{f_y(pixels)}{d_y(pixels)} = \frac{d(cms)}{D_y(cms)}$$

$$f_y(pixels) = \frac{d(cms) * d_y(pixels)}{D_y(cms)} \tag{7.1}$$

$$Similarly,\ f_x(pixels) = \frac{d(cms) * d_x(pixels)}{D_x(cms)} \tag{7.2}$$

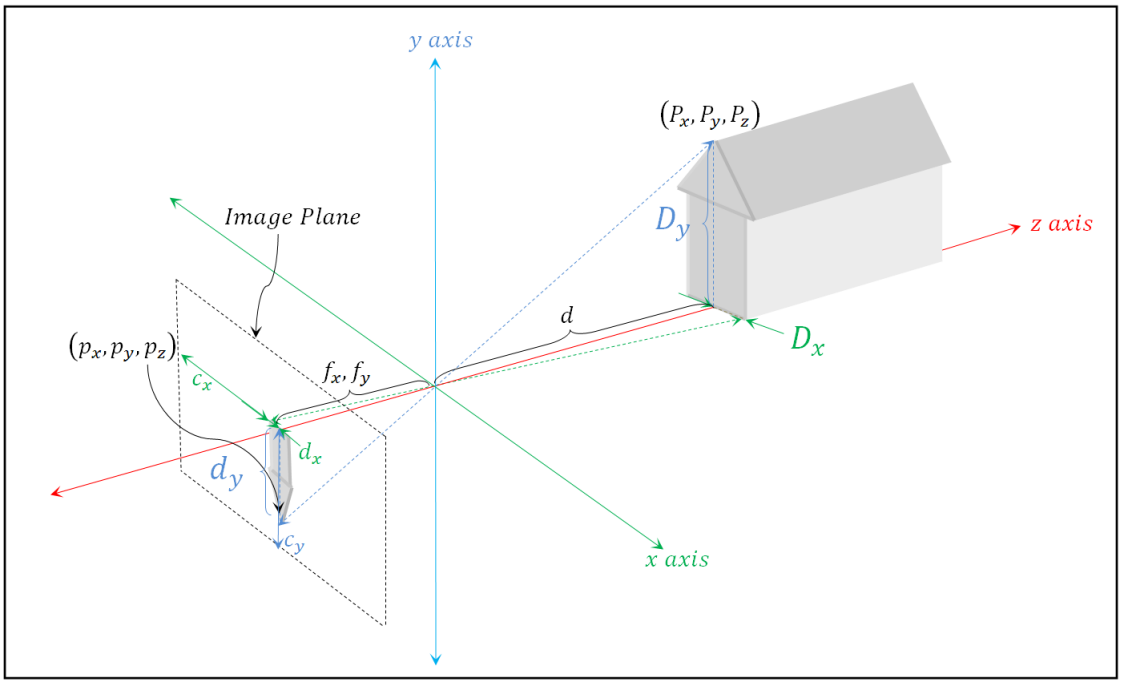


Figure 7.2: Derivation of Intrinsic Matrix

There are a couple of other parameters that we need to construct the intrinsic matrix. (c_x, c_y) defines the image center. This is just $(width/2, height/2)$ in pixels, of the image obtained on the computer. There can be also another parameter called skewness, s_k, which represents pixel shape distortions. A supposed to be square pixel can be a rhombus. A complete intrinsic matrix is constructed as

$$Intrinsic\ Matrix\ = \begin{bmatrix} f_x & s_k & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{7.3}$$

Most modern cameras have very negligible values of skew and hence it can be neglected; $s_k = 0$

In this section, we discuss two methods 1) manual and 2) Using OpenCV's chessboard calibration method to arrive at the *intrinsic matrix*.

7.1.1 Manual Calibration

For this method, we require a rectangular box/book or some other object with a definite rectangular shape. I will use the book shown in figure 7.3. Now position the object in front of the camera such that you get a clear shot of it roughly occupying the image center. A snapshot of the image can be obtained using cheese (we installed and used cheese in section 6.1). Measure the distance between the camera and book in *centimeters* using a ruler/measuring-tape. Measure the height and the width of the book in *centimeters*. From equation 7.1 and 7.2 we have, d, D_y and D_x.

Figure 7.3: Calculation of d, D_x and D_y

We now need to calculate d_x and d_y in pixels. There is a software called gimp, which is an opensource equivalent of photoshop. This software gives us the pixel information of the image under the cursor. To install gimp, type the following in your terminal.

```
sudo apt-get install gimp
gimp <path-to-image>
```

Find the pixel coordinates of all the corners of the book on the image. Find the span of the book in x and y axes as shown in figure 7.4.

Figure 7.4: Calculation of d_x and d_y

This gives us d_x and d_y. Now we can calculate f_x and f_y using equations 7.1 and 7.2. c_x and c_y are simply $image\_width/2$ and $image\_height/2$. These values can be substituted in equation 7.3 to obtain the intrinsic matrix.

This method of calibration is not very accurate though as we obtain pixel information from the image by hovering the mouse above the image. And quite often a low quality camera (in addition to lossy compression) yeilds blurry corners, thus finding corners of the object in the image can be imprecise.

7.1.2 Chessboard Camera Calibration with OpenCV

Next, we calculate the intrinsic matrix using opencv's chessboard method. We already did that using OpenCV's binaries in section 6.3.2. However, since the implementation is a very small extension of the code in section 6.3.1, and it is far more accurate, we will code it. Create a file named *intrinsic_camera_calibration.py* in your work directory. Copy the code in section 6.3.1. The *cv2.calibrateCamera()* function takes in *object_points* and *image_points*. *object_points* are 3D points corresponding to *image_points*. In the physical world, we can define one corner of the chessboard as the origin as shown in figure 7.5. Any point on the x-y plane (indicated by a semi-transparent blue rectangle) has z value = 0. Thus, for each *image_point* detected, we can associate a unique *object_point*.

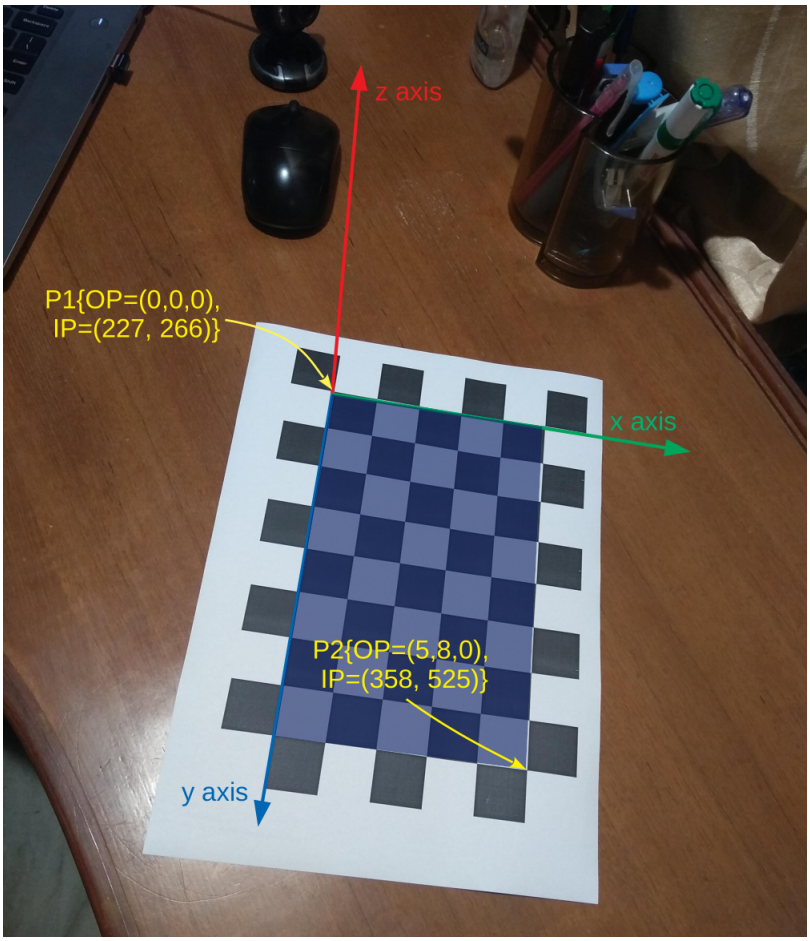


Figure 7.5: Image Points and Object Points for P1 and P2

```
import numpy as np
import cv2

criteria = ( cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT, 30, 0.1 )

if __name__ == "__main__":
    cap = cv2.VideoCapture(0)

    pattern_size = (9, 6) # number of corners in the chessboard

    # we need to describe some points that we are going to track
    # we need something like [[0, 0, 0], [1, 0, 0] . . . . [cols-1, 0, 0], [0, 1, 0], [1,
       1, 0] . . . . [cols-1, rows-1, 0]]
    # we compile each col separately and stack them together
    col0 = np.tile(np.arange(0, pattern_size[0]), pattern_size[1])
    col1 = np.reshape((np.ones((pattern_size[0], pattern_size[1]))*
       np.arange(pattern_size[1])).T, (1, pattern_size[0]*pattern_size[1] ))[0]
    col2 = np.zeros(pattern_size[1]*pattern_size[0])
    pattern_points = np.float32(np.vstack([col0, col1,
       col2]).T.reshape(pattern_size[1]*pattern_size[0], 3)) # a small addendum,
       converted float64 into float32
```

```
obj_points = []
img_points = []
flag, frame = cap.read()
h, w = frame.shape[:2]

count = 0
```

Inside main, we have to create some points, *pattern_points* for the chessboard to track. The first column of the points should be $[0, 1, 2....(cols-1), 0, 1, ...(cols-1).....repeated\ (rows-1)\ times]$. The second column is of the form $[0, 0.....(col-1)times.....1, 1.....(col-1)times.....repeated\ (rows-1)\ times]$. Finally col3 $[0, 0.....repeated\ rows * cols\ times]$. Then we append these cols together to obtain *pattern_points*. *pattern_points* from several images will be appended to obtain *obj_points* for greater accuracy.

```
while(count<30): # we are using 30 images to calculate the intrinsic matrix. This will
    yeild higher accuracy
    flag, frame = cap.read()
    found, corners = cv2.findChessboardCorners(frame, pattern_size)

    if found:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        cv2.cornerSubPix(gray, corners, (5, 5), (-1, -1), criteria) # this yeilds
            corners with subpixel accuracy
        cv2.drawChessboardCorners(frame, pattern_size, corners, found)

        img_points.append(corners.reshape(-1, 2)) # for each of the 30 images, we
            append the list of image points
        obj_points.append(pattern_points) # likewise we append the object points for
            the 30 images
        count=count+1
```

OpenCV has a builtin function *cornerSubPix()*, which yeilds sub-pixel tracking accuracy. At each step the *pattern_points* and *corners* are appended over 30 images to the array *obj_points* and *img_points* respectively to obtain greater accuracy. Finally, the *calibrateCamera()* function calibrates the *img_points* and its corresponding *obj_points* to obtain the ***intrinsic_matrix*** and *dist_coefs*. Distortion coefficients are use to correct radial aberrations.

```
flag, intrinsic_matrix, dist_coefs, rvecs, tvecs = cv2.calibrateCamera(obj_points,
    img_points, (w, h))
```

For the complete code visit ch7/intrinsics/intrinsic_matrix_calibration.py on github.

7.2 Extrinsic Matrix

In this section, we are going to create the extrinsic matrix. Remember, we defined an origin and three axes on the chessboard in section 7.1. The extrinsic matrix describes the position and orientation of the

camera in the physical space. The extrinsic matrix takes the form of a 3x3 rotation matrix, R augmented by a 3x1 translation matrix, t. Optionally, some folks append a $[0, 0, 0, 1]$ at the bottom to make a square matrix. However, we will leave it as a 3x4 matrix.

$$[R|t] = \left(\begin{array}{ccc|c} r_{0,0} & r_{0,1} & r_{0,2} & t_0 \\ r_{1,0} & r_{1,1} & r_{2,2} & t_1 \\ r_{2,0} & r_{2,1} & r_{2,2} & t_2 \end{array}\right) \tag{7.4}$$

A complete camera matrix is of the form,

$$P = K[R|t] \tag{7.5}$$

Here K refers to the intrinsic matrix. Remember, in section 4.1 and 4.2 we discussed several rotation and transaltion matrices. With corners detected on a chessboard, we are going to derive the $[R|t]$ matrix.

$$image\_points = K[R|t] * object\_points$$

In the above equation, we define *object_points* as discussed in section 7.1.2 and *image_points* are nothing but the detected corners of the chessboard. We already obtained intrinsic matrix, K in the previous section. The only component we do not know is $[R|t]$. This can be calculated using DLT (similar to what was discussed in section 4.3). But, DLT is a very naive algorithm and there are several optimizations such as *SolvePnP*, *Posit* etc. Here, we will be using a method called *solvePnPRansac()* which outputs *tvecs* and *rvecs* which are translation and rotation values in quaternions. Quaternions are a different form of representation of rotations which avoid the problem of gimbal lock. However, we need to obtain $[R|t]$ as matrices, we will use the *Rodrigues()* function to convert the quaternion values into a matrix. In your work directory, create a file called *extrinsics.py*.

```
import numpy as np
import cv2
import yaml

criteria = ( cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT, 30, 0.1 )

if __name__ == "__main__":
    cap = cv2.VideoCapture(0)

    pattern_size = (9, 6)

    col0 = np.tile(np.arange(0, pattern_size[0]), pattern_size[1])
    col1 = np.reshape((np.ones((pattern_size[0], pattern_size[1]))*
        np.arange(pattern_size[1])).T, (1, pattern_size[0]*pattern_size[1] ))[0]
    col2 = np.zeros(pattern_size[1]*pattern_size[0])
    object_points = np.float32(np.vstack([col0, col1,
        col2]).T.reshape(pattern_size[1]*pattern_size[0], 3)) # a small addendum,
        converted float64 into float32

    stream = open("./cameraConfig.yml", "r") # read the intrinsic matrix from the
        camera_parameters.yml file
    stream_info = yaml.load(stream)
    intrinsic_matrix = np.array(stream_info["camera_matrix"]["data"]).reshape(3, 3) # we
        obtain the camera matrix
```

```
    dist_coefs = np.array(stream_info["distortion_coefficients"]["data"]) # we also obtain
        the distortion coefficients

    while True:
        flag, frame = cap.read()
        found, corners = cv2.findChessboardCorners(frame, pattern_size) # corners of the
            chessboard are then detected. We dont find cornerSubPix() because its an
            expensive operation and number of positive hits are much less compared to the
            gross findChessboardCorners() method. Also its an overkill
        if(found):
            rvecs, tvecs, inliers = cv2.solvePnPRansac(object_points, corners,
                intrinsic_matrix, dist_coefs) # we use solvePnP() method to calculate the
                extrinsics -> rvecs and tvecs
            rotation_matrix = cv2.Rodrigues(rvecs)[0] # then we convert the quaternions
                rvecs into the rotation matrix
            extrinsic_matrix = np.concatenate((rotation_matrix, tvecs), axis = 1) # tvecs
                can be appended onto the rotation matrix

            print extrinsic_matrix # print the extrinsic matrix
        cv2.imshow("extrinsic", frame)
        key = cv2.waitKey(20)
        if(key==27):
            break
    cv2.destroyAllWindows()
```

We start of by obtaining *image_points*, *object_points* and K and $distortion\_coefficients$. Ofcourse, *object_points* are hard coded. K and $distortion\_coefficients$ are obtained by reading data from the *cameraConfig.yml* file we created in section 5.3.2. We will be using a *.yml* file parser which can be installed by typing on your terminal.

```
sudo pip install python-yaml
```

When I tried reading the cameraConfig.yml file, I recieved an error. This can be bypassed by editing the yml file. Comment out these lines by adding a hash, # symbol as follows
 # %YAML:1.0
#!!opencv-matrix - at two places
Check out the sample file. I do not know why *python-yaml* fails to read without this edit.
Since, *image_points* keep changing with every frame, we calculate them inside while loop using findChessboardCorners() method. Now we have all the ingredients. We calculate the *extrinsic_matrix* using *solvePnPRansac()* method which returns rvecs and tvecs. *Rodrigues()* is used to convert *rvecs* into a matrix, which is then prepended to tvecs to obtain the *extrinsic_matrix*. For the complete code visit ch7/extrinsics/extrinsics.py.

7.3 Exercise

1) Create a *cpp* program to calculate extrinsics using aruco.
Solution: ch7/exercises/Aruco_extrinsics/main.cpp

Chapter 8

Augmented Reality with OpenCV

Finally, in this chapter we are going to delve into the entire pipeline that makes an AR application. We learnt how to derive intrinsics and extrinsics. The below flowchart in figure 8.1 describes how each module falls in place.

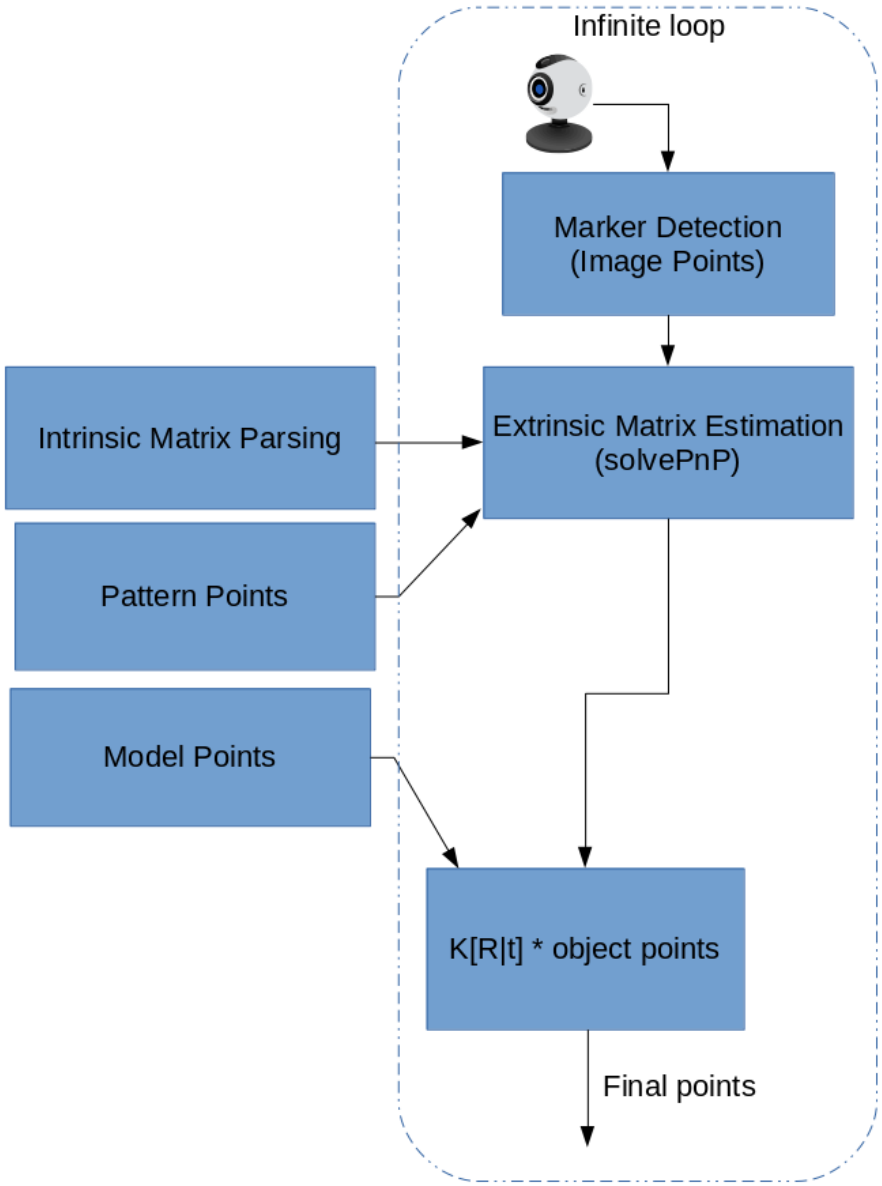


Figure 8.1: Flowchart depicting data AR using OpenCV

Model_points here refer to the 3D points of the object that we are going to overlay. The simplest object that can be overlaid is three axes of unit length with points as follows,

1. $(0,0,0) \rightarrow (1,0,0)$
2. $(0,0,0) \rightarrow (0,1,0)$
3. $(0,0,0) \rightarrow (0,0,1)$

Since we have all the ingredients lets code it up. This code is a continuation of the code from exercise 7.2.

```
camera_matrix = np.dot(intrinsic_matrix, extrinsic_matrix) # finally we compute the
    camera_matrix by combining the intrinsic and extrinsic matrix
homog_coord = np.dot(axis1, camera_matrix.T).T # Then we calculate the projection of 3D
    points on the image plane of the camera. Output is in homogeneous coordinate system
x = homog_coord[0]/homog_coord[-1] # x /
y = homog_coord[1]/homog_coord[-1] # and y carries the coordinates for the four points
    of the axes
```

Since $P = K[R|t]$, we multiply the *intrinsic* and *extrinsic* matrix to obtain the complete *camera_matrix*. We then multiply the homogenized *model_points* with the *camera_matrix* to obtain *homog_coord*, which is the homogenized *image_points* of the model after projection. x and y are de-homogenized by dividing by w (this was discussed in the last part of section 4.1). Now all that remains is to draw the obtained *image_points* on the screen.

```
cv2.line(frame, (int(x[0]), int(y[0])), (int(x[1]), int(y[1])), (255,0,0), 5)
cv2.line(frame, (int(x[0]), int(y[0])), (int(x[2]), int(y[2])), (0,255,0), 5)
cv2.line(frame, (int(x[0]), int(y[0])), (int(x[3]), int(y[3])), (0,0,255), 5)
```

For the complete code, visit ch8/chessboard/OpenCV_Augmented_Reality.

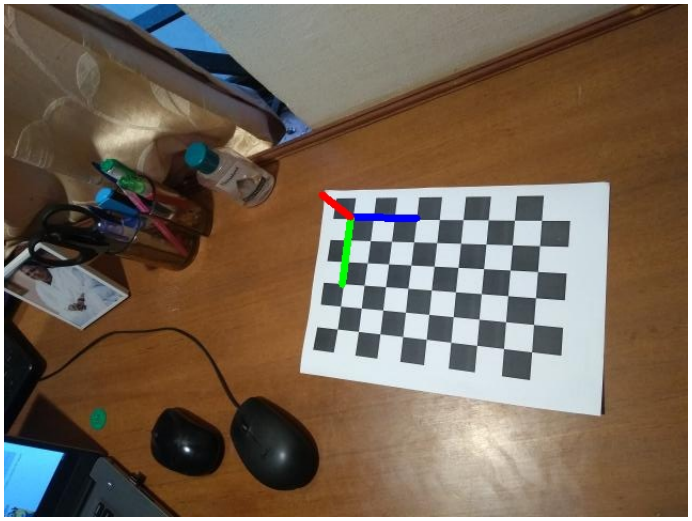


Figure 8.2

Figure 8.1 describes the output. Congratulations, you have coded your first AR application. Click this link to see a video of the same.

8.1 Exercise

1) Code a wireframe cube on aruco marker.
Solution: ch8/Aruco_Augmented_reality/main.cpp and for video of the same click here.
The above code has been refactored and made modular as, in the forthcoming chapters we will be using this as our default marker.

———————————————x———————————————

Chapter 9

Let us begin OpenGL

Congratulations if you have made it this far. OpenCV is predominantly a computer vision library and it was not built for drawing complicated realistic 3D objects on the screen with lighting effects like specular highlights. To make our learning complete, we indeed have to learn how to grapple with the monster called OpenGL. Brace yourselves, OpenGL is indeed a force to be reckoned with.

In the upcoming chapters we will solely be focussing on OpenGL; we will suspend all our work with OpenCV till a reasonable point in the future.

Why OpenGL?
Simple reason: Large and clunky game engines are whilst good for quick prototype development, they are not very good for learning. The analogy I would like to use is, using a game engine is like making coffee with a coffee maker. With a coffee maker, all you need to know is to push a button and out comes coffee. Sure, you might get good coffee, but the kind of coffee that gives you the "out of the world" experience is when brewed or concocted with great care - and not in some coffee maker. That being said, I do not mean one does not need to know how to program using a game engine. I just mean, knowing how things work just makes you a better programmer. - *I hope this adequetly answers the question.*
Disclaimer: *This book is not a comprehensive guide to OpenGL. We will simply be discussing the essentials that empowers you to make an AR application. For a more comprehensive guide, follow the tutorials by Joey de Vries. A big thanks to him for writing such a wonderful book.*

With that out of the way, lets dive into OpenGL. We will be henceforth using C++ for programming. OpenGL can be viewed as a graphics API which just provides access to the computers graphics platform. As such for drawing and user input we will be using GLFW - don't try to pronounce the name, just spell it out. Their symbiosis can be viewed as OpenGL being the middleware interacting with the hardware doing large parllel computations required to render the object. GLFW simply blits the data onto the screen. GLFW also handles user input such as mouse, keyboard, joystick clicks etc. There are many other packages that perform these operations such as SDL, SFML, MSDN etc.

9.1 Intro to OpenGL

A company called Silicon Graphics released OpenGL in 1992. Since then, OpenGL underwent several revisions. The first versions of OpenGL viz., OpenGL 1.x and 2.x had a fixed pipeline architecture. This architecture was easy to understand and program but the trade off was, it was extremely inefficient. You can still find several codebases written using these deprecated versions of pipeline. Overtime, programmers got more hungry for power and to facilitate that, around 2008, the khronos group decided to overhaul the OpenGL architecture and release what we call "modern opengl" starting from version 3.2. The code database of this version came to be called the "core-profile". The architectures prior to that called the "immediate profile" became deprecated since then. OpenGL comes pre-installed on most operating systems with the graphics driver, especially if you have a graphics card and have installed the driver for it. Currently, at the time of writing this book, the latest version of OpenGL is 4.6. Type the following command on the terminal to identify the core version of OpenGL you have.

```
glxinfo | grep "OpenGL core profile version"
```

The output should be something like "OpenGL core profile version string: 4.5 (Core Profile) Mesa 18.0.3". If you have a sufficiently decent hardware and any version of Ubuntu >12.04 or Operating Systems released since then, you will have modern OpenGL installed by default on your computer.

9.2 Setup your Environment to Program OpenGL

Now, the next task is to install GLFW. Visit *www.glfw.org* and hit the download button on the top. It will download the latest stable version of GLFW. I have downloaded it to my *~/Downloads* directory. Type the following commands on your terminal pointing to the downloaded path.

```
unzip glfw-3.x.x.zip
cd glfw*
mkdir build && cd build
cmake ..
make
sudo make install
```

The first command unzips the zip file and extracts everything to a glfw-3.x.x directory. Remember to substitute the "x.x" with the appropriate version number. Then we move into the directory with the cd command which stands for "change directory". Next, we create a directory called "build" using the "make directory" command. Its a practice to sort all the build files generated during the compilation of the library into a directory. Then we type "cmake .." which invokes the cmake build utility whilst pointing to the source of the files i.e. one directory above indicated by the two dots. Finally, we compile the project using the make command. Finally, we install the library after compilation.

The next step is to install the OpenGL Extension Wrangler Library or GLEW. The folks who write driver programs have this daunting task of having to play around with so many versions of OpenGL that best suits the hardware specifications. Whether or not an OpenGL function call is supported by your hardware is not determined at compile time. The GLEW library is responsible for ascertaining the location of the function definition at runtime. There are many libraries that does this. However, GLEW is the most widely used and tested one. OpenGL's core functions are made available as soon as you include the GLEW header. To install GLEW, type the following command.

```
sudo apt-get install libglew-dev
```

OpenGL, GLFW commands can be quite cryptic and its truly a nightmare to program these without the aid of autocompletion. Now, some of you might be using an IDE. Others, as I had suggested in chapter 1 might be using a simple text editor. Now, is about the time to migrate to an IDE to harness the awesome semantic autocompletion features. Or, like me you could simply configure your editor to support this feature (FYI, I use vim+youcompleteme). Codeblocks is a free opensource, incredible IDE that works great for OpenGL programming. To install codeblocks type the following on your terminal.

```
sudo apt-get install codeblocks
```

Since we will be coding on codeblocks quite a bit, I have made a couple of tweaks to enhance the aesthetics and usability. That wont be included in this document; click here for that.

9.3 GLFW Window

Finally, lets get to the actual coding part. Let us begin by writing the *Makefile*. In your work directory, create a *Makefile* and insert the following lines.

```
CC = g++

CFLAGS = -std=c++11 -g -Wall

INCLUDE = -I/usr/local/include/GLFW

LIBS = -lglfw3 -lm -lGLEW -lGL -lGLU -ldl -lXinerama -lXrandr -lXi -lXcursor -lX11
    -lXxf86vm -lpthread

# enter filename without extension
FILENAME = window_GLFW

all: $(FILENAME).cpp
   $(CC) $(CFLAGS) $(INCLUDE) $(FILENAME).cpp -o $(FILENAME) $(LIBS)

run:
   ./$(FILENAME)

clean:
   rm ./$(FILENAME) *.depend *.layout *.o
```

We will be using the c++11 standard for gcc compiler. The include files are in the */usr/local/include/GLFW* directory and the next line describes all the *LIBS*. Finally, the compile and run commands, and also the clean command.

Now, create a file named *window_ GLFW.cpp* in your work directory. Insert the following lines of code in the cpp file.

```
#include <iostream>
// GLEW
#define GLEW_STATIC
#include <GL/glew.h>
// GLFW
#include <GLFW/glfw3.h>

// call back function
void key_callback(GLFWwindow *window, int key, int scancode, int action, int mode)
{
    if(key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);
}

int main()
{
   glfwInit();  // Initialize GLFW
   glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);  // Describe the major version of OpenGL
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);  // Describe the minor version of OpenGL
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);  // Use the core profile
   glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);  // make sure the window is not resizable

    GLFWwindow *window = glfwCreateWindow(800, 600, "My first GLFW window", nullptr,
        nullptr);  // create a window of size 800, 600
    if(window == nullptr)
    {
        std::cerr << "failed to create a GLFW window" << std::endl;  // print out saying
            that GLFW has failed to initialize
        glfwTerminate(); // stop GLFW
        return -1;     // return -1 to the program terminating execution of the remaining
            program
    }
    glfwMakeContextCurrent(window);  // context is something like a handle of entire
        OpenGL's operations and states.
    glfwSetKeyCallback(window, key_callback); // associating the key_callback() function
        to control the glfw window.

    // some glew initializations
    glewExperimental = GL_TRUE;
    if(glewInit() != GLEW_OK)
    { // check for fails
        std::cerr << "Unable to initialize GLEW" << std::endl;
        return -1;
    }

    // setting the viewport
    glViewport(0, 0, 800, 600);

    while(!glfwWindowShouldClose(window))  // until the window recieves a termination
        signal continue the loop
    {
```

```
        glfwPollEvents(); // check for termination events such as esc press or x press on
            the dialogue window
        glClearColor(0.09f, 0.105f, 0.11f, 1.0f); // set the window color
        glClear(GL_COLOR_BUFFER_BIT); // clear the color buffer
        glfwSwapBuffers(window); // Swap the buffer
    }

    glfwTerminate(); // Close and free up all the resourses used
    return EXIT_SUCCESS;
}
```

Make sure you initialize *glew* before *glfw*. GLEW contains OpenGL definitions so you need not redefine it. The first line in *main()* initializes GLFW. The next few lines describes not to use the deprecated version of OpenGL. We also make the window non resizable. Then we create a window with size *width=800* and *height=600* and the title of the window as *My first GLFW window*. Then we set the current context of GLFW as our newly created window. A context in OpenGL can be assumed to be a handle on the entire OpenGL's states, operations etc. Multiple contexts can be defined. Then we initialize GLEW; *glewExperimental* asks glew to use the latest version. The next line sets the Viewport. A viewport describes the area of the window that can be used for viewing. Since we want to use the whole window, we describe the top coordinate as $(0, 0)$ and the *width* and *height* of the window. Now comes the main loop of the program. Every game or application that you create has a infinitely iterative loop that keeps quering for a termination condition. Termination condition can be a user exit input or a logical termination sequence such as end of the game. We can query for user termination trigger using the *glfwWindowShouldClose()* command. The *glfwPollEvents* seeks for changes to the window such as locking up etc. The *glClearColor()* command paints sets the color buffer with dark grey color; parameters are RGBA values clamped to the range, $(0, 1)$. The *glClear()* command will clear the screen to the buffer color set by the *glClearColor* command. *glClearColor* command sets the state and *glClear* command applies it. *glfwSwapBuffers* creates two color buffers. One buffer is being displayed while the other buffer is getting drawn. The advantage of this method is, only fully drawn buffers are displayed ensuring no flickering or partially drawn images being displayed on the screen. The *glfwTerminate()* command ensures all the used up resources are freed and the application is properly terminated(). The output should be a dark grey window, which upon clicking *esc, alt+F4* or the *x* button on the window terminates the program.
The *key_ callback()* function polls for *esc* key press or other window close actions. It returns *True* upon recieving a termination from the user.

Goto ch9/window_GLFW/window_GLFW.cpp to obtain the source code. Figure 9.1 shows the output.

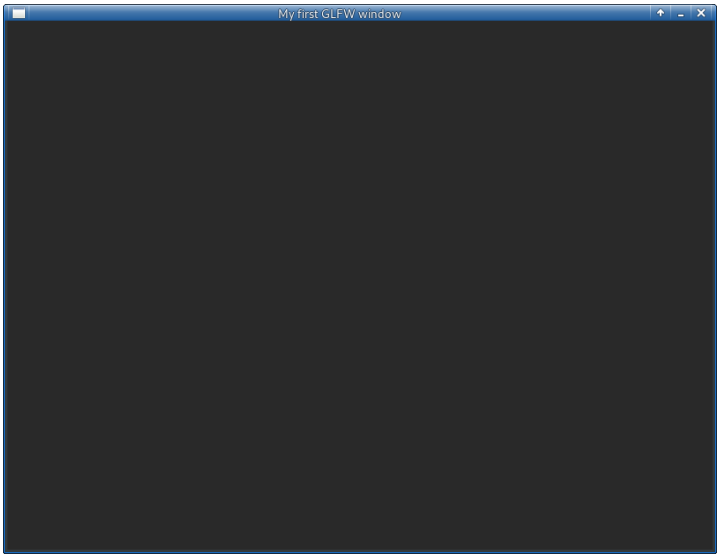
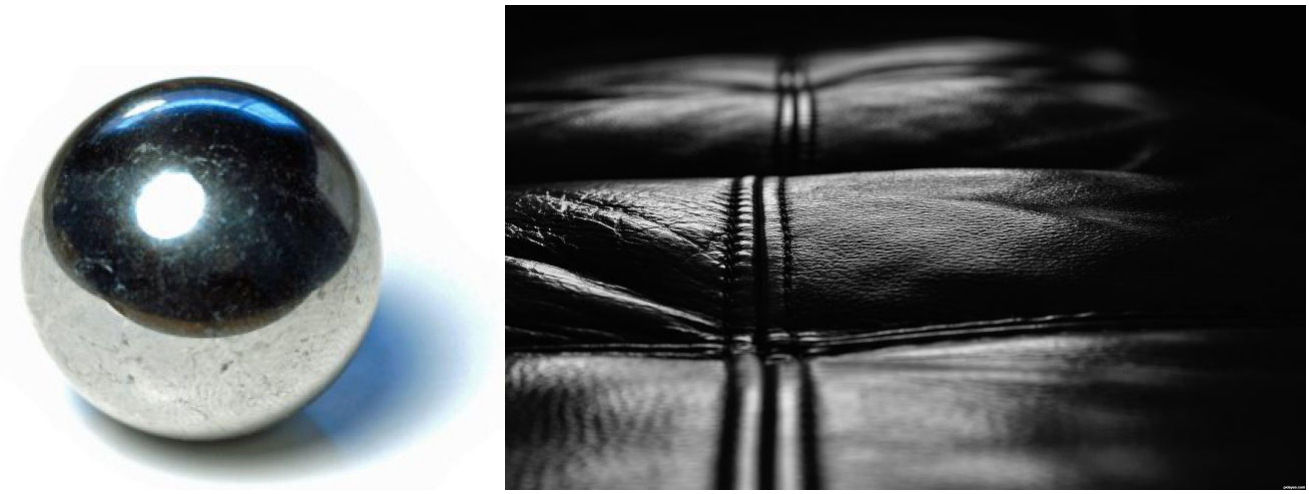


Figure 9.1: GLFW window

Chapter 10

Hello Triangle

Finally, after addressing all the paraphernalia, lets get started with OpenGL. Essentially, any graphics API or engine does one thing i.e. transform coordinates and numeric values associated with a 3D object and project it as pixels onto the screen. That sounds a lot like what we discussed in Chapter 7. The only difference is, the same object has to be rendered better. An object has several attributes such as shapes, colors, textures, materials, response to lighting etc. The outcome of what we perceive around us depends on all of these. For instance a shiny steel ball has a spherical shape and has uniform colouration of silver with spots of specular highlights where it reflects lights of objects around it. On the other hand, a leather bag wont have a definite shape and might have striations and weak specular highlights. One can notice the difference as shown in figure 10.1. These wonderful shapes, textures and colourations can be rendered through programming certain mathematical functions, based on the lighting and several other object properties the color values of pixels on the viewport. The values of each little pixel is calculated using several parallel programs called shaders. Shaders are strung together where the output of one is fed to the next to obtain the desired effect. Effects can include transformations in 3D space, response to lighting etc. Shaders are programmed using OpenGL Shading Language, GLSL.



(a) (b)

Figure 10.1: Differences in specular highlights, textures etc between a) shiny steel ball and b) leather bag

OpenGL's shader pipeline can be quite complex. A birds eye view of the shader pipeline is depicted in figure 10.2.

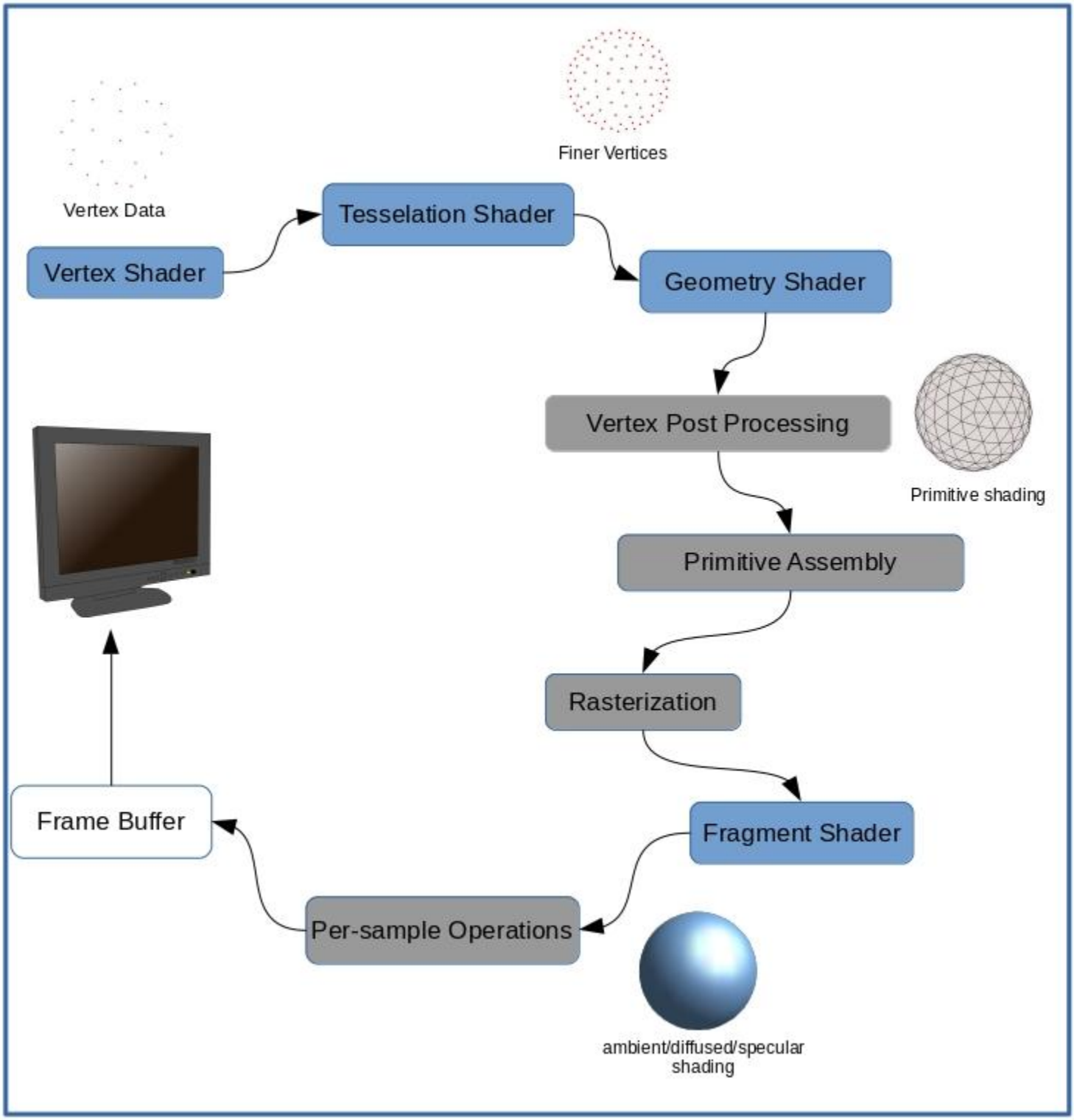


Figure 10.2: OpenGL shader pipeline

1. ***Vertex Shader:*** The vertex shader is a programmable shader that processes a vertex-stream from a a data-buffer called Vertex Buffer Object(VBO). Data which includes vertices, texture coordinates, lighting normals etc are stored in these buffers. The layout of the data is described by the user. The vertex shader obtains the vertex data and performs operations such as projections into different spaces (World Space, View Space, Clip Space etc. - we discussed these in chapter 7, and we will again deal with this in detail in the upcoming chapters).

2. ***Tesselation Shader:*** Suppose as shown the above figure you have the vertex list for a sphere. But

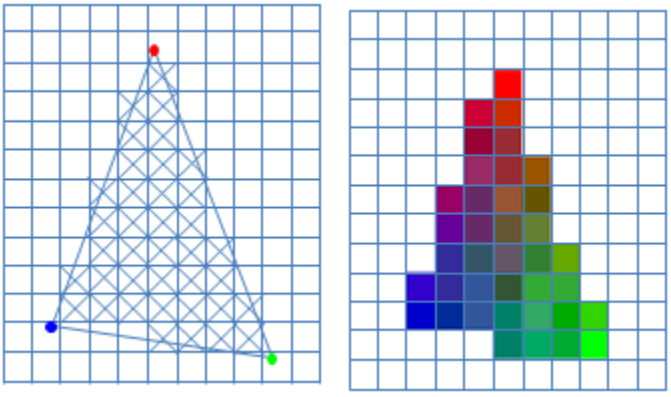
to obtain a smoother surface, you need more vertices to be interpolated in between the vertices. This task is most often done using the Tesselation Shader.

3. ***Geometry Shader:*** Suppose you are rendering a 3D model of water with ripples. An ideal choice of base mesh (list of vertices, edges etc) would be a rectangular grid. There will be several extrusions where the water moves up and down forming wavy patterns of ripples. Small periodic modifications such as these can be achieved using the Geometry Shader. Another example would be a 3D model of confetti or season salts on a pizza. One would ideally position particles on the model of pizza and extrude the individual particles into small bits of chilly/jalapeno flakes. Such extrusions are handled by Geometry Shader. Though tesselation and geometry shader create new vertex points, the difference between them is, with tesselation shader, one can only augment points within the base mesh and with geomety shader new points can be generated outside the base mesh - much like extrusion. Ofcourse, geometry shader is used only in cases of small extrusion requirements.

4. ***Vertex Post Processing:*** Vertex Post Processing deals does several operations such as clipping, clamping, perspective divide etc. At this stage several chunks of data are discarded. For instance, all the points lying outside the viewport is discarded. Points lying outside the volume of the *view frustum* (more about this will be discussed later) is discarded. When shapes are clipped, extra vertices are also added to reduce the resulting section into a composite of smaller primitives.

5. ***Primitive Assembly:*** At this stage, the vertices are connected together by edges and composited to form primitives. Primitives can be vertices, lines, triangles, polygons etc. A primitive represents small building blocks of a larger complex shape. Primitive Shading is performed where each primitive is associated with a uniform color tone.

6. ***Rasterization:*** Up till now we have been dealing with points and surfaces in 3D space. At this stage, each primitive is then discretized into pixels. Rasterization decides which pixel should be associated with what value of base color and depth as shown in figure 10.3. A gross color tone is described at this stage.



(a) (b)

Figure 10.3: Rasteriztion; image source:link

7. ***Fragment Shader:*** This shader calculates the finer color values based on several parameters such as light, material, texture, shadows etc. The output of the fragment shader is a color and depth value.

8. ***Per-sample Operations:*** At this stage blending and stensil tests are performed.

During implementation, several steps are abstracted. A complete pipeline is far more complicated. Take a look at this.
Though there are several stages to this pipeline, we will only be implementing the vertex and fragment shader. A complete OpenGL program requires atleast these two shaders to be programmed.

10.1 Data Flow Pipeline

Since, modern OpenGL uses GPU's, there are basically three main steps involved in processing and data transfer

1. Data Upload into GPU memory
2. Process Data in GPU
3. Output results on screen

Before GPU's became widely used in graphics, large loop constructs were used to process the data. However, while handling large number of data points, this process flow became extremely inefficient. Often, systems required each vertex/fragment to undergo the same process. GPU's were best suited for the job. GPU's contains several tiny cores that can process each data unit in parallel. Here data unit can mean vertex, fragment or in parallel processing paradigms, just small individual complete bundles of data.

Data is first transfered from RAM to the GPU memory (remember we spoke of buffers). This is done because transferring data from RAM to GPU takes up a lot of time. So whenever we can, we try to transfer large buffers of data at once. Processing is much faster. A special program called ***kernel*** is executed. Kernel is the generic name used in the parallel processing paradigm. In OpenGL's language, its called a ***shader***. For processing each vertex, a ***kernel instance*** is instantiated. A large buffer of vertices are processed together. The GPU program creates a ***thread*** which is a list of homogeneous kernel-instances running on individual cores. The number of cores allocated to running a thread is usually hidden from us when we code in OpenGL. GPU's are particularly not very good at running complex control flows. Homogeneous operations over large number of items is what is ideal for parallel processing.

Also, recall in section 9.1 we discussed, the actual implementation of OpenGL is done by the graphics driver writers. Since, there are several hardware architectures there is bound to be implementation differences between the manufacturers; the actual implementations of OpenGL will differ from one party to another. Understandably, its not like Intel/Nvidia/AMD would sit down, have a meeting to reach a consensus. Thus, its impossible to have a pre-compiled binary (or) atleast improbable/impractical for compiling and linking a shader program. Thus, as we will see next, we are going to dynamically compile and link our shader program at runtime.

The final step is to output the result on the screen. Usually, most parallel programs have calls to transfer the data from the GPU to the CPU accessible memory. Since, video cards are directly tied with graphics cards onto which our monitors are connected, painting the data on the screen is directly handled by the GPU.

10.2 Hello Triangle

Its always been a customary to start any 'computer language' with 'Hello World' so as not get cursed by the 'programming Gods'. Our graphics counterpart of 'Hello World' is a simple triangle program. The simple reason being, most of the complex shapes use triangles as base primitives. However, unlike other languages where the 'Hello World' program is relatively simple and it ramps up as we go further, OpenGL requires so many building blocks in place right before we start. A simple 'Hello Triangle' program can seem pretty daunting. But the rest of the strides are much simpler.

All the documentation of OpenGL's calls can be referred to from the site "docs.gl".
Let us start by defining vertices to be input into the vertex shader.

```
// Lets define an array of vertices
GLfloat vertices[] = {
   -0.5f, -0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    0.0f,  0.5f, 0.0f };

GLuint VAO, VBO; // we need an unsigned int to be the passed to the next function call
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO); // Generates buffers. We only need '1' buffer.
glBindVertexArray(VAO); // Bind vertex array objects first before VBOs

glBindBuffer(GL_ARRAY_BUFFER, VBO); // We are selecting the buffer. Binding in OpenGL
    means selecting.
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW); // Let put
    data into the buffer. This is similar to memcpy function in c.

// right now we threw some data into the VRAM. Ofcourse the next logical step is to
    describe how this data is organized. This is achieved by glVertexAttribPointer()
    whose parameters are
/* void glVertexAttribPointer( GLuint index,
                      GLint size,
                      GLenum type,
                      GLboolean normalized,
                      GLsizei stride,
                      const GLvoid * pointer); */
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3*sizeof(GLfloat), (GLvoid*)0);

glEnableVertexAttribArray(0);
glBindVertexArray(0);           // unbinding VAO
```

In the above snippet of code, we first create an array of GLfloats (*GLfloat* is similar to *float*). These form the vertices of a triangle in the *xy plane* with *z=0*. The next step is to create a memory buffer (buffer here simply means a space in the memory) in the GPU so that we can transfer the data from the RAM to VRAM (GPU memory). To do that, we need to request the GPU to give us a block of memory. The *glGenBuffers()* call does that. Since it requires an address of integer variable as input, we pass the *GLuint VBO* variable to it.

What is a VAO (Vertex Array Object)? : Let us consider the following scenario. We have a scene with many 3D objects to be drawn on the screen. Each object being complex, consists of many vertices (similar to above, where we described for a triangle). In addition to vertices, the object may have other ***attributes*** like 2D texture coordinates and 3D normals. More on textures and normals will come in later chapters. There are many ways how the attributes could be organized. Fig 10.4a describes an *array of structures* and fig 10.4b describes a *structure of arrays*.

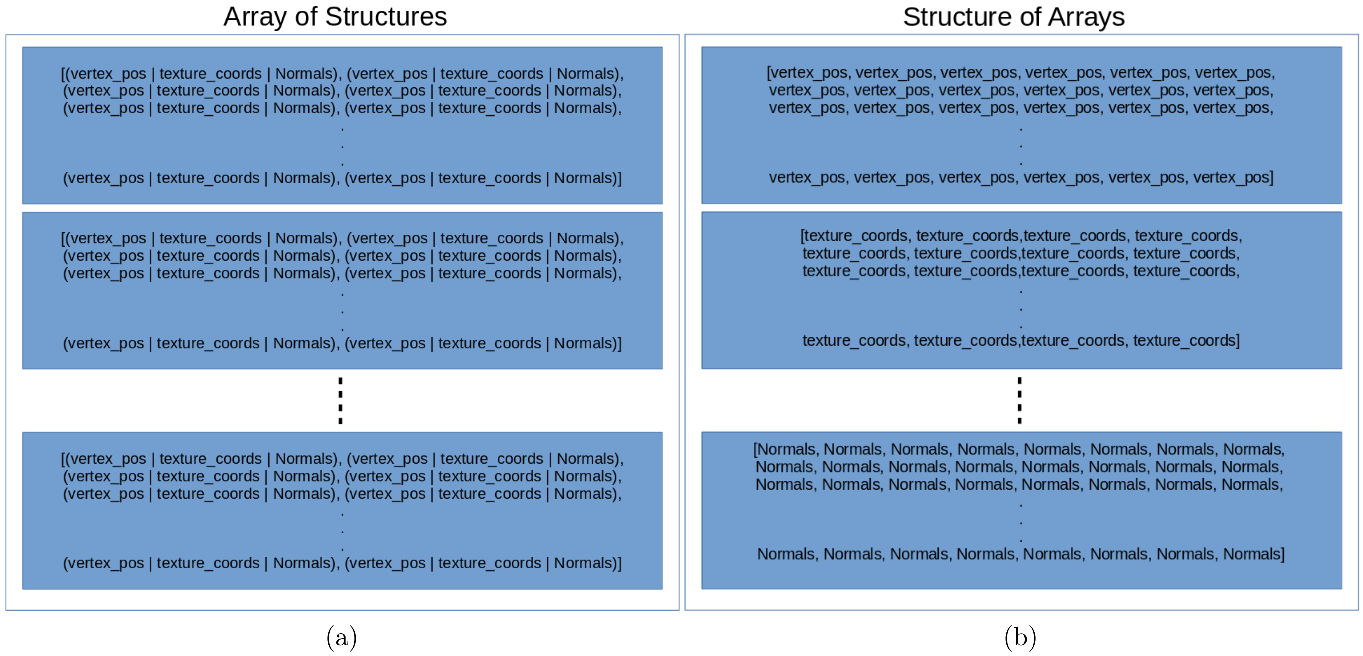


Figure 10.4: Data organization as (a) array of structures and (b) as structure of arrays

Thus the layout of the data within a memory buffer is completely user defined. For instance, if we are drawing a dining table with a teacup-and-saucer and a plate containing a donut, each of these form individual objects/meshes. The data describing a teacup might be ordered in a totally different way as compared to data describing a saucer. Also, each of these objects might have a different number of attributes described within the data. To sort all this out, we use a data structure called VAO. The advantage of a VAO is, we just have to bind the VAO once and describe the layout of the buffer (how to do this - we will discuss in the upcoming paragraphs). Then we simply unbind the VAO. When needed next, say when we are about to draw it, we simply bind the VAO and issue a draw call; then we unbind it. In short, ***VAO*** can be viewed as something that encapsulates the pointers to all the attributes required to draw that model/models. OpenGL being a big state machine operates like this: Create and load data and programs required to process the data. For instance, vertices, textures, normals etc., of teacup, saucer, donut, table and other stuff. Also describe their layouts as different VAO's. Describe shader programs that act on these data. Finally, when we are about to render something on the screen, bind the required items alone, their shader programs and issue the draw call. Then unbind everything. Its similar to a carpentry shop, where the carpenter (OpenGL) takes the necessary materials (buffer data organized into VAO's) and the tools (shader programs) to create an object (rendering some thing on the screen).

Similar to VBO's, VAO's also require an integer pointer to be generated. *GlBindBuffer()* binds the buffer. It gets bound and unbound along with VAO. *glBufferData()* actually dumps the data in the "vertices" array into the GPU memory. *GL_ARRAY_BUFFER* describes to the GPU that the buffer is indeed an array. We will look at some of the other kinds later. Now, when and how the data is transffered from the memory or the memory management details of the GPU is abstracted away from us by the GPU manufacturers. Details of the internal workings of most GPU's are outright trade secrets which wont be divulged to us.

Uptil now we have discussed the logistics of how the data is transferred and the data structures and containers used for storing data. Now we are going to describe the actual layout of the data and how we are going to describe it to the computer. This is done by *glVertexAttribPointer()* function. The description of the parameters list can be obtained from the documentation - link.

1. ***GLuint index*** - in the above example we have described only vertex positions as attributes. Usually we also describe other attributes like texture coordinates, normals etc with these. For instance a

typical array would look somewhat like figure 10.5.

2. ***GLint size*** - describes the number of components for the attribute under consideration. In the above case, we have (x, y, z) values for the vertices so this number is 3.

3. ***GLenum type*** - This describe the datatype of the values of the attribute. You can refer the list of supported datatypes in the documentation.

4. ***GLboolean normalized*** - Should you normalize the data?

5. ***GLsizei stride*** - How far should you jump from the start of the first attribute to obtain the next attribute. In our case we only have vertices as attributes.

6. ***const GLvoid* pointer*** - This field describes the offset incase you want to start from "not the first GLfloat value"

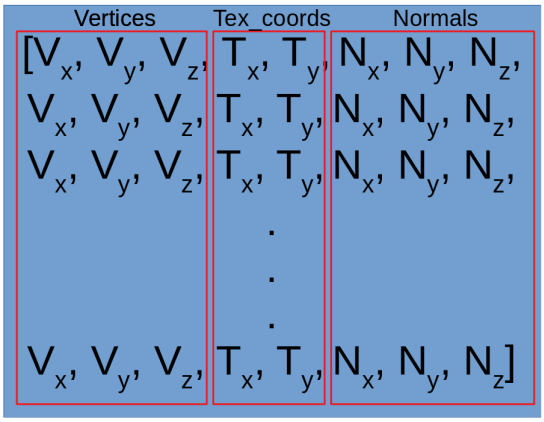


Figure 10.5: A typical data layout in OpenGL consisting of Vertices, Textures and Normals

Let us write two shaders viz., the vertex and the fragment shader to compile and link this data to be run on the GPU. There are many ways to input the shader program into the OpenGL program. We can write the shader code in a new file, they can be distributed as binaries or infact we can include them as a long string inside our program. In the interest of keeping the logic simple for the start, we will, for now, be including the shader program within our source code as a long string. In the upcoming chapters we will learn how to read off the shader code from a separate file.

```
#version 330 core // Description of shader version
layout (location=0) in vec3 position; // attribute (vertex_position) location

void main()
{
   gl_Position = vec4(position, 1.0f);
}
```

This is an example of a rudimentary vertex shader. Each shader begins with the declaration of the shader version. Since we are using OpenGL version 3.3, its corresponding GLSL version is 330. We also mention that we are going to use the core functionality.

in's and *out*'s are keywords that assign variables types to be either input or output type. Input type variables take data in from OpenGL program and output type variables pass the to the next block in the

shader pipeline. 'layout (location=0)' describes the location of the attribute-vertex (more on this later). "gl_Position" is passed to tesselation shader.

```
#version 330 core
out vec4 color;

void main()
{
   color = vec4(0.85f, 0.26f, 0.22f, 1.0f); //RGBA
}
```

The fragment shader code is very similar to the vertex shader code. The color variable is declared as output and it passes it on to the next block in the shader pipeline.

Just like we would compile and link any program, our next job is to compile and link the shaders within our OpenGL source code. To compile the vertex shader,

```
// Compiling Vertex Shader
GLuint vs = glCreateShader(GL_VERTEX_SHADER); // we first create a shader and get a
    shader handle named vertexShader
glShaderSource(vs, 1, &vertexShaderSource, NULL); // Then we pass the vertex shader
    string we created -> vertexShaderSource
glCompileShader(vs);

// now lets check for compile time errors
bool success; // flag for errors
GLchar infoLog[512]; // memory for storing errors
glGetShaderiv(vs, GL_COMPILE_STAUTS, &success);
if(!success)
{
   glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
   std::cerr << "Vertex Shader compilation failed : " << infoLog << std::endl;
}
```

Cool now we have compiled our vertex shader and checked for compile time errors. Lets do the same for the fragment shader.

```
// Compiling Fragment Shader
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER); // we create the fragment shader and
    otbain the handle
glShaderSource(fs, 1, &vertexShaderSource, NULL);
glCompileShader(fs);

// Checking for compile time errors
// using the same compile check flag and the error memory buffer
glGetShaderiv(fs, GL_COMPILE_STAUTS, &success);
if(!success)
{
```

```
   glGetShaderInfoLog(fs, 512, NULL, infoLog);
   std::cerr << "Fragment Shader compilation failed : " << infoLog << std::endl;
}
```

Okay now lets link the two shaders into a program. The below program must be fairly self explanatory.

```
// Linking Shader
GLuint program = glCreateProgram();
glAttachShader(program, vs);
glAttachShader(program, fs);
glLinkProgram(program);

// Lets check for link errors
glGetProgramiv(program, GL_LINK_STATUS, &success);
if(!success)
{
   glGetProgramInfoLog(program, 512, NULL, infoLog);
   std::cerr << "Error linking shader program " << infoLog << std::endl;
}

glDeleteShader(vs);
glDeleteShader(fs);
```

All that remains now is to actually perform the draw call. The draw call is done within the while loop.

```
while(!glfwWindowShouldClose(window))
{
   /* ------------------------
   ---- other glfw calls ----
   ------------------------ */

   glUseProgram(Program); // Binding the shader program
   glBindVertexArray(VAO); // Binding VAO of our triangle
   glDrawArrays(GL_TRIANGLES, 0, 3); // Draw call for our triangle
   glBindVertexArray(0); // unbind VAO

   /* ------------------------
   ---- other glfw calls ----
   ------------------------ */
}

// Deleting the used resources ------------------------------------------
glDeleteVertexArrays(1, &VAO);
glDeleteBuffers(1, &VBO);
glfwTerminate();
return 0;
```

In this snippet of code, we simply bind the shader program and the VAO. Then we call the draw function. The draw function is define as, *glDrawArrays(GLenum mode, GLint first, GLsizei count);*

- GLenum mode - which primitive to be used?
- GLint first - starting index
- GLsizei count - How many indices to be rendered

The docs contain detailed information.
Visit ch10/draw_triangle.cpp to obtain the full source code. If everything works well, then you should get a triangle as shown in figure 10.6.

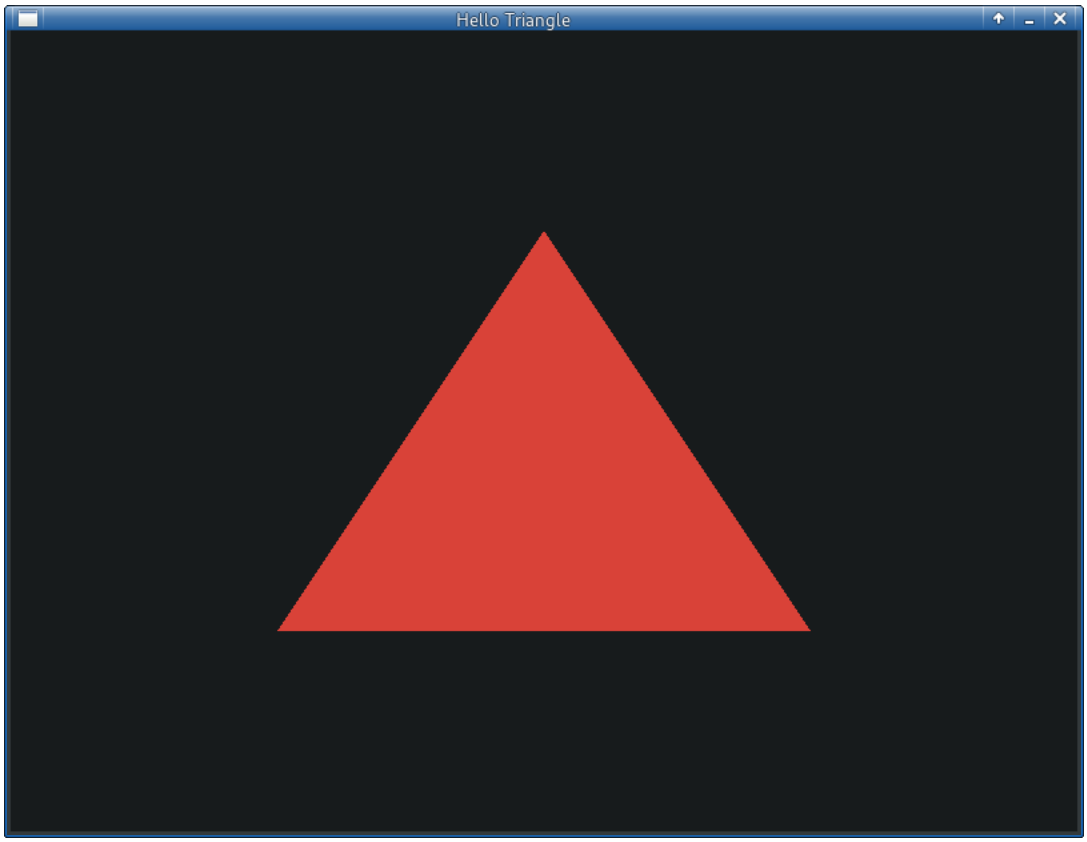


Figure 10.6: Our first OpenGL triangle using Shaders

10.3 Just a few more little details to tie it all up

Now that we have created a triangle, take it as an exercise to create a rectangle. You can find the source code under ch10/draw_rectangle.cpp.

Notice when you drew a rectangle, you had to manually describe vertex points again causing redundancy. Instead of describing a rectangle with 4 points we used 6 points.

```
GLfloat vertices[] = { -0.5f, -0.5f, 0.0f, /* left bottom */
                       -0.5f,  0.5f, 0.0f,  /* left top */
                       0.5f, -0.5f, 0.0f, /* right bottom */
```

```
                        -0.5f, 0.5f, 0.0f,  /* left top */
                         0.5f, 0.5f, 0.0f,  /* right top */
                        0.5f, -0.5f, 0.0f}; /* right bottom */
```

This is not a very efficient method to describe vertices when it comes to describing objects with a lot of vertices and interconnections. Ideally, we would like to describe a list of vertices, [vertex1, vertex2, vertex3, vertex4] and then tell OpenGL to draw the first triangle using vertex1, vertex2 and vertex3 and the second triangle using vertex2, vertex3 and vertex4.

Thankfully, there is a method to do this. OpenGL allows us to create what is called an *element_ buffer_ object*. This buffer essentially hosts the ordered index list. Upon binding this to a VAO, OpenGL draws the object taking the number of vertices successively as required by the primitive.

```
 GLfloat vertices[] = { -0.5f, -0.5f, 0.0f, /* left bottom */
                        -0.5f, 0.5f, 0.0f,  /* left top */
                        0.5f, -0.5f, 0.0f, /* right bottom */
                 0.5f, 0.5f, 0.0f}  /* right top */

GLuint indices[] = {1, 2, 3, 2, 3, 4};

 GLuint VBO, VAO, EBO;
 glGenVertexArrays(1, &VAO);
 glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);
 glBindVertexArray(VAO); // Bind vertex array objects first before VBOs

 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3*sizeof(GL_FLOAT), (GLvoid*)0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

 glEnableVertexAttribArray(0);
 glBindVertexArray(0);           // unbinding VAO

 while(!glfwWindowShouldClose(window))
 {
   /*----------------------------
   --------- other calls --------
   ----------------------------*/
     glUseProgram(program);
     glBindVertexArray(VAO);

   // A new draw call instead of glDrawArrays()
   glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0); // parameters -> primitive, #
       of indices, type, offset

   /*----------------------------
   --------- other calls --------
```

```
    ----------------------------*/
}
```

In the above program we have used *glDrawElements()* as our draw call, which takes the values from the *element_buffer_object* (or) *EBO* to render on the screen. You will get an output similar to figure 10.7. You can find the complete source code on ch10/draw_rectangle_ebo.cpp.

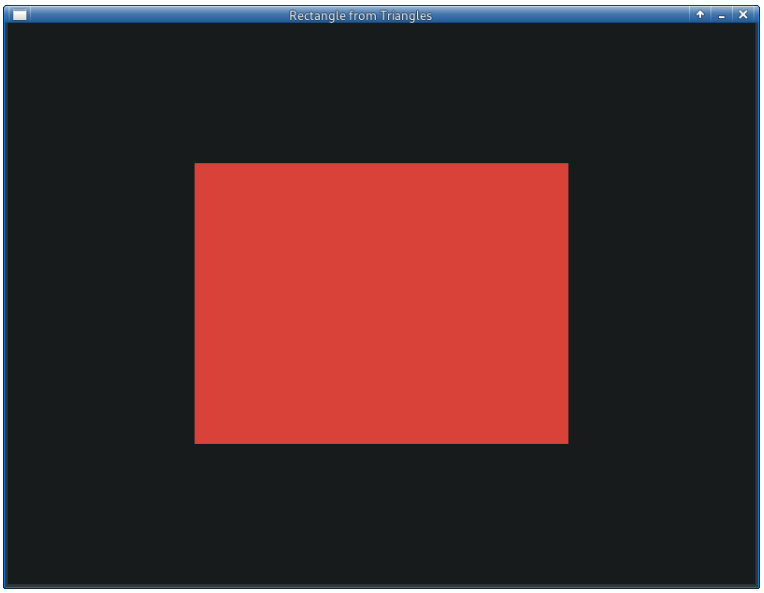


Figure 10.7: Rectangle using two triangles

10.4 wireframe objects

Now that you have created a triangle, how do u create a wireframe object(Chapter 1 illustrates a wireframe teapot)? To draw a wireframe, we just have call a function called glPolygonMode() with parameters *GL_FRONT_AND_BACK* to display the wire on both sides of the triangle and *GL_LINE* to describe OpenGL to draw everything using a line object. Uncomment *glPolygonMode()* line in the previous code ch10/draw_rectangle_ebo.cpp to obtain the a wiremesh as shown in the figure 10.8.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); // Describes the primitive to be used.
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0); // Draw call
```

A triangle has two faces i.e., front and back. *glPolygonMode()* describes to OpenGL, to use *GL_LINE* as primitive. *glDrawElements()* is another popular draw call. Just comment the *glDrawArrays()* line in the triangle program and uncomment the *glPolygonMode()* and *glDrawElements()* function to obtain the wireframe of the image.

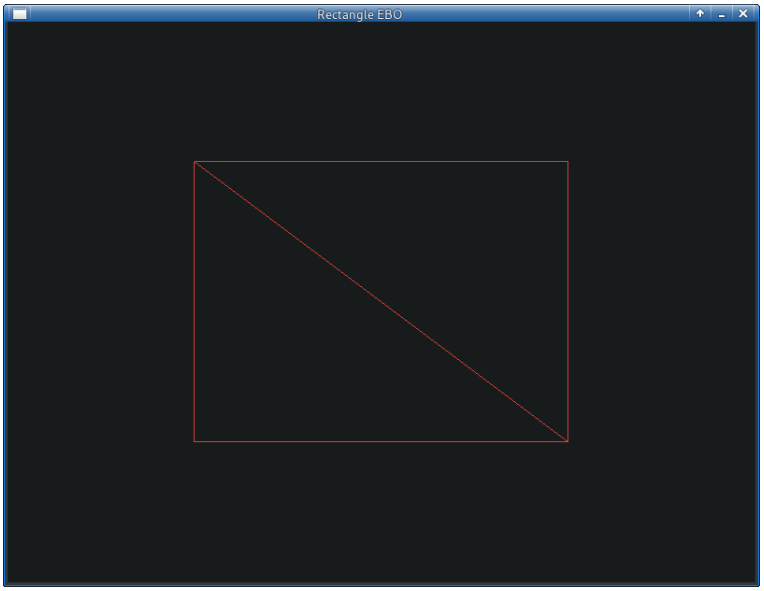
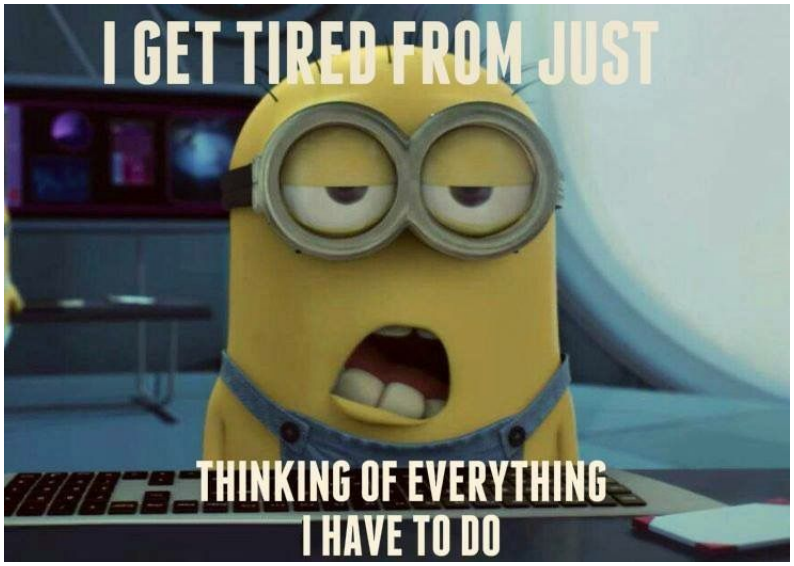


Figure 10.8: Wireframe rectangle



That was a lot of concepts we just discussed. I am sure it must have felt like a ton of bricks and you must be exhausted! Congratulations, that was one of the hardest topics we just covered.

Chapter 11

Shaders

Before we start with more code, lets clean our solution a bit so that its nice and sorted. Here is a directory tree I in accordance to which I sort my files. The main directory contains *bin, debug, include* and *src*. *bin* hosts all the executables generated. *debug* hosts certain debug logs. *include* has all the user defined and/or library header files. *src* houses the source files with other *resources* and *shaders* directories for some images and stuff we will use later and the shader files. The associated files are inserted into the appropriate directories and the *Makefile* is modified appropriately.

11.1 Shader compilation and linking - ABSTRACTION

Back to shaders! Its not too fun to write shaders as long cryptic strings. Fortunately, we can write shaders on separate files and include them in our OpenGL program. Lets go ahead and do that. We will be modifing the code written in ch10/draw_rectangle/draw_rectangle.cpp. Go ahead and copy the vertex and fragment shader code from the *.cpp* file into a *vertex_shader.vert* and *fragment_shader.frag* file respectively. Make sure to store the files in the *shaders* directory.

In addition to including the shaders as separate files we can also abstract the compilation and linking process so that we only need the shader-program call statement in *main.cpp*. Create a header file called *shader.h* and *shader.cpp* in the *include* directory.

In *shader.h* insert the following code.

```
#ifndef SHADER_H
#define SHADER_H

#define GLEW_STATIC
```

```
#include <GL/glew.h>

// does compilation, linking of shaders when called
class Shader
{
public:
   // shader program handler
   GLuint program;

   // constructor
   Shader(const char* vertex_shader_path, const char* fragment_shader_path);

   // Destructor
   ~Shader();
};

#endif
```

Its a typical header file with a constructor and destructor - nothing fancy.
shader.cpp is organized as follows

```
// Usual includes -------------
#define GLEW_STATIC
#include <GL/glew.h>
#include <iostream>
#include <fstream>
#include <cstring>
#include "shader.h"
#include <sstream>

GLchar* get_program_from_file(bool*, const char* )
{
   // this function reads the shader code from files and dumps it into a string
   /* --------- function code here ----------------- */
}

GLuint compile_shader(std::string, const char*)
{
   // simply the compile command we did in chapter 10
   /* --------- function code here ----------------- */
}

Shader::Shader(const char* vs_path, const char* fs_path)
{
   // here we do the read-from-file and compile calls
   // since linking is relatively small we implement it in the constructor itself
   /* ------- constructor code here ------------------ */
}

Shader::~Shader()
```

```
{
   /* ------- destructor code here ------------------ */
}
```

Now lets look at each function block separately. We will start with the constructor.

```
Shader::Shader(const char* vs_path, const char* fs_path)
{
   bool s_fail; // flag for shader compilation fail

   // get the vertex shader code
   const GLchar* vs_code = get_program_from_file(&s_fail, vs_path); // fxn for parsing
      vertex shader code from file.
   if(s_fail) std::cerr << "ERROR: VERTEX SHADER FILE COULD NOT BE OPENED" << std::endl;
      // error check for open

   // get the fragment shader code
   const GLchar* fs_code = get_program_from_file(&s_fail, fs_path); // fxn for parsing
      fragment shader code from file
   if(s_fail) std::cerr << "ERROR: FRAGMENT SHADER FILE COULD NOT BE OPENED" << std::endl;
      // error check

    //compile vs and fs ------------------------------------------
   GLuint vs = compile_shader("VERTEX", vs_code); // compile function
   GLuint fs = compile_shader("FRAGMENT", fs_code); // compile function

   // link the shaders -------------------------------------------
   // since linking is not very lengthy lets do it here. Its the same program as discussed
      in section 10.2 under the "Linking Shader" block.
   this->program = glCreateProgram();
   glAttachShader(this->program, vs);
   glAttachShader(this->program, fs);
   glLinkProgram(this->program);

   // check for linking errors
   GLint success;
   GLchar errorLog[512];
   glGetProgramiv(program, GL_LINK_STATUS, &success);
   if(!success)
   {
      glGetProgramInfoLog(this->program, 512, NULL, errorLog);
      std::cerr << "ERROR: SHADER PROGRAM LINKING FAILED" << errorLog << std::endl;
   }

   //-----------------------------------------------------------------
    // Delete shaders after compilation
   glDeleteShader(vs);
   glDeleteShader(fs);
}
```

Here we called two functions viz., *get_program_from_file()* and *compile_shader()*. We will look at *get_program_from_file()* function next. The main purpose of this function block is to parse the shader codes from files and return a "code-string" handle.

```
GLchar* get_program_from_file(bool* s_fail, const char* file_path) // s_fail checks for
    file open success/faliure, and file_path is the path to the file
{
   std::ifstream f(file_path); // file handler
   std::stringstream f_stream; // reading all the data as a string stream

   // error_check
   if(!f) *s_fail = 1; // assign file open success/faliure status
   else *s_fail = 0;

   // read all the contents of that file
   f_stream << f.rdbuf(); // read the contents of the file buffer into a string stream.

   // copy the data to a memory in the heap so that after the scope of this function runs
       out it still remains
   GLchar* code = new GLchar[f_stream.str().length()+1];
   std::memcpy(code, f_stream.str().c_str(), f_stream.str().length()+1);

   // lets close the open files
   f.close();

   return code;
}
```

Next we will write a function block called *compile_shader()* that compiles a shader string obtained from the *get_program_from_file()*.

```
GLuint compile_shader(std::string shader_type, const char* code) // takes a string
    describing shader type and the shader string as parameters.
{
   GLuint s = glCreateShader(GL_VERTEX_SHADER); // default we assume its a vertex shader

   if(shader_type == "FRAGMENT") // else its a fragment shader
      s = glCreateShader(GL_FRAGMENT_SHADER);

   glShaderSource(s, 1, &code, NULL); // here on its the usual code discussed in section
       10.2
   glCompileShader(s);

   GLint success;
   GLchar errorLog[512];
   glGetShaderiv(s, GL_COMPILE_STATUS, &success);
   if(!success)
```

```
    {
        glGetShaderInfoLog(s, 512, NULL, errorLog);
        std::cerr << "ERROR: " << shader_type << " SHADER COMPILATION FAILED. " << errorLog
            << std::endl;
    }
    return s; // returns the compiled shader object
}
```

For the entire code visit ch11/shader_in_files. Cool! I hope you obtained the same rectangle that you got in figure 10.7.

11.2 Uniforms

Now that we know how to create a simple rectangle, let us learn how to change the color of the rectangle over time as it runs. 'Shaders' being small snippets of code run on the GPU are very introverted in nature. They do not interact with other programs too often. Infact, their primary mode of interaction is via the *ins* and *outs*. But this often becomes insufficient when we want to send small pieces of data directly into shaders. To facilitate that OpenGL has ***Uniforms***. Uniforms are located in shaders and the CPU upon knowing the location of a uniform, can directly insert the data onto the GPU. In our scenario, we want to change the color of our rectangle. Let us go ahead and code this. We will be using the output of section 11.1 as our base code.

In the *fragment shader*, lets add a uniform.

```
#version 330 core
out vec4 color;

uniform vec4 u_color; // we added a uniform which is a vector of 4 float values

void main()
{
    color = u_color; // here we assigned the color variable with our variable from the
        uniform.
}
```

Now in our *.cpp* file,

```
    GLint color_location = glGetUniformLocation(our_shader.program, "u_color"); // this
        finds the location of the uniform -> ourColor

    /* other lines of code */

    // let us write a logic to change the color of green in our code
    GLfloat timeValue = glfwGetTime();
    GLfloat greenValue = (sin(timeValue)/2) + 0.5;

    // now insert the green color after binding the shader program.
    glUseProgram(our_shader.program);
```

```
glUniform4f(color_location, 0.0f, greenValue, 0.0f, 1.0f); // we use this function to
    update the color value.
```

The first thing we need to do is to find the location of the uniform variable. To do this we pass our shader program and the uniform variable to *glGetUniformLocation()* function. This returns an integer that describes the *color location*. The next two lines of code simply uses time to calculate a shade of green. We then bind the shader program. Now we insert the calculated green color into our uniform. *glUniform4f()* function takes the *color location* and the four color values as input and inserts it into the uniform. You should get a window that shows a green rectangle that continuously changes in green color intensity as shown in figure 11.1. For the complete code, click ch11/uniforms.

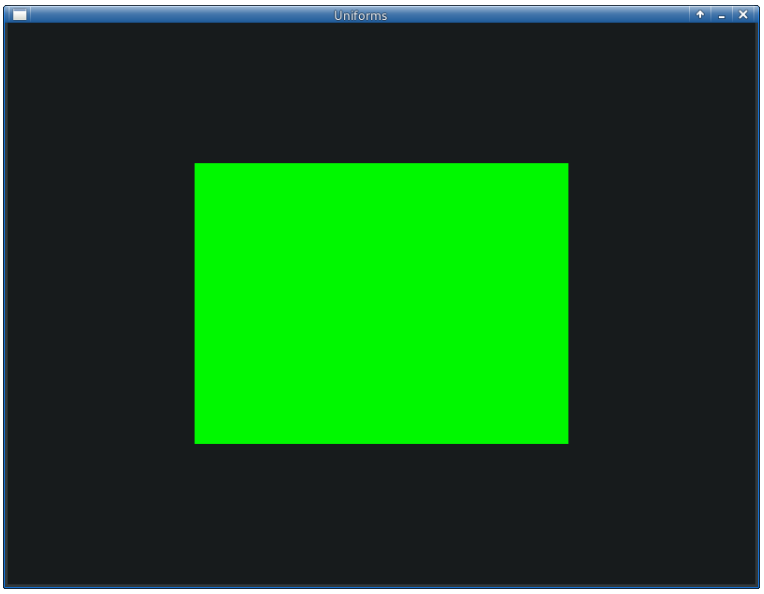


Figure 11.1: Uniforms - color varying time lapse

11.3 Multi-colored Triangle

How do you create a multicolored triangle? Say you want something like in figure 11.2. Let us see how its done. For this assignment we are going to modify the code from ch11/shader_in_files. We will modify the code to make a triangle. Let us do that first.

In *main.cpp*,

```
// updated vertex list with vertex positions and vertex colors
                // vertex pos      vertex colors
GLfloat vertices[] = { -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, // left bottom
                0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f,// right bottom
                0.0f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f}; // top
```

Note: right beside the vertex positions, we describe the associated color of the vertex.
Now the next step is to modify the attribute description.

```
// attribute 0 vertex positions
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6*sizeof(GL_FLOAT), (GLvoid*)0);
glEnableVertexAttribArray(0);

// attribute 1 vertex colors
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6*sizeof(GL_FLOAT),
    (GLvoid*)(3*sizeof(GL_FLOAT)));
glEnableVertexAttribArray(1);
```

We discussed attribute layout description in section 10.2 above figure 10.5. Now we have attribute 0 and 1. Lastly, we also modify the number of vertices to be rendered onto the screen. This is done at the draw call. Modify the draw call parameters to 3.

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Now in the vertex shader, we have to account for 2 attributes. Then we have to create an out-type shader variable that will be passed on to the fragment shader. We assign the obtained vertex color values, (vertex_color) to out variable v_color.

```
#version 330 core
layout (location = 0) in vec3 position; // vertex_position attribute
layout (location = 1) in vec3 vertex_color; // vertex_color attribute

out vec3 v_color; // since we will be passing on these values to the fragment shader, we
    need an out type shader variable

void main()
{
   gl_Position = vec4(position, 1.0);
   v_color = vertex_color; // obtained color from each vertex is passed to the fragment
       shader
}
```

In the fragment shader, we simply collect the values from the vertex shader using an in-type shader variable and make it a $vec4$ by appending $alpha = 1$ value to it.

```
#version 330 core

out vec4 color;
in vec3 v_color; // the color values are obtained from the vertex shader so we need an in
    variable

void main()
{
```

```
    color = vec4(v_color, 1.0f); // v_color is assigned
}
```

You should now get a colorful triangle. Each vertex takes the fullest tone of R, G and B. Fragment shader interpolates the colors of the remaining pixels to obtain a nice color blend. The output should resemble figure 11.2.

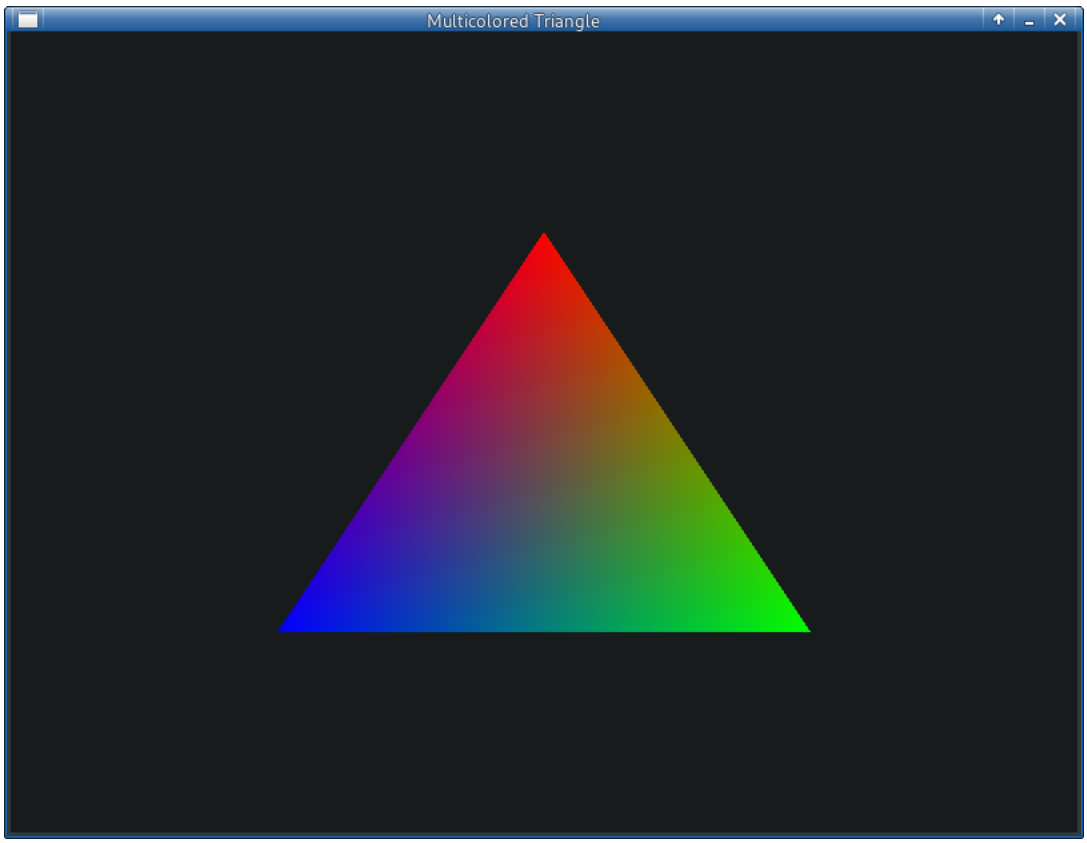


Figure 11.2: Multicolored triangle

You can obtain the complete source code in ch11/multi_colored_triangle.

Chapter 12

Textures

This is going to be an interesting chapter. Textures are going to provide a leap in the realism of the 3D object you are creating. Let us start of with, **what is a texture?** A texture is nothing but an image wrapped over a 3D object. For instance, a cube can be made a crate by wrapping a wooden texture over it. A texture wrapped image of a crate is shown below.

12.1 Loading Textures

We need a mechanism to load a textures into OpenGL. Innately, OpenGL does not give us the facility to load images. So we will use an external library to load textures and then we will use OpenGL to create texture objects. There are several libraries that does the job for us. But the most commonly used one is *Simple OpenGL Image Loader - SOIL*. To install SOIL, type

```
sudo apt-get install libsoil-dev
```

When we load a texture, an image wraps around a set of vertices. Its our job to specify to OpenGL the coordinates of the image mapped to each vertex. This process is called *UV mapping*. Here U and V denotes the texture coordinates. Below figure illustrates UV mapping.

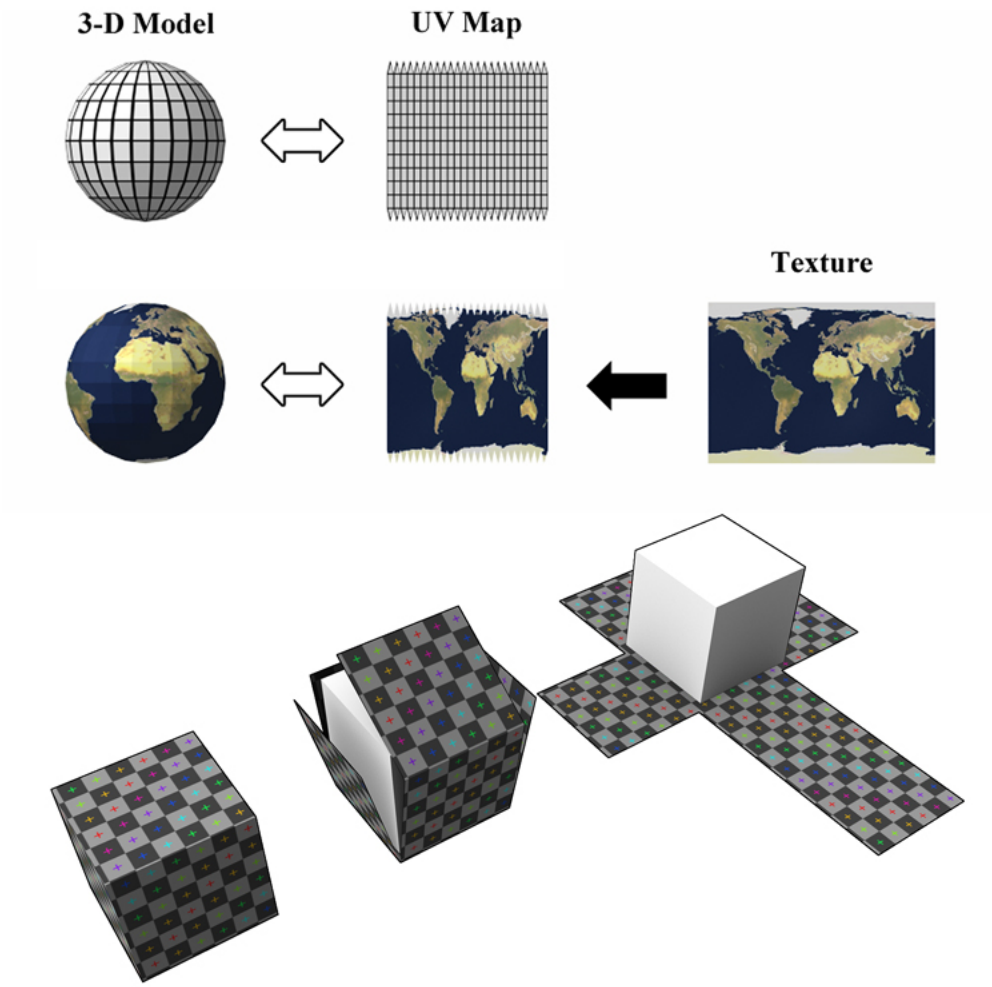


Figure 12.1: UV mapping. Image source: link

In the previous chapter, we specified colors for each vertex to create the multicolored triangle. And since we have to mention the texture coordinate for each vertex, its but natural to pair each vertex with a texture coordinate. Let us do that. Also, we can get rid of the color coordinates from the multicolored triangle because we dont want any coloration over our texture for now.

```
                        // vertex pos     texture coords
GLfloat vertices[] = { -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, // left bottom
                        0.5f, -0.5f, 0.0f, 1.0f, 0.0f,// right bottom
                        0.0f,  0.5f, 0.0f, 0.5f, 1.0f}; // top
```

Next, we need to create a texture object just like we create any other OpenGL object.

```
// textures
GLuint my_texture; // create a texture object; just like vao, vbo, ebo etc
glGenTextures(1, &my_texture); // generate textures
glBindTexture(GL_TEXTURE_2D, my_texture); // Bind texture; all usual procedure
```

Next we need to specify the wrapping and interpolation type.

12.1.1 Texture Wrapping

A texture can be pasted on a 3D model in many ways. Some of them are as shown below

1. **GL_REPEAT**: This is the default setting. In this mode the texture is repeated along both x and y axis.
2. **GL_MIRRORED_REPEAT**: same as the above mode, except, every time it repeats, it flips the image along the axis its repeating.
3. **GL_CLAMP_TO_EDGE**: In this mode, the edges of the texture gets stretched all the way till the border.
4. **GL_CLAMP_TO_BORDER**: Coordinates outside the texture are assigned a color value.

Figure 12.2: Various texture wrapping modes in OpenGL (image source: link)

12.1.2 Texture Interpolation/Filtering

A texture pixel is called *texel*. Often, the texture is either smaller or larger than the 3D model under consideration. Suppose the texture is smaller, the image has to be stretched to be wrapped around the model. In such a case the image size should be scaled or some form of affine transformation has to take place. There are many options to fill the generated holes. Most common ones include *GL_NEAREST* and *GL_LINEAR*. *GL_NEAREST* assigns the value of nearest neighbouring pixel to the void. In case of *GL_LINEAR* a linear interpolation is performed as discussed in section 4.1.

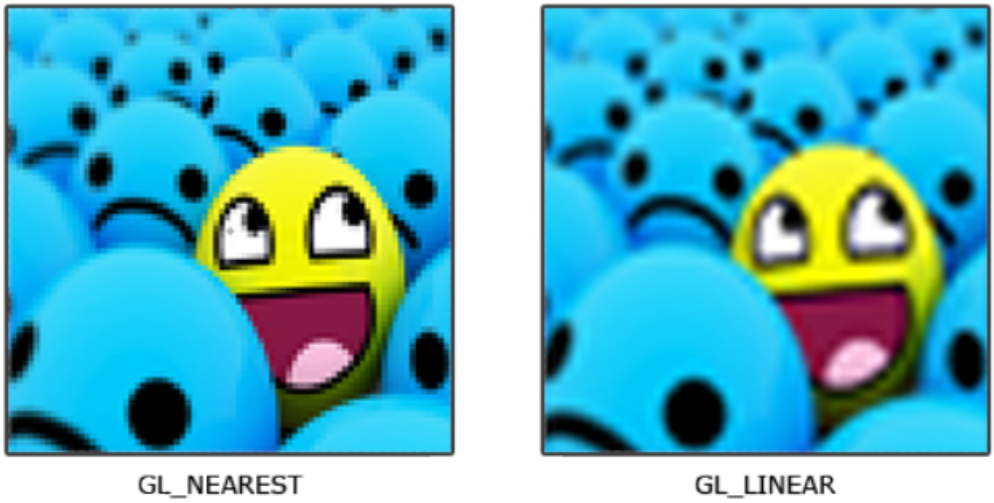
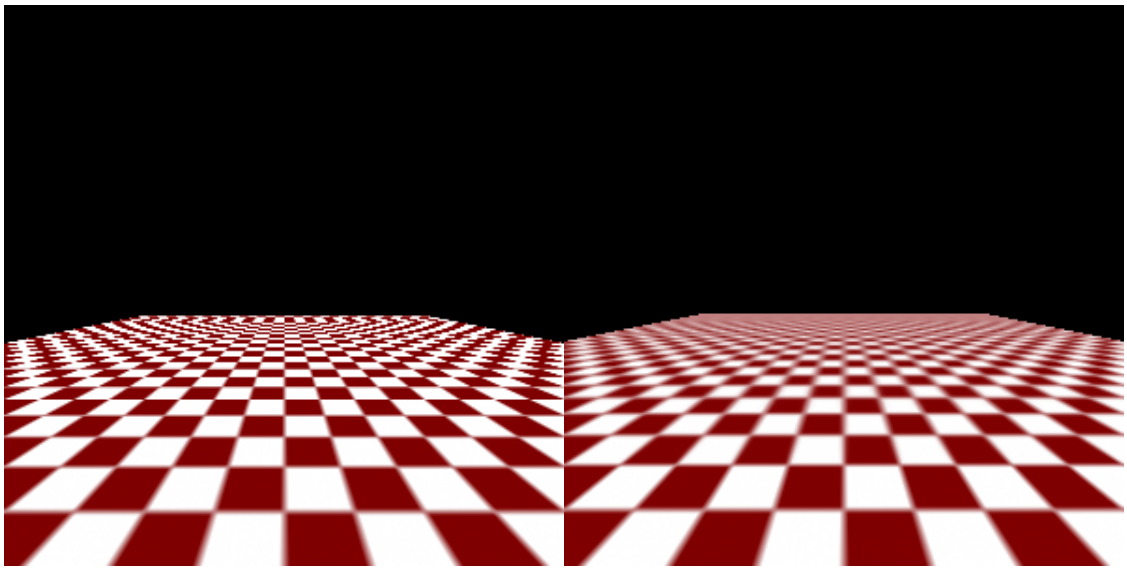


Figure 12.3: Difference between *GL_NEAREST* and *GL_LINEAR*. Image source: link

12.1.3 Mipmaps

When a pattern is repeated as shown in figure 12.4(a), the texture used creates aberrations due to filtering. This is clearly noticeable as depth increases. Each image must be downsized to avoid this problem. This is called mipmapping. The same figure with mipmapping does not create those aberrations.



(a) (b)

Figure 12.4: (a) without mipmapping, (b) with mipmapping. image source: link

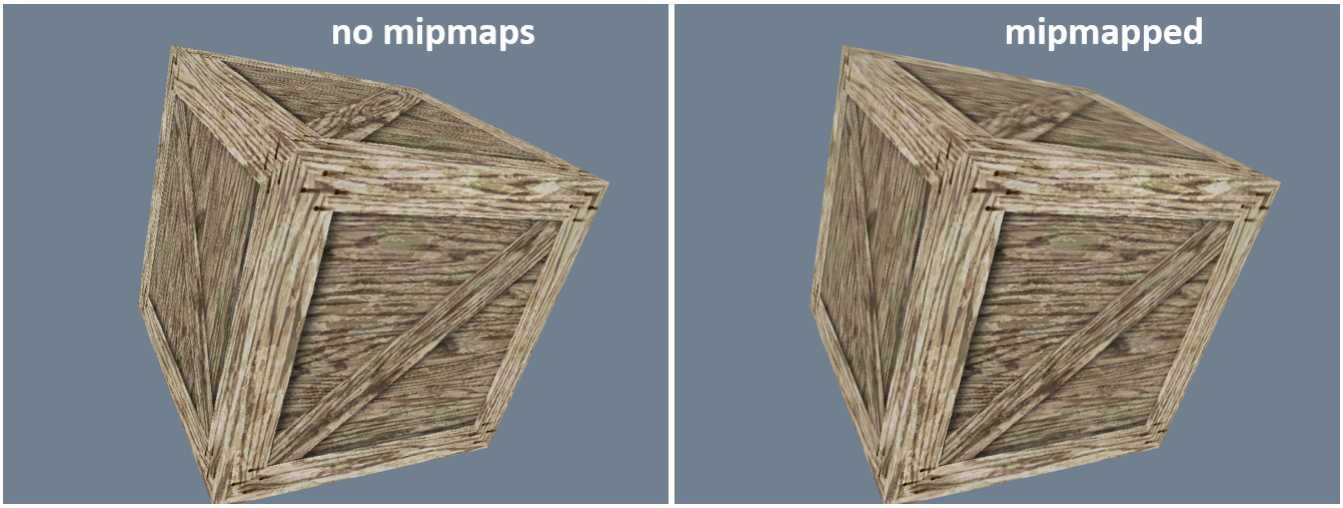


Figure 12.5: Another mipmap vs no mipmap example. image source: link

What does mipmapping do exactly? Given an image, it downsizes the image by several orders and as the distance between the camera and the object location increases a suitable downsized version of the texture is applied. Figure 12.6 illustrates the phenomenon.

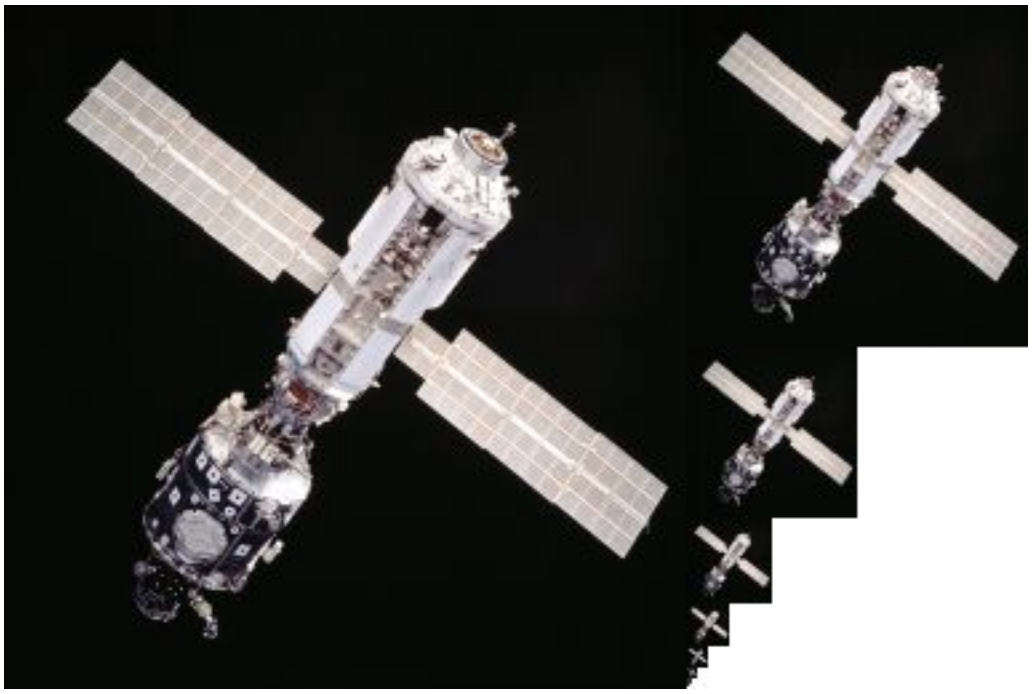


Figure 12.6: filtering down to various sizes during mipmapping. image source: link

To get OpenGL to enable mipmap on the texture we just loaded, we insert the following line.

```
    glGenerateMipmap(GL_TEXTURE_2D);
```

And thats just it. OpenGL automatically filters the loaded textures and picks out the appropriate downsized sample thats right for the job without the need for our intervention.

12.2 Shader code for textures

12.2.1 Vertex Shader

Textures are processed in the fragment shader. The *vertex shader* just has to pass the texture coordinates to the *fragment shader*. We are going to use the vertex and fragment shader codes from ch11/uniforms.

```
#version 330 core
layout (location = 0) in vec3 position; // vertex_position attribute
layout (location = 1) in vec2 tex_coords; // texture_coordinate attribute

out vec2 texCoords;

void main()
{
   gl_Position = vec4(position, 1.0);
   texCoords = tex_coords;
}
```

Line 1 and 2 are the same from the previous code. We add a new attribute named *tex_ coords* whih has two vector values. We also need to create *texCoords* to be passed to the fragment shader hence its an *out* variable. Finally on line 10 we assign the out variable with *tex_ coords*.

12.2.2 Fragment Shader

Next, we modify the fragment shader.

```
#version 330 core

out vec4 color;
in vec2 texCoords;

uniform sampler2D minion_texture; // texture is assigned to this uniform variable

void main()
{
   color = texture(minion_texture, texCoords); // assign texture for the fragment shader
       to process
}
```

We have a vec2 type for *texCoords* that we input from the vertex shader's output. Then we use a uniform of type *sampler2D* to read a texture. Then finally we assign the output *color* value with our texture.

12.3 Draw call

Finally, lets draw our triangle with a texture. We will be using *main.cpp* as our boiler plate code from ch11/uniforms.

```
// Binding our texture
glActiveTexture(GL_TEXTURE0); // We make the first texture active
glBindTexture(GL_TEXTURE_2D, minion_texture); // then we bind the texture
glUniform1i(glGetUniformLocation(our_shader.program, "minion_texture"), 0); // we look
    for the uniform variable in our shader program

/*************************************/
/******** issue draw calls ***********/
/*************************************/
```

You shoud get a nice fancy triangle with a funny samurai minion as shown in figure 12.7.

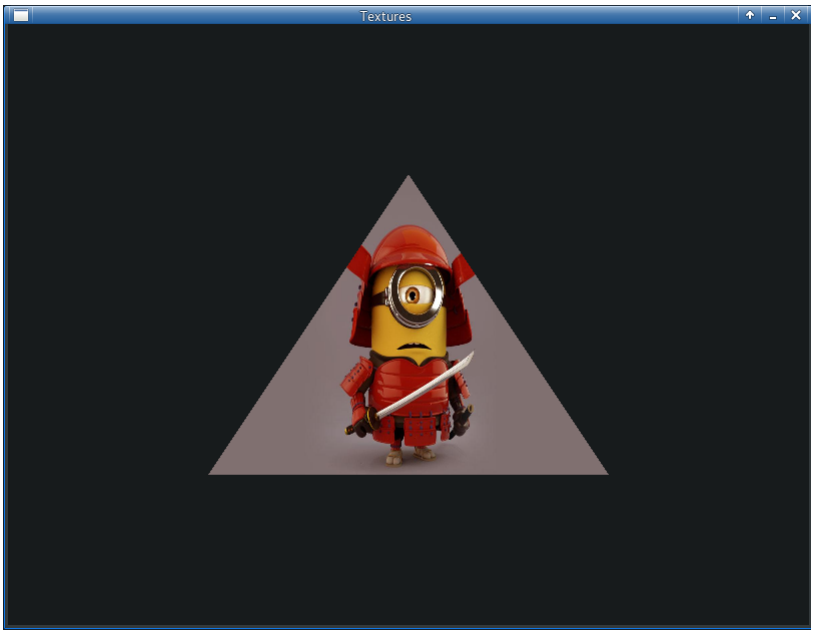


Figure 12.7: Textures with minions

If you were not able to compile the various parts of the code, take a peak at ch12/basic_textures

12.4 Mixture of Textures

Let us now try to mix two textures and overlay it on the same object. We will use the same boiler plate code in ch11/uniforms for this.

The creation and binding procedure for the new texture is the same. We add a new uniform in the *fragment_shader* for the wood texture. Up till this point, its the same code as discussed in the previous section. In addition to that, we want a nice smooth mix of textures.

```
color = mix(texture(wood_texture, texCoords), texture(arch_texture, texCoords), 0.3);
    // mix function takes a mixture of both the textures and 0.3 here is mix ratio
```

mix() function mixes the two textures. It also takes in a mix strength parameter that decides the strength or propotion of the mix. You should obtain an image similar to what is shown in figure 12.8.

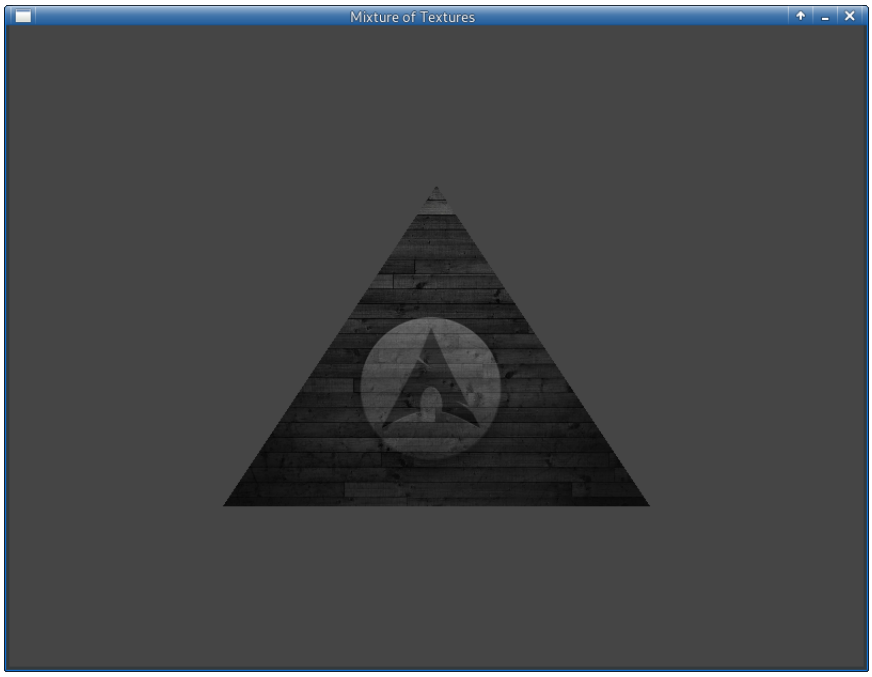


Figure 12.8

Visit ch12/mixture_of_textures for the complete source code.

Chapter 13

OpenGL Coordinate Systems and Projections

OpenGL has a number of coordinate systems. In this chapter, we will be exploring some of the coordinate systems that will help us segregate transformations clearly into various coordinate systems/spaces.

13.1 Coordinate Systems or Spaces

Recap: We have already learnt transformations in chapter 4. We are going to be doing more of the same, but segregating the several transformations into various coordinate-systems/spaces. We will be discussing five spaces.

13.1.1 Object Space

This space consists of nothing but the concerned object alone. Imagine you are creating a game with several characters. While creating each character - designing the head, torso, legs and other little body parts, you wont be concerned with where or how the object is placed in the game world. All you will be concerned with is designing the object and the details of the object. For instance, you created the nose of the character which has to be placed a little down from and in between the eyebrow. There you need a local coordinate system on its own called the *Object Space*. The object space coordinate system will have an origin and coordinate system just like any other 3D spatial coordinate system. This space allows us to perform individual object manipulation like size and default orientation of the object when it is going to be placed in the game world.

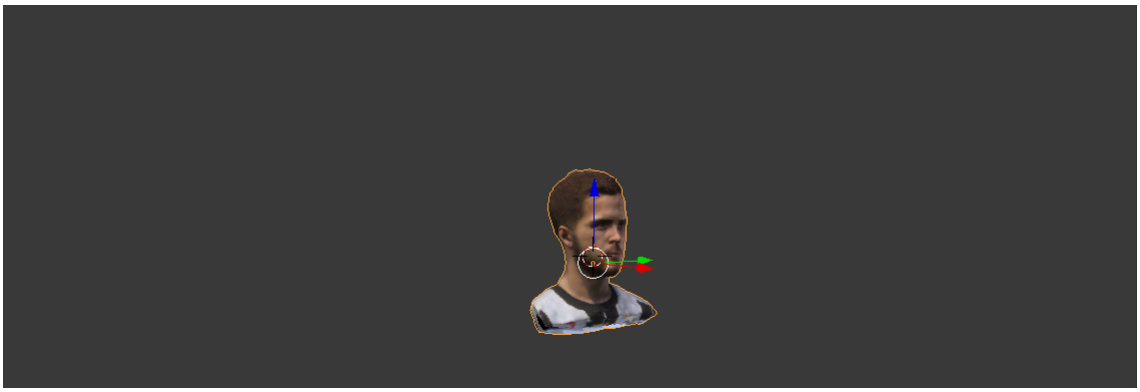


Figure 13.1: Object Space

13.1.2 World Space

OpenGL *World Space* refers to the game world. One can imagine it to be something like a God's view of the entire world or a birds eye view of the world. In this space all the coordinates are represented with respect to the world space coordinate system (world space axes and origin). Here, we do manipulations such as moving objects of about. We can develop spatial relations between objects of the game.

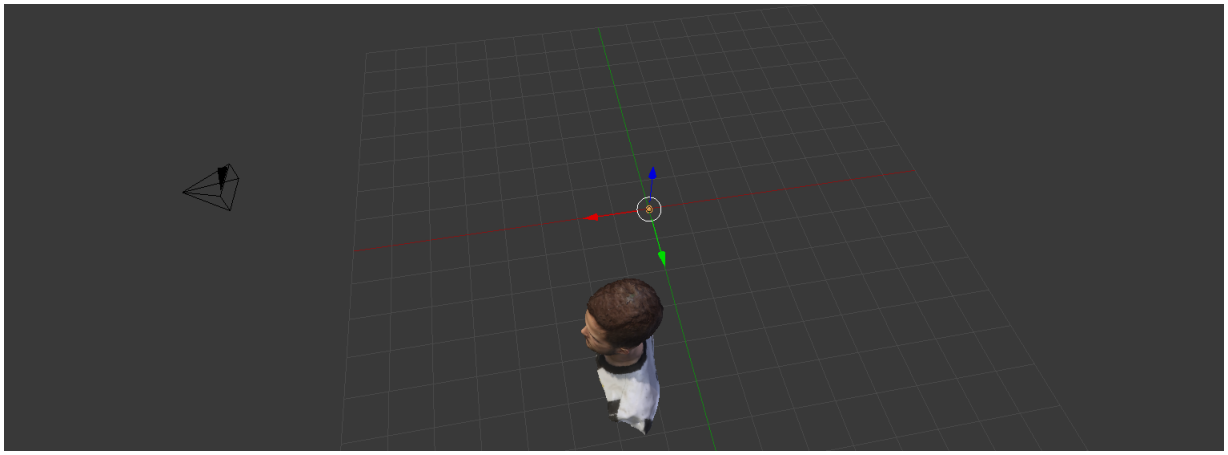


Figure 13.2: World Space

13.1.3 View Space

This space is analogous to a first person shooter view. OpenGL (or for that matter any 3D rendering platform) had a virtual camera. Through the camera view is how we see and perceive the objects of the world space. Thus, this is often also called the *eye-space*. The optical origin of the virtual camera is the view space origin. In this space, all coordinates are represented with respect to the camera origin. In our normal world, when we have to take the video of something, we move the camera around to capture the video. The funny reversal in the graphics world is, the camera remains static and we move the entire game world around in reverse to create the same effect.

"I understand how the engines work now. It came to me in a dream. The engines don't move the ship at all. The ship stays where it is and the engines move the universe around it." — Cubert Farnsworth

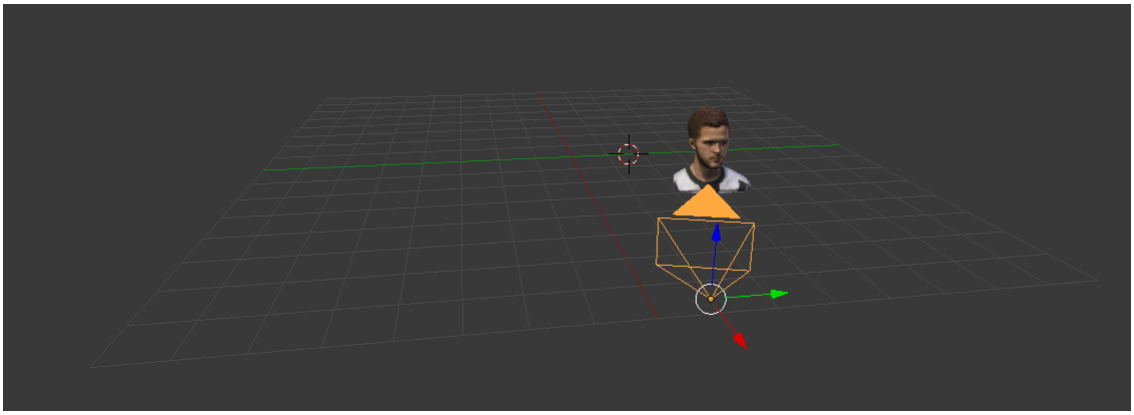


Figure 13.3: View Space

13.1.4 Clip Space

After vertex shader is processed, OpenGL clips the coordinates of all the objects. Any object lying outside the clip space is clipped. All the vertices lying outside the clip space are discarded. The others are passed on to the next stage. This space by default ranges between -1.0 and 1.0. So what we do is we work with larger coordinate numbers and then normalize the scale of the objects to -1 and 1.

What is enclosed by the clip space is called the *view frustum*. Imagine your eyes are the camera. When you stand somewhere you see the whole world in front of you. Ideally speaking, even objects at infinity. But, when we view through the OpenGL camera, rendering all objects till infinity is a bit of an impossible ordeal. So is rendering objects too close to the camera. Thus we tell OpenGL to render objects between a ***near plane*** and a ***far plane***.

There are basically two most common types of projections viz., ***Orthographic Projection*** and ***Perspective Projection***. Orthographic projection is a type of projection where distance does not affect the size of the object. One can encounter this type of projection in CAD softwares and architecture diagrams etc. It gives a realistic scaled size of all the objects. This, however is unnatural in real life. We want objects further away to be smaller in size. For instance, rail tracks should seem to converge. We discussed this briefly in section 4.1. Figure 13.4 illustrates the difference between orthographic and perspective projection.

(a) (b)

Figure 13.4: (a) orthographic projection of a cuboid, (b) perspective projection of a cuboid

13.1.5 Screen Space

All the objects still are represented in 3D. At each cycle of the game loop, the object should then be flattened to be projected onto the 2D screen. This coordinate space is called Screen Space.

Each of these transformations can be captured in the form of matrices. Figure 13.5 describes the various transformations from one space to another.

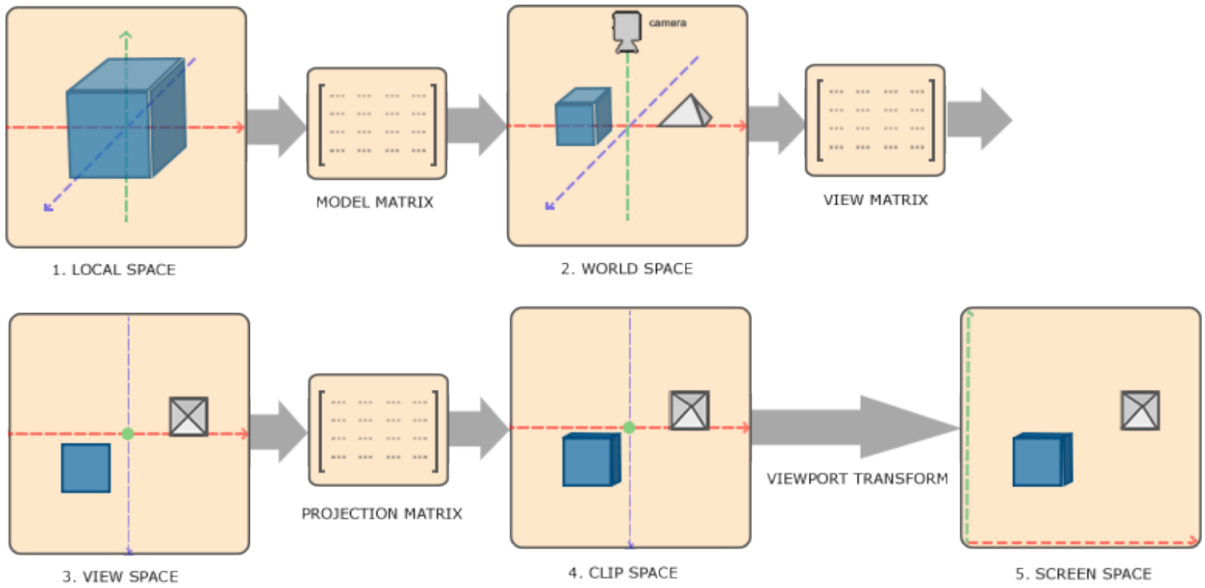


Figure 13.5: Transformation from one space to another. Image source: link

13.2 GLM - Math Library

In the previous section, we discussed we need matrices to transform us from one space to another. How exactly are we going to acheive that? With the help of OpenGL mathematics library called GLM. Visit https://glm.g-truc.net/0.9.9/index.html and checkout the GLM website. To install GLM on ubuntu, type the following on your terminal.

```
sudo apt-get install libglm-dev
```

GLM is a header only library and does not need linking. To add the glm headers insert the following header lines into the code. The documentation for the various functions and features can be found here. But we won't be using all the features. I have compiled some of the most commonly used ones below.

```
#include <iostream>

// GLM headers ------------------------
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
// ------------------------------------
```

The above three lines are the headers that packs all the GLM functions that we will be using.

```
int main()
{
   // let us create a vector of 3 elements ---------------------------------------
   glm::vec3 a(1.0f, 2.0f, 3.0f);
   std::cout << a[0] << "\t" << a[1] << "\t" << a[2] << std::endl;
```

```
// ------------------------------------------------------------------------
```

The above lines indicate how to initialize a vector of 3 elements.

```
// let us create a 4x4 array of matrices ----------------------------------
glm::mat4 my_first_matrix(0, 1, 2, 3,
                          4, 5, 6, 7,
                          8, 9, 10, 11,
                          12, 13, 14, 15);
// ------------------------------------------------------------------------

// access individual elements of the matrix -------------------------------
std::cout << "printing each element of the matrix ->" << std::endl;
std::cout << my_first_matrix[0][0] << std::endl;
// ------------------------------------------------------------------------
```

Next, we create a 4x4 matrix. We can access the individual elements of the matrix by doing the following.

```
// now if you want to assign a array of floats to a glm matrix -------------
float my_second_matrix_data[16] = {0, 1, 2, 3,
                          4, 5, 6, 7,
                          8, 9, 10, 11,
                          12, 13, 14, 15};
glm::mat4 my_second_matrix = glm::make_mat4(my_second_matrix_data);
// ------------------------------------------------------------------------
```

We often need to convert OpenCV's *cv::Mat* data structure to *glm::mat4* data structure. We can do the following to convert the data. *my_matrix.data* yields an array of floats/double or whatever datatype we assigned during the creation of cv::Mat.

```
// Multiply matrices ------------------------------------------------------
float my_third_matrix_data[16] = {15, 14, 13, 12,
                          11, 10, 9, 8,
                          7, 6, 5, 4,
                          3, 2, 1, 0};
glm::mat4 my_third_matrix = glm::make_mat4(my_third_matrix_data);
glm::mat4 multiplied_matrix = my_second_matrix * my_third_matrix;
// ------------------------------------------------------------------------

// print the multiplied matrix --------------------------------------------
std::cout << "printing the multiplied matrix ->" << std::endl;
for(int i=0; i<4; i++)
   for(int j=0; j<4; j++)
      std::cout << multiplied_matrix[i][j] << std::endl;
// ------------------------------------------------------------------------
```

This multiplies two matrices and prints each element. You can find the complete source code under ch13/glm_practice.

Remember, we learnt *intrinsics* and *extrinsics* in section 7.1 and 7.2. We have to somehow translate those intrinsics and extrinsics to model, view and projection matrices. That will be the main goal in the next chapter.

13.3 Lets Draw a Box

This section may come a bit abruptly. We are going to render a wooden crate with an insignia. We will then use this box to study the different transformations. For this part of the program we will be using ch12/mixture_of_textures as our boiler plate code. To draw a crate we need the list of vertices and indices that make up the crate. You can find it here. All the concepts used to draw a box is what we have learnt before so we wont be discussing the code in detail. Instead, we can just go through the steps involved.

1. Create a list of vertices and indices.
2. Create the buffers necessary i.e., VAO, VBO and EBO
3. Bind the buffers and describe the layout of the buffer data.
4. Unbind the VAO
5. Create two textures
6. In the while() loop
   (a) bind the two textures
   (b) draw the primitives using *glDrawElements()* draw call

Also appropriately modify the vertex and fragment shader. If you want to take a peek at the working code, take a look at ch13/box. The box will look similar to the rectangle you obtained in figure 11.1. This is because you are viewing only the front side of the box. Once the viewpoint is changed or the box is rotated you will be able to better see the box.

You should get an output similar to that of a rectangle as shown in figure 13.6.

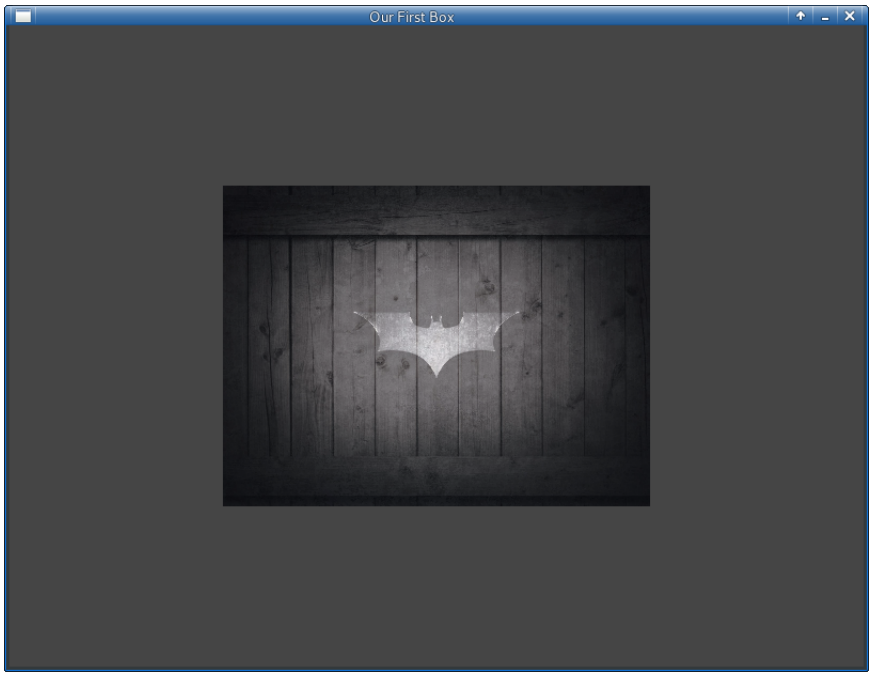


Figure 13.6: Figure of a wooden crate - front view

13.4 Model Matrix in Practice

In this section, we are play around with the wooden crate doing a lot of scaling, rotation and translations on it. We will be using matrices for the process. This section will be a combination of the concepts learnt in chapter 4 and section 13.1. If you have for some reason forgotten it or you are unfamiliar with those topics, now would be a good time to revist those, just to jog your memory.

13.4.1 Halve the Crate

The first transformation we will be learning is - *from model space to world space* using the model matrix to halve the crate size. Its a 4x4 matrix and with this we can control the default size and orientation when the object is placed in the world space. Ofcourse, we will be using ch13/box as our template code.

In the main.cpp file, let us create a matrix call the *half_ size_ matrix*.

```
glm::mat4 half_size_matrix = {0.5f, 0.0f, 0.0f, 0.0f,
                      0.0f, 0.5f, 0.0f, 0.0f,
                      0.0f, 0.0f, 0.5f, 0.0f,
                      0.0f, 0.0f, 0.0f, 1.0f};
```

The first three elements of the leading diagonal of the matrix is made 0.5 which reduces the sizes of the box along the x, y and z axes (as discussed in chapter 4). We then transfer the matrix to the GPU memory.

```
// find the location of the matrix in the shader
GLuint half_size_location = glGetUniformLocation(our_shader.program, "half_size");
glUniformMatrix4fv(half_size_location, 1, GL_FALSE, glm::value_ptr(half_size_matrix));
```

In the vertex shader code, we create a uniform to host our new half_size matrix. Then we multiply the matrix with our vertex position vector.

```
uniform mat4 half_size;

void main()
{
   gl_Position = half_size * vec4(position, 1.0);
   /***********************************************/
   /****** remainder of the vertex shader code ******/
   /***********************************************/
```

If you managed to get it right the crate size should have reduced by a factor of 0.5. For the complete source code visit ch13/halve_the_crate.

13.4.2 Rotate the Crate

Let us do something more interesting now. Let us try to rotate the crate. There are two way we can achieve this. The first way is to rotate using the model matrix. As we learnt before, this sets the orientation at the time of transfer from object space to world space. We can choose to insert this in the while loop so that an illusion of a continuously spinning crate can be achieved. The second way is to do the operation using the view matrix which we will discuss shortly.
RECAP:

$$Rotation\ around\ x\ axis,\ R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & \cos\theta & -\sin\theta & t_y \\ 0 & \sin\theta & \cos\theta & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where t_x, t_y and t_z denotes the translations along x, y and z.

Let us create the rotation matrix. Since we want the box to continuously rotate, we will do it inside our *while()* loop.

```
float time = glfwGetTime(); // gets the time since the start of execution

// lets create the rotation matrix. Lets use time to create the rotation matrix
glm::mat4 rotation_matrix = {1.0f, 0.0f, 0.0f, 0.0f,
                    0.0f, cos(time), -sin(time), 0.0f,
                    0.0f, sin(time), cos(time), 0.0f,
                    0.0f, 0.0f, 0.0f, 1.0f};

// To find the rotation matrix in from the shader
GLuint rotation_matrix_location = glGetUniformLocation(our_shader.program, "rotation");
glUniformMatrix4fv(rotation_matrix_location, 1, GL_FALSE,
    glm::value_ptr(rotation_matrix));
```

Now the modification in the vertex shader can be done as follows.

```
uniform mat4 rotation;

void main()
{
   gl_Position = rotation * half_size * vec4(position, 1.0);
   /************************************************/
   /****** remainder of the vertex shader code ******/
   /************************************************/
```

If up till here you did everything correctly, you should get a weirdly rotating window. Something like what is shown in the video.

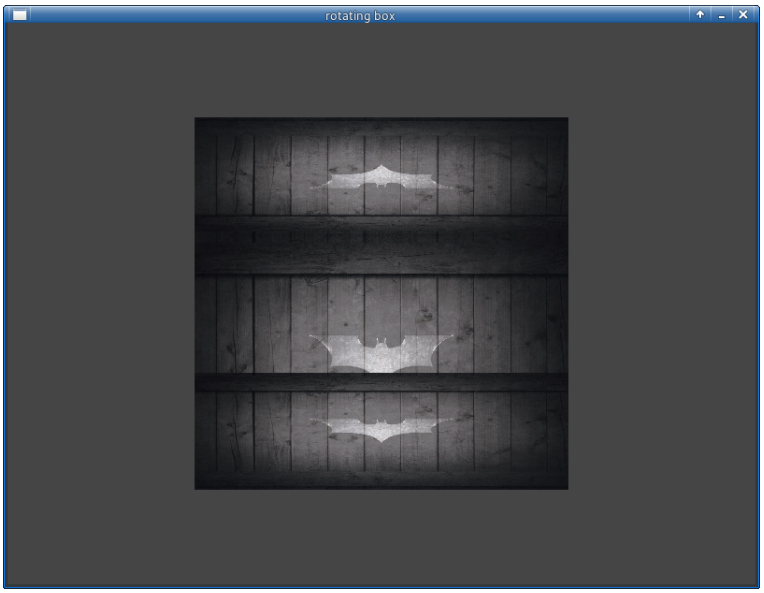


Figure 13.7: Box without depth enabled

This weirdness is because we did not enable depth in OpenGL. By default OpenGL does not enable the depth buffer. To enable the depth buffer, we need to clear the depth buffer bit and then call *glEnable(GL_DEPTH_TEST)*.
Modify *glClear()* function as follows.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/************************************************/
/****************** other code *****************/
/************************************************/

// To enable depth call this function outside while()
```

```
glEnable(GL_DEPTH_TEST);
```

Now the box should look somewhat normal as in the following video. The code for the following can be found under ch13/rotate_the_crate.

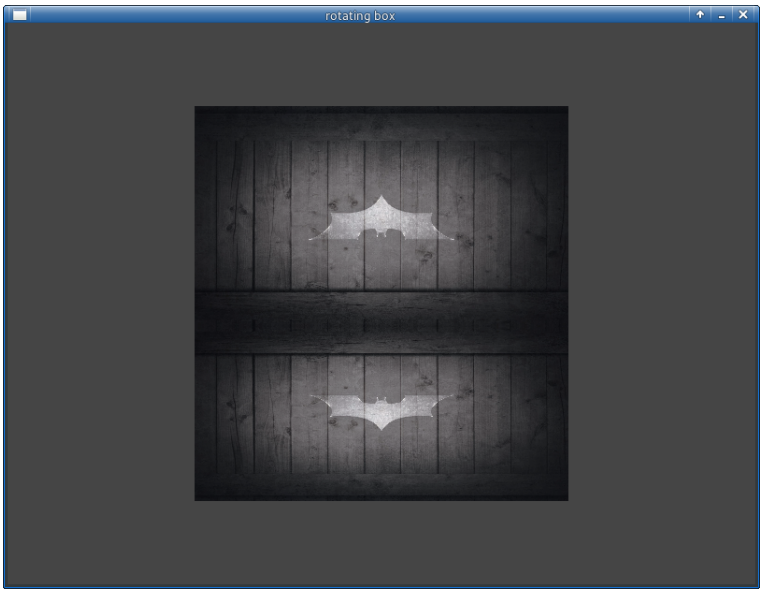


Figure 13.8: With depth buffer enabled in OpenGL

In the above sections, we actually created the *half_size_matrix* and the *rotation_matrix* manually. OpenGL actually provides inbuilt methods to create this matrix automatically.

```
// to create a scaling matrix one can use
glm::mat4 scaling_matrix = glm::scale(0.5f, 0.5f, 0.5f); // parameters are -> scale
    factors along x, y and z axis

// to create a rotation matrix
glm::mat4 rotation_matrix; // this will create an identity matrix
rotation_matrix = glm::rotate(rotation_matrix, angle, glm::vec3(0.5f, 1.0f, 0.0f)); //
    parameters are -> rotation_matrix, angle of rotation, axis around which the object
    should be rotated
```

But we intentionally did not opt this method of initializing matrices because we have to translate the *intrinsics* and *extrinsics* into OpenGL matrices.

If you notice carefully, the box still does not look all that perfect. There is still something wrong with it. This because of the default orthographic projection which we will delve into in subsequent sections.

13.5 View Matrix in Practice

The main purpose of the view matrix is to translate the origin and the entire coordinate system to the camera center. To do this we need to know the camera's position in world space, and its orientation. Camera's Position in world space can be obtained by its displacement along x, y and z axes from the world

space origin. To define the orientation of the camera, we need to define the ***direction the camera*** is pointing at (strictly speaking its opposite of this direction), the ***Up*** direction of the camera, and the ***Right*** direction.

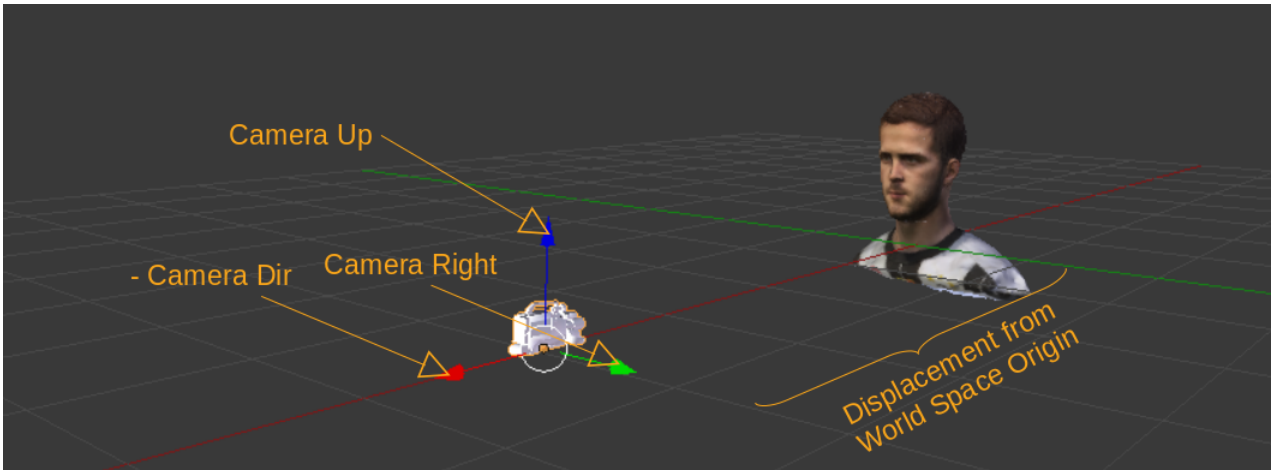


Figure 13.9: Camera Up, Right and -(camera direction)

With these we can define what is known as the ***lookAt*** matrix which is another fancy term for the *view matrix*. A view matrix is defined as

$$View\ Matrix\ = \begin{bmatrix} R_x & R_y & R_z & -C_x \\ U_x & U_y & U_z & -C_y \\ D_x & D_y & D_z & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13.1}$$

where, (R_x, R_y, R_z), (U_x, U_y, U_z) and (D_x, D_y, D_z) are vectors denoting ***Camera Right***, ***Camera Up*** and negative of ***Camera Direction*** respectively. But ***how do we actually calculate these vectors?*** In generic cases, we need to know some parametes to calculate these vectors. For our scenario, let us take a simplistic scenario. Look at figure 13.9. The object is at the origin and the camera is on the xz plane. Furthermore, the camera is always facing the world space origin.

1. ***Camera Direction:*** This is the difference between the camera position and the origin i.e., $(C_x, C_y, C_z) - (0, 0, 0) = (C_x, C_y, C_z)$. Note: the negative of the camera direction is represented in the view matrix equation (eq 13.1) itself.

2. ***Camera Up:*** Since the camera is on the xz plane, the UP direction is simply $(0, 1, 0)$.

3. ***Camera Right:***To calculate the camera right we use the *Gram Schmidt* process. Its a very rudimentary and simple method where we use the cross product of the UP and the *Direction* vectors to calculate the *Right* vector.

4. ***Camera Position:*** Camera Position is the distance of the camera from the world space origin.

We can now populate the above View matrix equation (eq 13.1). But calculating all this just to get the view matrix is ridiculuous right? Ofcourse, OpenGL provides an inbuilt method called *lookAt()*. We just have to feed in the position, direction and Up vector and it calculates the view matrix for us. Lets look at an example of this.

```
glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 5.0f), glm::vec3(0.0f, 0.0f, 0.0f),
    glm::vec3(0.0f, 1.0f, 0.0f)); // parameters are -> position, direction of camera
    point, camera Up vector
```

We modify the vertex shader code with a uniform named view.

```
uniform mat4 view;
int main{}
{
   gl_Position = view * model * vec4(position, 1.0);
   /************************************************/
   /****** remainder of the vertex shader code ******/
   /************************************************/
```

When we dealt with *rotation of crate*, OpenGL actually setup a default view and projection matrix. In our case we are explicitly defining the view and projection matrices. Without defining both of them, OpenGL will not be able to render any output. Thus we will obtain any visible output only in the next section.

13.6 Projection Matrices in Practice

In the rotating crate example, something was not quite right. If you view the box carefully, you might notice that the distance between vertices further away from the camera did not seem smaller that the distance between vertices close by. Controlling this is the main function of a projection matrix. Also OpenGL cannot render literally every thing in the game world all the time. If that were the case, it would require ridiculuous processing power. What OpenGL cleverly does is, it renders everything withing the ***view frustum***. Imagine view frustum to be a box (or a pyramid - we ll see later) right in front of the OpenGL camera. Everything that falls within the box is rendered. The box ofcourse moves with the camera.

This box is called the view frustum. There are most commonly two types of projections 1) orthographic and 2) perspective.

13.6.1 Orthographic Projection

Orthographic Projection has a view frustum in the shape of a cuboid just as shown in figure 13.10. Orthographic projection is a very simplistic form of projection where the object near or far within the view frustum is of the same size. The effect of it was shown in figure 13.4(a). This is popularly used in Computer Aided Design softwares such as Solid Works, Cadian etc. It is used to convey accurate and realistic sizes of objects when architecting something like perhaps an engine design or a house.

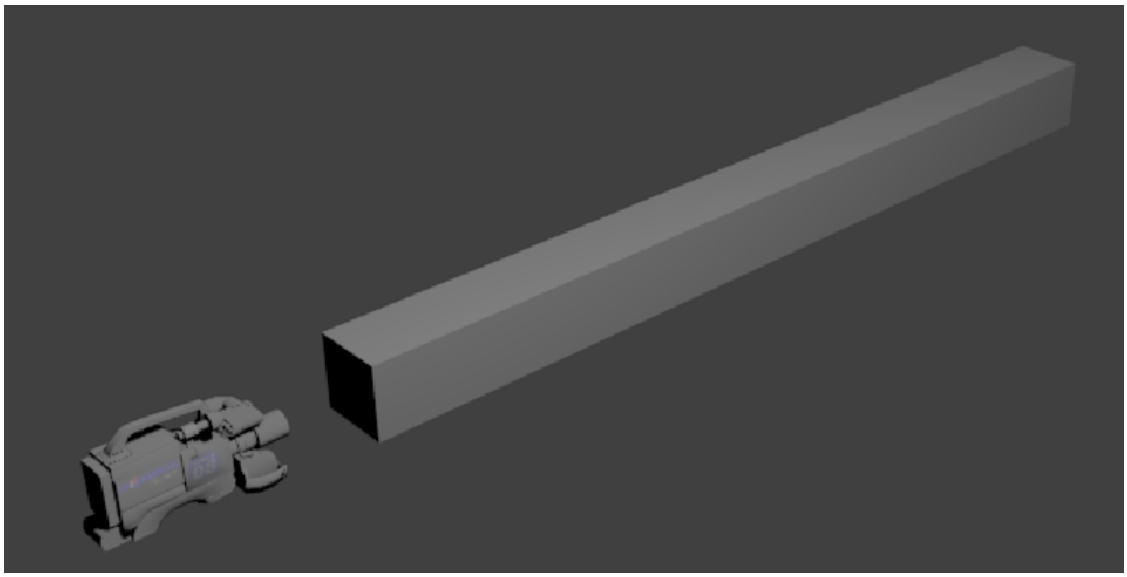
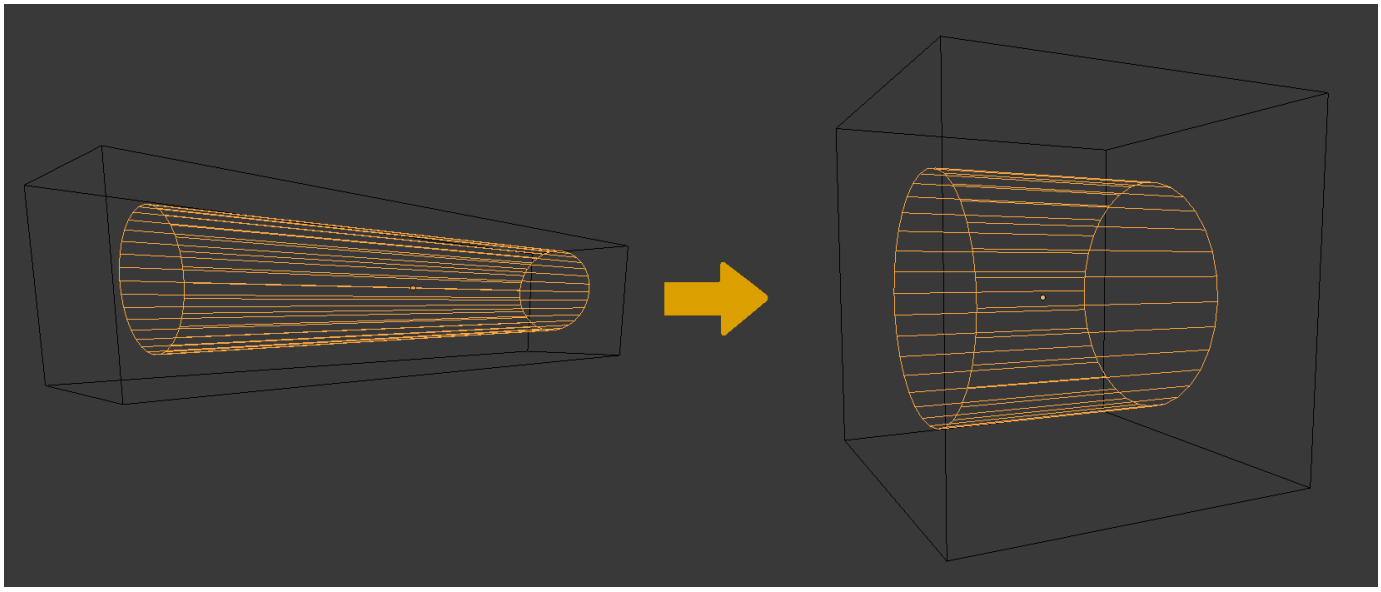


Figure 13.10: Illustration of a View Frustum

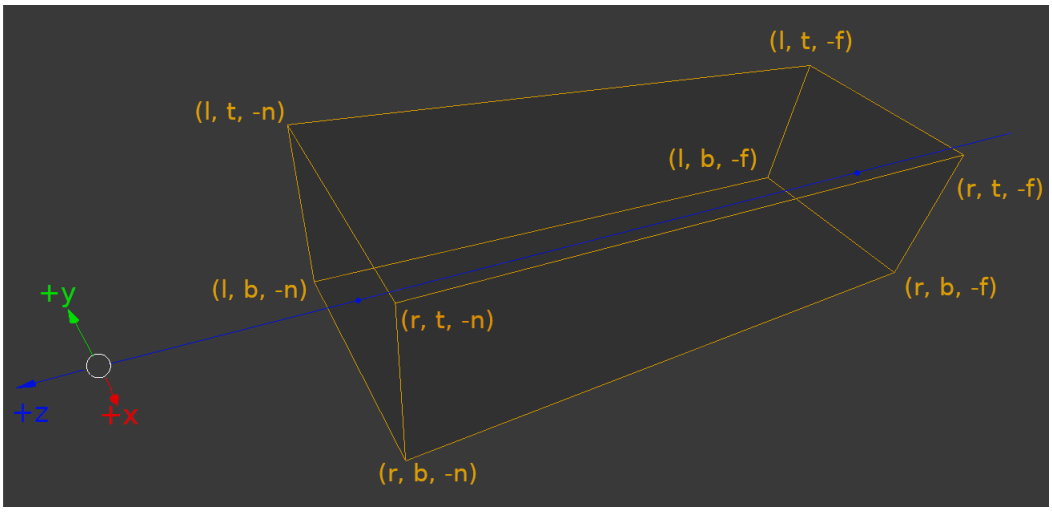
To describe an orthographic projection to OpenGL, we have to describe a cuboid. Cuboid has six planes viz., left, right, top, bottom, near (plane close to the camera) and far (plane far away from the camera).

In OpenGL, the orthographic projection translates the 3D point from a cuboid shaped view frustum (eye space) to a cube shaped view frustum (Normalized Device Coordiantes - NDC) as shown in figure 13.11. NDC has all the vertices scaled down and normalized to have a range between $[-1, 1]$ in all the three directions as shown in figure 13.12 (b). In addition, do notice, the coordinates change from ***Right hand coordinate system*** in the cuboidal view frustum to ***Left hand coordinate system*** on NDC. I don't think (as far as I know) there is any logical reason for this move. My guess is (merely a speculation), the team that designed modules that govern post eye-space projection did not communicate well with the folks that designed pre eye-space projection and they had an 'OOPS!' moment. So they decided to wing it by fixing it at the projection matrix.

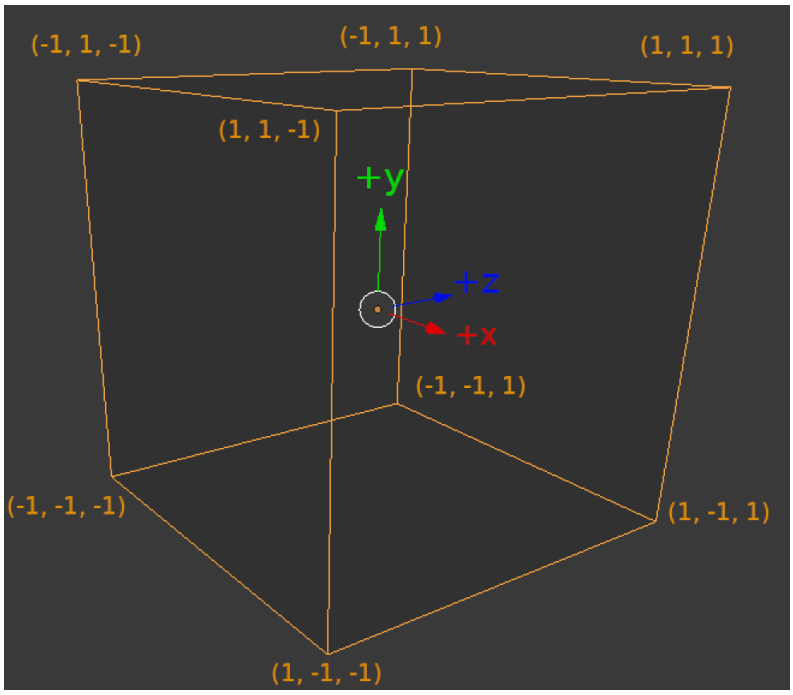


(a)

Figure 13.11: Effect of orthographic transform



(a)



(b)

Figure 13.12: (a) Cuboid View Frustum, (b) NDC

Conversion from eye-space coordinates to NDC happens by multiplying with the orthographic projection matrix, M_{ortho}

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ 1 \end{bmatrix} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}_{M_{ortho}} * \begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ 1 \end{bmatrix} \tag{13.2}$$

Let us derive the transformation matrix for orthographic projection. We start with an base identity matrix. We will modify the matrix row by row. Consider any point, (x_a, y_a, z_a) within the cuboid view frustum. Let us try to derive the first row of M_{ortho}.

$$l \leq x_a \leq r$$

$$0 \leq x_a - l \leq r - l$$

$$0 \leq \frac{x_a - l}{r - l} \leq 1$$

$$0 \leq 2 * \frac{x_a - l}{r - l} \leq 2$$

$$-1 \leq 2 * \frac{x_a - l}{r - l} - 1 \leq 1$$

Simplyfing we get,

$$-1 \leq \frac{2x_a}{r - l} - \frac{r + l}{r - l} \leq 1$$

We need to encode this into the first row of M_{ortho} matrix in equation 13.2 as

$$[x_{0,0}, x_{0,1}, x_{0,2}, x_{0,3}] = [\frac{2}{r - l}, 0, 0, -\frac{r + l}{r - l}] \tag{13.3}$$

y_a is used to derive the second row of M_{ortho}

$$b \leq y_a \leq t$$

Performing similar steps to above, we get,

$$-1 \leq \frac{2y_a}{t - b} - \frac{t + b}{t - b} \leq 1$$

Encoding this into the second row of M_{ortho} we get,

$$[x_{1,0}, x_{1,1}, x_{1,2}, x_{1,3}] = [0, \frac{2}{t - b}, 0, -\frac{t + b}{t - b}] \tag{13.4}$$

Similarly, we compute the third row of M_{ortho}

$$n \leq z_a \leq f$$

$$-1 \leq \frac{-2z_a}{f - n} - \frac{f + n}{f - n} \leq 1$$

Encoding this into the third row of M_{ortho} we get,

$$[x_{2,0}, x_{2,1}, x_{2,2}, x_{2,3}] = [0, 0, \frac{-2}{f - n}, -\frac{f + n}{f - n}] \tag{13.5}$$

The last row is inconsequential and remains as it is from the identity matrix. Thus the complete orthographic projection matrix is

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}_{M_{ortho}} \tag{13.6}$$

Let us code this up. We can assume the values of the cuboid view frustum as $[l, r, b, t, n, f] = [-1.5f, 1.5f, 1.5f, -1.5f, 0.1f, 100.0f]$. These values are arbitary and I chose them randomly.

Do note that OpenGL reads matrices in *column major* order. This means the matrices are read as shown in figure 13.13.

Thus, the obtained projection matrix has to be transposed before usage.

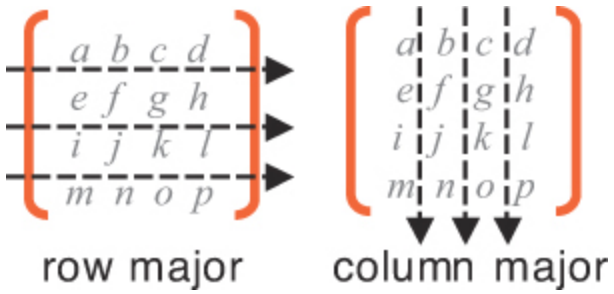


Figure 13.13: row major and column major. Image source : link

```
GLfloat l = -1.5f, r = 1.5f;
GLfloat b = -1.5f, t = 1.5f;
GLfloat n = 0.1f, f = 100.0f;

// defining the projection matrix
glm::mat4 projection_orthographic = {2/(r-l), 0, 0, -(r+l)/(r-l),
                          0, 2/(t-b), 0, -(t+b)/(t-b),
                          0, 0, -2/(f-n), -(f+n)/(f-n),
                          0, 0, 0, 1};

// OpenGL reads matrices in column major order
projection_orthographic = glm::transpose(projection_orthographic);

glUniformMatrix4fv(glGetUniformLocation(our_shader.program, "projection_orthographic"),
    1, GL_FALSE, glm::value_ptr(projection_orthographic)); // we need to locate the
    uniform 'projection_orthographic' ofcourse.
```

Let us add an orthographic projection matrix to our vertex shader as well

```
uniform mat4 projection_orthographic;
int main{}
{
   gl_Position = projection_orthographic * view * model * vec4(position, 1.0);
   /************************************************/
   /****** remainder of the vertex shader code ******/
   /************************************************/
```

You should obtain a box as shown in figure 13.14. You can find the code for this in ch13/orthographic_projection. You can find a video of the sample implementation here.

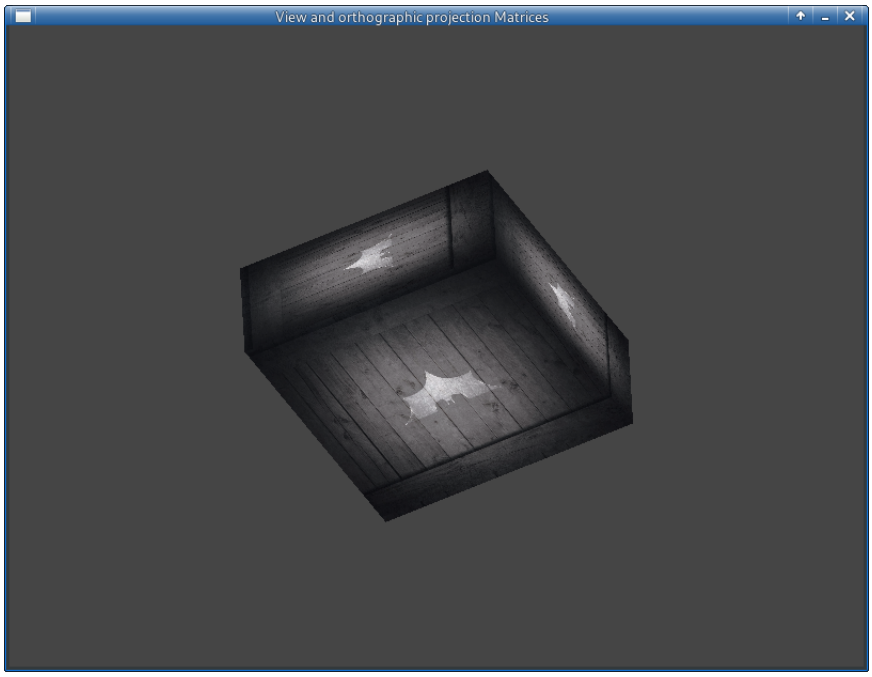


Figure 13.14: Orthographic Projection of the rotating crate

GLM provides a method to construct the orthographic projection matrix without having to manually construct it yourself.

```
glm::mat4 projection_orthographic = glm::ortho(left, right, top, bottom, near, far);
```

13.6.2 Perspective Projection

The above projection looks a bit unnatural because it lacks a sense of depth. A perspective projection does that job exactly. We discussed this for planar structures in chapter 4. We are going to do more of the same in this chapter. What if we took a pyramid shaped volume and squished it into a cuboid like in figure 13.15? A cylindrical object within the view frustum gets squished into a conical shape.

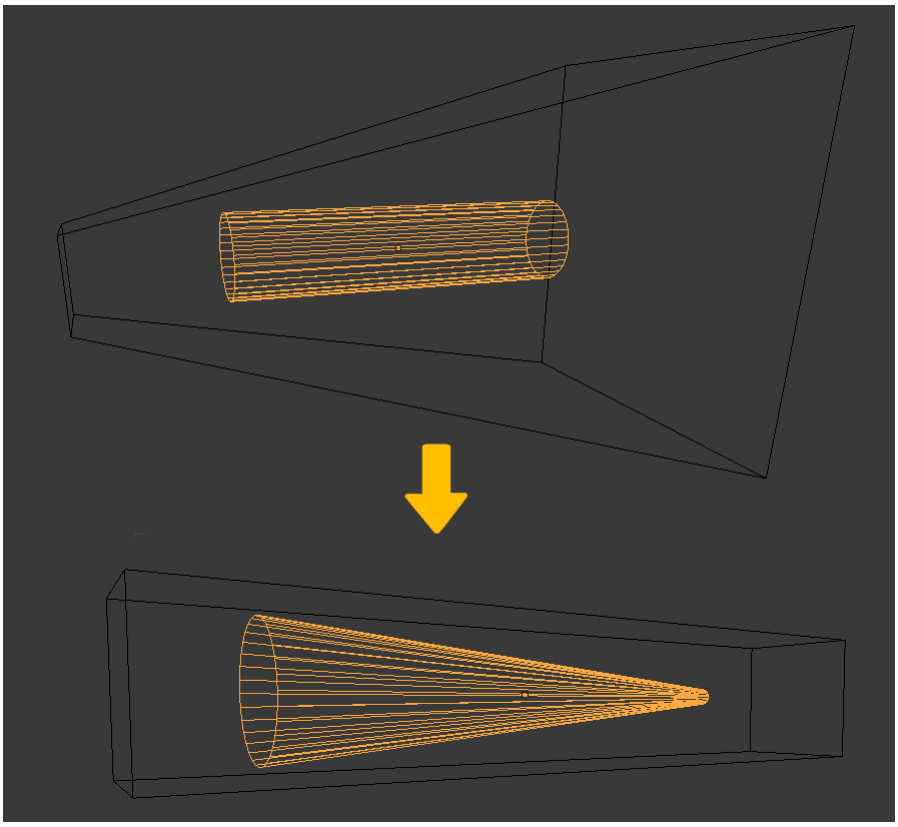


Figure 13.15: Effect of perspective transform

In addition to this, the cuboidal view frustum is further scaled to $[1, -1]$ as shown in the figure 13.16 to make it conform to NDC just as in the orthographic projection.

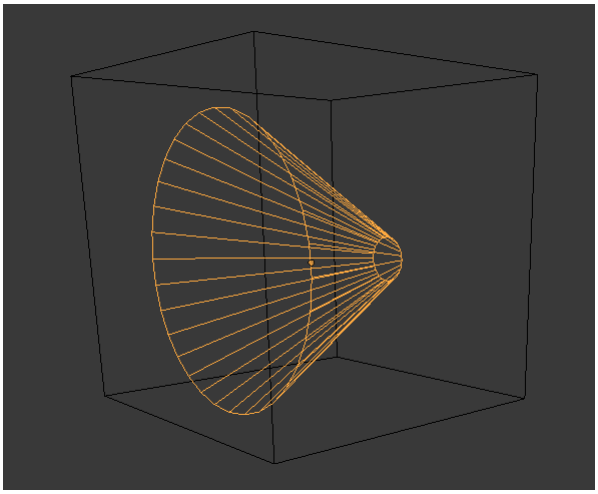
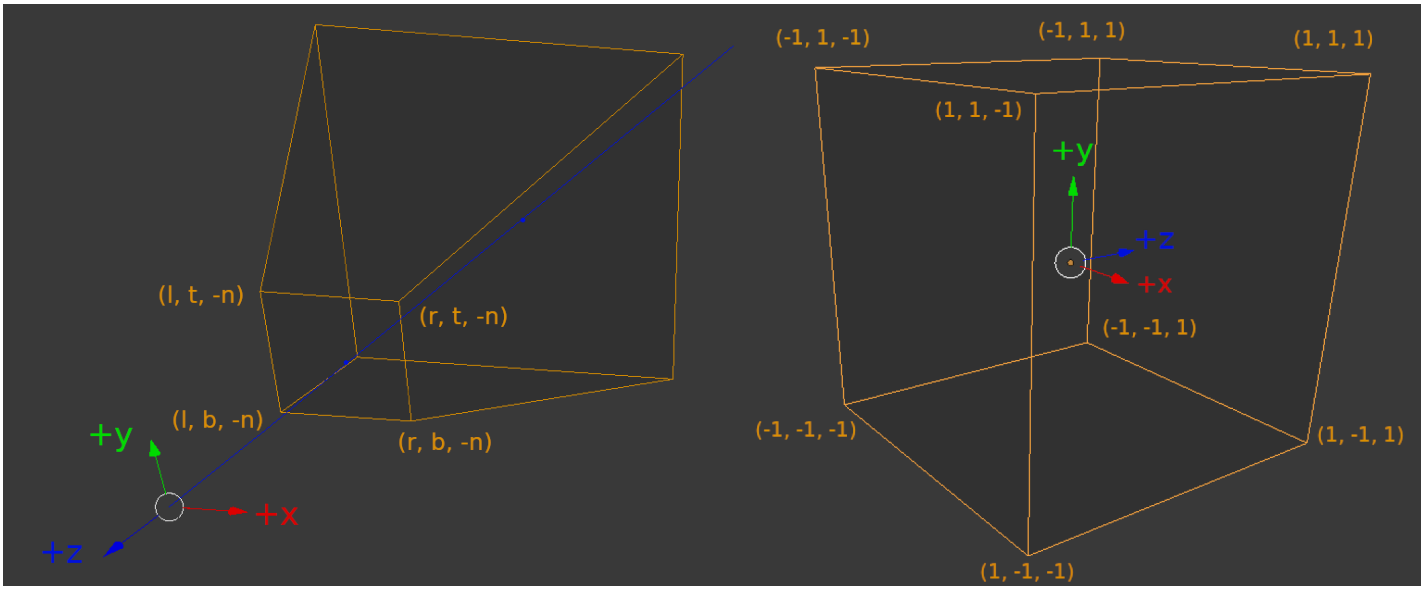


Figure 13.16: Clip to Normalized Device Coordinates

This is an illustration of what happens during a perspective divide. Our job is to convert the pyramidal shaped view frustum to NDC view frustum. Explicitly, we have to map the location of any point inside the pyramidal shaped view frustum to the NDC view frustum.



(a) (b)

Figure 13.17: (a) Pyramid View Frustum, (b) NDC

Again, do notice the coordinates change from ***Right hand coordinate system*** in the pyramidal view frustum to ***Left hand coordinate system*** on NDC.

Let us derive the perspective projection matrix. The whole process of translating from eye-space (pyramid view frustum) to NDC happens in 2 stages. The first stage is multiplying the each eye-space coordinate with the perspective projection matrix, M_{persp} to obtain the clip-coordinates. The second stage is each value of the clip coordinate has to be divided by w_{clip} to obtain the NDC coordinates.

Stage 1: To obtain *clip coordinates*

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}_{M_{persp}} * \begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ 1 \end{bmatrix} \tag{13.7}$$

Stage 2: To obtain NDC coordinates

$$\begin{bmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{bmatrix} = \begin{bmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{bmatrix}$$

Our job is to find out the values of $x_{0,0}$, $x_{0,1}$, ... $x_{3,3}$. To do that there is a small linear algebra idea/concept you have to get acquainted with.

Concept: How do you guess what numbers go into each value of $x_{i,j}$ in M_{persp}? Well lets take a small example where we have to perform a similar operation,

$$\begin{bmatrix} a \\ a \end{bmatrix} = \begin{bmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{bmatrix} * \begin{bmatrix} a \\ 0 \end{bmatrix}$$

Looking at this equation, we should be able to set the values of $[x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}] = [1, DC, 1, DC]$ without actually solving the matrix (DC refers 'Don't Care what the value is; it can be anything'). The argument is, since we want to take a from R.H.S. vector and make it appear on the on the bottom at L.H.S., we make $x_{1,0}$ as 1. Because, that value uses the top value from the vector in R.H.S., and affects the bottom

row of L.H.S.. That was the tricky number. Using this logic, we can judge the values of the other numbers too. I hope you can see it. Anyway, we can also choose to compute the values of the matrix.

$$x_{0,0} * a = a \ and,$$

$$x_{1,0} * a = a$$

$$x_{0,0} = 1 \ and,$$

$$x_{1,0} = 1$$

values of $x_{0,1}$ and $x_{1,1}$ are inconsequential and can have any value as they get multiplied by 0.

Back to our scenario of deriving M_{persp}. For reasons such as depth test which is performed after the fragment shader, we somehow need to preserve the value of z_{eye} and we also need it as $-z_{eye}$ because of the conversion from *Right Hand Coordinate system* to *Left Hand Coordinate system*. To do that, we set the value of $x_{3,2}$ to -1. We do not need to preserve any other vector value on the R.H.S. (eye coordinate). Thus, the last row of M_{persp} is

$$[x_{3,0}, x_{3,1}, x_{3,2}, x_{3,3}] = [0,\ 0,\ -1,\ 0] \tag{13.8}$$

Next, let us focus on the first row of M_{persp}. Consider the top view and side views of the pyramidal view frustum in figure 13.18.

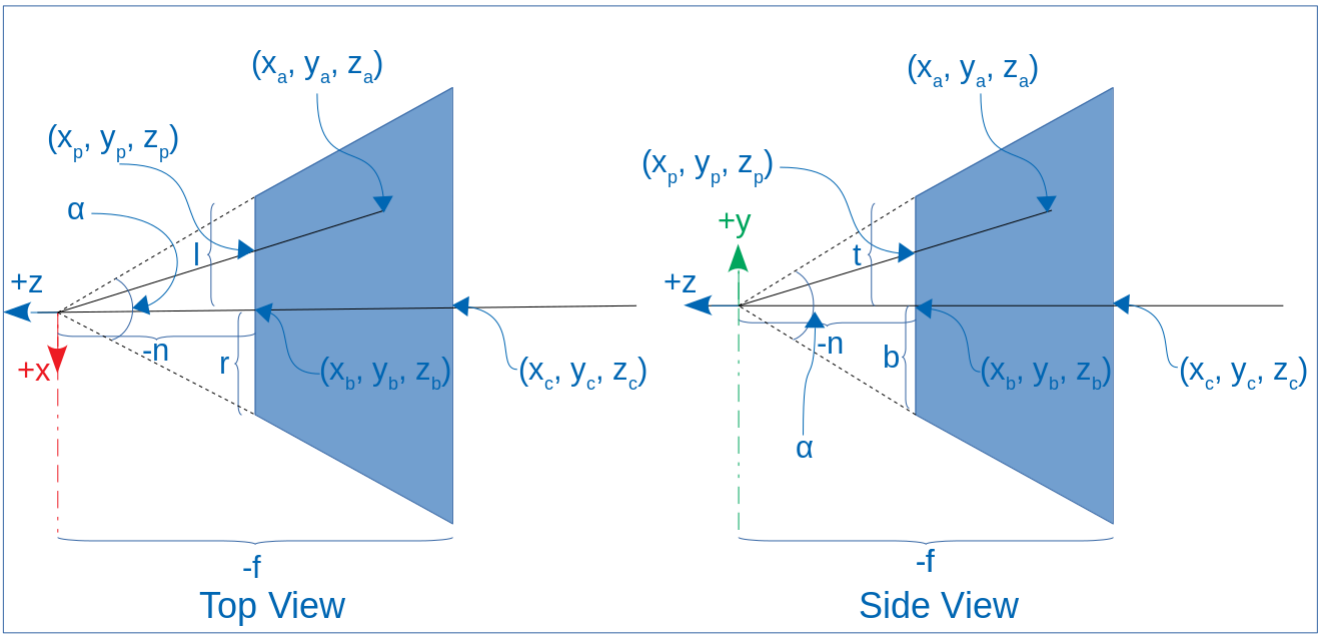


Figure 13.18: Side and Top views of the pyramidal view frustum

From the Top View,

$$\frac{x_a}{z_a} = \frac{x_p}{z_p}$$

Since $z_p = -n$,

$$x_p = \frac{-n * x_a}{z_a} \tag{13.9}$$

Likewise, from the Side View,

$$\frac{y_a}{z_a} = \frac{y_p}{z_p}$$

and,

$$y_p = \frac{-n * y_a}{z_a} \tag{13.10}$$

Now from figure 13.18, Top View,

$$l \leq x_p \leq r,$$
$$0 \leq x_p - l \leq r - l,$$
$$0 \leq \frac{x_p - l}{r - l} \leq 1,$$
$$0 \leq 2\frac{x_p - l}{r - l} \leq 2,$$
$$-1 \leq 2\frac{x_p - l}{r - l} - 1 \leq 1,$$
$$-1 \leq 2\frac{x_p - l}{r - l} - \frac{r - l}{r - l} \leq 1,$$
$$-1 \leq \frac{2x_p - 2l - r + l}{r - l} \leq 1,$$
$$-1 \leq \frac{2x_p - r - l}{r - l} \leq 1,$$
$$-1 \leq \frac{2x_p}{r - l} - \frac{r + l}{r - l} \leq 1,$$

At this stage, we substitute from eq 13.9.

$$-1 \leq \frac{-2 * n * x_a}{z_a(r - l)} - \frac{r + l}{r - l} \leq 1$$

Now, what we did here was convert a point (only the x dimension yet) from pyramidal view frustum to NDC view frustum. And, this takes place in 2 steps: 1) eye-space $\rightarrow$ clip-space 2) division by w which is equal to $-z_a$. But, in clip-space, the value is $\frac{2*n*x_a}{r-l} + \frac{z_a*(r+l)}{r-l}$. We have to encode this into a matrix row of M_{persp} in equation 13.7. The first row becomes,

$$[x_{0,0}, x_{0,1}, x_{0,2}, x_{0,3}] = [\frac{2n}{r - l}, 0, \frac{r + l}{r - l}, 0] \tag{13.11}$$

We perform the similar operations on y_a as we did for x_a

$$b \leq y_a \leq t$$

and this yeilds,

$$-1 \leq \frac{2y_p}{t - b} - \frac{t + b}{t - b} \leq 1,$$

Substituting for y_p from eq 13.10 and dividing the expression by z_a as we did above, we get $\frac{2n*y_a}{t-b} + \frac{z_a*(t+b)}{t-b}$. Encoding this as the second matrix row of M_{persp} in eq 13.7, we get,

$$[x_{1,0}, x_{1,1}, x_{1,2}, x_{1,3}] = [0, \frac{2n}{t - b}, \frac{t + b}{t - b}, 0] \tag{13.12}$$

We now have three rows of M_{persp} from equations 13.8, 13.11 and 13.12 as,

$$M_{persp} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ ... & ... & ... & ... \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{13.13}$$

To obtain the third row of M_{persp}, we take a slightly different approach. x, y and z be it in any space are variables. However, w depends on z. Based on this reasoning, $x_{2,0}$ and $x_{2,1}$ in M_{persp} from equation 13.7 can be made 0. Thus, the third row of M_{persp} takes the form $[0, 0, a, b]$, where we have to derive a and b. To do that, we use the points (x_b, y_b, z_b) and (x_c, y_c, z_c) in figure 13.17 (Top-View). Since, both these points lie on the z axis, they become $(0, 0, -n)$ and $(0, 0, -f)$ respectively.

Now, we multiply the points with our M_{persp} matrix.

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix}}_{M_{persp}} * \begin{bmatrix} 0 \\ 0 \\ k \\ 1 \end{bmatrix} \tag{13.14}$$

Here, k can be either $-n$ or $-f$ depending on which point we are using. Ofcourse, the 1 under k is due to the conversion to homogeneous coordinate system as we discussed in section 4.1.

$$z_{clip} = ka + b$$

and

$$w_{clip} = -k$$

$$z_{ndc} = \frac{z_{clip}}{w_{clip}}$$

$$z_{ndc} = \frac{ka + b}{-k} \tag{13.15}$$

Now, we take point $(0, 0, -n)$ and substitute in equation 13.15 to obtain,

$$-1 = \frac{-na + b}{n} \tag{13.16}$$

as near plane maps to -1 in NDC coordinate system (figure 13.17)

Substituting $(0, 0, -f)$ we get,

$$1 = \frac{-fa + b}{f} \tag{13.17}$$

as far plane maps to 1 in NDC coordinate system (figure 13.17)

Solving 13.16 and 13.17, we get,

$$a = -\frac{f + n}{f - n}$$

$$b = -\frac{2fn}{f - n}$$

Substituting in equation 13.14, we get the complete perspective projection transformation matrix as,

$$M_{persp} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{13.18}$$

In most of the cases we encounter, the pyramid in figure 13.17 (a) is symmetrical about the xz plane and the yz plane. In such a case, $t = -b$ and $r = -l$. Thus, equation 13.18 now becomes,

$$M_{persp} = \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{13.19}$$

Since we have assumed the view frustum is symmetrical, r/t gives the aspect ratio, ar. We also see from figure 13.18 (Side-View) that $n/t = 1/tan(\alpha/2)$ and $n/r = 1/(ar * tan(\alpha/2))$. Substituting these values in equation 13.19, we get,

$$M_{persp} = \begin{bmatrix} \frac{1}{ar*tan\alpha/2} & 0 & 0 & 0 \\ 0 & \frac{1}{tan\alpha/2} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{13.20}$$

As discussed in the orthographic projection section, OpenGL reads matrices in column major order thus during the actual implementation, ***we have to transpose*** M_{persp} ***before usage***.

Let us code this up. In *main.cpp*,

```
// defining the perspective projection matrix manually ----------------------------
GLfloat angle = 45.0f;
GLfloat n = 0.1f, f = 100.0f;
GLfloat ar = (GLfloat)width/(GLfloat)height; // aspect ratio

glm::mat4 projection_perspective = {1/(ar*tan(angle/2)), 0, 0, 0,
                        0, 1/tan(angle/2), 0, 0,
                        0, 0, -(f+n)/(f-n), -2*f*n/(f-n),
                        0, 0, -1, 0};

// OpenGL reads matrices in column major order
projection_perspective = glm::transpose(projection_perspective);

glUniformMatrix4fv(glGetUniformLocation(our_shader.program, "projection_perspective"), 1,
    GL_FALSE, glm::value_ptr(projection_perspective));
```

In *vertex_ shader.vert*,

```
uniform mat4 projection_perspective;

void main()
{
   gl_Position = projection_perspective * view * model * vec4(position, 1.0);
   /************************************************/
   /****** remainder of the vertex shader code ******/
   /************************************************/
```

You should obtain a box as shown in figure 13.19. You can find the code for this in ch13/perspective_projection. You can also find a video of the same here.

GLM provides an inbuilt method to construct the perspective projection matrix.

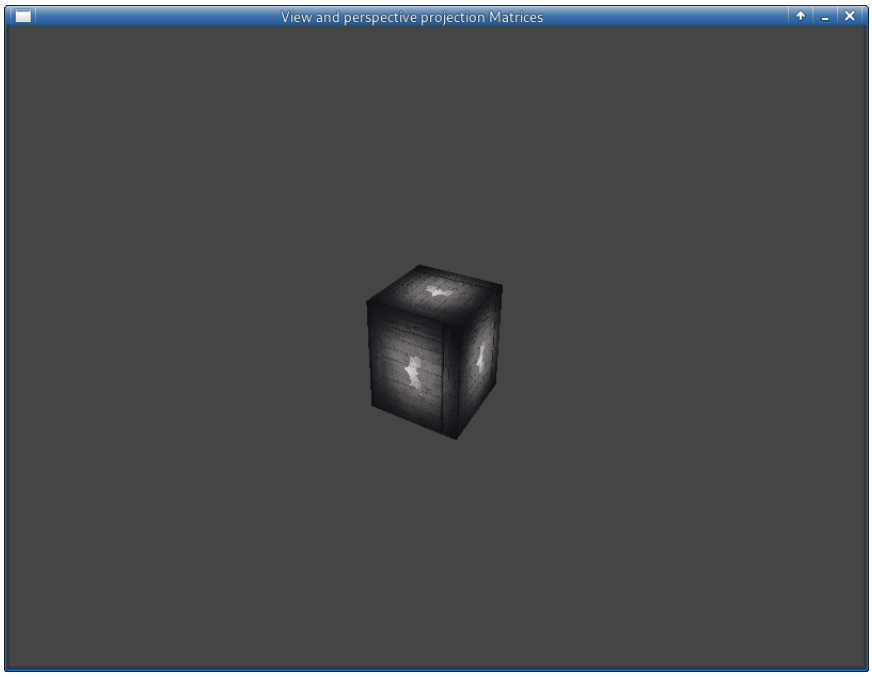


Figure 13.19

```
glm::mat4 projection_perspective = glm::perspective(angle, aspect_ratio, near, far);
```

13.7 Exercises

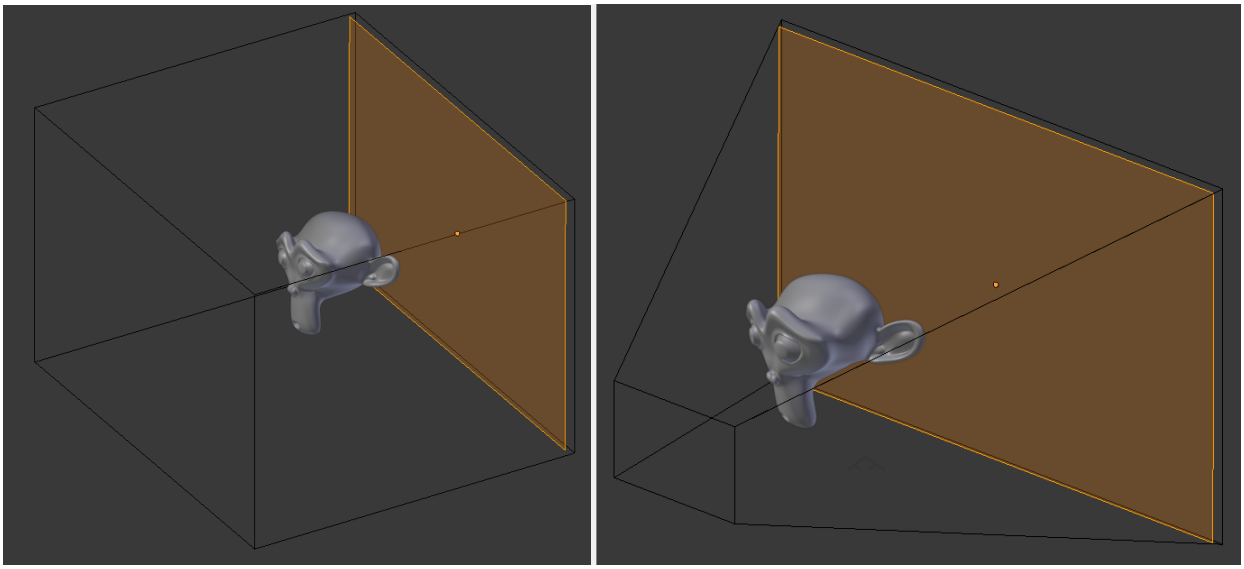
1. Try to implement a rotating box revolving around the camera.
2. Play around with orthographic and projection matrices.

Chapter 14

Augmented Reality with a Box

Finally, in this chapter we are going to learn how to overlay our crate on a marker. We have got a simple object rendered and why not complete the AR pipeline with that before we overlay more complicated 3D models. There are two objects we need to render for our AR application. They are,

1. ***Background Object:*** AR requires a background texture to be applied on a rectangular frame. This frame will act as the background as shown in figure 14.1. This frame is positioned just a bit (arbitrarily chosen) in front of the 'far' plane. The contents of the texture applied on this frame is captured by the webcam. We will use OpenCV to grab frames from the webcam and pass those frames as textures. The next question is, which would be an appropriate projection pattern to apply for the background object?



(a) (b)

Figure 14.1: Projection of Background Object as a) orthographic, b) perspective

The answer to that is, actually we can technically use both. Here in the upcoming subsections, we will be learning how to use both these projections for AR. 3D object requires us to use perspective projection and as a personal recommendation, I would recommend preserving homogeneity in projections thus using **perspective projection** alone. Its a bit unusual to mix perspective and orthographic projections, but we ll see it works just fine.

2. ***3D Object to be overlayed:*** At this stage we only know how to render simple shapes like our crate (object shown in figure 14 is a monkey-head). And, we are going to overlay the 3D object on

an ARUCO marker. We learnt in the previous chapter how to manipulate the position of the object using the 'model' and 'view' matrices (together they are addressed as 'modelview'); also to set the perspective projection using the 'perspective projection' matrix. Using OpenCV we did the same on a wireframe cube using 'extrinsic' matrix to set the position and orientation. 'Intrinsic' matrix was used to set the projection. Somehow we now need to translate the 'extrinsic and intrinsic' matrix to 'modelview and perspective-projection' matrix, during which, we should also handle all the quirks such as OpenCV coordinate system to OpenGL's coordinate system.

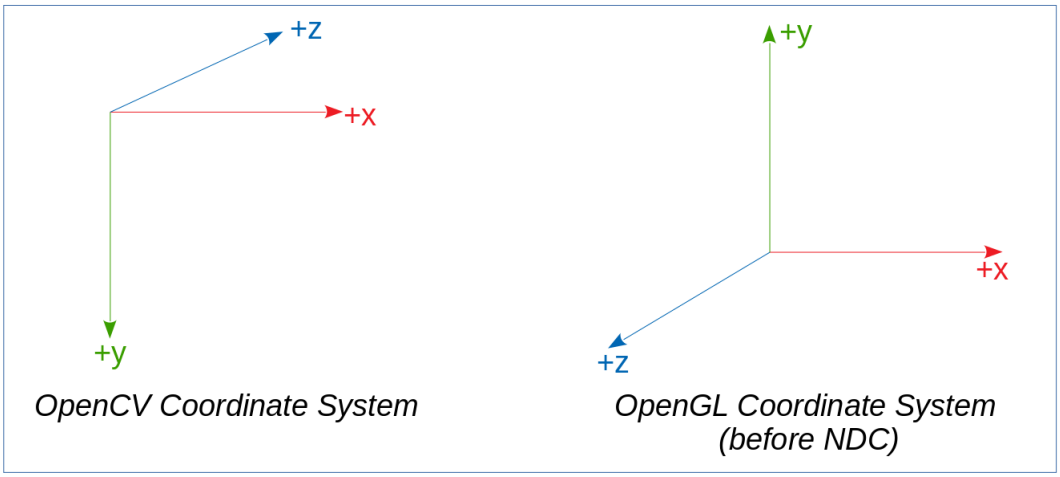


Figure 14.2: OpenCV and OpenGL coordinate system (before NDC)

14.1 OpenCV-to-OpenGL projection transform

OpenCV's intrinsic matrix creates the projection in the OpenCV world. We are going to derive the OpenCV to OpenGL perspective projection matrix. Consider a pin-hole camera model as shown in figure 14.3. This is the same as in figure 7.1, except the triangle on the left was flipped to the right. Since we are dealing with similar triangles and ratios, this wont cause any discrepancies.

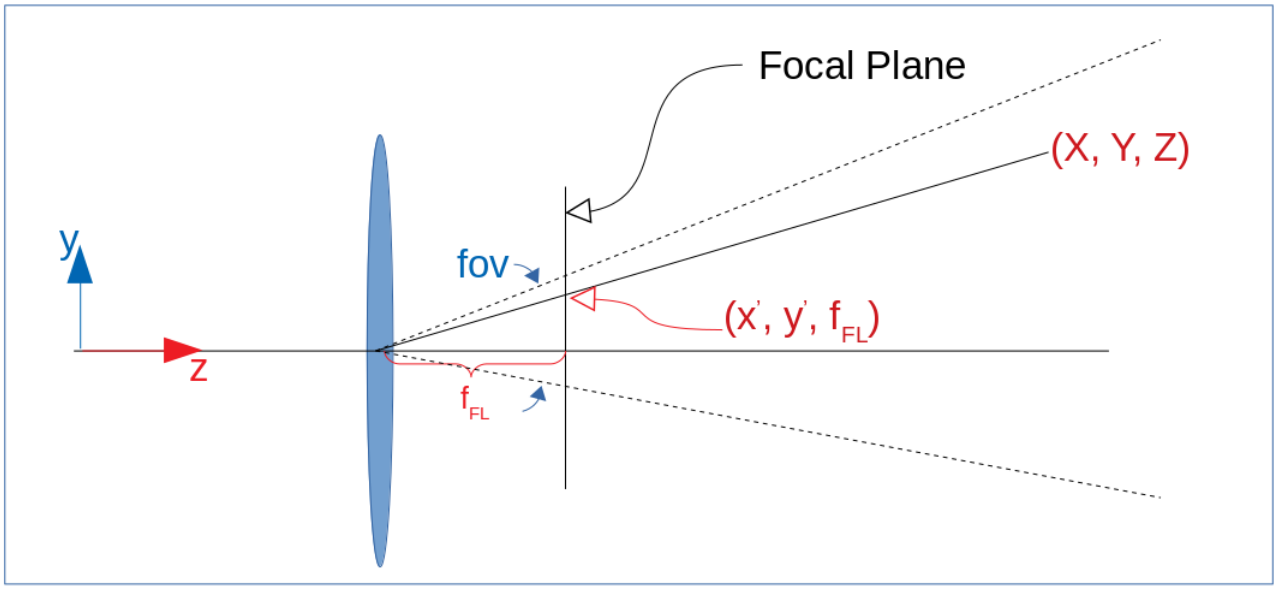


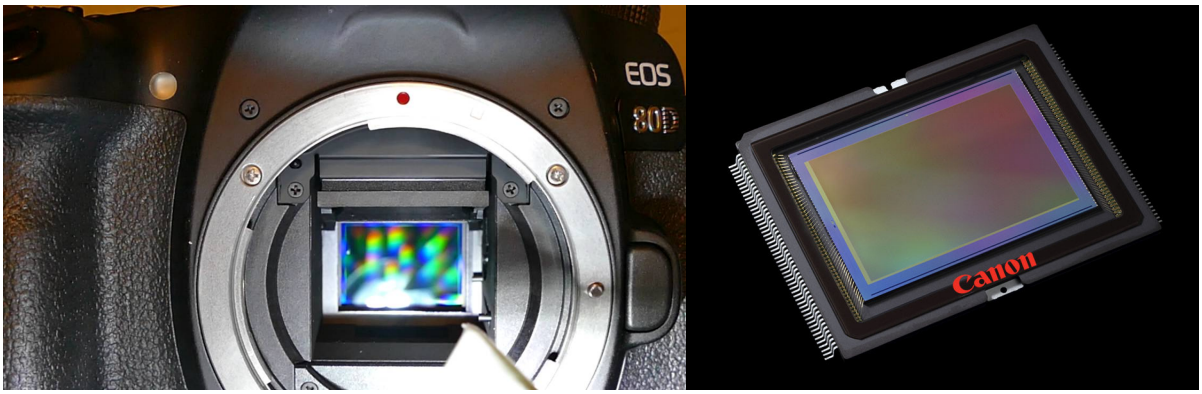
Figure 14.3: OpenCV Projection on the focal plane

In this figure, the focal length is f_{FL} because we used the symbol f for defining the far plane. Any point $[X, Y, Z, 1]$ can be projected onto the focal plane as $[x^{'}, y^{'}, f_{FL}, 1]$.

$$\begin{bmatrix} x^{'} \\ y^{'} \\ f_{FL} \\ 1 \end{bmatrix} = \begin{bmatrix} Xf_{FL}/Z \\ Yf_{FL}/Z \\ Zf_{FL}/Z \\ Z/Z \end{bmatrix} \approx \begin{bmatrix} Xf_{FL} \\ Yf_{FL} \\ Zf_{FL} \\ Z \end{bmatrix} = \begin{bmatrix} f_{FL} & 0 & 0 & 0 \\ 0 & f_{FL} & 0 & 0 \\ 0 & 0 & f_{FL} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{14.1}$$

The '1' in the last row is to capture the 'z' value exactly as discussed in section 13.6.2 where we used '-1'. This change is because of the z axis inversion in OpenCV when compared to OpenGL.

A careful reader might argue that, in equation 7.3 we had f_x and f_y and here we have only f_{FL}. This is because we did not deal with the concept of camera in completion there. Let us do that now. A basic camera has a lens which converges light rays onto a *focal plane* as shown in figure 14.3 (except for the flipping of the triangle). Usually camera's have a cascade of lenses used for zoom. Then the distances between the focal plane and the aggregate lens center is adjusted to obtain a sharp image. A *focal plane* is nothing but the distance from the lens center to the plane where the light entering through the lens converges. On the focal plane, camera manufacturers place a photo-sensitive film strip or a image sensor that is able to record light intensities. Most digital cameras use a CMOS image sensor placed exactly on the focal plane. The image sensor is nothing but a collection of several millions of sensors packed very finely into that tiny little space as shown in figure 14.4.



(a) (b)

Figure 14.4: a) canon image sensor attached to the camera image source, b) closeup of image sensor image source

Each of the million sensors are responsible for capturing intensities of RGB values. These values together are assigned as a color tone to each pixel on the digital image we see on our monitors. Sometimes, each of these small individual sensors that assign values to these pixels might not be square - symmetrical. And they also introduce a scaling factor both along the 'x' and 'y' axis. Let us use the notation k_x and k_y to denote the scaling factors. Moreover, while translating from *focal plane* to the *image plane* we observe, that there is a translation of origin from the center of the image to the corner as shown in figure . We capture this by modifying OpenCV's projection matrix as the following.

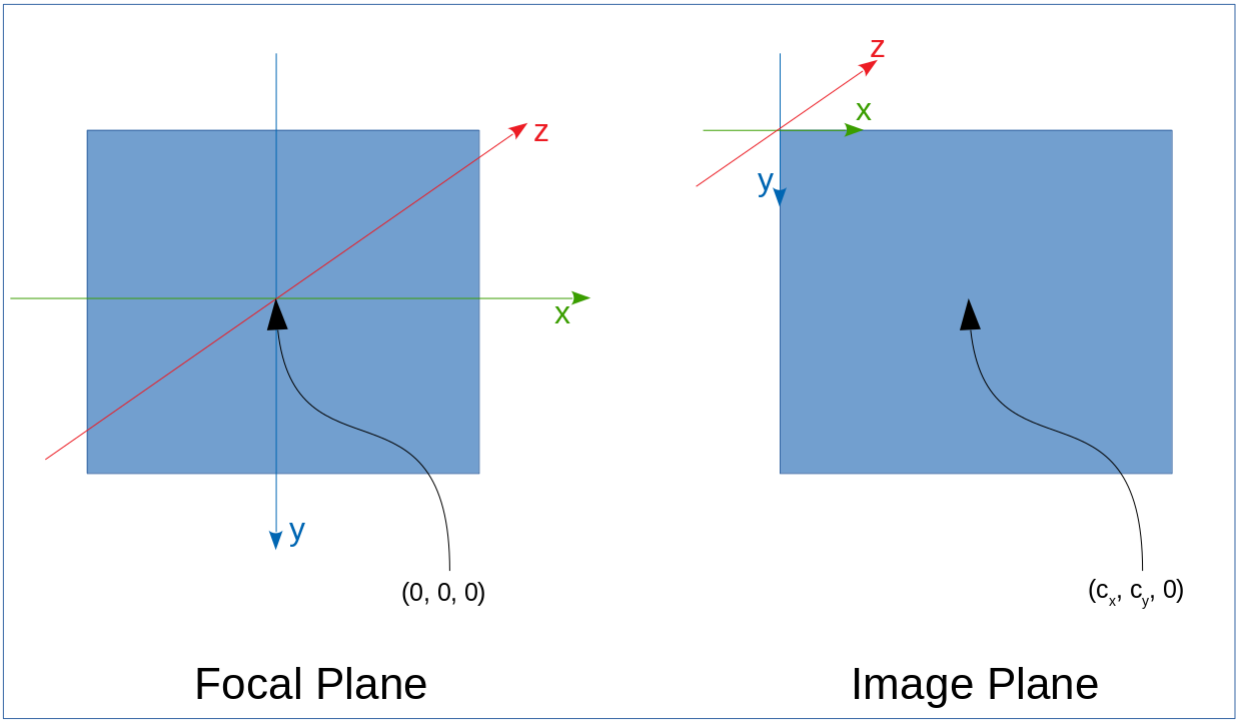


Figure 14.5: OpenCV focal plane to image plane

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} k_x & 0 & c_x/f_{FL} & 0 \\ 0 & k_y & c_y/f_{FL} & 0 \\ 0 & 0 & 1/f_{FL} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ f_{FL} \\ 1 \end{bmatrix} \tag{14.2}$$

Here, (x, y) denotes the pixel position on the image plane, and c_x and c_y represent image length/2 and width/2.

Combining Eq 14.1 and 14.2, we get

$$\begin{bmatrix} x \\ y \\ 1 \\ Z \end{bmatrix} \approx \begin{bmatrix} k_x & 0 & c_x/f_{FL} & 0 \\ 0 & k_y & c_y/f_{FL} & 0 \\ 0 & 0 & 1/f_{FL} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f_{FL} & 0 & 0 & 0 \\ 0 & f_{FL} & 0 & 0 \\ 0 & 0 & f_{FL} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{14.3}$$

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} k_x f_{FL} & 0 & c_x & 0 \\ 0 & k_y f_{FL} & c_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{14.4}$$

The top 3x3 is exactly the same as equation 7.3.

$$\begin{bmatrix} f_x & s_k & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} k_x f_{FL} & 0 & c_x \\ 0 & k_y f_{FL} & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{14.5}$$

$$f_x = k_x * f_{FL} \tag{14.6}$$

$$f_y = k_y * f_{FL} \tag{14.7}$$

Now, we have the complete OpenCV perspective projection matrix that projects a 3D point onto the image plane. Notice the only difference is we introduced the concept of *focal plane* and *image plane*.

Below (figure 14.6) we have an OpenGL pyramidal view frustum - side view. Its a fair assumption to make that, the view frustum is symmetrical about the xz and yz planes. Thus we use equation 13.19. M_{persp} is a factor of (r, t, n, f). Here, n and f are user defined. We have to somehow figure out r and t. Notice r and t are the only parameters that we have to determine to make OpenCV's view pyramid congruent to OpenGL's view pyramid.

$$M_{persp} = \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

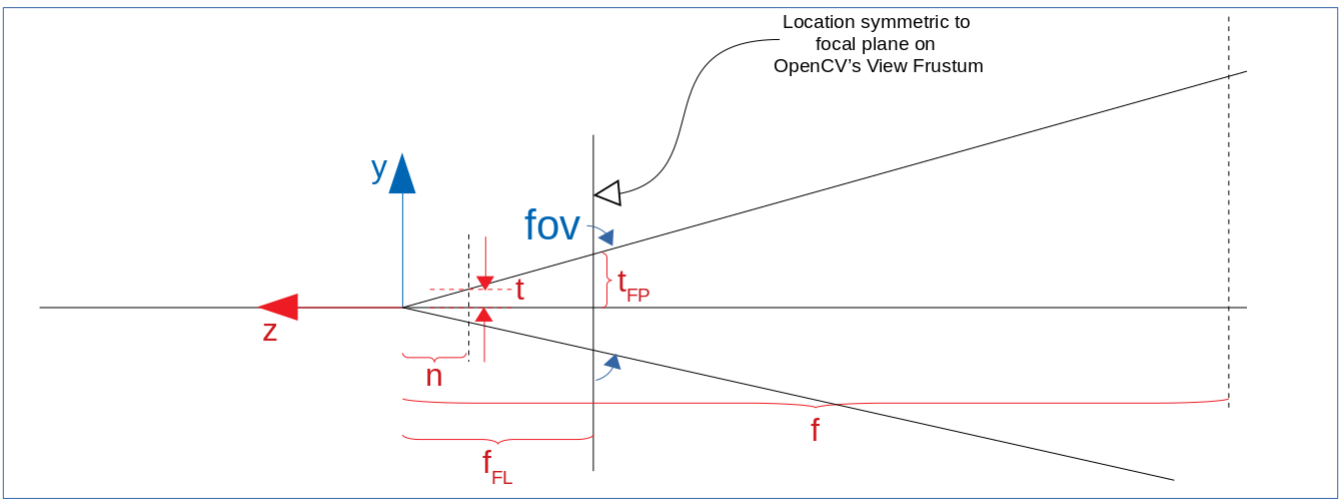


Figure 14.6: OpenGL view frustum

$$\frac{t}{n} = \frac{t_{FP}}{f_{FL}}$$

Since any point, (x', y') on the focal plane is equal to $(x'k_x, y'k_y)$ on the image plane.

$$k_y * t_{FP} = c_y$$

Substituting the above equation,

$$t = \frac{nc_y}{k_y * f_{FL}}$$

Substituting from 14.7, we get,

$$t = \frac{nc_y}{f_y}$$

Similarly solving for x, we get,

$$r = \frac{nc_x}{f_x}$$

Substituting in equation 13.19 for M_{persp}, we get

$$M_{persp-OpenCV} = \begin{bmatrix} \frac{f_x}{c_x} & 0 & 0 & 0 \\ 0 & \frac{f_y}{c_y} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{14.8}$$

Ofcourse, just like in chapter 13, we have to transpose the matrix as OpenGL reads the matrices in column major order. Also we cannot view the output until we fix the modelview matrix from the extrinsics. Recall, we require both the projection and modelview matrices to view any tangible output.

14.2 OpenGL's Modelview from OpenCV's Extrinsic Matrix

Modelview (view_matrix * model_matrix) is the equivalent matrix that governs rotations and translations in 3D space. In short, it decides where the 3D object, in our case the crate, has to be positioned in world space. The equivalent operation in OpenCV world is done using the extrinsic matrix defined in section 7.2. In this section, we are going to translate OpenCV's extrinsic matrix to OpenGL's modelview matrix. This process is not very complicated like the previous projection matrix translation and its almost a one to one mapping. There are two key changes that we have to make during the translation.

- Change of coordinates as shown in figure 14.2.
- OpenCV row major → OpenGL column major (this is the usual step we did for all the transformations we dealt with so far).

$$modelview = \begin{bmatrix} mv_{0,0} & mv_{0,1} & mv_{0,2} & mv_{0,3} \\ mv_{1,0} & mv_{1,1} & mv_{1,2} & mv_{1,3} \\ mv_{2,0} & mv_{2,1} & mv_{2,2} & mv_{2,3} \\ mv_{3,0} & mv_{3,1} & mv_{3,2} & mv_{3,3} \end{bmatrix} \tag{14.9}$$

We start of by converting the $[R|t]$ matrix from equation 7.4 into a 4x4 matrix by appending $[0, 0, 0, 1]$ as the last row.

$$modelview_{OpenCV-to-OpenGL} = \begin{bmatrix} r_{0,0} & r_{0,1} & r_{0,2} & t_0 \\ r_{1,0} & r_{1,1} & r_{1,2} & t_1 \\ r_{2,0} & r_{2,1} & r_{2,2} & t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The first change we have to make is flipping the y and z axis. Remember, OpenGL uses a column major matrix parsing style, thus *row 2* and *row 3* should be prefixed with $-^{ve}$.

$$modelview_{OpenCV-to-OpenGL} = \begin{bmatrix} r_{0,0} & r_{0,1} & r_{0,2} & t_0 \\ -r_{1,0} & -r_{1,1} & -r_{1,2} & -t_1 \\ -r_{2,0} & -r_{2,1} & -r_{2,2} & -t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We then transpose this matrix before using.

$$modelview_{OpenCV-to-OpenGL} = \begin{bmatrix} r_{0,0} & -r_{1,0} & -r_{2,0} & 0 \\ r_{0,1} & -r_{1,1} & -r_{2,1} & 0 \\ r_{0,2} & -r_{1,2} & -r_{2,2} & 0 \\ t_0 & -t_1 & -t_2 & 1 \end{bmatrix} \tag{14.10}$$

That is it. This is the final modelview matrix we will be using.

Its time to code these matrices up. Before we do that, we have to combine marker detection code written in OpenCV with OpenGL code. We will be using ch7/exercises/Aruco_extrinsics and ch13/perspective_projection as our boiler plate code.

First lets combine the *Makefiles* of both the projects together.

```
CC = g++

CFLAGS = -std=c++0x -g

INCLUDE = `pkg-config --cflags opencv` -I/usr/local/include -I/usr/local/include/GLFW
    -I/usr/include/SOIL

LIBS = -lglfw3 -lm -lGLEW -lGL -lGLU -lSOIL -ldl -lXinerama -lXrandr -lXi -lXcursor -lX11
    -lXxf86vm -lpthread `pkg-config --libs opencv` -laruco

FILENAME1 = main
FILENAME2 = ../include/cube
FILENAME3 = ../include/marker_detect
FILENAME4 = ../include/shader

all: ${FILENAME4}.cpp $(FILENAME3).cpp $(FILENAME2).cpp $(FILENAME1).cpp
   $(CC) $(CFLAGS) $(INCLUDE) ${FILENAME4}.cpp $(FILENAME3).cpp $(FILENAME2).cpp
      $(FILENAME1).cpp -o ../bin/$(FILENAME1) $(LIBS)

run:
   @./run.sh

clean:
   rm ../bin/./$(FILENAME1)
```

Contents of both the projects (*marker tracking* and *perspective projection*) are put in the same file. I will only go through the changes that have to be made. In *main.cpp*, we check the resolution of the image that is captured by OpenCV (or you can use *cheese* as described in section 6.1). Let's say it is 640x480.

```
int width = 640, height = 480; // capture properties of the camera using OpenCV.

int main(int argc, char **argv)
{
   cv::Mat rot_mat;

   /*********************************/
   /**** other lines of code ********/
   /*********************************/

   // default values of modelview
   glm::mat4 modelview;
   modelview = glm::translate(modelview, glm::vec3(0.0f, 0.0f, 10.0f);
```

Inside *int main(int argc, char **argv)*, we create a new *rot_mat* for the rotation matrix. We also will specify a default value of *modelview matrix* as initially. This is done to avoid the crate from appearing before we hover the aruco marker against the webcam. We will move the crate to behind the camera i.e., towards the $+^{ve}$ of the *z axis*.

Now, we define the perspective projection matrix outside the *while()* loop as it does not change with every iteration.

```
float near = 0.1f;
float far = 100.0f;
float fx = intrinsic_matrix.at<float>(0,0);
float fy = intrinsic_matrix.at<float>(1,1);
float cx = intrinsic_matrix.at<float>(0,2);
float cy = intrinsic_matrix.at<float>(1,2);

// OpenCV -> OpenGL perspective projection matrix generation.
glm::mat4 projection_perspective = {fx/cx, 0, 0, 0,
                  0, fy/cy, 0, 0,
                  0, 0, -(far+near)/(far-near), -(2*far*near)/(far-near),
                  0, 0, -1, 0};

// OpenGL reads matrices in column major order so we need to transpose
   projection_perspective
projection_perspective = glm::transpose(projection_perspective);
```

Awesome! Now, we need to define the modelview matrix. And, we also make changes to the *uniform* locator to look for *modelview*.

```
if(marker1.detect_flag)
{
   frame = cube1.drawcube(frame, intrinsic_matrix, distortion_parameters,
      marker1.rvecs, marker1.tvecs); // Draws the cube
   cv::Rodrigues(marker1.rvecs, rot_mat); // rvecs -> rotation matrix

   // OpenCV -> OpenGL modelview matrix generation.
   modelview = {rot_mat.at<double>(0,0), rot_mat.at<double>(0,1),
      rot_mat.at<double>(0,2), marker1.tvecs.at<double>(0), -rot_mat.at<double>(1,0),
      -rot_mat.at<double>(1,1), -rot_mat.at<double>(1,2),
      -marker1.tvecs.at<double>(1), -rot_mat.at<double>(2,0),
      -rot_mat.at<double>(2,1), -rot_mat.at<double>(2,2),
      -marker1.tvecs.at<double>(2), 0.0f, 0.0f, 0.0f, 1.0f};
   // OpenGL reads matrices in column major order so we need to transpose modelview
      matrix
   modelview = glm::transpose(modelview);
}
/*********************************/
/**** other lines of code ********/
/*********************************/

// we have model and view matrices combined as modelview
glUniformMatrix4fv(glGetUniformLocation(our_shader.program, "modelview"), 1, GL_FALSE,
   glm::value_ptr(modelview));
```

In the vertex shader,

```
uniform mat4 modelview;
void main()
{
   gl_Position = projection_perspective * modelview * vec4(position, 1.0);
```

Awesome, you should see two windows appear side by side. OpenCV window shows the webcam with aruco marker tracking with our cube overlayed. OpenGL window shows a very tiny crate that moves in accordance to the marker as its moved in front of the webcam. The crate is tiny, why? During camera calibration we physically measured the square size of each of the chessboard square and set input the value (section 6.3.2 where we entered the terminal command *./cpp-example-calibration -w 6 -h 7 -s 0.025*). Each square on my chessboard measured 0.025 meters. We also set the size of the vertices of the crate ranging from -0.5 to 0.5, which is 1 unit length. The equivalence is now 1 unit is equal to 0.025. In the file *run.sh*, we also enter the aruco per-marker-square size to be 0.05. By this equivalence to fill each square, we have to double the scaling factor of the vertices of the crate. This small exercise is left for the reader to implement.

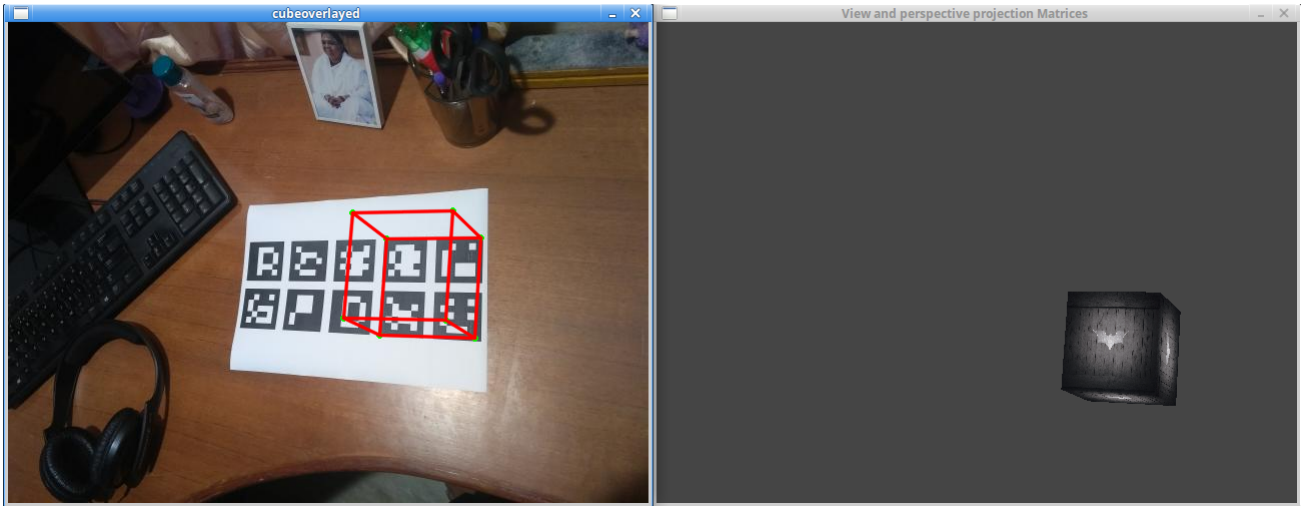


Figure 14.7: Implementation screenshot of background less AR

You can find the source code for this project in ch14/backgroundLess_AR. To compile this type 'make' and to execute this project type './run.sh'. And, a video of this can be found here.

14.3 Let us fix the background - Background object Perspective Projection

Cool, we are almost getting there huh? Let us get onto fixing the background. First, let us start with perspective projection, then we will move onto orthographic projection. Our first obvious task is to calculate the dimensions of the background object and its position. Let us say we are going to render the background object a small distance before the far plane.A *yz* plane, view of the pyramidal view frustum is shown in figure 14.8. We do not want the background object to exactly lie on the far plane as those are extreme cases (remember objects beyond the far plane do not get rendered). Also we want it far enough so that the 3D object (our crate) never crosses the background object. We can arbitarily choose the distance.

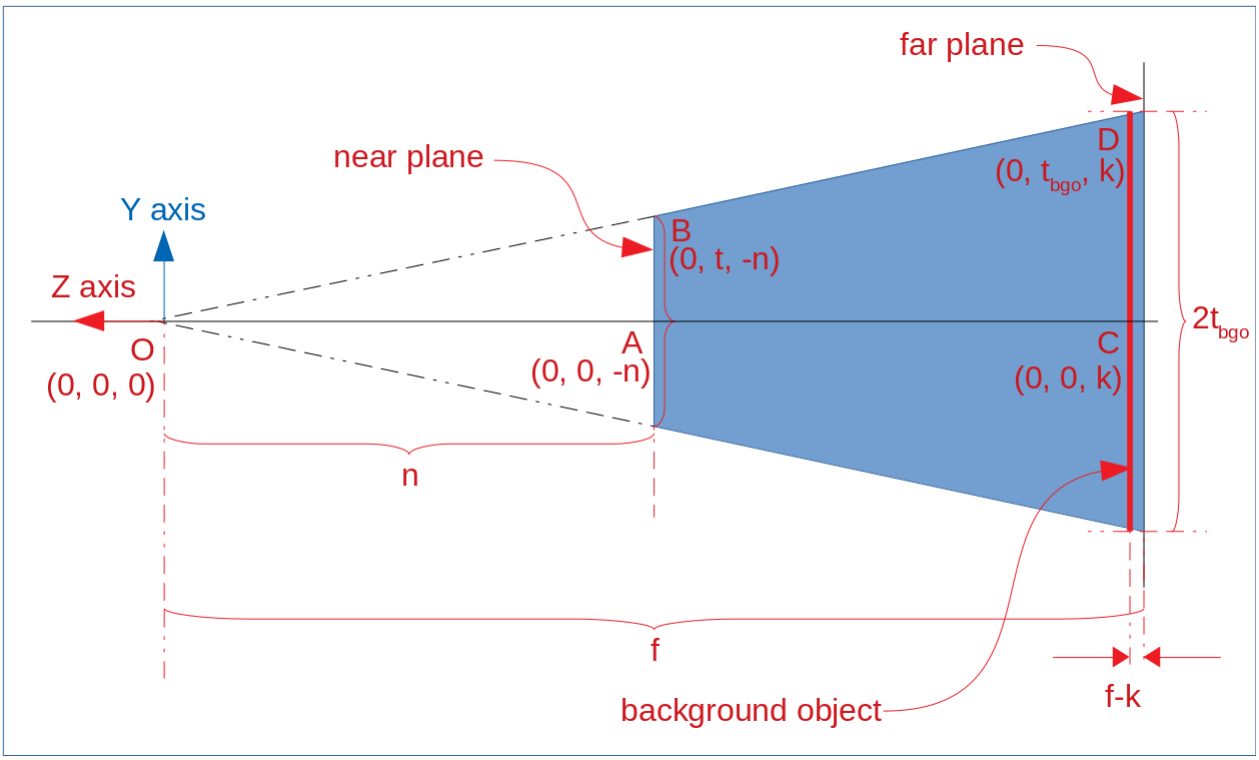


Figure 14.8: YZ plane view of pyramid view frustum with background object

Take a look at figure 14.8 which shows the pyramidal view frustum. Notice the location of the background object (denoted with a subscript *bgo*). Since our objective is to calculate the size of the background object, we need to calculate t_{bgo} and r_{bgo}. r_{bgo} is not described in the figure as it is similar to calculating t_{bgo} except that we would be looking down from the top view at the xz plane. Triangles OAB and OCD are similar. Thus

$$\frac{BA}{OA} = \frac{DC}{OC}$$

$$\frac{t}{n} = \frac{t_{bgo}}{k}$$

Substituting from equation above 14.8 i.e., $t = \frac{nc_y}{f_y}$, we get,

$$\frac{c_y}{f_y} = \frac{t_{bgo}}{k}$$

$$t_{bgo} = \frac{kc_y}{f_y} \tag{14.11}$$

and,

$$r_{bgo} = \frac{kc_x}{f_x} \tag{14.12}$$

Remember, t_{bgo} and r_{bgo} are only half the length and width of the background object. **To exactly span the pyramidal view frustum, a unit cube placed at a distance of k from the origin would have to be scaled by $2r_{bgo}$ on the *x axis* and $2t_{bgo}$ on the *y axis*.**

All that is remaining is to code this up. We wont be applying the modelview matrix derived from OpenCV for the background object because we do not want the background object to move along the marker. Since, we will be using a different modelview matrix, we can in fact use different shaders. In *main.cpp* we add,

```
// background object definitions
GLfloat vertices_bg[] =
{
   -0.5f, -0.5f, 0.0f, 0.0f, 0.0f,
    0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
    0.5f,  0.5f, 0.0f, 1.0f, 1.0f,
    0.5f,  0.5f, 0.0f, 1.0f, 1.0f,
   -0.5f,  0.5f, 0.0f, 0.0f, 1.0f,
   -0.5f, -0.5f, 0.0f, 0.0f, 0.0f,
};

// ------------------------------------------------------------------------
// background object
GLuint VBO_bg, VAO_bg;
glGenVertexArrays(1, &VAO_bg);
glGenBuffers(1, &VBO_bg);

glBindVertexArray(VAO_bg);

glBindBuffer(GL_ARRAY_BUFFER, VBO_bg);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices_bg), vertices_bg, GL_STATIC_DRAW);

// Postion Attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5*sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);

// Texture Attribute
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5*sizeof(GLfloat),
    (GLvoid*)(3*sizeof(GLfloat)));
glEnableVertexAttribArray(1);

glBindVertexArray(0); // Unbind VAO_bg

// ------------------------------------------------------------------------
// webcam texture
cap >> frame;
width = frame.size().width;
height = frame.size().height;

GLuint texture_bg;
glGenTextures(1, &texture_bg);
glBindTexture(GL_TEXTURE_2D, texture_bg);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
    frame.data);
```

```
   glGenerateMipmap(GL_TEXTURE_2D);
   glBindTexture(GL_TEXTURE_2D, 0);
   // ================================================================================
   // transformation presets for the background object (model, view and orthographic
      projection)
   glm::mat4 modelview_bg = {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, -499, 0, 0, 0, 1};
   modelview_bg = glm::transpose(modelview_bg);
   modelview_bg = glm::scale(modelview_bg, glm::vec3(499*cx/fx, 499*cy/fy, 0));
   // ================================================================================
```

Here, we have used $(f, n) = (500.0f, 1.0f)$, and $k = 499$. Thus the scaling factor along x and y axis is $\frac{998c_x}{f_x}$ and $\frac{998c_y}{f_y}$. Now, we draw the background object inside the *while()* loop.

```
   // draw bg ------------------------------------------------------------------------
   glUseProgram(bg_shader.program);
   glBindVertexArray(VAO_bg);

   glActiveTexture(GL_TEXTURE2);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, frame.cols, frame.rows, 0, GL_BGR,
      GL_UNSIGNED_BYTE, frame.data);
   glBindTexture(GL_TEXTURE_2D, texture_bg);
   glUniform1i(glGetUniformLocation(bg_shader.program, "webcam_texture"), 2);

   glUniformMatrix4fv(glGetUniformLocation(bg_shader.program, "modelview_bg"), 1,
      GL_FALSE, glm::value_ptr(modelview_bg));
   glUniformMatrix4fv(glGetUniformLocation(bg_shader.program,
      "perspective_projection_bg"), 1, GL_FALSE, glm::value_ptr(perspective_projection));

   glDrawArrays(GL_TRIANGLES, 0, 6);
   glBindVertexArray(0);
```

We also create additional shaders viz., *bg_ vertex_ shader.vert* and *bg_ fragment_ shader.frag*.

```
// background object vertex shader
#version 330 core
layout (location = 0) in vec3 position; // vertex_position attribute
layout (location = 1) in vec2 tex_coords; // texture_coordinate attribute

out vec2 texCoords;

uniform mat4 modelview_bg;
//uniform mat4 view_bg;
uniform mat4 perspective_projection_bg;

void main()
{
   gl_Position = perspective_projection_bg * modelview_bg * vec4(position, 1.0);
   texCoords = vec2(tex_coords.x, 1.0-tex_coords.y); // to invert the texture
```

```
}
```

```
// background object fragment shader
#version 330 core

out vec4 color;
in vec2 texCoords;

uniform sampler2D webcam_texture;

void main()
{
   color = texture(webcam_texture, texCoords);
}
```

If you have managed to code everything correctly, then you should obtain output similar to the following image as shown in figure 14.9. Click the following link to see the video. You can obtain the source code for the implementation in ch14/AR_perspective.

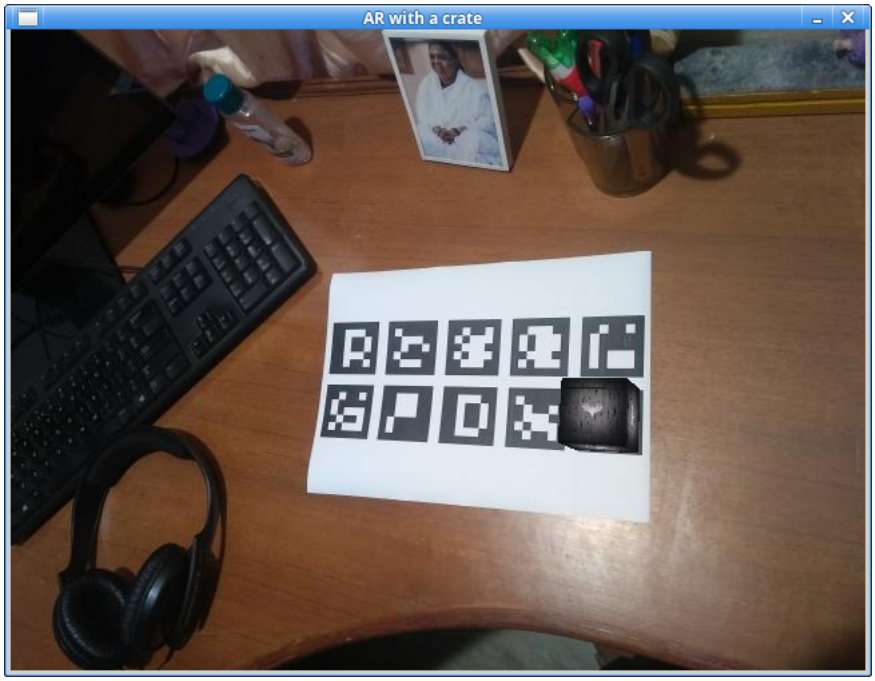


Figure 14.9: AR overlay with a crate

Sometimes, you might find that the crate does not quite rest on the marker image. Instead, it simply floats above the marker image. If you are sure your code works perfectly, then most likely the fault lies in camera calibration. Please read section 14.5 for correcting this.

14.4 Background object Orthographic Projection

Orthographic projection for the background object is pretty simple. There are no new concepts to be learnt and we can jump straight into implementation. Just like background perspective projection, we have to

define vertices, create VAO's and VBO's. In addition, we have to define a separate *orthographic projection* matrix and a default scaling *model* matrix. In *main.cpp*,

```
// transformation presets for the background object (model, view and orthographic
    projection)
glm::mat4 model_bg;
glm::mat4 view_bg;
GLfloat ortho_far = 10000.0f;

model_bg = glm::scale(model_bg, glm::vec3(width_window, height_window, 1));
view_bg = glm::translate(view_bg, glm::vec3(0, 0, -ortho_far));

// we will use orthographic projection for the background
glm::mat4 orthographic_projection_bg = glm::ortho(0.0f, (GLfloat)width_window, 0.0f,
    (GLfloat)height_window, near, ortho_far);
```

and inside the *while()* loop,

```
// draw bg ------------------------------------------------------------------------
glUseProgram(bg_shader.program);
glUniformMatrix4fv(glGetUniformLocation(bg_shader.program, "model_bg"), 1, GL_FALSE,
    glm::value_ptr(model_bg));
glUniformMatrix4fv(glGetUniformLocation(bg_shader.program, "view_bg"), 1, GL_FALSE,
    glm::value_ptr(view_bg));
glUniformMatrix4fv(glGetUniformLocation(bg_shader.program,
    "orthographic_projection_bg"), 1, GL_FALSE,
    glm::value_ptr(orthographic_projection_bg));

glBindVertexArray(VAO_bg);

glActiveTexture(GL_TEXTURE2);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, frame.cols, frame.rows, 0, GL_BGR,
    GL_UNSIGNED_BYTE, frame.data);
glBindTexture(GL_TEXTURE_2D, texture_bg);
glUniform1i(glGetUniformLocation(bg_shader.program, "webcam_texture"), 2);

glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
```

And finally ofcourse, we have to change the *bg_vertex_shader.vert* as follows,

```
uniform mat4 model_bg;
uniform mat4 view_bg;
uniform mat4 orthographic_projection_bg;

void main()
{
```

```
    gl_Position = orthographic_projection_bg * view_bg * model_bg * vec4(position, 1.0);
```

The output will be exactly identical to figure 14.9. For the complete source code, visit ch14/AR_orthographic. Ofcourse, just like in the previous section, if the crate does not rest on the marker, there is a good chance that the camera calibration is not perfectly done. We will discuss that in the next section.

14.5 Implementation Quirks - Camera Calibration

We implemented *backgroundLess_AR*, *AR_projection* and *AR_orthographic*. All three programs have */src/aruco_files* which hosts a *camera_parameters.yml* file. This file contains the camera matrix. This is the same as in section 6.3.2 where we generated a *cameraConfig.yml*. It turns out, the binaries we used to create the *.yml* file does not yield accurate results. I really don't know why it does not. It looks like at this stage we have to rework on the camera calibration a bit. Luckily for us if we compile the source code for camera calibration found deep inside the bowels of the OpenCV source code, we can obtain a pretty accurate camera matrix in the *.yml* file. Follow the below instructions for obtaining a new *camera_parameters.yml* file.

```
cd <your-path-to-opencv>/samples/cpp/tutorial_code/calib3d/camera_calibration
```

The files in the directory include *camera_calibration.cpp* and *in_VID5.xml*. These two files are the only ones we will be using. However, to compile the *camera_calibration.cpp* file (link), we need a *Makefile*. Let us create one.

```
CC = g++

CFLAGS = -g

INCLUDE = `pkg-config --cflags opencv`

LIBS = `pkg-config --libs opencv`

FILENAME = camera_calibration

all: $(FILENAME).cpp
   @$(CC) $(CFLAGS) $(INCLUDE) $(FILENAME).cpp -o $(FILENAME) $(LIBS)

run:
   @./$(FILENAME)
```

in_VID5.xml is the location where we insert our parameters such as the size of the square on the chessboard, camera # (or path-to-images or video_file). Let us do that now.

1. *<BoardSize_Width>width</BoardSize_Width>* and *<BoardSize_Height>height</BoardSize_Height>* - Between the tags, enter the number of inner corners of the chessboard. On this chessboard image, the *width=9* and *height=6*.

2. *<Square_Size>size</Square_Size>* - take a print out of the chessboard and measure the length of a side of the square. On an A4 paper, mine turned out to be 25mm. Insert *25* between the tags.

3. *<Input>device_number</Input>* - Between the input tags enter the camera device number in use. For instance, *<Input>"0"</Input>* for the first camera (we discussed the camera number convention in section 6.1).

4. Change this tag as follows *<Write_outputFileName>camera_parameters.yml</Write_outputFileName>*. Remember, we need a *.yml* file as output.

Now, we are ready to run our program. Type *make* and *./camera_calibration in_VID5.xml* on the terminal to run this code. Paste, the print out of the chessboard on a flat horizontal board. Show it in front of the camera and press 'g' to start the program. It will output *out_camera_data.yml*. This is our required file. Let us rename it to *camera_parameters.yml*.

There are a few more quirks that we have to sort out before we can use this *yml* file. We are using an old version of aruco which parses for yml tags that have all lower cases. Open the yml file and change the following all to lower case.

1. *image_Width* → *image_width*

2. *image_Height* → *image_height*

3. *board_Width* → *board_width*

4. *board_Height* → *board_height*

5. *square_Size* → *square_size*

6. *Camera_Matrix* → *caemra_matrix*

7. *Distortion_Coefficients* → *distortion_coefficients*

Perfect, this is the new *camera_parameters.yml* file we will be using henceforth. Click here to obtain the project files.

———————————x———————————

If you have made it this far, give yourself a big treat! You have learnt and implemented the entire OpenCV + OpenGL AR pipeline successfully. Concepts relating to AR end here and the upcoming chapters will deal with trying to overlay fancy 3D objects instead of a boring crate. And this will all be OpenGL concepts.

Chapter 15

Colors

After finishing off the essentials, we are going to get into fancy stuff now. In this chapter, we will be discussing colors. What are colors? In real world, sun/light-source emits light and it falls on an object. The source emits many wavelengths of light both visible and invisible. Visible spectrum is broadly categorized as a rainbow of colors with the mnemonics VIBGYOR (Violet, Indigo, Blue, Green, Yellow, Orange and Red). When this mixture of light or a subset of this mixture in case of other light sources, fall on the object, the object absorbs certain wavelengths and re-emits certain others. This re-emitted light falls on our retinal vision receptors as we see the object. The wavelength that reaches our eye corresponds to the characteristic color of the object.

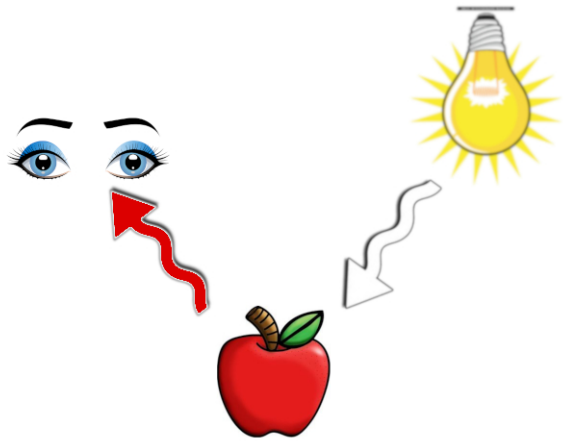


Figure 15.1: Depiction of how we percieve colors

In the real world there are infinite shades of colors. However, we really cannot represent so many colors on a computer. What we do instead is represent tones of primary colors Red, Green and Blue. Further each tone can be represented by a set of bits. Say we choose to represent each of the RGB tones using an 8 bit color scheme, we can represent $2^8 = 256$ tones of each color. This totally makes up 24 color bits. Suppose, the light is red in color, all the objects lit up by that light will have a tint of red color. Thus, the perceived color of the object is affected by both the color of light from the source and the inherent nature of emission of certain wavelengths of incident light by the object.

We already learnt how to apply color to an object in OpenGL (remember the multicolored triangle we coded in section 11.3). What we did was to define color values for each vertex. The color values are assigned by the fragment shader. For all the fragments between the vertices of the triangle, the color values assigned for each of the vertices is interpolated. This is shown in figure 11.2. In this section, let us draw a blue box and placed under red color lighting. We pass the color of the light and the color of the toy

as uniforms to the fragment shader. Lets use the code from ch13/perspective_projection as boiler plate code. Firstly, lets get rid of all the unwanted textures from both *main.cpp* and *fragment_shader.frag*.

```
//Add a uniform for box color
glUniform3f(glGetUniformLocation(our_shader.program, "box_color"), 0.3f, 0.6f, 0.7f);

//Add a uniform for light color
glUniform3f(glGetUniformLocation(our_shader.program, "light_color"), 0.93f, 0.83f,
    0.8f);
```

Now, inside the fragment shader, we perform a scalar multiplication to obtain the resultant color value.

```
color = vec4(light_color * box_color, 1.0f);
```

Your output should resemble figure 15.2. Please find the implementation source code in ch15/colors.

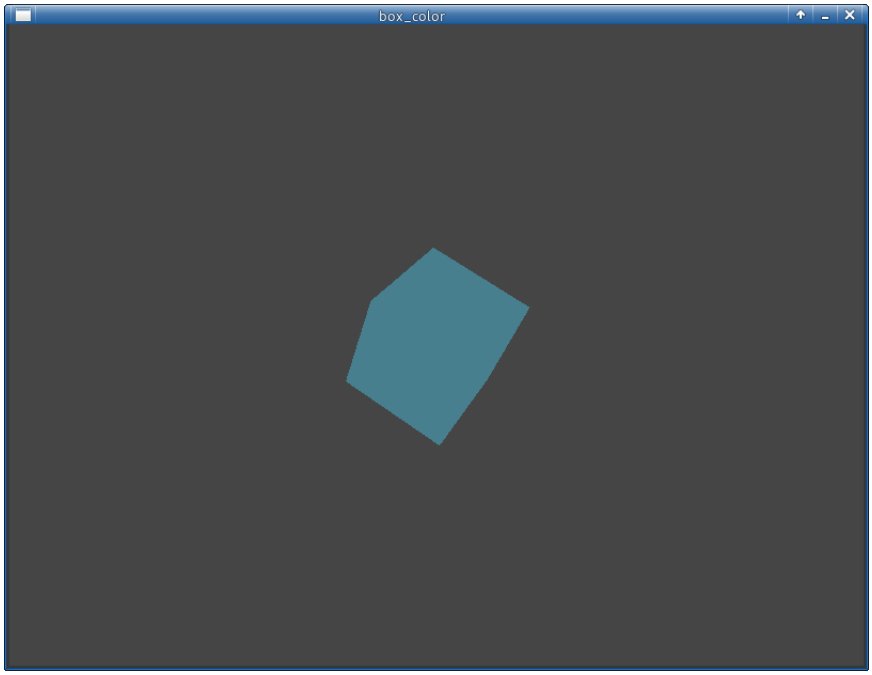


Figure 15.2: colored box

I agree, the output is kind of bland. Let us spice it up more, shall we.

Chapter 16

Lighting

In the last chapter, we implemented a pretty basic lighting scheme. But thats (figure 15.2) is not really how objects look. In the real world, usually we have directional light sources; for instance you can say "hey, the sun is over there", or "the bulb is attached to the ceiling in the middle of the room". Depending on the position and distance of the light, the side or faces of the object light up differently. The side facing the light is brighter than the opposite side. Broadly, we have three categories of lighting.

1. ***Ambient Lighting:*** This is the old scheme of lighting that we discussed in the last chapter where we assume some intensity of ambient light everywhere and it uniformly lights up the object just like in figure 15.2. This is usually because there is somehow ambient light everywhere due to multiple reflections from things around the object. Ambient lighting simulates this.

2. ***Diffused Lighting:*** In this lighting scheme, fragments facing the light are brighter than fragments away from the light.

3. ***Specular Lighting:*** This one is the coolest of them all. See the bright spot of white on figure 10.1(a) where there is a spike in intensity of light reflected off the object, that is called *specular highlights*. We will be implementing those as well.

Most objects are rendered with a combination of the above three lighting schemes.

16.1 Ambient Lighting

The code for this lighting scheme is the same as what we discussed in the last chapter (ch15/colors). Let us modify the code a bit so that we can have a variable that can control ambient light strength. In *main.cpp*, we add

```
int main()
{
   /********** other lines of code ***********/
   float ambient_strength = 0.6;
   glm::vec3 light_color = glm::vec3(1.0f, 1.0f, 1.0f);
   glm::vec3 ambient_light = ambient_strength * light_color;

   while(1)
   {
      /***** other lines of code **********/
      glUniform3f(glGetUniformLocation(our_shader.program, "ambient_light"),
          ambient_light.x, ambient_light.y, ambient_light.z);
```

```
}
```

In the fragment shader, we add the uniform *ambient_light*.

```
uniform vec3 ambient_light;

int main()
{
   color = vec4(ambient_light * box_color, 1.0f);
}
```

In figure 16.1, the intensity of light falling on the object is reduced; a proportional reduction in the intensity of reflected light can be seen too.

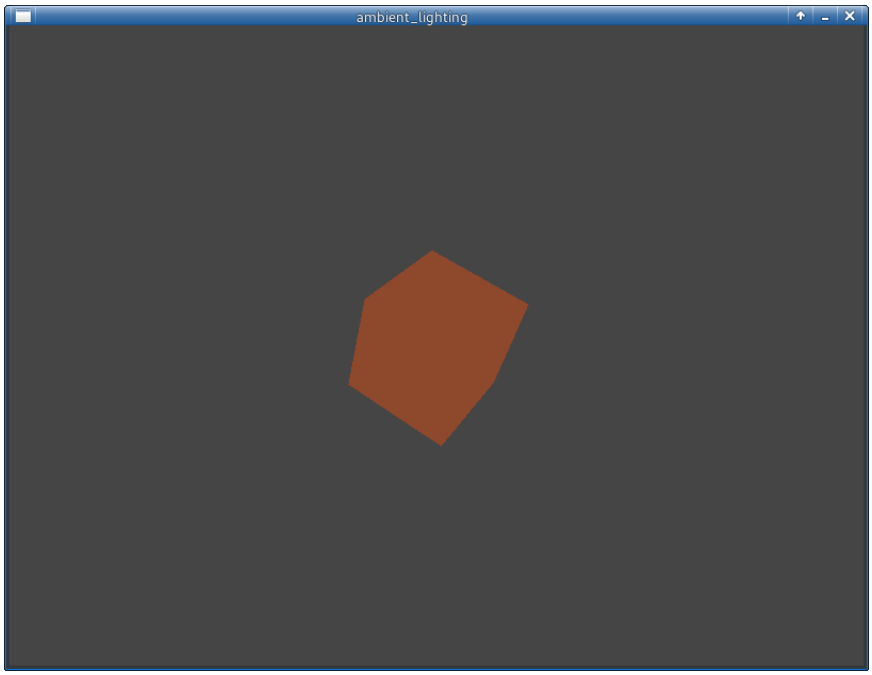


Figure 16.1: Reduction of ambient light on an object

Source code is in ch16/ambient_lighting.

16.2 Diffused Lighting

In this scheme of lighting, intensities of fragments facing the light source should be brighter than the others. Also, fragments closer to the light source should be brighter. In figure 16.2, we see that the face of the box towards the light is brighter and the face opposite to the light is darker.

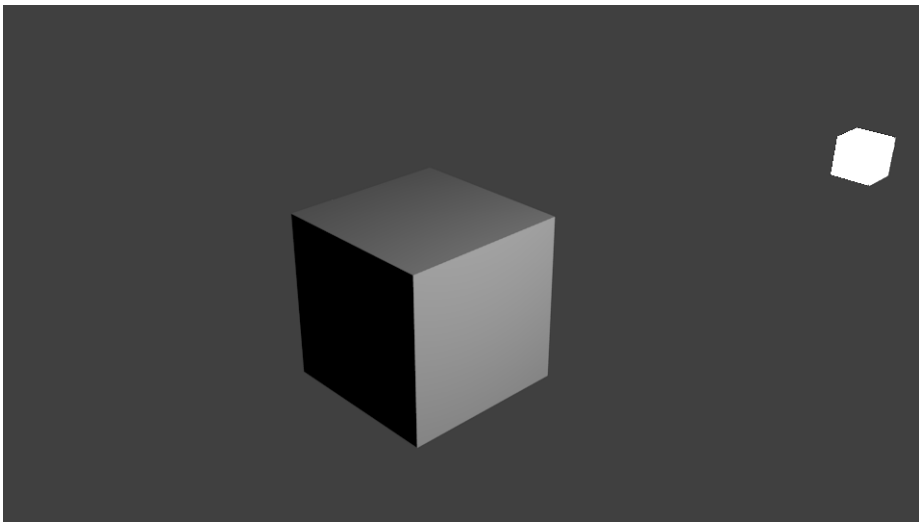


Figure 16.2: Diffused Lighting scheme

What are the factors contributing to lighting diffusion? Simply put, there are two factors,

1. Distance of the fragment from the light source.
2. The angle of incidence of light to the fragment.

16.2.1 Diffused Lighting - Effect of distance

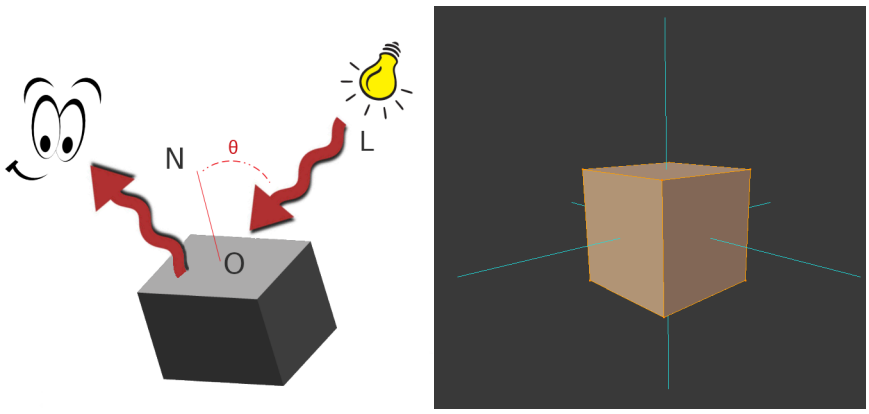
Intensity variation of light follows the inverse square law i.e., the intensity of light is inversely proportional to the square of the distance of the fragment from the source.

$$Light\_intensity\_at\_fragment \propto \frac{1}{distance^2}$$

$$Light\_intensity\_at\_fragment = \frac{k}{distance^2}$$

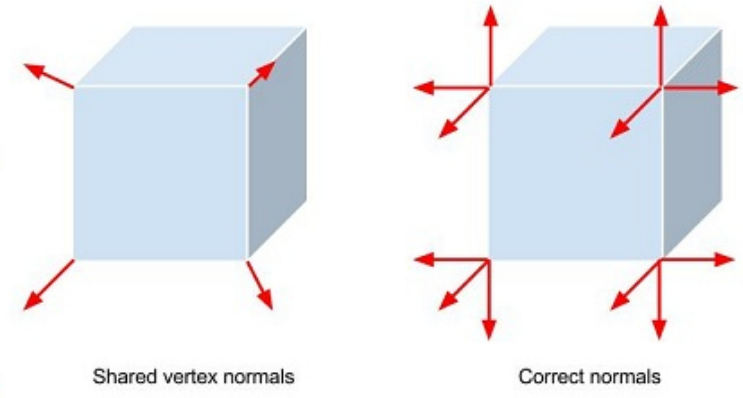
16.2.2 Diffused Lighting - Effect of angle of incidence

Consider figure 16.3 (a). LO describes the incident light. ON describes the normal to a surface on the box. ***Normals*** are vectors perpendicular to a surface. Commonly, there are surface normals, vertex normals as described by figure 16.3 (b) and (c).



(a)

(b)



(c)

Figure 16.3: (a) diffused lighting scheme, (b) surface normals and (c) shared and corrected vertex normals

In figure 16.3(a), we see a box and a light source. Rays from the light source fall on fall on a surface and the perceived intensity of light is directly proportional to the angle of incidence of the light on that surface. Notice the lines drawn perpendicular to the surface. These are called *surface normals* and they are used for calculating lighting effects.

$$Diffused\_light = (normal \bullet incident\_light\_vector) * light\_strength$$

Let us code this up. For our application, we will be using *surface normals* from figure 16(b). But how do we give normals as input to the shader programs? Remember, we described texture coordinates along with vertex coordinates in section 12.1? Similar to that, we can supply normals as another attribute to OpenGL. You can find a complete vertex and normal list here. Do add the vertices to *main.cpp* and modify the attributes layout description.

```
GLfloat vertices[] =
{
   //    vertices          Normals
   // front ------------------------------
   // face1, triangle1
   -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, 1.0f,
    0.5f, 0.5f, 0.5f, 0.0f, 0.0f, 1.0f,
              .
```

```
                .
                .
    0.5f, 0.5f, -0.5f, 0.0f, -1.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 0.0f, -1.0f, 0.0f
};
```

We also use these vertices to create another lamp object. This will define the position of the lamp. Lets make our lamp revolve around our object. We also pass the position of the lamp as a uniform.

```
while()
{
   /**** other lines of code *********/
   float time = glfwGetTime();
   float lamp_pos_x = 3*sin(time);
   float lamp_pos_z = 3*cos(time);
   /**** other lines of code *********/

   glUniform3f(glGetUniformLocation(our_shader.program, "lamp_pos"), lamp_pos_x, 0.0f,
      lamp_pos_z);
```

In the vertex shader, we add the following.

```
uniform vec3 lamp_pos;

int main()
{
   gl_Position = projection * view * model * vec4(position, 1.0);


   vec3 norm = normalize(vec3(model * vec4(normals, 1.0f)));
   vec3 fragment_position = vec3(model * vec4(position, 1.0f));
   vec3 light_direction = lamp_pos - fragment_position;
   float dist = distance(fragment_position, lamp_pos);
   float k = 4.0;

   diffuse_value = k/pow(dist, 2) * max(dot(norm, light_direction), 0.0f);
}
```

The second line inside *main()* multiplies the normals by the *model* matrix. Then we normalize the normal vectors into unit vectors. At this juncture, do refer figure 11.2. What we did there was to define color values at the fragments of the vertices. The remaining intermediate fragments will get interpolated. We are going to do the same here. We obtain the positions of all the fragments of the vertices of the cube. With the fragment positions we can calculate the direction of light. Now we have both unit *normals* and *light_ direction* to calculate the angle of incidence (as discussed just before section 16.2.1). We now calculate the distance between the *fragment position* and *lamp position*. Finally, we calculate the diffuse

value as the product of these two factors. You can try changing the values of k to obtain a reasonable output. Your output should be similar to figure.

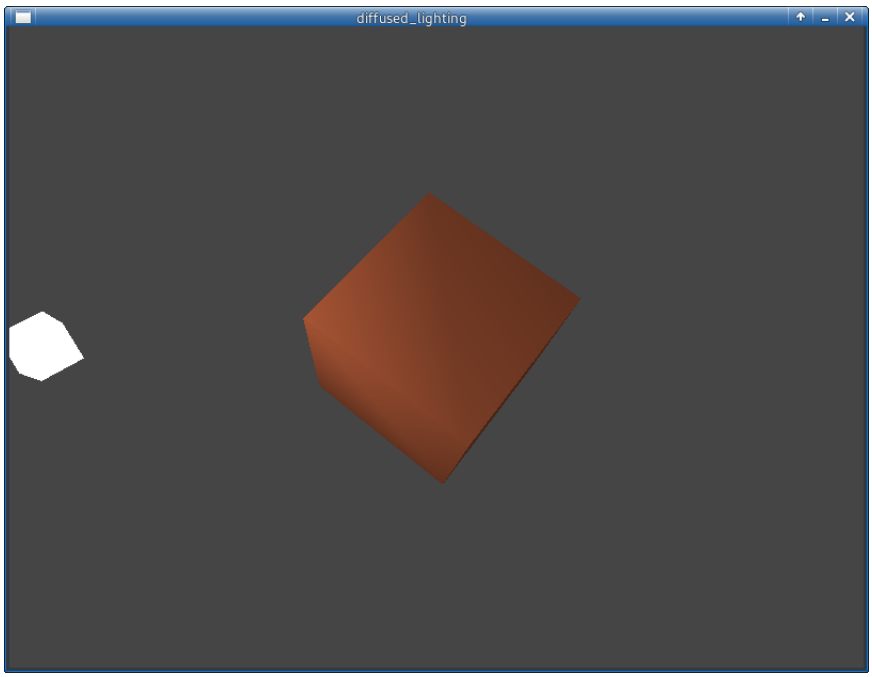


Figure 16.4: Diffused + ambient light

Source code for the same can be obtained under ch16/diffused_lighting. You can also see a video of the implemetation here.

Strictly speaking, we are not supposed to multiply normals with model matrices. Normals by definition are vectors that are perpendicular to a particular surface. If we perform non-uniform scaling operations, then the normal vectors can get skewed. This would result in vectors not being normal to the surface. However, since we are not doing non-uniform scaling, we got away with it.

Why did we calculate ambient light in *main.cpp* and diffused light in *vertex_shader.vert*?
For every iteration, vertex shader spawns as many number of threads as the number of vertices. These vertices are then passed to the fragment shader, which spawns so many more threads enough to paint all the pixels between these vertices to form objects. Thus, any calculation in the fragment shader becomes hugely more expensive than when its done in the vertex shader. Likewise, when operations are performed in the *main()* program become computationally less expensive. Ofcourse, there are several overheads such as RAM to VRAM data transfer time etc. Those need to be taken under consideration as well.

16.3 Specular Lighting

Specular highlights on any object gives a marked enhancement in realism. Remember the extraordinarily bright spots of light on the wine glass as the source of light passes by? These are called specular highlights. Just like diffused lighting, specular highlights depend on the inbound rays of light and the normal. However, in addition it also depends on the viewers angular displacement with the reflected light. Consider figure 16.5 below. LO is the incident light, OW is the reflected light that goes out into the world. ON is the normal to that *surface/fragment*. EO is the vector from the eye to the object. θ is the angle between the reflected light, OW and -EO i.e., the negative of the vector from the eye to the object.

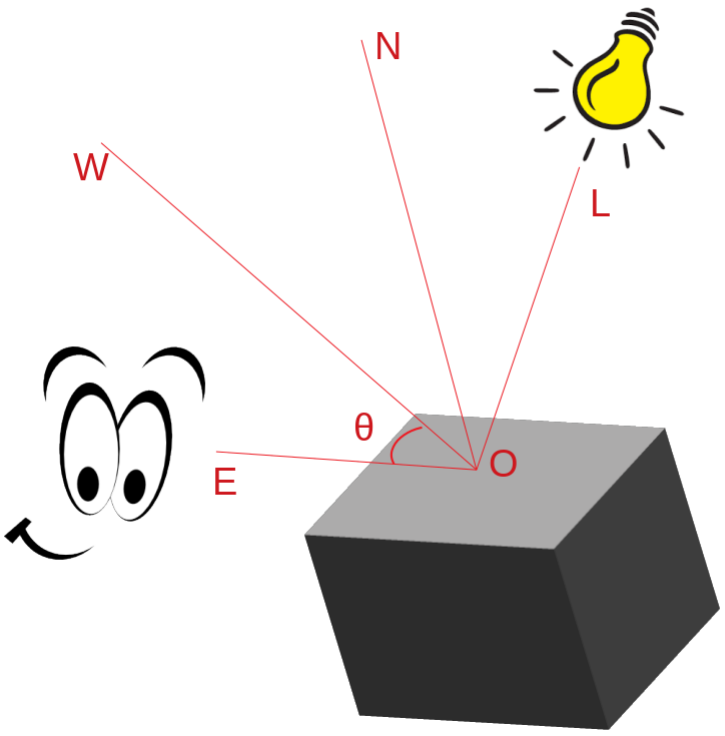


Figure 16.5: Specular light occurs due to the displacement of the view vector with the reflected ray

How do we calculate the reflected light, OW? GLM provides a *reflect* function that flips a vector around another. We pass incident light and normal to *reflect()* to obtain reflected light, OW.

```
vec3 reflected_light = reflect(light_direction, norm);
```

To calculate *EO*, we simply compute the vector between the *camera* and the *fragment*.

```
vec3 view_direction = normalize(camera_pos - fragment_position);
```

Now that we have both the view and the reflected vector, we can calculate the specular value as

$$specular\ value = specular\ strength * (view\_vector \bullet reflected\_vector)^{shininess}$$

Specular strength is a float value that can be set arbitrarily. The dot product of view_vector and reflected_vector raised to the *shininess* value causes the specular highlight. The *shininess* value can be altered for varied levels of specular highlight intensity. And the final output can be obtained by adding ambient, diffused and specular lights.

```
float specular_value = specular_strength * pow(max(dot(view_vector, reflected_vector),
    0.0), 16);
vec3 specular_light = specular_value * light_color;

color = vec4((ambient_light + diffuse_light + specular_light) * box_color, 1.0f);
```

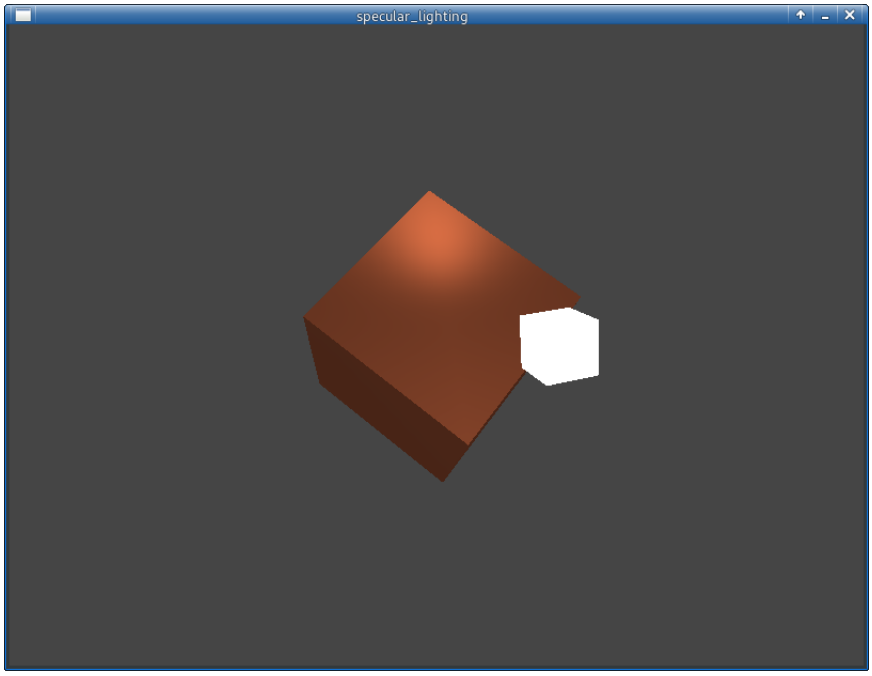


Figure 16.6: Illustration of specular highlighting

Source code for this can be found in ch16/specular_lighting. And a video of the implementation can be found here.

Chapter 17

Materials

This chapter is going to be fairly easy. In real world, not all objects reflect light in the same way. Their response to light will be different. For instance, a white eraser reflects a huge portion of ambient and diffused light while its specular component is almost negligible. On the other hand, a black marble has a high specular component. To simulate real world materials closely, we require a way to describe the material properties; specifically, their response to light. What we do here is to separate out a factor of response to each type of light (ambient, diffuse, specular and shininess). The granularity of separation also extends to the materials response to each of the RGB values.

17.1 Basic Materials

Let us get straight to coding this. We will be using ch16/specular_lighting as our base code for this.

For each material, its properties have been tabulated in http://devernay.free.fr/cours/opengl/materials.html. Let us pick the first entry - emerald. In *main.cpp*, we create variables to hold those values. As described at the bottom of the webpage, we multiply the shininess value by 128. We also pass these values to the shaders via uniforms.

```
glm::vec3 material_ambient = glm::vec3(0.0215f, 0.1745f, 0.0215f);
glm::vec3 material_ambient = glm::vec3(0.07568f, 0.61424f, 0.07568f);
glm::vec3 material_specular = glm::vec3(0.633f, 0.727811f, 0.633f);
float material_shininess = 0.6 * 128;

// ambient lighting calculations are performed in main.
glm::vec3 ambient_light = ambient_strength * (light_color * material_ambient);

glUniform3f(glGetUniformLocation(our_shader.program, "material_diffuse"),
    material_diffuse.x, material_diffuse.y, material_diffuse.z);
glUniform1f(glGetUniformLocation(our_shader.program, "material_shininess"),
    material_shininess*128);
glUniform3f(glGetUniformLocation(our_shader.program, "material_specular"),
    material_specular.x, material_specular.y, material_specular.z);
```

Since we defined diffused lighting in the vertex shader, we add a uniform.

```
uniform vec3 material_diffuse;

/***** other lines of code *********/
diffuse_light = diffuse_value * (light_color * material_diffuse);
/***** other lines of code *********/
```

And specular lighting in the fragment shader.

```
uniform vec3 material_specular;
uniform float material_shininess;

/***** other lines of code *********/
vec3 specular_light = specular_value * (light_color * material_specular);
/***** other lines of code *********/
```

The output should resemble figure 17.1.

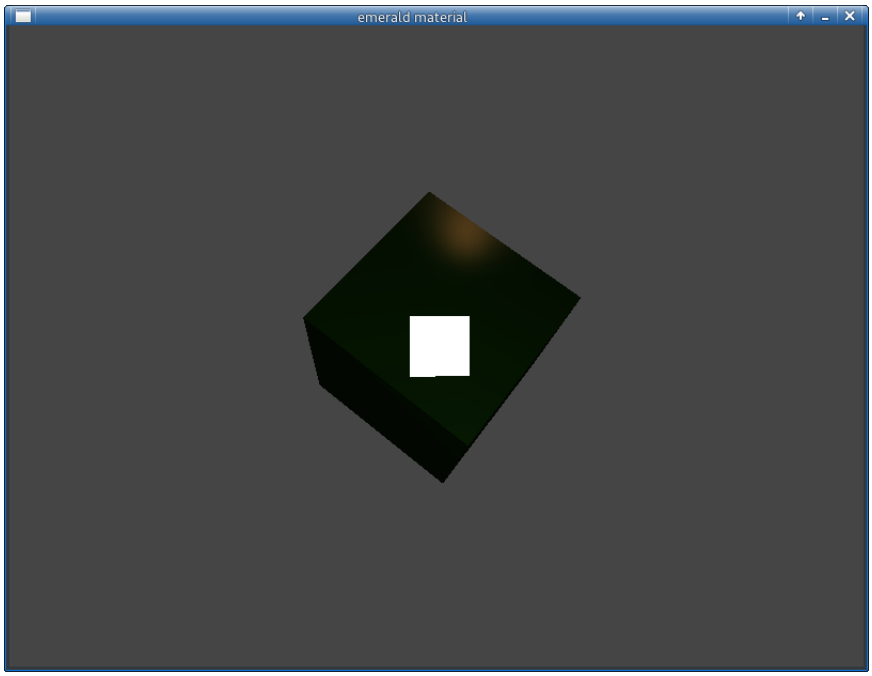


Figure 17.1: Material properties of Emerald

You can find the source code in ch17/materials.

17.2 Lighting for Materials

We described the response of lighting by materials. And uptill now we have been using pure white light which shines with full force all the time. We can also add several components to light. For instance, the nature of ambience, diffusion and specularity of the types of light.

```
glm::vec3 light_ambient = glm::vec3(0.2f, 0.2f, 0.2f);
glm::vec3 light_diffused = glm::vec3(0.5f, 0.5f, 0.5f);
glm::vec3 light_specular = glm::vec3(1.0f, 1.0f, 1.0f);

glm::vec3 ambient_light = ambient_strength * (light_ambient * material_ambient);

glUniform3f(glGetUniformLocation(our_shader.program, "light_diffuse"), light_diffuse.x,
    light_diffuse.y, light_diffuse.z);
glUniform3f(glGetUniformLocation(our_shader.program, "light_specular"), light_specular.x,
    light_specular.y, light_specular.z);
```

```
uniform vec3 light_diffuse;

/***** other lines of code *********/
diffuse_light = diffuse_value * (light_diffuse * material_diffuse);
/***** other lines of code *********/
```

And specular lighting in the fragment shader.

```
uniform vec3 light_specular;

/***** other lines of code *********/
vec3 specular_light = specular_value * (light_specular * material_specular);
/***** other lines of code *********/
```

Play around with these values. You can create really fancy effects. The complete source code can be obtained in ch17/materials_lighting_control.

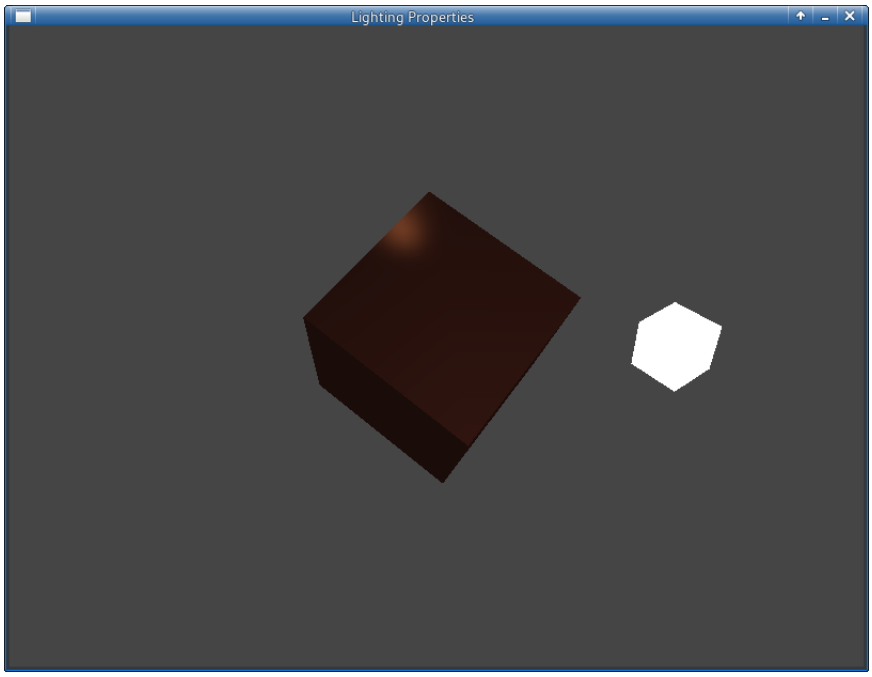


Figure 17.2: Lighting properties on materials

This chapter was a bit dull. The next chapter is going to be amazing!

Chapter 18

Assimp

This is the last and final chapter of this book. Here, we are going to learn how to import 3D models designed in softwares such as Maya, Blender, 3DS Max etc. Boxes, crates and cubes are becoming boring. We really want to render cool shapes on our AR screen. How do we do that? Using a library called ***Assimp***. Each model is made of several small primitives, mostly triangles patched up one over the other to from layers of meshes. Most often there is a heirarchy of meshes forming a tree with root node at its head. Now, there are several softwares that create these models and thus several standards exists. Some of the most popular formats include *.obj*, *.dae*, *.fbx* etc. Each of these files contain many vertices, texture coordinates, normals, material information etc. Now, we can ofcourse write our own parser for all these files but it would be a laborious task to be able to write a generic parser to support all (if not most popular) filetypes. Assimp comes to our rescue there. Assimp is that generic parser that reads these files and sorts the information into its data structures. We then take the data from assimp's data structures and construct VAO's and VBO's to store the data so that it is in a language that OpenGL understands. Thus its a two step process where assimp reads the data from a model file and the data stored within assimp's data structures are then put into OpenGL's data structures. The first step is automatically done by assimp and there is not much work to do other than pointing assimp to the object file. The second step will be the main focus of this chapter.

Before we start working with assimp, we need a bit of boiler plate code. Since we did not implement anything perfectly suitable for the job, we will use ch18/vertex_template_assimp. This code does nothing but draw a rotating wireframe cube which by now you should be well versed with.

Parsing data from an object model can be a bit tricky. The first step is to install assimp. To do that type the following on your terminal.

```
sudo apt-get install libassimp-dev
```

Now that we have installed assimp, we need to add it into our Makefile. We do that by adding *-lassimp* in the *LIBS* section of our *Makefile*.

```
LIBS = -lglfw3 -lm -lGLEW -lGL -lGLU -lSOIL -ldl -lXinerama -lXrandr -lXi -lXcursor -lX11
    -lXxf86vm -lpthread -lassimp
```

We also need to include headers to include assimp. Assimp contains three main headers, viz.,

```
// Assimp includes
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

#include <string>
#include <vector>
```

We also add *string* and *vector* methods as we will be using them soon.

```
   Assimp::Importer importer;
   const std::string& path = "<path-to-file>/3d_model.obj"; // Remember to give absolute
       path
   const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate |
       aiProcess_FlipUVs);
```

Inside *main()*, we create an *importer* object which does the import of the model given the path. Then a string *path* that points to the obj file under consideration. Remember to give the absolute path to the model. Let us start with trying to import a simple model - ch18/wireframe_suzanne/src/resources/3d_model/suzanne.obj. Suzanne is a cute little monkey head as shown in the figure 18.1.

Figure 18.1: Suzanne

```
const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate |
    aiProcess_FlipUVs);
if(!scene || scene->mFlags == AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
{
   std::cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << std::endl;
   return 0;
}
```

Importer reads the file. We set two flags *aiProcess_ Triangulate* and *aiProcess_ FlipUVs*. The first flag translates all the primitives into triangles if they are partially or fully not triangles. Remember in the textures lesson where we had to flip our textures, the next flag does that exact thing for us. We then check if the model has be successfully loaded. If this does not return any errors, we can safely assume that our model is successfully loaded and the data is stored within assimp's data structures.

Next, to parse the data out from Assimp, we need to know where and how the data is laid out. Once we load the model, assimp creates a ***scene*** object. A scene object hosts all the information about the object. Some of the following instances/methods it has are

1. mRootNode - root node handle. return type: aiNode *
2. mNumMeshes - number of meshes: return type: unsigned int
3. mMeshes - array of meshes. return type: aiMesh **
4. mNumTextures - number of textures. return type: unsigned int
5. mTextures - array of textures. return type: aiTexture **
6. mNumMaterials - number of materials. return type: unsigned int
7. mMaterials - array of Materials. return type: aiMaterials **
8. HasMeshes() - checks if there are any meshes. return type: bool
9. HasTextures() - checks if any textures are present. return type: bool
10. HasMaterials() - checks if any material information is present. return type: bool

There are some more but we wont be using them now. We are mainly concerned with meshes, textures and materials. Thus, any methods pertaining to that will be relavant to us.

18.1 Wireframe Suzanne

Our main goal in this section is to create a wireframe model of suzanne. To do that we will be needing vertices and indices. Since there are a lot of vertices and edges we will be dealing with, we will be using *STL vectors* from the *standard template library* of cpp.

```
// grabbing vertices from assimp data structures
    ---------------------------------------
std::vector<glm::vec3> vertices;
for(unsigned int num_meshes=0; num_meshes<scene->mNumMeshes; num_meshes++){
   for(unsigned int num_vertices_per_mesh=0;
```

```
        num_vertices_per_mesh<scene->mMeshes[num_meshes]->mNumVertices;
        num_vertices_per_mesh++){
       glm::vec3 a(scene->mMeshes[num_meshes]->mVertices[num_vertices_per_mesh].x,
          scene->mMeshes[num_meshes]->mVertices[num_vertices_per_mesh].y,
          scene->mMeshes[num_meshes]->mVertices[num_vertices_per_mesh].z);
       vertices.push_back(a);
    }
  }
  GLfloat *vertices_array = &vertices[0].x;
  GLuint size_of_vertices_array = vertices.size() * 3 * sizeof(GLfloat); // vertices data
     size * x,y,z values per vertex * sizeof(GLfloat)
```

The first line creates an stl for vertices. This will be our container for all the vertices of the model. The model ofcourse can have many meshes. *scene->mNumMeshes* will give us the number of meshes in the model. Our model suzanne has only one mesh (let us start simple) i.e., the 0^{th} mesh. For the o^{th} mesh, we need to know the number of vertices. This is obtained by *scene->mMeshes[0]->mNumVertices*. Now we have the number of vertices contained in the 0^{th} mesh. We then create a dummy vector, a and assign it with the x, y and z value of the vertex. Finally, we push it into our stl vector stack. Once this is done, we have all the vertex data we need. We also need a pointer to the vertex data which is obtained in *vertices_ array*. Also the size of each vertex is important as we will use them while describing the attributes during VAO creation. The next thing we need is the indices for EBO's.

```
  // grabbing indices from assimp data structures
     ----------------------------------------
  //std::cout << scene->mMeshes[0]->mFaces[0].mIndices[0] << std::endl; // this is how
     you access the first index of the first face of the first mesh of the scene.
  std::vector<GLuint> indices;
  for(unsigned int i=0; i<scene->mNumMeshes; i++)
     for(unsigned int j=0; j<scene->mMeshes[i]->mNumFaces; j++){
        indices.push_back(scene->mMeshes[i]->mFaces[j].mIndices[0]);
        indices.push_back(scene->mMeshes[i]->mFaces[j].mIndices[1]);
        indices.push_back(scene->mMeshes[i]->mFaces[j].mIndices[2]);
     }
  GLuint *indices_array = indices.data();
  GLuint size_of_indices_array = indices.size()*sizeof(GLuint);
```

For each mesh, several faces can be defined. Since all our primitives are triangles, each face is a triangle. And, each face has three indices. We gather all the indices and dump them into a stack of indices. Ofcourse, just like before we also require the data of the indices as well as its size. Remember, the number of vertices to be rendered is equal to the number of indices. Awesome! we have all that is required to render the model. For the VAO, VBO and EBO,

```
   GLuint VBO, VAO, EBO;
   glGenVertexArrays(1, &VAO);
   glGenBuffers(1, &VBO);
  glGenBuffers(1, &EBO);
```

```
    glBindVertexArray(VAO); // Bind vertex array objects first before VBOs
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, size_of_vertices_array, vertices_array, GL_STATIC_DRAW);

   // attribute 0 vertex positions
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3*sizeof(GL_FLOAT), (GLvoid*)0);
    glEnableVertexAttribArray(0);

   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
   glBufferData(GL_ELEMENT_ARRAY_BUFFER, size_of_indices_array, indices_array,
       GL_STATIC_DRAW);

    glBindVertexArray(0);            // unbinding VAO
```

Finally the draw call.

```
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); // looks way cooler if there is no
       lighting or material information.
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

You should get an output similar to figure 18.2.

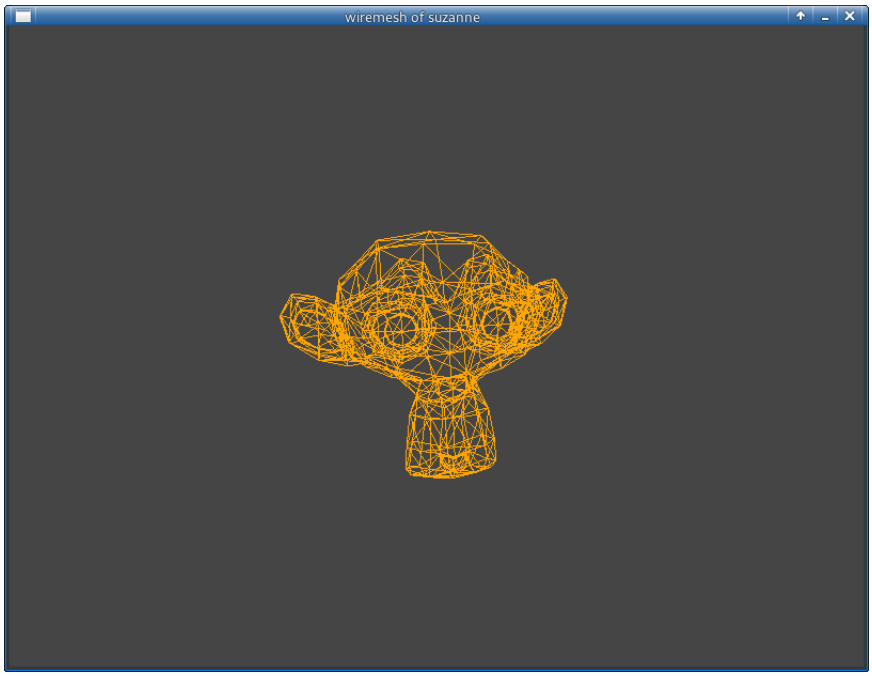


Figure 18.2: Wireframe suzanne rendered using assimp

You can find the source code for this under ch18/wireframe_suzanne. A video of the implementation can be found here.

18.2 Suzanne with Lighting

It will be interesting to see the effect of lighting on suzanne. But for lighting we cannot have a wireframe mesh. For this exercise we will combine the previous section's output; specifically assimp read with lighting from ch16/specular_lighting. Also we will observe the output of this on high poly. *High/low poly* is a term used to denote high or low polygon count for an object. A high poly object has a lot vertices and will be smoother as compared to a low poly model. Ofcourse, its about time we get a model that looks great under lighting and all the cool effects. When we say lighting, we know normals play a big role. We will first create a struct to contain our vertices+normals. In *main.cpp* add a structure named *VN* (Vertex and Normals).

```
struct VN{
   glm::vec3 Position;
   glm::vec3 Normal;
};
int main()
{
   /******** other code *************/
   // grabbing vertices and normals from assimp data structures
   std::vector<VN> mesh_data;
   VN vec;
   aiMesh *mesh = scene->mMeshes[0];
   for(unsigned int i=0; i<mesh->mNumVertices; i++) {
      glm::vec3 p(mesh->mVertices[i].x, mesh->mVertices[i].y, mesh->mVertices[i].z);
      glm::vec3 n(mesh->mNormals[i].x, mesh->mNormals[i].y, mesh->mNormals[i].z);
      vec.Position = p;
      vec.Normal = n;
      mesh_data.push_back(vec);
   }
   GLfloat *meshData = &mesh_data[0].Position.x;
   GLuint mesh_size = mesh->mNumVertices * 6 * sizeof(GLfloat);
   /******** other code *************/
}
```

We then add positions and normals into mesh_data stack. We also calculate the size of the data. Finally in the draw elements call, we modify the draw call to,

```
   glDrawElements(GL_TRIANGLES, size_of_indices_array/sizeof(GLuint), GL_UNSIGNED_INT, 0);
```

And viola, we have a great looking monkey head with a light spinning around it to bring out the specular highlights. I hope you obtained an output similar to figure 18.3. Cool isn't it?

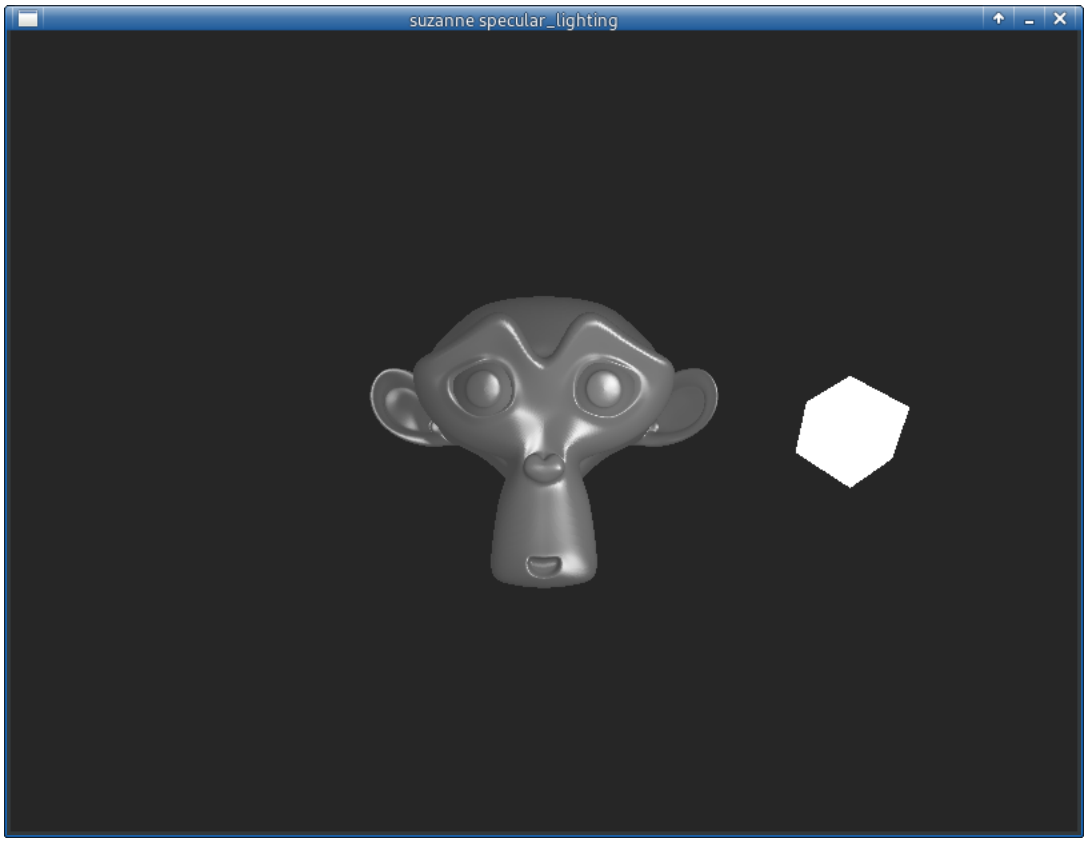


Figure 18.3: Suzanne with lighting

You can find the source code in ch18/suzanne_specular_lighting. Click here to see a video of the implementation.

18.3 AR with Suzanne

Now, our next task is to integrate ch18/suzanne_specular_lighting and ch14/AR_perspective. This is a bit of a tricky process. How I chose to do it was to make small progressions starting with *AR_perspective* code. Progressions include,

1. Simple box without textures: progression_1_simple_box
2. Box with a rotating light: progression_2_rotating_lamp
3. AR box with lighting effects from the rotating light: progression_3_lighting_effects
4. Integrating suzanne with AR (final code): AR_suzanne

A detailed explanation of the code is unnecessary as no new concepts were introduced; this section deals with only integration of various modules.

The resulting outcome is a really cool suzanne model with a rotating light around the object with ambient, diffused and specular lighting effects. A sample implementation output is shown in figure 18.4. You can find a video of the implementation here.

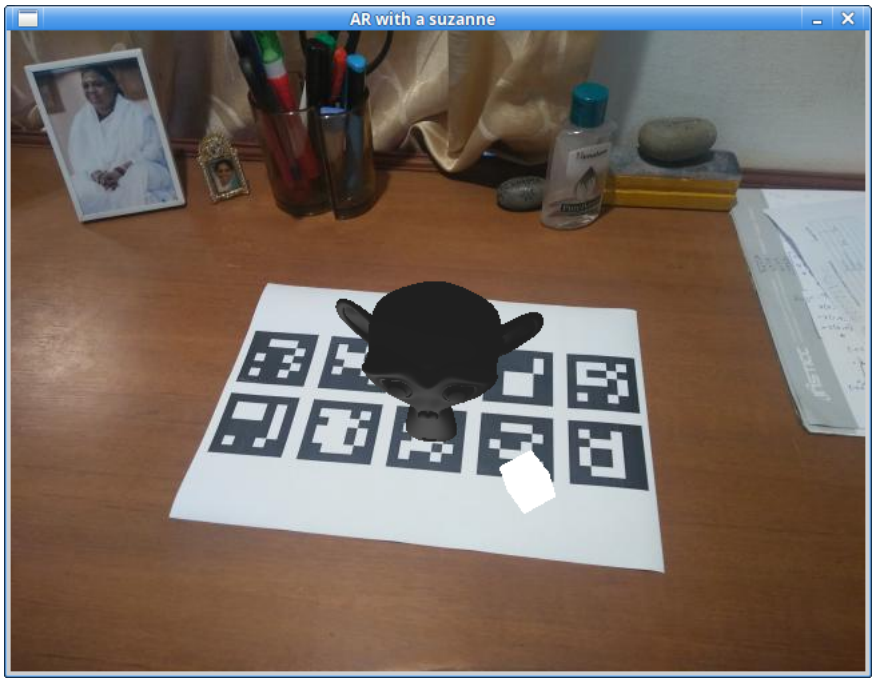


Figure 18.4: AR with Suzanne

Congratulations! I hope you have now acquired the essential tools required to create your own AR applications. Now go ahead and get creative :)