CS4670 Computer Vision: Final Report

Supasorn Suwajanakorn, Yu Cheng, Cooper Findley

This report briefly talks about our final project for computer vision with detail information on the implementation, the decisions we made and their justifications.

Introduction

Our final project, Augmented Reality 3D Graphing, aims to build a system that takes a picture from a camera, find the interested square in it, and paint some 3D patterns on top of that. The initiative was for the visualization of 3D graphs, but it could also be used in various other situations.

1. Use of existing work:

This idea was not based on an existing implementation or paper. We do intensively use openCV functions for various purposes as transformations on the images. We used the function *cvApproxPoly* and *FindExtrinsicCameraParams2,* which automatically approximates the polygon given contour information and calculates the object pose from 3D-2D point correspondences by solving for the closest rotation and transformation matrix given in class. More details could be found at the references below

http://opencv.willowgarage.com/documentation/structural_analysis_and_shape_descriptors.html?highlight=approxpoly#cvApproxPoly

http://opencv.willowgarage.com/documentation/camera_calibration_and_3d_reconstruction.html?highlight=findextrinsiccameraparams2#cvFindExtrinsicCameraParams2

2. Work separation:

Supasorn mostly worked on the algorithm to detect edges and interested planes, finding 3D-2D point mappings, solving for the extrinsic, and setting up the coordinate system for OpenGL on desktop. Yu helped on finding the plane and point mapping, and focused on localization of the algorithm on cellphone, including porting the existing GL program to ES2 and various other components. Cooper researched on abstracting the pattern from the image, and handled the generation of fans OpenGL draws on the pattern page.

3. Future work

We originally planned to encode more information onto the pattern, but did not actually perform that, as our speed of execution was not ideal on the phone, while the orientation detection was also not performing to our expectation. We decided just to use the intensity values for the height at this point, and applied homography to retrieve the intensity at the point interested.

We would also spend more time optimizing the performance on the phone, as the current frame rate, 1-2 frames per second, is not be ideal for delivery.

Details on implementation

The procedure that programs takes could be simply described as the following steps:

1. Identify the interested plane

We would need to firstly take a picture from the camera and identify the plane that we would take as the basic xOy plane. Currently, our design use the patterns surrounded by a square box as below, where the bounding box serves as the marker for the algorithm to detect the plane.

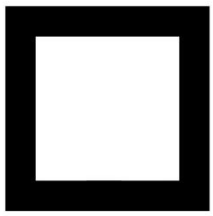


Figure.1 The bounding box for patterns

1a. Decision on type of marker

We first started with using four color circles to indicate corners, but it turned out that white balance greatly affects the color detection algorithm and there is no easy way we can control the white balance on laptop's camera that we use to do initial experiment. Another problem with circles was that when the marker or the camera moves quickly around the screen, the circle tends to get blurred out and lose sharp edges. We then tried black squares which worked better providing that the width is at least 0.8 inches. Also it's easier for users to generate a pattern inside this simple black square.

1b. Finding the bounding box

As suggested by Prof. Snavely, we first tried Canny edge detection and Hough transform. Canny edge detection worked reasonably well to extract the edges of the square, but Hough transform usually detects only perfectly straight lines even we tried a couple of configurations. In our experiment, we do not always have a hard card board to stick the pattern to, so the edges of the square tend to bend a little which usually got ignored by Hough line detection. To be able to detect bended edges, we tried approximating polygons using OpenCV's cvApproxPoly(), which required us to use the contour of the image instead of the result from Canny edge detection for our first-step features. After that, we look through all the detected polygons and decide which one is a square by checking basic characteristics such as the number of corners has to be four, the shape has to be convex. We also managed to detect a very flat square, one that is almost parallel to the viewing direction, by adjusting some parameters.

1c. Identifying the orientation

To find the orientation, we detect the intensity of the area near four inner edges and align the pattern so that the darkest area is always on the bottom. This could be implemented by adding a little back box near the inner edge of our bounding box.

2. Camera Calibration

Once we have detected a square in the scene, we determine the four corners and use it to solve for camera's extrinsic. In order to correctly remap 3D back to 2D in OpenGL, these four corners' coordinates have to be converted to OpenGL's clip coordinates or canonical cube first. The four 3D points that correspond to those four corners are given by (-1,-1,0), (-1,1,0), (1,-1,0), (1,1,0). The intrinsic matrix we use is.

F	0	0
0	F*a	0
0	0	1

Where a = 1.333

However, we do not know the focal length and the real focal length of the camera is not very useful because the unit in the intrinsic is pixel. So we tried to first approximate the range of focal length then do binary search by hand to minimize the re-projection error, and we computed F to be around 2.

Once we have the camera's extrinsic, we use it as a model-view matrix in OpenGL. For OpenGL's projection matrix, we need to add two special rows to preserve relative z-coordinates (used in z-buffer test) and homogenous w coordinate. Otherwise, OpenGL wouldn't be able to do depth test. There are many correct projection matrices that can preserve relative z-coordinate, but we pick a simple (0, 0, 1, 1). So Z' = -1 - 1/Z. The original Z has to be negative so that the OpenGL's clip coordinates will be in the range -1 to 1.

F	0	0	0
0	F*a	0	0
0	0	1	1
0	0	-1	0

Where the last row is an augmented row to preserve relative z-values or depth value

For OpenGL ES on the phone, however, there is no matrix stack, or the implementation of model-view and projection. So we simply multiply Projection x Model-View to produce the transformation matrix in the shader of OpenGL ES 2.

3. Detecting the information on the pattern

For extracting the pattern inside the square, because we know the coordinates of the four corners, we simply find the homography and inverse warp from the camera image to get the pattern inside. Alternatively, we also consider calculating the inverse of Intrinsic x Extrinsic which can solve the same

problem if we fix the z-coordinate of the input and may be faster if we can find a closed form which we didn't have enough time to optimize.

For generating the actual GL fans, we take in the information by the inverse warping, therefore getting the 3D coordinates for all the points that we want to draw. Then we generate a quadrilateral for each four adjacent points in the matrix, and push it to the GL pipeline. The desktop version runs fine with ease, while we had more troubles with the ES 2.0 system, which will be discussed later in part 5.

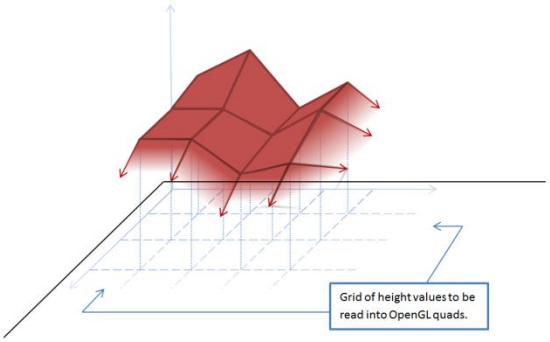


Figure.2 Quadrilaterals based on point information

4. Drawing with OpenGL

Our program runs on OpenGL windows, and in order to display the image from the camera, we basically copy the data from IplImage to a texture in OpenGL and setup orthographic projection, draw a quad with given texture, and clear the depth bit in OpenGL so that the new shape can be drawn on top of it. Then, we would simply draw the quadrilaterals described above, while changing the viewport to the appropriate location.

5. Integration with phone

After the desktop version is working, we performed the integration from desktop version to cellphone version. ES2 uses a pretty different mechanism than the normal GL, and we saw some problems on migrating, especially on change of viewport and rendering the shades, mainly due to the unfamiliarity to the ES2 frame. In the ES2 structure, all shaders are customized, and compiled at runtime. While providing the user with a lot of freedom, we had troubles identifying the correct shader functionalities to write. The lack of sufficient examples/references made the situation a little worse. Fortunately we did not require anything more complicated than a triangular fan and direct light rendering, and we got it to work rather soon. We performed some optimization, but as the frame rate is so low, we are thinking of adding more optimizations if appropriate.

Results and Summary:

1. Brief Summary

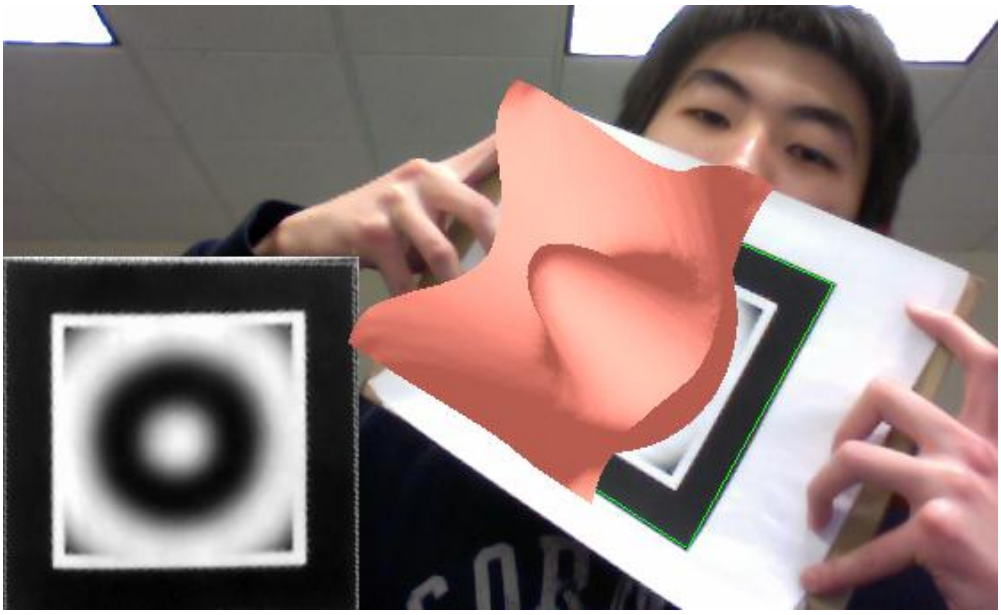
We are relatively satisfied with the performance of the desktop version, where we could steadily reach the maximum frame rate constraint by the webcam. Although the orientation detection is not perfect, it works reasonably well under good lighting conditions without glares. However, the cellphone version does not quite meet our expectation yet, where frame-rate being the biggest problem. Currently we have roughly 1-2 frames per second, where we would really like to see something like more than 10-20 frames per second. Also, on automatic mode, the camera produces rather dark/noisy photos compared with some smartphones we personally use, and that adds a little difficulty to the recognition algorithm.

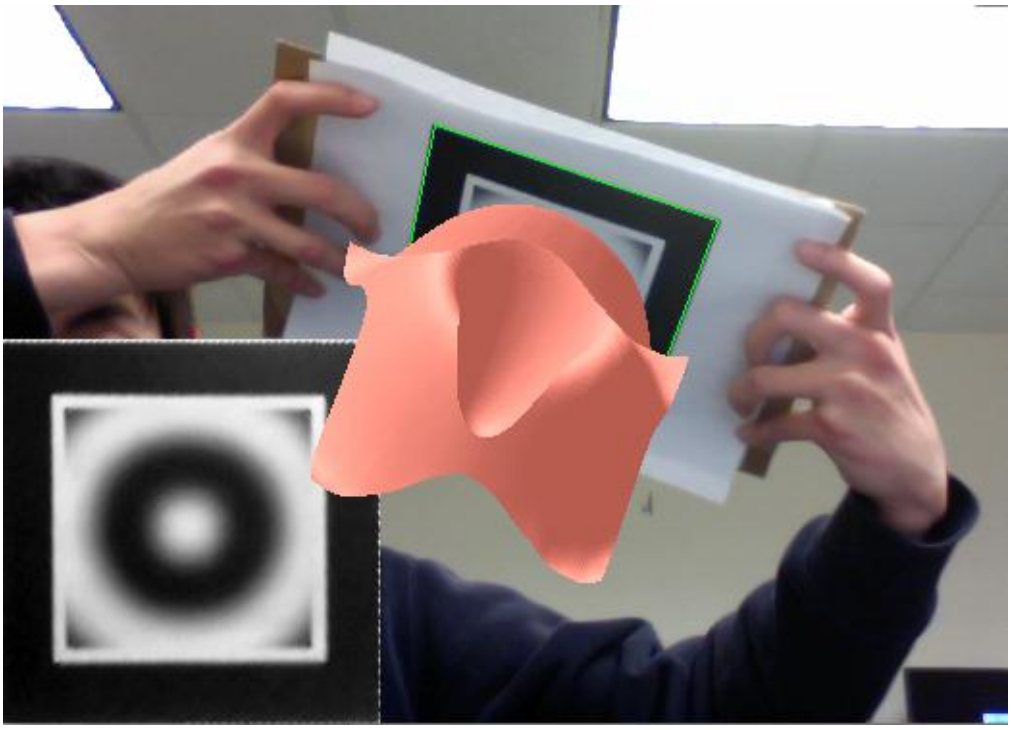
2. Results

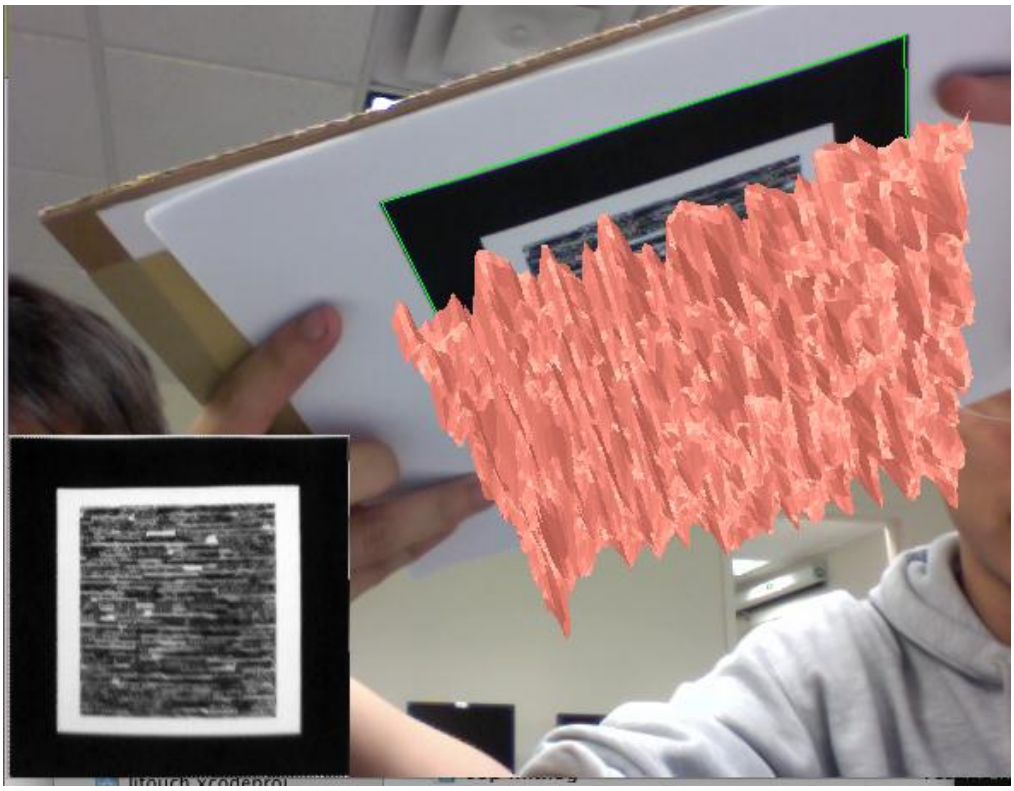
Here are some screen shots for the desktop version and phone version respectively.

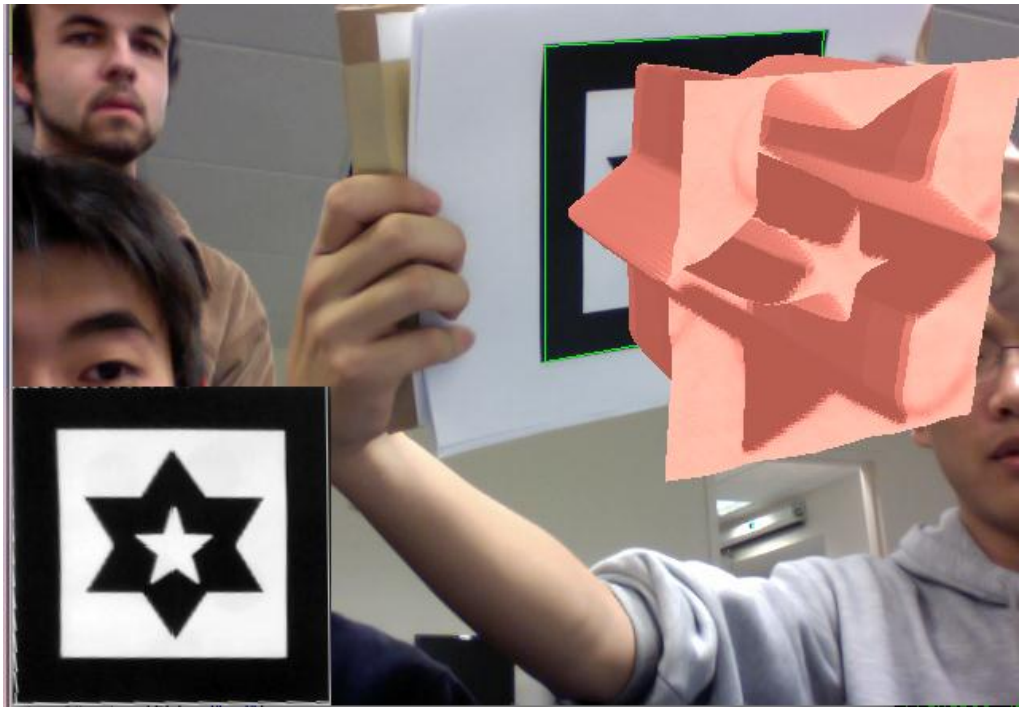
2a. Desktop version

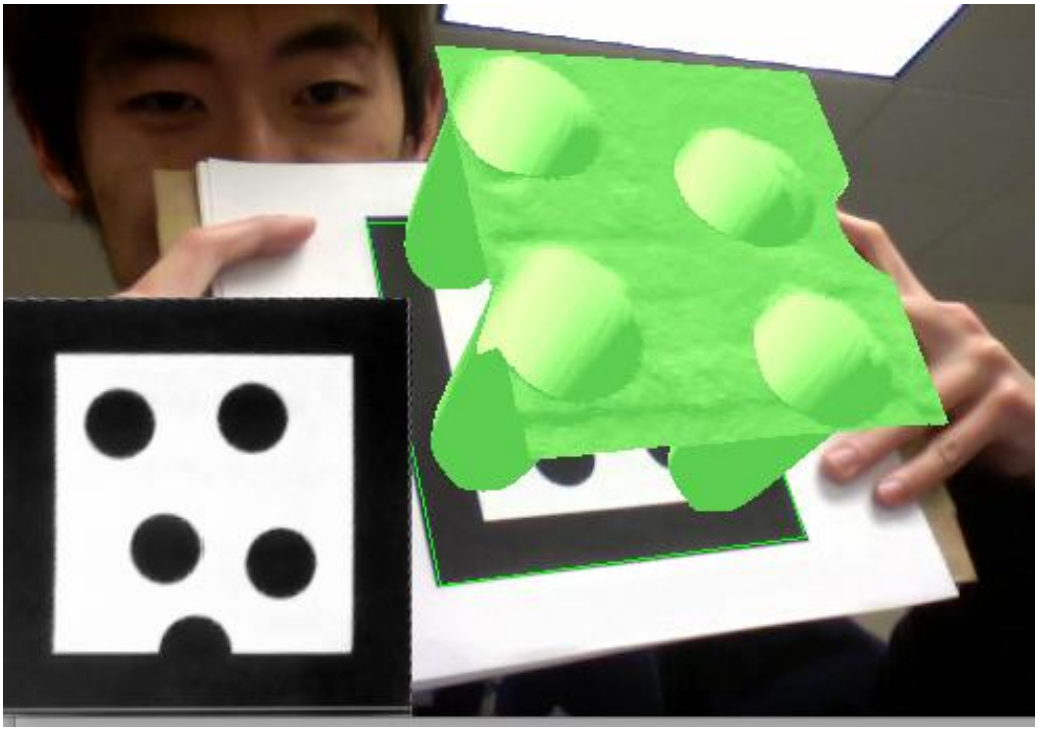
Overall, we are rather satisfied with the performance of the desktop version. The examples of screenshots show pretty promising results. Also notice is that the desktop version is running at 30 frames per second, the limit of webcam. We had some problems with the glare caused by the patterns printed by the laser printer under certain lighting conditions, and that counted for the reason why we were holding the pattern upside down for some of the examples.

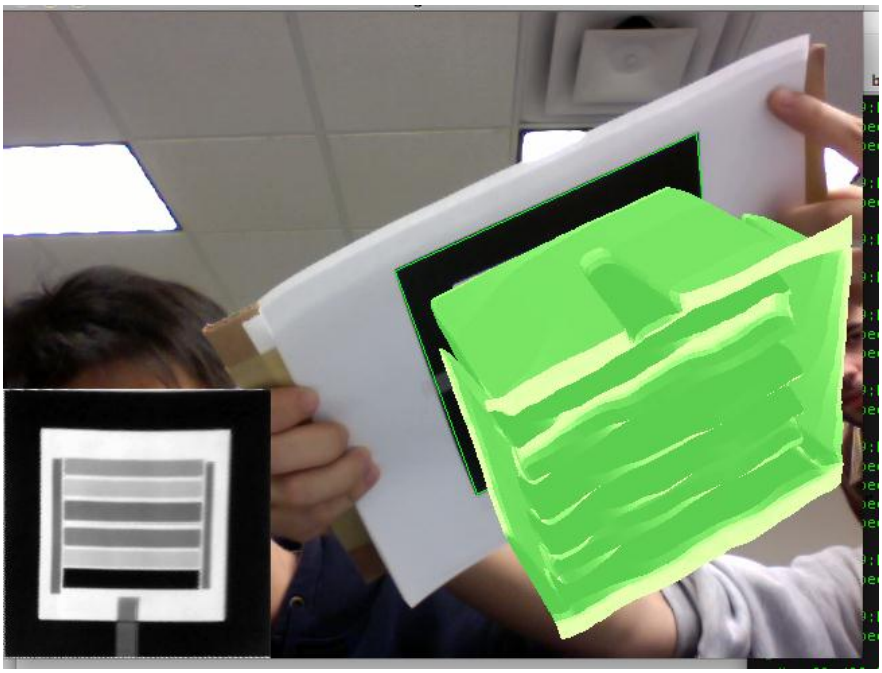






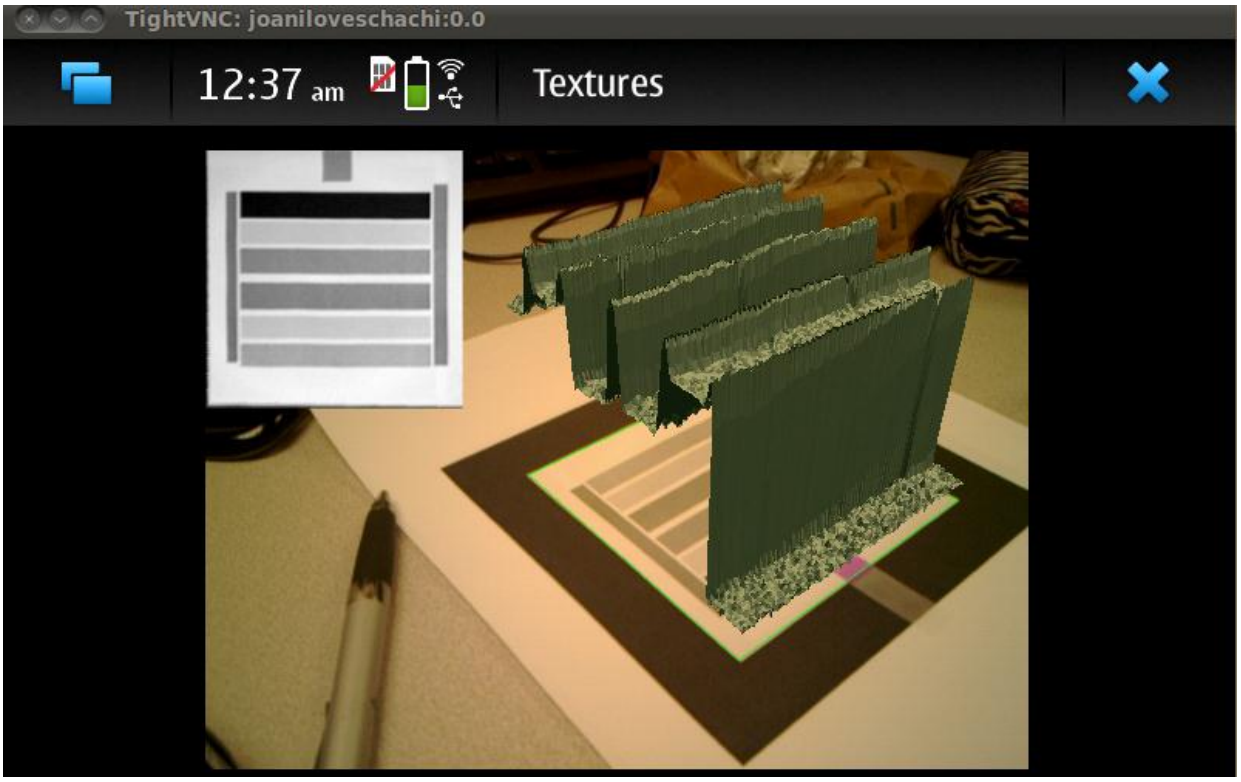


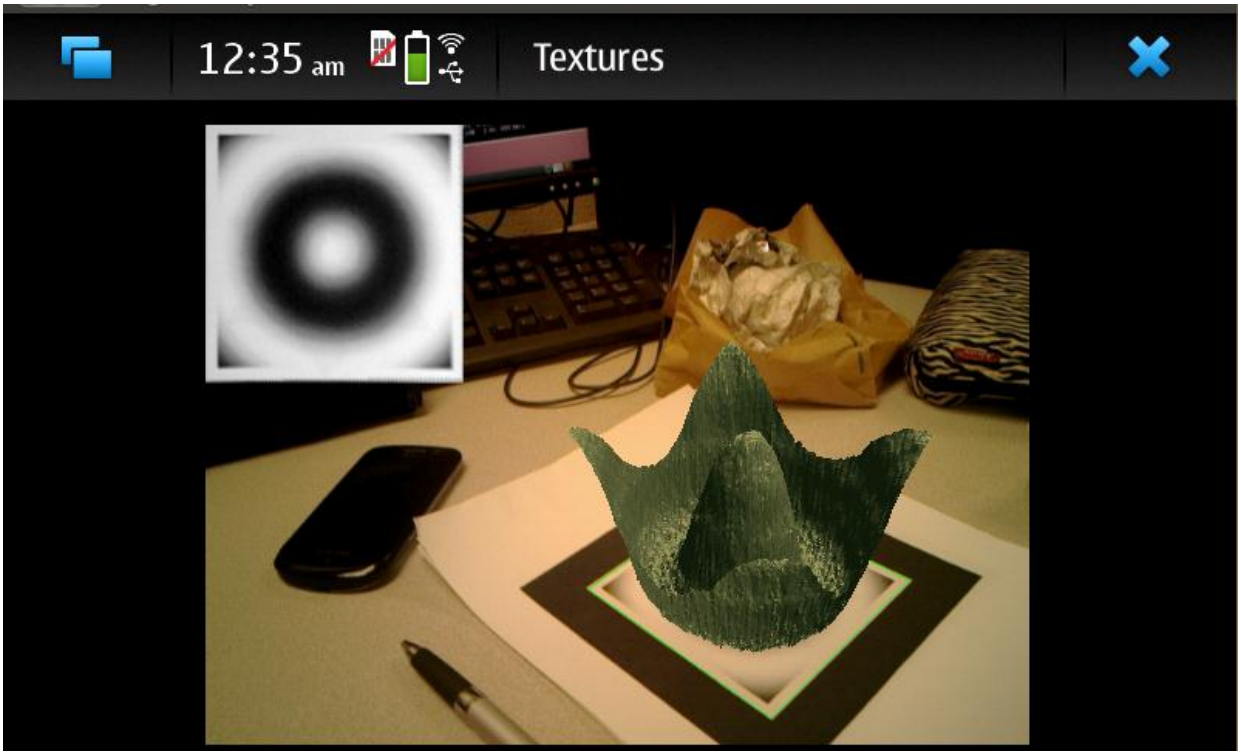


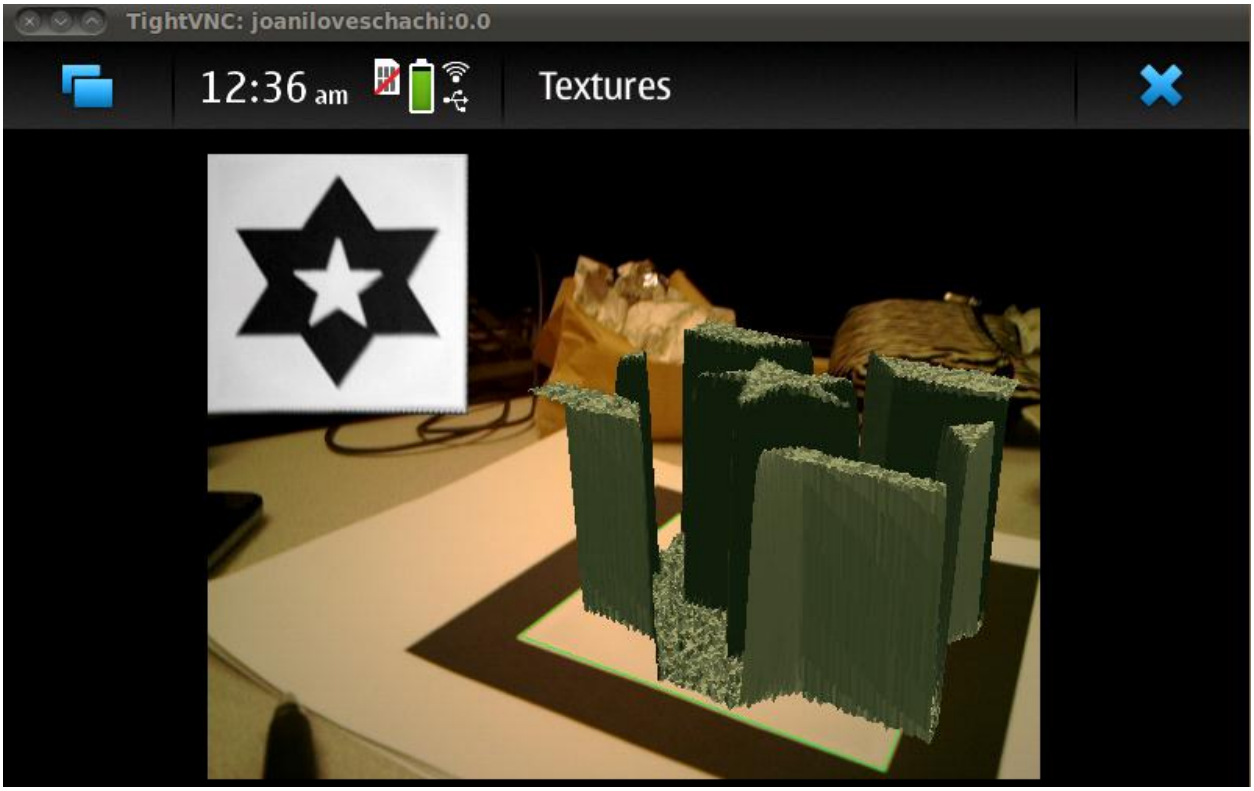


2b. Cellphone version

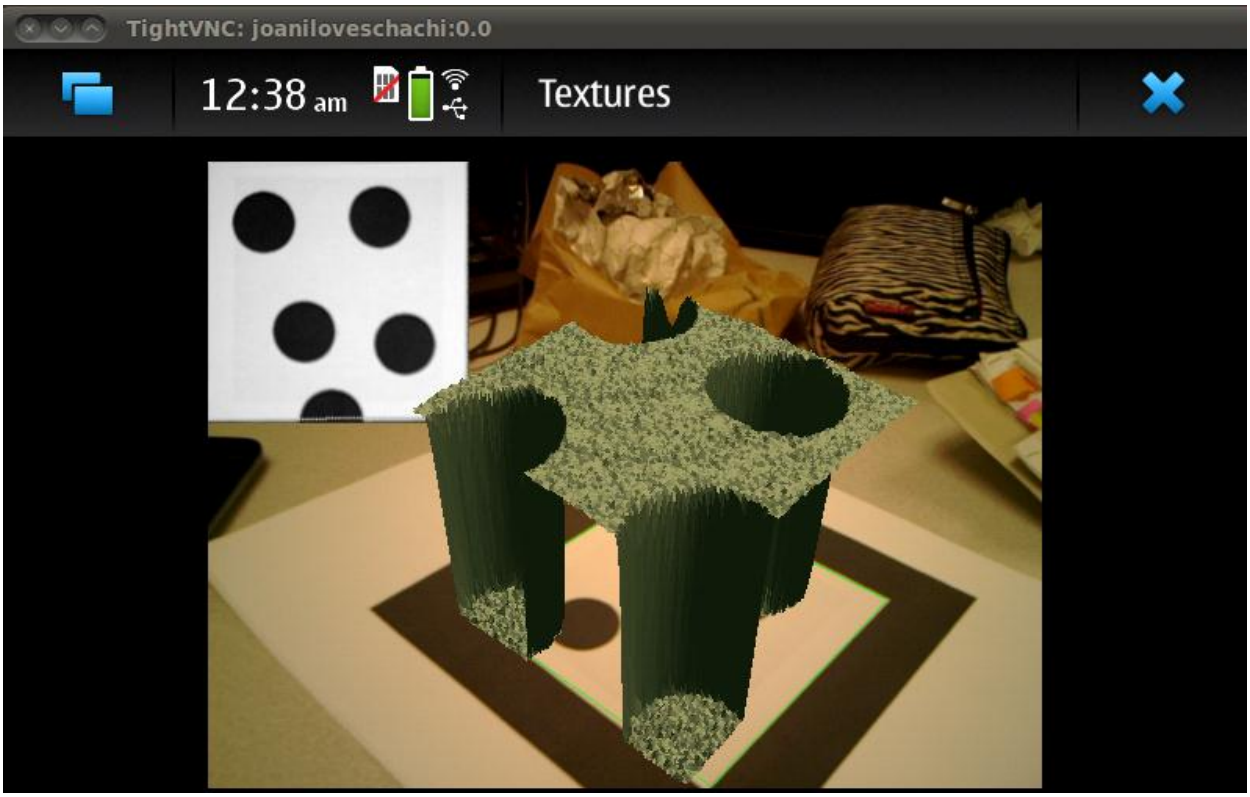
For the cellphone, we are relatively satisfied with the results, but not quite for the performance side. The resulting pictures look relatively close to the desired results, where the shapes and shades look normal. However as mentioned earlier, the system could only run at roughly 1-2 frames per second. Further optimization would be desired and it is one of the things we would like to work on in the future.

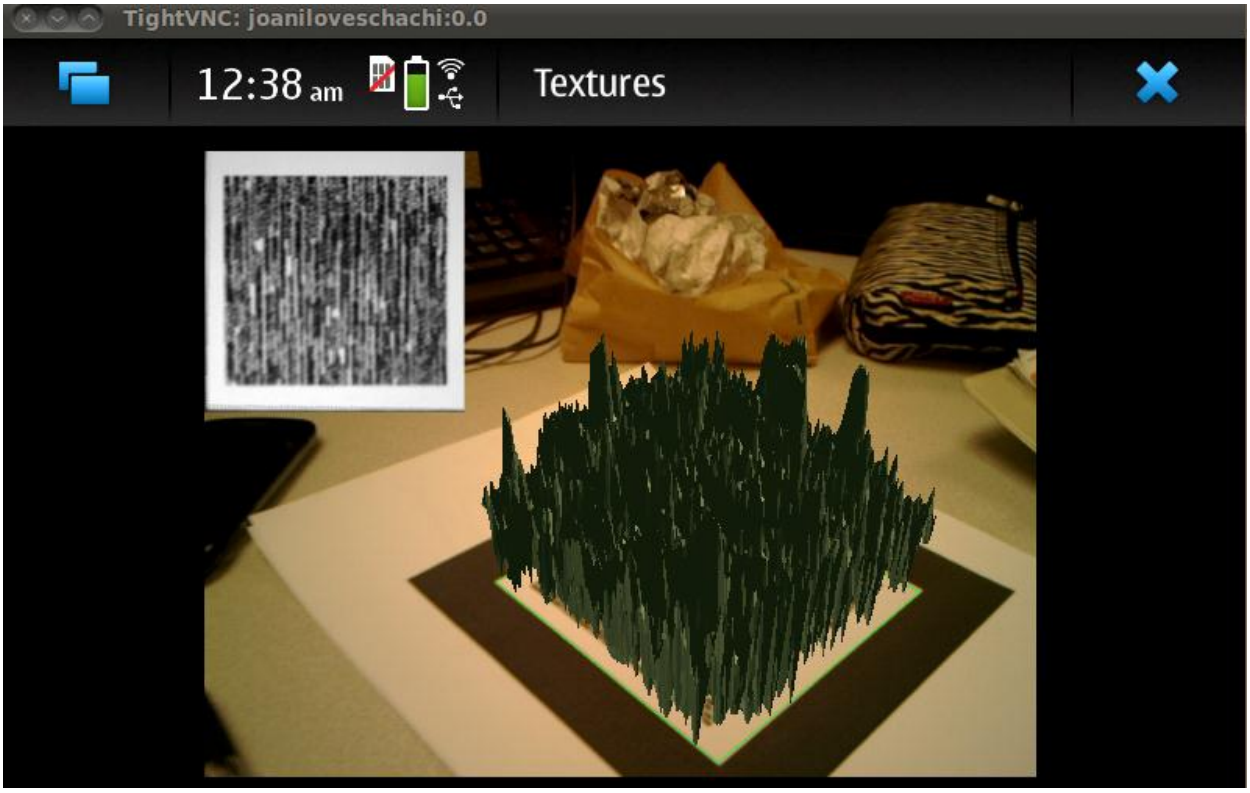



12:35 am
Textures


TightVNC: joaniloveschachi:0.0
12:36 am
Textures


TightVNC: joaniloveschachi:0.0
12:36 am
Textures


TightVNC: joaniloveschachi:0.0
12:38 am
Textures


TightVNC: joaniloveschachi:0.0
12:38 am
Textures