PROJECT 2
AUGMENTED REALITY GAMING SYSTEM

OVERVIEW
For this project, you will implement the augmented reality gaming system that you began to design during Exam 1. The system consists of a computer, projector, camera, and whiteboard or chalkboard. A human player can compete against the computer in traditional pencil and paper games like Tic-Tac-Toe by drawing on the board, while the computer projects images:

![System Components Diagram]

The project consists of four main tasks: establishing a homography between the projected image and the camera image, rectifying the camera image, determining whether the board is obstructed, and classifying the marks made by the human player.

TASKS
1. **Establishing a homography.** Before you can implement interesting games, you must first establish a homography between the camera image and the projected image, as shown in Figure 2. Your system must establish the homography automatically, with no input required from the user.

There are many ways to accomplish this, but here is the strategy I would recommend:
Figure 2: Projected and camera images

- Start by projecting a blank image onto the board. After 10 or so frames have been captured by the camera, pick the last one and designate it the reference image.\footnote{Why capture 10 frames? You want to wait until the auto-exposure and other settings of the camera adjust before grabbing a reference image.}

- Then, project a sequence of circular dots (one at a time) onto the board, laid out in a regular grid, where the game play will occur.

- For each dot image, take the difference (\texttt{cv2.absdiff}) of the current dot image and the reference image.

- Threshold each difference image to get a binary image, and use the blob-finding code from previous assignments to find the centroid of the dominant blob in the frame.

- The homography $H$ is the mapping that maps the blob centroids to the original, projected coordinates.

\textit{Tip:} To eliminate oddities from auto-exposure, people in front of the camera, etc., you may want to make sure that the dominant blob position is stable (i.e. does not change much over 3 or 4 frames) before moving onto the next position.

2. Rectifying images. Once you have the homography, you can use it to map the camera image to a rectified image of the game board. Again, you will want to wait several frames after projecting the game board before setting the reference rectified image, to allow auto-exposure to adjust.

My rectified image in Figure 3 is black, because I used a chalkboard. Note that both the dark areas and the light areas in the image have varying brightness. Neither one of these issues is a problem, because in general we will be operating on relative (i.e. differenced) intensity values, not absolute ones.
3. Determining board obstruction. Your system should not only be able to classify a mark on the board as to its shape and position (e.g. X’s and O’s for Tic-Tac-Toe), but it should also be able to determine whether there is an object (i.e. a hand) obstructing the board from view.

As shown in Figure 4 and Figure 5, there is a substantial difference between marks on the board, and objects obstructing the board. Since the human player is drawing fine lines on the board, eroding the thresholded difference image by a small structuring element should completely remove anything that is a valid mark; any image that has many foreground pixels remaining after thresholding and erosion is likely to be an obstruction.
4. **Classifying marks.** Once you have determined there is a mark on the board, the system should classify its position and type as required by the game. For games like Tic-Tac-Toe, you may find it useful to use the `cv2.matchTemplate` function to perform normalized correlation of a template at all possible locations within an image. It returns the correlation of each result (i.e. a score from -1 to 1, for each location).

![Image against which templates will be matched. This image was obtained by dilating the thresholded image from Figure 4 with a small structuring element.](image)

Possible templates to match X’s and O’s are shown in Figure 7. They were created using OpenCV `cv2.line` and `cv2.circle` functions, followed by a `cv2.GaussianBlur` operation.\(^2\) Matching with multiple line strengths and template scales should help cope with the fact that different players will mark at slightly different locations, sizes, and line widths.

![Templates for matching X and O shapes. Note variation in both size and line width.](image)

In the case of Tic-Tac-Toe, you may wish to constrain your template matching to a particular sub-region of the rectified image (i.e. the 1/9\(^{th}\) of the difference image which has the maximum number of thresholded pixels).

The class of the mark can be determined by hysteresis thresholding: let \(X_{\text{max}}\) denote the maximum normalized correlation for all X templates, at all locations within the image, and \(O_{\text{max}}\) the corresponding maximum for all O templates.\(^3\) Pick two thresholds \(\tau_1 > \tau_0\). If \(X_{\text{max}} > \tau_1\) and \(\tau_0 > O_{\text{max}}\), then the mark is determined to be an X; if \(O_{\text{max}} > \tau_1\) and \(\tau_0 > X_{\text{max}}\), then the mark is an O; otherwise, the mark is not classified as either one.

On the other hand, if you are playing the classic “dots and boxes” game, you need to use the

\(^2\)In my experiments, the blur seemed to help slightly, but is not strictly necessary.

\(^3\)Note that in Python, the max of two scalars is simply `max(s1, s2)`, and the maximum element of an array is `numpy.max(A)`. 

---

4
difference image to classify horizontal vs vertical lines instead, which might benefit from its own specialized move detection approach, such as searching each horizontal and vertical edge for the brightest area in the difference image.

Once a move is detected and classified, the rectified reference image can be updated to continue play. Again, you may want to wait for auto-exposure to adjust before continuing (i.e., enforce a delay of a small number of frames after recognizing a move).

GAME PLAY

This lab handout so far has only consisted of a technical description of the capabilities required for your project. Your job is to combine these capabilities into a fully functional game program. Your program may allow the computer to play against a human, or serve as a referee between two human players. In either case, your program should be robust enough to detect invalid or unrecognized moves, and recover accordingly. It should also be able to successfully recognize whatever criteria end the game (e.g. in Tic-Tac-Toe, a full board or three in a row), and declare a winner.

If you program a computer player, I will not be grading you based upon its skill – this is a computer vision course, not an AI course! All I ask is that the computer makes legitimate (if not particularly smart) moves.

There are several things you can do to add extra “cool factor” to your project. Examples include visual effects like highlighting the player’s marks. At minimum, you should provide the human with some visual feedback through the projector indicating what the computer is doing as either referee or player.

In the end, you don’t need to play Tic-Tac-Toe – I’m open to other games, as long as they involve the four tasks outlined above. It would be a bad idea to try to recognize too many types of marks, however. While discriminating between X’s and O’s is relatively easy, recognizing all possible handwritten letters is much more difficult. If you’re interested, you can find a compilation of pencil-and-paper games here: http://www.papg.com/. My suggestions from that site are: Sim, Obstruction, Go-Moku, 3D Noughts and Crosses, Dots and Boxes, and Cram/Domineering.

EVALUATION CRITERIA

Your project will be evaluated by the following criteria (percentages are approximate):

a. **source code and video (10%)** - Turn in the full source code for your project, including any additional files needed to compile and run it. I expect your code to be neatly indented and reasonably commented. You should also submit video of your system running (note you can just dump the camera stream to a file, or record video with a separate camera/cellphone).

b. **homography (15%)** - Automatically calculate a homography with no human intervention.
If human intervention is needed, you will lose points here.

c. **rectification (15%)** - Display the rectified camera image once the homography has been applied. This can be on a segment of the board not used for gameplay, or on another screen.

d. **determining obstruction (15%)** - Determine whether its view of the board is obstructed, and indicate to the user if so (i.e., by projecting a message outside of the board area).

e. **mark classification (15%)** - Discriminate between two types of marks using normalized correlation, with reasonable accuracy. It’s generally better to have false negatives than false positives (i.e., failing to recognize an X as such is better than mis-labeling it an O).

f. **“cool factor” (15%)** - Provide some visual feedback to the human user, and be interactive somehow. Be creative!

g. **demo (15%)** - We will schedule demonstrations to be held outside of class after spring break. During the demo, your group will be assessed on the criteria above. You should be prepared to address the criteria listed above, and answer questions about your group’s approach.

Instructions for submitting your code and video online will be provided. Since you are demonstrating your code live, no written report is needed.