Facilitating Interaction with Projection-Based Environments using a Laser Device and Webcam

Ben Brown and Mark Philip

Abstract

Using just a laser pointer, our system allows a user to remotely interact with a laptop that is hooked up to a projector (and currently being used to project). The only requirement is that the laptop's webcam must have a view that constitutes the entirety of the projected image. By pointing a laser pointer at the projected image, the user is able to interact with a virtual whiteboard as if he or she were using a mouse directly connected to the laptop. We accomplish this by first using an automated calibration step to map the webcam's view of the projected image to the laptop's screen. Once this is completed, our system detects whenever a laser pointer is pointed at the projection screen, and executes an action based on the location of the point existing. We have prepared a demonstration of the application's ability to track a laser point on a planar surface with a simple drawing app which allows the laser pointer to be used as a remote paintbrush. Images of the setup as well as the projected screen detecting the laser point are provided in the results.

Background

This project started as a hackathon idea. In applications such as augmented reality and virtual reality, we are always trying to discover new methods of interaction with our computers. Laser pointers are often used in virtual reality as a method of extending one's own hand or pointer finger. What if you could do that in real life?

This is not the first implementation of laser tracking as a method for computer interaction, but we believe that it is the most accurate, and our automatic calibration is something that we have not seen used before for, at least for this purpose. Our pipeline involves two major parts, the automatic webcam-to-projector calibration and correctional warping, and the conversion of a warped image to an (X, Y) coordinate system of the window/screen, enabling us to interact with the computer's screen at any location.

We found several papers on tracking a laser point using a camera, however every implementation ignored very warped perspectives and they (for the most part) seemed inaccurate and outdated. One paper [Kuo and Tang] was written in 2006 and utilized a specially-made lens on the camera to blur everything and accentuate red circles (the laser point). Their implementation has a calibration step which involves displaying 20-25 points on the screen, and using a least squares polynomial approximation, they calculate a function to map the camera image sensor to the projector screen. Kuo and Tang likely utilized this implementation due to the fact that they had an old webcam, and barrel distortion was apparent. We are hoping that modern webcams built into laptops are more professionally built today and will have less apparent offsets, and extreme accuracy is not necessary.

Our implementation first automatically detects the warp between what the webcam sees and the original image that is projected. This is done by first creating four points, one in each corner of the image, sending that image to the projector, and then extracting those points by watching their change over time. Those four points are then used to create the transformation matrix, which is then used to dewarp the webcam image. The dewarped image can then be checked for the presence of a laser. If a laser is found, the corresponding point is drawn to the image.

Previous attempts at detecting change over time in a scene have involved some usage of cv::absdiff() or some form of background modeling and foreground masking. After evaluating our use-case, we decided the former did not quite do what we wanted. If we were to use foreground masking, we would have to retrain the model every time the "background" changed, which would be happening quite often in most cases (e.g. projecting a paint application). Instead, we wanted a method that would be background-agnostic. To this end, we implemented our own algorithm for finding the corner points using absdiff(), based on their change over time.

Our calibration works as follows: four circles are placed near the four corners of the window, which is fullscreen and projected onto a flat wall with the laptop's camera facing the projector screen. All four points must be fully visible to the laptop's camera for the entire duration of the application's life. However, it is important to note that this is the only requirement; as long as all four points are visible and detectable, the camera can be any reasonable distance away from and any angle relative to the projection. These circles are then extracted from the camera's captured image and used for mapping the projected image back onto the original image.

Once the corner points have been detected, calculating the transformation matrix is trivial. The

cv::getPerspectiveTransform() method is a widely-used function, and has its roots in linear algebra. It essentially computes a transformation matrix based on the four source and destination points we feed it, i.e. it maps the source and destination points to each other and computes a function that describes this mapping. The next body of work involved using the dewarped webcam image to detect the presence of a laser, and if present, drawing on the appropriate location on the projected image. We ended up using a precalculated map for warpPerspective() to save on CPU cycles and thereby increasing performance quite drastically [4], since we are using that function on every single captured webcam frame for laser detection and drawing.

Our implementation detects the laser faster and more accurately than any of the implementations we have found, just by using simple thresholding. We were originally hoping to find the laser point using a histogram comparison between the unwarped rectangle of camera capture and the actual window that's being displayed. We were planning that the biggest difference between the images should be the laser point. However, we had to revise this method after some tests. We also liked the idea of subtracting the images because this would allow for multiple lasers, although it would have required us to run a color calibration between the camera and the screen.

Approach Used

The first problem we came across for this project was figuring out the screen dimensions and fitting a fullscreen image (the virtual whiteboard) to a secondary display. This is done to make the image displayed by the projector be full screen, to make calibration easier for both us and the user, as well as allowing for minimal interaction from the user to be necessary to use the program. The intended use of the application involves starting it up while already connected to the projector screen in display-extended mode (rather than mirrored), and simply following the prompts displayed on the laptop's screen. This way, we could communicate to the user how to manipulate their laptop by displaying on the laptop's screen what the webcam is currently seeing to ensure the projector screen is fully visible. We wanted to automate the fullscreen process but this actually proved to be a fruitless gesture, as without QTKit (MacOS window drawing framework) new versions of MacOS had no way to get screen dimensions using first-party frameworks. Once all four calibration dots are visible by the webcam, the automatic calibration begins.

For the calibration, we originally used a Hough transform to locate crosses in the four corners of the screen. After experimentation, we moved away from that method and switched to using circles instead, with the intention of using the Hough transform specifically for circles. During the course of experimentation, we determined that using Hough transform was actually quite difficult due to the fact that the circles, when warped due to perspective, become oblong ellipses, often with quite a large amount of variance in color within the same circle. A Hough transform turned out to be useless in the cases where the viewing angle difference was too great. Instead, we ended up using the color of the circles to our advantage. By configuring the circles to change color over time, we can run an active calibration on the four circles by tracking this change across several frames.

Specifically, we switch the circles every frame between pure red and pure blue. We capture 18 of these frames from the webcam at a time and process them together. Between each consecutive frame, we use cv::absdiff() to get the absolute pixel change between the two frames. From this diff image, we create a binary image that is set to "on" for pixels in the diff image that have a combined red and blue channel value above our threshold value, and "off" otherwise. This binary image is calculated between each consecutive frame, and the regions that show up in all 17 binary images can safely be assumed to be the flashing red and blue circles that were placed in the corners of the projected image. In other words, we detect consistent change in the red and blue channels of 18 frames taken from the webcam.

The 17 binary images are effectively combined into one image with a bitwise "and". From there, we run a median blur (kernel size = 5x5) across the image to get rid of most single-pixel noise, and then dilate to make the blobs in this image more prominent. From there, we can run cv::connectedComponentsWithStats() to extract the locations of the blobs from the image, and then only keep the four largest blobs, which should be the four corner circles, although not in any particular order.

These blobs are then sorted into clockwise order starting at the top-left (to match the order of the original drawn points). We then used OpenCV's implementation of calculating a warp perspective matrix based on four points to then transform the warped image into a standard

rectangular image. The reference points (onto which to map these newly-extracted points) are the originally-created points that were drawn onto the image that gets projected. This essentially creates a transformation matrix that allows us to undo the warp that the webcam "sees". From here, the dewarped image can then be used to extract the location of the laser's dot, if present in that frame.

The second part of our pipeline occurs after the initial warp points have been determined. Once we know how to perspective warp our image of the projector screen, we then correct this warp using another OpenCV function, cv::warpPerspective(), which takes each warped frame we take from the user's webcam and flattens it to display normally. Using the rectified image, we determine the location of the point of the laser, if found.

To get rid of noise and highlight the laser blob, we first use an elliptical kernel (6x6) to dilate the rectified image. Next, we use an elliptical kernel (12x12) to apply a morphological opening operation to the image.

Using the cleaned-up rectified image, we use connectedComponentsWithStats() to extract all the blobs, and then keep only the largest one. This largest blob must also be larger than a minimum threshold area (we used 500 px^2). Based on its size in comparison to the projector frame, we can then determine if this point is a laser point or just noise. If it is in fact a laser point, we draw a dot (or line depending on the number of dots and laser velocity) of a specified color pertaining to the laser point on our whiteboard matrix, denoting that the user had wanted to paint on that spot. This process repeats for every frame.

Experimentation

Before we even got into testing the final project, we had to devise a testing environment and pipeline to ensure that each stage in our project worked independently. This also allowed us to work separately for a majority of the project and just meet up occasionally to hook things together and tweak values. This test driven development was not something either of us had done before, especially for a computer vision project that relied heavily on the webcam and a specific setup (a projector in a classroom with a nearby podium). Especially for the calibration step, we did not want to spend a lot of time in the classroom testing every iteration of the code and trying new things. To augment this, we utilized a secondary class that would mimic the webcam. Instead of pulling a new frame from a CV capture object, we just inputted a created matrix which faked what the webcam might be seeing.

For instance, our calibration step works by displaying four flashing circles which are placed on a window, and should be projected onto the external projection screen. However, in the absence of a projector, we fed the calibration tool the matrix used for displaying the calibration points instead. Once we knew this worked in a perfect scenario, we applied some random noise and perspective warps to the calibration display matrix, to mimic a laptop being off to the side. Only when this part was working reliably did we actually plug into a projector for the first time.

Similarly, for detecting the laser point, we utilized fake display matrices with noise and slight perspective warps. This did not help us tune our thresholding (which ended up being mostly specific to the webcam being used), however it did help us make sure that our laser tracking worked before official testing. We also spent some time testing our near-completed application using a standard flat screen television, however we found out that the reflective screen made the laser near impossible to detect. We were able to use it for testing our calibration step, but we had to go to a classroom to fully test our application.

By the time we had gotten to an actual projector to test, it was just a matter of tweaking our threshold values. These values pertained to the size of the dot to look for, as well as the brightness cutoff to reduce noise and therefore false positives. We also found out that using a difference between the matrix representing the whiteboard and the rectified image did not really help, and it just proved easier to make sure the laser was a different color than any of the drawing colors. Actually, after tweaking our threshold values, we found out that the camera saw the laser point as nearly entirely white, which meant that it would be easy to spot no matter which color was on the board already.

Once we tweaked our thresholding values slightly, the pipeline worked without a hitch. Drawing was a bit imprecise, but we fixed that quickly by drawing lines instead of circles when the distance between the points was short. This distance is a global variable, and could be potentially adjusted based on the user's preference.

Finally, once we were up and running with testing our application on a real projector, we noticed that the warpPerspective command was running quite slowly. It is important to run this step every frame before finding the

laser point, but something had to be done because one of the slowest parts of our system ended up being the dewarping. We found a blog post from 2015 [4] that demonstrated how to create the map generated by warpPerspective beforehand. After generating two matrices -- a prewarped X transformation matrix and a Y transformation matrix -- we only then had to remap the camera frame with those two matrices to produce the intended dewarped image of the projection screen alone. This proved to be much faster and resulted in a much higher framerate, with no observable drawbacks.

Discussion and Analysis

The hardest part of this project was the automatic calibration step. We started off looking for Hough lines, then Hough circles, but each implementation failed as soon as the input was skewed in any way. Therefore we had to determine a calibration method that would be agnostic of perspective. To reduce noise as much as possible, we used time as a variable to let the webcam "watch" the calibration circles flash and then used the frame difference to determine the corner blobs. This method was much better than any of the other ones we had tried, and while it took a bit longer (approximately one to three seconds) than any of the static calibration techniques we tried, it was by far the most successful. However, there is currently no way to distinguish a false positive from a real one. This is among a few changes we would make if we had more time.

Currently, the application is very simple in actual presentation. This is due to the fact that we underestimated the amount of time and work that automated perspective warp calibration and testing would take. We did make a plan for a fully-fledged application with more usability, but those plans slowly moved out of scope as the perceived difficulty of our pipeline increased. The original application of this tool is to mimic mouse clicks and drags on the user's screen. While we do believe that this is possible, it was moved out of scope because we both use different computer environments. Ben uses a MacBook, while Mark has uses a Linux environment. We found that mimicking clicks on both systems *is* possible, but we would need to write it differently for each environment, not to mention the potential complications involved with attempting to make the application truly cross-platform by throwing Windows into the mix.

Due to technical difficulties with generating fake clicks, we decided to focus on a whiteboard drawing application. Of course we wanted this to be a beautiful-looking application with all the bells and whistles, but time was our enemy. Instead, we decided that creating an application that showed a single laser point being tracked, which in turn places a single red dot on that location of the screen would be sufficient to show the merits of the system we had designed. The feature we most wanted to implement was drawing in the same color as the laser point, however this proved to be nearly impossible as the laser point only showed up as bright white to each of our webcams. Furthermore, we would have liked to support multiple lasers on the screen simultaneously, however again time became an issue and we decided to focus on locating a single dot very consistently rather than multiple dots only fairly-consistently.

For each of our webcams, we had to adjust the threshold values slightly (size of the dot to look for as well as the brightness cutoff to reduce noise). We did have ideas to make this part of our automatic calibration. The proposal was to display a small outline of a circle on the projection screen (after the corner points and perspective warp had been determined) and to ask the user to point the laser inside of the circle for a specified duration (five frames for instance). From this, we could determine some quick thresholding values that would be more accurate for the consumer's setup. Another idea would be to automatically adjust the threshold values based on the results from connectedComponentsWithStats() or based on the output after our processing for highlighting blobs and the laser point. However, once again time became our enemy as we rushed towards the final submission. In the end, simple value tweaking worked for each of our laptops. These values seemed to be almost completely depended on the webcam itself, as testing the same laptop on multiple different projectors did not seem to make much of a difference.

Results

In the end, we were both very please with what we had created. The final application can automatically dewarp the webcam's view of any projector screen, and then we can reasonably find the laser point and draw remotely on any projection screen. Of course our results

were more favorable with more accurate thresholding values, but because of our test-driven development mindset for this project, by the time we got to actually testing on a projector, the code was mostly working completely. The close proximity of the drawn points to the external laser point was actually a nice surprise for both of us.

Our drawing application worked quite nicely. It was an interesting way to interact with a computer. Neither of us had used a laser pointer to interact with a screen, and it was something fun to play with for both of us. There were several drawbacks to using the laser pointer as an electronic interaction tool. The biggest drawback so far was the fact that the laser is not visible when not active. This means that as both a clicking tool and a drawing tool, the laser was not ideal. It ended up being a bit of a gimmick but in the end, it was a fun project to work on and we learned quite a bit. Two images from our final project are shown to the right (Fig 1 and 2).

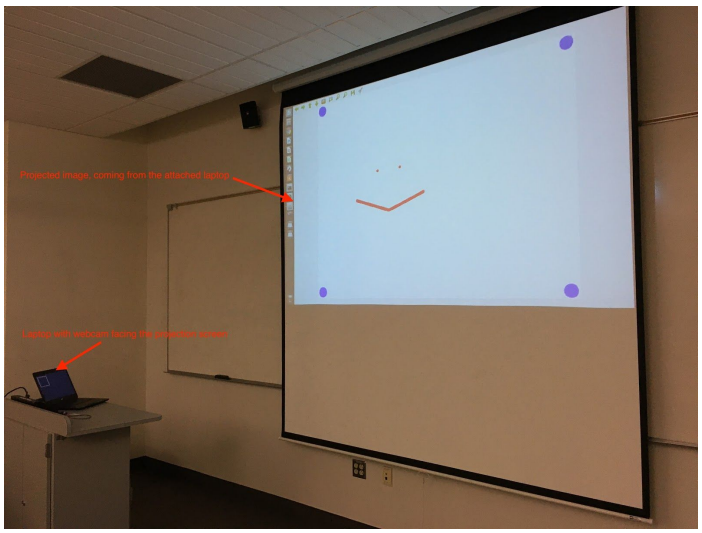

Fig 1: The completed setup, with the laptop on the left projecting the drawn image on the right

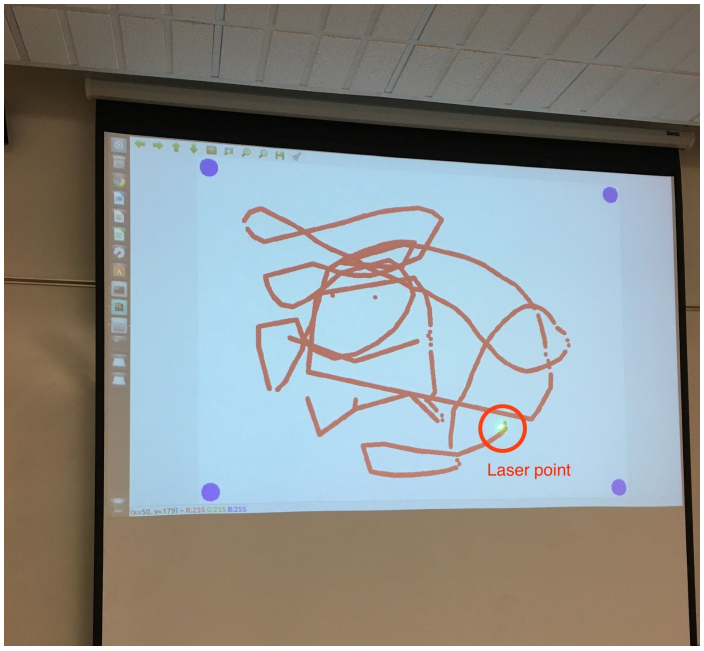

Fig 2: The projected screen in the midst of drawing from the laser point.

Conclusions

We learned quite a bit from this project. By far, the most useful thing we learned was the importance of test driven development for computer vision applications. This allowed us to work independently and ensure that each phase of the pipeline worked, even when our processes and algorithms changed. That is, even though the final results were the same, building a test environment that mimicked the projection screen allowed us to change every part of the internals of our code while still enabling iterative testing at each stage.

We also learned more about the internals of Hough transforms (even though we did not end up using it), and perspective transformations. In the end, it was the automatic calibration system that we were the most proud of. It has applications much further than detecting a laser point. In a sense, we have created a way to link up a external projection screen or television with a computer using just the webcam, and that is a much greater feat. We definitely had some fun and experienced many struggles during this project but we are both proud of the finished product.

References

1. Hough, P. V. (1962). U.S. Patent No. US3069654A. Washington, DC: U.S. Patent and Trademark Office.

2. Le, H., Doan, M., & Tran, M. (2010). Webcam-Based Laser Dot Detection Technique in Computer Remote Control. *Journal of Science and Technology*, 48(4), 73-81. Retrieved from https://www.researchgate.net.

3. Kuo, W.-K., & Tang, D.-T. (2007). Laser spot location method for a laser pointer interaction application using a diffractive optical element. *Optics & Laser Technology*, 39(6), 1288–1294. https://doi.org/10.1016/j.optlastec.2006.07.013.

4. Russell, M. (2015, July 6). Rom's Rants. Retrieved April 20, 2018, from http://romsteady.blogspot.in/2015/07/calculate-opencv-warpperspective-map.html.