Twist Your Fingers: A Hand Shadow Game

Yixin Wang
Stanford University
450 Serra Mall, Stanford, CA 94305
wyixin@stanford.edu

Chenyue Meng
Stanford University
450 Serra Mall, Stanford, CA 94305
chenyue@stanford.edu

Abstract

In this project, we designed a hand shadow game in a VR environment built in Unity. To capture the shadows made by hands, we use Intel RealSense camera to acquire RGB-D images and manage to segment hands from background with Sample and Filter method. The hand gestures will then be transmitted to Unity through a TCP socket. Unity renders the segmented hand as texture on a plane to interact with users.

1. Introduction and Motivation

Hand shadow is an ancient form of art, which requires people using their hands to cast shadows of animals or objects by obstructing lights from behind. Mimicking shapes of a bird, dog and other animal figures has been a great entertainment for family and friends, which delights both children and adults [1] [6]. Unfortunately, as a traditional and old-fashioned form of games, hand shadow game has been less popular since the late 19th century when electricity became available to homes. Because shadows are greatly and clearly defined most by candlelight, the appearance of light bulbs and electric lamps has limited the growth of this game. Therefore, hand shadows were more common in earlier centuries.

Nevertheless, the emerging interests and developments in the field of virtual reality (VR) could have been a savior to the ancient hand shadow game. As VR technology has been more and more promising [9] [2], we have the chance to build a standalone virtual environment, which deprives interference from outside light sources, and make hand shadow game achievable within.

Motivated by this idea, in this project, we will create a hand shadow game with a ViewMaster HMD and an Intel RealSense RGB-D camera. To clearly illustrate the game setting, it will be easier to explain with a solid example, shown in Figure 1.

In this specific example, the user will be asked to play the shadow of a crab with hands. The scene of this game

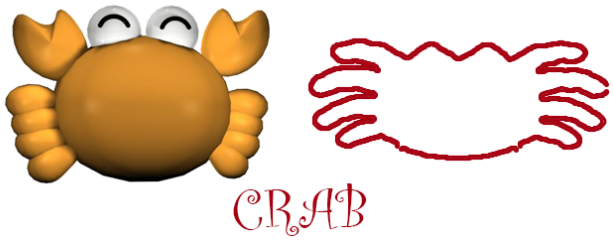

Figure 1. Illustrative example of our hand shadow game

consists of two major parts:

- A 3D model of the target object/animal (crab on the left). This model will serve as a hint to our users.
- The expected contour of the target hand shadow (contour of a hand-shadowed crab on the right). This contour will be a reference for our users to mimic.

When a user poses his/her hand(s) in front of the RealSense camera, the following steps happen.

(1) The camera will capture both RGB and depth data, which are used to distinguish users hand from background. Since we do not have much prior knowledge or expectations of what the background could be (a colorful wall or a non-static scene), it is essential for us to find a real-time and robust approach to extract user's hand(s) from the unknown background, which will be elaborated in Section 2 and Section 3.3.

(2) After being segmented, user's hand(s) will be displayed onto the HMD screen so that user can actually see his/her hand moving and modify hand gestures accordingly. The goal of this game is to make user try to move his/her hand into the expected contour (the right part of Figure 1) and fit the shape of it. With this step, we ensure that users hand is of fixed size on image plane, so that no further detection or scaling is necessary in step (3).

(3) To determine whether or not the user is making a correct hand shadow, compare segmented hand image with ground truth shadow. If they are sufficiently similar, the user moves forward to the next level. There are 5 levels (including a tutorial level) in total.

The organization of this report is as follows. In Section 2, we will discuss more about image segmentation and the current situation of hand shadow game. In Section 3, we will elaborate how we build the whole pipeline and how the scenes in Unity interact with the RealSense camera. Section 4 shows a qualitative evaluation procedure of hand segmentation and our game.

2. Related Work

2.1. Image Segmentation for Gesture Recognition

Segmenting the hand out of complex background is a hard problem. Some people use Fully Convolutional Neural Network to do image segmentation [7]. It can achieve very good results but is computationally expensive (requires large training dataset, lots of training, and GPU support), and cannot be done in real time with our current resources. It is even harder for mobile devices. On one hand, computation of neural network requires several million parameters and cannot be done locally. On the other hand, transmitting hand image to a remote server to compute via mobile network introduces delay. Thus Convolutional Neural Network is not a good option for our real-time game.

Most real time algorithms require user to place his/her hand in front of a white wall, and uses RGB value to determine hand contour directly. This can be done fast and easily. However, forcing the user to sit in front of a white wall to play this game may sacrifice user satisfaction. Some real time algorithms require user to fix camera position and take a photo of the background, so that background can be subtracted and hand contour will be revealed. However, this method cannot be used in HMD scenario, because users head is going to move. Another problem with most current approaches of hand segmentation is that they are used for gesture recognition, and thus do not need to be super accurate because down-stream classifier for hand gesture will take care of this offset. For our game, however, we need segmentation to be accurate, since we want users to twist their fingers and make the exact shadow we want.

We consider two real-time algorithms as suitable candidates for our scenario. The first candidate is Normalized Graph Cut [10]. The idea is to treat the entire image as a graph, and to break links that have low affinity, so that the image is cut in such a way that similar pixels in the graph remain in the same sub-graph. Each cut is "normalized", meaning that each cut has its own cost, so that we can avoid the algorithm from cutting the image into very small components. The second candidate is Sample and Filter. More specifically, we first sample hand pixels and construct a color profile of the hand. Then we look for pixels with the same color profile as hand, and use median filter [4] to suppress spurious detection.

2.2. Hand Shadow Game Apps Today

There are few hand shadow games apps to our knowledge. The most prevalent Android app is "Hand Shadows Puppets". A screen shot is shown in Figure 2. Most of these games feature an image or video to show users how to make a hand shadow. There is no real-time capture of hand, and few interaction with user. There are similar apps on iOS of similar form. There hasn't been any VR games of this kind to our knowledge, which makes our project a fresh and fun game idea to try out.

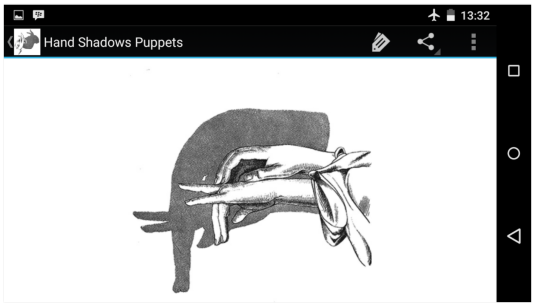

Figure 2. Screen shot from "Hand Shadows Puppets" game app.

3. Methods

3.1. Overview of Pipeline

The pipeline of our project is shown in Figure 3. It follows a server-client model which communicates through TCP socket.

Before game starts, we start a Python server which in turn starts RealSense camera driver. Then the server side begins to capture RGB-D images and performs hand segmentation using Graph Cut or Sample and Filter method, which will be discussed in Section 3.3. Meanwhile, the server listens on client's connections. The communication is asynchronous, that is, whenever there is a client request, the server has a frame prepared to transmit, rather than starting to compute the frame at that time. This makes user experience smoother by saving the time used for image processing. Both RealSense camera and hand segmentation uses 640×480 frames, which are down-sampled to 80×60 to allow for faster transmission. The reason of choosing socket communication instead of file I/O is that, socket is far more efficient because it bypasses slow disk read/write, and that it doesn't have the overhead of resolving read/write conflicts. For each frame, the server also compares user hand contour

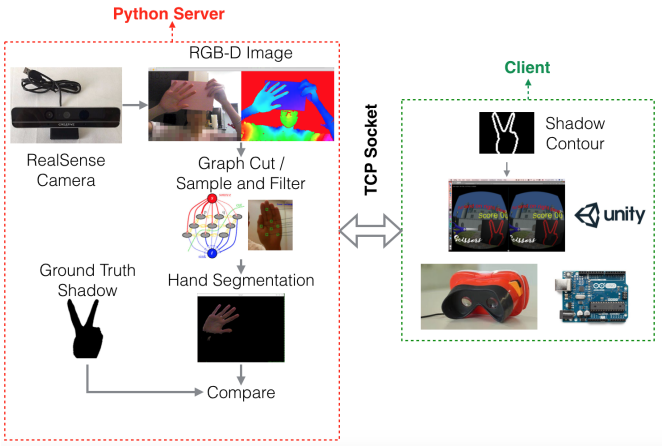

Figure 3. Pipeline of Hand Shadow Game: hand segmentation and scene rendering.

with ground truth shadow, determines a score and whether the shadow of game level is successfully completed. This information is sent along with segmented hand image to the client upon request.

Unity program acts as client side. As it renders each frame (in the `update()` function), it requests a real-time segmented hand image (80×60 PNG) from the Python server and renders it as texture on a plane. TCP sockets are chosen over UDP sockets because it eliminates the need for recovering from lost packets, and that communicating to local server is already pretty fast. Users can see their hands moving in the VR world in real-time, enabling him/her to decide how to move his/her hands to complete the hand shadow. Meanwhile, user's score is rendered real-time in Unity scene. Achieving a score higher than threshold moves the game forward to the next level.

The hardware setup for our project is shown in Figure 4. We stick VRduino and RealSense camera onto the head mounted display. We added head-straps to free user's both hands. However, the device is quite heavy so it is not very comfortable to wear.

3.2. Image Acquisition

We use an Intel RealSense RGB-D Camera to acquire images. Depth sensing is quite good in 10cm to 1m range, encoded in 16-bit integers. We use librealsense on Mac OS X for camera capture[1]. We visualize the RGB image and Depth heat map in Figure 5.

[1]Open source library for Cross-platform camera capture for Intel RealSense F200, SR300 and R200, can be accessed here https://github.com/IntelRealSense/librealsense.

Figure 4. Hardware setup for VR headset.

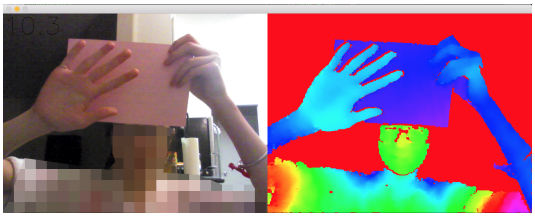

Figure 5. Camera capture: RGB image and Depth heat map.

3.3. Hand Segmentation

3.3.1 Depth Threshold

The Depth Threshold method is very simple but effective. It treats any pixel with depth between 10cm and 40cm as hand region, and sends the RGB data of this region to Unity. The results of this method are shown and discussed in Section 4.2.

3.3.2 Sample and Filter

The drawback of Depth Threshold method is that the hand contour it generates is not very accurate, which will be further discussed in Section 4.2. Sample and Filter method is based on Depth Threshold method, and aims exactly to resolve this problem. An illustration of this method is shown in Figure . It consists of the following steps.

(1) The image is transformed to HSV space (H is 0-360° and S and V are each 0%-100%.), in order to rule out illumination effects.

(2) Raw hand region $\mathcal{H}$ is determined using Depth Threshold method (Section 3.3.1).

(3) Hue values are sampled from pixels in raw hand region, we calculate its mean μ and variance σ, and consider hue values within the confidence interval $(\mu - 2\sigma, \mu + 2\sigma)$ as hand color profile.

(4) Refine the edge areas of raw hand region.

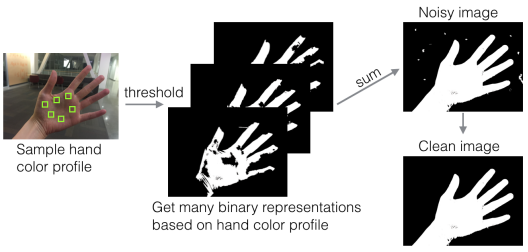

Figure 6. Illustration of Sample and Filter algorithm.

We consider pixels that are within 5 pixels to the left/right/above/below the hand region. The extended hand region is denoted as $\mathcal{H}'$. Each color in the hand color profile produces a binary image, which is intersected with $\mathcal{H}'$. Then we sum them and run nonlinear median filter [4] to get a smooth and noise free binary representation of the hand.

(5) Finally the refined hand image is sent to Unity. The results of this method are shown and discussed in Section 4.2.

3.3.3 Normalized Graph Cut

The idea is to treat the image as a graph $G(V, E)$, with node set V being pixels and edges E between every pair of pixels. Each edge is weighted by the similarity of the two pixels it connects. The similarity is measured by pixel distance, difference of intensity and difference of color, each normalized by variance. The similarity matrix is denoted as W, where W_{ij} is the similarity between pixel i and j. There is also a diagonal degree matrix $D_{ii} = \sum_j W_{ij}$. We use $x \in [0, 1]^n$ where n is the dimension of the image to denote a cut, such that pixels where $x = 0$ belongs to one side of the graph after the cut, while pixels where $x = 1$ belongs to the other side. It can be shown that the cost of a cut is

$$\text{cut}(A, V - A) = x^T(D - W)x$$

Each cut needs to be normalized so that the graph is not cut into too many small components (i.e. to avoid too many cuts). The normalization factor is

$$\text{vol}(A) = x^T Dx$$

Therefore, finding the lowest-cost cut can be formalized as the following optimization problem.

$$\min_x \text{ncut}(x) = \min_x \frac{\text{cut}(A, V - A)}{\text{vol}(A)}$$

We implemented Normalized Graph Cut and show the results in Figure 7. The results show that NGC very sensitive to color and illumination, and that the contour it generates is not very accurate. This is problematic in our game setting, because we want accurate hand contour and robustness against the change of game environment, i.e. the change of illumination. Therefore, we did not incorporate this algorithm into our Python server, and left it with Matlab implementation for performance comparison purposes.

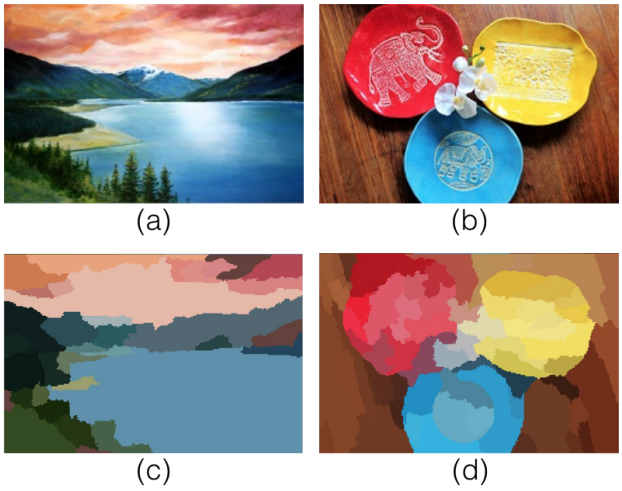

Figure 7. Graph cut performance. (a), (b) are original images, (c),(d) are their graph cut segmentation results, respectively.

3.4. Comparing with Ground Truth Shadow

Suppose we have a point light at the position of the camera. It emits rays which hits the hand and generates shadow $\mathcal{I}$. With the same hand posture, we denote a hand image captured by the camera as $\mathcal{I}'$. Since capturing hand images through the camera is equivalent to tracing rays from the camera to the outside world (we borrow this idea from ray tracing[11]), it can be inferred that $\mathcal{I}'$ is up to a scaling factor from $\mathcal{I}$. Thus it is reasonable to use binarized camera captured images $\mathcal{I}$ to represent hand shadows.

We convert the hand segment to a binary image, and compare it with the expected shadow image (also converted into a binary image). As a simple metric, we compare these images by calculating the area of the region that these two image do not overlap. This can be easily done by subtracting them and count the area of non-zero pixels (normalized by total area of the image).

4. Evaluation

4.1. Experiment Setup

We use librealsense to capture RGB-D images from RealSense camera. librealsense has a third-party Python interface, we use this interface in our project. We use OpenCV [5] for image processing, specifically, its Python interface. All hand shadow data are collected by us making a shadow

and taking pictures. Then we use Canny Edge Detector [3] to obtain shadow contours, which are displayed as hint for users. In Unity, we use `NetworkStream` to read and write from a TCP client socket.

4.2. Hand Segmentation

We experimented hand segmentation with Depth Threshold method (described in Section 3.3.1) and Sample and Filter method (described in Section 3.3.2). Here we compare their performance with real-time camera captures.

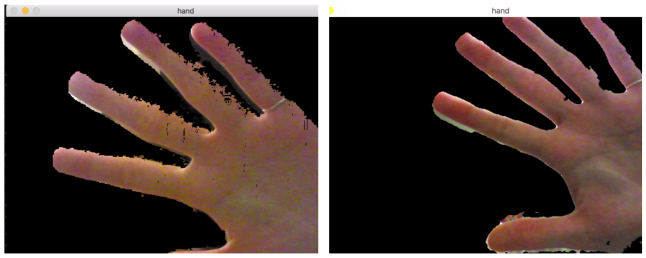

Figure 8. Hand segmentation when hand is near depth sensor. (a) Using Depth Threshold method. (b) Using Sample and Filter method.

Figure 8 shows hand segmentation when hand is very close to camera sensor. Image (a) shows the result using Depth Threshold method. If we look at edges of the fingers, we can see there are lots of fragmented artifacts. Image (b) shows the result using Sample and Filter method. The finger edges are much smoother, because pixels on the edge are refined according to their color.

Figure 9 shows hand segmentation when hand is far from camera sensor. We can see from image (a) that Depth Threshold method creates giggling edges. Image (b) shows that Sample and Filter method generates much smoother and more accurate edges of hand.

Figure 9. Hand segmentation when hand is far from depth sensor. (a) Using Depth Threshold method. (b) Using Sample and Filter method.

We further analyze the reason that Depth Threshold methods generates giggling edges with fragmented artifacts. Figure 10 shows the depth heat map of a frame. We can see that there are artifacts next to each finger, which we marked as "depth inaccurate area". The reason is that depth cameras like RealSense and Kinect use an infrared projector to scatter points onto object surface, and then the infrared camera captures these points and triangulate them with the image it sees. However, because the infrared projector and camera capture the scene from different viewpoints, they capture slightly different scenes and thus the occluded parts cannot be triangulated. Therefore, simple Depth Threshold method suffers from this problem. Sample and Filter method complement this by image processing knowledge, and that's why it can generate smooth and accurate edges of hand.

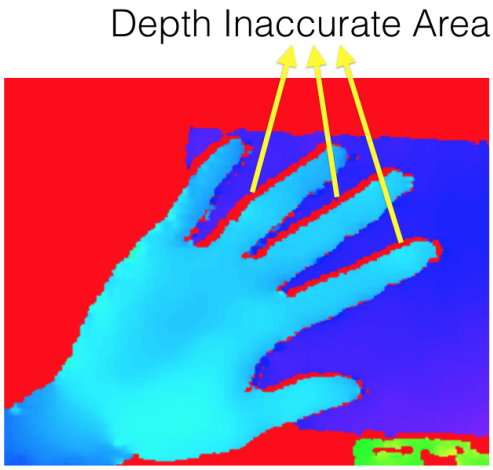

Figure 10. Depth heat map that shows the depth of some areas of the hand cannot be accurately determined.

4.3. Game Experience

Figure 11. Game start scene.

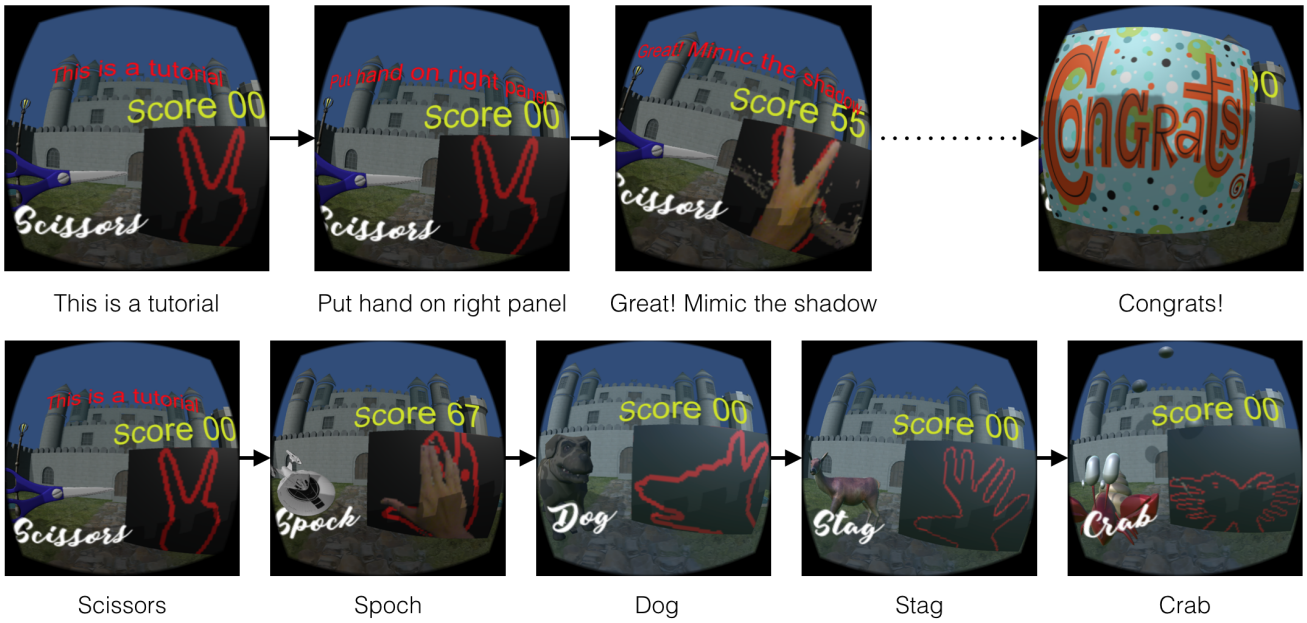

Figure 12. From right eye viewpoint: tutorial scene (top) and game levels changing from easy to hard (bottom)

4.3.1 Communication Efficiency

Using TCP socket to communicate between Python server (hand image capture and segmentation) and Unity client turns out to be pretty fast, and this contributes to real-time smooth game experience. The frame rate is mostly above 10fps.

4.3.2 Scene Design

The game start scene is shown in Figure 11. Subsequent game levels have slightly different scenes (Figure 12), but layout is similar. (1) 3D models: We use a tall castle and walls to surround the camera, so that the horizon won't be revealed to the user. Apart from texture map, we also use bump maps on castle walls and the stones on the ground to make them look more realistic.

(2) Lighting: We added point lights and spotlight to adjust scene illumination. Shadows are enabled to make the scene look more realistic. We render 2 bounces of light to capture global illumination.

4.3.3 Interaction with User

To make sure that users can get started with our hand shadow game easily and smoothly, we provide feedbacks consistently so users could quickly understand what rules they have for invoking objects and actions.

Tutorial Scene: As a starter-level round, we choose a scene with scissor hand to inform users of the game rule gradually, shown in the top part of Figure 12.

- Different from following rounds of the game, detailed instructions are displayed besides the reference model and target contour. For example, *This is a tutorial* and *Put hand on right panel*. With these instructions, we guide users to move their hands in front of the camera.

- On the backend, we keep streaming RGB-D data from camera and running hand segmentation algorithms on it. Once user's hand is detected and successfully segmented, the score shown above the panel jumps to a non-zero number. Here the score represents the completeness of the target hand shadow on a scale of 0 to 100.

- If the score is higher than some threshold, we assume user starts to know what the rule is. Then a new instruction pops up as *Great! Mimic the shadow* to encourage user modify hand gestures accordingly. As the hand shadow gets closer to the target, the score increases as well, together with other encouraging instruction *So close!* to give user some feedback upon how well they have been playing at this game.

- Once the score is high enough to reach another threshold, we display a huge *Congrats* to tell users that they have passed this level and transition to the scene of the next level.

Level design: Another important feature of a great game is to make users feel entertained the whole time with delicately designed game levels. In our hand shadow game,

we have designed 5 rounds with difficulty increasing gradually, shown in the bottom of Figure 12. For starters, the first scene (*Scissors*) serves as a tutorial, which user could pass easily using one hand to make gestures. As followed, the second round is a *Spock* hand gesture[2] which is slightly more difficult but still only requires one hand to pass. Then the rest rounds (*Dog*, *Stag* and *Crab*) would have a transition from using single hand to both, and become more and more hard to accomplish.

5. Discussion

This project has granted us the chance to combine a computer vision task and a VR application. It might have been a cool demo but still needs polishing. As for future work, we'd like to explore the following aspects.

- Smaller latency. Right now, the TCP server is implemented with blocking sockets. In this case, the server reads in data and processes the images. Then it will wait for the client to call, and in the meanwhile, block the input data stream. When the client has a request, the server will send back a packet which was finished a few milliseconds ago. To reduce this delay, one way is to implement a non-blocking socket, which will keep reading and processing data while wait for clients to call. Then when client calls, the packet received will always be the latest.

- Better approach of determining whether user has completed a hand shadow. In this project, we use the naive method of subtracting user hand shape from ground-truth hand shadow and counting non-zero pixels. The drawback is that, users have very different hand shapes (e.g. some may have thick fingers while some don't), not everyone has the same hand shape as the person that made those ground-truth shadows. Taking this into account, we can use SIFT [8] features so that local invariance is taken care of.

Acknowledgement

We would like to thank Professor Gordon Wetzstein, and all the TAs for a great class experience. Special thanks to Keenan who has provided us with lots of help and feedbacks on this project.

References

[1] A. Almoznino and Y. Pinas. *The Art of Hand Shadows*. Courier Corporation, 2002.

[2] G. Burdea and G. Coiffet. Virtual reality technology. 1993.

[3] J. Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[4] G. Devarajan, V. Aatre, and C. Sridhar. Analysis of median filter. In *ACE'90. Proceedings of [XVI Annual Convention and Exhibition of the IEEE In India]*, pages 274–276. IEEE, 1990.

[5] Itseez. Open source computer vision library. `https://github.com/itseez/opencv`, 2015.

[6] F. Jacobs and H. Bursill. *Fun with Hand Shadows*. Courier Corporation, 1996.

[7] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[8] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[9] H. Rheingold. *Virtual Reality: Exploring the Brave New Technologies*. Simon & Schuster Adult Publishing Group, 1991.

[10] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[11] T. Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, page 4. ACM, 2005.

[2] From Star Trek, has a meaning of *Live long and prosper*.