Real-time augmentation of a children's card game

Jordan Rabet
Stanford University
jrabet@stanford.edu

***Abstract*—Still today, trading card games make up a sizable part of the entertainment industry. However, their tired paper form is being rapidly taken over by digital equivalents. In this paper, we propose a way to reinvigorate interest in physical trading card games by making them more interactive through real-time 3D augmentation. While our ideal application would run on an AR headset such as the Hololens, we build a proof-of-concept application on a commercially available Android tablet which is able to augment existing Pokemon trading card. We show that using a modern CPU in conjunction with a GPU, this task is entirely tractable on a mobile device with few restrictions.**

I. Introduction

Many trading card games are based on the idea of cards representing monsters or other entities which are used to fight one another. Playing one prominent example of such a game, Yu-Gi-Oh, is often portrayed in media as being accompanied by holograms representing those monsters and their actions, which make the game more fun and engaging to play. The goal of this project is to build a mobile application which is able to augment an existing trading card game in a similar fashion. While the ideal target for such an application would be an augmented reality headset, the application was developed for a commercially available Android tablet (specifically, the Nvidia SHIELD tablet we have been provided) because of the current lack of such headsets as consumer products.

A. Previous work

There are several commercial products which offer similar functionality to this project. The most notable of these is the Playstation 3 game "The Eye of Judgment", developed by Sony Computer Entertainment and released in 2007. This game implemented the same concept of augmenting physical gamecards to make playing for engaging; however, it differed in several key aspects :

1) Game cards were designed specifically for the purpose of being augmented in this game. In fact, they were all marked with special tags meant to make detection and classification easier : CyberCodes [**?**].
2) The scene was controlled : players had to place a mat supplied with the game to place their cards on. Additionally, the camera had to be mounted above said mat using the supplied camera stand. The camera had to be immobile during the duration of the game.
3) The game ran on the Playstation 3, a powerful non-mobile device, more powerful than modern tablets.
4) The camera used (the Playstation Eye) was designed for computer vision purposes, and was therefore capable of streaming uncompressed video at high framerates (up to 120 frames per second).

A more recent, similar example is "Drakerz", a game released in 2014 for the PC. While it does not require a playmat, it still requires that the camera be immobile and mounted above the playing field. It also runs only on modern computers.

B. Contributions

The main contribution of this project is building an end-to-end system which is able to detect, classify, track and augment multiple commercially-available trading cards at once in real time on a mobile device. It mostly differs from previous comparable applications by the fact that the camera need not be fixed, that the required computations were optimized for a mobile device, and that the gamecards being augmented were not designed for the purpose of augmentation, making the task more challenging.

II. Technical execution

A. System overview

Our system's goal is to take sequential camera frames as input, detect Pokemon game cards in them, track their position through time and finally augment them with the appropriate 3D representation. In order to achieve this goal, we divided our system into five distinct components :

- Card detector : takes a single frame as input and attempts to find all cards it contains, outputting a list of 2D quads corresponding to the detected cards.
- Card classifier : takes a single image extracted and rectified from a frame (likely by the card detector) and matches it against known game cards to find which one

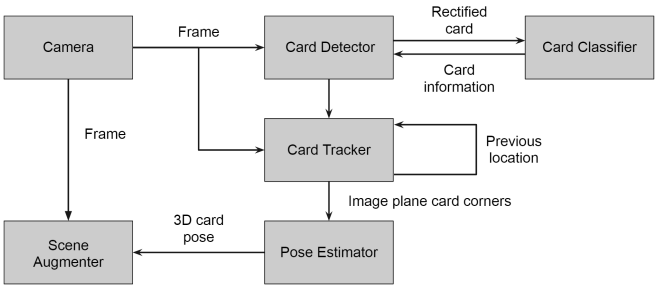

Fig. 1. Diagram representing our augmentation system's organization .

it is. If found, returns the name of the corresponding Pokemon as well as the quad's proper orientation.

- Card tracker : takes frame $n-1$, the location of a card at frame $n-1$ (in the image plane) as well as frame n as input and outputs the location of that same card (in the image plane) at frame n.
- Pose estimator : takes 4 points in the image plane representing a square in world space and outputs its world space position and orientation.
- Scene augmenter : takes a single frame accompanied by the corresponding 3D locations and orientations of cards in world space and outputs the scene augmented with 3D models on top of corresponding cards.

A system's flow can be seen in Figure 1. The main loop consists of the latest camera frame being sent to the card tracker, with the tracker's output being fed the pose estimator which in turn goes to the scene augmenter. When initializing the system, the camera's latest frame is also sent to the card detector, which uses the card classifier to identify cards before injecting its findings into the card tracker. The card detection system is run at a much lower frequency than the tracker due to higher computational requirements. In practice, for the purposes of our tests, the detector was only run when the user tapped the tablet's screen, which ended up being sufficient thanks to the tracker's robustness. For the final product however, we envision the detector continuously running in its own thread/core in parallel to the main loop.
In order to simplify the problem, we currently make the assumption that all cards are located in the same plane, and that the camera's intrinsics are known (ie, we assume that we know that camera matrix K).

B. Card detector

The game card detector's goal is to allow the system to automatically find relevant targets in the current scene without having the user manually tag them. Unlike previous works, our application is built to work with the approximately 10,000 Pokemon cards already in circulation [5], meaning we cannot equip them with specially designed markers making detection easier. That being said, as can be seen in Figure 2, Pokemon cards do have a very distinctive feature : their thick, yellow border, which is what our detector targets. Interestingly, most trading card games have a similarly thick border on all their

Fig. 2. Sample Pokemon card. Note the thick yellow outline .

cards (though it isn't usually yellow), making the idea of detecting cards by looking for their border applicable to more than just the Pokemon trading card game.

The borders we are looking for have two main characteristics : as tick borders to a quadrilateral, we can expect them to show up as two quadrilaterals in edge maps, and being bright yellow, we can expect to be able to distinguish them from the rest of the scene based on color. As such, the first step in our detection process is finding all pixels which might belong to a card border based on color. In order to do this, we first apply a white balance filter in order to correct colors and make sure our detector will work in different lighting environments. In the current version, this is done using global histogram equalization on individual RGB channels. While this method was only supposed to be a baseline, it ended up yielding very good results under multiple different lighting conditions, so it was kept. Then, we convert the color corrected image to the HSV color space, where we apply a simple threshold function to keep only pixels which are approximately the right color. The values for this threshold function were manually tuned and chosen in order to be more permissive than restrictive; in most cases, this filter will catch a large amount of pixels outside of card borders, but this is remedied later in the detection pipeline.
Then, we compute an edge map of our image using the Canny edge detection filter applied to a Gaussian-blurred version of the frame. We then blur the resulting edge map using a flat $5x5$ kernel, blur the color threshold image with a $21x21$ Gaussian kernel, and compute the element-wise product of those two images. The result is a bitmap where all lit pixels have to be near both a well defined edge and the color yellow. An example of those steps can be seen in Figure 3. As can be seen, while the color map and edge map show a lot of data which is not related to the targets, once put together they yield a decently accurate search area.
With this bitmap generated, the goal becomes to isolate

parts of it which might be cards and eliminate those which aren't. This is done by finding all connected components in the image. This can be done rather efficiently in a single top-to-bottom left-to-right pass on the image, by connecting each lit pixel to its left and top neighbors if either is also lit, and merging components together if they both are. That done, we filter connected components to eliminate unlikely candidates; doing this, we assume that there should be a 1:1 mapping between components and cards, and therefore we discard components which are too small (likely noise), those which are contained in another component (likely yellow markings on the card's illustration), as well as components which do not have the right proportions (likely due to noise, glare, or an extraneous and unfortunately colored object). The results of this filter can be seen in Figure 3.

With only components deemed likely to be an actual card's border left, we consider them each individually. Doing this means taking each component's bitmap and taking its bitwise AND product with the original edge map. Doing this gives us an edge map which should correspond to the border, which we then use to fit a quadrilateral to represent the card. An example of the output of the AND product can be seen in Figure 4. Quad fitting here is done using the Hough transform, and is a fairly simple process. We run the Hough line detection algorithm (with an angular resolution of 1 degree) on the card candidate's edgemap, which returns a list of local maxima lines sorted by number of votes. We go through this list in descending order of votes and only keep lines which are different enough from previously picked lines, in order to only keep the best lines of each cluster, stopping at a maximum of 8 lines (though we typically end up with fewer than 6). That done, we go through combinations of the lines, computing all possible intersections, and only keeping combinations which result in exactly 4 intersections within the region of interest. Assuming there is more than one such combination, we pick the one which results in the largest area quad, which becomes our initial estimate for the card's border.

We then refine this estimate by using a targeted version of the Hough transform with an angular resolution of $1/128$ degree, whose angular space is limited to angles less than 0.5 degrees away from our initially estimated. This allows us to get a better angular estimate of the lines simply by running the Hough line detection algorithm again, at no extra cost compared to the one run earlier in the detection algorithm.
The results of the detector are then used to compute a homography between the found borders and a rectangle with the dimensions of a card, which is in turn used to perspective-rectify cards. Those rectified images are then sent to the classifier.

C. Card classifier

Image classification was not the focus of this project, so in the interest of time little of our resources were put into making it. As such, the chosen classifier is not particularly

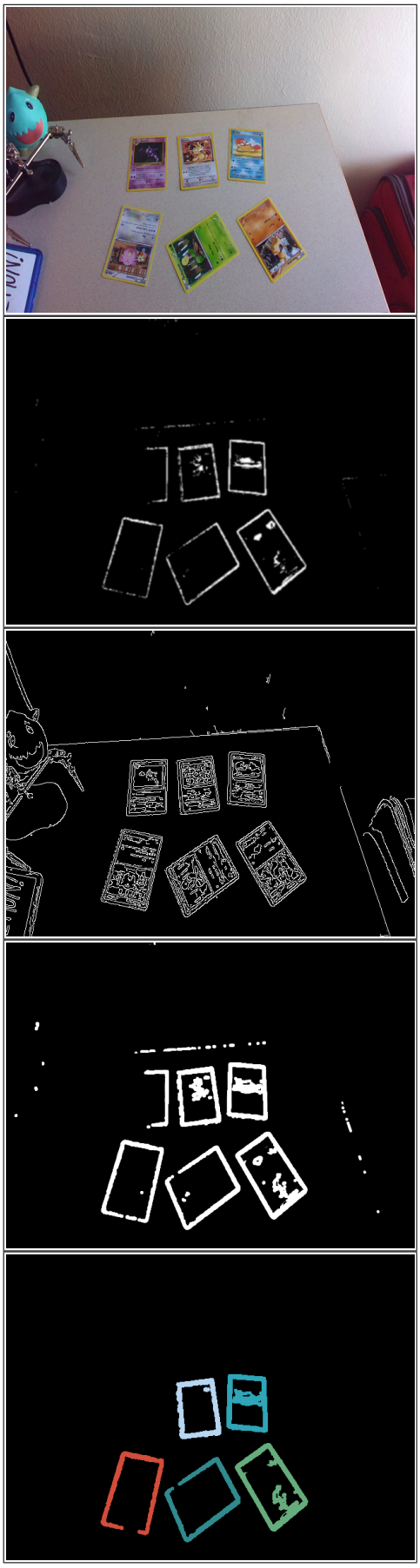

Fig. 3. Sample run of detection pipeline. From top to bottom : original image, color-filtered image, edge-map, "border-map", filtered connected components. (each color represents a different component)

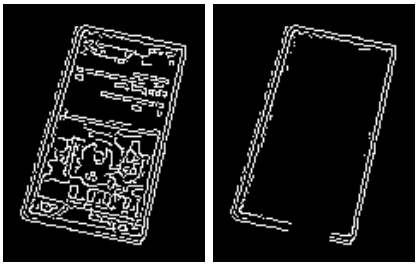

Fig. 4. Sample product of a connected component by the edge map. Left : original edge map. Right : product of edge map by connected component border. .

well adapted to the problem; it was mostly made as a proof-of-concept, as well as to take advantage of the geometry consistency test's result in the detector and to make the final augmenter's output look nicer without having to manually pick the types of the detected cards.

The method which was implemented is a simple version of a bag-of-words classifier. First, we extract SURF features from training images in order to cluster them into a visual vocabulary. Then, we train a 1-versus-all SVM for each of the target images based on its histogram response to the full visual vocabulary. With that done, classification can be performed by taking the rectified card candidate query image extracted by the detector, extracting SURF features from it, matching those features to the vocabulary, computing the histogram response and running it through all the SVMs. We then get a score indicating how the confidence in the query image matching one of the given training images. We take all n cards which have a confidence score above a certain threshold, and then run a geometry consistency test on them, by matching SURF features from the query image with those from training images directly and computing a homography using RANSAC. Finally, the training image whose homography is computed with the highest number of inliers is chosen as the one which matches the query image. Doing this geometry consistency test is especially useful because it gives us the query image's absolute orientation, which in practice is necessary information for our pose estimator.

As previously mentioned, classification is not the focus of this project; as a result, this approach is fairly slow and not immediately scalable to large numbers of card types. In practice, given the high number of individual Pokemon (over 700, each of which require their own 3D model, textures, animations, sounds...) and of individual Pokemon cards (on the order of $10,000$), one could imagine that hosting the classifier as a service on a distant server which the client would send query images to, and would in return give information on the card accompanied by the assets necessary for augmentation.

D. Card tracker

The goal of the card tracker is to determine the movement of a card between two frames so that neither the detector nor the classifier has to be run again each frame, which is of course desirable in order to achieve good performance and smooth augmentation. The tracker works separately for each card in order to make independent card movement possible, though tasks which can are batched together for peformance reasons. The tracker is first initialized for a card when it receives an initial position from the card detector. When that happens, the tracker detects Shi-Tomasi "Good Features to Track" [4] as implemented in OpenCV's goodFeaturesToTrack method and saves those which are located within the initial quadrilateral estimate.

When a new frame is received, the first thing done by the tracker is computing KLT optical flow for the card's features. This is done to get an initial estimate of the motion between the two frames : since our target is planar, a homography is computed between the frames using those feature matches (with RANSAC for outlier resiliance), and this homography is then applied to the card's four corners. Unfortunately, doing just this is not enough : this kind of optical-flow based tracking is extremely prone to drift, especially in low resolution and noisy environments. While it might be able to maintain a good estimate of the card's location, the card's shape slowly changes over time, which is extremely problematic given that the card's shape being accurate is essential to good pose estimation.

Due to this, the optical flow tracking is only used as an initial estimate each frame. Once this initial estimate is done, it is used as a search region for the actual new position of the card. A quad of slightly larger size is used as region of interest in an edge map (again computed using the Canny edge detector), which is then fed to the same quad fitting algorithm as the one previously described in the card detector, which a few differences. First, the edge map has to be filtered. Instead of relying on color (which can be unreliable, partly due to glare) to isolate the border, we filter out the card's contents by only keeping the left-most and right-most pixels on each row, as the top-most and bottom-most pixels on each column. This filtering can be done efficiently and in practice gives completely usable results, as can be seen in Figure **??**. Additionally, instead of having the quad fitting algorithm look for the quadrilateral with the largest area, it instead looks for the one that minimizes the cumulative distance between the old and new quads. In case the tracker is unable to fit a quad, it sticks with the initial estimate based on optical flow. An overview of the card tracker can be found in Figure 6.

Another problem which the tracker has to deal with is sudden camera movements, both big and small, which throw off the KLT tracker. Indeed, a problem with the approach presented above is that if the optical flow tracker's initial estimate is too far off, then the quad fitting algorithm will fail to recover from it. While smooth movements typically result in good initial estimates, quick camera jerks (typically unintentional) can ruin the tracker's accuracy. In order to deal with this, we first attempt to detect those instance by computing the variance of the distance betwen corresponding card corners as estimated by the optical flow tracker. The

Fig. 5. Sample product of the edge map border filter. Left : original edge map. Right : border-filtered edge map. .

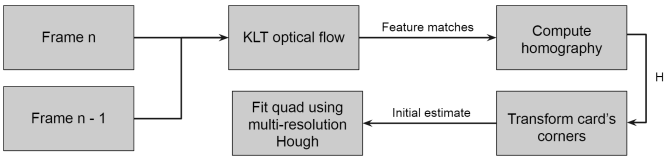

Fig. 6. Diagram representing the card tracking system .

idea is that if the movement is smooth and the timestep is small enough, then the shape of the card will have changed relatively little between two sequential frames, meaning that the variance would be low. On the other hand, if the movement is not smooth and the optical flow tracker results in an estimate which is largely off, then the variance will be high. In practice, we were able to find a threshold on that variance above which we determine that a problem occured with the optical flow tracker. When this happens, we replace the homography computation with a (still RANSAC-based) the estimation of an affine transformation between the matching features. This way, we are able to keep a sane shape for the card without sacrificing the knowledge the optical flow tracker gives us about the card's likely translation and rotation between the two frames. Implementing this largely helped improve our tracker's robustness.

In practice, we observed that this tracker performs quite well, is robust to a number of potentially problematic situations (partial occlusion, glare, camera jerk, slight motion blur) and at a relatively low cost in performance overhead thanks to optimizations which are described below.

E. Pose estimator

For every frame, once we are confident that we have found a reasonably good estimate of a card's corners in the image plane, we need to determine the card's pose in world space, which entails finding its position (represented by a translation vector) as well as its orientation (represented by a 3x3 rotation matrix). There are many so-called PnP (Perspective-n-Point) methods which can be used to estimate camera's pose relative to an object given the object's three dimensional shape and its corresponding image plane points. These methods can in fact be directly applied to this problem (the baseline we used for this part of the system was OpenCV's solvePnP

Fig. 7. An example of pose ambiguity; both images represent correct mathematical solutions to the pose estimation problem. .

method), however they are not ideal for multiple reasons. The first reason is that being geared towards a more general problem (that of an arbitrary 3D object), they are less than optimal for our problem which only involves planar objects, and performance is definitely a concern for real-time mobile applications. The second reason is that the best PnP methods are iterative and only converge towards a single solution, which is problematic as, in the case of planar objects, it was shown that there are in fact two local minima which each yield a separate solution pose to the problem [2]. In order to deal with this ambiguity, it is necessary to know what both of these solutions are, making generic PnP solutions inadequate. An example of the pose ambiguity problem can be seen in Figure 7.

Thankfully, there exists a method made to estimate the camera's pose relative to a square marker which deals with both of those issue [3]. Essentially, the method introduces a parameterization of the pose which depends on a single variable, the primary angle β, which both allows us to find the two ambiguous solutions and make extensive use of look up tables (LUTs) to accelerate the process. Making use of both the original paper and a matlab implementation of the algorithm provided by the authors, we were able to recreate the algorithm

in C++ and integrate it into our system. Of course, because the algorithm is made to work with square targets while ours are rectangular, we apply the right scaling transform before calling the pose estimator. This makes it especially criticaly that the card's orientation be known, as applying the scaling over the wrong direction would yield a wrong pose.
In order to deal with pose ambiguities, we rely having more than a single card being tracked by our system. Each card pose yields 4 points in 3D, on which we can use RANSAC to fit a plane. Given the nature of the ambiguities, this should allow us to find the plane the cards are actually located on, since we assume that all cards are coplanar. Then, using this plane's normal vector, we can, for each card, find the pose which is closest in orientation to that plane, which should be the pose we are after.
In addition to this, we make use of the fact that all cards are located in the same plane to deal with potential outlier cards. For example, there are situations in which 3 cards will have been tracked properly, but the fourth's corners are improperly registered for a few frames. By default, the fourth card's pose would be computated as being completely different from the others. In order to deal with this, we cluster plane normals extracted from each card's pose, determine which normals might be outliers, and then compute the plane normal as the average of inlier normals. We then reinject that average normal into each card's pose by setting it and then applying Gramm-Schmidt orthonormalization to the rotation matrix.

F. Scene augmenter

The scene augmenter module was made using OpenGL ES 2.0 for rendering. It works by first transferring the current from to the GPU as a texture, and rendering it as a flat image, clearing the depth buffer at the same time. Then, a projection matrix is computed for the actual augmentation. This camera matrix is based on the camera matrix K, but is modified in order to preserve depth information, which is needed for OpenGL rendering. If we have :

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Then the projection matrix is as follows :

$$\begin{bmatrix} \frac{2f_x}{w} & 0 & \frac{2c_x}{w} - 1 & 0 \\ 0 & \frac{2f_y}{h} & 1 - \frac{2c_y}{h} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{nf}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Where w and h are the frame's width and height respectively, n is the near plane's distance and f is the far plane's distance. The models used for augmentation are rendered using an OpenGL ES 2.0 OBJ parser and renderer written for the occasion.

G. Platform-specific optimizations

In order to get this complex system to run at acceptable speeds on a mobile device, many optimization specific to the platform had to be made. The first one is of course the use of look-up tables for pose estimation, which was mentioned above. The second one has to do with the quad fitting algorithm also described above. As it is used in the tracker, it is important that it be very efficient. Its initial implementation, based on the HoughLines function defined in OpenCV, was found to perform very poorly. After investigation, it turned out that while the function was based on look-up tables, those were re-generated each time the function was called. Additionally, parts of the function made use of doubles which are far slower than single precision floats on our target device. In order to improve this design, the hough line function was reimplemented, based on the OpenCV code, but changing the way look-up tables are generated (only once per resolution value), all the code which used double precision floats, and actually doing away with floats in some parts of the function by switching the look-up tables to use fixed point math. No loss in precision was measured, while the average call to the function was made 6 times faster.
Another costly operation in the tracking pipeline was found to be the sparse KLT optical flow calls. First of all, the individual calls made for each card's tracking were mutualized into a single call using for all tracked features at once, which helped make the use of optical flow more tractable. Additionally, the optical flow computation was completely moved from the CPU to the GPU using CUDA, which made the process twice as fast. Similarly, the computation of the Canny edge map (after the Gaussian kernel) was changed to be done using CUDA, multiplying the speed threefold. A related design choice had all "fundamental" image transformations such as the edge map computation, color correction and conversions all be stored in a single "frame sequence" object in order to make sure that the same computation would never be done more than it needed to be each frame.
Finally, in order to make use of the fact that our target device has not only powerful GPGPU capabilities but also a quad core CPU, the card tracking process was changed to make good use of multithreading by spawning 4 worker threads each able of processing an individual card's tracking in parallel to others. In most cases, this multiplied the tracker's overall speed by 4. A visualization of the final tracking pipeline can be seen in Figure 8.

III. Results

A. Performance

As is indicated by the previous section, performance was a major concern while making this application. In the end, we were able to achieve an overall average of 14 frames per second in most situations with 5 or fewer cards, which is more than enough to be considered real time application. In Table I, the average time taken for each task of main loop can be found. What we can see is that the optimizations made to the

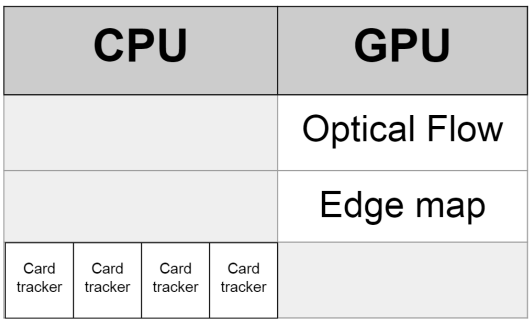

Fig. 8. Diagram representing the current tracking pipeline's distribution across computing devices .

TABLE I
PERFORMANCE OF THE MAIN LOOP. PERFORMANCE RECORDED WHEN TRACKING 4 CARDS.

Task	Device	Average time taken
Sparse optical flow	GPU	$19.6ms$
Canny edge map	GPU	$19.3ms$
Single Hough transform	CPU	$10.2ms$
Quad fitting	CPU	$23.6ms$
All cards tracking update	GPU and CPU	$45.5ms$
Single card pose estimate	CPU	$0.58ms$
All cards pose estimate	CPU	$3.2ms$
Processing entire frame	GPU and CPU	$73.5ms$

system's various modules were very effective. For instance, each pose estimate is done in only half a millisecond; similarly, the multithreading used to parallelize card tracking results in very little overhead, allowing the application to process 4 cards almost as fast as a single core is able to process a single one.

B. Quality of augmentation

It is difficult to accurately gauge the accuracy of the augmentation as we do not have any kind of ground truth, especially in the case of orientation. A (flawed) metric which can be used to somewhat assess the quality of the tracker is the number of cards for which the tracked is able to fit a new quad each frame. A graph representing this metric can be found in Figure 9; interpreting it, we can see that most cards don't seem to go long without getting a new quad fit to prevent drift, except for one which apparently doesn't get a new fit for the entire middle section of the sequence.
Qualitatively, we can say that the tracker manages to be robust to a number of problematic situations. For example, it is able to be robust to glare in certain situations, as can be seen in Figure 10. Additionally, it is able to withstand partial occlusions for short periods of time, as can be seen in Figure 11. It is also robust enough that a variety of angles work well, even very low angles which let the user look at augmented models from the side, as can be seen in Figure 12. More impressively, it is robust enough that having a user move cards while the camera itself is being moved around with their finger is not a problem, be it rotations or translations, as can be seen in Figure 13.
That being said, while the tracking is overall robust and accurate, the final result is not perfect. Unfortunately, the

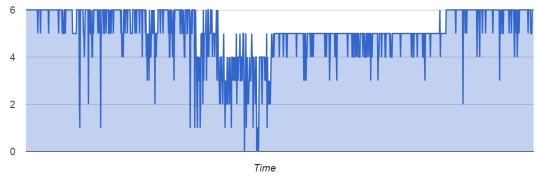

Fig. 9. Graph representing the number of frames which the tracker successfully fit a quad to over time, in a scene with a total of 6 cards being tracked .

Fig. 10. An example of the tracker showing robustness to glare. (bottom left card) .

output still suffers from intermittent jitters, typically due to poor quad fits happening in certain frames. This indicates that tracking could still be improved, as well as the normal vector outlier rejection mentioned previously.
A video showing the augmenter running under a variety of conditions (including the ones described above) can be found attached to this paper.

IV. FUTURE WORK

While this project's results are very encouraging, they do not make a full product. Due to lack of time, all potential optimizations and features could not be included in this project. For example, the tracking pipeline presented in Figure 8, while already producing good enough performance, leaves to be desired in that it does not make optimal use of all computing devices at all times. Instead, one could imaging a tracking pipeline closer to that described in Figure 14, which has the GPU processing data for frame n while the CPU is

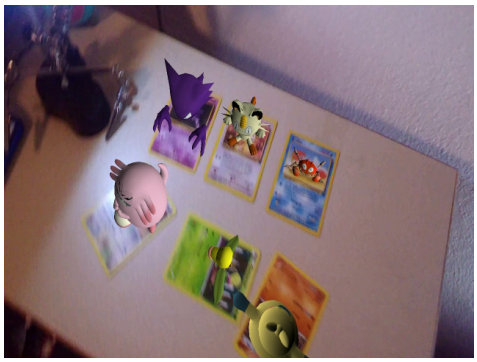

Fig. 11. An example of the tracker showing robustness to partial occlusion. (bottom right card) .

Fig. 12. An example of the tracker showing robustness to a low viewing angle. .

Fig. 13. An example of the tracker keeping up with a variety of user card movements.

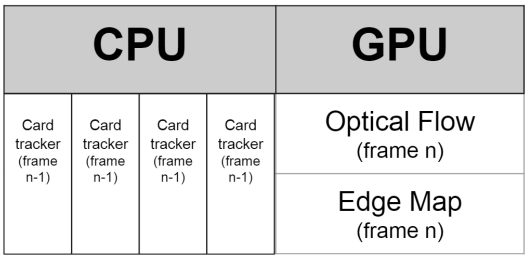

Fig. 14. Diagram representing the a possibly better tracking pipeline's distribution across computing devices .

still processing data from frame $n-1$, as such a pipeline could as much as double the currently observed framerate.

Optimization is not the only part of the project which could be expanded upon. Indeed, although the project runs on a mobile device and requires an accurate estimate of the device's location relative to its environment for proper augmentation, it currently makes no use of any of the on-board inertial sensors. One could imagine that integrating signals from those sensors could help make tracking even more robust, as well as give a good heuristic to help decide between ambiguous poses in the pose estimator.

Finally, a big part of trading card games is actually playing with them. While augmenting cards with 3D models already helps make the game more engaging, it is not enough for an application which aims to fully augment card game duels, by for example animating Pokemon attacking other Pokemon. While the underlying rendering part is trivial, detecting that a player is commanding one of its creatures to attack is not. Doing so would require maintaining a coherent map of the battle field, including attached semantics such as card ownership and card status (for example, detecting that a card is sideways which in a lot of card games is indicative of an action). One could imagine implementing some kind of card-based gesture recognition system to allow the application to fully follow the duel.

V. Conclusion

With this paper, we demonstrated that it is entirely possible to augment a trading card game in real time even in spite of the absence of specially designed markers on cards, of a fixed camera or of heavy computation capabilities. We paid close attention to code and algorithm optimization in order to get the system running smoothly on a mobile device, the Nvidia SHIELD tablet. We leveraged of both the CPU and the GPU for computation, and found that our card tracker is robust enough to handle users moving cards while the camera is also moving independently. We made suggestions for ways to expand on the project in both purely technical ways and more feature-oriented ones.

References

[1] Jun Rekimoto and Yuji Ayatsuka, *CyberCode: Designing Augmented Reality Environments with Visual Tags* 2001.

[2] G. Schweighofer and A. Pinz, *Robust pose estimation from a planar target* 2006.
[3] Shiqi Li and Chi Xu, *Efficient Lookup Table Based Camera Pose Estimation for Augmented Reality* 2011.
[4] Jianbo Shi and Carlo Tomasi, *Good Features to Track* 1994.
[5] Pokepedia, *Pokepedia list of Pokemon cards*. http://www.pokepedia.net/