DrawingBook

16-423 Final Project

William Willsey & Tommy Lau

Abstract:
In this project we have developed an application on iOS devices in order to project augmented reality 3D cube onto a coloring book, where the colors filled in are projected onto the cube itself. Any coloring on the drawing book page will then be reflected and projected to the augmented reality 3D model. Our goal is to familiarize ourselves with computer vision algorithms and iOS development in the process, while developing a creative upgrade to ordinary coloring books. We work to tackle issues created by limitations of mobile computing such as processing power and rapid movement of the device, both of which are rather non-issues on conventional high-performance computers.

Introduction:
This project works well as a mobile app, as the principle idea behind it is expanding the creativity of traditional coloring books by adding a new dimension to the drawing. Being able to quickly view and move around a drawing book with ease dictates that an app such as ours must be on a mobile platform like an iOS device. There are many challenges associated with creating our mobile computer vision app such as efficient detecting/tracking of the page and robust error and noise handling.

Background:
Our project is based off of much more in-depth research performed by Disney:
http://www.disneyresearch.com/publication/live-texturing-of-augmented-reality-characters/
We set out to create a functioning prototype that recognizes a template with an unfolded cube, then projects an augmented reality 3D cube and any drawings on it above the page. We placed an intricate pattern on the coloring sheet with a complex color gradient so that we can more easily match features on the page with an internal template. We can then compute necessary parameters including homography transforms and camera extrinsic matrices. We investigated many feature point detectors, descriptor generators, and matching techniques. Additionally, we investigated the use of the Armadillo library for faster and more efficient matrix computations, as well as the Lucas Kanade algorithm for tracking the page accurately and quickly. In the interest of time and getting a working prototype completed we opted to do much of our computations through OpenCV leaving further optimization for later. Another optimization that would certainly come into play if our object was much more complex than a cube would be utilization of the GPU and libraries such as OpenGL ES.

Approach:
Our application works in 3 steps:

1) Recognize the coloring page
2) Recover the drawings on the template
3) Project the cube above the page

The first step is recognizing the coloring page from the camera stream. We do this by using an ORB feature point detector and description extractor, and a brute force matcher between our coloring page template and the camera stream input. Using the well-matched pairs, we compute a homography matrix to warp from the drawing page template to the actual drawing page from the camera input. The quality of the matching is then determined by the percentage of matching pairs being inliers, and the average Hamming distance of each pair of matching. Additionally, we use the device's IMU to determine if we should expect a large jump or motion blur in the input, and factor this movement into the overall quality of input. If the quality is high, we will then move on to the next steps.

Second step, we recover the drawings on the coloring page by warping the camera input using the inverse homography matrix. The result should be the coloring sheet template as seen by the camera, with the drawings on it. The application then recovers the drawings on each square simply by cropping out the images.

The last step is to create the box. The application uses the homography matrix to extract the extrinsic camera parameters. Using these parameters, the application calculates the uv-coordinates in the image plane using the real world xyz-coordinates of the 8 corners of the cube. Using the 8 points, the image plane consists of the 6 sides of the cubes. For each side of the cube, we compute the norm of the plane using the cross product of two of the sides, and only project those planes which the norm is negative, i.e. if they are facing the camera. This technique is known as occlusion culling, and it ensures that the the back side of the cube doesn't appear in front of the front side on accident. In fact, the back sides of the cube aren't drawn at all until they are in view of the camera. To project each plane, we compute the homography warp from the extracted drawings, to each respective plane of the image frame. We then warp the extracted drawing by the computed homography, and then add it to the camera input.

Results:
A YouTube clip of the application in action: https://youtu.be/y49zOtfOqGU

After much tuning and improvements, our application successfully projects a cube in 3D and applies the necessary textures on each side effectively from most camera perspectives, though the efficiency and frame rate have left something to be desired. During the cube projection, our ORB feature matching and the resulting homography are stable and accurate, thus allowing the application to find the coloring sheet and the drawings inside of each box. On the other hand, the extrinsic camera parameters we extract using the homography matrix and solvePnp are quite error prone and can lead to our cube jumping around quickly at times. Because we are performing feature matching and computing homography matrix at every frame, the run time of the computation is too large, and therefore leaves our frame rate at about 10 frame per seconds.

We can optimize the algorithms used by the application in many ways. In the hardware perspective, we could parallelize part of the computations using the GPU and reduce run-time as well as CPU usage that way.
Software-wise, many of our computations are simply done using the OpenCV framework and are thus far from optimized. Basic Linear Algebra Subprograms (BLAS) or Linear Algebra Packages (LAPACK) can reduce the run time on those computations. Libraries such as Armadillo (http://arma.sourceforge.net/) and Eigen (http://eigen.tuxfamily.org/) allow matrices compuations to be run much faster, and thus can help with the run time of many of the computations done by the application, including bottlenecks such as RANSAC when the application is computing homography matching.
On an algorithmic level, to ensure that the application runs efficiently, computationally intensive steps need to be reduced in order to optimize our application. Currently, our application performs feature matching and RANSAC to find the homography at each frame, and that is computationally expensive due to the iterative nature of RANSAC. We explored many options to reduce the usage of RANSAC, mainly by replacing RANSAC with tracking techniques such as implementations of Lucas-Kanade algorithms or others. The use of tracking such as correlation filters or Lucas-Kanade algorithms will allow the application to only run feature matching and RANSAC at least once, and then track the video stream input at every frame instead of naively and exhaustively perform matching at every single frame. The implementation of robust tracking features will sacrifice some of the correctness, as feature matching at every frame will give us the best matching overall. Tracking will also significantly reduce run time on each frame by getting rid of the most computationally expensive part of our algorithms, and can thus reduce the choppiness and improve the framerate to a more acceptable level.

List of Work:
Equal work was performed by both project members

GitHub Page:
https://github.com/tommylau523/DrawingBook

References:
http://docs.opencv.org/2.4/
http://www.disneyresearch.com/publication/live-texturing-of-augmented-reality-characters/