Real-time Image Blending for Augmented Reality on Mobile Phones

Yaron Inger
statusreport@cs.huji.ac.il

Supervised by:

Mr. Zeev Farbman
farbman@gmail.com

Prof. Dani Lischinski
danix@cs.huji.ac.il

The Rachel and Selim Benin School of Engineering and Computer Science
The Hebrew University of Jerusalem
1. Abstract

Augmented reality is currently a hot trend in the mobile industry, allowing the addition of external data on top of camera input. Browsing app stores for mobile devices we find augmented reality applications in many interest areas such as gaming, navigation, tourism and even education. Augmented reality is currently being researched both in the academy (i.e. in computer vision, image processing and user experience labs) and at the industry by companies like Sony, which plans to officially integrate this technology in further versions of its gaming consoles.

Using computer vision and digital image processing algorithms, we’ve created an augmented reality iPhone application which examines the environment and searches for a known visual marker, then replaces its image with a new image selected by the user. In contrast to other augmented reality applications in the market which display the data to the user without focusing on blending it into the scene, our application was designed to provide the user with the ability to experience a scene with additional images as if it really existed in the real world.

The application runs in real-time, which requires heavy optimizations of the algorithms used, with respect to the relatively weak hardware it runs on.
2. Introduction

Imagine you moved to a new apartment and you’re looking to buy new pictures to cover your empty walls. Today, there are two leading options available for buying artwork. The first is going out to the nearest gallery, finding a picture that you like and imagine it in your room. The second is to sit in your home and browse the online catalog of a gallery using the Internet. But why can’t we combine the two? Recent advances in technology should allow us to create software that runs on a PC or a mobile phone and makes it possible to select and watch a desired picture in a room, as it’s really hang on the wall. This technology involves computer vision, digital image processing and real-time computation techniques – all of them form the experience that is called “Augmented Reality”.

The term “Augmented Reality” stands for sensing a real-world environment whose elements are augmented with artificial data created by a computer. The interest in Augmented Reality started in the beginning of the 90’s and was part of the hype around Virtual Reality. However, only a few years ago with the uprising of the mobile computing and smartphones industry, it was integrated into consumer devices.

Most of the Augmented Reality applications available today use a camera (connected to a PC or a smartphone) as the input device. On top of the camera input layer the applications present 2D or 3D layers that include additional information about the environment the user is in, depending on the application’s context. This data can be used in various implementations such as navigation (displaying driving or walking directions on top of street view), tourism (showing names and historic data of places on top of them), games (mixing real and virtual world) and more.

The Augmented Reality applications can be divided to two major groups:

1. Orientation aided applications: Augmented Reality software which uses built-in or external sensors such as accelerometer, gyroscope, GPS and compass to find the current position of the device in the world and display the data according to their input. These applications are mostly used outdoors to display additional text or images on top of the landscape. Example of such application is the Wikitude [11] Android application.

2. Computer vision aided applications: software which mostly depends on the camera (may use other sensors to aid the complex computer vision task) and uses feature tracking and image registration techniques to gain a correct position and proper context in the world. Such applications are used indoors to display 2D or 3D objects on other objects such as walls or tables. To make the computer vision algorithms robust and efficient it is common to request the user to print and put a known marker in the scene. The application will recognize the marker and replace it with the object to display. Example for such application is the Konstruct [12] iPhone application.
In this project we focus on the second type of applications. There are currently two major drawbacks for computer vision aided augmented reality applications:

1. Almost all of the applications don’t display the augmented data in real-time by introducing delays. This degrades the user experience of the application because of lags comparing to the movement of the device in the real world.
2. The displayed objects, which should naturally blend into the scene, don’t go through any visual transform before displayed on the camera input. Therefore, they are rendered to the user’s screen with their original chromatic properties and don’t fit to the real world. This breaks the seamless integration between the virtual and real world and spoils the user experience.

The purpose of this work is to create an Augmented Reality application on the iPhone 4 which will demonstrate that these two issues can be overcome simultaneously. The theme of the application is the same as described in the first paragraph of this introduction: a software piece that runs on a smartphone and allows a user to select a painting and see it on her wall as if it exists in the real world.

The success of the project will be measures by the following criterions:

1. Real-time interaction: the application must be responsive to the user movements in the world and work at the maximal frame rate available on the device (30 frames per second).
2. Accurate marker identification: the marker should be detected in reasonable lighting conditions (daylight to dark “romantic” light), in reasonable distance (0.5-3.5m) and in reasonable viewing angle (0-150 degrees between the user and the normal vector of the marker).
3. Image blending: images displayed on top of the marker should be transformed both geometrically and chromatically in order to adapt to the changes in the environment compared to the original geometry and chromaticity of the image.
3. Methods

3.1. The marker

In order to identify the region in the 3D world to put the desired picture at, we’ve used a special marker. The marker is a well-known image given to the user in advance to print and put on the desired wall used to display the picture on. The application is programmed (using computer vision and machine learning algorithms) to recognize the marker in an arbitrary scene and give details about its location and orientation.

The chosen marker is an image of 7 circles:

![Marker Image](image)

This image has some special properties: First of all, it’s very unique in natural scenes and therefore easy to detect. Second, by fixing the top row of three circles as the top of the marker we can easily tell the marker’s orientation (as will be explained later). Third, by knowing the real-world size of the marker, we can display a real-world fixed-size image or object on this marker using augmented reality.

To allow the user a wide view of the environment while looking at the blended image, we selected a marker size of 12.5x12.5 cm. This size is large enough to be easily detectable from the required distances as specified in part 2.

3.2. Development process

This section will briefly describe the development process without going into very technical details about the implementation.

The development process was spread across 3 main phases: designing the marker recognition algorithm using Matlab, porting the Matlab code to C++ using the OpenCV library and writing the iPhone application using Objective-C and C++.

3.2.1. Marker recognition algorithm design

First of all, we’ve captured sample videos of several scenes with and without the marker on the iPhone. In order to do so we wrote a simple video recording iPhone application because the default video recorder application that comes with the iPhone doesn’t support capturing video
in medium resolution (480x360 pixels) with custom camera settings (such as disabling the auto-exposure mode).

Then, we imported the movies to Matlab and started working on the algorithm. The Matlab environment is equipped with a lot of computer vision / image processing functionality which makes the algorithm development very easy to implement and validate across the various recorded movies.

### 3.2.2. Porting Matlab code to C++

When the recognition algorithm supplied the desirable results, we re-wrote it in C++ using OpenCV under Microsoft Visual Studio 2010. OpenCV is an open-source library written in C which includes a large base of computer vision algorithms implementations in C. The library is highly optimized to work with Intel’s x86 processors, but can be ported to other architectures (such as the iPhone). Because OpenCV doesn’t have a connected component labeling and analysis tools, we’ve also used the cvblob library [10] that uses the same interface as OpenCV and thus makes the development easier.

The final code runs under Windows with 45.4 FPS (Frames Per Second).

### 3.2.3. iPhone application development

Having the C++ code working, we proceeded to writing the iPhone application. The iPhone architecture uses Objective-C (or Objective-C++) as its primary language, which is a superset of C (or C++). Using this property we were able to port the C++ PC code to iPhone almost instantly. The application’s GUI and user interaction (using touch gestures) were implemented in Objective-C. However, because the iPhone 4 is an embedded device with relatively weak CPU and because OpenCV is not optimized for its hardware (ARMv7 processor) the performance of the code was very low and achieved only 5.4 FPS. This result is far from the desired 30 FPS required for real-time feedback.

After heavy optimizations (some of them will be further explained in the following part) that was based on replacing OpenCV functionality with code that takes advantage of the iPhone’s GPU (Graphics Processing Unit) and knowledge of iOS internals, we’ve succeeded to raise the FPS to 32.1 FPS and gain a smooth, real-time experience to the user.
3.3. The processing pipeline

3.3.1. Input

A video frame is generated by the hardware (Figure 3a). We receive the frame as a buffer with a metadata structure to our application through a callback function. This function is called at the maximal FPS rate of the hardware which is 30 times per second. Any delays processing the image will cause future frame drops and therefore should be avoided.

To enhance performance, we define the ROI (region of interest) as the bounding box \( B = \{ P_1, ..., P_3 \} \) of the marker that was found in the previous frame (if we did find any). Then, in the current frame, we process only the sub-image in the bounds \( B \cdot R(w) = \{ P_1 \cdot R(w), ..., P_4 \cdot R(w) \} \) where \( R > 1 \) is a resizing factor which depends on the largest diagonal \( w \) of the marker found in the previous frame.

3.3.2. Grayscale conversion

The entire marker detection processing is done in the intensity plane, and therefore the color input frame needs to be converted to grayscale (Figure 3b). In a preliminary iPhone version the input frame was given in RGB format, which requires conversion in \( O(n) \) by converting each pixel independently. However, it’s possible to configure the camera controller to output each frame in YCbCr 4:2:2 format which gives the gray channel automatically (on the Y plane), and therefore saves the conversion time and this entire step (As shown in figure 4).
3.3.3. Adaptive thresholding

Next we’d like to convert the image to a binary image to use a binary component analysis algorithm. Two methods of conversion using thresholding are implemented in the application:

**Global thresholding:** in this method we define a threshold \( T \in [0, 255] \) where this threshold applies for all the pixels in the image. Formally let \( I(x, y) \) be the input image. The threshold image is:

\[
I_T(x, y) = \begin{cases} 
0, & x < T \\
255, & x \geq T 
\end{cases}
\]

This method is fast to implement and because of its simplicity yields a low execution time and can even be parallelized using a fragment shader on the GPU. However, few problems arise. First, we need to find a valid thresholding value \( T \) for each frame, and second and more important, such value may not exist in scenarios where the scene is illuminated with spatially slow changing light. In this case, each pixel (or a group of pixels) should have a different threshold value. Because of that reason the following method is preferred:

**Adaptive thresholding:** this method applies a different threshold \( T(x, y) \) for each input pixel. The output image is calculated as:

\[
I_T(x, y) = \begin{cases} 
0, & x < T(x, y) \\
255, & x \geq T(x, y) 
\end{cases}
\]

There are many methods for calculating this threshold. We’ve chosen Bradley et al. method [2] of calculating the integral image and looking at the average of each pixel’s neighborhood to create the threshold function. This gives a good thresholding for our needs that runs in linear time with independence on the neighborhood size.

Figure 5: Comparison between global and adaptive threshold. (a) Input image, (b) Globally thresholded image with \( T = 100 \), (c) Adaptive threshold with block size = 50 and \( T = 100 \)
3.3.4. Blob analysis

In this phase we move from the image plane to a list of connected components in the image, with geometrical properties of each one. The algorithm we use was invented by Chang et al. [1] and implemented in the cvblob library. This algorithm runs twice faster than traditional algorithms because it requires only a single iteration over the image.

After the run of the connected component analysis algorithm, we’re left with all of the connected objects in the image with properties such as zero, first and second moments, major and minor axis length, centroid, perimeter and area.

First, we remove very small objects (with area $A < A_{\text{min}}$). This is a rough approximation to the use of the morphological operators "open" and "close", which take a relatively long time to process. Then, we filter the non-circular objects by calculating their unit-less circularity value:

$$C = \frac{4\pi A}{p^2}$$

Here $P$ stands for the blob’s perimeter. It’s easy to see that as $C$ gets closer to 1 our shape becomes more circular. We define a lower and upper bound for values of $C$ and keep only the objects with circularity in this interval. Note that because of perspective effects the marker’s circles may transform to wide ellipses in the image plane, therefore the circularity bounds are not very strict and we use this method to remove objects that doesn’t look as part of the marker at all.

![Figure 6: Filtering blobs in thresholded image. (a) input image, (b) thresholded image, (c) blobs after connected component analysis and filtering (each non filtered blob is painted with color different than white)](image)

3.3.5. Finding the marker

Given the $n$ blobs from the previous stage, we move to the feature space $(C_x, C_y, A)$ where $C$ stands for the $(x, y)$ position of the blob’s centroid. Note that we need to normalize the area $A$ to have units of pixels. We run mean shift [15] on this feature space $n$ times where each run starts at blob $i \in \{1, 2, ..., n\}$. We search for 7 starting points that at the end of the algorithm run
converged to the same point in the feature space and have 7 circles in its attention radius. These are the 7 circles of the marker.

Next we find the circles that lie in the corners of the marker: we search for 3 lines that intersect with 3 circles of our marker. These are the lines that give minimal $L_2$ norm distances to the centroids of the circles. We then take the outer blobs of each line to get the 4 marker corners $\{(x_1, y_1), \ldots, (x_4, y_4)\}$.

Finally, we find out the orientation of the marker by finding the “up” vector of the marker. We observe the line which intersects only 2 circles and calculate its normal vector $a + t\vec{b}$. The vector should have an intersection with a previously found marker line for $t > 0$, otherwise we flip $\vec{b}$.

![Figure 7: Marker found in the image](image)

### 3.3.6. Transforming source image and blending

We now have the 4 points of the marker and the up vector. Given the image we want to blend, we define the points $\{(x'_1, y'_1), \ldots, (x'_4, y'_4)\}$ as the top-left, top-right, bottom-left and bottom-right pixel coordinates of the image respectively. Next we need to create a correspondence between each point in this image to the points we found in the previous step. We use the up vector to arrange the points in a consistent manner (so that the first point will always be the top-left point of the marker when it’s up vector points towards the positive y axis and so forth).

We look for a homography $H_{3\times3}$ such that for each $i \in \{1,2,3,4\}$:

$$\begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{pmatrix} \begin{pmatrix} x'_i \\ y'_i \\ 1 \end{pmatrix}$$

This equation system is easy to solve with the 4 given points (8 equations and 8 variables).

For the color transfer of the source image, we sample the value $L$ of the pixel at the center of the marker. This pixel should have a value of bright white (255 on all channels) in natural light, and will change its value upon different illuminations (because illumination is a multiplicative
factor to the material’s albedo). Therefore the value of the pixel is a coarse approximation to the light reflected from the marker.

The final step in the process is to multiply each pixel in the source image $S(x, y)$ with the sampled color $L$ and then warp it using the homography to the scene (using backward warping). A naïve implementation will implement this algorithm using the CPU. Such implementation will work flawlessly in real-time under modern PC’s, but doesn’t scale well to embedded devices such as the iPhone. A better approach is to use the GPU’s vertex and fragment shaders to create the appropriate geometric and color transforms:

```
uniform mat4 modelViewProjectionMatrix;
attribute vec4 position;
attribute vec4 inputTextureCoordinate;
varying vec2 textureCoordinate;
void main()
{
  gl_Position = position * modelViewProjectionMatrix;
  textureCoordinate = inputTextureCoordinate.xy;
}
```

**Code 1: The vertex shader**

```
varying highp vec2 textureCoordinate;
uniform sampler2D targetImage;
precision mediump float;
uniform vec4 backgroundColor;
void main()
{
  gl_FragColor = backgroundColor * texture2D(targetImage, textureCoordinate);
}
```

**Code 2: The fragment shader**

The source image is uploaded to the GPU as a 2D texture, the sampled color value $L$ is defined as a 4D vector (RGB + alpha channel) and the homography $H_{3x3}$ is mapped to the model view and projection matrix $M_{4x4}$ as:

$$
M = \begin{pmatrix}
H_{11} & H_{12} & 0 & H_{13} \\
H_{21} & H_{22} & 0 & H_{23} \\
0 & 0 & 0 & 0 \\
H_{31} & H_{32} & 0 & 1
\end{pmatrix}
$$

Moving the computational load from the CPU to the GPU completely removes the processing bottleneck of this phase and enables real-time processing on the iPhone.
3.4. The application

The iPhone application uses the previously discussed processing pipeline with a simple user interface that allows the user to select an image to blend in the scene and select its real-world size. The images should be uploaded to the iPhone’s photo database beforehand and may include an alpha channel (to display a non-rectangular image). The application has 3 modes: Off (no marker detection and blending are made), Debug (detects the marker and shows the underlying mechanism of the detection pipeline on top of the camera input) and Blend (detects the marker and shows a blended image on top). When in blend mode, the user can change the displayed image’s size using a pinch touch gesture and adjust its size to the desired width and height in the real world.

Figure 8: Main screen of the application

Figure 9: Application in debug mode

Figure 10: Image selection window

Figure 11: Application in blend mode. User is given the real size of the blended image
4. Conclusions

4.1. Summary

We’ve shown that it’s possible to blend images into real scenes in real-time using augmented reality on mobile platforms. Our application runs in regular distance to the marker (more than 30cm) at more than 30 FPS, which gives a nice smooth experience to the user. Unfortunately, there are only few mobile Augmented Reality apps that give that kind of experience at present. We believe this is due to the high optimizations and knowledge needed to design and create such application.

A major aspect of the project dealt with converting real-time computer vision and image processing code from a PC environment to the embedded environment of the iPhone. In such environments, the knowledge of available hardware and software internals (such as low-level API and OS-hardware interaction) is crucial to gain high-performance abilities. We had to reflect this in the algorithm design, where accuracy (here, the correct marker identification percentage) is traded for faster execution. For example, the marker identification algorithm was designed in the lines of the previous observation: we wanted a fast and robust way to find the marker and therefore we traded the possibly generic methods of machine learning with a heuristic algorithm that is specifically stitched to the marker’s geometry.

We’ve also observed that high-performance image processing algorithms on smartphones should be designed in a way that allows them to be parallelized on the GPU because of the large amounts of data that needs to be processed every frame. The warping and color transfer of the source image cannot run on a single-core CPU, and wouldn’t run even on a dual-core CPU (as we start seeing in the mobile industry in the past few months) when dealing with higher resolutions.

A blending effect was also built and demonstrated in this application. The light approximation from one pixel worked better than expected (see video demo of the project), and can be extended in many ways and forms (more about it in the next section). As far as we know, this is the first Augmented Reality application which focuses on matching embedded objects to the scene. We believe this is the first step to a wide research on this topic.
4.2. Future work

4.2.1. Inter-frame pose estimation

Currently there’s no inter-frame pose estimation algorithm implemented in the application. This starts to affect when watching the marker from a distance (more than 3m) as the blended image starts to shake a bit because of noisy input. To mitigate this problem, the noise, observed from sample movies that will be recorded on the iPhone should first be fit to a statistical model and then a filter (such as Kalman filter) should be designed and applied to remove the noise. We assume that implementing such filter won’t affect the real-time property of the algorithm.

4.2.2. Support multiple markers

We’d like to support multiple markers in the same scene to allow the user to blend two or more images simultaneously. This feature requires visual changes in the marker to define it in a unique way in the scene and a change in the processing pipeline. The first can be done by drawing a unique barcode at the center of each marker. The second can be done by warping the marker to a plane parallel to the camera and then run the new marker-ID detection algorithm to decide which marker is it.

4.2.3. Generic marker detection

As specified before, the current marker identification algorithm was designed for a specific marker type. However, it’s possible that a more generic algorithm using tools from machine learning can be adapted to this problem and run in real-time using the GPU. We can try and implement a SVM classifier with vectors of a relatively small dimension (R^{30^2}) that were learned using PCA. Such vectors assume that the possible marker is warped to a plane parallel to the camera plane and because of that the major real-time pitfall here is to warp all the possible markers in a single frame to the same plane. However, our current implementation of the image warping using the GPU shows that it’s probably possible to warp such patches and still keep the real-time demands of the algorithm.

4.2.4. Better cloning

The current cloning method relies on a single pixel’s color value. This is of course a coarse approximation to the cloning that can be done in order to make the blended object look more real in the scene. Possible ways to enhance the cloning abilities are:

- Better environment sampling: sample more point on the marker and instead of using a sampled lighting factor \( L \) use a factor \( L(x, y) \) which is a smooth interpolant over the sampled points.
Real-time Image Blending for Augmented Reality on Mobile Phones

- Poisson cloning: use the Poisson cloning method such as in [3] to match the boundaries of the source image with the scene. This works best when the source image is not rectangular (includes an alpha channel which acts as a mask on the image).
- Motion blur and noise imitation: we’ve noticed that one of the issues that caused users to believe that the blended image doesn’t exist in reality is the fact that it looks too sharp and noiseless, comparing to the noisy input image that comes from the iPhone’s camera. To improve this it’s possible to imitate the noise by first modeling the real-world noise that comes from the camera’s CCD and then add noise with the same statistical parameters to the source image. Another issue is that such small CCD as in the iPhone is very prone to movements and shows a motion blur, even in outdoor scenes. One can measure the movement of the device using its built-in gyroscope and accelerometer and produce a motion blur of the source image to match the scene it will be blended into.
5. References


