PAPER KEYBOARD: A PRACTICAL IMPLEMENTATION OF OPEN CV

ZUBAIR MASOOD
B.A.I. Engineering
Final Year Project April 2014
Supervisor: Dr Fergal Shevlin
S/N: 10901248

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

ZUBAIR MASOOD

Name

02/04/2014

Date
ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Fergal Shevlin for his invaluable guidance and support throughout the project.

Furthermore I would like to thank my family and friends, who provided instrumental feedback and support throughout the project.
ABSTRACT:

The number of smartphones is increasing rapidly, primarily due to the advancements in technology which have made today’s smartphones durable, accessible, faster, more powerful etc. One driving factor is Android OS. Acquired by Google in 2005, today Android powers more than 79% of mobile devices worldwide.

Due to Android’s large market size a host of mobile applications (or simply apps) are being developed. One such section includes word processing apps. As the screen sizes of android phones have increased, more and more users are using their phone for viewing and editing documents. However to keep the phones slim, a physical keyboard is usually absent from such devices. This makes typing cumbersome.

The purpose of this project is to develop a solution to this problem: A “paper keyboard”. The paper keyboard involves a keyboard printed on a piece of paper which the user can use to type documents on a mobile device. The mobile device contains an application which uses the phones front camera to detect key strikes.

This report discusses in detail, the viability of such a keyboard, the key board design and implementation and the practicality of the solution.
Table of Contents:

Declaration 2
Acknowledgements 3
Abstract 4
Introduction 6
  Why Android & Project Motivation 6
  Development Tools 8
  Proposed Solution 10
Design 11
  Design Requirements 11
  Draft 1 12
  Draft 2 13
  Draft 3 14
  Draft 4 15
  Draft 5 16
  Final Keyboard Design 17
  UI Design 18
Implementation 19
  Application Overview 19
  UI Creation 20
  Virtual Keyboard 21
  Key Detection 22
  Key Stroke Detection 23
  Finger Detection 24
  Colour Blob Detection 25
Testing 27
  LogCat 27
  Toasts 28
  Response Time 29
  Changing Lighting Conditions 30
  Changing Background Conditions 31
  Colour Analysis 32
  Finger Calibration 33
  Key Stroke Detection Analysis 34
Conclusion 36
  Similar Products 36
  Appraisal 39
  Future Work 40
  Concluding Remarks 41
References 42

Source Code and Project Demo are located in a CD at the back of this booklet
Introduction:

Why Android? & Project Motivation:

In October 2003 Andy Rubin along with 3 other technology executives established Android Inc.\(^1\) to develop in Rubin’s words “smarter mobile devices”. On August 17 2005, Google acquired Android Inc\(^2\).

Today Android is the world’s most popular mobile operating system powering 79% of mobile devices worldwide\(^3\). Androids popularity transcends mobile phones. It has been used in televisions, game consoles, car dashboards etc.

Android is based on the Linux kernel, is open source and released under the Apache license\(^4\). Because of its open source nature and excellent development tools by Google, Android has a large collection of applications which extend the capabilities of Android devices. Due to the large market share, App development is a lucrative industry. As of May 2013 1 million apps have been downloaded 48 billion times\(^5\).

Despite at least 11000 distinct android devices\(^6\) most of the market share belongs to approximately 20 high end devices manufactured by Samsung, HTC, and ASUS\(^6\). These devices have extraordinary processing power, giving developers opportunities to develop power intensive apps.

Due to relatively larger screen sizes of these devices (> 4 in) word processing applications are really popular on such devices. For example Documents to Go, one of the most popular office suites has over 40 million downloads\(^7\).

However, even though large screens are excellent for reading documents. The screen size is not ideal for typing efficiently. Alternate keyboards are really successful. For Example Swift Key a redesigned keyboard app, is the
bestselling app in over 58 countries\textsuperscript{8}.

However alternate keyboards are still restricted to the screen size. The purpose of this project was to remove this restriction by developing an Android app which uses the front facing camera, and the openCV library to detect user inputs on a paper keyboard in real time.

\textbf{FIGURE 1: ANDROID SCREEN FRAGMENTATION}

\textbf{FIGURE 2: SWYPE AND SWIFTKEY ARE AMONG THE BEST SELLING APPS ON GOOGLE PLAY SHOWING CONSUMER INTEREST IN TYPING FASTER}
**Development Tools:**

**HTC Sensation XE** is the phone this solution was developed on. This is my primary phone, and was therefore always available to test on. There are certain features of the phone which made it the ideal candidate for this purpose. The phone boasts a high quality 4.3 inch screen (540 * 960 pixels). More importantly the phone contains a 1.5 gigahertz processor with 768 Mb ram. This processing power would allow resource intensive calculations later on. More importantly the front camera is of sufficiently high resolution making it ideal for this application. The back cover was used as support when running the application.

![Figure 3 : HTC Sensation XE](image-url)
Tegra Android Development Pack (TADP) 2.0:

This suite provided by NVIDIA contained almost all the development tools needed for this project\(^\text{10}\). Even though this suite is primarily designed for Tegra devices it worked perfectly on my phone. It contained the latest tools Android development tools coupled with openCV 2.4 and a host of NVIDIA’s tools. All the libraries were pre linked and all the paths were correctly configured. This effortless installation was in stark contrast to the steps required to configure each library and path independently.

Development Language:

Having previously worked with openCV in Visual Studio, I initially chose C++ as the development language. Not only was I much more familiar with the open cv C++ framework it is much more extensive than its Java counterpart (Java was only introduced in openCV 2.4.4\(^\text{11}\)). Similarly most of the literature available covers openCV in either C, C++ or Python.

However the openCV samples for Android changed my opinion. There were sample programs showing, how to open the camera view, take and manipulate images. I felt that this was an essential framework which to build my project on: therefore this project was developed in Java.

Paper Keyboard:

The actual keyboard was designed on PAINT.NET which is essentially a freeware image manipulation diagram. It was then printed on an A4 sheet of paper in colour. Even though laminating it or printing on a hard cover would have made it more robust, I kept it realistic as I wanted the end client to be able to easily replace/use it.
The primary advantage of this solution is ease of use. It is primarily aimed at college students who carry paper regularly but find it cumbersome to always carry laptops. As no other cost except that of printing the keyboard is borne by the final user, it is a relatively inexpensive solution.

Furthermore if successfully implemented the paper keyboard can be used in many other applications. Mechanical keyboards are expensive and impractical in public kiosks. The same principle can be used to make wireless gamepads, drawing tablets etc.

The next chapters look at how this solution was actually implemented.
Design:

Design Requirements:

“Design is the fundamental soul of a man-made creation that ends up expressing itself in successive outer layers of the project, or service.”

-Steve Jobs

The actual keyboard design was fundamental to the success of this project. The keyboard needed to fit the following specifications:

1. Must fit on an a4 page
2. Must contain at least all alpha numerical characters
3. Must fit inside the front camera view
4. Each key must be distinguishable from its neighbouring keys
5. Must contain extra keys that can be used for calibration
6. Must be intuitive to use

Designing a keyboard which met all the above criteria was always going to be difficult. The designs started with trying to fulfil one criterion (such as fit inside the camera view) and then they gradually progressed to fulfilling most of the above.

The following pages discuss the design process from the first draft to the final keyboard. After each design was made it was presented to friends and the supervisor. Their feedback was invaluable in providing guidelines on how to make the keyboard intuitive for the final user.
Draft 1:

This was the first draft. By positioning the phone on the black rectangle, the keyboard was consistently positioned inside the camera view. Due to the position of the camera the keys were aligned to the right side. The keys were clearly visible, however they were not distinguishable from the neighbouring keys. Due to the perspective distortion of the camera the numerical keys were not clearly visible. To resolve this the keys needed to move closer towards the camera or increase in magnitude. Nevertheless the character keys were intuitive (modified QWERTY) therefore this was sufficient for the earliest draft.
Draft 2:

By changing the orientation of the page (to landscape) the camera view could see more of the page. The overall size of each key was increased and a black background was chosen for better contrast during segmentation. This would allow the end user to print using a black and white printer thus increasing potential market size.

Instead of having dedicated numerical keys a shift key combination was chosen as it would be more practical. This is what allowed for the significant increase in key size.

Despite being a clear improvement from the last draft, the general feedback was that even this keyboard was too small. Therefore the size of the keys
needed to increase while still positioned inside the camera view.

Draft 3:

It was chosen to revert back to the vertical orientation for this draft. Camera distortion was taken into account by making each subsequent line bigger than the next one. Even though the keys were clearly visible and in theory easy to segment, there are some clear flaws. The number of keys is severely limited and the user would need to adapt to a new keyboard layout. Nevertheless taking distortion into account was inevitable and this was one possible solution going forward. One feasible solution to the capacity problem was adding more sheets, but this solution was dropped due to negative feedback.
This was a natural progression from the last draft. By using a diagonal orientation the number of keys could be maximised, while keeping them relatively far apart and of suitable size. The number of keys was sufficiently enough for characters however could not contain all the alphanumerical keys. This was the maximum number of suitable keys that could fit on paper; however the number was still not enough for a practical keyboard with all alphanumerical and special keys (shift, space etc.). Furthermore the feedback highlighted that this layout would be cumbersome for the user to adapt to. Therefore a completely different approach was needed.
Draft 5:

This was the penultimate draft. As shown this keyboard contained most of the keys present on a normal physical keyboard. So how was I able to significantly (almost double) increase the keys the camera could see? I moved the camera off the page approximately 5 inches back from the first row of keys. Even though this was by far the most practical draft yet, there were still problems. Firstly by moving the camera back meant a much more rigorous calibration mechanism would be required. Secondly the keys at the 2nd last row were so compact that the camera could not tell them apart. However the feedback was excellent and by tweaking this design a little bit, I was able to create the final draft.
Final Keyboard Design:

This design was chosen as the final keyboard. It is a modified version of the QWERTY keyboard present on most commercial netbooks. The 4 red corners were chosen as the calibration keys (discussed later). Even though the keys are not separated they are mostly distinct from their neighbours.

This keyboard fulfills most of the requirements stated in the design requirements section. It fits nicely on an A4 keyboard with room to spare (to allow the user to rest their hands). The general feedback was fantastic especially when compared to the other drafts. Most users expressed a genuine interest in the application when this design was presented as the final one. No physical distortion is made to the keys. This meant that this would have to be done later digitally.

The design process was time consumable but nevertheless fundamental to the overall success of the project. If earlier designs were chosen, ample problems which occurred later would never have had to be resolved; however the overall quality of the project would have suffered.

Having designed the physical keyboard, the next step was designing a suitable UI (User Interface).
UI Design:

“In many cases, the user interface to a program is the most important part for a commercial company: whether the programs works correctly or not seems to be secondary”

-Linus Torvalds

Creating a clean and intuitive user interface was equally as important as designing the physical keyboard. From onset it was decided that the program would start immediately instead of having a Menu. The lay out is simple. When the program starts a camera view opens with two windows. The top bottom is just the bottom window stretched. Then notifications appear frequently to guide the user (such as “calibration complete”). Similarly feedback is provided through small notifications on screen. An onscreen keyboard is also drawn, by aligning the virtual keyboard with the physical one the user enables the key detection algorithm to correctly identify the position of each key.
Implementation:

Application Overview:
The application primarily consists of one activity. Activities are essentially blankets which focus on multiple aspects of an application. Because there is essentially only one view, one primary activity is sufficient to encapsulate the entire application.

Android applications tend to have 4 states which describe the circumstances of the application. They are onCreate, onPause, onResume and onDestroy. These states create the basic guidelines outlining how the app behaves in each state.

OnCreate is called when the app starts initially. It is primarily used to initialize the user interface. In this app the onCreate function just starts the camera view and initializes any toasts (notifications) which may be needed later.

OnPause dictates what happens when the application is paused (for example the user receives a call). This is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, etc. In this application the camera view is disabled.

OnResume is called when the activity starts interacting with the user after onPause. In this application onResume resumes the camera view before the pause occurred.

OnDestroy is the final call received before the activity is destroyed. In this application the camera view is disabled.

The above functions are staple functions of most applications therefore nothing time consuming or complex is done in them. The most fundamental function of this application is onCameraframe(). On each frame the calibration counter is updated which is essentially a counter used for various functions. On each frame the keyboard is redrawn and the calibration counter updated. Depending on the frame counter either the calibration or key stroke detection algorithms are called. Essentially all the algorithms are processed in this function. The next few pages outline how these functions were implemented.
UI Creation:

The design section of the layout highlights the rationale behind the UI. Firstly the openCV libraries had to be reconfigured to target the front camera. After configuring the front camera the appropriate size for the zoom window (the part being zoomed) had to be chosen. Through experimentation it was found out that the last 20 % of the image was the required part.

```c
mZoomWindow = mRgb.submat(rows*88/100, rows, 0, cols);
```

The next part was stretching the last 20 % to fill the remainder of the screen. OpenCV contains a resize function which takes in an input matrix and resizes it to fit an output matrix. This function was used to stretch zoom window to the zoom corner (part holding the stretched image).

```c
Imgproc.resize(mZoomWindow, mZoomCorner, mZoomCorner.size());
```

Even though the size of the keyboard was now ideal, the orientation wasn’t. The keyboard was mirrored as expected but also inverted which made the UI unintuitive. To complete the UI the image was essentially flipped on both axis.

```c
Core.flip(mZoomCorner,mZoomCorner,-1);
```
Virtual Keyboard:

Creating the virtual keyboard was by far the most time consuming aspect of the project. First a key was drawn approximately. Then the keyboard was positioned. By judging how far the actual key was from the approximated one, a better approximation made. This step was repeated until the virtual keyboard could almost perfectly align with the real one. This process was repeated for each key. As the keyboard and UI adjustments were improved the whole was repeated.

The rectangle function from the core library was used to draw the virtual keyboard.

```java
// L-1
Core.rectangle(mZoomCorner, new Point(54,113), new Point((79),(138)), new Scalar(255, 0, 255, 255), 2);
/*Parameters
 mZoomCorner = input matrix;
 point(54,113 = first vertex);
 point(79,138 = vertix of the rectangle opposite first vertix);
 Scalar(255, 0, 255, 255)= colour;
 2 = line thickness;
 */
```

After the keyboard was appropriately drawn, the next step was drawing an outline. The purpose of the outline was to facilitate the user in aligning the real keyboard with the virtual one. The line function from the core library was used as a rectangle could not accurately encapsulate the keyboard sufficiently.

```java
// outline
Core.line(mZoomCorner, new Point(40,100), new Point(800,100), new Scalar(255, 255, 255, 255), 2, 8, 0);
/*Parameters
 point1,2 = start and end points of the line;
 Scalar(255, 255, 255, 255) = colour (white);
 2 = line thickness;
 8 = line type;
 0 = line shift;
 */

Core.newLine(mZoomCorner, new Point(110,100), new Point(690,270), new Scalar(255, 255, 255, 255), 2, 8, 0);
Core.newLine(mZoomCorner, new Point(110,100), new Point(690,270), new Scalar(255, 255, 255, 255), 2, 8, 0);
Core.newLine(mZoomCorner, new Point(690,270), new Point(690,100), new Scalar(255, 255, 255, 255), 2, 8, 0);
```
**Key Detection:**

Key detection is what turns coloured boxes on a paper to a keyboard. It is arguably the most important algorithm in this application. A lot of time was dedicated into ensuring a reliable key detection algorithm as without it, the application would ultimately be useless.

Each key is defined by the rectangular coordinates used to draw it.

```cpp
Core::rectangle(mZoomCorner, new Point(348,157), new Point((368),(173)), new Scalar(0, 0, 0, 255), 2,
Rect touchedRect23 = new Rect();
touchedRect23.x = 348;
touchedRect23.y = 157;
touchedRect23.width = 20;
touchedRect23.height = 15;
```

A new variable touched region which is the key region cropped is declared.

This region is then converted to HSV for further processing. Blob detection only works in grayscale, so the image must be split into its 3 channels. To preserve the original a new variable is used, and for the image to be split it needs to be in the HSV format.

```cpp
Improc::cvtColor(touchedRegionRgba23, touchedRegionHsv23, Improc.COLOR_RGB2HSV_FULL);
```

The RGB value is calculated and stored in a new variable (mBlobColorRgba) by summing the elements in HSV and then converting to RGB.

```cpp
// Calculate average color of touched region 23
mBlobColorHsv23 = Core::sumElems(touchedRegionHsv23);
int pointCount23 = touchedRect23.width * touchedRect23.height;
for (int i = 0; i < mBlobColorHsv23.val.length; i++)
    mBlobColorHsv23.val[i] /= pointCount23;
mBlobColorRgba23 = convertScalarHsv2Rgba(mBlobColorHsv23);
```

During the calibration phase this variable is summed to give a key value:

```cpp
```

This is the unique value which is used to identify each key. Its variance is used to determine a possible keystroke.
Key Stroke Detection:

After key detection, this is the most crucial algorithm. Appropriate key detection has three functions:

1. Registering the right key
2. Ignoring the other keys
3. Displaying the right key

The basic key detection algorithm is intuitive. After the calibration period has passed, each frame each key has its key value recalculated. If this value lies within a certain threshold a counter is updated. If this counter reaches a certain value the key is registered as being pressed and a notification pops up displaying the key. Even though this algorithm is sufficient for detecting keystrokes it does not help in ignoring other keys (for example keys below would have their key values changed as well).

To counteract this problem a row evaluation system is used. Initially each row has its Boolean set to false. When a keystroke is detected the rows Boolean is set to true. For a key to be displayed, all the Booleans above must be set to false.

```java
//
if (calcounter2 > 250 && 1 == false )
{

t int ukey2 = (int) mBlobColorRgb23.val[9]; (int) mBlobColorRgb23.val[9] = (int) mBlobColorRgb23.val[9];
if (ukey2 > (ukey -70) \&\& ukey2 < (ukey -30) )
{
    ucounter1++; 
    L1 = true; 
}

if (ucounter == 5)
{
    u.show();
}
if (ucounter > 8)
{
    L1 = false; 
    ucounter = 0;
}
}
**Finger Detection:**

One possible solution to the shadowing problem is finger detection. Ideally the finger should be calibrated at the start of the calibration process. This would allow more robust key stroke detection, as keystrokes from other variables could be ignored. A highly robust finger detection algorithm could also eliminate interference from hand gestures, or sudden changes in the environment.

A finger calibration method was developed towards the conclusion of the project. Due to time constraints, finger detection was ultimately dropped from the final solution.

After the application has started and a few seconds have elapsed to allow the user to align the keyboard, a “beginning calibration” message appears. The user puts his finger on the calibration key (beside the right shift key) until the “calibration complete” message appears. During this time (50 frames) the key value of the calibration key is summed. This sum is then divided by the number of frames to get the average finger key value.

By averaging over a large number of frames an accurate finger value is obtained. However another reason this was dropped was that the key value changes depending on the position of the key. Essentially this meant that calibration would have to occur on both sides. This would have doubled the calibration time, therefore I decided against implementing finger detection.

```java
if (calcounter2 == 100)
{
    ffing1 = fing1/50;
    ffing2 = fing2/50;
    ffing3 = fing3/50;

timer2.show();
    calibrationcompleted = true;
    finger = ffing1 + ffing2 + ffing3;
}
```
**Colour Blob Detection:**

Initially blob detection was selected as the principle method of key detection. In this section the pros and cons of blob detection will be outlined as well as why it was not implemented in the final solution.

Blob detection is basically a tool for detecting regions which differ in basic properties such as colour in contrast to neighbouring areas. A single blob is therefore a region of interest which contains elements of similar properties when compared to its neighbouring areas.

Blob detection has various applications. It is used for object tracking and object recognition. Blob detection can also be used for texture tracking and recognition\(^{14}\). Another application is peak detection when analysing histograms.

There are numerous blob detectors, but mathematically their base is similar\(^{15}\).
Blob detection is scale invariant which makes it ideal for applications in which the target objects suffer from perspective distortion. OpenCV android is packaged with a sample application which demonstrates how to implement blob detection.

Initially blob detection was implemented on the corner keys:

```java
mDetector.setHsvColor(mBlobColorHsv);
Improc.resize(mDetector.getSpectrum(), mSpectrum, SPECTRUM_SIZE);
mDetector.process(mRgba);
List<MatOfPoint> contours = mDetector.getContours();
Log.e(TAG, "Contours count: " + contours.size());
Mat colorLabel = mRgba.submat(4, 68, 4, 68);
colorLabel.setTo(mBlobColorRgb);
```

After the average colour of the key was calculated, it was passed to the ColorBlobDetector activity. Here using blob detection the average colour was calculated then displayed on screen. A spectrum of all the colours was also displayed. Finally the number of contours (Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity\(^{16}\)) was recorded.

Even though the result was impressive it was unnecessary. Compared to simply summing the elements this was slow and inefficient. Ideally blob detection could be used to detect the corner keys, so the user does not have to align the keyboard. Similarly it could be used for finger tracking to improve the robustness of the key stroke detection algorithm.

Blob detection is an invaluable tool in computer vision. It can be used to track objects regardless of spatial distortion. However once the virtual keyboard was fully implemented blob detection’s processing time was too much to be used for each key.
Testing:

LogCat:

The purpose of software testing is to determine the overall quality of the product. It should be done in a methodological manner to evaluate the quality – related information of the product\(^{17}\).

The Android logging system provides a mechanism for collecting and viewing system debug output. Logs from various applications and portions of the system are collected in a series of circular buffers, which then can be viewed and filtered by the LogCat command\(^{18}\).

LogCat was used extensively when testing different aspects of the project. Whenever the application was started directly from eclipse, the LogCat window started displaying a constant stream of messages about the state of the application such as “OpenCV loaded successfully”. This is primarily due to the excellently written openCV library.

OpenCV library facilitated easier debugging. When the program crashed it logged possible exceptions. For example if the dimensions of the virtual keyboard keys exceeded the screen size, the application crashed, however LogCat displayed that the size of matrix was greater than the screen size.

To handle the large number of messages, LogCat provides an excellent filtering system. By using filter words such as colour, LogCat can be requested to display only the messages which contained the tag colour.

LogCat was used extensively when measuring performance metrics such as response time (discussed later).
Toasts:

LogCat only works when the device is connected to the computer. The phone cannot be connected to the computer and be placed near the paper keyboard in the right position due to the location of the usb port. Therefore an alternate method had to be devised to make sure the key values metrics were as expected.

The solution was to use small notifications that pop up on the screen when required. A toast provides simple feedback about an operation in a small popup. It only fills the amount of space required for the message and the current activity remains visible and interactive. For example, navigating away from an email before you send it triggers a "Draft saved" toast to let you know that you can continue editing later. Toasts automatically disappear after a timeout\textsuperscript{19}.

However unlike LogCat updates, managing toasts was challenging. If two toasts occurred at the same time the application would sometimes crash. Toasts also only display strings so the metrics were converted to strings before they could be displayed.

Displaying a toast was done as follows. First the toast was initiated in the OnCreate state. Then it was shown when appropriate. To avoid toast collisions a gap of 50 frames was preferred.

```java
CharSequence text = "Calibration Failed!";
int duration = Toast.LENGTH_SHORT;
Context context = getApplicationContext();
toast = Toast.makeText(context, text, duration);
toast.show();
```
**Response Time:**

Response time is one of the most crucial performance characteristic of any application. As stated before LogCat only works if the phone is plugged in and toasts are time consuming thus unsuitable for this task.

Therefore Response time without the keyboard needed to be measured. To achieve this, a white sheet of paper was held against the phone screen and the response of the “A” key recorded.

```java
Log.i(TAG, "A key value: " + akey + " calcounter: " + calcounter2 );
```

As can be seen response time was measured with reference to the calibration counter which is updated every frame. To get a better understanding of the results the frame rate needed to be calculated. To calculate the frame rate a stop clock was operated when the calibration counter was between 50 -150. After performing the experiment 5 times, an approximate frame rate of 10 fps was calculated.

When the cal. counter reached 100 a white piece of paper was held up. Therefore the value spiked initially then remained around 300 (which reflects the true value), then returned to the previous low. As the spike was near instant it shows that the response time is instantaneous and can be approximated at 150 ms. As human response time is 250 ms, in theory if designed perfectly there should be no lag.
Changing Lighting Conditions:

For the keyboard to be practical, it needs to be robust to work in different environments. Most computer vision systems in the industry have constant lighting, or controlled variables. However a keyboard cannot request such privileges. As long as the paper and the phone are kept relatively stationary the keyboard should work.

To analyse the response to different lighting conditions, key values were recorded multiple times in two distinct lighting conditions. In the first case lighting conditions were kept ideal; in the second case a heavy book was placed beside the keyboard to produce constant shadows. Toasts were used every 50 frames to display the key value.

<table>
<thead>
<tr>
<th>Room Lighting</th>
<th>Run</th>
<th>Z</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>I</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>305</td>
<td>244</td>
<td>288</td>
<td>265</td>
<td>232</td>
<td>364</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>295</td>
<td>236</td>
<td>267</td>
<td>253</td>
<td>237</td>
<td>352</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>286</td>
<td>237</td>
<td>242</td>
<td>273</td>
<td>271</td>
<td>359</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Lighting</th>
<th>Run</th>
<th>Z</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>I</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>171</td>
<td>186</td>
<td>149</td>
<td>204</td>
<td>252</td>
<td>360</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>182</td>
<td>192</td>
<td>153</td>
<td>209</td>
<td>260</td>
<td>363</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>165</td>
<td>179</td>
<td>134</td>
<td>213</td>
<td>265</td>
<td>361</td>
</tr>
</tbody>
</table>

This table shows that in constant lighting even though small deviations occur the key values are similar. The slight change is caused by how the key board is aligned. Even though shadows caused a decrease in the key values it was generally consistent. This shows that if a proper key stroke detection mechanism is used results should be consistent.
Changing Background Conditions:

One aspect of environment which is likely to change is the background. Since the last 20 % of the camera view is stretched, all the changes are also magnified. Even though the keyboard is displayed perfectly, one needs to remember that this is a 2d representation of a 3d world.

There are three backgrounds that were used for this test. In the first scenario it is just me and a stationary background. In the 2\textsuperscript{nd} case it’s me on a different day in different clothes and in the third day the background is dynamic (basically a friend walking back and forth).

Again the key values were calculated for the letters which spell out my name. Like last time the first reading was taken after 200 frames and each subsequent value was recorded in intervals of 50 frames. To make the experiment authentic the keyboard was realigned each turn. All other variables were kept constant.

<table>
<thead>
<tr>
<th>Stationary 1</th>
<th>Z</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>I</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>306</td>
<td>250</td>
<td>290</td>
<td>262</td>
<td>237</td>
<td>383</td>
</tr>
<tr>
<td>2</td>
<td>269</td>
<td>231</td>
<td>263</td>
<td>241</td>
<td>251</td>
<td>348</td>
</tr>
<tr>
<td>3</td>
<td>274</td>
<td>245</td>
<td>277</td>
<td>254</td>
<td>242</td>
<td>355</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stationary 2</th>
<th>Z</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>I</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>310</td>
<td>227</td>
<td>283</td>
<td>274</td>
<td>184</td>
<td>375</td>
</tr>
<tr>
<td>2</td>
<td>307</td>
<td>210</td>
<td>279</td>
<td>262</td>
<td>193</td>
<td>370</td>
</tr>
<tr>
<td>3</td>
<td>304</td>
<td>203</td>
<td>285</td>
<td>255</td>
<td>201</td>
<td>377</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic</th>
<th>Z</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>I</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>296</td>
<td>235</td>
<td>287</td>
<td>257</td>
<td>245</td>
<td>386</td>
</tr>
<tr>
<td>2</td>
<td>287</td>
<td>215</td>
<td>276</td>
<td>264</td>
<td>263</td>
<td>363</td>
</tr>
<tr>
<td>3</td>
<td>292</td>
<td>201</td>
<td>291</td>
<td>271</td>
<td>232</td>
<td>358</td>
</tr>
</tbody>
</table>

These numbers suggest that background has a negligible effect on key values. In stationary 1 and dynamic I was wearing a luminous jersey which may justify the low values of “I” in stationary 2.
Colour Analysis:

Selecting the most appropriate colour for each key was also necessary. Even though the primary aim of colours was to differentiate between keys, the colour was also dependent on the position of the key. As the keys at the top behave differently when compared to keys at the bottom choosing an appropriate colour was required.

To achieve this, a better understanding of how each colour behaves was required. So a 1 cent coin was wrapped in a unique colour and placed in different key positions. The key detection algorithm was run, to determine the change in colour depending on position. Again letters which spell out my name were used. Dark shades of green were to get contrasting results (light green is basically yellow).

<table>
<thead>
<tr>
<th>Colour</th>
<th>Z</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>I</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>301</td>
<td>378</td>
<td>304</td>
<td>300</td>
<td>373</td>
<td>390</td>
</tr>
<tr>
<td>Green</td>
<td>161</td>
<td>237</td>
<td>166</td>
<td>158</td>
<td>235</td>
<td>253</td>
</tr>
<tr>
<td>Red</td>
<td>284</td>
<td>364</td>
<td>287</td>
<td>276</td>
<td>358</td>
<td>374</td>
</tr>
<tr>
<td>Blue</td>
<td>172</td>
<td>226</td>
<td>154</td>
<td>149</td>
<td>221</td>
<td>236</td>
</tr>
</tbody>
</table>

As predicted yellow and red performed similarly whereas green and blue behaved in a similar fashion. The values of green and blue were very low in the lower half of the keyboard (Z and B). This meant that dark colours could only be used in the upper half of the keyboard. This is reflected in the final keyboard design as the Z, A and B keys are essentially different composites of red and yellow. Similarly dark colours are used mostly in the upper half of the keyboard.

To keep the keys distinct green and blue were used sparingly in the lower part, however lighter shades were used. Similarly darker shades of yellow and red were used in the upper half.

Colour analysis allowed the design of the keyboard to be done in a more systematic manner, ensuring better key detection performance.
Finger Calibration:

Ideally, the application should recognise a finger and attempt to track its position. This would prevent false positives in terms of key stroke detection. If this algorithm is robust, other object interference would have negligible effect on keystroke detection.

Due to the many advantages of finger calibration, a finger calibration feasibility test was done. The first part of the test involved testing a fingers key value and then comparing it with other objects’ key value. To measure the key value the finger (or object) was placed beside the right shift key. Then the key value was recorded each frame for 50 frames. This was then averaged to obtain a single key value. The results are as follows:

<table>
<thead>
<tr>
<th>Run</th>
<th>Finger</th>
<th>Pen</th>
<th>Pencil</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>507</td>
<td>542</td>
<td>538</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>539</td>
<td>544</td>
</tr>
<tr>
<td>3</td>
<td>537</td>
<td>541</td>
<td>568</td>
</tr>
<tr>
<td>4</td>
<td>503</td>
<td>512</td>
<td>526</td>
</tr>
<tr>
<td>5</td>
<td>490</td>
<td>525</td>
<td>557</td>
</tr>
</tbody>
</table>

It is clear from the results above that finger detection with the aim of avoiding interference is unreliable. Not only was the finger least consistent, it was also improbable to differentiate from other objects. The next test was designed to see weather finger calibration would aid key stroke detection. Up to now fixed thresholds were used to determine a key stroke. If dynamic thresholds could provide a more reliable result, this would be the way forward. To test this each key value was recalculated with the finger on the key.

<table>
<thead>
<tr>
<th>Finger V</th>
<th>576</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>I</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>328</td>
<td>404</td>
<td>330</td>
<td>329</td>
<td>460</td>
<td>412</td>
</tr>
</tbody>
</table>

Finger

<table>
<thead>
<tr>
<th>Z</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>I</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>253</td>
<td>334</td>
<td>267</td>
<td>254</td>
<td>382</td>
<td>338</td>
</tr>
</tbody>
</table>
As the value of the finger changes depending on the position, knowing the initial finger is useless. The results gathered fall in the thresholds, therefore dynamic thresholds were not used. These results were recorded on an exceptionally bright day therefore the key values are a little higher than usual.

**Key Stroke Detection Analysis:**

Ultimately key stroke detection accuracy is the most important performance metric. If it is below par, the application is useless. Therefore sufficient effort was made to ensure high accuracy.

Testing was simple. A key was pressed and the output was recorded. The results were as follows:

<table>
<thead>
<tr>
<th>Key Pressed</th>
<th>Output</th>
<th>Key Pressed</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>B</td>
<td>N</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>Q</td>
<td>W</td>
</tr>
<tr>
<td>E</td>
<td>R</td>
<td>R</td>
<td>Q</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>G</td>
<td>D</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>J</td>
<td>J</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>K</td>
<td>K</td>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>Y</td>
<td>U</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>Z</td>
<td>Z</td>
</tr>
</tbody>
</table>

Even though most of the results were as expected, there were some clear miss matches. Most of the errors involve neighbouring keys, so it might have been due to poor alignment etc. This was one of the “better” runs as no phantom keys (key appearing when nothing is touched) appeared.
It is important to note that due to time constraints only the alphabetical keys were implemented. Also during testing the keyboard was attached to a paper clipboard to keep it stationary.

To validate the results, the experiment was repeated numerous times in different areas. The following results were recorded:

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Phantom keys</th>
<th>Max Delay (s)</th>
<th>Incorrect keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

The maximum delay usually occurs at the first key press as there is a 50 frame pause after initial calibration. Phantom keys vary depending on the quality of the alignment. However the question arises, what is causing the incorrect keys to appear? The answer is probably shadows. As most of the incorrect keys are neighbouring keys, shadows from the hand must trigger wrong key stroke detections. Significant improvements in the thresholds can still be made to reduce this problem.

Nevertheless the solution works. Rigorous testing led to the right colours to be chosen for the keyboard, appropriate thresholds strategic algorithmic decisions were also made on the basis of these test results.

Testing took significant time. However, without it, ample mistakes would have been made which would have been far costlier when compared to the time it took to complete suitable testing.
Conclusion:

Similar Products:
Similar products validate the fact that a solution is required. Also similar products might offer insight on how to improve this application. In this section some of the existing solutions are critiqued.

Vibrating Keyboard:
Florian Kraeutli application uses the iPhone’s accelerometer to track the vibrations caused by tapping on a key. Key detection is essentially done by analysing the vibrations\textsuperscript{21}.

Florian has stated that this application has an accuracy of 80 % and is limited by the hardware of the iPhone. Florian’s application is resistant to change in lighting background etc. and uses predictive text to minimise errors. Although an excellent solution it has some flaws. All the demo videos show the application working on the same surface, so its performance in other environments cannot be evaluated. Also by limiting it to the iPhone, the potential market size is diminished. The calibration process is long and unintuitive. Nevertheless this solution is impressive.
Celluon Epic Laser Keyboard:

Celluon Epic is a commercial product unlike the previous solution. For $199 a small transmitter is provided which projects a laser keyboard. The Epic can be synced to most devices via Bluetooth. It is a full QWERTY keyboard, and in theory promises a full typing experience. The response time is fast and no calibration is required. As it is projected on the surface, it should withstand noise, background change, lighting change etc. The customer reviews on this product are mixed. Most customers like the idea but are not happy with the learning curve required. Cost is also a disadvantage.

My application was designed to be accessible due to its low cost. Also since the Epic pairs via Bluetooth, the end user would be better off purchasing a physical Bluetooth keyboard.

This technology has better alternatives. It can be used in kiosks where the wear and tear costs are huge. It can also be used to project controls in showrooms, class rooms, lecture theatres etc. If the cost is ignored this is a practical and novel solution.
Paper Keyboard by Gyorgyi Kerekes:

After I had started designing my keyboard, in November 2013 Gyorgyi Kerekes further validated my idea by releasing a paper keyboard for the iPhone. The principle technology behind his application is unknown, but behaves similarly to this application; A QWERTY keyboard is printed on a piece of paper and placed in front of the phone. The phone uses its front camera to detect key positions and keystrokes.

Unfortunately due to the lack of an iPhone I didn’t get the opportunity to use the application myself. Nevertheless the response to this app has been positive. This is a fully featured application which includes built in chat and email client. Like my application this application suffers from shadows. The keyboard is spaced apart and lacks numerical or special keys. The key stroke detection algorithm is also primitive in that only one finger can be used at a time. Nevertheless the application UI looks impressive and built in games will help it reach a bigger target market.

This is also the only application of its kind on the App store and no similar
application is available on the Google Play Store.

**Appraisal:**

There are some aspects of this solution, which were implemented exactly as the initial design, while others could have been implemented better.

The user interface implementation is commendable. The final implementation looks exactly like the design. It is minimalistic yet intuitive. Some improvements can be made on the virtual keyboard to make it aesthetically pleasing.

Similarly the key detection algorithm is robust. It is highly accurate and adaptable. It is also relatively quick and intuitive. Some a priori knowledge (what colour to expect) can be fed in to it to make it even more robust.

Unfortunately the final key stroke detection algorithm is lacking. At best it can detect a single finger keystroke efficiently. To ensure accuracy the response is slow, however from the end user point of view it should be faster than it is. Even though it is adaptable, it is not as robust as the other algorithms. However the speed and accuracy can be improved, if the thresholds are calculated from a much larger sample space (more tests). The effect of shadow also needs to be eradicated at this point.

Similarly the initial calibration can be improved. An initial system was built to detect if the keyboard moved significantly; it was also designed to inform the user if the calibration had failed. Due to time constraints this was not fully implemented. The 4 corner blocks were designed to aid initial calibration, and can still provide a viable solution to this problem.

More testing could also have been done to produce a better result. The tests should have been done with multiple users to make the results more authentic. Similarly more devices should have been used to see the differences in performance between devices. Unfortunately the resources available limited the quality and quantity of the tests performed.

Nevertheless the overall solution is highly commendable. For a proof of concept it works brilliantly and with more resources has the potential to be a highly successful application.
Future Work:

Significant work can be done to turn this solution into a viable product. Displaying the user keys without storing them is useless. Ideally an overlay textbox should be developed which pops up after the calibration to show and store the text as it is typed. The text box should use predictive text dictionaries to improve the overall accuracy. The user should then be able to either paste the text in their chosen word processing application or save it for later use.

By using simple “hacks” the performance of the program can be significantly improved. For example if all the calculations are done in only odd numbered frames, performance is almost doubled. Similarly in between words the key stroke detection on special keys can be hibernated. The source code was written sequentially as the project progressed; therefore performance can be increased through code optimization.

Before the application can be released, the effect of shadows has to be eradicated. There are multiple ways this can be done. One such method involves invariant images. Invariant images are obtained by finding images orthogonal to the direction of the shadow. Shadow edges are the difference between the edge maps of the invariant and non-invariant images. To obtain the shadow free image simply integrate the gradient field of the image using a poisson equation$^{25}$. It can also be done by duplicating the image, applying Gaussian blur to the duplicate, inverting the gray channel on the duplicate and adding the two images$^{26}$. However such methods need to be tested rigorously before being implemented.

To increase accessibility the application can also be ported to other platforms such as IOS, Windows phone etc. Despite being aimed at college going students, the target market can be far bigger. In developing countries where computers are scarce, this application can be used to provide cheap alternatives to PCs. Similarly the same technology can be used for digital signatures (the user signs on a paper, the app verifies and produces a digital signature) or drawing applications (the user draws on paper, it produces a digital output, or
helps the user draw by tracing the virtual drawing). The application could also be used as an accessory (paper piano, paper joystick) or augmented reality games.

Closing Remarks:

I found this project both challenging and rewarding. Even though due to time constraints, I was unable to polish the final application, I am proud of the outcome. By undertaking this project I have gained invaluable experience in image processing and mobile application development.

On December 17, 1903 the Wright brothers recorded their first flight\textsuperscript{27}. The flight lasted 12 seconds and covered just over 37 meters. Today aviation is at the forefront of the transportation industry.

On a similar note the current paper keyboard records just single keystrokes at an approximate accuracy of 83 %. Is it ready to replace the on screen keyboard yet? No, but it shows that it has the potential to and hopefully one day it will be.
References:


