World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:7, No:10, 2013

Augmented Reality on Android

Chunghan Li, Chang-Shyh Peng, Daisy F. Sang

***Abstract*—**Augmented Reality is an application which combines a live view of real-world environment and computer-generated images. This paper studies and demonstrates an efficient Augmented Reality development in the mobile Android environment with the native Java language and Android SDK. Major components include Barcode Reader, File Loader, Marker Detector, Transform Matrix Generator, and a cloud database.

***Keywords*—**Augmented Reality, Android.

I. Introduction

AUGMENTED Reality(AR) combines real and virtual images for presentation. Augmented Reality is interactive in real time and registered in 3D [2]. One of the most well-known applications is the 1st & Ten System [1]. In the 1st & Ten System the real-world elements are football field and players, and the virtual element is a yellow line which augments the image in real time. In the early stages of research in AR, developments were limited by slow processors and insufficient computer memories. However in the last few years, due to much improved processing power and broad availability of digital cameras, research in AR has been advancing on both the personal computer and mobile devices [12], [16].

Among all AR-capable mobile devices, the smart phone is the most representative instrument. Consumers' perspectives of smart phones have been as fast changing as the growth of the market. Phones are no longer only for voice communications. Instead, they have evolved into multi-tasking gadgets. Most smart phones are running high-end multi-processing mobile operating systems, which come with a suite of utilities such as camera, location positioning, internet browsing, and so on. These smart phones are packaged with general purpose processors that are as powerful as that in a conventional personal computer in most daily tasks. Such processing power enables application developers to unleash their creativity on an entire different platform. And, with the advantage of portability and ease of accessing cloud services, many new ideas can now be realized; such as Google Goggles [7] and Shazam [17].

Major modern mobile operating systems include Android, iOS, Symbian, and Windows Mobile. Android, built specifically for mobile devices, is the most popular and widely adopted operating system. The learning curve of Android is smooth, in large due to its openly available technical documentation. Android provides a custom built virtual machine that enables application testing and debugging [8]. Android apps can be efficiently sold via Android Market. Current mobile devices do come with the constraint of limited computational power and system resources. AR applications with 3D visualization are mostly resource-draining tasks. Low level languages, such as C/C++, are frequently adopted in hope to improve the efficiency. For instance, QCAR [14] was written in C. Such libraries are very much device dependent, and supports only selective mobile devices [15].

In consideration of application portability and the support of all mobile devices, this paper takes on the challenge of developing an AR application in only the native Java and Android's official SDK. Paper continues with discussion of application framework in Section II and resultant sample runs and screenshots in Section III, and concludes in Section IV.

II. Application Framework

Augmented Reality on Android builds an interactive and distributed application on an Android phone. The application uses an Android phone's built-in camera to capture the initial input in the format of a barcode, which corresponds to one of the predefined categories (e.g. animals). Based on the initial input, a cloud-based database is queried for a list of options (e.g. dog, cat, mouse, etc.). This list of options is presented to the user for a selection. Upon user's selection, the cloud-based database is queried again for details of the selected 3D model. Data of the selected 3D model is then loaded and converted in preparation for presentation. Lastly, the 3D model data goes through all necessary 3D transformation before being displayed in the Android phone's screen in Augmented Reality.

The choice of language is Java; the most popular object-oriented, system-independent, and portable language. With the wide availability of Java Virtual Machine, Android apps in Java can be readily used on any Android devices. Java Virtual Machine on the other hand can introduce execution latency. Better and more efficient algorithms are designed to minimize side effects such as execution latency. To optimize the portability, native Java is used without any 3rd party library in all areas including 3D graphics and computer vision. Further, this application also aims to ensure an acceptable user experience.

According to the outlined framework, major application functions include Barcode Reader, Model Loader, Marker Detector, and Transform Matrix. Barcode Reader reads the initial input in the form of a barcode. Model Loader has the heavy responsibility of loading the representation data of the

Chunghan Li was with the Master of Science in Computer Science Program, California Lutheran University, Thousand Oaks, CA 91360, USA.

Chang-Shyh Peng and Daisy F. Sang are with the Department of Computer Science, California Lutheran University, Thousand Oaks, CA 91360, USA, (e-mail: peng@callutheran.edu, fcsang@csupomona.edu).

user-selected model. Marker Detector works with other functions to calibrate the imaging device and prepare for the output. Transform Matrix handles the image presentation process.

A. Barcode Reader

With Android's built-in camera and object recognition functionality, initial input is recognized in the form of a barcode that is surrounded by a rectangle, which is further enclosed by a square (referred to as a marker throughout the rest of this paper). Fig. 1 depicts a sample input.

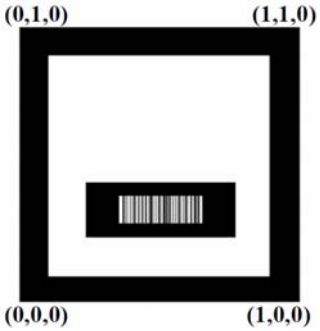

Fig. 1 Sample Initial Input

Fig. 2 Standard Code 128 Barcode

Code 128 [5], a widely used standard and a very high-density barcode symbology [9], is the selected encoding scheme. It can encode all 128 ASCII characters and use a check digit mechanism to minimize the chance of misreading. As shown in Fig. 2, a standard Code 128 barcode will have six sections: Quiet Zone, Start Character, Encoded Data, Check Character, Stop Character, and Quiet Zone.

Quiet Zones, at least 10 times wider than the narrowest element, are used to delimit the barcode in the front as well as the end. In this application, barcodes are used not only for input but also for orientating the marker. To improve the accuracy and efficiency, Quiet Zones are replaced by a surrounding rectangle. Each encoded character in the barcode is composed of three bars and three spaces, where the stop adds an additional extra bar of length 2. Both each bar or each space can be 1, 2, 3 or 4 units (e.g. pixels) wide, the sum of the widths of bars must be even, the sum of the widths of the spaces must be odd, and with total 11 units per character. For instance character "A" is encoded as 10100011000 where 1 is a bar and 0 is a space. The check digit is a modulus 103 checksum. It is calculated by summing the start code value to the products of each character's value multiplied by its position in the barcode string. The start character and first encoded value is in position 1. The sum of the start code value and the products is divided by 103. The remainder is the check digit's value, which is then converted into a character and appended to the end of the barcode.

To improve the correctness of barcode reading, the Barcode Reader employs multi-reading algorithm. A barcode is scanned one line at a time. For each scan, neighboring lines (couple of pixels above or below) are scanned as well. A check character is factored in to verify the correctness.

B. Model Loader

With the recognized barcode, the Model Loader queries the cloud database for a list of models. This list of models is presented in the smart phone/device display for user to choose from. Once the user makes the selection, the Model Loader again queries the cloud database for the definition file of the chosen model. Challenges in the design of Model Loader include the compatibilities of various modeling definitions, calculation efficiencies, and preparation of displaying a 3D model on a 2D screen.

3D models are denoted by combinations of polygons (or faces) and corresponding textures on each face. The selected 3D model is then, via UV Mapping [20], mapped to a 2D image. Each 3D model is represented in the OBJ format [13]. OBJ developed by Wavefront Technology (now formally known as Alias|Wavefront) [21], is a popular open geometry definition in ASCII file format. It can be generated or converted by most 3D graphic editor applications. On the other hand, the native language for Android graphics is OpenGL for Embedded Systems (OpenGL ES). OpenGL ES, a subset of OpenGL, is developed for embedded devices such as mobile phones, PDAs, and video game consoles [6]. There are some critical incompatibilities, between OBJ and OpenGL ES, that the Model Loader has to accommodate. In OBJ, it is acceptable to use three or more vertices to define a face. OpenGL ES allows only three. Triangulation is therefore required to OBJ and OpenGL ES. Secondly, OpenGL ES requires that each texture image be in the shape of a square with the length of an integral power of 2. Further, each vertex is defined once and can correspond to multiple faces In OBJ. In OpenGL ES however, vertices shared in adjacent faces need to be recognized separately for each face. Thus definition of the model in OBJ needs to be translated according to the OpenGL ES's specification.

Within the constraints of available processing power and on board memory in the Android devices, the efficiency of the application is equally critical. Improvement of efficiency starts with downsizing the input file. Android has native support of zip files in its Java library. The OBJ input files are text files, which typically can be compressed to one third of the original size. The efficiency of loading of 3D model data can be improved accordingly. As soon as the OBJ file is fetched, the Model Loader needs to parse the input. While there are built-in parsing utilities in Java's String class, its execution speed is

unbearably slow. Thus Model Loader's parsing is developed from ground up with the character type instead of String class and is executed upon each line of input.

Upon the parsing of the selected 3D model, the Model Loader performs UV Mapping to create a 2D image representation. UV Mapping maps a texture map to the surface of a 3D object. The texture map is a 2D map, and can be in various formats such as a patterned grid, a bitmap, a raster image, etc. UV Mapping assigns pixels in the texture map (UV-vertex) to vertices on the surface of 3D object. The rendering process can then properly paint the 3D object in a 2D display. The surface of a 3D object is typically modeled as an assembly of polygons or faces. In most 3D models, there are many vertices at the intersection of multiple faces. Each of such vertices corresponds to several UV-vertices, which in turn is unique to each corresponding face. Such one-to-many relationship unfortunately is not supported in the OpenGL ES data structure. Those vertices will have to be duplicated in order to meet the requirement of the one-to-one relationship in OpenGL ES. The duplication of vertices introduces a challenge to data storage. Since the number of vertices and corresponding UV-vertices is not known until the entire OBJ file is loaded, an array is not an option for data storage. Instead, Model Loader implements a vector of vectors. Each vertex is denoted as a node in the vector, and its corresponding UV-vertices are stored as a vector of the node. Once the loading of OBJ file is finished, information of vertices and their UV-vertices are successfully calculated and stored.

C. Marker Detector

Marker is used to calibrate the camera and to position the chosen model in 3D reality. As shown in Fig. 1, a marker is in the shape of a square in which each side is identified with a straight line of thickness. The space taken up by the four sides of the square is referred to as the negative region, in contract to the positive region that denotes the space inside the marker. The Marker Detector first converts the original image into a binary true/false image where a bright pixel is denoted true and a non-bright pixel is denoted false. The input image in Android is recognized in YCbCr color space, which directly contains brightness information [11]. By bit shifting AND/OR operation, the brightness can be calculated as follows:

```
for(j=2;j<height;j++){
   temp = j*width*downScRate;
   for(i=2;i<width;i++){
      y = (0xff&((int) _image[i*downScRate+temp]))-16;
         if (y > white_limit){
            Image[i+j*width] = true;
         }else{
            Image[i+j*width] = false;
         }
   }
}
```

The downScRate is instrumented to improve efficiency. For instance, if downScRate is 2, the algorithm will pick every other pixel in both dimensions. The size of the converted image will then be reduced by 75%.

From the converted binary image, adjacent bright pixels are grouped into individual regions. The structure of the region is:

```
class Region{
   public int ID;
   public long volume;
   public boolean merged;
   public int x, y;
   public int top, bottom, left, right;
   public boolean isMarker;
   public Vector<Spot> ContourSP;
   public Vector<Spot> OutContourSP;
   public Line[] Lines;
}
```

ID is the region's unique identification number. *Volume* records the size of the region. If the *volume* is too small, the region would be ignored as noise. *Merged* is set true when the region is adjacent to another and is thus combined into one. *X* and *y* denote the coordinate of a pixel in the region. This pair of coordinates will be used in the outlining process. *Top*, *bottom*, *left*, and *right* are the coordinates of the boundary of the region. *isMarker* is the flag that indicates whether the region can be a marker or not. *ContourSP* and *OutContourSP* are custom classes that contain the coordinates of pixels, inside or outside of a region, visited in the outlining process. *Lines* is another custom class that identifies the four sides of the marker.

The Marker Detector then outlines each region to identify the four sides of the marker. If a positive region enclosed by four straight lines is identified, the Marker Detector will recognize the negative region and its perimeter. Once the perimeter is found to be a quadrilateral, this negative region is marked as a marker (denoted by *isMarker* being true). For the precise equation of each side of the quadrilateral, detected markers are outlined pixel by pixel. The coordinates of these pixels are grouped and averaged for each side. With the help of the equations of these four sides, the four corners' coordinates can then be correctly calculated. These coordinates will later be used for the presentation of the chosen 3D model in the smart phone's display.

D. Transform Matrix Generator

The next step is for the Transform Matrix Generator to generate the transform matrices which translate between the 2D display and OpenGL ES 3D presentation in Android systems. First, a transformation is calculated to coordinate the 2D display (on which the detected marker was defined) and OpenGL ES 3D presentation in Android systems. Secondly, a normalization process is implemented to estimate the model size in a constant unit distance between the marker and camera. A system of corresponding equations is defined with

the use of matrix production. Equations are then solved again with Gaussian Elimination [10], [18], [19]. The pose matrix, resulting from these heavy calculations, enables OpenGL ES to present the model in correct rotation and orientation.

III. Sample Runs

The application was deployed and tested on a Google Nexus One with 3Mbps downstream bandwidth. Figs. 3 and 4 showcase the efficiencies of the Model Loader. In Fig. 3, the OBJ file size is 75Kb and texture data file size is 103Kb. There are 944 vertices and 1911 triangles definitions in the model. The total loading time is 1.6 second. In Fig. 4, the OBJ file size is 240Kb and texture data files (10 files in all) size is 213Kb. There are 3783 vertices and 6576 triangles definitions in the model. The total loading time is 4.7 second. Figs. 5 to 8 highlight the execution of the entire AR application. Fig. 5 shows the barcode scanning. The corresponding model list via the cloud server is depicted in Fig. 6. Upon the successful transmission of the model file, a 3D model can be displayed on the Android screen as in Figs. 7 and 8. A complete demo video and the source codes are available online at [3] and [4], respectively.

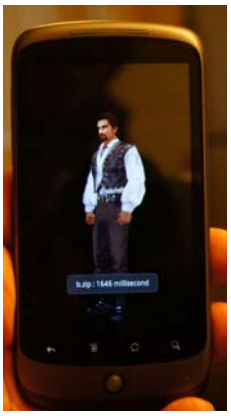

Fig. 3 Model Loader Timing 1

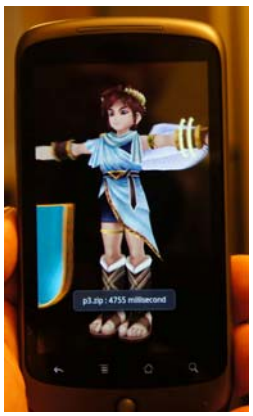

Fig. 4 Model Loader Timing 2

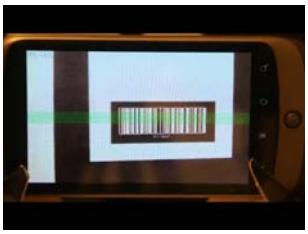

Fig. 5 Barcode Scanning

Fig. 6 Model List

Fig. 7 Initial 3D Model Display

Fig. 8 Rotated 3D Model Display

IV. Conclusion

This paper studied and developed an interactive and real-time application of Augmented Reality on an Android device with cloud database and only the native Java and Android's official SDK. As technology improves, AR on mobile devices will be more accessible to the general public. With maturing cloud services, mobile AR apps will invade users' daily life in the near future. Monopoly players can visualize each property's weather, landscape, and seasonal features via smart phone. Sports card traders can remotely examine their card of interest via a 4G touchpad. Ad feeders can present product details, in 3D AR, to their prospective buyers. The outlook of AR applications is even more interesting with the development of mobile wearable devices; just imagine a 3D holographic image overlaying the real world right in front our naked eyes!

APPENDIX

Following is the hierarchy of the source code. Details are presented in [4].

com.graduate.aa
- AndroidAR.java
- OBJItems.java
- Setting.java

com.graduate.barcode
- CodeConverter.java
- CodeReader.java

com.graduate.detector
- Detector.java
- DetectorFeedback.java
- DetectorView.java
- Label.java
- LabelMap.java
- Line.java
- Marker.java
- Region.java
- Spot.java

com.graduate.loader
- DefaultPlane.java
- Mtl.java
- OBJLoader.java
- OBJPart.java
- OBJRender.java
- Vertex.java
- VertexLink.java

com.graduate.TMgenerator
- Matrix.java
- TMgenerato.java

REFERENCES

[1] 1st & Ten system (graphic system), http://en.wikipedia.org/wiki/1st_%26_Ten_(graphics_system), last accessed 2012.
[2] Augmented Reality, http://en.wikipedia.org/wiki/Augmented_reality, last accessed, 2012
[3] Augmented Reality on Android demo, https://docs.google.com/file/d/0B5TOGbkpt0cVTXB5djNOQTBDaVU/edit?usp=sharing, last accessed 2013.
[4] Augmented Reality on Android source code, https://docs.google.com/file/d/0B5TOGbkpt0cVNEhmZEtySWl4bnM/edit?usp=sharing, last accessed 2013.
[5] Code 128, http://en.wikipedia.org/wiki/Code_128, last accessed, 2012
[6] S. Conder and L. Darcey, "Android wireless Application Development, 2nd Edition ", Addison-Wesley, 2010.
[7] Google Goggles, http://www.google.com/mobile/goggles/, last accessed 2012.
[8] T. Grønli, J. Hansen, and G. Ghinea, "Android vs Windows Mobile vs. Java ME: A Comparative Study of Mobile Development Environments", Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assitive Environments, 2010.
[9] T. Kan, C. Teng, and W. Chou, "Applying QR Code in Augmented Reality Applications", Proceedings of the 8th International Conference on Virtual Reality Continuum and Its Applications in Industry, 2009.
[10] H. Katon and M. Billinghurst, "Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing", Proceedings of the 2nd International Workshop on Augmented Reality, 1999.
[11] H. B. Kekre, S. D. Thepade, A. Athawale, and A. Parkar, "Using Assorted Color Spaces and Pixel Window Sizes for Colorization of Grayscale Image", Proceedings of the International Conference and Workshop on Emerging Trends in Technology, 2010.
[12] R. Meier, "Professional Android 2 Application Development", Wrox, 2010.
[13] OBJ File, http://www.martinreddy.net/gfx/3d/OBJ.spec, last accessed 2013.
[14] QDevNet, http://developer.qualcomm.com/dev/augmented-reality, last accessed 2013.
[15] QDevNet AR SDK, http://ar.qualcomm.at/qdevnet/sdk, last accessed 2013.
[16] D. Schmalatieg and D. Wagner, "Experiences with Handheld Augmented Reality", Proceeding of the 6th International Symposium on Mixed and Augmented Reality, 2007.
[17] Shazam, http://www.shazam.com/, last accessed 2012.
[18] Systems of Linear Equations: Solving by Gaussian Elimination, http://www.purplemath.com/modules/systlin1.htm, last accessed, 2012.
[19] L. N. Trefethen, "Three Mysteries of Gaussian Elimination", ACM SIGNUM Newsletter, Volume 20, Issue 4, 1985.
[20] UV Mapping, http://en.wikipedia.org/wiki/UV_mapping, last accessed 2013.
[21] Wavefront Technology (Alias|Wavefront), http://www.autodesk.com/, last accessed 2013.