Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 25 (2013) 436 – 442

Procedia Computer Science

2013 International Conference on Virtual and Augmented Reality in Education

Considerations on Designing a Geo-targeted AR Application

Jose Rodríguez-Rosa, Jorge Martín-Gutiérrez*

Vicerrectorado Tecnologías de la Información y las Comunicaciones, Universidad de La Laguna, Spain

Abstract

When designing an AR application for mobile devices there are some factors to take into account such as gyroscope drift and device orientation, compass noise and six axis or accelerometer and GPS sensors accuracy. This work focuses on Geo-targeted Augmented Reality applications. We expose the procedure followed to design an AR framework almost from scratch from the software design point of view. We have also detailed the technical difficulties encountered during the development of the application and how they were addressed. Finally we propose some guidelines and recommendations for future work and improvements which can also serve as a common pattern design for Geo-tagging AR applications.

Keywords: Geotargeting, AR, Sensors, Gyroscope, GPS, Tracking

1. Introduction

Despite there are already many AR customizable browser, sometimes it is desirable to create one for scratch. E.g. the AR browser is not customizable enough, it lacks some features you would like in your application, or one simply wants to learn how to design an AR application for learning purposes.

On the other side, starting a developing project from scratch might be at first a daunting task not always being worth the hassle, especially when the target architecture is so fragmented and diverse as it happens with mobile devices (Android and iOS). We first evaluated several AR and cross-platforms development tools, and finally settled on Unity 3D using Qualcomm Vuforia AR Toolkit. These tools are available for free and provide a fast development cycle for multiple target architecture will little or none customizations needed in the project.

* Corresponding author. Tel.: +34-22- 319035
E-mail address: jmargu@ull.edu.es

doi:10.1016/j.procs.2013.11.055

Nomenclature

POI – Point of Interest

Geo-target – A trackable AR target defined by its GPS (latitude, longitude) coordinates.

Further nomenclature continues down the page inside the text box

2. Project Goal

The main goal of the project is to provide a generic purpose geo-targeted AR application template, which will be used at first on a domestic project related to tourism information and points of interest. In this application example, users will interact with their surrounding environment scanning outdoor places for local Points of Interest which will appear on their mobile devices screen when available.

3. Application Design

Some features were either mandatory or high desirable, including among them:

- **Multiplatform and multiarchitecture –** The application should run on as many devices as possible. Unity provided a very robust cross-platform development framework, mainly focused on 3D applications[†]. Notice that the fact an application is compilable for a given architecture does not implies it is compatible with all available devices within such architecture. Hardware models present many variations even for a single platform (e.g. Android) and some of them might lack sensors (GPS, gyroscope, compass, etc.) which makes the application not usable in such cases.
- **Usable and intuitive** – The application must be intuitive and easy to use for the novel user. It should not require any preparation for its usage other than starting it, and might adapt to the user interaction. E.g. it should support automatic screen orientation (portrait, or landscape for both left and right handed users), etc.
- **Stable and agile** – Heavy applications might drain mobile devices batteries. The application should be robust and fast, keeping its rendering frame rate around 30 frames per second. This will improve user experience and encourage the user to keep using it.
- **Versatile and highly customizable** – The application might connect to internet to retrieve new information on runtime and change its behavior or some other aspects depending on some events or circumstances (e.g. current geographical coordinates, time of the day, etc...)

For this purpose, Unity3D and Qualcomm Vuforia AR Toolkit were used. They provided a rapid application development and testing cycle whilst maintaining multi-platform compatibility.

[†] Latest versions now support iOS, Android, Blackberry and Windows Phone platforms.

4. Technical Considerations

For an outdoor geo-targeted AR application, we will heavy relay in mobile GPS, gyroscope and any other available orientation sensors since our AR targets will mostly be GPS targets and not AR markers. So the first step is to get the device geo-coordinates and orientation in and accurate way.

Next step is to determine which AR information will be taken into account and displayed for the current user position and orientation. We want to discard as much information as possible for the current scene to maintain application performance.

Once this has been achieved, the next step is to draw the AR objects over the device camera input in concordance with the device orientation, maintaining an illusion of coherence. This also requires periodic readjustments and auto-calibration to correct eventual drift that might happen during application usage.

Finally the application must (optionally) respond input events generated by the user, such as touch events, device screen orientation, etc.

4.1. Gyroscope attitude

Accessing device gyroscope information in Unity3D is easy, since it provides a generic interface for any architecture. Unity3D uses quaternions for treating object orientation, and so the gyroscope attitude is given with another one.

However, we encountered some issues when using gyroscope input data since it is very device dependent: Sometimes gyroscope attitude readings are noisy, update slowly or require a quaternion correction if the device screen orientation is changed (e.g. from Portrait to Landscape)[1].

This was addressed using a quaternion spherical linear interpolation (*slerp)* between the current camera attitude and the desired one (which might be multiplied by a second quaternion to correct the screen orientation if needed) using a step factor of 0.2. This value was determined empirically and effectively makes the interpolation to act as a low-pass filter, stabilizing the gyroscope reading while keeping enough responsiveness for a good user experience. The sampling rate used for this sensor was 15Hz (higher rates shortens device battery).

In our tests, we found gyroscope on Apple iPhone smartphone to be very stable and accurate over time; on the other hand, Android it´s very device dependent, but having the low-pass filter implemented as stated above, effectively fixes this problem on most devices.

4.2. Compass and Orientation

For simplicity and performance reasons, we indicate the positive Z axis in the 3D scene to be oriented to the Geographic North Pole. Gyroscopes might experience some drift during application runtime, and depending on the device they might have the positive Z axis oriented to a random direction (usually the device orientation at the moment the application was launched). Figure 1 shows an example of a device pointing almost 30 degrees to the East. Notice the gyroscope attitude (denoted by circles around the camera axis) is not aligned with the XYZ axis which are aligned with the geographic North.

Magnetic sensors allow the device to implement a compass. Unlike gyroscopes they do not experience drift over time, but are more prone to noise and erratic readings. So to implement a gyro-compass with an acceptable accuracy we tested two common low-pass filter methods: moving average and exponential moving average.

For the moving average, we have the well-known formula (1).

$$S_n = \frac{\sum_{i=0}^{N-1} Y_{N-i}}{N} \tag{1}$$

Testing the application with N=15 showed rather stable results for the compass filtered reading (denoted here by S_n), but at the cost of slow responsiveness (partly because oldest values weight as much as more recent ones). So this formula has a linear complexity of N. Since this method is executed on each frame this might cause frame-rate drop and high battery consumption, mainly in older devices.

Another more efficient method is the exponential moving average which is denoted by the formula (2).

$$S_n = \alpha Y_{n-1} + (1-\alpha) S_{n-1} \tag{2}$$

This method requires fewer operations and gives much better results when filtering noise from compass sensor input data. This formula is computed in constant time. We empirically determined the value of parameter α=0.2 as an optimal setting for this method, and the result is more responsive than the previous one.

The S_n value is the horizontal rotation (around the Y axis) in degrees. This value is converted to a quaternion and interpolated (using *slerp* as done before with the gyroscope stabilization method) with the previous value using the number of seconds since the last reading as interpolation *step parameter* (since this method is executed several times per second, this is a fractional value between 0 and 1).

Now to correct the gyroscope horizontal rotation, we need to cancel its original rotation around the Y axis and rotate counter-clockwise the current compass data input. Results obtained with this method are very stable and accurate enough to bring a good experience to the user.

4.3. GPS and Geo-location

To place the objects in the 3D scene over the camera device view we choose a plane projection. This is accurate enough for projecting nearby objects (the application only renders objects within a predefined radius of 500 meters). The XYZ coordinate system has its origin (0, 0, 0) in the device when application starts.

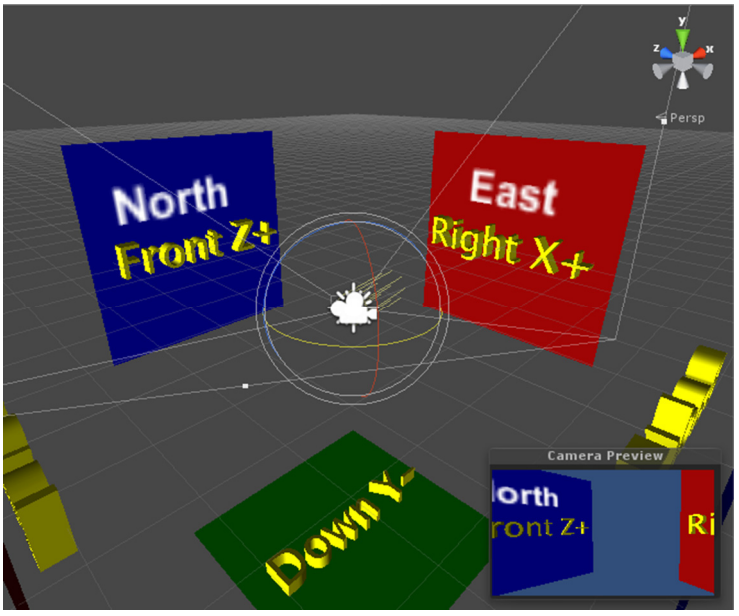

Fig. 1. Device (camera) orientation and compass

The GPS sensor already returns the latitude and longitude of the device (camera). The application obtains 3D objects metadata from an external source (e.g. from a database on a remote internet service, or preloaded and bundled with the application). Each object is usually a POI, which is a Geo-target trackable target, denoted by its latitude and longitude coordinates. It is possible to obtain the distance from two geographical coordinates using the *haversine* [2] formula (3). These formulae provide better accuracy on close distances around the Earth surface and have little overhead.

$$\begin{aligned} a &= \sin^2(\Delta\varphi/2) + \sin^2(\Delta\lambda/2)\cdot\cos(\varphi_1)\cdot\cos(\varphi_2) \\ b &= 2\cdot\arctan_2(\sqrt{a},\sqrt{1-a}) \\ \rho &= 6471\cdot b \end{aligned} \tag{3}$$

The formula (3) gives the distance (ρ) in kilometers between two given geographical coordinates (φ_1, λ_1), (φ_2, λ_2). This formula is applied for every object placed in the scene every 30 seconds (this time periods is adjustable), and those geo-markers not falling within the expected radius are deleted from memory to improve performance.

The next step is to get the POI bearing with respect to the device. This is represented in Figure 2: the angle between the positive Z axis (Geographic North) and the vector from the device to the target (in blue), also called bearing in navigation, gives the information needed to place the 3D Object (in Fig. 2, a white sphere) into the scene relative to the device position. The bearing between two coordinates (φ_1, λ_1), (φ_2, λ_2) is given by formula (4).

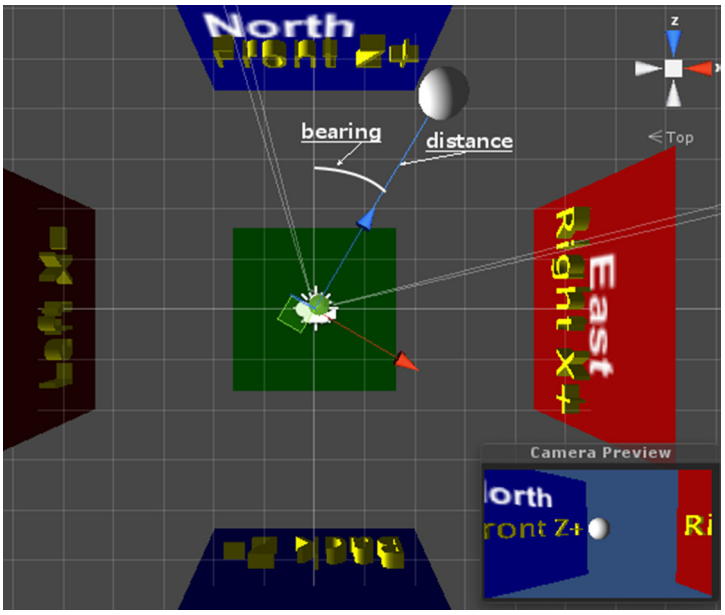

Fig. 2. Distance and bearing to a Geo-Marker

Having both the bearing and the distance, we have completely defined the vector from the device to the geo-target. We can now place the object in the 3D scene (as in Fig. 2) by setting its (x, z) coordinates to [ρ·cos(φ), ρ·sin(φ)]. At the moment we do not take the height (Y axis) into account, since many devices do not detect their current altitude yet.

$$\varphi = \arctan_2\left(\sin(\Delta\lambda)\cos(\varphi_2),\ \cos(\varphi_1)\sin(\varphi_2) - \sin(\varphi_1)\cos(\varphi_2)\cos(\Delta\lambda)\right) \tag{4}$$

5. Implementation and results

As stated before, we used Unity3D engine + Qualcomm Vuforia AR toolkit. Implementing the steps described in the previous section lead us to a very usable and stable AR application. FourSquare (FourSquare Labs, INC.) provides an open API to retrieve geo-target POIs near a given geographical coordinate, using HTTP+JSON.

We used this API as a data source for displaying POIs in the University Campus, but it can be used anywhere to get the FourSquare information all over the world. Users can tap with the finger on the targets to get extra information about a place, pictures, etc. See figures 3a and 3b.

Since the GPS input from the device is required to place the POI elements in the 3D scene, the application might suffer from error on weak GPS signal reception (e.g. indoor). We have determined an error of up-to 20 degrees for indoor usage (current devices uses even Wifi signals to get a geo-positioning estimation), which is automatically corrected once the device is brought outside.

Fig. 3 (a) Locating a place (red arrow show direction); (b) at a place (displayed at bottom)

6. Conclusion and Future Work

Creating the AR application has been an invaluable learning experience. This application also serves as a generic template for rapid AR application development, if the project requisites match this design pattern.

The application is very robust and open to many possibilities which range from AR outdoor gaming [3] to guided tourism, local POI information and orientation, outdoor learning, etc.

References

[1] Azuma R, Hoff B, Neely H, Sarfaty R. A Motion-Stabilized Outdoor Augmented Reality System. In: Proceedings of IEEE VR'99, p.252 –259

[2] Zdeb M. Driving Distances and Times Using SAS® and Google Maps, SAS Global Forum 2010, Paper 50-2010. URL: http://support.sas.com/resources/papers/proceedings10/050-2010.pdf

[3] Thomas B. Challenges of making outdoor augmented reality games playable. In Proceedings of the 2nd CREST Workshop on Advanced Computing and Communicating Techniques for Wearable Information Playing, 2010.