MASTER THESIS

TITLE: Location based augmented reality application on Unity 3D

DEGREE: Enginyeria de Telecomunicació (segon cicle)

AUTOR: Antoni Serra Font

DIRECTOR: Dolors Royo Vallés

DATE: November 5, 2013

Títol: Location based augmented reality application on Unity 3D

Autor: Antoni Serra Font

Director: Dolors Royo Vallés

Data: November 5, 2013

Resum

Aquest document presenta el desenvolupament I els passos seguits per crear una aplicació de realitat augmentada utilitzant el software Unity 3D amb un dispositiu mòbil. Es descriurà pas a pas el desenvolupament i se n'avaluarà el rendiment.

Title: Location based augmented reality application on Unity 3D

Author: Antoni Serra Font

Director: Dolors Royo Vallés

Date: October, 15th 2013

Overview

This document presents the development and steps taken in order to create an augmented reality application using the Unity 3D software using a mobile handheld device. The steps of the development will be explained and the performance evaluated.

INDEX

TABLE OF FIGURES

INTRODUCTION

Augmented reality is one of the newest and more exciting technologies to come out in later years; yet it seems it is everywhere by now: the applications and ramifications of this technology have not yet been fully realized. The main idea is simple: combine virtual elements into a real world display and allow some interaction between them. In these first years of development, many solutions offer a way to develop and use this elements within our computer, phone or tablet, and in essence anything with a camera display.

Since its origin, augmented reality has been closely tied to the videogame industry; and its most prolific development comes for the entertainment application. Unity 3D is a software application dedicated exclusively to the videogame world: everything it has is designed to allow a seamless development for anyone who wants to create a videogame.

The aim of this project is to develop a complete augmented reality project using only tools available within unity's capabilities. The reason for that choice is the videogame industry has been one of the pioneers on the creation and development of this now fast growing industry. Unity is more now than ever seen as one of the main user friendly software for game developers to fully integrate all videogame capabilities within one single IDE.

There are on the market other proprietary applications developing Augmented Reality over this platform (or at least, they include a plugin for being able to develop on it). These applications allow for development of games or other applications. The focus of the project is, however, to develop an augmented reality application from the ground: using no plugins and no external tools which depend on proprietary libraries, evaluate the different solutions the software allows, and check its performance on a real device.

1. Augmented Reality

1.1. Section Overview

This section will present augmented reality as a concept: goals, origin, evolution and how the project uses it in its particular case.

1.2. Description and brief history

Augmented reality (AR) is a live, direct or indirect, view of a physical, real-world environment whose elements are augmented (or supplemented) by computer-generated sensory input such as sound, video, graphics or GPS data.

The aim of AR is to modify our perception of the reality, changing it in order to add or subtract information with a singular objective. It could be used to enhance a particular element of it in order to render it more relevant, eliminate elements that distract from its purpose; or simply add virtual elements to complement the actual input.

The more common applications use real time augmentation in order for the user to feel that changes are applying as it happened in real world. In some cases, the augmented elements introduced in the scene can be manipulated by the user, becoming and interactive virtual world using computer vision and object recognition technologies.

Essentially, the main idea behind any augmented reality application is to render a view of the real world for the user, and then use sensor capabilities to be able to detect and interact with it.

The term Augmented Reality has been attributed to Thomas Caudell, a former Boeing researcher, who is believed to have coined the term in 1990. However AR technology has been around for a few decades. The first applications of AR appeared in the late 1960s and 1970s.

- In 1962 Morton Heilig, a cinematographer, created a motorcycle simulator called Sensorama with visuals, sound, vibration, and smell.

- In 1966 Ivan Sutherland developed the first head mounted display.

- In 1975 Myron Krueger creates Videoplace that allows users to interact with virtual objects for the first time

- Early mobile applications began to appear in 2008

Nowadays, the augmented reality market is in meteoric rise: marketing companies use them to promote their products; videogame companies develop new games using the technology every year.

1.3. Types and uses

AR can be divided on several types, according the hardware they use to display it (handhelds, glasses, PC ...) or the kind of “reality” they are trying to “augment”. Essentially from this project point of view there are two main classes to differentiate how the virtual elements are introduced onto the real world representation: The position of the virtual objects. The main difference being if the position is chosen with a reference image that has to be recognized, or if the position is obtained using GPS coordinates.

1.3.1 Pattern recognition

There is a specific element that the software is trained to recognize; this element acts as trigger for the virtual elements to be displayed. Those elements may be a shape, like the human body, or a symbol, like a QR, or a company logo. These virtual reality applications require image processing techniques in order to compare the pattern with the reference they store in their database. The math behind it can get quite complex, especially if the algorithm has to take into account logo orientation and spatial distance. This is the more common use of AR.

1.3.2 Location based

This approach aims to locate the virtual object position referencing it with a location triangulation using a GPS device. Usually this kind of applications are developed for GPS related applications such as your city map with your favorite restaurants on it, visualized onto the screen using the camera for a real world feel. This kind of approach doesn’t need an external reference prepared for it (such a precise logo or image) and it can be used in any situation worldwide.

2. Unity

2.1. Section Overview

In this section it is going to be introduced the basic concept of the software used for the development, as well as the reasons for its choosing. The main characteristics will also be outlined and discussed its advantages over other possible alternatives.

2.2. Introduction to Unity

Unity is a cross-platform game engine with built-in IDE developed by Unity Technologies. The software is mainly used for videogame development, and all of its structure is based around it to make the development as easy as possible. In figure 1 it can be seen a promotional capture provided by the developer of its editor.

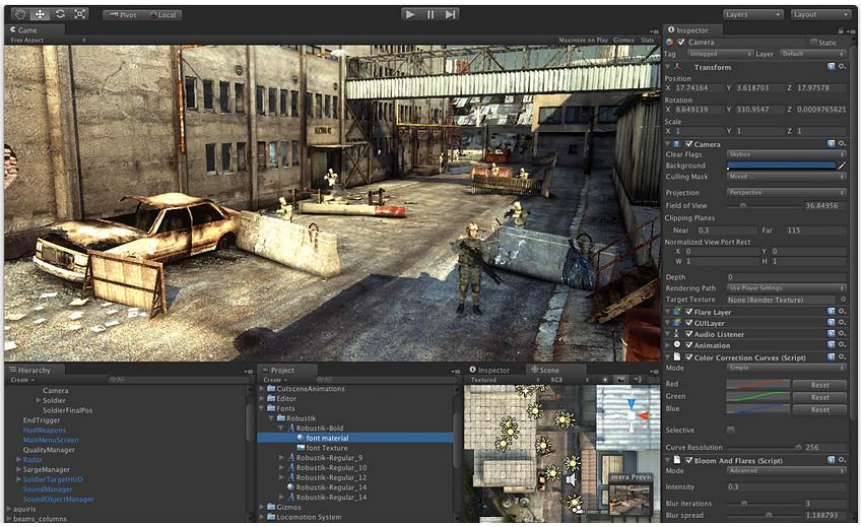


Figure 1: Unity framework

The software includes a Rendering engine supporting Direct3D, OpenGL and other proprietary software as well as processing and post-processing real time tools (such as light mapping).

The game engine is built around the use of scripts for the interaction between elements. Unity includes an open-source implementation of the .NET framework called MonoDevelop for IDE. It is capable of interpreting JavaScript, C# and Boo(which has a Python-inspired syntax).

The main feature of Unity is its cross-platform capabilities, which allow for the user to develop in a generic environment provided by the software, and then build a solution for each of the possible platforms (figure 2).

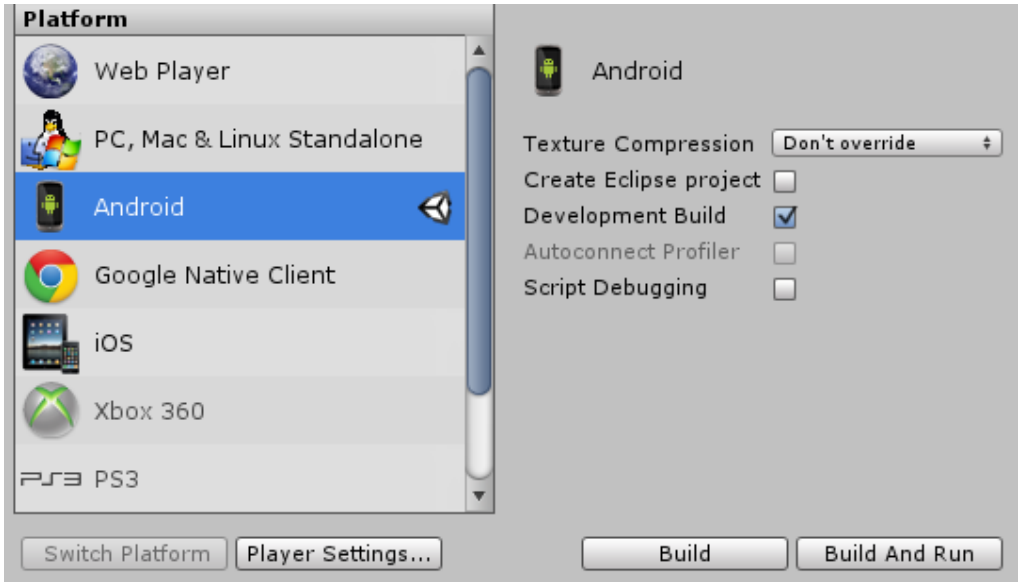


Figure 2: Different outputs for a single project build

Unity includes a built-in support for NVidia's PhysX physics engine with added support for real-time cloth simulation on arbitrary and skinned meshes, thick ray casts, and collision layers

2.3. Why Unity?

The main focus of this project is not to implement an AR application but to do it over Unity. Unity provides a full software package support that allows for any kind of game developing. Its characteristics as a full complete environment allows for simplicity and completeness inside a unified editor. Its role as a multiplatform development tool, allows for default interaction with the portable devices features, as well as its sensor output, and it includes an in-built system to access its data, simplifying the process. It is easy to see that Unity 3D holds in it the capabilities to develop an augmented reality application, although it not includes specifically design functions for this purpose yet.

2.4. Software *description*

Unity is a very unique software design for videogame creation. It is important in order to understand the implementation of the project, to describe Unity's components and how they interact with each other. It is important to remark that unity is not the focus of the project, just the tool used to develop, and because of that, it won't be an in-depth description.

2.4.1. Unity framework

The main editor of Unity shows a 3D display of the scene. The basic elements of any projects are called “Game Objects”. This is the definition of almost everything you create in the game, from lights to cameras to 3D models; they are simply the way Unity has to identify any element of the game that is not a script, a texture, a video, an audio etc… They are physical elements that have some behavior and some measure of interaction with each other. Each element differentiates using “components”; add-ons to the game objects that give some property to the game object. For example, a camera needs a “camera” component, a 3D element needs a collider to interact with physics, a light needs a “light” component to project illumination etc (see figure 3).

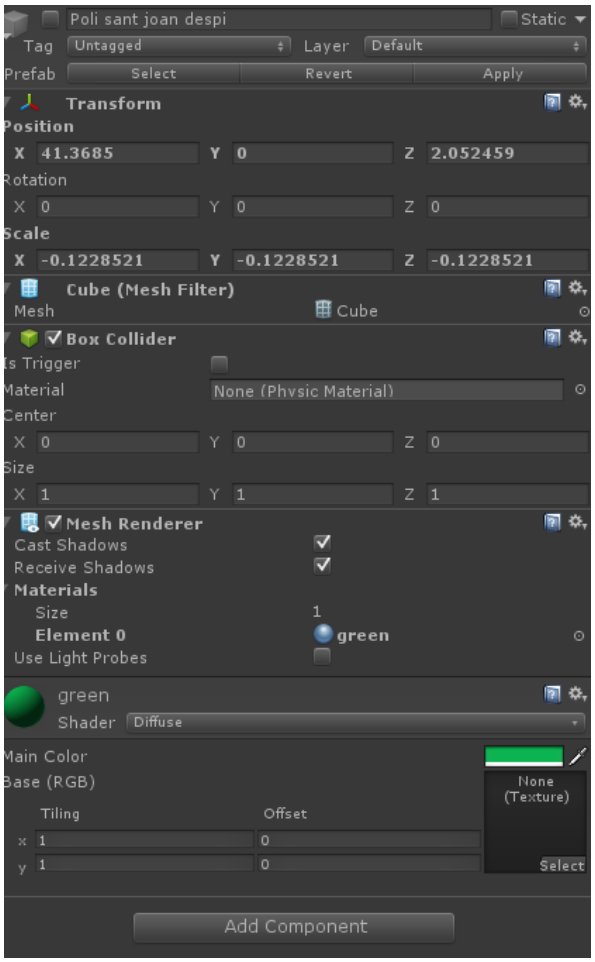


Figure 3: Example components in a Game Object

The interactions between these elements are defined inside a “script”. A script is a programming element which allow for custom unity functions to be executed to the object they are attached to (or any element referenced in it). All scripts include two basic functions: Start () and Update () (see figure 4). Start is a constructor, and defines properties and interactions called when the object is rendered in the scene. Update is called every frame time, and executes all its method calls periodically. In figure 4 it can be seen the scripts folder.

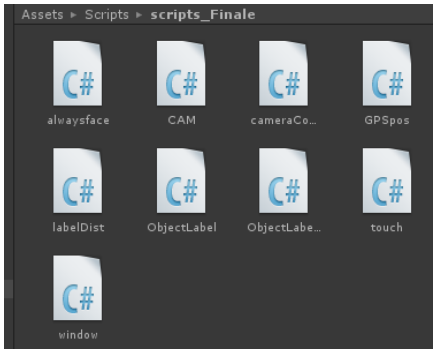


Figure 4: Script folder

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class NewBehaviourScript : MonoBehaviour {
5
6      // Use this for initialization
7      void Start () {
8
9      }
10
11     // Update is called once per frame
12     void Update () {
13
14     }
15 }
16
```

Figure 5: Sample new script basic features

2.5. Unity as an Augmented Reality development platform

Unity is a software platform which doesn't natively support augmented reality features. Nonetheless as it supports Mobile development, it has access to the devices' sensor capabilities, and thus, it has the capacity to do so.

2.5.1. Important Features

Unity contains a series of features that make it capable of Augmented Reality development:

- Libraries interacting with the device sensors and able to output its values in real time
- Application deployment on a real device (android or iOS)
- Monitoring of the software performance using a profiler, even over the device in real time
- Its structure using Game Objects and scripts makes it intuitive and easy to deploy

2.5.2. Notable Augmented Reality Libraries over Unity

Development of Augmented Reality in Unity is something different companies have tried already. Each provides either a development libraries to work with, or full functional applications using this interface. In this section a brief summary of the more relevant ones are listed and compared.

- **Qualcomm's Vuforia**: One of the more remarkable and professional examples of Unity's use of Augmented Reality. It uses Computer Vision technology to recognize and track planar images (Image Targets) and simple 3D objects, such as boxes, in real-time. It takes into account 3D objects perspective and rotation with respect the real world. This library, however doesn't support GPS positioning, and although it was considered as a setup for the project, it was discarded.

- **String:** String is an alternate to Qualcomm's solution; it prides itself to be the fastest AR solution for iOS. However, it doesn't have a free license, doesn't have an Android solution, and doesn't work with GPS specifically.

- **Metaio:** Metaio is a German company founded in 2003. Works well with its unity integration; it even has a free version with a tutorial and examples of different augmented reality features, and a simple SDK module that can be adapted to any project. It also features webinars for new users. The code itself is limited to their plugin, and it cannot be accessed, thus its performance, although it is top notch, cannot be evaluated. It was impossible to integrate it from the basic point of view our project has.

3. Project Description

3.1. Section Overview

In this section we will introduce the focus of the project, the different parts and the approach followed in order to create the application. It will also include a description of the device used and its components.

What the project aims to is to create an augmented reality environment which places real world elements into a virtual world representation of it. In the final solution, the device will be able to focus onto a real world element and display its positioning using a virtual 3D shape.

For this particular project, the EETAC buildings will do as placeholders for the virtual elements. The objects would be able to be touched and an information window will be displayed about them.

3.2. Parts and implementation

The idea of the project is to test a custom made Augmented Reality project over unity using only the tools it provides.

- **One camera focusing the scene**. This camera has to act as the eyes of the user, so it should always be showing what the user sees in the real world. To achieve that, the project will use the camera feed from the device.

- **One mean to project the video feed.** The video from the camera is projected into a canvas or texture able to reproduce it. Then the camera will be pointing at it, giving the illusion of real world feed.

- **A way to rotate the camera.** In order for the real world to interact with the virtual world, we need a tie-in that allows for Unity to recognize when the device turns and apply it to the Unity camera.

- **Real world positioning.** We need a system able to read the world position of the device and the real world elements and accurately position them in reference to each other. It is also important to keep distances and size of the elements to scale.

All the elements need to be able to interact with each other without consuming too much of the CPU performance while doing it. In order to test each particular feature, separate scenes will be tested for each element, and then put the whole together.

3.3. The device

The device used for the project is critical in more ways than one. First in sets the maximum complexity the program can achieve, since its performance might be limited by its CPU capabilities. Second, the sensors presents and its accuracy bias the project with its availability and precision.

HTC DESIRE

In the following table, it is presented the relevant specifications regarding the chosen device for the project: The HTC desire.

OS: Google Android 2.1
CPU: Qualcomm Snapdragon S1 QSD8250 @ 100MHz
RAM: 576 MiB (accessible: 412 MiB) GPU: AMD Z430
Multi-touch screen & TouchPad/TrackPad
5 Megapixels Camera
Sensors: Accelerometer, GPS, digital compass and proximity sensor.

Figure 6: Description table of the HTC Desire

It is important to remark that the Desire is an old model for today's standard, both in capacity and performance. It's critical the fact that it not possess a gyroscope, which is a fairly standard sensor in most of the market devices.

Another limitation regarding the device is the camera. It is quite old and provides some blurriness. Also the device seems to dedicate a lot of resources for the camera to run smoothly.

4. CAMERA FEED

4.1. Section Overview

The background of our scene has to be able to show the real world, so the objects superpose in a real scenario. In order to do so, we have to be able to access the camera from the device. Our HTC Desire has a built in camera as do all the smartphones, and Unity provides the means to access it using custom functions. The problem won't be the access of the camera per se, but the representation with unity's main camera in order to visualize it correctly.

Device camera: The HTC desire comes with a 5 MP camera 2592 x 1944 pixels and is capable of video imaging at 480p at 15fps.

4.2. Accessing the camera feed

Unity provides a series of function inside the class "WebCamTexture" that allows us to access the camera feed. It accesses video feed whether is coming from a webcam or from a smartphone or tablet device. The class outputs a texture which you can apply to any surface, that is, to any game object with a material that can render it.

4.3. First solution: The plane

The first idea that was implemented was a version of the camera that proves rather intuitive: The Main camera of the scene would be facing a plane, and we would render the camera image.

So for the first test it was set up a model with a plane facing the camera in a way that completely covers the vision. (see figure 7, notice that although the camera preview doesn't show the plane covering the full camera field of view, it does so with the app running).

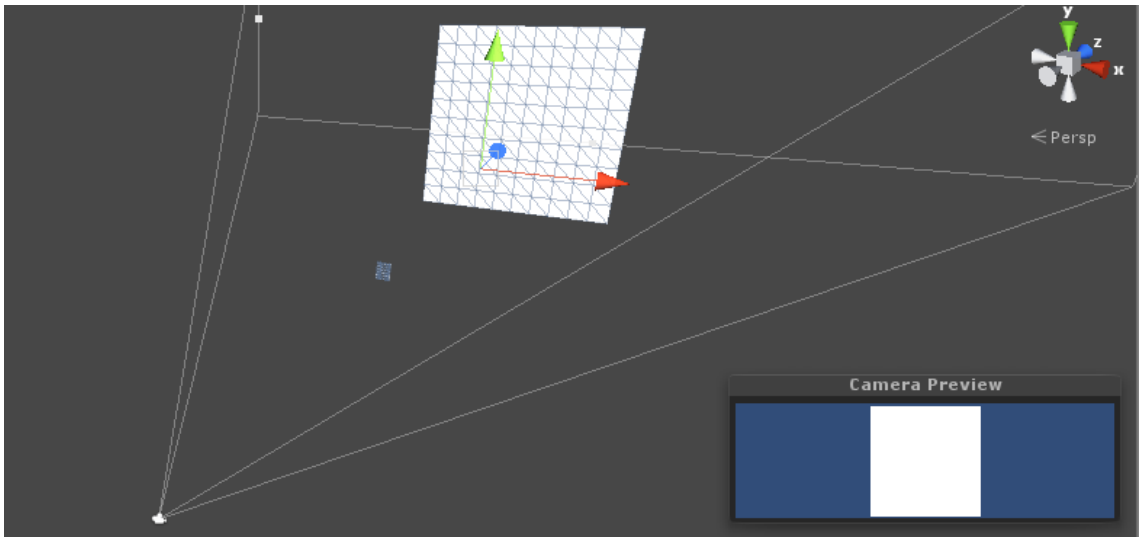


Figure 7: Model on Unity of the plane test

The main advantage of this model is that the camera deepness is very intuitive; the space between the camera and the plane is the rendering location and everything behind won't be displayed; also it is more intuitive since it acts as a cinema background: it's a screen where the video is projected.

This model had some problems nonetheless: first, it is difficult to adjust the plane so it covers the full image without distortion; this is due to the fact that many factors have to be taken into account: the device orientation, the plane width the depth or distance from the camera to the plane, the overture of the field of view… In the end by trial and error we achieved 90% coverage without cutting the edges or full view with some peripheral loss.

Video quality was not the best that can be; it tends to slow down in fps due to the processing of the application through unity. Overall it has a good enough quality for the purposes of the project.

The real problems, however, come when implemented with the other parts of the project. The camera processing add a huge load to the processor; causing the software to slow the whole fps from the application, not just the camera rendering. This leads to a jumping and erratic camera movement when implementing the rotation with it.

In figure 8 we can see the CPU usage of the device when executing the program. It can be seen that almost 20% of the CPU is not dedicated to the camera, since it is occupied rendering the plane (as is shown in the rendering section)

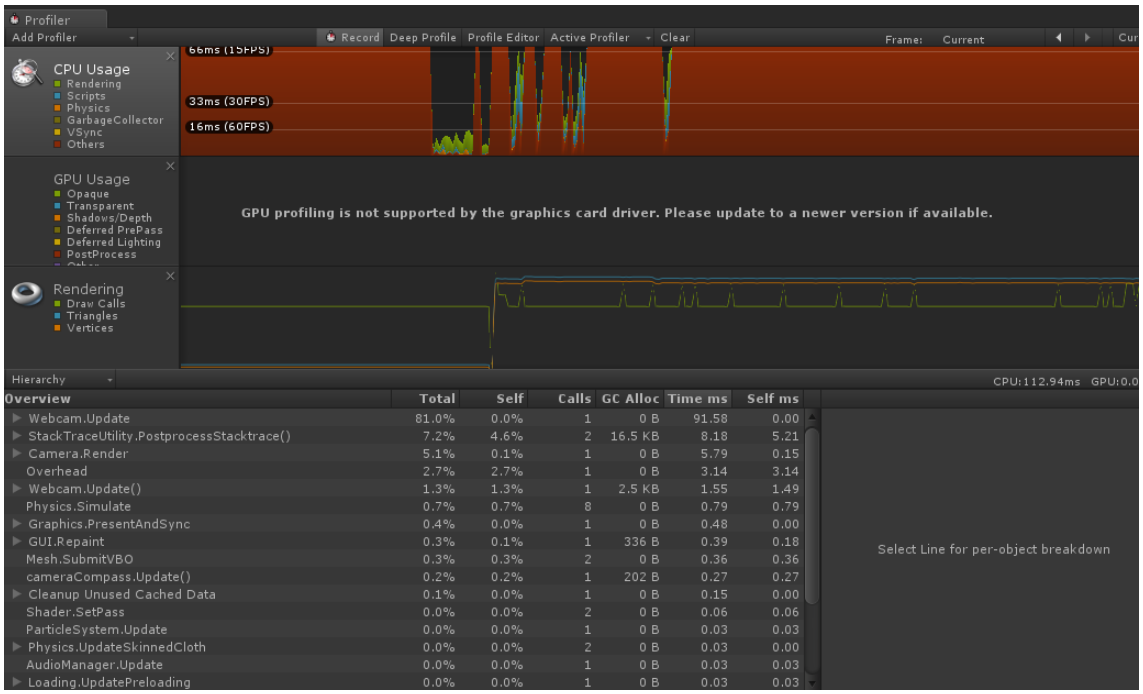


Figure 8: Profiler showing the performance of the test of the plane solution

4.3.1 Improving the plane solution: Simpler plane

With the last results, it was clear that the plane solution provide some issues in the performance area. As remarked, this idea's main problem was the extra load in CPU usage produced by the plane rendering. Trying to give it a second look, instead of using the default plane provided we used a custom function to create one; this approach has the advantage that we can define the plane with as many simple geometry as we want. If we recall last section, one of the problems in rendering the plane is that it was composed of multiple simple geometry shapes (triangles) that need to be rendered each time. For this approach we defined the plane as a simple one shape geometry, almost a texture with some background (see figure 10). In figure 9 it can be seen the setup which is identical to the last test.

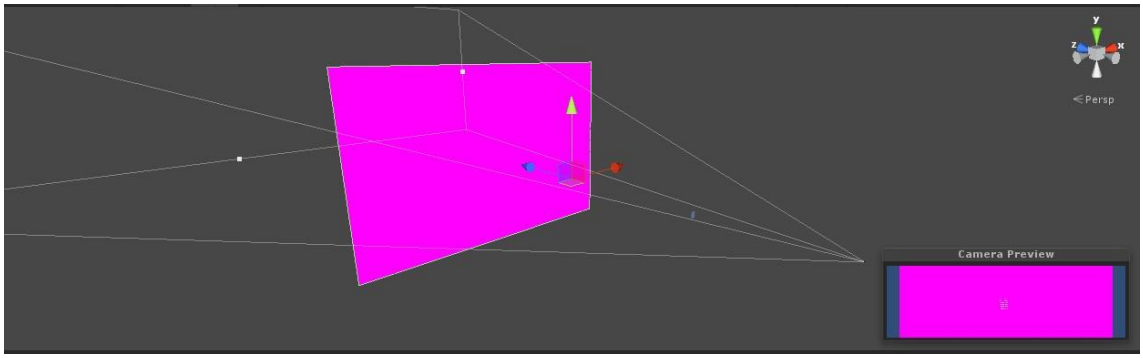


Figure 9: Unity modeling of the simpler plane solution

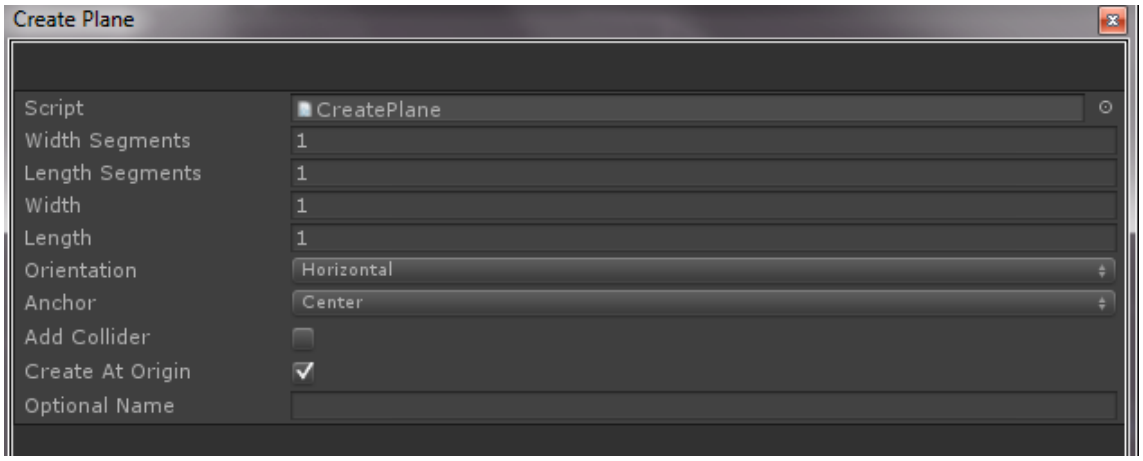


Figure 10: Plane creator window

When executing the profiler, it can be seen that the CPU usage dedicated to the scripts is higher, which means less percentage is dedicated to rendering the plane (see figure 11). Although it is an improvement, it is still far from the better solution, since the other problems associated with the plane solution are still there.

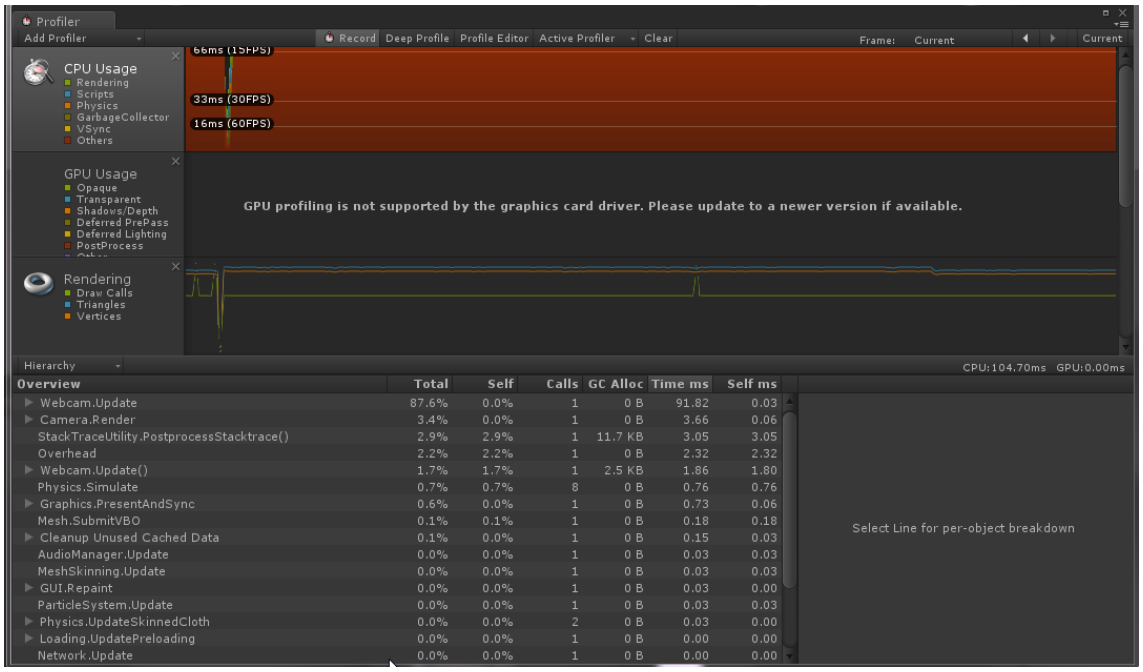


Figure 11: Profiler showing the performance on the simpler plane solution test

All and all both have the problematic that the plane is not a camera, and suffer from non-intuitive behavior when coupled with the compass movement and the device orientation (such as turning up when rotating right, changing the camera to upside down when the device is tilted etc), it needs additional script references to the compass or the plane position in order to behave as expected.

4.4. Second solution: Camera Texture

The first was the most obvious solution and it allowed for an easy implementation; however it had its flaws and in the end introduced too many issues for the final project.

The major problems with the plane solution were the plane rendering and the camera visualization; so this approach aims to solve those problems.

The idea is to create a game object call “GUI Texture” which is not a physical model, and thus, doesn’t have the rendering problem associated with the plane. GUI Textures are displayed as flat images in 2D. They are made especially for user interface elements, buttons, or decorations; but in this case it will serve as a background for the video feed (see figure 12).

The concept is essentially the same; create a flat surface where to insert the webcam texture. In this case though, it is not important where to put the object relative to the camera since what it is done is create a texture and then fix its size to the screen size.

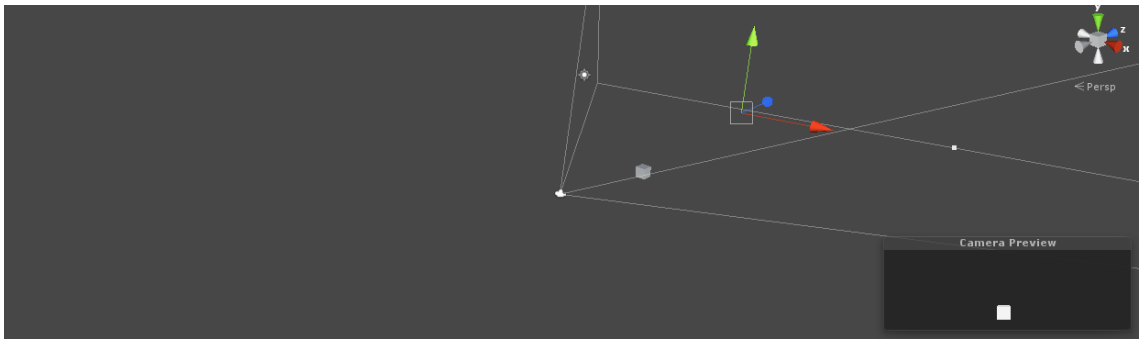


Figure 12: Unity design of the GUI texture solution test

This solution allows for a less rendering usage of the CPU; it is still a surface but it is not a physic surface, so it doesn’t require triangles to be rendered in the scene to create the plane; it also doesn’t require for the plane to follow the camera position when it rotates, and thus it generates less CPU load.

The main problem with this solution is that the texture position and orientation are fixed by script; rotating the device changes the relative position of the texture; the other main problem is that, although we are not creating a plane, it still acts as a plane. At all effects it is a wall reproducing a video. Also as it is now, there is no way to represent the 3D scene objects since there is no real space (physical) between the camera layer and the objects.

In figure 13 it is shown the build performance with and without the camera on. As can be seen the camera feed is the responsible of a high percentage of the

CPU usage, which shows a gain with respect to the last test where it also had to render the plane (also shown in the rendering section)

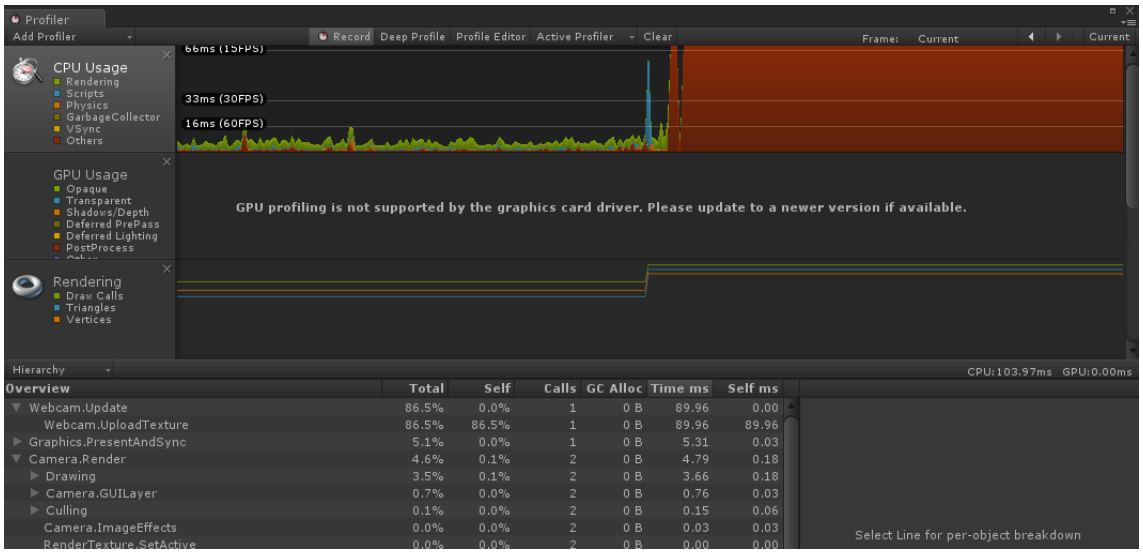


Figure 13: Profiler showing the performance on the GUI texture solution test

4.5. Final Solution: Second camera

So a third model was then implemented. We require a second element to display the image without rendering a real plane, and without interfering in the 3D elements that need to be displayed in order to create the augmented reality effect.

The third and final model is a variation of the texture model. For this solution we create a second camera which will be responsible of drawing the background image that will hold the video feed. Then afterwards, the main camera will draw the 3D scene without clearing the background so the image shows through.

The main difference with the previous solution is the creation of “layers”. Layers are a feature introduced by Unity, and are used to render only part of the scene. Are mainly used by cameras and lights to show or illuminate a limited number of elements. Those layers will be rendered in a way that doesn’t interfere with each other. This way, the main camera will render the objects in 3D and the second camera will render only the background (see figure 14).

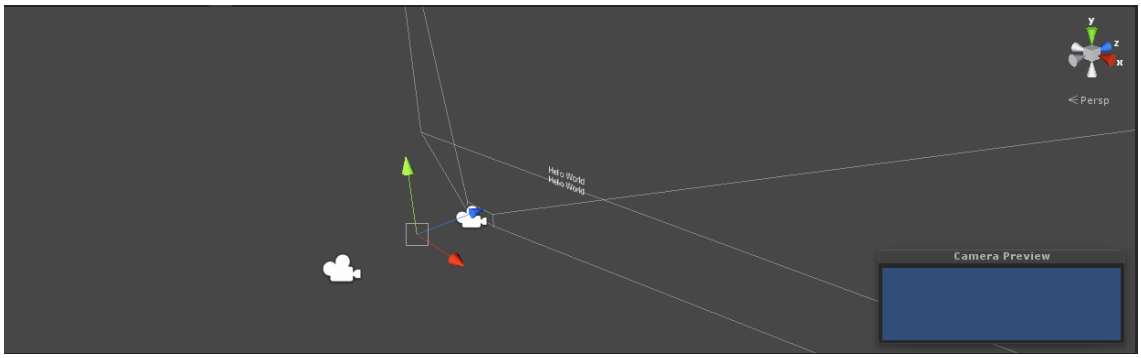


Figure 14: Second camera solution setup on Unity

For this solution, it is required the presence of three elements:

- Main Camera
- Secondary Camera
- GUI Texture
- A second layer

For that purpose we create a layer called “background” and assign it to the texture that will hold the camera feed, and also to the second camera. On the game object, it is possible to select which layer it is displayed on, using the culling mask option. Finally, on the main camera, it is chosen to display all the layers except for the background one.

To bind all the elements, a script will be attached to the secondary camera. This script will relate the texture that will hold the camera feed and the secondary camera that will be displaying it. The way it works, the background layer will only be rendering the camera and its texture; and the main camera will be displaying the rest (all except those).

This allows for the models to be displayed before, and then the camera feed is shown as a background. This is independent of the camera movement or rotation so it doesn’t affect the other parts of the project.

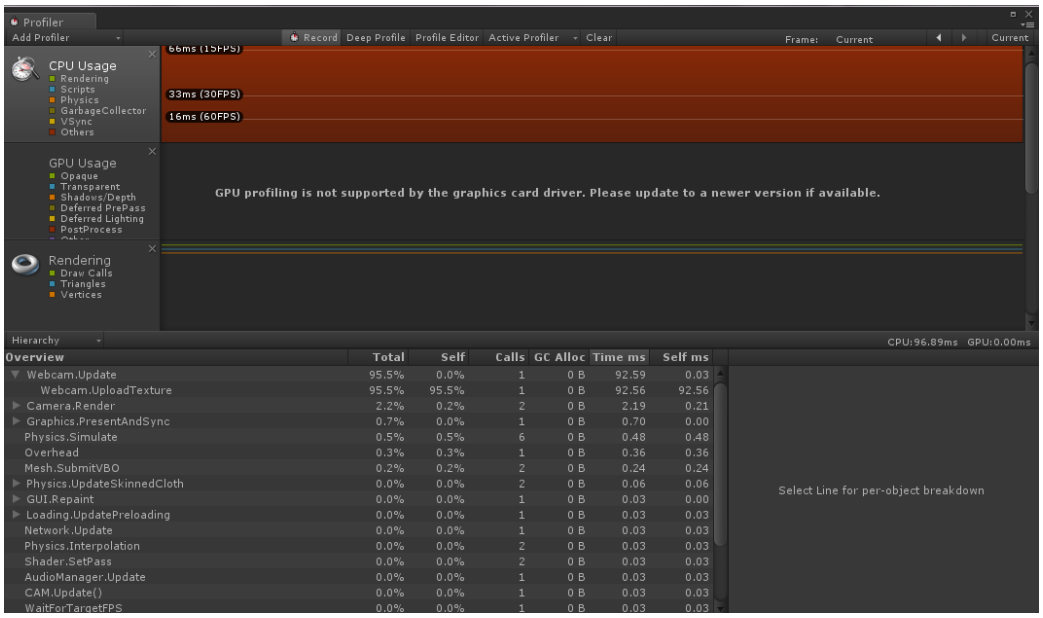


Figure 15: Profiler showing the performance on the second camera solution test

Figure 15 shows that the percentage of CPU dedicated to the scripts is much larger this time, meaning they are less influenced by other elements in the scene.

5. GPS

The GPS is the core of the project. The whole objective is to be able to see around you and detect those important elements in the surrounding area where you find yourself, anywhere in the world.

5.1. What is a GPS?

The Global Positioning System (GPS) is a space-based global navigation satellite system (GNSS) that provide reliable location and time information in all weather and at all times and anywhere on or near the Earth when and where there is an unobstructed line of sight to four or more GPS satellites.
By capturing the signals from three or more satellites, GPS receivers are able to triangulate data and pinpoint your location.

With the addition of computing power, and data stored in memory such as road maps, points of interest, topographic information, and much more, GPS receivers are able to convert location, speed, and time information into a useful display format.

5.2. What does unity offer?

Location services for unity are summed up in the input class. Location.lastdata returns a LocationInfo variable that holds the last data of the GPS reading. This value is updated in every frame update. The readings it can provide include latitude, longitude, altitude and its accuracies.

5.3. Testing location services

For the first test it was intended to display the readings for the longitude and latitude of the device and compare its accuracy. A sample program was deployed in order to get the data at every update of the program. The results prove accurate enough for our purpose (see figure 16).

Figure 16: Picture of the phone showing the basic location services

Getting location values good enough for the project was the easy part; the complex task appear when trying to translate those values into unity position values.

5.4. Representing location in unity: Cartesian translation

The first idea, and most intuitive was to try to design a function which transforms latitude and longitude into x,y,z values into a plane; but the question was: referenced to what? Since the (0,0,0) position of unity doesn't have any real meaning in the real world, first had to be assigned a reference point.

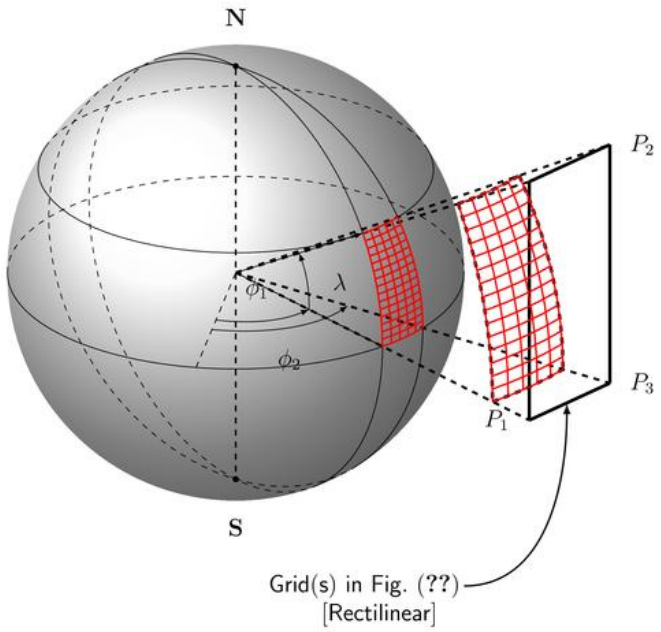


Figure 17: Cartesian to polar representation of a curved plane

As a first test, it was decided that current position will be the reference value for the initial position, and then all the real point objects that would be displayed, would do that in reference to that point.

For the latitude, is easy to convert into a north-south distance; the earth is a 360 dregrees full circle and that distance is 40008000 meters, assuming non spherical errors negligible, it can be compute fairly easily.

Converting the longitude it's a little bit more complex, since it's dependent on the latitude value (after all, it is a sphere). We know the circumference of the circle in the equator (at latitude 0) is 40075160 so the circumference at a given latitude will be proportional to the cosine so the formula can be extracted (all the formulas will be at the annex X to ease the reading complexity).

So basically once the two points had they values in meters, the position of the main camera would be set to (0,0,0) and the position of the relative points of interest in the map would be calculated as the difference between the real world position of the point, and the real world position of the GPS reading, obtaining a relative distance to the device position (which will be updated every frame).

Overall this solution can work in theory, but the output doesn't seem very intuitive as a "meters" calculation; and since the result is easier to read referenced to the actual position of the device, a second solution was implemented.

5.5. The plain LLN solution

It was decided that the most intuitive decision is the simple one. We assumed the unity world to be an infinite plane, and the earth curvature negligible (For the purpose of our tests and the relative distance between the object, it is acceptable) and we assign the latitude and longitude values as a unity-world position units.

Overall this solution is accurate enough since we consider earth a 2D plane and the latitude and longitude as a y and x. The relative distance between the points doesn't need to be calculated; so the position of the camera in the unity world is updated at frame time; since the project is not designed to be used on the move due to the changes in calibration of the compass; the change of position is barely noticeable.

The problem with this solution is the scale: since we are not referring to it as meters, the relative distance between two points might be of about 10^{-4} units in latitude reference, and thus, the accuracy needed to represent it is not attainable for the software's precision.

It was decided to apply an x10.000 scale to the position of the device and the reference points; and then adjust the size of the 3D objects accordingly. This value has to be so large since the changes in the scale are very small at lower

values; thus for two points too much close, the distance between them is barely noticeable (especially since the software tends to round up the decimal values, and it produced sometimes the overlapping of two different points separated 50 meters.

5.5.1. GPS marks

The Objects used to represent the real world highlighted positions, were plain cubes. The reason for it is that the cube is simple enough in rendering computational cost to not add too much to the already loaded CPU of the device (as does for example, a plane as seen in the camera section); another reason is that the cube is symmetrical from every approach and so it doesn't depend on where the device is located with respect to the cube. Finally, the cube is large enough and compact to be easily selectable with a touch command.

The position consists of three components(see figure 18): The Mark label, displaying the name of the object, the distance label, displaying the straight line physical distance between the device and the object; and the cube itself. The three elements are included into a main game object which holds the main scripts of the group.

Figure 18: Representation of a Cube marker in Unity editor and its setup in the inspector

As design extras, the cube displays the UPC logo to mark it as an UPC point of interest. Also, the cube itself contains a script attached that makes it rotate slightly to make it visible and relevant no matter what real world image is displayed by the camera; it also contributes to the fact that it is the same view no matter where it is observed from.

5.5.2. The distance label

As it was explained, the cube size is not at scale due to the fact that the real world exact scale is only approximated; since the cube sizes doesn't represent its real size and can mislead as to where the exact location of the point is (also it isn't affected by real world opaque objects such as walls), a distance calculator was added in order for the viewer to always have an idea of how far the object really is.

To calculate the distance between the device and the points, it was used the Harvesine formula (included in the annex). This formula uses de mathematical law of Harvesine to point the linear distance between to spherical points given its longitude and latitude. It is important to notice that although the result is not really a straight line but a spherical distance, the result is as valid since the distances are small enough to consider the difference negligible.

The distance label includes a script to make it always look at the camera, making sure it is always readable no matter where we are facing the object from

6. CAMERA ORIENTATION

In order for the camera to point to where the objects are, first it needs a reference of where it exists in the real world. This task is performed by the device sensors. To acknowledge its position relative to the earth in terms of rotation and inclination, it uses 3 elements:

The gyroscope:
The gyroscope is a device that measures or maintains orientation. It is formed by a body with rotation symmetry which rotates around the axis of this simmetry. In mobile devices though, they are called MEMS gyroscopes; a vibrating mass is placed in the center of the chip. The mass will be vibrated whenever an electrical signal goes through it. Moving the phone will cause the changes of electrical signals that are picked by the sensors. The sensors will send instructions to be interpreted by the software to provide the necessary feedback to the user. This sensor is usually used alongside the accelerometer to an accurate measure of the phone orientation, but our device doesn't have one integrated.

The compass (magnetometer):
The compass was originally a navigation instrument used to show directions in a frame of reference that is stationary relative to the surface of the earth. This frame defines de four cardinal directions.
Compasses work by detecting the magnetic field generated by the earth core. Early compasses were created by a magnetized pointer, free to align itself with the earth magnetic field. In general, a compass is any magnetically sensitive device capable of indicating the direction of the magnetic north of a planet's magnetosphere.
We refer to the instrument used to detect north as "compass", but it is more accurate to define it as "magnetometer". This distinction is important, since the compasses are usually programs built using magnetometers in modern devices.

A magnetometer is an instrument designed to measure the strength of a magnetic field (scalar devices) or in some cases it can be used to point out the direction (vector devices). Their most common usage is to calculate the strength of earth's magnetic field, but they have also other uses such as detecting magnetic anomalies and submarine detection in military applications.

The accelerometer:

An accelerometer is an electromechanical device that will measure acceleration forces. These forces may be static, like the constant force of gravity pulling at your feet, or they could be dynamic - caused by moving or vibrating the accelerometer.

By measuring the amount of static acceleration due to gravity, you can find out the angle the device is tilted at with respect to the earth. By sensing the amount of dynamic acceleration, you can analyze the way the device is moving

6.1. Sensor limitations

The sensor implemented in the device has a series of limitations due to the inherent design of the electronics, that it is important to remark.

Ferromagnetic materials and other sources of magnetic fields tend to make the readings fluctuate, so it may happen that the value is not accurate or it changes over time; there is nothing that can be done in order to compensate this, since the variations are temporal.

Also, the sensor needs to be calibrated at every location (by executing the infinite sign with the device) sincein each position tends to create a unique offset that changes the measurement. That pretty much excludes measurements on the move, although it is necessary for our project and the changes are not that critical for our purpose, it is still remarkable.

6.2. Earth's magnetic field

The magnetic field of earth extends from the Earth's inner core to where it meets the solar wind. Its magnitude at the Earth's surface ranges from 25 to 65 μT.

At any location, the Earth's magnetic field can be represented by a three-dimensional vector (see figure 19).

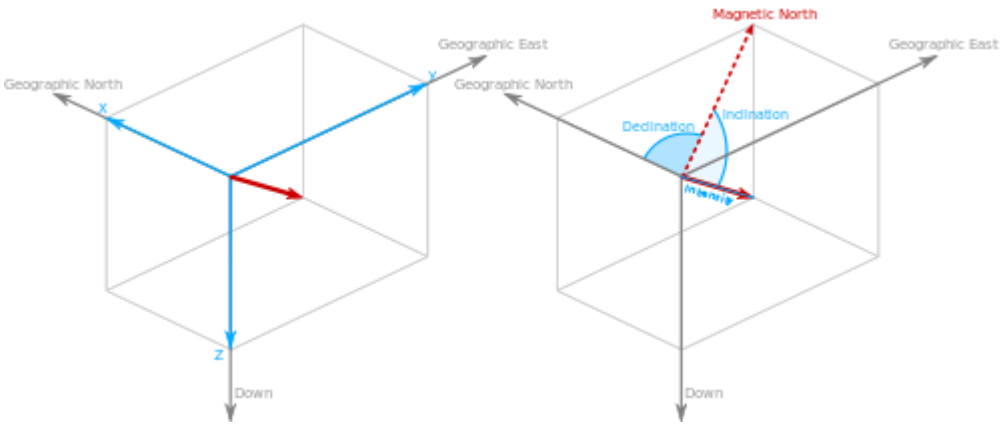


Figure 19: Schematic of the difference between the magnetic and geographic north (Wikipedia)

It has two main components: the horizontal element that points to toward the magnetic North (this is what we use for compass) and the magnetic inclination that has variable degree but points mostly down on the Northern Hemisphere.

This provides with an important differentiation: There's a deviation between geographic north and magnetic north. It chances depending on where you are located in the world. Magnetic north is always moving, and we call this margin of error declination. Declination is an angle that measures the difference between true north and magnetic north (figure 20).

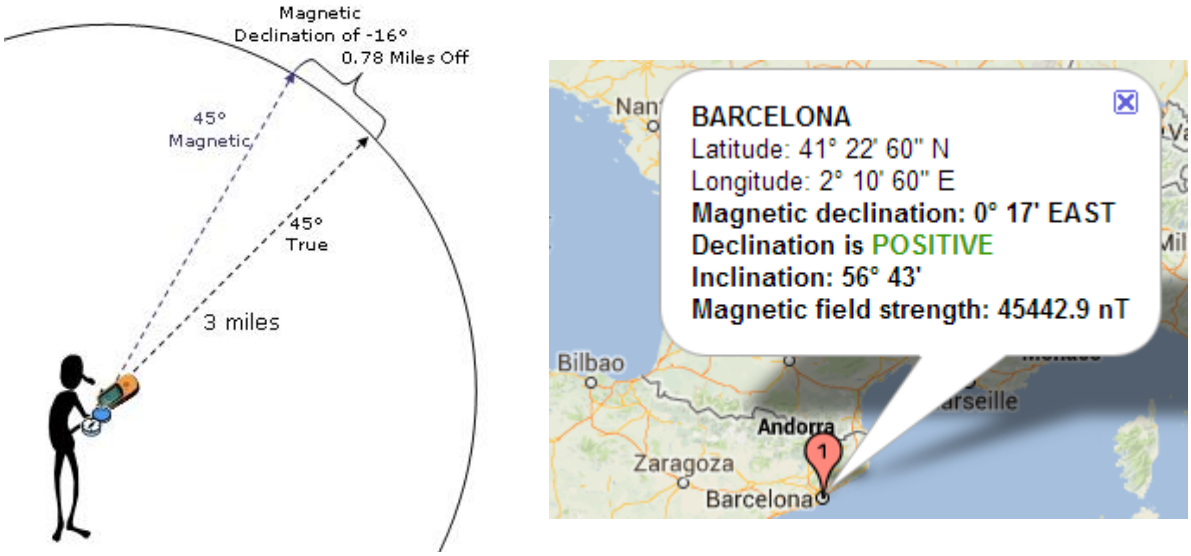


Figure 20: Magnetic declination representation and real world example

We call true north the direction towards the North Pole, and is the direction needed in order to be able to get oriented on a map. So our magnetometer device points towards the magnetic north and a correction is needed in order to achieve the correct direction.

For our current tests, the magnetic declination of Barcelona is about 17' as can be seen in figure; this is not so important but still it has to be taken into account since the software aims to be accessible anywhere in the world.

6.3. Compass implementation

The device used for the testing doesn't include a gyroscope, so all the tests were performed using only the compass and the accelerometer.

For the preliminary tests, it was considered the main camera as the moving factor of the whole world; both in orientation and positioning. Thus, the camera will have to update its position to the changes in the real world device.

Unity provides several methods for accessing the device sensors, specifically we can calculate the device positioning to the North Pole using several methods.

6.3.1. What unity offers

Unity offers an output result in the compass variable from the input class. The compass access values provides with several alternatives in order to get the magnetometer readings:

- magneticHeading: The heading in degrees relative to the magnetic North Pole
- rawVector: The raw geomagnetic data measured in microteslas
- trueHeading: The heading in degrees relative to the geographic North Pole

For our purpose it might be that trueHeading is the better choice, but since we have no control over how the data is treated, we also tried a different approach.

The magneticHeading data gives us nothing extra, is the same data as the trueHeading but with the correction of the magnetic deviation. The rawVector, on the other hand, can allow us to test a bit if we are able to create a more efficient/accurate version of the compass.

6.4. The trueHeading solution

First let's try the obvious solution. A model accessing the north projection of the internal unity methods is going to be tested. Let's keep in mind that one of the main purposes of the project is to test the Unity capabilities in working on Augmented Reality, so it is important to test all the tools the software offers for us.

The trueHeading variable outputs a float value with the rotation in reference to the geographic north that we are deviating. Thus, when the camera is pointing into the north, the result will be 0, and when its facing south, it will output 180 etc.

As a first test, a mini-scene has been implemented consisting of a main camera which will hold the vision, and four cubes in the four cardinal points around the camera (see figure 21)

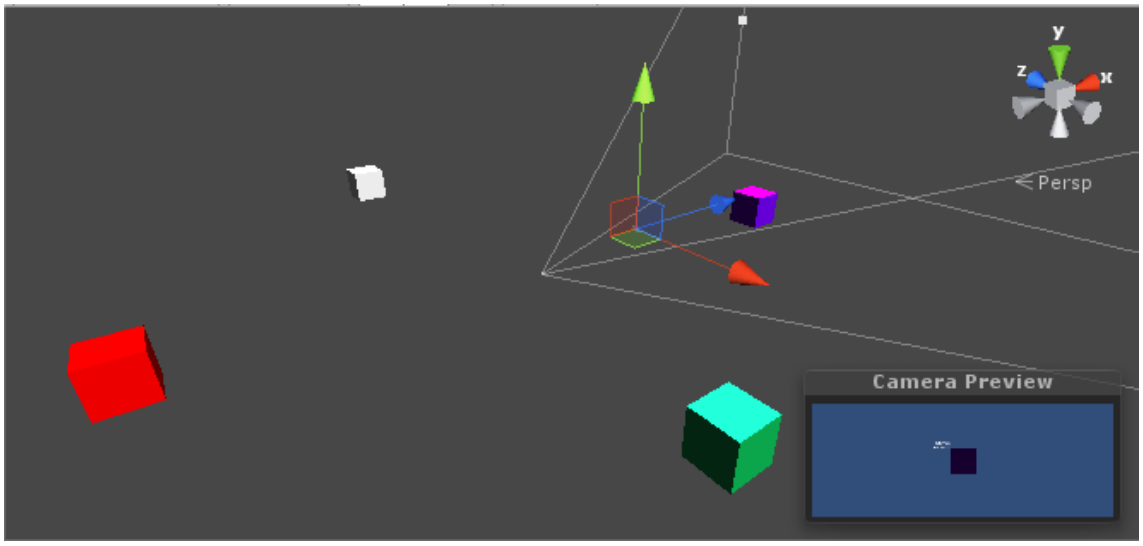


Figure 21: Setup for the compass rotation test in Unity

A script will be attached to the main camera, and it will control its movement. For the purpose of the test, the camera will only be able to rotate around its “y” axis, using the data obtained from the sensor reading.

In order to convert the compass value into a realistic camera rotation there are several methods at our disposal.

First, the script will access the device sensor data using the Input class, enabling its reading with the sentence:

Input.compass.enabled = true;

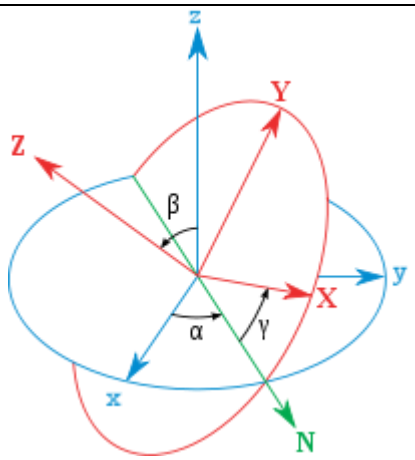
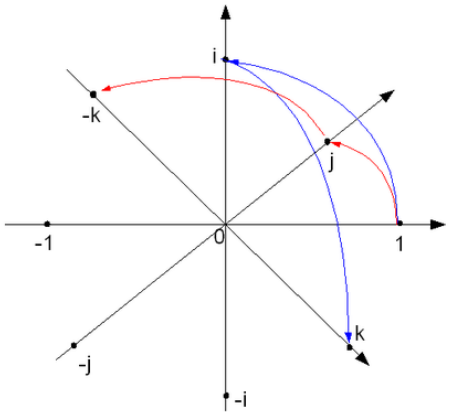
in the constructor method. This has to be repeated in any subsequent access to the sensors’ readings, so it will be omitted in future sections.

In order to use the data from the sensor to apply the rotation we used to approaches: quaternions and eulerAngles.

Quaternions are used to describe rotations in Unity, they have the advantage that doesn’t suffer from gimbal lock and they are the internally used format by unity rather than transformation matrices.

Euler Angles are representations in a 3 component vector of the rotation for each of the three axis in a 3d environment of a 3d object.

It is a complex mathematical tool that is far from the scope of the project, since are also never accessed directly; suffice to say they have some advantages to be calculated directly, as can be seen in the following table for comparison:

<table>
<tr><th>EULER ANGLES</th><th>QUATERNIONS</th></tr>
<tr><td> 
</td><td> 

Graphical representation of quaternion units product as 90°-rotation in 4D-space

ij = k
ji = -k
ij = -ji</td></tr>
<tr><td>Pros:
Easy to understand and visualize since they use 3D vectors.
Are computed faster since only 3 values to work with compared to 4 for Quaternions.</td><td>Pros:
Avoid the gimbal lock effect.
Perform the rotation based on a target vector and not a set of rotation on axes.</td></tr>
<tr><td>Cons:
One rotation can yield different result. Different engine use different orders (x,y,z) or (y,x,z) or else. The result is each time different
Gimbal lock effect. When one rotation aligns with another, the get locked together and one dimension is lost.</td><td>Cons:
Are slighlty slower since there is one more value than eulerAngles
Complex mathematics</td></tr>
</table>

The aim in any case is to obtain the more accurate and stable rotation of the camera so that's what we will observe when trying both methods, rather than its performance.

6.4.1. The euler angles solution

This solution considers the reading of the compass directly as the rotation value for the camera. For that, an eulerAngles vector is created and then passed as a rotation vector to the unity rotation function.

6.4.2. The Quaternion method

This method consists on passing the compass reading as a component of a quaternion vector, and then transforms the result into a rotation value output for the camera to read.

Testing results:
In both cases the result is sound and functional; the camera shows each cube as we turn around ourselves with the device in hand. We decided to choose the euler angles approach since it is easier for us to understand, although they almost get the same result.

6.5. Testing the raw Vector

As stated above, the program does all the calculation for us in order to obtain the Geographic north pointer; however, the project also tried to generate a custom-made rotation function for the camera using the original magnetic reading. What is intended to obtain is for the compass to point into the geographic north direction using a custom made function.

6.5.1. Implementation and results

The implementation is a simple mathematical equation using arctangents. A compass heading can be determined by using just the Hx and Hy component of the earth's magnetic field, that is, the directions planar with the earth's surface (see figure 22). The mathematical equations can be found in the annex.

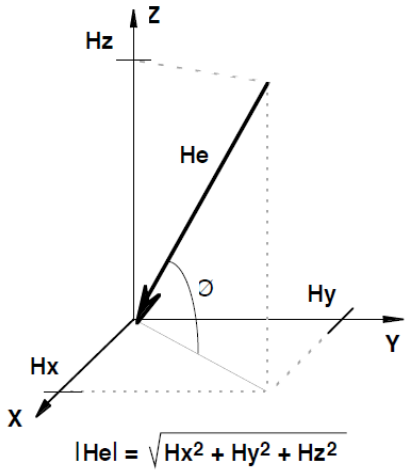


Figure 22: Schema of the geographic north calculation trigonometric function

The results are stable enough although there are a couple of issues to consider.

- First of all, the deviation of the magnetic north with respect the geographic north has not been taken into account. It doesn't really matter

for this test because the results were compared with the magnetic north output.

- The output results are not the same as the function results, they have a little deviation that tends to change dramatically whenever the device is held upwards or sideways or simply flat on a surface.

- There is a slight drift in values, which are due to the miss calibration of the device perhaps.

The results show that a custom calculation using the raw data from the sensor is possible, although more calibrations and tweaking are needed in order to achieve the precision that the default unity functions present.

In conclusion, we will be using the default functions since we are trying to achieve maximum precision.

6.6. Final camera implementation: smoothing the results

After the previous test, we decided on the trueHeading function to calculate camera orientation, and applied it with the eulerAngles approach. The result on screen is a bit unstable: the camera flickers as the numbers change in rapid succession; the compass precision is not as good as it should be.

In order to try and smooth the result we tried on different methods:

- *Update less often the camera position*: this has the disadvantage that mess up with the update function; on the other hand although it jumps less often, it still jumps position giving an awkward look to it.

- *Update only on absolute values*: This allows us to make sure the decimal changes don't affect the measure. Again, the problem is that the changes in degrees occur more often than expected and not only in decimal order of magnitude; the changes, if any, are not really appreciated.

The solution finally implemented was a smoothing function between changes; the function takes an initial value (current position) and final value (current sensor reading) and it creates a seamless transition between the two. The time of the transition can be adjusted, and it has, considering the feel of the camera; that is: if done too soon the flickering affect don't banish, and if done too late it creates a delay effect on the movement that doesn't feel natural.

This function was based on the code of a plugin called Itween (linked on the references) that allows tricking the update function to not change the value so often. The code for the plugin was open and it is been included in the annex for reviewing.

7. Final project Implementation

7.1. Section Overview

Once all the parts have been developed and tested individually, we added all of them into a single scene to work together. As mentioned, few concessions had to be done in order to ensure the performance quality and smoothness of the different elements

7.2. Final build elements

The final Scene containing all elements will be built using parts from the other test scenes (figure 23). In the final build the elements will be put together as described:

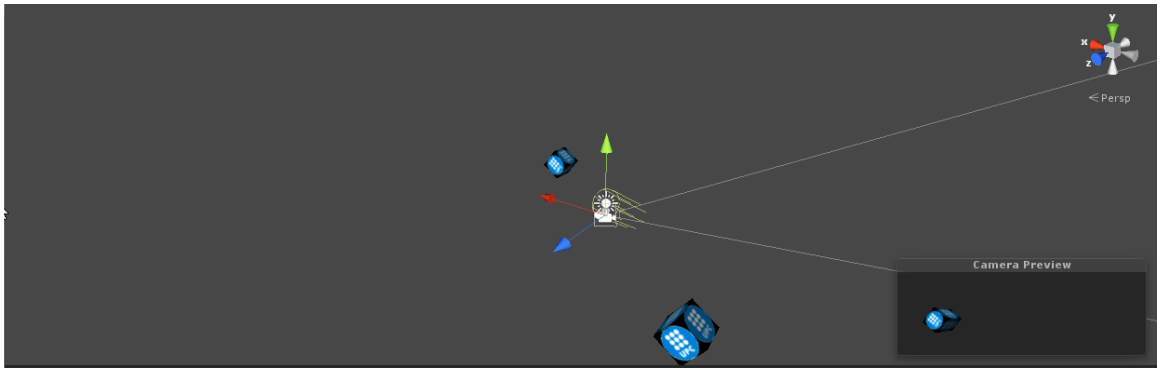


Figure 23: Final scene setup on Unity

The main camera
This camera will show the virtual elements and it will act as the user point of view. Since the main camera also has to be able to react to the compass it will also hold the script dedicated to it and it will handle the GPS. It will also have attached the scripts dedicated to the touchscreen and the information windows which will be addressed later.

- Compass script
- Touch screen script
- GPS script
- Information window script

The GPS script changes the position of the camera inside the real world to match with the readings of the location service functions.

The video feed secondary camera
As indicated in the previous chapters, a secondary camera will be the responsible of rendering the video feed of the device camera. Attached to it, there's the script dedicated to bind the texture and the camera.

The virtual elements: Markers
The final project contains two virtual cubes used as a representation of the real world positions (figure 24). For demonstration purposes, they will point to two buildings inside the UPC from Castelldefels.

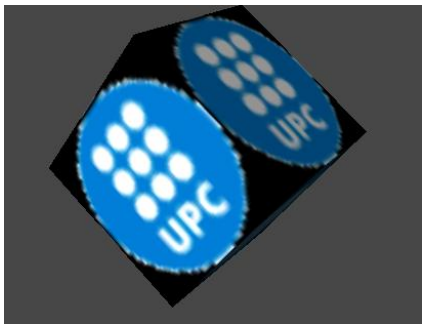


Figure 24: Close-up of the cube with the UPC logo used as a marker

This marker points to the DEAB and EETAC faculties. They are formed by a Game Object container which will include the cube, the distance label, and the name label. The game object container includes a hard-coded position on the Unity world with its latitude and longitude values.

- **Cube element:** The design is a rotating cube around one of its corners. It has a UPC logo as a texture and it has a basic rotation script attached (basically, it rotates at a constant speed)

- **Name and distance labels:** These labels are game objects with a GUI text component attached. Basically are texture 2D texts which are referenced as a position on the screen. Previous tests included a 3D modeled text which also had a script attached with a code that makes them face the player view; but it was finally substituted by GUI Texts since it doesn't require 3D render (which helps in the performance) and are always facing the camera. As an addition, they include a code which allows the text to stay on screen when they are not on the main camera's field of view, as a reference to rotate to it.

The project was limited to 2 markers at the same time so the performance stayed on reasonable levels.

7.3. Testing the final build

We tested the result using the profiler in a steady situation (figure 25); we do not want to strain the camera so it does not affect the readings on the other parts. Although it is difficult to see numerically, it can be observed that the camera feed still takes all the CPU processing memory. It is difficult to judge numerically since the other elements take so less computing power compared to that.

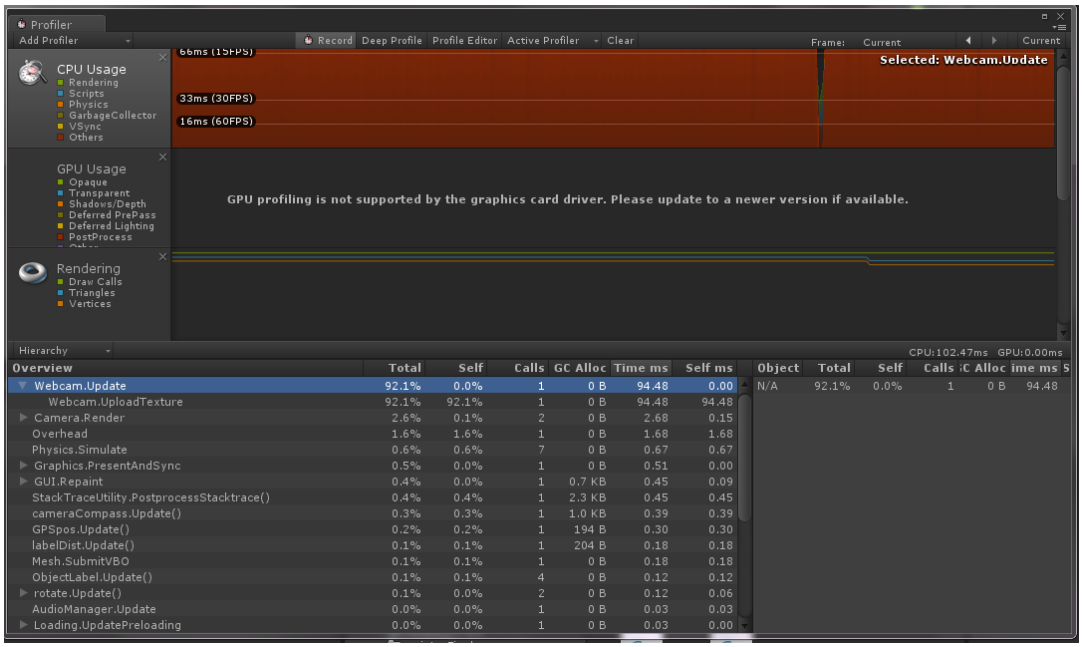


Figure 25: Profiler showing the performance on the final build

When switching the camera off we obtain a more balanced result between the other functions(figure 26), taking the rotation of the camera with the compass the most part of it. It is important to remember though that the device was remaining still throughout the whole process, so displacing the camera would put more load to that particular function.

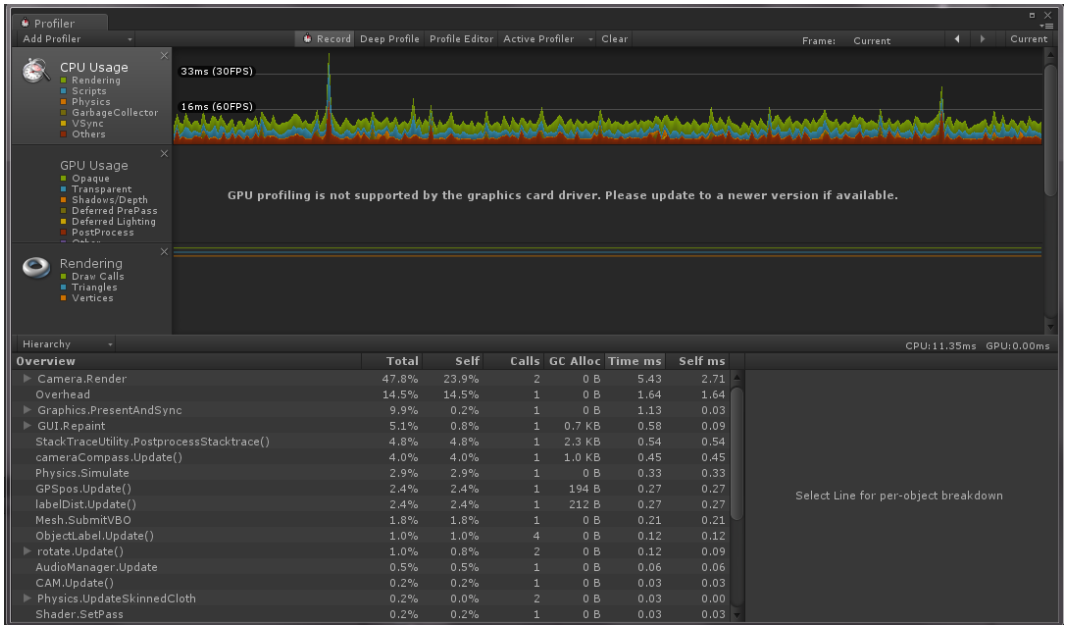


Figure 26: Profiler showing the performance on the final build without the camera

After this test it was also tried to include more markers in order to evaluate the performance in a stress situation. For that matter, the markers were duplicated on random positions on the vicinity of the original two up to 10 markers on the screen (see figure 27). The objective was to assess the quality in case all markers were present at the same time.

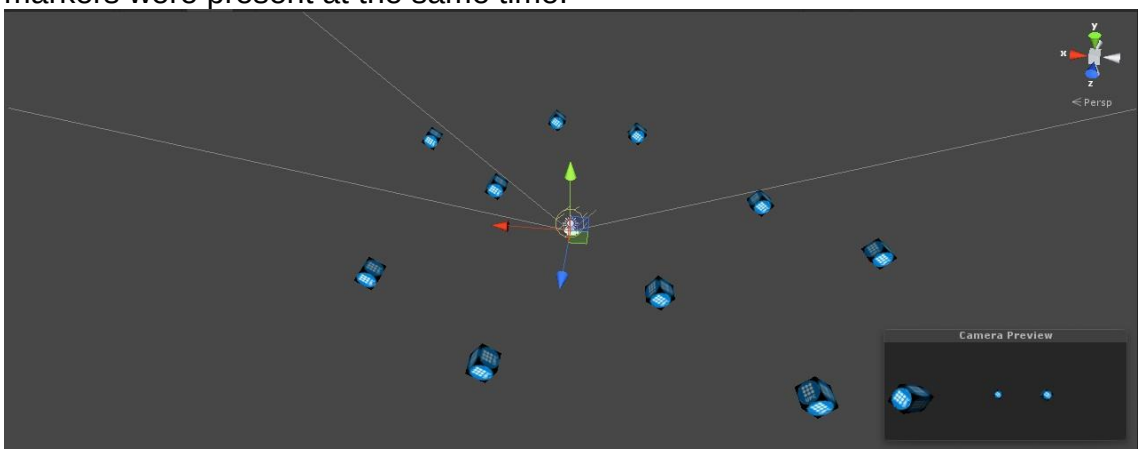


Figure 27: Editor's capture for the multiple markers test.

For that purpose the tests in both cases were repeated: with the camera on and with the camera off to see exactly what is the repercussion. It is also worth noting that, again, in order to obtain a stabke reading, the camera was mantained steady during the tests.

For the case of the ON camera, there is a slight change on the percentage dedicated to the Camera rendering (from 92 to 88); that means more resources are dedicated to the marker rendering. This is represented in a diminish of the quality in the playback of the camera video, all seems to move at less fps.

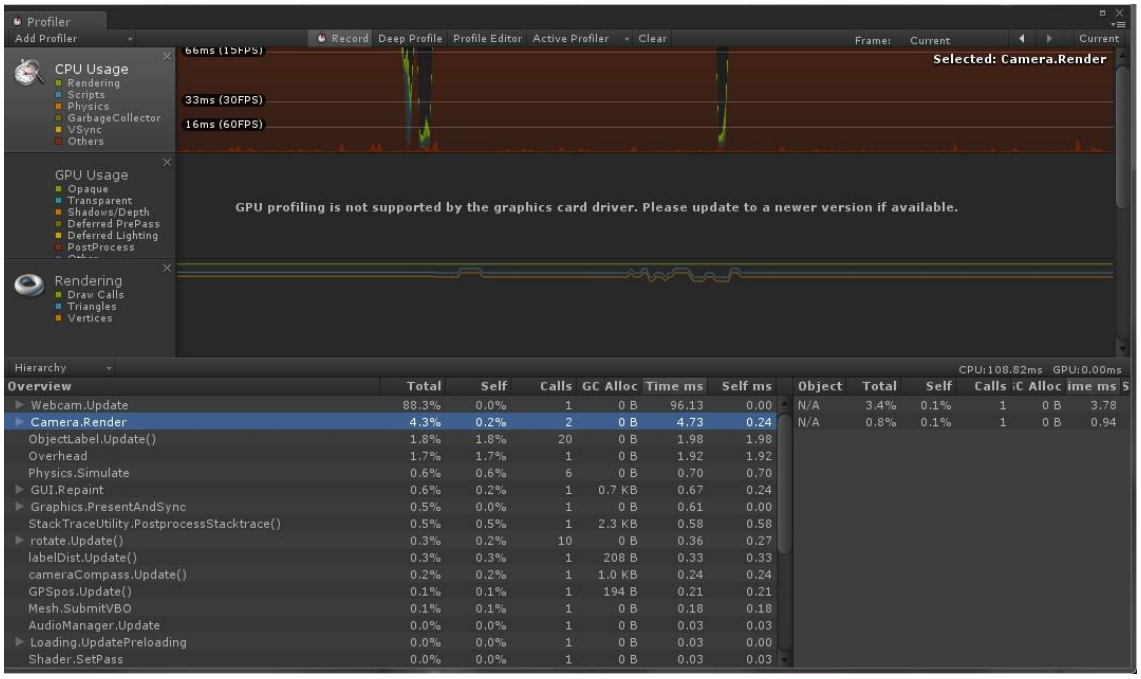


Figure 28: CPU usage with the camera ON

For the case of the OFF camera, it also suffers from some slight reduction in performance, but much less noticeable that in the case of the camera; that means the device is capable of handling the extra load since it is not nearly the saturation point. In the graph below, it can also be seen the green lines are more prevalent this time around. Although due the way unity works, it probably doesn't render the elements not present inside the camera view, and that is why the results doesn't show a more relevant change.

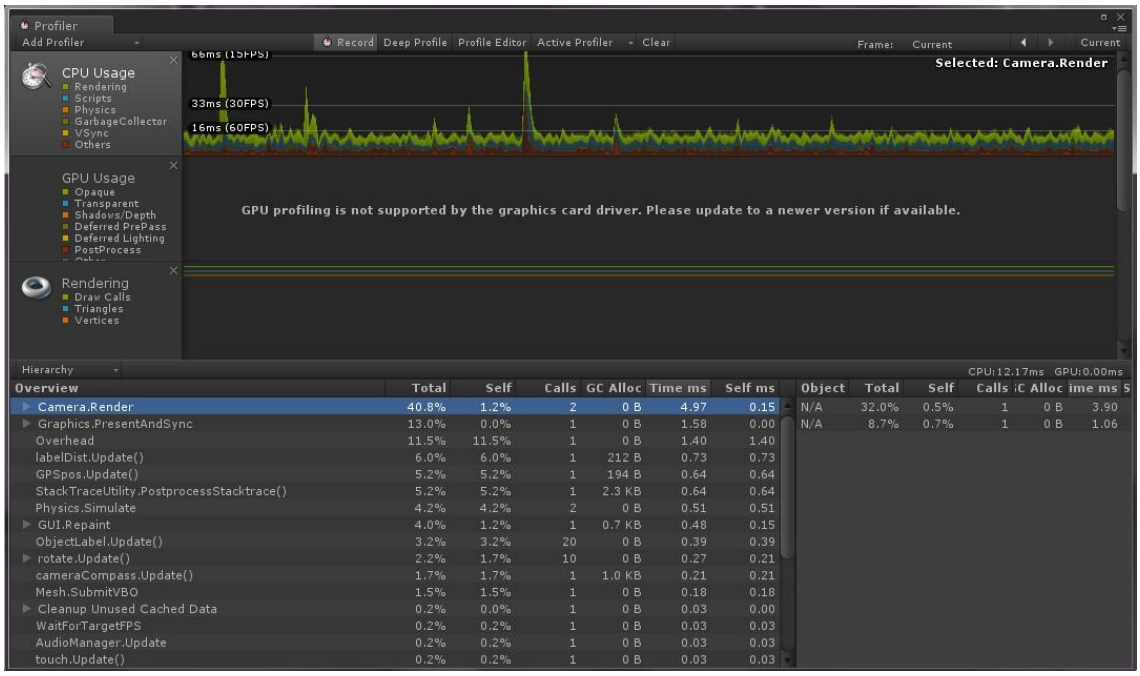


Figure 29: CPU usage with the camera OFF

All and all, the visual quality is good enough, although both the camera video feed and the rotation movement lose some framerate in the process. The case is specially noticeable in the case of the compass rotation element, since the camera rendering takes almost all the load of the CPU. As can be seen in the picture, the virtual elements integrate good enough with the real world; and the reference of the other markers are visible in the upper right corner (see figure 30).

Figure 30: Picture of the device playing the final application and the information window displayed when the marker is touched

When touching the markers, a brief description with a picture of the element can be seen. This takes almost no load of the CPU since the movement of the device feels almost identical, so there was no need to pause the camera when displaying it.

8. Timeline and monetization

8.1. Timeline

The timeline of the project was rather unusual. The main development of the project wasn't started until the second half of the year. There were discussed several other options including multiplayer games over Unity or multiplayer in augmented reality. Finally after some tests the documentation was not enough and we found an impass on the development. At that time, the knowledge of the software was deep enough that the idea for a custom made augmented reality project was proposed. The development was quite fast due to that, and in the summer months the project was implemented in their several parts with their test scenes.

Coming september, the designs were merged, tuned and we added the details such as the touchscreen features or the rotation and textures of the markers. The final month was dedicated to writing the thesis and additional testing and completition (see figure 31)

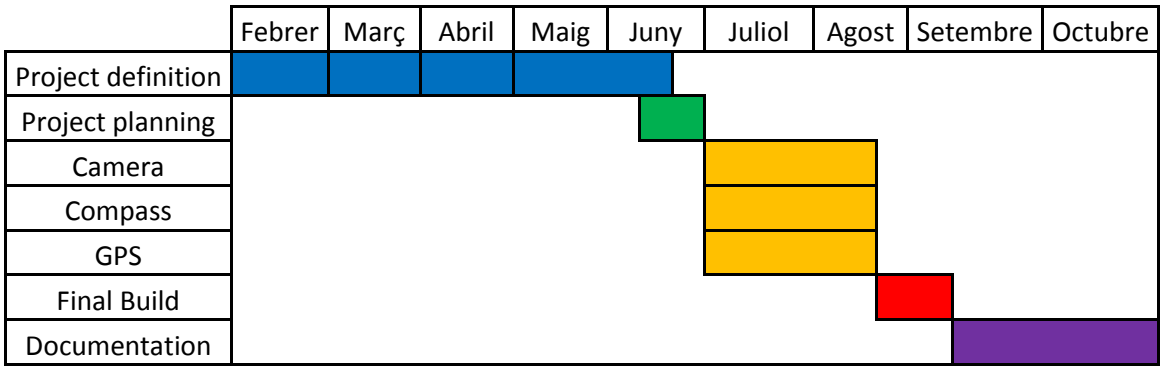


Figure 31: Time Schedule of the project

8.2. Project cost

This section is not easy to define since the project got a few changes. Considering the project didn't start until June, we may say that this is the point where the engineer begins working, and thus, when it begins getting paid since the final specifications didn't arrive until that point. However one may assume that the specification problems may or may not be a client issue with their requirements.

We assume 40€/h as a reasonable fee and commit the hours as 5 hours a day for the working 2 months and 2 hours a day for the rest. We also will add 1 h/day for the three previous months as a research time (considering only working days)

- 20 days x 1 hour x 3 Months x 40€ = 240€ for reasearch time
- 20 days x 5 hours x 4 Months x 40€ = 1600€ for development time
- 20 days x 2 hours x 2 Months x 40€ = 320€ for final details

2160€ for the whole project

The project is reasonably priced since my level is of junior engineering. The project is more of a test of functionality than a full priced application, and due to that is not ready for the market, and the client won't make any profit of it. Besides, the project definition took way too long and those are payed hours who got no result.

9. Conclusions and future work

The main purpose of this project was to prove how viable it is for Unity to develop an augmented reality application, how easy it can get and what is its performance.

After a few tests of the different elements it is fair to say that Unity can perform very well as a development software for this kind of applications. Its architecture game oriented allows for a simple and intuitive distribution of elements, and it natively supports all the elements necessary for its implementation.

As seen in the tests, the Camera rendering is an issue with Unity; at least without a more thorough coding of its rendering. It is obvious that the software is not ready for a constant video feed rendering of each frame; and perhaps a further investigation on that matter will allow for a better camera function (the default camera on the device works a lot better than ours). A better device will surely improve the experience and smoothness of it, but in general it worked well enough for the purpose.

Since the camera rendering takes a lot of the processing capabilites, it is difficult to evaluate the other parts without taking out the camera. When tried with more amrkers, the result shown as expected since the camera saturates easly the CPU capabilities og the device, and thus the perfomance of the application gets noticeable affected. If the elements are distant enough, unity auto computes the rendering so if it gets out of camera range they are simply not rendered as would do in a videogame project. That is the main reason why Unity is such a good platform for AR development of videogames.

This is far from a perfect project though, the world coordinates were approximated with its latitude and longitude and it works in large distances and scales; but if it requires precision, the actual design doesn’t work very well. Working a bit more with the conversion to cartesian coordinates may improve the project on this matter.

The compass detection and rotation of the camera are functional, but not precise enough and it loses calibration when moving around with the device. The movements are not as smooth as they should be: it is erratic in its value and although it could be partially solved taking a smoothening function, again it fails when working in a precision levels, that is, when the object is too close. With the aid of a gyroscope it may be compensated, but this project is a valid test in case one is not available.

The compass is sensitive with pitching and rolling, so the reading goes off very quickly when not mantained steady. This could be partially compensated with the aid of the accelerometer: we performed some tests on it but it proved a bit more complex for the actual results, and it was abandoned in favor of more important goals. The problem is that the compass of the phone is designed to work flat as a real compass would, and this it could never be precise enough as

it is now. A bit more work on the compass function to compensate for this fluctuations may lead to a more solid result.

10. Enviromental Impact

Since this is a software project there is no noticeable envorimental impact besides the energy produced and the natural emissions of the device. This application can be used, however, in substitution of actual information panels on site allowing to preserve some natural enviroment without the intervention of man made elements.

11. References

[1] MSDN Magazine. Orientación con la brújula de Windows Phone
Charles Petzold. June 2012
http://msdn.microsoft.com/es-es/magazine/jj133827.aspx

[2] Holistic 3D. Unity Location and Compass for Mobile
November 5 2012, By Penslayer
http://www.holistic3d.com/?p=574

[3] Unity Answers. Polar (spherical) coordinates to xyz, and vice versa.
November 28 2011
http://answers.unity3d.com/questions/189724/polar-spherical-coordinates-to-xyz-and-vice-versa.html

[4] Unity Answers. Problem with localEulerAngles (180 degrees error?)
September 2011
http://answers.unity3d.com/questions/313966/problem-with-localeulerangles-180-degrees-error.html

[5] Unity Answers. Compass conntected to camera is jerky
January 2007
http://answers.unity3d.com/questions/313966/problem-with-localeulerangles-180-degrees-error.html

[6] Unity Wiki. OpenStreetMap for unity iPhone
28 November 2012
http://wiki.unity3d.com/index.php?title=OpenStreetMap_for_unity_iPhone

[7] Unity Wiki. GPS Global Positioning System
10 January2012
http://wiki.unity3d.com/index.php?title=GPS_Global_Positioning_System

[8] Sundh.com.QolNreNr.dpuf. Stabalize compass of iPhone with gyroscope
18 September,2011
http://www.sundh.com/blog/2011/09/stabalize-compass-of-iphone-with-gyroscope/

[9] Love Electronics. Tilt Compensating a Compass with an Accelerometer
18 October,2011
http://www.loveelectronics.co.uk/Tutorials/13/tilt-compensated-compass-arduino-tutorial

[10] MatD. Unity GPS plugin development tutorial: building a Android plugin for Unity with Eclipse and Ant
2012
http://www.mat-d.com/site/unity-gps-plugin-development-tutorial-building-a-android-plugin-for-unity-with-eclipse-and-ant/

[11] Stack Overflow. transform longitude latitude into meters
11 June,2010
http://stackoverflow.com/questions/3024404/transform-longitude-latitude-into-meters

[12] Wikipedia. Haversine formula
13 September,2013
http://en.wikipedia.org/wiki/Haversine_formula

[13] Ray Wenderlich. Introduction to Augmented Reality on the iPhone
30 June,2011
http://www.raywenderlich.com/3997/introduction-to-augmented-reality-on-the-iphone

[14] Ray Wenderlich. Augmented Reality on Android: Using GPS and the Accelerometer
10 October,2010. Chris Haseman
http://www.devx.com/wireless/Article/43005

[15] MatD. Unity 3D WebCamTexture rotation problem with Android – How to rotate the camera picture ?
2012
http://www.mat-d.com/site/unity-3d-webcamtexture-rotation-problem-with-android-how-to-rotate-the-camera-picture/

[16] Sfonge. Compensating accelerometer with the compass - the limitations
2012. Gabor Paller
http://www.sfonge.com/forum/topic/compensating-accelerometer-compass-limitations

[17] Random Programmer Babble. Camera Feed Background in Unity
25 August, 2012
http://www.sfonge.com/forum/topic/compensating-accelerometer-compass-limitations

[18] Sensor Wiki. Magnetometers
3 April, 2013
http://sensorwiki.org/doku.php/sensors/compass_magnetoresistive

[19] Unity Answers. Raycasting to check what object (with a particular tag) is underneath the Player
31 October, 2010
http://sensorwiki.org/doku.php/sensors/compass_magnetoresistive

[20] How Stuff Works. How Augmented Reality Works 2012. Kevin Bonsor
http://wiki.unity3d.com/index.php?title=ObjectLabel

[20] iTween.
http://itween.pixelplacement.com/index.php

12. Annex

Second Camera solution script

```
        void Start () {
                cameras = WebCamTexture.devices[0];


   camfeed = new WebCamTexture(cameras.name, 640, 480, 30);

   camfeed.Play();

   GameObject bgimg = GameObject.Find("background image");
   bgimg.guiTexture.texture = camfeed;


        }
```

Calculation of the north direction using the raw magnetic reading

```
public void rotateraw()
        {
                float direction=0;

                if (Input.compass.rawVector.y > 0)
                        direction = 270 - (Mathf.Atan(Input.compass.rawVector.x
                        /Input.compass.rawVector.y))*180/Mathf.PI;

                else if(Input.compass.rawVector.y < 0)
                        direction = 90 - (Mathf.Atan(Input.compass.rawVector.x
                        /Input.compass.rawVector.y))*180/Mathf.PI;

                else if (Input.compass.rawVector.y == 0 && Input.compass.rawVector.x<0)
                        direction = 180;

                else if (Input.compass.rawVector.y ==0 && Input.compass.rawVector.x >0)
                        direction = 0;

                textaccel.GetComponent<TextMesh>().text = "Compass : "+
                Input.compass.magneticHeading;

                textCompass.GetComponent<TextMesh>().text = "Compass: "+direction;
                comp = new Vector3( 0,direction,0);
                iTween.RotateUpdate(camera2,comp,3f);


        }
```

Harvesin function for distance calculation

```
private float harvesin (GameObject pos)
        {
                float R = 6371; // km
                float dLat = (pos.transform.position.z/10000-Input.location.lastData.latitude)
                *(Mathf.PI/180);
                float dLon = (pos.transform.position.x/10000-Input.location.lastData.longitude)
                *(Mathf.PI/180);
                float lat1 = (Input.location.lastData.latitude)*(Mathf.PI/180);

                float lat2 = (pos.transform.position.z/10000)*(Mathf.PI/180);
```

```
                float a = Mathf.Sin(dLat/2) * Mathf.Sin(dLat/2) +
                          Mathf.Sin(dLon/2) * Mathf.Sin(dLon/2) * Mathf.Cos(lat1) * Mathf.Cos(lat2);
                float c = 2*Mathf.Atan2(Mathf.Sqrt(a), Mathf.Sqrt(1-a)); //2*Mathf.Asin (Mathf.Sqrt(a));
                float d = R * c;
                return d;


        }
```

Smoothing function from the itween plugin

```
public static void RotateUpdate(GameObject target, Hashtable args){
    CleanArgs(args);

bool isLocal;
float time;
Vector3[] vector3s = new Vector3[4];
Vector3 preUpdate = target.transform.eulerAngles;

//set smooth time:
if(args.Contains("time")){
     time=(float)args["time"];
     time*=Defaults.updateTimePercentage;
}else{
     time=Defaults.updateTime;
}

   //set isLocal:
 f(args.Contains("islocal")){
     isLocal = (bool)args["islocal"];
}else{
    isLocal = Defaults.isLocal;
}

//from values:
if(isLocal){
   vector3s[0] = target.transform.localEulerAngles;
}else{
   vector3s[0] = target.transform.eulerAngles;
}

//set to:
if(args.Contains("rotation")){
if (args["rotation"].GetType() == typeof(Transform)){
   Transform trans = (Transform)args["rotation"];
   vector3s[1]=trans.eulerAngles;
}else if(args["rotation"].GetType() == typeof(Vector3)){
   vector3s[1]=(Vector3)args["rotation"];
  }
}

//calculate:
                        vector3s[3].x=Mathf.SmoothDampAngle(vector3s[0].x,vector3s[1].x,ref
vector3s[2].x,time);
                                vector3s[3].y=Mathf.SmoothDampAngle(vector3s[0].y,vector3s[1].y,ref
vector3s[2].y,time);
                                vector3s[3].z=Mathf.SmoothDampAngle(vector3s[0].z,vector3s[1].z,ref
vector3s[2].z,time);
```

```
//apply:
if(isLocal){
   target.transform.localEulerAngles=vector3s[3];
}else{
   target.transform.eulerAngles=vector3s[3];
}

//need physics?
if(target.rigidbody != null){
   Vector3 postUpdate=target.transform.eulerAngles;
   target.transform.eulerAngles=preUpdate;
   target.rigidbody.MoveRotation(Quaternion.Euler(postUpdate));
                                                }
                                    }
```