Augmented Reality on the PlayStation Vita

1. Introduction

This project required building a game which used either stereoscopic 3D or the PlayStation Vita's augmented reality (AR) functionality. The chosen technology needed to be integrated into gameplay, not merely used as an effect.

The augmented reality option was chosen. In the prototype game that was developed, 'You AR Being Watched', the AR markers are arranged on a flat surface upon which the game is played. In the game the player is trying to score points by quickly identifying and eliminating target civilians (represented as untextured spheres) from among a crowd. By moving the PS Vita around the player can view the game world from different perspectives, zooming in on smaller groups of civilians as they narrow their search.

The game exploits AR technology to implement a novel control scheme but, as there are no mechanics which involve actually playing with the AR markers, the game idea could easily be adapted to work without it. In fact, it might be better off that way – the platform has limitations which provide interesting challenges to the developer but ultimately constrain the game from reaching its full design (and market) potential.

If the game was taken further as an AR game it would double-down on the technology by adding mechanics which actually directly involve the AR markers and more mechanics which require the player to move around and view the game world from different angles and distances.

2. How to Play

You AR Being Watched

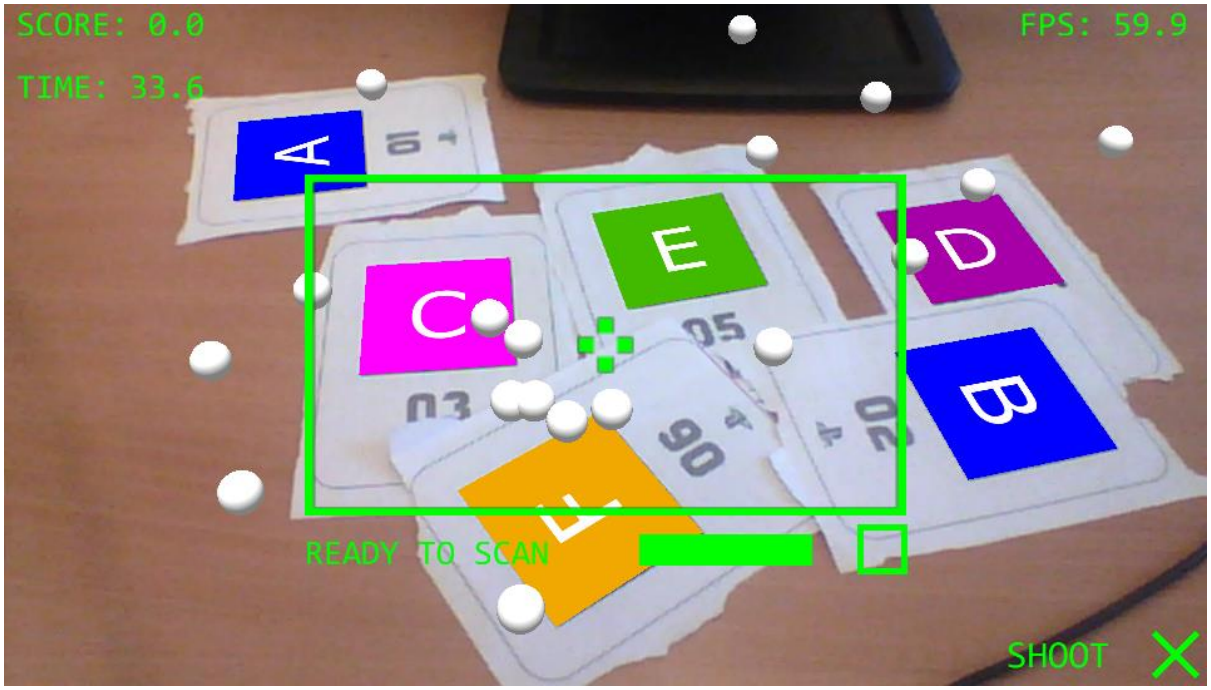
You are manning a police drone in a totalitarian state using an augmented reality interface, searching a crowd of civilians for a known criminal. Use your drone's PerpFinder™ Scanner to identify your current target, then neutralise them with your drone's in-built Pacification Cannon™. You will then be assigned a new target to find and apprehend – a drone operator's work never ends.

Setup: *Arrange your AR markers on a flat surface. Only one is required to play, but the more you use, the larger the game's area can be: you can spread them out as much or as little as you'd like.*

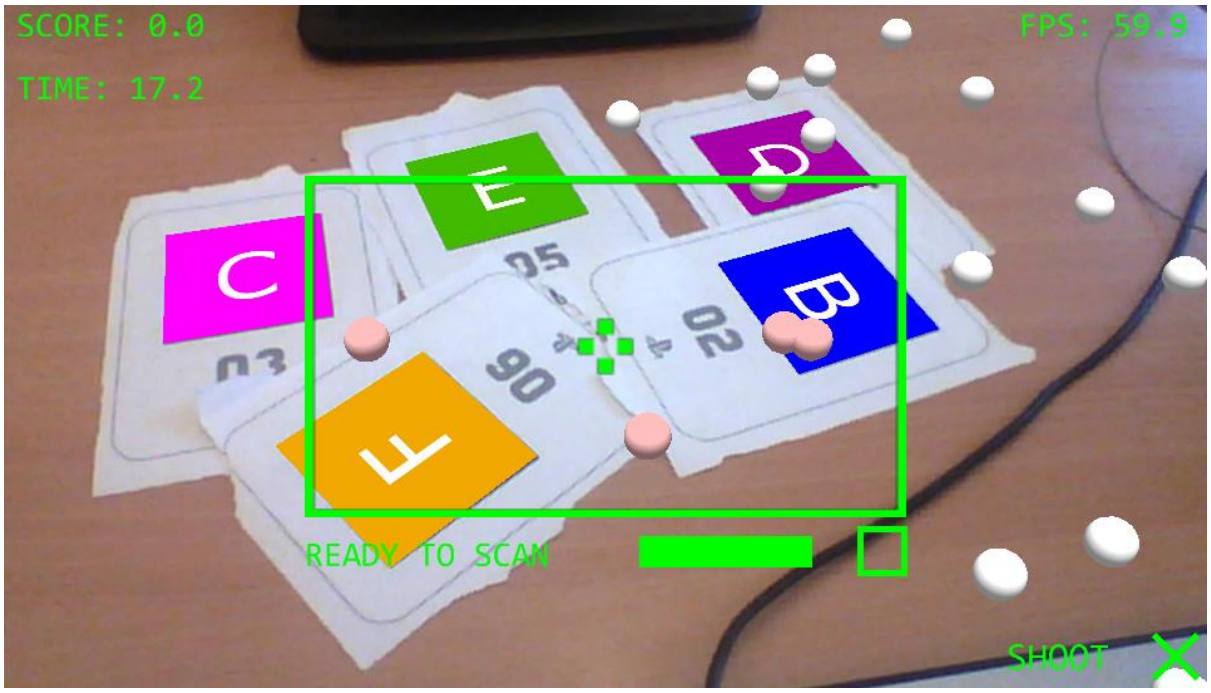
Once the game starts, don't move the markers!

Controls:

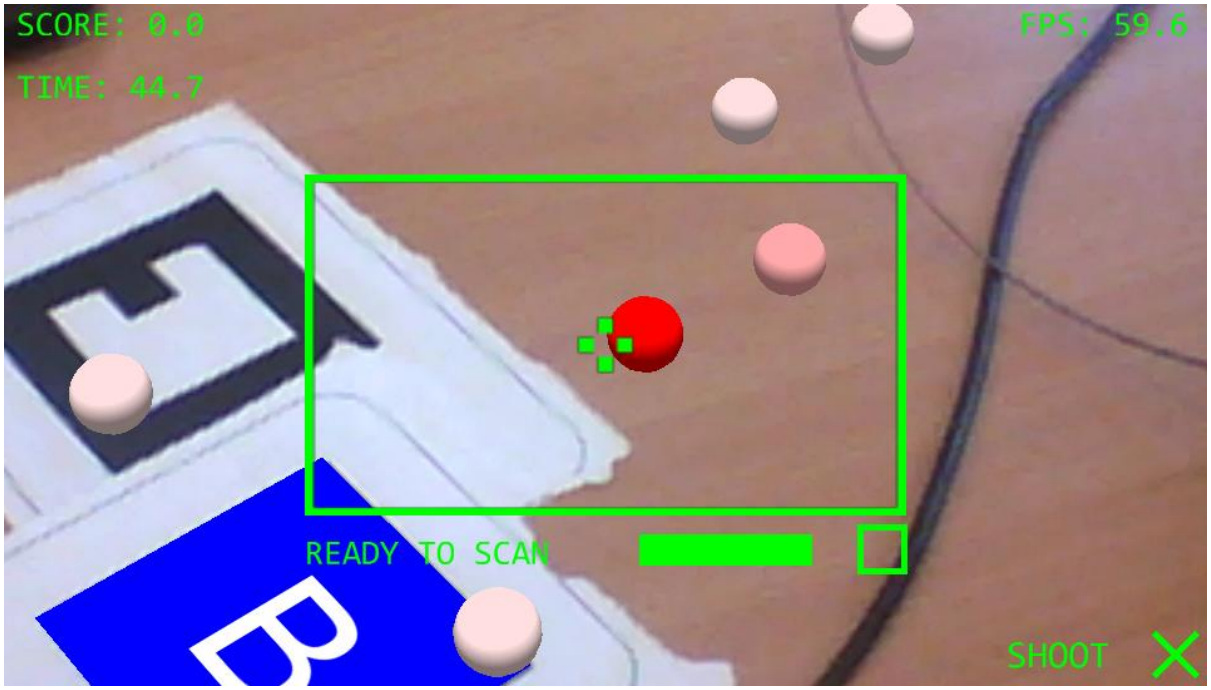
*Press **SQUARE** to scan the civilians inside the rectangle on the heads-up display.*



*After scanning, the civilians in the rectangle might turn red! This means one of them is the **TARGET**. The fewer civilians are in the rectangle, the redder they'll all become; the redder a civilian is, the more likely it is to be the target.*



*Once you think you know who the target is, aim at them with your crosshair and press **X (CROSS)** to neutralise them! You will lose points for neutralising the wrong civilian, so don't shoot until you're certain.*



*You will now be awarded **POINTS** and be assigned a new **TARGET**.*

Hints

- *Start by scanning large groups of civilians, then close in on smaller groups once you start seeing red. The **ZOOM** feature **(LEFT SHOULDER / RIGHT SHOULDER)** can help with this.*
- *If you neutralise your **TARGET** in less than 20 seconds you will receive bonus points.*
- *The game never ends. Sorry.*

3. Application Design

3.1 Overview

The programming process involved a lot of experimentation at first, as it was not known whether certain things were possible or how they could be achieved. Gradually things were refactored into a reasonably-organised state.

Here are the less interesting parts of the application:

- **ARApp** (**ar_app.h**, **ar_app.cpp**) is the 'application class' which manages the game and program execution at the top level. It doesn't do anything out of the ordinary.
- **HUDMessenger** (**hud_messenger.h**, **hud_messenger.cpp**) is a class which controls the management and rendering of a queue of short, temporary messages (such as how many points the player just scored), one at a time, on the screen. It is very rudimentary, but does what it's supposed to do.
- The **ARCamera** class has two jobs: to bridge the gap between the PS Vita's screen and the PS Vita's in-built camera by providing utility functions, and to construct a perspective projection matrix which has the correct width and height. It exists because all its state and functionality was refactored out of **ARApp**.
- **Collision module** (**collision.h**, **collision.cpp**): Groups together classes and functions which are used to test intersection between entities, such as the **Frustrum** class and the **LineSegIntersectsSphere** function.

- **Mesh-gen module** (**meshgen.h**, **meshgen.cpp**): contains functions for generating **abfw::Mesh** objects.

Entities which require some more explanation will now be covered.

3.2 Civilian Class

Civilian (**civilian.h**, **civilian.cpp**) used to inherit from a GameObject class that was removed from the project in a refactoring pass when it was decided that non-Civilian children of GameObject weren't going to make it into the prototype.

Update(): Every frame the Civilian moves towards its current target, checks whether it is within a certain distance of it and, if it is, finds a new target (**Civilian::NewTarget()**).

To enable the ability to change the colour of Civilians the Abertay Framework was modified. It was easier to modify the framework than to extend it. The additions in **renderer_3d.h** and **.cpp**, **default_3d_shader_vita.h** and **.cpp**, and **default_3d_shader_f.cg** are highlighted with comments. The programmer can now call **renderer_3d->set_blend_colour()** as needed and the colour they supply to the function will be multiplied with the colour of the texture used when rendering the **abfw::MeshInstance**.

3.3 Marker Class and Scene Hierarchy

Creating a **Marker** class (**marker.h**, **marker.cpp**) was an obvious first step to group together data and functionality regarding the AR markers. An AR marker has three elements of state: its marker ID (an integer from 0 to 5), whether it have been found on-screen by the tracking software and, if so, its transform in camera space. The **Marker::Update** method uses the Sony framework to attempt to find the marker matching the ID and then get its current transform.

One of the key application features which makes the game possible is a simple scene hierarchy system: Civilians are 'parented' to Markers. Civilians own local transforms which define their position, orientation and scale relative to the transform of their parent Marker. When it is time to render or otherwise determine the transform of the Civilian in world (actually camera) space the local transform is multiplied by the parent's transform. At present the scene hierarchy is only one layer deep for simplicity's sake.

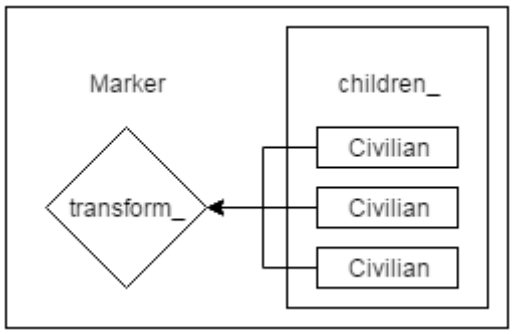


Figure 1: Diagram illustrating the structure of the Marker class and the parenting system.

Civilians are parented to Markers dynamically such that when their current parent is lost by the marker-tracker they are re-parented to the nearest available marker. This helps create the impression of a game world that is larger than the screen. When a marker goes off-screen or is obscured the game can continue running as if nothing happened. When all markers are off-screen, the game enters a paused state. All of this management is handled by ARApp.

The re-parenting system (which can be seen in **Civilian::set_parent()** in **civilian.cpp**) isn't perfect. Sometimes the game object ends up with a weird orientation, so the **set_parent** method simply gets rid of the object's orientation entirely and only cares about the object's translation and scale. This isn't really a solution and means that the player can tell when a Civilian is re-parented as it will rotate suddenly to match the orientation of its new parent. In the prototype this visual annoyance is concealed by the fact that all the Civilians are represented as untextured spheres. In the full game an actual fix would be needed.

Sometimes Civilians will appear to 'jump' when they are re-parented, because the two Markers involved are not co-planar or their orientations are different such that the Civilian is now above or below the plane of their new parent. The Civilian's z-position is snapped close to zero so that it lies on the same 'plane' as its parent, causing it to jump downwards or upwards. As long as all the markers are arranged on the same flat surface this isn't too much of a problem. The jumping is worse when a Civilian is parented to a Marker that is a long distance from it, so ARApp re-parents Civilians to their nearest available Marker rather than just any available Marker.

A way of determining the shared plane on which the Markers all roughly lie is needed to avoid these errors. Perhaps the average position and orientation of all the Markers could be used as the parent transform for all Civilians.

When the tracking software is struggling to get a 'lock' on the exact position of 'jitter' may occur, and Civilians parented to that Marker will appear to move back and forth on the spot rapidly. Usually, but not always, this means that the marker is far away from the camera and therefore takes up a small number of pixels, so its transform is difficult to calculate. This cannot be fixed but game objects could be re-parented to markers which are closer to the camera if the marker they are parented to is too far away from the camera. Sometimes other factors cause the jitter such as rapidly-changing lighting conditions, or the marker not being printed black enough for the camera to pick up well. The program could detect if a marker's transform is changing between frames to an unexpected degree and re-parent its children to less jittery markers, but the detection system would have to take in to account movement of the camera by the player.

Markers store their child Civilians in an std::vector by value. This keeps code simple but has some drawbacks. Civilians don't own anything they point to so that's not a problem when they're copied, but the Civilian class is very large. This means that cache usage is probably very bad, and when a re-parenting event occurs the program needs to stop in order to copy lots of data from one std::vector to another, likely requiring a fair bit of allocation and deallocation of memory. With the low number of Civilians currently in the game, and with such rudimentary graphics, any performance hitches caused are invisible to the user, but the current system is not scalable and needs to be re-written.

3.4 Scanning Civilians

When the player 'scans' the area in front of them, all of the civilians contained within the rectangle in the middle of the screen are collected into a group. This is done using frustrum-versus-sphere collision detection (see the **FrustrumContainsSphere** function in **collision.cpp**).

1. Set up scanning frustrum
2. scanned_civilians := empty collection of pointers to Civilians
3. For each Marker (including those not on screen):
    a. For each Civilian in the Marker's children:
        i. Get the Civilian's bounding sphere in camera space
        ii. If scanning frustrum contains the sphere:

            1. Add a pointer to the Civilian to scanned_civilians.
4. Determine if the target Civilian is in scanned_civilians.
    a. For each Civilian in scanned_civilians:
        i. If the civilian's ID variable matches the ID of the target civilian:
            1. scanned_civilians contains the target
            2. End for loop (early)
5. If scanned_civilians contains the target:
    a. Calculate a colour between white (RGB 1, 1, 1) and red (RGB 1, 0, 0). The larger scanned_civilians is, the whiter the colour is.
    b. For each Civilian in scanned_civilians:
        i. Set the Civilian's colour variable to the redness we just calculated.

This pseudocode is implemented in the ARApp class' **TryScan()** method. There is some room for improvement, or at least for doing things differently. For example, instead of looping for every Marker it could just skip over those which are not on-screen.

The application has not yet been tested and profiled with huge numbers of Civilians, so whether or not optimizations such as spatial partitioning are required to increase the speed of collision detection is unknown. (In a scene defined in camera space, what exactly does a spatial partition look like, anyway?)

3.5 Shooting Civilians

The player's goal is to find the target civilian and then to shoot them. This is handled by **ARApp::TryShoot()**. Like **TryScan()**, it loops through all the Civilians in the scene and runs an intersection test on them. In this case the civilian's bounding spheres are tested for intersection with a line segment (see **LineSeg** class) that points down the negative Z-axis. That is, it points from the camera, through the centre of the screen, into the scene. If multiple civilians intersect this line segment, then the first is selected as the one that has been hit by the player's 'bullet'. This just means finding the one with the smallest absolute Z-position.

Bounding spheres were chosen because axis-aligned bounding boxes would be inaccurate and oriented bounding boxes would have required more development time and computation time (without necessarily giving better results).

The same points about optimizations apply here as they do above.

3.6 Source Control

Subversion (Apache Software Foundation 2013) was used through the interface software TortoiseSVN (Kung and Onken 2014) to help with development. The full repository will not be included in the coursework submission but is available on request.

4. Reflection

4.1 Context

It is difficult to talk about AR without talking about its sibling virtual reality. AR and VR are very much still emerging technologies in terms of their reliability, usefulness and success in finding a market. Recent years have, however, seen an explosion of interest and investment in augmented and virtual reality among developers, but it is yet to be seen whether this is also felt by general consumers.

Developers are still figuring out how AR and VR games can be played and what constitutes good design. Formal research has been done into how AR games of various genres should be controlled by their players (Pollmann et al. 2013, p. 120–125). Some of this work dates from long before the

current AR surge, for example 'Possession Techniques for Interaction in Real Time Strategy Augmented Reality Games' (Phillips and Piekarski 2005, p. 2), showing that there has been a long-term interest among developers and designers in using the technology. What is interesting about 'Possession Techniques…' is that it was written in the context of an AR landscape in which AR displays were mostly head-mounted, as is noted in 'Spatial Augmented Reality' (p7) (Bimber and Raskar 2005):

> *The virtual reality community has oriented themselves away from head-mounted displays and towards spatial displays. Today, a large variety of spatial displays make up an estimated 90% of all VR displays. Head-mounted displays, however, are still the dominant displays for augmented reality. The reason for this might lie in the strong focus of mobile AR applications – requiring mobile displays.*

Looking at today's AR/VR landscape, it is clear that a huge shift has taken place in both communities. Head-mounted displays like the Oculus Rift, HTC Vive and Samsung Gear VR are now dominant in VR while spatial displays have been largely forgotten about. Meanwhile handheld AR has grown a great deal with the rise of smartphones and tablets, leading to its inclusion as a feature of the PlayStation Vita. Head-mounted AR has yet to prove itself in the consumer world: the two main stories about it in 2015 were the failure of Google Glass (Kaplan 2015; Bilton 2015) and Microsoft's HoloLens demonstrations, which were received lukewarmly.

Marker-tracking functionality is impressive but there is still a degree of 'jankiness' which disrupts the experience. The limited resolution of the camera, the natural shaking of the player's hands, and the speed of the image-processing software all contribute to an experience that is far from perfect, even on high-end dedicated systems like the PlayStation Vita. The reliance on markers is big limitation of the technology. Upcoming AR technologies are powerful enough to not need them. Google's Project Tango and Microsoft's HoloLens both use a combination of advanced sensory equipment such as depth cameras to gather as much information as possible about the world in front of them so that they can accurately project 3D objects onto the display. Projects such as *InnAR Wars*[1] (Barbagello 2015) and *Minecraft* on HoloLens demonstrate this technology.

4.2 Project Assessment

Requiring the player to physically move around while playing a game is somewhat interesting, but arguably the markers should be incorporated into gameplay to make full use of the technology. Unfortunately the Vita is heavy, expensive and designed to be held in both hands, so it was decided to avoid involving game mechanics which required the player to put down the Vita or hold it in one hand in order to move markers around because they make for an awkward experience.

Some Vita AR games which use the markers have the player rearranging them as they play to solve puzzles, but others make use of them without do not incorporate moving them around into gameplay (Silvester 2014). By using different combinations of markers the player selects how difficult the game will be, or what different elements will be involved.

In *You AR Being Watched* the markers could represent different buildings. When the player sets up the markers on their table, the markers they select could define which buildings exist in the town. Civilians and the player could make use of the buildings in meaningful ways to add more depth. The buildings would also obstruct the player's view.

[1] Is this the worst game name involving a pun on AR yet? It's not even a pun, at least not in English.

The Civilians could display more complex behaviour. They could path-find around each other on their way to their targets, pause, maybe stop and talk to each other. Unfortunately, creating a convincing crowd simulation which enabled more engaging and visually exciting gameplay was definitely beyond the scope of the project.

Audio would improve the experience a great deal as sound effects could help to get around the limitations of the abstract graphics, allowing the player to more quickly understand their role in the game and what they're trying to do in it. Unfortunately there wasn't enough time to add sound effects.

The zoom feature was a good idea; it lets the game get more out of the limited space it exists in, helping to sell the idea that the Vita is a window onto a world. The player can zoom in on and isolate smaller groups of civilians without having to physically move up close and potentially lose AR markers.

In conclusion, the prototype demonstrates a game idea that isn't impossible to implement without AR technology, but becomes a more novel and original game thanks by making use of it. AR is interesting to develop for and presents challenges which are enjoyable to overcome.

5. References

Apache Software Foundation 2013, Apache Subversion.

Barbagello, R 2015, The Challenge of Building Augmented Reality Games in the Real World, viewed 20 March, 2016, <http://ralphbarbagallo.com/2015/08/17/the-challenge-of-building-augmented-reality-games-in-the-real-world/>.

Bilton, N 2015, Why Google Glass Broke, *The New York Times*, viewed 21 March, 2016, <http://www.nytimes.com/2015/02/05/style/why-google-glass-broke.html>.

Bimber, O and Raskar, R 2005, Spatial Augmented Reality Merging Real and Virtual Worlds, *Scientist*.

Kaplan, D 2015, The Real Reasons Google Glass Failed, *The Threadling*, viewed 21 March, 2016, <http://threadling.com/product-marketing-google-glass/>.

Kung, S and Onken, L 2014, TortoiseSVN.

Phillips, K and Piekarski, W 2005, Possession techniques for interaction in real-time strategy augmented reality games, *Proc. ACM SIGCHI International Conference on Advances in computer entertainment technology*, p. 2.

Pollmann, F, Wenig, D, Picklum, M and Malaka, R 2013, Evaluation of interaction methods for a real-time augmented reality game, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 120–125.

Silvester, N 2014, Sony's Free Augmented Reality Games for the PS Vita, *About.com*, viewed 30 March, 2016, <http://psp.about.com/od/Series-and-Top-Picks/tp/Ar-Games-For-The-Ps-Vita.htm>.