iOS game using real time algorithms University of Coimbra Mestrado em Engenharia Informática

Francisco Manuel da Silva Ferreira
Number: 2006124182

12/07/2012

Contents

1 Introduction

1.1 Resume

This thesis started as the development of a game incorporated a solution which allowed for real time head tracking a mobile devices. Namely the iOS family composed by the iPod, iPhone and iPad. It would allow the user to experience a 3D sensation, as the head tracking would be used to plot the view port's position in a 3D space. But in the middle of the development, Apple released it's own solution which was native to the iOS environment and was highly optimized.

Due to this issue, the project deliverables changed to the development of a game which would capture objects from the camera and transport them into a virtual world, where physics could be applied to them. This re-used some of the work already performed in the first semester, namely the blob detection system. A new research had to be done on how to transform these objects into polygons and how to map them into a physics engine.

In the end it was almost as doing two separate research thesis comprised into one.

1.2 Keywords

Mobile programming, Mobile image processing, Head tracking algorithms, Blob detection, Mobile games with camera, polygon construction algorithms

1.3 Overview

The original purpose of this thesis was to develop a new kind of game, unique to the iOS environment, using head tracking for a simulated 3D experience. The game would be a 3D game which would track the user's head to change the view-port's position and give the user the feeling that he was looking through a window into a new world. This would be achieved through a real-time head tracking system fast enough to run in a small processing device while still saving resources to make a decent 3D render to offer a good gaming experience to the user. The user's head would be tracked so we could update the 3D world to match the view angle with the device, allowing the user to look from the sides, up, bottom, closer and further away, giving him an unique experience.

As better explained in the section Update to the project during the development of this thesis, Apple released a native solution which would allow, without any effort, to perform real time head tracking natively in any iOS 5.0 device, thus making this

thesis focus obsolete. Although this task could be easily achieved on a traditional computer, under this environment, it was a hard objective because of hardware limitations. The resources would have to be shared between the rendering engine and the head tracking, meaning that the algorithms involved would have to be extremely fast to achieve playable frames per second.

The reason why the iOS environment is targeted by this thesis, is because it is the only one with guaranteed hardware quality. Other platforms (i.e.: Android devices) suffer from a high fragmentation of processing power in their hardware, while on the iOS it is guaranteed that all users will be able to use the application due to hardware minimum specifications.

The original requirements of fast processing and optimized algorithms were kept after the change in the iOS api, and this thesis is now split into two parts. These parts represent the first and second semester. The first sums up the research about real time head tracking systems, while the second part involves a completely new research in order to develop a different, yet challenging, type of game.

1.4 Motivation

My personal motivation was to increase my skill as a developer and engineer, bring innovative content to the iOS environment and if possible making a fun game known worldwide while at it. I've always been interested in video game development and fast performing algorithms, thus my deep interest in this thesis subject.

Due to the speed technology evolves, what was innovative in the beginning of this thesis, was already easily accessibly in the middle of it. Nevertheless my interest in developing something unique for the iOS environment kept me pushing through, ending up in adapting a project whose prototype had already won a contest at Sapo Codebits and which I've always wanted to develop for the iOS.

I knew that the complexity to develop this new project was very high, but I didn't wanted to make something that was now easy, ending up in choosing to change the project to be able to deliver unique content in the iOS platform. Having always enjoyed algorithms and code sprints this was perceived as a challenge.

1.5 Research Problem

Normally the biggest problem when doing development targeting mobile devices is the limited processing power [5] they possess. This leads to a big secrecy among

developers of mobile platforms to ensure competitive advantages to their companies. This context leads to a lack of libraries available to use in projects which include image processing. The only freely available library is the well known Open Computer Vision (OpenCV) from IBM which can be ported to the iOS environment.

The mobile devices targeted by this thesis include the iPhone 4 which has a 800 MHz ARM Cortex-A8 processor. This device's processor was set as the minimum requirements for the project.

Initially the biggest problem was building a system which allowed for real time head tracking to be achieved while saving resources to do 3D rendering in the mobile devices environment, which is characterized by having slow processors, when compared to a normal computer [5]. If not optimized both tasks would consume too much processing, therefore a balance between both of them had to be found. For the head-tracking, the OpenCV library included a 'Haar-like head tracking' API which was used during the first stage of the semester, but which wasn't the optimal solution. A solution was researched and found to have an acceptable performance while combined with 3D rendering, before the thesis changed deliverables.

After the change, presented in section Update to the project, the research problem continued to be the same making algorithms fast enough to process in a mobile environment, but now implied the implementation of multiple auxiliary algorithms described in section Applied algorithms.

1.6 Goals

There were two main goals for this thesis, one of them, as referred in the Research Problem, was to write algorithms with low complexity therefore with fast image processing in mobile devices. The other one was to make a game which is fun to play and unique in the Apple store and in the mobile devices world.

The Apple store is the biggest game retail service in the world, being solely aimed, and exclusive, for Apple hardware (iPhone, iPad, iPod or the Macintosh computers).

There was a set of secondary objectives which are personal achievements. But due to the change in the thesis, the project couldn't be refined to be released and will only be achievable later after this thesis is complete and the projects is further tweaked. These personal objectives were:

- Having over 4 stars average ranting with at least 1000 ratings – This is the first secondary objective, having an average of 4 stars with one thousand ratings

means that the game is really good and the users actually appreciate it.

- Being featured in the Apple store main page – It is incredibly hard given the amount of applications that are submitted to the Appstore every day. Achieving a feature by apple is a guarantee of the quality of a game.
- Achieve over 100.000 downloads – This objective is very hard to achieve, having 100.000 downloads will be like a dream come true.
- Have the game known as a "mini"-milestone in game development innovation – Having the game mark a milestone, which would show other developers/teams that this kind of games is possible. Would be the cherry on top of the cake.

1.7 Development environment

Unlike most software pieces, where the developer can focus only on the software, and expect the hardware to be capable of supporting it, in the particular scenario of this application, where the hardware is very limited, one must also understand the limitations of the hardware so we can bring the application to its full potential. This means that the developer needs to address a balance between the two aspects, the software and the hardware.

1.7.1 Hardware

At table 1 you can find the differences between the target devices. The reason why the Apple's platform was chosen is because the main features required, processing power and camera resolution, don't diverge much from device to device. This allows the development to focus its full attention in the development of the game and the challenges it provides, instead of having to worry with compatibility issues.

The table 1 illustrates this:

Table 1: Hardware description

	iPod 4th gen	iPhone 4	iPhone 4S	iPad 2	new iPad
Display	640x960	640x960	640x960	768x1024	2048x1536
Memory	256Mb	512Mb	512Mb	512Mb	1GB
CPU Clock	800 MHz	800 MHz	800 MHz	1200 MHz	1000MHz
Cores	1	1	2	2	4
Camera	Yes	Yes	Yes	Yes	Yes

As can be seen the specifications for all the devices hardly changes, making the hardware very stable and predictable. One can easily say that what will work for

one will most likely work on all. It is specially important that the clock doesn't vary much. All the algorithms written in these two projects are single threaded, making use of a single core, so the clock speed is very important.

This specification stability gives extra security for the applications, should the development hardware be the android platform where, for example, the display resolution is not standardized and processor clock varies a lot. This would lead to only a very small segment of the users targeted as possible players.

1.7.2 Software

This thesis was done in Mac OSX operative system, while the software development environment was the Xcode IDE, which allows native development of Objective-C. This was not a personal choice, but rather the only solution to be fully compatible with the devices targeted by the thesis.

In this IDE, there is available a wide range of profilers, debugging tools, built in code completing, code colouring,etc. as any other IDE also has, but completely integrated and compatible with the iPhone/iPad. The main advantage of Xcode is that it automatically loads the certificates needed to deploy the application, and can also directly build and run/debug in the mobile devices without having to hack into them as it happens in other platforms.

2 Development plan

The development plan for the first semester was very linear and without any surprises. It started with a research to establish the state of the art, followed by two months of implementation and experimentation with head tracking algorithms. After a good solution was achieved, a prototype was implemented and the algorithms where tuned to have a better performance. The first semester plan culminated in writing an intermediate report and presenting the work done so far to the jury.

In the Figure 1 it is shown detailed information about these activities, representing the dependencies and time frames where each of the tasks had to be completed. Followed by a description of each of them.

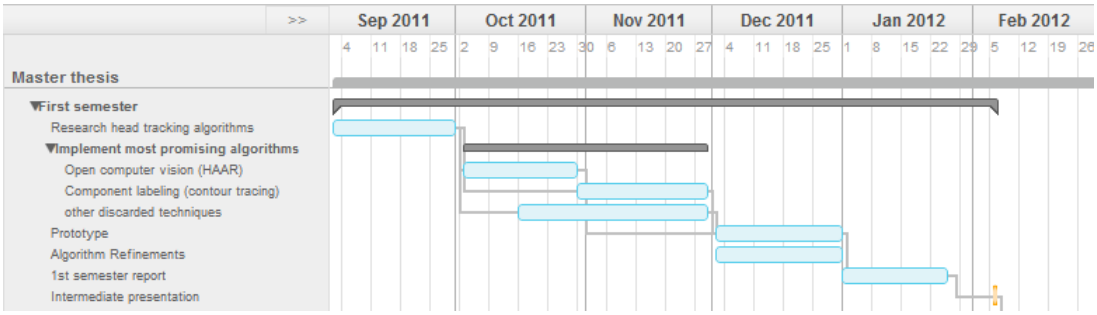


Figure 1: First semester's plan

Each task individually explained:

- Research head tracking algorithms - This consisted in building a state of the art, targeting the iOS environment, about head tracking systems. During this phase there was no available API to do it natively on the iOS.
- Implement most promising algorithms - This step consisted in the implementation and analysis of the best solutions found during the initial research.
  - Open Computer Vision - It consumed a good amount of time to properly build Open Computer Vision in XCode, which wasn't a direct build, and at the same time it was needed to learn how the library worked.
  - Component labelling - Running out of ways to do it, this iteration was an attempt to innovate and come up with a new way to do the head tracking without consuming many resources.

  - other - Other techniques where found which weren't adaptable or simply were technologically impossible to implement.

- Prototype - A prototype on a 3D environment had to be built to be sure that the 3D positioning of the head actually gave the '3D perspective' feeling.

- Algorithm Refinements - During the construction of the prototype the selected algorithms where also refined and combined to achieve better results.

This plan worked out perfectly and there was a viable demonstration during the intermediate thesis presentation. During the second semester the plan was to start with the final development of the game with the selected head tracking algorithms. This was when apple released the new version of it's operative system iOS 5.0, which included a head tracking API, built natively in the operative system and supported with hardware acceleration. This lead this thesis to change scope, trying to re-use some of the work done, but mostly starting from scrach. More information about this change can be found in section: Update to the project. Due to the update to the project's scope, this thesis became the sum of two separate thesis which share a common technological base.

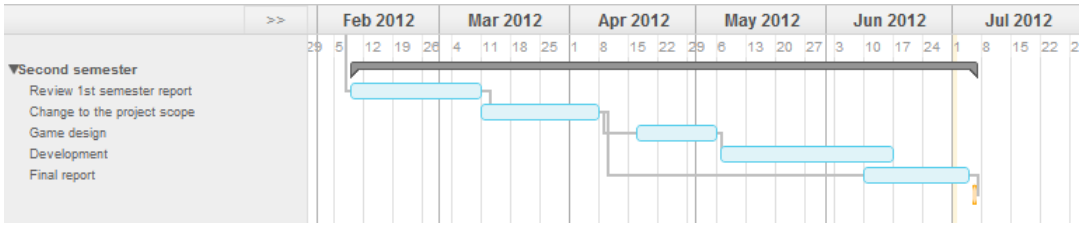


Figure 2: Second semester's plan

Figure 2 represents the work planning and structure during the second semester. This gantt chart doesn't describe what was originally planned, but instead it describes the new planning done during the change to the project's scope.

- Review first semester report - The first step, after the intermediate presentation, was to review de feedback from the juri, analyse how to improve the thesis and what needed to be changed. This stage was focused on planning and making preparations to start the implementation.

- Change to project scope - During this period the project's libraries where updated to iOS 5.0, and the problem found in section: Update to the project

emerged. During this time, tests where made to isolate the problem and check if the new solution provided by apple was better than the one being developed. This lead to a change in the deliverables of the thesis as well as a planning change. This period took almost a full month because a new state of the art had to be researched.

- Game design - This period was reserved to conceptualize how the new game would work and which rules would be available. It was also used to start doing tests to speed up the development processes.
- Development - The development consisted in implementing the core of the game so it could be usable.

3 State of the Art

This state of the art is a description of the results from a research about head tracking algorithms and subsequent experimentation and analysis. It includes the best solutions found which work in the iOS environment as well as other illustrative solutions which are very good hacks or adapt well in other situations.

3.1 Open Computer Graphics (HAAR algorithms)

Haar like features are used to detect elements through the use of objects instead of analysing raw pixels. The idea behind the use of Haar-like features as input to a learning algorithm instead of feeding raw pixels directly into it is that raw pixels represent small amounts of knowledge about the environment [1]. While features usually contain associated knowledge about what they represent, Haar-like features are used to map an image into an object, allowing for easier identification of objects. Each Haar feature contains a small amount of information, but when clustered together they can be combined into an input learned by the AdaBoost system [1].

It was proposed by P.Viola and M. Jones to use a haar classifier, namely AdaBoost [1], trained with a specific set of learning examples, to detect objects in images [2] [3]. In the case of complex objects, like a face, it becomes hard to find a good general detection strategy which can match a high percentage of faces [3]. Especially when considering that one of the biggest problems is to negate the noise in the image, often the target faces present themselves with an angle that needs to be compensated, or surrounded by colours that match the limits of the face, making it harder to isolate.

Having to compensate for noise and low processing speeds of a mobile device, arises the problem that for real time face detection to occur, the image must be scaled down, which in turn, increases the probability of error. This scaling down of the image also brings an advantage namely the reduction of noise, which could make the algorithm lose time with candidates that should be ignored at first. As in our system a real time feed was used, it was easy to compensate for a failed frame, it would just have been ignored and the detection would had returned the values for the previous successful detection. This ignoring strategy, can lead to "jumpiness" but most likely won't be noticed by the user.

For this thesis the Haar classifier of OpenCV was tested with a different set of platforms and resolutions. What is shown in the table 2 is the number of frames per second achieved while doing real time face detection with each of the different resolutions provided by the camera API. The values shown are frames per second.

Table 2: OpenCV Haar classifier tests

Resolution	iPod/iPhone 4	iPad 2	iPhone 4S
640x480	$[0.1, 1]$	$[2, 6]$	$[1, 5]$
480x360	$[7, 8]$	$[11, 13]$	$[11, 12]$
192x144	$[14, 15]$	$[20, 25]$	$[20, 25]$

This shows us that the OpenCV haar classifier can be used to make real time face detection, although there is a major problem which is that we still need to calculate the positioning of the persons face in relation to our virtual world, in order to calculate the distances. To calculate the distance of a person's face and the viewing angle, one must first understand what is returned and how to we can use it. The Haar-like features algorithms, where openCV is included, return an area, characterized by a rectangle containing the target object (in our case a face). If multiple faces are detected, as it can happen due to noise or actually having multiple faces in the picture, the biggest (closest) one is returned and we use that one, discarding the others [4].

The concept applied here is simple arithmetic's, first the centroid of the image is calculated to get an exact reference under a two dimensional plot; afterwards, the area of the detected rectangle is calculated. And if this area is big then it means the user is close to the screen, as it gets smaller the distance increases.

$$distancemultiplier = \frac{1}{plottedarea}$$

After the three coordinates are calculated, the algorithm would then proceed to update the camera coordinates, giving the user the effect of looking through a window, and being able to look from side to side.

The main issues related to the use of this approach are:

- The image is scaled down to a low resolution, so we can make a real time detection. This will:
  - limit the maximum distance that the user can be from the screen.
  - limit quality of the detection, meaning that if the user moves in small distance steps, the area generated will be equal or nearly the same size as the previous detection.
  - If there are extreme light sources the detail loss will make the detection fail.

- It won't work for all faces though. Having a non-symmetric face will lower the probability of detection, using the Haar-like features for head tracking [3] multiple training environments would be required to guarantee more faces to be detected [1].
- Dark coloured skin tones won't be detected.

To be able to use this method, openCV was selected because of it's robust implementation. Although it was not prepared for the Objective-C environment, a C++ port was found and edited a bit to be able to execute properly without leaks or sudden segmentation faults.

In conclusion:

OpenCV with haar-like feature detection is a great candidate to be used as the main algorithm behind the head tracking system. It's commonly used as a head-tracking system [3] [13] [14], the biggest problem is that if we want detection quality sometimes it is too slow for the real time requirements of the head-tracking application, under the limited processing environment.

3.2 Component labelling with contour tracing

Another set of tests performed was an experiment. Using a linear time contour tracing technique on a threshold image we can easily find multiple blobs [11]. Having the premise that we have a face in front of the camera, and knowing that face's coordinates we can afterwards do quick blob scans to detect the current position of the head.

Some of the best algorithms for blob detection have a linear time complexity [11]. This is very good to match against a fast stream of images, as this will keep the amount of processing needed stable.

The algorithm described in 'A linear-time component-labelling algorithm using contour tracing technique' [11] was implemented from scratch for the iOS in objective-c for this thesis and it works remarkably fast but with some limitations.

This algorithm allows us to map blobs and label them independently of each other [11]. Although we don't know exactly what they are, my expectations of this algorithm is that when used in an alternated fashion with the Haar-like features which can give us a set of previous coordinates we can make fast detections of where the face is.

The concept of this algorithm is that if the image is threshold into a black and white image we will get with a very good distinction of the contours of objects. We can linearly go through all pixels once, and every time we find a contour candidate we just follow the contour until we are back at the initial position, to get the map of the blob [11]. We can then label it and store it for later processing. A second step is the tracing process, were we are forced to go through all inner pixels of the blob, meaning that the area is automatically calculated during the trace function. This will lower the needed mathematics at a later stage, meaning that we won't need to process extra square roots and divisions on all blobs, which is computational intensive, to calculate areas. A more formal description of this algorithm can be found in section: Blob detection.

The table 3 shows the performance achieved when tested against the various target platforms with different video resolutions, all values represent frames per second:

Table 3: Blob detection in real time

Resolution	iPod/iPhone 4	iPad 2	iPhone 4S
640x480	$[4, 5]$	$[8, 9]$	$[7, 8]$
480x360	$[7, 10]$	$[20, 25]$	$[15, 20]$
192x144	$[15, 20]$	$[40, 60]$	$[30, 50]$

As shown in this example this technique is much faster than the openCV haar-like features detection, but will not do work on its own, requiring to have an auxiliary detection system to adjust the values detected.

A proposal on how to integrate this process is to first use another (slower) technique and store the previous position of the head in the image. Afterwards, search for the biggest blob near the centroid. This means that blobs within blobs would either be ignored or add the total area detected, the adding would be justified because of the eyes, nose and mouth which can generate inner blobs and be part of the main face blob.

The main problem with this approach is, the fact that we are not detecting the face itself but rather a 'shadow' or a 'footprint' of it, making the algorithm dependent on a previous successful detection to properly identify it.

Other problems that directly relate to the algorithm are, once again, the dependency in the amount of specular light. Clear shadows will automatically remove valid contours of blob candidates.

3.3 WiiMote head tracking

Head tracking can also be performed using a Wiimote hack. This requires a Wii controller which has an integrated infra-red camera and two infra-red LEDs around the head to track the positioning and distance of the user [15]. Although this isn't directly associated to our scenario, constrained to the iOS environment, it shows us that head tracking doesn't solely need to rely on computational intensive algorithms.

The concept applied is that it is very easy to track blobs of light and with them we can get the correct position for the user's head. Although impossible to do with a single camera and a single blob, this concept lead to the idea of combining both OpenCV and the blob detection algorithms.

3.4 i3D for iphone

There is also an experimental iPhone application called i3D which applies a technique that combines OpenCV Haar face detection with an extended Camshift tracking algorithm [14] to perform real time face tracking. In this case, their aim is the same of this thesis, which is to be able to do a head tracking algorithm working in real time and still have processing power for the auxiliary tasks.

This application is a Human Computer Interaction (HCI) demo and their paper covers most of the HCI potential of the application while leaving vague notes about the implementation itself. They do note a set of problems that have also been encountered during the tests of my implementations [14].

Using the technique of calculating the area of the detected face to plot the distance from the screen, they assume that the user is at most at a distance of 30 centimetres from the screen, which compensates for half detected faces [14]. They also assume that the user is always in front of a neutral white background, with specular light pointing directly at the user. These constraints easily boost the detection, but leave a poor margin for actually using the application in uncontrolled environments.

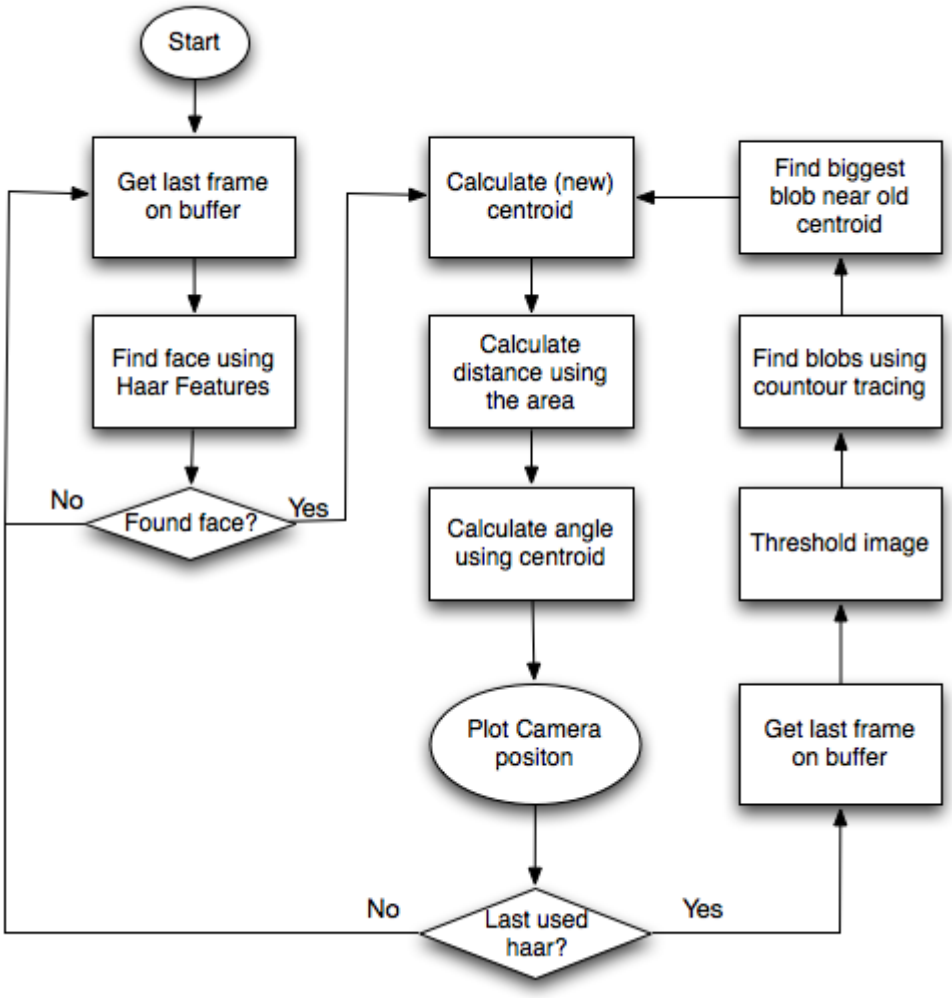
On the other hand, the conclusion of their HCI work is that the users increasingly desire applications that incorporate this technology. And the overall questionnaire response was that they would easily and in a natural way adapt to this environment. This questionnaire was answered by participants ranging from 12 to 83 years of age [14].

4 System Design

4.1 Algorithm to be used

The hardest part of the implementation of this game will be the algorithm for the head-tracking system which must be lightweight enough to run in real time and still save resources for the rendering engine to perform its tasks.

Figure 3: Solution's flow chart



Therefore, it was implemented an algorithm which alternates between the Algorithms described in section 3.1 and 3.2. This will allow the system to take the best of both, on one side the pure speed but sometimes erratic behaviour of the blob detection, compensated with the high quality of the OpenCV Haar detection.

In the figure 3 the structure of the algorithm is described. This was the mock-up developed for testing purposes and will most likely have changes during the end development process. This algorithm is explained in detail as follows:

- Whenever there is no head detected, the Haar kicks in and tries to detect a head in the picture.
- Once a head has been detected a centroid and an area is calculated. This centroid is then stored to be re-used by the blob detection system.
- While the new head is successful the blob detection system is used:
  - The image is threshold to transform it into a black and white image to trace
  - The contour tracing algorithm is used and detects the blobs in the image.
  - The blobs are then sorted by size and to the top 20%:
    - The average distance of all points to the old centroid is calculated
    - The centroid of this blob is calculated
    - If a blob contains multiple edges of the image, it is discarded
    - Else it is added to a new list
  - This new list is then sorted in relation to:
    - (The average distance to the old centroid + blob centroid) / 2
    - The first element of the list is our most valid candidate and we use that one to:
      - Replace the old centroid with this centroid
      - Use the centroid + the area to calculate the new camera position
  - If no valid blob is detected a "HeadNotFound" is thrown and the algorithm defaults to the OpenCV Haar detection, and this frame is discarded.
- If the blob system was used more than 10 times, it also defaults to the OpenCV detection, and the frame is discarded

The astute reader is probably wondering why the frames are discarded. This is because the iPhone streams a new image from the camera 30 times per second (once every 0.03 seconds) [16]. This happens on another thread meaning that it is independent from the current processing lock caused by this algorithm [16]. As it was shown on the tests made to the two algorithms which backbone this one, both are much slower than 0.03 seconds, meaning that if for any reason the algorithm fails, there is always a new updated frame waiting to be processed, which would most likely be dropped should we repeat the process on the same frame.

It should also be noted that we will use the preset 'low' for the face detection. This means that the average frames per second should be an average from both algorithms will be analysing images of size 192x144 and will have an expected processing speed of at least the values shown in table 4:

Table 4: Combined processing times

Resolution	iPod/iPhone 4	iPad 2	iPhone 4S
Before render	$[15, 18]$	$[40, 50]$	$[30, 40]$
After remder	$[20, 21]$	$[30, 31]$	$[24, 25]$

4.2 First semester's completed objectives

Going through each of the first semester's goals, it will now explained how they were achieved:

4.2.1 Research head tracking algorithms and libraries

The initial research about libs that would allow us to do what was required was a bit of a dead end. The only real solution found was the Open Computer Vision which had to be adapted to run properly on the iOS platform. As there weren't much choices we turned our attention to the algorithms themselves, and also found that there weren't many alternatives to do what was intended. One of the algorithms that worked was already included into the OpenCV lib and this were good news. Actually, not finding a solution that left enough computing power available was positive as it opened the way to devise a new solution which works much more effectivelly in this constrained environment.

4.2.2 Implementation of the most promising algorithms

The implementations where successfully completed, although they had lots of problems. To integrate the iOS video streaming system with the algorithms, lots of hacking through the system to access the data was required and this was quite a challenge. But in the end they were effectively working and it was rewarding to see them run.

Having successfully implemented them, we keept optimizing the code so they could run faster. Due to this, and the long testing devoted to this software components, at the moment we can ensure that the solution is very stable, consistent and robust.

4.2.3 Test the implemented algorithms in real devices

Originally this game was to run on the iPad 2 which is the most powerful device of the platform. The iPhone 4S which was released in the end of last year, was also a very good surprise, because it was processing as fast as the iPad 2 and very good results were being achieved on it. After having successfully tested the application on it, and wishing to go further, we tested the code in all major iOS devices with positive results (iPad 2, iPhone 4S, iPhone 4 and iPod touch 4th gen).

5 Second phase

5.1 Update to the project

After the first stage of this work (in the first semester) Apple released the IOS 5.0 and IOS 5.1 which includes a new face detection API [6]. This release made the face detection very simple to implement. Using the new API we can quickly detect the sizes and positions of the face, mouth and eyes, with adjustable levels of accuracy. This API update can be used in situations demanding high speed, turning our optimizations obsolete.

This meant that this Master thesis focus needed to be changed, so after some brainstorming and with the jury approval, we decided to proceed with one of the algorithms presented in the first phase of this project, the blob detection. The new application will not include head tracking, but will in turn use real-time blob detection to track elements drawn by the user in flat surface such as a piece of paper. This new focus brought a completely new set of elements required to be researched and implemented.

5.2 New deliverables

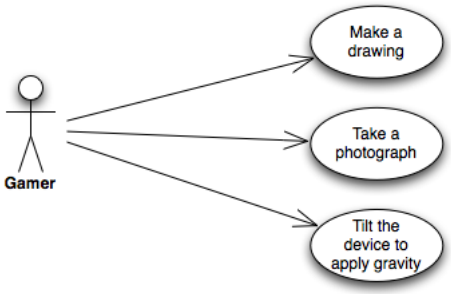
The project now aims at developing a game where the user can draw a 'world' in a paper using multiple colours. This will allow the user to express it's creativity by drawing, allowing him afterwards to apply Augmented Reality to the game by integrating the drawing into a physics engine. This presents some extra challenges not covered by the previous state of the art, which include transforming a blob into a polygon, triangulating it and the use of a physics engine.

6 Game Design

The effective usage of a persons creativity is the ultimate driver. With this in mind, this game is designed to allow the user's to express their will to construct as most as possible, giving him control over all game elements.

The game consists in making drawings in a sheet of white paper, and using the camera in the device to allow the application to transport the drawings into a virtual world where physics will be applied. This simple concept empowers the user to link reality with the virtual world, allowing him to build as he wish. After the world is constructed and mapped into the device, the user can control the gravity by tilting the device in different directions. This will make the objects drawn to change positions in the screen. To avoid that all objects have the same behaviour, different colours will apply different physical properties. For example, any item drawn in black will be a fixed object, while all others are affected by gravity forces.

Figure 4: Use cases



6.1 Specification

To properly specify the game, we can take a programmer's approach of simplification and abstraction to isolate two base elements which can be easily identified and explained. These elements are actions and objects.

6.1.1 Actions

An action is defined by an interaction with the physical environment PE. Our physical environment is composed by a paper, two or more coloured pens and the

mobile device running the game.

$$PE = [paper, blackpen, bluepen, device]$$

While an action is purely defined as the interactions which the user can perform on both virtual and real world with the device and the game engine. An object O is a game element which transits from the physical environment PE into the game's virtual world VW. An action is always dependent on the available game objects and the possible interactions with them to be performed. Thus the physical environment contains a collection of n of these objects drawn in the paper.

$$PE \Leftarrow [O_1, O_1, O_2, ..., O_n]$$

While the virtual world VW is a transformation of the PE which maps and processes all the n objects into virtual objects V_o

$$PE \Rightarrow [O_1, O_1, O_2, ..., O_n] \Rightarrow mapping \Rightarrow [V_{o_1}, V_{o_2}, ..., V_{o_n}] \Rightarrow VW$$

There exists a set of main actions which can be isolated, these consist of the building blocks for the game's interaction. In the table 6 these are briefly described.

Table 5: Action descriptions

Action	Description
Draw objects	Use a pen to draw fixed or movable objects in a paper
Focus image	Point the camera at the paper and see the objects getting selected
Capture objects	Tap the screen to capture objects
Tilt device	Tilt the device to apply gravity changes to the world

- Drawing objects - this is the most common and time consuming action in the game. It is performed by using one or more coloured pens to draw multiple objects. Using different colours which later will be identified to set the object's properties. Two different colours should not touch during drawing time unless the user wants the detector to merge the objects together.

- Focus image - Upon starting the application's detection system, the user can see in the screen the images being captured. This stream contains highlights where the objects are present. The processing is done in real time, which means that the user has time to allow the camera to auto-focus and detect all objects.

- Capture objects - By tapping the screen while at the focus process, the user instructs the application to apply all transformations to the detected objects and plot them into the physical and rendering engine.
- Tilt the device - After all objects are placed inside the physical engine, the user can physically move the device, tilting it either horizontally or vertically to change the gravity being applied. This will make the drawing animate and move, allowing the user to play with the world he has previously drawn on paper.

6.1.2 Objects

As previously explained, an action can generate or interact with multiple game objects. These objects are pieces of the actual physical drawing simulated in the physics engine and drawn in the screen of the user's mobile device. They can be affected by gravity and are internally defined as polygons. This polygon definition is explained in detail in section: Polygon construction.

There are two main types of game objects: movable objects which are affected by gravity, and fixed objects which are fixed in space and are used as walls to limit the movement of the other ones. While on paper (literately while drawn on paper) each these objects is defined by a continuous drawing always using the same colour.

An object can have any form the user desires, it can be a circle, a square, a convex or concave polygon, but it must have a minimum line width so that the camera can perceive it as being at least three pixels thick. This constraint happens because anything smaller can generate errors and might be discarded as too thin to be triangulated (more information about triangulation can be found in section: Triangulation).

Objects can be drawn solid or with holes and with multiple colours. The table bellow defines the properties of each colour and type of object.

Table 6: Action descriptions

Type	Behaviour
Solid object	Directly transformed to a polygon and filled with colour
Object with hole	Contains one or more holes, accepts other objects in these holes
Black coloured	Fixed in space (gravity immune)
Other colours	Movable, allows gravity to affect it

7 Applied algorithms

7.1 Blob detection

By definition a blob is a representation of a large object in binary. In this project a blob refers to an ordered sequence of points that define the contour of an object detected in an image. These points are extracted from a two dimension black and white image (after the threshold process is applied). The list L is a circular ordered list containing n points $P_{(x,y)}$ the point P_{n-1} connects to P_0.

The blob detection is split in two parts, a finder which searches and tags all pixels of an image, and a tracer which traces the contours of blobs. The main step of the blob detection is to find external and internal contours to be afterwards transformed into objects [11]. This detection is the main functionality for this project to work, it needs to be fast because it is shown in real time to the user(the blobs being detected). The blob detection works directly with raw images from the camera's image feed. This allows for lower memory consumption as no wrapping is used to prepare the image for processing.

An image feed from the camera consists in an array of sequential pixels in binary. Let's assume that this array is in form of list L of sequential pixels P, each of these pixels is composed by four binary values $RGBA$ for this explanation. The pixels are listed sequentially meaning that L is one dimensional and of size $w(width) * h(height) * 4$, and a simple mapping was done to access the correct pixels.

A visualization example for the sequential list:

$$L = [RBGA_{0,0}, RBGA_{0,1}, ..., RBGA_{w,0}, RBGA_{1,0}, ..., RBGA_{0,h}, RBGA_{1,h}, ..., RBGA_{w,h}]$$

Mapping for accessing each pixel and colours:

$$\begin{gathered} P_{(x,y)} = L[x + width * y] \\ Red_{(x,y)} = L[(x + width * y) + 0] \\ Green_{(x,y)} = L[(x + width * y) + 1] \\ Blue_{(x,y)} = L[(x + width * y) + 2] \\ Alpha_{(x,y)} = L[(x + width * y) + 3] \text{ (not used)} \end{gathered}$$

With this mapping the image is now traversable and all pixels are easy to access. The finder will run sequentially through L which will mean that the reverse function

of $P_{(x,y)}$ is also required and declared as a two variable equation:

$$\begin{cases} \text{x = i - w * y} \\ \text{y = (int) i/w} \end{cases} \tag{1}$$

7.1.1 Threshold

Threshold is defined by the transformation of a grey scale image into a black and white bitmap [12]. Threshold processes are often used in image processing to simplify the environment for the algorithms to take effect without having complex computations to handle the colours [12] [11].

In this application, it will be making contour detection to find blobs, which means that before starting the process, the image must be converted into a black and white image, so we can detect the contours easier, meaning that a threshold must be applied before reading each pixel. As referred a pixel $P_{(x,y)}$ is composed by three colours RGB in binary format. Each colour is represented by one full byte, varying in value from $[0, 255]$ a byte with a value of 0 is black while one with the value of 255 is the brightest possible. It is in the interest of the algorithm that image is seen uniquely with values of $(0, 0, 0)$ and $(255, 255, 255)$.

The threshold applied considers the average of the three colours, and transforms to white if the average is 70% bright. This can be described with simple mathematics:

$$T_{(x,y)} = \begin{cases} a = \frac{Red_{(x,y)} + Green_{(x,y)} + Blue_{(x,y)}}{3} & \\ 0 & \text{if } a < 178; \\ 255 & \text{if } a \geq 178; \end{cases} \tag{2}$$

The 70% threshold point was selected because the target of the game will be a sheet of white paper. The paper will be always white or greyish $T \leq 178$ making everything else a valid drawn contour to follow. With this margin, light variations such as shadows will be slightly compensated.

7.1.2 Finder

The *finder*'s job is to traverse the array of points and verify which ones are part of a contour. It is used a binary image where white pixels are considered empty and black pixels found are the beginning of the contour that will define a blob, say B. The image is traced from top to bottom and from left to right [11], as previously referenced in the *Blob Detection* section. The finder also stores and labels each pixel found with an integer label, say BL (for '**B**lob **L**abel'). White pixels are labelled as

visited ($BL = -1$), while the contours are labelled in increasing order $BL \in [1;\ 2^{31}]$. Each new blob found will use an increment of the label $BL_i = BL_{i-1} + 1$ while inner blobs, representing holes, will use the label of the parent blob, let's call an inner blob's label IBL (for '**I**nner **B**lob **L**abel'). The blobs are stored in a list L where the index of the blobs represents the order with which they were found.

So in the end, the *finder* will generate two lists, a list of blobs L_{blobs} and a a list of labels L_{labels}

$$L_{blobs} = [B_1, B_2, B_3, B_4, B_5, B_6, B_7, ..., B_n]$$

Let's assume that B_3 and B_7 are holes of B_1, while B_5 is a hole of B_2. As the indexes in each of the lists represent the matching between blob and label, it would generate a list of labels such as:

$$L_{labels} = [LB_1, LB_2, LB_1, LB_3, LB_1, LB_4, LB_2, ...]$$

Both L_{blobs} and L_{labels} are sorted by the order in which they where found. This is the same as saying the sorting comparator for L_{blobs} between two blobs Ba and Bb both composed by a list of n and m points $Pa_{i_{(x,y)}}$ and $Pb_{i_{(x,y)}}$:

Let w be the width of the image.

$$Ba > Bb \Leftrightarrow Pa_{0_{(x,y)}} > Pb_{0_{(x,y)}} \Leftrightarrow Pa_{0_x} + Pa_{0_y} * w > Pb_{0_x} + Pb_{0_y} * w$$

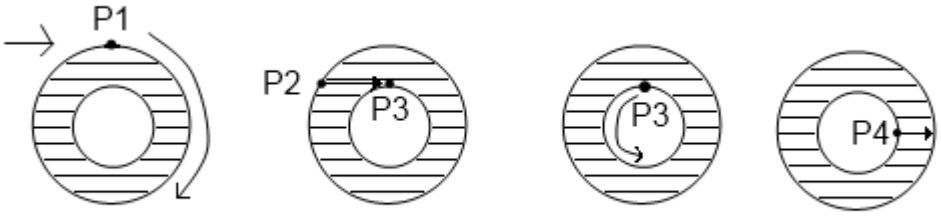
To make this happen, any white pixel found during the traversing will be labelled as visited -1 while each time the *finder* finds a non labelled black pixel P_i it will consider this as the beginning of a new contour if and only if $P_{i_(x,y+1)}$ was a white pixel and P_i is unlabelled. In this case it will increment the label LB and call the *tracer* to find the contour. The tracer will mark all points of the contour with LB. When the tracer finishes it returns a list of points with the contour found. The finder will then store this contour in L_{blobs} and the label in L_{labels}.

On the other hand, if the *finder* finds an unlabelled black pixel $P_{i_unlabelled}$ and $P_{i_(x,y+1)}$ is a labelled black pixel then the label from $P_{i_(x,y+1)}$ will be applied to P_i. Being in this state means that the algorithm is currently inside the blob and all that's needed to be done is label the pixel we are currently at. If P_i a labelled black pixel then the *finder* is currently looking at a contour point.

If the *finder* traverses over an unlabelled black point P_i and $P_{i_(x,y-1)}$ is a white pixel, then this means that an internal contour was found. The tracer will be called again to find this internal contour and label all the points with the label being assigned to P_i. The result will then be stored in L_{blobs} and the label in L_{labels}. As the internal contour is always found after the external contour. The first time any label is found in L_{labels} the matching blob will be the external contour.

In image 5 you can see the four steps described. P_1 represents a point found in the start of a blob and a trace is started. P_2 to P_3 is the filling of the blob, where the pixel above is always labelled. P_3 is then identified as the beginning of an inner blob and an inner trace is started. From P_4 to the right the top pixel is always labelled and the filling continues.

Figure 5: Finder sequence



7.1.3 Tracer

The *Tracer* is called by the *finder* to find the contour for the blobs that will later be used by the application to map the polygons. The goal of the tracer is to build the blob by finding for each point P the next point in P's neighbours that belongs to the contour. The search is done in a clockwise order, using the last found point as a reference to search the next one. For finding the next point the following matrix is used [11]:

reference matrix:

$$\begin{matrix} 5 & 6 & 7 \\ 4 & P & 0 \\ 3 & 2 & 1 \end{matrix}$$

When a point P is visited it is in one of two circumstances, a new contour is being started or the process has already begun. If a new contour is being found, a new list B is created and we know that we're in P and the pixel (6) $\Rightarrow P_{x,y-1}$ is white. So the first pixel to verify is the pixel (7) $\Rightarrow P_{x+1,y-1}$ and if this pixel is also white, we follow the search in a clockwise order through the *reference matrix* until a black pixel is found. The previous pixel is then stored in B, labelled with the label

given by the *finder* and the algorithm is now in the second state, which is when the process has already begun.

When the process has already started it is known which position was previously being visited, so if the black pixel was found at (1) $\Rightarrow P_{x+1,y+1}$ the next pixel to be searched is two positions afterwards. Saying a was the previous entry position in the *reference matrix*, the next point to follow would be $a + 2(mod8)$ [11].

When the *tracer* is at the starting position again, the *tracer* as followed through the entire blob and every point is now labelled and stored in B. The blob can then be returned to to the finder.

If it is an inner blob that is being traced, this is a special case, the starting position cannot be (7) $\Rightarrow P_{x+1,y-1}$, it must be (3) $\Rightarrow P(x, y+1)$ since it is known that (2) $\Rightarrow P(x, y+1)$ is known to be white and the next point in clockwise order is (3) $\Rightarrow P(x, y+1)$. Independently of being the inner blob, the rest of the *tracer* processes the same way as when doing the external trace.

7.2 Polygon construction

As referenced each blob is composed by all the points of the detected contour. If the blob's contour is composed of 300 points, this would lead to 298 triangles composing a single object. We wanted to limit the amount of polygons in the bodies set in the physics engine, so after the triangulation is applied we drop the total number of points to $\frac{1}{5}$ of the original vertex. The exclusion process is linear: for every 5 points the first will remain and the last 4 will be removed. The bridges injected during the triangulation are special cases that will never be removed.

7.2.1 Triangulation

In this project a crucial part is to transform the list L of points $P_{[0,n-1]}$ into triangles $\widehat{ABC}$, that can later on be injected into a physics engine. As previously referenced, L is a circular ordered list containing n points $P_{(x,y)}$ the point P_{n-1} connects to P_0. Each vertex $\widehat{abc}$ shares exactly two edges $\overline{ab}$ and $\overline{bc}$ and given the nature of the blob detection, it is guaranteed to be a simple polygon.

Given this pre-set a triangulation algorithm by Ear Clipping [7] has been implemented. Having a simple polygon of n vertexes, it is guaranteed that while the vertexes are traversed, the interior region is always to the same side, and it is a fact from computational geometry [7], that a triangulation of this kind of polygons always has $n-2$ triangles. There are multiple algorithms for triangulating polygons [8] [9], this one was chosen because it's the simplest to implement. The complexity order

is $O(n^3)$ but it is an acceptable complexity considering the low amount of vertexes per polygon, and that it will be processed only once and between scenes (no requirements for real time). It wasn't worth it for time differences smaller than one or two seconds to loose 3 or 4 times more time to implement one of the other faster triangulation strategies

The main trick in this algorithm is to interactivly extract Ears. An Ear of a polygon is a polygon itself composed by any three candidate vertices (V_1, V_2, V_3), but without having any other vertex present in L inside the triangle formed by the candidates [7]. The central vertex V_2 in $V_1\widehat{V_2}V_3$ is called the Ear tip [7]. If these three candidates successfully represent an Ear they will be stored in a polygon List and the Ear tip will be removed from L. The process is afterwards repeated until there are only three polygons left. The following pseudo code shows the logic of this algorithm. One thing that will eventually happen is that a reflex triangle will eventually be detected and these ones need to be ignored. A reflex triangle is a triangle which will contain interior vertices with angles larger than 180º. We only want to test convex triangles, which are the ones in which all interior angles are smaller than 180º.

Ear clipper pseudo algorithm

```
begin
  while lenght(L) > 3
        V1(x, y) = L0(x, y)
        V2(x, y) = L1(x, y)
        for i := 2 to n do
            V3(x, y) = Li(x, y)
            innerPoint = false
            if isReflex(V1, V2, V3)
              continue
            end
            forj := i + 1 to ndo
                                    V4(x, y) = Lj(x, y)
                                    innerPoint = innerPoint((V1, V2, V3), V4)
            end
            if innerPoint == false
              StorePolygon(V1, V2, V3)
              L.remove(V2)
              break
            end
        end
  end
```

$StorePolygon(L_0, L_1, L_2)$
end

This successfully triangulates a simple polygon. The algorithm-interested reader can easily notice that each Ear Clip has a complexity of $O(n^2)$ thus leading to a total of $O(n^3)$ complexity per polygon. This leads us to a $O(n^3)$ complexity for the full set of polygons detected. This can be easily mitigated by lowering the number of vertexes per polygon. The lowering of the detail is mostly unnoticed by the user and it also helps to produce polygons with smaller sets of triangles, thus making the Physics Engine run faster.

7.2.2 Points in triangle

To successfully detect the ears to extract there was a sub algorithm that needed to be applied. This sub algorithm is a common computer graphics problem [10]. The problem consists in giving a set of three points P_1, P_2 and P_3 that form a triangle $T(P_1, P_2, P_3)$ verify if P_4 is inside T. To solve this problem it is required some basic Algebra concepts, such as calculating the Determinant or DOT product. The determinant is required as they correspond to signed areas, thus barycentric coordinates are simply ratios of the areas inside the triangle.

To calculate if a point is inside a triangle we require three vectors all originating in the same point. Let that point be P_1 and the vectors will be $\overrightarrow{P_1, P_3}$ $\overrightarrow{P_1, P_2}$ $\overrightarrow{P_1, P_4}$. Using this vectors we require to calculate the dot products of each of them:

$$\begin{aligned} dot00 &= \overrightarrow{P_1, P_3} \cdot \overrightarrow{P_1, P_3} \\ dot01 &= \overrightarrow{P_1, P_3} \cdot \overrightarrow{P_1, P_2} \\ dot02 &= \overrightarrow{P_1, P_3} \cdot \overrightarrow{P_1, P_4} \\ dot11 &= \overrightarrow{P_1, P_2} \cdot \overrightarrow{P_1, P_2} \\ dot12 &= \overrightarrow{P_1, P_2} \cdot \overrightarrow{P_1, P_4} \end{aligned}$$

With the dot products calculated we can now proceed to calculating the barycentric coordinates:

$$\begin{aligned} inverseDenomitor &= \tfrac{1}{dot00 * dot11 - dot01 * dot01} \\ u &= (dot11 * dot02 - dot01 * dot12) * inverseDenominator \\ v &= (dot00 * dot12 - dot01 * dot02) * inverseDenominator \end{aligned}$$

Now that we have the barycentric coordinates calculated, the verification is simple, should P_4 lie inside the triangle $T(P_1, P_2, P_3)$ then u and v must both be positive and summed up together, they must be smaller than one:

$$(u \geq 0) \wedge (v \geq 0) \wedge (u + v < 1)$$

If this expression is true then P_4 is inside the triangle.

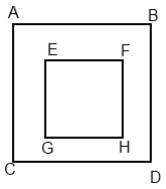
7.2.3 Polygons with holes

A major problem that needed to be solved was the problem of how to apply holes to the polygon. During the blob detection holes were already successfully detected and labelled, so it is known which holes match which outline. The polygon contains it's main body B and n number of holes listed as $H_0, H_1, ..., H_{n-1}$. All objects of both B and H have unique points in space, so therefore:

$$P_{(x,y)} \neq P'_{(x,y)}, \forall P_{(x,y)} \in B \wedge \forall P'_{(x,y)} \in H_i, \wedge 0 \leq i < n$$

With this premise we can now consider that the blob B is composed by all the points of its contour, as well as all holes $H_0, H_1, ..., H_{n-1}$ also contain all points of their inner contour. David Eberly suggests [7] for each H_i we should reverse the order of points and find two mutually visible vertices to establish a 'bridge' between these two. Merging both blobs into a single bigger blob. This will convert into B and H_i into a simple polygon containing two edges that touch themselves. These two mutually visible vertices must be duplicate in order to retain the form of the original blob. Consider $B = [A, B, C, D]$ and $H = [E, F, G, H]$

Figure 6: Two contours before connection

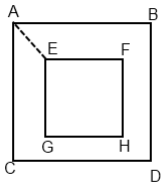


Following David Eberly's [7] approach, two mutually visible points should be found and linked together. Having B in clockwise order, H needs to be reversed in counter-clockwise, creating $H' = [H, G, F, E]$. All that's left to do is to select the points where the linking will occur. For this example A and E are selected. To link these two polygons the bridge points need to be duplicated and H' needs to be copied into B. Duplicating the original points in B will generate:

$$B' = [A, A, B, C, D] \text{ and } H'' = [E, H, G, F, E].$$
$$B' + H'' = [A, E, H, G, F, E, A, B, C, D]$$

This will result in a single blob B' composed of both B and H. If more holes existed the same process would repeat itself until all holes were added to the new main blob B'. The process suggested by David Eberly involves casting rays and making multiple comparisons and vectorial mathematics to compare the vertexes and find

Figure 7: Two contours connection



the proper mutually visible ones. The ray casting is mostly due to the fact that not knowing where the other points are, the presence of edges E_i need to be verified around the vertexes VH of the hole, and follow them to their respective vertexes VB. Verifications must be made to be sure that no other vertex or edge crosses the straight line formed by $\overline{VH, VB}$.

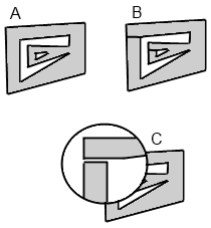
In this implementation however,there is access to all points of the contour in their raw form, as vertexes point by point. Therefore a simpler way to do this without having to recur to complex calculations that could slow down the processing time. Let's assume that B contains all points of it's contour, and it is a clockwise ordered list, while H also contains all points and is a counter clockwise ordered list. Given the nature of the blob detector we can assume that $H_{0_{(x,y)}}$ is the top left point of the blob. Knowing this, it is guaranteed that if we follow a line to the left we will find a point $B_{?_{(x,y)}}$ which represents a visible vertex from $H_{0_{(x,y)}}$.

Considering there are two holes H_0 and H_1, each of these holes is a blob represented as a counter clock wise list, and the order of the holes themselves is the order of the capture by the tracer (sort order comparator value: $v = P_x + P_y * w$), and a blob B where the holes should be merged. In the implementation, first we extract $H_{0_{(x,y)}}$ and iterate over all points $P_{i_{(x,y)}} \forall B, 0 \leq i < size(B)$. Extract into an auxiliary list L all points that match the condition $P_x = H_{0_x} \wedge P_y < H_{0_y}$. Once L is populated it is now guaranteed that the point in L closest to P is a mutually visible vertex. When at this step, P can now be duplicated in B and also push a copy of H_{n-1} to H_0. Afterwards, $B' = [P_0, ..., P_i, H_0, H_1, ...H_n - 1, P_i, P_i + 1, ...P_{n-1}]$ can be generated and replace B by B' getting $B = B'$ so we can repeat this process for H_1.

Now that H_1 is going to be merged, the information regarding where H_0 was merged or even if there was a previous merge is irrelevant. From the point of view of the *merger* it is simply merging a blob with a hole and the closest point in x to $H_{1_{(x,y)}}$ is for sure going to be a mutually visible vertex and a valid point to merge H_1 with

B. The following picture shows a highlight of how the blob will look like after the merge of two holes is complete.

Figure 8: hole merging detail



In A the blob with two holes can be seen. This example was drawn to show that the order of merging matters, we must always merge first holes closest to $(0, 0)$ (top left corner).

In B it is shown the blob after the merge. The lines that now cut through the grey part of the main blob, are edges that connect the main body and the holes. It is visible that the inner hole is connected to the outer hole. Although when this happens the outer hole is already part of the main blob.

In C an example of a merge section is displayed. The gap is widely exaggerated to facilitate the visualisation of how the connection will look like after the merge has occurred.

8 Conclusions

A game's core has been developed but it still needs some extra tweaking specially in terms of artwork to be released into the public. It works in real time and the algorithms that handle both the head tracking (for the first semester) and the image mapping (second semester) were successfully implemented.

The change in deliverables made the full thesis much harder to complete, because in reality, changing the deliverables in the middle of the project meant almost doing a full second thesis work. Yet this was achieved and the implementation works, which means that there is a lot of unexplored ground to work, develop and innovate in the mobile development world.

After this what's missing is artwork for the game and visual design, which will require some more time to develop. But still aiming at giving something new to the users, the development will continue.

This thesis was extremely challenging, but because of that also very interesting to develop. The experience gained from developing new algorithms and refining old ones is invaluable. The required balance between the thesis pedagogical goals, algorithmic complexity and computational constraints made us realize how important is the role of the engineer to seek a compromise betwen cost, requirements and technical possibilities.

9 glossary

3D Rendering - Process of producing a 2D image from 3D data such as models or scenes

API - Interface from where two sofwatere components can comunicate with one another (Application Programming Interface)

Blob - Binary large object

IDE - Software development tool used by computer programmers (Integrated Development Environment)

FPS - Frames per second

Frame - Static image which comes from a stream of images.

IOS - Mobile operating system distributed and developed by Apple.

Physics Engine - Physics Simulator. A set of bodies composed by triangles is defined in the engine and a set of forces is placed in action, the most common being gravity. The engine will then make all the calculations for collisions and general physics.

Xcode - IDE developed by Apple for OSX and IOS software development

Wii - Game console developed by the Nintendo Corporation

References

[1] Michael J. Jones, Paul Viola, *Face Recognition Using Boosted Local Features*, Mitsubishi Electric Research Laboratories, Technical Report, 2003

[2] Takeshi Mita, Toshimitsu Kaneko, Osamu Hori *Joint Haar-like Features for Face Detection*, Multimedia Laboratory, Corporate Research & Development Center, Toshiba Corporation, Technical Report, 2005

[3] Anjo Vahldiek, Ansgar Schneider, Stefan Schubert, Baden-Wuerttemberg *Evaluation of Optimizations for Object Tracking Feedback-Based Head-Tracking*, Computer Science Department, State University Stuttgart

[4] Banjo Pérez, Corchado J.M., Moreno M.N., Julián V., Mathieu P., Canada-Bago J., Ortega A., Fernandez Caballero A. *Highlights in Pratical Applications of Agents and Multiagent Systems* 9th International Conference on Practical Applications of Agents and Multiagent Systems, 2011

[5] Huang Jingshu, Bue Brian, Pattath Avin, Ebert David S., Thomas Krystal M. *Interactive Illustrative Rendering on Mobile Devices*, Purdue University, West Lafayette, 2007

[6] IOS Developer Library, *CIFaceFeature Class Reference*, 2011, available at http://developer.apple.com/library/ios/documentation/CoreImage/Reference/CIFaceFeature/CIFaceFeature.pdf

[7] David Eberly, *Triangulation by Ear Clipping*, Geometric Tools LCC, 2008

[8] Atul Narkhede, Dinesh Manocha *Fast Polygon Triangulation based on Seidel's Algorithm*, Department of Computer Science, UNC Chapel Hill 1995.

[9] Jeff Erickson *Non-Lecture G: Polygon Triangulation*, Department of Computer and Information Science and Engineering, University of Florida, 2006

[10] Dianne Hansford *Barycentric Coordinates, Introduction to Computer Graphics* Arizona State University, 2007

[11] Fu Chang, Chun-Jen Chen, and Chi-Jen Lu *A Linear-Time Component-Labeling Algorithm Using Contour Trancing Technique*, Institute of Information Science, Academia Sinica, Taiwan

[12] Paul L. Rosin, Tim Ellis *Image difference threshold strategies and shadow detection,* Institute Remote Sensing applications, Ispra, Italy and Centre for Information Engineering City University London

[13] Frank Loewenich, Frederic Maire *A Head-Tracker based on the Lucas-Kanade Optical Flow Algorithm*, Queensland University of Technology, Brisbane Australia, 2006

[14] Jérémie Francone, Laurence Nigay *Using the User Point of View for Interaction on Mobile Devices* Laboratoire d' Informatique de Grenoble, Université Joseph Fourier Grenoble, France, 2011

[15] Kevin Hejn, Jens Peter Rosenkvist *Headtracking using a Wiimote*, Department of Computer Science, University of Copenhagen

[16] *AV Foundation Programming Guide*, iOS development center resources, Apple