Abstract

Compositing is an artistic and scientific challenge which attempts to convincingly combine multiple visual elements into a single image. Previsualisation is a cost-effective time-saving function to visualise complex scenes before committing resources to a full-scale production. This project presents a novel previsualisation tool for compositing synthetic objects into captured photographs. The rendering takes place as augmented reality, driven by appropriate context-aware data relating a model to its environment. The novelty and added-value of this solution lies in reducing the information gap between the composited elements and in producing instantaneous visual feedback to a compositor. The project is implemented on a commercial mobile device using the latest hardware sensors and graphics software available.

Contributions & Achievements

- I designed a novel compositing system which bypasses a traditional image post-processing stage in favour of a flexible, real-time rendering approach.
- I devised a simplistic method for recreating environment shading information for use in colour-based compositing. I have named the collected data *colour priors* [pp. 19-20].
- I have extended the hemisphere lighting algorithm with device sensor data and colour priors to simulate ambient occlusion, resulting in a technique I call *hemisphere occlusion* [pp. 24-26].
- I have thoroughly investigated the capabilities of a modern mobile device and used my findings to bridge an information gap between the current state of compositing on embedded systems compared to desktop PCs.

Acknowledgements

I would like to thank my project supervisor, Colin Dalton, for being extremely invested in the progress of this project; from helping me propose the initial research topic to preparing me for the final report and presentation, and everything in between. His patience and attention has been nothing short of magnificent.

I would also like to thank my project markers, Ian Holyer, Tilo Burghardt, and Martijn Stam, for their valuable feedback on every deliverable of the research skills unit and the demonstration event. Their critical analysis helped to shape this project and identify any possible shortcomings.

Finally, a special acknowledgement to my fellow graduates, Haolin Li, Ting Xu, and Yu Zhao, whose work and progress over the summer in our weekly meetings with Colin was a great incentive to push myself harder and gauge my work amongst my peers.

Table of Contents

1. Introduction

1.1. Project Overview

Compositing is an artistic and scientific challenge which attempts to convincingly combine multiple visual elements captured or rendered in separate environments into a single image. The costly information gap between the source and target material has kept this task as a largely manual process tackled by seasoned professionals working from inferred knowledge and artistic judgement.

Previsualisation is a cost-effective and time-saving function to visualise complex scenes before committing resources to a full-scale production. It is extremely helpful to the whole film crew as a powerful means of communication and collaboration. After all, the film industry produces a mountainous series of images above all else.

With the current quality and availability of computer-generated imagery (CGI), more and more projects are using 3D models throughout the whole production pipeline, from pre-production (previsualisation) to post-production (compositing).

This project combines these two stages into a single process, producing a novel previsualisation tool for compositing synthetic objects into captured photographs, with a preview-quality finish.

The rendering takes place as augmented reality, driven by appropriate context-aware data relating a model to its environment. The novelty and added-value of this solution lies in reducing the information gap between the composited elements and in delivering instantaneous visual feedback to a compositor. The application mostly focuses on outdoor, daytime scenes due to the relative simplicity of a dual illumination environment (sunlight and daylight), compared to more complex indoor scenes with multiple unknown illumination sources.

The project is implemented on a commercial mobile device using the latest hardware sensors and graphics software available. This thesis documents the work carried out by this project from the initial research review, through to the design and development, and finally to the analysis and evaluation.

1.2. Objectives

- Develop a mobile app targeted for the Apple iPhone 4, using Xcode and the iOS 5 SDK.
- Harness the advantages and address the limitations of context-aware sensors.
- Design a system for effective multi-source data management, considering restrictions in memory and processing power.
- Investigate the current state of 2D computer graphics on embedded systems, using pixel-based image-processing techniques.
- Investigate the current state of 3D computer graphics on embedded systems, using OpenGL ES 2.0.
- Produce and adhere to a framework for colour-based compositing with context-aware information.
- Implement a flexible and interactive compositing tool, using a novel, real-time approach.

1.3. Hardware

Below is a list of the relevant technical specifications for both the mobile device used throughout the project and the laptop computer used for development and comparative rendering analysis [Section 7.2].

Apple iPhone 4[1]

Year:	2010
CPU:	1 GHz ARM Cortex-A8
GPU:	PowerVR SGX535
RAM:	512 MB
OS:	iOS 5
Sensors:	Assisted GPS, Digital Compass, Three-Axis Gyroscope
Camera:	5 MP iSight
Display:	960x640 [2] pixels (330 PPI pixel density)

MacBook Pro

Year:	2011
CPU:	2.2 GHz Intel Core i7
GPU:	AMD Radeon HD 6750M
RAM:	8 GB
OS:	Mac OS X 10.7.4

1.4. Software

Below is a list of the main software used throughout the project, accompanied by a brief description of each component's use.

iOS SDK

Developer:	Apple Inc.
Version:	5.1.1
Description:	The main set of resources for developing native iPhone applications. Used to handle the overall communication and processing on the device. For a detailed listing of the iOS frameworks used, see Appendix A.
Source:	[Apple Inc., 2012a]

Maya

Developer:	Autodesk, Inc.
Version:	2012 x64
Description:	Multi-use 3D computer graphics application. Used for model editing, OBJ exporting, UV mapping, and comparative rendering analysis [Section 7.2].

---

[1] A word on mobile devices: This project was chosen to run on iOS rather than Android due to personal preference, previous development experience, and standardisation of hardware components across common devices. This choice was not a hindrance to the project and the resulting tool can in fact be ported to any device with similar hardware and compatibility with OpenGL ES 2.0 (as is the case with the Android family), with minor platform-specific alterations.

[2] All screen resolution and image dimensions are given as *width x height*, conforming to the landscape orientation of the device.

Source: [Autodesk, Inc., 2012]

Mental Ray
Developer: NVIDIA
Version: 3.9.1.36
Description: Production-quality rendering application that utilises ray tracing. Used as a Maya plugin (*Mayatomr*) for comparative rendering analysis [Section 7.2].
Source: [NVIDIA ARC GmbH, 2012]

obj2opengl
Developer: Heiko Behrens
Version: 2009
Description: A script which converts OBJ files to C++ header files. Used to make models compatible with the OpenGL ES rendering cycle.
Source: [Behrens, 2012]

OpenGL ES
Developer: Khronos Group
Version: 2.0
Description: An API for writing graphics applications on embedded systems. Used for handling and rendering 3D graphics on the Apple iPhone 4 device.
For a detailed listing of the GLSL functions used, see Appendix B.
Source: [Khronos Group, 2012]

Photoshop
Developer: Adobe Systems
Version: 12.1 x64 (CS5 Extended)
Description: 2D graphics editing application. Used for texture editing, image processing (for testing), and image analysis (for interpreting results).
Source: [Adobe Systems Incorporated, 2012]

Solar Position Algorithm
Developer: National Renewable Energy Laboratory
Version: 2003
Description: An algorithm for calculating the solar zenith and azimuth angles, based on the date, time, and location on Earth. Used to drive the position of a directional light.
Source: [National Renewable Energy Laboratory, 2003]

Xcode
Developer: Apple Inc.
Version: 4.3.2
Description: The main IDE for developing native iPhone applications. Used to write, edit, and compile Objective-C code, manage resources, and construct the application UI. Includes the *Instruments* tool for performance analysis.
Source: [Apple Inc., 2012b]

2. Background & Context

Contextual Caveat

This project makes use of very new technologies[3] and computing concepts that are yet to be fully explored. Therefore, it does not directly continue a particular line of research or previous work, rather, it develops a new point of reference for future applications. Furthermore, many algorithms and techniques are implemented from renowned solutions, but in these cases the challenge lies in adapting them to fit a new approach.

While mobile devices carry a plethora of sensors and data, they are, for the time being, massively restricted in terms of memory and processing power. This project navigates this largely uncharted territory and attempts to find the right balance between these two extremes, always keeping a common goal in sight: real-time, context-aware compositing.

With the contextual caveat issued, this section examines the related work surrounding this project in scientific literature, mobile apps, and non-commercial technology.

2.1. Scientific Literature

In preparation for this thesis I conducted an extensive research review [Rendon Cepeda, 2012] in order to understand the significance, novelty, and added value of this project in relation to previous work in three separate areas of computing: context-awareness, 3D graphics, and image-processing. These areas are still quite broad and hence further examined in terms of photography, lighting/shading, and compositing, respectively. Below I present a summary of my most relevant findings.

2.1.1. Context-Aware Photography

Context-awareness is a branch of ubiquitous computing in which a device can both sense and react to its environment. Location and mobility are the main exponents of this field, with modern devices providing instant access to integrated sensors such as the gyroscope and accelerometer. This availability allows a device to be constantly aware of its relationship with the environment, and hence annotate data accordingly.

Context-awareness has been mostly combined with photography as metadata, for example, by geo-tagging[4] or embedding camera settings for personal reference. Though this is not uncommon to the project, there are certainly more sensors to be explored and several different ways to process data [Section 4.1].

Experiments in image manipulation through sound are presented by Ljungblad et al. [2004], where user screaming manages to distort and pixelise a portrait. Similarly, Hakansson et al. [2006] use background noise to colorise shadows. While very interesting, this research lacks focus and instead the results read like a *carte blanche* issued along with a multitude of cameras and sensors.

---

[3] For example: OpenGL ES 2.0 was introduced on the iPhone 3GS (June 2009) and the three-axis gyroscope was introduced on the iPhone 4 (June 2010).

[4] The process of adding location data - usually a place name, or geographical coordinates - to a photograph.

Photo tourism is an interesting concept from Snavely et al. [2006], where a series of similar photographs - taken roughly at the same location, with different angles - are composited into a 3D model of the area. With the use of both image-based modelling (IBM) and rendering (IBR), the system creates enough scenery to allow a virtual exploration of the area, after minimal user input. IBM and IBR are both outside the scope of this project, but this research does lead into the development of *Photosynth* [Microsoft Corporation, 2012], which is a popular mobile app that uses sensor data to guide a user into capturing better-aligned photographs for a high-quality panoramic stitch. This app's concept and design was a direct inspiration for the project's data capture phase [Section 4.2].

Lastly, Lalonde et al. [2010] provide a method for annotating photographs with their inferred camera parameters by using the sun position and sky appearance within the image as calibration targets. Their work is essentially the reverse of this project and although at least one of our goals is similar - collect all relevant data about an outdoor photograph - their implementation is much more complicated. It's a very impressive solution, but more than anything it helps highlight the added value and novelty of a context-aware solution.

2.1.2. 3D Graphics Lighting & Shading

Lighting refers to the simulation of light in a virtual environment - where this project implements ambient and directional lighting (as well as indirect hemisphere lighting). Shading refers to the process of altering the colour of a model based on its position and orientation to said lights - where this project implements ambient, diffuse, and specular shading. These two concepts are both well-known and largely standardised, so they will not be discussed any further for now [Section 5.3]. However, it is important to note that they go hand-in-hand, helping define depth, material characteristics, and the overall atmosphere of a scene. Furthermore, it is the lighting that the project is more interested in since shading is mostly defined by unique model attributes, whereas lighting constantly changes according to the relationship with the environment.

The most accurate and common way to translate a real lighting environment to a virtual one is through image-based lighting[5] (IBL). This technique uses environment mapping to project an omni-directional image representing real-world lighting information (gathered from a light probe) onto a sphere, used to simulate the lighting in the virtual world. IBL was first conceived by Paul Debevec in 1998, and he's since become the ultimate specialist on the subject, publishing new work year after year [Debevec, 2012] and specialist tools such as *HDR Shop* [USC Institute for Creative Technologies, 2012]. His results are fascinating and incredibly photorealistic, but far too processor-intensive for this project's device. Furthermore, they are founded atop high-dynamic range imaging (HDR) to best understand the pose[6] and colour attributes of all light sources within the environment, requiring pre-computed diffuse and specular maps from many source images, best suited for large-scale rendering systems rather than the simplified capabilities of a mobile device[7].

Spherical harmonics lighting is a very similar concept presented by Ramamoorthi and Hanrahan [2001], which generally has a better runtime performance (particularly because it only handles a diffuse component) but also requires large amounts of prior work in calculating lighting coefficients from light probe images

---

[5] Also known as *light-probe mapping* when created from photographs of reflective spheres.
[6] Position and orientation.
[7] The iPhone is currently capable of capturing HDR images, but this feature has not been released to developers.

Another form of IBL is by cube mapping, as explained by Alnasser and Foroosh [2006]. With their methodology, they prove that cube mapping is superior to spherical mapping - a discovery which goes well with this project's approach, considering the compatibility of OpenGL ES with the former, but not the latter.

An alternative solution for all outdoor lighting is to create a whole physically-based sky/sun model [Preetham et al., 1999; Dobashi et al., 2002]. The advantages to this solution include full control over the perceived colour of the sky and atmospheric phenomena such as scattering and haze. The results of these models are often beautiful images with highly realistic sunsets/sunrises and light shafts. However, atmospheric and meteorological data is nowhere near accurate enough to be able to simulate the real world for a given place and timestamp - which involves a large portion of this project. Furthermore, there would be a lot of photometric information lost compared to environment mapping when it comes to capturing surrounding reflections. This technique is much better suited for modelling a whole virtual landscape, rather than compositing. Halawani and Sunar [2010] present a similar solution for a virtual sky/sun model, but unfortunately their paper is not written very well and proves difficult to understand. Their equations aren't clearly explained either, specially since they have very precise constants whose origin is never clarified. A real shame because it's one of the more recent papers available on the subject.

2.1.3. Automated Compositing

Compositing is the process of combining several visual elements into a single finished shot. Effective compositing of CGI onto a camera view is very complex, since the former is rendered and the latter captured under different conditions.

However, every pixel of an image, whether captured or rendered, has a colour and intensity associated with it. Therefore, light-matching can be achieved by comparing the values between the two images and adjusting them until the right balance is found. The sensor data used to set up the virtual environment is a great start, but unlikely to be perfect. Compositing is largely an intuitive task with lots of artistic and contextual judgement required from the user. An experienced compositor would know how to manually modify one or both images to achieve harmony, but this project examines algorithms that can help automate this process.

Brightness and contrast adjustments are standard light-matching solutions examined in Section 5.6.1, based on greyscale images calculated from luma coefficients. Colour-matching is a more complex subject, tackled by Reinhard et al. [2001; 2004] with a statistical approach. Their methods are suitable for offline processing between two 2D images [2001] (as a colour-transfer algorithm) or real-time processing between rendered and captured video [2004] (as a colour-blending algorithm). These methods are also examined in Section 5.6.2

Karsch et al. [2011] produce some of the most impressive composites as a result of an IBM system with minor user input, which are not within the scope of this project. Regardless, their work is worth mentioning as the forefront of rendered-to-captured compositing - which actually involves reconstructing the photograph as a virtual environment, and then inserting the synthetic objects there. Lopez-Moreno et al. [2010] are the close runner-ups to foolproof compositing, with their approach involving object-extraction from images in order to be used as light-probes based on their silhouettes and surface normals. This is conceptually similar to Lalonde et al.'s [2011] method for inferred camera parameters [Section 2.1.1], which once again serves to emphasise the added value and novelty of a context-aware solution.

Currently, there isn't a fully automated compositing solution yet and there certainly isn't one that can handle all aspects of the job (scale, orientation, position, as well as lighting). However, there does seem to be a lot of literature explaining the elements of realistic compositing and the instant giveaways of fake images. For example, Lalonde and Efros [2007] focus on the importance of colour compatibility within the scene while Kee and Farid [2010] pay close attention to discrepancies in light direction instead. Furthermore, it's always helpful to understand the basics and refer to a general overview from both a scientific [Willis, 2007] and artistic [Wright, 2010] point of view.

2.2. Mobile Apps

The scientific literature survey covers a lot of ground, touching on many extensively-researched topics with multiple collaborators and many years of previous work behind them. However, the comparison to this project can feel inadequate at times considering the differences in resources that work as both an advantage (sensor data/capabilities) and disadvantage (memory/processing power). Therefore, a more appropriate placement for this project may be found amongst similar mobile apps in today's market.

2.2.1. Augmented Reality Compositing

CSL Sofas (Fig. 1a) is an interior decoration app for viewing a wide range of sofas in your living room. *My Wild Night With Ted* (Fig. 1b) is a entertainment app for adding the titular film character to your pictures.

Fig. 1a: *CSL Sofas* app [Apposing, 2012]

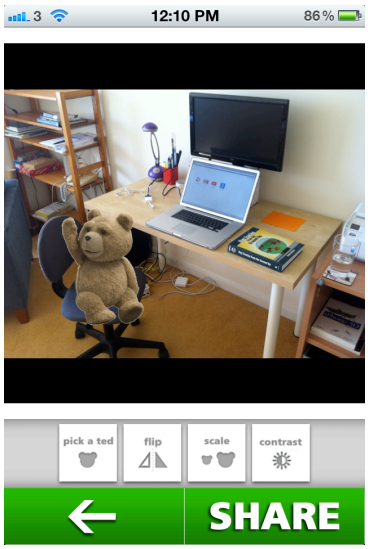


Fig. 1b: *My Wild Night With Ted* app[MEDL Mobile, 2012]

Both apps fall within the *faux*-AR (augmented reality) category since their capabilities simply extend to overlaying graphics onto a camera view (or previously captured photograph) and they don't actually gather any information from their environment. However, they are true compositing apps, albeit very basic ones. Both apps allow the user to scale and move the virtual object, whilst *Ted* goes a step further and enables a simple contrast control. They're both closed-source, but I strongly suspect they're rendered with a 2D graphics engine.

Clearly, there is plenty of room for improvement in both cases. Simply offering 3D capabilities opens up a whole new set of possible object orientations. While a contrast control is an adequate addition, this does not take into account brightness or colour levels which are equally, if not more, important. Adding a semi-automated feature to these tools would help users with little compositing experience. Finally, if a similar app to *CSL Sofas* existed for exterior or garden decorating (gazebos, benches, fountains, even plants, etc.), then the addition of context-aware sunlight would create a much more complete composite.

2.2.2. Ray Tracing

iGPUTrace (Fig. 2a) and *Quaternion Julia Raytracer* (Fig. 2b) are both graphics-intensive ray tracing applications. These apps clearly demonstrate both the rendering capabilities and performance limitations of OpenGL ES 2.0 on the iPhone.

Fig. 2a: *iGPUTrace* app [Angisoft, 2009]

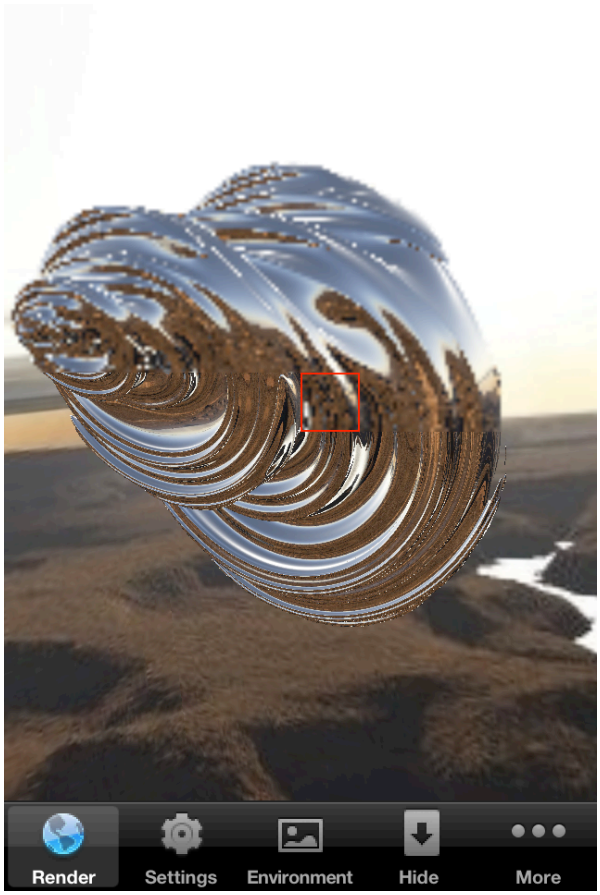


Fig. 2b: *Quaternion Julia Raytracer* app [Angisoft, 2011]

In reality, rays travels from a light source in all directions, colliding and reflecting off objects until they reach the viewer. In simulations, the process is the opposite: rays travel from the view point to the visible world, calculating the ray paths between an object and a light source. Computer graphics applications which use this method are known as *ray tracers*. Ray tracing is the most accurate rendering method in computer graphics, capable of multi-depth reflection as seen in *iGPUTrace* and self-reflection as seen in *Quaternion Julia Raytracer*. This method is also capable of rendering many other photorealistic effects such as ambient occlusion and global illumination, thoroughly covered in specialised literature [Suffern, 2007] and examined in Section 5.4.

However, ray tracing carries very expensive render times. *iGPUTrace* performs in real-time, which is a very impressive feat, but once more complex models are introduced, such as those in the *Quaternion Julia Raytracer*, the real-time rendering element is lost. Fig. 2b actually shows the app mid-render. Therefore, we can expect real-time ray tracing in the future, but for now this project must utilise alternative rendering techniques.

2.3. Non-Commercial Technology

Finally, I present a brief glimpse into the latest previsualisation work for film and augmented reality for broadcast television. This is the absolute forefront of technology in context-awareness, computer graphics, compositing, and any other areas covered in this section. These are the technologies attached to multi-million pound productions, re-shaping the industry.

2.3.1. Previsualisation in Film

Simulcam for *Avatar* (Fig. 3a) by Digital Domain Productions, Inc. [2012] and *Zviz* for *Rango* (Fig. 3b) by Industrial Light & Magic [2012] are magnificent previsualisation tools that adapt motion-capture technology to a camera. These devices move inside a physical space rigged with location sensors, thereby placing the camera inside a virtual environment. The device renders this view in real-time, and anything on set that has a CGI counterpart is rendered in the view as well. This allows a director to see his live-action actors as the virtual characters they are playing, thus being able to guide the action with instant feedback as opposed to waiting until post-production to see the final results (or using an incredible amount of imagination). In *Avatar*, for example, this allows the filmmakers to see the *Na'vi* in *Pandora* rather than just the actors playing the *Na'vi* on a set meant to be *Pandora*.

Fig. 3a: *Simulcam* on the set of *Avatar* [Digital Domain Productions, Inc., 2010]

Fig. 3b: *Zviz* on the set of *Rango* [Industrial Light & Magic, 2011]

Though clearly not within the same technology (or budget) league, this project does have a very similar goal to these previsualisation cameras: the ability to combine real and virtual elements into a unified preview-quality shot, in real-time. Their approach inserts real elements into a virtual world; my approach inserts virtual elements into the real world, more similar to...

2.3.2. Augmented Reality in Broadcast Television

The BBC's *Match of the Day* programme has a very intricate setup for AR rendering, with a system combining technologies from *Vizrt* [2012], *Deltatre* [2012], *MotionAnalysis* [2012b], and their own in-house graphics department. Unlike the previsualisation cameras for film, which aid the filmmakers, the benefits here are directed towards the viewer for a richer television experience. The graphics are very light-neutral and not intended for a realistic composite, they're simply meant to add a "fresh appeal" to the show. The technical focus of this system instead lies within real-time motion-tracking, which is outside the scope of this project. Fig. 4 shows two screen caps from a recent broadcast of the programme, highlighting the *CamTrak* technology.

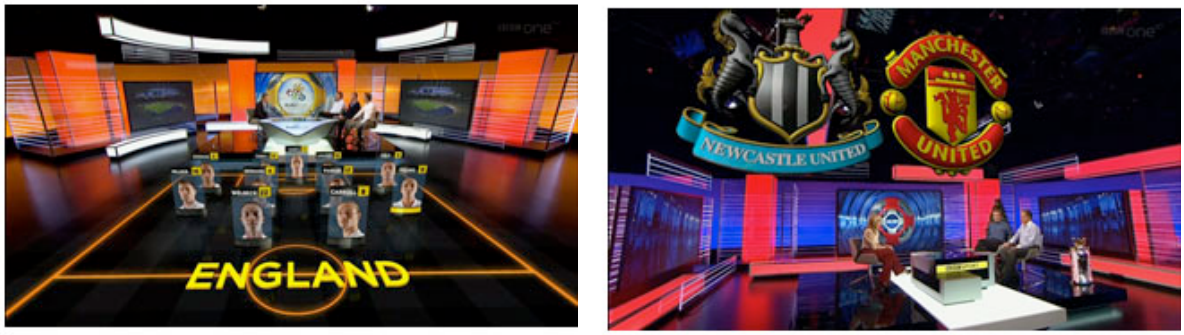


Fig. 4: Augmented reality on the BBC's *Match of the Day* [Motion Analysis, 2012a]

3. Application Environment

3.1. iOS SDK

iOS is the operating system that runs on the Apple iPhone 4. It manages the device hardware and provides the technologies required to implement native apps. The iOS SDK contains the tools and interfaces needed to develop, install, run, and test these apps. Apps communicate with the hardware through well-defined system interfaces that can be viewed as a set of layers (Fig. 5). Higher-level frameworks can be found towards the top of the stack, with lower-level frameworks towards the bottom.

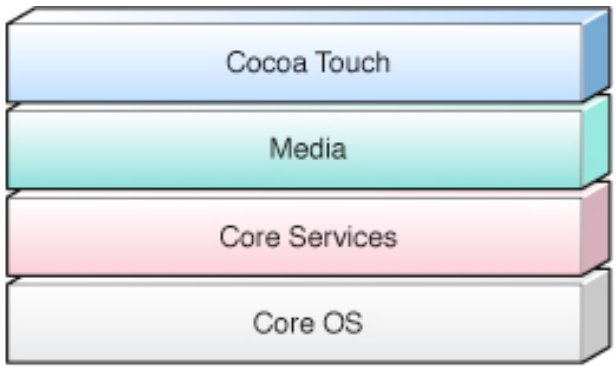


Fig. 5: Layers of iOS [Apple Inc., 2012a]

3.1.1. Cocoa Touch Layer

The Cocoa Touch layer contains the key frameworks for building iOS applications. In this layer the project sets up the majority of the user interface (UI) and navigation controls using *UIKit* and *Storyboards*. These are both high-level interfaces that handle the application layout, view controller hierarchy, and user input. They make connections between UI elements (sliders, buttons, views, etc.) and code snippets, which are then used for further processing.

3.1.2. Media Layer

The Media layer contains graphics, audio, and video technologies.

3.1.2.1. Camera

There is a basic camera class found within *UIKit*: *UIImagePickerController*. This type of camera controller is used in both AR compositing apps mentioned in Section 2.2.1, which simply overlay graphics onto a camera view or previously captured photograph. This project requires finer control over the camera, which is handled in a lower-level layer with a setup cost of four frameworks instead of one: *AVFoundation*, *CoreImage*, *CoreMedia*, and *CoreVideo*.

Fig. 6: Camera view resolutions

The key design issue here is video resolution. The camera quality level must conform to one of many presets available, ranging from 192x144 pixels to 1280x720 pixels. This project aims for preview-quality composites rendered in real-time, so there needs to be

a balance between quality and performance. With this in mind, the application uses a medium preset of 480x360, which is a quarter of the screen resolution (960x640) with 20 pixels of vertical cropping on the top and bottom of the view (Fig. 6). This is an adequate choice given the screen resolution of the previous iPhone (3GS, 480x320) and that neither the aims or objectives of the project are compromised as a result.

3.1.2.2. 2D Graphics

CoreGraphics handles native 2D image-based rendering. This framework is used to aid basic processing and analysis of cube map faces and textures.

3.1.2.3. 3D Graphics

The *OpenGLES* and *GLKit* frameworks combine to handle 3D graphics rendering on OpenGL ES 2.0. Fig. 7 depicts the OpenGL ES client-server model on iOS, where the application calls OpenGL ES functions via an OpenGL ES client. The client processes the function call and converts it into commands delivered to the OpenGL server. There are plenty of built-in function calls to the OpenGL ES API - which mostly constitute boilerplate code - so instead of a full listing of these functions, Fig. 8 describes the pseudo-code for the OpenGL ES setup within the application (iOS properties are *italicised*).

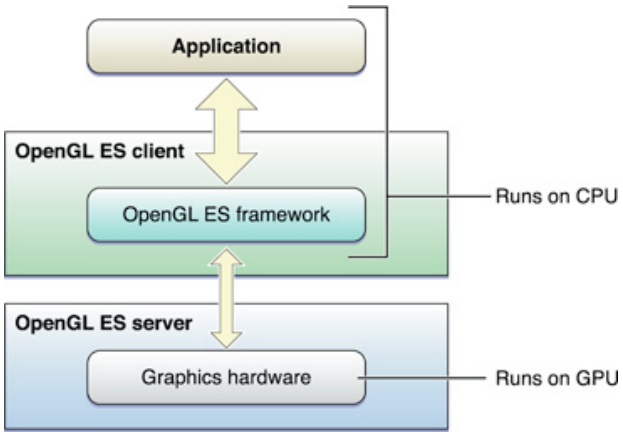


Fig. 7: OpenGL ES client-server model [Apple Inc., 2012a]

```
Procedure OPENGL ES SETUP
begin PROCESS
1:    Initialise view GLKView
2:    Set current context EAGLContext
3:    Enable depth testing
4:    Build shaders
5:    Build GLSL program
6:    Extract handles to shader attributes
7:    Extract handles to shader uniforms
8:    Load 2D texture
9:    Load cube map
10:   while (RENDERING)
11:         Clear buffers: colour, depth
12:         Load model
13:         Draw scene
14:         Update shader attributes
15:         Update shader uniforms
16:   end while
end PROCESS
```

Fig. 8: Pseudo-code for OpenGL ES Setup

3.1.3. Core Services Layer

The Core Services layer contains fundamental system services. It includes the *Foundation* framework which defines a base of Objective-C classes providing a set of useful primitives.

3.1.3.1. Compass

The digital compass is part of the *CoreLocation* framework, providing heading information to the application. It's very easy to set up and includes its own manager to handle event updates. The key design decision here is accuracy. Like the camera, the compass needs to find a balance between composite quality and real-time performance. The compass calculates the viewer orientation (azimuth angle) relative to the sun and the virtual environment [Sections 5.3.3, 5.3.4, and 5.5.2], but is a huge drain on resources. Therefore, the compass uses *magnetic* heading updates (as opposed to *true* heading updates, which are GPS-assisted) with a one-degree filter.

3.1.3.2. GPS

The GPS is also part of the *CoreLocation* framework, providing location information to the application. Like the gyroscope, it is very easy to set up and includes it own manager to handle event updates. The key design decision here is, once again, accuracy. As we will see in Section 4.1, accurate physical motion tracking (translation in 3D space) via sensors is not yet within the reach of mobile devices and hence only used in the application to calculate the device position relative to the sun. The accuracy presets range from navigation-performance to three kilometres; this application uses the 10 metre preset (which includes changes in altitude).

3.1.3.3. Gyroscope

The gyroscope is part of the *CoreMotion* framework, providing motion-based data from the device. Unlike the location services, it requires a self-declared manager to handle event updates. For this, I have chosen to deploy a timer firing every 0.1 seconds. It's quite a drain on resources but a focal point of the application, driving the same data as the compass (as the zenith angle) as well as the hemisphere shading [Section 5.4].

3.2. OpenGL ES

OpenGL ES is a subset of the desktop OpenGL API for rendering graphics on embedded systems, creating a powerful low-level interface between software and graphics acceleration. OpenGL ES 1.1 uses a fixed function pipeline (Fig. 9a) which is easier to handle, but has many restrictions. OpenGL ES 2.0 uses a programmable pipeline (Fig. 9b) which leverages the power of shaders.

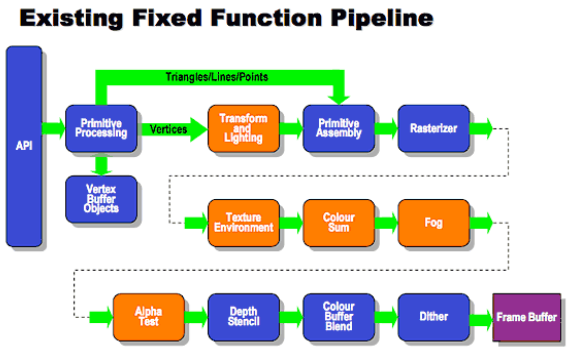


Fig. 9a: Open GL ES 1.1 fixed function pipeline [Khronos Group, 2012]

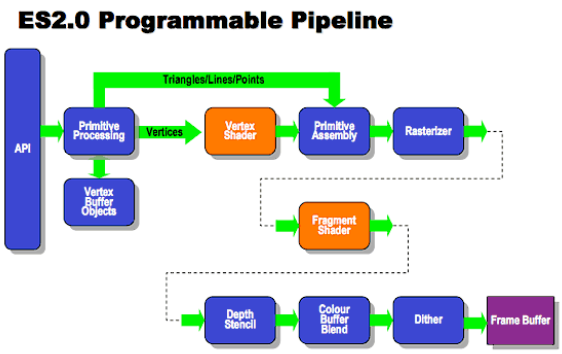


Fig. 9b: Open GL ES 2.0 programmable pipeline [Khronos Group, 2012]

Shaders are code that run on the GPU, written in GLSL (OpenGL Shading Language). They do not compile within the Xcode IDE, instead they are delivered as strings to the OpenGL ES server and compile at runtime, on the iPhone itself. They add so much power and flexibility to the OpenGL ES 2.0 pipeline and are capable of rendering procedural textures, particle effects, complex materials, and so much more that isn't possible in OpenGL ES 1.1. However, a challenging limitation is that their individual vertex or fragment output data can't be read directly, instead the full render is drawn straight onto the frame buffer. This is both a development and analysis obstacle, but still the overall benefits of shaders far outweigh this hindrance.

3.2.1. Vertex Shader

The vertex shader handles per-vertex data such as vertices, normals, and texture coordinates. It can perform complex calculations and pass multiple user-defined parameters to the fragment shader, but must always output a vertex position through the *gl_Position* variable to the next stage of the pipeline. For a complete implementation of this project's vertex shader, see Appendix D.

3.2.2. Fragment Shader

The fragment shader handles per-pixel data such as material characteristics, light intensity, and texture colour. It can receive multiple user-defined parameters from the vertex shader and perform many complex calculations, but its one goal is to output the final colour of a pixel and pass it through to the next stage of the pipeline via the *gl_FragColor* variable. For a complete implementation of this project's fragment shader, see Appendix E.

3.2.3. Per-Vertex vs. Per-Pixel Shading

Generally speaking, there are less vertices in a scene than there are pixels, making it advisable to carry out complex calculations in the vertex shader rather than in the fragment shader. In fact, fragment shaders even require precision qualifiers for variables to ensure better code optimisation. For some models, performing the shading calculations at the vertex level can result in a loss of detail and quality due to inaccurate interpolation. Shifting this work to the fragment shader for per-pixel shading is the solution to this problem, but can be detrimental to performance.

If we consider the largest model of this project - the *Dragon* with 50,000 vertices [Appendix C]- and the screen at full display - 960x640 equalling 614,400 pixels - then indeed the vertices < pixels rule holds true. However, most of the OpenGL ES canvas is rendered transparent, with the model taking up approximately 10-20% of the screen, or roughly 100,000 pixels. This is a much more manageable number for the fragment shader. The application carries out most shading calculations in the fragment shader, but the vertex shader is used for indirect views of the environment (such as those produced by reflections and refractions) which are distorted anyway and don't need to be as detailed as a model texture.

4. Data Capture

4.1. Sensor Analysis

A compositing tool is primarily concerned with visual data, in this case using captured photographs and rendered models to produce a final image. Secondary data from the device sensors can form a bridge between these two separate entities by creating a common environment. Geometric data is needed to inform the position of virtual elements according to their real counterparts. This type of data can be gathered from the following sensors[8]:

- Accelerometer
- Compass
- GPS
- Gyroscope

For these motion sensors, Benford et al. [2003] describe a framework for analysing the movement of physical interfaces in terms of sensibility, *sensability*, and desirability. Sensible movements are those that users naturally perform; sensable movements are those that can be measured by a computer; and desirable movements are those that are required by a given application. In this data capture phase, the user's physical movements include:

1) Be outdoors
2) Choose a camera position
3) Choose a camera orientation

This simple list of activities can be analysed against the motion framework to determine which sensors are most suitable for implementation (Fig. 10)[9].

Description	Movement	Sensors	Sensible	Sensable	Desirable
Be outdoors	Geolocation • Lat., Long., Alt.	GPS	✓	✓	✓
Camera position	Translation in 3D space • X, Y, Z (↑)	Accelerometer, GPS	✓	✗	✓
Camera orientation	Rotation in 3D space • X, Y, Z (θ)	Compass, Gyroscope	✓	✓	✓

Fig. 10: The sensible, sensable, and desirable framework

All listed movements are desirable as they will determine the geometric structure of the virtual environment for the light [Sections 5.3.3 and 5.4], camera [Section 5.3.4] and cube map [Section 5.5.2]. They are also completely sensible as they are naturally occurring, unrestricted movements - the device can move and rotate anywhere in 3D space, while the user can travel anywhere in the world.

[8] The camera could also be used to track movement by implementing select motion-capture or augmented reality tracking techniques. This is beyond the scope of this project, which is exclusively based on pure motion sensors.
[9] It's worth noting that there are other ways to obtain geolocation (e.g. WiFi, cellular towers). However, these are less accurate than GPS and this system has the inherent advantage of outdoor use where GPS signal is usually much stronger and more reliable. Hence, the reason why the alternatives have been largely ignored.

Geolocation is sensable by GPS to an accuracy of approximately ±10 metres horizontally and ±15 metres vertically [Coast Guard Navigation Service, 2011]. This is an acceptable margin of error, as the properties of sunlight are highly unlikely to change within a few metres. Furthermore, this conforms to the chosen sensor accuracy preset [Section 3.1.3.2].

The camera, however, will move about the user in much smaller distances trying to set up the desired shot, resulting in a positional shift that will not be picked up by the GPS. This affects the virtual camera's position as would a zoom or pan about the scene. Accelerometers are much more responsive to smaller movements, detecting changes in acceleration (specifically, *g-force*). This data can then be converted into positional movement using Eq. 1:

$$a = Accelerometer - Gravity$$
$$v = \int a\,dt$$
$$x = \int v\,dt$$

a is the linear acceleration
v is the velocity
x is the position

Eq. 1: Calculating position from accelerometer data

The double integral may be straight-forward, but the real problem lies with the readings as even the slightest sensor error will grow exponentially and deem the whole calculation useless. The iPhone (and most mobile devices) have very noisy accelerometers that require a lot of calibration and filtering. Therefore, calculating movement accurately in 3D space is impossible in practice (for now). Seifert and Camacho [2007] provide a detailed explanation and optimistic implementation; however, this project will have to classify the movement as not sensable.

Finally, the camera orientation can be calculated from the device's compass[10] (heading/azimuth angle) and gyroscope (roll/zenith angle). This is a much easier data set to work with as there are a finite number of angles per rotation axis ($0<x<360$ and $-90<x<90$[11], respectively). These values can be comfortably replicated by a virtual camera as they total two axes, each with an origin reference (magnetic north and device orientation, respectively).

4.2. Environment Capture

Due to the native support for cube maps in OpenGL ES, the environment will be captured as six faces, with the front face annotated with geometric reference data applicable to the rest of the cube map. Fig. 11 shows the user interface for this application phase.

[10] A GPS-assisted compass gives more accurate readings with a true heading, as opposed to a magnetic heading, but in practice the difference between them is never more than three degrees and it is best to conserve resources (and real-time performance) with the less processor-intensive option.

[11] In reality, the gyroscope roll data lies between $0<x<180$ but the $-90<x<90$ scale is more user-friendly as it places 0 at the front-facing landscape orientation.

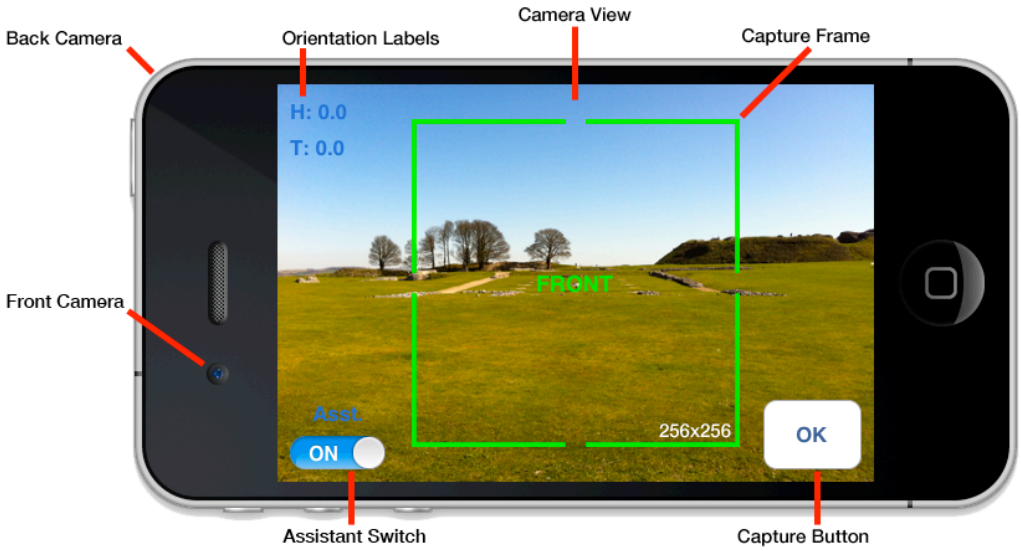


Fig. 11: User interface for the **Environment Capture** phase

The camera view has a total size of 480x360, with a visible display of 480x320. The iPhone only accepts square, power-of-two textures[12] for OpenGL ES rendering, so the choice for cube map face dimensions is clear at 256x256 (2^8x2^8). The next size up lies outside the camera view bounds (512x512)[13] and the next size down is too low-resolution (128x128). The UI contains a capture frame with these dimensions, centred in the screen, for the benefit of the user. The portion of the camera view that lies within this frame at the time of capture will be used as a cube map face. Meanwhile, the device orientation sensors (compass and gyroscope) are running in the background and serve to annotate the final cube map with a geometric reference point.

Fig. 12a describes the pseudo-code for the unassisted environment capture phase of the application. In addition, each picture phase includes a small amount of custom image editing which crops the camera view (480x360) to the capture frame (256x256) and rotates the output by 180 degrees (cube map faces are passed into OpenGL ES with an upside-down orientation). These steps are omitted from the pseudo-code for clarity and brevity. Furthermore, it's worth noting that the user is guided through the capture phase by displaying the name of the next face to be captured on-screen.

[12] Both cube map faces and model textures.

[13] This is the size used for model textures though, since UV texture maps rarely cover the whole area and thus need a slightly higher resolution.

```
Procedure ENVIRONMENT CAPTURE (UNASSISTED)
Inputs: CAMERA; HEADING
begin PROCESS
1:    Take picture from CAMERA (FACE FRONT)
2:    Store HEADING as REFERENCE
3:    Take picture from CAMERA (FACE UP)
4:    Take picture from CAMERA (FACE DOWN)
5:    Take picture from CAMERA (FACE RIGHT)
6:    Take picture from CAMERA (FACE BACK)
7:    Take picture from CAMERA (FACE LEFT)
end PROCESS
Outputs: FACES (x6); REFERENCE
```

Fig. 12a: Pseudo-code for Environment Capture (Unassisted)

This process is very simple but not too user-friendly. A second implementation includes an on-screen assistant which creates orientation boundaries for a better cube map capture experience. Fig. 12b describes the pseudo-code for the assisted version of the environment capture phase. It depicts the system with a ±two-degree boundary for both azimuth and zenith angles.

```
Procedure ENVIRONMENT CAPTURE (ASSISTED)
Inputs: CAMERA; HEADING; ROLL
begin PROCESS
1:    Take picture from CAMERA (FACE FRONT)
2:    Store HEADING as REFERENCE
3:    while (CUBE MAP IS INCOMPLETE)
4:          if (ROLL > 88)
5:                Take picture from CAMERA (FACE UP)
6:          else if (ROLL < -88)
7:                Take picture from CAMERA (FACE DOWN)
8:          else if (-2 < ROLL < 2) AND
                    ((REFERENCE+88) < HEADING < (REFERENCE+92))
9:                Take picture from CAMERA (FACE RIGHT)
10:         else if (-2 < ROLL < 2) AND
                    ((REFERENCE+178) < HEADING < (REFERENCE+182))
11:               Take picture from CAMERA (FACE BACK)
12:         else if (-2 < ROLL < 2) AND
                    ((REFERENCE+268) < HEADING < (REFERENCE+272))
13:               Take picture from CAMERA (FACE LEFT)
14:         end if
15:   end while
end PROCESS
Outputs: FACES (x6); REFERENCE
```

Fig. 12b: Pseudo-code for Environment Capture (Assisted)

This process uses both orientation sensors continuously, but still only annotates the cube map with a heading reference. This is because the user is forced to keep the horizontal faces within a reasonable roll range centred about 0 (front-facing landscape orientation) since the compass becomes unresponsive and error-prone at upward-facing (zenith angle 90) and downward-facing (zenith angle -90) orientations. Furthermore, thanks to gravity, up will always be up and down will always be down for all objects/environments. The azimuth orientation is what really matters since the user's position defines which way is front, right, back, and left.

5. Model Rendering & Compositing

This section covers the main algorithms implemented by this project's previsualisation tool, specifically relating to rendering/compositing. Compositing occurs at the render stage, on the target (synthetic object), in real-time, by leveraging the OpenGL ES 2.0 capability of accessing the vertex and fragment shaders directly. There is no post-processing involved and the synthetic objects are continuously drawn straight from the shaders onto the screen, on top of a live video camera view. The bulk of these algorithms can be found within the code listings for the vertex and fragment shaders [Appendices D and E]. Their resulting images, further discussion, and analysis can be found in Section 7.

5.1. Colour Priors

In some cases, the following algorithms require visual environment/model data (such as mean and standard deviation values) gathered prior to rendering by a custom image analysis interface; I have dubbed these elements *colour priors*.

5.1.1. Model Textures

For the environment (the *source* image), the colour priors are calculated from the cube map faces. For the model (the *target* image), the colour priors are calculated from an edited "green screen" version of the UV texture (Figs. 13a-c). This edited texture is necessary for accurate results, as it is very rare for a UV texture map to cover the whole model texture [Appendix C]. In the case of the banana texture, the yellow background and black blotches would skew the results. When encountered, green screen pixels[14] are discarded from the calculations. The original texture is still used for the main rendering as some models may pick up a green edge in certain areas (much like real chroma key compositing).

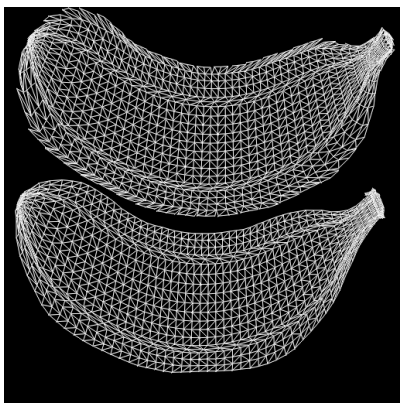


Fig. 13a: UV Texture Map

Fig. 13b: Unedited Texture

Fig. 13c: "Green Screen" Texture

[14] RGB value (0, 255, 0). Interestingly, the RGB colour system justifies the inclusion of a green channel because the human eye is most sensitive to green light. Green screens are designed this way because (1) most computers/cameras can pick up a pure green colour easily thanks to the RGB system, and (2) it is a very uncommon colour, unlikely to be found on real or rendered imagery.

5.1.2. Greyscale Priors

Some algorithms only use intensity priors, requiring a conversion from three-channel RGB to single-channel greyscale. There are many greyscale conversion techniques, each serving a different purpose [Helland, 2012], but the one used throughout the implementation is the *luma* conversion[15] (Eq. 2) using the coefficients defined by ITU-R recommendation BT.709 [International Telecommunication Union, 2009].

$$C = 0.2126R + 0.7152G + 0.0722B$$

R is the red channel
G is the green channel
B is the blue channel
C is the greyscale colour

Eq. 2: Luma greyscale conversion

5.2. Geometry in OpenGL ES

At its most basic level, geometry in OpenGL ES requires two 4x4[16] matrices: the projection matrix and the model-view matrix.

5.2.1. Projection Matrix

The projection matrix defines a perspective view for the virtual camera. For a more accurate AR experience, this matrix draws its parameter values from the physical camera's technical specifications: field of view = 61.048, aspect ratio = 1.5 [Caramba App Development, 2012; Lumo, F., 2010]. All models are processed with Behrens' *obj2opengl* script [2012] and therefore have normalised vertices, so the near and far clipping planes of the projection matrix are set to 0.1[17] and 1.1, respectively.

5.2.2. Model-View Matrix

The model-view matrix is a combination of a model matrix and a view matrix. This is only relevant to explain the name, as later on we'll see that the actual view is controlled in different ways depending on the task at hand. Effectively, the model-view matrix transforms the environment into object-space, centred about the model. In terms of the application, the model-view matrix handles standard object transformations (translate, rotate, scale) controlled by the user.

---

[15] Similar to the green-screen and RGB colour theory, luma is also designed to accommodate the human eye's increased sensitivity to green light (followed by red, then blue light).

[16] There is a practical reason for 4x4 matrices in 3D space, in terms of handling homogeneous coordinates, quaternions, the concept of infinity, etc. This is a very deep and complex mathematical subject, but the resulting model-view-projection matrix system in OpenGL ES literature is largely presented as standard practice without further discussion. Bomfin [2011] delivers a detailed guide covering the fundamentals of this area, based on work by Baker [2012] and Lengyel [2004]. Buck [2012] also provides a very solid coverage of the topic.

[17] Negative values are invalid and really small values close to zero may cause *z-fighting*.

5.3. Lighting & Shading

5.3.1. Lighting Model

Both real and simulated light are separated into two categories: *direct* and *indirect* illumination[18]. Direct illumination is light that hits a surface by travelling directly from a light source. Indirect illumination is light that hits a surface after being reflected off another object. A combination of both is needed to replicate natural illumination.

5.3.1.1. Ambient Light (Daylight)

Daytime refers to the time of day between sunrise and sunset, where a given point on the planet's surface experiences natural illumination as a cause of daylight. This includes direct illumination from sunlight, as well as indirect illumination caused by diffuse sky radiation[19] and all reflections off outdoor surfaces. Thus, no objects are completely obscured during daytime, and the way to simulate this in a virtual environment is through ambient light.
Ambient light performs a constant illumination throughout the scene, emitted uniformly in all directions, with a constant magnitude[20].

In this implementation, daylight only has an intensity value, l_d, as it is very difficult to calculate a suitable colour, affected by many uncontrollable factors such as weather and pollution. This, however, can be compensated for with colour-correction algorithms [Section 5.6]. Daylight intensity is a decision left to the user, as image-processing techniques would need more information (such as ground truth images) to distinguish between the natural colour of objects and their perceived colour given the current lighting conditions.

5.3.1.2 Directional Light (Sunlight)

Sunlight is simulated by directional light, in which all emitted rays are parallel to each other. They originate from a single point and travel in a single direction. Furthermore, their incidence radiance is not affected by the position of the light source along the ray's direction of travel (no decay). Essentially, this means that directional light simulates a light source that is infinitely far away, exclusively emitting parallel rays[21]. This is physically impossible in reality, due this synthetic light source having no surface area, but it's a very close approximation to sunlight.

For the same reasons explained by daylight, sunlight has an intensity value, l_s, defined by the user, and no colour[22]. Furthermore, it has a position, L, described in Section 5.3.3.

5.3.2. Shading Model

The implementation adopts a modified version of the *Blinn-Phong* shading model [Blinn, 1977] for its simplicity, effectiveness, and ability to accommodate a mobile light source and viewer[23]. The surface attributes that fit this model include an ambient, diffuse, and specular

---

[18] Also referred to as *local* and *global* illumination, respectively.

[19] As a side note, this phenomenon explains the perceived colour of the sky, which is blue due to the fact that the atmosphere scatters blue light more effectively than any other colour.

[20] This is known as a 3D isotropic radiance field.

[21] The mathematical theory behind directional light comes from the *Dirac delta function*: a generalised function that is zero everywhere except at the origin, where it is infinite.

[22] Technically, the intensity associated with both lights is a colour ranging from pure black (intensity = 0.0) to pure white (intensity = 1.0).

[23] In fact, this is the default shading model used in the fixed function OpenGL ES 1.1 pipeline.

component, as well as a gloss factor (best used for rendering refractive materials). A diagram (Fig. 14) and equation (Eq. 3) describe the model, with naming conventions to suit this project.

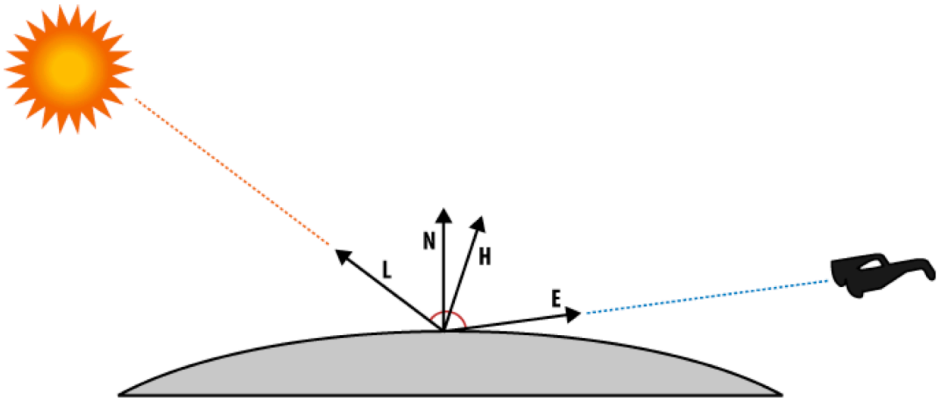


Fig. 14: Diagram of the Blinn-Phong shading model [Rideout, 2010]

$$d_f = N \bullet L$$

$$H = \frac{L+E}{|L+E|}$$

$$s_f = (N \bullet H)^{g_f}$$

$$C = l_d * (a_c) + l_s * ((d_c * d_f) + (s_c * s_f))$$

N is the surface normal
L is the sun position
E is the camera position
H is the halfway vector between L,E
d_f is the diffuse factor
s_f is the specular factor
g_f is the gloss factor
a_c is the ambient colour
d_c is the diffuse colour
s_c is the specular colour
l_d is the daylight intensity
l_s is the sunlight intensity
C is the output colour

Eq. 3: The modified Blinn-Phong shading model

The values of d_f *and* s_f are clamped to a non-negative value as only visible surfaces are calculated (one-sided lighting). All vectors (N, L, E, H) must be normalised if they haven't done so already.

Textures dictate the colour of the model, therefore the ambient, diffuse and specular properties simply adjust the intensity values, i.e., there is one colour channel for each rather than three for each. Below are brief definitions of these components, given for ease of reading and comprehension:

- Ambient: The colour of an object in shadow.
- Diffuse: The colour of an object under pure white light.
- Specular: The colour of light reflected off the object.

Since the application is in object-space, normals[24] need to be transformed into viewer-space by calculating the inverse-transpose of the model-view matrix. The resulting 4x4 matrix is

[24] A surface normal is a vector perpendicular to the surface, defining its orientation.

further reduced to a 3x3 viewer-space normal matrix for calculating correct surface normals[25]. The process is described by Eq. 4.

$$N_v = \left(M^{-1}\right)^T * N_o$$

M is the model-view matrix
N_o is the object-space surface normal
N_v is the viewer-space surface normal

Eq. 4: Surface normal conversion from object-space to viewer-space

5.3.3. Sun Position

The sun position, *L*, is calculated with the aid of the third-party *Solar Position Algorithm* (SPA) [National Renewable Energy Laboratory, 2003]. The algorithm inputs are shown in Fig. 15, with constants ΔT[26] provided by the SPA itself and meteorological data for Bristol, UK[27] provided by Alveston Bristol UK Weather [2012]. Variable inputs are provided by the device and updated as needed. Surface rotation (azimuth) and slope (zenith) angles could be gathered from the compass and gyroscope respectively, but the SPA doesn't seem to take these values into account for the final output. Instead, these values are used for matrix transformations covered further in this section.

Variables	**Constants**	
Year	Timezone:	0
Month	Delta T (ΔT):	64.797
Day	Average annual pressure:	1015
Hour	Average annual temperature:	11
Minute	Atmospheric refraction*:	0.5667
Second	Surface rotation*:	-
Latitude	Surface slope*:	-
Longitude		
Altitude		*Optional values

Fig. 15: SPA inputs

The SPA outputs a zenith angle, θ, and azimuth angle, ϕ, for the position of the sun in the sky. This data conforms to a horizontal coordinate system for celestial bodies, which needs to be converted to 3D cartesian coordinates for use in the OpenGL ES environment. This conversion is done via the spherical coordinate system. Figs. 16a-c depict diagrams for each coordinate system and Eq. 5 outlines the mathematical conversion.

[25] This removes the unnecessary translation of normal vectors by extracting the upper-left 3x3 matrix. Once again, the mathematical intricacies are explained by Bomfin [2011] and Rideout [2010], amongst others.
[26] ΔT = Terrestrial Time (TT) - Universal Time (UT).
[27] Development and testing site. A web-query script or geographical database would be needed for the application to be accurate worldwide

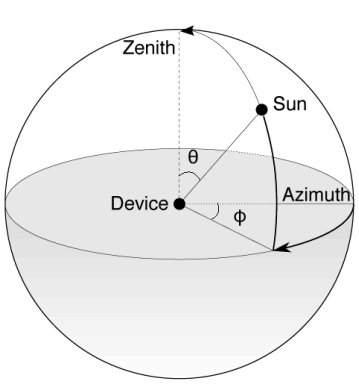


Fig.16a: Horizontal coordinate system

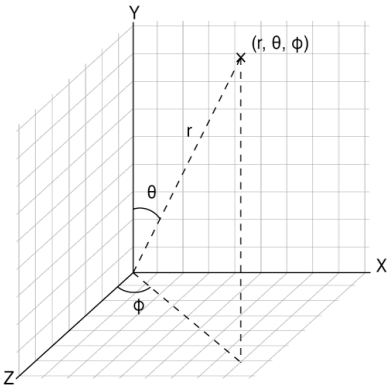


Fig. 16b: Spherical coordinate system

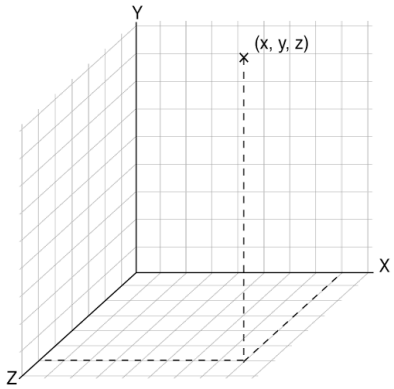


Fig. 16c: Cartesian coordinate system

$$x = r * \sin\theta * \sin\phi$$
$$y = r * \cos\theta$$
$$z = r * \sin\theta * \cos\phi$$

r is the radial distance
θ is the zenith angle
ϕ is the azimuth angle
x, y, z are the cartesian coordinates

Eq. 5: Coordinate system conversion from horizontal to cartesian, via spherical

Furthermore, there are two practical modifications to this process:

1) r is negligible because the sun is simulated by an infinitely far directional light
2) z is inverted because in OpenGL ES negative values go "into" the screen

Finally, the sun position relative to the device orientation is calculated with a matrix driven by sensor data. This matrix is rotated about the x-axis by the gyroscope roll angle and rotated about the y-axis by the compass heading angle. The resultant matrix is then multiplied by the initial sun position vector *(x, y, z)*, giving the final sun position vector

5.3.4. Camera Position

The camera position, *E*, is calculated in a similar way to the final step of the sun position algorithm. The shading model assumes an infinitely far viewer, much like a directional light, since the device can't actually detect the user's view pose relative to the screen. Therefore, the initial camera position vector is *(1, 1, 1)* - this is for effective vector-matrix multiplications rather than representing a default position (it's more like an "identity vector"). The orientation identity matrix is rotated about the x-axis by the gyroscope roll angle and rotated about the y-axis by the compass heading angle. The resultant matrix is then multiplied by the initial camera position vector, giving the final camera position vector.

5.4. Hemisphere Shading

Hemisphere shading is a term I coined to describe the combination of hemisphere lighting and hemisphere occlusion in a context-aware rendering environment.

5.4.1. Hemisphere Lighting

Hemisphere lighting is an indirect illumination algorithm simulating the effect of daylight, based on two hemispheres. An object in a scene is prone to global illumination effects from

its surroundings[28], for which hemisphere lighting isolates a sky plane directly above the object and a ground plane directly below it. These planes are also referred to as the upper and lower hemisphere, respectively. In hemisphere lighting, a surface location with a normal pointing straight up receives all of its illumination from the upper hemisphere. Consequently, a surface location with a normal pointing straight down receives all of its illumination from the lower hemisphere. Surface points in between these two extremes receive their illumination from a combination of both, depending on the angle of the normal. Fig. 17 depicts a sphere illuminated by hemisphere lighting and Eq. 6 outlines the maths.

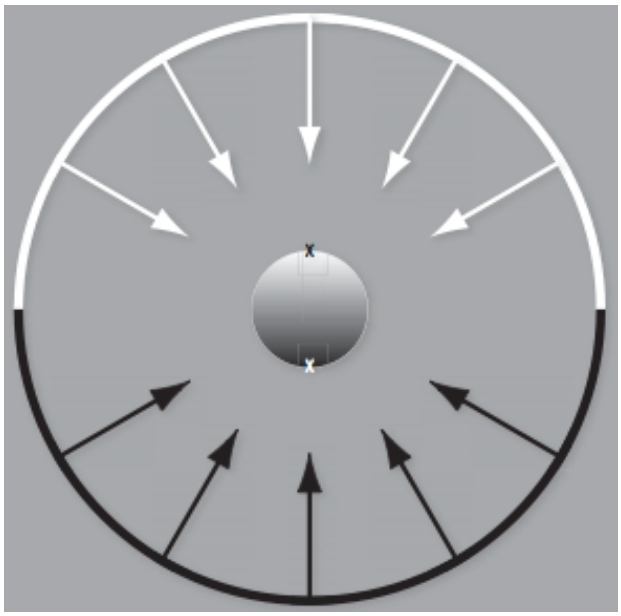


Fig. 17: Hemisphere lighting model [Rost and Licea-Kane, 2009]

$$\cos\theta = N \bullet E_r$$
$$a = 0.5 + (0.5 * \cos\theta)$$
$$C = a * c_u + (1.0 - a) * c_l$$

N is the surface normal
E_r is the camera roll vector
a is the blend coefficient
c_u is the upper hemisphere colour
c_l is the lower hemisphere colour
C is the output colour

Eq. 6: Hemisphere lighting calculation

The original algorithm [Rost and Licea-Kane, 2009] simply takes θ as the surface normal angle calculated from a pre-determined light position vector. However, this implementation leverages the advantage of a mobile camera to define θ based on the camera's view of the surface normal, given a certain gyroscope roll value[29]. The hemisphere lighting model operates within the range $0<\theta<180$, so gyroscope data is adjusted accordingly from $-90<\theta<90$. It's then passed as a 3D vector to compute the dot product with the surface normal.

The colour of the hemispheres is decided by *colour priors* calculated from the environment, for an effective composite. The upper hemisphere colour, c_u, is extracted from the mean RGB value of the top cube map face. The lower hemisphere colour, c_l, is extracted from the mean RGB value of the bottom cube map face.

5.4.2. Hemisphere Occlusion

Ambient occlusion is a ray-tracing technique for rendering photo-realistic surface shadows. Ambient light on its own simply illuminates objects with a flat, constant colour, creating a dangerous compositing territory in the absence of sunlight on a cloudy day. Ambient occlusion addresses this situation by performing self-shading of objects based on their own

[28] In close-quarters, many other objects would affect the colour of the model, but in wide open areas (such as a field) the effects of hemisphere lighting are more apparent.
[29] Both N and E_r are normalised (length is 1), so $cos\theta$ can be taken straight from the dot product of the two vectors.

(and surrounding) geometry. The basic idea of ambient occlusion is to determine the amount of ambient light received by a surface point. For each point, this amount depends on how much of the upper hemisphere is visible and how much is blocked (occluded) by surrounding geometry [Suffern, 2007]. This is why true ambient occlusion is only possible with ray-tracing, as each object vertex must perform ray-casting to determine its occlusion factor[30]. Like the diffuse and specular components previously covered, this occlusion factor is then multiplied by the model's ambient colour to determine the final ambient appearance.

Hemisphere lighting can be used to simulate a crude version of ambient occlusion good enough for use in previsualisation compositing [Section 7.2.2], as long as appropriate values are chosen. Once again, this is decided by *colour priors* calculated from the environment. Ambient occlusion is a shadowing algorithm, and therefore uses greyscale values. It would be inadequate to use data from the top and bottom cube map faces because, theoretically, they are both flat planes which are not susceptible to environment shadowing effects: looking up means looking directly at the light source, while looking down means looking at a perfectly illuminated surface. Given that natural illumination always occurs from top to bottom - never from the ground-up - then it's more appropriate to examine its effect on objects within the horizontal planes: front, back, left, and right cube map faces. These images show objects in light and in shadow, and thus are a better gauge of the intensity information of the environment. The upper hemisphere colour (pure light), c_u, is extracted from the lightest greyscale value of the horizontal cube map faces. The lower hemisphere colour (pure shadow), c_l, is extracted from the darkest greyscale value of the horizontal cube map faces.

5.5. Environment Mapping

Environment mapping is a form of image-based lighting which captures the scene surrounding an object as one or more texture maps. These specialised textures are usually viewed indirectly through accurate reflections or refractions on the object surface, computed at render time.

5.5.1. Reflections & Refractions

Reflections and refractions are standard mathematical procedures built into OpenGL ES (Fig. 18), but briefly examined here for cohesiveness:

Function	Description
reflect(I, N)	For the incident vector *I* and surface normal *N*, returns the reflection direction.
refract(I, N, eta)	For the incident vector *I*, surface normal N, and the refractive index *eta*, returns the refraction direction.

Fig. 18: Built-in OpenGL ES functions for reflection and refraction [Khronos Group, 2012]

For both functions, the incident vector *I* is a vertex of the object and the surface normal *N* is the corresponding normal (both are 3D vectors in object-space). Instead of transforming these into viewer-space with the usual normal matrix, as per the shading model, they are transformed post-reflection/refraction with an environment matrix. This is because the

[30] Also referred to as *visibility* or *accessibility* factor.

surface reflections/refractions "stick" to the model and are dependent on the orientation of the device. The environment matrix for reflections is simply the normal matrix rotated about the x-axis by the gyroscope roll angle and rotated about the y-axis by the compass heading angle. The environment matrix for refractions is calculated in the same way, but with negative-value angles (a transparent material sees through to the opposite cube map face).

Transparent, refractive materials conform to a reduced version of the Blinn-Phong shading model, described by Eq. 7.

$$H = \frac{L+E}{|L+E|}$$

$$s_f = (N \bullet H)^{g_f}$$

$$C = r_c + s_f$$

N is the surface normal
L is the sun position
E is the camera position
H is the halfway vector between L,E
s_f is the specular factor
g_f is the gloss factor
r_c is the refraction colour
C is the output colour

Eq. 7: Blinn-Phong shading model for transparent, refractive materials

In this implementation, the material doesn't have a colour (nor a model texture) and is rendered completely transparent, only with specular highlights[31]. The model uses two-sided lighting as s_f is not clamped to a non-negative value, as it is for opaque surfaces. This is a very basic simulation of refraction as more accurate techniques require ray-tracing. However, it's adequate enough for simulating further techniques.

5.5.2. Cube Map

Environment mapping is traditionally accomplished with either spherical or cube mapping[32]. Both methods are supported by desktop OpenGL but OpenGL ES only uses cube mapping which is thankfully the method best suited to the implementation, from the capture phase up until the final render.

A cube map is a specialised texture constructed from six individual 2D textures, organised to represent the faces of a cube: Left (-X), Right (+X), Down (-Y), Up (+Y), Back (-Z), Front (+Z). With each face defined by a sign and an axis, a 3D space is created. Within this space, a 3D texture coordinate used as a direction vector can decide which face(s) it's looking at, and return a texel[33] value intersecting the cube map, thus computing an accurate colour for the reflection/refraction algorithms. The built-in GLSL function *textureCube(sampler, coord)* handles the lookup.

In AR compositing, cube maps are most suitable for rendering reflections because they produce an indirect view of elements outside the camera view - there is no direct comparison to what isn't part of the visible scene[34]. In some cases, cube maps produce even better results because the camera/photographer is not rendered by the reflections (since they weren't captured for the cube map). Refractions, on the other hand, can be an instant giveaway of a bad composite because the transparent object creates the illusion of seeing

---

[31] These properties are particularly helpful in reducing calculations for the processor-intensive *live feed* procedure.
[32] Lat-long maps are also popular, specially within Debevec's IBL literature [2012].
[33] Texture element, akin to pixel (picture element).
[34] This is mostly true, as the model does pick up reflections from nearby objects.

right through it to the AR camera view of the scene (with the proper distortion, of course). A cube map misaligned with the current video frame or a change in scenery (e.g., a person walking behind the object, not captured by the original cube map) will produce an obviously inaccurate composite, which brings us to the next section...

5.5.3. Live Feed

Transparent objects blend right through to the underlaying camera view, creating a challenging AR compositing problem as outlined above. This is solved with continuous re-texturing from a live camera feed, which definitely straddles the boundary of real-time rendering, massively increasing the system load. There isn't a particularly complex algorithm for this procedure[35], but special care must be taken towards efficient data management and optimised code execution. Leveraging the power of the *self.pause* function in *GLKit* for pausing and resuming OpenGL ES rendering (exclusive to iOS 5), Fig. 19 describes the pseudo-code for processing refractions from a live camera feed.

```
Procedure LIVE REFRACTIONS
Inputs: CAMERA
begin PROCESS
1:    Initialise iOS environment
2:    Initialise OpenGL ES environment
3:    while (RUNNING)
4:          pause rendering cycle
5:                Retrieve CAMERA image
6:                Crop image to satisfy texture dimensions (256x256)
7:                Convert image to PNG format
8:                Assign image to texture
9:          resume rendering cycle
10:         Clear buffers: colour, depth
11:         Load model
12:         Attach texture
13:         Draw scene
14:         Update shaders: attributes, uniforms
15:   end while
end PROCESS
```

Fig. 19: Pseudo-code for Live Refractions

The critical component of this process lies within lines 5-8, between the rendering cycle pause and resume commands. Retrieving the camera image (5) is a smooth procedure since the raw camera output has already been converted to the native *UIImage* format for the AR display. Cropping the image (6) to conform to texture specifications (square, power-of-two dimensions) involves quick and easy image-editing code, still operating at a relatively high level within the iOS hierarchy (Media layer). Converting the image to the PNG format (7) is the most expensive procedure, incurring a costly read and write cycle since OpenGL ES will not accept the native *UIImage* format. This step requires the use of lower-level frameworks from the Core Services layer. Finally, the image is assigned to an OpenGL ES texture node (8) with standard texture-loading code.

This process can also be used for rendering live reflections, using the front-facing device camera. This is simply an experimental feature since, from a compositing point-of-view, it doesn't make sense to render live reflections at the cost of losing the back-facing camera.

[35] Just the simple task of reducing a texture coordinate from 3D to 2D.

This means that the whole display would act as a mirror, rather than a lens, with the synthetic object on top. It may be possible to use both cameras in the future, but not for now.

5.6. Colour Correction

Finally, colour correction algorithms take charge of the more traditional aspects of compositing: comparing a source and target image, then adjusting the colour values of the latter to match the former.

5.6.1. Intensity Adjustments

Brightness and contrast algorithms are image interpolation procedures that adjust the intensity of a pixel, based on a target image. These algorithms were first devised by Haeberli and Voorhies [1994] but are now considered standard image manipulation techniques for 2D [Brinkmann, 2008] and 3D [Rost and Licea-Kane, 2009] graphics. The general form of these algorithms is represented by Eq. 8[36].

$$C_o = C_s * (1.0 - \alpha) + C_t * (\alpha)$$

α is the interpolation coefficient
C_s is the source colour
C_t is the target colour
C_o is the output colour

Eq. 8: General form of image interpolation algorithms

The source images are usually a uniform shade of grey, which we can extract from greyscale *colour priors* calculated from the environment's horizontal planes (front, back, left, and right cube map faces). The first task for these priors is to fit the luma-converted target pixel between the lowest-value and highest-value priors using the built-in OpenGL ES *smoothstep* function[37]. This ensures that the rendered model is never darker or lighter than the environment (much like the hemisphere occlusion algorithm values [Section 5.4.2]).

5.6.1.1. Brightness

Brightness is a simple scaling operation, multiplying each fragment shader pixel by an intensity factor between two specific bounds. Like the smoothstep clamping process, these bounds are defined by the darkest and lightest priors from the environment's horizontal planes. The floating-point intensity value of the current luma-converted target pixel is then used as the interpolation coefficient for performing a linear blend between these two priors. The result is the intensity factor for adjusting the brightness of the same target pixel. Eq. 9 outlines this process.

$$b = s_l * (1.0 - t) + s_h * (t)$$
$$C_o = t * b$$

t is the target pixel
s_l is the lowest prior value
s_h is the highest prior value
b is the brightness intensity factor
C_o is the output colour

Eq. 9: Brightness adjustment

[36] This is such a common algorithm that it is built into GLSL as the *mix* function.
[37] This is a essentially a clamping function with smooth Hermite interpolation (an "S" curve).

5.6.1.2. Contrast

Contrast is a step further from brightness, adjusting the relationship between the higher and lower intensities of an image. High contrast increases the differences in the intensity range, whereas low contrast reduces them. A simple contrast operator will likely use a grey mid-tone value (intensity ≈ 0.5) to compute the contrast curve, making everything above this point lighter and everything below it darker. This curve point is adequate for images with fairly even intensity distributions, but this may not always be the case (e.g. sunny scenes are much brighter than overcast scenes). *Biased* contrast operators have an adjustable curve point, usually selected manually by the user [Brinkmann, 2008]. This implementation defines this value automatically, extracted from the mean greyscale prior of the environment's horizontal planes. Eqs. 10a and 10b outline the contrast adjustment procedure for low and high intensities. If the target pixel, t, is less than the source mean, s_μ, then the lower range procedure is performed, else the higher range procedure takes precedence.

$$t' = \frac{(t - s_l)}{(s_\mu - s_l)}$$

$$c = s_l * (1.0 - t') + s_\mu * (t')$$

$$C_o = t * c$$

t is the target pixel
t' is the range-adjusted target pixel
s_l is the lowest prior value
s_μ is the mean prior value
c is the contrast intensity factor
C_o is the output colour

Eq. 10a: Contrast adjustment (lower range)

$$t' = \frac{(t - s_\mu)}{(s_h - s_\mu)}$$

$$c = s_\mu * (1.0 - t') + s_h * (t')$$

$$C_o = t * c$$

t is the target pixel
t' is the range-adjusted target pixel
s_μ is the mean prior value
s_h is the highest prior value
c is the contrast intensity factor
C_o is the output colour

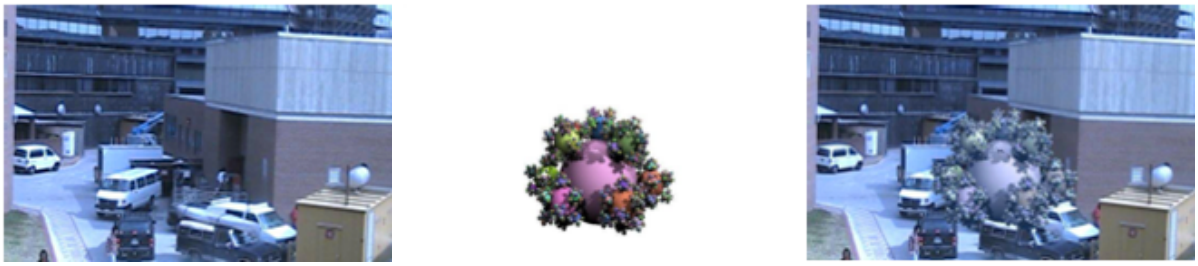
Eq. 10b: Contrast adjustment (higher range)

5.6.2. Colour Matching

The colour matching method in this section is based on an offline colour transfer algorithm for images [Reinhard et al., 2001] and a real-time colour blending algorithm for video [Reinhard et al., 2004] with similar foundations. The results of these algorithms are respectively reproduced in Fig. 20 and Fig. 21 to help distinguish the two in the subsequent analysis.

Source Target Composite

Fig. 20: Colour transfer algorithm [Reinhard et al., 2001]



Environment Model Composite

Fig. 21: Colour blending algorithm [Reinhard et al., 2004]

Both algorithms adjust a target (model) from source (environment) data. They perform the compositing in post-processing once all imagery has been rendered or captured and there is a full set of data available.

The colour-transfer algorithm manipulates data in the Lαβ[38] colour space [Ruderman et al., 1998; cited in Reinhard et al, 2001], while the colour-blending algorithm manipulates data in a version of the CIELAB[39] colour space [Wyszecki and Stiles, 1982; cited in Reinhard et al, 2004]. The key difference is that CIELAB is preferred as an optimised version of *Lαβ* which omits a superfluous conversion to logarithmic space, in order to fast-process data.

The 2004 solution includes two user parameters, simply referred to as *s* and *d*, for adjusting the intensity of the blending effect. A strange observation made in their own paper is that parameter *d* "did not appreciably change the result" [p.6] and may be removed from the calculations. It's odd that it's still included in the final version of their algorithm, but an even more curious fact is that omitting *d* effectively excludes the source mean and leaves only the following data to perform the compositing: target value, target mean, target variance, and source variance. The 2001 solution doesn't have any user parameters but does have a very robust model, using the following data to perform the compositing: target value, target mean, target standard deviation, source mean, and source standard deviation.

This project implements a combination of the two algorithms, adopting:

1) The CIELAB colour space approximation from Reinhard et al. [2004] (Eq. 11a)[40]; and,
2) The mathematical process from Reinhard et al. [2001] (Eq. 11b)

[38] Lightness (L) and colour-opponent (A,B) dimensions.
[39] *Commission Internationale de L'Eclairage* Laβ.
[40] In the GLSL implementation, this data must be converted to column-major order.

$$\begin{bmatrix} L \\ \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0.3475 & 0.8231 & 0.5559 \\ 0.2162 & 0.4316 & -0.6411 \\ 0.1304 & -0.1033 & -0.0269 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.5773 & 0.2621 & 5.6947 \\ 0.5774 & 0.6072 & -2.5444 \\ 0.5832 & -1.0627 & 0.2073 \end{bmatrix} \begin{bmatrix} L \\ \alpha \\ \beta \end{bmatrix}$$

Eq. 11a: RGB-CIELAB colour space conversion

$$C_o = \left(t - t_\mu\right) * \left(\frac{t_\sigma}{s_\sigma}\right) + s_\mu$$

t is the target pixel
t_μ is the target mean
t_σ is the target standard deviation
s_μ is the source mean
s_σ is the source standard deviation
C_o is the output colour

Eq. 11b: Colour transfer

The source, s, and target, t, mean (μ) and standard deviation (σ) values are extracted from colour priors. The target priors use data from edited green-screen model textures while the source priors use data from the cube map's horizontal planes.

6. Final Implementation - Visual Summary

This section provides a visual summary of the final implementation, with the aid of a system block diagram (Fig. 22) and user interface diagram (Fig. 23).

6.1. System Block Diagram[41]

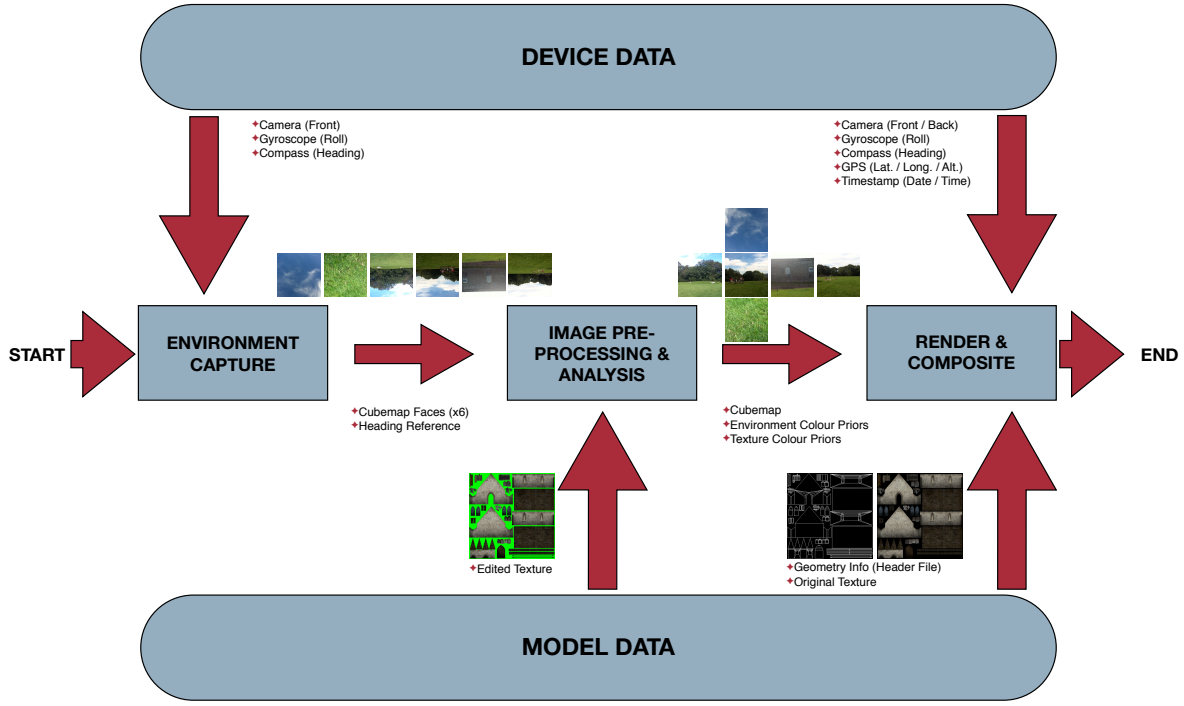
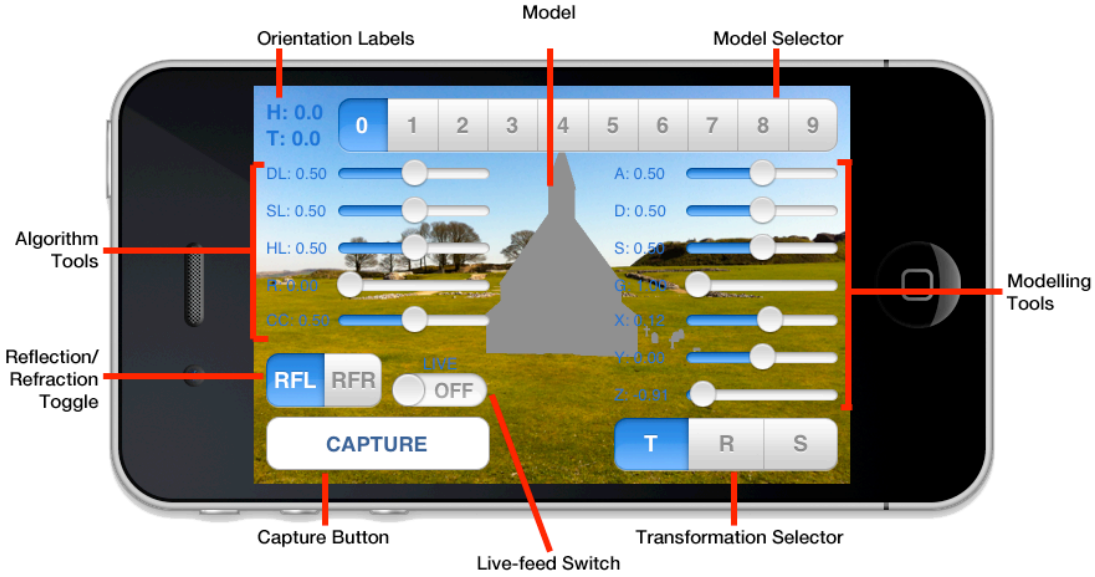


Fig. 22: System Block Diagram

[41] User inputs have been purposely omitted for brevity and clarity, but they affect the *Environment Capture* and *Render & Composite* stages.

6.2. User Interface Diagram[42]



Slider Key:

Algorithm tools

DL:	Daylight
SL:	Sunlight
HL:	Hemisphere shading
R:	Reflection/Refraction
CC:	Colour correction

Modelling tools

A:	Ambient intensity
D:	Diffuse intensity
S:	Specular intensity
G:	Gloss factor
X:	X-axis transformations
Y:	Y-axis transformations
Z:	Z-axis transformations

Fig. 23: User interface diagram for the **Render & Composite** stage

[42] The model has been purposely rendered with a flat grey colour in order to preserve the surprise of the next section...

7. Testing & Results

The testing of the system was carried out by isolating the main algorithms into separate applications, in order to clearly appreciate key elements and individual effects. For the same reason, most results are rendered against a black background or pre-loaded environment as the underlaying camera view would be too inconsistent and distracting.

The final implementation combines all the algorithms into a single application which still performs in real-time, for which fully composited renderings are presented at the very end of this section.

There is no post-processing involved, be it manual or automatic, in any of these images.

7.1. Sunlight

This section examines the basic lighting and shading algorithm only, with particular emphasis on the SPA output when calculated and transformed with real device data (final sun position).

Test Parameters:
Ambient: 0.50; Diffuse: 0.50; Specular: 0.50; Gloss: 1.0; Daylight: 0.50; Sunlight; 0.50

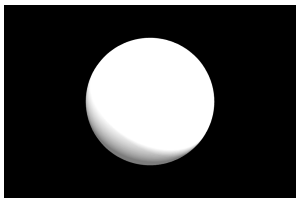


Fig. 24a: Test Sphere, Sun position (0.3, 1.0, 1.0)

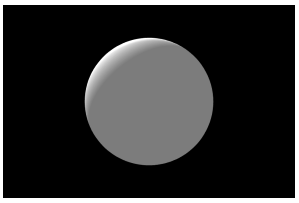


Fig. 24b: Test Sphere, Sun position (-0.3, 1.0, -1.0)

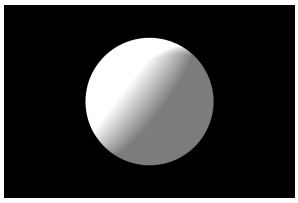


Fig. 24c: Test Sphere, Sun position (-1.0, 1.0, 0.3)

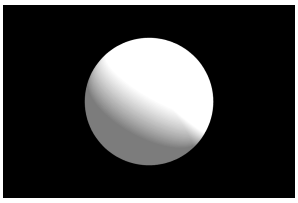


Fig. 24d: Test Sphere, Sun position (1.0, 1.0, -0.3)

Fig. 24e: Chapel, Sun position (0.3, 1.0, 1.0)

Fig. 24f: Chapel, Sun position (-0.3, 1.0, -0.9)

Fig. 24g: Chapel, Sun position (-1.0, 1.0, 0.2)

Fig. 24h: Chapel, Sun position (1.0, 1.0, -0.4)

The shading model performs as expected, albeit with slightly rough interpolation between the ambient, diffuse, and specular components. This is most apparent on the *Test Sphere* model, particularly Figs. 24c and 24d, but unnoticeable on the *Chapel* model. The more computationally-expensive *Phong* shading algorithm could be used instead of the optimised *Blinn-Phong* solution (which uses *Gouraud* shading), thus increasing the smoothness of the shading interpolation [Phong, 1975]. However, the difference is likely to go unnoticed on more expressive, textured models than the simple test sphere.

The SPA and matrix transformations are highly responsive to device data, resulting in an instant and accurate calculation of the virtual directional light's position with regards to the real sun (not pictured).

7.2. Hemisphere Shading

The hemisphere shading algorithms simulate crude approximations of more advanced ray tracing techniques which can't perform in real-time on commercial mobile devices. This section compares the visual results and render times of hemisphere shading algorithms rendered with OpenGL ES versus desktop ray tracing algorithms rendered with *Mental Ray* on *Maya*.

In this section, the OpenGL ES rendering time is fixed and defined by the iOS *GLKViewController* frame rate speed of 30 FPS, or approximately 0.03 seconds. In order to reduce experimental bias, both algorithms use the same *Dragon* model, very similar projection views, the same resolution (480x320), and particular care has been taken to approximate the model transformations (translate, rotate, scale).

A particular point of note is that the the desktop setup does not have access to instantaneous gyroscope data, highlighting an inherent advantage of the mobile device

7.2.1. Hemisphere Lighting

Hemisphere lighting in OpenGL ES attempts to simulate global illumination in Mental Ray. In the desktop setup, the model lies between two actual vertical polygonal planes.

Test Parameters:

Sky RGB colour: (135, 206, 235) Ground RGB colour: (77, 189, 51)

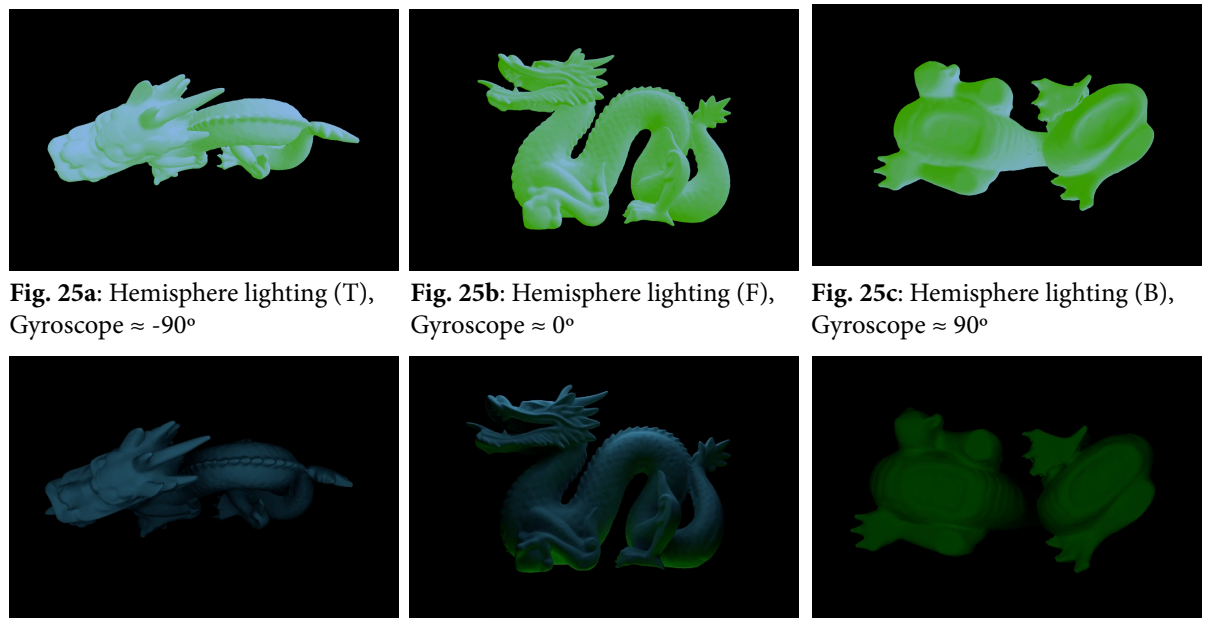


Fig. 25a: Hemisphere lighting (T), Gyroscope ≈ -90°

Fig. 25b: Hemisphere lighting (F), Gyroscope ≈ 0°

Fig. 25c: Hemisphere lighting (B), Gyroscope ≈ 90°

Fig. 25d: Global illumination (T), Render time ≈ 4.0 seconds

Fig. 25e: Global illumination (F), Render time ≈ 5.0 seconds

Fig. 25f: Global illumination (B), Render time ≈ 4.0 seconds

The first thing to note is that Mental Ray uses many parameters to perform the render (including final gathering), explaining the darker blue and green colours. Nonetheless, the results are very different from each other.

Global illumination produces a much more accurate render of the top (Fig. 25d) and bottom views (Fig. 25f), each shading the object entirely in shades of their respective sky and ground planes. This is mostly the case for hemisphere lighting (Figs. 25a and 25c), but there is a noticeable amount of spill from the opposite hemisphere.

The front views are much more telling as global illumination (Fig. 25e) accurately simulates a dominance of the sky plane, since the direction of travel of natural illumination is top-to-bottom (followed by ground-up reflections and many other combinations). In hemisphere lighting, a front view should theoretically render an equal blend of both hemisphere colours, but Fig. 25b shows a slight lighting dominance of the lower hemisphere.

Hemisphere lighting is far from convincingly simulating global illumination, but the render times disclose that global illumination is far from real-time performance.

7.2.2. Hemisphere Occlusion

Hemisphere occlusion in OpenGL ES attempts to simulate ambient occlusion in Mental Ray. Results are further compared to default rendering with ambient light in Maya, also using Mental Ray.

Test Parameters:

Light intensity: 0.9 Shadow intensity: 0.1

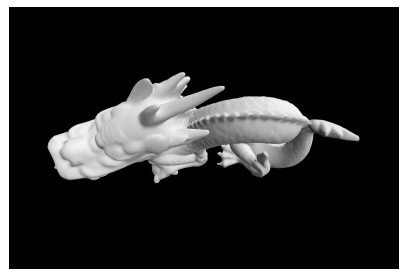


Fig. 26a: Hemisphere occlusion (T), Gyroscope ≈ -90°

Fig. 26b: Hemisphere occlusion (F), Gyroscope ≈ 0°

Fig. 26c: Hemisphere occlusion (B), Gyroscope ≈ 90°

Fig. 26d: Ambient occlusion (T), Render time ≈ 15.0 seconds

Fig. 26e: Ambient occlusion (F), Render time ≈ 13.0 seconds

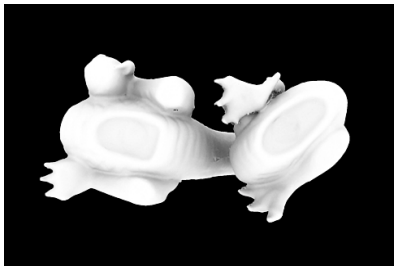


Fig. 26f: Ambient occlusion (B), Render time ≈ 12.0 seconds

Fig. 26g: Default illumination (T), Render time ≈ 1.0 seconds

Fig. 26h: Default illumination (F), Render time ≈ 2.0 seconds

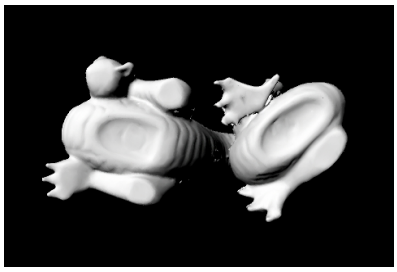


Fig. 26i: Default illumination (B), Render time ≈ 1.0 seconds

Once again, Mental Ray uses many parameters to control the final render, resulting in intensities outside the bounds specified. However, we can definitely see much more similarity between the techniques and the shape of the model is very neatly defined in all cases.

The render times for ambient occlusion (Figs. 26d-f) instantly stand out at a staggering 13.33 seconds average per image. This clearly indicates that ray-traced ambient occlusion is still a long way away from real-time rendering.

The front views (Figs. 26b, 26e, and 26h) show that the hemisphere occlusion pass is noticeably darker than the ambient occlusion pass, caused by a dominance of the shadow hemisphere. It is very similar to the default illumination though, which has a surprisingly high render time of 2.0 seconds given it's simplicity.

Since there is no additional geometry in the ray-traced scenes, the model is not occluded by a ground plane and hence has a high visibility factor. This produces an accurate, but incorrect render of the models if they were viewed on a mobile device. The top views (Figs. 26d and 26g) have a very similar intensity distribution to the bottom views (Figs. 26f and 26i), when in reality the former should be much lighter than the latter. Thanks to the instantaneous gyroscope roll data, hemisphere occlusion renders these views correctly with a light top (Fig. 26a) and a dark bottom (Fig. 26c).

For this last reason, *baked lighting* - previously-rendered lighting information stored as a texture map (e.g. per-vertex ambient occlusion coefficients) - is an unsuitable alternative to ambient occlusion on OpenGL ES because the compositing tool needs the ability to view the model from any angle, rather than a single pre-determined position.

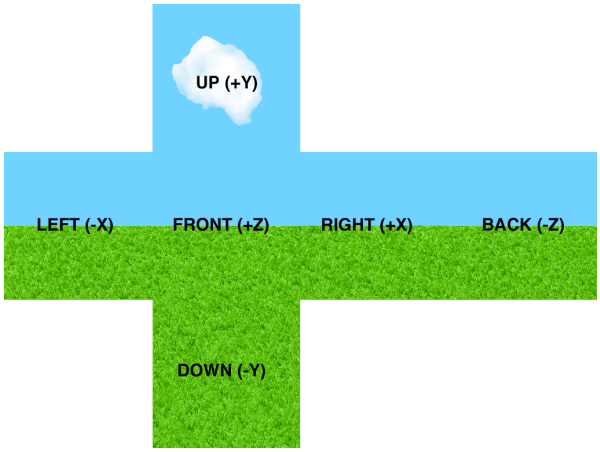
7.3. Environment Mapping

This section examines the resulting renders of reflective and refractive materials using two different types of environment mapping. Emphasis is placed on the mapping techniques, rather than the accuracy of the reflection and refraction calculations. For the comparative performance analysis, both applications render the refractive test sphere model.

7.3.1. Cube Map

Test Parameters:

Synthetic cube map



Captured cube map

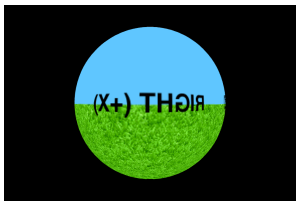


Fig. 27a: Test Sphere, Reflection (Synthetic)

Fig. 27b: Test Sphere, Reflection (Captured)

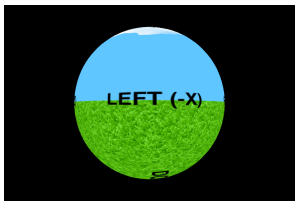


Fig. 27c: Test Sphere, Refraction (Synthetic)

Fig. 27d: Test Sphere, Refraction (Captured)

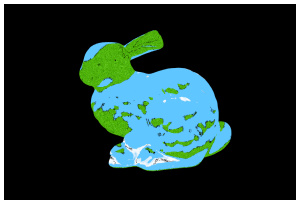


Fig. 27e: Stanford Bunny, Reflection (Synthetic)

Fig. 27f: Stanford Bunny, Reflection (Captured)

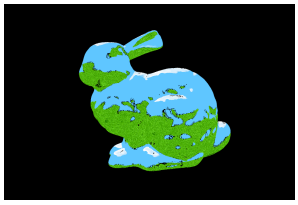


Fig. 27g: Stanford Bunny, Refraction (Synthetic)

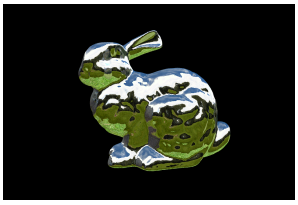


Fig. 27h: Stanford Bunny, Refraction (Captured)

The test spheres clearly show the seams between the captured cube map faces (Figs. 27b and 27d). A lot of time and effort from this project could have been devoted to creating a seamless cube map by implementing stitching algorithms directly [Brown and Lowe, 2003; 2007] or setting up a system to interface with remote applications such as *Autostitch* [Brown, 2011]. Both methods would considerably slow down the system pipeline and needlessly spend time solving a problem with an already well-known solution. Figs. 27f and 27h show that simply using an unstitched cube map produces very accurate preview-quality results, with less noticeable seams on rich models (such as the pictured Stanford bunny).

The cube map and its matrix transformations are highly responsive to device sensor data, resulting in an instant and accurate calculation of the camera orientation within the environment.

7.3.2. Live Feed

Test Parameters:
N/A

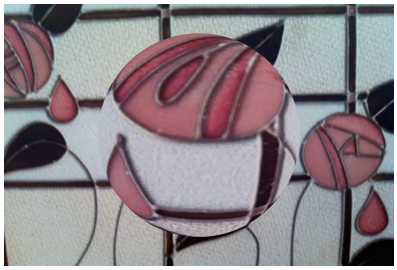


Fig. 28a: Test Sphere,
Live feed refraction (1)

Fig. 28b: Test Sphere,
Live feed refraction (2)

Fig. 28c: Test Sphere,
Live feed reflection

Fig. 28d: Mask,
Live feed refraction (1)

Fig. 28e: Mask,
Live feed refraction (2)

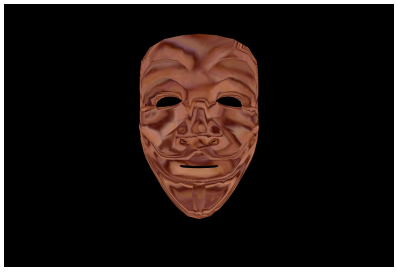


Fig. 28f: Mask,
Live feed reflection

Mapping refractions from a live environment produces a highly realistic effect that will composite exceptionally well for any model and background (Figs. 28a, 28b, 28d, and 28e). Live reflections (Figs. 28c and 28f) on the other hand, are a largely pointless feature for now due to the reduced quality of the front-facing camera and the inability to render a simultaneous view from the back-facing camera (i.e. the scene). However, the concept is there for the taking when future device improvements come along.

There is a slight visual limitation in the live feed feature though, only noticeable with certain setups. The mask model in Fig. 29a should be refracting light from the green and black bands in the background, but it isn't. This is because the sampling area for the live mapping doesn't contain these elements, as seen in Fig. 29b. Just like in the data capture stage, the sampled texture is taken from a cropped version of the camera view with dimensions 256x256 in order to comply with OpenGL ES requirements (square, power-of-two textures). A larger source image size 512x512 would result in even more inaccurate refractions, by over-sampling the camera view.

Fig. 29a: Live feed error

Fig. 29b: Live feed sampling area

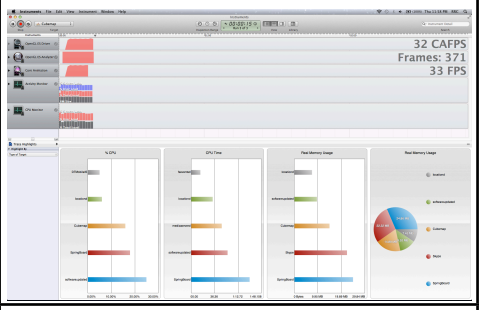
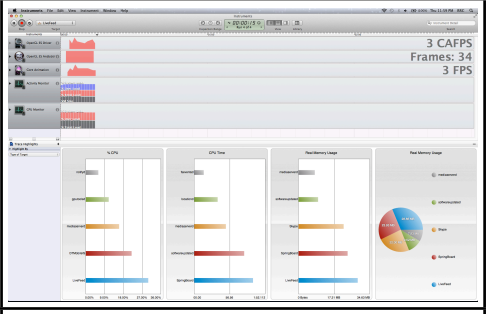
	Cube Map	Live Feed
Instruments **Profile Output**		
Rate	33 FPS	3 FPS
CPU	15%	30%
Memory	14.88 MB	28.86 MB

Fig. 30: Comparative performance analysis - Cube Map vs. Live Feed

Finally, a comparative performance analysis (Fig. 30) really highlights how costly live feed rendering is compared to its cube map counterpart. The task is twice as expensive in terms of memory and CPU consumption, and an astounding 11 times slower in the rendering rate - which may or may not be considered real-time performance, depending on the strictness of the term's definition.

7.4. Colour Correction

7.4.1. Intensity Adjustments

In this section, the test implementation renders a greyscale, untextured Gargoyle model. The scene needs some form of lighting/shading to perform properly, for which I have chosen hemisphere occlusion with ambient light. As usual, the hemisphere occlusion algorithm extracts its intensity priors from the environment.

Test Parameters:
Ambient: 1.00; Diffuse: 0.00; Specular: 0.00; Gloss: 1.0; Daylight: 1.00; Sunlight; 0.00
Hemisphere Occlusion: YES

High brightness/contrast

Low brightness/contrast

Normal brightness/contrast

Fig. 31a: Intensity (High), Adjustment value = 0.0

Fig. 31b: Intensity (High), Adjustment value = 0.5

Fig. 31c: Intensity (High), Adjustment value = 1.0

Fig. 31d: Intensity (Low), Adjustment value = 0.0

Fig. 31e: Intensity (Low), Adjustment value = 0.5

Fig. 31f: Intensity (Low), Adjustment value = 1.0

Fig. 31g: Intensity (Normal), Adjustment value = 0.0

Fig. 31h: Intensity (Normal), Adjustment value = 0.5

Fig. 31i: Intensity (Normal), Adjustment value = 1.0

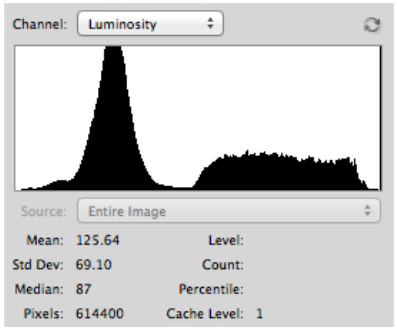


Fig. 31j: Histogram (Normal), σ = 69.10

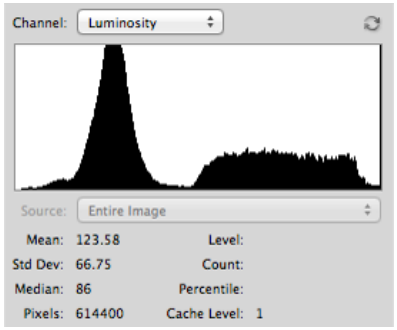


Fig. 31k: Histogram (Normal), σ = 66.75

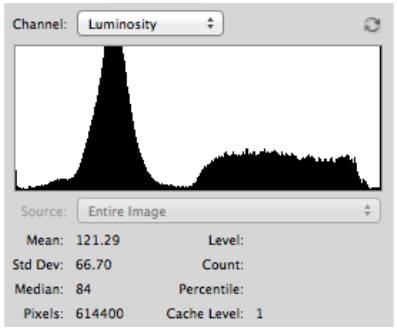


Fig. 31l: Histogram (Normal), σ = 66.70

In this section, both intensity adjustment algorithms (brightness and contrast) have been combined into a single procedure. The results are very pleasing, as the fully adjusted models (value = 1.0; Figs. 31c, 31f, and 31l) composite very well with the background and there is a clear difference between the three different environments. For the sake of brevity, only histograms for the normal environment are displayed, where the standard deviation value drops from 69.10 (no adjustment) to 66.70 (full adjustment) signifying a closer distribution of intensities.

7.4.2. Colour Matching

For the following results, the models have no shading or lighting associations and are simply coloured by their respective textures. The results have been analysed with the same histogram tool as the intensity adjustment algorithm but the full output has been omitted, including only the standard deviation, σ, for presentation purposes.

Test Parameters:

Blue / cyan saturation	Green / yellow saturation	Red / magenta saturation

Fig. 32a: Colour (Blue/cyan), Adjustment value = 0.0, σ = 79.16

Fig. 32b: Colour (Blue/cyan), Adjustment value = 0.5, σ = 78.73

Fig. 32c: Colour (Blue/cyan), Adjustment value = 1.0, σ = 78.47

Fig. 32d: Colour (Green/yellow), Adjustment value = 0.0, σ = 79.81

Fig. 32e: Colour (Green/yellow), Adjustment value = 0.5, σ = 79.40

Fig. 32f: Colour (Green/yellow), Adjustment value = 1.0, σ = 78.70

Fig. 32g: Colour (Red/magenta), Adjustment value = 0.0, σ = 84.54

Fig. 32h: Colour (Red/magenta), Adjustment value = 0.5, σ = 84.17

Fig. 32i: Colour (Red/magenta), Adjustment value = 1.0, σ = 83.99

The algorithm is successful in performing a correct colour transfer, with noticeable differences amongst the three environments. The standard deviation behaves as expected, decreasing in value as the colour distribution moves closer together (exhibiting the

appropriate inverse relationship with the adjustment value). However, these results aren't as rich as the ones presented by Reinhard et al. [2001; 2004] in their original publications, even with a full adjustment. Furthermore, the algorithm's results are less than satisfactory when applied to models with textures dominated by shades of a single colour (Figs. 33a-f). Due to OpenGL ES limitations, it's difficult to deduce where the problems exist since access to shader output data is strictly forbidden (the OpenGL ES server only delivers fully rendered images to the frame buffer).

Fig. 33a: Banana (Blue/cyan), Full adjustment

Fig. 33b: Banana (Green/yellow), Full adjustment

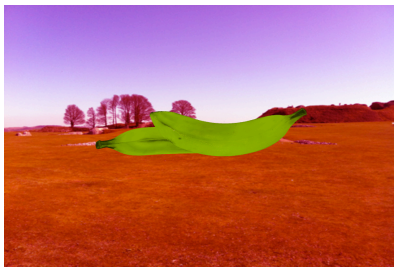


Fig. 33c: Banana (Red/magenta), Full adjustment

Fig. 33d: Hulk (Blue/cyan), Full adjustment

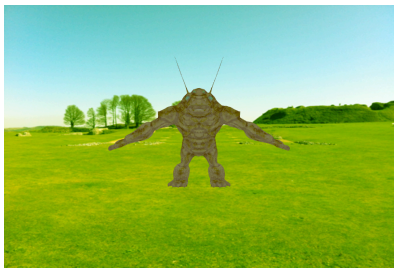


Fig. 33e: Hulk (Green/yellow), Full adjustment

Fig. 33f: Hulk (Red/magenta), Full adjustment

7.5. Full Implementation

To conclude the testing and results section, Figs. 34a-f showcase a selection of compositions using the full range of modelling and algorithm tools available to the application. Each image attempts to highlight a certain aspect of the implementation, but the ultimate goal is to create interesting preview-quality composites.

Fig. 34a: Park Bench,
Sunlight

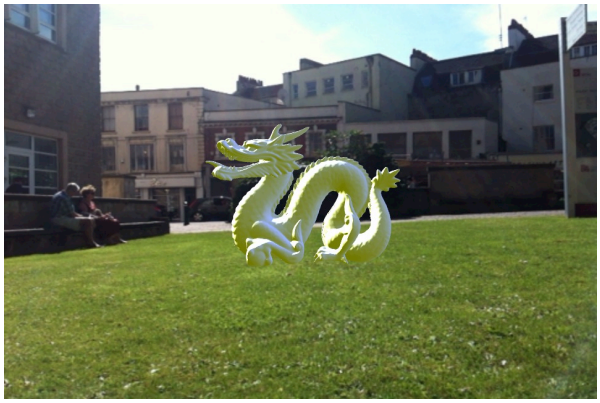


Fig. 34b: Dragon,
Hemisphere shading

Fig. 34c: Stanford Bunny,
Cube map reflection

Fig. 34d: V for Vendetta Guy Fawkes Mask,
Live feed refraction

Fig. 34e: Gargoyle,
Intensity adjustments

Fig. 34f: Honda Jazz,
Colour matching

8. Further Work

8.1. Attainable Extensions

The topics discussed in this section can extend the application without drastically modifying the current structure. They were not implemented in the final solution due to either limited time constraints or low degree of added value.

8.1.1. Texture Compression

Block compression with the PVRTC format [Imagination Technologies Limited, 2012] is the recommended approach to reducing texture memory usage. There is a lot of boilerplate code involved in the setup, but this format could boost performance of the live feed environment mapping function, bypassing the write/read cycle in generating a PNG image. Mipmaps could boost the quality (and performance) of the models by caching multiple textures of different dimensions for appropriate size sampling, depending on the current view.

8.1.2. Higher Resolution

The application currently renders the camera view at a resolution of 480x320, which is actually a quarter of the available display (960x640). Careful data management and performance optimisation could free-up processing bandwidth in order to accommodate the higher resolution. Special care needs to be taken though, since this would cause a knock-on effect on the rest of the application such as increased texture sizes of the environment (256x256 to 512x512).

8.1.3. Bump Mapping

Bump mapping is a texturing effect for rendering rough surfaces without affecting a model's geometry, only its appearance by modifying its normals. This technique is very useful for uneven material surfaces, such as wood or stone. This simply requires an additional 2D texture attachment in OpenGL ES and the accompanying shader code.

8.1.4. Multiple Materials

The application currently supports a single material for the whole model. This is a very basic and inflexible type of modelling which renders quite unrealistic objects (for example, a real car will have shiny rims with matte tires). An MTL file usually accompanies an OBJ file with per-face shading data. Behrens' python script [2012] only supports OBJ conversion to readable OpenGL ES header files, but a similar tool could be developed to parse MTL files, with a corresponding modification to the shader code.

8.2. Alternative Implementations

The methods featured in this section are not suitable for use within the current application and would require a massive concept, system, and implementation redesign. They would likely need both an extra post-processing stage and communication with a dedicated server for additional memory and processing power. The real-time characteristic would be all but lost, but the rendering and compositing would gain a massive boost in quality.

8.2.1. Ray Tracing

Ray tracing is common computer graphics techniques which generate images by tracing the path of light from pixels on a viewing plane to objects in a virtual scene. Ray casting performs the reverse procedure, casting rays from objects and tracing them through the virtual scene to the view plane.

The thoroughly discussed ambient occlusion algorithm is a form of ray tracing. A similar effect useful to the application is self-reflection with multiple depth passes. Ray-traced refractions create wonderful images by tracing the path of light all the way through a transparent object.

Ray casting can be used to create shadows, but in an AR application it would be very difficult to map these accurately onto real objects (and even more difficult to map real shadows onto virtual objects). Shadow detection/generation is a common and highly challenging problem in 2D scenes, but work in this field [Chuang et al., 2003; Guo et al., 2011] shows promising techniques for automated shadow matting relevant to compositing. Nonetheless, shadows could work really well on flat, clear foregrounds (e.g. an open field) adding realism to a scene by anchoring synthetic objects to the ground and transmitting size/depth.

8.2.2. Image-Based Techniques

Image-based techniques include modelling (IBM), lighting (IBL), and rendering (IBR), all of which have been discussed in Section 2.1. The common idea is that a set of 2D images is used to generate a 3D environment recreating the real world. This approach requires specialised equipment as well as a render farm, leaving the device to simply be used as a basic motion capture sensor for orientation and location data, annotating a large set of reference images.

8.2.3. Post-Processing

Image post-processing with shaders [Munshi, 2009] and [Rost and Licea-Kane, 2009] is the most attainable and least disruptive goal in this section. The current application would still remain as a real-time, context-aware *preview* tool, but the final composite algorithms would occur in a post-processing stage. If the preview results are rendered off-screen to a frame buffer OpenGL ES object, then kernel/convolution-based image-processing techniques can be implemented at the shader level to act on this data and then overlay the resulting render onto the camera view image. Image post-processing techniques that would benefit the composite include blurring and light blooming.

8.3. Technology Upgrades

This section takes a quick glance at the relevant features of the upcoming[43] iPhone 5, iOS 6 , OpenGL ES 3.0, and GLSL ES 3.0 technologies.

[43] At the time of this document's submission:
- The iPhone 5 and iOS 6 have just completed their worldwide release.
- The OpenGL ES 3.0 specification has been announced but not yet implemented.

8.3.1. iPhone 5 & iOS 6

The main upgrades of the new device include a larger display (640x1136), better camera quality (8 MP), increased processing power (dual-core CPU and triple-core GPU), and increased memory (1 GB RAM).

iOS 6 includes the new function *CVOpenGLESTextureCache* for fast-loading a camera view as an OpenGL ES texture (incredible news for the live feed algorithm). New camera APIs include exposure control, which could be very useful for HDR image capture.

8.3.2. OpenGL ES 3.0 & GLSL ES 3.0

OpenGL ES 3.0 mainly upgrades texture-centric features such as a standard compression function, non-power-of-two compatibility, and spherical maps (useful for most IBL methods). It also implements occlusion queries for minimising function calls and computing time.

GLSL ES 3.0 includes full support for 32-bit integer and floating point operations, removing many previous shader limitations.

9. Conclusions

The device has been investigated to its limit and the resulting tool succeeds in delivering preview-quality composites rendered in real-time, using a direct approach with minimal user input. The novel solution bypasses traditional compositing techniques and effectively combines all relevant sensor, user, and model data into an efficient system. The two new concepts developed - *colour priors* and *hemisphere occlusion* - will hopefully become a useful point of reference for future projects in this relatively new field of context-aware compositing. Similarly, this thesis will hopefully serve as a useful literature survey on the current state of compositing on embedded systems compared to desktop PCs.

On a personal note, this has been my most rewarding experience in computer science and largest achievement in the field. With the new iOS 6 and iPhone 5 just released, a whole world of Android devices to explore, an additional post-processing phase, and time on my hands, all I can say is let's keep going!

APPENDIX A: Index of iOS Frameworks Used

The following table lists the proprietary iOS 5 frameworks included in the implementation. They are accompanied with a brief description of their use within the final application [Apple Inc., 2012].

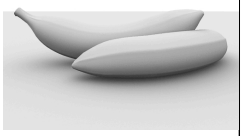
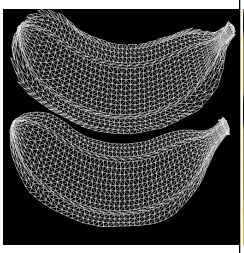
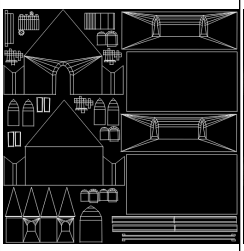
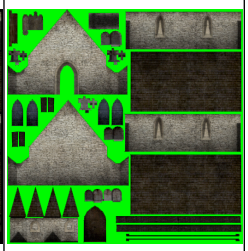
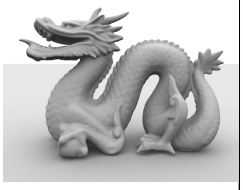
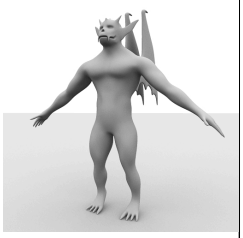
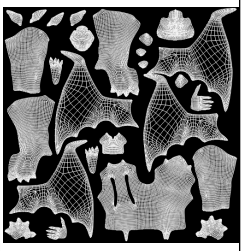
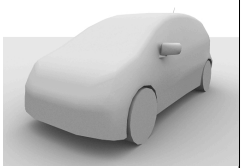
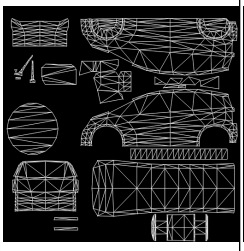
Framework	Description
AVFoundation	Contains Objective-C interfaces for playing and recording audio and video.
CoreGraphics	Contains the interfaces for Quartz 2D.
CoreImage	Contains interfaces for manipulating video and still images.
CoreLocation	Contains the interfaces for determining the user's location.
CoreMedia	Contains low-level routines for manipulating audio and video.
CoreMotion	Contains interfaces for accessing accelerometer and gyro data.
CoreVideo	Contains low-level routines for manipulating audio and video. Do not use this framework directly.
Foundation	Contains interfaces for managing strings, collections, and other low-level data types.
GLKit	Contains Objective-C utility classes for building complex OpenGL ES applications.
OpenGLES	Contains the interfaces for OpenGL ES, which is an embedded version of the OpenGL cross-platform 2D and 3D graphics rendering library.
UIKit	Contains classes and methods for the iOS application user interface layer.

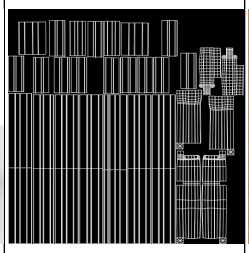
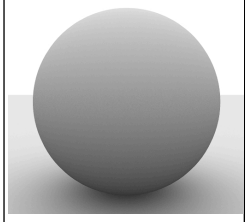
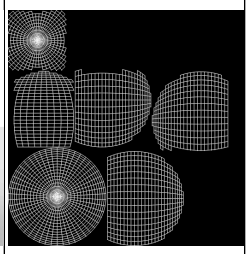
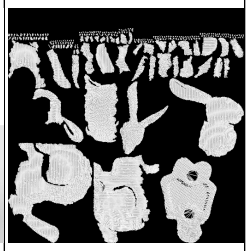
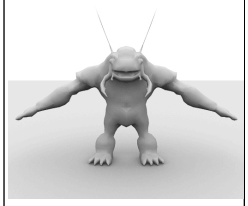
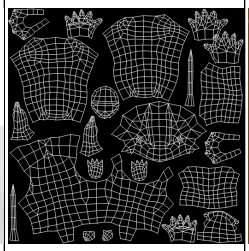
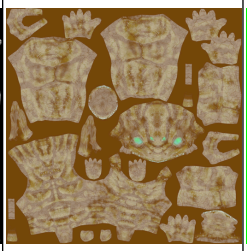
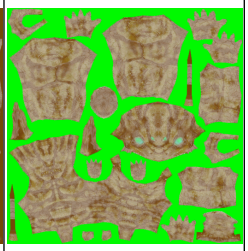
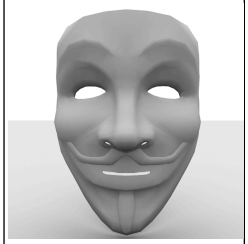
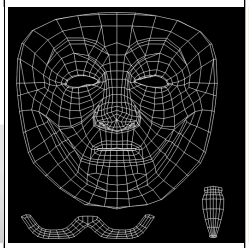
APPENDIX B: Index of GLSL Functions Used

The following table lists the built-in GLSL functions called in the implementation. They are accompanied with a brief description of their use within the final shaders [Khronos Group, 2012]. This list excludes calls to basic maths functions such as *max(x, y)* and *pow(x, y)*.

Function	Description
dot(x, y)	Returns the dot product of *x* and *y*.
mix(x, y, a)	Returns the linear blend of *x* and *y* using the value of *a*.
normalize(x)	Returns a vector in the same direction as *x,* but with a length of 1.0.
reflect(I, N)	For the incident vector *I* and surface normal *N*, returns the reflection direction.
refract(I, N, eta)	For the incident vector *I,* surface normal N, and the refractive index *eta*, returns the refraction direction.
smoothstep(edge1, edge2, x)	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0.0 and 1.0 when *edge0* < *x* < *edge1*.
texture2D(sampler, coord)	Uses the texture coordinate *coord* to access the 2D texture currently specified by *sampler*.
textureCube(sampler, coord)	Uses the texture coordinate *coord* to access the cube map texture currently specified by *sampler*. The direction of *coord* selects the face in which to do a two-dimensional texture lookup.

APPENDIX C: Model Specifications

Info.	Model	UV Texture Map	Original Texture	Edited Texture
Banana Vertices: 4032 Faces: 8056 Normals: 4032 Coords.: 4420 Size: 4.7 MB Source: Behrens [2012]				
Chapel Vertices: 484 Faces: 878 Normals: 1828 Coords.: 826 Size: 474 KB Source: Gerzi 3D ART [2009]				
Dragon Vertices: 50000 Faces: 100000 Normals: 300000 Coords.: 71706 Size: 61.7 MB Source: Stanford University Computer Graphics Laboratory [2011]			N/A	N/A
Gargoyle Vertices: 27178 Faces: 54352 Normals: 27178 Coords.: 29427 Size: 35.7 MB Source: Veleran [2011a]			N/A	N/A
Honda Jazz Vertices: 650 Faces: 1259 Normals: 3777 Coords.: 1760 Size: 661 KB Source: Vladimiroquai [2010]				

Info.	Model	UV Texture Map	Original Texture	Edited Texture
Park Bench Vertices: 1454 Faces: 2420 Normals: 2758 Coords.: 1699 Size: 1.4 MB Source: DenisVFX [2012]				
Sphere Vertices: 2452 Faces: 4900 Normals: 2452 Coords.: 2681 Size: 3 MB Source: Original model			N/A	N/A
Stanford Bunny Vertices: 34835 Faces: 69666 Normals: 34835 Coords.: 39088 Size: 42.2 MB Source: Stanford University Computer Graphics Laboratory [2011]			N/A	N/A
Umber Hulk Vertices: 1527 Faces: 3050 Normals: 6274 Coords.: 1944 Size: 2 MB Source: Veleran [2011b]				
V for Vendetta Guy Fawkes Mask Vertices: 936 Faces: 1693 Normals: 3412 Coords.: 936 Size: 1 MB Source: Ghonma [2011]				

```
// APPENDIX D: Vertex Shader Source Code

// VERTEX SHADER
static const char* ShaderVS = STRINGIFY
(

// Attribute variables are typically changed per vertex
attribute vec4 aVertex;
attribute vec3 aNormal;
attribute vec2 aTexture;

// Uniform variables are changed at most once per vertex
uniform mat4 uProjectionMatrix;
uniform mat4 uModelviewMatrix;
uniform mat3 uNormalMatrix;
uniform mat3 uEnvironmentMatrix;

uniform float uRefraction;

// OUTPUTS to the Fragment Shader
varying vec3 vNormal;
varying vec2 vTexture;
varying vec3 vReflection;
varying vec3 vRefraction;

void main(void)
{
    vNormal = uNormalMatrix * aNormal;
    vTexture = aTexture;

    // Compute reflection
    vReflection = uEnvironmentMatrix * reflect(aVertex.xyz, aNormal);

    // Compute refraction
    vec3 refraction = uEnvironmentMatrix * refract(aVertex.xyz, aNormal,
            uRefraction);
    refraction = vec3(uProjectionMatrix * vec4(refraction, 0.0));
    vRefraction = refraction;

    // Every vertex shader must output a position into the gl_Position
            variable
    // This defines the position that is passed through to the next stage
            of the pipeline
    gl_Position = uProjectionMatrix * uModelviewMatrix * aVertex;
}

);
```

```
// APPENDIX E: Fragment Shader Source Code

// FRAGMENT SHADER
static const char* ShaderFS = STRINGIFY
(

// INPUTS from the Vertex Shader
varying mediump vec3 vNormal;
varying mediump vec2 vTexture;
varying mediump vec3 vReflection;
varying mediump vec3 vRefraction;

// Uniform variables are changed at most once per vertex
uniform highp vec3 uSunPosition;
uniform highp vec3 uCameraPosition;
uniform highp vec3 uCameraTilt;

uniform highp float uMaterialAmbient;
uniform highp float uMaterialDiffuse;
uniform highp float uMaterialSpecular;
uniform highp float uMaterialGloss;

uniform highp float uIntensityDay;
uniform highp float uIntensitySun;

uniform highp vec3 uHemisphereGround;
uniform highp vec3 uHemisphereSky;
uniform highp float uHemisphereMix;
uniform highp float uHemisphereOcclusionL;
uniform highp float uHemisphereOcclusionH;

uniform highp float uReflection;

uniform highp vec3 uEnvironmentGS;
uniform highp vec3 uEnvironmentM;
uniform highp vec3 uEnvironmentSD;
uniform highp vec3 uModelM;
uniform highp vec3 uModelSD;
uniform highp float uCC;

uniform highp float uMaterialMode;
uniform highp float uTextureMode;
uniform highp float uModelMode;

uniform sampler2D uTexture;
uniform samplerCube uCubemap;

highp vec3 rgbTOlab(highp vec3 rgb)
{
    rgb = rgb * 255.0;

    highp mat3 convert =
    mat3
    (
     0.3475, 0.2162, 0.1304,
     0.8231, 0.4316, -0.1033,
```

```
     0.5559, -0.6411, -0.0269
    );

    highp vec3 lab = convert * rgb;

    return lab;
}

highp vec3 labTOrgb(highp vec3 lab)
{
    highp mat3 convert =
    mat3
    (
     0.5773, 0.5774, 0.5832,
     0.2621, 0.6072, -1.0627,
     5.6947, -2.5444, 0.2073
    );

    highp vec3 rgb = convert * lab;

    rgb = rgb / 255.0;

    return rgb;
}

highp vec3 correctColor(highp vec3 inputColor)
{
    highp vec3 lab = rgbTOlab(inputColor);

    highp float l1 = lab.r - uModelM.r;
    highp float a1 = lab.g - uModelM.g;
    highp float b1 = lab.b - uModelM.b;

    highp float l2 = (uModelSD.r / uEnvironmentSD.r) * l1;
    highp float a2 = (uModelSD.g / uEnvironmentSD.g) * a1;
    highp float b2 = (uModelSD.b / uEnvironmentSD.b) * b1;

    highp float l3 = l2 + uEnvironmentM.r;
    highp float a3 = a2 + uEnvironmentM.g;
    highp float b3 = b2 + uEnvironmentM.b;

    highp vec3 rgb = labTOrgb(vec3(l3, a3, b3));

    highp vec3 outputColor = rgb;

    outputColor = mix(inputColor, outputColor, uCC);

    return outputColor;
}

highp vec3 correctGS(highp vec3 inputColor)
{
    // Greyscale w/ Luma coefficients
    highp float colorGS = (inputColor.r * 0.2126) + (inputColor.g * 0.7152)
            + (inputColor.b * 0.0722);
    colorGS = smoothstep(uEnvironmentGS.x, uEnvironmentGS.z, colorGS);
```

```
    // Brightness
    highp float brightness = mix(uEnvironmentGS.x, uEnvironmentGS.z, colorGS
            );

    // Contrast
    highp float contrast = 0.0;
    highp float range = 0.0;
    highp float value = 0.0;

    if(colorGS > uEnvironmentGS.y)
    {
        range = uEnvironmentGS.z - uEnvironmentGS.y;
        value = colorGS - uEnvironmentGS.y;
        contrast = mix(uEnvironmentGS.y, uEnvironmentGS.z, (value / range));
    }

    else
    {
        range = uEnvironmentGS.y - uEnvironmentGS.x;
        value = colorGS - uEnvironmentGS.x;
        contrast = mix(uEnvironmentGS.x, uEnvironmentGS.y, (value / range));
    }

    highp vec3 colorBrightness = inputColor * brightness;
    highp vec3 colorContrast = inputColor * contrast;

    highp vec3 outputColor = mix(colorBrightness, colorContrast, 0.5);
    outputColor = mix(inputColor, outputColor, uCC);

    return outputColor;
}


highp vec3 opaqueColor(void)
{
    lowp vec3 colorOpaque = vec3(0.0);
    lowp vec3 colorAmbient = vec3(uMaterialAmbient);
    lowp vec3 colorDiffuse = vec3(uMaterialDiffuse);
    lowp vec3 colorSpecular = vec3(uMaterialSpecular);
    lowp vec3 colorOcclusionL = vec3(uHemisphereOcclusionL);
    lowp vec3 colorOcclusionH = vec3(uHemisphereOcclusionH);

    // Light model
    highp vec3 N = vNormal;                     // Surface normal
    highp vec3 L = normalize(uSunPosition);     // Sun position
    highp vec3 E = normalize(uCameraPosition);  // Camera position
    highp vec3 H = normalize(L + E);            // Normalized half-plane
            vector
    highp vec3 Ct = normalize(uCameraTilt);     // Camera tilt (X-axis
            rotation)

    // Ambient lighting
    highp float costheta = dot(N, Ct);
    highp float a = 0.5 + (0.5 * costheta);
    highp vec3 hemisphere = mix(uHemisphereGround, uHemisphereSky, a);  //
```

```
        Hemisphere lighting
    highp vec3 occlusion = mix(colorOcclusionL, colorOcclusionH, a);    //
        Hemisphere occlusion
    highp vec3 af = mix(occlusion, hemisphere, uHemisphereMix);
    highp vec3 ambient = mix(colorAmbient, af, uCC);

    // Diffuse lighting
    highp float df = max(0.0, dot(N, L));   // One-sided lighting

    // Specular lighting
    highp float sf = max(0.0, dot(N, H));   // One-sided lighting
    sf = pow(sf, uMaterialGloss);

    // Intensity adjustments
    lowp vec3 colorBase = (ambient * uIntensityDay) + (((df * colorDiffuse)
        + (sf * colorSpecular)) * uIntensitySun);

    // Cube map
    if(uTextureMode > 0.0)
    {
        // Texture color
        lowp vec3 colorTexture = vec3(texture2D(uTexture, vTexture));

        // Model color
        lowp vec3 colorModel = colorBase * colorTexture;

        // Reflection color
        lowp vec3 colorCubemap = vec3(textureCube(uCubemap, vReflection));
        lowp vec3 colorReflection = colorModel * colorCubemap;

        colorOpaque = mix(colorModel, colorReflection, uReflection);
    }

    // Live feed
    else
    {
        lowp vec2 conversion = vReflection.xy * 0.5 + 0.5;
        lowp vec3 colorLive = vec3(texture2D(uTexture, conversion));
        lowp vec3 colorReflection = colorBase * colorLive;
        colorOpaque = colorReflection;
    }

    return colorOpaque;
}

highp vec3 transparentColor(void)
{
    lowp vec3 colorTransparent = vec3(0.0);

    // Light model
    highp vec3 N = vNormal;                     // Surface normal
    highp vec3 L = normalize(uSunPosition);     // Sun position
    highp vec3 E = normalize(uCameraPosition);  // Camera position
    highp vec3 H = normalize(L + E);            // Normalized half-plane
        vector
```

```
    // Specular highlights
    highp float sf = dot(N, H);                 // Two-sided lighting
    sf = pow(sf, uMaterialGloss);

    // Refraction color
    lowp vec3 colorRefraction = vec3(0.0);

    // Cube map
    if(uTextureMode > 0.0)
    {
        lowp vec3 colorCubemap = vec3(textureCube(uCubemap, vRefraction));
        colorRefraction = (colorCubemap + sf);
    }

    // Live feed
    else
    {
        lowp vec2 conversion = vRefraction.xy * 0.5 + 0.5;
        lowp vec3 colorLive = vec3(texture2D(uTexture, conversion));
        colorRefraction = (colorLive + sf);
    }

    colorTransparent = colorRefraction;

    return colorTransparent;
}

void main(void)
{
    lowp vec3 color = vec3(0.0);

    // Reflect / Refract
    if(uMaterialMode > 0.0)
        color = opaqueColor();
    else
        color = transparentColor();

    if(uTextureMode > 0.0)
    {
        // Non-Textured
        if(uModelMode > 0.0)
            color = correctGS(color);
        // Textured
        else
            color = correctColor(color);
    }

    // The gl_FragColor variable contains the final output color for the
            fragment shader
    gl_FragColor = vec4(color, 1.0);
}

);
```

BIBLIOGRAPHY

Literature References

1. Alnasser, M. and Foroosh, H. (2006) "Image-based rendering of synthetic diffuse objects in natural scenes." In **Proceedings of the 18th International Conference on Pattern Recognition. Hong Kong, 20-24 August 2006**. Washington, DC: IEEE Computer Society. pp. 787-790
2. Alveston Bristol UK Weather (2012) **Alveston Bristol UK Weather** [online]. Available from: http://www.alvestonweather.co.uk/wxtempsummary.php? [Accessed July - September 2012]
3. Baker, M.J. (2012) **Euclidean Space - Building a 3D World** [online]. Available from: http://www.euclideanspace.com/ [Accessed July - September 2012]
4. Benford, S., Schnadelbach, H. and Koleva, B. et al . (2003) Sensible, sensable and desirable: a framework for designing physical interfaces. **Technical Report Equator**, 03-003
5. Blinn, J.F. (1977) "Models of light reflection for computer synthesized pictures." In **Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques.** New York, NY: ACM. pp. 192-198
6. Bomfin, D. (2011) **Cameras on OpenGL ES 2.x - The ModelViewProjection Matrix** [online]. Available from: http://db-in.com/blog/2011/04/cameras-on-opengl-es-2-x/ [Accessed July - September 2012]
7. Brinkmann, R. (2008) **The Art and Science of Digital Compositing**. Second Edition. Burlington, MA: Morgan Kaufmann Publishers.
8. Brown, M. and Lowe, D. (2003) "Recognising panoramas" In **Proceeding of the Ninth IEEE International Conference on Computer Vision. Vancouver, 13-16 October 2003.** Washington: IEEE Computer Society. pp. 1218-1227
9. Brown, M. and Lowe, D. (2007) Automatic panoramic image stitching using invariant features. **International Journal of Computer Vision**, 74(1): 59-73
10. Buck, E.M. (2012) **Learning OpenGL ES for iOS**. Upper Saddle River, NJ: Addison-Wesley.
11. Caramba App Development (2012) **Field of View Angles for iPad/iPhone** [online]. Available from: http://www.caramba-apps.com/blog/files/field-of-view-angles-ipad-iphone.html [Accessed July 2012]
12. Chuang, Y.Y., Goldman, D.B. and Curless, B. et al. (2003) Shadow matting and compositing. **ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003**, 22 (3): 494-500
13. Coast Guard Navigation Service (2011) **GPS Accuracy** [online]. Available from: http://www.gps.gov/systems/gps/performance/accuracy/ [Accessed May 2012]
14. Debevec, P. (1998) "Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography" In **Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques. Orlando, 19-24 July 1998.** New York, NY: ACM. pp. 189-198
15. Debevec, P. (2012) **Paul Debevec Home Page** [online]. Available from: http://www.pauldebevec.com/ [Accessed April - September 2012]
16. Dobashi, Y., Yamamoto, T. and Nishita, T. (2002) "Interactive rendering of atmospheric scattering effects using graphics hardware" In **Proceedings of the ACM SIGGRAPH/**

EUROGRAPHICS Conference on Graphics Hardware. Saarbrucken, 1-2 September 2002. Switzerland: Eurographics Association. pp. 99-107
17. Guo, R., Dai, Q. and Hoiem, D. (2011) "Single-image shadow detection and removal using paired regions" In **Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition. Colorado Springs, 20-25 June 2011.** Washington: IEEE Computer Society. pp. 2033-2040
18. Lengyel, E. (2004) **Mathematics for 3D Game Programming and Computer Graphics**. Hingham, MA: Charles River Media, Inc.
19. Lumo, F. (2010) **Apple iPhone 4 Camera Specs** [online]. Available from: http://falklumo.blogspot.co.uk/2010/06/apple-iphone-4-camera-specs.html [Accessed July 2012]
20. Haeberli, P. and Voorhies, D. (1994) Image processing by linear interpolation and extrapolation. **IRIS Universe Magazine, Silicon Graphics**, August 1994 (28)
21. Hakansson, M., Gaye, L. and Ljungblad, S. et al. (2006) "More than meets the eye: an exploratory study of context photography" In **Proceedings of the 4th Nordic Conference on Human-Computer Interaction: Changing Roles. Oslo, 14-18 August 2006.** New York, NY: ACM. pp. 262-271
22. Halawani, S.M. and Sunar, M.S. (2010) "Interaction between sunlight and the sky colour with 3D objects in the outdoor virtual environment." In **Proceedings of the 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation. Kota Kinabalu, 26 May 2010**. Washington, DC: IEEE Computer Society. pp. 470-475
23. Helland, T. (2012) **Seven Grayscale Conversion Algorithms** [online]. Available from: http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/ [Accessed August - September 2012]
24. International Telecommunication Union (2009) Parameter values for the HDTV standards for production an international programme exchange. **Recommendation ITU-R BT.709-5**, 04/2002. International Telecommunication Union
25. Karsch, K., Hedau, V. and Forsyth, D. et al. (2011) Rendering synthetic objects into legacy photographs. **ACM Transaction on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2011**, 30 (6): 157:1-157:12
26. Kee, E. and Farid, H. (2010) "Exposing digital forgeries from 3-D lighting environments" In **Proceeding of the 2010 IEEE International Workshop on Information Forensics and Security. Seattle, 12-15 December 2010.** Washington, DC: IEEE Computer Society. pp. 1-6
27. Ljungblad, S., Hakansson, M. and Gaye, L. et al. (2004) "Context photography: modifying the digital camera into a new creative tool" In **Conference on Human Factors in Computing Systems. Vienna, 24-29 April 2004.** New York, NY: ACM. pp. 1191-1194
28. Lalonde, J.F. and Efros, A.A. (2007) "Using color compatibility for assessing image realism" In **Proceedings of the 11th IEEE International Conference on Computer Vision. Rio de Janeiro, 14-20 October 2007.** Washington, DC: IEEE Computer Society. pp. 1-8
29. Lalonde, J.F., Narasimhan, S.G. and Efros, A.A. (2010) What do the sun and sky tell us about the camera? **International Journal of Computer Vision**, 88 (1): 24-51
30. Munshi, A., Ginsburg, D. and Shreiner, D. (2009) **OpenGL ES 2.0 Programming Guide**. Upper Saddle River, NJ: Addison-Wesley
31. Phong, B.T. (1975) Illumination for computer generated pictures. **Communications of the ACM**, 18 (6): 311-317

32. Preetham, A.J., Shirley, P. and Smits, B. (1999) "A practical analytic model for daylight." In **Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques. Los Angeles, 8-13 August 1999**. New York, NY ACM Press / Addison-Wesley Publishing Co. pp. 91-100
33. Ramamoorthi, R. and Hanrahan, P. (2001) "An efficient representation for irradiance environment maps." In **Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques. Los Angeles, 12-17 August 2001**. New York, NY: ACM Press. pp. 497-500
34. Reinhard, E., Ashikhmin, M. and Gooch, B. et al. (2001) Color transfer between images. **IEEE Computer Graphics and Applications**, 21(5): 34-41
35. Reinhard, E., Akyuz, A. and Colbert, M. et al. (2004) Real-time color blending of rendered and captured video. **The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)**, 2004: 1502:1-1502:9
36. Rendon Cepeda, R. (2012) **Context-Aware Compositing**. Research Review, University of Bristol
37. Rideout, P. (2010) **iPhone 3D Programming**. Sebastopol, CA: O'Reilly Media, Inc.
38. Rost, R.J. and Licea-Kane, B. (2009) **OpenGL Shading Language**. Third Edition. Upper Saddle River, NJ: Addison-Wesley
39. Seifert, K. and Camacho, O. (2007) Implementing positioning algorithms using accelerometers. **Freescale Semiconductor Application Note**, 02-007. Freescale Semiconductor, Inc.
40. Snavely, N., Seitz, S. and Szeliski, R. (2006) Photo tourism: exploring photo collections in 3D. **ACM Transaction on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2006**, 25 (3): 835-846
41. Suffern, K. (2007) **Ray Tracing from the Ground Up**. Wellesley, MA: A K Peters, Ltd.
42. Willis, P. (2007) "Generalised compositing" In **Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia. Perth, 1-4 December 2007.** New York, NY: ACM. pp. 129-134
43. Wright, S. (2010) **Digital Compositing for Film and Video**. Third Edition. Burlington, MA: Focal Press.

Media & Software References

1. Adobe Systems Incorporated (2012) **Image Editor Software | Adobe Photoshop** [online]. Available from: http://www.adobe.com/products/photoshop.html [Accessed October 2011]
2. Angisoft (2009) **iGPUTrace** [online]. Available from: http://itunes.apple.com/us/app/igputrace/id322063527?mt=8 [Accessed July - September 2012]
3. Angisoft (2011) **Quaternion Julia Raytracer** [online]. Available from: http://itunes.apple.com/us/app/quaternion-julia-raytracer/id321524703?mt=8 [Accessed July - September 2012]
4. Apple Inc. (2012a) **iOS Dev Center - Apple Developer** [online]. Available from: https://developer.apple.com/devcenter/ios/index.action [Accessed June - September 2012]
5. Apple Inc. (2012b) **Xcode 4 Downloads and Resources - Apple Developer** [online]. Available from: https://developer.apple.com/xcode/ [Accessed June - September 2012]
6. Apposing (2012) **CSL Sofas** [online]. Available from: http://itunes.apple.com/gb/app/csl-sofas/id489696164?mt=8 [Accessed July - September 2012]
7. Autodesk, Inc. (2012) **Maya - 3D Animation Software** [online]. Available from: http://usa.autodesk.com/maya/ [Accessed October 2011]

8. Behrens, H. (2012) **Obj2opengl: Convert Obj 3D Models to Arrays Compatible with iPhone OpenGL ES** [online]. Available from: http://heikobehrens.net/2009/08/27/obj2opengl/ [Accessed July 2012]
9. Brown, M. (2011) **Autostitch** [online]. Available from: http://www.cs.bath.ac.uk/brown/autostitch/autostitch.html [Accessed May 2012]
10. Deltatre (2012) **Deltatre** [online]. Available from: http://www.deltatre.com/ [Accessed September 2012]
11. DenisVFX (2012) **Park Bench** [online]. Available from: http://www.turbosquid.com/FullPreview/Index.cfm/ID/649163 [Accessed August 2012]
12. Digital Domain Productions, Inc. (2010) **Amazing Demonstration of the Virtual Camera Invented for Avatar** [online]. Available from: http://www.youtube.com/watch?v=Wy7IvrizRok [Accessed September 2012]
13. Digital Domain Productions, Inc. (2012) **Digital Domain Productions** [online]. Available from: http://digitaldomain.com/ [Accessed September 2012]
14. Gerzi 3D ART (2009) **Chapel** [online]. Available from: http://www.turbosquid.com/FullPreview/Index.cfm/ID/464431 [Accessed August 2012]
15. Ghonma (2011) **V for Vendetta Guy Fawkes Mask** [online]. Available from: http://thefree3dmodels.com/stuff/accessories/v_for_vendetta_guy_fawkes_mask/21-1-0-345 [Accessed August 2012]
16. Imagination Technologies Limited (2012) **PowerVR Insider SDK** [online]. Available from: http://www.imgtec.com/powervr/insider/powervr-sdk-docs.asp [Accessed July - September 2012]
17. Industrial Light & Magic (2011) **ILM's Virtual Camera that "Filmed" Rango** [online]. Available from: http://www.youtube.com/watch?v=evmBM6oOwok [Accessed September 2012]
18. Industrial Light & Magic (2012) **Industrial Light & Magic** [online]. Available from: http://www.ilm.com/ [Accessed September 2012]
19. Khronos Group (2012) **OpenGL ES 2_X - The Standard for Embedded Accelerated 3D Graphics** [online]. Available from: http://www.khronos.org/opengles/2_X/ [Accessed June - September 2012]
20. MEDL Mobile (2012) **My Wild Night With Ted - Ted the Movie** [online]. Available from: http://itunes.apple.com/gb/app/my-wild-night-ted-ted-movie/id540457675?mt=8 [Accessed July - September 2012]
21. Microsoft Corporation (2012) **Photosynth - Capture Your World in 3D** [online] Available from: http://photosynth.net/ [Accessed May 2012]
22. Motion Analysis (2012a) **BBC and TV2 Install Motion Analysis' CamTrak Systems** [online]. Available from: http://www.motionanalysis.com/html/temp/BBC_Sport.html [Accessed September 2012]
23. Motion Analysis (2012b) **CamTrak** [online]. Available from: http://www.motionanalysis.com/html/animation/camtrak.html [Accessed September 2012]
24. National Renewable Energy Laboratory (2003) **Solar Position Algorithm (SPA)** [online]. Available from: http://rredc.nrel.gov/solar/codesandalgorithms/spa/ [Accessed June - July 2012]
25. NVIDIA ARC GmbH (2012) **NVIDIA Advanced Rendering: Mental Ray** [online]. Available from: http://www.mentalimages.com/products/mental-ray/ [Accessed October 2011]
26. Stanford University Computer Graphics Laboratory (2011) **The Stanford 3D Scanning Repository** [online]. Available from: http://graphics.stanford.edu/data/3Dscanrep/ [Accessed June 2012]

27. USC Institute for Creative Technologies (2012) **HDR Shop** [online]. Available from: http://www.hdrshop.com/ [Accessed August - September 2012]
28. Veleran (2011a) **Gargoyle 1** [online]. Available from: http://www.turbosquid.com/FullPreview/Index.cfm/ID/624130 [Accessed August 2012]
29. Veleran (2011b) **Umber Hulk 1** [online]. Available from: http://www.turbosquid.com/FullPreview/Index.cfm/ID/624128 [Accessed August 2012]
30. Vizrt (2012) **Viz Virtual Studio** [online]. Available from: http://www.vizrt.com/products/viz_virtual_studio/ [Accessed September 2012]
31. Vladimiroquai (2010) **Honda Jazz** [online]. Available from: http://www.turbosquid.com/FullPreview/Index.cfm/ID/529737 [Accessed August 2012]

Additional Sources

The following is a list of supplementary material consulted throughout the project, but not cited in the thesis.

A. Baranski, S., Gundersen, J. and Hollemans, M. et al. (2011) **iOS 5 by Tutorials** [online]. Razeware LLC. Available from: http://www.raywenderlich.com/store/ios-5-by-tutorials [Accessed July 2012]
B. Hegarty, P. (2011) **iPad and iPhone Application Development** [online]. Available from: http://itunes.apple.com/itunes-u/ipad-iphone-application-development/id473757255?mt=10 [Accessed July - September 2012]
C. Larson, B (2010) **Advanced iPhone Development** [online]. Available from: http://itunes.apple.com/gb/itunes-u/advanced-iphone-development/id407243028 [Accessed July - September 2012]
D. Previsualization Society (2012) **The Previsualization Society** [online]. Available from: http://previssociety.com/ [Accessed August - September 2012]
E. Roche, K. (2011) **Pro iOS 5 Augmented Reality**. New York, NY: Springer Science + Business Media, LLC.
F. Smithwick, M. (2011) **Pro OpenGL ES for iOS**. New York, NY: Springer Science + Business Media, LLC.
G. Wikibooks (2012) **Movie Making Manual** [online]. Available from: http://en.wikibooks.org/wiki/Movie_Making_Manual [Accessed May - September 2012]

GLOSSARY: Acronyms & Abbreviations

Term	Definition
AR	Augmented Reality
CGI	Computer-Generated Imagery
CIELAB	*Commission Internationale de L'Eclairage* L$\alpha\beta$.
FPS	Frames Per Second
GLSL	OpenGL Shading Language
GPS	Global Positioning System
HDR	High Dynamic Range Imaging
IBL	Image-Based Lighting
IBM	Image-Based Modelling
IBR	Image-Based Rendering
iOS	iPhone Operating System
L$\alpha\beta$	Lightness (L) and colour-opponent (A,B) dimensions
MP	Mega Pixels (1 MP = 1 Million Pixels)
MTL	Material Template Library
OBJ	Wavefront .obj file
OpenGL	Open Graphics Library
OpenGL ES	OpenGL for Embedded Systems
PNG	Portable Network Graphics
PPI	Pixels Per Inch
PVRTC	PowerVR Texture Compression
RGB	Red-Green-Blue
SPA	Solar Position Algorithm
UV	Denotes the axes of a 2D texture