# Illusion SDK: An Augmented Reality Engine for Flash 11

by

Joseph Howse

Submitted in partial fulfilment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
November 2012

© Copyright by Joseph Howse, 2012

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled “Illusion SDK: An Augmented Reality Engine for Flash 11” by Joseph Howse in partial fulfilment of the requirements for the degree of Master of Computer Science.

Dated: November 20, 2012

Supervisor: _____

Readers: _____

_____

DALHOUSIE UNIVERSITY

DATE: November 20, 2012

AUTHOR: Joseph Howse

TITLE: Illusion SDK: An Augmented Reality Engine for Flash 11

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: MCSc CONVOCATION: May YEAR: 2013

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions. I understand that my thesis will be electronically available to the public.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests that permission has been obtained for the use of any copyrighted material appearing in the thesis (other than the brief excerpts requiring only proper acknowledgement in scholarly writing), and that all such use is clearly acknowledged.

_____

Signature of Author

For the inspirational Dr. L. S. River, and Nummists everywhere.
Visit http://nummist.com/.

**Table of Contents**

**List of Tables**

**List of Figures**

**Abstract**

Augmented reality (AR) software attempts to track real-world objects while creating the illusion that virtual objects exist in real space. To be convincing and relevant, AR software must be responsive—thus, efficient—and available ubiquitously wherever the tracked object is used. Difficulties arise because ubiquity demands a general, extensible model of the platform, while efficiency demands tailoring to a particular set of resources.

This thesis presents Illusion SDK: a general, extensible framework for AR. Illusion provides loosely coupled or decoupled abstractions of sensors, trackers, and compositors. Implementations are optimized for particular use cases. Illusion's architecture depends on only an event system and a 3D scene graph, so it is highly portable. Wrapping of third-party trackers is supported.

Illusion's current implementation targets Flash 11.4 and integrates with the Alternativa3D 8 graphics engine. To our knowledge, Illusion's support for wrapping third-party trackers is unique among toolkits targeting the GPU-accelerated Web. Illusion performs well on MacBook Pro 13" mid-2010, where an intensive camera application can exceed 45 FPS. Generally, Illusion should perform well on hardware that uses shared video memory. Optimizations are needed for hardware that uses dedicated video memory. These optimizations are problematic in Flash 11.4 but should not generally be problematic in ports to other platforms.

**List of Abbreviations Used**

| | |
|---|---|
| AGAL | Adobe Graphics Assembly Language |
| AIR | Adobe Integrated Runtime |
| API | application programming interface |
| AS3 | ActionScript 3.0 |
| AR | augmented reality |
| AV | augmented virtuality |
| AugCog | augmented cognition |
| CPU | central processing unit |
| CUDA | Compute Unified Device Architecture |
| CV | computer vision |
| DVI | Digital Visual Interface |
| FPS | frames per second |
| GPU | graphics processing unit |
| GUI | graphical user interface |
| HD | high-definition |
| HDMI | High-Definition Multimedia Interface |
| HMD | head-mounted display |
| HITLab | Human Interface and Technology Lab [at the University of Washington] |
| ICGV | Institute for Computer Graphics and Vision [at the Graz University of Technology] |
| I/O | input/output |
| LSO | local shared object |
| MR | mixed reality |
| NFT | natural feature tracking |
| PB3D | Pixel Bender 3D |

| | |
|---|---|
| QCAR | Qualcomm Augmented Reality |
| QR | Quick Response [Code] |
| RAM | random access memory |
| RFID | radio frequency identification |
| RGB | red, green, blue |
| RTMFP | Real-Time Media Flow Protocol |
| SDK | software development kit |
| SIFT | scale-invariant feature transform |
| SURF | speeded-up robust features |
| UbiComp | ubiquitous computing |
| UC | ubiquitous computing |
| UML | Unified Modeling Language |
| URL | uniform resource locator |
| USB | Universal Serial Bus |
| VRAM | video random access memory |
| VS | visual servoing |
| VVS | virtual visual servoing |

**Acknowledgements**

This thesis has been written at odd hours: late nights at home, lunch breaks at work, weekend visits to friends, and while lying in an emergency ward, where time stops. People around me have shown support and kindness no matter how distracted I have been in return.

My thesis supervisor, Dr. Alex Brodsky, has been a constant source of good advice on research methods, editing, software architecture, optimization, and getting things done. My directed studies supervisor, Dr. Derek Reilly, suggested many useful readings on AR. Derek and Dr. Stephen Brooks, as my thesis readers, have given thoughtful editing comments leading to this final version.

Bernhard Jung and Imagination Computer Services GmbH have provided free licenses and support for flare*tracker and flare*nft (tracking libraries that are wrapped by Illusion).

My managers and coworkers at the IWK Health Centre, Ad-Dispatch, and MindSea Development have encouraged my research and provided valuable training and experience. Nathan Kroll of Ad-Dispatch suggested my research goal: an efficient, ubiquitous AR Web framework.

Many people have helped me when my health has been bad. Among them are: the staff of the QEII Health Sciences Centre; the staff of the Mayo Clinic in Jacksonville; Dr. Jack Graham; Mirna and Jim; and Sue and Phil.

My friends in my role-playing group have given me time to forget reality and to be a minotaur, seraph, cowboy, smuggler, or spy.

My longtime friend Paul has been a model of dedication, patience, and hard work.

My friend Vika, a fellow thesis writer, has commiserated from the other side of the globe.

My cats—Plasma, Sanibel, Lambda, and Josephine—have created simplicity. Plasma and Sanibel kept vigil over the writing of this thesis every night for a year. For my benefit, Plasma developed a time management tool using her chew-toys. At the start of our worknight, she would bring me a fish patterned with purple flowers. At the end, she would bring me a cushion patterned with cow spots. She would stand over it and sing a dirge until I followed her to bed.

My parents, Jan and Bob, have given me a life and many new leases on it. Listening about thesis progress is only the smallest part of what they have done.

The memory of my brother Sam, and of the cats and people he loved, sustains me every day.

## Chapter 1: Introduction

**Mixed reality (MR)** is an emerging medium that links arbitrary physical objects to arbitrary software content in (nearly) real time. As a blanket term, MR includes both **augmented virtuality (AV)**, in which the user experience focuses on virtual environments, and **augmented reality (AR)**, in which the user experience focuses on real environments. Typically, in AR, the means of linking physical objects and software content is **computer vision (CV)**, such that the application receives event data when physical objects appear, move, or disappear from the perspective of the computer's video camera. Then, for example, relevant graphics can be dynamically positioned atop the live video feed (Figure 1).

Computer vision, more generally, is the capture and analysis of data about color, brightness, or line-of-sight distance (depth). For capture, CV uses light sensors, sometimes in combination with light emitters. For analysis, CV often relies on measuring local contrast and simulating various perspectives to gauge what is being viewed and how it is posed in space. If it runs continuously in real time, this process is known as **tracking** and the relevant software component is called a **tracker**. Outside controlled environments, robust CV tends to require running computationally expensive algorithms on big (video-quality) streams of sensor input (Comport, 2005).

**Ubiquity** is an essential concern in AR. Wherever relevant physical objects are found, the user may want to run the application using whatever computer platform is on hand. Kiosks, mobile devices, and the Web have the potential to make AR applications available to roving users. These deployment channels are crowded, so AR technology should have

low barriers to adoption for the sake of busy developers, promoters, and users. For example, entry barriers can be kept low by modular, extensible designs or by leveraging an existing software platform that is popular.

**Figure 1: An Example of AR**


A screenshot from "Apples and Goblets", our demo project. A live video background shows two sheets of paper. An identifiable image is printed in the center of each sheet. A tracker has found the images, and 3D models are being superimposed atop them. The apple and goblet models are courtesy of Teinye Horsfall at WireCASE Ltd (http://www.wirecase.com) and Sven Dännart at Medievalworlds (http://www.medievalworlds.com), respectively.

Another essential concern is **responsiveness**, which requires **efficiency**. Whenever presented with relevant physical changes, the

application should respond in (nearly) real time; otherwise, discrepancies between the real and virtual worlds become obvious and distracting. A typical AR application spends most of its time in image capture, image processing, and rendering. These tasks are slow when implemented naively but they can be accelerated by modern hardware if the software platform supports it. Mobile users tend to have limited hardware and Web users tend to have limited software platforms, so AR's efficiency requires special attention in these contexts.

These two concerns—ubiquity and efficiency—are somewhat at odds. A ubiquitous solution must achieve abstraction from the system features and resources that may be unavailable in relevant contexts; an efficient solution must work closely with the system to access appropriate resources. This thesis attempts to reconcile concerns of ubiquity and efficiency, especially in AR applications that target the Web.

The combination of ubiquity and efficiency is critical in an industry-grade AR engine—and finding or creating this combination is nontrivial. A good solution can potentially deliver this combination, provided that care is taken to use the strengths and avoid the weaknesses of a platform's idiosyncrasies.

As a means to this end, the thesis presents a high-level AR framework that is agnostic about its platform's I/O capabilities, yet is sufficiently modular and extensible to support optimized implementations for particular systems and use cases. Agnosticism about I/O makes the framework portable, in principle, to a wide variety of ubiquitous computers that may have unconventional interfaces. Also, it enables the framework to wrap any medium of AR: that is to say, any source of sensor data (visual or otherwise), any type of tracking, and any destination for

composited scenes. Sensors, trackers, and compositors are the framework's core abstractions. The potential for modular optimizations makes it feasible, in principle, to adapt to various underlying performance characteristics as the need arises, without needing to re-architect the application.

A few specific features of the framework are worth noting. Multiple trackers (potentially, wrappers for multiple third-party tracking libraries) may run at the same time, even if using the same source of sensor data. For efficiency, sensor data is shared by reference rather than by copy. A tracker may, in principle, distinguish between duplicates of real-world objects (though most existing third-party trackers do not do so). Trackers are agnostic about compositors and vice versa. Thus, an application can use the framework's tracking functionality with or without rendering a virtual scene atop a real scene. This decoupling makes the framework applicable to non-AR scenarios, such as video games that use tracking to control purely virtual scenes.

The next two chapters—"Background" and "Exploratory Work"—deal with the context and motivation of our work in AR. "Background" draws on published sources whereas "Exploratory Work" draws on the author's experience in the AR industry. One theme in these chapters is the fragmentation of AR solutions, leading (in industry) to redundant integration work that could be alleviated by a unifying framework. The remaining chapters—"Design and Contribution", "Evaluation", and "Discussion"—deal with the new framework, called Illusion SDK, which the author has developed as a potential basis for efficient, ubiquitous AR applications. "Design and Contribution" presents Illusion's architecture, along with diagrams and code samples, and compares this architecture to

existing alternatives. "Evaluation" quantifies the efficiency characteristics of two test applications built atop Illusion. "Discussion" reviews the levels of ubiquity and efficiency currently achievable with Illusion, and proposes future work on optimizations, extensions, and ports. Finally, appendices deal with Illusion's availability and non-AR functionality.

# Chapter 2: Background

This chapter gives an overview of the obstacles faced in this thesis project. Then, it proceeds with an explanation of the history, literature, and technology of several relevant domains: computer vision, augmented reality, ubiquity, efficiency, and Web platforms.

## 2.1: Problems in Web AR

A number of CV and AR libraries already target a Web platform such as Flash, Silverlight, Java, or (rarely) JavaScript. (See "2.3.3: Frameworks".) Problems with the status quo include:

- **Fragmentation:** The libraries have dissimilar programming interfaces for complementary functionality. (Some track barcodes, others track faces, and others track photos.) They lack high-level integration with game or graphics engines. Due to the amount of developer time required to study and integrate many different interfaces, industry may have difficulty adapting to new technology. (See "3.1.1: At Ad-Dispatch".)
- **Platforms in decline:** Many Web users do not have Silverlight or Java. The market penetration of these platforms is declining. (See "2.6.2: Adoption".)
- **Immature platforms:** Many Web users have browsers that lack support for the relevant features of JavaScript, such as WebRTC (for camera access) and WebGL (for GPU acceleration). (See "2.6.1: System Access".)
- **Inefficiencies:** Many libraries for Flash do not leverage recent platform optimizations such as GPU acceleration. They integrate

most easily with code that also does not leverage these optimizations. (See "3.2: Refinements".)

Better designs are feasible. Since 2003, mobile AR has benefitted from research toward an optimized, high-level toolkit supporting multiple tracking strategies and multiple operating systems. Recent iterations, Studierstube ES and Vuforia, use GPU acceleration and integrate with game engines. (See "2.3.3: Frameworks".)

We argue that there is an untapped opportunity to unify and optimize AR- and game-related functionality in one toolkit capable of targeting the Web. We use Flash 11 as our testbed. To avoid contributing to fragmentation, this new toolkit facilitates the wrapping of existing AR libraries, which may implement many different tracking algorithms. Besides making the underlying libraries easily swappable, Illusion allows their functionality to be used simultaneously in one application, through one interface. For example, two underlying tracking libraries could be used for different types of subjects in the environment or they could crosscheck each other's results for one type of subject.

Unifying relevant functionality is non-trivial because AR technology and applications are evolving rapidly. To be extensible and maintainable, the unified toolkit must provide abstractions that support current and foreseeable features and requirements.

Optimization is non-trivial because AR includes three expensive stages—image capture, image processing, and rendering—with competing data formats and resource requirements. This problem is exemplified in Flash 11, which has three dissimilar graphics pipelines. (See "2.6.4: Focus on Flash".) Here, a custom approach to compositing (mixing different pipelines' content) is needed to leverage a GPU-enabled pipeline and a

camera-enabled pipeline in a way that supports rendering 3D models atop a live video that trackers can read.

The combination of unification and optimization is non-trivial because trackers from various vendors should share resources such as the camera feed. To avoid duplicating resources, the wrappers around the trackers must be agnostic about the way sensor data (ex. image data) are obtained and managed.

### 2.2: Vision, Space, and Colorspace

"Did you see the stop sign, sir?" a police officer might ask a driver. The hypothetical stop sign is in plain view but the driver thinks he did not see it.

Seeing (or vision) implies more than just looking in the right direction and having one's retinas stimulated. Seeing may imply awareness, recognition, an appropriate reflex, or even an understanding of events. ("I saw him cheat at cards.") Like human vision, CV is a multilayered concept that pertains to sensors, intelligence, and everything in between.

At the sensor level, vision is about **light** striking **lenses** and **photoreceptors**. A lens is a curved, transparent surface that focuses (funnels) light toward a surface of a different size. This second surface—a retina, film, or digital sensor—is covered with photoreceptors. A photoreceptor has a chemical or electrical response to light: the more light striking the photoreceptor, the stronger the response. A system typically contains several types of photoreceptors, each with a different **spectral response**: a function that maps the light's wavelength (color) to the scale of the response. Photoreceptors, in general, may respond to wavelengths

that are invisible to humans (the infrared and ultraviolet spectra) (Baines & Bomback, 1967).

By the time a photoreceptor responds to light, this light has already journeyed through **space** and **colorspace**. It has bounced off surfaces (a motion in space) and, in doing so, has changed color and lost some intensity (a motion in colorspace). (More precisely, only certain colors of light, in certain amounts, have bounced off each surface in a given direction.) A particularly important waypoint in this journey is the last surface that the light bounces off before entering the lens. Typically, we perceive this surface as a thing we "see", though we may also perceive it as a thing we "see in" (ex. a mirror). As users of vision, we may care about this waypoint's position in space and about the way it transforms light in colorspace. Where is the surface and what color is it?

Given a set of densely positioned photoreceptors, with various known spectral responses, we should be able to estimate a surface's color by triangulating the photoreceptors' responses. We can imagine the various spectral responses as a "surround view", consisting of multiple, simultaneous vantage points in colorspace. However, in terms of regular space, we seldom have the luxury of a surround view: we have just one camera or two narrowly spaced eyes. As such, estimating a surface's spatial properties is a harder problem than estimating its colorspatial properties. Even harder is the more abstract problem of reconstructing a relationship among surfaces: in other words, recognizing a shape or object.

There are two major theories about the means by which human vision achieves object recognition without a surround view in space. One theory, pioneered by Marr and Nishihara (1978) and Biederman (1987), posits a "**structural-description**" approach. Supposedly, we memorize an

object's 3D structure and we mentally pose and draw such structures, as if they were models posing for an artist, until our mental drawing matches the actual projection in the eye. The other theory, pioneered by Poggio and Edelman (1990), posits an "**image-based**" approach. Supposedly, we memorize many 2D projections of an object from various vantage points and, like a witness looking through a set of mugshots, we search our memory for pictures that match the actual projection in the eye.

The two theories need not be mutually exclusive. Tarr and Bülthoff (1998), surveying previous experimental work, conclude that the actual approach depends on viewing conditions, the subject, the viewer's expertise, and the specificity of the recognition (ex. a man, a soldier, a lieutenant, Lieutenant Dan).

CV approaches to object recognition (and tracking) mirror the supposed approaches in human vision. The image-based approach uses a 2D image (or a 2D pattern that many images may match) as a ground truth for recognition. The structural-description approach uses a 3D model as a ground truth. Both approaches require transformations to be applied to the reference image/model and to the actual, captured image. These transformations make it possible to compare the arrangements of certain salient features (ex. vertices) in pseudo-3D space. Techniques that build on the two general approaches are described further in "2.3.2: Techniques".

## 2.3: Augmented Reality

### *2.3.1: Origins and Examples*

On January 24, 1990, Tom Caudell—a postdoctoral researcher at Boeing—proposed the idea of using head-mounted displays (HMDs) to project wiring schematics onto formboards (large electrical panels used in airplane manufacture). He and a colleague, David Mizell, expanded this proposed application into a domain that they called "see-through virtual reality" or, later, "augmented reality" (Caudel & Mizell, 1992; Mizell, 2001; Henn, 2010)—the superimposition of an interactive virtual space atop real space, in real time (Milgrim et al, 1994; Azuma, 1997). The codification of AR helped inspire a wide range of workplace applications, especially in fields with low automation but high costs of failure. Examples include: a synchronized digital/paper interface to facilitate communication in air traffic control (Mackay et al, 1998); a set of handheld tools to help a surgeon measure a 3D virtual model of the patient's liver (Reitinger et al 2005); a pharmaceutical pill recognition system (Hartl, 2010; Hartl et al, 2011); and a networked HMD allowing crime scene investigators to collaborate remotely with other experts (Poelman et al, 2012).

While they may have invented AR as a term, Caudell and Mizell acknowledge a long line of precedents. Mizell comments (in Henn, 2010):

> The technology is certainly older than the term. Ivan Sutherland's first head-mounted display, in 1968, was see-through and tracked. Military helicopter pilots used see-through, tracked, helmet-mounted gunsights in Vietnam. When Tom Caudell and I worked on the technology at Boeing in the early 1990's, Steve Feiner at Columbia University was working on very similar ideas. While

> Tom and I were prototyping the wire bundle assembly formboard application, Steve was demonstrating a system that could be used to guide a user through a maintenance procedure on a photocopier.

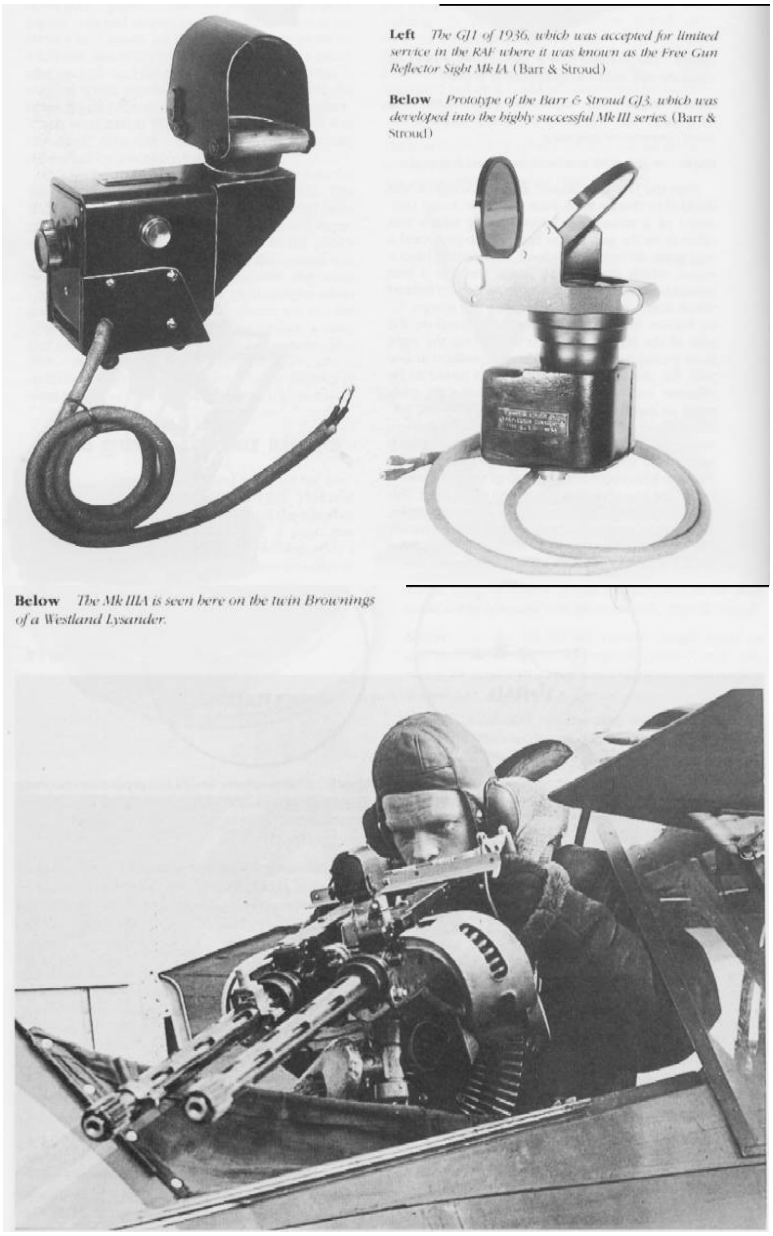
Mizell alludes to two prominent features of AR in his description of its lineage. First, AR involves tracking some identified target in the user's environment. Second, it provides the user with visual guide-marks that seem to exist in the same spatial context as the target. These guide-marks may assist the user in some task of hand-eye coordination involving the target.

The first feature—**automated identification and tracking**—is quite an old idea. An 1889 patent, pertaining to the problem of lost railcars, proposed an automated mechanism "to take the initials and numbers as the cars pass certain points…to form accurate information of the whereabouts of the cars" (in Collins, 2011). This patent later served as an inspiration to David Collins, who invented computerized **barcode tracking** and successfully applied it to railcars in 1961 (Collins, 2011) (Figure 2). Another well-established identification and tracking technology is the **radio frequency identification (RFID)** tag—a transponder that either emits or reflects a known radio frequency. RFID originates in 1940, when the Luftwaffe and then the Royal Air Force adopted it to distinguish friendly aircraft on radar (Dobkin & Wandinger, 2005).

**Figure 2: KarTrak Barcode**

KarTrak, the first barcode system, was deployed on railcars in 1961. Left: A KarTrak barcode, close-up. Right: A KarTrak barcode and two railcars. The barcode is the tall, dark rectangle near the photo's center.

**Figure 3: Reflex Gunsights**



**Left** *The GJ1 of 1936, which was accepted for limited service in the RAF where it was known as the Free Gun Reflector Sight Mk IA.* (Barr & Stroud)

**Below** *Prototype of the Barr & Stroud GJ3, which was developed into the highly successful Mk III series.* (Barr & Stroud)

**Below** *The Mk IIIA is seen here on the twin Brownings of a Westland Lysander.*

By the 1930s, reflex sights such as these Royal Air Force models were being mass-produced. Top: Prototypes, close-up. Bottom: A gunner uses his reflex sight with both eyes open. The reflex sight's design makes the illusionary targeting guide-marks appear at infinity. Thus, the illusion's alignment with the firing path is viewpoint-invariant and it is usable by either eye, or both at once, from any angle (Clarke, 1994).

The second feature—**illusionary guide-marks**—is likewise quite an old idea. Sir Howard Grubb, in 1900, patented a "Gun Sight for large and small Ordnance" that used a light source and a mirror to project illusionary guide-marks into the gunner's eye as he viewed his target through the scope. This optical invention, known as the **reflector sight** or **reflex sight**, was deployed in German fighter planes in 1918 (Clarke, 1994). By the 1940s, it was common for heavy weaponry of all kinds to include reflex sights (Figure 3), which were sometimes part of "electrically operated, computing" systems that used gyroscopes to predictively reposition some of the targeting guide-marks as the gunner attempted to track his moving target (United States Army Air Forces, c. 1944). These devices were known as **gyro sights**.

Clearly, AR's technological genesis owes much to aeronautics, which—along with other mechanized forms of transportation and warfare—has transformed mankind's thinking about space, tracking, and targeting. However, AR's cultural genesis goes beyond logistics and gunnery. Since antiquity, people have pondered how to create and place illusions—and how to animate normally inert objects—for the sake of art, pomp, and awe. ***Trompe-l'oeil*** **murals** are attributed to Greek painters of the fifth century BC (Pliny the Elder, c. 79 AD) and examples from the 70s AD are well preserved in the ruins of Pompeii. **Mechanical automata** are attested by authors as early as Pindar (c. 464 BC/1830, p. 40). He describes them as the pinnacle of human handiwork; as evidence of Athena's wisdom being manifest in mortal men:

> Meanwhile the maid with azure eye
> Her favor'd Rhodians deign'd to grace
> Above all else of mortal race,

With arts of manual industry.
Hence framed by the laborious hand,
The animated figures stand,
Adorning every public street,
And seem to breathe in stone, or move their marble feet.

Wisdom true glory can impart
Without the aid of magic art.

By the first century AD, Hellenic special effects technology—involving mirrors, magnets, mechanics, and pneumatics—was sufficiently advanced for Hero of Alexandria to devise interactive experiences, such as: programmable automata that were flexible enough to enact scenes from mythology; an automatic door triggered by lighting an altar fire; and dispensers that exchanged coins for holy water, or water for wine (Hero, c. 60 AD/1851). Although Hero wrote treatises on the construction of his devices, spectators would have seen only the miraculous facade.

Likewise in AR, technologically advanced interactions can be masked behind more traditional gestures and tokens. Posters and toy figurines are common targets for creative AR. They merge fantasy worlds into homes and public places much as *trompe-l'oeil* murals and automata did in antiquity (Figure Series 4). Storybooks are another common target for creative AR (Billinghurst et al, 2001). When used for entertainment purposes, the AR illusion may consist of elaborate 3D animations that are explorable from multiple perspectives and responsive to user input. A proposed social role of AR in the home is to encourage children to integrate multiple media into their imagination and to share this creative, storytelling experience with family and friends, in person (Hee, 2012). As such, AR entertainment is being contrasted (by its proponents) to purely

virtual entertainment that might not encourage as much interaction with one's surroundings.

Without relying as much on traditional artifacts, digital media giants are also embracing AR. The latest handheld game consoles, Nintendo 3DS and PlayStation Vita, come bundled with applications that combine tabletop gaming and video gaming. An arrangement of physical cards defines landmarks in the game world, while the device's camera and touchscreen provide the means of interaction. The player—or two players with separate devices—first build the game world by hand and then play in it via the device (Nintendo, "Nintendo 3DS - AR Cards at Nintendo"; Gutierrez, 2012). 19.2 million units of these devices (17.9 million of the 3DS and 2.3 million of the Vita) have been sold worldwide as of July 2012 (VGChartz, "Platform Totals"), so the bundled AR games are widely owned, whether or not they are widely played.

Anecdotally, AR has even inspired body modifications. One Nintendo 3DS player has had the physical part of an AR game tattooed onto his forearm (Shepherd, 2011). An earlier AR-enabled tattoo was demoed by ThinkAnApp of Buenos Aires, Argentina (Civantos, 2010). It is unclear whether ThinkAnApp had further plans or operations: its Twitter page contains just four posts, along with a link to a defunct domain that was once the company's website (ThinkAnApp, "thinkanapp (thinkanapp) on Twitter").

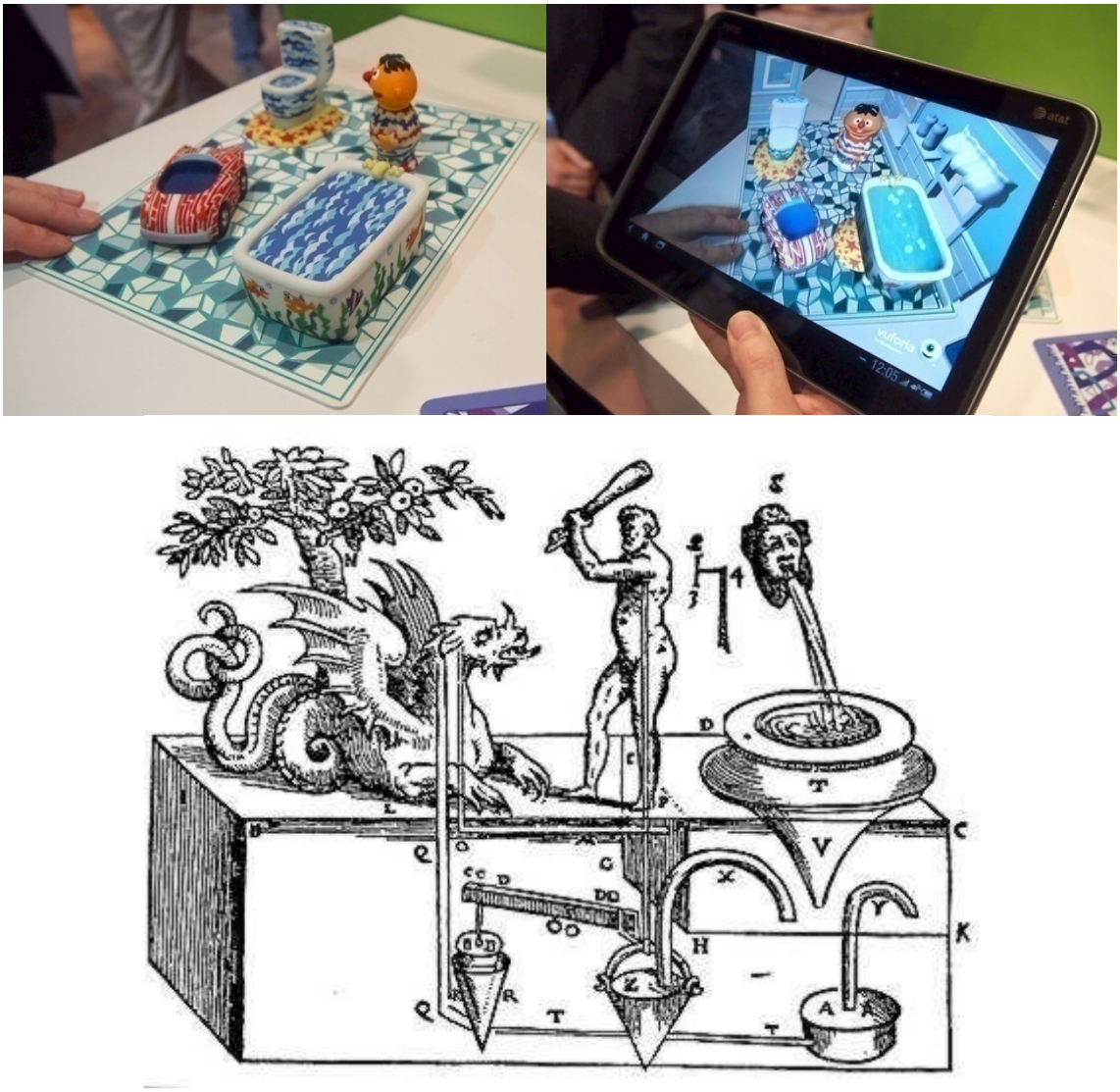
AR is an oddball mix of technologies and arts: the applications of it and its precursors run the gamut from deadly to constructive, and kitsch to sublime. It should prove popular.

## Figure Series 4: AR Compared to Ancient Art

### Figure 4A: AR Compared to *Trompe-l'oeil* Murals


Top: Marketing images from String Labs, an AR company. Viewed through the mobile device's camera, the posters turn into animated, interactive *trompe-l'oeil* murals. Bottom: A *trompe-l'oeil* mural painted c. 70 AD in Pompeii.

**Figure 4B: AR Compared to Automata**



Top: A prototype of an AR play-set from the Sesame Street franchise. Figurines and props can be positioned however the child chooses. Viewed through the mobile device's camera, they animate and interact in ways that depend on the physical setup (Hee, 2012). For example, Bert and Ernie might talk if they are close to each other. Bottom: "Hercules and the Dragon", an interpretation by Giovanni Battista Aleotti (1589) of an automaton designed by Hero of Alexandria. In this version, Hercules hits the dragon continually and the dragon spits water at him. In an alternative version (Hero 60 AD/1851), Hercules' attack is triggered by a pullstring when someone tries to pick an apple.

*2.3.2: Techniques*

A defining feature of AR is its use of **markers**—i.e. points of reference on physical objects. The format of these markers may be natural (ex. geography, anatomy), artistic (ex. logos, photos), or synthetic (ex. barcodes, electromagnetic tags). For ubiquity's sake, non-natural markers need to integrate well with existing production processes. For example, new images can be incorporated into printed products or packaging without requiring new steps in procurement or manufacturing. RFID tags can be inserted into layered products (ex. dresses, stuffed animals) but do require new steps in procurement and manufacturing. Our discussion focuses on image markers.

AR literature tends to apply the term **natural feature tracking (NFT)** to natural and artistic markers alike. Arguably, this conflation is appropriate: a live view of a face, a photo of a face, and an iconic smiley face might all qualify as "a face", for the purposes of a given AR application.

Given a predefined set of markers and a stream of input, the AR application must solve a classification problem: in each frame of input, which markers are represented? Moreover, what is the **pose** (position and orientation) of each represented marker? These questions, respectively, represent the problems of **recognition** and **tracking**.

For camera input, approaches to this classification problem rely on measures of local contrast. Our discussion focuses on local contrast in terms of color or brightness values, sampled in short, isolated timeframes. However, the same local contrast measures can also be considered in other terms (Figure Series 5), such as: depth (ex. sensed using an infrared camera and illuminator); **time-differenced** color, brightness, or depth (for

tracking a particular motion signature); or **time-averaged** color, brightness, or depth (for tracking a stationary marker in a motion-filled scene, ex. a briefcase abandoned in a subway station). An image marker in either the depth or time-differenced case would not look like a conventional photo. An image marker in the time-averaged case would look like a long-exposure photo.

Synthetic markers allow for very coarse contrast measures of local contrast because the relevant color palette (usually binary black-or-white) and edge patterns (usually right angles) are known *a priori*. Each frame of camera input can be analyzed by thresholding its colors, searching for the relevant edge patterns, comparing found edges to markers' edges, and (for any matches) iteratively refining an estimate of the transformation matrix (Wagner & Schmalstieg, 2007).

A common synthetic format is the so-called **square marker**, which is essentially a low-density 2D barcode: within a black-bordered square, each of a number of sub-squares is either black or white. Another common synthetic format is the **frame marker**, which is essentially a low-density 1D barcode bent in four places to form a square border around some arbitrary content. Also, a frame marker can be thought of as a square marker that omits all but the peripheral sub-squares. Square markers have appeared in AR literature since 1996 (Rekimoto, 1996) and are predated by numerous other concepts for automated barcode identification, going back to Collins' 1961 KarTrak system for railcars (Collins, 2011). However, most barcodes are not optimized for pose estimation tasks. For example, **quick response (QR) codes**, invented in 1994 for tracking automobiles during manufacture (Kan et al, 2011), are

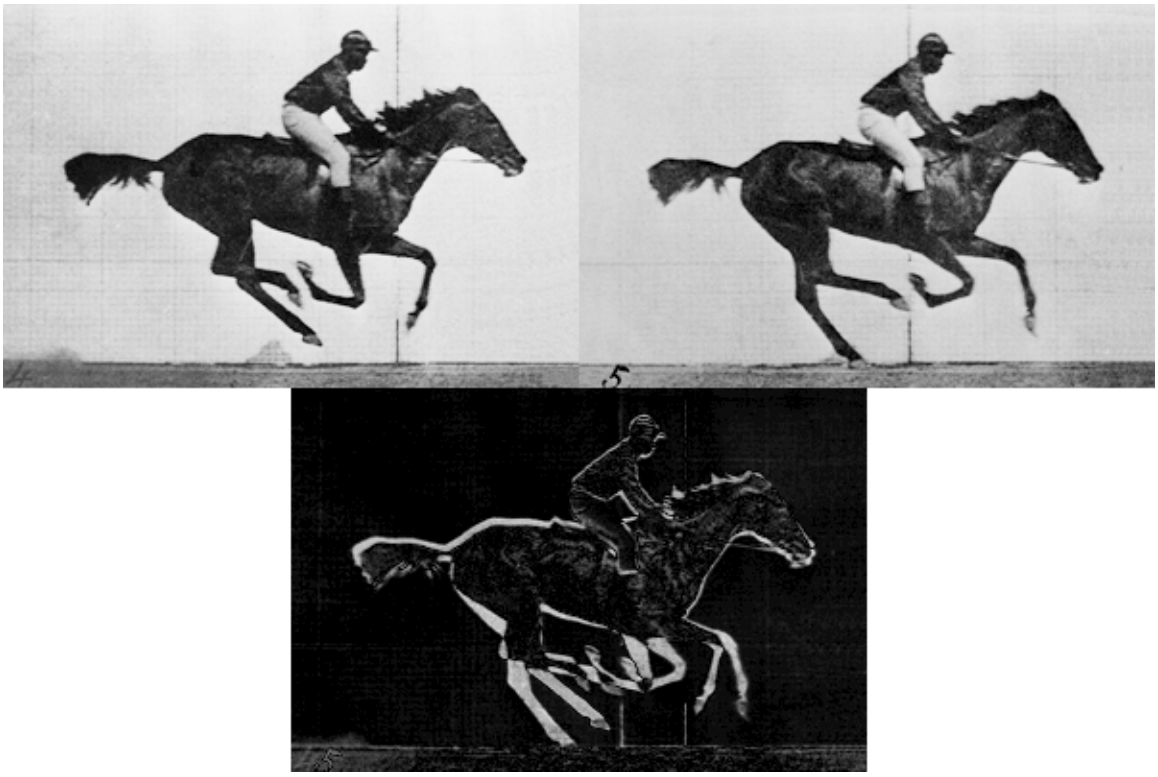
high-density 2D barcodes that are commonly used in identification tasks but not pose estimation tasks.

## Figure Series 5: Unconventional Image Types

### Figure 5A: Depth Image

"Self portrait with the Kinect. Robot in the back" (2011) by Martin Wojtczyk. This depth image is captured using the Microsoft Kinect camera and its official SDK (Wojtczyk, 2011). Bright areas may be interpreted as shallow (near) and dark areas may be interpreted as deep (far). The camera does not truly measure depth but rather infrared (IR) brightness. The source of IR light is an illuminator attached to the camera (akin to the flash on an ordinary camera). Note that areas shadowed from the IR illuminator, such as the wall behind the user, are falsely interpreted as being very deep. So are IR-absorbent materials, such as the frames of the user's glasses.

**Figure 5B: Time-differenced Image**



Top: Two consecutive frames from “The Horse in Motion” (1878) by Eadweard Muybridge. The galloping horse is captured in midair and then as its left hind leg lands. Bottom: The difference (later minus earlier) between the two frames. The bright areas may be interpreted as motion or an outline of motion.

**Figure 5C: Long-exposure Image**

"Office of Helmut Friedel" (1997) by Michael Wesley. This photo is an extremely long exposure, lasting one year (Kazmierczak, 2005). Subjects that move regularly, such as people, are not captured. Subjects that move occasionally, such as furniture, are captured as if semitransparent. Subjects that move rarely, or not at all, are captured as in normal photography. The same effects can be achieved by averaging video frames over time.

**Figure 6: Square Markers and Frame Markers**


Promotional images from the flare*tracker project. Top: Subtypes of square markers, including frame markers. Datamatrix markers are sufficiently dense to allow for URLs to be encoded. Bottom: Screenshots of the markers in use.

**Haar-like features** are more sophisticated local contrast measures, suitable for NFT. They were first proposed by Papageorgiou et al (1998) and then refined by Viola and Jones (2001). (Sometimes, Haar-like feature classification is called **Viola-Jones object detection**, after these authors.) Each Haar-like feature encodes differences in intensity among two or more adjacent image areas. (For example, an image area could be 4x4 pixels and its intensity could be the sum of those pixels' RGB components.) Each frame of live video can be subsampled and each

subsample's Haar-like features can be compared to each marker image's Haar-like features in order to evaluate similarity. At some similarity threshold, the live video subsample and the marker image are deemed to match. Scale (magnification) differences between the live image and marker image are handled by means of **feature cascades**, i.e. resampled versions of the features. If a marker is far away from the camera, its video image will match a low-resolution version of the feature set; if close, a high-resolution version. Feature cascades also provide an efficient means of screening out irrelevant image sections, using coarser comparisons first (Viola and Jones, 2001).

Haar-like feature classification, of the type described above, tends to be fast—capable of running at 15 FPS on Pentium III 700 MHz (Viola and Jones, 2001)—but it has shortcomings. It is insensitive to hue. It is sensitive to shadow edges and reflection edges. It is dependent on the camera's ability to expose the relevant contrasts (which are subtler than those of synthetic markers) under unpredictable lighting conditions. It is not necessarily robust to rotation and 3D transformations—though it can be if additional feature sets are generated (Lienhart & Maydt, 2002; Messom & Barczak, 2006).

An alternative to Haar-like feature classification is **scale-invariant feature transform (SIFT)**, first published by Lowe (1999). SIFT attempts to identify points of local contrast that change only minimally with respect to scale, rotation, illumination and 3D transformations. To identify such features, SIFT resamples the input images, whereas Haar-like feature classification resamples the features to potentially match input images. SIFT tends to emphasize geometric edges, whereas Haar-like

feature classification tends to emphasize an arrangement of darker and lighter regions, such as the eyes and mouth versus the rest of the face.

Another notable technique, **speeded-up robust features (SURF)**, combines SIFT with Haar-like feature classification. First proposed by Bay et al (2006/2008), SURF uses feature clusters, each consisting of one SIFT-like feature surrounded by multiple Haar-like features. SURF is demonstrated to be an advance in both performance and robustness, compared to its constituent techniques. Moreover, it handles 3D subjects relatively well by treating feature clusters as submarkers that may each be oriented differently (Bay et al 2006/2008).

The same means of generalizing 2D image tracking to 3D object tracking is found in earlier sources as well. Comport (2005) notes that object tracking problems in AR are essentially the same as those in robotics. A robot can estimate an object's pose by incrementally moving around the object in real space to find the perspective where the live 2D image of that object best matches a 2D reference image. This process is known as **visual servoing (VS)**. Comport describes an analogous process, for AR, as **virtual visual servoing (VVS)**. Unlike VS, VVS does not rely on camera movement. The change in perspective is virtualized by incrementally reposing a 3D reference model to find the 2D projection of the model that best matches the 2D live image. To simplify the comparison, Comport (like others) relies on edge tracking, such that the presence or absence of an expected feature can be confirmed by a "one dimensional search to the normal of a contour" (2005).

*2.3.3: Frameworks*

While the preceding works have advanced the algorithms that are relevant to AR, other works have contributed more to the body of available software components. Some of these components are suitable for integration into higher-level frameworks; others are themselves high-level frameworks to which we may compare Illusion. (See "4.3: Comparison to Other Designs".) For the moment, though, our focus is on understanding the range of tracking algorithms and AR application frameworks that are implemented on various platforms. The relative merits of some of certain major platforms are discussed later. (See "2.6: Web Platforms").

Starting in the mid-1990s, several frameworks have attempted to standardize and optimize AR interfaces across multiple platforms. Two cornerstones of this evolution have been Studierstube (an application framework) and ARToolKit (an image tracking component for square markers). Studierstube is developed at the Institute for Computer Graphics and Vision (ICGV) at the Graz University of Technology. ARToolKit is developed at the Human Interface and Technology Lab (HIT Lab) at the University of Washington and University of Canterbury (New Zealand), though major branches of it have merged into the work of the ICGV instead.

From 1995 to 2002, Studierstube focused on supporting collaborative AR work environments on heterogenous multicomputers (Schmalstieg et al, 2002). Using HMDs and numerous other I/O peripherals, these multiuser environments attempted to create new modes of office work by presenting physical tools and software tools in the same spatial context.

Similarly, ARToolKit originated from an experiment in AR video conferencing (Kato & Billinghurst, 1999).

These projects were put in another context in 2003, when elements of Studierstube and ARToolKit were ported for standalone use on the Pocket PC platform—the first mobile use of AR. The port's proof-of-concept application ran at just 5 FPS on iPAQ. It used CV and software rendering to draw lines around the edges of 2D barcodes (Wagner & Schmalstieg, 2003). A more performant and more useful application was reported in 2005, when the framework's authors were developing an AR guide to museum exhibits. Displaying animated 3D models atop binary-encoded markers, this prototype ran at 20 FPS on unspecified Pocket PC hardware (Schmalstieg & Wagner, 2005).

Despite these improvements, the performance of the original mobile port was deemed inadequate. A new port, Studierstube ES (Embedded Subset), commenced in 2006, with the goal of wedding mobile AR to an efficient, new, mobile game engine, including facilities for peer-to-peer cooperative play. A sample game, Cows vs. Aliens (2007), demonstrated the feasibility of running Studierstude ES's features on the Gizmondo mobile game console, which uses Windows CE (Mulloni, 2007). Subsequent work has also brought Studierstube ES to smartphones: Windows Phone and Android.

Besides these official ports of Studierstube, other libraries have adapted certain components of it and ARToolKit. NyARToolkit (Java, ActionScript, .NET, C++), FLARToolKit (ActionScript), SLARToolKit (C#), and JSARToolkit (JavaScript) are ports of ARToolKit (NyARToolkit, "Welcome to NyARToolkit.EN"; Heikkinen, "JSARToolkit"). JavaCV wraps ARToolKit plus several AR and CV

components from other parties (JavaCV, "JavaCV"). flare*tracker re-implements ARToolKit functionality in an original codebase by Imagination Computer Services GmbH, of Vienna, Austria (Imagination, "flare*tracker"). flare*nft (ActionScript) is Imagination's adaptation of Studierstube's NFT component (Imagination, "flare*nft"; Jung, 2011). FLARManager provides integration of FLARToolKit, flare*tracker, and flare*nft into the Papervision3D graphics engine (Socolofsky, "FLARManager: Augmented Reality in Flash"). Vuforia (formerly known as Qualcomm Augmented Reality or QCAR) is another adaptation of Studierstube's NFT component. Vuforia targets Android and iOS, with optional integration into the Unity game engine (Qualcomm, "Augmented Reality (Vuforia™)").

For licensing reasons, Studierstube is unlikely to spawn other ports in the near future. Equally, existing ports are unlikely to merge under an umbrella project. One of the developers of flare*nft comments, "The reason for flare* not being licensed to mobile devices [i.e. to mobile developers using Adobe AIR] is a contractual one. The tracker source that is also the basis of flare* has been sold to Qualcomm … (now available as QCAR SDK) and we are limited to licensing for PC platforms" (B. Jung, personal communication, September 14, 2011). Studierstube ES and other recent Studierstube developments (post-2008) are closed-source and not available for licensing (Studierstube, "Availability of Augmented Reality Software").

Independent of Studierstube, other proprietary solutions include D'Fusion SDK, from Total Immersion, of Paris, France; String SDK, from String Labs, of London, UK; IN2AR from Beyond Reality, of the

Netherlands; and Beyond Reality Face, from Tastenkunst, of Leipzig, Germany.

D'Fusion SDK, launched in 2004, implements various forms of NFT, including Comport's 3D object tracking (Comport, 2005). D'Fusion SDK is available in several versions, targeting desktops, mobiles (iOS, Android), the proprietary D'Fusion Web Player, or Flash. Some versions feature optional integration with visual editing suites: the company's own D'Fusion Studio; or Unity (Total Immersion, "Augmented Reality Software and Solutions by Total Immersion | Augmenting Your Reality"; Geffroy, 2012).

String SDK combines features of square markers and NFT. Each marker includes a freeform image but must be framed by a thick, black border on a white background. This hybrid approach seems to be original and unpublished. It might offer advantages in efficiency and robustness, compared to pure NFT. String SDK supports iOS only, with optional Unity integration (String Labs, "String™ Augmented Reality").

IN2AR (Beyond Reality, "IN2AR") offers NFT functionality, and Beyond Reality Face (Tastenkunst, "Beyond Reality Face") offers face tracking functionality. They are new libraries, having emerged during the writing of this thesis. They are written in ActionScript and include samples targeting the current version of Flash.

Many non-proprietary packages, too, are also relevant to AR. An influential, open-source library has been OpenCV (formerly, CVLib), launched by Intel in 2000. OpenCV initially focused on providing low-level optimizations to make CV functionality, including NFT, more feasible on single-core, consumer CPUs. When running hand-optimized MMX assembly, the library's alpha version achieved speedup ratios

ranging from 2.00 to 8.33, relative to its fallback of compiled C (Bradsky & Pisarevsky, 2000).

Like Studierstube, OpenCV has been ported to mobile and embedded platforms, and has suffered some performance setbacks in the process. Notably, these ports have tended to rely on OpenCV's single-threaded C fallbacks rather than contributing original optimizations for new architectures. No optimized port was completed until 2009, when the CVCell project ported OpenCV to the Cell Broadband Engine Architecture. Using up to six of the Cell's eight coprocessor cores, CVCell delivered mixed results—ranging from a 15.2 slowdown ratio to a 17.9 speedup ratio—in function-level benchmarks against OpenCV's optimized code for Intel Core 2 Duo E6850 3.00 GHz. However, an application-level benchmark (of NFT) favored CVCell, with a 1.37 speedup ratio, from 8.12 FPS to 11.2 FPS (Sugano & Miyamoto, 2010). Since 2010, OpenCV itself supports higher-order parallelism on NVIDIA GPUs via CUDA (OpenCV, "OpenCV Change Logs").

Besides being ported to other architectures, OpenCV has also been ported or wrapped for use with high-level languages, including C++, Ch, Python, and Java (OpenCV, "OpenCV Change Logs"; Yu et al, 2003; JavaCV, "JavaCV"). Many of these ports have been merged back into the main project. A small subset of OpenCV functionality, including facial tracking, has been ported to ActionScript as the Marilena project (Klingemann, 2009).

Tracking libraries sometimes build atop OpenCV. One example is an open-source library called ALVAR, which tracks square markers (VTT, "ALVAR Technical"). ALVAR is, in turn, wrapped by an open-source, C# game engine called Goblin XNA. Goblin XNA is, in principle,

designed to support other visual trackers as well, though it does not currently do so (Oda et al 2012).

Another camp of other open-source library development centers on the SURF algorithm specifically. The original implementation is patented but its source code is publicly released. Other implementations include OpenSURF (C++, C#) (Evans, 2009), Pan-o-Matic (C++), Parallel SURF (multithreaded C++), Speeded-Up SURF (CUDA) (Furgale et al, 2009), CUDA SURF, JavaSURF, ASSURF (ActionScript), and many more. Evaluations of various implementations have been undertaken by Gossow et al (2010) and by Abeles (2012). Notably, Pan-o-Matic has nearly identical performance to the original SURF, while Parallel SURF (adapted from Pan-o-Matic) offers large speedup ratios: 6.51 for 8 cores and 3.62 for 4 cores (Gossow et al 2010). To some extent, these SURF implementations are built with non-AR, non-real-time applications in mind. For example, Pan-o-Matic is purpose-built for panoramic photo stitching, i.e. merging multiple photos of adjacent, overlapping subjects into one, wider-format photo.

All of the preceding libraries and frameworks offer programming interfaces in general-purpose languages. However, some alternatives instead offer markup languages or visual programming tools, which both tend to treat applications as hierarchies of content. A notable example is the Argon AR browser, an iOS application that is the reference implementation for a proposed standard called KHARMA (Augmented Environments Laboratory, "KHARMA"). This standard includes a markup language, KARML, which allows 3D models and Web-like media to be anchored to geolocations or square markers.

Despite the range of relevant libraries and of library providers—academia, industry, the open-source community—most AR functionality is not yet widely known to application developers and consumers. Game interfaces based on square markers—relatively old technology—still have enough novelty value that they are heavily publicized in conjunction with the releases of new platforms such as the Nintendo 3DS and PlayStation Vita (Nintendo, "Nintendo 3DS - AR Cards at Nintendo"; Gutierrez, 2012). Thus, NFT remains one step ahead of the mainstream.

## 2.4: Ubiquity

AR, and the Web and mobile platforms it often targets, have evolved in tandem with the concept of **ubiquitous computing (UC, UbiComp, or ubiquity)**. Broadly, the literature on ubiquity predicts a massive proliferation of low-cost, networked, responsive computers that make use of sensor data. It also imputes certain expectations and behaviors to the people who will share an environment with these multitudinous computers. Certainly, AR is one computerized medium that may pervade an environment and change people's expectations and behaviors. Let us first survey the influences on UC and the different formulations that have emerged, and then consider UC's implications for an AR framework that targets the Web. Ultimately, we are interested in the role that a Web-based AR framework can play in furthering AR as a ubiquitous technology.

### *2.4.1: A Conflicted Concept*

On December 9, 1968, Douglas Engelbart demonstrated the tools that he believed people would use to achieve **augmented intelligence** (to

become "augmented intellectual worker[s]"), an aim he had characterized in terms of (Englebart, 1962):

> more-rapid comprehension, better comprehension, the possibility of gaining a useful degree of comprehension in a situation that previously was too complex, speedier solutions, better solutions, and the possibility of finding solutions to problems that before seemed insolvable. And by complex situations we include the professional problems of diplomats, executives, social scientists, life scientists, physical scientists, attorneys, designers—whether the problem situation exists for twenty minutes or twenty years.

Engelbart's demo introduced the public to the mouse, word processing, collaborative editing, hypertext, email, video conferencing, and many other future staples of **personal computing** and information technology (Engelbart, 1968). A subset of these tools would begin to reach users in 1973, when Xerox Palo Alto Research Center (PARC) developed the Alto, precursor to the Apple Macintosh.

If personal computing promised to augment our situational intelligence, then ubiquitous computing takes the promise one step further: to augment the situation (the reality) itself, with computers everywhere to quietly inform and serve us. Steve Jobs, in 1987, referred to the Apple II as "a ubiquitous computing resource that is powerful, reliable and flexible enough to be used everywhere on campus" (in Ronzani, 2007). A more nuanced meaning of the term "ubiquitous computing" was coined by Mark Weisner, at Xerox PARC, in 1988. He and his colleagues foresaw a world where networked computers would greatly outnumber human beings, and the role of single-user workstations would diminish in favor of a more diffuse, shared, peripheral, "invisible" (unobtrusive), and "calm" usership of embedded devices (Weiser & Brown, 1996).

Perhaps by the concept's nature, the predictions surrounding ubiquity are broad—lacking any monolithic prototype such as Englebart provided for the personal computer. Among their technological artifacts, the UC team at Xerox PARC made a handheld system called PARCTAB. It offered wireless networking, a stylus interface, and applications including a reverse pager (for mapping people's locations), a universal remote controller, and a weather forecast (Schilit et al, 1993). However, in reference to this invention, Weisner later writes, "[UC] is not the same thing as mobile computing, nor a superset nor a subset" (Weiser, March 17, 1996). Elsewhere, Weisner and John Seely Brown give the following forecast of UC's technological trajectory (1996):

> [UC's] cross-over point with personal computing will be around 2005-2020. The "UC" era will have lots of computers sharing each of us. Some of these computers will be the hundreds we may access in the course of a few minutes of Internet browsing. Others will be imbedded in walls, chairs, clothing, light switches, cars—in everything. … This will take place at a [*sic*] many scales, including the microscopic.
>
> … UC will see the creation of *thin server*s, costing only tens of dollars or less, that put a full Internet server into every household appliance and piece of office equipment. The next generation Internet protocol, IPv6, can address more than a thousand devices for every atom on the earth's surface. We will need them all.

Amid the prognostications, often the use cases are not pinned down. Why will we need to fill these thousands of IP addresses per atom? What data will be gathered and served by UC-enabled chairs and such?

Ubiquity's subsequent proponents have defined subdomains that yield more detailed prescriptions. One subdomain, **proactive computing** (also called **ambient intelligence** or **ubiquitous intelligence**), aims to fulfill the

criteria of invisibility and calmness by making computers measure, predict, and answer to human needs without any active human input (Tennenhouse, 2000). This concept has seen practical applications in continuing care and assisted living: for example, motion sensor data can be used to predict whether an individual is lost or incapacitated, in which case an automated call for help can be made (Consolvo et al, 2004). The same principle applies to security systems making automated calls. Certain other proactive computing proposals resemble the Jetson family's appliances: for example, the proactive refrigerator would be able to measure and classify the household's consumption, and place orders accordingly (Rogers, 2006).

Three main criticisms (Rogers, 2006) are levelled against proactive computing. First, as an AI problem, the inference of human needs and wants from sensor data is computationally difficult and perhaps ill-conceived: the needs and wants might change anytime for reasons that do not leave sensory artifacts. Second, the system's proactivity is invisible only to the person being monitored—and only until the help arrives. A nurse, policeman, deliveryman, or other intervening human is typically still assumed, such that the system is costlier and less private than implied. Third, the system does not encourage skill development (or skill maintenance) on the part of the person being served; this person may develop a dependency or a false sense of security.

Responding to these perceived dangers, another UC subdomain focuses on the concepts of "**proactive people**" and "**engaging user experiences**" (Rogers, 2006). According to this school, UC's best use cases would actually be in play, learning, scientific exploration, and persuasion—contexts that ostensibly demand an intense or excited

attitude instead of calm. Example applications include: educational robots for which students can write networked, sensory AI programs, with aesthetically pleasing outputs (Resnick et al, 2000); environmental monitoring systems consisting of distributed sensor nodes, which multiple users can deploy (Lane et al, 2010); and health promotion apps that monitor peer groups collectively, with the aim of generating positive peer pressure (Rogers, 2006). The latter proposal—to generate peer pressure—seems to risk compromising privacy and autonomous decision-making just as badly as proactive computing might do.

Another subdomain proposes that the level of calm or engagement should be **context-aware**, such that the computer automatically becomes invisible when it has no relevant information but highly visible when it does. Notification systems typify this approach (Rogers, 2006). An extreme example of context-aware computing is the US military's research into **augmented cognition (AugCog)**. This research consists of using real-time neuroimaging (EEG and fMRI scans) to predict which of the soldier's faculties are currently overstimulated, and to target new information at less stimulated faculties instead: for instance, by presenting text instead of graphics (Shachtman, 2007).

Still other subdomains or reformulations, such as **pervasive computing** and **mobile computing**, place less emphasis on invisibility; more on the technologies, protocols, and infrastructure that enable users to access information and electronic services "everywhere at anytime" (Hansmann et al, 2002). The term "pervasive computing" was popularized by representatives of Novell and then IBM in the 1990s (Ronzani, 2007). Originally, the term had much the same connotations as mobile computing. This conflation is clear in the book titled *Pervasive Computing:*

*The Mobile World*, by managers at IBM and Nokia (Hansmann et al, 2002).[1] Traditionally, in pervasive and mobile computing, the networks under discussion consist of conventional servers, personal computers, and personal phones, which do not seem to constitute shared or userless environments as envisioned by Weisner. Even in work on **mobile sensing**—exploring the use cases for continuously harvesting and broadcasting sensor data from communities of smartphone users—the role of the user is typically envisioned as active, hands-on, and computer-centric: for instance, as an annotator who types comments about the data for other users to read (Lane et al, 2010).

As this brief survey suggests, ubiquity and its successors are broad concepts, debated with somewhat ambiguous semantics, conflicting aims, and shifting technological focus. A study covering twenty years of newspaper articles (Ronzani, 2007) suggests that in popular media, the terms "ubiquitous computing", "pervasive computing", and "ambient intelligence" are used almost synonymously—and are used less and less since 2000, perhaps suggesting that the abstract and futuristic debates have been superseded by shared concerns about implementations today. Collectively, though, the literature on ubiquity and its kin still seems to offer pertinent advice for developing technology that is future-proof—technology that does not assume people's needs are bound to a desktop or a personal computer.

---

[1] More recently, the connotations of "pervasive computing" have grown to overlap with "ubiquitous computing" in general. The two fields' major conferences are merging as the *2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*.

*2.4.2: Relevance to an AR Framework*

As noted earlier, AR emphasizes interaction with markers—low-tech physical objects—that are identifiable and trackable by computers. Although AR researchers formerly required controlled environments with many processors and I/O devices handling few markers, the trend has been toward recognizing many markers with unspecialized computer systems that consumers can regularly access. (See "2.3.2: Techniques" and "2.3.3: Frameworks".) This trend seems to couple AR with ubiquity, though low-tech markers rather than computers are the proliferating element in this ubiquity.

Since our objective in this thesis is to develop an AR framework, not applications, we will avoid prescribing one or another set of user experience principles from the UC literature. The application developer is better positioned to make such choices, whereas the framework developer should provide underlying systems that let application developer to focus on these choices, without restrictions.

Despite their differences, most schools of ubiquity agree that their systems need to: be low-cost; leverage previous networking innovations and infrastructure; be available, responsive, and interconnected at an instant's notice; support collection, analysis, and sharing of sensor data; and not violate privacy expectations. Our framework for AR Web applications should be consistent with these goals.

**Cost** needs to be considered end-to-end, and in the context that people already have certain skills, software, and hardware: they need not invest from scratch. At least in the short term, the least costly solution—and also the least evangelical—is likely to be one that leverages commonplace

developer proficiencies and commonplace consumer platforms. Also, third-party dependencies must enter into the consideration of cost.

Here, it is worth noting that commonplace consumer platforms are entirely capable of controlling sensor networks, appliances, kiosks, and other hardware/interface configurations that do not match the conventional personal computing paradigm. Examples include: the Arduino family of electronic prototyping components, supporting wired or wireless control from Flash, Unity, iOS, Android, and other platforms (Arduino, "Interfacing with Other Software"); various smart TVs, supporting Flash (Magni, 2012); and the Samsung SUR40 spill-proof coffee table, running Windows 7 with APIs for collaborative touch input and real-time scanning (Samsung, "Samsung SUR40 for Microsoft® Surface®").

For **client-server networking**, high-level functionality is widely supported in Web platforms. For **peer-to-peer networking**, which is critical to the multiuser vision of ubiquity, high-level support is less widespread. Except in the situation where peers are discoverable on the local network, peer-to-peer networking tends to rely on remote servers, with proprietary protocols, to broker discovery of peers. The AR application framework should be compatible with some peer-to-peer networking service that is trusted: for example, one that is owned by the platform provider.

The concerns of **availability, responsiveness, and interconnectedness** are always fundamental to Web development and deployment. The proliferation of smartphones and Internet service over cellular networks has greatly extended the reach of Web applications. Unfortunately, at present, Web-based AR is not generally deployable in the mobile context

due to the lack of camera support in mobile Web browsers and plugins. For operating systems originating in the desktop world, the range of camera-enabled Web plugins has become quite broad. (See "2.6: Web Platforms".) Partly fulfilling the criteria for ubiquity, some of these plugins are in widespread use even on public computers, and are capable of fast load times and fast networking, such that they do not obtrude from the rest of the Web browsing experience.

For application developers who need to work with **sensor data**, the AR framework should generalize well in several respects. First, it should not make assumptions about what type of sensor data is being handled (ex. video, audio) or how the data are obtained (ex. camera, microphone, file). Second, it should not internalize the source of sensor data. Client code may need to read this source, preprocess it (affecting input to the tracking algorithm), or postprocess it (ex. affecting output to the video renderer). Third, it should expose an interface (which is needed internally anyway) for sharing sensor data among multiple subscribers, which are notified as data become available.

**Privacy** is problematic in Web-based AR. The user's picture, possession of an AR marker, and standard Web client data provide several means of tracing an identity. Either the platform or the application framework should insist on informed consent if there is the possibility that the AR application will transfer data to the server or third parties. Also, the platform should provide reasonable security against data theft on the client side and in network transit, since the application may be running on public computers or public networks where fellow users cannot trust each other.

We have described some ordinary (even pedestrian) computing problems that apply to ubiquitous AR. Indeed, of all the heralded aspects of ubiquity, the ones emphasized here are perhaps the least revolutionary—but they matter with respect to reaching (and not disappointing) the broadest audience, in diverse contexts, today. A good lesson in features' relative importance can be drawn from home appliances (which seem to loom large in thought on ubiquity). The dishwasher, refrigerator, washing machine, and dryer are subliminal parts of daily life not because they are invisible and silent (indeed, they are big and loud), nor because recent prototypes or luxury versions have AI and networking add-ons, but rather because the convenience they provide is affordable and reliable. At an exhibition in 1959, Nikita Khrushchev boasted of the Soviet Union's technological superiority over the United States; Richard Nixon regrouped by showing off the kitchen appliances that were affordable to the single-income family of the American worker (Safire, 2009).

## 2.5: Efficiency

### *2.5.1: Factors*

Efficiency is the capability to do much using little. "Much of what?" and "Little of what?" are questions that depend on the application's deliverables and the system's resources, respectively. Alternatively, all renewable resources can be abstracted as time. Efficiency in terms of deliverables per time is also called throughput. A related concept is latency: the amount of time between an event's occurrence and the completion of its handling.

Typically, an AR application must deliver real-time video and graphics. This deliverable can be measured in frames, i.e. redraw events that are handled by the software. Throughput may be expressed in frames per second (FPS). (Throughput, in this case, is also called frame rate). Latency may be expressed in milliseconds (ms) or another unit of time. Alternatively, latency may be expressed in number of frames, as frames are convertible to time when the frame rate is known. When latency is one frame or greater, the application is said to suffer from "frame lag".

High throughput yields the impression of smooth motion. 60 FPS is ideal for most computer screens. However, it is not imperative for AR to run at 60 FPS. For comparison, consider the frame rates used in cinematic productions that convincingly blend real and virtual footage. Most feature films are shot at 24 FPS; Peter Jackson's *The Hobbit* is shot at 48 FPS; and some Disneyland rides are shot at 60 FPS (Jackson, 2011).

With live video, low latency yields the natural impression of seeing in the present time. High latency yields the uncanny impression of seeing into the near past. Consider that an eyeblink lasts 100 to 400 ms (Schiffman, 2001). A latency of similar length can cause the viewer to see his own eyelids close fully and reopen—an uncanny effect indeed.

With tracking, low latency helps the user correlate his (or the tracked object's) motions to immediate results, such that he sees which motions are trackable and can quickly adapt to the tracker's strengths. Thus, the application seems to be responsive to the user and vice versa. High latency creates the frustrating impression that the tracker is unreliable, as the user tends not to see its successes until he has already moved again.

Camera-based interfaces in console games have received criticism for high latency. On Xbox 360, typical latency is about 150 ms to 200 ms in games that use gesture recognition with 640x480 video at 30 FPS (Leadbetter, 2010). For comparison, in Xbox 360 games that use gamepad input alone, typical latency is about 4 frames, or 67 ms for 60 FPS (Leadbetter, 2009).

Video transfer contributes significantly to the time cost of AR. A webcam, whether internal or external, typically sends its data via Universal Serial Bus (USB). Alternatively, it might use another peripheral bus such as FireWire, Ethernet, or Thunderbolt. Multiple peripherals (ex. plugged into different ports) may compete for bandwidth on the same bus. When competition for bandwidth is low, USB 2.0 can support 640x480 uncompressed video at 60 FPS and USB 3.0 can support 1080p (1920x1080) uncompressed video at 60 FPS (Table Series 1). The transfer time (the contribution to latency) would be about 15 ms and 10 ms in these respective cases.

## Table Series 1: Data Rates, FPS, and Transfer Time

**Table 1A: Data Rates of Uncompressed Video**

| Resolution | Data rate (MB/s) @ 30 FPS, RGB565 | Data rate (MB/s) @ 30 FPS, RGB888 | Data rate (MB/s) @ 60 FPS, RGB565 | Data rate (MB/s) @ 60 FPS, RGB888 |
|---|---|---|---|---|
| 320 x 240 | 4.4 | 6.6 | 8.8 | 13.2 |
| 640 x 480 | 17.6 | 26.4 | 35.2 | 52.7 |
| 1280 x 720 | 52.7 | 79.1 | 105.5 | 158.2 |
| 1920 x 1080 | 118.7 | 178.0 | 237.3 | 356.0 |

**Notes:** RGB565 is 2 bytes per pixel. RGB888 is 3 bytes per pixel.

**Table 1B: FPS and Latency of Uncompressed Video on Peripheral Buses**

| Type | Year | Max data rate (MB/s) | Max FPS @ 640x480, RGB888 | Transfer time (ms) @ 640x480, RGB888 | Max FPS @ 1920x1080, RGB888 | Transfer time (ms) @ 1920x1080, RGB888 |
|---|---|---|---|---|---|---|
| USB 1.1 | 1996 | 1.5 | 1.7 | 585.9 | 0.3 | 3955.1 |
| USB 2.0 | 2000 | 60.0 | 68.2 | 14.6 | 10.1 | 98.9 |
| FireWire 800 | 2002 | 98.3 | 111.8 | 8.9 | 16.6 | 60.3 |
| Gigabit Ethernet | 1999 | 125.0 | 142.2 | 7.0 | 21.1 | 47.5 |
| USB 3.0 | 2010 | 625.0 | 711.1 | 1.4 | 105.3 | 9.5 |
| Thun-derbolt | 2011 | 2,500.0 | 2,844.4 | 0.3 | 421.4 | 2.4 |
| 100 Gigabit Ethernet | 2008 | 12,500.0 | 14,222.2 | 0.1 | 2107.0 | 0.5 |

**Note:** Transfer time is per frame; thus, it is the minimum contribution to latency.

Note that uncompressed video is the norm for webcams. However, high-end models may feature H.264 encoding, which can yield excellent image quality at compression ratios of 50:1 (Kane Computing, "Compression Ratio Rules of Thumb"). For AR purposes, compression may be counterproductive: the tracker probably cannot read compressed video without it being decompressed again.

Depending on the camera, the programmer may be able to configure the captured resolution and maximum frame rate. However, little else can be done in software to control the costs of capturing video frames and sending them via the peripheral bus. Optimization efforts must focus on subsequent transfers and transformations instead. A typical application loop might interleave manipulations to camera-derived data and purely virtual data, as follows:

1. Capture the video frame, as previously discussed. Depending on the system and the use of its graphics libraries, the captured video frame may be stored in main memory, GPU memory, or both.
2. Analyze the video frame (in comparison to reference geometry and textures) to obtain tracking results. This step is processor-intensive because it involves many transformations of many vertices and pixels.
3. Update the 3D (virtual) scene based on the tracking results. Typically, few 3D transformations are directly dictated by the tracking results and thus this step is inexpensive.
4. Update the 3D scene again according to its own dynamics (ex. AI, physics, kinematics). This step is processor-intensive if there are many interacting entities.

5. Render the video frame. This step is bus-intensive if the approaches to video capture and rendering dictate that the frame must now be moved between different memory regions (i.e. main memory v. GPU memory). It is not processor-intensive unless complex filters are applied to the video (ex. to make it look more similar to the purely virtual content).
6. Render the 3D scene. This step is processor-intensive because it involves many transformations of many vertices and pixels. It is not bus-intensive as long as resources (ex. geometry, textures) are reused between frames.
7. Blend the 3D rendering and video rendering—if the video rendering was not already the 3D rendering's background during (6). This step is bus-intensive if the two renderings are in different memory regions (i.e. main memory v. graphics memory). It may be somewhat processor-intensive if it involves blending many pixels with transparency.
8. Display the rendered, composite scene. Like video capture, this step is bus-intensive and not programmatically controllable. Depending on the user's monitor interface (ex. DVI, HDMI, DisplayPort) and resolution settings, transfer times of about 5 ms to 10 ms can be expected. At non-native resolutions, LCD monitors take additional time to interpolate between input pixels and output dots. Combined transfer and interpolation time can be as much as 50 ms for some setups (Leadbetter, 2009).

To summarize, we find high, programmatically controllable costs in (2), (4), (6), and perhaps either (5) or (7). Our focus in this thesis is on steps (2) to (3), which concern tracking, and (5) to (7), which concern

rendering. Step (4), which concerns dynamics, is not addressed further, as the topic of optimizing dynamics is broad, somewhat application-specific, and not necessarily specific to AR.

As an alternative to interleaved, sequential execution, the processing of camera-derived data and purely virtual data can run in two parallel streams, potentially leveraging multiple processors more efficiently. These streams must synchronize at (3) and (7). However, it is not strictly necessary that synchronizations occur on every iteration. One stream's latency and frame rate can be compromised to improve the other's. For example, video capture and tracking could run at 30 FPS while dynamics and rendering ran at 60 FPS (or vice versa).

Given the wide variety of contributing factors, efficiency is difficult to predict from *a priori* knowledge alone. That is to say, in high-level programming, precise efficiency characteristics are not obvious from an inspection of source code and hardware specifications. Rather, efficiency must be measured at runtime, preferably in an itemized, modular fashion such that shortcomings can be traced to one factor or another and remedied in future revisions. We now turn our attention to these practical issues of measurement.

### *2.5.2: Measurement Techniques*

The developer should test software's efficiency on systems that are deemed typical of the target audience. Let us assume that the target audience for Web AR applications is similar to the target audience for Web games. According to one platform's recent user survey (Unity, "Web Player Hardware Statistics - 2012 Q2"), the typical Web gamer's system runs Windows 7 (51.0%) and has an Intel Core 2 CPU (33.3%), 2 GB

RAM (33.5%), Intel GMA 950 GPU (14.4%), 64 MB VRAM (32.2%), and 1366x768 desktop resolution (23.0%). At this desktop resolution, it is unlikely that a 1280x720 video would be visible all at once in a windowed Web application. Thus, 640x480 is more plausible as a "typical" video resolution that an application developer would support for Web AR.

After choosing a testbed, the developer should itemize costs. To measure the time cost of a block of code, the developer can programmatically start a timer at the block's beginning and stop the timer at the block's end. We may say that the block's time use is a **transparent cost**, since it is so readily measured. However, programmatic timers cannot readily capture the full time cost of a process that includes input/output (I/O) events. For such a process, some costs are hidden in software and hardware that are beyond the programmer's sandbox; some are hidden in the non-computing world, where the user moves and sees. We may say that time use in these contexts is an **opaque cost**.

An I/O process's total latency or total time cost (transparent plus opaque) can be estimated from a video recording of the user and I/O devices. A certain frame may be deemed to show an input event and another frame the corresponding output event. The interval between these two frames is the estimated latency. Such an approach has become popular in the gaming press and game development industry (West, 2008; Leadbetter, 2009). The estimate's precision is limited by the recording's frame rate. Its accuracy is limited by the difficulty of discretizing an input event: it will take several frames for a user to execute a button-press, let alone a gesture with an AR marker.

If we are willing to artificially omit some opaque costs, we may discretize input events more precisely and more accurately by means of

proxy. An **input proxy** is not regular input; rather, it is a stored or programmatically generated data stream that is treated as if it were regular input. For example, a video file or a live rendering could be a proxy for live camera input. The proxy's event timings are knowable because its data can be recorded in laboratory conditions or programmed deterministically.

For AR, an input proxy is especially useful in measuring latency between capturing a physical marker's appearance, motion, or disappearance and making the corresponding update to the virtual marker. These latencies affect the perceived responsiveness of the AR interface, in a way that may be somewhat independent of frame rate. Without use of an input proxy, another way to estimate these latencies would be to include (as a baseline) a simple tracker that is highly tailored to the marker and viewing conditions. For example, the coordinates of a yellow tennis ball in a blue scene should be determined quickly by a tracker that seeks a yellow region and considers its radius.

For applications that are bottlenecked by input, an estimate of opaque costs can be obtained by measuring frame-to-frame time in a minimal application that does little but gather input. For AR, this baseline application could be a live camera feed, a readout of timings, and nothing else.

## 2.6: Web Platforms

### *2.6.1: System Access*

Traditionally, client-side Web platforms have placed major restrictions on system access. Some of these restrictions have made CV and AR either infeasible or inefficient.

**Table 2: Browser Support for WebGL**

| Browser | Supports WebGL? |
|---|---|
| Internet Explorer 6+ | Only via third-party add-on: Chrome Frame or IEWebGL |
| Firefox 4.0+ | Yes |
| Google Chrome 9+ | Yes—but with significant platform incompatibilities, including: Windows XP; ATI on Linux |
| Safari 5.1+ | Yes—but disabled by default |
| Opera 12+ | Yes |
| Mobile Safari | No |
| Google Android | Depends on vendor's implementation |
| Internet Explorer Mobile | No |

**Sources:** Google, "Google Chrome Frame"; Google, "WebGL and 3D graphics"; IEWebGL, "IEWebGL - WebGL for Internet Explorer"; Opera, "An introduction to WebGL"

JavaScript, as implemented in Web browsers, does not provide access to multiple CPU cores. Camera access is just starting to gain support in

JavaScript: a draft specification called WebRTC is implemented in Firefox 17 (a development version), Chrome 21, and Opera 12 (Bidelman, 2012). The GPU may or may not be accessible from JavaScript via a standard called WebGL. WebGL's availability depends on the user's platform, browser, add-ons, and settings (Table 2).

Compared to JavaScript, plugin-based Web platforms are generally less restrictive, at least when comparing recent stable versions (Table 3). Most of these platforms can be characterized as either general-purpose application runtimes or game engines. General-purpose application runtimes tend to give client code full access to the system's multiprocessing capabilities. Game engines, on the other hand, tend to multiprocess certain functionality internally (ex. rendering, physics), while prescribing single-threaded client code. The most popular plugin platform, Flash, has mixed characteristics. It has traditionally prescribed single-threaded ActionScript code (with the possibility of hardware-accelerated rendering behind the scenes), yet in recent years it has added several programmable forms of multiprocessing: Pixel Bender shaders; AGAL shaders; and, most recently, ActionScript Workers. (See "2.6.3: Focus on Flash".) Broadly, Flash can be characterized as a general-purpose application runtime with roots as a scriptable media player.

As an alternative to building an application atop an existing plugin platform, developers can roll their own plugin application or platform using ActiveX for Internet Explorer (Microsoft, "ActiveX Controls"), NPAPI for other browsers (Mozilla, "Gecko Plugin API Reference"), or higher-level wrappers that may bridge the two (Mozilla, "External resources for plugin creation"). Broadly, what can be achieved in writing a desktop application or runtime, can be achieved in writing a plugin.

However, the challenges of creating an original plugin platform—and developer and user communities around it—are beyond the scope of this thesis.

**Table 3: Plugin-based Web Platforms that Support Camera Access and Multiprocessing**

| Plugin | Intent | Systems with camera access | Multiprocessing capabilities |
|---|---|---|---|
| Flash 10.0+ | General-purpose | Windows, Mac, Linux | Accelerated 2D rendering. General-purpose multiprocessing is feasible using Pixel Bender shaders but not ActionScript alone. |
| Flash 10.2+ | General-purpose | Windows, Mac, Linux | Accelerated video decoding. |
| Flash 11.0+ | General-purpose | Windows, Mac, Linux | (Not applicable to Linux.) Accelerated 3D rendering. General-purpose multiprocessing is feasible using AGAL or Pixel Bender 3D shaders but not ActionScript alone. |
| Flash 11.4+ | General-purpose | Windows, Mac | General-purpose multiprocessing is feasible using ActionScript Workers. |

continued on next page

continued from previous page

| Plugin | Intent | Systems with camera access | Multiprocessing capabilities |
|---|---|---|---|
| Java | General-purpose | Windows, Mac, Linux | Unrestricted. |
| Silverlight 4.0+ | General-purpose | Windows, Mac | Unrestricted. |
| Moonlight 4.0+ (in preview release) | General-purpose | Linux | Unrestricted. |
| Shockwave 11.5.8+ | Game engine | Windows, Mac | Superset of Flash 10.0 capabilities, as Shockwave can embed Flash 10.0 bytecode. Also, has its own route for accelerated rendering, physics, etc. |
| Unity Web Player 3.5+ | Game engine | Windows, Mac | Unrestricted but, generally, the Unity API is not thread-safe. Accelerated rendering, physics, etc. |
| ShiVa3D 1.9+ | Game engine | Windows, Mac, Linux | Via C++ plugins (1.9+), unrestricted. Via Lua scripts, general-purpose multiprocessing is infeasible. Accelerated rendering, physics, etc. |

continued on next page

continued from previous page

| Plugin | Intent | Systems with camera access | Multiprocessing capabilities |
|---|---|---|---|
| Panda3D | Game engine | Windows, Mac, Linux | Unrestricted. Includes multiprocessing extensions to Python. Accelerated rendering, physics, etc. Client code, culling, drawing optionally run parallel to each other. |
| D'Fusion | AR engine | Windows, Mac | Accelerated rendering. Maybe accelerated pixel buffer manipulations. General-purpose multiprocessing is infeasible via Lua scripts; no other kind of client code allowed. |

**Sources:** Adobe, "ActionScript Technology Center"; Adobe, "Director 11.5 Help"; Adobe, "Flash Player Release Notes"; Adobe, "How Stage3D Works"; Adobe, "Pixel Bender Technology Center"; Adobe, "What is AGAL"; Total Immersion, "Augmented Reality Software and Solutions by Total Immersion | Augmenting Your Reality"; Rose, 2011; Microsoft, "Silverlight"; Novell, "Moonlight"; Oracle, "Lesson: Java Applets"; Panda3D, "Documentation"; ShiVa3D, "Documentation"; ShiVa3D, "Documentation"; Unity3D, "Documentation".

*2.6.2: Adoption*

General-purpose plugin platforms, including their recent versions, are quite widely adopted, though some are in decline (Table 4). Special-purpose plugin platforms, such as game engines, do not exhibit the same potential in terms of market penetration. Within the latter category, Shockwave (Table 4) remains far more widely adopted than the alternatives. For comparison, consider that Unity Web Player had 113.3 million cumulative installations between its 2006 initial release and May 2012 (Unity, "Fast Facts"). Of this number, about 6 million or 5.3% would have been downloads by developers (Unity, 2012, April 9), suggesting that on average each developer has converted only 19 users to the Unity Web Player. For other vendors' special-purpose plugins, data are completely unavailable—in which case, the market penetration is presumably negligible.

Some special-purpose runtimes have been ported to run atop Flash, rather than requiring a dedicated Web plugin. Flash deployment targets are now offered by Unity and by Unreal Engine (a game engine geared toward large studios).

A plugin's market penetration is relevant to the initial user experience of any application targeting that plugin. A user who does not already have the plugin might not realize it is required, might not wish to install it, or might fail to install and run it despite trying. Even on successfully installing the plugin, the user might feel inconvenienced and be negatively predisposed toward the app. For these reasons, the installation experience becomes an increasingly crucial point of comparison where market penetration is lower. Factors affecting the installation experience may include: the size and hosting of the plugin download; system requirements;

the inclusion of any third-party software that might be perceived as adware; automated update checking; and the need for any restart/refresh steps that may cause the user to lose the webpage (Helgason, 2008).

**Table 4: Market Penetration of Selected Plugins**

| Plugin | Market penetration, October 2011 | Market penetration, April 2012 |
|---|---|---|
| Flash 11.0+ | 23.36% | 68.86% |
| Flash 10.0+ | 93.89% | 94.39% |
| Java, any version | 76.57% | 68.70% |
| Silverlight 4.0+ | 63.33% | 59.16% |
| Shockwave 11.0+ | 26.54% | 26.42% |

**Source:** StatOwl.com, "Statistical analysis and market research of Internet usage trends".

High market penetration does not necessarily imply that users are proactive in installing the plugin and seeking content that uses it. The plugin might come preinstalled on the user's system or it might be suggested to the user by websites that require it.

Another factor in initial user experience is loading time, which depends partly on the plugin (and its standard libraries) and partly on the application. Anecdotally, Flash and Silverlight offer much faster loading times than Java and Shockwave do. However, no benchmarks of loading times seem to be available.

*2.6.3: Performance*

For performance benchmarks more generally, Ernst (2011) is an excellent source, offering tests of JavaScript, Flash, Java, Silverlight, and their standard libraries. However, even his work gives short shrift to the multiprocessing capabilities of certain platforms—suggesting the extent to which these capabilities remain immature or just underexploited. For example, Flash's and JavaScript's 3D acceleration capabilities receive mention but not testing, due to their unstable implementations as of 2010-2011. Meanwhile, Flash's shader-based multiprocessing capabilities, despite being more mature, receive no mention at all.

Subject to such limitations, Ernst finds that there is no clear, cross-category winner of his benchmarks. Java and Silverlight tend to lead in numerical (number-crunching) benchmarks, while Flash holds the advantage in 2D graphics and string manipulations, for example. Moreover, Ernst finds that some performance advantage may be gainable by mixing platforms in one application. He advocates the use of Flash/JavaScript intercommunication, with JavaScript hopefully compensating for weak numerical performance in single-threaded Flash client code. Admittedly, this approach is fragile: depending on the browser's JavaScript implementation and security settings, the intercommunication may adversely affect performance or fail (Ernst 2011).

*2.6.4: Focus on Flash*

Developments in the last three versions of the Flash platform are especially important to our research. Existing AR and 3D rendering libraries have emerged at various stages in the platform's evolution, so

multiple Flash versions are relevant to an understanding of the libraries' design. Where not otherwise noted, our remarks on Flash are also applicable to the corresponding versions of **Adobe Integrated Runtime (AIR)**, the non-Web platform that can typically be targeted from the same codebase as Flash.

**Table 5: Timeline of Recent Flash and AIR Versions**

| Date | Flash version | AIR version | Changes in platform support |
|---|---|---|---|
| 2012, August | 11.4 | 3.4 | |
| 2012, May | 11.3 | 3.3 | Flash drops Linux |
| 2012, March | 11.2 | 3.2 | Flash drops Android |
| 2012 | | | LG Smart TVs released with support for AIR 3.0 |
| 2011, November | 11.1 | 3.1 | |
| 2011, October | 11.0 | 3.0 | |
| 2011, June | | 2.7 | AIR drops Linux |
| 2011, May | 10.3 | | |
| 2011, April | | | BlackBerry Tablet OS released with support for Flash and AIR |

continued on next page

| continued from previous page | | | |
|---|---|---|---|
| **Date** | **Flash version** | **AIR version** | **Changes in platform support** |
| 2011, March | | | AIR adds iOS |
| 2011, February | 10.2 | 2.6 | |
| 2011, January | | | Flash adds Android |
| 2011 | | | Samsung Smart TVs released with support for AIR 2.5 |
| 2010, October | | 2.5 | AIR adds Android |
| 2010, June | 10.1 | | |
| 2009, November | | 2.0 | |
| 2008, November | | 1.5 | |
| 2008, October | 10.0 | | |
| 2008, June | | 1.1 | |
| 2008, February | | 1.0 | |
| 2007, December | 9 Update 3 | 1.0 Public Beta 3 | (Windows, Mac, and Linux already supported) |

**Sources:** Adobe, "Flash Player Release Notes"; Adobe, "Adobe AIR Release Notes"; Magni, 2012; Samsung, 2011.

Flash 11.4 (the latest version) targets Windows and Mac. AIR 3.4 (the latest version) additionally targets iOS and Android. Previous versions of

Flash and AIR have targeted other platforms (Table 5), though not necessarily with the same features as on Windows and Mac. For example, Flash for Android lacked camera access, though AIR for Android has it (Adobe, "Flash Player Release Notes"; Adobe, "Adobe AIR Release Notes"). BlackBerry Tablet OS and certain television sets (Magni, 2012; Samsung, 2011) have vendor-supported versions of Flash and AIR that lag behind the official Adobe versions.

Flash applications are able to run automatically when a user with the Flash plugin visits a webpage. To reduce the risks associated with auto-running Web applications, Flash provides certain privacy and security features (Adobe, "Flash Player security and privacy"). When an application wants to access the user's camera or microphone, the user is prompted for permission. This permission is per-application and, normally, per-use. Cross-domain scripting is prohibited, such that one Flash application cannot launch another, remote Flash application. Access to the user's local filesystem is prohibited, except for locally hosted applications that are not networked. Cookie-like data called local shared objects (LSOs) may be stored on the user's machine but, since Flash 10.1, this functionality is disabled when Internet Explorer, Firefox, Chrome, or Safari is in private browsing mode. Compared to Flash, AIR has looser security restrictions but AIR applications do not run automatically; they must be installed as regular desktop applications.

Despite the limitations imposed for privacy and security, Flash supports peer-to-peer networking and thus is a viable platform for a broad range of ubiquitous applications. (See "2.4.2: Relevance to an AR Framework".) Adobe's official peer-to-peer service, available in Flash 10 and later, is called Cirrus (Adobe, "Cirrus"). Under Cirrus, an Adobe

server helps peers discover each other and thereafter peers may communicate directly, exchanging arbitrary data. These communications use Adobe's Real-Time Media Flow Protocol (RTMFP), which is supposed to offer low latency, high reliability, and strong encryption (Adobe, "RTMFP FAQ").

For Flash 9 and later, applications are programmed primarily in ActionScript 3.0 (AS3). ActionScript 3 is an object-oriented, event-driven, reflective, imperative language, influenced by Java and JavaScript. ActionScript 3 compiles to bytecode and, when used reflectively, may emit bytecode or JavaScript. Types in AS3 may be either static or dynamic. They may not be generic (except in the case of vectors), nor may they overload the equality operator. Functions are first-class citizens (Adobe, "ActionScript Technology Center").

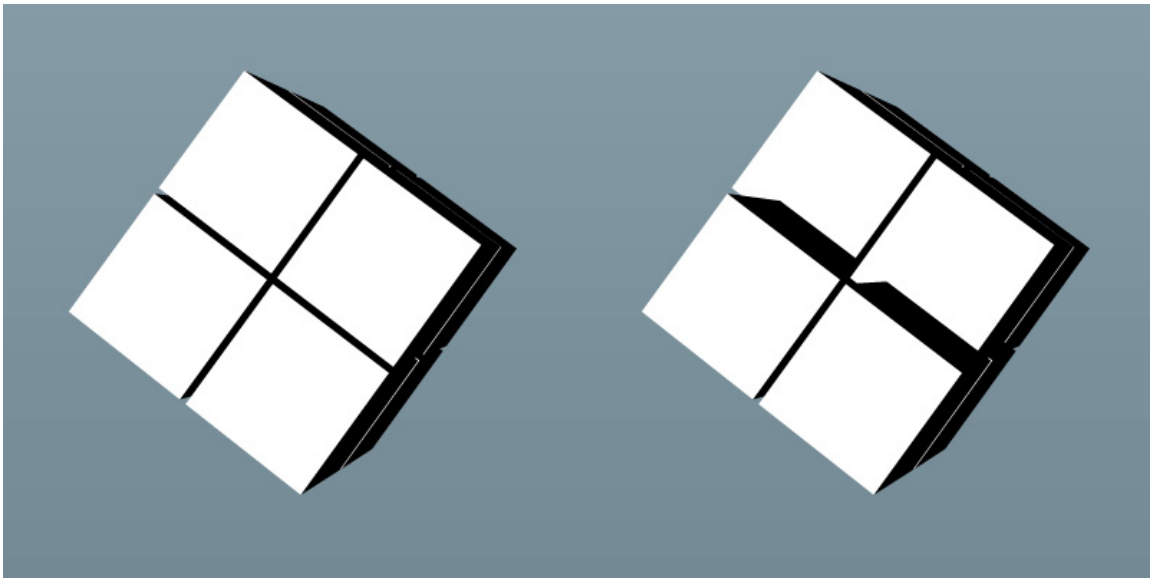
The standard libraries for AS3 emphasize GUIs, media capture/playback, and 2D vector graphics. An AS3 application is structured as a **scene graph**: a hierarchy of entities with spatial coordinates (in this case, pixels coordinates). Each node in the graph may dispatch and receive events. The scene graph may be laid out in MXML, a markup language that is interoperable with AS3.

Flash 9 and later also include experimental (prerelease) support for compiled C and C++. The Flash toolchain for these languages is known as Alchemy. Recent prerelease versions of Alchemy are available only to selected developers (Adobe, "Alchemy").

Adobe acknowledges that in Flash 10 and earlier, the platform was incapable of supporting high-performance 3D rendering. The company suggests that with "acceptable performance", Flash 10 applications could render at most 4,000 triangles per frame. Furthermore, depth sorting was

only feasible per-triangle rather than per-pixel; thus, portions of triangles would sometimes appear to be misplaced (Adobe, "How Stage3D Works") (Figure 7).

**Figure 7: Depth Sorting: Per-pixel v. Per-triangle**



Left: Per-pixel depth sorting, yielding the correct rendering of four cubes that are close to each other. Right: Per-triangle depth sorting, yielding an incorrect rendering. For performance reasons, implementations of depth sorting targeting Flash 10 are typically per-triangle (Adobe, "How Stage3D Works"; Chúťka, 2010). The screenshots are from Chúťka (2010).

An API called **Stage** is Flash's default rendering pipeline—and was, until recently, its only rendering pipeline. Stage is highly portable: all features can run on one CPU core—and many features can *only* run in this manner. By the same token, Stage is suboptimal for anything but low-end hardware. However, since Flash 9, Adobe has been adding platform-specific optimizations to Stage, such that some functionality has become well optimized, at least relative to other Web-based alternatives (Ernst,

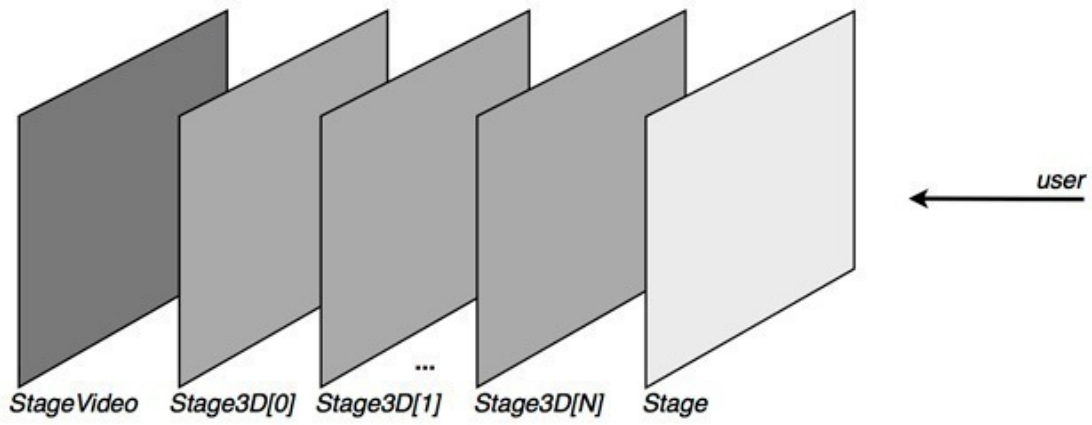
2011). Flash Player 9 Update 3 introduced optimizations for multicore CPUs, with the effect of improving the speed of built-in rendering functions for vectors, bitmaps, filters, and video (Adobe, "Flash Player Release Notes"; Ulloa, June 14, 2007). Video decoding is GPU-accelerated since Flash 10.1 (Adobe, "Flash Player Release Notes"). Flash 10.0 enabled developers to program their own optimizations for multicore CPUs via the Pixel Bender shader language, which parallelizes vector operations (Adobe, "Flash Player Release Notes"; Uro, 2008). Since Flash 11.4, clients can also do CPU multiprocessing in ActionScript by programmatically launching multiple "Workers" or virtual instances of the Flash runtime, each with its own thread of execution. Workers have high overhead cost but may cheaply communicate with each other via shared memory or message passing (Adobe, "Worker").

Starting in Flash 10.2, applications may be able to offload streaming video rendering to the GPU via an API called **StageVideo**. StageVideo is always rendered behind Stage—a suitable ordering for "the most common use case, which is a video player application" (Adobe, "Getting started with stage video"). There may be zero or more StageVideos available to an application, depending on the underlying platform. As each StageVideo is full-window and does not support transparency or other blending (Adobe, "Getting started with stage video"), the use case for multiple StageVideos would seem to be limited to multi-window or multi-display AIR applications.

For reasons of efficiency, previous versions of Flash made the contents of StageVideo write-only: the frames of video data could not be read by client code (Adobe, "Getting started with stage video"). Moreover, StageVideo did not support camera input. Without camera input or

readable video frames, StageVideo was doubly ill-suited to the needs of AR. However, a solution was previewed in a beta version of Flash in October 2011 (Imbert, 2011). Ten months later, StageVideo gained client-readable camera input in the release version of Flash 11.4.

**Figure 8: Ordering of Stages in Flash 11**




At the front (nearest the viewer), there is exactly one Stage, rendered via the old, CPU-bound, programmable pipeline. Behind it, there are zero or more Stage3Ds, rendered via the new, GPU-accelerated, programmable pipeline. At the rear, there are zero or more (though typically one) StageVideos, rendered via another GPU-accelerated pipeline, which is fixed-function. The diagram is from Adobe ("How Stage3D Works").

Since Flash 11.0, the original Stage and StageVideo are supplemented by an alternative called **Stage3D** (formerly, Molehill), which is GPU-accelerated. Stage3D supports parallel programming on the GPU via **Adobe Graphics Assembly Language (AGAL)** or the higher-level **Pixel Bender 3D (PB3D)** language, though the future of the latter language is uncertain (Adobe, "What is AGAL"; Adobe, "Preview 3 and the future of PB3D"). There may be zero or more Stage3Ds available to an

application, depending on the underlying platform (Adobe, "How Stage3D Works"). Windows and Mac can use Stage3D since Flash 11.0, while iOS and Android AIR applications can use it since Flash 11.2 (Adobe, "Flash Player Release Notes"; Adobe, "Adobe AIR release notes"). Stage3Ds are sandwiched between the Stage and StageVideo (Figure 8) (Adobe, "How Stage3D Works"). Stage3D does not yet support background transparency so if one Stage3D is visible, it obscures further-back Stage3Ds and StageVideo. Background transparency for Stage3D was previewed in a beta version of Flash in summer 2011. However, there is no news about further development of this feature.

Stage3D does not have any functionality for camera input or streaming video rendering. Therefore, in itself, Stage3D is an insufficient canvas for AR. To use Flash for GPU-accelerated 3D rendering in front of live video, one must make use of another graphics pipeline as well. Two approaches are feasible:

1. Marshall camera frames from Stage to Stage3D, and render them on Stage3D as a textured plane behind the virtual scene.
2. Marshall the virtual scene's frames from Stage3D to Stage, and render them as a bitmap in front of a live video that resides on either Stage or StageVideo.

Neither approach is ideal: it would be more efficient to have camera input stored to a Stage3D texture in the first place, or to have a transparent background in Stage3D so that StageVideo could show through. Thus, compositing is problematic for AR in Flash.

## Chapter 3: Exploratory Work

The original motivation for this thesis grew out of the author's work at Ad-Dispatch, Inc. (Dartmouth, NS), an AR company, in the period of July 2011 to March 2012. During this time, Ad-Dispatch saw the need to transition away from its main third-party software dependency, which, though efficient, was expensive, fragile, and troublesome to Web users. The company sought alternatives that would be equally efficient but more ubiquitous: practical even for low-budget projects, varying host environments, and novice users. For certain platforms, Ad-Dispatch found an off-the-shelf, integrated solution to its problem. For other platforms, including the Web, there was no such readymade engine, though there were relevant off-the-shelf components.

As part of his employment, the author integrated available AR, rendering, media, and GUI components in Flash (plus AIR) to test several alternative concepts of an AR engine targeting the Web (plus other platforms). The results of these efforts are detailed in the next section, "3.1: At Ad-Dispatch". Broadly, the concepts proved functional and deployable, though efficiency and workflow (particularly, turnaround time for testing art assets) were problematic to varying extents. Ad-Dispatch suspended its plans to develop an engine in-house.

Thereafter, the author entered into an informal collaboration with Bernhard Jung, one of the developers of flare*nft, to start assessing another round of integration concepts, with an emphasis on reducing marshalling costs, utilizing the GPU via thinner wrappers, and supporting asset imports more simply. The results of this dialogue are detailed in the "Refinements" section. Broadly, the consensus was that the new approach yielded improvements in efficiency and workflow, relative to the author's

and Jung's previous models. Further validating the use of thin wrappers, the author tried unsuccessfully to attain the same performance when adding AR atop an existing, high-level game engine.

There are two takeaway lessons from this exploratory work. First, the combination of ubiquity and efficiency is critical in an industry-grade AR engine—and finding or creating this combination is nontrivial. Second, Flash (or AIR) solutions can potentially deliver this combination, provided that care is taken to use the strengths and avoid the weaknesses of the platform's multiple graphics pipelines. Particularly, complications arise in sharing data among components. These lessons influence the architecture and choice of dependencies that we discuss in the next chapter, "Design and Contribution".

## 3.1: At Ad-Dispatch

### *3.1.1: Objectives and Problems*

Ad-Dispatch specializes in rapid development of multi-platform NFT applications with 3D content. The purpose of its applications is to add interactive value to clients' physical products, advertisements, and venues. Users include media technicians, commercial salespeople, retail salespeople, and consumers.

Within this business context, reusable application frameworks (both code and interface principles) are important for the sake of meeting the short timelines while still ensuring robustness, responsiveness, usability, and polish for the multiple platforms, contexts, and audiences. Of course, not all application frameworks fulfill these needs equally well.

As of 2011, Ad-Dispatch was heavily invested in the D'Fusion SDK but was encountering problems with this framework. First, D'Fusion carried high recurring costs per application and per revision of an application. This cost structure was prohibitive to many of the small-scale and recurring contracts that would have suited Ad-Dispatch's specialty in rapid development. Second, D'Fusion does not truly provide uniform development and deployment experiences across the supported platforms. For D'Fusion Web Player, many end users complained of installation failures. On iOS and Android, the application developer must write glue code to manage D'Fusion's proprietary runtime. The vendor-provided samples of glue code proved to be unreliable (in the author's attempts to adapt them)—suffering from memory leaks, camera resource leaks, and graphics resource leaks in certain situations where the runtime needed to be paused and resumed.

The residual appeal of D'Fusion lay in its convenient visual toolchain, and its consistent graphical features and performance across Web and desktop platforms. Ad-Dispatch's artists, in particular, were satisfied with D'Fusion because it could reliably import their 3D animations and show previews that corresponded well to runtime results (at least on the Web and desktops).

To understand the importance of previewing 3D animations early and often, let us look at the typcial steps in Ad-Dispatch's workflow:

1. Managers and salespeople establish project specifications in consultation with clients. These specifications are then communicated to artists and programmers.
2. Artists create content while programmers work on implementing other aspects of the application.
3. Artists send content to programmers, who attempt to incorporate the content into the application. The application or an in-editor simulation of it may be shown to artists for review. If there are problems at this stage and there does not seem to be any convenient programmatic solution, programmers and artists meet to discuss possible causes and solutions. These discussions may result in rework for the artists, in which case the workflow returns to (2).
4. A demo build of the application is sent to the client for review. This demo may be work-in-progress. The client may request changes. Depending on the client's requests and whether the demo is work-in-progress, the workflow might return to (1) or (2).
5. A client-approved application is tested by various staff and deployed by programmers.

If timelines for incorporating content are not met, then artists and programmers are tied up in the effort to resolve the technical problems, while managers and salespeople may be tied up in client-relations problems because there is no new demo content for the client to see.

Ad-Dispatch decided to try to replace D'Fusion with one or more alternative frameworks, which would need to be less costly and more robust but would ideally offer the same type of workflow, and at least

equal graphical features and performance. A comparison of mobile alternatives led to the adoption of Vuforia plus Unity as an obvious choice for iOS and Android projects, with Vuforia providing the NFT functionality and Unity providing the game engine functionality and visual toolchain. A comparison of Web and desktop alternatives did not produce any obvious match to the criteria, so the author was assigned to investigate further and to work on developing better matches.

### *3.1.2: Approaches and Outcomes*

At different times, the author explored possible Flash 10, Flash 11, and equivalent AIR solutions using flare*nft. As discussed in "2.3.3: Frameworks", flare*nft belongs to the same evolutionary group as Vuforia, so the choice of flare*nft would facilitate cross-platform feature parity for Ad-Dispatch. The scope of the explorations included identifying integration issues among the platform's standard libraries, various third-party rendering libraries, and flare*nft itself.

Performance profiling was done on Mac OS X 10.7, 2.4 GHz Core 2 Duo, 4 GB RAM, GeForce 320M. Functional testing was done on multiple platforms.

One proposed approach—the most conservative—was to closely follow flare*nft's demo application code, which integrated the Papervision renderer. As the pioneering 3D renderer for Flash, Papervision was first publicly released in 2007 (Ulloa, July 7, 2007). The latest stable version, dating to 2009, is optimized for Flash 9 (Ulloa, October 13, 2009). As such, Papervision is highly suboptimal for today's Flash audience. On Flash 10.3, it was possible to produce an NFT application that ran at 12 FPS with an animated, 20,000 polygon model and 640x480 video.

However, to achieve this frame rate, it was necessary to use an inaccurate depth-sorting algorithm, which is actually the default in Papervision (Chúťka, 2010). This option caused an inordinate amount of rework for the animator, since the model's geometry needed to be sliced up in unusual ways to compensate for the algorithm's flaws. With accurate depth-sorting, the application ran at only 4 FPS.

Other proposed approaches involved more up-to-date renderers, such as Away3D and Alternativa3D. These two renderers are available in both Flash 10-optimized and Flash 11-optimized versions. Away3D is forked from Papervision so may offer the easiest upgrade path for legacy code. Alternativa3D is an original library with an impressive portfolio, including Adobe's official launch demo for Stage3D and several massively multiplayer online games (MMOGs) (AlternativaPlatform, "Showcase"). Approaches using either of these two renderers were not explored very far, due to Ad-Dispatch's concerns about the time investment that might be required for programmers and animators to troubleshoot another new rendering pipeline.

A final proposed approach—the most radical—was to integrate flare*nft with Unity via interprocess communication, such that artists and front-end developers would only need to deal with the known Unity workflow. (This effort predated Unity's built-in support for targeting Flash). This approach was explored mainly with respect to AIR deployment for kiosks running Windows or Mac. Testing revealed an AIR incompatibility in flare*nft, so an AIR-compatible custom version was obtained from the vendor. An open-source demo application called UnityFlashCam (Rooney, 2011) was studied as an example of marshalling video frames from AIR's input to Unity's output via asynchronous socket

communication. Extending and optimizing this example, the author's demo marshalled both video frames and NFT pose estimates from AIR to Unity. The asynchronous communication had the desirable effect of decoupling the camera/NFT frame rate in Flash from the faster rendering frame rate in Unity, such that the animated foreground did not skip frames even when the video background and tracking did. On AIR 3.1 and Unity 3.4, rendering at 60 FPS was possible for millions of polygons, while a 320x240 live video ran at high frame rates in the background. However, video lag was somewhat noticeable, and at higher video resolutions the background's frame rate deteriorated, becoming unusable in the case of HD video. Ultimately, the bottleneck was not the socket communication (the marshalling between application contexts) but rather the upload of video frames from main memory to GPU memory (the marshalling between hardware contexts).

The company did not reach any decision on its technological strategy for the Web and desktop-based kiosks. For the short term, flare*nft plus Papervision saw small-scale use, while D'Fusion continued to be the mainstay. The investigation of other alternatives was shelved in December 2011, pending possible new information in Q1 2012 about the flare*nft roadmap, Unity roadmap, and business opportunities for the Web and kiosks. Ultimately, in that quarter, no new information proved conclusive with respect to Ad-Dispatch's criteria.

### 3.2: Refinements

The idea of integrating flare*nft with an up-to-date, Flash 11 renderer continued to seem plausible, except that the timeline differed from the expectations in an application development company specializing in rapid

turnaround. The author undertook to develop an integration demo for his own research purposes, and then validate this work in consultation with Bernhard Jung, one of flare*nft's developers.

The integration demo is simply called FlareNFTAlternativa3D. It is an optimized and parameterized port of flare*nft's "Austrian Cubes" demo (which integrates flare*nft and Papervision). "Austrian Cubes" uses three marker images: of Austria, Vienna, and Graz. When the user holds a marker in front of the webcam, a cube bearing the flag or crest of the given place is rendered over the marker in the video feed. On two of the markers, certain regions are enabled as **virtual buttons** that trigger a logging function when physically touched (or when otherwise occluded). To exercise the superior rendering efficiency of Alternativa3D, the port uses additional 3D content: atop each cube sits a 13,470-triangle apple.[2] The port's virtual buttons are not responsible for logging but instead for causing certain apples to disappear/reappear, or stop/start rotating.

Four main problems are addressed in the optimized port of "Austrian Cubes". These problems relate to:

1. rendering the 3D content in front of the video;
2. calibrating the virtual camera's perspective to match the video camera's supposed perspective;
3. sanity-checking assumptions about the video camera's perspective;
4. supporting HD video without additional burden on the tracking algorithm.

[2] The apple model is courtesy of Teinye Horsfall at WireCASE Ltd (http://www.wirecase.com).

Some elements of these problems arise from shortcomings in the original (Papervision) "Austrian Cubes"; others arise from differences between Papervision and Alternativa3D, or between Stage and Stage3D. The particular complications and solutions are as follows.

First, like the Flash 11 platform in general, Alternativa3D suffers from the rendering order problem described in "2.6.4: Focus on Flash": content that fully utilizes the GPU-accelerated Stage3D pipeline cannot be rendered in front of content that fully utilizes the CPU-bound Stage pipeline, where camera input resides. Moreover, Alternativa3D does not offer any built-in functionality for streaming video frames from Stage to Stage3D. However, the opposite route is well supported: Alternativa3D can do partially GPU-accelerated rendering to bitmaps residing on Stage—in our case, to an otherwise transparent bitmap in front of the video. This convenient approach proved to offer good enough performance to validate the choice of Alternativa3D over Papervision (Table Series 6).

Second, flare*nft assumes that certain values it provides will be written to the virtual camera's projection matrix, with the intent of matching the video camera's perspective. Papervision permits direct editing of the projection matrix, while Alternativa3D instead provides high-level functions that abstract the editing of the the projection matrix. To determine the correspondence between the raw matrix exposed by flare*nft and the abstractions exposed by Alternativa3D, it was necessary to reverse engineer the derivation of each, mostly via black-box testing. (Alternativa3D was closed-source at this time, though now it is open-source.) Notable parameters of flare*nft's projection matrix derivation are described below in relation to the third problem. A notable parameter

of Alternativa3D's projection matrix derivation is documented as "fov" or "Field of view" (AlternativaPlatform, "Camera3D - API Documentation") but more precisely it represents diagonal field of view. This meaning was unexpected because OpenGL and Direct3D use vertical field of view (OpenGL, "gluPerspective"; Microsoft, "D3DXMatrixPerspectiveFovLH function").

Third, flare*nft relies on the video camera's optical and digital characteristics being defined in a configuration file, in a format specified by the ARToolKitPlus project. The required data in this config file include the pixel dimensions of the captured video and the lens's focal length divided by the pixel pitch (B. Jung, personal communication, December 28, 2011; Christian Doppler Laboratory, "ARToolKitPlus"). The latter datum relies on *a priori* knowledge of the video camera's engineering specs. As such, the true value is unknowable except in controlled setups such as kiosks. The original "Austrian Cubes" invariably uses a configuration that represents a 320x240 video feed with a 72° diagonal field of view (medium-wide, equivalent to a 30mm focal length in 35mm photography). Testing revealed that flare*nft's tracking accuracy degrades drastically as the video feed's actual aspect ratio diverges from the configuration. FlareNFTAlternativa3D addresses this problem by choosing among multiple config files based on runtime measurements of the camera's aspect ratio. Generating the config file itself at runtime would be another option. Meanwhile, misconfiguration of focal length per pixel pitch produces lesser variances in tracking accuracy: across the ranges tested, this misalignment was barely noticeable in Austrian Cubes.[3] FlareNFTAlternativa3D does not do any runtime

[3] Other, less contrived scenarios might highlight misalignment more

validation of focal length per pixel pitch. Such validation could be done via user-assisted calibration exercises that would yield an estimate of the video camera's field of view.

Fourth, the cost of flare*nft's tracking algorithm increases with the resolution of the video being processed. Capturing HD video frames for display may be desirable in some contexts, yet processing them with flare*nft proved to be impractical in real time. (The original "Austrian Cubes" does not attempt it.) FlareNFTAlternativa3D addresses this problem by applying efficient downscaling (via the Stage pipeline) to the video data that is sent to flare*nft (while the rendered frame is not downscaled). Thus, the cost of the tracking algorithm remains constant regardless of the video resolution used in capture and display.

Table Series 6 summarizes FlareNFTAlternativa3D's performance, varying with respect to the marker, number of virtual buttons, video input resolution, and 3D content. "**Seeking**", in the table, refers to periods when no marker is recognized and, consequently, no 3D content is being rendered. "**Tracking**" refers to periods when one of the markers is recognized and, consequently, one of the sets of 3D content is being rendered. "**Video only**" refers to the NFT algorithm and 3D rendering both being turned off, for comparative purposes. Performance is capped at 60 FPS due to camera hardware limitations and Flash's synchronous processing of the video input frames. Between seeking and tracking, the cost of video rendering should stay constant, the cost of the NFT algorithm should decrease, and the cost of 3D rendering should increase

---

clearly. There is no particular reason for a user to expect a cube to sit dead-center atop a picture of Austria. However, in a virtual cutaway view of a medical drawing, a user would expect a particular alignment.

(from nothing). If the solution to the fourth problem is effective, the cost of the NFT algorithm should not vary with respect to the video input resolution. The cost of the 3D rendering may vary with respect to the video input resolution because the resolution of the 3D rendering is increased to match. The performance results suggest that resolution rather than triangle count is the dominant factor in the ranges tested. The solution to the fourth problem is validated, as performance during seeking remains constant across several video input resolutions.

We believe that in Web use, an AR application with these performance capabilities can meet or exceed consumer expectations. 640x480 and 1280x720 are large canvases in the context of Web design, and speeds approaching 60 FPS or 30 FPS are typical in games. For comparison, the popular Wii game console is limited to exactly 640x480 resolution at either 60 FPS or 30 FPS (depending on the television). One Wii developer claims that the system is capable of rendering approximately 80,000 polygons at 60 FPS (Richardson, 2009). Again, the performance seen in FlareNFTAlternativa3D comes close to this level.

## Table Series 6: Performance of FlareNFTAlternativa3D

**System:** Flash 11.2, Firefox 11.0, Mac OS X 10.7, MacBook Pro 13" mid-2010. Built-in iSight camera for 320x240 and 640x480 resolutions; USB Logitech HD Pro C920 for 1280x720 and 1920x1080 resolutions.

**Table 6A: Vienna Marker, 0 Virtual Buttons**

| Resolution of video input and canvas | Triangles | FPS, seeking | FPS, tracking | FPS, video only |
|---|---|---|---|---|
| 320 x 240 | 12 | 23 | 60 | 60 |
| 320 x 240 | 13,482 | 23 | 60 | 60 |
| 640 x 480 | 12 | † 23 | † 50 | 60 |
| 640 x 480 | 13,482 | 23 | 44 | 60 |
| 1280 x 720 | 12 | 23 | 22 | 60 |
| 1280 x 720 | 13,482 | 23 | 19 | 60 |
| 1920 x 1080 | 12 | 13 | 9 | 35 |
| 1920 x 1080 | 13,482 | 13 | 9 | 35 |

**† Baseline (original "Austrian Cubes"):** 24 FPS, seeking; 60 FPS, tracking

**Table 6B: Austria Marker, 1 Virtual Button**

| Resolution of video input and canvas | Triangles | FPS, seeking | FPS, tracking | FPS, video only |
|---|---|---|---|---|
| 320 x 240 | 12 | 23 | 55 | 60 |
| 320 x 240 | 13,482 | 23 | 55 | 60 |
| 640 x 480 | 12 | † 23 | † 43 | 60 |
| 640 x 480 | 13,482 | 23 | 40 | 60 |
| 1280 x 720 | 12 | 23 | 22 | 60 |
| 1280 x 720 | 13,482 | 23 | 19 | 60 |
| 1920 x 1080 | 12 | 13 | 9 | 35 |
| 1920 x 1080 | 13,482 | 13 | 8 | 35 |

**† Baseline (original "Austrian Cubes"):** 24 FPS, seeking; 50 FPS, tracking

**Table 6C: Graz Marker, 2 Virtual Buttons**

| Resolution of video input and canvas | Triangles | Rotating content | FPS, seeking | FPS, tracking | FPS, video only |
|---|---|---|---|---|---|
| 320 x 240 | 12 | No | 23 | 38 | 60 |
| 320 x 240 | 13,482 | No | 23 | 37 | 60 |
| 320 x 240 | 13,482 | Yes | 23 | 37 | 60 |
| 640 x 480 | 12 | No | † 23 | † 30 | 60 |
| 640 x 480 | 13,482 | No | 23 | 30 | 60 |
| 640 x 480 | 13,482 | Yes | 23 | 30 | 60 |
| 1280 x 720 | 12 | No | 23 | 19 | 60 |
| 1280 x 720 | 13,482 | No | 23 | 19 | 60 |
| 1280 x 720 | 13,482 | Yes | 23 | 19 | 60 |
| 1920 x 1080 | 12 | No | 13 | 9 | 35 |
| 1920 x 1080 | 13,482 | No | 13 | 8 | 35 |
| 1920 x 1080 | 13,482 | Yes | 13 | 8 | 35 |

**† Baseline (original "Austrian Cubes"):** 24 FPS, seeking; 40 FPS, tracking

By comparison, the original "Austrian Cubes" renders 12 triangles and 640x480 video at 40 FPS to 60 FPS (depending on the marker and number of virtual buttons), with visible rendering flaws such as distorted and non-antialiased textures, as well as noticeable lag in both the video and the tracking. Based on the author's Papervision-based work at Ad-Dispatch, as well as comments from Adobe ("How Stage3D Works") and the Papervision community (Grden, 2011), it is clear that Papervision's

performance deteriorates rapidly as the triangle count increases into the 1,000s or 10,000s, especially if accurate depth sorting is used. Although Alternativa3D seems to have greater overhead (handicapping it for small triangle counts), it does not suffer from these rendering flaws nor from such severe falloff of performance. Moreover, FlareNFTAlternativa3D does not suffer from any noticeable lag when using 640x480 video, and even when it uses 1280x720 video, its lag is less than that of the original "Austrian Cubes" (using 640x480 video). The reduction in lag is attributable to the efficient resampling of video frames for flare*nft's purposes.

For practical purposes, 1920x1080 video input proved to be unusable on the test system. Video lag exceeded 15 seconds, even when the video was the only content running.

The source code of FlareNFTAlternativa3D was provided to Jung, who confirmed that the approach and the output were valid (B. Jung, personal communication, January 3, 2012). Based on the demo and further experimentation of his own, Jung also noted that Alternativa3D's content pipeline is simpler and more robust than Papervision's (B. Jung, personal communication, January 16, 2012). As the result of these explorations, the development version of flare*nft now supports closer integration with Alternativa3D (B. Jung, personal communication, April 12, 2012), so further improvements in performance and workflow can be expected.

As an alternative to using Alternativa3D, the author also retested the concept of integrating flare*nft with Unity. By this time, new features in Unity 3.5 allowed for Unity applications to be embedded within Flash, making interprocess communication unnecessary. Unfortunately, the

author's efforts at merging Unity's 3D content with with video input from Stage degenerated into the same bottleneck as in the previous integration attempt. Overall performance actually worsened; the interprocess approach had, in its favor, the ability to place video capture and NFT tracking on separate threads.

Going forward, all of this exploratory work is important in suggesting that a renderer built from the ground up for Stage3D—specifically, Alternativa3D—is viable for use in a gaming-quality AR framework for the Web. By comparison, several alternatives are not as viable, due to problems of either performance (Papervision, Unity for Flash) or reliability (Papervision, D'Fusion).

## Chapter 4: Design and Contribution

We present an AS3 solution targeting Flash 11. This solution, called Illusion SDK, draws on existing third-party libraries to provide the foundations of its tracking and graphics engine functionality. (See "Appendix A: Availability and Licensing" for information on obtaining Illusion and dependencies.) However, compared to its dependencies and other previous work, Illusion offers greater generality and extensibility by virtue of a high-level, modular design.

General and extensible solutions are motivated by a desire to avoid rework, particularly in the event that the use case changes or a given implementation proves to be too limited. Relevant anecdotes of rework are offered in the previous chapter, "Exploratory Work". Although Illusion is not the only framework that attempts to abstract AR functionality, the alternatives miss some foreseeable types of rework. For example, they may obstruct the programmer from changing the source of sensor data, changing the tracking algorithm, using multiple tracking algorithms at once, tracking duplicates of physical markers, or changing or removing the rendering functionality. (See "4.3: Comparison to Other Designs".)

Illusion's design includes abstractions for sensors, trackers, and compositors. Via a simple wrapper, any tracker can integrate with any source of sensor data and—optionally—with any technique for compositing and rendering an AR scene.

Ubiquitous applications can be built atop Illusion, as its abstractions do not assume any I/O pattern that is specific to personal computing. For example, a sensor need not be a camera or any other local peripheral; it

could be a network of peers. The tracking results need not be rendered atop a video; indeed, they need not be rendered at all.

At the same time, applications built atop Illusion can be efficient. The abstractions are not tightly coupled to each other, so the client is free to pick and choose among possible implementations and interactions based on their optimality for the given application and platform. For example, multiple trackers can read from one sensor without caveats. The sensor and compositor can be selected independently of the tracker. (See the rest of this chapter.) Empirically, Illusion proves to be efficient insofar as it adds no frame lag and negligible time cost relative to the underlying library functions of the trackers, graphics engine, and Flash. (See the "Evaluation" chapter.)

Illusion is modular: it can be compiled as multiple libraries with sparse interdependencies. Thus, client code can use just part of the functionality without having to pay overhead for the whole. We provide an example application that uses camera input, multiple types of trackers, lighting, and textured models loaded from external files. Another application might use only one type of tracker, or might analyze a virtual drawing canvas instead of a camera feed. Yet another application might use only Illusion's model loading functionality. All of these use cases are intended (and none is handicapped) by Illusion's design.

This chapter proceeds by describing, in a top-down fashion, the design and usage of Illusion's AR-related functionality. Illusion's more general-purpose functionality (ex. for loading files and setting up 3D scenes) is described in "Appendix B: Non-AR Functionality" instead. After describing AR functionality, this chapter presents an example application.

Last, Illusion is compared to other designs in terms of generality and extensibility.

### 4.1: AR Functionality

This section illustrates the functionality of Illusion SDK through class diagrams and snippets of client code. The diagrams conform to the **Unified Modeling Language (UML)** standard (Object Management Group, "Unified Modeling Language"). Most of the code snippets are adapted from an example application that is listed in full in the next section, "4.2: Full Example Application". Each diagram and snippet are accompanied by brief remarks on the functionality's motivation, interface, implementation, and design patterns.

Our taxonomy of design patterns comes from Gamma et al (1995). Particularly, Illusion uses the observer pattern, mediator pattern, and adapter pattern. The observer pattern, also known as event-driven programming, relates one object (the observee) to many others (the observers) such that all observers receive notifications about changes to the observee's state. The mediator pattern relates many objects (the mediatees) to one object (the mediator) such that the mediator handles interactions among the mediatees. Often, the mediator pattern makes use of delegation: the mediator (a delegate) implements an interface that is called by at least one of the mediatees (a delegator). The adapter pattern, also known as the wrapper pattern, relates one object (the adaptee) to another (the adapter) such that the adapter's interface masks the adaptee's interface.

## Figure Series 9: Overview of Illusion SDK

### Figure 9A: Design of AR-related Classes and Interfaces

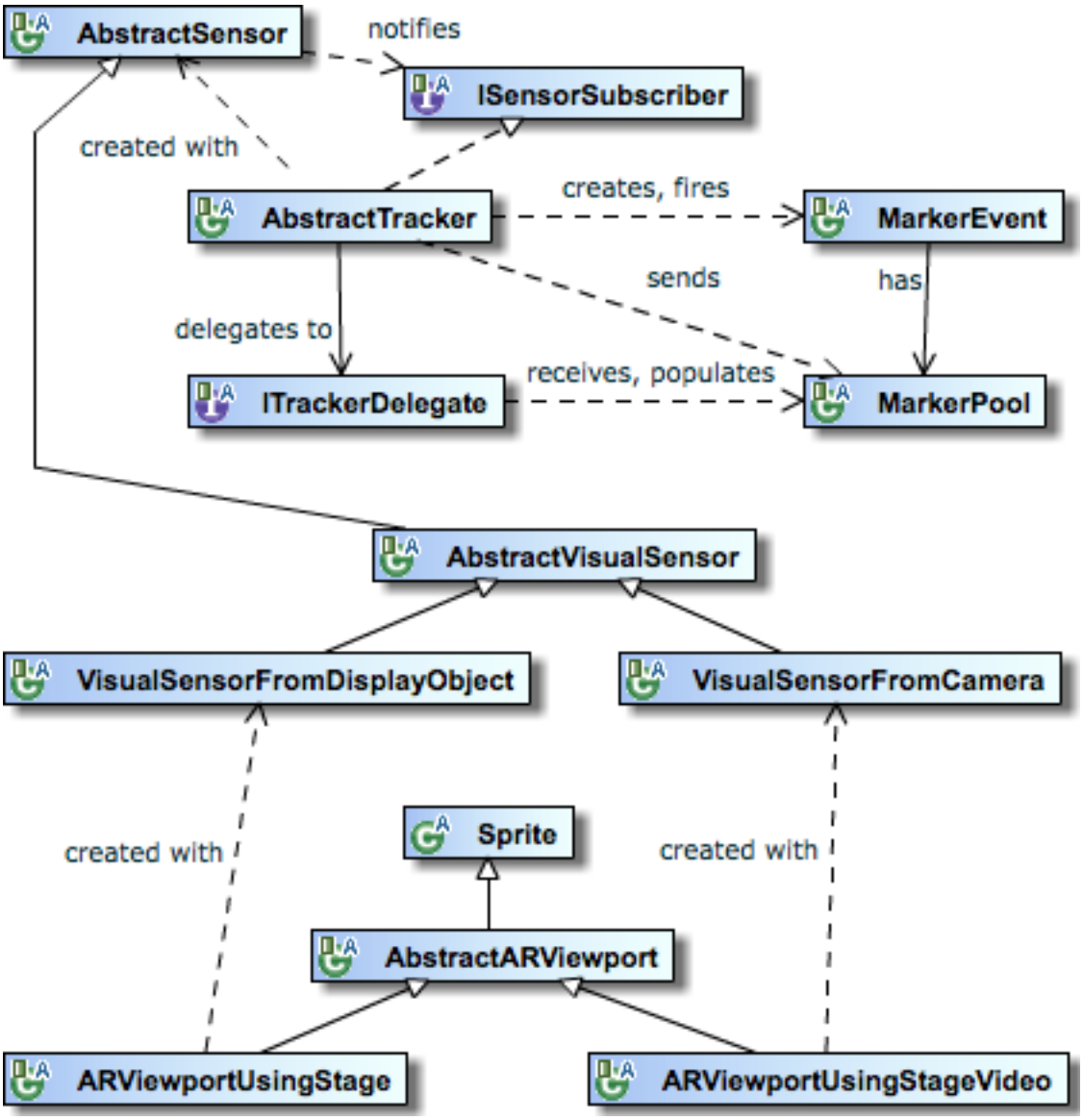
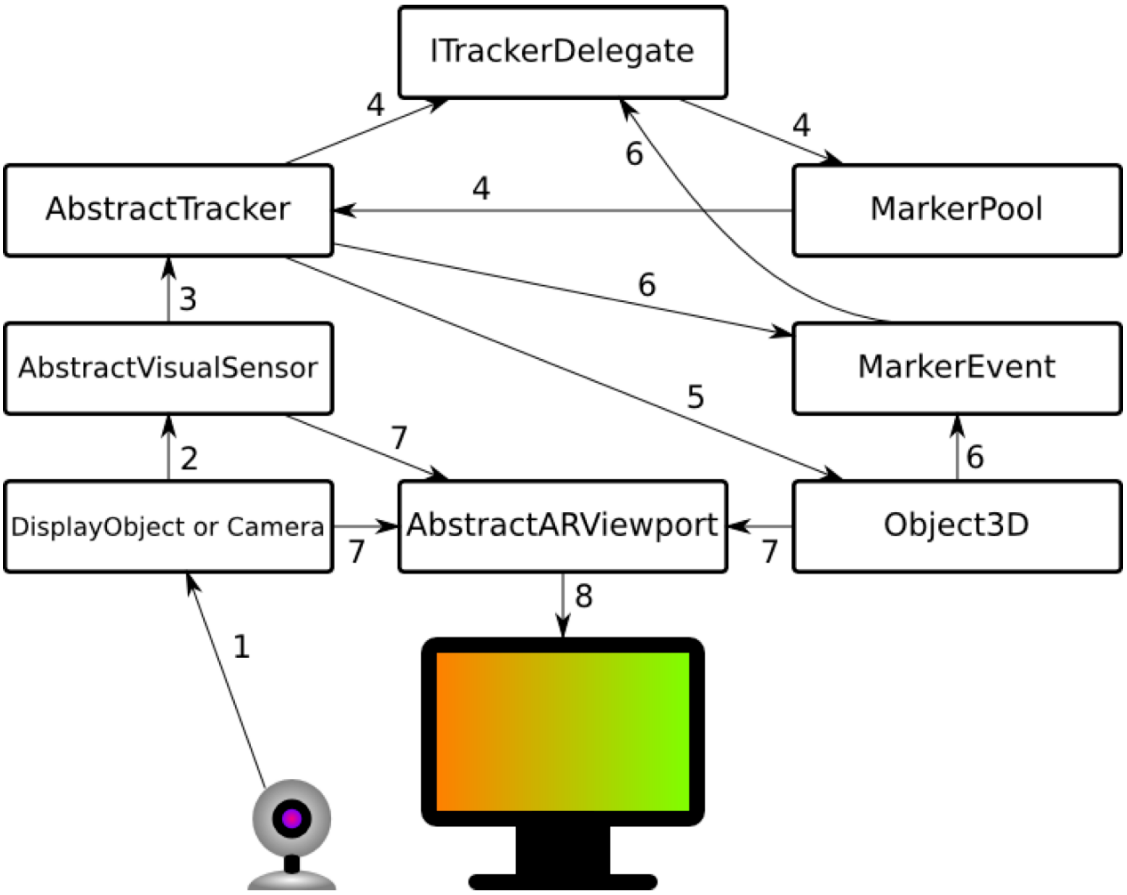


**Figure 9B: Example of AR-related Data Flow**




Our discussion of Illusion focuses on the components that are most specific to AR. Key concepts include (Figure 9A):

- A sensor that captures data about the real world. This concept is represented by the AbstractSensor class, which has specializations dealing with visual data: AbstractVisualSensor, VisualSensorFromDisplayObject (capturing data from a 2D scene node such as a video), and VisualSensorFromCamera (capturing data directly from a camera). Other classes may implement an interface called ISensorSubscriber, which allows instances to subscribe to updates about one or more sensors' data.

- A tracker that updates a 3D scene based on a sensor's data. This concept is represented by the AbstractTracker class, which implements ISensorSubscriber. (Specializations of AbstractTracker are introduced later in this chapter.) Client code provides a delegate and (optionally) event handlers to customize the contents of the tracker's 3D scene. The delegate must implement an interface called ITrackerDelegate, which allows it to receive and populate lists of virtual markers. These lists are of type MarkerPool. Tracked nodes in the 3D scene may receive events of type MarkerEvent.
- A viewport that composites a sensor's visual data and a tracker's 3D scene. This concept is represented by the AbstractARViewport class, which extends Sprite (a type of 2D scene node in Flash). The specializations of AbstractARViewport are ARViewportUsingStage (typically used alongside VisualSensorFromDisplayObject) and ARViewportUsingStageVideo (typically used alongside VisualSensorFromCamera).

For a visual AR application, a typical data flow among these components is (Figure 9B):

1. Flash updates a 2D node or camera.
2. An AbstractVisualSensor stores the node or camera's pixels.
3. An AbstractTracker reads the AbstractVisualSensor's pixels.
4. The AbstractTracker selects 3D nodes that it holds in MarkerPool instances. If it runs out of nodes to select, it asks an ITrackerDelegate to (optionally) supply more.
5. The AbstractTracker updates the selected 3D nodes. The nodes end up sharing a common parent while tracked.

6. The AbstractTracker fires MarkerEvent instances from newly found or lost 3D nodes. Listeners handle the events. The ITrackerDelegate may be a listener.
7. An AbstractARViewport composites the AbstractTracker's root 3D node and AbstractVisualSensor's 2D node or camera.
8. Flash displays the AbstractARViewport.

The design and usage of these components are detailed in the rest of this chapter.

For 3D scene graph functionality and 3D graphics functionality, Illusion interfaces with the Alternativa3D 8 graphics engine. All relationships between Illusion classes and Alternativa3D classes are achieved by composition, not inheritance. Therefore, Illusion's implementation is independent of Alternativa3D's.

For tracking functionality, Illusion interfaces with flare*nft and flare*tracker. However, Illusion provides abstractions to facilitate the future development of other trackers and tracker wrappers. As such, dependencies between Illusion's implementation and third-party trackers' implementations are localized in leaf nodes of Illusion's inheritance tree. Moreover, all relationships between Illusion classes and flare*nft or flare*tracker classes are achieved by composition, not inheritance.

### *4.1.1: Centralizing Access to Sensor Data*

For visual tracking (and image processing in general), access to pixel data is essential. Often, the relevant data are unstable because they correspond to an input device (ex. a camera), or to a branch of the 2D scene (ex. a video) that is being re-rendered continually. For efficiency's sake, when multiple trackers (or other image processing entities) need to

access the same unstable pixel data, they should do so through a shared manager that minimizes acquisition and copying of data. Similar issues apply to non-visual trackers and their access to data. For example, the audio data read from a microphone or a playing sound clip are unstable, too.

We have seen (in "3.2: Refinements") that a source of pixel data may be associated with *a priori* knowledge such as a field of view, and configuration values such as a resolution. These factors may affect the way the data are processed (ex. by trackers) and the way its source is presented (ex. by a compositor).

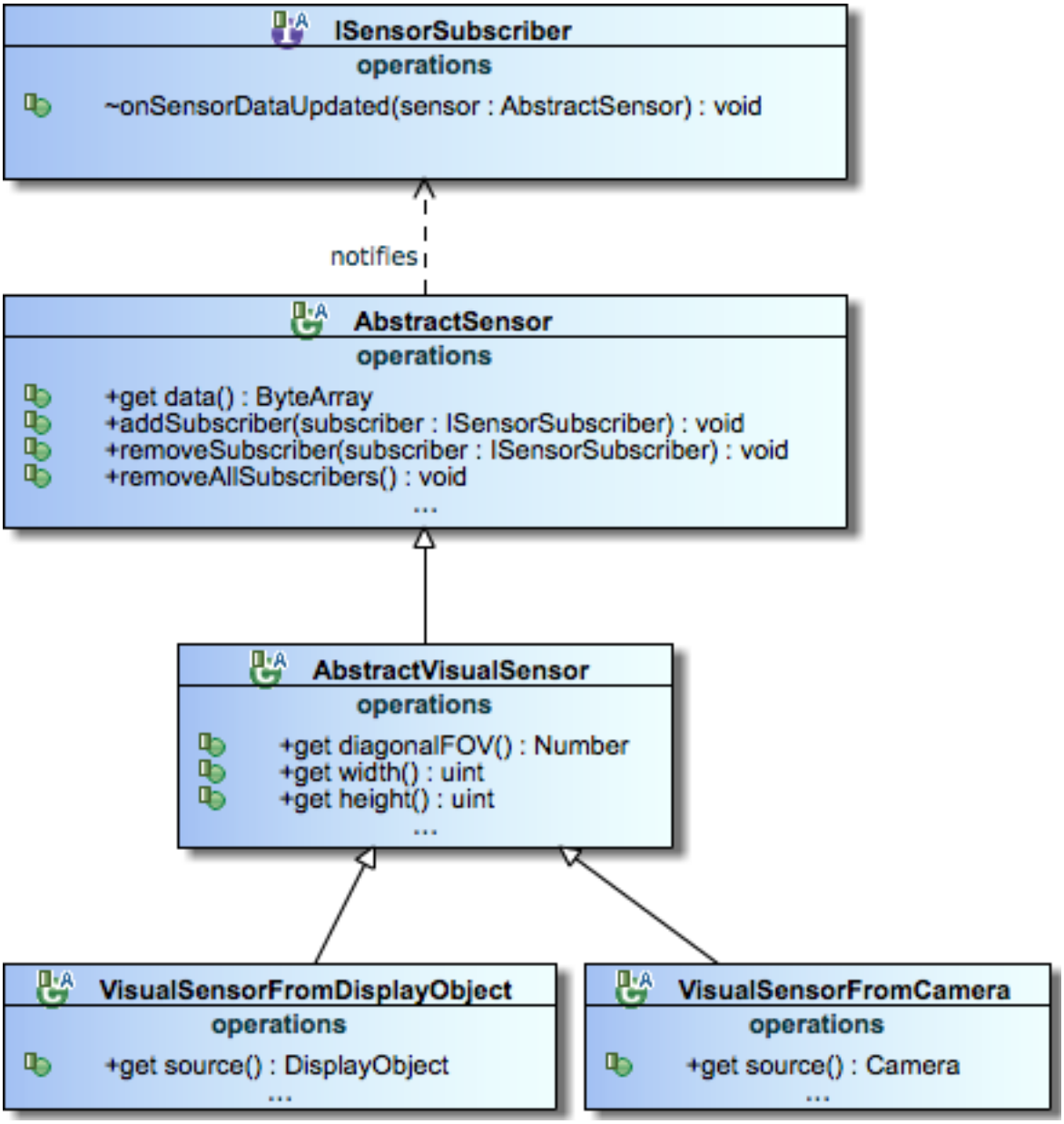
To represent a configurable stream of input, Illusion provides the AbstractSensor class (Figure 10). To more specifically represent a configurable stream of pixel data, Illusion provides a subclass, AbstractVisualSensor. AbstractVisualSensor has two implementations: VisualSensorFromDisplayObject (Code Sample 1) and VisualSensorFromCamera (Code Sample 2). VisualSensorFromDisplayObject captures pixels from a DisplayObject (a standard 2D scene node in Flash), which is specified at instantiation. For example, the DisplayObject could be a live video in the 2D scene. VisualSensorFromCamera captures data directly from a Camera, which is likewise specified at instantiation. The Camera need not be attached to a live video in the 2D scene. Optionally, a FOV and processing resolution may be specified at instantiation of an AbstractVisualSensor subclass; otherwise, certain default values are assumed.

Client code may implement an interface called ISensorSubscriber and subscribe to any AbstractSensor to receive notifications about new sensor data. The subscriber (or other code) may access a ByteArray representing

the current sensor data (ex. the current pixels of the DisplayObject or Camera). As such, AbstractSensor may be considered a mediator between some stream-like class (ex. DisplayObject or Camera) and ByteArray. Internally, AbstractSensor implementations use the observer pattern to coordinate the mediatees.

Typically, client code does not access an AbstractSensor's properties directly or subscribe to its notifications directly. Rather, an AbstractSensor is used in instantiating other types, which internalize the reading of the AbstractSensor's properties and the subscription to its notifications. (See the next two sections: "Compositing 2D and 3D Scenes" and "Tracking Markers".)

**Figure 10: Design of AbstractSensor and Related Types**



### Code Sample 1: Usage of VisualSensorFromDisplayObject

```
class MyVisualSensorFromDisplayObjectSubscriber
implements ISensorSubscriber
{
      var sensor_:VisualSensorFromDisplayObject;

      public function MyVisualSensorFromDisplayObjectSubscriber(
            source:DisplayObject)
      {
            // Create the sensor with the default values for the
            // FOV and resolution arguments.
            sensor_ = new VisualSensorFromDisplayObject(source);
      }

      // The ISensorSubscriber implementation.
      public function onSensorDataUpdated(
            sensor:AbstractSensor):void
      {
            // Get the FOV and resolution values.
            var diagonalFOV:Number = sensor_.diagonalFOV;
            var width:uint = sensor_.width;
            var height:uint = sensor_.height;

            // Get the latest frame of pixel data from the sensor.
            var pixels:ByteArray = sensor_.pixels;
      }
}
```

### Code Sample 2: Usage of VisualSensorFromCamera

```
class MyVisualSensorFromDisplayObjectSubscriber
implements ISensorSubscriber
{
      var sensor_:VisualSensorFromCamera;

      public function MyVisualSensorFromDisplayObjectSubscriber(
            source:Camera)
      {
            // Create the sensor with the default values for the
            // FOV and resolution arguments.
            sensor_ = new VisualSensorFromCamera(source);
      }

      // The ISensorSubscriber implementation.
      public function onSensorDataUpdated(
            sensor:AbstractSensor):void
      {
            // Get the FOV and resolution values.
```

```
            var diagonalFOV:Number = sensor_.diagonalFOV;
            var width:uint = sensor_.width;
            var height:uint = sensor_.height;

            // Get the latest frame of pixel data from the sensor.
            var pixels:ByteArray = sensor_.pixels;
        }
}
```

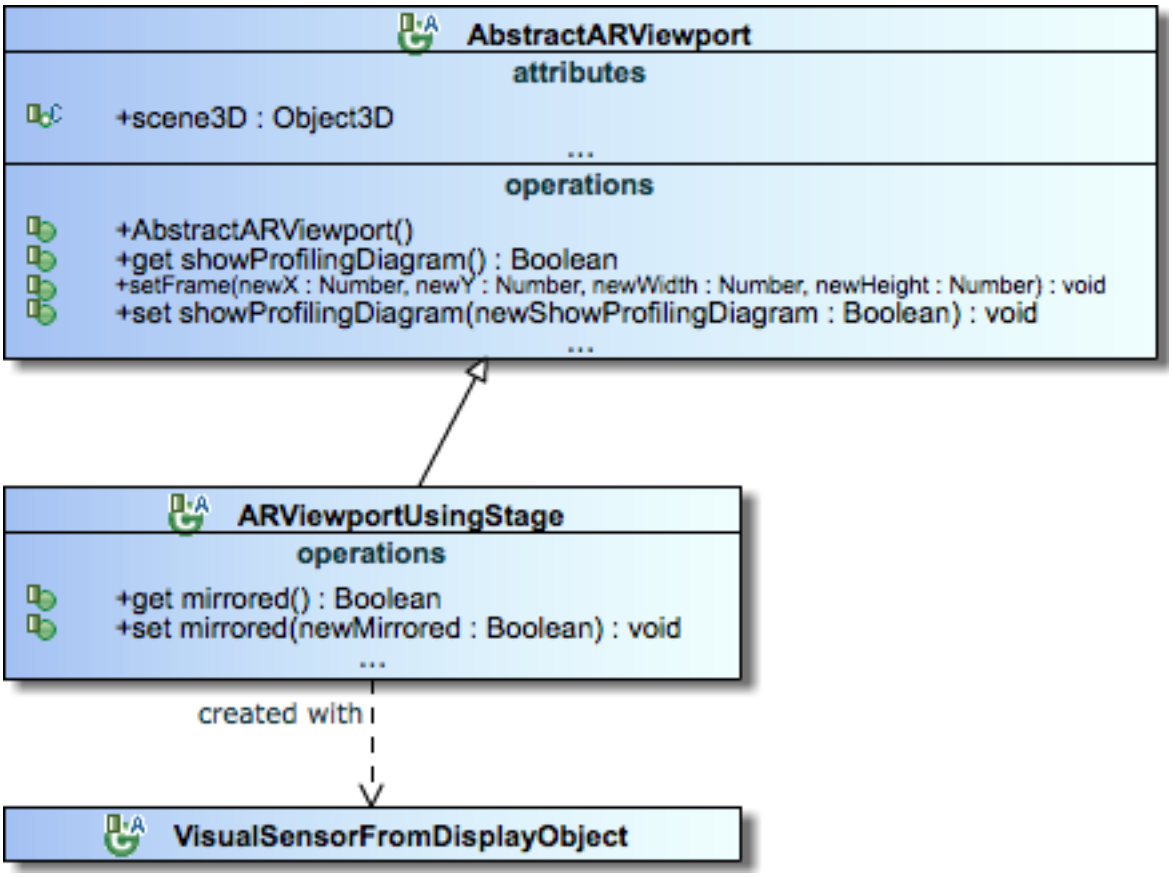
*4.1.2: Compositing 2D and 3D Scenes*

We have discussed the difficulties of overlaying 3D content atop 2D content in Flash, particularly where the 2D content is a camera feed. (See "2.6.4: Focus on Flash" and "3.1.2: Approaches and Outcomes".) To varying extents, the feasible solutions may promote efficiency by leveraging GPU acceleration but hurt efficiency by requiring data to be marshalled between main memory and graphics memory. One approach is to marshall the GPU-rendered 3D content into the 2D context. Illusion provides two implementations of this approach, with one implementation being general-purpose and the other being optimized for camera input.

Both implementations of 3D-to-2D marshalling are concrete subclasses of the AbstractARViewport class. AbstractARViewport extends Sprite (a type of node in Flash's standard 2D scene graph), so client code can add its instances to the Stage (alongside other standard 2D nodes). An AbstractARViewport is constructed using a Stage3D and an AbstractVisualSensor. Internally, the AbstractARViewport creates a 3D root node, adds it to the Stage3D, and continually manages the 3D scene's rendering to a bitmap that is the child of the AbstractARViewport. This bitmap's background is transparent. Beneath the bitmap lies a representation of the 2D scene.

For the general-purpose compositor, which is called ARViewportUsingStage, the representation of the 2D scene is a DisplayObject residing on Stage. This DisplayObject is obtained from a VisualSensorFromDisplayObject that is provided when instantiating the ARViewportUsingStage (Figure 11; Code Sample 3).

**Figure 11: Design of ARViewportUsingStage**



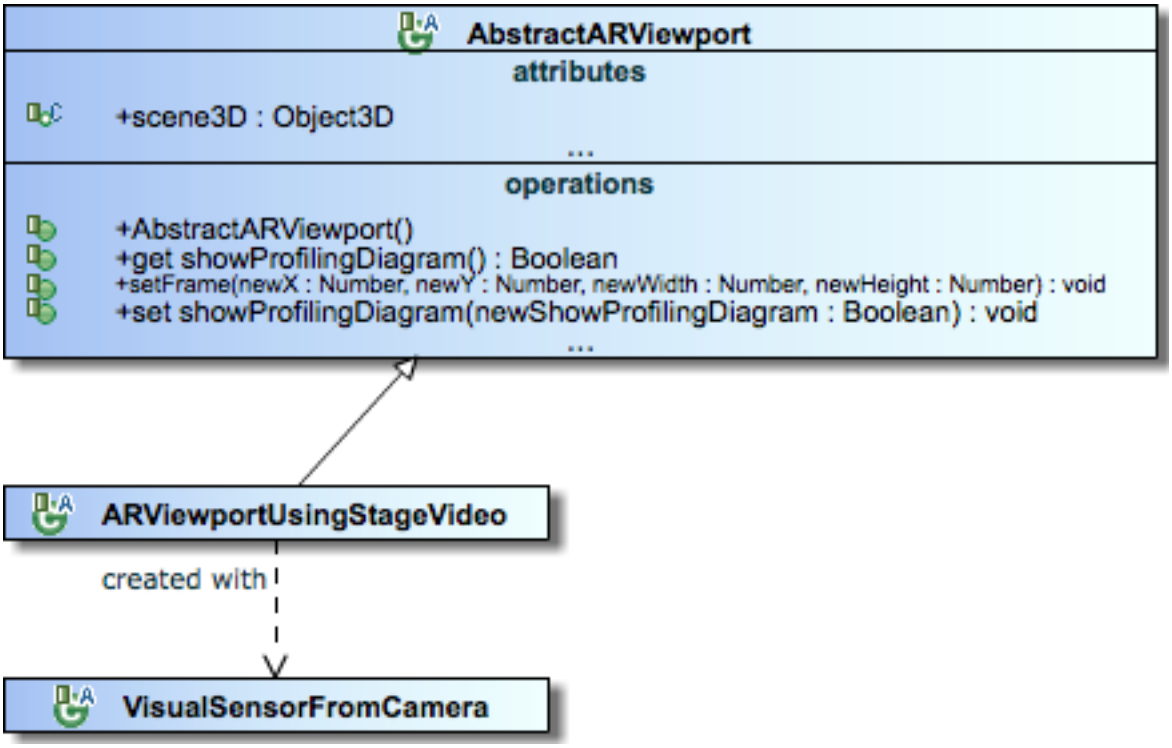
**Code Sample 3: Usage of ARViewportUsingStage**

```
var scene2D:DisplayObject;
var stage3D:Stage3D;
var sensor:VisualSensorFromDisplayObject;
…

// Create the AR viewport.
var arViewport:ARViewPortUsingStage =
      new ARViewportUsingStage(stage3D, sensor);

// Display profiling statistics such as FPS.
arViewport.showProfilingDiagram = true;

// Make the AR viewport mirrored (horizontally flipped).
arViewport.mirrored = true;

// Add the AR viewport to the 2D scene.
scene2D.addChild(arViewport);

// Get the 3D scene from the AR viewport.
var scene3D:Object3D = arViewport.scene3D;
```

For the camera-specific compositor, which is called ARViewportUsingStageVideo, the representation of the 2D scene is a StageVideo with an attached Camera. The Camera is obtained from a VisualSensorFromCamera that is provided when instantiating the ARViewportUsingStageVideo. The StageVideo is also provided at instantiation (Figure 12; Code Sample 4).

The 3D root node of an AbstractARViewport is exposed to client code, which is responsible for populating it with children. The 3D root node's type is Object3D—a building block of scene graphs in the Alternativa3D graphics engine. Object3D is analogous to Sprite: both are types of positional nodes that have event dispatching functionality.

**Figure 12: Design of ARViewportUsingStageVideo**



**Code Sample 4: Usage of ARViewportUsingStageVideo**

```
var scene2D:DisplayObject;
var stageVideo:StageVideo;
var stage3D:Stage3D;
var sensor:VisualSensorFromCamera;
…

// Create the AR viewport.
var arViewport:ARViewPortUsingStageVideo =
      new ARViewportUsingStageVideo(stageVideo, stage3D, sensor);

// Display profiling statistics such as FPS.
arViewport.showProfilingDiagram = true;

// Add the AR viewport to the 2D scene.
scene2D.addChild(arViewport);

// Get the 3D scene from the AR viewport.
var scene3D:Object3D = arViewport.scene3D;
```

Both implementations of AbstractARViewport are optimized to do nothing (i.e. just show the 2D scene) when the 3D scene contains no meshes. This optimization is important because compositing is expensive and, in AR, the 3D scene may be empty much of the time (when no physical markers are found).

At the design level, AbstractARViewport can be considered an adapter from AbstractVisualSensor to both Sprite and Object3D. Internally, it uses the observer pattern to coordinate with its adaptee.

Note that AbstractARViewport does not interface with Illusion's tracking system. Conversely, Illusion's tracking system does not interface with AbstractARViewport. Thus, 2D/3D compositing is decoupled from tracking. For example, clients are free to roll their own compositor for use with the tracking system, or to use the tracking system without rendering any underlying 2D content.

*4.1.3: Tracking Markers*

We have discussed a variety of trackers available in AS3 and other languages. (See "2.3.3: Frameworks".) None of these trackers provides a superset of the others' functionality, so clients might want to use multiple trackers in one project or at least across projects. However, interface differences are an obstacle. A naive approach—one that treats trackers' interfaces as non-generalizable—produces code that is difficult to maintain and perhaps inefficient due to duplicated processing and storage of sensor data.

Illusion provides a streamlined interface for wrapping (creating adapters for) existing and future trackers. This interface is called AbstractTracker. Currently, wrappers for flare*tracker and flare*nft are implemented. These wrappers are called FlareBarcodeTracker (Figure 13, Code Sample 5) and FlareNaturalFeatureTracker (Figure 14, Code Sample 6).

AbstractTracker and its implementations rely on a mediator pattern with delegation. The client-defined delegate must implement an interface called ITrackerDelegate, which provides callbacks for situations where the client probably wants to supply new virtual markers. The delegate can supply virtual markers to the AbstractTracker via mediatees of type MarkerPool. Specifically, MarkerPool exposes a Vector.<Object3D> whose elements are virtual markers that may represent a like number of tracked instances of a physical marker. The AbstractTracker has one MarkerPool per type of physical marker. For example, if an AbstractTracker can track three different images but only one instance of each image at a time, then it would have three MarkerPool objects, which client code should populate with one Object3D each.

AbstractTracker subclasses are instantiated with an ITrackerDelegate, an AbstractSensor (often, an AbstractVisualSensor representing the input images and their projection data), and an Object3D (representing the root node to which virtual markers will be attached). An optional boolean argument specifies whether the tracker should start automatically or wait for an instruction from client code. Automatic starting is the default.

Depending on the subclass, the constructor may take additional arguments. FlareBarcodeTracker must be instantiated with a configuration object of type FlareBarcodeFeatureSet, which specifies the quantity, patterns, and sizes of barcodes to be used. This configuration class serves to provide default values in a manner that is independent of constructor signatures.

FlareNaturalFeatureTracker reads its configuration from a file whose path may be provided as an optional constructor argument; otherwise, a default path is used. Both types of flare* wrappers need a license file for the underlying library. The license file's path may be provided as an optional constructor argument; otherwise, a default path is used.

Internally, the flare* wrappers' file loading is performed using the Loader class and ILoaderDelegate interface that are described in "B.1: Loading Binary or Text Files". The same approach to file loading is recommended for client implementations of AbstractTracker.

Internally, an AbstractTracker implementation should update the position and orientation of virtual markers based on the position and orientation of corresponding physical markers. Also, when physical markers are found or lost, corresponding virtual markers should be added to or removed from the root node, and each virtual marker should dispatch an event of type MarkerEvent, specifying its change in tracking

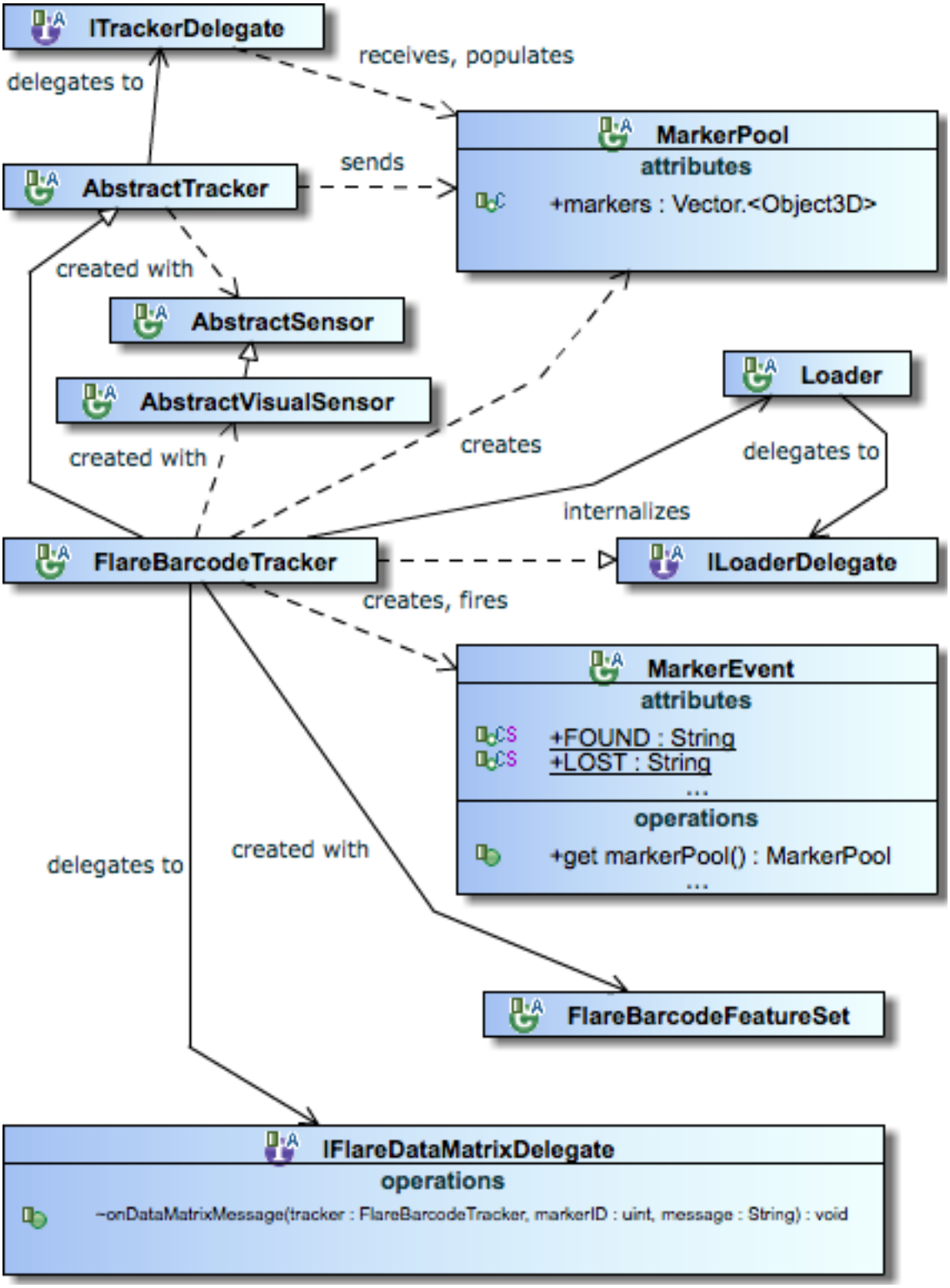
status. Client code may listen for a MarkerEvent on any marker, and may handle the event appropriately.

Optionally, the client-defined delegate for FlareBarcodeTracker may implement an interface called IFlareDataMatrixDelegate, which provides a callback for handling a barcode decoding event. The decoded message (ex. a URL) is provided to the callback as an argument. This callback is fired even if no virtual marker is provided.

Similarly, the client-defined delegate for FlareNaturalFeatureTracker may implement an optional interface called IFlareVirtualButtonDelegate, which provides a callback for handling occlusion events. An occlusion event is fired when a specified part of the physical marker is covered or uncovered. For example, such an event might happen when the user starts or stops touching the specified part of the physical marker. These events allow the specified part of the marker to be treated like a button.

Besides supporting its current implementations, AbstractTracker is designed to be extensible by future versions of Illusion and by client code. Illusion assumes only that the underlying tracker can take sensor data as input and give 3D spatial data as output. The sensor data need not be visual. A minimal wrapper requires just three method implementations: start, stop, and updateTrackedMarkers.

**Figure 13: Design of FlareBarcodeTracker and Related Types**

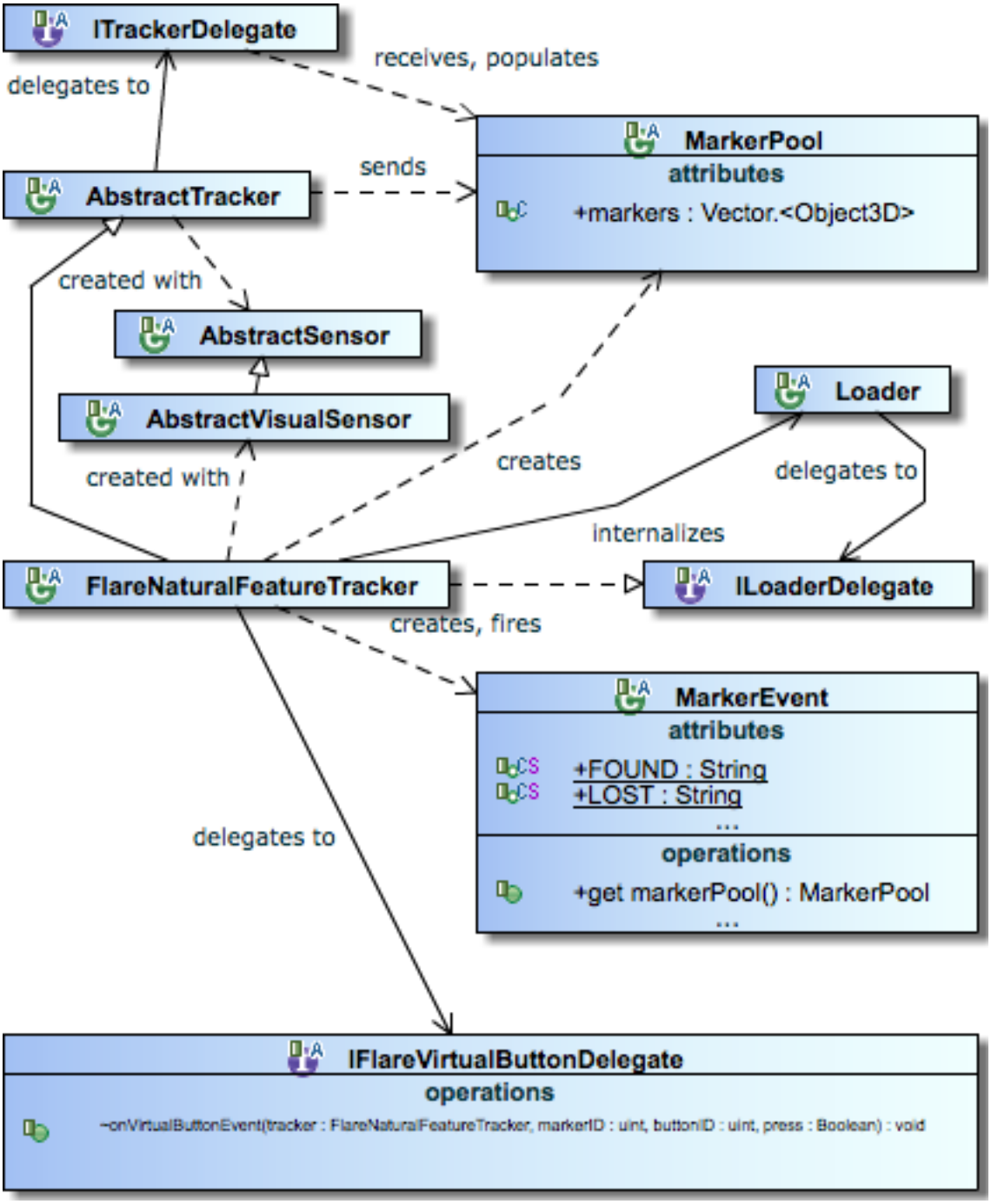


**Code Sample 5: Usage of FlareBarcodeTracker and Related Types**

```
class MyFlareBarcodeTrackerDelegate
implements ITrackerDelegate, IFlareDataMatrixDelegate
{
      var markers_:Vector.<Object3D>;
      var tracker_:FlareBarcodeTracker;

      public function MyFlareBarcodeTrackerDelegate(
            sensor:AbstractVisualSensor,
            scene3D:Object3D,
            markers:Vector.<Object3D>)
      {
            markers_ = markers;

            // Create the feature set.
            var featureSet:FlareBarcodeFeatureSet =
                  new FlareBarcodeFeatureSet();

            // Let there be one simple ID marker.
            flareBarcodeFeatureSet.numSimpleIDs = 1;

            // Let there be one BCH marker.
            flareBarcodeFeatureSet.numBCHs = 1;

            // Let there be one frame marker.
            flareBarcodeFeatureSet.numFrames = 1;

            // Let there be one data matrix marker.
            flareBarcodeFeatureSet.numDataMatrices = 1;

            // Create the barcode tracker with the default
            // file path for the license.
            tracker_ = new FlareBarcodeTracker(this,
                                               sensor,
                                               scene3D,
                                               featureSet);
      }

      // Part of the ITrackerDelegate implementation.
      public function onTrackerStarted(
            tracker:AbstractTracker,
            markerPools:Vector.<MarkerPool>):void
      {
            // Add one marker to each marker pool (as long as
            // enough markers were specified). Also add event
            // listeners to each marker.
            for(var i:uint = 0;
                i < markerPools.length && i < markers_.length;
                i++)
            {
                  var marker:Object3D = markers_[i];
```

```
                // Add the marker to the pool.
                markerPools[i].markers.push(marker);

                // Listen for the marker being found.
                marker.addEventListener(MarkerEvent.FOUND,
                                        onMarkerFound);

                // Listen for the marker being lost.
                marker.addEventListener(MarkerEvent.LOST,
                                        onMarkerLost);
            }
        }

        // Part of the ITrackerDelegate implementation.
        public function onMarkerPoolHasExcessDemand(
            tracker:AbstractTracker,
            markerPoolIndex:uint,
            markerPool:MarkerPool):void
        {
            // Do nothing. We do not add virtual markers
            // dynamically in this sample. Anyway,
            // FlareBarcodeTracker cannot detect multiple
            // instances of each physical marker.
        }

        // The IFlareDataMatrixDelegate implementation.
        public function onDataMatrixMessage(
            tracker:FlareBarcodeTracker,
            markerID:uint,
            message:String):void
        {
            // Do something with the decoded message.
            …
        }

        public function onMarkerFound(event:MarkerEvent):void
        {
            // Do something with the found marker or its
            // marker pool.
            var markerPool:MarkerPool = event.markerPool;
            …
        }

        public function onMarkerLost(event:MarkerEvent):void
        {
            // Do something with the lost marker or its
            // marker pool.
            var markerPool:MarkerPool = event.markerPool;
            …
        }
}
```

**Figure 14: Design of FlareNaturalFeatureTracker and Related Types**



**Code Sample 6: Usage of FlareNaturalFeatureTracker and Related Types**

```
class MyFlareNaturalFeatureTrackerDelegate
implements ITrackerDelegate, IFlareVirtualButtonDelegate
{
      var markers_:Vector.<Object3D>;
      var tracker_:FlareBarcodeTracker;

      public function MyFlareBarcodeTrackerDelegate(
            sensor:AbstractVisualSensor,
            scene3D:Object3D,
            markers:Vector.<Object3D>)
      {
            markers_ = markers;

            // Create the natural feature tracker with the
            // default file paths for the license and feature
            // set.
            tracker_ = new FlareBarcodeTracker(this,
                                               sensor,
                                               scene3D);
      }

      // Part of the ITrackerDelegate implementation.
      public function onTrackerStarted(
            tracker:AbstractTracker,
            markerPools:Vector.<MarkerPool>):void
      {
            // Add one marker to each marker pool (as long as
            // enough markers were specified). Also add event
            // listeners to each marker. Finally, add one
            // virtual button corresponding to each type of
            // physical marker.
            for(var i:uint = 0;
                i < markerPools.length && i < markers_.length;
                i++)
            {
                  // Add the marker to the marker pool.
                  markerPools[i].markers.push(markers_[i]);

                  // Listen for the marker being found.
                  marker.addEventListener(MarkerEvent.FOUND,
                                          onMarkerFound);

                  // Listen for the marker being lost.
                  marker.addEventListener(MarkerEvent.LOST,
                                          onMarkerLost);

                  // Add a virtual button with its upper-left
                  // corner at (128, 128) pixels and its
                  // lower-right corner at (384, 384) pixels, in
                  // terms of the marker type's source image.
```

```
                tracker_.addVirtualButton(i,
                                          128, 128, 384, 384);
            }
        }

        // Part of the ITrackerDelegate implementation.
        public function onMarkerPoolHasExcessDemand(
            tracker:AbstractTracker,
            markerPoolIndex:uint,
            markerPool:MarkerPool):void
        {
            // Do nothing. We do not add virtual markers
            // dynamically in this sample. Anyway,
            // FlareNaturalFeatureTracker cannot detect
            // multiple instances of each physical marker.
        }

        // The IFlareVirtualButtonDelegate implementation.
        public function onVirtualButtonEvent(
            tracker:FlareNaturalFeatureTracker,
            markerID:uint,
            buttonID:uint,
            press:Boolean):void
        {
            if (press)
            {
                // Handle the button press.
                …
            }
            else
            {
                // Handle the button release.
                …
            }
        }

        public function onMarkerFound(event:MarkerEvent):void
        {
            // Do something with the found marker or its
            // marker pool.
            var markerPool:MarkerPool = event.markerPool;
            …
        }

        public function onMarkerLost(event:MarkerEvent):void
        {
            // Do something with the lost marker or its
            // marker pool.
            var markerPool:MarkerPool = event.markerPool;
            …
        }
    }
```

### 4.2: Full Example Application

An example application called ApplesAndGoblets (Code Sample 7) exercises most of Illusion's functionality in unison. flare*tracker seeks 6 barcode markers (2 each from 3 families of barcode patterns) while flare*nft seeks 3 natural feature markers. Each nature feature marker has a "virtual button" (a separately tracked subregion) in its center.

Two high-polygon 3D models—an apple (13,470 triangles) and a goblet (3,324 triangles)—are loaded into the application.[4] (See "B.2: Loading 3D Model Files".) Each marker is given either a clone of the apple or a clone of the goblet as its virtual representation. For each natural feature marker, the corresponding apple or goblet clone starts/stops spinning when the user obscures the virtual button. A three-point lighting setup illuminates the scene. (See "B.3: Creating Lighting Setups").

The trackers analyze data from a live video. This live video is composited with the rendered apples and goblets. The code can be configured to use either the combination of VisualSensorFromCamera and ARViewportUsingStageVideo or the combination of VisualSensorFromDisplayObject and ARViewportUsingStage.

[4] The apple and goblet models are courtesy of Teinye Horsfall at WireCASE Ltd (http://www.wirecase.com) and Sven Dännart at Medievalworlds (http://www.medievalworlds.com), respectively.

### Code Sample 7: ApplesAndGoblets

```
[SWF(width='640', height='480', frameRate='60')]
public class ApplesAndGoblets extends Sprite implements
      IExternalModelPrefabLoaderDelegate,
      ITrackerDelegate,
      IFlareVirtualButtonDelegate
{
      private const ACCELERATE_WITH_STAGE_VIDEO:Boolean = true;

      private var stageFrameRate_:Number;
      private var stage3D_:Stage3D;
      private var sensor_:AbstractVisualSensor;
      private var arViewport_:AbstractARViewport;
      private var flareBarcodeTracker_:FlareBarcodeTracker;
      private var flareNFT_:FlareNaturalFeatureTracker;
      private var applePrefab_:ExternalModelPrefab;
      private var gobletPrefab_:ExternalModelPrefab;
      private var rotatingMarkers_:Vector.<Object3D> =
            new Vector.<Object3D>();
      private var lastMilliseconds_:int;


      public function ApplesAndGoblets()
      {
            if (stage)
            {
                  onAddedToStage();
            }
            else
            {
                  // Listen for being added to the 2D stage.
                  addEventListener(Event.ADDED_TO_STAGE,
                                   onAddedToStage);
            }
      }


      private function onAddedToStage(event:Event = null):void
      {
            if (hasEventListener(Event.ADDED_TO_STAGE))
            {
                  removeEventListener(Event.ADDED_TO_STAGE,
                                      onAddedToStage);
            }

            // Configure the 2D stage.
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            // Store the 2D stage's frame rate for use in
            // pausing/unpausing.
```

```
stageFrameRate_ = stage.frameRate;

// Create the video camera.
var videoCamera:Camera = Camera.getCamera();

// Get the 3D stage.
stage3D_ = stage.stage3Ds[0];

// Configure the video camera.
videoCamera.setMode(640, 480, 60); // 640x480 @ 60 FPS

if (ACCELERATE_WITH_STAGE_VIDEO &&
    stage.stageVideos.length > 0)
{
     var stageVideo:StageVideo =
          stage.stageVideos[0];

     // Create the visual sensor that draws data from
     // the camera.
     sensor_ =
          new VisualSensorFromCamera(videoCamera);

     // Create the AR viewport.
     arViewport_ = new ARViewportUsingStageVideo(
          stageVideo,
          stage3D_,
          sensor_ as VisualSensorFromCamera);
}
else
{
     // Create and configure the video.
     var video:Video = new Video(videoCamera.width,
                                 videoCamera.height);
     video.attachCamera(videoCamera);

     // Create the visual sensor that draws data from
     // the video.
     sensor_ =
          new VisualSensorFromDisplayObject(video);

     // Create and configure the AR viewport.
     arViewport_ = new ARViewportUsingStage(
          stage3D_,
          sensor_ as VisualSensorFromDisplayObject
     );
     (arViewport_ as ARViewportUsingStage).mirrored =
          true;
}

// Add the AR viewport to the 2D scene.
addChild(arViewport_);
```

```
        // Get the 3D scene from the AR viewport.
        var scene3D:Object3D = arViewport_.scene3D;

        // Add lights to the 3D scene.
        scene3D.addChild(SceneUtils.newThreePointLighting());

        // Listen for and request the 3D stage's graphics
        // context.
        stage3D_.addEventListener(Event.CONTEXT3D_CREATE,
                                  onContextCreate);
        stage3D_.requestContext3D();

        // Listen for keystrokes.
        stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
    }

    private function onContextCreate(event:Event):void
    {
        stage3D_.removeEventListener(Event.CONTEXT3D_CREATE,
                                     onContextCreate);

        // Create the model loader.
        var loader:ExternalModelPrefabLoader =
              new ExternalModelPrefabLoader(this, "data");

        // Load the models.
        loader.loadExternalModelPrefabs("apple.3ds",
                                        "goblet.3ds");

        // Listen for frame updates.
        addEventListener(Event.ENTER_FRAME, onEnterFrame);

        // Initialize the time.
        lastMilliseconds_ = getTimer();
    }


    public function onLoadExternalModelPrefabError(
        loader:ExternalModelPrefabLoader,
        filename:String,
        errorEventType:String):void
    {
        throw new Error("Failed to load \"" +
                        loader.basePath + filename + "\": " +
                        errorEventType);
    }

    public function onLoadExternalModelPrefabComplete(
        loader:ExternalModelPrefabLoader,
        filename:String,
        externalModelPrefab:ExternalModelPrefab):void
    {
```

```
if (filename == "apple.3ds")
{
     // Store the apple model.
     applePrefab_ = externalModelPrefab;

     // Set the apple model's scale.
     applePrefab_.scale = 2.5;

     // Load the resources for the apple model.
     applePrefab_.loadResources(stage3D_.context3D);
}
else // filename == "goblet.3ds"
{
     // Store the goblet model.
     gobletPrefab_ = externalModelPrefab;

     // Set the goblet model's scale.
     gobletPrefab_.scale = 1000;

     // Load the resources for the goblet model.
     gobletPrefab_.loadResources(stage3D_.context3D);
}

if (loader.numLoadsPending > 0)
{
     // The other model is still loading.

     // Wait for the other model to load.
     return;
}

// Both models have loaded.

// Create and configure the barcode tracker's feature
// set.
var flareBarcodeFeatureSet:FlareBarcodeFeatureSet =
     new FlareBarcodeFeatureSet();
flareBarcodeFeatureSet.numSimpleIDs = 2;
flareBarcodeFeatureSet.numBCHs = 2;
flareBarcodeFeatureSet.numFrames = 2;

// Create the barcode tracker.
flareBarcodeTracker_ =
     new FlareBarcodeTracker(this,
                             sensor_,
                             stage,
                             arViewport_.scene3D,
                             flareBarcodeFeatureSet);

// Create the natural feature tracker.
flareNFT_ =
     new FlareNaturalFeatureTracker(
```

```
                this,
                sensor_,
                stage,
                arViewport_.scene3D);
    }

    public function onTrackerStarted(
          tracker:AbstractTracker,
          markerPools:Vector.<MarkerPool>):void
    {
          // Add either an apple or a goblet to each marker
          // pool.
          for (var i:uint = 0; i < markerPools.length; i++)
          {
                var marker:Object3D;
                if (i % 2 == 0)
                {
                      marker = applePrefab_.newObject3D();
                }
                else
                {
                      marker = gobletPrefab_.newObject3D();
                }
                marker.addEventListener(MarkerEvent.LOST,
                                        onMarkerLost);
                markerPools[i].markers.push(marker);
          }

          if (tracker == flareNFT_)
          {
                // Set up a 48x48 pixel virtual button in the
                // center of each physical marker.

                // The Austria physical marker is 480x256.
                flareNFT_.addVirtualButton(
                      0, 216, 104, 264, 152);

                // The Vienna physical marker is 480x288.
                flareNFT_.addVirtualButton(
                      1, 216, 120, 264, 168);

                // The Graz physical marker is 336x480.
                flareNFT_.addVirtualButton(
                      2, 144, 216, 192, 264);
          }
    }

    public function onMarkerPoolHasExcessDemand(
          tracker:AbstractTracker,
          markerPoolIndex:uint,
          markerPool:MarkerPool):void
    {
```

```
        // Do nothing, such that the supply of markers is
        // inelastic.
    }

    public function onVirtualButtonEvent(
        tracker:FlareNaturalFeatureTracker,
        markerID:uint,
        buttonID:uint,
        press:Boolean):void
    {
        if (!press)
        {
            // The virtual button was released.

            // Do nothing.
            return;
        }

        // The virtual button was pressed.

        // Get the marker corresponding to the virtual button.
        var marker:Object3D =
            tracker.markerPools[markerID].markers[0];

        var i:int = rotatingMarkers_.indexOf(marker);
        if (i == -1)
        {
            // The marker was not rotating.

            // Start rotating the marker.
            rotatingMarkers_.push(marker);
        }
        else
        {
            // The marker was rotating.

            // Stop rotating the marker.
            rotatingMarkers_.splice(i, 1);
        }
    }


    private function onMarkerLost(event:MarkerEvent):void
    {
        var marker:Object3D = event.target as Object3D;

        var i:int = rotatingMarkers_.indexOf(marker);
        if (i != -1)
        {
            // The marker was rotating.

            // Stop rotating the marker.
```

```
                rotatingMarkers_.splice(i, 1);
            }
        }

        private function onEnterFrame(event:Event):void
        {
            // Update the time.
            var milliseconds:int = getTimer();
            var deltaMilliseconds:int =
                milliseconds - lastMilliseconds_;
            lastMilliseconds_ = milliseconds;

            for each (var marker:Object3D in rotatingMarkers_)
            {
                // Rotate the marker at 45 degrees per second.
                marker.getChildAt(0).rotationZ +=
                    deltaMilliseconds * 0.00025 * Math.PI;
            }
        }

        private function onKeyUp(event:KeyboardEvent):void
        {
            if (event.keyCode == 32) // spacebar
            {
                // Show or hide the Alternativa3D profiling
                // diagram.
                arViewport_.showProfilingDiagram =
                    !arViewport_.showProfilingDiagram;
            }
            else if (event.keyCode == 80) // 'p'
            {
                // Pause/unpause the 2D stage.
                if (stage.frameRate == stageFrameRate_)
                {
                    stage.frameRate = 0.0001;
                }
                else
                {
                    stage.frameRate = stageFrameRate_;
                }
            }
        }
    }
```

### 4.3: Comparison to Other Designs

Illusion's design is most closely paralleled in FLARManager (Socolofsky, "FLARManager: Augmented Reality in Flash"). FLARManager is an attempt at providing universal glue between tracking libraries and non-GPU-accelerated 3D graphics engines for Flash 10. The supported trackers are FLARToolkit, flare*tracker, and flare*nft. The supported graphics engine is Papervision (plus other options for FLARToolkit only). FLARManager internally creates one camera feed per tracker.

Compared to FLARManager, Illusion has the advantages of being more up-to-date in its choice of 3D graphics engine and more general in its design of trackers' data sources or sensors. Sensors in Illusion need not be based on camera feeds and may be shared among multiple trackers. FLARManager is more general in its design of 3D scenes, since there is a bridge layer between a tracker and a graphics engine. However, in practice, this bridge layer might make FLARManager harder to extend and maintain, as it introduces a degree of interaction among extensions. (The existence of a bridge from tracker "x" to graphics engine "y" may make a client expect that other supported trackers also have a bridge to "y".)

Goblin XNA (Oda et al, 2012), which targets Windows and Windows Phone, is another glue layer that can wrap trackers from multiple vendors. Goblin XNA implements an original game engine atop existing graphics and physics libraries. The game scene may have camera components attached to it and may contain markers as children. At a given time, only one camera may be active for tracking purposes and only one pose may be tracked per physical marker. The ALVAR tracker, for square markers, is

supported on Windows. No trackers are supported on Windows Phone. Certain HMDs, which can sense the wearer's head pose, are supported on Windows.

Compared to Goblin XNA, Illusion is again more general in its design of sensors. Illusion's sensors need not be based on camera input and need not be shared among all active trackers. Also, Illusion currently supports a natural feature tracker, while Goblin XNA does not. However, Goblin XNA has a broader range of game engine functionality and supports HMDs.

KHARMA (Augmented Environments Laboratory, "KHARMA"), the specification of the Argon mobile AR browser, is vendor-neutral insofar as it is offered as an open standard. However, this standard specifies a (limited) set of supported tracking functionality, so it is not broadly inclusive of existing and future trackers from third parties. KHARMA does offer a markup-based 3D scene graph in which any node may be anchored to a geolocation or a visually tracked square marker. An iPhone-like set of sensors is assumed.

Argon and KHARMA, being a dedicated AR platform and its specification, are not directly comparable to Illusion. Illusion is "just" a toolkit targeting a multipurpose Web platform and is designed for portability to other multipurpose platforms and for extensibility by any tracker developer. On the other hand, Argon and KHARMA integrate markers into a high-level markup language and support geolocation-based tracking.

While there are only a few attempts to provide a third-party-extensible glue layer between tracking libraries and graphics engines, there are numerous vendor-specific solutions for integrating tracking with a

graphics engine. For example, flare*tracker and flare*nft come with samples of Papervision integration. D'Fusion comes with a dedicated graphics engine for some platforms, as well as a sample of Away3D integration for Flash. Vuforia comes with a wrapper for Unity integration and a sample of raw OpenGL ES integration.

The typical vendor-specific offering is "just" an application framework, readymade for a fixed set of trackers. By contrast, Illusion is a toolkit that the client may extend for an arbitrary set of sensors, trackers, and compositing techniques. As seen in the "Evaluation" chapter, Illusion's generality and extensibility come at negligible cost in processing time. Moreover, due to its generality and extensibility, Illusion has the potential to reduce duplication of development effort (i.e. increase code reuse) in an organization where multiple sensors, trackers, or compositing techniques are used. For example, it can help an organization experiment with diverse approaches to ubiquitous computing. Conversely, some organizations might need to work with multiple game engines but only one tracking technology. Then, vendor-specific glue layers might reduce duplication of development effort, while Illusion would not.

From experience at Ad-Dispatch ("3.1: At Ad-Dispatch"), we know that the time cost of experimenting with multiple trackers, in the absence of a general and extensible framework, is indeed a practical problem. This problem arose even when targeting just one type of sensor (webcams) on conventional personal computers. Illusion's design addresses a superset of the fragmentation problem that was present in the author's frameworks at Ad-Dispatch.

# Chapter 5: Evaluation

We seek to evaluate the efficiency of Illusion. This concept is distinct from the efficiency of any particular tracking libraries that Illusion wraps, or of any particular I/O systems that underlie its sensors and compositors. That is to say, we are primarily concerned with the cost that Illusion adds atop the tracking algorithm and platform-specific function calls. If this overhead cost is low in the Flash implementation, then in general it should be low in ports, too. To lesser extents, we are concerned with assessing optimization opportunities in Illusion's use of Flash-specific function calls, and with subjectively describing the performance of an interactive demo.

To address these evaluation goals, this chapter poses questions about Illusion's performance in various use cases. Methods for capturing performance data are discussed. Based on these methods, measurements and impressions are presented. From these observations, we are able to conclude that Illusion's overhead cost is negligible even when there are hundreds of simultaneously active trackers or marker pools. However, the compositing implementation for Flash is expensive on some hardware.

## 5.1: Questions

For any given application, quantitive measures of efficiency include CPU usage,[5] frame rate, and latency. Frame rate and latency relate to subjective qualities such as the AR scene's smoothness (continuity of motion) and its responsiveness (timeliness of motion, i.e. synchronization with real-world events). (See "2.5: Efficiency" for a broader discussion of

[5] GPU usage is of course another measure of efficiency. However, Illusion does not directly use the GPU; rather, it integrates with Alternativa3D and Flash functionality that may be GPU-accelerated.

efficiency's meaning and its determining factors in the context of AR.) To capture these quantitative and qualitative characteristics over a variety of application configurations, our evaluation includes four sets of questions:

1. How many whole frame-lengths of latency does Illusion add in an application that uses tracking with VisualSensorFromDisplayObject and ARViewportUsingStage?[6] i.e. By how many frames does the tracker's output scene lag behind the input? An answer of "0" ("no frame lag") implies that the 3D rendering for the current frame can use tracking results based on the 2D rendering for the current frame (if the underlying tracker permits). In this case, total latency would depend on frame rate and on components other than Illusion.
2. How much CPU time per frame does Illusion add? i.e. How much time is spent in functions belonging to Illusion's namespace? This measure excludes the time spent in underlying tracker functions and underlying Alternativa3D and Flash functions. How does this measure of Illusion's cost vary with respect to tracking resolution, number of trackers, and number of markers?
3. How much CPU time per frame is spent in underlying Alternativa3D and Flash functions whose use is peculiar to Illusion? (i.e. An alternative implementation of an AR framework for Flash

[6] An application using VisualSensorFromCamera and ARViewportUsingStageVideo effectively has two frame rates: one for the video background and another for everything else. We have no tools to measure the former, and the difference between the two appears to be unstable. Thus, the effect of Illusion on latency is not easily isolated in this case.

would not necessarily use these functions.) How does this cost vary across hardware configurations?

4. Given some specification of an AR scene with "good" graphics and trackers, what frame rate is achievable and what is the author's subjective impression of the smoothness and responsiveness of this scene? How do the frame rate and impression vary across systems? How do they vary between an implementation that uses StageVideo (i.e. VisualSensorFromCamera and ARViewportUsingStageVideo) and one that does not (i.e. VisualSensorFromDisplayObject and ARViewportUsingStage)?

The first three sets of questions are explored in controlled conditions. A minimal test application, called MinimalProfiler, uses an image stream and tracking algorithm that are contrived for repeatability, low time cost, and low latency, rather than realism. (See the next section, "5.2: Methodology", for details on MinimalProfiler and its dependencies.) These controlled conditions make it easier to assess possible causes of variation in the quantitative data on Illusion's efficiency.

To address the fourth set of questions, we provide observations on the performance of our full-featured demo application, ApplesAndGoblets. (See "4.2: Full Example Application".) This exploration is less controlled but more realistic, as it exercises industry-grade trackers, complex artistic content, camera input, and human interaction. These realistic conditions give a sense of context (a likeness to the author's previous experience in using AR applications), such that it is easier to form subjective impressions about the application's performance.

## 5.2: Methodology

### *5.2.1: Dependencies and Platforms*

To measure CPU time costs (hereafter, "time costs"), we rely on third-party solutions. A tool called The Miner (Sociodox, "The Miner") is integrated into MinimalProfiler. The Miner provides detailed logs and statistics pertaining to time costs, which can be broken down by event type or function, per frame. At least some of the costs of running The Miner are measurable, and are excluded from our reporting.

The Miner requires the Flash runtime's debug version, which might run less efficiently than the release version. For comparison, we evaluate the performance of ApplesAndGoblets using both the debug runtime and the release runtime. This comparison is conducted using a basic measurement tool that is included in Alternativa3D. This tool does not require the debug runtime but does not provide breakdowns by event type, or function. It simply measures the time elapsed between frames. This value may exceed the time cost of the application per se, as the application may be throttled by I/O bottlenecks or may have to yield the processor to other processes.

Below the level of the Flash runtime, factors affecting performance also include the user's operating system and hardware. As discussed in "2.5.2: Measurement", a "typical" user might be running Windows 7 on a machine with an Intel Core 2 CPU, 2 GB RAM, Intel GMA 950 GPU, 64 MB VRAM, and 1366x768 desktop resolution (Unity, "Web Player Hardware Statistics - 2012 Q2"). The closest match available to us as a test machine is a Dell Inspiron 9400. We also have a MacBook Pro 13" mid-2010 and a custom gaming desktop (Table 7). Measurements are

taken on each machine. A 640x480 Flash canvas (and 640x480 camera feed) is used because this resolution seems to be most practical for embedding in web pages on typical desktop resolutions. An external webcam—a Microsoft LifeCam VX-2000—is used with all machines. Flash 11.4 and Firefox 15 (the latest stable versions) are used.[7]

**Table 7: Test Machines**

| Description | CPU; main memory | GPU; graphics memory | Operating system |
|---|---|---|---|
| Dell Inspiron 9400 | 2 GHz Intel Core 2; 2 GB DDR2 | ATI Mobility Radeon X1400; 128 MB dedicated DDR3 plus 128 MB shared DDR2 | Windows 7 32-bit |
| MacBook Pro 13" mid-2010 | 2.4 GHz Intel Core 2 Duo; 8 GB DDR3 | NVIDIA GeForce 320M; 256 MB shared DDR3 | Mac OS X Lion |
| Custom gaming desktop | 3.41 GHz AMD Phenom II X4 965; 4 GB DDR3 | ATI Radeon HD 5870; 1 GB dedicated DDR5 | Windows 7 64-bit |

[7] During development, preliminary tests were conducted with Flash 11.3 and Flash 11.4 Beta. The stable version of Flash 11.4 seems to yield worse performance, especially on the Dell Inspiron 9400.

*5.2.2: Application and Parameters*

MinimalProfiler (Code Sample 8), our contrived test application, has a predictable source of image data, a simple (non-realistic) tracking algorithm, a configurable number of trackers and marker pools, and an option to pause when a known frame number is reached. For any frame, the output includes a list of timing statistics and a composited scene based on the latest image data and tracking results. By pausing and inspecting the composited scene, we determine whether (and how much) the tracking results lag behind the image source (thus answering our first evaluation question). By configuring the number of trackers and marker pools, and then inspecting the timing statistics, we determine how CPU time costs scale with respect to the configured variables (thus answering our second and third evaluation questions). Details of these methods are given below.

MinimalProfiler uses VisualSensorFromDisplayObject and ARViewportUsingStage. It requires a tracker with the following characteristics:

1. The tracking algorithm is transparent. i.e. For any given input image, the virtual markers' correct transformations are known. This characteristic enables us to determine which frame of the input stream corresponds to the current frame of tracking results. Thus, we can measure the latency as a whole number of frame-lengths.
2. An arbitrary number of tracker instances can run at once. This characteristic enables us to scale the application.
3. Each tracker instance can have an arbitrary number of marker pools. This characteristic, too, enables us to scale the application.

We fulfill these characteristics in a contrived tracker called DebugTracker (Code Sample 9). DebugTracker creates one marker pool per pixel of input resolution and it transforms one marker from each non-empty pool. The tracking algorithm reinterprets a pixel's RGB color coordinates as a marker's spatial coordinates: red is x, green is y, and blue is z, with a color range of [0, 255] being mapped onto a spatial range of [-128, 127]. For example, the color red—RGB triplet (255, 0, 0)—is interpreted as the position (127, 0, 0).

**Code Sample 8: MinimalProfiler**

```
[SWF(width='640', height='480', frameRate='60')]
public class MinimalProfiler extends Sprite
                              implements ITrackerDelegate
{
      private const UPDATE_EVENT:String = Event.ENTER_FRAME;
      private const START_PAUSED:Boolean = true;
      private const NUM_TRACKERS:uint = 1;
      private const NUM_MARKER_POOLS_PER_TRACKER:uint = 1;


      private var stageFrameRate_:Number;
      private var background_:Shape;
      private var stage3D_:Stage3D;
      private var sensor_:VisualSensorFromDisplayObject;
      private var arViewport_:ARViewportUsingStage;
      private var trackers_:Vector.<DebugTracker> =
            new Vector.<DebugTracker>();
      private var lastMilliseconds_:int;


      public function MinimalProfiler()
      {
            if (stage)
            {
                  onAddedToStage();
            }
            else
            {
                  // Listen for being added to the 2D stage.
                  addEventListener(Event.ADDED_TO_STAGE,
                                   onAddedToStage);
            }
```

```
}


private function onAddedToStage(event:Event = null):void
{
      if (hasEventListener(Event.ADDED_TO_STAGE))
      {
            removeEventListener(Event.ADDED_TO_STAGE,
                                onAddedToStage);
      }

      // Configure the 2D stage.
      stage.align = StageAlign.TOP_LEFT;
      stage.scaleMode = StageScaleMode.NO_SCALE;

      // Store the 2D stage's frame rate for use in
      // pausing/unpausing.
      stageFrameRate_ = stage.frameRate;

      // Get the 3D stage.
      stage3D_ = stage.stage3Ds[0];

      // Integrate TheMiner profiling tools.
      addChild(new TheMiner());

      // Create the background and fill it with a color that
      // the debug tracker will interpret as xyz
      // coordinates: (0, 64, 127).
      background_ = new Shape();
      background_.graphics.beginFill(0x80c0ff);
      background_.graphics.drawRect(0,
                                    0,
                                    stage.stageWidth,
                                    stage.stageHeight);
      background_.graphics.endFill();

      // Create the visual sensor that draws data from the
      // shape.
      sensor_ = new VisualSensorFromDisplayObject(
            background_, /* source */
            1.2566370614359172, /* fov: 72 degrees */
            NUM_MARKER_POOLS_PER_TRACKER, /* width */
            1 /* height */);

      // Create the AR viewport.
      arViewport_ = new ARViewportUsingStage(stage3D_,
                                             sensor_);

      // Add the AR viewport to the 2D scene.
      addChild(arViewport_);

      // Listen for and request the 3D stage's graphics
```

```
      // context.
      stage3D_.addEventListener(Event.CONTEXT3D_CREATE,
                                onContextCreate);
      stage3D_.requestContext3D();

      // Listen for keystrokes.
      stage.addEventListener(KeyboardEvent.KEY_UP, onKeyUp);
}

private function onContextCreate(event:Event):void
{
      stage3D_.removeEventListener(Event.CONTEXT3D_CREATE,
                                   onContextCreate);

      // Create the tracker.
      for (var i:uint = 0; i < NUM_TRACKERS; i++)
      {
            trackers_.push(new DebugTracker(
                  this,
                  sensor_,
                  stage,
                  arViewport_.scene3D));
      }

      // Listen for frame updates.
      addEventListener(UPDATE_EVENT, raiseMarker);

      // Initialize the time.
      lastMilliseconds_ = getTimer();
}


public function onTrackerStarted(
      tracker:AbstractTracker,
      markerPools:Vector.<MarkerPool>):void
{
      // Create the marker.
      var marker:GeoSphere = new GeoSphere(
            25, /* radius */
            2, /* segments */
            false, /* reverse */
            new FillMaterial(0xffff80) /* material */);

      // Upload the marker's resources to the 3D context.
      for each (var resource:Resource
                in marker.getResources())
      {
            resource.upload(stage3D_.context3D);
      }

      markerPools[0].markers.push(marker);
}
```

```
public function onMarkerPoolHasExcessDemand(
      tracker:AbstractTracker,
      markerPoolIndex:uint,
      markerPool:MarkerPool):void
{
      // Do nothing, such that the supply of markers is
      // inelastic.
}


private function raiseMarker(event:Event):void
{
      removeEventListener(Event.ENTER_FRAME, raiseMarker);

      // Update the time.
      var milliseconds:int = getTimer();
      var deltaMilliseconds:int =
            milliseconds - lastMilliseconds_;
      lastMilliseconds_ = milliseconds;

      // Fill the background with a color that the debug
      // tracker will interpret as xyz coordinates:
      // (0, 0, 127).
      background_.graphics.beginFill(0x8080ff);
      background_.graphics.drawRect(0,
                                    0,
                                    stage.stageWidth,
                                    stage.stageHeight);
      background_.graphics.endFill();

      if (START_PAUSED)
      {
            // Pause the 2D stage.
            stage.frameRate = 0.0001;
      }
}

private function onKeyUp(event:KeyboardEvent):void
{
      if (event.keyCode == 32) // spacebar
      {
            // Show or hide the Alternativa3D profiling
            // diagram.
            arViewport_.showProfilingDiagram =
                  !arViewport_.showProfilingDiagram;
      }
      else if (event.keyCode == 80) // 'p'
      {
            // Unpause/repause the 2D stage.
            if (stage.frameRate == stageFrameRate_)
            {
```

```
                        stage.frameRate = 0.0001;
                  }
                  else
                  {
                        stage.frameRate = stageFrameRate_;
                  }
            }
      }
}
```

### Code Sample 9: DebugTracker

```
public class DebugTracker extends AbstractTracker
{
      public function DebugTracker(delegate:ITrackerDelegate,
                                   sensor:AbstractVisualSensor,
                                   stage:Stage,
                                   scene3D:Object3D,
                                   autoStart:Boolean=true)
      {
            super(delegate, sensor, stage, scene3D, autoStart);
      }


      override public function start():void
      {
            super.start();

            var visualSensor:AbstractVisualSensor =
                  sensor_ as AbstractVisualSensor;
            var numPixels:uint =
                  visualSensor.width * visualSensor.height;

            // Unfix the number of marker pools.
            markerPools.fixed = false;

            // Create the marker pools.
            for (var i:uint = 0; i < numPixels; i++)
            {
                  markerPools.push(new MarkerPool());
            }

            // Fix the number of marker pools.
            markerPools.fixed = true;

            // Notify the delegate that the marker pools have been
            // created and tracking has started.
            delegate_.onTrackerStarted(this, markerPools);
```

```
}

override public function stop():void
{
      super.stop();

      // Release the marker pools.
      markerPools.fixed = false;
      markerPools.splice(0, markerPools.length);
      markerPools.fixed = true;
}


override protected function updateTrackedMarkers(
      markerPoolIterators:Vector.<MarkerPoolIterator>,
      pixels:ByteArray):void
{
      for (var i:uint = 0; i < markerPools.length; i++)
      {
            // Skip the alpha value.
            pixels.readUnsignedByte();

            // Interpret the RGB values as xyz coordinates.
            var x:int = pixels.readUnsignedByte() - 128;
            var y:int = pixels.readUnsignedByte() - 128;
            var z:int = pixels.readUnsignedByte() - 128;

            var marker:Object3D =
                  markerPoolIterators[i].nextMarker();

            if (!marker)
            {
                  continue;
            }

            marker.x = x;
            marker.y = y;
            marker.z = z;

            if (marker.parent != scene3D_)
            {
                  // The marker is newly found.

                  // Add the marker to the 3D scene.
                  scene3D_.addChild(marker);

                  // Notify the marker that it has been
                  // found.
                  marker.dispatchEvent(new MarkerEvent(
                        MarkerEvent.FOUND,
                        markerPoolIterators[i].markerPool));
            }
```

```
                }
        }
}
```

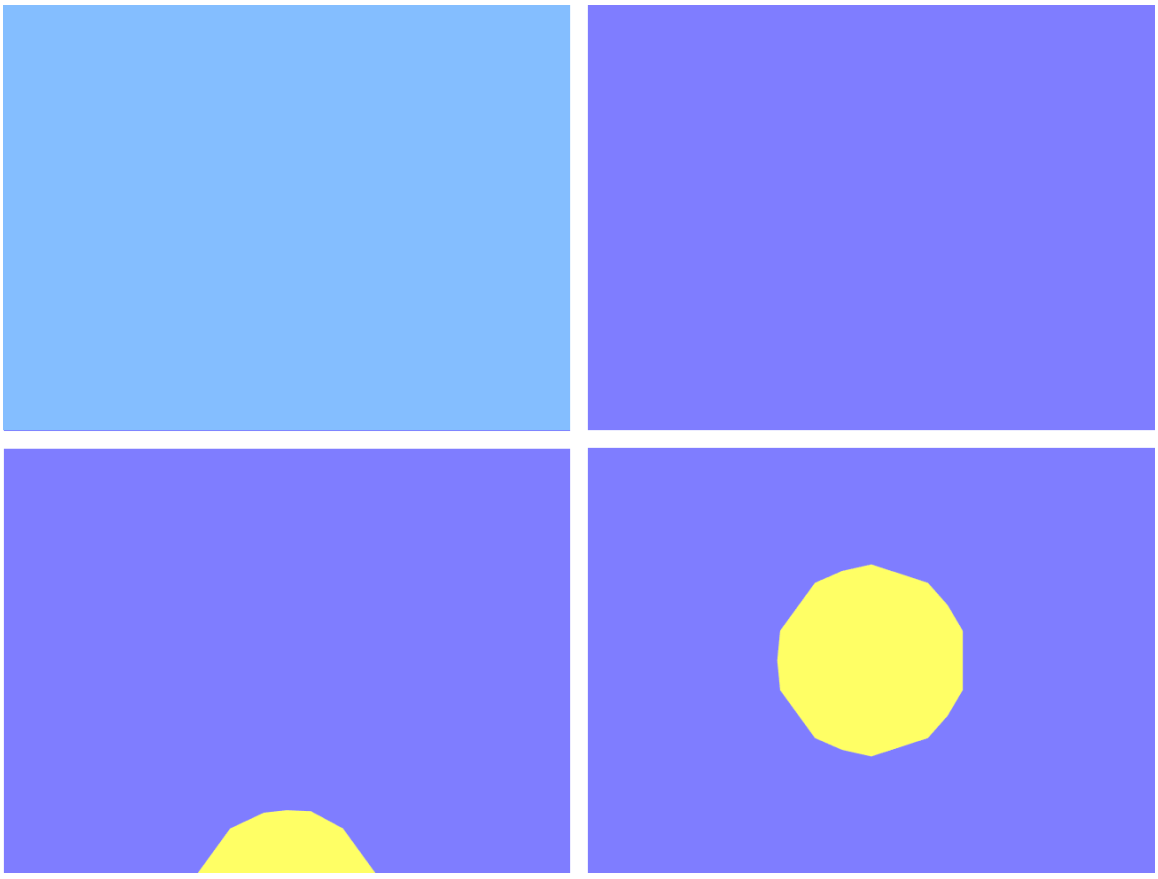
When feeding DebugTracker a predetermined stream of pixel data, we can correctly predict the virtual markers' positions as a function of time and latency, both measured in whole frames. When time is a given value and the stream of images contains no duplicates, this function is invertible such that we solve for latency. i.e. By inspecting the tracking results, we know what the pixels of the input image were. Then, we can search for this input image and find that it is a certain number of frames old (ex. 0 for the current frame of input or 1 for the previous frame). This number of frames is the latency.

Our search for the latency value is bounded. Based on *a priori* knowledge about Illusion and Flash, we can infer that Illusion's latency is at most 1 frame. The rationale is as follows. Illusion's trackers are updated every frame in an event-driven manner. So are the currently supported image sources (DisplayObject and Camera). Event dispatching in Flash is a single-threaded loop and Illusion's event handlers are likewise single-threaded. Therefore, a given frame's image source update must finish before the next frame's tracker update begins.

The up-to-dateness of a DebugTracker result can be observed visually in a screenshot or paused frame. Our experiment uses a contrived image source: a rectangle whose color changes sharply from one frame to the next. The experimenter can tell whether the position of a rendered 3D marker and the color of the rendered background match, according to DebugTracker's known algorithm. If they do match, the tracker has evaluated up-to-date input and thus the lag is 0 frames ("no frame lag").

If they do not match, the tracker has evaluated outdated input and thus the lag is 1 frame (Figure 15).

**Figure 15: A Test for Frame Lag or its Absence**



The contrived image source transitions from a cyan rectangle (upper left image) to a blue rectangle (upper right image). According to the contrived tracker, the cyan color corresponds to a lower point in space (a lesser y coordinate) than the blue color. If there is a lag of one frame, the tracker evaluates the cyan rectangle in the frame when the blue rectangle is being rendered; thus, the blue rectangle is rendered at the same time as a low 3D marker (lower left image). If there is no frame lag, the tracker evaluates the blue rectangle in the frame when the blue rectangle is rendered; thus, the blue rectangle is rendered at the same time as a high 3D marker (lower right image).

The presence or absence of frame lag is necessarily affected by the client code's approach to updating the 2D scene. Where in the Flash event cycle is the client code's update registered: near the start of the frame; near the end of the frame; with a high priority; with a low priority? How tardy can the client code's update be before a frame lag is introduced, i.e. before the tracker misses the opportunity to evaluate the updated data in the same frame? To find the answer, we vary our implementation of the test, incrementally delaying the 2D scene update's position in the Flash event cycle until a frame lag is introduced.

To evaluate scalability, we use up to 1000 marker pools and 1000 trackers. We consider 1000 marker pools to be an extreme case—at least for Web-based AR—due to the physical and computational complexity of seeking so many different markers simultaneously.

#### *5.2.3: Sampling*

When measuring various per-frame time costs in a given scenario, we take a sample consisting of 20 observations. Each observation consists of average per-frame costs in a 5-minute run of the application.

### 5.3: Observations

#### *5.3.1: Frame Lag*

Lag is measured to be 0 frames (i.e. the blue rectangle is rendered at the same time as a high 3D marker) as long as the 2D scene update occurs before the Flash runtime's EXIT_FRAME event. Typical practice among Flash developers is that application code performs 2D scene updates in response to the runtime's ENTER_FRAME event, which does precede

the EXIT_FRAME event. (ENTER_FRAME is followed by FRAME_CONSTRUCTED, which is followed by EXIT_FRAME.) Thus, 0 frames of lag are introduced by typical use of Illusion. Atypical use, involving 2D scene updates in response to EXIT_FRAME or subsequent events, can introduce a lag of 1 frame.

*5.3.2: CPU Time Breakdown*

The amount of time that MinimalProfiler spends inside all Illusion functions, combined, remains less than 1 ms in every case and less than 10 μs in most cases (Table Series 8). With a single marker pool and tracker, the per-frame cost of Illusion functions ranges from 0.45 μs to 1.1 μs, depending on the test machine. With 100 marker pools, the cost is at worst 8.5 μs; with 1000, at worst 940 μs. For a given number of marker pools, the cost is lowest when the ratio of marker pools to trackers is about 10:1.

Similarly, our tests of MinimalProfiler spend less than 10 μs per frame inside Alternativa3D functions. With a single marker pool and tracker, these functions' per-frame cost ranges from 3.6 μs to 7.5 μs, depending on the test machine.

Rather, an overwhelming proportion of the time cost in MinimalProfiler is associated with Illusion's and Alternativa3D's calls to lower-level functions provided by the Flash standard library. Depending on the test machine, 10.2 ms to 106 ms per frame (86% to 99% of the total time cost) is associated with Flash's Context3D.drawToBitmapData() function alone. This function is essential to the 2D/3D compositing technique used by ARViewportUsingStage and ARViewportUsingStageVideo. Another

0.73 ms to 3.4 ms (0.86% to 13%) is associated with Flash rendering functionality that is not specific to the compositing technique.

## Table Series 8: Time Costs per Frame, MinimalProfiler

**Table 8A: 1 Tracker, 1 Marker Pool, Varying Machine, Varying Category of Cost**

| Category | Dell Inspiron 9400 | MacBook Pro 13" mid-2010 | Custom gaming desktop |
|---|---|---|---|
| Total, excluding TheMiner | Mean: 110 ms<br>StD: 0.72 ms | Mean: 11.8 ms<br>StD: 0.50 ms | Mean: 84.6 ms<br>StD: 0.22 ms |
| Illusion SDK | Mean: 1.0 µs<br>(0.00091%)<br>StD: 0.47 µs | Mean: 1.1 µs<br>(0.0093%)<br>StD: 0.077 µs | Mean: 0.45 µs<br>(0.00053%)<br>StD: 0.29 µs |
| Alternativa3D | Mean: 7.5 µs<br>(0.0068%)<br>StD: 3.8 µs | Mean: 3.6 µs<br>(0.031%)<br>StD: 0.34 µs | Mean: 5.3 µs<br>(0.0035%)<br>StD: 3.0 µs |
| Flash rendering | Mean: 3.4 ms<br>(3.1%)<br>StD: 0.44 ms | Mean: 1.5 ms<br>(13%)<br>StD: 0.063 ms | Mean: 0.73 ms<br>(0.86%)<br>StD: 0.040 ms |
| flash.display3D.<br>Context3D.<br>drawToBitmap() | Mean: 106 ms<br>(96%)<br>StD: 0.70 ms | Mean: 10.2 ms<br>(86%)<br>StD: 0.57 ms | Mean: 83.8 ms<br>(99%)<br>StD: 0.19 ms |

**Note:** The breakdowns are not exhaustive, so they do not sum to the totals.

**Table 8B: MacBook Pro 13" Mid-2010, Cost of Illusion Functions Only, Varying Number of Marker Pools, Varying Number of Trackers**

| Marker pools<br>Trackers | 1 | 10 | 100 | 1000 |
|---|---|---|---|---|
| **1** | Mean: 1.1 µs<br>StD: 0.077 µs | Mean: 1.0 µs<br>StD: 0.42 µs | Mean: 8.5 µs<br>StD: 1.4 µs | Mean: 940 µs<br>StD: 210 µs |
| **10** | | Mean: 1.0 µs<br>StD: 0.33 µs | Mean: 3.0 µs<br>StD: 1.1 µs | Mean: 140 µs<br>StD: 41 µs |
| **100** | | | Mean: 3.6 µs<br>StD: 0.81 µs | Mean: 48 µs<br>StD: 8.9 µs |
| **1000** | | | | Mean: 63 µs<br>StD: 7.6 µs |

**Note:** The stated number of marker pools is the total; it is not per tracker.

*5.3.3: Overall Performance*

Subjectively, ApplesAndGoblets is choppy but usable on the Dell Inspiron and the custom gaming desktop. The application runs more smoothly on the MacBook Pro. Regardless of the test machine, use of StageVideo makes ApplesAndGoblets seem more responsive. Particularly, the video background runs more smoothly and with less lag, so the user can watch his own motions more comfortably. The improvement when using StageVideo is most pronounced on the MacBook Pro, where the StageVideo version runs very fluidly.

Timings of ApplesAndGoblets (Table 9) are consistent with the subjective impressions. The Dell Inspiron runs the application at

approximately 6 FPS; the custom gaming desktop, approximately 8 FPS. The MacBook Pro can exceed 17 FPS without StageVideo, or 46 FPS with StageVideo. These frame rates refer to the 3D scene; the video may run at another (higher) frame rate when using StageVideo.

The Flash release runtime consistently yields better performance than the debug runtime does. The release runtime's speedup ratio in ApplesAndGoblets ranges from 1.08 to 1.27, depending on the test machine and whether StageVideo is used.

**Table 9: Total Time Cost per Frame of ApplesAndGoblets while Tracking 1 Natural Feature Marker and Rendering 1 Spinning Apple**

| Flash version | Context of video background | Dell Inspiron 9400 | MacBook Pro 13" mid-2010 | Custom gaming desktop |
|---|---|---|---|---|
| 11.4 debug | Stage | Mean: 177 ms (5.65 FPS) StD: 2.9 ms | Mean: 72.8 ms (13.7 FPS) StD: 1.5 ms | Mean: 138 ms (7.25 FPS) StD: 0.78 ms |
| 11.4 debug | StageVideo | Mean: 182 ms (5.49 FPS) StD: 2.0 ms | Mean: 23.5 ms (42.6 FPS) StD: 1.7 ms | Mean: 142 ms (7.04 FPS) StD: 1.2 ms |
| 11.4 release | Stage | Mean: 164 ms (6.10 FPS) StD: 1.6 ms | Mean: 57.5 ms (17.4 FPS) StD: 1.7 ms | Mean: 124 ms (8.06 FPS) StD: 1.1 ms |
| 11.4 release | StageVideo | Mean: 161 ms (6.21 FPS) StD: 1.5 ms | Mean: 21.6 ms (46.3 FPS) StD: 0.50 ms | Mean: 128 ms (7.81 FPS) StD: 0.65 ms |

**Note:** For StageVideo, the video runs at an independent (higher) frame rate.

### 5.4: Analysis

Within the tested range, Illusion's time cost seems to grow at a less-than-linear rate with respect to number of trackers but at a linear or greater-than-linear rate with respect to number of marker pools. (See Table 8B in "5.3.2: CPU Time Breakdown".) Moreover, the time cost seems to grow with the number of marker pools per tracker. The differences in growth characteristics might be due to differences in iteration approaches. Illusion internally creates an iterator object each frame for each marker pool, whereas it does not use iterator objects for trackers. Moreover, a dynamically sized vector is created each frame for each tracker and is populated with the tracker's marker pool iterators. Here are some examples of the per-frame operations associated with these iterators and vectors:

- Each iterator is allocated, has its iteration function invoked multiple times, and is garbage collected.
- Each vector may be resized multiple times. The number of times the vector is resized depends on the number of iterators added to the vector—thus, the number of marker pools per tracker. The cost of each resizing depends on the same thing.

Thus, there is room for optimizing iteration over marker pools.[8] However, even in the worst test case, Illusion's per-marker cost is only 1.1 μs, which is likely to be negligible relative to the underlying tracker's per-marker cost.

Even if Illusion were ported to other platforms, we expect that its overhead costs, like iteration, would remain negligible relative to the

[8] To test the hypothesis that the iteration approach affects Illusion's scalability, we conducted some preliminary tests with revisions that do not

underlying costs of sensing, tracking, and compositing. For example, on any platform, it should be possible to achieve iteration costs that grow only linearly with respect to the number of elements. Two commonplace approaches with this characteristic are linked lists and indexing into fixed-size arrays.

The standard deviation of Illusion's time cost also seems to increase with the number of marker pools per tracker. This pattern might reflect underlying variations in the cost of resizing vectors.

Illusion's time cost is similar on the Dell Inspiron and MacBook Pro, yet is less on the custom gaming desktop. (See Table 8A in "5.3.2: CPU Time Breakdown".) This result is unsurprising because the former two machines have relatively similar CPUs, while the latter machine's CPU is clocked faster and has twice as many cores. (See Table *7* in "5.2.1: Dependencies and Platforms".)

Given its low cost at multiple scales and on multiple machines, Illusion proves to be lightweight despite its high-level architecture. That is to say, calls to Illusion's functions pass through to underlying platform and tracker functions at negligible cost in time. Moreover, an inspection of paused frames reveals that Illusion introduces no frame lag (in normal usage, i.e. when client code adheres to typical event handling practices). (See "5.3.1: Frame Lag".) Thus, latency is as good as possible for the given I/O systems, tracker, and overall frame rate.

Alternativa3D also seems to be lightweight, as far as we can judge from the simple 3D scene in MinimalProfiler. Alternativa3D's cost in

---

use iterator objects. These tests suggest that the time cost can be reduced by a factor of two or more in the case of a single tracker with 1000 marker pools. Note that in absolute terms this difference is still small.

MinimalProfiler is at worst 7.5 μs, which is negligible. Moreover, we know that Alternativa3D does not introduce frame lag (in normal usage); otherwise, Illusion would introduce frame lag too.

The time cost of Alternativa3D is lower on the MacBook Pro than on the Dell Inspiron or the custom gaming desktop. The reason for this difference is not apparent to us. However, in absolute terms, the difference is negligible—again, as far as we can judge from the simple 3D scene in MinimalProfiler.

The time cost of Illusion's 2D/3D compositing technique (as implemented in ARViewportUsingStage and ARViewportUsingStageVideo) is high and has surprising variations across the test machines: the MacBook Pro outperforms the custom gaming desktop. Differences in memory architecture are probably the cause. Whereas the desktop uses dedicated graphics memory (and the Dell Inspiron may use dedicated graphics memory, depending on the circumstances), the MacBook Pro uses shared memory. Shared memory would make GPU rendering results more cheaply accessible to the CPU, as the compositing technique demands.

The rationale for the current compositing technique is peculiar to the limitations of Flash. (See "2.6.4: Focus on Flash".) The current technique marshals the 3D rendering results from graphics memory to main memory. An alternative implementation could marshal camera data from main memory to graphics memory, though this approach would preclude the use of StageVideo. A GPU-accelerated compositing technique that avoids marshalling is currently infeasible in Flash because Stage3D supports neither camera input nor background transparency.

On future versions of Flash, or on other platforms, it is important that Illusion evolve to support fully GPU-accelerated compositing. Techniques that avoid reading back graphics memory (or, conversely, uploading video buffers to graphics memory each frame) should be both faster and more predictable, as they eliminate an entire category of bus costs.

For camera input, StageVideo (via ARViewportForStageVideo) offers some advantages over Stage (via ARViewportForStage). (See Table 9 in "5.3.3: Overall Performance".) StageVideo is categorically more efficient on the MacBook Pro, while on the other test machines it seems to improve the smoothness and responsiveness of the video background only. Even in the latter case, the user might perceive an overall improvement. Real-world objects are the focus of interactivity in AR, so the user might first watch for a real-world event in the live video scene, and only then watch for a virtual-world event in the 3D scene.

It is surprising that StageVideo improves the overall frame rate on the MacBook Pro but not on the other test machines. A possible explanation relates back to the compositing inefficiency, albeit indirectly. Recall that CPU time cost on the Dell Inspiron and the custom gaming desktop is dominated by compositing, more so than on the MacBook Pro. Use of StageVideo does not offload compositing to the GPU, though it does offload some rendering, which accounts for a higher proportion of CPU time cost on the MacBook Pro than on the other machines. Thus, the MacBook Pro should realize a greater speedup ratio from the use of StageVideo.

Note that the higher proportional cost of Flash rendering on the MacBook Pro does not imply a higher absolute cost. The absolute cost is

greater on the Dell Inspiron, though less on the custom gaming desktop. This ordering is consistent with the machines' relative processing power. Both CPU and GPU capabilities might be relevant in this case if Flash conditionally offloads some rendering to the GPU.

With some caveats, Illusion and its flare* wrappers are already usable in real-world applications. Graphically intensive AR applications—for example, using high-polygon models and dynamic lights—perform well on at least some hardware that uses shared graphics memory. Applications that do not use the compositing functionality—for example, games that use tracking to control purely virtual scenes—should perform well on diverse hardware.

# Chapter 6: Discussion

This chapter reviews the motivations behind this thesis project and the extent to which they are resolved by Illusion SDK. We conclude that Illusion's design can support efficient implementations for many platforms and many ubiquitous applications. For Flash, the current compositing implementations are inefficient on some hardware, though otherwise efficiency is not problematic.

Continued development of Illusion is proposed. Optimizations of the current compositing implementations are planned in anticipation of new features in Flash. Also, the author and other parties should be able to port and extend Illusion to cover more platforms and more techniques for sensing, tracking, and compositing.

## 6.1: Ubiquity

AR's potential for ubiquity was one of the motivations behind this thesis project. Illusion SDK addresses this potential by providing an extensible, modular, portable, high-level architecture that includes abstractions of sensors, trackers, and compositors. These abstractions are agnostic about the platform's I/O capabilities: they can support various sources of sensor data and destinations for composited scenes. Moreover, trackers and compositors have no dependency on each other. A notable consequence is that compositing can be omitted—or handled by a non-Illusion module—if the platform or application so requires. (See the "Design and Contribution" chapter.) For these reasons, Illusion's abstractions are portable to ubiquitous computers that may have unconventional interfaces.

Illusion's zlib license is friendly to third-party ports. For example, device manufacturers could make proprietary ports to their platforms. Our intent is to encourage the adoption of Illusion's design, even if our own implementation is inappropriate to a given ubiquitous computing platform.

Illusion's current implementation, targeting Flash, is not in itself ubiquitously deployable. However, it is easily deployable to Web users on Windows and Mac. (No incompatibilities were encountered in informal testing on our own and colleagues' machines.) Deployment in the form of desktop applications, via Adobe AIR, should also work (though this path is not yet tested). Besides being available on Windows and Mac, the AIR runtime might have a future on proprietary appliances that could be characterized as ubiquitous computers. For example, recent LG Smart TVs run AIR 3.0, which is equivalent to Flash 11.0 (Magni, 2012). Flash's security and privacy features, along with its support for peer-to-peer networking, may make it a suitable platform for applications that rely on a network of ubiquitous computers. (See "2.6.4: Focus on Flash".)

### 6.2: Efficiency

The need for AR applications to be responsive—and therefore efficient—was another motivation behind this thesis project. Illusion SDK's design is capable of supporting efficient implementations that have low time cost per frame and do not introduce any whole frames of latency. (See the "Evaluation" chapter.)

Most of Illusion's implementation for Flash is efficient. There is one major efficiency problem in Illusion's compositing implementation for Flash: the compositor may marshall data between main memory and

graphics memory every frame, at high cost in time. This problem arises from Flash's use of multiple graphics contexts, some of which cannot capture camera input and some of which cannot be layered in front of another context. (See "2.6.4: Focus on Flash" and "5.4: Analysis".) Generally, ports of Illusion to other platforms should not suffer from the same compositing problem.

On systems that use shared graphics memory, Illusion's current compositing implementation runs efficiently. Also, in applications that do not require compositing, the current efficiency problem is irrelevant. For example, client code could use Illusion's sensing and tracking functionality to control nodes in a pure 3D scene with no video background. Though it might not fit the definition of AR, this use case is envisioned by Illusion's modular design.

### 6.3 Conclusion

This thesis has contemplated the statement that ubiquity and efficiency are critical in an industry-grade AR engine—and that finding or creating this combination is nontrivial. A good solution, we have argued, can deliver this combination, provided that care is taken to use the strengths and avoid the weaknesses of a platform's idiosyncrasies.

Our proof and practical contribution lie in the design, implementation, and evaluation of Illusion SDK. Illusion is an AR engine whose modular, extensible design can potentially support ubiquitous and efficient applications. The Flash implementation takes account of certain platform peculiarities, strengthening its performance results on some hardware. The design is portable to other platforms regardless of I/O capabilities, and implementations are expected to have low overhead cost regardless of

platform. Compared to previous solutions by the author and others, Illusion has the potential to reduce rework in certain industry-relevant situations, as needs and technologies change.

### 6.4: Future Work

This thesis project has coincided with significant developments in Flash and other Web application runtimes. Flash 11.4 offers GPU-accelerated, readable camera input, increasing the platform's suitability for AR. More generally, Flash 11's prospects as a game platform are being bolstered by support from cross-platform engines such as Unity and Unreal. Where the latest Flash versions are supported (i.e. on Windows and Mac), they continue to see strong rates of adoption.

On the other hand, Flash has failed to address the problem of increasing fragmentation in Web clients' operating systems. It lacks iOS support and has recently dropped Android and Linux support. Silverlight and Java have similarly failed to support the mobile Web. Instead, it is likely that WebRTC (for camera access) and WebGL (for GPU acceleration) will make JavaScript a viable solution for cross-platform AR in future versions of mobile and desktop browsers.

A reexamination of target platforms is the first priority in future work on Illusion. Although Illusion's abstractions come at negligible performance cost (and so do Alternativa3D's), they are implemented atop Flash platform functions that may carry an undue cost. We want to optimize Illusion for future Flash versions and also port it to other Web application runtimes in order to determine how the overall performance of AR scenes is affected by the platform.

Beyond Flash 11.4, the platform might evolve to support transparent backgrounds in Stage3D, thus enabling a GPU-accelerated compositing solution. There is a precedent for this feature in a beta version of Flash 11.0. (See "2.6.4: Focus on Flash".) ARViewportUsingStageVideo is written in a way that anticipates this feature, so adopting it should require only a one-line change to the code.

We have not yet studied the challenges of porting Illusion to JavaScript (or any other target in particular). The range of third-party trackers will differ greatly: Flash is quite mature in this respect, while JavaScript (and some other targets) are not. Illusion and all its ports should share at least one functionally equivalent tracker for testing purposes. ARToolkit derivatives are a likely choice, since they are already available for Flash (FLARToolkit), JavaScript (JSARToolkit), and many other targets.

Another priority is the improvement of Illusion's content pipeline. Animations should be supported in the 3D model importer. (There is some underlying support for animation import in Alternativa3D but Illusion's higher-level importer does not provide any abstractions of it.) As the content pipeline is closely integrated with the underlying game or graphics engine, it might differ greatly across ports of Illusion. For example, a port that integrated with Unity would already get a high-level content pipeline as part of the Unity toolchain.

An ongoing concern is to expand the number of concrete tracker interfaces in Illusion, either by wrapping existing trackers or implementing new ones. Facial tracking and 3D object tracking are not covered by Illusion's current tracking wrappers, so these types of tracking could be priorities. Non-visual (ex. audio) sensing and tracking also

remains to be explored. Original trackers should take advantage of GPU acceleration and other forms of multiprocessing, where available. An eventual goal is for Illusion to support AR applications that are GPU-accelerated at the three costly stages of tracking, rendering, and compositing.

## Appendix A: Availability and Licensing

For the latest version of Illusion, visit https://github.com/JoeHowse/Illusion/. The latest version may have refinements over the version described in this thesis. (See "6.4: Future Work".)

Illusion is available under zlib license, a liberal open-source license that permits both open-source and closed-source extensions, for noncommercial or commercial use. For example, anyone is free to make and sell an application that interfaces with Illusion, a tracker that extends Illusion, or an authoring environment that embeds Illusion.

We believe that a liberal open-source license helps make a software library relevant to a broad audience and broad problem set. Open-source code fosters knowledge transfer among software architects: without it, students and even professionals would have relatively few opportunities to study large codebases (Brown and Wilson 2012). It can be audited—a requirement in critical applications such as military simulations (McDowell et al, 2006; McDowell, 2008) and forensic tools (Carrier, 2002). It can be forked and redistributed by anyone, so vendor lock-in is avoided—also a requirement in critical and long-term applications (McDowell et al, 2006; McDowell, 2008). It is available without cost, without embargo against any user group, and (under "liberal" licenses) without prohibition against any use case.

Compared to the Flash-based alternatives—notably, FLARManager and vendor-specific offerings—Illusion is available under a more liberal license, making it extensible by the broadest possible developer community. (See "4.3: Comparison to Other Designs" for more differences that pertain to extensibility and generality.) FLARManager is

open-source under GNU Public License (GPL) but this license prohibits closed-source extensions. For the latter purpose, FLARManager offers an alternative, paid license. Vendor-specific offerings are typically closed-source, paid, or both. Illusion's zlib license encourages both open-source and closed-source extensions, for noncommercial or commercial use.

Note that while Illusion is open-source under zlib license, implementations of its AbstractTracker type may wrap libraries that fall under other licenses. (See "4.1.3: Tracking Markers".) Clients who use or make such wrappers must abide by the tracking libraries' licenses. For example, flare*tracker and flare*nft are closed-source, commercial libraries that can be licensed from Imagination Computer Services GmbH (http://www.imagination.at/en/?Products:Augmented_Reality).
Imagination provides demo licenses, with no expiry, for use on localhost.

Alternativa3D (Illusion's dependency for 3D scene graph functionality in Flash) is open-source under Mozilla Public License. Alternativa3D's latest source code is available from https://github.com/AlternativaPlatform/Alternativa3D and the latest precompiled binary is available from http://alternativaplatform.com/en/download8/.

## Appendix B: Non-AR Functionality

### B.1: Loading Binary or Text Files

The Flash standard library provides an event-driven (observer) pattern for asynchronous file loading. The Flex standard library provides an alternative pattern of static prototypes, whereby arbitrary files may be embedded at compile-time as classes.

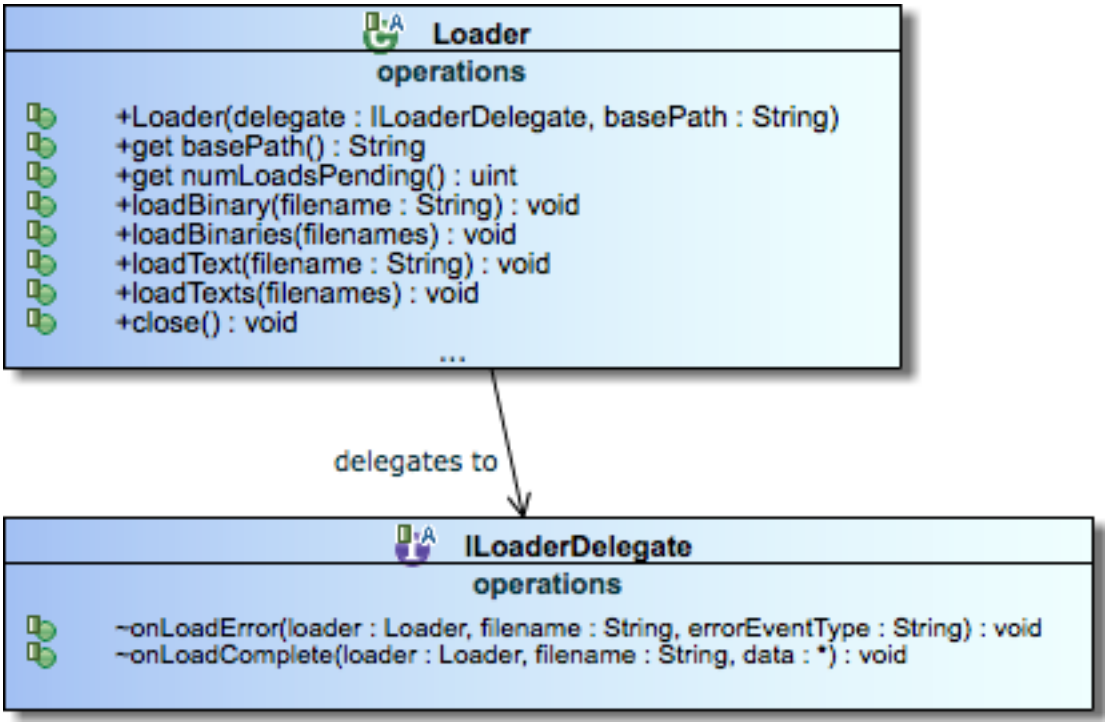
Neither of these pre-existing patterns is appropriate to Illusion's internal needs. Under the observer pattern, synchronizing multiple file loads is cumbersome. Under the static prototype pattern, Illusion and the client application become bloated (due to the Flex dependency and static data) and more tightly coupled (due to the lack of dynamic data).

As a substitute, Illusion provides a mediator pattern, with delegation, for asynchronous file loading (Figure A; Code Sample A). The delegator class, Loader, is instantiated with a delegate and a base file path. The delegate must implement an interface called ILoaderDelegate, which has functions for handling load successes and load errors. The delegator has functions for requesting that one or more file paths (relative to the specified base path) be loaded as either binary data or text. The delegator can be polled (ex. by the delegate, on a load success) to determine the number of pending loads: 0 pending loads indicates that a synchronization point has been reached. The delegator can be told (ex. by the delegate, on a load error) to cancel all pending loads.

Internally, Loader still uses the observer pattern. The particular implementation of this pattern ensures that a Loader and its delegate are never garbage-collected while there are pending loads.

Illusion already uses Loader and ILoaderDelegate internally, so client code does not necessarily need access to general-purpose file loading functionality. Also, client code is free to use either of the two standard file loading patterns, or some other alternative; Illusion just provides Loader as an option.

**Figure A: Design of Loader and ILoaderDelegate**



### Code Sample A: Usage of Loader and ILoaderDelegate

```
public class MyLoaderDelegate implements ILoaderDelegate
{
      var binaryData_:ByteArray;
      var textData_:String;

      public function MyLoaderDelegate()
      {
            // Create a loader with this delegate and a base
            // path of "data".
            var loader:Loader = new Loader(this, "data");

            // Load a binary file.
            loader.loadBinary("stuff.bin");

            // Load multiple binary files.
            //loader.loadBinaries("stuff.bin", "dreams.bin");

            // Load a text file.
            loader.loadText("prose.txt");

            // Load multiple text files.
            //loader.loadTexts("prose.txt", "verse.txt");
      }

      // Part of the ILoaderDelegate implementation.
      public function onLoadError(loader:Loader,
                                  filename:String,
                                  errorEventType:String):void
      {
            // Cancel any remaining loads.
            loader.close();

            // Throw an error, saying which load failed and
            // what the failure was.
            throw new Error("Failed to load \"" +
                            loader.basePath + filename + "\": " +
                            errorEventType);
      }

      // Part of the ILoaderDelegate implementation.
      public function onLoadComplete(loader:Loader,
                                     filename:String,
                                     data:*):void
      {
            if (filename == "stuff.bin")
            {
                  binaryData_ = data as ByteArray;
            }
            else // filename == "prose.txt"
            {
```

```
                textData_ = data as String;
            }

            if (loader.numLoadsPending > 0)
            {
                return;
            }

            // Both files have loaded. Now we can do something
            // that requires both files' data.
            …
        }
}
```

**B.2: Loading 3D Model Files**

Alternativa3D provides functionality for parsing certain binary and text formats as 3D scene graph branches (ex. 3D models), containing references to any external resources (ex. textures) that need to be loaded before render-time. However, this functionality falls short of being an asset pipeline. On one end, it does not facilitate loading the original binary or text data from files, nor does it generate any materials (just resource references and metadata). On the other end, it does not aggregate resource references so as to facilitate efficient uploading to the GPU and efficient disposal from the GPU. To find resource references and generate appropriate materials, client code must walk the parsed scene branch. Naively implemented client code might generate duplicate materials and make redundant resource uploads to the GPU, especially if the scene contains multiple copies of the branch.

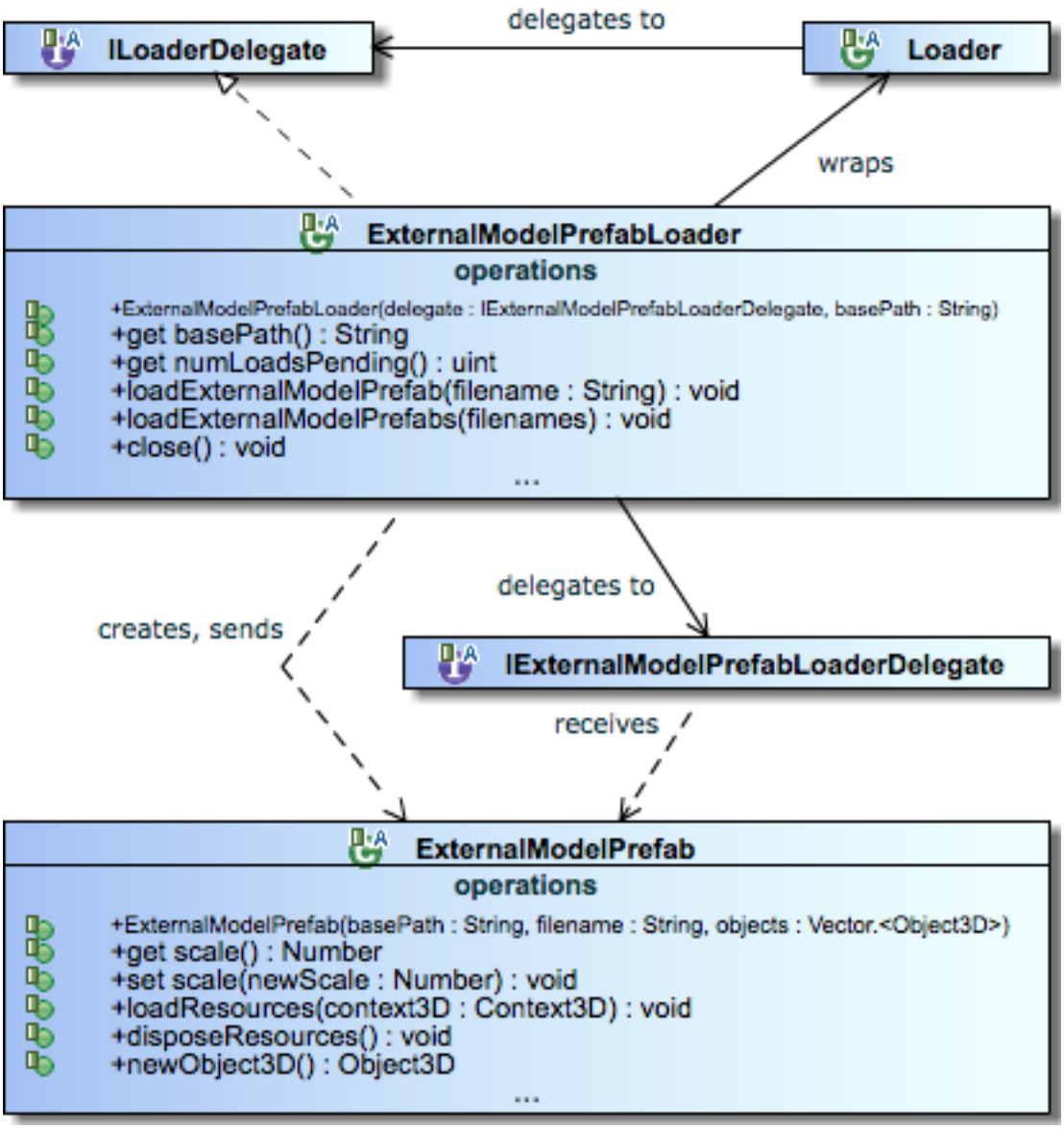
To address these shortcomings and pitfalls, Illusion provides higher-level functionality via a builder class, ExternalModelPrefab. This class is instantiated with a prototype of a 3D scene graph branch. Client code

typically does not instantiate a builder directly but instead gets one from a file loader of type ExternalModelPrefabLoader (Figure B; Code Sample B), which internalizes the steps of loading the 3D data file, parsing it as a prototype, and instantiating an ExternalModelPrefab with the prototype. ExternalModelPrefabLoader uses the mediator pattern and delegation to provide an interface that is similar to Loader's. The delegate must implement an interface called IExternalModelPrefabLoaderDelegate, which is similar to ILoaderDelegate. (See the previous section, "Loading Binary or Text Files".)

Internally, ExternalModelPrefabLoader creates a Loader and acts as its delegate. Being a Loader's delegate, an ExternalModelPrefabLoader is never garbage-collected while there are pending loads.

Supported file formats are .dae (COLLADA), .3ds (3D Studio), and .a3d (Alternativa3D). Meshes are imported, and materials are generated with the following components: diffuse, normal, specular, gloss, and alpha. Each component uses a texture map if one is specified by the data; failing that, a uniform color if one is specified by the data; and failing that, a default uniform color. The current version of ExternalModelPrefab does not parse non-mesh nodes, nor does it parse animations.

**Figure B: Design of ExternalModelPrefabLoader and Related Types**



**Code Sample B: Usage of ExternalModelPrefabLoader and Related Types**

```
public class MyExternalModelPrefabLoaderDelegate
implements IExternalModelPrefabLoaderDelegate
{
      var stage3D_:Stage3D;
      var scene3D_:Object3D;

      var applePrefab_:ExternalModelPrefab;
      var orangePrefab_:ExternalModelPrefab;

      var littleApple:Object3D;
      var bigApple:Object3D;
      var orange:Object3D;

      public function MyLoaderDelegate(stage3D:Stage3D,
                                       scene3D:Object3D)
      {
            stage3D_ = stage3D;
            scene3D_ = scene3D;

            // Create a loader with this delegate and a base
            // path of "data".
            var loader:ExternalModelPrefabLoader =
                  new ExternalModelPrefabLoader(this, "data");

            // Load a model file.
            //loader.loadExternalModelPrefab("apple.3ds”);

            // Load multiple model files.
            loader.loadExternalModelPrefabs
            (
                  "apple.3ds",
                  "orange.3ds"
            );
      }

      // Part of the IExternalModelPrefabLoaderDelegate
      // implementation.
      public function onLoadExternalModelPrefabError(
            loader:ExternalModelPrefabLoader,
            filename:String,
            errorEventType:String):void
      {
            // Cancel any remaining loads.
            loader.close();

            // Throw an error, saying which load failed and
            // what the failure was.
            throw new Error("Failed to load \"" +
                        loader.basePath + filename + "\": " +
                        errorEventType);
```

```
}

// Part of the IExternalModelPrefabLoaderDelegate
// implementation.
public function onLoadExternalModelPrefabComplete(
      loader:ExternalModelPrefabLoader,
      filename:String,
      data:*):void
{
      if (filename == "apple.3ds")
      {
            applePrefab_ = data as ByteArray;
      }
      else // filename == "orange.3ds"
      {
            orangePrefab_ = data as String;
      }

      if (loader.numLoadsPending > 0)
      {
            return;
      }

      // Both prefabs have loaded. Now we can do
      // something that requires both prefabs, such as
      // populating a scene with models.

      // Make models.

      littleApple = applePrefab_.newObject3D();

      applePrefeb_.scale = 2.5;
      bigApple = applePrefab_.newObject3D();

      orange = orangePrefab_.newObject3D();

      // Upload resources to the GPU.
      applePrefab_.loadResources(stage3D_.context3D);
      orangePrefab_.loadResources(stage3D_.context3D);

      // Populate the scene.
      scene3D_.addChild(littleApple);
      scene3D_.addChild(bigApple);
      scene3D_.addChild(orange);
      …
}

public function dispose():void
{
      // Depopulate the scene.
      scene3D_.removeChild(littleApple);
      scene3D_.removeChild(bigApple);
```

```
            scene3D_.removeChild(orange);

            // Unload resources from the GPU.
            applePrefab_.unloadResources();
            orangePrefab_.unloadResources();
            …
      }
}
```

### B.3: Creating Lighting Setups

The author's experience is that developers often begin a project with naive or lazy approaches to lighting. As such, the application prototype might give a misleadingly poor impression of the 3D artwork—particularly the materials. When lacking a more deliberate lighting design, developers should pick a standard cinematic setup that tends to produce a variety of hard and soft highlights and shadows (Arnold, 2011).

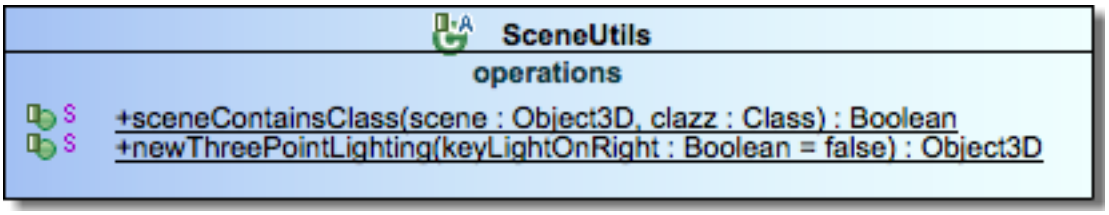
One such setup is **three-point lighting**, consisting of a **key light**, a **fill light**, and a **back light**. The key light is a bright light facing the subject from above-front-left or above-front-right. It provides broad illumination of the subject, though with partial shadow. The fill light is a dim light facing the subject from above-front-right or above-front-left. It softens the shadows on the subject. The back light is a bright light facing the subject from back-left or back-right. It highlights the subject's silhouette.

Illusion facilitates the use of a three-point lighting setup by providing a static factory method in the SceneUtils class (Figure C, Code Sample C). The constructee is a 3D node with three directional lights as children. As the lights are directional, their position relative to the subject is irrelevant, and usage is extremely simple (albeit inflexible).

This implementation of a three-point lighting setup is intended only for rapid prototyping of simple scenes, where light rays' points of origin are not of particular concern. Multiple directional lights, as used in this implementation, may produce strange-looking results in complex scenes.

SceneUtils also provides a static function for checking whether a specified 3D node or any of its subnodes are instances of a specified class. Such information can be useful for optimization purposes. For example, if an entire scene contains no meshes, it need not be rendered. The latter optimization is used internally by Illusion's compositors.

**Figure C: Design of SceneUtils**



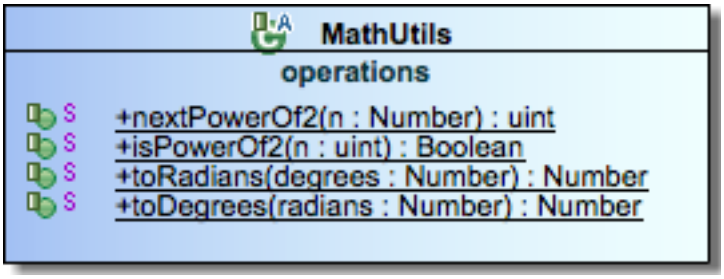

**Code Sample C: Usage of SceneUtils**

```
var scene3D:Object3D = new Object3D();
sceneContainsClass(scene3D, Light3D); // false
…
var threePointLighting:Object3D =
     LightingUtils.newThreePointLighting();
scene3D.addChild(threePointLighting);
sceneContainsClass(scene3D, Light3D); // true
```

### B.4: Miscellaneous Static Functions

Certain math functions are provided statically in the MathUtils class (Figure D; Code Sample D). Specifically, these functions relate to powers of 2 and angle conversions.

**Figure D: Design of MathUtils**

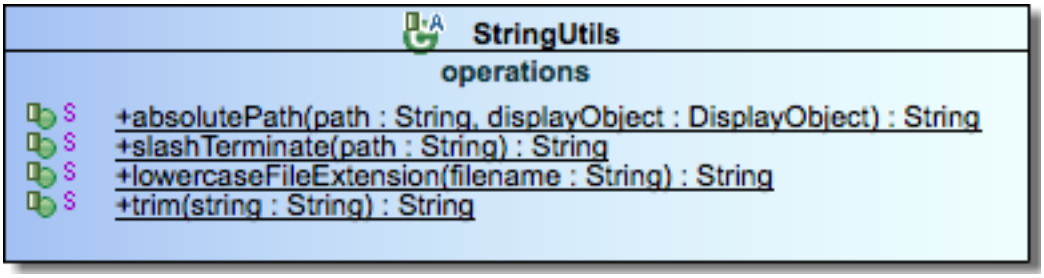



**Code Sample D: Usage of MathUtils**

```
MathUtils.nextPowerOf2(3); // 4
MathUtils.nextPowerOf2(4); // 4

MathUtils.isPowerOf2(3); // false
MathUtils.isPowerOf2(4); // true

MathUtils.toRadians(180); // approximately Math.PI

MathUtils.toDegrees(Math.PI); // approximately 180
```

Certain functions for parsing strings are provided statically in the StringUtils class (Figure E; Code Sample E). Specifically, most of these functions relate to parsing file paths. Classes in Illusion already use these functions internally to parse file path arguments; client code does not need to pre-parse file paths when interfacing with Illusion.

**Figure E: Design of StringUtils**



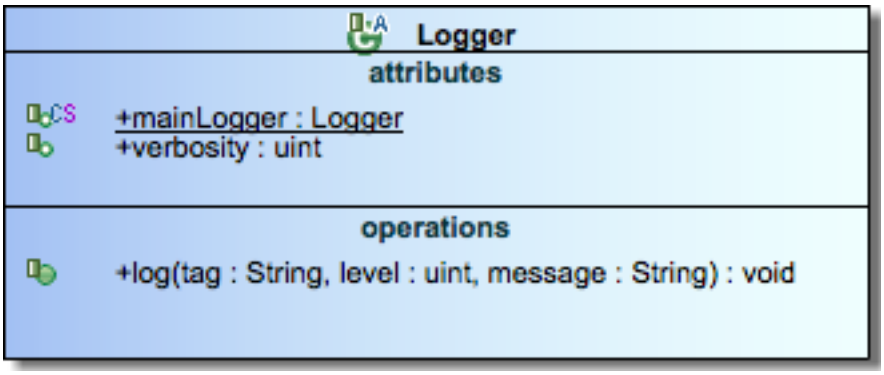

**Code Sample E: Usage of StringUtils**

```
var loadee:DisplayObject;
// Can get information about the application's loading path.
// Suppose the loading path is "http://nummist.com".
…

StringUtils.absolutePath("data/cat.gray.PNG", loadee);
     // http://nummist.com/data/cat.gray.PNG

StringUtils.lowercaseFileExtension("cat.gray.PNG"); // ".png"

StringUtils.slashTerminate("data"); // "data/"

StringUtils.trim(" \t\ngrayspace \t\n"); // "grayspace"
```

Logging functionality is provided in the Logger class (Figure F; Code Sample F). Logger exposes a static, constant instance that may be treated as if it were a singleton. Alternatively, client code may create its own instances of Logger for the sake of independent configurability. A

Logger's verbosity level can be set, and log requests can be made with a specified tag (such as a class name), priority level, and message. A log request will be fulfilled (using ActionScript's standard trace function) if its priority level is less than or equal to the verbosity level.

**Figure F: Design of Logger**




**Code Sample F: Usage of Logger**

```
Logger.mainLogger.log("My tag", 0, "My priority 0 message");
// Gets logged as "My tag [0]: My priority 0 message".

Logger.mainLogger.log("My tag", 1, "My priority 1 message");
// Does not get logged because verbosity defaults to 0.

// Increase verbosity to 1.
Logger.mainLogger.verbosity = 1;

Logger.mainLogger.log("My tag", 1, "My priority 1 message");
// Gets logged as "My tag [1]: My priority 1 message".
```

**Bibliography**


Abdeles, P. (2012). Resolving Implementation Ambiguity and Improving SURF. Retrieved February 24, 2012, from http://arxiv.org/pdf/1202.0492v1.

Adobe. ActionScript Technology Center. Retrieved February 1, 2012, from http://www.adobe.com/devnet/actionscript.html.

Adobe. Adobe AIR Release Notes. Retrieved March 29, 2012, from http://www.adobe.com/support/documentation/en/air/releasenotes.html.

Adobe. Alchemy. Retrieved August 10, 2012, from http://labs.adobe.com/technologies/alchemy/.

Adobe. Cirrus. Retrieved September 28, 2012, from http://labs.adobe.com/technologies/cirrus/.

Adobe. Director 11.5 Help. Retrieved February 1, 2012, from http://help.adobe.com/en_US/Director/11.5/UsingScripting/index.html.

Adobe. Flash Player security and privacy. Retrieved September 27, 2012, from http://www.adobe.com/security/flashplayer/.

Adobe. Flash Player Release Notes. Retrieved March 29, 2012, from https://www.adobe.com/support/documentation/en/flashplayer/releasenotes.html.

Adobe. Getting started with stage video. Retrieved March 28, 2012, from http://www.adobe.com/devnet/flashplayer/articles/stage_video.html.

Adobe. How Stage3D Works. Retrieved March 27, 2012, from http://www.adobe.com/devnet/flashplayer/articles/how-stage3d-works.html.

Adobe. Pixel Bender Technology Center. Retrieved February 1, 2012, from http://www.adobe.com/devnet/pixelbender.html.

Adobe. Preview 3 and the future of PB3D. Retrieved March 31, 2012, from http://blogs.adobe.com/pixel-bender/2011/09/22/preview-3-and-the-future-of-pb3d/.

Adobe. RTMFP FAQ. Retrieved October 1, 2012, from http://www.adobe.com/products/flash-media-enterprise/rtmfp-faq.html.

Adobe. What is AGAL. Retrieved April 1, 2012, from http://www.adobe.com/devnet/flashplayer/articles/what-is-agal.html.

Adobe. Worker. Retrieved October 1, 2012, from http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/system/Worker.html.

AlternativaPlatform. Camera3D - API Documentation. Retrieved April 8, 2012, from http://alternativaplatform.com/en/docs/8.27.0/.

AlternativaPlatform. Showcase. Retrieved March 25, 2012, from http://alternativaplatform.com/en/showcase/.

Arduino. "Interfacing with Other Software". Retrieved April 6, 2012, from http://arduino.cc/playground/Main/InterfacingWithSoftware.

Arnold, D. (2011). CSCI 4167/6608: Advanced Computer Animation. (Lecture slides). Dalhousie University.

Azuma, R. (1997). A Survey of Augmented Reality. *Presence: Teleoperators and Virtual Environments, 6*(4), 355-385.

Baines, H., & Bomback, E. (1967). *The Science of Photography* (3rd ed.). Fountain Press.

Bay, H., Tuytelaars, T., & Van Gool, L. (2006/2008). SURF: Speeded-Up Robust Features. Proceedings from ECCV 2006: *The 9th European Conference on Computer Vision. Reprinted in Computer Vision and Image Understanding, 110*(3), 346-359.

Beyond Reality. IN2AR. Retrieved July 12, 2012, from http://www.in2ar.com/.

Bidelman, E. (2012, July 20). Capturing Audio & Video in HTML5. Retrieved August 17, 2012, from http://www.html5rocks.com/en/tutorials/getusermedia/intro/.

Billinghurst, M., Kato, H., & Poupyrev, I. (2001). The Magic-Book—Moving Seamlessly between Reality and Virtuality. *IEEE Computer Graphics and Applications, 21*(3), 2-4.

Bradsky, G., & Pisarevsky, V. (2000). Intel's Computer Vision Library: Applications in calibration, stereo, segmentation, tracking, gesture, face and object recognition. Proceedings from CVPR 2000: *2000 Conference on Computer Vision and Pattern Recognition*.

Brown, A., & Wilson, G. (Eds.) (2012). *The Architecture of Open Source Applications*. Lulu.com.

Carrier, B. (2002, October). Open Source Digital Forensic Tools: The Legal Argument. @stake Research Report.

Caudell, T., & Mizell, D. (1992). Augmented Reality: an application of heads-up display technology to manual manufacturing processes. Proceedings from HICSS 1992: *The 25th Hawaii International Conference on Systems Science*(2), 659-669. Kauai, Hawaii.

Christian Doppler Laboratory. ARToolKitPlus. Retrieved April 8, 2012, from http://handheldar.icg.tugraz.at/artoolkitplus.php.

Chúťka, J. (2010, March 4). Fixing Z-sorting in Papervision 3D (update). Retrieved March 25, 2012, from http://blog.yoz.sk/2010/03/fixing-z-sorting-in-papervision-3d/.

Civantos, D. (2010, February 14). Por el amor de Dios: los tatuajes vivientes. Retrieved March 9, 2012, from http://www.cookingideas.es/por-el-amor-de-dios-los-tatuajes-vivientes-20100214.html.

Clarke, R. (1994). *British Aircraft Armament Vol. 2: RAF Guns and Gunsights from 1914 to the Present Day*. UK: Patrick Stephens.

Collins, D. (2011). Bar Codes: The 50th Anniversary of the First Bar Code Scanner. Retrieved March 12, 2012, from http://www.aimglobal.org/members/news/templates/template.aspx?articleid=3827&zoneid=46.

Comport, A. (2005). *Towards a Computer Imagination: Robust Real-time 3D Tracking of Rigid and Articulated Objects for Augmented Reality and Robotics*. (Unpublished dissertation). University of Rennes 1.

Consolvo, S., Roessler, P., Shelton, B., LaMarca, A., Schilit, B., & Bly, S. (2004). Technology for care networks for elders. *Pervasive Computing, 3*, 22-29.

Dobkin, D., & Wandinger, T. (2005, June). A Radio-Oriented Introduction to Radio Frequency Identification. *High Frequency Electronics*, 46-54.

Engelbart, D. (1962). *Augmenting Human Intellect: A Conceptual Framework*. Menlo Park, CA: Stanford Research Institute.

Engelbart, D. (1968). A Research Center for Augmenting Human Intellect. (Lecture). Video republication retrieved April 2, 2012, from http://sloan.stanford.edu/MouseSite/1968Demo.html.

Ernst, T. (2011). *Performance Analysis and Acceleration for Rich Internet Application Technologies*. (Unpublished diploma thesis). University of Ulm.

Evans, C. (2009, January 18). Notes on the OpenSURF Library. Retrieved February 17, 2012, from http://sites.google.com/site/chrisevansdev/files/opensurf.pdf.

Furgale, P., Tong, C., & Kenway, G. (2009). Speeded-Up Speeded-Up Robust Features. (Unpublished project report). University of Toronto.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995.) *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.

Geffroy, M. (2012, February 27.) Total Immersion to Demo D'Fusion AR Solutions on New Intel Ultrabook Platform at GDC 2012. Retrieved February 27, 2012, from http://www.t-immersion.com/blog.

Google. Google Chrome Frame. Retrieved February 10, 2012, from http://code.google.com/chrome/chromeframe/.

Google. WebGL and 3D graphics. Retrieved January 21, 2012, from http://support.google.com/chrome/bin/answer.py?hl=en&answer=1220892.

Gossow, D., Paulus, D., & Decker, P. (2010). An Evaluation of Open Source SURF Implementations. Proceedings from RoboCup 2010: *The 14th Annual RoboCup International Symposium*, 169-179. Singapore.

Grden, J. (2011, September 22, 1:08 a.m.) Re: Optimizing scene with animated DAE model. Message posted to http://papervision3d.758870.n4.nabble.com/Optimizing-scene-with-animated-DAE-model-td3832292.html.

Gutierrez, R. (2012, February 8). Dive Into PS Vita's Augmented Reality Suite. Retrieved February 29, 2012, from http://blog.eu.playstation.com/2012/02/08/dive-into-ps-vitas-augmented-reality-suite/.

Hartl, A. (2010). Computer-Vision based Pharmaceutical Pill Recognition on Mobile Phones. Proceedings from CESCG 2010: *The 14th Central European Seminar on Computer Graphics*.

Hartl, A., Arth, C., Schmalstieg, D. (2011). Instant Medical Pill Recognition on Mobile Phones. Proceedings from CV 2011: *IASTED International Conference on Computer Vision*.

Hee, S. (2012, January 11). Qualcomm Brings Sesame Street to Mobile Devices Through AR. Retrieved March 21, 2012, from http://www.hardwarezone.com/tech-news-qualcomm-brings-sesame-street-mobile-devices-through-ar.

Heikkinen, I. JSARToolkit. Retrieved August 17, 2012, from https://github.com/kig/JSARToolKit.

Helgason, D. (2008, March 31). Thoughts On Browser Plugin Penetration. Retrieved January 30, 2012, from http://blogs.unity3d.com/2008/03/31/thoughts-on-browser-plugin-penetration/.

Henn, S. (2010, October 11). Augmented Reality: drones attack Jimmy Fallon, Marketplace newsroom. *Marketplace*. Retrieved February 28, 2012, from http://www.marketplace.org/topics/tech/news-brief/augmented-reality-drones-attack-jimmy-fallon-marketplace-newsroom.

Hero of Alexandria. (1851). *The Pneumatics of Hero of Alexandria*. (B. Woodcroft, Trans.). London, UK: Taylor Walton and Maberly. (Original work written c. 60 AD).

IEWebGL. IEWebGL - WebGL for Internet Exlporer. Retrieved February 10, 2012, from http://iewebgl.com/.

Imagination. flare*nft. Retrieved February 13, 2012, from http://www.imagination.at/en/?Products:Augmented_Reality_for_Flash:flare*nft.

Imagination. flare*tracker. Retrieved March 25, 2012, from http://www.imagination.at/en/?Products:Augmented_Reality_for_Flash:flare*tracker.

Imbert, T. (2011, October 28). Flash 11.2 and AIR 3.2 beta builds hidden gems. Retrieved March 27, 2012, from http://www.bytearray.org/?p=3684.

Jackson, P. (2011, April 11). 48 Frames Per Second. Retrieved July 4, 2012, from https://www.facebook.com/notes/peter-jackson/48-frames-per-second/10150222861171558.

JavaCV. JavaCV. Retrieved February 12, 2012, from http://code.google.com/p/javacv/.

Kan, T., Teng, C., & Chen, M. (2011). QR Code Based Augmented Reality Applications. In Furht, B. (Ed.), *Handbook of Augmented Reality* (339-354). New York, NY: Springer.

Kane Computing. Compression Ratio Rules of Thumb. Retrieved July 8, 2012, from http://www.kanecomputing.co.uk/pdfs/compression_ratio_rules_of_thumb.pdf.

Kato, H., & Billinghurst, M. (1999). Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System. Proceedings from IWAR 1999: *The 2nd IEEE and ACM International Workshop on Augmented Reality*.

Kazmierczak, M. (2005, March 14). Photo Book Review: Open Shutter by Michael Wesely. Retrieved March 11, 2012, from http://mkaz.com/photography/photo-book-review-open-shutter-by-michael-wesely.html.

Augmented Environments Laboratory. KHARMA. Retrieved November 23, 2012, from https://research.cc.gatech.edu/kharma/.

Klingemann, M. (2009, March 14). Optimizing Flash Based Face Detection. Retrieved February 12, 2012, from http://www.quasimondo.com/archives/000687.php.

Lane, N. D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., & Campbell, A. T. (2010). A Survey of Mobile Phone Sensing. *IEEE Communications Magazine*, September 2010, 140-150.

Leadbetter, R. (2009, September 5). Console Gaming: The Lag Factor. Retrieved July 7, 2012, from http://www.eurogamer.net/articles/digitalfoundry-lag-factor-article.

Leadbetter, R. (2010, June 15). Tech Analysis: Kinect. Retrieved July 5, 2012, from http://www.eurogamer.net/articles/digitalfoundry-kinect-tech-analysis.

Lienhart, R., & Maydt, J. (2002). An extended set of Haar-like features for rapid object detection. Proceedings from ICIP 2002: *The 2002 International Conference on Image Processing*, 900-903.

Lowe, D. (1999). Object Recognition from Local Scale-Invariant Features. Proceedings from ICCV 1999: *The 7th International Conference on Computer Vision*, 1150-1157. Corfu, Greece.

Mackay, W., Fayard, A., Frobert, L., & Medini, L. (1998). Reinventing the Familiar: Exploring an Augmented Reality Design Space for Air Traffic Control. Proceedings from CHI 1998: *SIGCHI Conference on Human Factors in Computing Systems 1998*, 558-565.

Magni, S. (2012, June 23). LG SmartTVs, Adobe AIR 3.0 and App Test. Retrieved September 25, 2012, from http://www.noriste.com/lg-smarttvs-adobe-air-3-0-and-app-test/.

Marr, D., & Nishihara, H. (1978). Representation and recognition of the spatial organization of three-dimensional shapes. *Proceedings of the Royal Society of London B, 200*, 269–294.

McDowell, P. (2008, April 29). Delta3D and Open Source Software. Presentation at GameTech 2008: *Defense GameTech Users Conference 2008*.

McDowell, P., Darken, R., Sullivan, J., & Johnson, E. (2006, July). *Journal of Defense Modeling & Simulation, 3*(3), 143-154.

Messom, C., & Barczak, A. (2006). Fast and Efficient Rotated Haar-like Features Using Rotated Integral Images. Proceedings from ACRA 2006: *The Australasian Conference on Robotics and Automation 2006*, 1–6.

Microsoft. ActiveX Controls. Retrieved February 10, 2012, from http://msdn.microsoft.com/en-us/library/aa751968%28v=vs.85%29.aspx.

Microsoft. D3DXMatrixPerspectiveFovLH function. Retrieved April 8, 2012, from http://msdn.microsoft.com/en-us/library/windows/desktop/bb205350%28v=vs.85%29.aspx.

Microsoft. Silverlight. Retrieved February 1, 2012, from http://msdn.microsoft.com/en-us/library/cc838158%28v=vs.95%29.aspx.

Milgram, P., Takemura, H., Utsumi, A., & Kishino, F. (1994). Augmented Reality: A class of displays on the reality-virtuality continuum. *SPIE Vol. 2351: Telemanipulator and Telepresence Technologies*, 282-292.

Mizell, D. (2001). Boeing's Wire Bundle Assembly Project. In W. Barfield & T. Caudell (Eds.), *Fundamentals of wearable computers and augmented reality* (457-468). Mahwah, NJ: Lawrence Erlbaum Associates.

Mozilla. External resources for plugin creation. Retrieved February 10, 2012, from https://developer.mozilla.org/en/Plugins/External_resources_for_plugin_creation.

Mozilla. Gecko Plugin API Reference. Retrieved February 10, 2012, from https://developer.mozilla.org/en/Gecko_Plugin_API_Reference.

Mulloni, A. (2007). *A collaborative and location-aware application based on augmented reality for mobile devices*. (Unpublished master's thesis). University of Udine.

Nintendo. Nintendo 3DS - AR Cards at Nintendo. Retrieved March 11, 2012, from http://www.nintendo.com/3ds/ar-cards.

Novell. Moonlight. Retrieved February 1, 2012, from http://www.mono-project.com/Moonlight.

NyARToolkit. Welcome to NyARToolkit.EN. Retrieved February 15, 2012, from http://nyatla.jp/nyartoolkit/wp/?page_id=198.

Object Management Group. Unified Modeling Language. Retrieved May 31, 2012, from http://uml.org/.

Oda, O., MacAllister, C., & Feiner, S. (2012, February 2). Goblin XNA User Manual. Columbia University.

OpenCV. OpenCV Change Logs. Retrieved February 12, 2012, from http://opencv.willowgarage.com/wiki/OpenCV%20Change%20Logs.

OpenGL. gluPerspective. Retrieved April 8, 2012, from http://www.opengl.org/sdk/docs/man/xhtml/gluPerspective.xml.

Opera. An introduction to WebGL. Retrieved February 10, 2012, from http://dev.opera.com/articles/view/an-introduction-to-webgl/.

Oracle. Lesson: Java Applets. Retrieved February 2, 2012, from http://docs.oracle.com/javase/tutorial/deployment/applet/index.html.

Panda3D. Documentation. Retrieved February 1, 2012, from http://www.panda3d.org/documentation.php.

Papageorgiou, C., Oren, M., & Poggio, T. (1998). A General Framework for Object Detection. Proceedings from ICCV 1998: *The 6th International Conference on Computer Vision*, 555-562. Bombay, India.

Pindar. (1830). Seventh Olympic Ode. In C. Wheelwright (Trans.), *Pindar* (pp. 36-43). London, UK: Henry Colburn and Richard Bentley. (Original work written c. 464 BC).

Pliny the Elder. (1857). *The Natural History of Pliny.* (J. Bostcock and H. Riley, Trans.). London, UK: H.G. Bohn. (Original work written c. 77 AD).

Poelman, R., Akman, O, & Lukosch, S. (2012). As if Being There: Mediated Reality for Crime Scene Investigation. Proceedings from CSCW 2012: *The 2012 ACM Conference for Computer Supported Collaborative Work*. Seattle, WA.

Poggio, T., & Edelman, S. (1990.) A network that learns to recognize three-dimensional objects. *Nature, 343*, 263 – 266.

Qualcomm. Augmented Reality (Vuforia™). Retrieved February 13, 2012, from https://developer.qualcomm.com/develop/mobile-technologies/augmented-reality.

Reitinger, B., Werlberger, P., Bornik, A., Beichel, R., & Schmalstieg, D. (2005). Spatial Measurements for Medical Augmented Reality. Proceedings from ISMAR 2005: *International Symposium on Mixed and Augmented Reality 2005*.

Rekimoto, J. (1996). Augmented Reality Using the 2D Matrix Code. Proceedings from WISS 1996: *Workshop on Interactive Systems and Software 1996*.

Resnick, M., Berg, R., & Eisenberg, M. (2000). Beyond Black Boxes: Bringing Transparency and Aesthetics Back to Scientific Investigation. *Journal of the Learning Sciences, 9*(1), 7-30.

Richardson, N. (a.k.a. "Richy2k"). (2009, October 2, 10:21 a.m.). Re: Realistic Wii Polygon Counts. Message posted to http://www.gamedev.net/topic/549131-realistic-wii-polygon-counts/.

Rogers, Y. (2006). Moving on from Weiser's Vision of Calm Computing: Engaging UbiComp Experiences. Proceedings from UbiComp 2006: *The 8th International Conference of Ubiquitous Computing*.

Ronzani, D. (2007). The Battle of Concepts: Ubiquitous Computing, Pervasive Computing and Ambient Intelligence in Mass Media. *Ubiquitous Computing and Communication Journal, 4*(2), 9-19.

Rooney, C. (2011). UnityFlashCam. Retrieved March 18, 2012, from https://github.com/rooch84/UnityFlashCam.

Rose, D. (a.k.a. "drwr"). (2011, September 29). Triple your frame rate? Retrieved January 23, 2012, from http://www.panda3d.org/blog/?p=206.

Safire, W. (2009, July 23). The Cold War's Hot Kitchen. *The New York Times*. Retrieved April 7, 2012, from http://www.nytimes.com/2009/07/24/opinion/24safire.html?_r=1&pagewanted=all.

Samsung. Samsung SUR40 for Microsoft® Surface®. Retrieved April 6, 2012, from http://www.samsunglfd.com/product/feature.do?modelCd=SUR40.

Samsung. (2011, January 6). SAMSUNG And Adobe Bring Adobe Air [*sic*] To Smart TVS [*sic*]. Retrieved October 1, 2012.

Shachtman, N. (2007, March 21). Pentagon's PCs Bend to Your Brain. *Wired*. Retrieved April 27, 2012, from http://www.wired.com/dangerroom/2007/03/the_us_military/.

Schiffman, H. (2001). *Sensation and Perception: An Integrated Approach*. New York, NY: John Wiley and Sons.

Schilit, B., Adams, N., Gold, R., Tso, M., & Want, R. (1993). The PARCTAB Mobile Computing System. Proceedings from WWOS-IV: *The Fourth Workshop On Workstation Operating Systems*.

Schmalstieg, D., & Wagner, D. (2005). A Handheld Augmented Reality Museum Guide. Proceedings from ML 2005: *IADIS International Conference on Mobile Learning 2005*, 34-39.

Schmalstieg, D., Fuhrmann, A., Szalavári, G., Encarnação, L., Gervautz, M., & Purgathofer, W. (2002). The Studierstube Augmented Reality Project. *Presence: Teleoperators and Virtual Environments, 11*(1), 33-54.

Shepherd, O. (a.k.a. "cranberryzero"). (2011, May 2). Nintendo 3DS augmented reality tattoo is awesome, real. Retrieved March 9, 2012, from http://www.iheartchaos.com/post/5134010603/nintendo-3ds-augmented-reality-tattoo-is-awesome-real.

ShiVa3D. Documentation. Retrieved February 1, 2012, from http://www.stonetrip.com/developer/doc/.

Sociodox. The Miner. Retrieved August 3, 2012, from http://www.sociodox.com/theminer/.

Socolofsky, E. FLARManager: Augmented Reality in Flash. Retrieved August 13, 2012, from http://words.transmote.com/wp/flarmanager/.

StatOwl.com. Statistical analysis and market research of Internet usage trends. Retrieved January 28, 2012, from http://www.statowl.com/.

String Labs. String™ Augmented Reality. Retrieved February 27, 2012, from http://www.poweredbystring.com/.

Studierstube. Availability of Augmented Reality Software. Retrieved February 15, 2012, from http://studierstube.icg.tugraz.at/availability.

Sugano, H., & Miyamoto, R. (2010). Highly optimized implementation of OpenCV for the Cell Broadband Engine. *Computer Vision and Image Understanding, 114*, 1273–1281.

Tarr, J. & Bülthoff, H. (1998). Image-based object recognition in man, monkey and machine. *Cognition, 67*, 1-20.

Tastenkunst. Beyond Reality Face. Retrieved July 12, 2012, from http://www.beyond-reality-face.com/.

Tennenhouse, D. (2000). Proactive Computing. *Communications of the ACM, 43*(5), 43-50.

ThinkAnApp. (2010). thinkanapp (thinkanapp) on Twitter. Retrieved March 9, 2012, from https://twitter.com/#!/thinkanapp.

Total Immersion. Augmented Reality Software and Solutions by Total Immersion | Augmenting Your Reality. Retrieved February 28, 2012, from http://www.t-immersion.com/.

Ulloa, C. (2007, June 14). Need for speed. Retrieved March 18, 2012, from http://blog.papervision3d.org/2007/06/14/need-for-speed/.

Ulloa, C. (2007, July 7). Papervision3D Public Beta. Retrieved March 18, 2012, from http://blog.papervision3d.org/2007/07/07/papervision3d-public-beta/.

Ulloa, C. (2009, October 13). Papervision3D is Shifting Gears. Retrieved March 18, 2012, from http://blog.papervision3d.org/2009/10/13/papervision3d-is-shifting-gears/.

United States Army Air Forces. (c. 1944). Aircrewman's Gunnery Manual. Reprint retrieved March 18, 2012, from http://www.liberatorcrew.com/Manuals/AGM.htm.

Unity. Documentation. Retrieved February 1, 2012, from http://unity3d.com/support/documentation/.

Unity. Fast Facts. Retrieved July 10, 2012, from http://unity3d.com/company/fast-facts.

Unity. (2012, April 9). Unity Reaches One Million Registered Developers. Retrieved July 10, 2012, from http://www.marketwire.com/press-release/unity-reaches-one-million-registered-developers-1641486.htm.

Unity. Web Player Hardware Statistics - 2012 Q2. Retrieved July 10, 2012, from http://unity3d.com/webplayer/hwstats/pages/web-2012Q2.html.

Uro, T. (2008, May 20). Adobe Pixel Bender in Flash Player 10. Retrieved March 29, 2012, from http://www.kaourantin.net/2008/05/adobe-pixel-bender-in-flash-player-10.html.

VGChartz. Platform Totals. Retrieved February 29, 2012, from http://www.vgchartz.com/analysis/platform_totals/.

Viola, P., & Jones, M. (2001). Rapid Object Detection using a Boosted Cascade of Simple Features. Proceedings from CVPR 2001: *2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 511-518.

VTT. Alvar Technical. Retrieved November 23, 2012, from http://virtual.vtt.fi/virtual/proj2/multimedia/alvar/technical.html.

Wagner, D. & Schmalstieg, D. (2003). First Steps Towards Handheld Augmented Reality. Proceedings from ISWC 2003: *The 7th International Conference on Wearable Computers*.

Wagner, D. & Schmalstieg, D. (2007). ARToolKitPlus for Pose Tracking on Mobile Devices. Proceedings from CVWW 2007: *Computer Vision Winter Workshop 2007*.

Weisner, M. (1996, March 17). Ubiquitous Computing. Retrieved April 2, 2012, from http://sandbox.xerox.com/ubicomp/.

Weisner, M., & Brown, J. (1996, October 5). The Coming Age of Calm Technology. Retrieved April 4, 2012, from http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm.

West, M. (2008, July 16). Measuring Responsiveness in Video Games. Retrieved July 11, 2012, from http://www.gamasutra.com/view/feature/3725/measuring_responsiveness_in_video_.php.

Wojtczyk, M. (2011, January 17). Creating Depth Images with the Kinect Sensor. Retrieved March 11, 2012, from http://www.bing.com/images/search?q=kinect+depth+image&view=detail&id=B3ED8D7E8B010D08FAFE5F624E5515EDA0AA2567&first=0&FORM=IDFRIR.

Yu, Q., Cheng, H., Cheng, W., & Zhou, X. (2003). Interactive Open Architecture Computer Vision. Proceedings from ICTAI 2003: *The 15th IEEE International Conference on Tools with Artificial Intelligence*.