A SCRIPT BASED MODULAR GAME ENGINE FRAMEWORK FOR AUGMENTED REALITY APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUHAMMED FURKAN KURU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2009

Approval of the thesis

"**A SCRIPT BASED MODULAR GAME ENGINE FRAMEWORK FOR AUGMENTED REALITY APPLICATIONS**"

submitted by **Muhammed Furkan Kuru** in partial fullfillment of the requirements for the degree of **Master of Science in Computer Engineering, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit _____
Head of Department, **Computer Engineering**

Assist. Prof. Dr. Tolga Can _____
Supervisor, **Computer Engineering**

Examining Committee Members:

Assoc. Prof. Dr. Kürşat Çağıltay _____
Computer Education and Instructional Technology, METU

Assist. Prof. Dr. Tolga Can _____
Computer Engineering, METU

Assoc. Prof. Dr. Veysi İşler _____
Computer Engineering, METU

Dr. Çağatay Undeğer _____
Computer Science, BILKENT University

Assist. Prof. Dr. Tolga Çapın _____
Computer Science, BILKENT University

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Muhammed Furkan Kuru

Signature :

ABSTRACT

A SCRIPT BASED MODULAR GAME ENGINE FRAMEWORK FOR AUGMENTED REALITY APPLICATIONS

Kuru, Muhammed Furkan

M.Sc., Department of Computer Engineering

Supervisor: Assist. Prof. Dr. Tolga Can

September 2009, 71 pages

Augmented Reality (AR) is a technology which blends virtual and real worlds. The technology has various potential application domains such as broadcasting, architecture, manufacturing, and entertainment. As the tempting developments in AR technology continues, the solutions for rapid creation of AR applications become crucial. This thesis presents an AR application development framework with scripting capability as a solution for rapid application development and rapid prototyping in AR. The proposed AR framework shares several components with game engines. Thus, the framework is designed as an extension of a game engine. The components of the framework are designed to be changable in a plug-in system. The proposed framework provides the developers with the ability of agile coding through the scripting language. Our solution embeds a dynamic scripting programming language (Python) in a strictly typed static programming language (C++) in order to achieve both agility and performance. The communication between the AR framework components and the scripting programming language is established through a messaging mechanism.

Keywords: augmented reality, virtual reality, scripting, game engine

ÖZ

ARTIRILMIŞ GERÇEKLİK UYGULAMALARI İÇİN BETİK DİLİ TABANLI BİR UYGULAMA GELİŞTİRME ÇERÇEVESİ

Kuru, Muhammed Furkan

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Yard. Doç. Dr. Tolga Can

Eylül 2009, 71 sayfa

Artırılmış Gerçeklik (AG) sanal ve gerçek dünyayı birleştiren bir teknolojidir. Bu teknolojinin yayımcılık, mimari, üretim ve eğlence gibi bir çok potansiyel uygulanma alanı vardır. AG teknolojisindeki gelişmeler devam ederken, bu teknoloji üzerinde hızlı ve kolay uygulama geliştirme çözümlerinin önemi artmaktadır. Bu tezde betik dil destekli bir AG uygulama geliştirme çerçevesi sunulmaktadır. Bu AG çerçevesindeki betik dilinin çevikliğiyle birlikte AG bileşenleri kullanılarak uygulama geliştirilebilir. AG bileşenleri oyun motorlarındaki bileşenlere benzer olduğundan AG çerçevesi bir oyun motorunun genişletilmiş şekli olarak tasarlandı. Bununla birlikte, AG çerçevesinin bileşenleri değiştirilebilir bir şekilde tasarlanmıştır. Dinamik bir betik programlama dili (Python) statik tip bağlamalı bir dil (C++) içerisine gömülerek hem çeviklik hem de performans hedeflenmiştir. AG bileşenleri ile betik programlama dili arasındaki iletişim bir mesajlaşma yöntemiyle oluşturulmuştur.

Anahtar Kelimeler: artırılmış gerçeklik, sanal gerçeklik, betik dil, oyun motoru

ACKNOWLEDGMENTS

First and foremost, I want to thank my supervisor Asst. Prof. Dr. Tolga Can for all his support and guidance throughout this study. His motivating comments and suggestions during this study were invaluable.

I would also like to thank jury members for their comments on this thesis.

I deeply thank to my partners and close friends Mustafa Ayan, Münir Ercan, and Gürkan Caner Birer, the members of Rotasoft company, for their work related to this study. In addition, I thank to them for their help, support, understanding, and patience particularly during the thesis writing period.

Many thanks go to my friend Gökhan Çapar for his help and unique calming comments. Playing table tennis with him during the one-hour breaks from this study was enjoyable.

I want to thank to Doğacan Güney, my dear friend, for his useful advices and quick but clear answers to my questions regarding this work.

I would like to thank to my cousin Tacettin Kuru and my friend Fatih Yavuz for their encouragement and motivation in the time of writing this thesis.

I wish to specially thank to my friend Başak Demirel. This work may not have been finished on time without his daily check on my study progress.

I would like to thank TÜBİTAK (Turkish Scientific and Technical Research Council) for the graduate scholarship during this study.

Finally, I am very grateful to my beloved family for their unconditional support and love in all those years.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Augmented Reality (AR) is an interactive visualization technology in which virtual and real worlds are combined together to create a visually enhanced environment. AR differs from Virtual Reality (VR) in the sense that the environment created in VR is entirely virtual whereas AR mixes the real and virtual environments. In AR, the user's or real objects' position and orientation are tracked and virtual objects are superimposed into the user's surrounding environment. The technology uses a composition of techniques from computer vision and computer graphics fields.

AR can be applied to a wide range of domains such as broadcasting, computer games, medical training, architecture, entertainment, and manufacturing [1, 2]. In addition, as the tempting and promising development in AR continues, the solutions for building AR systems become crucial in the software market.

1.1 Motivation and Problem Definition

AR systems add a new dimension to visualization and interaction techniques. However, the development of an AR system is not a trivial task. AR systems include too many components each of which requires comprehensive knowledge and programming skill. In addition, for specific solutions, different components need to be incorporated into the system. Moreover, since AR itself is a new technology, each of its components is under research and development, thus are likely to be replaced by improved and better components. But still, the overall architecture is the same for most AR applications and a common framework with a modular structure can be developed. It is crucial for a modular AR framework to support substitution of components with alternatives without affecting the application's overall structure. Even if it is not required to change the underlying components, the framework can be used for

rapid application development.

Generally, in AR systems two main capabilities are needed: 1. Visualization 2. Tracking. The visualization problem is the one that is solved during game or VR application development. From software development and design perspective, AR applications can be thought as an extension to computer games or game-like multimedia softwares since they share functional properties such as real-time 3-dimensional (3D) rendering, user input handling, sound playing, physics, events and timers. However, game development itself is also a tedious task as it requires the programmer to implement these functionalities. But, in order to simplify game development and favor software reusability, the idea of Game Engine (GE) frameworks have come out. The philosophy behind the Game Engine framework is that the common parts in game development should be extracted and reused. This is necessary for building reusable, tested software infrastructure containing middle-ware software as sub-components. There is no specific definition of GE but GEs are the main independent components of games. They should be independent of the actual game project. So, GEs do not carry any game specific code [34]. This is achieved by separating the game content and capabilities to be used. A game engine just defines the application's potential abilities and different games can be created using the same game engine with different CG materials and game logic. Even though, the name includes the word "game", the usage of game engines is not restricted to computer games domain. It can be used for various kinds of multimedia projects, simulations, and real-time visualizations. Incorporating a game engine into an AR system reduces the complexity by creating a level of abstraction for the problems similar to game development involved in AR.

The tracking problem can be solved in various ways depending on the application type (e.g. indoor, or outdoor) and specifications. There have been studies in tracking techniques and different kinds of solutions exist such as computer vision based, inertial tracking, and GPS [35]. By providing a common interface, existing tracking solutions can be incorporated into an AR system.

Establishing an extended game engine with tracking capability enables us to develop various kinds of applications using the framework by favoring code reuse. Moreover, embedding a readable simple scripting language let the developers save time by facilitating the development of applications.

In this thesis, we propose a modular game engine framework for AR applications that is powered by scripting ability which enables rapid prototyping and application development. Firstly, for the framework development, an extensible game engine is to be designed as the

baseline. The game engine will handle the real-time and 3D rendering issues. Moreover, it may bring additional features commonly needed in games such as sound playing, graphical user interface, physics, and network support. 3D tracking, live image data acquisition, and other requirements will be satisfied through the extension of the game engine. In the framework, the tracking module will be plugged to the system as an input module providing user's position and orientation. The framework is to be powered by the scripting capability in Python, which provides us with fast development through less coding with simple syntax.

1.2 Related Work

There are a number of frameworks that encapsulate algorithms, implementation, and hardware communication. However, the software engineering concepts for system-wide AR development are not yet widely used in the AR community and a complete framework ready to develop AR applications is missing.

ARToolkit [16] is an example of a simple and minimal AR framework. It has vision-based tracking ability using a set of markers with pattern matching. The virtual objects are rendered as simple graphics using OpenGL for visual augmentation. In fact, ARToolkit is a software library rather than a framework. In a broad sense, it just provides the developer with an application programming interface (API) that enables the developer to call a set of library functions. However, it does not provide a pluggable interface for tracking and rendering. Thus, if a developer wants to change the underlying OpenGL renderer, the problem arises. Despite its simplicity, the library has been widely adopted and various AR applications have been developed with ARToolkit because of its popularity. ARToolkit is far from a scalable framework but the computer vision based tracking part of ARToolkit library, as any other tracking library, can be plugged into our proposed AR system by implementing a layer as an interface.

The DWARF project [3] proposed a high-level modular design concept that differs from traditional AR software designs. The framework in the DWARF project includes a set of services. Each service is a separate component that provides *abilities* to other components by revealing its *needs*. The connection between the services are established by matching one service's need to another service's ability. For example, trackers gather the pose information. Providing position and orientation data can be defined as their ability. A renderer needs position data in order to align virtual objects properly. So a renderer component and a tracker component, for example a 3D-renderer and a vision-based tracker, is matched

for augmentation. This system provides us with a very flexible architecture and enables us to replace any module with another one having the same interface, namely needs and abilities. This type of architecture abstracts the inter-module communication and module dependency. Thus changing a module's implementation in software or hardware does not affect the remaining system as long as the need and ability pins are not changed. We tried to adopt the modularity approach of DWARF in our framework and extend it with scripting capability. Without the scripting capability, the DWARF lacks support for rapid development of application behaviour which frees the developer from the burden of dealing with the underlying modules and low-level programming issues.

The Studierstube project [25] is a software framework that provides the foundation and basic software design layers. The goal of this framework is to support the technical requirements of AR applications. It has a reusable architecture and the underlying tracking system can be configured. It combines 3D tracking, rendering, and output to AR and VR devices. It has user management functions and distributed applications. This framework is used for developing mobile, collaborative, and ubiquitous AR applications. However, this framework also lacks scripting support which is crucial for rapid application development. Because, the implementation of the application logic in a strictly typed static language rather than a dynamic scripting language, increases the application development time.

1.2.1 Scripting Languages

There are various scripting languages each of which has its own advantages and disadvantages. For the candidate scripting languages in our proposed AR framework, integrability is the most critical feature. Thus, we examine the scripting languages which were designed to be embeddable in applications.

Lua [23] is a lightweight embeddable scripting language which provides a procedural syntax similar to Pascal language. It is fast and easily integrable to C language. Its syntax is simpler than C. In order to embed Lua in other applications, it provides an easy to use C API. Lua is widely used in contemporary computer games.

Perl [22] is another dynamic scripting language that is mostly used in web programming and system administration. It focuses on application-oriented tasks. However, its syntax is not highly readable.

Embeddable Common Lisp [17] is a functional programming language which is an implementation of Common Lisp that can be embedded into C programming language. However, it lacks popularity and documentation, particularly in game scripting.

Tcl [28] is designed to be an extension language for rapid prototyping, testing and graphical user interfaces. It has a flexible syntax which you can incorporate new control structures. However, Tcl holds all the data as strings which causes a performance drop due to data types' conversion to and from strings.

Visual Basic [31] is a beginner programming language with easy syntax which is proprietary to Microsoft. It is mostly used for developing Windows graphical user interface applications. It can be used by applications through component object model inter-process communication.

Python [18] is a general purpose dynamic programming language which can be used as a glue language as well. It provides high code readability through clear syntax. It is widely used as a scripting language in various types of applications, e.g. web, graphics, simulation, and computer games. Its has a broad standard library.

All of these scripting language alternatives are suitable for being embedded in our proposed AR framework. However, Tcl, Lua, and Python are one step ahead of others because of their easy integration with C++. We have chosen Python since it has more beginner documentation and more readable syntax like pseudocode. Moreover, Python has more extension libraries than Lua and Tcl.

1.3 Contributions

Our contribution in this thesis is a proposal of a scripting-based AR framework design and implementation. In this framework, we propose a scripting ability to program AR applications. It promises a reduction in the complexity and line of code needed to create AR applications. The focus is reusability and rapid development during the design and development.

Flexibility and extensibility

The system is designed to favor a flexible structure through the pluggable module interface. According to application needs, proper modules are plugged to the main core. A module can be defined as a component that is specialized to execute a specific job with an interface. The system can be extended by implementing new modules. For example, besides the basic modules such as the renderer and image processing modules, a sound player module can be attached to the system in order to increase augmentation by sound effects. Moreover, the ability to substitute a module with a new module is required to follow the fast developments in the area of underlying technologies. A different tracking method may be

incorporated to the system by just substituting the tracker module. Therefore, changes to the tracking module should be independent of the remaining parts. The modular structure just defines the interfaces and any two modules having the same interface can be replaced with each other.

Simple and rapid development

The level of abstraction in the framework is sufficiently high to support rapid development of AR applications. A large number of existing frameworks are commonly based on strictly typed languages that require compilation. Therefore they oppose frequent changes and it increases the development time. On the contrary, scripting languages allow rapid development by avoiding heavy programming task and strict style.

The editing and testing phase is pretty slow while writing in a static and strictly typed language such as C++. C++ is designed with an emphasis on run-time performance and any feature which has possible performance drawback is excluded from the language [27]. However, a dynamic scripting language does not require compiling and linking. It reduces development time by faster coding. The trade-off between run-time performance and development time is an important issue, especially in the course of real-time applications. Because of the critical performance issues, the system module's core functionalities can be implemented in C++ and an interface for the use of the scripting language can be defined. We choose this approach in our proposed framework.

Besides the abstraction provided by the modular structure, scripting brings another abstraction layer that separates the application specific data and the underlying implementation. Using a scripting language is an easy way to *glue* the modules together. In addition, a high level of reusability is achieved by the use of scripts. The application specific logic and data can be defined by written scripts and the capabilities are reused behind the scene. Since, scripting languages generally have a simple syntax and usage, learning and using it does not require high programming skills. Additionally, the script-writers do not need to know anything about the background technology, they have to focus merely on the data and logics. In this thesis, Python programming language is used as the scripting medium.

In this thesis, in order to demonstrate the effectiveness of the proposed AR framework, we develop a proof of concept AR application. We show that by taking advantage of the scripting capability, AR applications can be developed rapidly.

1.4 Organization of the Thesis

In Chapter 2, the background on AR technology is described. Chapter 3 summarizes the overall structure of the proposed AR framework and its components. In Chapter 4, the design and implementation details of the framework's scripting power are presented. Chapter 5 demonstrates the development of an AR application using our proposed framework. Finally, we conclude and present future work in Chapter 6.

CHAPTER 2

BACKGROUND ON AUGMENTED REALITY

2.1 Introduction

Computer-generated imageries (CGI) have been widely used in games, movies, commercials, and television broadcasting. Most of the time, CGIs are not used singly in pure form but rather they are combined with real world visuals. However, merging 3-dimensional (3D) virtual objects into the real world visuals is a tedious and time-consuming work. 3D and 2D graphics artists need hours for proper alignment of virtual objects into videos frame by frame. The efforts pay off as the composite end product becomes impressive and realistic. This way a seamless interaction can be achieved between virtual and real objects. But, still the user interaction from the viewer's point of view is missing. The user can not manipulate the virtual objects and can not change the viewing angle since the video is an offline material

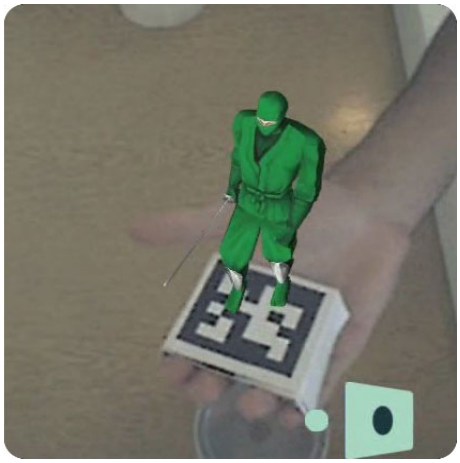

Figure 2.1: Example of Augmented Reality

and static. Thus, the result of merging process is far from interaction. A real-time merger is required for viewing both real and virtual objects in order to experience interaction. Augmented Reality (AR) is a new technology to present a solution for blending virtual and real world with real-time user interaction. An example of AR image is shown in Figure 2.1. There is a 3D CG ninja model that is aligned properly into the real scene, providing the user with a level of interaction.

2.2 Augmented Reality versus Virtual Reality

Augmented Reality is similar to Virtual Reality (VR) in the way of using CG materials as content to generate virtual objects. In virtual reality, the user's scene is completely computer generated and VR attempts to make users perceive virtual environment as real. In order to reach realistic scenes, VR requires high quality models for building the environment and powerful hardware for achieving smooth rendering performance in real time. On the contrary, Augmented Reality does not need to generate all the environment synthetically. AR acquires user's surrounding visual data from capture devices and video streams in order to use the real visuals as background environment. Then, the virtual objects are superimposed onto video frame sequence. A visual from real environment and a synthetic virtual environment are blended together to form an augmented visual.

In comparison to VR, the rendered scene content in AR is minimized and this reduces the requirement of high computing power for rendering. Another contrast between AR and VR is that Augmented Reality needs to *register* virtual objects precisely with real objects, whereas Virtual Reality does not have any problem related to registration since every object's position and orientation is defined and rendered accordingly. *Registration* refers to proper alignment and superimposition of virtual and real objects. Except the registration problem and the quantitative difference in CG rendering, AR and VR share common problems to be handled in order to create applications of their type.

2.3 Augmented Reality Applicable Domains

The concept of augmenting real world by virtual objects is a promising way of information visualization. It can be applied to various domains by AR systems. Consider visiting a foreign touristic city, and not having any knowledge about the nearby historical places. Instead of a guide, an AR system can help you retrieve the related information, instruct

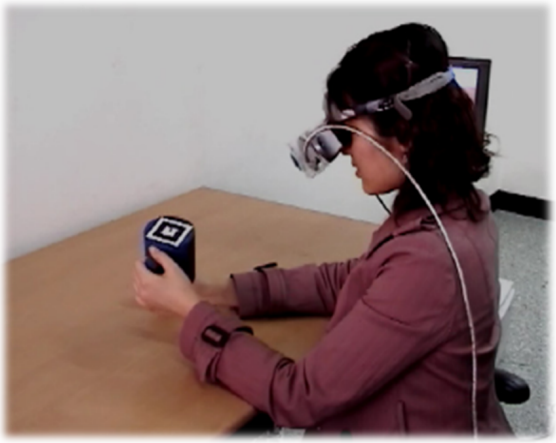

Figure 2.2: Example of indoor AR using Head Mounted Display

you to the right direction and augment the real world with audio-visuals. Head Mounted Display (HMD)s can be used for augmentation from your point of view. The signs in the real world can be translated to your language, audio and visual information can be acquired from internet. Your surrounding can be turned into a vivid environment. AR can be effective indoors as well. The user may interact with the 3D virtual objects that appear in front of him. An indoor HMD usage example is shown in Figure 2.2.

Another domain that AR can be applied is medical training. Students can learn about the surgery using an AR system that visualizes the invisible inner parts of human body. The surgery can be simulated using the AR system. Even the operations may be executed through the AR system's visual guidence.

During live broadcasting, especially in sports, 3D and 2D virtual advertisements can be blended into the scene as they are part of the game field or grandstands. Statistics and information about the game or players can be displayed over the live stream.

AR can be used in manufacturing and maintenance of machineries as well. The usage guidelines of tools in the industry could be visualized using AR. For example, 3D models and arrows can be drawn to instruct the user how to insert, put, remove tool's pieces. Informative text can be displayed throughout the guidence as well [1].

Computer games, as the most entertaining applications, can be enriched by AR. AR-Quake [29] is an example of outdoor first-person-shooter AR game. It gives opportunity to play legendary game Quake outside with global positioning system (GPS) data for defining the user's position and compass for the orientation.

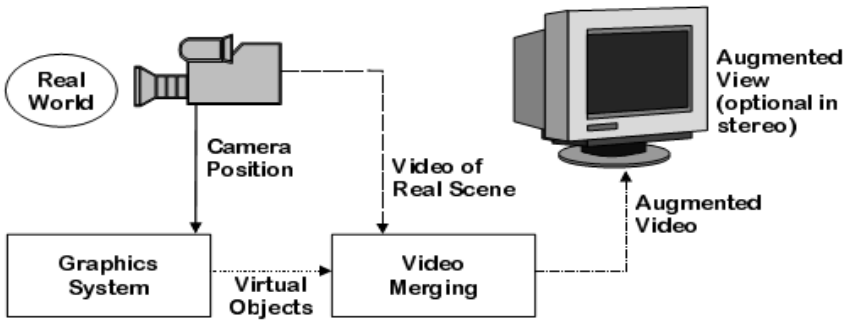

Figure 2.3: Monitor-Based AR Display (Courtesy of J.Vallino) [30]

2.4 Augmented Reality Environment

A typical AR environment consists of a set of hardwares and AR software. The overall system requires technology to get and process information, then display images of augmented view accordingly. The devices used in AR system directly affects the quality of the user's experience of feeling immersed in the mixed environment and the seamless interaction between real and virtual. Generally, a camera, a display device, and computers are used for basic AR setup environment.

2.4.1 Camera, Display, and Computers

Cameras are the devices that enable us to acquire the video images from the real world. The video images can be used as a source for gathering information about the user's context in real world. They can be rendered as background environment. Different types of display devices can be used for setting up a visualization medium.

Monitor-Based Systems

Monitor-based systems are the simplest and most common approach for AR systems. In this system a monitor is used to display the generated augmented scene. The real-time video stream of the real environment is gathered via a camera frame by frame continuously. The frames are fed to the AR system. Then the camera's pose (3D coordinates and orientation) in the real world can be calculated using vision-based approaches. The system is diagrammed in Figure 2.3. This system requires only a personal computer (PC) and any kind of camera plugged to the PC. Setting up this kind of system is affordable and simple. Low-cost consumer products such as web cameras having USB or Firewire interface or analog

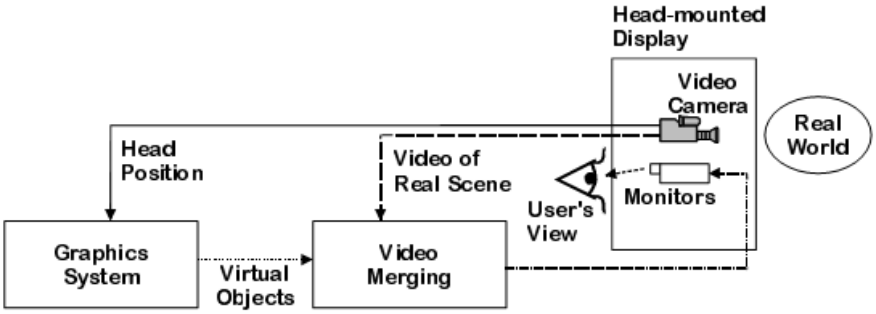

Figure 2.4: Video see-through HMD AR system (Courtesy of J. Vallino) [30]

cameras through a capture card can be connected to the PC. The image quality is limited by the camera's properties such as resolution and frames per second. The disadvantage of this system is that the user may have difficulty in sensing the mixed environment interaction from a display device. It achieves the real and virtual blending but the interaction level is not high enough because the user may feel like an outside viewer. The interaction level may be increased by placing the camera in a proper position to see user actions front and the screen displays what the camera captures. The images taken from camera should be flipped vertically to enable this effect. This way the user has feeling of a mirror [9] and see himself in action.

Video See-Through Systems

Head-mounted Displays (HMD) are widely used both in VR and AR to immerse the user completely in the computer generated scenery. The user wearing an HMD is able to see the images in front of himself. HMD is worn on the head and consists of two display screens for eyes. The augmented scene is fitted into the user's field of view increasing the sense of presence in the mixed scene. A scheme of video see-through system for AR is shown in Figure 2.4. The difference between this system and a VR HMD is the addition of a video camera attached to HMD. Using video see-through HMD system, the user does not see the the real world directly. The real world is rendered from the video frames and it may reduce the image quality depending on the camera used.

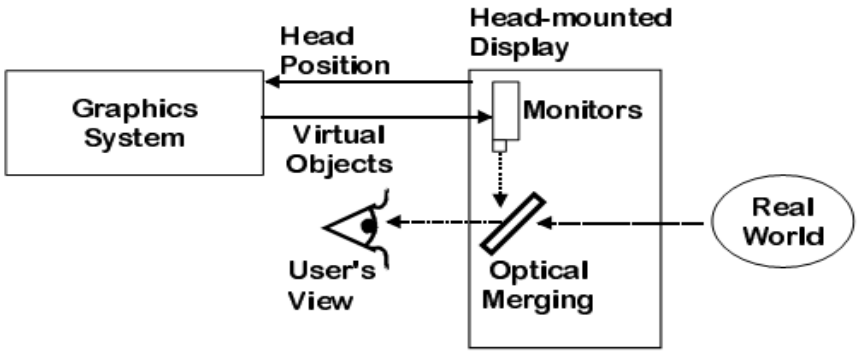

Figure 2.5: Optical see-through HMD AR system (Courtesy of J.Vallino) [30]

Optical See-Through Systems

Optical see-through systems provide us with a better HMD configuration for AR. Instead of drawing the real world with reduced quality, transparent displays like glasses are used to pass the light from environment to the user's eyes directly. Virtual objects are blended in these displays in the same way as video see-through displays. Figure 2.5 depicts this system. Since the real world is not rendered through a camera, HMDs used in this system do not have resolution limits. They do not even require cameras if computer-vision techniques are not used for tracking purposes. Inertial trackers on the HMD estimates the pose of the user's head. However, the quality of virtual images are still limited by the hardware capabilities. The mere disadvantage of this system is that the see-through displays are still not in the range of affordable costs for consumers.

2.4.2 User-Interaction Devices

In an interactive environment created by an AR system, the user interaction with the environment should be emphasized as much as visualization. Haptic input devices such as force-feedback gloves can be used to give the user feeling virtual objects as if they are in his hand. Another example for a tangible AR interface is shown in an application developed by Kato et al. [15]. In this example the user can select and manipulate virtual furniture in an AR living room design application. The motions of the paddle are mapped to gesture based commands, such as tilting the paddle to place a virtual model in the scene and hitting a model to delete it. An example image is shown in Figure 2.6. New interaction techniques have been researched to increase the interaction level [35].

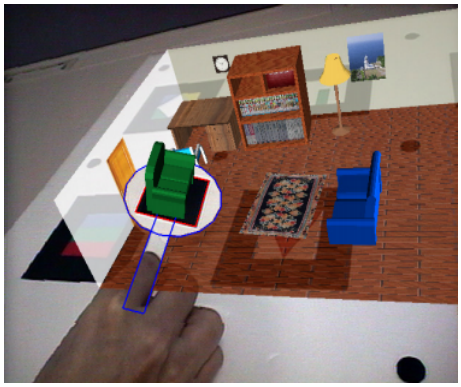

Figure 2.6: A real paddle is used to pick up, move, drop, and destroy virtual models (Courtesy of Hirokazu Kato)

2.5 Tracking Methods

It is crucial that the position and orientation of the user be tracked precisely for proper registration in AR. Registration can be defined as aligning virtual objects with the real world. Several approaches have been developed for tracking the user or an object in the real environment. They can be grouped into two: Sensor-Based Tracking and Computer Vision-Based Tracking.

2.5.1 Sensor Based Tracking

By sensor-based tracking techniques, the user or a real object can be tracked using inertial, magnetic, acoustic, and mechanical sensors. These types of sensors have both advantages and disadvantages. For example, magnetic sensors have a high update rate and are light-weighted, but they are noisy and can be distorted by any material containing metallic substance that disturbs the magnetic field [35]. Another example is inertial sensors. They are used in HMDs to track the head's motions. They contain gyroscopes and accelerometers. Gyroscopes measure rotation rate whereas accelerometers measure linear acceleration vectors with respect to the inertial reference frame. To eliminate the effect of gravity, the acceleration due to the gravity should be subtracted from the observed acceleration value. The gyroscope determines the relative orientation changes with respect to the reference frame. But the accumulation of signal and error may raise the problem of increasing orientation drift. A magnetic compass may be incorporated to compansate the accumulative errors but they are also subject to errors by ferrous materials [33]. In order to achieve accurate tracking vision based tracking can be incorporated into a sensor-based system in a hybrid-manner.

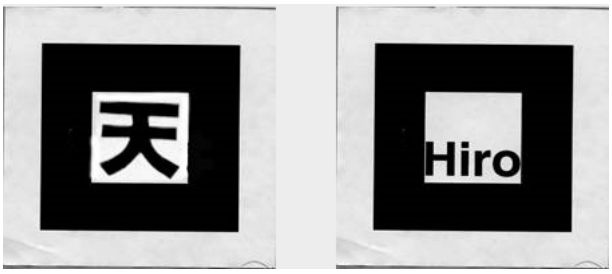

Figure 2.7: Sample ARToolkit markers (Courtesy of Hirokazu Kato)

2.5.2 Computer-Vision Based Tracking

Vision based tracking techniques use image processing methods to calculate the camera's position and orientation relative to the real world objects. This is the most active tracking research area in AR. In this technique, the video input provides the information about the camera's pose relative to the scene. The intrinsic parameters of the camera and the information in the video frames can be used to calculate the camera position and orientation. In video frames, features can be tracked to extract the scene information. Since finding strong features in video frames can be difficult, manual markers are placed in the real scene to aid the tracking. In this feature based method, a correspondance is found between the 2D image features of the markers and 3D world model coordinates. The camera's position and orientation is calculated using this correspondance. Using the image processing methods artificial markers can be tracked in real-time. The ARToolKit library can track black and white square markers with pattern matching [16]. In this library, every image frame captured from camera is thresholded into binary values. Then the black regions that can be fitted into a region by four lines are found by segmentation and edge detection techniques. These image regions are normalized and then checked against a pattern database, if a match is found, the region is marked as the identified region. Figure 2.7 shows two example markers used in ARToolkit. After marker identification, the position and orientation of the marker is calculated with respect to the camera. Then, the virtual objects linked to the identified marker can be aligned and blended in the video frame. The flow diagram of a marker based AR system is shown in Figure 2.8.

An alternative to the pattern matching method is the Reactivision's topological fiducial tracking [4]. In this method, instead of checking 2D patterns against a database, a topology of the image is extracted and the regions having the same topology of markers are marked as candidates. This method does not require line fitting or edge detection and it does not

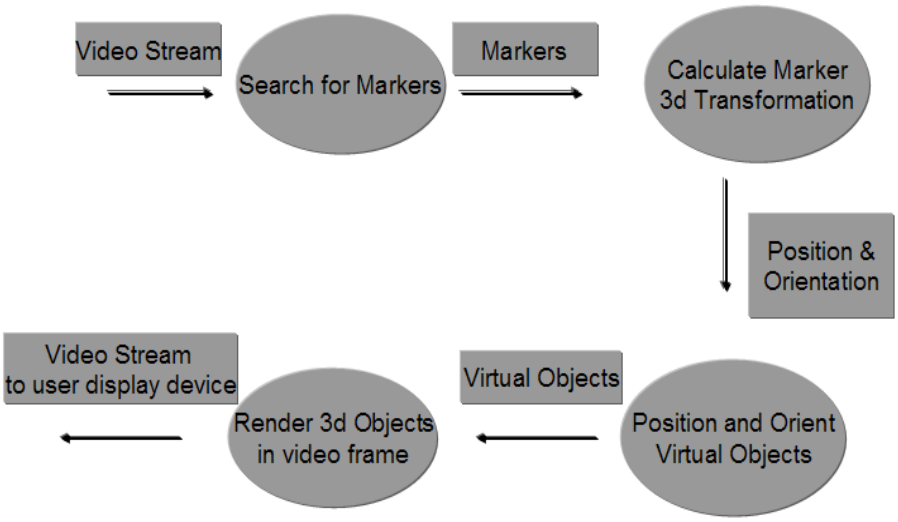

Figure 2.8: A marker based AR system

Figure 2.9: Sample Reactivision fiducials [4]

restrict the marker's shape to square but it can only estimate 2D positions and orientation lacking 3D pose estimation. Example fiducials are shown in Figure 2.9.

These types of marker-tracking techniques are simple and they can exhibit high performance. Having a simple set up with a PC and USB camera, an interactive environment with decent frame rates can be created. However, any occlusion of some part of the markers may lead the system lose tracking or match wrong markers. Ongoing researches in vision based tracking are focused on the robustness of these systems.

The markers are usually unnatural shapes which are placed in the environment to facilitate the tracking. Generally, they are colored black and white in order to maximize the contrast and minimize the effect of the light in the environment. This way a better thresholding can be applied without losing any part of the marker due to poor lighting conditions.

The regions in the marker shape can be identified by segmentation successfully. Moreover, they have features such as strong edges and corners. These all make tracking the markers easier. But in a large-scale environment, the use of markers is not feasible. For this reason, instead of using artificial fiducial markers, the natural features in a scene such as points, lines, edges, and textures can be used to calculate camera pose. After determining camera position and orientation from known visual features, the system can dynamically update the pose calculation by using natural features acquired consequently. In this way the system can provide us with robust tracking even when the original natural features are no longer in view. There are various natural feature tracking techniques which are applied to AR. In recent years, research on natural feature tracking has been highly active one in computer vision [35].

CHAPTER 3

AUGMENTED REALITY GAME ENGINE FRAMEWORK

In this chapter, our proposed framework and its components are described. Our AR framework is built upon a main core to which any functional component can be attached. The main core does not execute anything but updates every component in each cycle. The 3D render engine, the Script Manager, and Input-Output handler are the basic required components that are statically binded to the core. A general view of the framework can be seen in Figure 3.1. Graphical User Interface (GUI), Image Processor, Sound Library, and Video Player can be plugged to the main core depending on the application needs. Additional components, for example a Physics Component, can be incorporated into the AR system like these plug-ins. The core and static components are implemented in C++. However, plugins can be implemented in both C++ and Python programming languages.

3.1 Render Engine

Render engines are the most critical components of Game Engines. The render engine component is tightly connected to the main core and forms the basis of our framework. The term *rendering* can be used to define the operation of generating images from model data in the process of virtual visualization. The model data holds the visual information of 2D or 3D objects, that is the shape geometry and material properties such as texture, lighting, and shading. The generated output would be a 2D image that can be saved in a file or displayed on the monitor device immediately.

Depending on the scene complexity in terms of both object and light quantity and the techniques used for rendering objects, the rendering process may need extensive computing time. However, real-time applications such as computer games and in our case AR appli-

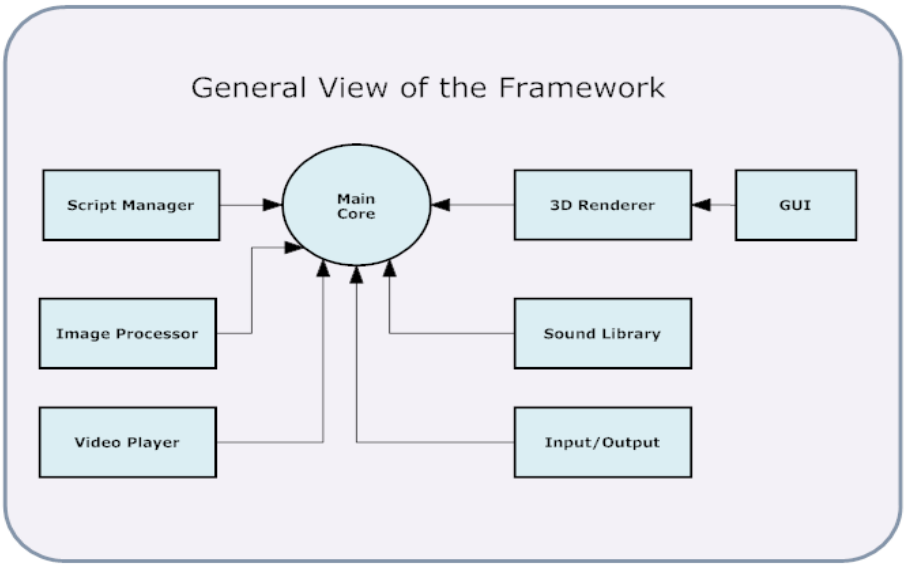

Figure 3.1: General view of the framework

cations need real-time rendering. The user should see the output immediately without any delay. Thus, the rendering process should be executed at a rate more than 24 frames per second which is the minimum rate for smoothness for the human eye. The video adapters with 3D hardware acceleration capability generates the required computing power to aid the system achieve acceptable frame drawing rates.

In the framework, the render engine is the part that is responsible for all the rendering process. 2D and 3D rendering is executed using the render engine as an interface between the framework and the video output hardware. Its basic abilities are:

- 2D and 3D computer graphics drawing: 2D graphics include digital images, text, and 2D geometric shapes. 2D graphics do not have any depth. The graphical user interface elements are examples of 2D graphics.

3D computer graphics are the visuals of 3D geometries. The model data of 3D objects are used for visualization. Rendering 3D graphics is performed by calculating the viewing angle, 3D model's position, and orientation in the virtual world coordinates. The final graphic output would be a 2D graphic that can be displayed on monitor devices.

- Lighting and Shading: Lighting is the crucial ability to create realistic images. If the building blocks of the geometries, namely the polygons, are filled with flat colors, the final shape would be sketch-like and far from realistic view. In order to increase photo realistic effects, the virtual light sources are defined and positioned in the environment and the objects in the scene are colored according to the light intensity. The light sources can

diffuse different colors. The amount of reflecting light of the object varies by the incoming light angle. The objects seem shaded as in the real world. They can reflect the light or seem transparent and shine. The specular light is the light that heads towards the user with a steep angle from shiny surfaces.

- **Texture Mapping**: The geometric shapes can be wrapped by texture images to increase reality. For example, in order to draw a 3D earth, a primitive sphere object is created and the polygons of the sphere are filled from a 2D image file of the earth.

- **Render to Texture**: This is a feature for creating picture in picture effects in the scene. Some part of the scene is rendered as a 2D image and that is used as a texture on a object in the scene. This feature is also essential for background video drawing in monitor and video see-through based augmented reality. The real world stream is loaded to the defined texture data space and then rendered accordingly.

The render engine has some additional high level abilities such as scene management to control and manage the virtual objects. It is also capable of drawing video stream data to the background for achieving augmented visual. The render engine basicly keeps geometry data of the scene and the camera properties. For every frame it calculates the drawing position and color of the entities in the scene and fills the framebuffer accordingly, finally creating a 2D image on the screen.

In order to create a 3D augmented scene, the virtual camera should be configured to match with the real camera in the real environment. The virtual objects drawn in the 3D virtual environment will seem as if they are part of the real world.

As the underlying 3D renderer, we choose open-source Object-oriented Rendering Engine (Ogre3D) [26].

3.1.1 Object-Oriented Rendering Engine (Ogre3D)

Ogre3D is one of the leading open source projects which is ranked in top 100 on sourceforge.net. It has all the features to satisfy the render engine requirements described above. Ogre3D is not a Game Engine; it is just a render engine. It has numerous capabilities such as polygon rendering (shader language support), geometry encapsulation, material system, object oriented scene management, and a robust plug-in system. It can run both on Direct3D and OpenGL graphic library APIs [13]. Ogre3D has a consistent documentation and active forum support which allow us to use it effectively and find solutions easily for the problems encountered.

From the design perspective, Ogre3D sits on the heart of our C++ side. Most of the

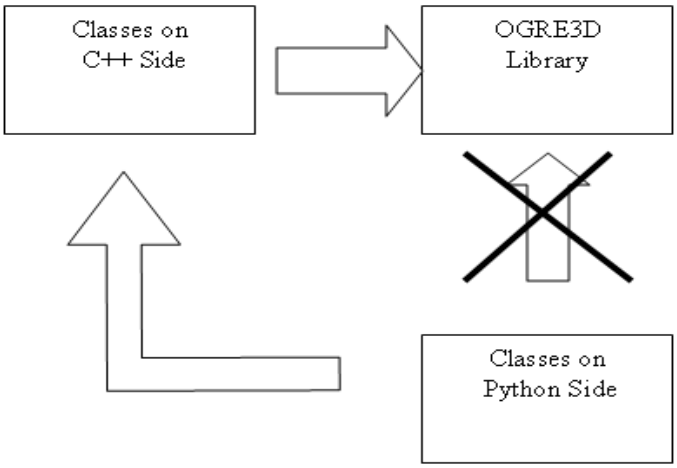

Figure 3.2: Scripting mechanisim for Render Engine function calls

code pieces invoke Ogre3D methods. However, the scripting power let us encapsulate the Ogre3D API. Through the Script Manager, the Python scripts call Ogre3D functions but the Python side does not know anything about Ogre3D. The orders or messages sent from the Python side, are independent of their corresponding implementation by the underlying render engine. The scripting part does not carry anything related to the render engine. Thus, the script writer does not need to know about Ogre3D. This mechanism is shown in Figure 3.2.

3.2 Input Library

In AR applications, acquiring inputs from the user is essential in order to enhance the level of user interaction. Apart from the conventional user input devices such as mice, keyboards, and joysticks, the haptic input devices such as gloves or accelerometers (e.g. Nintendo's Wii Remote [20]) can be plugged into the system. We choose the Object-oriented Input System (OIS) [32] as the input handler in our framework.

3.2.1 OIS

OIS is an open-source object oriented input system. It is written in C++ and easily integrated with Ogre3D. It supports mice, keyboards, joysticks, and Wii remote. In our framework, input devices are initialized by the OIS library. The library supports both buffered and

unbuffered mode. The inputs taken from input devices through this library are fed to the Script Manager's "input listeners". This way a connection between inputs and the scripting is established. In scripting part, one can add a listener for key events such as "key pressed" and "key released". Logical connection between keys and their corresponding triggering actions is formed by listener and observer pattern in the scripting part. This mechanism will be described throughly in the Scripting chapter.

3.3 Graphical User Interface (GUI)

GUI is an important way of interacting with software applications. In our framework, 2D buttons and texts are rendered using a GUI library that is compatible with the render engine Ogre3D. There are many alternative GUIs, each of which has several advantages and disadvantages. We chose ButtonGUI, that is written with a very minimalistic approach, as our GUI component. We implemented the wrapping of the ButtonGUI library to our scripting language Python. ButtonGUI can use both 2D and 3D geometry on the buttons. It can render any type of fonts with Unicode support. The GUI elements can be created and changed dynamically by the scripting capability. ButtonGUI also allows us to define separate material scripts for mouse events such as onClick, onRelease, mouseOver, and mouseOff. It also sends messages when a button's position is changed or it is dropped on any other button. The button events can be associated with triggering function calls at the Python side. This way, the logic behind the GUI elements can be written by Python scripts.

3.4 Image Processing Component

In our framework, for vision-based augmented reality applications, a component which is responsible for acquiring video images and processing the image frames is required. This module is an interface between a video acquisition device or video file and the renderer. In a video see-through or monitor based AR system, the acquired images should be displayed in front of user. Image grabber takes frames from the source and sends to the renderer. Most of the time, the renderer draws the real world images to the background, behind all of the virtual objects. Apart from background drawing, the images are sent to an image processing library in which the user's context in the real world is extracted. Computer-vision libraries which uses marker-based techniques in order to search for markers and calculate the position and pose information of the user or a real object are incorporated in the image processing

module. The position and orientation information is fed into the render engine which rearranges the camera or virtual objects' positions accordingly. We use an exclusive image processing library, developed by RotaSoft company [24], in the Image Processing Component for marker identification and tracking. The library has been developed together with our AR Framework. This image processing library is tightly connected to the main core and the render engine. It basicly acquires image frames from video sources through DirectShow [19] API. On every frame, it sends the frames to the render engine. The render engine draws the frames on the background for monitor-based and video see-through based AR setups. The library adapts a topological marker search approach similar to the Reactivision library [14]. In every frame, it searches for markers. After identifying a marker in an image frame, it calculates the marker's position and orientation relative to the camera. The steps of marker detection are listed below.

- Acquire image frame.
- Binarize image by applying an adaptive threshold.
- Apply segmentation in order to define regions.
- Generate frame's topological tree.
- Search for markers whose topology trees are defined in advance.
- Calculate position and orientation of the marker using the marker features such as corners and inner regions

The design and implementation details of this underlying image library is out of this thesis' scope.

3.5 Sound Library

The sound playing capability of an AR system is crucial for the enhancement of the augmentation level. The visual animations lacking sound would diminish the user's feeling of being in the mixed environment. The Sound Library is designed to provide sound playing capability to our AR framework. We chose the open source OpenAL library [11] as the underlying sound player library which is easily integrated to the AR system. OpenAL supports concurrent sound playing, mono and stereo sounds. It is also designed for 3D positional sounds. It has hardware acceleration support. Many commercial applications, especially games, use OpenAL [11].

3.6 Video Player

Video Player component is designed in order to support picture-in-picture effects. The streaming videos can be drawn on both 2D screen and 3D virtual objects. For example, a video can be displayed on the screen of a virtual television, which is rendered in the 3D scene. The Video Player uses the wmvideo plugin [12], an open source video plugin which has been implemented using DirectShow API [19].

3.7 Script Manager

Script Manager is responsible for handling scripts written for developing applications using the AR framework. The Script Manager is the communication interface between the scripting language Python and the core language C++. It interprets the incoming messages from the Python side and dispatches them to the components in the AR framework in order to be handled. It is also designed for sending messages to the Python side from the components on the C++ side. The details of the communication and the scripting architecture is described in the Scripting chapter.

CHAPTER 4

SCRIPTING

4.1 Introduction

The scripting power of the AR framework provides a Rapid Application Development (RAD) environment by allowing us to control AR components in a flexible way. Normally, most of the components of the AR framework have been developed in C++, which is a strongly typed system programming language. Using the framework, any AR application can be developed in C++ as well. However, the development time of an application in C++ increases because of the long edit-compile-link-run cycles. A higher level interpreted scripting language, on the contrary, needs just edit-run development cycle and it handles some programming details automatically. This provides the programmer with less coding to achieve the same job. Moreover, since the programmer does not deal with low level programming issues, such as memory handling, he can solely focus on the application itself. This increases the productivity and make the application contain fewer bugs. Certainly, there is a trade-off between development time and performance. Using a scripting language may decrease the execution speed and increase memory consumption. This is due to the missing compile time optimizations. Nevertheless, as the computing power of computers increases rapidly, the performance loss caused by scripting overhead becomes negligible, especially considering the dramatic decrease in the development time and the significant increase in programmer productivity and software reuse by use of scripting languages [21].

The main reason for using a scripting language in our AR Framework is to develop AR applications rapidly without getting into complicated C++ syntax. The AR application developer should be able to create applications by just writing scripts and run them with the already compiled and linked executables. We chose Python as the scripting language and integrated it with C++. Throughout this process, we followed these steps:

1. Choose a scripting language

2. Design a communication mechanism

3. Design an event triggering system for the game logic

Before going into the details of this process, we may have a quick look at how to program in the scripting language.

4.1.1 A Quick Look at Script Programming

The scripts in the AR framework are written in Python programming language. The main executable, which is developed in C++, imports and interprets the written Python scripts. The structure of the scripting has been built on three logical units: *Sensor*s, *Controller*s, and *Actuator*s. Sensors sense whenever an event happens, such as a key press, mouse movement, and a marker visibility. Sensors are linked to Controllers, which controls the activation of the Actuators. The actuators are the actual units that have effect on the application. For example, they can change position, motion, visibility, and size of an object. *Sensor*s, *Controller*s, and *Actuator*s are connected in a *Link*. The application levels or scenes are composed of *Link*s. Thus, we build an application by creating *Link*s and adding them to the *Scene*s.

For example, we can create a 3D cube when the scene is loaded and rotate it around y-axis by pressing "space" key. We use CreateModelActuator for loading and rendering 3D cube geometry stored in *cube.mesh* file. PositionSetActuator is used to locate the cube in x,y,z coordinate system relative to the marker. We use RotateActuator and KeySensor to associate the "space" key with rotation of the cube.

```
# we create a Scene instance
Scene01 = Scene("Scene01")

# add Links to the Scene instance
Scene01.addLink(Link([ActuatorSensor("sceneLoadedSensor", "onScene01")],
ANDController(""),
[CreateModelActuator("CubeCreate", "Cube", "cube.mesh")
,PositionSetActuator("CubePos", "Cube", (0, 0, 0))
]
))

# when space key is pressed, cube is rotated
Scene01.addLink(Link([KeySensor("", "Space")],
```

```
ANDController(""),
[RotateActuator("CubeRot", "Cube", (0, 90, 0))
]
))
```

The details of the scripting and its usage are described in this chapter.

4.2 Scripting Language

There were two alternatives for incorporating a scripting language to the AR system:

1. Design a new scripting language
2. Use an existing embeddable scripting language

At first, we thought of designing our own scripting language with a very simple, human-readable syntax. But after exploring this idea, we discarded it because of the need of extra development time for designing the scripting language and implementing a parser and manager for it. Designing an exclusive scripting language for the AR framework seemed to be risky because it might make us lose our focus.

As an alternative, choosing an existing scripting language and embedding it has several advantages. It lets developers start faster. There is no need to spend time for development or maintenance. Moreover, a language we would design would not be as efficient as the existing scripting languages which are robust and cleverly designed. As scripting language alternatives, we considered Lua [23] and Python [18], both of which are easily embeddable scripting languages and used in several games and multimedia applications. We chose Python as our scripting language because of its broad documentation, extension libraries, being widely-used, and our familiarity with its syntax.

4.3 General Scripting Structure

Most of the components of the AR framework are developed in C++. These components, e.g. Image Processor, Sound Manager, and Video Player are time-critical so they have to benefit from the C++'s high performance. The Script Manager component handles Python scripts through the Script Manager. The Script Manager creates the connection between two languages: one strongly typed static language, 'C++', and one typeless dynamic language 'Python'. The AR framework can be viewed as two main parts:

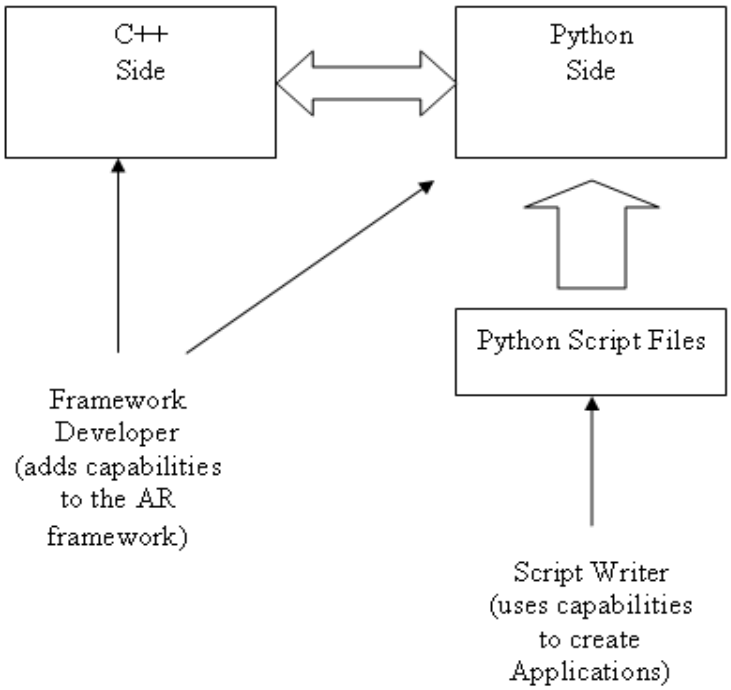

Figure 4.1: The Scripting Structure and The Developer Roles

C++ Side: the core engine

Python Side: the interface

C++ has been used as the core system language and Python has been used as a 'glue' language to create applications. Python scripts do not define new capabilities. Instead, they just create the logical links between components. The interfaces for communicating with components are defined in Python and their corresponding handlers are implemented in C++ in order to achieve better performance. If performance is not an issue, the implementation can be done on the Python side as well, without the need for communication with C++.

From developers' point of view, there are two roles defined: AR Framework Developer and Application Developer. AR Framework Developer can add new capabilities by writing Python interface classes and their corresponding delegate classes on the C++ side. There is no restriction for the framework developer.

An application developer (script writer) does not have to go into details on the C++ side. He only uses the AR system's capability by just looking at the abilities provided by interfaces on Python and selecting any of them. He does not need to know anything related to implementation details running in the background. This working mechanism and developer roles are plotted in Figure 4.1.

Using this structure, the designed division of labor between C++ and Python provides a significant software re-usability. In order to develop AR applications, the framework components developed in C++ can be used with Python scripts without rebuilding.

4.4 C++ and Python Integration

In our AR Framework, Boost Python library has been used to export C++ classes to Python and embed Python in C++. Boost [7] is a robust open source package that provides various portable C++ libraries. Its Boost.Python package has an easy high-level interface for exporting C++ data to Python and vice versa. Any class defined in C++ can be exported to Python using the Boost library. Sometimes wrapper classes are needed to be written in order to comply with data types and interfaces. Boost.Python library generates Python dynamic link library (DLL) files with .pyd extension that can be used from a Python shell or embedded Python.

4.5 Communication Between C++ and Python

All of the framework component classes and methods could be exported to Python using Boost Python library and they could be accessed by the Script Writer. But it would make the Script Writer have excessive control over components' parts. A level of encapsulation seemed to be necessary to restrict and control Python script usage. In addition, this type of complete exposure to Python by exporting all of the capabilities as it is, would only simplify code writing using Python's syntax. What we want was some sort of independence of scripting from the underlying components on the C++ side. Thus, instead of a direct access to component elements (classes and functions), we considered building another layer between Python and C++. This layer would be taking orders from Python scripts and send them to the corresponding connected component. Even if any underlying component changes, the written Python script will not need any change but this intermediate communication layer would just call the replaced component. Two different approaches were followed while implementing this layer.

In the first approach, the layer was implemented as wrapper classes for every component. The wrapper classes become like an interface to component classes. They have their own methods that are needed on the Python side and they use the component classes as delegates. In Python scripts, the Script Writer might use the Python extension of these wrapper classes.

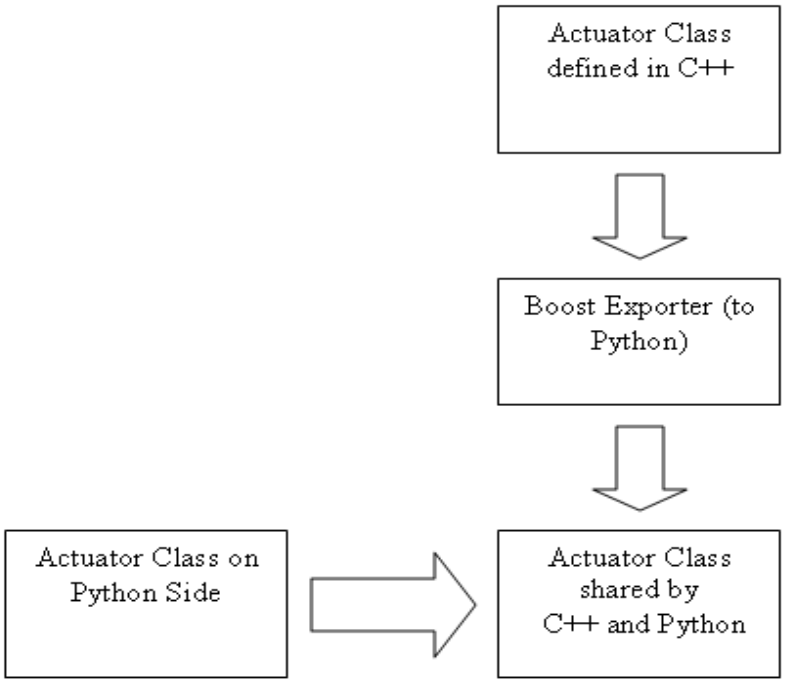

Figure 4.2: Actuator Sharing Between C++ and Python

If any component changes or replaced by another one, only the delegate component class of the wrapper class requires to be changed. If the replaced component has different interface, the wrapper class will change its delegate method invocation. There is no need to change anything on the Python side, as it does not carry any component specific code. We used this wrapping mechanism for some of the components in the AR framework, e.g. GUI.

However, instead of this class wrapping mechanism, we considered adapting a messaging system as a communication between C++ and Python. We treated every order from Python side as messages to be handled on the C++ side. We named the messages as 'actuators'. Every capability that we wanted to be available on Python side has been defined as an 'Actuator'. A C++ class that is to be shared with Python, has been created for each Actuator. This Actuator class has data fields that carry the message information. In Python, another Actuator class is defined that uses this shared class as data holder. This is depicted in Figure 4.2.

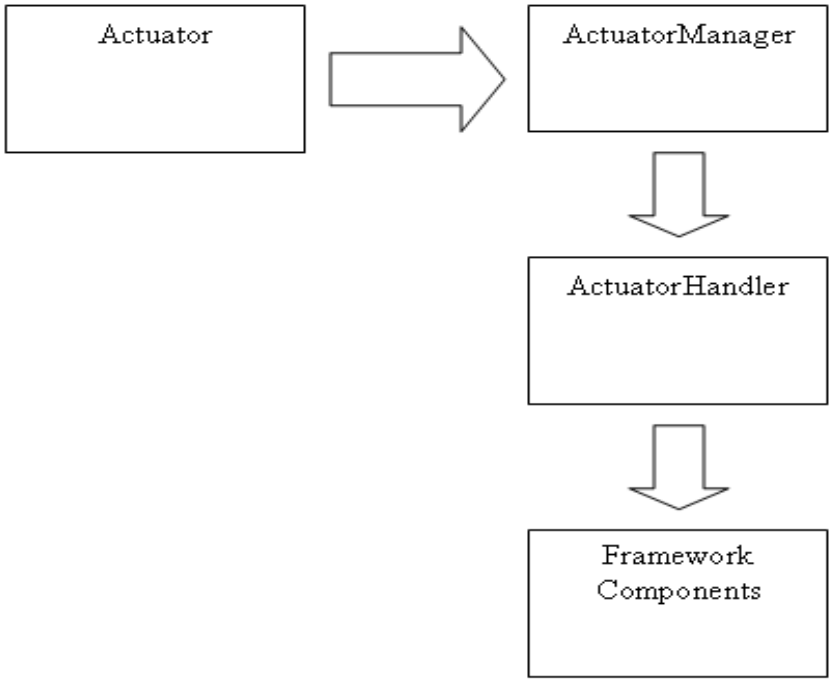

Figure 4.3: Actuator Handling

4.6 Actuator Handling

On the C++ side, we define an ActuatorHandler which will be responsible for handling the received Actuator. The ActuatorHandler uses the component available as a delegate to do the desired job. The Python script just creates the actuator and fills its parameters and sends it to the ActuatorList. ActuatorList is monitored by ActuatorManager in every frame. ActuatorManager is in Script Manager component and responsible for dispatching Actuators in the ActuatorList to their corresponding ActuatorHandlers looking at their type. Actuator Handling steps are shown in Figure 4.3

The connection between ActuatorManager and ActuatorHandlers is established in a way that it needs minimum code change whenever a new capability is added to the system by implementing a new ActuatorHandler. The class should be inherited from ActuatorHandler class and override the *handleActuator()* method. If any time dependent action is to be defined, handleActuator(float t) method should be implemented to define the behaviour of the handler. There are also *isActive()*, *isDead()* methods to decide whenever the ActuatorHandler has to be run and should be deleted respectively. After an ActuatorHandler finishes its job and gets into the *dead* state, the ActuatorListener on the Python side is notified with

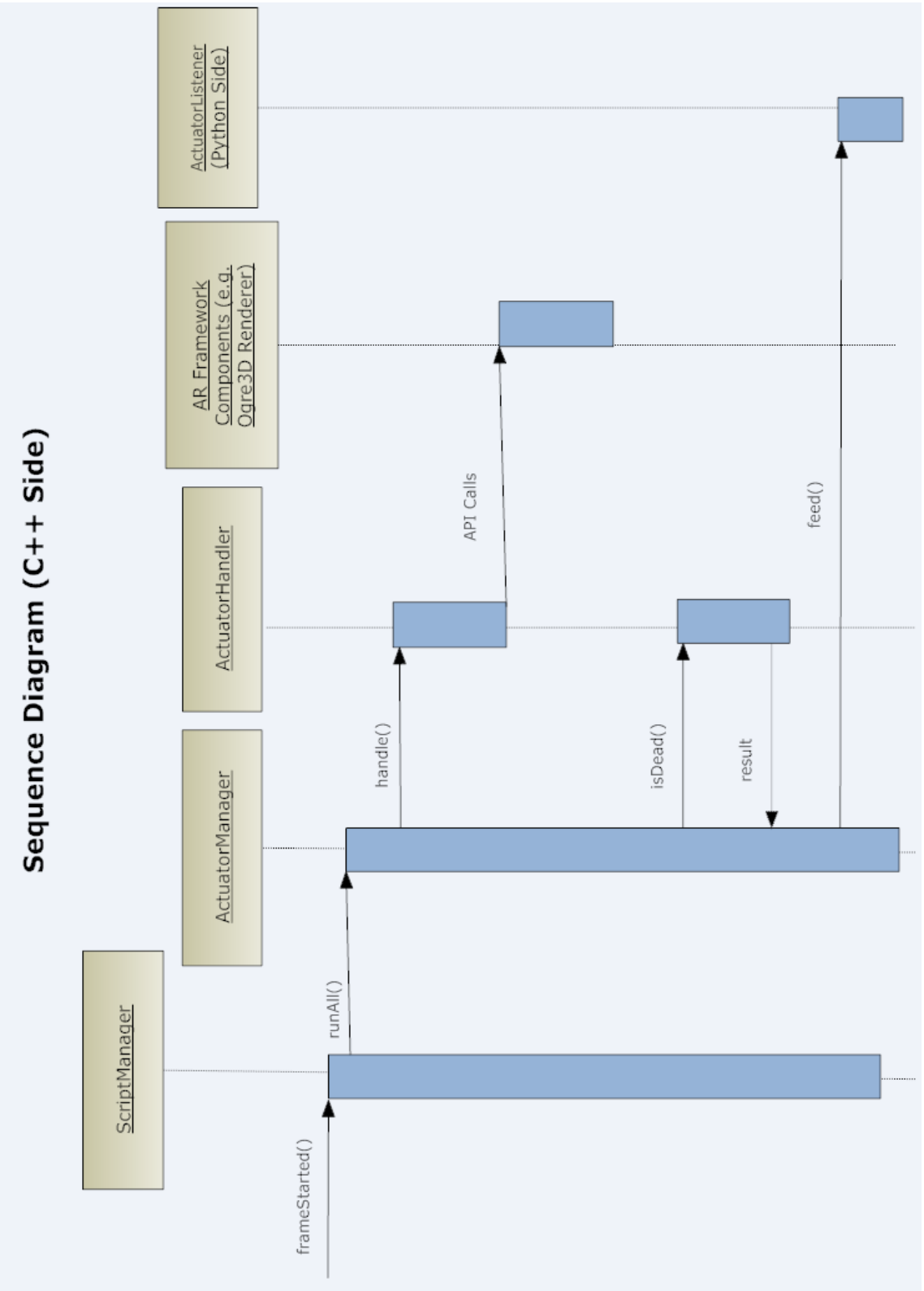

Figure 4.4: The sequence diagram of Actuator Handling

the Actuator ID. The sequence diagram of this process is shown in Figure 4.4.

As an example to this message system, let us think, we need to give the Script Writer an ability to change virtual lights in the augmented 3D scene. First of all, we define a 'LightActuator' on the C++ side. It will be shared by Python as a data change medium. We expose the 'LightActuator' to the Python using Boost.Python library. We write an encapsulating 'LightActuator' class on Python side that creates an instance of the shared 'LightActuator'.

Step 1: Actuator Definition on the C++ side

```
class LightActuator: public Actuator {

public:
    LightActuator() : Actuator() {
        create = false;
    }
    std::string lightName;
    bool create; // create or change
    double posx, posy, posz; // 3D position of light
    double difr, difg, difb; // diffuse colors
    int type; // 0 point, 1 directional, 2 spot light
    virtual Type getType()
    {
        return LIGHT;
    }
};
```

Step 2: Export Class to Python

```
...
using namespace boost::python;

BOOST_PYTHON_MODULE( Actuator )
{
    class_<LightActuator, bases<Actuator>>("LightActuator")
        .def_readwrite("lightName", &LightActuator::lightName)
        .def_readwrite("create", &LightActuator::create)
        .def_readwrite("type", &LightActuator::type)
        .def_readwrite("posx", &LightActuator::posx)
        .def_readwrite("posy", &LightActuator::posy)
        .def_readwrite("posz", &LightActuator::posz)
        .def_readwrite("difr", &LightActuator::difr)
        .def_readwrite("difg", &LightActuator::difg)
        .def_readwrite("difb", &LightActuator::difb);
}
```

Step 3: Python Interface class encapsulating the data holder

```
# Creates light with
# name: lightName, 3D pos: pos, diffuse color: color, light type: type
class CreateLightActuator(Actuator):
    def __init__(self, name, lightName, pos, color, type = 0):
        self.actuator = ActuatorModule.LightActuator()
        Actuator.__init__(self, name)
        self.actuator.lightName = lightName
        self.actuator.create = True
        self.actuator.type = 0
        self.actuator.posx = pos[0]
        self.actuator.posy = pos[1]
        self.actuator.posz = pos[2]
        self.actuator.difr = color[0]
        self.actuator.difg = color[1]
        self.actuator.difb = color[2]
    def onUpdate(self):
        activeActuatorList.append(self.actuator) #fill actuatorlist
```

The above steps set up the data exchange part. In addition, we need to define how to handle LightActuator in LightActuatorHandler on the C++ side. After implementing LightActuatorHandler, we need to add some dispatcher code in ActuatorManager class to send incoming LightActuators to LightActuatorHandlers.

Step 4: ActuatorHandler Definition

```
class LightActuatorHandler: public  ActuatorHandler {

...
    virtual void handleActuator() {
        Light *light;
        if (actuator.create) {
            try {
                light = mSceneMgr->getLight(actuator.lightName);
            }
            // if the light- does not exist
            catch (Ogre::Exception& e) {
                light = mSceneMgr->createLight(actuator.lightName);
            }
        }
        try {
        light = mSceneMgr->getLight(actuator.lightName);
        light->setPosition(actuator.posx, actuator.posy, actuator.posz);
        switch(actuator.type) {
            case 0:
                light->setType(Light::LT_POINT);
                break;
            case 1:
                light->setType(Light::LT_DIRECTIONAL);
```

```
                break;
            case 2:
                light->setType(Light::LT_SPOTLIGHT);
                {Vector3 dir(-light->getPosition());
                dir.normalise();
                light->setDirection(dir);
                }
                break;
            default:
                break;
        }

        light->setDiffuseColour(ColourValue(actuator.difr, actuator.difg, actuator.difb));
        light->setSpecularColour(ColourValue(1, 1, 1));
    ...
```

Step 5: Dispatching Actuator to ActuatorHandler in ActuatorManager

```
...
boost::python::object cur = actList[i];
Actuator &s = boost::python::extract<Actuator&> (cur);
switch(s.getType())
{
...
case Actuator::LIGHT:
activeHandlers.push_back(new LightActuatorHandler(static_cast<LightActuator&> (s), mSceneMgr));
                break;
...
```

The class diagram of Actuator handling mechanism in general is shown in 4.6

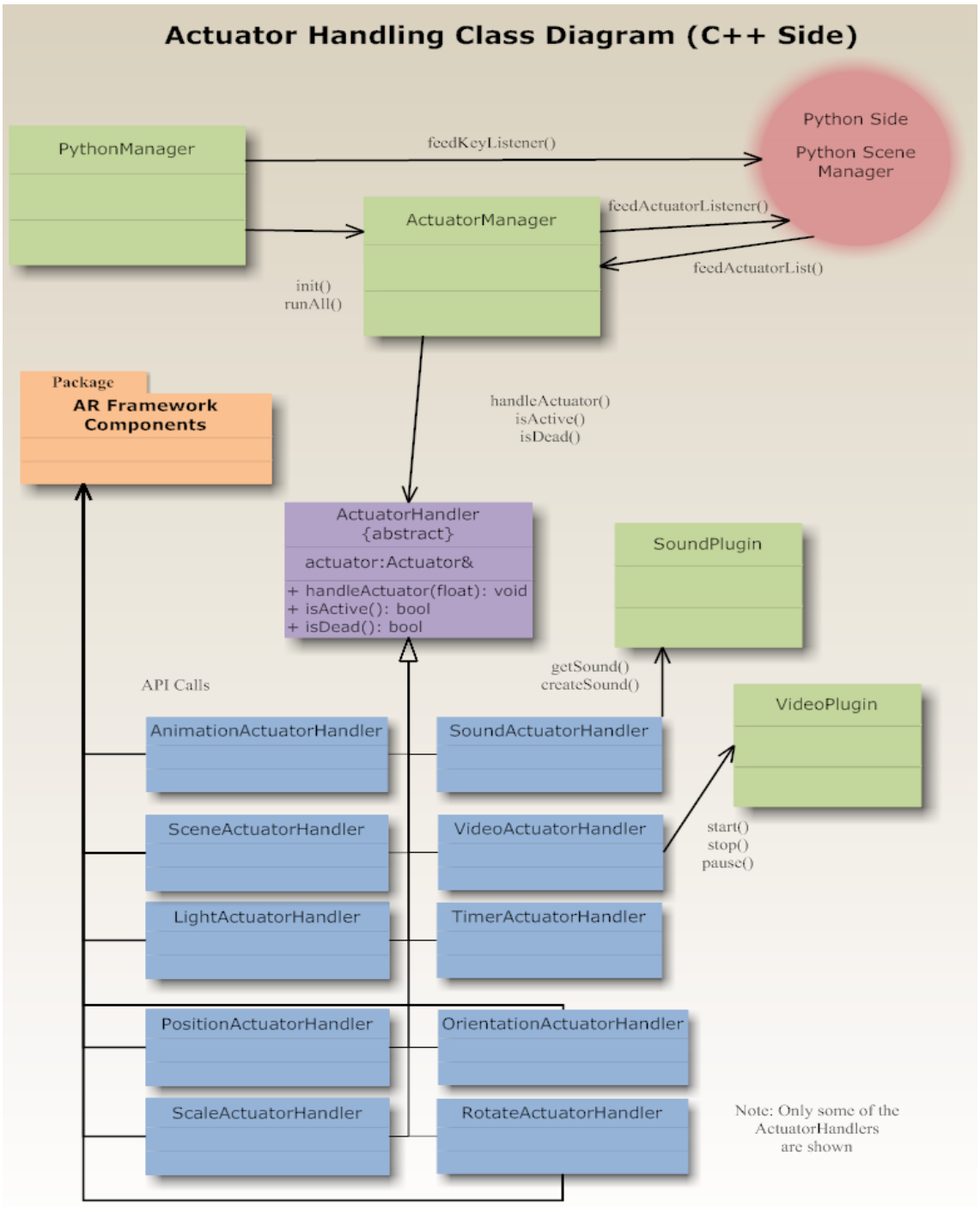

Figure 4.5: The class diagram of Actuator Handling

This messaging method provides an encapsulation between scripting and components. The scripting part becomes loosely coupled with the components. In addition, the Script Writer just sends messages by creating Actuator instances from the Python side and he does not need to know anything about how they are handled on the C++ side.

4.7 Messaging from C++ to Python

The one way communication from Python to C++ may not be enough for the Script Writer to create AR applications. He may need to inquiry the state of the system, receive messages and user inputs or even ticks for every new frame. In C++, the Script Manager is responsible for sending messages to the Python side. We adapted a structure with the listener-observer pattern [10].

The scripting architecture and messaging between C++ and Python is shown in Figure 4.6.

We define Listener classes in Python and append any data to these Listeners from C++. Using Boost.Python library, C++ can reach these Python Listener class instances.

There are 4 types of Listeners:

KeyListener: The Script Manager, on the C++ side, is connected to the Input Library. It appends user input keys to the KeyListener.

ActuatorListener: ActuatorManager class in the Script Manager appends finished actuators' IDs to this Listener.

MarkerListener: This Listener is fed by the Image Processor Component with the recently detected marker ID, position, and orientation.

AlwaysListener: This Listener is notified by the Script Manager for every frame tick.

4.8 Message Triggering

The data flow starts from the C++ side and reaches to the Python Listeners. ActuatorManager and the Script Manager notify listeners. The listeners then notify the Sensors. The triggering continues over Controllers and Actuators, ends up back on the C++ side through ActuatorManager's ActuatorList. [Sensors -> Controllers -> Actuators] connectivity and triggering is encapsulated in the Link class. The message triggering is shown in Figure 4.7

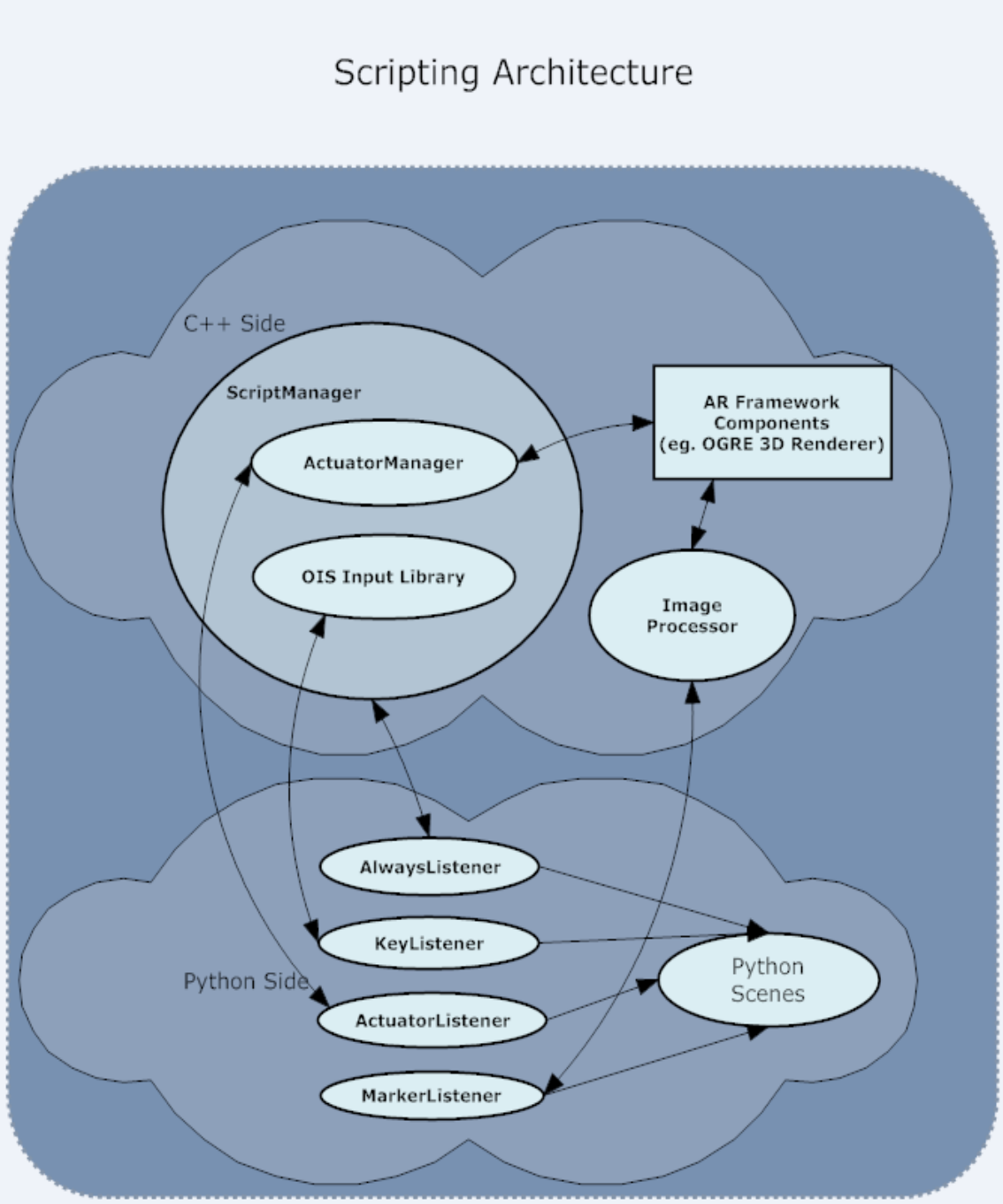

Figure 4.6: Scripting architecture built between C++ and Python

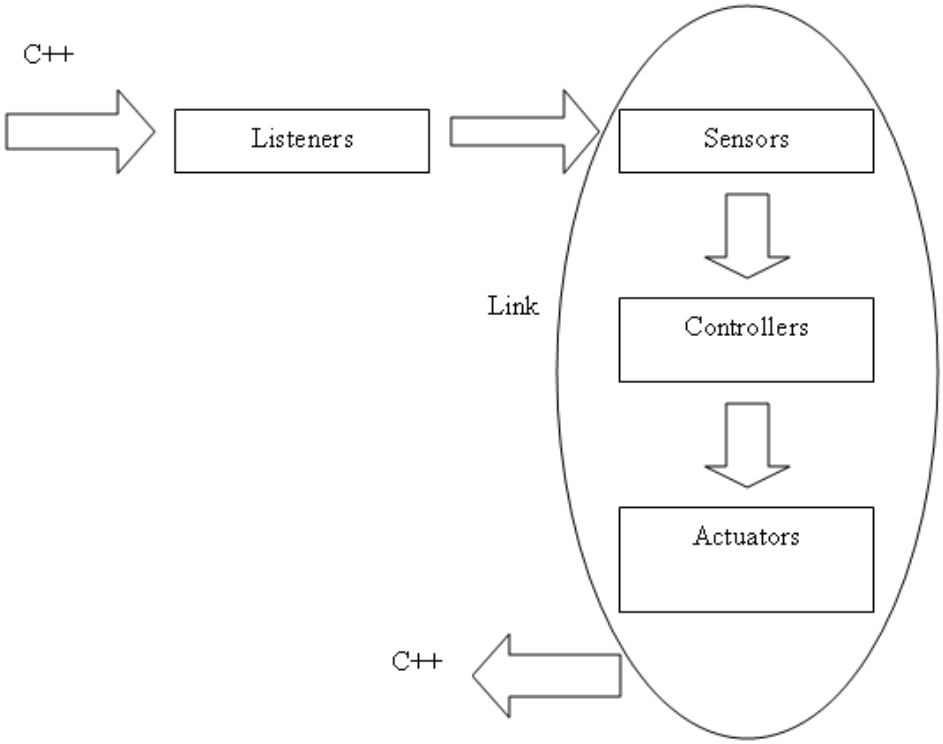

Figure 4.7: Triggering Steps

4.8.1 Sensor-Controller-Actuator Structure

We use the Sensor, Controller, and Actuator structure which forces event-driven programming. The flow of the application is controlled by events that are sensed by Sensors. We are inspired by Blender Game Engine [6] which uses this structure to create computer games rapidly using the 3D graphics prepared in Blender. The logical elements in this structure, e.g. Actuators and Sensors are independent of the objects they affect. The pattern is very similar to the command pattern [10]. Actuators are like command objects which change the properties of the object whose ID is given as a parameter. An alternative to this method could have been achieved by exposing all the components' classes and methods to Python. This way an imperative object-oriented script writing could be possible. But, the scripts would be closely coupled with underlying components. Any change on the C++ side would affect the Python scripts immediately. Wrapper classes can be written to encapsulate the underlying components in order to remove the scripts' dependency. However, it would not be practical to wrap and expose every class. Instead, Python side classes and methods can be written in order to hide Actuator creation and sending.

4.8.2 Sensors

Sensors are designed to sense the key, action, marker, and frame change events on Python side. There are 4 kinds of sensors:

KeySensor: It checks if the pressed key is the same as this sensor's key.

AlwaysSensor: This sensor is activated for every frame.

ActuatorSensor: It checks if actuator which finished its execution has the same name with this sensor's registered actuator name.

MarkerSensor: It checks if the current marker is the same as this sensor's marker.

The Sensors are registered to the related Listeners. For example, a KeySensor("A"), that is used to sense if user presses key "A", registers itself to KeyListener. KeyListener gets notification from the C++ side for every key strokes and for every registered KeySensor it checks if the key labels match. Then it notifies the Sensor accordingly. If user presses key "A", the registered KeySensor("A") will be activated.

4.8.3 Controllers

The triggering between Sensors and Actuators is established through Controllers. When a Sensor in a Link is activated, it checks the Controller and if the controller returns true it sends its Actuators to the ActuatorList. There are 3 types of Controllers.

ANDController: This controller triggers Actuators when all of the sensors in the Link become notified.

ORController: This controller triggers Actuators when at least one of the sensors in the Link become notified.

PythonController: This controller holds a link to the Python function returning *true* or *false*. It triggers Actuators whenever the Python script returns *true*.

Additional Controllers may be added to the system.

4.8.4 Triggering on Key Inputs

On the C++ side, the Script Manager receives both buffered and unbuffered keys via OIS library. It feeds these keys to the KeyListener instance on the Python side. The active KeySensors registered to the KeyListener are activated depending on the key value. The triggering to the Actuators continues if the Controllers which are linked to the KeySensors return *true* value. The actuators are appended into the ActuatorList. The sequence diagram of key input triggering is shown in Figure 4.8.

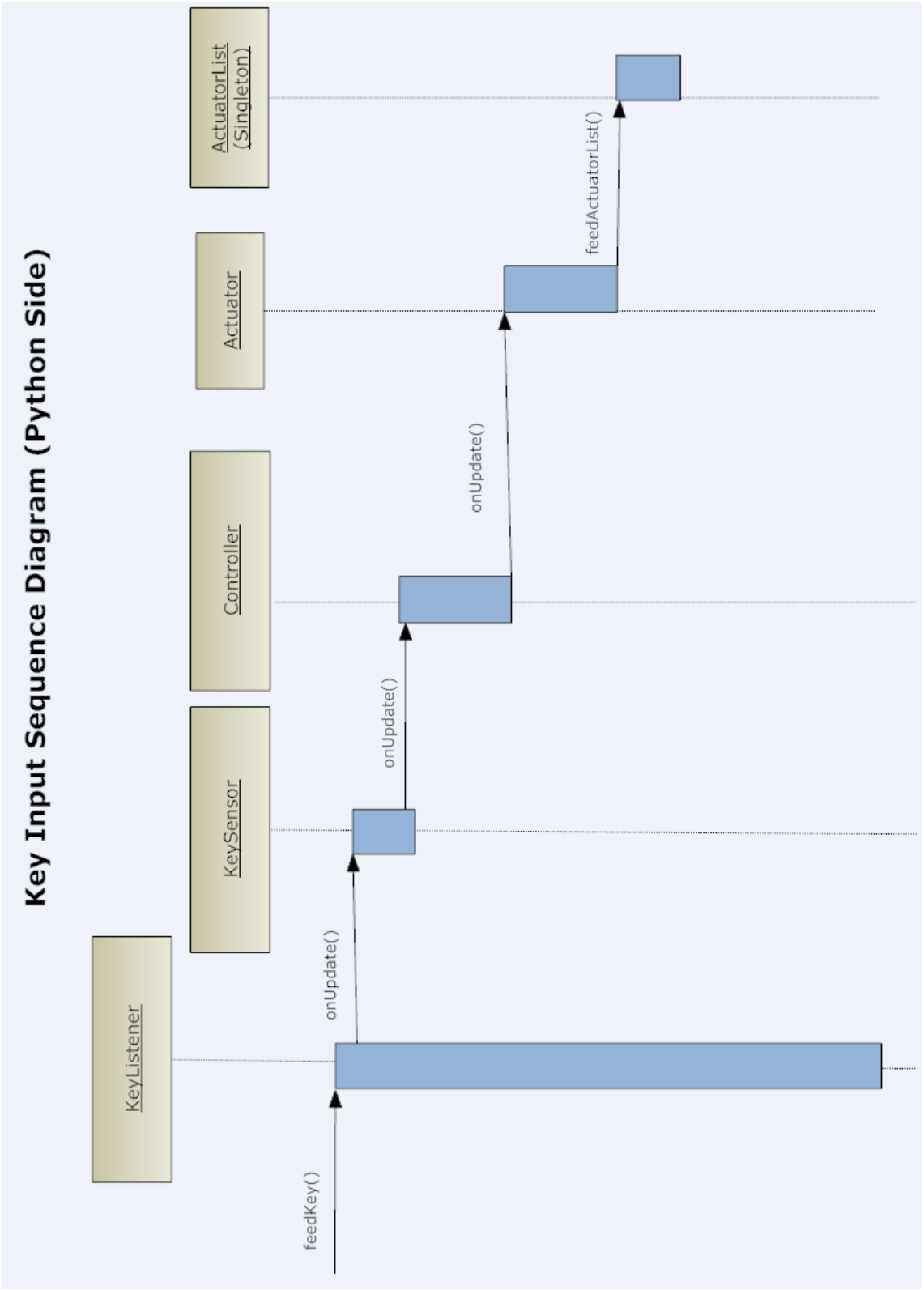

Figure 4.8: The sequence diagram of key input triggering

4.8.5 Triggering on New Frames

The Script Writer might need a Python function called or message sent to the C++ side on every frame. In order to provide this, the Script Manager notifies AlwaysListener instance on the Python side. The active AlwaysSensors, which are registered to the AlwaysListener, are activated on every new frame. The cascaded triggering continues till the Actuators depending on the Controller's return values.

4.8.6 Triggering on Events

Most of the time, it may be needed that events trigger other events. In our scripting architecture, the actuators generate a message after they are handled. The ActuatorManager class in the Script Manager, feeds the handled Actuators' IDs to the ActuatorListener instance on the Python side. The active ActuatorSensors, which are registered to the ActuatorListener are activated depending on the Actuator ID. As an example, an ActuatorSensor can be defined to be activated after 5 seconds passed. At first, a TimerActuator with a unique ID and 5 seconds parameters. The ActuatorSensor instance is created with this TimerActuator's ID as its parameter. After the TimerActuator is sent to the ActuatorList and handled on the C++ side, its ID is fed to the ActuatorListener. Then the ActuatorSensor instance is notified.

4.8.7 Triggering on Marker Inputs

Markers that are visible on the acquired real world images are treated as inputs. The Image Processor component is designed to notify MarkerListener on every new real world frame. MarkerSensors are defined to sense marker changes on the Python side. The active MarkerSensors, which are registered to the MarkerListener, are notified whenever a marker with the same ID is detected in the Image Processor component. The marker IDs are integer values starting from 1. The marker data appended to the MarkerListener, contains position and orientation values. They are used by PositionSetActuator and OrientationSetActuator in order to *register* 3D objects with the marker in the real world.

4.8.8 Actuators

As previously described, Actuators carry the action messages from Python to C++. They have an *onUpdate* method which appends the Actuator to the end of the ActuatorList queue. Some of the Actuators are listed below.

1. **AnimationActuator**: It creates 3D models. It can play, pause or stop an animation of a 3D object. This actuator is also used for changing visibility property of 3D objects.

2. **PositionActuator**: It sets the 3D position (x, y, z) of a virtual object. It is also used for *registering* the root scene node, to which all the objects are connected, relative to the marker.

3. **OrientationActuator**: It sets the orientation (w, x, y, z) of an object in 3D space. It is also used for *registering* the root scene node, to which all the objects are connected, relative to the marker.

4. **AlignActuator**: It consists of a PositionActuator and an OrientationActuator. It aligns the root scene node with the visible marker in the real images.

5. **RotateActuator**: It rotates an object in 3D space around 3 axes (pitch, yaw, roll).

6. **ScaleActuator**: It scales an object in 3D space relative to 3 axes (x,y,z) defined by the marker plane.

7. **SceneActuator**: It handles scene changes by resetting every animation object, deleting timers, destroying sounds, stoping and clearing videos. It also can change the overall speed of animation speed in the AR application. It supports pause, play mode for the applications.

8. **TimerActuator**: It creates timers and it can be used for callbacks and delays in the applications.

9. **CameraActuator**: It handles the virtual camera's position, orientation in the 3D scene. It can be used to give camera effects in the virtual world.

10. **MaterialActuator**: It changes the materials (textures, colors, opacity), visual properties of objects in the 3D scene.

11. **ModelActuator**: It is similar to the AnimationActuator but it only deals with static objects (objects without animation or motion). It is used for simplicity.

12. **MovableTextActuator**: It creates 3D texts that face to the virtual camera in the 3D scene. It can be used to label and tag virtual objects. In addition, it can be used for debug purposes in a scene including too many objects.

13. **OverlayActuator**: It creates 2D texts and images on the screen. It can be used to give textual and visual information. It can be used for debugging as well.

14. **ParticleActuator**: It creates, starts, stops, and destroys particle effects in the 3D scene.

15. **SoundActuator**: It creates sound objects from sound files using SoundPlugin component. It is used for playing, pausing and stopping sounds by SoundActuatorHandler using SoundPlugin API calls.

16. **VideoActuator**: It is similar to SoundActuator. It is used for playing, pausing, stopping videos by VideoPlayerActuatorHandler using VideoPlugin API calls.

The class diagram of the Python side containing Actuators is shown in Figure 4.9 and Figure 4.10.

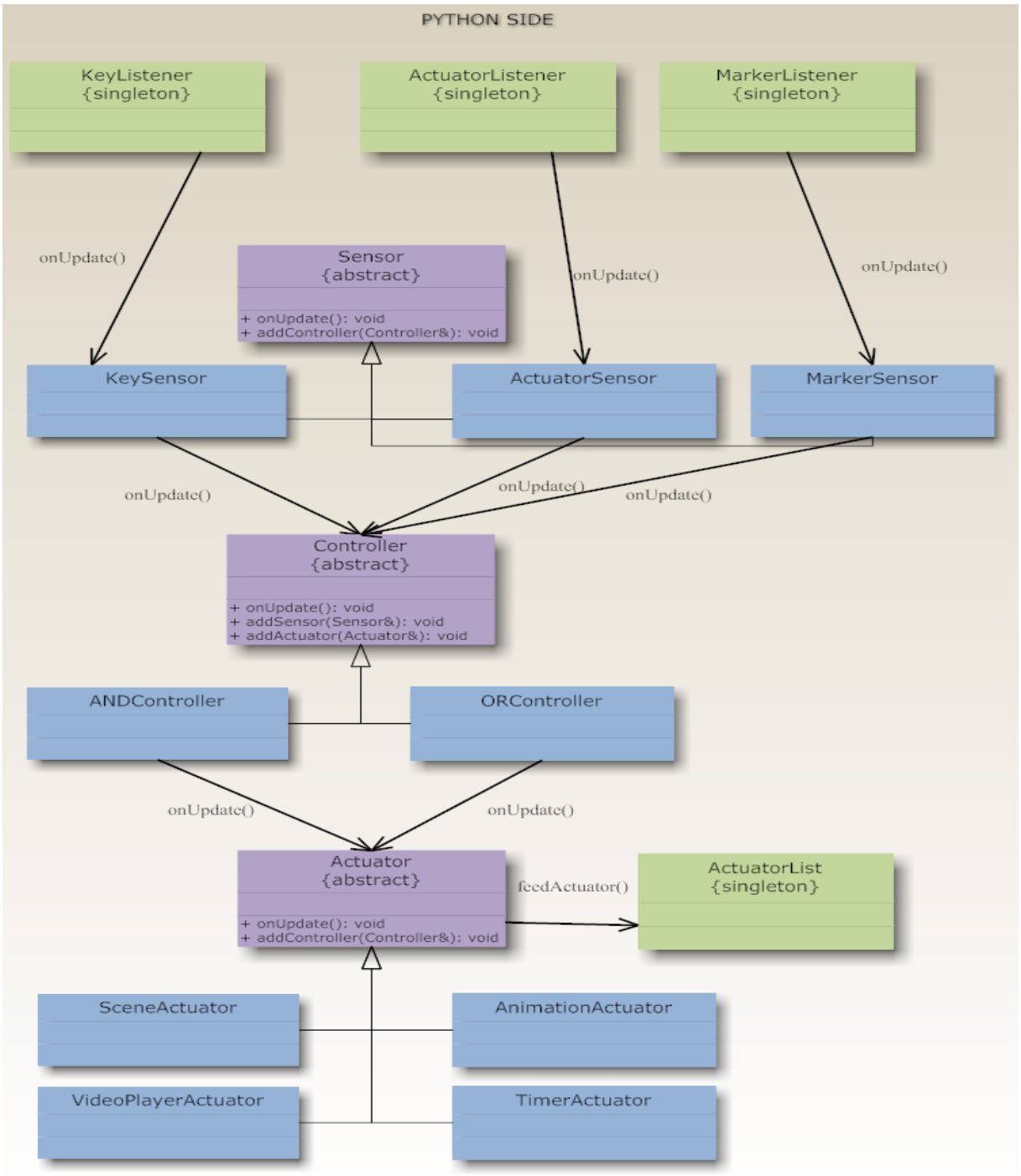

Figure 4.9: Python class diagram

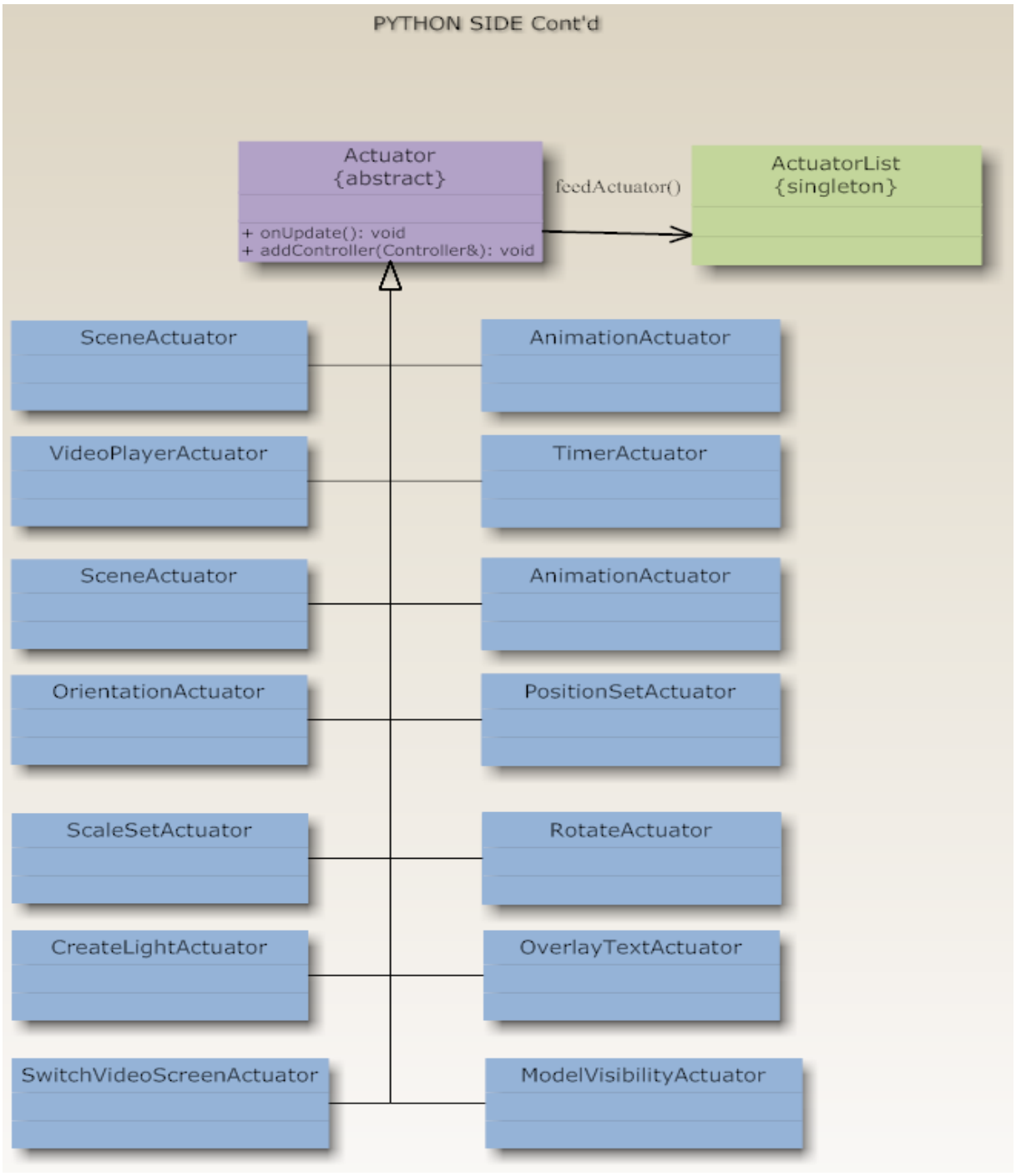

Figure 4.10: Python class diagram continued

4.8.9 Links

Links hold Sensor, Controller, and Actuator arrays. The logical connection between these 3 elements are encapsulated in Links. Links form the building blocks of Scenes.

4.8.10 Scenes

Each part, scene, and level in the game logic is defined in **Scene** class. Actually a level is created by the composition of Links (used for active messages and logic). So, Scene is just a container of Links. Scene specifies the active messages, active sensors, and actuators by updating listeners.

Python Side Scene, Link, Sensor, Controller, and Actuator class diagram and encapsulation structure is shown in Figure 4.11.

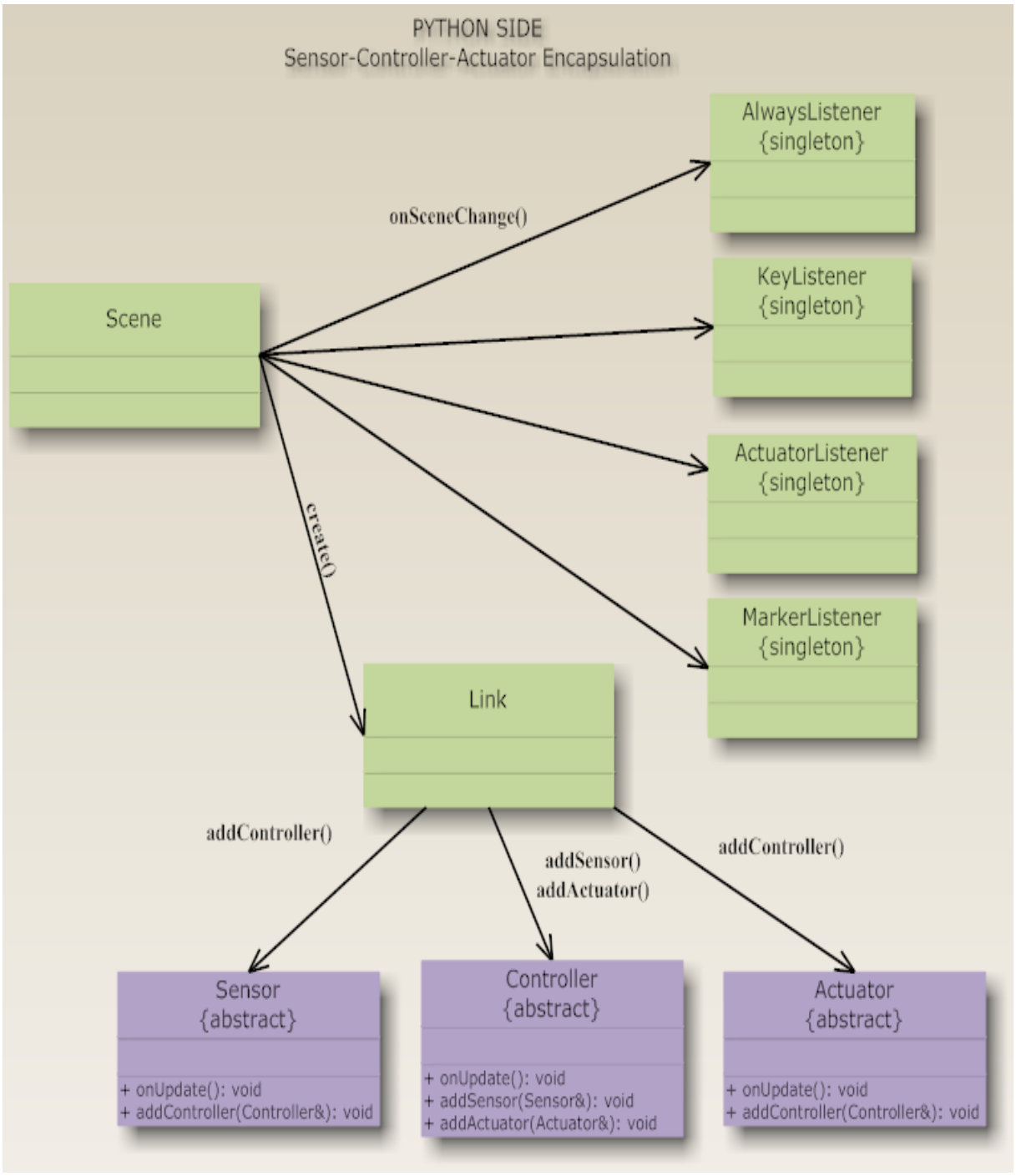

Figure 4.11: Python Class Diagram and Encapsulation of Sensor, Controller, and Actuators

The general view of the scene management via message triggering is diagrammed in Figure 4.12.

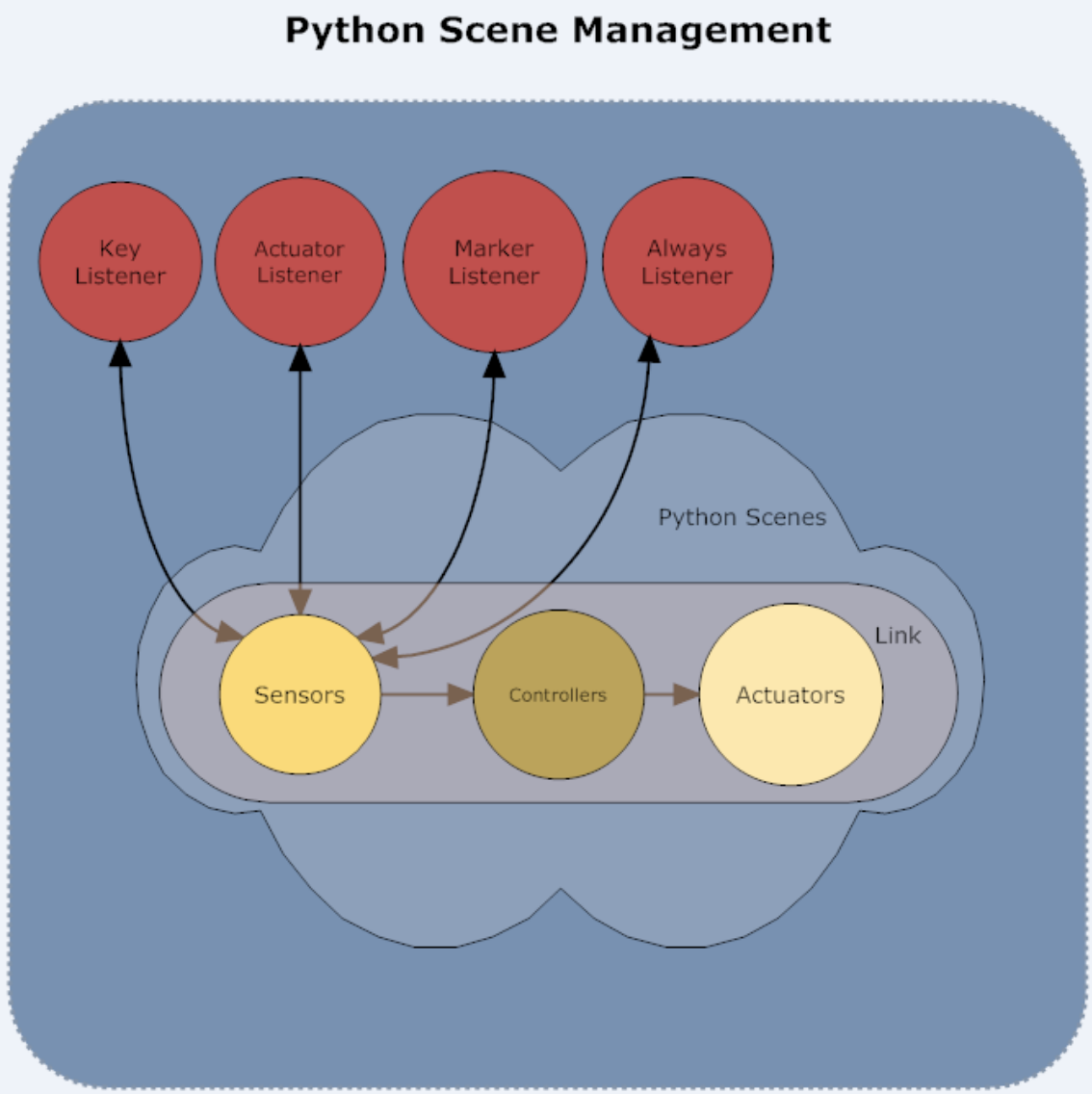

Figure 4.12: Python Scene Management

4.9 Loading Scenes

The Script Manager loads Scenes through importing only one Python file: *scenario.py*. This file consists of Scene imports and Marker associations. Depending on the application's complexity, countless Scene files can be created and these files can be imported in *scenario.py*. Any common order that will affect all of the Scenes can be written in this file. For example,

every Scene can be matched with a distinct Marker, such that whenever the Marker becomes visible, the Scene is loaded. The changing Scene conditions are defined in *scenario.py* as well. Some part of an example scenario.py file is shown below.

```
SCENE_COUNT = 10
Scenes = [ ]

# import all the scenes from scenes folder.( Scene01, Scene02,...)
for i in xrange(1, SCENE_COUNT+1):
    scene_str = "0" + str(i) if (i < 10)  else str(i)
    import_str = 'from ' + "scenes.Scene" + scene_str + ' import Scene' + str(i)
    exec import_str
    append_str = "Scenes.append(Scene"+ str(i)"
    exec append_str

# associate Marker1 with Scene1, Marker2 with Scene2, ...
# when Marker1 becomes visible in Scene1,
# its position and orientation will be used to align virtual objects
for i in xrange(len(Scenes)):
    Scenes[i].addLink(Link([MarkerSensor("", i + 1)], ANDController(""),
                           [AlignActuator("MarkerCenterAlign")]))

# marker change conditions: when Marker1 is found in a scene other than Scene1, Scene1 is loaded.
for i in xrange(1, SCENE_COUNT+1):
  for j in xrange(len(Scenes)):
    if j == i:
      continue
    Scenes[i].addLink(Link([MarkerSensor("", j + 1)], ANDController(""),
                [ChangeSceneActuator("onScene" + str(j + 1), Scenes[j], listeners)],
               ))
```

4.10 Example Scripting Usage

An AR application, which is developed through our proposed framework, will be composed of various Scenes. Every Scene is used as a level or stage in the application. The Script Writer will just add new Scenes. Every Scene can be written in a separate Python file (.py extension) starting with Scene01.py and incrementing the numeric part. (Scene02.py, Scene03.py ...)

Creating a Scene is just adding Links to the Scene. Scene class' addLink method is used to add Links. Scene is to be initialized with a scene name. The convention is to give the same name with the Scene variable.

Example: Scene01 = Scene("Scene01")

The starting state is specified by the Link that has the ActuatorSensor with actuator name "onSceneXX" (XX digits). This is not strict and other methods may be used for defining the entry point.

An example Scene file is shown below. It creates a light in the 3D world and plays a music file when the Scene is loaded. After the music ends, a ninja model created and it walks from left to the right relative to the Marker if there is any associated:

```
Scene01 = Scene("Scene01")
ninja_scale = (0.5, 0.5, 0.5)
ninja_start_pos = (-200, 0, 200)
ninja_dest = (200, 0, 200)
Scene01.addLink(Link([ActuatorSensor("Scene01LoadedSensor", "onScene01")],
          ANDController(""),
          [
   CreateLightActuator("LightCreator1", "Light1", (500,500,0) , (1, 1, 1), 0)
          ,SoundPlayerActuator("TestSound", "test.wav" , play = True, loop = False)
          ]))

Scene01.addLink(Link([ActuatorSensor("TestSoundFinished", "TestSound")]
          ANDController(""),
          [ CreateAnimationActuator("CreateNinja", "Ninja", "ninja.mesh")
          ,ScaleSetActuator("NinjaScale", "Ninja", ninja_scale)
          ,PositionSetActuator("NinjaPos", "Ninja", ninja_start_pos)
          ,AnimationActuator("NinjaWalk", "Ninja",
                       playAnim = "walk", loop = True, dest=, turn=True))
          ]))
```

4.11 Graphical User Interface Control

We followed a different control mechanism for GUI elements. Instead of sending messages by actuators to the GUI component, we created a wrapper class PyButtonManager that provides an interface for controlling all of the GUI component (buttonGUI) elements and events on the Python side. The PyButtonManager creates only one GUI instance and registers the underlying buttonGUI to the renderer Ogre3D. A higher level of encapsulation is achieved by writing a *Button* class on the Python side, which uses PyButtonManager as a delegate for creating, changing, hiding, showing GUI elements.

All of the GUI elements (texts, images, buttons, areas) are defined of type *Button*. Each *Button* has a unique id. Its material properties, size (width, height), 2D position, alignment and, text can be changed dynamically. In addition, *Button*s which contain an input field,

have additional getInputText method for getting input text values from the user. *Button*s may contain extra child *Button*s. Their layout can be arranged by changing position and alignment values of the items in the script files. For every Scene file, an additional GUI file may be attached. The Scene file imports the GUI file and GUI code. In consequence, the application code is seperated into two different files.

The event handling of *Button*s is performed by defining various *Button* actions: *OnClick*, *OnRelease*, *MouseOver*, *MouseOff*, *MouseWheelUp*, *MouseWheelDown*. Any Python method can be registered to any action of a *Button*. In order to access to the other AR components by GUI element actions, still actuators are used. Actuators are created and their onUpdate methods are invoked in a Python method. Then, the Python method is registered to a *Button*'s event. For example, changing the visibility of a 3D screen object in the scene by pressing a mouse button on a *Button* would require a Python method that creates ModelVisibilityActuator instance and calls its onUpdate method. Then this method needs to be registered to the *Button*'s OnClick event. The implementation of this process is shown below.

```
def flipScreenVisibility():
    mActuator = ModelVisibilityActuator('screenVisibilityFlipper', 'ScreenNode', flip = True)
    mActuator.onUpdate() # add Actuator to the ActuatorList in order to be handled on the C++ side

screenVisButton = Button('visButton', 'visButtonMaterial', (128, 128), pos = (10, 10),
                         align = relativePosition.TopLeft, onClick = flipScreenVisibility)
```

The *Button*s are rendered according to the their associated material file by the renderer. In the material file, their color, texture images, and opacity are defined. For every *Button* action event, different material can be defined in order to achieve fancy visual effects. For example, by adding a buttonMaterial.mouseOver material script, a diffent image can be displayed on a *Button* whenever mouse cursor goes over it.

4.12 Extension to the Sensor-Controller-Actuator Structure

Most of the time, the Script Writer accesses to the AR components through Python scripts. However, he sometimes does not need any component access. Rather than sending a message, he may just need to invoke a Python method. He can keep the state of application objects on the Python side in any data structure making use of the Python which is very capable language supporting object-oriented programming by classes. But the data on the Python

side may need to be kept synchronized with the C++ side objects. The Script Writer may keep synchronization and change the state on the Python side on every frame or on an event change.

By extending the Link structure, we provide the ability of Python method registration to the Links. We enable Python methods to be called on any event change. In the Sensor-Controller-Actuator pattern, we add a new field Method to the end of the Link. Similar to the way of sending Actuators to the ActuatorList, the Methods are invoked whenever the sensors are activated and the controllers return true value. The Link structure may involve both Actuators and Methods. They are defined in two seperate lists. The Python methods with arguments may be invoked by adding them to Methods as well. Method list contains (method, [args]) tuples. If method does not need any argument, args list can be omitted.

Example Link:

```
Class Car:
    ...
    def up(self):
        if self.speed <= 4:
            self.speed += 1
            self.update()
    def down(self):
        if self.speed > 1:
            self.speed -= 1
            self.update()
    ...

scene.addLink(Link([KeySensor("", "Up")],
                   ANDController(""),
                   [ ],  #Actuators (empty)
         [(car.up)] #Methods
))
scene.addLink(Link([KeySensor("", "Down")],
                   ANDController(""),
                   [ ],#Actuators (empty)
 [(car.down)] #Methods
))
```

CHAPTER 5

RESULTS

In this chapter, we present a case study in which an AR application has been accomplished by using our proposed AR framework. This case study shows the simple usage of the framework for rapid application development by describing the steps followed for creating an AR advertisement tool which augments physical ad papers and brochures.

In addition, in this chapter, we evaluate the development time of the application, performance, disadvantages, and advantages of script usage.

5.1 Augmented Reality Advertisement Tool

AR Advertisement Tool is designed to be an attractive promotion tool, particularly for commercial products. It is inspired from the Magic Book [5] which displays virtual items on a real book through the AR displays. Basically, this AR Advertisement Tool is an interactive brochure creator. Even though the brochure paper used is not different from a standard brochure in terms of appearance, when it is seen through AR displays, it turns into a material which enables us to see any virtual 3D model and animation on it. The users can see the product's 3D model on the paper from all angles. In addition, 2D videos or images can be displayed in a frame area on the paper. The 3D fictional objects and characters visible on the brochure move as the brochure's real world position changes. Thus, the user gets the chance of seeing the models of the product he wants from any position and angle he wants without using keyboard and mouse but only by translating and rotating the brochure.

AR Advertisement Tool is also designed to enable the users to see the products they want at any place. Particularly for the customers who do not want to spend time in stores, the application may be downloaded and the user may print a copy of the brochure. The commercial products models pop out from the concrete paper brochure as they use it with

their computer in the display.

In this case study, we decided to prepare an advertisement brochure for a car. The user will see a 3D model of the car as he holds the brochure in the camera's range of vision. It will be possible to change the color of the car by key inputs and GUI elements. The 3D models of the interior parts of the car will be displayed on another page of the brochure.

5.1.1 Digital Content Creation

The digital content (3D models, animations, textures) of the application may be produced using external tools. There are various digital content creation tools for creating models, animations, terrains, etc. These tools may use proprietary formats for the digital content. So, they need to be exported to our proposed AR framework's underlying render engine Ogre3D. There are exporters for most of the modelling tools. For our application, the 3D models and animations of the car and its sub-parts are exported to the render engine format.

5.1.2 Layout and Markers

Depending on the brochure page count and layout, we may choose any number of markers up to the maximum number of Markers defined in the Image Processor Component. Since, we have only two pages, one marker for each page and two markers in total are enough for the application. We choose Marker with label 1 and Marker with label 2. The brochure sheets need to be at least as large as the preferred marker dimensions. As the markers get smaller, it becomes difficult to track them when they get farther from the camera. It is best to choose 10 cm square markers for 40-50 cm distance from the camera. The markers are placed on the brochure pages. The remaining empty parts of the pages are reserved for textual and visual information. The pages may be filled in any style without occluding the markers. The final layout of the brochure pages are shown in Figure 5.1 and Figure 5.2.

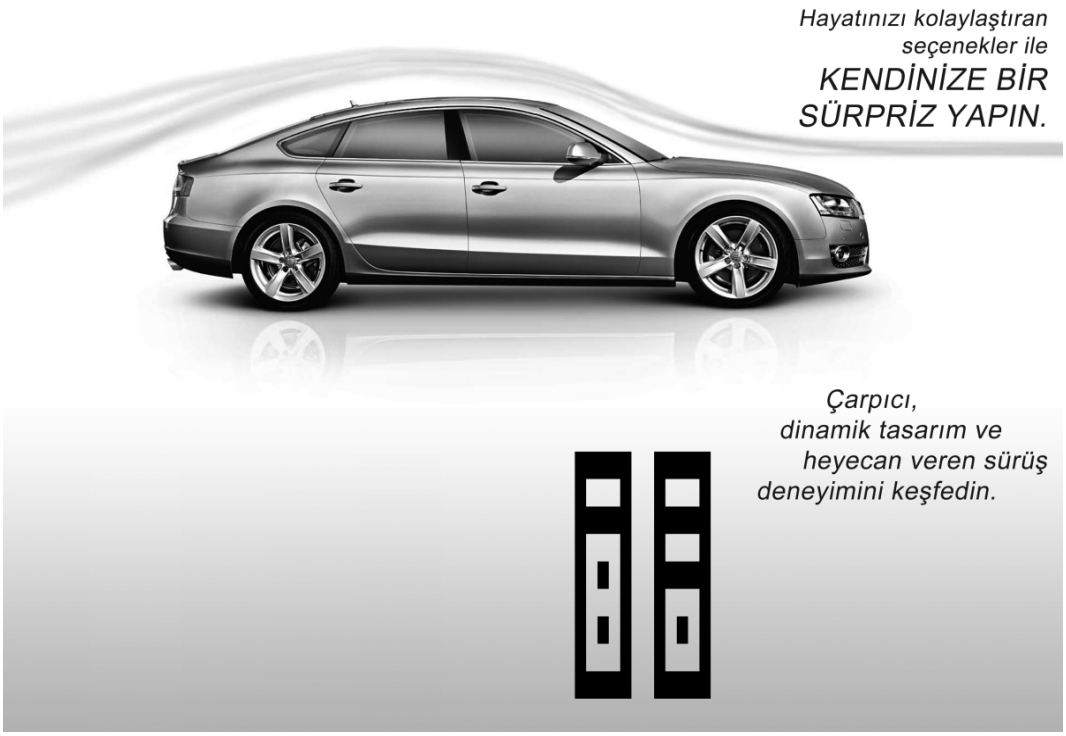

Figure 5.1: The first page of the brochure

5.1.3 Scenes

We separate the pages of the brochure into two different *Scene*s. Then, we create one file for each *Scene*: Scene01.py and Scene02.py.

In Scene01.py file, we create a *Scene* object with name 'Scene01' in Python. Then, the *Link*s, which are needed to create models and form the logical connections, are added to the *Scene*.

Whenever the first *Scene* is loaded, the 3D model of the car is rendered on the AR display. We want a background music play as well. CreateAnimationActuator is used for creating the 3D model. The car model is also scaled and positioned using ScaleSetActuator and PositionSetActuator respectively. In addition we need a virtual light source in order to see the virtual objects in their diffuse colors. Otherwise, they are rendered in only ambient colors. A LightActuator instance can be used to create the virtual point light. The first part of the *Scene* file is shown below:

```
Scene01 = Scene("Scene01")
car_scale = (1.0, 1.0, 1.0)
car_pos = (0, 0, 0) #just center on the marker
car_rotate_sec = 0.3 # 30 degree rotation seconds
car_rotates = True
```

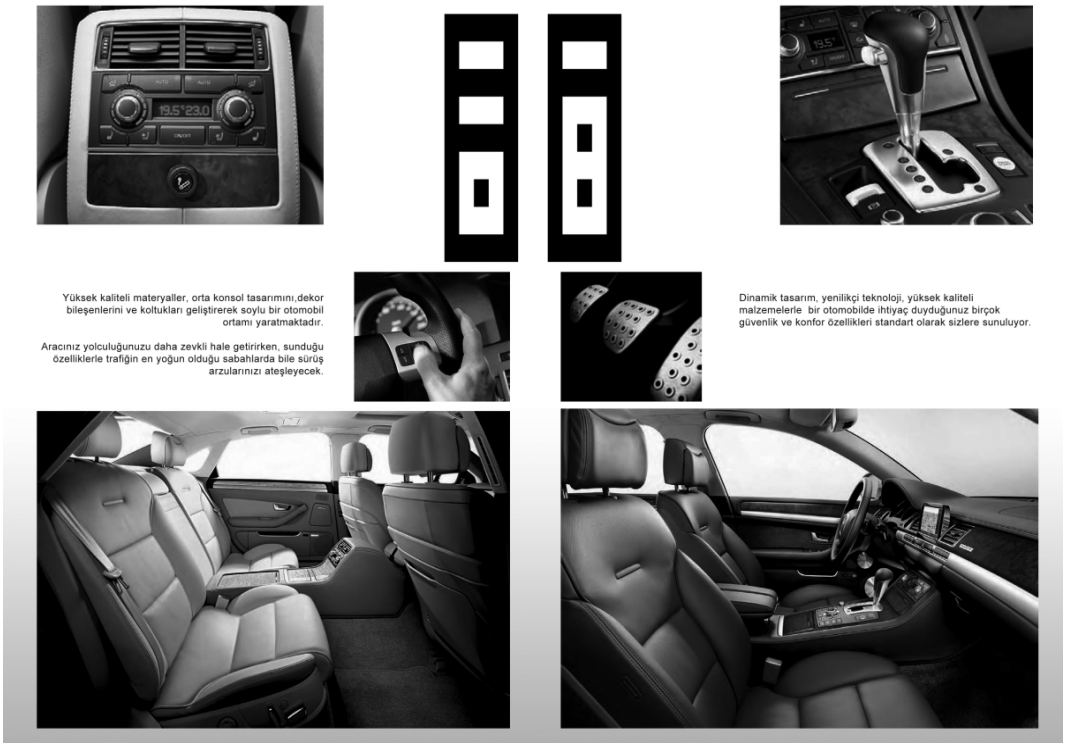

Figure 5.2: The second page of the brochure

```
car_rot_actuator = RotateActuator("CarRot", "Car", (0.0, 30, 0), time = car_rotate_sec)

Scene01.addLink(Link([ActuatorSensor("Scene01LoadedSensor", "onScene01")],
          ANDController(""),
          [
          CreateLightActuator("LightCreator1", "Light1", (500,500,0) , (1, 1, 1), 0)
          ,SoundPlayerActuator("BackgroundMusic", "music01.wav" , play = True, loop = True)
          ,CreateAnimationActuator("CreateCar", "Car", "car-main.mesh")
          ,ScaleSetActuator("CarScale", "Car", car_scale)
          ,PositionSetActuator("CarPos", "Car", car_pos)
          ,car_rot_actuator
          ]))
```

We may add a constant rotate around vertical y-axis perpendicular to the brochure plane. This is achieved by creating a RotateActuator and an ActuatorSensor for perpetual rotation.

```
#continuous rotate link for car
Scene01.addLink(Link([ActuatorSensor("CarRotFinishedSensor", "CarRot")], \
        ANDController(""),
        [
        car_rot_actuator
        ]))
```

We may add a key binding for pausing and restarting the rotation of the car:

```
def carRotationFlip():
    global car_rotates
    if (car_rotates):
        SensorInactivate("", "CarRotFinishedSensor").onUpdate()

    else:
        SensorActivate("", "CarRotFinishedSensor").onUpdate()
        car_rot_actuator.onUpdate()

    car_rotates = not car_rotates

# space key stops/starts rotation
Scene01.addLink(Link([KeySensor("", "Space")],
                    ANDController(""),
                    [ ],
                    [(carRotationFlip)]
                    ))
```

A GUI button may be used for pausing and restarting the rotation of the car as well. The button is placed to the top right position of the screen. The button's visual properties are defined in a material file. The material name is given as second parameter to *Button* class:

```
rotBut = Button("rotB", "rotButMaterial", (50, 50), align = relativePosition.TopRight,
               pos = (10, 10), onClick = carRotationFlip)
```

We choose "tab" key for changing the color of the car from a given color set with (red, green, blue, alpha) values.

```
color_set = [(1,0,0,1), (1,1,0,1), (1,1,1,1), (0.5, 0.5, 0.5, 1)]
color_index = 0
def carChangeColor():
    color_index = color_index + 1
    if (color_index >= len(color_set))
        color_index = 0
    MaterialActuator("CarMaterial", "Car", color_set[color_index]).onUpdate()
# tab key changes color
Scene01.addLink(Link([KeySensor("", "Tab")],
                    ANDController(""),
                    [ ],
                    [(carChangeColor)]
                    ))
```

In addition, we can add another GUI button for changing the color or texture of the 3D car model. This GUI button is positioned below the rotation button. The method *carChangeColor* is registered to the button's *onClick* event.

Figure 5.3: A snapshot of the first scene in AR advertisement tool

```
colorBut = Button("colorB", "colorButMaterial", (50, 50), align = relativePosition.TopRight,
                pos = (10, 100), onClick = carChangeColor)
```

A snapshot of the first Scene is shown in Figure 5.3.

Similar to the Scene01.py file, in Scene02.py, we create a *Scene* object with name 'Scene02'. Then we create *Link*s to form the application logic.

When the second *Scene* starts, a point light source is positioned in the scene and one of the interior parts of the car is displayed on the brochure. A music file starts playing in loop.

```
Scene02 = Scene("Scene02")
Scene02.addLink(Link([ActuatorSensor("Scene02LoadedSensor", "onScene02")],
          ANDController(""),
          [
          CreateLightActuator("LightCreator2", "Light2", (500,500,0) , (1, 1, 1), 0)
          ,SoundPlayerActuator("BackgroundMusic", "music02.wav" , play = True, loop = True)
          ]
          ))
```

Four different sub-parts of the car are modelled and they are exhibited one at a time. For each of these 3D models CreateAnimationActuator, ScaleSetActuator and PositionSetActuator are needed. At the beginning, only one of them is visible.

```
models = [{"name" : "Steer", "mesh_name" : "car_steer.mesh", "scale" : (2, 2, 2)}
    ,{"name" : "Engine", "mesh_name" : "car_engine.mesh", "scale" : (1, 1, 1)}
    ,{"name" : "Interior", "mesh_name" : "car_interior.mesh", "scale" : (1, 1, 1)}
    ,{"name" : "Interior-Detail", "mesh_name" : "car_int_det.mesh", "scale" : (1, 1, 1)}
 ]
models_pos = (0, 0, 0) # all of the 3D models are positioned exactly on the marker.
active_model = 0
Scene02.addLink(Link([ActuatorSensor("Scene02LoadedSensor", "onScene02")],
          ANDController(""),
          [ CreateAnimationActuator("", model["name"], model["mesh_name"], visible = False)
           for model in models
          ] +
          [ ScaleSetActuator("", model["name"], model["scale"])
           for model in models
          ] +
          [ PositionSetActuator("", model["name"], models_pos)
            for model in models
          ] +
          [ ModelVisibilityActuator("", models[active_model]["name"], visible = True) ]
          ))
```

The "space" key is registered for changing the active visible object. It makes the visible object vanish and another object get rendered.

```
def changeVisibleModel():
    ModelVisibilityActuator("", models[active_model]["name"], visible = False).onUpdate()
    active_model = active_model + 1
    if (active_model >= len(models))
        active_model = 0
    ModelVisibilityActuator("", models[active_model]["name"], visible = True).onUpdate()
# space key changes active visible model
Scene02.addLink(Link([KeySensor("", "Space")],
                    ANDController(""),
                    [ ],
                    [(changeVisibleModel)]
                    ))
```

Similar to the rotation and color change GUI buttons, a button may be drawn to the screen for switching between the 3D models. The method *changeVisibleModel* is registered to the button's *onClick* event.

```
changeModelBut = Button("switchB", "switchButMaterial", (50, 50), align = relativePosition.TopRight,
               pos = (10, 100), onClick = changeVisibleModel)
```

A snapshot of the second Scene is shown in Figure 5.4. Depending on the application's extra requirements, any additional program logic may be added to the *Scene* files. Apart from scene creation, the transitions between scenes and common properties of the *Scene*s have to

Figure 5.4: A snapshot of the second scene in AR advertisement tool

be defined. The scenario.py is the Python file that is loaded by the core. In this file, we need to import our two *Scene*s. We want Scene01 is loaded whenever Marker1 becomes visible. Likewise, Scene02 is to be loaded as Marker2 becomes visible. We use *MarkerSensor*s and *ChangeSceneActuator*s in order to switch *Scene*s.

```
# scenario.py
# Scene files are in scenes folder
from scenes.Scene01 import Scene01
from scenes.Scene02 import Scene02

Scene01.addLink(Link([MarkerSensor("Marker2Sensor", 2)],
                     ANDController(""),
                     [ChangeSceneActuator("onScene2", Scene02, listeners)]
                     ))
Scene02.addLink(Link([MarkerSensor("Marker1Sensor", 1)],
                     ANDController(""),
                     [ChangeSceneActuator("onScene1", Scene01, listeners)]
                     ))
```

The whole scene objects are to be drawn relative to the associated marker. The created 3D models are connected to a root scene graph node, "RootSceneNode". For every associated marker observation, this root node is aligned with the marker in the real world image.

```
# aligns RootSceneNode (parent node of all objects)
# with the marker in the real image
Scene01.addLink(Link([MarkerSensor("Marker1Sensor", 1)],
                     ANDController(""),
                     [AlignActuator("", "RootSceneNode")]))
Scene02.addLink(Link([MarkerSensor("Marker2Sensor", 2)],
                     ANDController(""),
                     [AlignActuator("", "RootSceneNode")]))
```

5.2 Evaluation

5.2.1 Development Time

As we use a very high level scripting language (Python), we can express 100 to 1000 instructions per statement. In addition, a set of useful data structures already exists in Python scripting language. Thus, compared to C++, we can write fewer lines of code to express the same application behaviour. Through the less coding, we benefit a reduction in development time. In various types of application development, the scripting version requires less code and development time. The difference varies from a factor of two to a factor of 60 [21]. In our case study, we only write around 100 lines of code. The C++ version would take at least 500 lines of code. If we consider the implementation of the built-in Python data structures such as lists and dictionaries, it would reach 750 lines of code. In addition, there is no compile step in Python. After writing or changing the script file, we can run the main executable in order to test it. This also reduces the debugging time.

Using our proposed framework, 3 interns, who are second and third grade Computer Science students, have developed "Canlı Kitap" applications at Rotasoft company [24]. In average, they have learned the Python and scripting syntax in 1 week. After having the 3D and 2D materials ready, the script writing part has taken 3 weeks time for a book with 16 scenes. The scene files contained 100 lines of code in average.

5.2.2 Performance

AR applications need real-time performance for seemless user interaction. Thus, performance is a critical issue for the AR framework. The graphics rendering can be loaded on the video adapter with hardware acceleration. Yet, the Image Processor requires high central processing unit (CPU) time. Embedding the Python programming language in C++ and implementing a communication layer, gives rise to an overhead. This overhead's size changes depending on the application's instructions.

Python is a slower language than C++. Particularly, the performance difference become obvious in floating point operations, tight loops, and function calls [8]. However, Python can be used as a *glue* language rather than implementation of jobs which need high CPU usage. In our scripting structure, the Python side is responsible for establishing and controlling the logical bricks of the application. We avoid the implementation of the underlying components in Python.

Python script codes are invoked by the notification of the Sensors when an event happens. For example, in our case study, whenever the "space" key is pressed, the KeySensor is notified and ModelVisibilityActuator is added to the queue which is shared by Python and C++. Among the Sensors, only the AlwaysSensor is noticed every frame and the Controllers and Actuators connected to this sensor are activated in every frame. This requires the message transfer in every frame which causes a noticeable drop in frame rate. Still, smooth frame rates are achieved in a 1.83 gigahertz machine with 1024 megabytes (MB) of random access memory (RAM) and a hardware accelerated video adapter that has an exclusive 128 MB RAM. Average, best and worst FPS results for our case study is shown in Table 5.1. The results are taken from Ogre3D log file. We have tested the two scenes separately. In addition, we have tested the empty scene performance with and without the camera. The results show that major drop in FPS is due to scene geometry complexity and image processing. The screen resolution is 640 x 480 and the hardware consists of 1.83 ghz CPU, 1 GB RAM, and NVidia 7300 video adapter.

Table 5.1: FPS results

Scene	Triangle Count	Average FPS	Best FPS	Worst FPS
Empty (w/o camera)	~0	582.98	598.402	568.432
Empty	~0	450.506	481.629	372.627
Scene01	~50.000	150.078	458.541	83.7487
Scene02	~15.000	269.162	451.548	100.298

Scripting Overhead

In order to estimate the overhead caused by the scripting, we have added counters in the Script Manager code where it invokes Python methods. The pinpoints to the Python side

are the listeners. Thus, we have calculated the latencies in milliseconds during the communication to Python listeners. The scripting complexity changes depending on the number of Sensors, Controllers, Actuators, and Python methods. In our case study, the first scene (Scene01) has 5 Links with 5 Sensors, 5 Controllers, 12 Actuators, and 2 methods in total. The second scene has 5 Links with 5 Sensors, 5 Controllers, 24 Actuators, and 2 methods in total. The scripting complexity of the scenes are similar. So, for testing purpose, we added a test scene 100 extra Links having an AlwaysSensor with a Python method just printing short text to the console. The results shown in Table 5.2 are obtained from 20.000 frames. The empty scenes shown in the table contain only 2 active Links. Amongst the listeners, AlwaysListener notification is dominant. Likewise, the increase in latency due to many ActuatorSensors can be seen in the test scene result. As the number of active sensors and actuators increases, the delay caused by the scripting overhead increases.

Table 5.2: Scripting Latency Results

Scene	Total Latency	Duration	FPS	Average Load
Empty (w/o camera)	17 ms	34.46 sec	580	0.05%
Empty	21 ms	41.66 sec	480	0.0504%
Scene01	77 ms	132.15 sec	151	0.0583%
Scene02	61 ms	62.5 sec	320	0.098%
Test (+100 AlwaysSensor)	340 ms	54.05 sec	370	0.63%

5.2.3 Disadvantages

The scripting in Python has few disadvantages apart from the performance drop. There may be problems during script file writing. Since Python forces indentation for scope and flow control, mixing tab and space characters may cause syntax errors. We encountered indentation errors while copying from other Python files. We had import problems with files created in different editors. Because, different end of line conventions in Windows and Unix was problematic while importing the file from the embedded Python. Still, the file format can be set to UNIX format in advance to solve this problem.

Python has no type checking before run-time. This may lead to errors in run-time when an object is wrongly passed as a parameter to a function. But through the Python's excep-

tion message, it can easily be fixed. During the development and testing, we encountered confusing errors in the output file. Still, after tracing the error message, we fixed the bug quickly.

5.2.4 Advantages

The scripting increases the productivity and development quality by making the developer focus merely on the application behaviour. In addition, coding in a scripting language, particularly in Python, is enjoyable. Thanks to its simpler syntax, Python is much easier to learn than C++. Python removes the burden of manual memory management. Moreover, Python has numerous extension modules which can be imported with a single line. For example, if we need an XML parser, we may use the Python's xml module by just writing "import xml".

CHAPTER 6

CONCLUSIONS

The presented thesis work focused on the design of a reusable augmented reality framework. The demonstrative application showed how to develop an AR application by making use of the scripting capability of the AR framework. The high-level scripting capability of the proposed AR framework provides a rapid development environment. As Python scripting language has a simpler syntax than C++, it is easier to learn. In addition, particularly, the abstraction layer between the core and the application through scripting increases the productivity of the application developers by freeing them from the implementation details of the core functionalities.

The designed messaging mechanism between two different programming languages (C++ and Python) brings an important encapsulation level. Embedding Python in the C++ enabled us to benefit from the simple syntax of the Python compared to C++ complex style. Furthermore, instead of a strictly typed static language, the dynamic typing and binding feature of Python provides a much free environment. The code writing takes much less time by avoiding compile time and speeding up the edit-test cycle. The developers may express the orders with fewer lines of code by high level instructions of the scripting language.

The scripting has significant memory and instruction overhead which may reduce the application's speed. Particularly, the tight loops on the Python side may degrade the performance considerably. However, the operations that need tight loops can be implemented on the C++ side and exported to the Python side. Embedding a dynamic language like Python or exporting methods and classes to the Python accomplishes flexible structure for rapid application development.

6.1 Future Work

6.1.1 Object-oriented Programming in Scripting

The scripting language, Python, in the AR system, has a very high-level syntax. However, the coding style is not object-oriented. It is like event-driven programming where the flow of the program is determined by events. The current Sensor-Controller-Actuator structure does not hold the objects in a data structure which causes the loss of object states. The objects are reached through their ID, and the operations on the objects are executed through actuators. For example, a 3D car model object is created by CreateAnimationActuator and it is positioned by using PositionSetActuator which takes the object ID and the position as parameters. However, *car.setPosition(pos)* would be an object-oriented way to change the position of the car. In current implementation, Python classes can be created to associate with the objects. In addition, the Actuators can be hidden in object's class functions. For example, a *Car* class can be written to hold the state of a car such as its velocity, position, and visibility. The methods of *Car* class can make use of Actuators. For example, *setPosition* method may create a PositionSetActuator and invoke its *onUpdate* method. When the scene is loaded the car instance registers its functions and actuators to the *Scene* instance. In order to force object-orientation in the scripting, a generic solution can be provided by an *Object* class implementation on the Python side and all of the 3D models and animation classes may inherit from this class. The script statements sometimes seem confusing because of long Link creation strings. It may become clearer in imperative style with object instances.

6.1.2 Automatic Script Generation

The script writer have to know the Python programming language for building scripts. However, for simple AR application and development and prototyping, the scripts can be automatically generated through a graphical user interface by a user without any programming language knowledge. The *Link* structure and its coding is suitable for creation through a GUI. Blender game engine [6] provides a GUI with buttons to create logical bricks for game development. In a similar way, a user may create AR application by just selecting appropriate Sensors, Controllers, and Actuators in a GUI panel.

6.1.3 Run-time Coding

A shell access to the embedded Python may provide extra flexibility, particularly for prototyping and testing. While an AR application is running in the background, we may change

the application behaviour through a simple shell. In order to change the objects' properties, we may create Actuator instances and activate them. The shell may be implemented as a separate Python thread that is created by the embedded Python.

6.1.4 Support for other Scripting Languages

The same scripting structure can be developed for other scripting languages such as Lua, Ruby, and Perl. The Script Manager's interface to the Python can be altered in order to embed an alternative language. The underlying components and the handlers of the Actuators do not need to be modified. The classes and methods on the Python side can be defined and implemented in the alternative scripting language.

REFERENCES

[1] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments 6*, pages 355–385, August 1997.

[2] Ronald T. Azuma, Y. Balliot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *IEEE Computer Graphics and Applications*, pages 34–47, November 2001.

[3] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reichner, S. Riss, C. Sandor, and M. Wagner. Design of a component-based augmented reality framework. In *ISAR'01*, pages 45–54, 2001.

[4] R. Bencina, M. Kaltenbrunner, and S. Jordà. Improved topological fiducial tracking in the reactivision system. In *Proceedings of the IEEE International Workshop on Projector-Camera Systems (Procams 2005), San Diego (USA)*, 2005.

[5] Mark Billinghurst, Hirkazu Kato, and Ivan Poupyrev. The magicbook - moving seamlessly between reality and virtuality. *IEEE Computer Graphics and Applications, vol. 21, no. 3*, pages 6–8, May/June 2001.

[6] Blender 3d. `http://www.blender.org/`, 2002. Last accessed, August 2009.

[7] Boost portable c++ libraries. `http://www.boost.org/`, 1999. Last accessed, August 2009.

[8] Bruce Dawson. Game scripting in python, 2002.

[9] Morten Fjeld and Benedikt M. Voegtli. Augmented chemistry: An interactive educational workbench. In *International Symposium on Mixed and Augmented Reality (ISMAR'02)*, pages 259–321, 2002.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Reading, 1995.

[11] G. Hiebert. Openal 1.1 specification and reference. `http://connect.creativelabs.com/openal/Documentation/OpenAL%201.1%20Specification.pdf`, 2005. Last accessed, August 2009.

[12] G. Hiebert. Wmvideo video player for ogre3d. `http://connect.creativelabs.com/openal/Documentation/OpenAL%201.1%20Specification.pdf`, 2005. Last accessed, August 2009.

[13] Gregory Junker. *The Ogre 3D Programming*. APress, 2006.

[14] M. Kaltenbrunner and R. Bencina. Reactivision: A computer-vision framework for table-based tangible interaction. In *Proceedings of TEI'07. Baton Rouge, Louisiana*, 2007.

[15] H. Kato, M. Billinghurst, I. Poupyrev, K. Imamoto, and K. Tachibana. Virtual object manipulation on a table-top ar environment. In *International Symposium on Augmented Reality (ISAR'00)*, pages 111–119, 2000.

[16] Hirokazu Kato, Mark Billinghurst, and Ivan Poupyrev. *ARToolKit User Manual Human Interface Technology Lab, University of Washington*. 2000.

[17] Embeddable common lisp. `http://ecls.sourceforge.net/`. Last accessed, August 2009.

[18] Mark Lutz. *Programming Python*. O'Reilly Media, Inc, 2006.

[19] Microsoft. Directshow system overview. `http://msdn.microsoft.com/en-us/library/ms783354(VS.85).aspx`, 2009. Last accessed, August 2009.

[20] Nintendo. Nintendo wii. `http://www.nintendo.co.uk/NOE/en_GB/systems/about_wii_1069.html`, 2008. Last accessed, August 2009.

[21] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer, vol. 31, no. 3*, pages 23–30, March 1998.

[22] Perl programming language. `http://www.perl.org/`. Last accessed, August 2009.

[23] W. Celes R. Ierusalimschy, L. H. de Figueiredo. Lua 5.1 reference manual. `http://www.lua.org/manual/5.1/`, 2006. Last accessed, August 2009.

[24] Rotasoft company. http://www.rotasoft.com.tr/, 2007. Last accessed, August 2009.

[25] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalav´ari, L. Miguel Encarnacao, M. Gervautz, and W. Purgathofer. The studierstube augmented reality project. *Presence, Vol. 11, No. 1*, pages 33–54, February 2002.

[26] Steve Streeting. Object-oriented graphics rendering engine (ogre) 3d. http://www.ogre3d.org, 2000. Last accessed, August 2009.

[27] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.

[28] Tcl (toolkit command language). http://www.tcl.tk/. Last accessed, August 2009.

[29] Bruce Thomas, Ben Close, John Donoghue, John Squires, Phillip De Bondi, and Wayne Piekarskiet. Arquake: An outdoor/indoor augmented reality first person application. In *Proceedings of the 4th International Symposium on Wearable Computers*, pages 139–146, 2000.

[30] James R. Vallino. *Interactive Augmented Reality*. PhD thesis, University of Rochester, New York, 1998.

[31] Microsoft, visual basic for applications. http://msdn.microsoft.com/en-us/vbasic/default.aspx. Last accessed, August 2009.

[32] WreckedGames. Object oriented input system. http://sourceforge.net/projects/wgois/, 2009. Last accessed, August 2009.

[33] S. You, U. Neumann, and R. Azuma. Hybrid inertial and vision tracking for augmented reality registration. In *Proceedings of IEEE Virtual Reality*, pages 260–267, 1999.

[34] Stefan Zerbst and Oliver Düvel. *3D Game Engine Programming*. 2004.

[35] Feng Zhou, Henry Been-Lirn Duh, and Mark Billinghurst. Trends in augmented reality tracking, interaction and display: A review of ten years of ismar. In *International Symposium on Augmented Reality (ISMAR'08)*, pages 193–202, 2008.