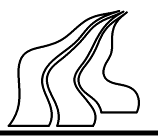
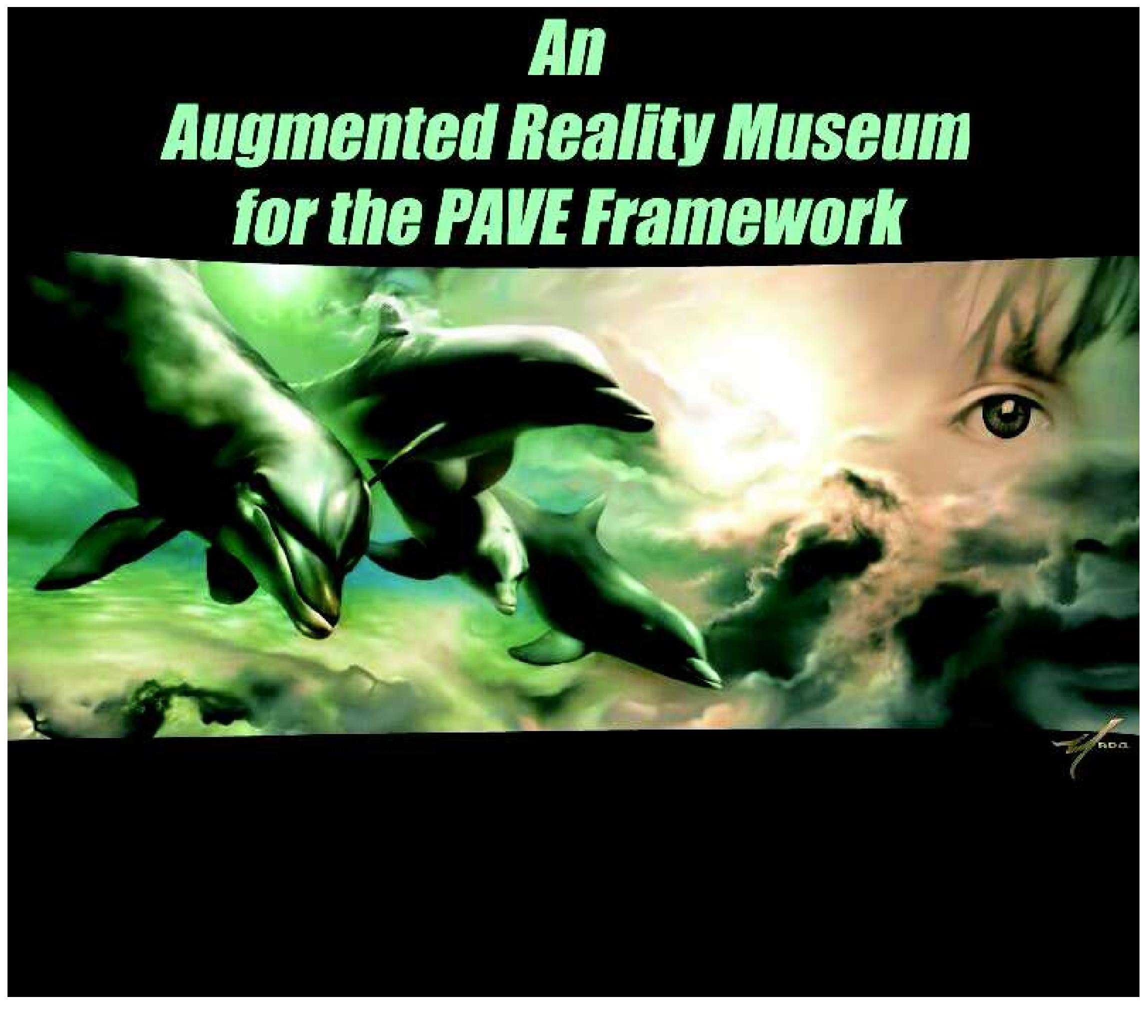
Aalborg University
Institute of Computer Science

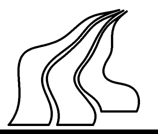
Fredrik Bajersvej 7E ▪ DK-9220 Aalborg Ø ▪ Phone: +45 96 35 80 80





Group E3-101A, DAT6 project, spring 2001.

Aalborg University
Institute of Computer Science



Fredrik Bajersvej 7E ▪ DK-9220 Aalborg Ø ▪ Phone: +45 96 35 80 80

Theme: Distributed Systems,
Computer Vision and Virtual Reality

Title: An Augmented Reality Museum for the
PAVE Framework

Period: Feb 4th 2001 to
Jun 13th 2001

Project members:

Rune Elmgaard Laursen
Mikkel Kierkegaard Stausholm
Henrik Lunardi Weide

Supervisors:

Anders P. Ravn
Claus B. Madsen

Copies: 10

Pages: 90

Abstract:

This report covers the analysis, design and implementation of an augmented reality (AR) museum prototype. The system is built as modules for the PAVE (Parallel Architecture for Visual Effects) framework, and hence the performance of the system scales on multiprocessor PCs with a measured gain of 54% from 1 processor to 2 processors. The system utilizes 3D graphics hardware for visualization in real time and supports binding of dynamic visual effects to 3D objects. The system runs in real time with a minimum frame rate of 9,1 frames/second on the test machine used. The design covers the overall system architecture, enhancements made on the PAVE framework and the description of suitable algorithms to meet the requirements for the AR museum.

Delivered: Jun 13th 2001

Copyright © 2001, Institute of Computer Science, AAU.

Aalborg Universitet
Institut for Datalogi

Fredrik Bajersvej 7E ▪ DK-9220 Aalborg Ø ▪ Tlf: +45 96 35 80 80

Fagområde: Distribuerede systemer, Computer Vision og Virtual Reality

Titel: An Augmented Reality Museum for the PAVE Framework

Periode: 4.feb 2000 til 13.jun 2001

Projektdeltagere:

Rune Elmgaard Laursen
Mikkel Kierkegaard Stausholm
Henrik Lunardi Weide

Projektvejledere:

Anders P. Ravn
Claus B. Madsen

Oplag: 10

Antal sider: 90

Synopsis:

Denne rapport omhandler analyse, design og implementation af et prototype “augmented reality” museum. Systemet er bygget som moduler til PAVE (Parallel Architecture for Visual Effects) frameworket, og ydelsen af systemet skalerer på multiprocessor PC’ere. Forbedringen af ydelsen er målt til at være 54% fra 1 processor til 2 processorer. Til visualisering i reel tid udnytter systemet 3D grafik hardware og supporterer tilknytning af dynamiske visuelle effekter til 3D objekter. Design aspekterne omhandler den overordnede system arkitektur, udvidelser foretaget på PAVE frameworket og beskrivelsen af passende algoritmer til at imødekomme kravene til et “augmented reality” museum.

Dato: 13.jun 2001

Copyright © 2001, Institut for Datalogi, AAU.

Preface

This report is the result of group E3-101A's project work done on the DAT6 semester, in the spring of 2001, at the Institute of Computer Science at Aalborg University. This report describes the analysis, design and test of a prototype AR system built as modules in the PAVE (Parallel Architecture for Visual Effects) framework.

We wish to thank: Flemming N. Larsen, Aalborg University, for showing us some basics about 3D StudioMAX scene modelling. Peter Hounum, Preben S. Nielsen and Sven Vestergaard at AM:3D for lending us a Polhemus FASTRAK for several critical weeks. Karin Husballe Munk, Aalborg University, for lending us a Polhemus ISOTRAK.

Aalborg University, Denmark
June 13th, 2001

Rune Elmgaard Laursen

Mikkel Kierkegaard Stausholm

Henrik Lunardi Weide

Contents

1 Introduction **11**
1.1 Motivation . . . 11
1.2 Project Goals . . . 12

2 Related Work **15**

3 Analysis **17**
3.1 What is Augmented Reality? . . . 17
3.2 Typical AR systems . . . 18
3.3 Choosing the AR Museum . . . 20
3.4 The AR Museum . . . 20
3.4.1 Requirements . . . 22
3.4.2 Delimitations . . . 23
3.5 Summary . . . 23

4 Baseline **25**
4.1 Hardware . . . 25
4.2 Software . . . 26
4.2.1 Calculation of Camera Field of View . . . 27
4.2.2 Rendering Pipeline . . . 28
4.3 The PAVE framework . . . 30
4.3.1 Render Model and Data Flow . . . 30
4.3.2 Modules . . . 31
4.3.3 Parallelization in PAVE . . . 32
4.3.4 PAVE Design . . . 34
4.3.5 The Generic Components . . . 38
4.3.6 Graph description Components . . . 40
4.3.7 The Communication and Syncronization Components . . . 41
4.3.8 Management Components . . . 45
4.3.9 Execution engine Components . . . 47
4.4 Summary . . . 49

5 Design **51**
5.1 Architectural Style . . . 51
5.2 PAVE Design Enhancements . . . 53
5.2.1 Optional Inputs and Outputs on Nodes . . . 53

5.2.2 Trigger Capable Node . . . 54
5.2.3 Empty Triggering . . . 55
5.2.4 Garbage Collection . . . 56
5.2.5 Read-Only and Writable Data . . . 57
5.3 Components . . . 59
5.3.1 Data Structures . . . 59
5.3.2 World Model Module . . . 61
5.3.3 Magnetic Tracker Module . . . 61
5.3.4 Camera Module . . . 63
5.3.5 Distortion Correction Module . . . 64
5.3.6 Frustum Culling Module . . . 65
5.3.7 World Object Module . . . 67
5.3.8 World Update Module . . . 68
5.3.9 Render Module . . . 69
5.3.10 Summary . . . 71

6 Experiments **73**
6.1 Performance Scaling Test . . . 73
6.1.1 Test Setup . . . 75
6.1.2 Test Results . . . 75
6.2 User Impression Test . . . 75
6.2.1 Test Setup . . . 75
6.2.2 Test Results . . . 75
6.3 Magnetical Tracker Precision Test . . . 78
6.3.1 Test Setup . . . 78
6.3.2 Test Results . . . 78
6.4 Test Conclusion . . . 80

7 Conclusion **81**

8 Future Work **83**
8.0.1 Enhanced Tracking . . . 83
8.0.2 Multiple Users . . . 83
8.0.3 Hardware Accelerated Image Distortion . . . 83
8.0.4 Shadows . . . 84
8.0.5 Depth of Field Blur . . . 84

A BaseModule Specialization Example **85**

1 Introduction

Last semester we designed and implemented a framework, called PAVE (Parallel Architecture for Visual Effects), that supported parallel real time rendering of graphics in a generic fashion. The motivation was that a lot of current real time graphics applications are hardcoded to support particular hardware configurations to maximize rendering speed. Most multimedia programs offer utilization of e.g. 2 CPUs by design, but typically not for more CPUs. Therefore, we wanted to contribute to the real time graphics area, by developing a general model for parallel rendering that could be used to create scalable real time graphics applications.

As a consequence we designed a dataflow model that supports generic parallel execution, useful for making e.g. graphics rendering performance scalable over multiple CPUs on a given machine. The number of CPUs that can be utilized is not limited by the dataflow model and the way we have designed the parallel execution engine. Futhermore the dataflow model was designed as a modular framework that can be used by a programmer to create different kinds of real time graphics rendering applications. The point was that a programmer should only worry about creating modules (plugins) according to a template interface. The modules can then be used as components in e.g. a larger graphics rendering scheme, by plugging them into an dataflow graph and the parallel execution will then be taken care of by the framework. The ideal goal was to make it easier and less error prone for a programmer to create performance scalable graphics software with our framework, by adhering to a small set of simple rules, and not worry about parallelization issues.

1.1 Motivation

Now that we have a framework for creating scalable graphics applications, we find it to be a natural step to create a graphics application ourselves and use the framework as the underlying software platform. Particularly we want to create an application that has high demands in terms of acceptable graphics rendering speed and at the same time have a large room for potential enhancements.

This in particular and the fact that we have an interest in real time computer graphics for entertainment and art, let us to look into the area of creating augmented reality (AR) systems. In the following we will describe our specific goals.

1.2 Project Goals

The goal is to design and implement a simple augmented reality (AR) system, where a real room acts as the frame of reference to a virtual world. In order to have a potential entertaining product as a goal, we will use the concept of a museum. We will call this an *AR Museum*. We define it to be the following:

- It should be possible for a spectator, to inspect a given room in the real world, and on a computer screen or a head mounted display observe the room augmented with virtual objects in real time.
- Objects in the virtual world must appear, as if they were part of the real world and hence follow the cameras view. In essence the objective is to blend computer graphics with input from a camera in a suitable manner to obtain these goals.
- The virtual objects could be artistic pictures on the walls, sculptures standing at certain positions etc. The virtual objects could be animated in some fashion in order to give the spectator a more entertaining experience and to further enhance reality.

The performance goals, that we have for the AR Museum, can be summarized in the following:

- Real time presentation. Since a spectator must be able to freely move around in a room and observe events, the presentation speed, also called rendering speed, must have a minimum acceptable lower bound.
- Performance scalability. The system must be able to scale over multiple processors, in order to increase rendering speed.
- The system should utilize special graphics hardware that can assist in accelerating the graphics rendering, and ideally do it in a fashion that favours parallel execution.

It is our intention to create such a system by analyzing what components are neccessary, designing and implementing them as modules that can be run in the PAVE (Parallel Architecture for Visual Effects) framework, which is introduced in chapter 4. We will design and implement neccessary generic enhancements of PAVE in order to support the creation of the AR Museum. The reason that we wish to use PAVE as the underlying architecture, is its inherent ability to scale on multi

processor PCs, and hence give the AR system better performance scalability.

There are two issues that we will consider. The first aspect of problems is dealing with the architecture of the AR system, like dataflow between components and what role each component in the AR system must have. This is influenced by the way PAVE works in terms of its inherent features and underlying modular design philosophy.

The other aspect of problems are based on each components role in the AR system, and therefore consists of applying suitable algorithms to the components. Datastructures to pass information between the various algorithms, will be addressed as well.

2 Related Work

In the field of Augmented Reality, many different application areas exist. People working with the area have different focuses, some contributes to the area by developing new more robust algorithms for e.g. camera tracking, others focus on visual quality etc.

In augmented reality systems, a main issue for achieving acceptable results is that registration of the users position and gaze must be relatively precise, so that augmented objects appear in the correct position.

A calibration-free AR system, using optical tracking that relies on tracking at least four non-co-planar points in the filmed image is described in [KV98].

The task of making a fast and robust tracking algorithm that combines magnetical tracking and optical tracking is addressed in [SHC$^+$96].

The *StudierStube*, described in [SFH00], is an example of a multi user AR system. Its main focus is on building multi user AR environments / user interfaces. It also comprises a tracking algorithm which combines optical and magnetical tracking. This is described in [APG98]. In *StudierStube* head-mounted see-through displays are used for augmentation.

Another AR system that focuses on more users doing collaborative work, is described in [Rek96]. Here a hand-held see-through display is used to view and interact with augmented objects.

An AR system with the same application context as in our project, building a museum / gallery, is described in [MKN96]. Here the focus is in particular the interaction and communication between visitors, virtual guides and the experts behind the exhibition.

Our approach to creating an AR museum differs in focus from the above. In our

project we apply magnetical tracking for determining position and gaze of the user. As we have created our AR museum, we have done it under the conditions that the system must be scalable over multiple processors and thereby part of the focus has been on creating a flexible and robust dataflow software architecture. We have made our system capable of handling real time graphical effects on virtual objects. The handling of the graphical effects has been made a part of the systems dataflow architecture, in order to make optimizations and enhance scalability at a higher level than on particular effect algorithms. Lastly, we have made the architecture modular in a strong sense, so that alterations and enhancements of the system are possible and supported. The whole system runs on a single computer and takes advantage of multiple processors.

Analysis

In this chapter, we begin by describing what augmented reality is, and what a typical augmented reality system contains and its general requirements. Then we describe the choices made in order to conceive the AR Museum as defined in section 1.2 and give an overview of the elements it must contain. Finally, requirements and delimitations for the AR Museum are given.

3.1 What is Augmented Reality?

Augmented reality (AR)[1] describes reality that is augmented with virtual objects. An augmented reality system presents the spectator with a composite view of the world, that consists of the real scene viewed by the spectator combined with a virtual computer generated scene that augments the real world with additional information.

In [Val01] the relation between augmented reality and virtual reality is defined as a Reality-Virtuality Continuum, shown in figure 3.1.

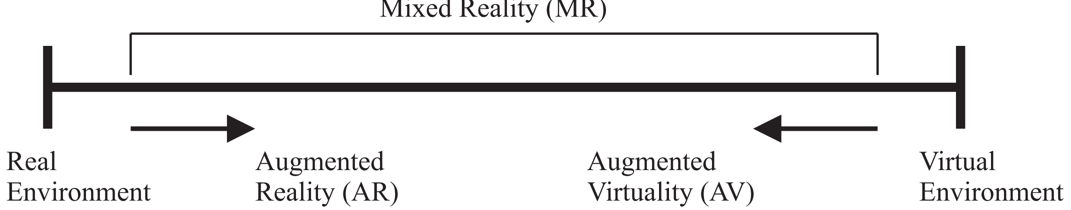


Figure 3.1: *Milgram's Reality-Virtuality Continuum*

Augmented reality lies somewhere in between reality and virtual reality. It lies closer to reality than it is to virtual reality, the predominant perception being the real environment, augmented with virtual objects.

[1] Augmented reality will in this report be denoted AR.

Furthermore, [Val01] proposes a taxonomy for categorizing mixed reality systems, i.e. system that creates an environment which is somewhere in between pure reality and pure virtuality. It consists of the three axes: *Reproduction Fidelity*, *Extent of Presence Metaphor*, and *Extent of World Knowledge*.

Reproduction Fidelity is a measurement of the image quality of the objects that augment reality. As it is required for AR systems to run in real time, the result of rendering images with today's graphical hardware, is far from the photorealism, one could wish for. This puts AR in the low end of the Reproduction Fidelity scale. An ultimate and ambitious goal would of course be to make the objects that augment the real environment indistinguishable from reality.

Extent of Presence Metaphor describes the degree of immersiveness that is felt by a spectator when looking at the displayed scene. The degree of imersiveness will, of course, be highly dependant of the display technique used in the AR system. There is a great difference in the feeling of looking around **in** an AR environment, and looking **at** an AR environment.

The *Extent of World Knowledge* dimension, in the mixed reality categorization proposed by Milgram, measures how much information about the real world is available to the system. In AR systems it is imperative that accurate registration of objects in the world can be maintained. The real and virtual parts of the world must be combined so they match as a whole, as an augmented reality. For this to be possible, the AR system typically must have information about the frames of reference for the real world, the camera viewing it and the spectator. The less world knowledge needed by an AR system, the more robust and flexible it is, as it can be used in changing and different settings.

3.2 Typical AR systems

Although AR systems have many different application contexts with different demands of the system, several key components are common to a typical AR system. These are described in the following.

The main task of the AR system is to register the real scene viewed by the spectator and the corresponding virtual scene, merge the two scenes correctly and display the result, all this in real time. The real scene is viewed by an imaging device, a video camera or the human equivalent, the spectator's eyes. Likewise the virtual scene containing the objects to augment the real scene, is viewed by a virtual camera, and is rendered by the computer. These two "cameras" must be aligned, so that the rendering of images from the virtual camera is correctly performed. This means that the virtual camera must know the intrinsic (focal length, lens distortion) and extrinsic (position, orientation) parameters of the real camera. Merging of the

two images can then be done, producing the augmented reality images to display. In figure 3.2, an overview of a typical AR system is presented.

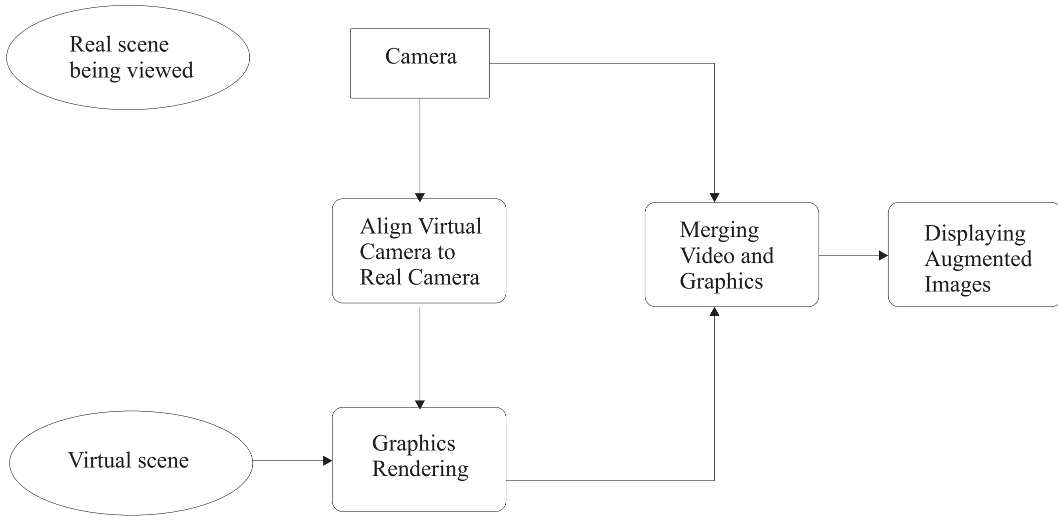


Figure 3.2: *Overview of a typical AR system*

To correctly align the two cameras, so that the virtual scene is rendered from the same point of view as the real scene, the position and orientation of the real camera must be tracked in real time as the spectator moves around. A common method for tracking, is using orientation angles and coordinates coming from a magnetical tracker. This method can be somewhat inaccurate due to low resolution in the tracked orientation, limited range and disturbances in the magnetic field created by the tracker. A combination of magnetical tracking and optical tracking based on computer vision algorithms, has been used to obtain higher accuracy. Such hybrid tracking is described in [SHC+96, APG98].

As mentioned above different techniques for displaying the augmented images can be used, giving different degrees of imersiveness. The simplest technique is to use a monitor for displaying the augmented reality. This is referred to as Fish Tank virtual reality. Alternatively, see-through head-mounted displays (HMDs) can be used. HMDs come in two forms. The equivalent of the monitor display is the video see-through HMDs, which contains a camera. They project the augmented image on the displays, like on a monitor. With optical see-through HMDs the spectator can actually see through, meaning only the virtual objects are displayed, the rest of the display is transparent, giving the spectator a free view of reality.

To conclude the overview of AR systems, we summarize the general performance criteria of an AR system. As real time high quality visualization of the augmented scene, in which the spectator must be able to move around in, is imperative to an

AR system the three following criteria are put on the system:

- Frame rate of the rendered virtual images.
- High visual quality rendering of the virtual objects.
- Accuracy in registration of real and virtual images.

A frame rate of minimum 10 frames per second are considered acceptable for the virtual objects in the AR scene not to appear jumpy. A low accuracy in registering the images will result in the virtual objects not being perceived as stationary. If, in the AR system, time delays occur in computing the alignment of the virtual camera, the virtual objects will lag behind the motion in the real world. To put it in short, in an AR system, speed, accuracy and low latency is of the essence. High visual quality and high frame rate in the rendering of virtual objects is linked directly to the *Reproduction Fidelity* measurement, and there is a trade-off between image quality and frame rate. The better image quality, the lower the frame rate and vice versa. Here 3D render acceleration hardware is useful, in order to speed up rendering of virtual objects without stressing the main CPU(s) too much.

3.3 Choosing the AR Museum

To choose an AR application case, we need to find one where the PAVE[2] architecture can be exploited to its full potential in terms of parallelization. Since PAVE was designed with parallelization of real time graphical effects in mind, we find that an AR application embodying such effects should be an obvious choice. We have therefore chosen a museum as our case, as this provides a possibility for experiencing dynamic real time effects, or computer art, in a fitting environment, which should give the spectator a more immersive experience. We also wanted a simple application, which would not rely on interaction between the spectator and the augmented world. A virtual museum is well justified without interaction, but it is of course an area of great possibilities for enhancing the spectators experience. Furthermore a virtual museum filled with dynamic real time art would call for a large amount of processing power and therefore the automatic scalability of PAVE makes it a suitable platform. Later on, we will explain how real time graphical effects can be tied into the AR Museum. In the following we will setup the requirements for the AR Museum.

3.4 The AR Museum

The AR Museum is seen as an arbitrary real world room with virtual objects placed in it. The virtual objects constitute the art in the museum and should ideally be

[2]Introduced in chapter 4.

seen as existing in the room. There should be no real restrictions on the virtual objects. They can be static or dynamic in the sense that they change properties such as e.g. form or material which will allow for e.g. pictures on the wall with dynamic content. An interface between dynamic effects and such objects should be designed, so frame data generated by algorithms consistuting the effects can be linked to an object's properties. This could be used for updating the material on a virtual object.

The museum should be able to be viewed from any position and orientation desired, which is achieved by using a magnetical tracker to determine position and gaze of the camera. We delimit the system from using optical tracking, such as using e.g. image analysis algorithms to gain world knowledge. As a consequence the AR Museum will have a relatively low *Extent of World Knowledge*. The only world knowledge the system must have apart from the camera position and gaze, is a given rooms geometry.

Since we have had no access to see-through head-mounted displays, the museum is presented on a monitor and the real world is captured by a standard web camera. This is also the reason why the system will be delimited from supporting multiple users/spectators.

The knowledge about the world (museum) is contained in a structure called the *World Model*. It holds information both about reality and virtual reality. Part of the reality description includes the picture grabbed by the camera and information about the cameras position and gaze. It also includes a description of the static geometry of the world such as walls, ceilings, floors and possible lightsources. The knowledge of the virtual reality includes descriptions of the geometry of the art objects present in the museum and their properties such as position and material. In the process of constructing the World Model, we find that adding support for using a modeling tool, e.g. 3DStudio MAX, is an important feature, since it allows easier creation of complex geometry.

To enhance the overall performance of the AR system we have chosen to use 3D graphical hardware for rendering the virtual objects. For this purpose we will use the Microsoft Direct3D API, which encapsulates hardware accelerated rendering of 3D primitives. Furthermore we will focus on enhancing performance by detecting objects that are not present in the cameras field of view. This information can be used for deactivating dynamic effect algorithms, linked to certain virtual objects, which generates frames used exclusively by these objects.

For merging the camera image of the real world with the computer generated image of the virtual objects, camera calibration must be performed to compensate for lens distortion, for this we need a *Camera calibration* component, that delivers calibrated camera images to the World Model structure. The cameras position and orientation read from the magnetical tracker must be collected by a *Magnetic*

tracker component, that delivers the data to the World Model structure. When the information about the real world is known, based in camera input, position and orientation and geometry descriptions encapsulated in the World Model structure, *Virtual object clipping* must be done for objects that are outside the cameras view. Finally, a *Rendering* component, that displays all the collected information on screen is need. See figure 3.3 for a conceptual overview of the elements we imagine the AR Museum must contain. The small circles denotes objects in the virtual world, such as pictures on the walls or sculptures. The small graphs linked to each virtual object, symbolizes dynamic effect algorithms, where each effect is tied to a given objects material.

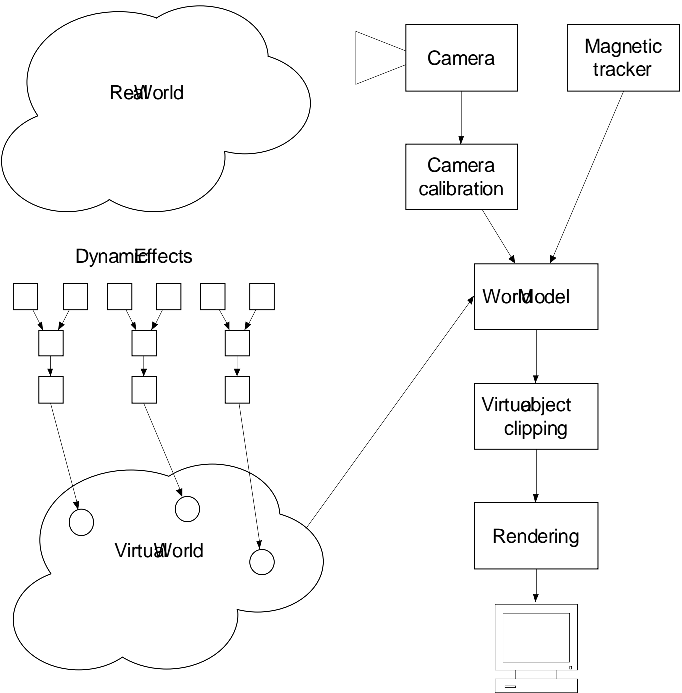


Figure 3.3: *AR Museum overview.*

Based on the above the specific requirements for the AR Museum, that will influence the design and choice of algorithms for the components is summarized below along with the delimitations we have made.

3.4.1 Requirements

Specific requirements for the AR Museum are:

- One camera for filming the real world.
- Magnetic tracker is used to track the position and orientation of the camera.

- Calibration of the images grabbed by the camera.
- Dynamic virtual objects.
- Interfacing between dynamic effects and virtual object properties.
- Culling of virtual objects that are out of sight and dynamic effects tied to those objects must be deactivated.
- Graphic rendering must take advantage of 3D hardware.
- World Model geometry can be described in a modeling tool.

3.4.2 Delimitations

The delimitations of the AR Museum are:

- Single user system.
- No interaction with objects in world model.
- The augmented result is seen on a standard monitor.
- No optical tracking.

3.5 Summary

We have described what augmented reality is, and listed the general requirements for AR systems. We have setup requirements and made delimitations for our specific application case, the AR Museum. The general requirements and specific requirements, will form the focus in the AR Museum design. Figure 3.3 shows how we imagine the AR Museum must work conceptually. We will in the next chapter present the baseline for the AR Museum, before the design is presented in chapter 5.

4 Baseline

In this chapter we describe our baseline, consisting of the hardware, software and programming tools we will use to create the AR Museum. Following after, we give a description of how we calculate a cameras field of view. After that we explain some of the fundamental principles in 3D graphics rendering. Finally, we describe the PAVE framework, which will become the foundation of the AR Museums architectural design.

4.1 Hardware

Based on the requirements for the AR Museum, we defined in section 3.4.1 on page 22, we choose the following hardware.

- Polhemus 3SPACE FASTRAK magnetical tracker
- Creative Video Blaster WebCam 3 USB
- PC with two 400 MHz Intel Celeron CPU's, 128 Kb cache each, 128 MB 66 MHz RAM and a NVIDIA TNT2 3D graphics card.

In figure 4.1 on the next page the hardware setup for the AR Museum system is depicted.

The magnetical tracker is connected to the computer via a serial port. It has a magnetical transmitter that creates a magnetical field around it. The receiver connected to the magnetical tracker measures the magnetical field so that its position and orientation can be determined. The receiver is placed on a wooden stick behind the camera. The reason for this is that the camera generates too much electromagnetic noise for the receiver to be placed on top of the camera. The Polhemus 3SPACE FASTRAK has the following specifications:

- **Position Coverage:** 76 cm distance from receiver to transmitter with specified accuracy and up to 305 cm with slightly reduced accuracy.

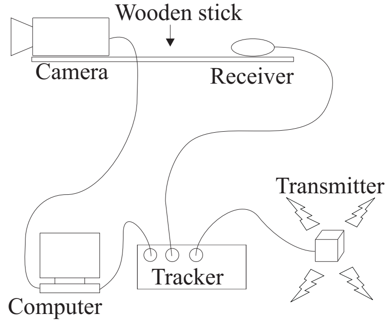


Figure 4.1: *The AR museum hardware setup.*

- **Static Accuracy:** 0.08 cm RMS for the x, y and z receiver position, 0.15 degrees RMS for receiver orientation.
- **Resolution:** 0.0005 cms/cm of range and 0.25 degrees.
- **Latency:** 4.0 milliseconds from receiver measurement to output.
- **Update Rate:** 120 updates/second.

The camera used is a standard commercially available low-price web camera connected to the computer via the USB port. Its specifications in our setup are:

- **Frame Rate:** Maximum of 30 frames/second.
- **Resolution:** 320x240 pixels with 24bit color depth.
- **Field of View:** horizontal field of view: 43.6 degrees, vertical field of view: 34.4 degrees.

The field of view values have been empirically obtained, as described in section 4.2.1 on the facing page.

The specifications for the FASTRAK tracker and the Creative camera can be found at [FAS00] and [Cre01], respectively.

4.2 Software

The software platform the AR Museum system will be based upon, is listed below:

- The MS Windows 2000 operating system.
- The DirectX 8.0a subsystem (includes Direct3D) in Windows 2000.
- The C++ programming language.

- The PAVE (Parallel Architecture for Visual Effects) framework, introduced in section 4.3.

Now that we have presented the hardware and software, we will describe how we have obtained the two field of view angles for the camera. Following after we will explain the basic principles in a 3D graphics render pipeline, which is related to the way the Direct3D subsystem works and hence influences how we design the visualization algorithms of the virtual objects in the AR Museum. Lastly, we will introduce the PAVE framework.

4.2.1 Calculation of Camera Field of View

To ensure that the real camera and the virtual camera are aligned, thus filming the same part of the scenes, the virtual camera must have the same horizontal and vertical field of view as the real camera. The field of view (FOV) is the angle covered by the lens. It can be calculated as shown below:

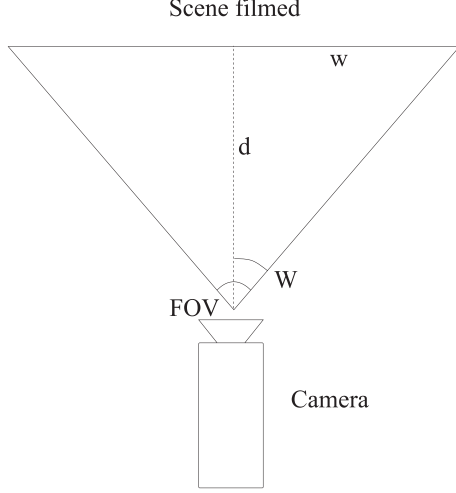


Figure 4.2: *Field of view calculation.*

In figure 4.2 w is the distance from the center to the edge of the scene filmed, d is the distance from the camera to the scene, and angle W is half the FOV. In a right-angled triangle, it holds that:

$$tan(W) = \frac{w}{d} \tag{4.1}$$

Hence, FOV can be expressed as:

$$FOV = 2arctan\left(\frac{w}{d}\right) \tag{4.2}$$

The above equation has been used to calculate the FOV for the camera used, by using distances w and d obtained empirically. This method is used to calculate both the horizontal and the vertical FOV.

4.2.2 Rendering Pipeline

This section gives a brief explanation of some of the elements used for rendering 3D graphics. A 3D primitive is a collection of vertices that form a single 3D entity. The simplest primitive is a collection of points in a 3D coordinate system. Throughout this report we describe 3D graphics using a left-handed cartesian coordinate system (see figure 4.3). Often, 3D primitives are polygons. A polygon

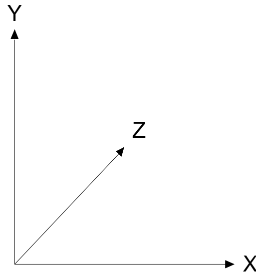


Figure 4.3: *Left-handed cartesian coordinate system.*

is a closed 3D figure delineated by at least three vertices. The simplest polygon is a triangle which can be combined to form large, complex polygons and meshes (a collection of polygons). To enhance the realism of computer-generated 3D images a texture can be mapped onto a polygon. A texture is a bitmap which is strecthed onto the polygon by specifying what coordinates the vertices maps to in the bitmap.

A 3D model (one or more meshes) is described in *model space* which is a frame of reference that uses vertices relative to the 3D model's local origin. For visualizing the 3D model, it must be sent through the rendering pipeline which applies three transformations, the world, view, and projection transformations, to it.

The first stage of the pipeline transforms a model's vertices from their local coordinate system to a coordinate system that is used by all the objects in a scene. The process of reorienting the vertices is called the *world transformation*. This new orientation is commonly referred to as *world space*, and each vertex in world space is declared using world coordinates.

In the second stage, the vertices that describe the 3D world are oriented with respect to a camera. That is, a chosen point-of-view for the scene, and world space coordinates are relocated and rotated around the camera's view, turning world space

into *camera space*. This is the *view transformation*.

The third stage is the *projection transformation*. In this part of the pipeline, objects are usually scaled with relation to their distance from the viewer in order to give the illusion of depth to a scene; close objects are made to appear larger than distant objects, and so on. This transformation can be seen as projecting 3D coordinates into 2D space. The projection transformation can also be said to describe a *viewing frustum* which is an 3D volume in which a scene is positioned relative to the camera. For perspective viewing, the viewing frustum can be visualized as a pyramid (see figure 4.4), with the camera positioned at the tip. This pyramid is intersected by a front and back clipping plane. The volume within the pyramid between the front and back clipping planes is the viewing frustum. Objects are visible only when they are in this volume.

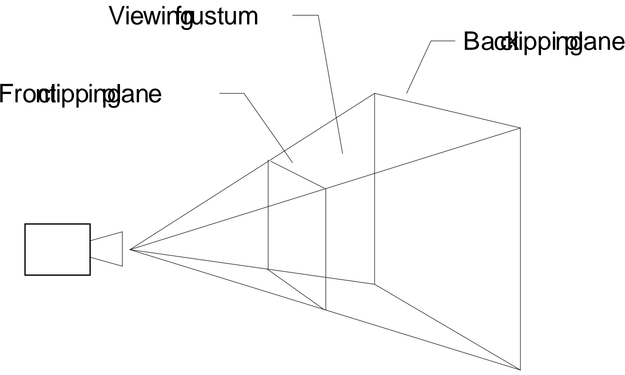


Figure 4.4: *Viewing frustum.*

In the final part of the pipeline, any vertices that will not be visible on the screen are removed, so that it does not take up time to calculate the colors and lighting for something that will never be seen. This process is called clipping. After clipping, the remaining vertices are scaled according to the window viewport and converted into screen coordinates. The resulting vertices seen on the screen when the scene is visualized exist in *screen space.*

To handle occlusion of the drawn polygons a *depth buffer* can be used. A depth buffer is holding depth information for each pixel on the screen. Whenever a pixel is drawn, its depth value is checked against the value in the depth buffer. If the value is smaller than the value in the buffer the pixel is drawn and the new depth value stored in the depth buffer, otherwise the pixel is discarded.

The Microsoft Direct3D API, which is a drawing interface that provides access to 3D video-display hardware in a device-independent manner, will be used for

rendering the graphics used in the AR museum. It provides mechanisms for using the above described elements plus many more.

4.3 The PAVE framework

PAVE is a framework supporting parallel execution of tasks in a generic fashion, where the tasks can be algorithms in graphics rendering. We will describe the key aspects of the underlying model in PAVE and describe the design of the components in PAVE that have a direct influence on the architectural style and algorithmic functionality of the AR Museum as it is introduced in section 1.2 and further explained in section 3.4.

PAVE was designed to facilitate parallel rendering of visual effects out of the requirement that speed is important in graphics rendering, especially for realtime purposes. Another requirement was that it should be relatively simple for a programmer to add new well-defined render components (denoted modules) to PAVE without having to think about parallelization issues. Hence the parallelization is designed to be generic and is taken care of by the underlying PAVE architecture. The last requirement, we had, was that an end user should be able to take the modules, made by the programmer, and connect them together in a dataflow graph to form his/her own composition of graphical algorithms. It was very important to us that the concept of a dataflow graph consisting of render components, should both be intuitive to a user and at the same time reflect the algorithmic dataflow.

4.3.1 Render Model and Data Flow

The process of rendering can be seen as, in turn, applying a number of algorithms to the data that must be rendered. In the context of an AR system, the data is typically bitmap data, world model descriptions etc., that is, graphical data that is supposed to be rendered to a computer display. The algorithms are typically those that generate bitmap data, world model descriptions and/or manipulate such data structures. The rendering process can be divided into subtasks, which must be evaluated in some predefined order. Rendering lends itself very easily to partitioning into subtasks, as the graphical manipulation algorithms commonly used are designed for one small specific purpose, such as a gaussian blur filter algorithm. The task partitioning is implicitly given, as each algorithm can be seen as a subtask itself, and it is the composition of the graph by the end user that defines the order of subtasks.

Task Graph

The order of the tasks describe the flow of data. As mentioned this can be represented as a **task graph**, as it is done in [CT95]. We call the data flowing between the tasks **frames**. A frame can literally be any datatype, even a reference to a datatype instance. The rendering flow starts with one or more **input nodes**, which

produces input frames, that travel through a number of algorithmic tasks and end in one or more **output nodes**. The input and output nodes are the start and end points of the graph, respectively. An input node, is a node that has no predecessors in the graph. The data it works on comes either from an external source or resides in the node itself. An output node has no successors, and its data is typically output on an external device, such as a display.

To have a meaningful graph representation, the graph must be directed, showing the direction in which data flows. This leads to using a directed acyclic graph, see [Cor98], as our task ordering representation. In figure 4.5, an example task graph is shown.

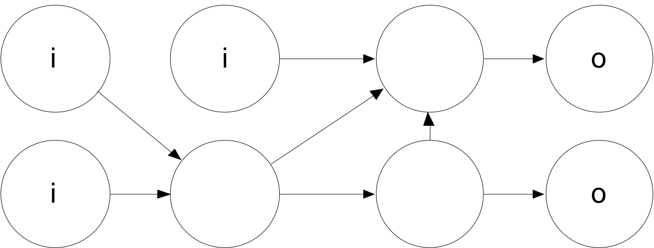


Figure 4.5: *An example of a task graph.* **i** *denotes input nodes,* **o** *denotes output nodes*

4.3.2 Modules

We have denoted the tasks of the graph **modules**. Each module in the graph is conceptually considered as a "black box". The role of a module is to apply its algorithm to the input frames it receives from its predecessors, and pass on the result to its successors.

Furthermore, the end user should easily be able to adjust a module, so that the graphical effect the module represents, can appear to his liking. For that purpose, we introduced **Module states**. To any given time a module is in a certain state, determined by a number of **parameters**, which the end user can change the values of. The decision of which parameters that exist on a given module, is made by the programmer depending on the algorithm he uses in the module he is creating. For instance we can imagine the programmer creating a module that can take bitmap frames (images) as input, blur them with a Gaussian Blur algorithm and then output blurred bitmap frames. An example of a parameter on such a module could be "Blurriness", that describes how blurry the module's Gaussian Blur algorithm should make each input image.

The algorithm of the module manipulates the input data, given the state of the module. The manipulated result is delivered as the output.

We have defined the input/output relationship in a module as follows:

- Receive i inputs from its p predecessors, where $i \geq p \geq 0$. Wait until every input from the predecessors has arrived.

- Apply the algorithm to the input, according to the state.

- Send o outputs to its s successors, where $o \geq s \geq 0$. If one or more of the successors are busy, wait until all successors are available to receive.

As we will explain in section 4.3.7, this input/output definition is the key to the syncronization between modules in a task graph.

4.3.3 Parallelization in PAVE

The parallelization in PAVE is based on two parallization methods, called **Functional Parallelism** and **Temporal Parallelism**, as discussed in [Cro97].

Functional Parallelism is achieved by splitting the rendering up into smaller distinct functions, which is then applied to a sequence of data frames. If a processor is then assigned to a function or a group of functions, called a functional unit, and data frames are communicated between the functional units, this constitutes a pipeline. When the first data frame, that have entered the pipeline, have reached the last functional unit of the pipeline, all functional units will be working in parallel, hence the degree of parallelism is proportional to the number of functional units and relies on repeated inputs i.e. a stream. Furthermore it is the slowest unit in the pipeline that will determine the overall speed of the rendering. This parallelization technique fits the task graph concept in a very straightforward way. In our framework a functional unit is a module.

Temporal Parallelism comes from partitioning the rendering task in the time domain (e.g. by frame index number). The sequence of frames to be rendered are split into frame sequence subsets. A frame sequence subset can consist of one or more frames. Each frame in a subset is rendered concurrently, by having a replication of the rendering unit working on each frame. In our case this means that a given module in the graph is replicated a number of times to obtain "local" temporal parallelism for that given module. The temporal parallelism in PAVE is optional in the way, that a programmer can choose to specify that a certain module must be replicated a number of times if the given module is a potential bottleneck in the pipeline.

Se figure 4.6 for an illustration of Functional Parallelism and "local" Temporal Parallelism respectively.

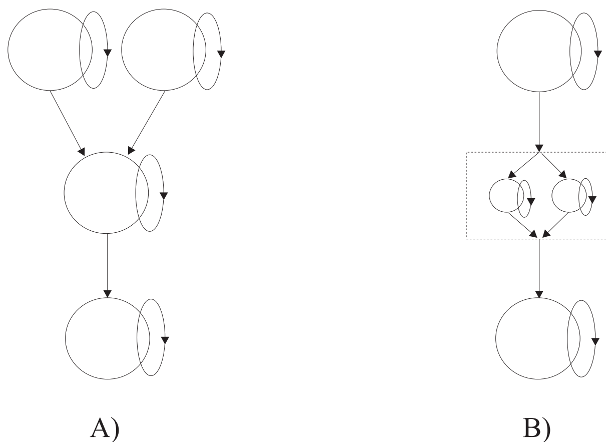


Figure 4.6: *A) The default Functional Parallelism. B) Temporal Parallelism by replication. The rectangle depicts two identical module replica working on each their frame.*

Task scheduling

A task can be a module or a group of modules. The main problem of scheduling tasks, is that of assigning tasks to processing units, in a way that optimizes the processing. A typical parameter, one wishes to minimize is the total execution time of the scheduled tasks compared to the sequential total running time, and hence dividing the tasks as optimal as possible among the processing units.

In [Ull75], it is shown that the general problem of scheduling a set of tasks is *NP*-complete, unless some task parameters are constrained, one of these parameters could be that all tasks must have the same execution time (as it is done in [CT95]). In the PAVE framework there is no actual knowledge of what goes on inside a module, and no static measurement of the execution time of modules, as this differs hugely depending on the module's input, algorithm and state. As we want to support different kinds of events, such as changing parameters, that in turn modifies the state of a module over time in a dynamic fashion, this causes modules to have varying running times per frame.

From these observations we chose to abstract away the scheduling of modules, and let the Windows 2000 SMP[1] kernel do the scheduling. This is done in the framework by using threads[2] as the processing units. The Windows 2000 scheduler is capable of scheduling threads between a pool of available processors. Every time a thread is ready to be scheduled it is assigned to one of the available processors for execution for a given time slice. This means that when PAVE is running on e.g. a dual-CPU machine, two treads are always running concurrent, one for each of the

[1] Symmetrical Multi Processing as defined in [Sta97]

[2] In abstraction, a thread can be seen as a virtual processing unit.

two processors. The threads are set to have the same priority, which is the default priority of the process creating the threads. The Windows 2000 kernel utilizes a round-robin scheduling policy.

In addition, test results showed us that minimizing the number of threads in an application under Windows 2000 did not provide any significant benefit in terms of less overhead and better speed. Last semester, we tested the Windows 2000 scheduling capabilities by running a process that created up to 2000 threads, partitioning a job into the same number of parts as threads, each part having one thread running it. It showed no significant overhead in terms of completion time of the overall job by going from one thread to 2000 threads. As a consequence the default behaviour in PAVE is to assign a seperate thread of execution to each module in a task graph, as illustrated in figure 4.6 where the arcs shown at each module denotes a thread. That way the scheduler in Windows 2000 takes care of all the scheduling, and modules that have precedence over other modules will be evaluated first automatically. Modules that are independent of each other are automatically run in parallel, and a module that has multiple parents will wait until all of its parents have delivered data. This is possible, since the input/output relationship in a module works as a syncronization mechanism. A module is only allowed to invoke its algorithm when all inputs are present and only allowed to deliver outputs when all its children are ready to receive them.

4.3.4 PAVE Design

We have introduced the key aspects and definitions of the render model in PAVE. Now we will describe how the render model has been designed as an object-oriented framework to support modular parallel graphics rendering.

The PAVE design can be said to consist of five major parts, each part consisting of a number of classes. In the following we will introduce the five parts and describe their overall roles. After that we will go into describing the classes in each part in more detail.

Management

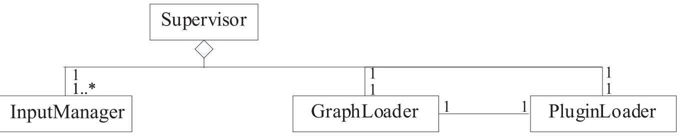


Figure 4.7: *The management classes.*

The management part consists of classes that that manages the whole system. The **Supervisor** class is instantiated by the main application that wishes to use PAVE. The Supervisor utilizes a **GraphLoader** to load a graph description from a text file. The **PluginLoader** is used to load the plugins used in the loaded graph. A plugin contains a collection of modules. The **InputManager** is used to trigger the graph, and is controlled by the Supervisor. The Supervisor is controlled by the main application. The management classes can be seen in figure 4.7.

Graph Description

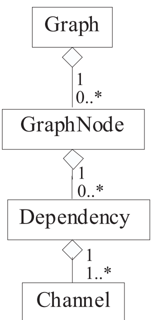


Figure 4.8: *The graph desciption classes.*

The graph description classes are the ones who are instantiated by the GraphLoader when a graph is loaded from a text file. Instances of those classes consistute the complete **Graph** description, such as dependencies between **GraphNodes**, and for each **Dependency** a number of **Channels** exist. The classes are depicted in figure 4.8.

Communication and Syncronization

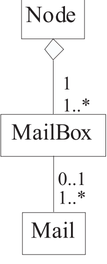


Figure 4.9: *Communication and syncronization classes.*

Three classes form the data communication between modules in a graph. The **Node** class contains the input/output relationship algorithm defined in section

4.3.2. A Node instance contains a **MailBox** for each input it has. For each module in a graph, there is a Node instance belonging to it, and for each Node instance there is a GraphNode instance belonging to it. When data is communicated between modules, a **Mail** containing the data is sent from one Node to another. The three classes are shown in figure 4.9

Generic Components

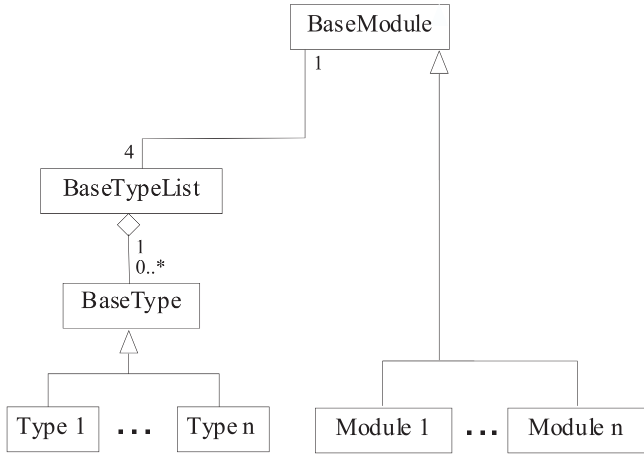


Figure 4.10: *The generic classes.*

The generic components contains two base classes and a container class. A module programmer that wishes to contribute new modules or new datatypes to the framework, must inherit from **BaseModule** and **BaseType** respectively. These classes define the interface for the module programmer. The **BaseTypeList** is a utility container class used both by the communication classes and the programmer to specify module inputs and outputs. On figure 4.10 the three classes are shown, and possible specializations of them.

Execution Engine

The three classes, **Threader**, **WorkerThread** and **Job**, works as the execution engine. The Threader creates WorkerThreads to execute a graph. A **Job** tells a WorkerThread what nodes in the graph it must execute. See figure 4.11 for the relationships between these classes.

The PAVE Classes

In figure 4.12 a class diagram depicting all the components and their relationship, in the PAVE framework, can be seen. We will now continue to explain each class in detail, starting with the generic component classes.

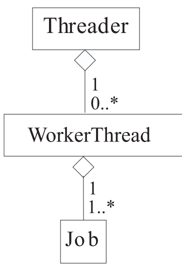


Figure 4.11: *The classes consituting the execution engine.*

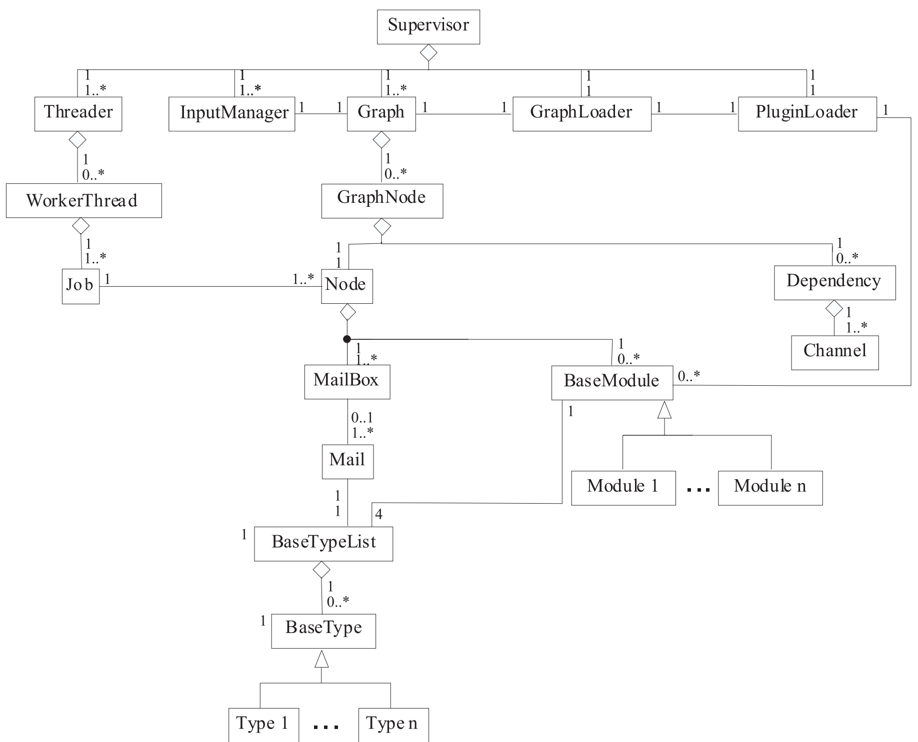


Figure 4.12: *Class diagram illustrating relationships between components in the PAVE framework.*

4.3.5 The Generic Components

Here we will describe the generic component classes in more detail.

BaseType

BaseType is the super class for all data types that are passed between modules in a graph. When a new type is made by the programmer, the BaseType is specialized and the neccessary attributes and methods are added. The framework does not need to know the interface for derived types since it only passes them between modules and does not interact with their methods or data. It does however need to be able to get an id for a derived type so it can check if modules can receive that type. It is the programmer's responsibility to ensure that each new type created is assigned a unique type id. On derived types a *clone* interface method has to be implemented so the framework can duplicate types generically by calling the *clone* method. The BaseType has a **FrameIndex** and a **TimeStamp** as attributes, which are used when instances of BaseType specializations (frames) are passed between modules. Modules receiving frames can use this information to determine the current frame index and the time for that particular frame index.

BaseTypeList

Can contain a list of BaseType specializations. This is a container class used in the communication between modules, and used by the module programmer to declare and utilize inputs and outputs.

BaseModule

The BaseModule is the super class for all modules that can be inserted in a graph in the framework. It is inherited for each new module, a programmer wishes to write. This class defines the template, that is the interface, for all modules. For writing a BaseModule specialization, defining the modules functionality and exposing information about its input/output interface and unique type id to the framework, is necessary. The constructor of the derived module must specify type information for the following:

- Unique type id
- Inputs - The number of inputs and the type of each.
- Outputs - The number of outputs and the type of each.
- Parameters - The number of parameters and the type of each.

Specifying inputs, outputs and parameters, is done by adding empty *BaseType* specializations to predefined lists (BaseTypeList containers) for inputs, outputs and parameters, respectively. The *unique type id* is specified by setting an internal base

string attribute. The *type id* of the BaseType specializations added are used for type checking in the framework. The order in which BaseType specializations are added to the lists defines the numbering of each input/output/parameter element exposed to the framework. For instance if a BaseType representing a bitmap frame type is added first to the input list then that frame input will be seen as input number one of type *bitmap frame* on the module. The values of the specialized BaseTypes added to the parameter list defines the module's initial state.

In addition to implementing the constructor, two interface methods must be implemented in the specialization:

- Init
- Action

Init is implemented when the programmer wishes to define initialization for a module after the constructor has been called, but before executing the module algorithm for the first time. The *Action* method is implemented to specify the module's functionality. The *Action* method in a BaseModule specialization contains the module's algorithm, that is called for each frame index. When the *Action* method is called, input and parameter BaseTypeLists are passed to it and when finished, it must return an output BaseTypeList. For instance, if the programmer wants to create a blur module, the *Action* method will contain e.g. a Gaussian Blur filter algorithm. The algorithm manipulates a bitmap frame delivered at input number one and puts the resulting bitmap frame into an output list and returns it, for each frame index.

The three above mentioned base classes form the fundamentals when it comes to creating new render modules for PAVE and creating datatypes that such new modules might need. It is actually the only things that a module programmer needs to be concerned with. In appendix A, a small example of how a BaseModule specialization is implemented, is shown.

Until now we have occasionally talked about modules as being the nodes in a graph, although this is conceptually true, we needed some kind of abstraction from a module and a node in a graph. We decided to seperate the communication aspects of a module and the internal algorithm of the module. The internal algorithm part is contained in a BaseModule specialization as mentioned above. In addition we have seperated the communication aspects and graph integrity properties also. As a consequence we have two additional classes, each with their distinct roles. They are called Node and GraphNode. A Node contains a BaseModule specialization instance and takes care of the communication and syncronization between its module and other Nodes. The GraphNode represents a vertex in a graph, and it contains a reference to a Node instance and information such as dependencies between the vertices. See figure 4.13 for an illustration of the run-time relationship between a

GraphNode instance, Node instance and its BaseModule instance. The associations between the instances are made at initialization and build time of the graph. The WorkerThread shown is created by the Threader component and enters through a ThreadEntry method in Node. GraphNode, Node, WorkerThread and Threader along with the other components will be explained below.

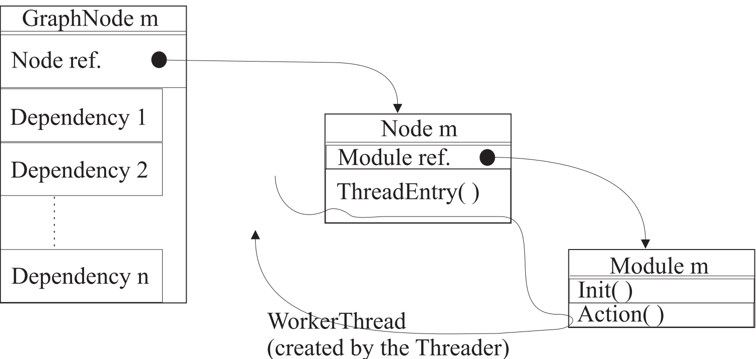


Figure 4.13: *Run-time relationship between GraphNode, Node and BaseModule instances. Inputs and outputs are not shown.*

4.3.6 Graph description Components

Here we will describe the graph description component classes in detail.

GraphNode

A GraphNode is a vertex in a graph. It holds a Node and lists of Dependencies to other GraphNodes. The GraphNode has methods for building the connections in the graph, and for checking its consistency, i.e. that all nodes are connected properly. The GraphNode can add and remove Channels to another GraphNode. When a Channel is added, it is checked if the source and destination MailBoxes, on the source GraphNode's Node and the destination GraphNode's Node respectively, holds data of the same BaseType specialization by inspecting the type id. Otherwise, a Channel cannot be created. If a Channel is added, and no Dependency exists to the destination GraphNode, a Dependency is created and the Channel is added to it. Likewise, if all Channels in a Dependency are removed, the empty Dependency is removed.

The GraphNode has a method for returning its dependencies to the Node instance it gets at creation time (from the GraphLoader), so the Node can send mails to the right recipients.

Dependency

A Dependency describes a dependency connection to a GraphNode. In other words, it is an edge in the graph of GraphNodes. A Dependency holds a reference to a GraphNode, to which the dependency exists, and a list of Channels, between the GraphNode that holds the Dependency (source) and the one referenced in the Dependency (destination). When a Dependency holds several Channels, it means that several outputs are sent to different (input) MailBoxes on the receiving Node in the destination GraphNode.

Channel

A Channel describes a connection between an output and an input of two nodes. It holds two numbers, source and destination, which are the index number of a MailBox on each node, respectively.

Graph

This class represents and contains a graph with connected GraphNodes. It has methods to build and modify the graph, check its integrity such as detecting cycles, find paths from one graph to another, add and remove dependencies between GraphNodes.

The Graph consists of a list of GraphNodes. An example graph is shown in figure 4.14.

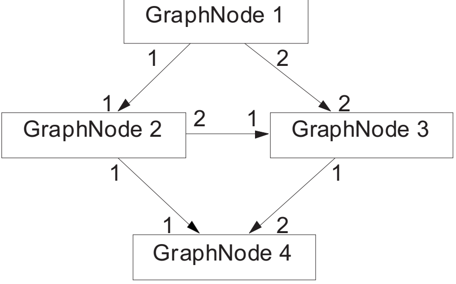


Figure 4.14: *An example graph. The numbers on the arcs denote input and output index numbers, respectively.*

The contents of the GraphNodes it contains, are shown in figure 4.15.

4.3.7 The Communication and Syncronization Components

Here we will describe the communication and syncronization component classes in detail.

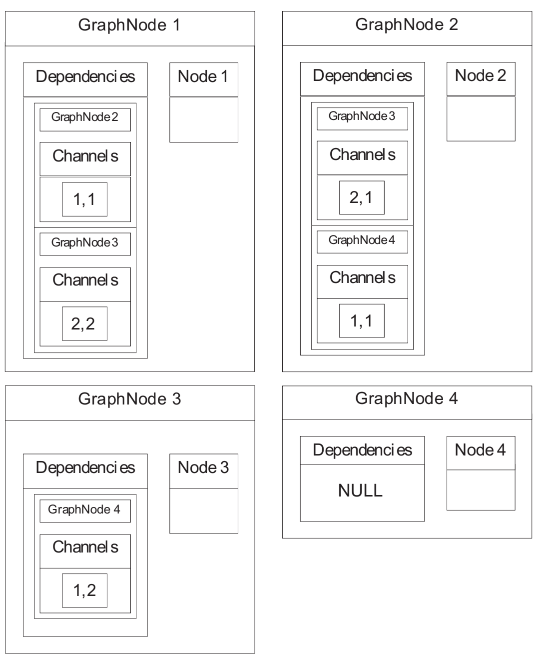


Figure 4.15: *The contents of the four GraphNodes in figure 4.14.*

Node

This class is perhaps the most central component in PAVE's render core, as it contains the syncronization between modules. It encapsulates an instance (more instances if replication is present, see 4.3.3) of a BaseModule specialization. It controls all the input and output to/from its encapsulated module, it contains the input/output relationship syncronization algorithm presented in 4.3.2. The algorithm is situated in a *ThreadEntry* method that is called by an associated WorkerThread. The Node holds a MailBox for each input the module has. When the module is executed the Node retrieves a Mail from each MailBox and inserts these into the module's input list. When the module has completed its processing of the input data, it returns an output list to the Node which then asks its parent GraphNode, what mailboxes to send the output to. If more than one Channel exists for an output, that data for that output must be cloned for every Channel to prevent succeeding modules in the graph from writing in the same data concurrently.

The realization of temporal parallelism in the framework is handled in the Node by replication of modules i.e. several instances of the same type of module. All instances of modules inside a Node are put in a ready queue and when a WorkerThread wants to execute a module, it enters the Node and takes the first module off the ready queue and executes its action method through the *ThreadEntry* method. After execution the WorkerThread puts the module back at the end of the ready queue. This way a number of WorkerThreads can be assigned to a Node for

executing each of the replicated modules (see figure 4.16).

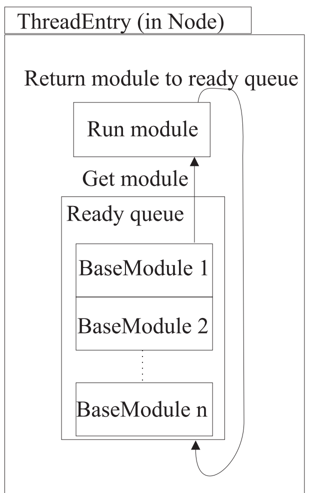


Figure 4.16: *Illustration of a ready queue inside a Node.*

When a Node holds replicated modules it is neccessary to have a mechanism in *ThreadEntry* for syncronizing modules so they pass on their outputs in the same order as inputs are received on the Node. This mechanism was inspired by token ring networks. The modules can be seen as nodes in a token ring and only the module who has the token is allowed to send its output. The first module that retrieves input from the mailboxes also receives the token. When it has sent its output it sends the token to the module that retrieved input after it, and so forth.

In the following the *ThreadEntry* method for executing a Node's module(s) is described in object oriented pseudo code:

```
ThreadEntry method (for WorkerThreads):
{
  InputList = empty BaseTypeList;
  BaseModule = ReadyQueue.GetModule();

  EnterCriticalSection;

  // get mail tuple
  for each MailBox
  {
    BaseType = MailBox.GetBaseType();
    InputList.AddBaseType(BaseType);
  }

  ExitCriticalSection;

  TimeStamp = InputList.GetTimeStamp();
```

```
  ParameterList = GetParameterList(TimeStamp);

  OutputList = BaseModule.Action(InputList, ParameterList);

  WaitForToken;
  for each BaseType in OutputList
  {
    BaseType.SetTimeStamp(TimeStamp);
    MailBox = GraphNode.GetNextOutputMailBox();
    MailBox.PostMail(BaseType);
  }
  if (an OutputList was returned from the module's Action method)
    delete OutputList;
  delete InputList;
  SendTokenToNextModule;
}
```

It should be noted that the critical section is necessary for ensuring that replicated modules does not receive mails with the same timestamp. All mails with the same time stamp must be retrieved by the same module.

MailBox

The MailBox is used by a Node for sending data between modules and holds a FIFO list of elements of type Mail. A MailBox is created in Node for each input its module has specified. At creation time, it is possible to specify how many elements the MailBox should be able to hold. After it has been created the size remains fixed. Methods for posting and retrieving mails from the MailBox are blocking in the following sense:

- If a client call tries to retrieve a mail from the MailBox and it is empty, the call blocks until a mail arrives.
- If a client call tries to post a mail to the MailBox and it is full, the call blocks until a mail has been retrieved by another call (from another thread).

All access is protected by mutexes.

Mail

A container for BaseTypes, used for packaging frames between Nodes. It contains the following:

- BaseTypeList of BaseType specialization elements.
- Time stamp
- Frame index

The BaseTypeList allows that user defined types can be sent via Mails.

4.3.8 Management Components

Here the management component classes are described in detail.

Supervisor

The Supervisor acts as the main book keeping component in the framework. It is the component that glues the Graph, InputManager and Threader components together and manages them. In addition the Supervisor holds the PluginLoader. The Supervisor is also intended as the interface between the main application and the rest of the framework.

The Supervisor manages a Graph instance by associating it to an InputManager and a Threader, giving a 3-tuple as follows:

(**Graph**, **InputManager**, **Threader**).

The overall roles that the Supervisor has are the following:

1. Loading a Graph using the GraphLoader by giving it a graph description file and a reference to the PluginLoader.

2. Creating a Threader and InputManager and associate the Graph to them, creating a 3-tuple. For each GraphNode, it creates a Job to which it adds a Node reference (coming from GraphNode). The Threader creates a WorkerThread for each Job. References to input Nodes are added to the InputManager.

3. Starting, stopping or pausing the input frame flow to a Graph through the InputManager instance.

4. Through the Threader instance controlling how many WorkerThreads are assigned to each Job (multiple WorkerThreads for one Job if Module replications are present in one or more Nodes in a Job). Starting, stopping or if the Graph is to be deleted terminating execution of the nodes.

5. Deleting a Graph and its associated InputManager and Threader. Upon deletion of a Graph, the Supervisor stops the InputManager, so that input Nodes stop creating frames. After that the Supervisor posts "Shutdown" mails to all Nodes in the Threader's job list, telling them not to wait for input mails any more. That way the Supervisor can delete the Threader safely, since each WorkerThread will no longer block in the ThreadEntry function in Node.

See figure 4.17 for the data structures the Supervisor contains. The GraphLoader instance is temporary, in the sense that every time a new Graph is loaded a new GraphLoader is created and old instances are discarded. The PluginLoader instance contains all the loaded plugins and instantiated BaseModule specializations during the whole life of the Supervisor. The number of elements in the 3-tuple list, denotes the number of Graph instances at any given time.

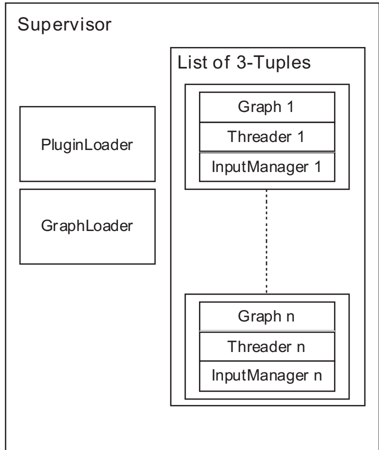


Figure 4.17: *Illustration of the data structures in a Supervisor.*

PluginLoader

The PluginLoader is used for loading a group of modules, called a plugin, into the framework. For creating a plugin, it is needed to derive the BaseModule class and implement a number of methods (see definition of BaseModule). One or more derived BaseModules are compiled to a DLL[3] file which the PluginLoader can load at run time. Once the plugin is loaded, the PluginLoader can return instances of the BaseModule specializations existing in the plugin. Which specialization that should be instantiated is specified by the calling GraphLoader by giving a type id name to PluginLoader.

GraphLoader

Loads a graph description from a text file, instantiates a Graph and creates GraphNodes and in turn adds a Node to each GraphNode. The description file includes the following information:

- Which plugins are used.
- Which modules are used (by type id) and what their instance name should be.
- How the module instances are connected (the Graph description).
- Which modules instances are replicated.

The GraphLoader is responsible for building up a Graph upon parsing the script. This can be described in the following steps:

[3] A DLL (Dynamic Link Library) is a library of functions that uses dynamic linking. This allows an executable to include only the information needed at run time to locate the executable code for a DLL function.

- Calls PluginLoader to load plugins.
- Asks PluginLoader for instances of modules by type id and give each instance a name, according to the script.
- Create a Node for each module instance and assign the instance to the Node.
- Creates GraphNodes and assigns Nodes.
- Creates a Graph and adds the GraphNodes.
- Adds Channels between GraphNodes, while checking for type compatibility between inputs and outputs.

InputManager

The InputManger is responsible for starting and stopping input to a graph. Its functionality could be compared to the functionality of a CD player which can play, stop, pause and set the play position of a song. For starting the flow in the graph it is necessary to tell the input nodes to generate their outputs since they do not have any inputs themselves to trigger them. This is achieved by assigning one MailBox to each input Node in a Graph, so they internally can act as normal Nodes. The InputManager's job is then to send a timestamped "trigger" command to each input Node, that tells them to generate output. The Graph can be seen as a pipeline, which means that "trigger" commands are the mechanisms that insures that the pipeline is fed whenever the input Nodes are ready to produce output. The "play position" is determined by the timestamp contained in the mail and stopping the presentation means that the InputManager stops sending "trigger" commands.

The InputManager has a list of input Nodes, assigned by the Supervisor by inspecting what Nodes that does not have any frame input. A separate thread in the InputManager goes though the list of Nodes posting "trigger" command mails to their mailboxes in the same manner as a Node posts its modules output to another Node. Figure 4.18 on the following page illustrates this mechanism, where each input Node contain a module that streams frames from e.g. a video camera. The two frame streams are then delivered to a Node (having two MailBoxes) containing a blending module (having two inputs and one output) that blends the input frames together and outputs the result as one frame consisting of a composition of the two inputs.

4.3.9 Execution engine Components

The execution engine component classes are in the following described in detail.

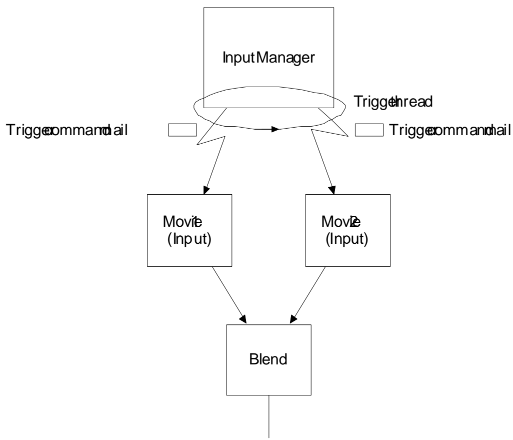


Figure 4.18: *An InputManager sends "trigger" commands to two input Nodes.*

Threader

The Theader creates and handles all the WorkerThreads created when a graph is running. It has a list of created WorkerThreads, each WorkerThread has a reference to a Job. A Job can contain one or more Node references. If a module replication (temporal parallelism) is present, more threads are assigned to the same Job. The Threader controls starting, suspending and stopping of WorkerThreads on behalf of the Supervisor.

WorkerThread

A WorkerThread is a thread of execution that is assigned to a Job (by the Threader). The WorkerThread repeatedly executes its Job, by asking it which Node to execute, until it is explicitly stopped. Several WorkerThreads can be assigned to the same Job which is neccessary when a Node holds replicated modules.

Job

A Job is a task description for one or more WorkerThreads. The Job consists of a list of Nodes that must be executed. This way a WorkerThread can execute several Nodes. This implies that the list must be ordered by precedence so deadlock does not occur, in the case of more than one Node in the list[4]. The default behaviour is simply to assign one Node to a Job's list. We decided on this approach from the observations made in section 4.3.3. As a service the Job has a method, that each time it is called by a WorkerThread, tells what Node in the list must be executed. This method is protected by a mutex.

[4]The Supervisor is responsible for ordering Nodes in a job's list by inspecting the dependencies in the Graph.

4.4 Summary

This concludes the baseline for our AR Museum. We have listed the hardware we will use, the software, programming tools, explained how we calculate field of view for a camera and the fundamentals in 3D graphics rendering. Finally, we have described the most important aspects of the PAVE framework. As mentioned in section 1.2 it is our goal to design the AR Museum so that it runs on top of PAVE in order to make the system scale on multiprocessor PC's. This consists of creating specializations of the BaseModule class and connect instances of those modules in a graph in a manner that corresponds to the conceptual view of the AR Museum as shown in figure 3.3 on page 22 along with the requirements in section 3.4.1 on page 22. The specifics about the design of the graph and the modules along with some necessary enhancements of PAVE, are presented in the next chapter.

Design

In this chapter, we describe the architectural style of our AR museum. It is designed as modules in a graph for the PAVE framework. Each of the modules and their functionality will be described and enhancements made to PAVE to facilitate the design issues will be described.

5.1 Architectural Style

Based on the requirements of the AR museum in section 3.4.1 on page 22, we describe which algorithmic components, denoted modules in PAVE, are needed to build our AR museum. We describe the way in which the modules are connected, that is, as an AR museum graph in PAVE. The role of each of these modules are briefly described. In section 5.3 the design of each module and their algorithmic content is described.

In figure 5.1 on the following page the graph constituting the AR museum is depicted. Each box contains a module. Arcs in the figure represent frame data output from a module passed on as input to successive modules. The small boxes are modules connected to form subgraphs constituting the dynamic effects that produce the works of art outputted to the *world object modules*. A subgraph is typically various bitmap effect modules combined. A subgraph exists for each world object where it is desired to have some kind of changing material over time. The dotted arcs between the *frustum culling module* and the subgraphs, symbolizes that the data sent is a form of triggering that causes receiving modules to run. This triggering mechanism is described in sections 5.2.2 on page 54 and 5.2.3 on page 55.

Three input modules are needed in the AR museum graph, one for grabbing frames from the camera filming the real world, one that registers the position and orientation of the camera, obtained by the magnetic tracker and a module that holds information about the real and virtual world. These modules are denoted *camera*

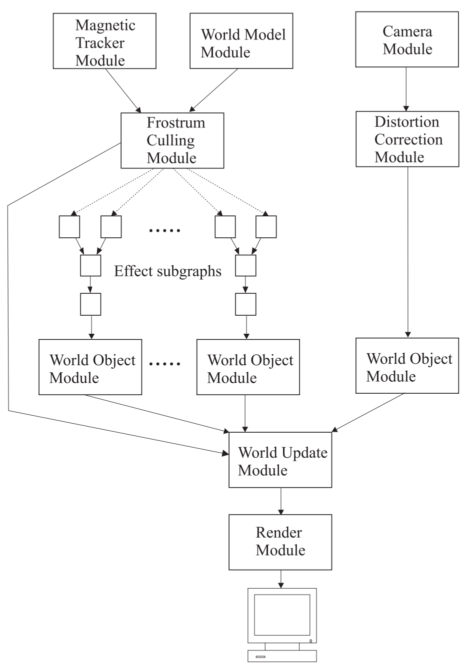


Figure 5.1: *The AR museum graph.*

module, *magnetic tracker module* and *world model module*, respectively. The *world model module* and the *magnetic tracker module* are connected to the *frustum culling module*. The task of this module is to decide which of the *world object modules*, to which it is connected to through subgraphs, holds WorldObjects that are visible. Only these subgraphs must be triggered to run. The frames grabbed from the camera needs to be corrected for the lens distortion that occurs in the camera. This is done in the *distortion correction module*. The undistorted image of the real world is passed on to a *world object module*, which creates a description of a virtual world object. It is thus treated like the virtual world objects that has input (e.g. bitmap textures) from subgraphs. These subgraphs are triggered to run by the *frustum culling module*. All *world object modules* are connected to the *world update module*. The task of this module is to update the WorldModel according to its WorldObject inputs. The WorldModel and WorldObject data structure is described in section 5.3.1 on page 59. The *world update module* receives a WorldModel reference and outputs an updated WorldModel to the *render module*. The role of the *render module* is to render the WorldModel that it gets as input, so the resulting rendered image can be displayed on a monitor.

The graph design described above, gives rise to some enhancements to the existing PAVE framework. Firstly, the *frustum culling module* must give output to a variable number of subgraphs and the *world update module* must take input from the same variable number of *world object modules* connected to those subgraphs. This gives a need for allowing an optional number of inputs and outputs on mod-

ules, where the number of outputs and inputs are not known at module creation time, but first when the graph is being built.

Another issue in the design of the graph is that the *frustum culling module* is responsible for triggering the subgraphs connected to it. The triggering of input modules is a task in PAVE which originally was the exclusive responsibility of the InputManager (see section 4.3.8 on page 47). Triggering in some form should also be possible for a Node. Furthermore, subgraphs can be inactive, implying that no processing by their modules is done. This is the case e.g. when the *frustum culling module* detects that a WorldObject is not in the field of view, hence its subgraphs does not need to be computed. The triggering mechanism must be enhanced so that input modules to a certain subgraph connected to a *world object module* can be set to be inactive in some way, so that modules in a given subgraph are not executed, when it is not needed.

These PAVE enhancements among other general robustness enhancements, are described in the following section.

5.2 PAVE Design Enhancements

The aforementioned design observations give rise to a collection of design enhancements of PAVE that are desirable in order for the system as a whole to be both flexible, robust and service the need for performance optimization at a higher level than internal module algorithms. We start by describing the mechanism needed to support optional inputs and outputs on a module. Next we describe the triggering and how this can be used to deactivate subgraphs. These aspects can be categorized as making graph building more flexible and serving to performance optimize graph execution in general, respectively. Finally we describe the aspects for making graph execution more robust, which involves designing a policy for general garbage collection of data and controlling what data is read-only and what is writable.

5.2.1 Optional Inputs and Outputs on Nodes

Originally it was necessary for a BaseModule specialization to specify exactly how many inputs and outputs (and their type) it needs in order to be able to function (see 4.3.5). Clearly this imposes a limitation, since it is necessary to write a new module, e.g. a *frustum culling module*, each time the number of world objects in the virtual world changes and hence the number of *world object modules* in the AR Museum graph. The reason for this is that the *frustum culling module* and *world update module* would have changed their number of outputs and inputs, respectively. It would be possible to set an upper limit on how many objects in the world model that could be manipulated by subgraphs, regardless of how many actually existing in the virtual world. But such a limitation seems rather unflexible and possibly

serves to add more complexity in the AR museum graph. Therefore the need for a rule that allows a given module to have an optional number of inputs and outputs seems apparent.

We have decided to make the rule simple and so that it does not conflict with the original design of PAVE. The modification contains the following:

- When a module has specified its "static" input and output types, in its constructor (see section 4.3.5 on page 38), in the conventional way, it should be possible to tell the framework that it can have n optional inputs and/or outputs of some type following after the statically defined ones. If the module has specified j "static" inputs, minimum j connections must be made to it, and $j + n$ connections are possible, where $n \geq 0$. The same goes for optional outputs.

- When the module is inserted into a graph, the conditions for inputs and outputs must hold in the same way as originally designed.

- When connections are made to the optional inputs, additional MailBoxes on Node must be made at connection time.

As a simple example, a module specifies that it has optional inputs only. If i connections are made to a seperate input number in the module, the framework automatically adds i Channels to the associated GraphNode as before, but in addition i MailBoxes in the associated Node must be added. The difference is that the module specifies that it can have optional inputs, but does not know in advance how many. So the framework must check how many Channels are made at graph build time to the module, and add MailBoxes to the associated Node as needed. That means that when the Supervisor (see section 4.3.8 on page 45) has loaded a graph from a description file, it will go through every GraphNode, inspecting the number of Channels added to each GraphNode. For each GraphNode, update the associated Node's number of MailBoxes.

5.2.2 Trigger Capable Node

In order for a Node to be able to gain control over certain input Nodes, it is neccessary to be able to connect it to these input Nodes. Originally it was not possible to connect a Node to an input Node, since an input Node contains a module with no frame input. The assumption was that since no frame input is desired, the rule was that the input Node simply needed to be triggered by the InputManager (see 4.3.8 on page 47). So we need to modify the rule for the special case where a Node is connected to an input Node, when the Supervisor inspects what Nodes that must be associated with the InputManager.

We describe the modified rule in the following way:

- If a Node has 0 inputs (its module has no frame inputs), it is considered an input Node.
- If an input Node has a channel connected to it, it must *not* be associated to the InputManager. Otherwise, the Node must be associated to the InputManager.

Now a Node can be triggered by either the InputManager as before, or be triggered by another Node.

5.2.3 Empty Triggering

The idea of a Node being able to send "trigger" commands to another Node becomes useful when we need to be able to deactivate subgraphs. We do this by generalizing the functionality from sending "trigger" commands, to be able to send any type of command. The decision of what type of command a Node will send to another Node, is placed in a module's action method belonging to the sending Node. That way a module programmer can create a module that can send commands of any type, but if the receiving Node would have to react on the command, the Node's ThreadEntry method must be modified to process a given command. Otherwise only the module in the receiving Node can process the command.

To be able to deactivate given subgraphs at given times during rendering, we add a few extra conditions in the Node's ThreadEntry method. It it described in the following:

1. Check to see whether the frame received at the *first* input is of the *Command* type (by inspecting the type id), and if that is the case, see 2. If not, see 4.
2. If the received command is a normal "trigger" command (by inspecting the command type's string attribute), see 4. If not, see 3.
3. Check to see if the command is a so called "empty trigger". If that is the case, the action method of the Node's module is not called and hence no execution of the module for the given frame index. In addition send "empty trigger" commands on all outputs.
4. The module's action method is executed exactly as originally intended, and output returned from the action method call is delivered on all outputs.

Another approach is that a Node could choose not the send any "trigger" commands for given frame indexes, causing flow to stop completely in the subgraphs. Unfortunately, this would stop *all* dependent succeeding nodes, since a Node waits until input has arrived on all input MailBoxes and hence block the whole graph. It would be possible to make a rule on the Node's ThreadEntry method, where certain input numbers could be optional, but the idea seem to complicate matters in far too many cases. Sending an "empty trigger" also insures that frame index order is not corrupted when subgraphs are made active/inactive.

5.2.4 Garbage Collection

When modules are sending data in a graph, they are sending references to data. A module that sends a result, to another module, typically creates the data and sends a reference to that data as output. So in that case the creating module owns the data by default. But cases also exist where a module merely forwards data that it got from some other module. In such a case the ownership belongs to some other module.

The problem is that a given module receiving data does not know to whom the data belongs. And since data that is not being used anymore must be deleted in some fashion, some mechanism in e.g. a module must take the initiative to delete that data. This gives another problem since a given module does not know whether another module also has a reference to the data, and hence cannot safely delete data. In fact it is impossible for a module to determine whether it is safe to delete data or not, since modules can be connected in many different ways, and a module does not know the topology of the graph it is inserted in and does not know anything about other modules. The only thing a module knows for sure is what input data types it can receive and what output data types it delivers, and whether these only can be read from or are writable (see 5.2.5 on the facing page). Therefore the algorithm in the action method of a module can not determine who owns the data it gets and whether that data is referenced in some other module in the graph.

The only thing a module knows about the data it sends along is whether it wants to keep the data for future calculations or not. If it wants to keep the data, it also has the responsibility of deleting it eventually, and hence other modules must not delete it.

From this observation it is quite clear that some rule regarding the integrity of the data being sent between modules, should be at hand. It is also neccessary that the mechanism itself is not part of any module algorithm, since a module only knows what it creates, wants to keep and what it does not want to keep. Another reason is that a module programmer's role should not be further complicated, as stressed in our original goals for the PAVE design.

For this mechanism we have decided to add reference counting with garbage collection on the BaseType class, so that all specializations of BaseType, e.g. WorldModel, WorldObject, FrameBuffer etc., have reference counting.

This is done by adding an integer attribute as a counter in the BaseType baseclass. In addition we add two methods to the BaseType base class called **AddRef** and **ReleaseRef**. These two methods are decribed below:

- **AddRef**. Increments the reference counter attribute by one.

- **ReleaseRef**. Decreases the reference counter attribute by one. If the counter has reached zero, this BaseType instance deletes itself.

The two above methods are encapsulated in a critical region, protected by a mutex. When a BaseType specialization is instatiated, its initial reference count is one, meaning that the creator owns the BaseType instance initially. It is neccessary to hold a reference variable to the created BaseType instance in order to e.g. give up ownership (by calling the ReleaseRef method) of the reference to the instance. If e.g. a module is receiving a reference to a BaseType instance and it wants to keep that instance for future calculations, it calls the AddRef method on the instance. If a module algorithm in the action method creates a new BaseType instance for each frame index, that it wants to deliver as a result on one of its outputs, it adds the reference to the output list (which is a BaseTypeList) and gives up its own ownership by decreasing the reference count. That is unless it wants to keep the result for future calculations.

When a BaseType instance is added to a BaseTypeList, the reference count is increased by one, and when a BaseTypeList is deleted all its elements (BaseTypes) have their reference count decreased. In section 4.3.7 on page 42, the pseudo code for the original ThreadEntry method can be seen. It can be seen that the local variable called InputList (which is a BaseTypeList) is assigned a number of BaseTypes from each input MailBox and given to the module's action method. The InputList is deleted when thread execution returns from ThreadEntry. This means that if a module has not obtained ownership to one or more of the incoming input BaseTypes they are deleted if no other modules has ownership to them.

This approach to garbage collection and reference counting is similar to e.g. Microsoft's Component Object Model (COM) and was inspired from that model.

5.2.5 Read-Only and Writable Data

To ensure that modules do not write in data they are not supposed to, and to avoid making superfluous copies of data, we introduce read-only data, and writable data. When a module's input and output is declared, it must be stated if it is read-only or writable. To support read-only and writable data, a permission attribute is put on the BaseType, denoting if the data is read-only or writable.

The BaseTypes contained in a module's input BaseTypeList and the BaseTypes it generates as output must all have the permission attribute set. With the permission attribute set, the following semantics is used for sending data between modules. For all Channels it holds:

- **R $\rightarrow$ R:** If the receiving module has specified its input data as read-only, it will not write in it, and a reference to the original data can be sent.

- **W → R:** As above.

- **R → W:** If the receiving module has specified its input data as writable, and the data outputted is readable, a reference to a clone of the data is created and sent.

- **W → W:** If both the receiving module and the sending module has specified the data as writable, two cases exist:

  - If the Channel in question is the last one stemming from the given output, and no Channels from the output sends to a read-only destination, then a reference to the original data is sent, as no other modules will access the original data.

  - Otherwise a reference to a clone of the output data is sent.

It should be noted that when sending output from a module, all clones of data are made before the original reference may be sent. This avoids that data is cloned, while residing in another module. A clone is always writable and its reference count is set to one.

In figure 5.2, an example graph containing all different cases of R/W semantics is depicted. With the R/W semantics introduced, modules can be connected

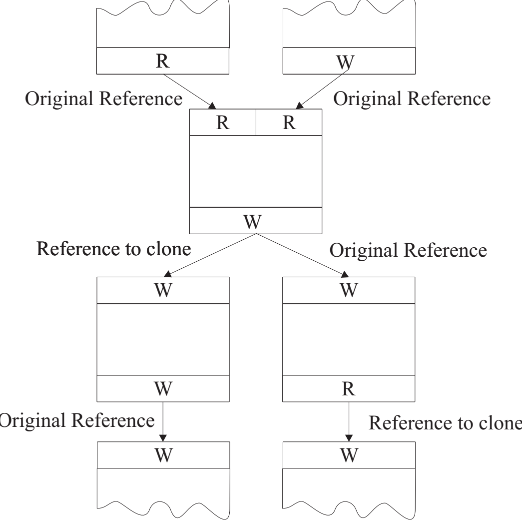


Figure 5.2: *Graph extract containing all different R/W data semantic cases.*

in different ways, without explicitly stating for each possible graph built, whether module outputs should be sent as references or references to clones.

5.3 Components

Here, we will describe the necessary PAVE data structures and modules (see figure 5.1 on page 52), what algorithms each module must contain and how those algorithms work.

5.3.1 Data Structures

In this section we describe the PAVE data types used by the AR museum modules. All the types are specializations from BaseType (see section 4.3.5 on page 38) and described in the following.

WorldObject

The WorldObject is a data type which encapsulates information about a real-world or virtual object present in the world. The WorldObject consist of the following elements:

- **Name**:
  A unique string identifier.

- **Mesh**:
  List of polygons describing the geometry of the WorldObject.

- **Texture**:
  A bitmap image which is mapped onto the surface of the WorldObject.

- **Bounding box**:
  An axis aligned minimal enclosing box that covers the geometry of the WorldObject in all 3 dimensions. This will be used for visibility testing (see section 5.3.6).

- **State**:
  The state string can either be "static" or "dynamic". See section 5.3.8 for a description of how the state string is used.

- **Visible attribute**:
  Determines whether the WorldObject is visible or not.

WorldModel

The WorldModel is a data type which holds information about the real and the virtual world. In particular it is used for holding descriptions of geometry and from what position and orientation the world is viewed. The WorldModel consist of the following elements:

- **List of WorldObjects describing real-world objects**:
  Used for describing objects such as walls, floors, ceilings and other static real-world objects. Its usage is described in section 5.3.9.

- **List of WorldObjects describing virtual objects**:
  Used for describing all the virtual objects in the world.

- **List of lightsource descriptions**:
  Used for simulating real-world lighting on the virtual objects.

- **View transformation matrix**:
  Represents the position and orientation of the camera viewing the world.

- **Projection transformation matrix**:
  Represents the viewing frustum in which the world is visible.

FrameBuffer

The FrameBuffer is a type which encapsulates a representation of a bitmap image. It has a buffer holding the raw pixels in the image and descriptions of pixelformat and resolution.

Matrix

The Matrix type is a representation of an arbitrarily sized matrix.

String

The String type encapsulates an arbitrarily sized string.

Float

The Float type encapsulates a single precision floating point value.

Command

The Command type is used for sending commands to Nodes in a PAVE graph. Two types of commands will be used in the PAVE museum graph:

- Triggers (see section 4.3.8 on page 47).

- Empty triggers (see section 5.2.3 on page 55).

In the following sections we describe the modules constituting the AR museum graph.

5.3.2 World Model Module

- Inputs:
  - none
- Outputs:
  - WorldModel (read-only)
- Parameters:
  - String: name of world model file

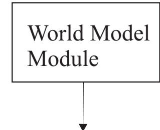


Figure 5.3: *World model module*

The *world model module* is responsible for loading and initializing the WorldModel. Based on the filename given in its parameters it loads a geometry description of the world from disk. The geometry is specified in a Microsoft X file which is a template-driven format that enables storage of meshes, textures, animations, and user-definable objects [msd01]. After the geometry is loaded the WorldModel structure is instantiated and the geometry is inserted into it. For each mesh in the X file a WorldObject is created and inserted into the WorldModel. For each WorldObject a bounding box is computed and stored in the WorldObject.

Furthermore a background WorldObject is created and inserted into the WorldModel. The background WorldObject represents the current image of the real world which is grabbed by the camera. It is simply a rectangular mesh with the current real-world image mapped onto it as a texture. See section 5.3.9 on page 69 for descripition of how the background WorldObject is used when visualizing the AR museum.

The *world model module* outputs a read-only WorldModel data structure each time its action method is called and it keeps a reference to the WorldModel itself so the WorldModel will not be deleted by the automatic garbage collection (see section 5.2.4 on page 56).

5.3.3 Magnetic Tracker Module

- Inputs:

– None

- Outputs:
  – Matrix: 4x4 view transformation matrix. (writable)
- Parameters:
  – Matrix: transmitter translation vector relative to world model origin.
  – Matrix: receiver translation vector relative to camera.

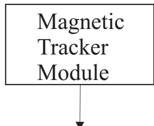


Figure 5.4: *Magnetic tracker module*

The task of the *magnetic tracker module* is to read the position and orientation of the magnetical receiver, that registers the camera's extrinsic parameters.

When the module is initiated, the position of the magnetical transmitter must be known. The transmitter translation vector V_t, received in the parameters, describes the translation from the origin in world space to the transmitter (see figure 5.5).

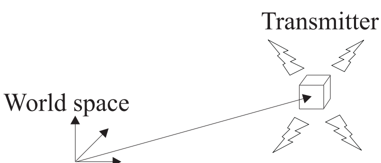


Figure 5.5: *Transmitter translation vector*

As the magnetical receiver is highly sensitive, we can not place it on top of the camera, as the camera corrupts the magnetical field. As described in section 4.1 on page 25, the magnetical receiver is placed behind the camera on a wooden stick to avoid corruption of the position and orientation measurements (see figure 5.6 on the next page). The receiver translation vector V_r, received in the parameters, describes the translation from the receiver device to the camera lens.

When the action method is run a rotation matrix M_{rot} and a receiver position vector V_{pos} is obtained from the orientation and position measured. Since the receiver translation must be done in the camera's local coordinate system, the receiver translation vector V_r is multiplied by the transposed rotation matrix:

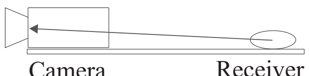


Figure 5.6: *Receiver translation vector*

$$V_{rl} = V_r \cdot \overline{M_{rot}}$$

The total translation vector V_{total} is then described by:

$$V_{total} = V_{pos} + V_{rl} + V_t$$

From V_{total} a 4x4 translation matrix M_{tran} is created. The view transformation matrix M_{view} is computed by multiplying M_{tran} by M_{rot} and is sent as output.

5.3.4 Camera Module

- Inputs:
  - None
- Outputs:
  - FrameBuffer (writable)
- Parameters:
  - None

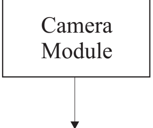


Figure 5.7: *Camera module.*

The task of the camera module is to grab frames of the reality filmed by a camera. It has no input and delivers a FrameBuffer containing the grabbed image as output. To grab the frames, functionality in Microsoft's DirectShow API was used. When the action method is called, the current image grabbed is outputted as a FrameBuffer.

5.3.5 Distortion Correction Module

- Inputs:
  - FrameBuffer (read-only)
- Outputs:
  - FrameBuffer (writable)
- Parameters:
  - 4 Floats: distortion coefficients

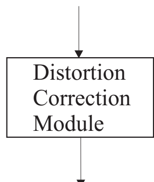


Figure 5.8: *Distortion correction module*

The task of the *Distortion Correction module* is to compensate for the distortion that occurs in the image grabbed by the web camera.

Distortion is an optical error in the camera lens that causes a displacement of pixels at different points in the image. The pixels in the image are misplaced relative to the center of the field, hence it is called radial distortion.

Radial distortion comes in two forms: pincushion distortion (positive) and barrel distortion (negative). The two forms of distortion is depicted in figure 5.9.

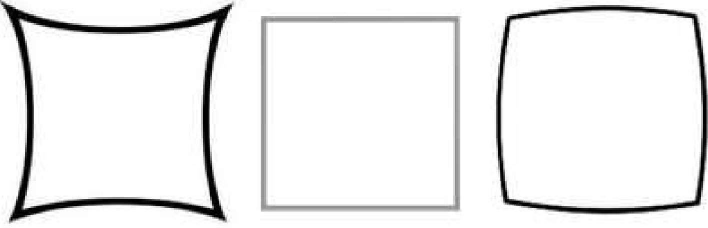


Figure 5.9: *The result of pincushion distortion(left), non-distorted image (middle) and barrel distortion(right)*

The distortion of pixels is not linearly correlated to the distance to the center of the image. At small distances to the center of the image there will be very little

displacement and in the edges of the image displacement of pixels will be very large. This can be approximated by the following equation:

$$r_{src} = r_{dest}(a{r_{dest}}^3 + b{r_{dest}}^2 + c r_{dest} + d) \tag{5.1}$$

r_{dest} denotes the distance from the pixel to the center of the image, in the distorted image and r_{src} denotes the corresponding distance in the distortion corrected image.

The a, b and c coefficients is a measurement of the distortion in the image. The parameter d describes the linear scaling of the image. Using $d = 1$, and $a = b = c = 0$ leaves the image as it is. If the distortion corrected image must have the same size as the original image, it must hold that:

$$a + b + c + d = 1 \tag{5.2}$$

The above equations were taken from [Der99].

The difference between the actual (distorted image) and the "real" predicted (non-distorted image), as it would look taken by an ideal pinhole camera, can be counteracted by displacing each point in the image along the direction vector spanned by the center point of the image and the distorted point, as shown in the above equation.

For each pixel index in the destination image, the *distortion correction module* calculates which pixel index in the source image it must contain, and saves this in a lookup table.

This table is used in the action method to correct pixel positions in every frame the *distortion correction module* receives. The distortion coefficients used have been obtained empirically by filming a checkerboard, and adjusting them to get an undistorted image.

5.3.6 Frustum Culling Module

- Inputs:
  - WorldModel (read-only)
  - Matrix: 4x4 view transformation matrix. (read-only)
- Outputs:
  - WorldModel (writable)
  - Multiple triggers/empty triggers (writable)
- Parameters:
  - String: list of object names that needs to be checked for visibility.

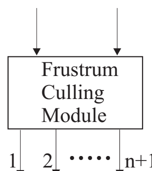


Figure 5.10: *Frustum culling module*

- Float: horizontal field of view.
- Float: vertical field of view.

The task of the *frustum culling module* is to determine which of the WorldObjects are visible and make sure that only subgraphs that produce frames for visible objects are executed. This allows for more virtual objects in the world that use frames generated by subgraphs, since all the virtual objects are rarely visible at the same time and hence not all of the subgraphs needs to be executed. In terms of rendering speed this can provide for a significant speedup.

Each of the module's optional outputs (see section 5.2.1 on page 53) is dedicated to a separate subgraph, which the *frustum culling module* can either activate or deactivate by sending a trigger mail or an empty trigger mail respectively (see section 5.2.3 on page 55). So all input nodes in a given subgraph *must* be connected to the same output on the *frustum culling module*. Furthermore there may not exist any dependencies between any of the subgraphs, since it could result in unpredicted results when a module in an active subgraph expects valid data from a module in an inactive subgraph. See figure 5.11 for an illustration of subgraphs connected correctly to the *frustum culling module*.

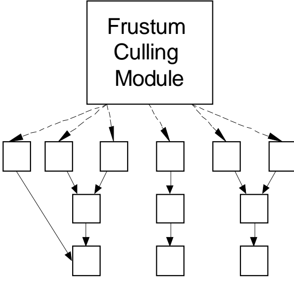


Figure 5.11: *Illustration of three subgraphs connected to each their output on the frustum culling module. The dotted lines represents trigger/empty trigger mails being sent to input nodes.*

When the action method is called it must determine which of the optional outputs to send triggers/empty triggers to. For this two things must be known. Firstly, it must be known what output is linked to what WorldObject. This is achieved by

giving the *frustum culling module* a string parameter containing the names of the WorldModel objects that needs to be checked for visibility. Names are listed in the string in the order which correspond to the numbering of the multiple outputs. If for instance the string is "*object1, object2*", a trigger will be sent on first of the optional outputs if *object1* is visible. If *object1* is not visible an empty trigger will be sent on the first of the optional outputs. This will be true for all the object name/output pairs.
Secondly, the actual visibility test must be performed for each WorldObject. For this to be achieved the view matrix and the projection matrix (see section 4.2.2 on page 28) must be known. The view matrix is received on the second input and the projection matrix is generated based on the vertical FOV and the horizontal FOV received in the parameters and statical values for front and back clipping planes. These values are set to span the entire world model. Alternately, they could be determined by the nearest and farthest away visible geometry.

The projection matrix represents the viewing frustum (see figure 4.4 on page 29) and the view matrix represents the the camera's position and orientation. When these two matrices are multiplied, the resulting matrix represents a transformed viewing frustum that is the volume in which objects are visible to the camera. The six planes front, back, left, right, top and bottom which constitutes the viewing frustum are extracted from the multiplied matrix.

The visibility test checks if an object lies inside or outside of the transformed viewing frustum. For a single point this is done by checking for each of the six planes planes if the point lies in the halfspace not containing the frustum. If for one or more of the planes this is true, the point can be classified invisible. This check can be very expensive if the WorldObject's mesh consists of many polygons, since the check needs to be done for each vertex in all the polygons. Therefore the WorldObject's bounding box is used instead since it only requires to test if the box is inside the viewing frustum. If it is inside, the WorldObject is marked visible. This ensures that if the bounding box is invisible then the mesh is also invisible. Though, in some cases the bounding box can be visible but the mesh invisible, this is still a highly preferable method performance-wise.

After the visibility test has been performed for each WorldObject in the WorldModel, a clone of the WorldModel is made and the view matrix and the projection matrix are stored in it. Instead of forwarding the WorldModel received on the input, the clone is made to ensure that the new view and projection matrices stored for next frame do not overwrite the current ones before they have been used for rendering.

5.3.7 World Object Module

- Inputs:

  - FrameBuffer (read-only)

- Outputs:
  - WorldObject (writable)

- Parameters:
  - String: WorldObject name
  - String: WorldObject state, ”dynamic” or ”static”

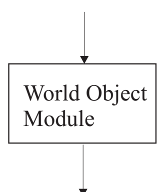


Figure 5.12: *World object module*

The task of the *world object module* is to create a WorldObject based on its input and parameters. The WorldObject is representing a virtual object, in the WorldModel, which needs to be created or modified in some way. When the action method is called the module names the WorldObject and sets its state as specified in its parameters and assigns it a texture which is present in the FrameBuffer it receives in its input.

5.3.8 World Update Module

- Inputs:
  - WorldModel (read-only)
  - Multiple WorldObjects/EmptyTriggers (read-only)

- Outputs:
  - World Model (read-only).

- Parameters:
  - none

The *world update module* is responsible for updating the WorldModel based on the WorldObjects received in its inputs. The second input on this module is a multiple type which allows an arbitrary number of modules to be connected to it. These modules must output either a WorldObject or an empty trigger. When the

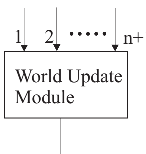


Figure 5.13: *World update module*

action method is called all empty triggers are ignored while all the WorldObjects are collected in a list and processed. This involves checking each WorldObject if it exist in the WorldModel and if its state is set to dynamic. If this is the case the WorldObjects's FrameBuffer is read and uploaded to the texture memory on the display adapter. This will overwrite the texture memory assigned to WorldObject's texture so that next time it is drawn the contents of the framebuffer is mapped onto the surface of WorldObject. Note that the *world update module* is designed only to update textures on the virtual objects, but it is easily extendable to support adding new WorldObjects to the WorldModel or updating other properties on existing WorldObjects.

5.3.9 Render Module

- Inputs:
  - WorldModel (read-only)
- Outputs:
  - none
- Parameters:
  - none

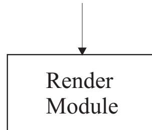


Figure 5.14: *Render module*

The *render module* is responsible for visualizing the visible objects in the world. When the module is initialized it setups the Direct3D rendering pipeline and creates a window in which the rendered image will be displayed.
When the action method is called the *render module* examines the WorldModel

received on the input to render the final image. Two requirements exist for the rendering:

- Requirement 1: Virtual object must be rendered overlayed onto the image of the real world

- Requirement 2: Occlusion between virtual objects and real-world geometry must be handled

Requirement 1 is achieved by first rendering the background WorldObject (see section 5.3.2). The background WorldObject is a rectangular mesh with the image of the real world mapped onto it. It can be seen as a plane always positioned perpendicularly to the camera view direction. Therefore it should not be transformed by the view and projection matrices in the WorldModel. Instead it is projected directly into screen space using an orthogonal projection so it covers the entire render window. Since the background WorldObject does not contain any depth information the depth buffer is disabled while rendering it.
Next, the virtual objects must be rendered, but for achieving requirement 2, some depth information about the real world is needed. An example of this could be a room with pillars in and virtual pictures on the walls. It is likely that in some positions a virtual picture is occluded by a pillar, and in this situation it is necessary to know depth information about the pillar. Since all depth information rely on the provided descriptions of geometry , descriptions of the real-world objects that can occlude virtual objects are needed. When rendering the real-world geometry descriptions they should not be rendered to the color buffer since they are already present in the image of the real world. Instead we disable writes to the color buffer and enable writes to the depth buffer so only their depth information is present in the scene. Afterwards when rendering the virtual objects, writes to the color buffer are enabled and the visibility of the virtual objects will depend on the depth information present in the depth buffer.

The view and projection matrices in the WorldModel are used to transform both the real-world objects and the virtual objects so they appear correctly according to the camera position and orientation. Lighting is enabled only when rendering the virtual objects. Information about the lightsources are obtained in the WorldModel and used for rendering the virtual objects so their appear illuminated by the real-world lightsources. The lightsources in the WorldModel should of course be placed at positions that correspond to where lightsources are positioned in the real world.

The render steps involved in rendering the complete scene is listed cronologically below. For description of render specific terms see section 4.2.2.

- Clear depth buffer.

- Disable depth buffer.

- Disable lighting.
- Set view transformation to the identity matrix.
- Set projection transformation to orthogonal projection.
- Render background WorldObject.
- Enable depth buffer.
- Disable writes to color buffer.
- Set view transformation to view matrix in WorldModel.
- Set projection transformation to projection matrix in WorldModel.
- Render WorldObjects that represent real-world objects.
- Enable writes to color buffer.
- Enable lighting.
- Render WorldObjects that represent virtual objects.

5.3.10 Summary

This chapter has described the overall design the AR museum by identifying what components was needed to realize the AR museum using the PAVE framework. This resulted in designing enhancements to PAVE and designing the functionality of each component. This design has resulted in a implementation of a prototype of the AR museum which will be subject to experiments in the following chapter.

6 Experiments

In this chapter, we perform three types of tests to our AR museum. The performance test will determine the scalability of the AR system by comparing frame rates when running the system on a single and a dual processor machine. The user impression test is less concrete. It is based on examining screenshots of the running system to see how well the augmented objects blend in with reality. In some screenshots a displacement error of the virtual objects can be seen and therefore a test was constructed that determines the precision of the magnetical tracker.

The world model used for the performance test and the user impression test contains four virtual objects. Three pictures placed on the walls in the room and a teapot placed on a table. The PAVE AR museum graph used for these tests is depicted in figure 6.1 on the following page. It contains three subgraphs which are linked to the virtual pictures on the walls and therefore the textures on these pictures are the result of frame output from these subgraphs. The first subgraph contains an *image loader module* which simply loads a bitmap image from disk when initialized and outputs this image[1] each time it is called. The second subgraph contains a *plasma module* which generates a swirling color pattern based on trigonometric functions. The third subgraph contains a *circle flower module* which generates a moving "flower like" pattern. Two of the virtual pictures are placed in positions where real pictures exist in the room. This allows for observing if the virtual pictures are situated at their correct positions.

All tests were performed on a Dual Celeron 400 MHz with 128 Kb cache, 128 Mb 66 MHz ram and a NVIDIA TNT2 graphics card.

6.1 Performance Scaling Test

This test measures the scalability of the AR system by comparing frame rates when running the system on a single and a dual processor machine. Multiprocessor sup-

[1] We have used the famous painting called "Skriget" by Edvard Munch.

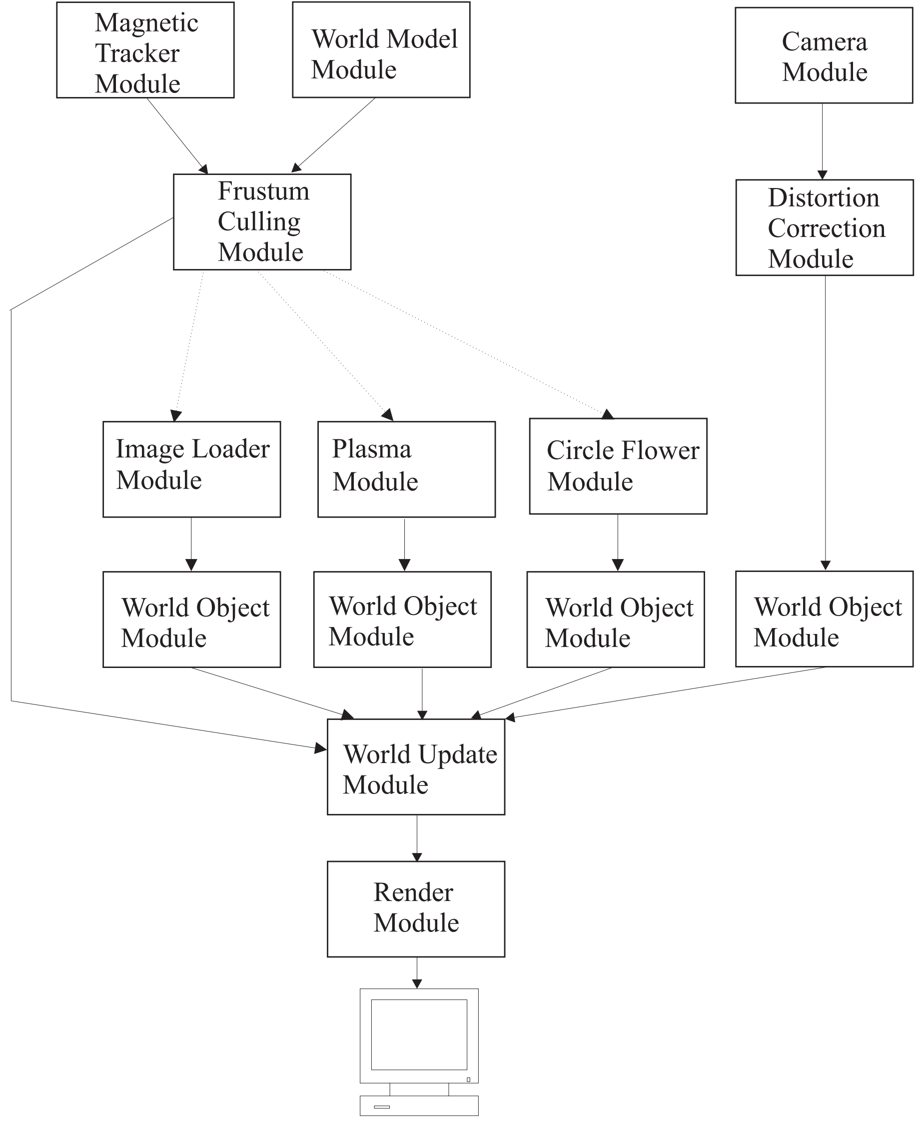


Figure 6.1: *PAVE AR museum test graph.*

port was disabled in Windows 2000 for the single processor frame rate measurement.

6.1.1 Test Setup

The PAVE graph depicted in figure 6.1 on the preceding page was executed in the PAVE framework. The frame rates measured are average frame rates obtained by running the the AR museum graph for several minutes using arbitrary camera-positions and orientations.

6.1.2 Test Results

The measured frame rates are listed in table 6.1. We consider the 54.2 percentage gain in frame rate a satisfactory result.

Fps on 1 CPU	**Fps on 2 CPUs**	**gain in %**
5,9	9,1	54,2

Table 6.1: *Performance scaling result table.*

6.2 User Impression Test

This test is based on examining screenshots of the running system to see how well the augmented objects blend in with reality.

6.2.1 Test Setup

The PAVE graph depicted in figure 6.1 on the preceding page was executed in the PAVE framework. Screenshots were taken while the system was running.

6.2.2 Test Results

In figure 6.2 screenshots of the augmented objects can be seen. How well the virtual objects blend in with the real world differs in the screenshots. It turned out that it was highly dependent on camera position and orientation. Furthermore it is most noticeable on the screenshots containing the virtual pictures since the virtual edges do not allign the real-world edges. The virtual teapot, on the other hand, blends in quite well.

In figure 6.3 three locations in the room can be seen with and without virtual objects.

In figure 6.4 on page 78 an example of gross displacement of the virtual object is seen. For this screenshot the camera was rolled 180 degrees which resulted in a

(a) Virtual picture with dynamic texture.

(b) Virtual picture with static texture.

(c) Virtual teapot.

(d) Virtual teapot.

Figure 6.2: Screenshots of virtual objects blended with a real world image.

(a) First location without augmented object.

(b) First location with augmented object.

(c) Second location without augmented object.

(d) Second location with augmented object.

(e) Third location without augmented object.

(f) Third location with augmented object.

Figure 6.3: Screenshots of three locations with and without virtual objects.

displacement of the virtual picture. The displacements in the x and y directions are approximately 18 and 12 centimetres respectively.

Figure 6.4: *Screenshot showing the displacement of a virtual picture.*

6.3 Magnetical Tracker Precision Test

Given the displacement of virtual objects experienced in the user impression test, we wish to test whether the displacement can be put down to the inaccuracy in the magnetical tracker measurements.

6.3.1 Test Setup

We wish to compare position and orientation measured by the magnetical tracker, with the position and orientation that should be obtained. We have marked angles covering 180 degrees rotation about the y-axis, in the xz-halfplane with 15 degrees between them. The magnetical receiver is put on a stick, which is placed from the transmitter to the 13 marked points. Each of the 13 angles (0 to 180 degrees), and the corresponding x and z position is measured 10 times, and an average angle and position is computed, to counteract inaccuracy in the test setup itself. In figure 6.5 on the next page, the tracker precision test setup is depicted.

6.3.2 Test Results

In table 6.2 on page 80 the results of the magnetical tracker test are listed. Angle errors are measured in degrees and position errors are measured in centimeters. If we look at the *average*, *min* and *max* values for angle and position errors, we can see that position errors are less than angle errors. Because of the fact that position errors do not impact the quality of AR as much as angle errors does, we only examine the angle errors further. To estimate how much impact the angle errors have

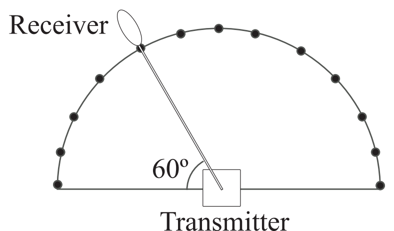


Figure 6.5: *Tracker precision test setup.*

on displacement of the virtual objects, a simple case depicted in figure 6.6 is used. This figure shows a camera looking at a wall from a distance d. The dotted line represents the measured erroneous looking direction of the camera. This results in a point being displaced by the distance x on the wall.

If we return to figure 6.4 on the facing page showing gross displacement of a virtual picture on a wall and use the information that the camera was placed approximately 2 meters from the wall, the displacement x can be calculated for the *average*, *min* and *max* angles errors. This gives 11.9, 1.0 and 23.5 centimeters respectively. The displacements observed in the screenshot were 18 and 12 centimeters respectively, so this lies within the value for the *max* angle error. From this, we concluded that the angle error introduced in the measurements of the magnetical tracker's orientation is significant enough to cause the displacement of the virtual objects. These errors are most likely due to the magnetical tracker's sensibility to metal and other magnetic fields in the environment.

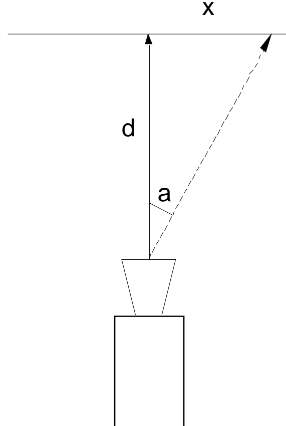


Figure 6.6: *Displacement due to angular error.*

Angle	Angle err.	X pos. err.	Z pos. err.
0	0,3	0,9	0,0
15	1,6	1,5	0,6
30	2,0	1,5	1,0
45	2,4	1,3	1,2
60	2,3	0,8	1,0
75	2,5	0,4	1,3
90	3,1	0,2	1,6
105	3,8	0,0	1,3
120	4,3	0,5	1,0
135	4,9	0,6	0,7
150	5,4	1,2	0,7
175	6,7	2,3	0,1
180	5,3	0,7	0,3
Average	3,4	0,9	0,8
Min	0,3	0,0	0,0
Max	6,7	2,3	1,6

Table 6.2: *Tracker precision result table.*

6.4 Test Conclusion

The tests performed on the AR museum system shows that the system scales well. In terms of the visual quality, the main reason for the less than perfect aligment of the virtual objects, can be put down to the magnetical tracker, as mentioned in the precision tests. Regarding the requirement, that the minimum acceptable frame rate being 10 frames/second, in an AR system, this is barely satisfied. Our system achieved a frame rate of 9.1 frames/second on the 2 processor test machine with a TNT2 graphic card and hence is a little bit slower than the acceptable frame rate. The dual Celeron machine is a rather slow machine, compared to even contemporary single processor machines, and therefore we do not see this as a problem. We tested the AR Museum on a 500 Mhz Pentium-III single processor machine with a GeForce 256 card, where the system ran with a frame rate of 20 frames/second, which gave satisfactory visual results.

7 Conclusion

In this chapter we will conclude upon our goals and requirements set in chapters 1 and 3.

We have designed an Augmented Reality Museum architecture, that allows a user to inspect a room augmented with virtual objects constituting the art in the museum. The museum is described by a world model holding information about the real world and the virtual objects residing in it. The world model can be created by using a modelling tool so that complex environments can be created. The virtual objects can be dynamic and thereby giving the user a more immersive experience.

The design has been modularized and described by modules connected in a graph in the PAVE framework. The inherent capability of PAVE to execute modules in parallel ensures that the AR system will scale over multiple processors.

The PAVE framework allows creation of visual effects by building subgraphs connecting effect modules. An interface has been designed that can link these subgraphs to virtual objects. This allows for easy integration of visual effects into the AR environment.

Since the visual effects can be very time consuming we have designed a mechanism that determines whether virtual objects are visible or not, and on that basis control whether effect subgraphs should be active or inactive.

The AR museum's functionalty has been analysed to identify areas in the PAVE framework that needed new features and enhancements. This includes garbage collection, read-only and write semantics for datatypes, optional inputs and outputs on modules and the triggering/empty triggering mechanism. The optional inputs and outputs mechanism on modules has greatly improved the flexibility in PAVE and as a consequence also made the AR Museum more flexible. The garbage collection mechanism manages memory allocation efficiently and makes PAVE more robust to changing configurations of graphs. The triggering mechanism enhances

performance by reducing the execution of subgraphs to only when it is necessary. The read-only and write semantics helps reducing cloning of data, and increases overall efficiency of the system.

As part of realizing the AR museum, the intrinsic and extrinsic parameters are determined for the camera, which allows calibration of the camera image and alignment of the virtual and the real camera. The visualization has been made to take advantage of 3D acceleration hardware for better performance.

A prototype of the AR museum has been implemented and tested. It proved to scale well on a dual processor machine and maintain a stable acceptable framerate. The visual quality is quite good and the dynamic effects adds to the user experience. Regarding the registration of camera positions and orientation, our tests showed that the magnetic tracker can be somewhat inaccurate, esspecially when it comes to measuring orientation angles. This caused virtual objects to be more or less displaced, where some displacements where gross. The conclusion from this is, that more precise tracking equipment that is less sensitive to electromagnetical noise, is a neccessity to obtain better results.

In general the system satisfies our goals and has proven to be quite robust, fairly fast and stable. The concept of art on objects has shown to work quite well visually.

8 Future Work

In this section we briefly describe possible future directions that can be taken to expand on our project.

8.0.1 Enhanced Tracking

Since the magnetical tracking proved to be a somewhat imprecise method of registering the position and gaze of the camera, additional tracking methods would be required to reach an acceptable user immersiveness. A of way achieving this could be to examine ways of doing hybrid tracking by e.g. designing an optical tracking component.

8.0.2 Multiple Users

If the AR museum were to be experienced by using head-mounted displays, it would be desriable to allow multiple users at the same time. This would require modifying the museum graph and most likely the functionality of the components too. For instance, it should be investigated how the control of activating/deactivating the effect subgraphs will change when considering multiple users field of view.

8.0.3 Hardware Accelerated Image Distortion

We have chosen to calibrate the camera image in software. It would be possible to do this by taking advantage of graphics hardware by mapping the camera image onto a distorted mesh. This would not give a per-pixel accuracy and therefore it might be of interest to investigate a way of subdividing the mesh in areas where the image distortion is most pronounced. Another possibility could be to render the virtual objects into a texture and distort this in the above mentioned way instead of distorting the camera image.

8.0.4 Shadows

Making virtual objects cast shadows on each other and on real-world objects would be a obvious way of enhancing the user immersiveness. With the world knowledge and the rendering approach we have chosen it is possible to generate hardware accelerated shadows on modelled real-world objects.

8.0.5 Depth of Field Blur

A way of making the rendered objects blend better in with the real-world image could be to simulate a depth of field effect on the rendered image. The depth of field effect is due to different parts of an image being in focus and others not. The rendered image would need to be blurred in areas where the real-world image is out of focus. Realtime depth of field blur effects are possible on todays consumer graphics hardware [nvd01].

A BaseModule Specialization Example

This example shows how a module is implemented for PAVE framework, the module shown is a real module called BlendModule, we implemented last semester. The BlendModule takes two bitmap images, of FrameBuffer type[1], as input. Its Action method blends two images by some factor specified by an integer Value parameter. It delivers a resulting bitmap image (of FrameBuffer type) that contains a composite of the two inputs. Below is a C++ code example of the BlendModule implementation, the blend algorithm resides in the action method.

The source code shown, is for a plugin package, we have called *StandardModules*, that contains several other modules, but only the code for BlendModule is shown. The code for registering the BlendModule into the plugin DLL-file representing the *StandardModules* plugin package is also shown.

StandardModules.h:

```
#ifndef __StandardModules__
#define __StandardModules__

#include "Types.h"
#include "Base.h"
..........

//---------------------------//
// Blend module (declaration)
//---------------------------//
```

[1] A specialization of BaseType that contains a bitmap image buffer.

```
class BlendModule : public BaseModule
{
public:
   BlendModule(); // Constructor.
   ~BlendModule(); // Destructor.

   virtual void init(BaseTypeList* staticParams); // Intialization method.
   BaseTypeList* action(BaseTypeList* inputs, BaseTypeList* params); // Action method.

private:
   unsigned int m_imgSize;
};

..........

#endif
```

StandardModules.cpp:

```
#include "StandardModules.h"
..........

//---------------------------//
// Blend module - begin
//---------------------------//

BlendModule::BlendModule() // Constructor
{
   this->setTypeID("BlendModule");
   m_inputs->addBaseType(new FrameBuffer()); // Declare a FrameBuffer as input number one.
   m_inputs->addBaseType(new FrameBuffer()); // Declare a FrameBuffer as input number two.
   m_params->addBaseType(new Value()); // Declare a Value as parameter one.
   m_outputs->addBaseType(new FrameBuffer()); // Declare a FrameBuffer as output number one.

   m_imgSize = 256*256*4; // Fixed size of the images
}

BlendModule::~BlendModule() // Destructor
{
   // does not own anything that needs to be deleted.
};

void BlendModule::init(BaseTypeList* staticParams)
{
   // no initialization is necessary.
}

BaseTypeList* BlendModule::action(BaseTypeList* inputs, BaseTypeList* params)
{
   FrameBuffer* image1 = (FrameBuffer*) inputs->getBaseType(0);
   FrameBuffer* image2 = (FrameBuffer*) inputs->getBaseType(1);

   // create a result FrameBuffer to send along.
```

```
   FrameBuffer* resultImage = new FrameBuffer(256, 256, 4);
   unsigned char* drawbuf = resultImage->getPtr();
   unsigned char* rgb_pic1 = image1->getPtr();
   unsigned char* rgb_pic2 = image2->getPtr();

   Value* v = (Value*) params->getBaseType(0);
   unsigned int step = v->getValue();

   unsigned int invstep = 255 - step;
   unsigned int imgSize = m_imgSize;

   // Intel x86 assembler version of a blending algorithm.
   __asm
   {
     mov   edi, drawbuf
     dec edi
     mov esi, rgb_pic1
     mov ebx, rgb_pic2
     mov ecx, imgSize

     innerloop:
       mov edx, [esi]
       and edx, 0x000000FF
       imul edx, step
       inc esi
       shr edx, 8
       mov eax, [ebx]
       and eax, 0x000000FF
       imul eax, invstep
       inc ebx
       shr eax, 8
       inc edi
       add eax, edx  //eax = (color1*step)/256 + (color2*(255-step))/256
       mov [edi], al
       dec ecx
       jnz innerloop
   }

   // Create an outputlist container
   BaseTypeList* output = new BaseTypeList();

   // Add the result image to the outputlist
   output->addBaseType(resultImage);

   // Release the reference to the result.
   // Added after the enhancements of PAVE were implemented.
   resultImage->releaseRef();

   return output; // return the output list to parent Node.
}

//---------------------------//
// Blend module - end
```

```
//---------------------------//

...........

//---------------------------//
// Export module(s) here
//---------------------------//
__declspec(dllexport) BaseModule* moduleQueryFunc(int moduleNr)
{
   switch(moduleNr)
   {
   case 0:
      return (BaseModule*) new BlendModule();
   case 1:
      return (BaseModule*) new .................
   ...............................
   .
   .
   default:
      return NULL;
   }
}
//---------------------------//
//---------------------------//
```

The idea of exporting modules, is that an application can load the plugin DLL file at run time. When loaded it can call one single generic factory function (*moduleQueryFunc*) in the plugin and on behalf of the parameter given, the function returns the desired module instance.

Bibliography

[APG98] T. Auer, A. Pinz, and M. Gervautz. Tracking in a multi-user augmented reality system. In *Proceedings of the IASTED International Conference Computer Graphics and Imaging 1998*, pages 249–252, 1998.

[Cor98] T. H. Cormen. *Introduction to Algorithms*, page 89. MIT Press, 1998.

[Cre01] Creative Video Blaster WebCam 3 USB Manual, 2001. http://www.europe.creative.com/support/manuals/.

[Cro97] Thomas W. Crokett. An introduction to parallel rendering. 1997.

[CT95] Michel Cosnard and Dennis Trystram. *Parallel Algorithms and Architectures*. International Thompson Computer Press, 1995.

[Der99] Helmut Dersch. Correcting barrel distortion, 1999. http://www.fh-furtwangen.de/ dersch/barrel/barrel.html.

[FAS00] Fastrak specifications, 2000. http://www.polhemus.com/ftrakds.htm.

[KV98] Kiriakos N. Kutulakos and James R. Vallino. Calibration-Free Augmented Reality. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):1–20, January 1998.

[MKN96] K. Mase, R. Kadobayashi, and R. Nakatsu. Meta-museum: A supportive augmented reality environment for knowledge sharing. In *Proceedings of International Conference on Virtual Systems and Multimedia '96*, pages 107 – 110, 1996.

[msd01] Microsoft online developer center, 2001. http://msdn.microsoft.com/directx.

[nvd01] Nvidia developer relations site!, 2001. http://partners.nvidia.com/developer.

[Rek96] J. Rekimoto. Transvision: A hand-held augmented reality system for collaborative design. In *Proceedings of Virtual Systems and Multimedia (VSMM) '96*, 1996.

[SFH00] D. Schmalstieg, A. Fuhrmann, and G. Hesina. Bridging multiple user interface dimensions with augmented reality. In *Proceedings of ISAR 2000*, pages 249–252, 2000.

[SHC+96] Andrei State, Gentaro Hirota, David T. Chen, Bill Garrett, and Mark Livingston. Superior augmented reality registration by integrating landmark tracking and magnetic tracking. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 429–438. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Sta97] William Stallings. *Operating Systems*. Alan Apt, 1997.

[Ull75] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, June 1975.

[Val01] Jim Vallino. *Introduction to Augmented Reality*. PhD thesis, Department of Computer Science, Rochester Institute of Technology, 2001. Chapter 1, http://www.cs.rit.edu/ jrv/research/ar/introduction.html.