SIGGRAPH 2008
Class Notes: Don't be a WIMP
(http://www.not-for-wimps.org)

Johannes Behr[1]
Fraunofer IGD
Germany

Dirk Reiners[2]
University of Louisiana at Lafayette
USA

October 6, 2008

[1]johannes.behr@igd.fraunhofer.de
[2]dirk@lite3d.com

Class Description

Virtual and augmented reality have been around for a long time, but for most people they are movie fantasies. Very few people outside a few research labs have worked with or experienced these systems for themselves. On the other hand, interactive 3D graphics applications are ubiquitous, mostly in the form of games. More and more people are working in animation and games, creating models and programs for interactive 3D applications on standard monitors.

The goal of this class is to demonstrate that the leap to actual immersive or augmented environments is not as big as you might think. It explains how high-powered 3D graphics cards, mainstream applications of stereoscopic displays in 3D TV and movies, and webcams that achieve TV-quality images have significantly lowered the barriers to entry. And how, in combination with those hardware advances, freely available software based on open standards like X3D provides all the tools you need to access the elusive world of virtual and augmented reality applications. Following a summary of the basic principles of stereo displays, tracking systems and post-WIMP interaction metaphors, the main part of the course is a practical introduction to creating and running your own interactive and immersive applications.

Prerequisites

Basic knowledge of computer graphics. Understanding of what polygons, lights, and cameras are. Helpful but not required: graphics programming or 3D animation experience. This class is intended for attendees who are interested in interactive 3D graphics and might want to move beyond the WIMP (Window, Icon, Menu, Pointer) environment.

Instructors

Johannes Behr, Fraunhofer IGD, Germany

Johannes Behr leads the VR group at the Fraunhofer Institut für Graphische Datenverarbeitung in Darmstadt, Germany. His areas of interest focus on virtual reality, computer graphics and 3D interaction techniques. Most of the results of his recent work are available as part of the InstantReality Framework. He has an MS from the University of Wolverhampton and received his PhD from the Technische Universität Darmstadt.

Dirk Reiners, University of Louisiana at Lafayette, US

Dirk Reiners is a faculty member in the Center for Advanced Computer Studies (CACS) at the University of Louisiana at Lafayette. His research interests are in interactive 3D graphics and software systems to make building 3D applications easier. He has an MS and a PhD from the Technical Technische Universität Darmstadt and is the project lead for the OpenSG Open Source scenegraph.

Contents

Chapter 1

Introduction

This document contains additional material and tutorials to support and simplify different aspects of he VR/AR application development process. It's not about basic visualisation or tracking methods but about building applications based on available results, standards and systems. The target audience are people interested in using the technology rather than developing it.

1.1 What is VR/AR

We give some basic definitions for VR/AR but these are very limited and defining this topics is not the goal of the course, partially because even though the topics have been around for a long time there is no commonly agreed definition. The goal of the course is to give people the ability to start building interactive, immersive applications. Basic definitions of AR and VR can be found in the literature ([47, 53, 34, 30]) or online ([55, 56]).

Most of the literature defines some basic elements which are critical but variable for all VR/AR applications:

Virtual Content The content of the medium defines the virtual world. This imaginary space, manifested through a medium, defining any collection or number of objects in a space and the relations and rules in the corresponding simulation.

Immersion Being mentally and physically immersive are important criteria for VR applications. Whereby physical immersion is a defining characteristic of virtual reality and mental immersion is probably the goal of most media creators. Therefore it is important that there is a synthetic stimulus of the body's senses via the use of technology. This does not imply all senses or that the entire body is immersed/engulfed. Most systems focus on vision and sound. Some include touch and haptics, generally known as force feedback. Other senses are much harder to stimulate by a computer and are only touched in very few research environments.

Sensory feedback Sensory feadback is an ingredient essential to virtual reality. The VR system provides direct sensory feedback to the participants based on their physical position. In most cases it is the visual sense that receives the most feedback, as it is also the sense that brings most of the information from the environment into the human system anyway.

Interactivity For VR to seem authentic, it must respond to user actions, namely, be interactive. The system must produce sensory feedbacks according to the user action. Most systems give visual feedback with an updaterate from at least 30 times per second. More is desirable, and immersion gets lost when the time lag between user actions and sensable reactions exceeds 100 ms.

There is not a single interaction and navigation method or device that would define VR nor AR. There is no Window, Icon, Menu, 2D-Pointer defining a single methoper but every application designer is free to choose whatever is most attractive or appropriate for the current set of goals. This gives the developer on the one hand a lot of freedom but on the other hand asks for a new set of development tools, standards and methods.

1.2 Building VR/AR Applications

Early Virtual Reality (VR) applications where mainly focusing on "virtual prototypes", "ergonomic evaluation", "assembly-disassembly" and "design review". The automotive industry was one of the driving forces of Virtual and Augmented Reality and today this technology is used in many application areas. The automotive industry learned how to use VR successfully, saving time and costs by building and evaluating virtual prototypes. But in fact, only large companies could take advantage of virtual reality, because of the high initial hardware and software costs. Consequently small companies were not using VR/AR and also for many application domains, using VR/AR was not profitable.

Things have changed significantly, and today the initial costs of the main expense factors, the computer graphics hardware and projection systems, are at least an order of magnitude lower than 5-10 years ago. Nearly every modern PC is equipped with a 3D-graphics card (e.g. nvidia Geforce, ATI Radeon), which is able to deliver graphics performance easily outperforming the high end graphics systems from last decade (e.g. Silicon Graphics Onyx systems), which were built to run VR/AR applications. Furthermore VR/AR systems have evolved a lot: Abstract device management is commonplace today, there are some well accepted base technologies, e.g. X3D, and the introduction of component models offer great flexibility, allowing application designers to tackle a wide variety of application domains with a single software system (all-purpose), including Augmented Reality (AR) based applications.

Chapter 2

X3D as a Basic Technolgy

There is no standard method or tool for VR/AR application development. This course material and tutorials are based on X3D as one of the few widely accepted industry standards in the field and the InstantReality framework, because it is freely available and supports the wide variety of applications and technologies that are covered in this course.

2.1 Utilizing X3D for VR/AR Development

The InstantReality toolkit (IR) [16, 11] is an open/free environment for VR/AR applications developed at the Institute for Computer Graphics (IGD) in Darmstadt, Germany, which utilizes the X3D ISO standard as application description language.

One of the main reasons for starting the project was the ever present need for a faster and more efficient application development and teaching tool. Therefore, the goal was to define abstractions for interaction and behaviour descriptions, which work well for VR and AR applications but which is also suitable for beginnes

Like most traditional toolkits, IR uses a scene-graph to organize the data, for spatial as well as logical relations. In addition to the scene description, a VR/AR application needs to deal with dynamic behaviour of objects, and the user interaction via the available input devices. The major drawback of traditional VR/AR-toolkits is the fact that the application developer has to implement a controller to handle all aspects of both behaviour and interaction for most applications anew. To overcome this drawback, modern component design models providing techniques like connection oriented programming [51] were employed for IR. The basic idea is that the system should provide a component-framework which allows the user to load and wire different components to fulfill the widest possible variety of different application requests.

Instead of designing another application model, we adopted the basic idears of the X3D ISO standard [24].

The X3D node is not just a static graph-element but defines a state, state-changes, and input/output slots for every node. Application development is done by instantiating and wiring nodes, which are provided by the system.

The use of X3D as an application programming language leads to a number of advantages over a proprietary language:

- It is integral to an efficient development cycle to have access to simple yet powerful scene development tools. With X3D, the application developer can use a wide range of systems for modelling, optimizing and converting, as the X3D standard is supported by most major modelc reation tools.
- The interface is well defined by a company-independent ISO standard.
- Due to platform independence, development and testing can even be done on standard desktop computers.
- X3D and the used scripting language based on the widely used ECMAScript (aka JavaScript) are much easier to learn and teach than the low level interfaces often provided by traditional VR/AR toolkits.

- There is a growing number of CAD and simulation packages which export static and dynamic X3D worlds.
- There is a great number of books and tutorials available.

From an application developer's point of view, the IR runtime environment acts and feels very much like a web-client based browser.

However, since our goal was not to build another X3D web client but a full feature VR environment, the system includes some additional aspects not found in traditional X3D clients:

- The X3D specification includes only high level sensors. The X3D sensors recognize that "the user" has touched or turned some part of the world but do not define how a specific input device or user action has to be handled. The X3D specification of high level sensors has to be reinterpreted for immersive environments. Additionally, the system must provide low level sensors that stream data (e.g. stream of float values) from the real to the virtual world and vice versa.
- The system utilizes the concept of nodes and routes not only for the scene itself but also for the browser configuration and any other dynamic aspect. Every dynamic component (e.g. Job, Window) in IR is a node with fields, communicating with slots living in a specific namespace and hierarchy. The namespace type defines the use and the types of nodes that can live in the namespace. The route mechanism not only defines the communication channels directly, but also the thread paths used to parallelize the system execution indirectly.
- The VRML standard allows the developer to create prototypes with behaviour scripting in Java or JavaScript/ECMAScript. In order to achieve maximum performance, we need the ability to extend the node pool with native C++ implementations and fetch these nodes without recompiling or linking the toolkit.
- We should provide standard interfaces (e.g. HTTP, SOAP) so that other software packages (e.g. simulation packages) can utilize IR just like a visualisation and interaction service provider.

We tried to address these different requirements in the design and development of the IR system and present and discuss some of our results in this paper.

2.2 Related Work

These days, there are very few immersive VR applications which are really built from scratch, just using a hardware abstraction library for graphics (e.g. OpenGL [46]) and sound (e.g. OpenAL [38]). Most systems utilize at least a scene graph library like OpenSG [43], OpenSceneGraph [39], Performer [45] or Inventor [50].

Most VR frameworks include abstractions for input devices and multi-screen and stereo setups [21, 32, 33]. But there are only very few, which really provide abstractions for behaviour and animation description [25, 57]. However, there are some VR Systems which even provide connection oriented programming techniques as part of the application development model: the Avocado [52] and Lightning [23] systems are VR systems using the concepts of routes for internode communication to define the behaviour graph, but are not related to VRML or X3D. Both systems are based on the Performer [45] library and define a scene-graph behaviour graph, which is very closely related to the Performer rendering system. The OpenWorld [27] system is a VRML based development system, which also provides support for stereo views, but is not a full feature VR-System with various abstractions for input and output systems.

Stiles et al. [49] adapted VRML for immersive use and was especially concerned about sensor and viewpoint handling in immersive environments. Our work on low level and high level sensors is partly based on his results.

2.3 Hello world in X3D

This section shows you how to create a "Hello world!" in X3D

2.3.1 What we need

This section will show you how to write the famous “Hello world!” program in X3D. All you need is a text-editor and a X3D-browser like the IR viewer.

2.3.2 How we do it

First start up the text-editor and enter the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D profile='Immersive'>
    <Scene>
    </Scene>
</X3D>
```

Save the file and call it *simple.x3d*

Sweet! You just created the most simplest X3D world possible - an empty one. You can check that it is empty by opening it with your favourite X3D browser and watch if the browser gives any warnings. If he does consult the browser documentation if it supports the X3D *XML encoding* - some browser might only support the X3D *Classic VRML* encoding.

2.3.2.1 Understanding the code

Let’s have a look at the code:

```
<?xml version="1.0" encoding="UTF-8"?>
```

This is the XML file declaration and it’s used for easy identification. The file declaration should be present in every valid XML file so just copy-and-paste it there.

The file declaration is followed by the X3D document root element which specifies a *profile* :

```
<X3D profile='Immersive'>
```

A complete overview of the profiles concept can be found in the X3D specification . Simply put it tells the browser which kind of nodes the world uses so that the browser can check if he supports the profile (and the nodes associated to that profile). The *Immersive* profile used here is targeted at *“implementing immersive virtual worlds with complete navigational and environmental sensor control”*

Next comes the empty Scene element:

```
<Scene></Scene>
```

In the following we will put some text into our scene.

2.3.3 Show the words

What is missing in our world is the content. Since we want the two words “Hello world!” in fully blown 3D we add the following Text element to the Scene so that the code now looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D profile='Immersive'>
  <Scene>
    <Shape>
      <Text string="Hello world!" />
    </Shape>
  </Scene>
</X3D>
```

We added a Shape which contains a Text geometry - and that’s it! Save the file to `helloworld.x3d` and start it up. The words “Hello World” should be shown by your X3D browser.

You could now continue to play around with the Text , e.g. changing the *depth* or the *fontStyle* .

Files:

- simple.x3d
- helloworld.x3d

2.4 Creating X3D Applications

This section is the first in a group of sections which tries to teach how to develop VR/AR applications with the InstantReality Framework. This section tries to explain some basics but is mainly a starting point to get you pushed into the right direction. These sections are no sections on VR/AR in general or developer sections for framework internals.

2.4.1 Relation to X3D

One goal of the IR design was to make it really easy to write immersive VR and AR applications. One basic idea was, not to use a single GUI-Tool or method but a software abstraction layer which would allow us to define rich, dynamic and highly interactive content. Since there is no ISO/ANSI/whatever standard for such an VR/AR application-interface, we tried to adapt something which is well known, practical and very close to our domain: X3D.

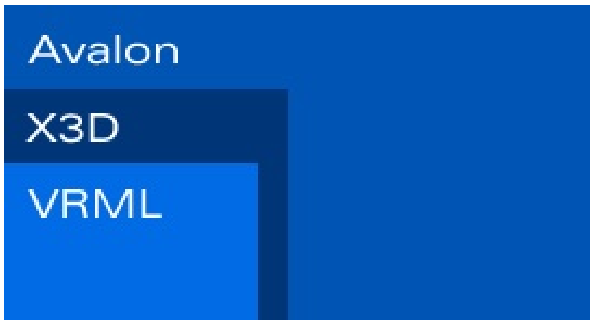


Figure 2.1: X3D as feature and node subset of Avalon/IR

The X3D standard is a royalty-free ISO standard file format and run-time architecture to represent and communicate 3D scenes and objects using XML. The standard itself is a successor of the VRML97 ISO standard. We utilized and extended this standard to fit the requirements we had from the VR/AR domain. This way our nodes and features are really a superset of X3D/VRML, and every X3D application is a valid Avalon application.

2.4.2 X3D Sources to read

To get started you have at least to understand the basic concepts of VRML/X3D. The official webpage has the X3D spec online which is not what you would like to read in the beginning. The developer section on the Web3d page holds some interesting links to sections and software tools. If you prefer some text books you should check out the X3D Book from Don Brutzman and Leonard Daly. Sometimes you can find some interesting, possibly used and really cheap VRML books, like the “VRML Handbook” or e.g. “VRML - 3D-Welten im Internet”.

2.4.3 X3D Conformance

Most VRML/X3D files should work right away. If you have some files that do not perform right, please visit the forum or write us a mail including the file and/or a short description.

However, there are some known bugs and not yet implemented VRML/X3D features. The online documentation should list the state of every implementation.

2.4.4 Growing number of Nodes and Components

VRML was a static ISO standard which only defined a fixed set of 54 nodes. X3D almost doubled this number in the first version and indroduced Components and Profiles to group and organize all nodes. X3D, in contrast to VRML, was designed to be extensible and therefore people keep revising the standard or build application specific node-sets like we did. If you look at the documentation you can see how the node-sets were growing over time from VRML 2.0, X3D 3.0, X3D 3.1 up to X3D 3.2. In addition to the nodes new components (e.g. for shaders) where also introduced.

2.4.5 Mimetypes and encodings

X3D supports not just a single data encoding but three in order to fulfill different application requirements. First of all there is an XML based encoding, which is easy to read and write for humans as well as for computers.

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D profile='Immersive'>
  <Scene>
    <Shape>
      <Text string="Hello world!" />
    </Shape>
  </Scene>
</X3D>
```

The VRML syntax has some disadvantages concerning parsing and the like. However for historical reasons the VRML encoding is still supported in X3D as the so-called classic encoding:

```
#X3D 3.0 utf8
Shape {
    geometry Text {
            string "Hello, World"
    }
}
```

The binary format can be loaded very efficiently but is is not readable by human beings at all (well I know somebody that can read parts of it but this is another story).

```
0000000 00e0 0100 7800 02cf 7378 2864 7468 7074
0000010 2f3a 772f 7777 772e 2e33 726f 2f67 3032
0000020 3130 582f 4c4d 6353 6568 616d 692d 736e
0000030 6174 636e f065 023c 3358 7844 7006 6f72
0000040 6966 656c 4643 6c75 786c 7606 7265 6973
0000050 6e6f 3342 302e 817b 1881 6f6e 614e 656d
0000060 7073 6361 5365 6863 6d65 4c61 636f 7461
0000070 6f69 086e 6826 7474 3a70 2f2f 7777 2e77
0000080 6577 3362 2e64 726f 2f67 7073 6365 6669
0000090 6369 7461 6f69 736e 782f 6433 332d 302e
00000a0 782e 6473 3cf0 5304 6563 656e 043c 6853
00000b0 7061 7c65 5403 7865 7874 7305 7274 6e69
00000c0 0867 2205 6548 6c6c 206f 6f77 6c72 2164
00000d0 ff22 ffff
```

All those data files can be gzip-compressed and the system loader can handle the compression automatically. The loader/writer framework supports all three encodings and VRML 2.0 equally well and in addition tools (e.g. aopt) to convert between these formats. You can not mix different encodings in the same file but you are free to mix the encodings in a single application (e.g. having a foo.wrl inline in you bar.x3d world) .

Right now we are using the XML and VRML encoding through-out the sections. But we will include some translation mechanisms later on.

Files:

- helloworld.x3d
- helloworld.x3db
- helloworld.x3dv

2.5 Get your engine right

This section shows you how to build up different elements of a more complex application by providing a scene, engine and context-setup description. The different parts allow the author to cluster and specify various aspects of your final application. The Scene contains the dynamic spatial content, the Engine contains device and environment specific parts (e.g. window or cluster setup) and the ContextSetup defines the application independent parts like the path of your java installation. The ContextSetup parameter are normally stored in the system preferences (e.g. plist-files on Mac or registry entries on Windows) but this sections shows how to overwrite the settings per application

2.5.1 Scene

The Scene defines the default SAI execution context and the root node of your spatial world. As shown in the following example it is only valid as X3D element. There is only one Scene per execution context or file allowed. However you can Inline extra X3D-files which must include a Scene element.

```
<X3D>
    <Scene>
        <Shape>
            <Box size='1 2 9'/>
        </Shape>
    </Scene>
</X3D>
```

2.5.2 Engine

The scene describes the content of and interaction with the world - they do not describe on which kind of output device or how the content should be displayed. This is where the engine file comes in. Imagine you want to display your world in red/green stereo, or you want it to be displayed on a three-sided power-wall, etc. This can all be done via an engine setting without even touching your world. This separation between the world and the engine file allows you to visualize your world on a variety of output devices without changing the world itself.

The next example shows a very simple engine setting as an example of a basic engine file, which only contains a RenderJob . The RenderJob includes a WindowGroup, an abstraction used to combine render threads, which itself includes two windows. Therefore the engine will start two windows on the desktop showing the same scene and camera.

The Engine sub-tree is a own X3D-namespace and can include any nodes which are not SceneBaseNodes. For example scripts and event-filter are very usefull as part of your engine. The Engine also does not have to be complete. Missing jobs will automatically be inserted by using the Engine.requiredJobList values. More complex engine settings can be found in the cluster section.

```
<X3D>
<Engine>
    <RenderJob DEF='render'/>
        <WindowGroup>
           <Window DEF='win1'/>
           <Window DEF='win2'/>
        </WindowGroup>
    </RenderJob>
</Engine>
<Scene>
  <Inline url='theWorld.X3D'/>
<Scene>
</X3D>
```

2.5.3 ContextSetup

The ContextSetup stores all setup and preferences per execution context which are related to specific nodes or components but independent of a specfic instance of a node. You can set different limits, e.g. texture size and system pathes (like the Java classpath and various other properties which are usual stored in the player plist or registry entry).

If and only if you really *have* to overwrite it in the application file you can create an extra ContextSetup child as part of the X3D element.

```
<X3D>
<ContextSetup maxTexSize='2048'>
<Engine>
  ...
</Engine>
<Scene>
  ...
<Scene>
</X3D>
```

The attributes and the values are not fixed and dynamically build up from the loaded component. Use the WebInterface access point in order to see the configuration of your running system. The following section includes a list of attributes for the beta3 player including all components.

```
cdfPath /Users/jbehr/src/Avalon/tmp/cdfFolder ()
  defines the for the cdf data
multiContext FALSE ()
  Run more than one context
appThreadCount 16 ()
  The number of application threads
forbidFieldFree FALSE ()
  forbid Field free optimize
forbidNodeFree FALSE ()
  forbid Node free optimize
rayIntersectMode auto (auto,lowMem,fast,fastest)
  defines the ray intersection mode
forceSingleDataTypeRoute FALSE ()
  force routes to connect only slots with same type
explicitGZIPTest FALSE ()
  use the explicit gzip test
forbidReindex FALSE ()
  forbid geos to reindex to share properties
forbidStripFan FALSE ()
  forbid geos to strip/fan the mesh data
forbidVertexResort FALSE ()
  forbid vertex resort to improve cache usage
forbidSingleIndex FALSE ()
  forbid singe-index opt. for static/unshared obj.
forbidIndexResize FALSE ()
  forbid optimized-index opt. for static/unshared obj.
forbidDList FALSE ()
  forbid DisplayLists for static obj
forbidVBO FALSE ()
  forbid VertexBufferObjects for static obj
forbidNormalUpdate FALSE ()
  forbid normal updates for dynamic obj.
geoPropertyUpdateMode memcpy (none,memcpy,stl-assign,stl-swap)
  mode used to update backend geoProps
maxTexSize 4096 ()
  max texture u,v,w size, 0 means hardware limit
forceTexCompress FALSE ()
  force textures to use compressed internal types
frontCollision TRUE ()
  do front-collision check while navigating
zRatio 20000 ()
```

```
  ratio to specify the z-near limit
defaultJobList TimerJob,InteractionJob,ExternalInterfaceJob,WebServiceJob,CollisionJob,SoundJ
  Defines the default jobs
showStatusMessage true ()
  show the status message in render view
infoScreenAnimationTime 0.5 ()
  info screen animation time in seconds
logoMode auto (auto,on,off)
  logo mode
forceSingleThread FALSE ()
  force single thread app/render cycle
keyAssignMode ctrlMod (autoSwitch,app,sys,ctrlMod,altMod,shiftMod)
  defines how key-events are assigned to sys/app
binSearchKeyIndex FALSE ()
  use binSearch to find the key value
tessellationFactor 1 ()
  tessellationFactor (from 0 to 1)
ecmaScriptShareRunTime FALSE ()
  ecmascript share-runTime
ecmaScriptGlobalMem 8388608 ()
  ecmascript global system mem size
ecmaScriptLocalMem 8192 ()
  ecmascript local script mem size
ecmaGCPerFrame none (none,auto,force)
  set GarbageCollection mode per frame
javaVMPath  ()
  The path to the Java virtual machine
javaClassPath /Users/jbehr/src/Avalon/java/instantreality.jar ()
  The Java class path
javaOptions  ()
  Options that are transfered to the Java virtual machine
cgVertexProfile auto (auto,arbvp1,vp20,vp30,optimal)
  Cg vertex shader profile
cgFragmentProfile auto (auto,arbfp1,fp20,fp30,optimal)
  Cg fragment shader profile
rigidBodyTrigger Collision ()
  name of the rigid body physics trigger
```

2.5.4 The Scene node in classic encoding

The classic encoding normally does not include an explicit Scene node. All root nodes of a single file are added to the scene as children. Here we extent the spec by allowing to explicitly set context root nodes like ContextSetup, Engine or Scene nodes.

```
#X3D 3.0 utf8

Engine {
}
Scene {
  children [
    Transform {
            ...
    }
  ]
}
```

This section just tries to explain the Context root nodes, like Scene, Engine and ContextSetup-nodes, and how to build up an application using only one or none of them. More useful examples can be found in the cluster and rendering sections .

An execution context will always have an Engine and ContextSetup node. In most cases there is only an explicit (=X3D encoding) or implicit Scene (=classic encoding) and the runtime system will automatically create the object.

Chapter 3

MultipleViews and Stereo

In this section we will outline some of the modifications and extensions for running X3D applications in immersive environments with multiple views and stereo setups: which nodes and techniques we have to adopt and which additional nodes are useful and necessary.

The X3D standard only supports a single Camera with the bound Viewpoint. The Viewpoint is a node in the scengraph defining a position, orientation and fieldOfView. There is no Window, Viewport or clipping area defined. For different VR/AR scenarios it is essential to drive Multi-screen and Multi-pipe setups. Therefore we have to define some extensions and refinements to fulfill the requirements we have.

3.1 Rendering

The proposed X3D Specification Revision 1 [24] includes two new components, Layering and Layout, which provide nodes and functionality to render and layout different scene-parts in different layers. The Layer and LayerSet nodes define the sub-trees and rendering order but do not define what kind of composition method is used. Layer nodes are intended to create special 2D-/3D-interaction elements such as heads-up displays or non-transforming control elements. With the new Viewport node additional clip boundaries can be defined, but they only refer to a single render window. There is no notion of how this information shall be treated in a multi screen cluster setup. This gets even worse when having a closer look at the Layout component. Because of nodes like the ScreenFontStyle and its pixel-specific addressing it is mainly designed as a means to provide some additional information and with desktop applications and interaction metaphors in mind, but is not applicable for immersive systems.

Augmented (AR) or Mixed Reality (MR) applications in general require methods that render different layers of information - at least some sort of video-stream as background and 2D- and 3D-annotations. Even more complex layering techniques with certain compositing methods, including mechanisms for general multi-pass techniques as proposed in [**?**] with the *RenderedTexture* node, are needed to implement image-based rendering techniques like for instance dynamic HDR glow or motion-blur. For such advanced rendering effects which are - depending of the type of application - quite essential for evoking the sensation of presence and immersion, the possibility to render window-sized and view-aligned quads and additionally some way to control the composition method is needed. Besides this especially in immersive environments real time shadows are needed for depth cues and correct perception.

For desktop applications and simple mono projector systems image-based rendering can be achieved by using an extended LayoutLayer node from the Rev1 specification, and additionally introducing novel *X3DAppearanceChildNode* types for advanced render state control for image compositing. This approach can be extended for stereo systems by introducing special shader uniform variables denoting such information like left/right eye (named 'StereoLeftEye' in the following pixel shader code fragment) or even a cluster window id. This way creating stereo textures can be easily accomplished.

Another challenge is how to treat these different kinds of layers in really immersive environments like e.g. the CAVE. Here, the correct positioning of user interface elements can be consistently handled with our special *Viewspace* node, which transforms its children to the coordinate system of the current active viewpoint. But there still exists no continuous conceptual model for handling pure effects layer nodes for mono/stereo and all display types ranging from a desktop PC to a tiled display cluster system.

Another important issue are multi resolution surfaces (e.g. LOD or adaptively tessellated NURBS) on the one hand and simulation systems (like physics and particle systems) on the other hand on multi screen setups. The latter can especially lead to problems when the simulation is non-deterministic and distributed across a cluster for all render windows. If the simulation calculations are not bound to the application thread but run on different client PCs, the result is not necessarily the same and might therefore lead to rendering errors. Similar problems can occur with multi resolution meshes. If the view dependent tessellation is different on neighboring windows of tiled display systems this can also lead to artifacts.

3.2 Stereo Basics

This chapter is not a section at all but a brief excursion into optics. It gives an overview over the human eye as a tool to perceive our world stereoscopically. It will also introduce those factors which are responsible for depth perception and which are further used to generate an artificial stereoscopic view in virtual reality environments. As the trick is to create and present a different image for each of your eyes, the last section will discuss the most popular technologies to get this done in a more or less simple way.

There are no preconditions.

3.2.1 Depth perception of human eyes

There are two categories of depth perception, the monocular perception related to only one eye and the binocular perception related to both eyes.

3.2.1.1 Monocular depth perception

Occlusion Farther objects are being occluded by nearer objects.

Perspective Objects of the same size are bigger if they are nearer to the viewer.

Depth of field Adaption of the eye lens warping, so called accomodation, focuses objects in a specific distance to the viewer. Other objects with a different distance to the viewer appear blurred.

Movement parallax If objects with a different distance to the viewer move with the same speed, nearer objects appear to be faster than farther objects.

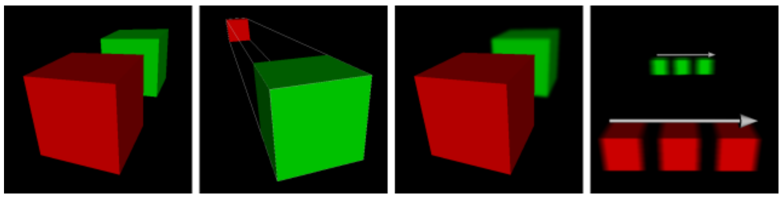


Figure 3.1: Monocular depth perception: Occlusion, perspective, depth of field and movement parallax

3.2.1.2 Binocular depth perception

- **Parallax** (**Each of our eyes sees a slightly different image of the world. This is because each eye is a separate camera which**)

  grabs the environment from a different position. In the image below you see two eyes focusing the black point. The **optical axes** show the alignment of the eyes and the projected point on the retinas. There is another point in green which is also projected. Those points have the same distance *L* and *R* to the optical axes on both retinas, they are **corresponding points** . Each point on the **horopter** , let's say a focus circle, has these characteristics. Points which are in front or behind the horopter have different distances, *L'* and *R'* , to the optical axes on the retina. The difference between *R* and *L* is the so called **disparity** , which is positive or negative dependent on the position in front or behind the horopter. Points on the horopter have a disparity of 0.

- **Convergence** (**The distance between our eyes is fixed but the angle depends on the distance of a focused object. If we watch**)

  the clouds in the sky the eye orientation is nearly the same but when we look at a fly which sits on our nose, the left eye looks more to the right and the right eye to the left, so we are squiting. This angle between the optical axes of both eyes, the convergence, gives our brain a hint about the distance to an object.

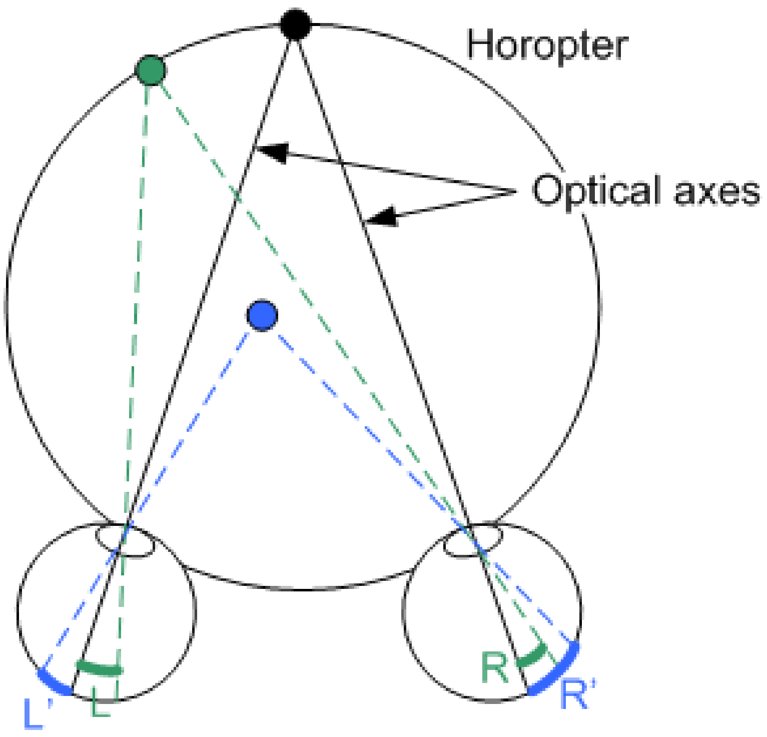


Figure 3.2: Binocular depth perception: Disparity on focused and non-focused points

3.2.2 Depth generation in VR

Generation of stereoscopic viewing in virtual reality environments depends mainly on the above described parallax and convergence, which go hand in hand. On a projection plane two separate images are displayed, one for the left and one for the right eye. Therefore the scene must be rendered from two different cameras, next to each other like the human eyes. A separation technique has to make sure that one eye receives only one image but this issue is discussed in the next chapter.

As each eye receives its appropriate image, their optical axes have a specific angle which yields to the convergence. The brain recognizes both points as the same point and interprets it as it was behind the projection plane or in front of it.

Parameters which affect the convergence are the distance to the projection plane which is called **zero parallax distance** and the **eye distance** . With these parameters depth impression can be adjusted.

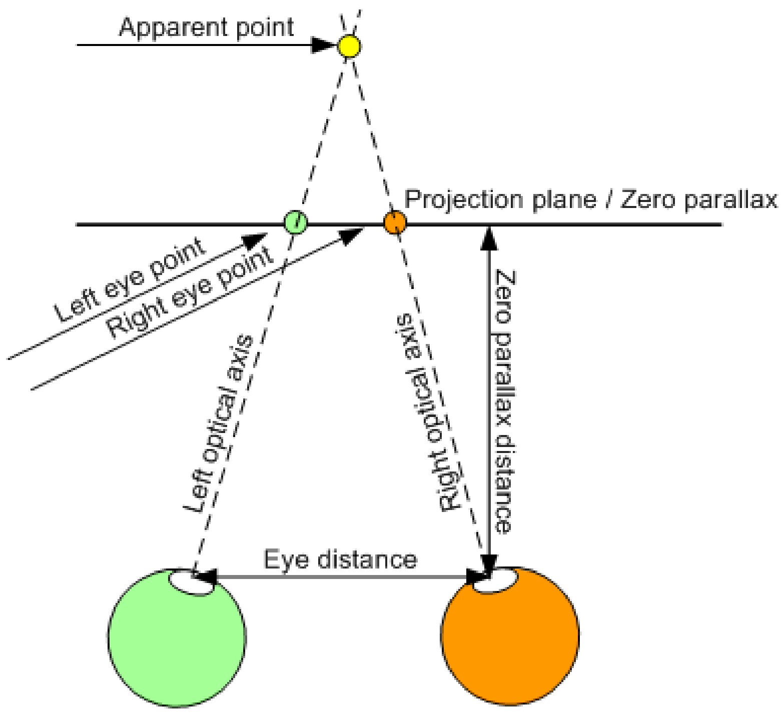


Figure 3.3: Stereo projection in VR

3.2.3 Eye separation

Different images for both eyes have to be generated, presented and redivided to the eyes. How these three steps are performed depends on the chosen technology. Those are categorized into active and passive approaches.

Active eye separation means a presentation of left and right images one after each other with high frequency while the eyes are alternately being shut in the same frequency by shutter glasses (see image below). The advantage is to see the original images without reduction of colors or color spectrums. On the other hand you need hardware (graphics card, beamer) which is able to display images in a high frequency of about 120 Hz as well as synchronisation with shutter glasses.

With passive stereo, images for the left and right eye are displayed simultaneously. A common approach is to separate color channels in order to show the blue channel for the left eye and and the red channel for the right eye. A user wears glasses that filter the red or blue channel away as you can see in the left image below. The big disadvantage is the corruption of proper colors.

A better approach in this direction is the technology of Infitec , which doesn't split whole color channels but color spectrums in each channel. For each eye three different bands - for red, green and blue wavelenths - are filtered, with the result to get a total of six bands: right eye red, right eye green, right eye blue, left eye red, left eye green and left eye blue. The desired colors are not completely modified like in the color channel separation case, but only a few frequency regions are lost instead of a whole color channel.

Another concept of passive stereo is the separation by polarization filters. Light oscillates in different directions and a polarization filter just filters light in a way to let only parts of the light in a specific direction come through. So, for one eye, vertical oscillating and for the other, horizontal oscillating light is permitted. It's a simple solution as the filters are not very expensive and can be easily put in front of a video beamer and into cheap paper glasses. The downside is a loss of brightness and interesting effects when inclining the head. This effect can be reduced with **radial polarization** approaches instead of **horizontal and vertical** .

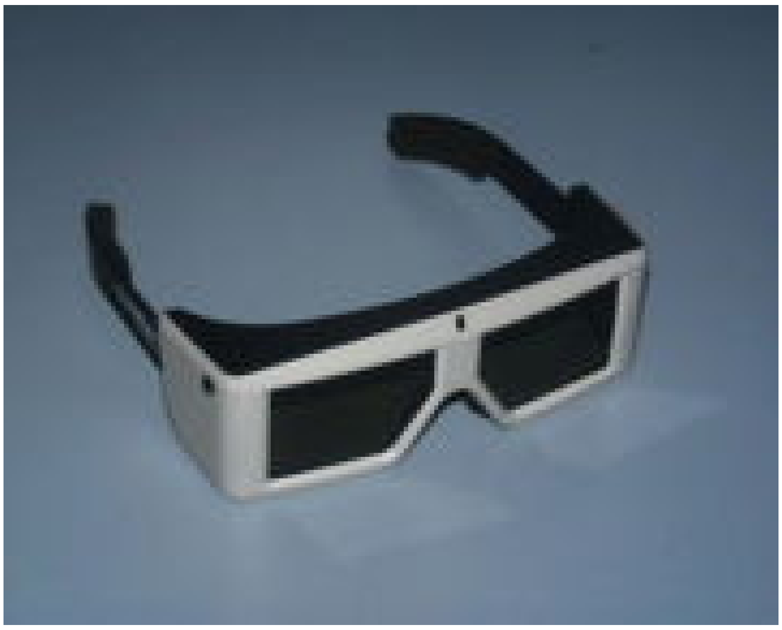


Figure 3.4: Active eye separation by shutter glasses

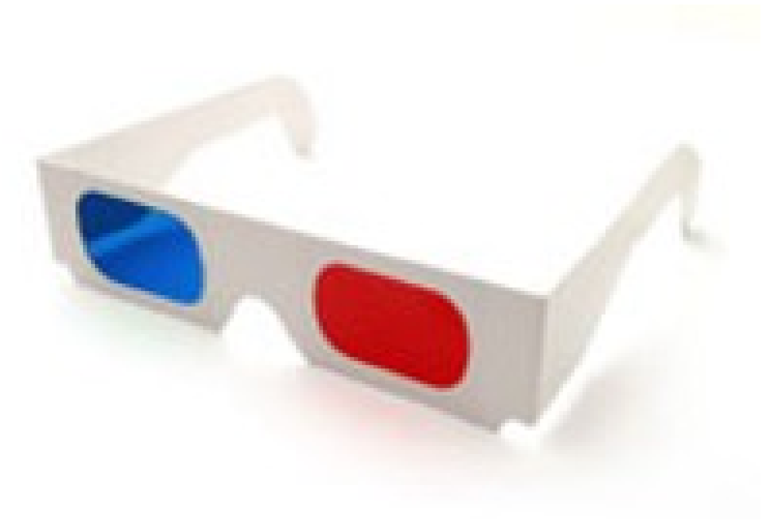


Figure 3.5: Eye separation by color

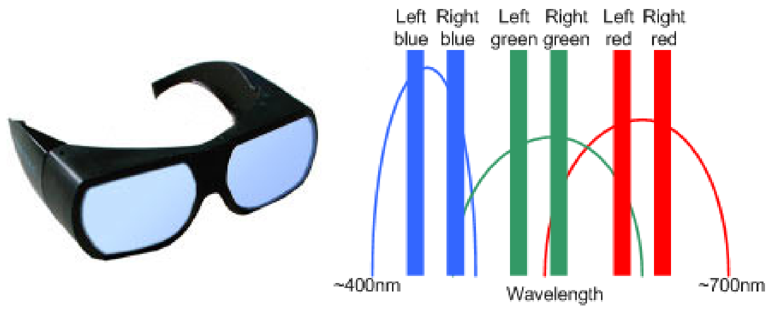


Figure 3.6: Eye separation by color spectrum (schematic illustration)

Figure 3.7: Eye separation by polarisation filter

3.3 Multiple Windows and Views

This section describes the configuration of multiple windows and multiple view areas per window. It is a precondition for the later parts of this section and the Clustering section.

Please read the “Engine” section if you are not yet familiar with the concept of different Context base elements.

3.3.1 Engine configuration

First we try to get some basic window settings right. Therefore we have to setup the RenderJob. In the engine section that is used for local rendering, the definition for the rendering in most cases looks like the following line:

```
DEF render RenderJob {}
```

Instant Reality automatically adds the missing configuration to produce an image on a local window. To be able to understand more complex rendering configurations, we’ll have a short look at the automatically generated configuration for a local window.

```
...
DEF render RenderJob {
  windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
          size 512 512
          views [
            Viewarea {
              lowerLeft  0 0
              upperRight 1 1
            }
          ]
        }
      ]
    }
  ]
}
```

```
...
```

With this configuration we have one local window with one Viewarea that covers the whole window.

3.3.2 Multiple view areas

To create a second view area we just have to add a Viewarea node and define the region where it should appear in the window. 0 means the left or bottom side and 1 is the right or top. If you set values greater than 1, they are interpreted as pixel values. In each view area, the complete scene is rendered.

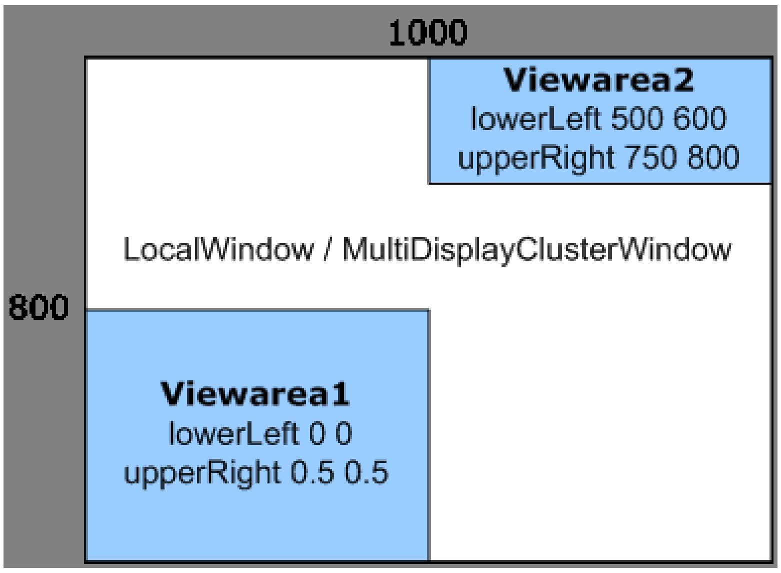


Figure 3.8: Concept of view areas in a window

The code for putting two view areas next to each other will look like this:

```
...
DEF render RenderJob {
  windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
          size 800 400
          views [
            Viewarea {
              lowerLeft  0 0
              upperRight 0.5 1
            }
            Viewarea {
              lowerLeft  0.5 0
              upperRight 1 1
            }
          ]
        }
      ]
    }
  ]
}
...
```

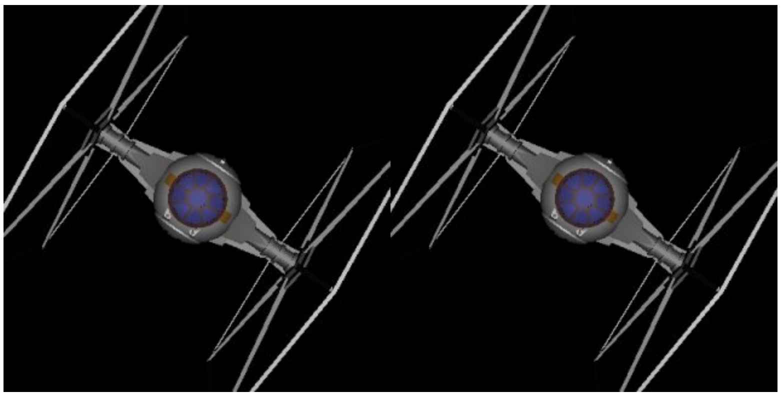


Figure 3.9: Two view areas next to each other a local window

View areas can be modified in a way to change the camera position and orientation by ViewModifier nodes. This is especially used for stereo configurations and CAVE environments (see appropriate sections or nodetype tree documentation).

3.3.3 Multiple windows

Multiple windows can be configured by adding LocalWindow nodes into the RenderJob section:

```
...
DEF render RenderJob {
  windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
          size 800 400
          position 0 0
          views [
            Viewarea {
              lowerLeft  0 0
              upperRight 0.5 1
            }
            Viewarea {
              lowerLeft  0.5 0
              upperRight 1 1
            }
          ]
        }
        LocalWindow {
          size 300 300
          position 500 500
          views [
            Viewarea {
              lowerLeft  0 0
              upperRight 1 1
            }
          ]
        }
      ]
    }
  ]
}
...
```

Files:

- MultipleViewareas.wrl
- MultipleWindows.wrl
- tie.wrl

3.4 Active Stereo

This section describes the configuration of the engine to achieve an active stereoscopic view of scenes by using synchronized shutter glasses.

Please read the sections “Multiple Windows and Views” as well as “Stereo Basics” in this category to get a good overview about stereo approaches and basic configuration issues regarding multiple views in Instant Reality.

3.4.1 Hardware

A “normal” graphics card uses a double buffer approach, a back buffer to write into and a front buffer to display in the meantime to avoid flickering. To use active stereo you should take a graphics card with quad buffer, i.e. four buffers. That means it uses a front and a back buffer for each eye.

As display you can either use a monitor or a video beamer which is able to display active stereo images interleaved in time.

Now you just need some shutter glasses which let you see the correct image for the appropriate eye. It is synchronized with the graphics card, mostly using infrared as you can see in the image below.

Figure 3.10: Infrared synchronized shutter glasses

3.4.2 Stereo modifier

If we want to do stereo, then we need two view areas. One for the left eye and one for the right eye. For stereo it is neccessary to modify viewing parameters. For this kind of modification there exists a number of modifiers in Instant Reality. For a simple stereo projection we have to use the ShearedStereoViewModifier :

```
...
Viewarea {
  modifier [
        SharedStereoViewModifier {
          leftEye  TRUE
          rightEye FALSE
          eyeSeparation 0.08
          zeroParallaxDistance 1
        }
      ]
    }
    ...
```

Depending on the eye which should be represented by this modifier, *leftEye* and *rightEye* has to be set to TRUE or FALSE. *zeroParallaxDistance* and *eyeSeparation* values are in metres, so they have good default values, if your scene is also modeled in metres. Otherwise you could either adapt the values or as a better approach, you should use a NavigationInfo node in the *Scene* namespace and set the *sceneScale* field to 0.01 if the scene is modeled in centimetres or 0.001 if the scene is modeled in millimetres and so on. The advantage is you can keep the stereo configuration fix for your setup and each scene and just need to change one value.

```
...
Scene {
  children [
    NavigationInfo {
      sceneScale 0.01
    }
    ...
  ]
}
```

3.4.3 Quad Buffer Stereo

Active stereo configuration is simple. First you have to tell the LocalWindow to use four instead of two buffers, the default setting.

```
...
LocalWindow {
  buffer 4
  ...
}
...
```

In the window we need two view areas, one for the left and one for the right eye. These areas are overlapping as we don't set specific regions for them. For each Viewarea we define a SharedStereoViewModifier which is responsible for the camera modification of the left or right eye respectively. It has also to be defined which buffers on the graphics card should be used by which view area. Therefore we set

```
...
Viewarea {
  leftBuffer TRUE
  rightBuffer FALSE
  ...
}
...
```

for the left eye view area and

```
...
Viewarea {
  leftBuffer FALSE
  rightBuffer TRUE
  ...
}
...
```

for the right eye view area.
After all the configuration looks like the one below:

```
RenderJob {
  windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
        buffer 4
        size 1024 768
          views [
            Viewarea {
              leftBuffer TRUE
              rightBuffer FALSE
              modifier [
                ShearedStereoViewModifier {
                  leftEye  TRUE
                  rightEye FALSE
                }
              ]
            }
            Viewarea {
              leftBuffer FALSE
              rightBuffer TRUE
              modifier [
                ShearedStereoViewModifier {
                  leftEye  FALSE
                  rightEye TRUE
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

Files:

- activeStereo.wrl
- tie.wrl (test model)

3.5 Passive Stereo

This section describes the configuration of the engine to achieve a passive stereoscopic view of scenes. It will distinguish between overlapped view areas and separated side by side view areas. The first one is performed by splitting color channels and using red/blue glasses. The latter can be used to achieve a stereo setup with two video beamers and appropriate filters for example. You should also check the preconditions for this section.

Please read the sections “Multiple Windows and Views” as well as “Stereo Basics” in this category to get a good overview about stereo approaches and basic configuration issues regarding multiple views in Instant Reality. Notice that this section is just about **passive** stereo.

3.5.1 Stereo modifier

If we want to do stereo, then we need two view areas. One for the left eye and one for the right eye. For stereo it is neccessary to modify viewing parameters. For this kind of modification there exists a number of modifiers in Instant Reality. For a simple stereo projection we have to use the ShearedStereoViewModifier :

```
...
Viewarea {
  modifier [
    ShearedStereoViewModifier {
       leftEye  TRUE
       rightEye FALSE
       eyeSeparation 0.08
       zeroParallaxDistance 1
    }
  ]
}
...
```

Depending on the eye which should be represented by this modifier, *leftEye* and *rightEye* has to be set to TRUE or FALSE. *zeroParallaxDistance* and *eyeSeparation* values are in metres, so they have good default values, if your scene is also modeled in metres. Otherwise you could either adapt the values or as a better approach, you should use a NavigationInfo node in the *Scene* namespace and set the *sceneScale* field to 0.01 if the scene is modeled in centimetres or 0.001 if the scene is modeled in millimetres and so on. The advantage is you can keep the stereo configuration fix for your setup and each scene and just need to change one value.

```
...
Scene {
  children [
    NavigationInfo {
      sceneScale 0.01
    }
    ...
  ]
}
```

3.5.2 Stereo by overlapping view areas

To receive a simple red/blue stereoscopic view, we have to overlap two view areas, display only one color channel per area (red or blue) and put a ShearedStereoViewModifier into both areas. After all the code looks like this:

```
DEF render RenderJob {
  windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
          views [
            Viewarea {
              red TRUE
              green FALSE
              blue FALSE
              lowerLeft 0 0
              upperRight 1 1
              modifier [
                ShearedStereoViewModifier {
                  leftEye  TRUE
                  rightEye FALSE
```

```
                    }
                  ]
                }
                Viewarea {
                  red FALSE
                  green FALSE
                  blue TRUE
                  lowerLeft 0 0
                  upperRight 1 1
                  modifier [
                    ShearedStereoViewModifier {
                      leftEye  FALSE
                      rightEye TRUE
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
```

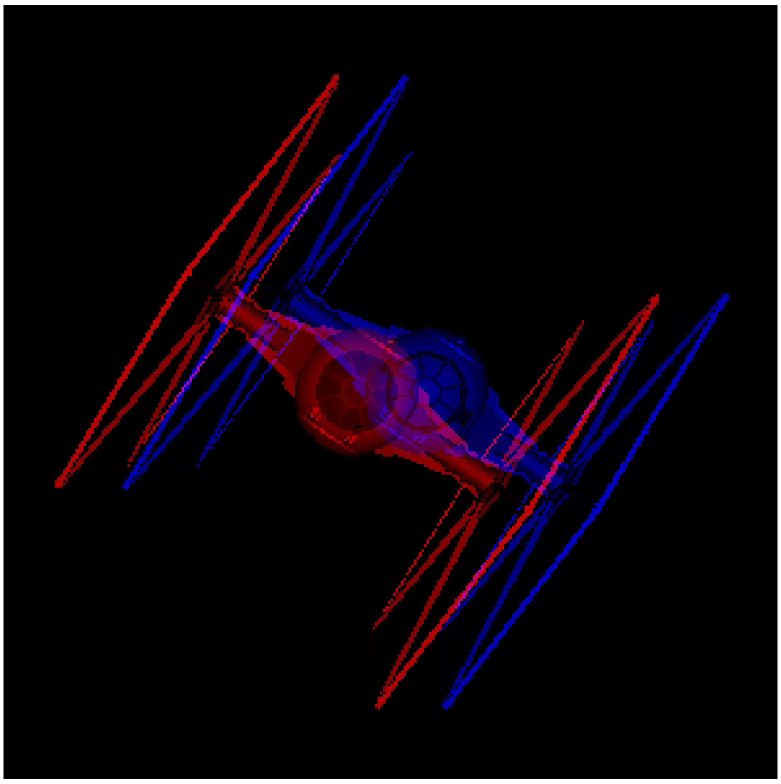


Figure 3.11: Stereo view due to separated color channels

The result will look like this. Everything you need now are some glasses with a red foil for the left eye and a blue foil for the right eye.

3.5.3 Stereo by separate view areas

Splitting color channels is the fastest variant of stereoscopic viewing. But we get much better results if we use the full color of “left” and “right” images. That’s the reason why we render both images side-by-side, choose our output device, let's say a graphics card with two outputs and a video beamer on each output. The beamer images are then superposed. To receive only one image per eye we use polarization or color spectrum filters (see Stereo Basics section) in front of the beamers and in front of our eyes.

To be able to show the image for the left eye on the left side of our window and the image for the right eye on the right side, we use two view areas again. The first is located from 0 - 0.5 and the second from 0.5 to 1.

```
...
LocalWindow {
  size 600 300
  views [
    Viewarea {
      lowerLeft  0   0
      upperRight 0.5 1
      modifier [
        ShearedStereoViewModifier {
          leftEye  TRUE
          rightEye FALSE
        }
      ]
    }
    Viewarea {
      lowerLeft  0.5 0
      upperRight 1   1
      modifier [
        ShearedStereoViewModifier {
          leftEye  FALSE
          rightEye TRUE
        }
      ]
    }
  ]
}
...
```

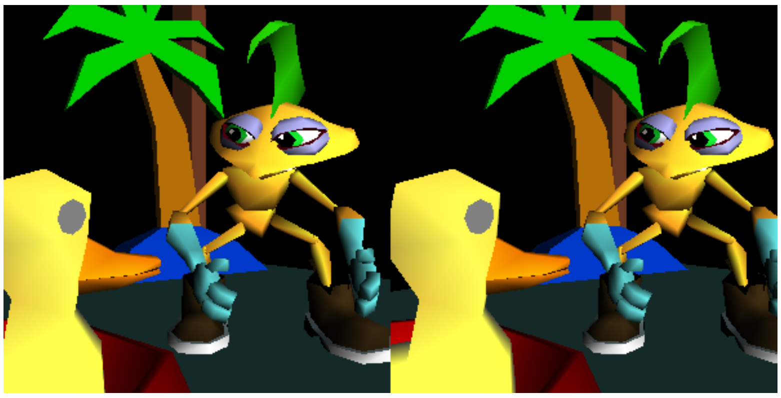


Figure 3.12: Stereo view side-by-side view areas

Your graphics card has to be configured so that the left side of the desktop is visible on the left output and the right side of the desktop is visible on the right graphics output. Additionally the size of the window must cover the whole desktop. This can be done by the following code.

```
...
LocalWindow {
  fullScreen TRUE
}
...
```

Now we have a simple stereo setup. The disadvantage is that a single PC is responsible for the simulation and the rendering of two images per frame. We can get a much better performance, if we are using 3 (or more) hosts where one is responsible for the simulation, one to calculate the right eye image and one to calculate the left eye image. To find out more about this, read the sections in the **Cluster section** .

Files:

- stereoOverlap.wrl (color channel split)
- stereoSeparated.wrl
- tie.wrl (test model)

Chapter 4

Interaction and Devices

4.1 Interaction

One major drawback of the X3D specification is the extremely limited support for Input/Output (IO) devices. The X3D specification does not mention devices at all - it only specifies some very high-level nodes that allow to control the way the user navigates in the scene (NavigationInfo node) and interacts with objects (PointingDeviceSensor nodes). The actual mapping between these nodes and the concrete devices connected to the computer is up to the browser. While this interaction model is sufficient for web-based 3D applications consisting of simple walk-through scenarios running on a desktop machine, it is much too limited for immersive VR and AR applications. For example, consider a driving simulator with a mock-up of the dashboard. The simulator (written in X3D) should be able to get the status of the switches on the dashboard e.g. used to switch the headlights on or off. Or consider a video see-through AR application where the X3D scene needs to get the video images from a camera attached to the system to put them into the background of the virtual scene. This demonstrates that there are much more usage scenarios for IO devices than the simple navigation and point-and-click scenarios currently supported by the X3D specification.

Our proposal, and the one implemented in IR, is to use a layered approach to integrate support for IO devices into X3D. On the basic layer, we propose a set of low-level sensors that allow the application to receive data streams from or to send data streams to devices or external software components. On top of this layer is a set of high-level sensors consisting of the traditional PointingDeviceSensor nodes mentioned in the X3D specification. This approach is similar to that of traditional 2D user interfaces where we have a low-level layer consisting of simple mouse and keyboard events and higher-level layers consisting of user interface elements like buttons, text fields and menus.

4.1.1 Low-Level Sensors

The purpose of the low-level sensors is to send or receive raw data streams without imposing any interpretation of these streams by the X3D browser. It is the sole responsibility of the X3D application to handle these data streams in a use- and meaningful way. Recently, there have been some competing proposals for low-level sensors [9, **?**, **?**]. These proposals suffer from two design flaws:

- It is generally not a good idea to specify nodes like “JoystickSensor”, “MidiSensor” or “TrackerSensor”. An approach like this means that we have to specify nodes for all kinds of devices available, which is obviously not possible. There will always be devices that do not fit into the set of device classes available in the X3D specification. As a result, this approach does not reliably and convincingly solve the problem.
- Even worse, these types of nodes require that the X3D application developer has to foresee what kinds of devices are available to the users of his application. This is obviously not possible and conflicts with the typical use-case of X3D applications - downloading them from a web server and running them on any kind of hardware available.

For these reasons, we proposed another solution [**?**] which does not have these drawbacks. The idea is to treat devices as entities consisting of typed input and output data streams. We admit that this is not as straightforward as using special nodes for each kind of device, but the benefits far

```
DEF cam Viewpoint { ... }
DEF headPos SFVec3fSensor
  { label "Head Position" }
ROUTE headPos.value_changed
   TO cam.set_position
DEF headRot SFRotationSensor
  { label "Head Orientation" }
ROUTE headRot.value_changed
   TO cam.set_orientation
Shape {
  appearance Appearance {
    DEF videoTex PixelTexture {}
  }
  geometry IndexedFaceSet { ... }
}
DEF frame SFImageSensor
  { label "Video Frames" }
ROUTE frame.value_changed
   TO videoTex.set_image
```

Figure 4.1: Template of an AR application in X3D.

outweigh this disadvantage. We get maximum flexibility, we do not bloat the X3D standard, and we get a stable basis for higher layers of abstraction. So we propose a set of sensors, one for each X3D field type. The interface of these nodes looks like this (“x” is just a placeholder for the concrete X3D field types SFBool, SFFloat, ..., MFBool, MFFloat, ...):

```
xSensor : X3DDirectSensorNode {
  x        [in,out] value
  SFBool   []       out   FALSE
  SFString []       label ""
}
```

The “value” exposed field is used to send or receive data values. The “out” field specifies whether the node is used to receive data values (“FALSE”) or to send data values (“TRUE”). Finally, the “label” provides means to map the sensor node to a concrete data stream of a device. The important point here is that we do not specify where the data comes from (e.g “Joystick 1/X-Axis”). Instead, in the label field we specify what the data values are used for (e.g. “Move left/right”). When the user loads an X3D scene that contains sensors that are not mapped to devices, a dialog window opens that lists the labels of the sensors. Next to each label is a drop-down menu that contains all devices that are currently connected to the machine and that have a matching type and direction. This is just the same procedure that we are used to when we start a game, e.g. a first-person-shooter, for the first time. Before we can start playing the game, we have to go into a “Configuration” dialog to specify which joystick to use and which of the joystick’s buttons is the fire button and so on. We propose to do the same for X3D scenes. After the user specified a mapping for the sensors, the X3D browser can save the mapping in a database (using the URL of the X3D scene as a key), so the user does not have to do this configuration each time he starts the X3D scene later on. It is also possible to define some kind of default mapping, e.g. we could specify that an SFFloat input sensor with the label “Move left/right” by default gets mapped to the x-axis of the first joystick.

Figure 4.1 shows a simplified template of a video see-through AR application written in X3D that demonstrates how to use low-level sensors. There is a SFVec3fSensor that provides the current head position from the tracking system, and a SFRotationSensor that provides the orientation. We simply route both values into a Viewpoint node. Furthermore, there is a SFImageSensor that provides video images from a camera. These images are routed into a PixelTexture node that is mapped onto an IndexedFaceSet in the background of the virtual scene.

```
DEF pointerTransform Transform {
  children DEF userBody UserBody {
    children Shape { ... }
  }
}
DEF handPos SFVec3fSensor
  { label "Hand Position" }
ROUTE handPos.value_changed
   TO pointerTransform.set_position
DEF handRot SFRotationSensor
  { label "Hand Orientation" }
ROUTE handRot.value_changed
   TO pointerTransform.set_orientation
DEF handHot SFBoolSensor
  { label "Hand Active" }
ROUTE handHot.value_changed
   TO userBody.set_hot
```

Figure 4.2: Using the UserBody.

4.1.2 High-Level Sensors

High-level sensors are sensors that are built on top of low-level sensors. They provide a more abstract interface to devices. Examples for high-level sensors are the X3D PointingDeviceSensor nodes as well as all means for navigating in the virtual scene. The PointingDeviceSensor nodes allow to interact with objects in the scene. The user can choose which object to manipulate by locating a pointing device “over” the object. In the case of 2D projections on desktop clients, “over” is defined by moving the mouse pointer over the object. But unfortunately, the X3D specification does not give any hints about how to interpret “over” in immersive, stereo projections using 3D or even 6D devices. Stiles et al. [49] describe possible solutions. In our system, we use a special node called “UserBody” that defines a 3D pointer. Its interface looks like this:

```
UserBody : Group {
  SFBool  [in,out]  hot  FALSE
}
```

It is simply a Group node that has one additional field, “hot”. The children of this group node consist of geometries that form the shape of the 3D pointer. The “hot” fields specifies whether the pointer is active (i.e. “clicked”) or not. There can be an arbitrary number of UserBodies, e.g. for multiuser applications. The pointer gets transformed in the 3D scene the usual way by putting Transform nodes in the transformation hierarchy above the UserBody and by routing position and orientation values into these transform nodes. We usually get the position and orientation values via low-level sensors from a tracking system, e.g. when using a stylus to interact with a scene, but it is possible to use arbitrary sources for these values, e.g. Script nodes. This is similar to the proposal made by Polys et al. in [**?**]. Figure 4.2 shows an example that demonstrates how to connect the UserBody to a tracking system.

To interact with PointingDeviceSensors, our systems provides three different kinds of interaction modes, “project”, “intersect” and “collision”, which have specific advantages depending on the kind of application and interaction device. The user can select one of these interaction modes by using the user interface of our system. “project” is useful for interaction devices that only provide position values (3 degrees of freedom). We shoot a ray from the Viewpoint through the center of origin of the UserBody node. The first object that gets hit by this ray is the object our pointer is currently “over”. “intersect” and “collision” are useful for interaction devices that provide position values as well as rotation values (6 degrees of freedom). When using “intersect”, we shoot a ray from the origin of the UserBody node along the negative z axis. Again, the first object that gets hit is the object we are “over”. When using “collision”, the user actually has to collide the geometry of the UserBody with another object.

4.2 Input/Output streams

This section shows you how to use an IOSensor to connect input/output streams; as example we will use two joystick-axes to control the diffuse-color of a box.

4.2.1 Introduction

IR supports various ways to get the device data in/out of your application/scene to handle different classes of applications and scenarios. For web-applications it's desirable to have device-independent setups. On the other hand the system must provide concrete and low-level access to services to handle complex and extensive application-/device-setups which are part of a fixed and controlled desktop or even immersive environments. For device-independent setups, you can just define labeled streams inside of a scene and map this streams outside (e.g. interactive or as part of your engine) to a logical devices. To access the device directly you have to specify a concrete service-type.

This section shows how to utilize IOSensor-nodes to handle both setups.

4.2.2 IOSensor basics

The IOSensor allows you to connect a local or network-device or service. (Side node: To be more precise: it abstracts a backend or namespace of the device-subsystem which is in most cases a single device) There are other X3D extensions and systems which also provide low-level device access but most of them provide one node per logical device-class. We followed a different approach: We have a single IOSensor node type. Every IOSensor node instance can be used to connect to a service/namespace of the supported device-subsystems. Interfaces and parameters of a concrete device, if any, will be mapped to dynamic fields, similar to the Script node. Therefore the node has, more or less, only two fields.

```
IOSensor : X3DSensorNode {
    SFString [] type   [auto] [joystick,video,...]
    SFString [] name   []
}
```

The type specifies the type of device/namespace/backend which should be used or can be set to 'auto'. This switches two essential modes: The implicit or explicit naming of services which leads in most cases to device independent respectively dependent setups.

4.2.3 Implicit Service definition

If your application needs IO-data-streams but would not like to specify which device or service should be used the 'type' field should be set to 'auto'. In this case, the system and runtime environment will try to map the user-given IO-slots to concrete devices automatically or by asking the user.

```
DEF camNav IOSensor {
    eventOut SFFloat speed
    eventOut SFFloat direction
}
```

4.2.4 Explicit Service definitions

If the type field is not auto it should define one of the supported device/backend/namespace types. Ths standard device-abstraction system supports more than 30 backends. All definitions can be accessed by using the 'device management' system provided in the Help-Menu.

Which devices are available depends on the system and environment. The name-field is used as an identifier in the device-subsystem. It is not used inside of your X3D-application.

4.2.4.1 Parameter of Services

Every device type provides a list of dynamic SFString fields which can be used as parameters. Most backend types provide e.g. a 'device' field which takes a number (e.g. 0,1,2) or even a description substring of the description:

```
DEF myStick IOSensor {
    type "joystick"
    device "microsoft"
}
```

If the type 'joystick' does not provide a dynamic 'device'-field you will get a warning and a list of valid fields in the application log. You can also lookup valid backend-fields using the online interface which is available through the help menu. The parameter-fields only depend on the device type but not on a single service instance. For example all joystick-backends provide a 'device'-field but every joystick instance provides a different number of buttons and axes.

4.2.4.2 IO Slots of Services

The IOSensors use user-provided dynamic in/out-slots to connect to these devices: e.g. the x/y axis of a joystick could be accessed in the following way

```
DEF myStick IOSensor {
    type "joystick"
    eventOut SFFloat *x*axis*
    eventOut SFFloat *y*axis*
}
DEF myScript Script {
    eventIn SFFloat speed
}
ROUTE myStick.*x*axis TO myScript.speed
```

These slots can be used together with ROUTES to process and transmit the events from or to the device. The names of the slots are used to map the device interfaces to the node interface. To increase the usability file-system like wild-cards (* and ?) are supported. Again: Use the backend-interfaces which can be accessed in your Help-menu to check the provided slots.

The last example shows how to map the image data of a local camera onto a box.

```
DEF video IOSensor {
  type "video"
  eventOut SFImage frame*
}

Shape {
  appearance Appearance {
    texture DEF tex PixelTexture { }
  }
  geometry Box { }
}

ROUTE video.frame* TO tex.image
```

4.2.5 Conclusion

This section only shows how to get your data in. It does not show how to use the data for interaction and navigation purpose. If you would like to use the data together with high level PointingSensors look at the section Immersive PointingSensor Interaction. For a section on abstract 3D-Navigators please have a look at the Navigator section .

Files:

- joystickToColorTutorial.x3d
- videoCameraTutorial.x3d

4.3 Connecting Devices

This section shows you how to connect a device to the framework and how to find all in/out slots of the device interactively. This is necessary if you would like to write a application which automatically connects a device using e.g. a IOSensor (See Input/Output streams section). It only shows how to start a IO-Node and how to find the coresponding input/output slots. You still need to connect the streams to the application (e.g. using a IOSensor) and some code to process the input data (e.g. SteeringNavigator); The section uses a simple joystick but the steps are in general the same for every device

4.3.1 Introduction

The framework does not classify a specific interface for a device-type but dynamically creates input/output slots for every device instance. This has the advantage that we can connect any kind of device providing any kind of data channels. But the design has also one major drawback: You always have to test a specific physical device to see what kind of in/out slots it provides. There is for e.g. only a single "Joystick"-Type which supports any number of buttons and axis. Therefore the same code can be used to connect a 12-Button/6-axis joypad or an 5-Button/0-axis apple-remote (which is also registered as joystick in OSX).

You can connect the in/out slots interactively which gives you the most flexibility e.g. using the Web-Interface. However, if you really know what you are doing and would like to write a application which uses for e.g. an IOSensor to start and connect a specific device you have to know what kind of slots there are. This first part of the section shows you to find those slots. The second part shows how to use this information with an IOSensor.

4.3.2 Finding your slots

4.3.2.1 Connect your device to the local machine

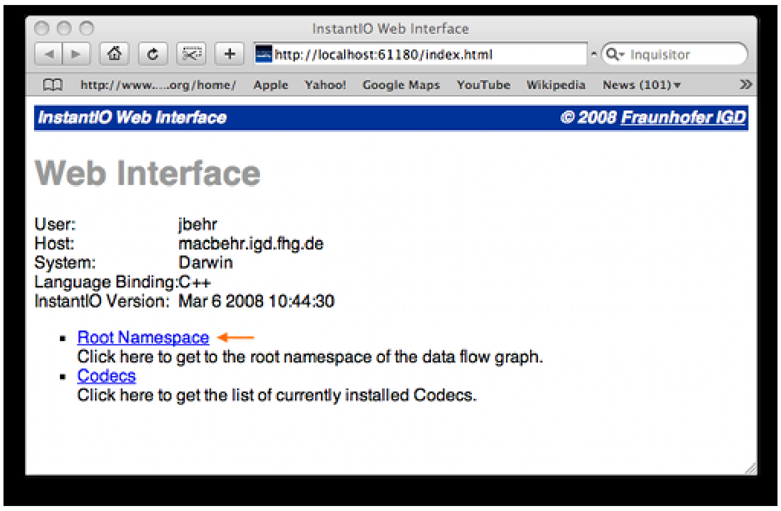
Connect the device to the same machine where you would like to run the InstantPlayer system (You can use devices remotely easily using the InstantIO-Server but this is not the topic of this section). Here I connect a logitech joypad to my notebook.

Figure 4.3: plug the device in

4.3.2.2 Start the correct Device-handler

Start the InstantPlayer and open the Web-Interface of the Device-Managment system by clicking on “Help->Web Interface Device Managment” in the menu bar entry. This opens your preferred web-browser showing the Web Interface of the device managment system called InstantIO.

This page may look like a static page but this is actual the dynamic User-Interface for the device managment system. The Web interface allows you to control the system remotely which is very handy for Immersive or mobile-AR applications.



InstantIO Web Interface © 2008 Fraunhofer IGD

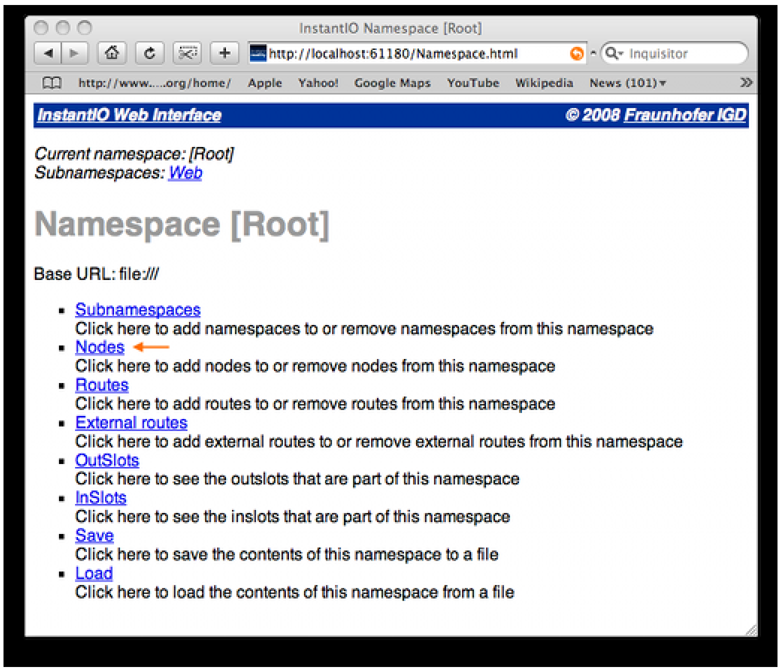
Web Interface

User: jbehr
Host: macbehr.igd.fhg.de
System: Darwin
Language Binding:C++
InstantIO Version: Mar 6 2008 10:44:30

- Root Namespace
  Click here to get to the root namespace of the data flow graph.
- Codecs
  Click here to get the list of currently installed Codecs.

Figure 4.4: start the interface and go to the root namespace

Go to RootNamespace to open the root Namespace page.



InstantIO Web Interface © 2008 Fraunhofer IGD

Current namespace: [Root]
Subnamespaces: Web

Namespace [Root]

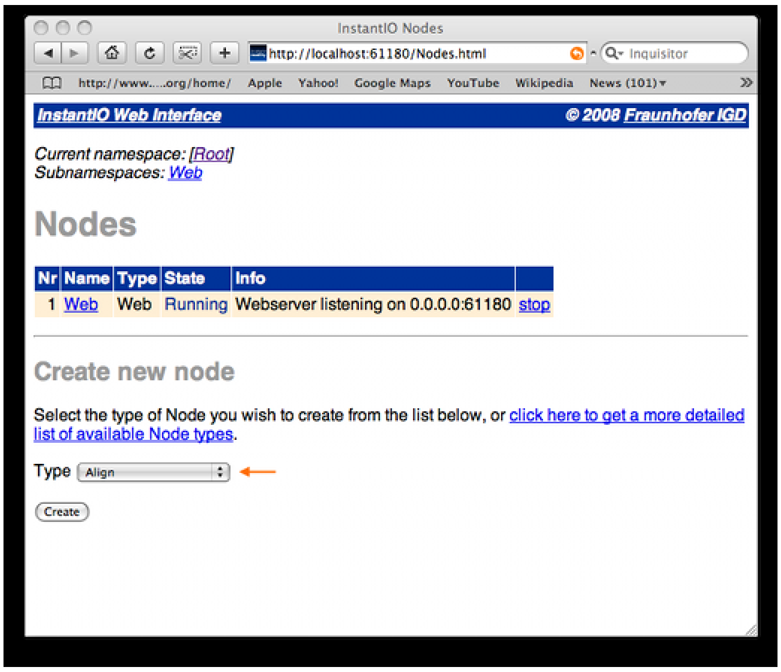
Base URL: file:///

- Subnamespaces
  Click here to add namespaces to or remove namespaces from this namespace
- Nodes
  Click here to add nodes to or remove nodes from this namespace
- Routes
  Click here to add routes to or remove routes from this namespace
- External routes
  Click here to add external routes to or remove external routes from this namespace
- OutSlots
  Click here to see the outslots that are part of this namespace
- InSlots
  Click here to see the inslots that are part of this namespace
- Save
  Click here to save the contents of this namespace to a file
- Load
  Click here to load the contents of this namespace from a file

Figure 4.5: go to nodes page to get a list of active nodes

Click further on Nodes to open the page to add and remove device handler to the current namespace.

In the lower side of the page you find the “Create new node” section which allows you to create a new device handler. Select the correct type for the connected device. In our case we select “Joystick”


InstantIO Nodes

http://localhost:61180/Nodes.html

InstantIO Web Interface © 2008 Fraunhofer IGD

Current namespace: [Root]
Subnamespaces: Web

Nodes

Nr	Name	Type	State	Info	
1	Web	Web	Running	Webserver listening on 0.0.0.0:61180	stop

Create new node

Select the type of Node you wish to create from the list below, or click here to get a more detailed list of available Node types.

Type Align

Create

Figure 4.6: select a type to create a new handler

and push the “Create” Button.

The following page allows you to set some parameter for the device. In our case it is the name and the device identifier. More information can be found on the specific page

This should start the device handler and bring you back to the list of active devices.

There is now a “Operating Logitech Dual Action” which is sleeping since no Sensor is connected. Click on the Root link at the top of the page to get back to the namespace page.

4.3.2.3 Get a list of all out-slots

Now since we have an active device handler for the joystick we can see what kind of dynamic slots are available.

Now click on OutSlots to get a list of all available OutSlots for the new device

This list shows 12 Buttons and 6 Axis which the joystick provides. These are all slots of the physical device.

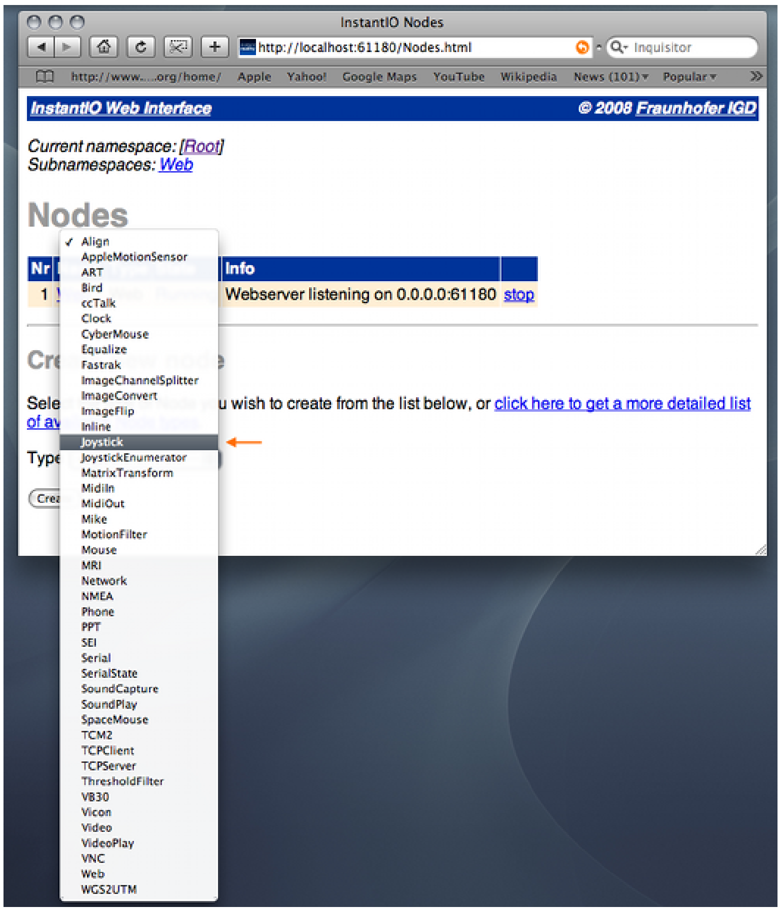
4.3.3 Use the information to start an IOSensor

The IOSensor allows you to start a InstantIO type directly from your Scene. Use dynamic outputOnly slots, the same way you use them in Script nodes, to connect the slots. The Name of the slots have to be the same as in the above list. There is one aspect you have to keep in mind. The InstantIO slots sometimes contain spaces and you have to replace those with wildcards (* and ?) to create a single token.

```
DEF myStick IOSensor {
        type "joystick"
        outputOnly SFFloat *Hat*x*axis*
        outputOnly SFFloat *Hat*y*axis*
}
```

If you use the xml encoding you can use the full name including any kind of spaces:

```
<IOSensor DEF='myStick' type='joystick'>
   <field accessType='outputOnly' name='Hatswitch X-Axis' type='SFFloat'/>
   <field accessType='outputOnly' name='Hatswitch Y-Axis' type='SFFloat'/>
 </IOSensor>
```



Current namespace: [Root]
Subnamespaces: Web

Nodes

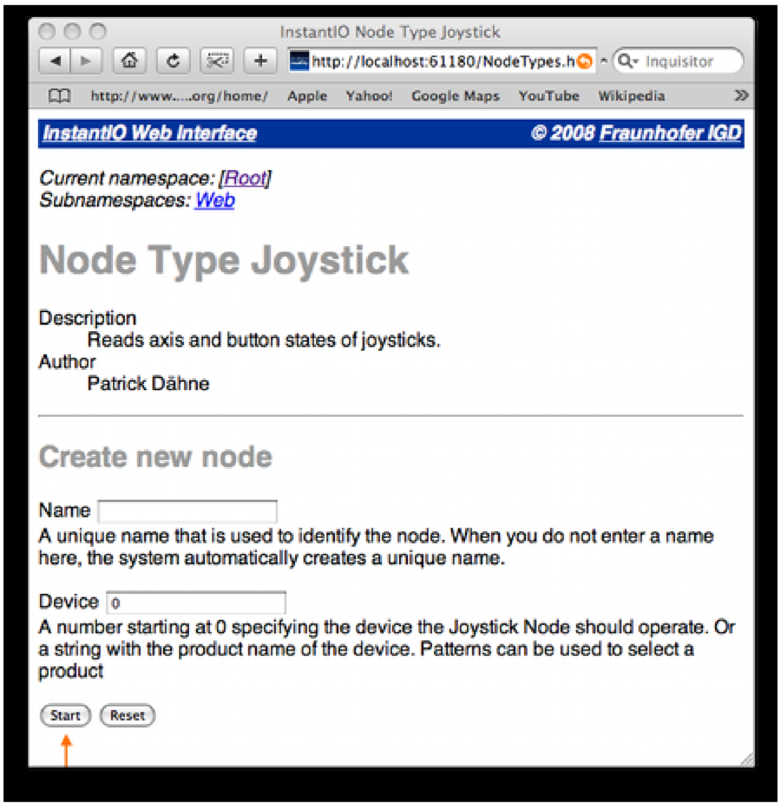
Nr		Info	
1		Webserver listening on 0.0.0.0:61180	stop

Cre

Sele u wish to create from the list below, or click here to get a more detailed list of av

Type

Figure 4.7: pick the right type for you device


InstantIO Node Type Joystick
http://localhost:61180/NodeTypes.h
Inquisitor
http://www....org/home/ Apple Yahoo! Google Maps YouTube Wikipedia
Start
Reset

InstantIO Web Interface © 2008 Fraunhofer IGD

Current namespace: [Root]
Subnamespaces: Web

Node Type Joystick

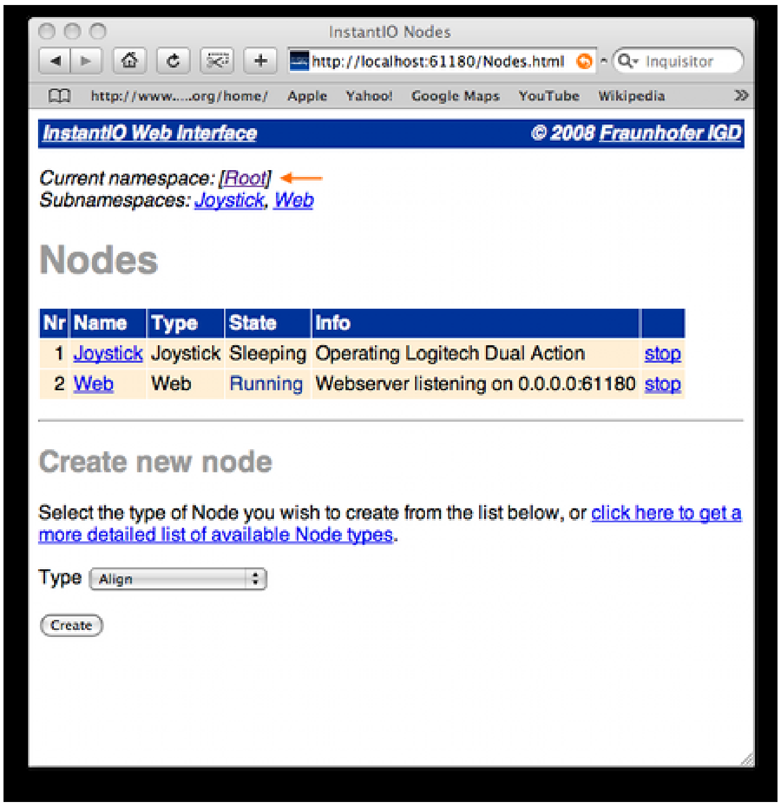
Description
Reads axis and button states of joysticks.
Author
Patrick Dähne

Create new node

Name
A unique name that is used to identify the node. When you do not enter a name here, the system automatically creates a unique name.

Device 0
A number starting at 0 specifying the device the Joystick Node should operate. Or a string with the product name of the device. Patterns can be used to select a product

Figure 4.8: fill in the parameter the start the type


InstantIO Nodes
http://localhost:61180/Nodes.html
Inquisitor
http://www....org/home/ Apple Yahoo! Google Maps YouTube Wikipedia
Create

InstantIO Web Interface © 2008 Fraunhofer IGD

Current namespace: [Root]
Subnamespaces: Joystick, Web

Nodes

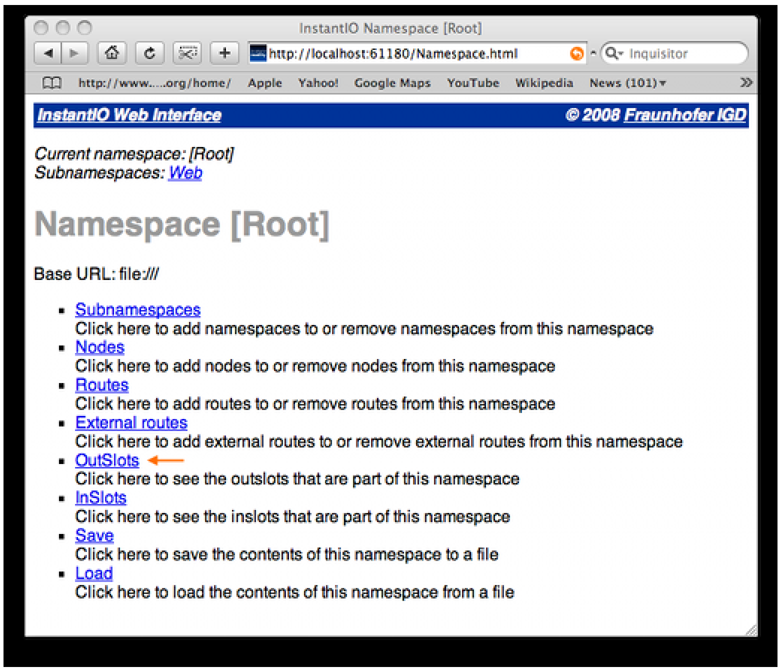
Nr	Name	Type	State	Info	
1	Joystick	Joystick	Sleeping	Operating Logitech Dual Action	stop
2	Web	Web	Running	Webserver listening on 0.0.0.0:61180	stop

Create new node

Select the type of Node you wish to create from the list below, or click here to get a more detailed list of available Node types.

Type Align

Figure 4.9: fill in the parameter the start the type


InstantIO Namespace [Root]
http://localhost:61180/Namespace.html
Inquisitor
http://www.....org/home/ Apple Yahoo! Google Maps YouTube Wikipedia News (101)

InstantIO Web Interface © 2008 Fraunhofer IGD

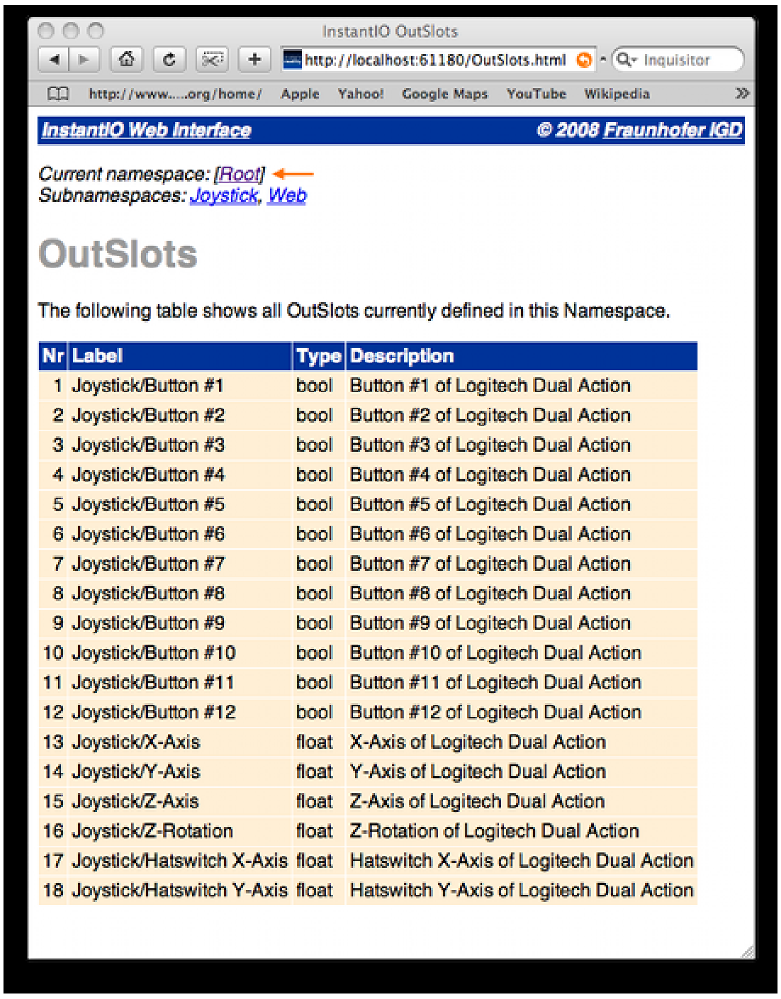
Current namespace: [Root]
Subnamespaces: Web

Namespace [Root]

Base URL: file:///

- Subnamespaces
  Click here to add namespaces to or remove namespaces from this namespace
- Nodes
  Click here to add nodes to or remove nodes from this namespace
- Routes
  Click here to add routes to or remove routes from this namespace
- External routes
  Click here to add external routes to or remove external routes from this namespace
- OutSlots
  Click here to see the outslots that are part of this namespace
- InSlots
  Click here to see the inslots that are part of this namespace
- Save
  Click here to save the contents of this namespace to a file
- Load
  Click here to load the contents of this namespace from a file

Figure 4.10: Scene


InstantIO OutSlots
http://localhost:61180/OutSlots.html
Inquisitor
http://www.....org/home/ Apple Yahoo! Google Maps YouTube Wikipedia

InstantIO Web Interface © 2008 Fraunhofer IGD

Current namespace: [Root]
Subnamespaces: Joystick, Web

OutSlots

The following table shows all OutSlots currently defined in this Namespace.

Nr	Label	Type	Description
1	Joystick/Button #1	bool	Button #1 of Logitech Dual Action
2	Joystick/Button #2	bool	Button #2 of Logitech Dual Action
3	Joystick/Button #3	bool	Button #3 of Logitech Dual Action
4	Joystick/Button #4	bool	Button #4 of Logitech Dual Action
5	Joystick/Button #5	bool	Button #5 of Logitech Dual Action
6	Joystick/Button #6	bool	Button #6 of Logitech Dual Action
7	Joystick/Button #7	bool	Button #7 of Logitech Dual Action
8	Joystick/Button #8	bool	Button #8 of Logitech Dual Action
9	Joystick/Button #9	bool	Button #9 of Logitech Dual Action
10	Joystick/Button #10	bool	Button #10 of Logitech Dual Action
11	Joystick/Button #11	bool	Button #11 of Logitech Dual Action
12	Joystick/Button #12	bool	Button #12 of Logitech Dual Action
13	Joystick/X-Axis	float	X-Axis of Logitech Dual Action
14	Joystick/Y-Axis	float	Y-Axis of Logitech Dual Action
15	Joystick/Z-Axis	float	Z-Axis of Logitech Dual Action
16	Joystick/Z-Rotation	float	Z-Rotation of Logitech Dual Action
17	Joystick/Hatswitch X-Axis	float	Hatswitch X-Axis of Logitech Dual Action
18	Joystick/Hatswitch Y-Axis	float	Hatswitch Y-Axis of Logitech Dual Action

Figure 4.11: got to OutSlots to get a list

You can connect those slots to any kind of other node (e.g Navigator or Script) to process the incoming values. Look at the next parts in this section to get further information. Look at the Input-Output stream section to get further information about the IOSensor.

4.3.4 Conclusion

This section shows how to connect external devices and how to find their provided slots. This is the information you need to connect the service to the framework. You can do it interatively or as part of your scene useing an IOSensor or similiar services.

4.4 Space-Mouse/Navigator/Pilot Device

This tutorial demonstrates how to connect a 3Dconnexion (former LogiCad3D) SpaceMouse, Space-Navigator, SpacePilot or compatible device to Instant Player.

4.4.1 Introduction

This tutorial demonstrates how to connect a 3Dconnexion (former LogiCad3D) SpaceMouse, Space-Navigator, SpacePilot or compatible device to Instant Player. There are three different approaches, depending on the type of device and the operating system you are using.

In general all the devices provide 6 degrees of input. Three axes for translation and three axes for rotation. In addition the devices support a varying number of buttons, ranging from 2 to more than 20. The main difference is the type of connection.

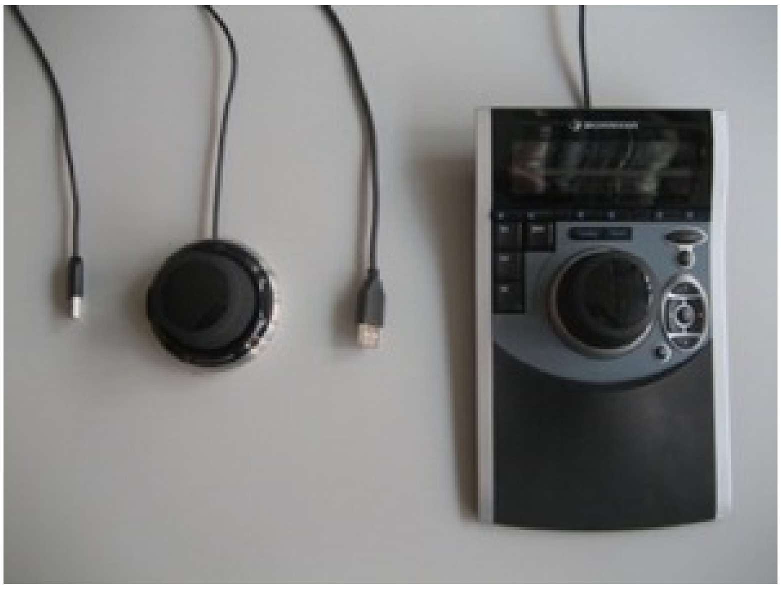


Figure 4.12: SpaceNavigator and SpacePilot using an USB-Connector

New devices, e.g. SpaceNavigator and SpacePilot, use a USB connector.

Older devices, like the classic DLR SpaceMouse are usually connected via a serial slot. These are becoming rare, so in most cases a USB slot will be fine.

4.4.2 Start the correct device Handler

The correct InstantIO Handler depends on the physical device and opperating system. Look for the “Input/Output streams” tutorials to get more background information about the IOSensor.

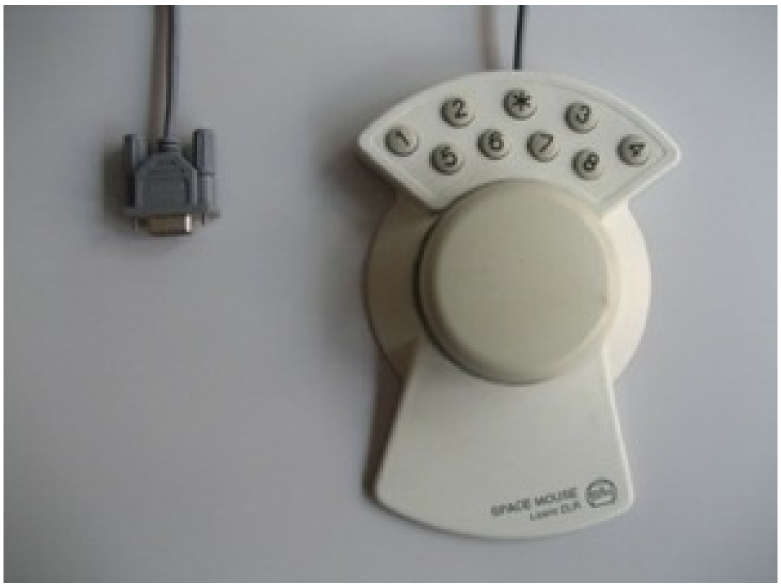


Figure 4.13: SpaceMouse using an Serial-Connector

4.4.2.1 USB Device on Windows

The SpaceNavigator backend is the recommended way to connect SpaceNavigator and SpacePilot devices on Windows. It does not exist on Mac OS X or Linux, and it does not work out of the box for classic SpaceMice connected to the serial port.

To get the backend working, first install the appropriate driver from the 3Dconnexion web site (it is actually the same driver for all USB devices, “3DxSoftware 3.x”). Then, integrate an IOSensor node into the scene whose type is “SpaceNavigator”:

```
DEF ios IOSensor {
  type "SpaceNavigator"
  eventOut SFFloat X?translation
  eventOut SFFloat Y?translation
  eventOut SFFloat Z?translation
  eventOut SFRotation Rotation
  eventOut SFBool Button??1
  eventOut SFBool Button??2
}
```

The SpaceNavigator backend has three float outslots that provide values between -1 and 1 for the x, y and z translation of the cap, one rotation outslot for the rotation of the cap, and two boolean outslots for the buttons “1” and “2” of the SpaceNavigator (we currently do not support the other buttons available on other 3Dconnexion devices like the SpacePilot).

4.4.2.2 USB Device on OSX

The Joystick backend is the recommended way to connect SpaceNavigator and SpacePilot devices on Mac OS X. It does not work on Windows or Linux, and it does not work for classic SpaceMice connected to the serial port.

To get the backend working, do not install any drivers - just integrate an IOSensor node into the scene whose type is “Joystick”. In the “device” SFString field, you can either specify the name of the device or its index (0 is the first joystick device in the system, 1 the second, and so on):

```
DEF ios IOSensor {
  type "Joystick"
  device "0"
  eventOut SFFloat X-Axis
```

```
    eventOut SFFloat Y-Axis
    eventOut SFFloat Z-Axis
    eventOut SFFloat X-Rotation
    eventOut SFFloat Y-Rotation
    eventOut SFFloat Z-Rotation
    eventOut SFBool Button??1
    eventOut SFBool Button??2
  }
```

The Joystick backend has three float outslots that provide values between 0 and 1 for the x, y and z translation of the cap, three float outslots that provide values between 0 and 1 for the x, y and z rotation of the cap, and boolean outslots for each button of the device.

4.4.2.3 Serial-Device on all Systems

The SpaceMouse backend is the recommended way to connect classic (serial) SpaceMouse devices on all operating systems.

To get the backend working, do not install any drivers - just integrate an IOSensor node into the scene whose type is "SpaceMouse". In the "device" SFString field, you have to specify the serial port the SpaceMouse is connected to (0 is COM1, 1 is COM2, and so on).

```
  DEF ios IOSensor {
    type "SpaceMouse"
    device "0"
    eventOut SFFloat X?translation
    eventOut SFFloat Y?translation
    eventOut SFFloat Z?translation
    eventOut SFFloat X?rotation
    eventOut SFFloat Y?rotation
    eventOut SFFloat Z?rotation
    eventOut SFBool Button?1
    eventOut SFBool Button?2
    eventOut SFBool Button?3
    eventOut SFBool Button?4
    eventOut SFBool Button?5
    eventOut SFBool Button?6
    eventOut SFBool Button?7
    eventOut SFBool Button?8
  }
```

The SpaceMouse backend has three float outslots that provide values between 0 and 1 for the x, y and z translation of the cap, three float outslots that provide values between 0 and 1 for the x, y and z rotation of the cap, and eight boolean outslots for the buttons "1" - "8" of the device. It is currently not possible to access the "*" button of the device via the IOSensor node.

Side Node: It is also possible to get older (serial) SpaceMouse devices working with the SpaceNavigator backend on Windows, but that involves a little bit of hacking. You have to install the appropriate driver from the 3Dconnexion web site ("3DxSoftware 2.x"). Additionally you have to get and register a library (TDxInput.dll). This library only comes with the drivers for newer (USB) devices ("3DxSoftware 3.x"). So you have to perform the following steps to get older (serial) devices working:

- Get and install the driver for newer (USB) devices ("3DxSoftware 3.x"). As I already said, this driver is the same for all USB devices, so it does not matter whether you choose the driver for the SpaceNavigator, the SpacePilot or any other USB device.
- Locate the library "TDxInput.dll". On an english version of Windows, when you installed the driver into the default location, the location of that library is "C:Program Files3Dconnexion3Dconnexion 3DxSoftware3DxWarewin32". Copy that library into a safe location.
- Uninstall the 3DxSoftware 3.x driver.
- Get and install the driver for older (serial) devices ("3DxSoftware 2.x").

- Register the library “TDxInput.dll”. To to that, you have to log in as an administrator, open the command line, go into the directory that contains the library, and enter the following command: “regsvr32 TDxInput.dll”. Do not move the library to another location or remove it from the hard disk - it is registered at that specific location.

4.4.3 Controlling the Application

The IOSensor nodes give you the raw datastreams of the devices. You, as application developer, are totally free to use it to change various application states. You can e.g. navigate the camera, transform BodyPart nodes to trigger PointSensors or change the color of an object according to the current rotation state.

You can use Scripts to code this behavior or use helper Nodes like an SteeringNavigator. Check the “Navigator” and “Immersive PointingSensor Interaction” tutorial for more details.

Attached to this tutorial you find a simple example which shows most usual case. Using a Space-Navigator/SpacePilot on Windows to control a Navigator while walking/flying in a Virtual Environment

Files:

- space-nav.x3d

4.5 Navigator

This sections shows how to use 2D and 3D navigators together with device inputs to move the user camera.

4.5.1 Introduction

For desktop applications navigation is simply accomplished by using the mouse. Internally a so-called Navigator2D node, which is part of the engine, especially the Viewarea node, is used to navigate with a mouse through the 3D scene. Thus it has three input fields, “mousePress”, “mouseRelease”, and “mouseMove”. Actually they were designed for reacting to mouse events, but as other devices may produce similiar events, for the purpose of generality those events may be routed to the 2D Navigator as well. But generally the user doesn’t need to worry about that.

This is different for the 3D Navigators, which were especially developed for joysticks or VR-devices. They also inherit from the abstract Navigator base node, but for convenience they are part of the scene and therefore have to be suitably parameterized, which will be explained in later sections.

4.5.2 Getting the data - the IOSensor

If you want to navigate or interact with your scenes using a joystick, spacemouse or a similiar external device you first need an IOSensor in your scene, for retrieving the input values. Below is an example for addressing a joystick. Usage of e.g. a spacemouse would be quite similiar, with the ’type’ field set to type “spacemouse”.

```
DEF ios IOSensor {
    type "joyStick"
    eventOut SFFloat *x*axis
    eventOut SFFloat *z*rot*
    eventOut SFBool *button*8
}
```

4.5.3 Moving around - the Navigator3D

Now that you have the values of your device, there are basically two options for navigating. On the one hand you can route the translational and rotational values to a Script node, calculate the corresponding transformations and route the results to your Viewpoint. Because this might be quite cumbersome on the other hand you can alternatively use a Navigator3D node. Currently there are three types: the PointingNavigator , the SpaceNavigator , and the SteerNavigator .

- The PointingNavigator is especially useful for fully immersive navigation in combination with a Pen. Via point and “click” you can fly to the chosen location, which is conceptually similiar to the ’lookat’ navigation type. The calculated movement is generated from relative movements of the device.
- The SpaceNavigator allows navigation regarding all six degrees of freedom by directly manipulating the camera position in 3D space, but therefore usually a lot of practice is needed.
- The SteerNavigator tries to alleviate this by providing a simpler interface for walk-through and fly modes with devices like joystick and spacemouse. In the following the latter navigation type will be further explained exemplarily.

After having outlined what type of navigators exist, it will now be explained, how they are used.

One possibility is to instantiate a navigator as a child of a Viewpoint , which is shown in the following code fragment. This has the great advantage that the navigator is automatically connected to the currently active Viewpoint.

```
Viewpoint {
    position 45.15 0.42 5.11813
    orientation -0.21 0.97 0.0644 0.59193
    navigator [
        DEF nav SteerNavigator {
            inputRange [0 1]
            rotationSpeed -0.2 -0.2 -0.2
            translationSpeed 10 10 10
        }
    ]
}

ROUTE ios.*x*axis TO nav.set_yRotation
ROUTE ios.*z*rot* TO nav.set_zTranslation
```

As can be seen in the next code fragment, despite fields for the type of navigation etc., the NavigationInfo also contains a MFNode field “navigator” for holding the 3D navigator, which will be called for the currently bound ViewBindable node.

```
NavigationInfo {
    type "walk"
    navigator [
        DEF nav SteerNavigator {
            inputRange [0 1]
            rotationSpeed -0.2 -0.2 -0.2
            translationSpeed 10 10 10
        }
    ]
}

ROUTE ios.*x*axis TO nav.set_yRotation
ROUTE ios.*z*rot* TO nav.set_zTranslation
```

Now there remains one question. How do the navigators update their internal state? The SteerNavigator for instance has six input fields for [x|y|z]Rotation as well as for [x|y|z]Translation, which define a rotation around the corresponding axis or a translation in the appropriate direction respectively. For updating camera movement, you only need to route the corresponding values from your device sensor node to the navigator node as shown above.

Furthermore there exist some interesting fields for fine-tuning your navigator. The “inputRange” field specifies the input value range e.g. [-1;1] or [0;1]. It is possible to specify one value for all inputs or a single range for all 6 input values. The “rotationSpeed” field defines the rotations per second for each axis; the values can also be negative for inverting the direction of rotation. The “translationSpeed” field defines the speed of the translation in meters per second for each axis.

In order to avoid drift when not interacting with the input device the SteerNavigator has two SFVec3f fields for defining the values of zero deflection for each axis (meaning the control sticks, after initially having moved them already, are now at rest): “zeroDeflectionTrans” and “zeroDeflectionRot”. Last but not least the SteerNavigator node has an SFBool eventIn slot called “updateRotationCenter”. If this slot is triggered a point along the viewing ray, usually the point of intersection with the scene geometry, is set as the new center of rotation (default is the origin), which is used in examine mode as the center point around which the viewpoint is rotated.

The example file shows a simple walk-through world using an IOSensor for joystick movement and a SteerNavigator for showing the previously explained fields in action.

Files:

- walk_through.wrl

4.6 Immersive PointingSensor Interaction

This sections shows how to use a UserBody together with immersive interaction devices in order to trigger pointing sensors.

4.6.1 Desktop based interaction

For desktop applications object manipulation is simply accomplished by using the mouse or similiar devices. The X3D PointingDeviceSensor nodes therefore allow to interact with objects in the scene. The user can choose which object to manipulate by locating the mouse pointer “over” the object. Here the interaction concepts directly follow the way they are described in the pointing device sensor component of the X3D specification.

Hint: This concept can easily be generalized for any screen-space based input data. Internally the so-called Navigator2D node, which is part of the engine, is used to handle navigation and interaction with a mouse within a 3D scene. But other devices like e.g. optical tracking may produce similiar events, which can also be used. Because those concepts were already explained in the context of 2D/3D navigation, the interested reader may refer to the corresponding navigation section .

4.6.2 Fully immersive interaction

Within X3D a pointing-device sensor is activated when the user locates the pointing device “over” geometry that is influenced by that specific pointing-device sensor. For desktop clients with a 2D mouse this is just defined by the mouse pointer. In immersive environments (e.g. a CAVE using a 6DOF interaction device) it is not so straightforward how “over” should be understood.

Therefore one additional node to generalize the immersive implementation is provided. The UserBody derived from the Group node defines a sub-graph as so-called user body. The UserBody has only one extra SFBool field “hot”. The hot-field is analogous to a mouse button for 2D interaction and corresponds to the “button pressed” state.

If the UserBody is instantiated as child of a Transform node it can be transformed by external interaction devices like a spacemouse or a pen (whose values can be accessed by means of the IOSensor node), and can be used for direct visual feedback of pointing tasks as well as for colliding with real scene geometry, equivalent to a 3D mouse cursor.

The type of interaction is set in the NavigationInfo node. Currently the following interaction types are possible:

- `none` - no interaction
- `ray` - handles ray selection in 3D; the ray origin is the position of the user body, and the ray points into the negative z direction (typically an array, by grouping a Cone and a Cylinder, is used for representing the proxy geometry, in this case don’t forget to add an additional rotation of ’1 0 0 -1.5707963’ for correct adjustment to the parent Transform)
- `nearest` - also ray based, but uses the nearest sensor, because sometimes it might be quite difficult to really hit an object by means of a ray intersect
- `projection` - like ’ray’ this type also handles ray selection in 3D, but this time the ray points from the camera through the origin of the user body’s coordinate system, what is especially useful for desktop applications. Be careful not to mix up the origin (which might not be visible) with the

real position of your object. Hint: When using the Viewspace a Geometry2D node works best as user body.

- `collision` - here the notion of being “over” is modelled by means of a collision of the user body geometry with the sensor geometry

With the help of the following code fragment (the complete version can be found in the example) a typical usage scenary will finally be exemplarily discussed.

```
DEF script Script {
        eventIn SFTime update
        eventIn SFFloat set_xRotation
        eventIn SFFloat set_yRotation
        eventIn SFFloat set_zRotation
        eventIn SFFloat set_xTranslation
        eventIn SFFloat set_yTranslation
        eventIn SFFloat set_zTranslation
        eventOut SFRotation rotation_changed
        eventOut SFVec3f translation_changed
        url "javascript: ..."
}

DEF timeSensor TimeSensor { loop TRUE }
ROUTE timeSensor.time TO script.update

DEF ios IOSensor {
        type "spacemouse"
        eventOut SFFloat X*Rotation
        eventOut SFFloat Y*Rotation
        eventOut SFFloat Z*Rotation
        eventOut SFFloat X*Translation
        eventOut SFFloat Y*Translation
        eventOut SFFloat Z*Translation
        eventOut SFBool Button*
}

DEF navInfo NavigationInfo {
        interactionType "ray"
        sceneScale 0.01
}

Viewspace {
        scaleToScene TRUE
        children [
                DEF userBodyTrans Transform {
                        children [
                                DEF userBody UserBody {
                                        ...
                                }
                        ]
                }
        ]
}

ROUTE ios.X*Rotation TO script.set_xRotation
ROUTE ios.Y*Rotation TO script.set_yRotation
ROUTE ios.Z*Rotation TO script.set_zRotation
ROUTE ios.X*Translation TO script.set_xTranslation
ROUTE ios.Y*Translation TO script.set_yTranslation
ROUTE ios.Z*Translation TO script.set_zTranslation
ROUTE ios.Button* TO userBody.hot
ROUTE script.rotation_changed TO userBodyTrans.set_rotation
ROUTE script.translation_changed TO userBodyTrans.set_translation
```

Because a UserBody can only have an effect when being moved around, you first have to update its position and orientation to determine which 3D objects are to be hit. This can be done with the help of an IOSensor for receiving the input data of your desired interaction device. In this example a spacemouse was chosen.

Because a spacemouse has six SFFloat eventOut slots, three for translation along the x, y, and z axis, and three for rotation about these axes, the final translation (of type SFVec3f) and rotation (of type SFRotation) have to be assembled in a script. After that the results are routed to the parent transform of the UserBody node, which contains the pointer geometry.

In this example the user body is also a child of a Viewspace node. This is due to the fact, that usually the pointer geometry is not really considered as being part of the scene but rather a tool for interacting in immersive environments.

In this context two fields are quite important: If `scaleToScene` is true, the Viewspace is scaled to the same size as defined in `sceneScale` of the NavigationInfo. This is very useful in case the scene wasn't modelled in meters; hence if the scene was modelled e.g. in centimeters, the sceneScale field should be set to 0.01.

Warning

Please note, that currently only the first UserBody can activate pointing device sensors in ray, nearest and collision mode; whereas the projection mode may not work in multi-viewport/ highly immersive environments.

Files:

- Ray intersect
- Projective intersect
- Projective intersect with HUD

4.7 Vision Marker Tracking

This section shows you how to use instant reality's vision module for marker tracking.

4.7.1 Introduction

instant *vision* is a set of visual tracking systems starting with simple marker tracking going to markerless tracking like line trackers and KLT. The true power of the system lies in the ability to combine several such tracking procedures, for instance using a line tracker for initialisation with an absolute pose and KLT for frame to frame tracking.

In this example we will focuson a simple marker tracking example using the VisionLib backend.

4.7.2 IOSensor

The marker tracking is loaded like any other HID device via an IOSensor. These are instant *vision's* fields:

- *VideoSourceImage (SFImage)* : Camera image
- *TrackedObjectCamera_ModelView (SFMatrix)* : Camera's modelview matrix
- *TrackedObjectCamera_Projection (SFMatrix)* : Camera' projection matrix
- *TrackedObjectCamera_Position (SFVec3f)* : Camera's position
- *TrackedObjectCamera_Orientation (SFRotation)* : Camera's orientation

```
<IOSensor DEF='VisionLib' type='VisionLib' configFile='visionlib.pm'>
    <field accessType='outputOnly' name='VideoSourceImage' type='SFImage'/>
    <field accessType='outputOnly' name='TrackedObjectCamera_ModelView' type='SFMatrix4f'/>
    <field accessType='outputOnly' name='TrackedObjectCamera_Projection' type='SFMatrix4f'/>
    <field accessType='outputOnly' name='TrackedObjectCamera_Position' type='SFVec3f'/>
    <field accessType='outputOnly' name='TrackedObjectCamera_Orientation' type='SFRotation'/>
</IOSensor>
```

In order to use the camera's correct modelview and projection in the scene we are using a Viewfrustrum instead of a standard Viewpoint. By routing the IOSensor's *TrackedObjectCamera_ModelView* and *TrackedObjectCamera_Projection* to the ViewFrustrum's *modelview* and *projection* the virtual camera matches the real camera's position and orientation relative to the marker.

```
<Viewfrustum DEF='vf' />

<ROUTE fromNode='VisionLib' fromField='TrackedObjectCamera_ModelView' toNode='vf' toField='mo
<ROUTE fromNode='VisionLib' fromField='TrackedObjectCamera_Projection' toNode='vf' toField='p
```

With the camera's image in the background we are creating an Augmented Reality scenario.

```
<PolygonBackground>
    <Appearance positions='0 0, 1 0, 1 1, 0 1' >
            <TextureTransform rotation='0' scale='1 -1'/>
            <PixelTexture2D DEF='tex' autoScale='false'/>
    </Appearance>
</PolygonBackground>

<ROUTE fromNode='VisionLib' fromField='VideoSourceImage' toNode='tex' toField='image'/>
```

This example works with standard webcams with vga resolution. An example how to manipulate instant *vision's* configuration file and to create and use different markers will follow.

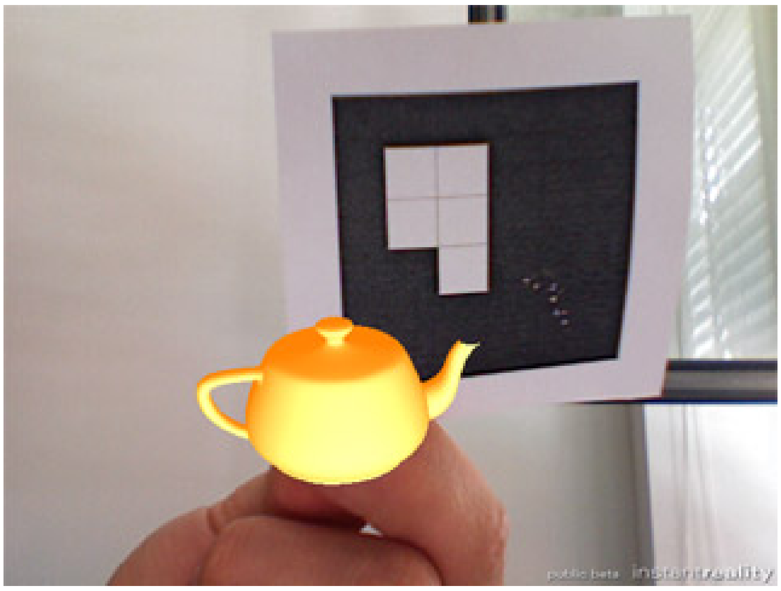


Figure 4.14: visionlib.jpg

Files:

- visionlib.x3d (Example)
- visionlib.pm (Configuration File)
- visionlib.pdf (Marker)

4.8 Vision Tracking Device

This tutorial shows you how to vision based tracking; As example we will create a marker tracker.

4.8.1 Introduction

4.8.2 Tracking in General

World The desciption of the world how the tracking system sees it. It contains one ore more TrackedObjects.

TrackedObject Describes the objects to be tracked, thus this node contains all information about an object which should be tracked

Marker One way to track things is the use of a marker. To describe a TrackedObject with a marker this node is added to the TrackedObject

Camera A camera is used in every vision tracking system. The node contains one Intrinsic and one Extrinsic data node.

ExtrinsicData Part of the camera description, contains the parameters descibing the position and orientation of the camera in the world.

IntrinsicData Second part of the camera description, describes the internal parameters of a camera like resolution, focal length or distortion.

ActionPipe The execution units (Actions) in InstantVision are arranged in an execution pipe which is called an ActionPipe.

DataSet All data items used in InstantVision are placed in the DataSet and have a key(name) to refere to them.

4.8.3 The Example

Two files will be needed to setup an InstantReality scene with a vision tracking device. The first is a VisionLib configuration file, which describes the tracking setup, the second is a scene file for IR.

The VisionLib config (visionlib.pm). All images and cameras used in the VisionLib config are exported to InstantPlayer, so you can use the images as textures or backgrounds and the cameras as transformations for Viewpoint, Viewfrustum or ComponentTransform. The names of the Images are the same as in the VisionLib config, the cameras are split into 4 names where the first part names the TrackedObject from which the camera is derived and the postfix names the output type. In the Example these names are “TrackedObjectCamera_ModelView”, “TrackedObjectCamera_Projection”, “TrackedObjectCamera_Position”, “TrackedObjectCamera_Orientation”. The camera here is derived from World.TrackedObject which gave the name.

Tracking multiple markers can be achieved by duplicating the TrackedObject sections and give the TrackedObjects distinctive keys and marker codes. As described above, you will get the camera (inverted object) transformations named like the TrackedObject key + (e.g.) “Camera_ModelView”.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<VisionLib2 Version="2.0">

  <Plugins size="0">
  </Plugins>

  <ActionPipe category="Action" name="AbstractApplication-AP">
    <VideoSourceAction category="Action" name="VideoSourceAction">
      <Keys size="2">
        <key val="VideoSourceImage"/>
        <key val=""/>
      </Keys>
      <ActionConfig preferred_height="480" preferred_width="640" shutter="-1" source_url="ds"
    </VideoSourceAction>
    <ImageConvertActionT__ImageT__RGB_FrameImageT__GREY_Frame category="Action" name="ImageCo
      <Keys size="2">
        <key val="VideoSourceImage"/>
        <key val="ConvertedImage"/>
      </Keys>
    </ImageConvertActionT__ImageT__RGB_FrameImageT__GREY_Frame>
    <MarkerTrackerAction category="Action">
      <Keys size="5">
        <key val="ConvertedImage"/>
        <key val="IntrinsicData"/>
        <key val="World"/>
        <key val="MarkerTrackerInternalContour"/>
        <key val="MarkerTrackerInternalSquares"/>
      </Keys>
      <ActionConfig MTAThresh="140" MTAcontrast="0" MTAlogbase="10" WithKalman="0" WithPoseNl
    </MarkerTrackerAction>
    <TrackedObject2CameraAction category="Action" name="TrackedObject2Camera">
      <Keys size="3">
        <key val="World"/>
        <key val="IntrinsicData"/>
        <key val="Camera"/>
      </Keys>
    </TrackedObject2CameraAction>
  </ActionPipe>

  <DataSet key="">
    <IntrinsicDataPerspective calibrated="1" key="IntrinsicData">
      <!--Image resolution (application-dependant)-->
      <Image_Resolution h="480" w="640"/>
      <!--Normalized principal point (invariant for a given camera)-->
      <Normalized_Principal_Point cx="5.0037218855e-01" cy="5.0014036507e-01"/>
      <!--Normalized focal length and skew (invariant for a given camera)-->
      <Normalized_Focal_Length_and_Skew fx="1.6826109287e+00" fy="2.2557202465e+00" s="-5.734
      <!--Radial and tangential lens distortion (invariant for a given camera)-->
      <Lens_Distortion k1="-1.6826758076e-01" k2="2.5034542035e-01" k3="-1.1740904370e-03" k4
    </IntrinsicDataPerspective>
    <World key="World">
      <TrackedObject key="TrackedObject">
        <ExtrinsicData calibrated="0">
          <R rotation="1 0 0 &#xA;"/>
          <t translation="0 0 0 &#xA;"/>
          <Cov covariance="0  0  0  0  0  0  &#xA;0  0  0  0  0  0  &#xA;0  0  0  0  0  0  &#
        </ExtrinsicData>
        <Marker BitSamples="2" MarkerSamples="6" NBPoints="4" key="Marker1">
          <Code Line1="1100" Line2="1100" Line3="0100" Line4="0000"/>
          <Points3D nb="4">
            <HomgPoint3Covd Cov3x3="0  0  0  &#xA;0  0  0  &#xA;0  0  0  &#xA;" w="1" x="0" y
            <HomgPoint3Covd Cov3x3="0  0  0  &#xA;0  0  0  &#xA;0  0  0  &#xA;" w="1" x="6" y
            <HomgPoint3Covd Cov3x3="0  0  0  &#xA;0  0  0  &#xA;0  0  0  &#xA;" w="1" x="6" y
            <HomgPoint3Covd Cov3x3="0  0  0  &#xA;0  0  0  &#xA;0  0  0  &#xA;" w="1" x="0" y
          </Points3D>
        </Marker>
      </TrackedObject>
    </World>
  </DataSet>

</VisionLib2>
```

The scene file instantiates an IOSensor based on the VisionLib config file. The output of this IO sensor is then routed to a texture and a Viewfrustum node.

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D>
    <Engine DEF='engine'>
            <TimerJob DEF='timer'/>
            <SynchronizeJob DEF='synchronize'/>
            <RenderJob DEF='render'>
                    <WindowGroup>
                            <Window position='10 50' size='640,480' fullScreen='false' />
                    </WindowGroup>
            </RenderJob>
    </Engine>

    <Scene DEF='scene'>
            <IOSensor DEF='VisionLib' type='VisionLib' configFile='visionlib.pm'>
                    <field accessType='outputOnly' name='VideoSourceImage' type='SFImage'/>
                    <field accessType='outputOnly' name='TrackedObjectCamera_ModelView' type=
                    <field accessType='outputOnly' name='TrackedObjectCamera_Projection' type
                    <field accessType='outputOnly' name='TrackedObjectCamera_Position' type='
                    <field accessType='outputOnly' name='TrackedObjectCamera_Orientation' typ
            </IOSensor>

            <Viewfrustum DEF='vf' />

            <PolygonBackground>
                    <Appearance positions='0 0, 1 0, 1 1, 0 1' >
                            <TextureTransform rotation='0' scale='1 -1'/>
                            <PixelTexture2D DEF='tex' autoScale='false'/>
                    </Appearance>
            </PolygonBackground>

            <Transform translation='0 0 0'>
                    <Shape DEF='geo2'>
                            <Appearance>
                                    <Material emissiveColor='1 0.5 0' />
                            </Appearance>
                            <Teapot size='5 5 5' />
                    </Shape>
            </Transform>

            <ROUTE fromNode='VisionLib' fromField='VideoSourceImage' toNode='tex' toField='im
            <ROUTE fromNode='VisionLib' fromField='TrackedObjectCamera_ModelView' toNode='vf'
            <ROUTE fromNode='VisionLib' fromField='TrackedObjectCamera_Projection' toNode='vf
    </Scene>
</X3D>

<Viewpoint DEF='vf' fieldOfView='0.5' />

<ROUTE fromNode='VisionLib' fromField='VideoSourceImage' toNode='tex' toField='image'/>
<ROUTE fromNode='VisionLib' fromField='TrackedObjectCamera_Position' toNode='vf' toField='pos
<ROUTE fromNode='VisionLib' fromField='TrackedObjectCamera_Orientation' toNode='vf' toField='

#VRML V2.0 utf8

DEF trackingSensor IOSensor {
    type "VisionLib"
    configFile "visionlib.pm"
    eventOut SFImage        VideoSourceImage
    eventOut SFMatrix4f TrackedObjectCamera_ModelView
    eventOut SFMatrix4f TrackedObjectCamera_Projection
    eventOut SFVec3f        TrackedObjectCamera_Position
    eventOut SFRotation TrackedObjectCamera_Orientation
}

DEF trans Transform {
  children [
    Shape {
      appearance Appearance {
        texture DEF tex PixelTexture2D {
        }
      }
      geometry Box {
      }
    }
  ]
}

ROUTE trackingSensor.VideoSourceImage TO tex.image
ROUTE trackingSensor.TrackedObjectCamera_Orientation TO     trans.rotation
```

4.8.4 Modifications

This section gives you some clues what to change to get your setup running.

4.8.4.1 VideoSource

The example above uses DirectShow (or QT on the Mac) to access a camera. This should work for all cameras which support it, these will usually have a WDM driver to be installed. To use other cameras you need to change the VideoSource:ActionConfig:source_url field in the .pm file. There are also a number of arguments which can be passed to the video source driver. The arguments are added to the source url like this: *driver://parameter1=value;parameter2=value* .

Some drivers and their parameters are (available on platform in parentheses):

ds (win32, darwin) Windows driver as mentioned above, on a Mac this is the same as “qtvd” Parameters are: device - string name of the camera, mode - string name of the mode, framerate - integer. The driver compares the given parameters to whatever DS reports about the camera, if there is a mach the maching parameters are used, other values are ignored.

vfw (win32) Old VideoForWindows driver. That is a good luck driver, no parameters implemented.

v4l (linux) Works with video4linux (old version 1). No parameter support yet but you can pass something like v4l:///dev/myvideodev to select a device and it reads environment variables VIDEO_SIZE which is an integer value [0-10] which selects a video size between 160x120 and 768x576

ieee1394 (win32) FireWire DC cameras which run with the CMU driver http://www.cs.cmu.edu/~iwan/1394/index.html on windows. No parameters available for now.

ieee1394 (linux) FireWire DC cameras which run with the video1394 kernel module and libdc1394 (coriander), which includes PGR devices. Parameters are: unit - integer value for selecting a camera at the bus, trigger - boolean [0,1] 1 switches on external trigger, downsample - boolean [0,1] downsamples a bayer coded image to half size, device - string like “/dev/video1394/0” the device file to use.

ieee1394pgr (win32) PointGreyResearch cameras, license needed. Parameters are: unit - integer value for selecting a camera at the bus, trigger - boolean [0,1] 1 switches on external trigger, downsample - boolean [0,1] downsamples a bayer coded image to half size, mode - string value to select a mode, when passing “mode=320” some mode with a resolution of 320x240 is selected.

ueye (win32, linux) IDS imaging uEye cameras, license needed (more adjustments then the ds drivers) Parameters: downsample - boolean [0,1] downsamples a bayer coded image to half size,

vrmc (win32) VRmagic cameras, license needed, No parameters supported yet.

qtvd (darwin) Mac QuickTimeVideoDigitizer, no parameters yet

Some of these drivers require additional libs/dlls which must be installed on your system and in the path.

4.8.4.2 Marker

A marker in IV is described by a 4x4 code mask and four corner points. You can easily change the marker code by editing the fields DataSet:World:TrackedObject:Marker:Code:LineX. The marker is made of 4 lines Line1 = 1100 Line2 = 1100 Line3 = 0100 Line4 = 0000 where 0 = black and 1 = white, e.g.

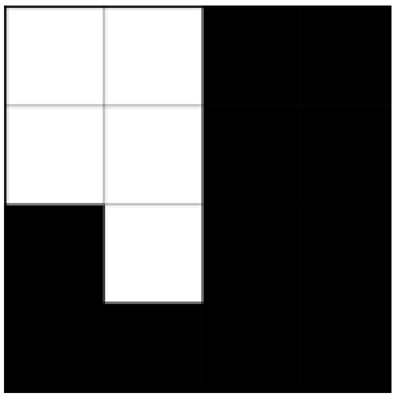


Figure 4.15: marker code

the real marker must have a black square around this and another white square around. It will look like

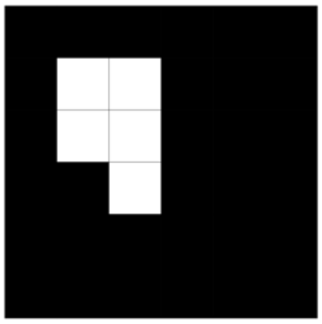


Figure 4.16: full marker

One way to create and print whose things is to go into word and create a 8x8 table, make the rows and cols the same size and color the cell background with black’n’white.

You can also change the position of the marker in the world by changing ...Marker:Points3D:HomgPoint3Covd:[xyz] values, make sure the marker stays rectangular and planar. The points describe the outer black border (6x6 field) not the white surrounding, they corrospond to upper left, upper right, lower right, lower left corners of the image.

You also use multiple markers in one TrackedObject, just duplicate DataSet:World:TrackedObject:Marker and change one of them to reflect its physical position on the object you want to track.

4.9 Apple Sudden Motion Sensor

This section shows you how to use the Apple Sudden Motion Sensor inside a 3d scene.

4.9.1 Introduction

In 2005 Apple introduced the Sudden Motion Sensor for its portable computers in order to protect the hardrive. This example only works on Apple Powerbooks, iBooks, MacBooks and MacBooks Pro build after 2005.

The sensor is a 3-axis accelerometer. The AppleMotionSensor backend delivers a Vec3f with the three acceleration values.

4.9.2 Shaking

In the first example we create an IOSensor of the type AppleMotionSensor and route its values to a Transform node. The values are getting smoothed by a PositionDamper.

The AppleMotionSensor is loaded like any other HID device via an IOSensor. The acceleration values are stored in the field *Motion* :

- *Motion (SFVec3f)* : Acceleration values (x, y, z)

```
<IOSensor DEF='AppleMotionSensor' type='AppleMotionSensor'>
    <field accessType='outputOnly' name='Motion' type='SFVec3f'/>
</IOSensor>
```

The acceleration values could be routed to a Transform node. But in order to smooth the values we are putting a PositionDamper inbetween.

```
<Transform DEF='tr'>
    <Shape>
            <Appearance>
                    <Material diffuseColor='1 1 1' />
            </Appearance>
            <Box/>
    </Shape>
</Transform>

<PositionDamper DEF='pd' tau='0.1' />

<ROUTE fromNode='AppleMotionSensor' fromField='Motion' toNode='pd' toField='set_destination'/
<ROUTE fromNode='pd' fromField='value_changed' toNode='tr' toField='set_translation'/>
```

Files:

- test_suddenMotion.x3d (Example)

4.9.3 Tilt

In this second example we are mapping the acceleration on the orientation of an object.

Calling the SFRotation() constructor with the acceleration vector and a vector SFVec3f(0,1,0) calculates the sensor's orientation. By routing that value on a Transform's *rotation* the object seems to keep its position while rotating the notebook.

```
<Script DEF='script'>
    <field accessType='inputOnly' name='set_motion' type='SFVec3f'/>
    <field accessType='outputOnly' name='rotation_changed' type='SFRotation'/>
    <![CDATA[javascript:

            var rotation_changed;
            var vector = new SFVec3f(0,1,0);

            function set_motion(motion)
            {
                    rotation_changed = new SFRotation(motion, vector);
            }

    ]]>
</Script>

<ROUTE fromNode='AppleMotionSensor' fromField='Motion' toNode='script' toField='set_motion'/>
<ROUTE fromNode='script' fromField='rotation_changed' toNode='tr' toField='set_rotation'/>
```

Files:

- test_suddenMotion_02.x3d (Example)

4.10 Serial Communication

This section shows you how to communicate with a serial device.

4.10.1 Introduction

This section shows how to communicate with a serial device within a 3D scene. Possible devices are microcontroller boards like Arduino or Wiring , but also for example Wacom tablets, GPS devices and rotary encoders.

4.10.2 Setting up the serial port

The serial port is set up with an IOSensor node at the beginning of the scene. The following parameters are available:

- *Device* : A number starting at 0 specifying the serial interface to use. 0 for COM1 or the first tty serial device. *(default: 0)*
- *BaudRate* : Baud rate of the serial port. 9600, 19200, ... *(default: 9600)*
- *DataBits* : Number of data bits used for the communication on the serial port. Possible values are 7 or 8. *(default: 8)*
- *Parity* : Type of parity used for the communication on the serial port. Possible values are even, odd or none. *(default: none)*
- *StopBits* : Number of stop bits used for the communication on the serial port. Possible values are 1 or 2. *(default: 1)*
- *Handshake* : Type of handshake (flow control). Possible values are none, hardware or software. *(default: none)*
- *DTR* : The status of the DTR line.
- *RTS* : The status of the RTS line.
- *Init String* : An initialisation string that is send to the serial device to start operation.

- *Deinit String* : A deinitialisation string that is send to the serial device to stop operation.
- *Delimiter* : Ascii value of the character that splits the serial message *(default: no delimiter)*
- *MaxBytes* : The maximum number of bytes a message consists of. A value of -1 means that there is no maximum number of bytes. *(default: -1)*

```
<IOSensor DEF='serial' type='serial' Device='0' Delimiter='10' BaudRate='9600'>
    <field accessType='outputOnly' name='Data out' type='SFString'/>
    <field accessType='inputOnly' name='Data in' type='SFString'/>
</IOSensor>
```

Here we are seeting up a serial device with 9600 baud rate at COM1 or tty.usbserial-00001. The delimiter is set to a line break (ASCII value: 10). All other parameters have the default values.
There are two fields for the incoming and outgoing data:

- *Data out* : Data from the serial device to the scene
- *Data in* : Data from the scene to the serial device

4.10.3 Sending Data to the Serial Port

By routing the KeySensors' *keyPress* field to the serial devices' *Data in* field we are sending each keystroke (SFString) to the serial port. We also specify a name for the device handler via the *name* field.

```
<KeySensor DEF='keysensor' />
<ROUTE fromNode='keysensor' fromField='keyPress' toNode='serial' toField='Data in'/>
```

4.10.4 Receiving Data from the Serial Port

In order to get the data from the serial port and to show it in the scene we are routing the values from the serial devices' *Data out* field to a Text nodes' *string* field.

```
<Transform>
    <Shape>
            <Text DEF='text' string='' solid='true'>
    </Shape>
</Transform>

<ROUTE fromNode='serial' fromField='Data out' toNode='text' toField='string'/>
```

4.10.5 Example Scene

This is a simple example for the communication between an Arduino microcontroller and instant *viewer* . We are sending keystrokes from the scene to the controller. The software on the Arduino board switches an LED on when "1" is sent and switches it off when "2" is sent. It sends the Strings "On" and "Off" back to the scene where it is routed on a Text node's string.

```
<X3D>
  <Scene DEF='scene'>

    <IOSensor DEF='serial' type='serial' Device='0' Delimiter='10' BaudRate='9600'>
            <field accessType='outputOnly' name='Data out' type='SFString'/>
            <field accessType='inputOnly' name='Data in' type='SFString'/>
    </IOSensor>

    <Viewpoint position='0.625 0.3 1.9' />

    <Transform>
```

```
            <Shape>
                    <Appearance>
                            <Material diffuseColor='1 1 1' />
                    </Appearance>
                    <Text DEF='text' string='/../' solid='true'>
                            <FontStyle justify='BEGIN' family='SANS' />
                    </Text>
            </Shape>
    </Transform>

    <ROUTE fromNode='serial' fromField='Data out' toNode='text' toField='string'/>


    <KeySensor DEF='keysensor' />
    <ROUTE fromNode='keysensor' fromField='keyPress' toNode='serial' toField='Data in'/>

  </Scene>
</X3D>
```

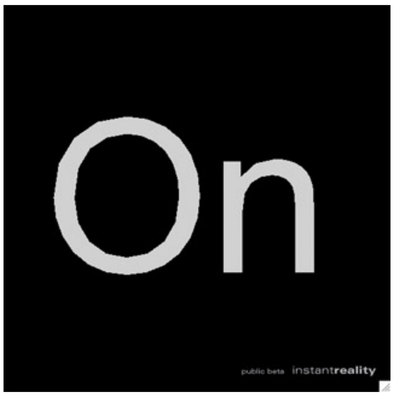


Figure 4.17: Scene

```
void setup()
{
  Serial.begin(9600);
  pinMode(13, OUTPUT);
}

void loop()
{
  if (Serial.available() > 0)
  {
    int incoming = Serial.read();

    if ((char)incoming == '1')
    {
      digitalWrite(13, HIGH);
      Serial.println("On");
    }
    else if ((char)incoming == '2')
    {
      digitalWrite(13, LOW);
      Serial.println("Off");
    }
    }
  }
```

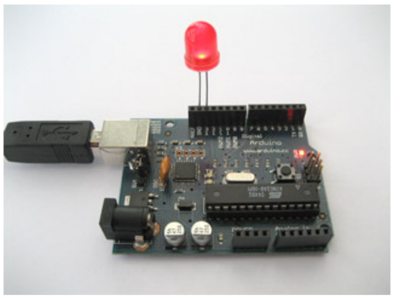


Figure 4.18: Arduino LED

Files:

- serialTutorial.x3d
- serialExample.pde

Chapter 5

Clustering

5.1 Cluster Basics and Load Balancing

This section gives an overview over cluster topologies in computer graphics. It continues with the often ignored topic of load balancing, as clusters are mostly used to set up multi display environments without a real distribution of load. In InstantReality the activation of load balancing is very easy. You will also get a basic introduction on how to setup your network and start cluster servers to be ready to use for your cluster. Different setups will be explained in the following sections of this Clustering section.

5.1.1 Topologies

To make expensive computations faster, a common approach nowadays is establishing a cluster of more or less convenient hardware, let's say PCs. In computer graphics we can imagine several configurations using a cluster of PCs as you can see in following scenarios.

Most common systems look like the illustration below, several PCs are connected over a network to render and display a virtual scene together on multiple displays. The advantage is a high resolution but as one PC is responsible for exactly one display the system is only as fast as the PC which has the highest render load.

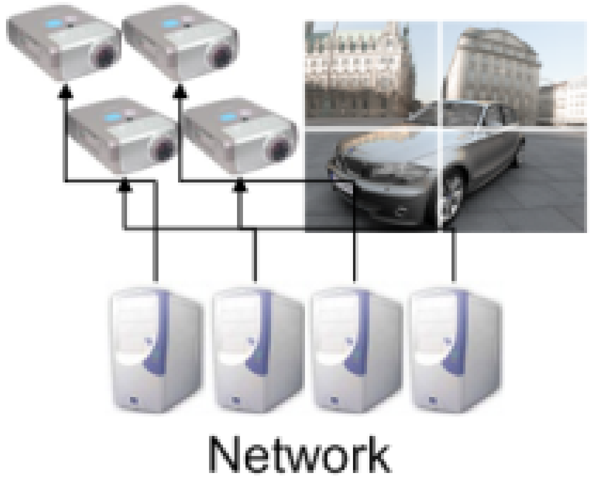


Figure 5.1: Multi display cluster configuration

To render complex geometries a configuration as shown below is possible. The scene has to be distributed to PCs in a cluster, they compute their task and send the results back to the PC which composes and displays the final image. In this case we use a cluster to render complex scenes but we lose the high resolution of a multi display system.

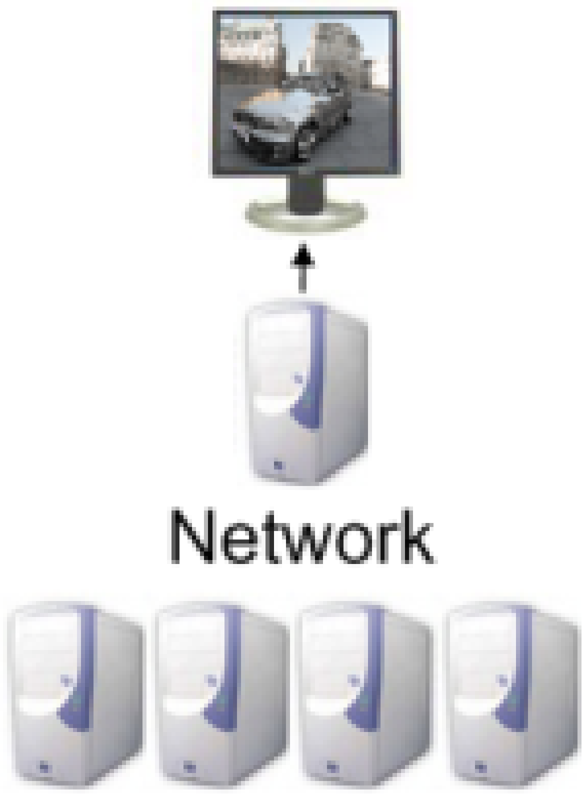


Figure 5.2: Single display cluster configuration

If we like to benefit of both advantages, i.e. high resolution and effective rendering of complex models, we need a flexible system. In this system we have several PCs in a network where some are playing a role as display and others that are not. This topology can also be used as stereo setup for instance.

The InstantReality system provides the concept of using an arbitrary number of displays in a specified alignment together with an arbitrary number of PCs. Those can be connected to a display or not. This flexible concept allows each of the above basic setups. But it can also be used to create more complex configurations like cave environments with displays which are orthogonal to each other or even displays in an arbitrary angle. To see how simple a cluster is configured please read the following sections in the Clustering section. But you should finish this one for load balancing aspects and basic IR cluster setup information.

5.1.2 Load balancing

In a cluster, PCs should share the overall load equally between each other to be effective and to get the best performance. In computer graphics all of them generate an equally cost intensive part of the scene. A proper and fast precomputation of the scene takes place before distributing parts of it to the PCs. Finally every rendered section which belongs to another display PC is copied over the network to its target. This approach offers arbitrary setups like single display and multi display systems, both as mono or stereo solution with an effective balancing of the upcoming load.

There are several approaches to balance the load of 3D scenes. Two big categories are image space distribution and geometry distribution.

5.1.2.1 Image space balancing (Sort-First)

In image space distributed balancing, also called *Sort-First* , a precomputation takes place which transforms only bounding boxes of objects into camera space to get the approximate position of geometries on the displays. By this information a cost function is estimated. This is based on

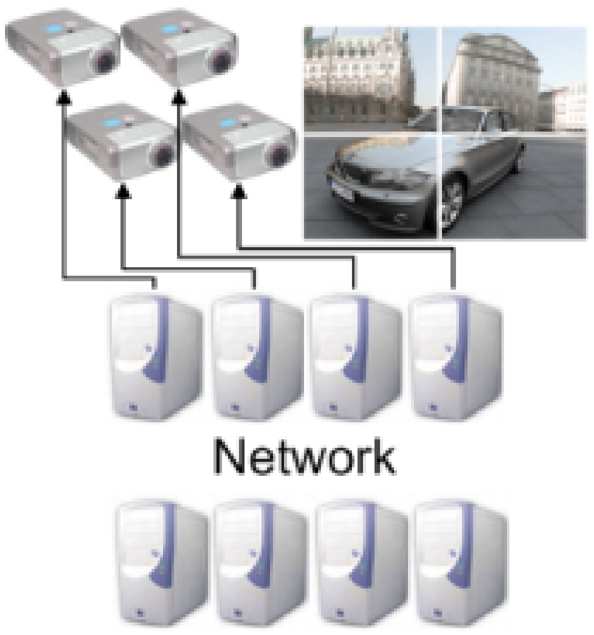


Figure 5.3: Multi display configuration with load balancing

transformation costs as well as the rasterization costs and therefore takes the number of vertices and size of rendered bounding boxes into account.

Now that each PC has it's estimated costs, parts of viewports are distributed to render on another PC which has only low costs. The resulting rendered parts are sent back to the display PC over network. If you want to learn more about the implemented cost estimation and load balancing algorithms, please check the *Technical Details* section at the end of this section.

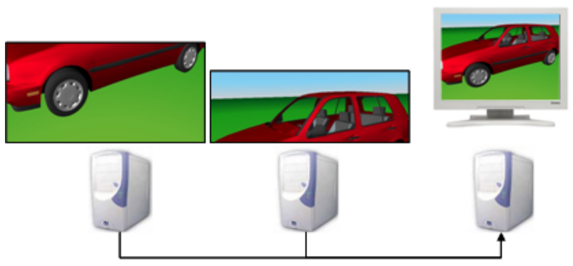


Figure 5.4: Image space based load balancing

5.1.2.2 Geometry based balancing (Sort-Last)

In this kind of load balancing, parts of the scenegraph are distributed. That means geometries are distributed between PCs and after rendering the pixel data including the depth information is copied back to the display PC. On the display PC all received images are composed to one image again by involving the depth. This approach shouldn't be used on multi display systems for one reason. Geometries can be bigger than a single display resolution, so it can't be rendered by only one graphics card as it doesn't fit to the framebuffer. But on a single display cluster system it is very fast and a better choice than *Sort-First* .

Important

To use load balancing effectively it is very important to have 1000Mbit network, because pixel data has to be sent over network. Otherwise you won't have the advantages of load balancing.

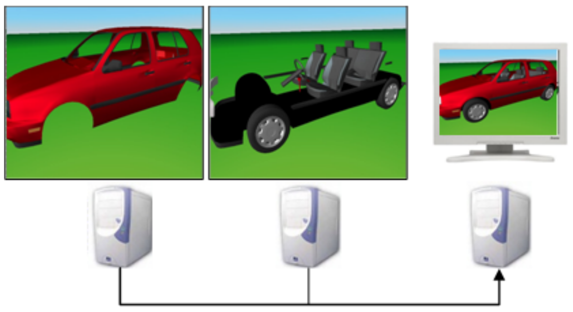


Figure 5.5: Geometry based load balancing

5.1.3 Network configuration

Setup your network. Each host must be reachable by name from each other host. Try this with `ping host` . If you have a local network (no access to the internet), you have to define a dummy gateway eg. 192.168.1.254 if your network uses the IP-Range 192.168.1.0 - 192.168.1.253. On Linux, if your /etc/hosts file contains a line like `127.0.0.1 myHostname` where myHostname is not localhost, then remove this line.

5.1.4 InstantPlayer and InstantCluster

Doing cluster rendering with the InstantReality framework you have one instance of the InstantPlayer running your X3D application. This application provides the user interface, loads the x3d file including the engine configuration and does all the simulation for your scene. To be able to produce a graphics output on another host, you have to run an InstantCluster on this host. Start the “InstantCluster” entry in your menu or application directory or use an autostart mechanism to run it all the time. It only needs resources while rendering and otherwise sleeps and waits for connections.

5.1.5 Technical Details

For technical details about the algorithms of the load balancing and some benchmarks I suggest to have a look into the paper Load Balancing on Cluster-Based Multi Projector Display Systems .

With sort-first approach, i.e. image space based distribution, we got a speedup of 3 to 6 for a single display setup and 16 PCs in a cluster. On multi display systems with 48 PCs (24 PCs for displaying in 6 x 4 alignment) animations were 3 to 4 times faster.

Sort-last balancing with a model of Standford’s David Statue and 56 millions of polygons achieved a speedup of 10 with 16 PCs in the cluster and even over 20 with more than 32 PCs. For the composition of image parts it uses a new pipeline approach.

5.2 CAVE cluster

This section demonstrates how to setup a CAVE environment with three projection walls. It will also take stereo functionality into account which is essential in a CAVE as well as load balancing between cluster PCs.

To get knowledge of multi display configurations and stereo setups, I suggest to work through the other *Clustering* sections and *Multiple Views and Stereo* . After that you will know everything about a ClusterWindow and Viewarea s. Those nodes will be used in this section and extended by a view modifier named ProjectionViewModifier .

5.2.1 Aspects of a CAVE

A CAVE consists of up to six projection walls which are usually aligned orthogonal to each other to build a cubic room or a part of a cubic room. That’s the main reason why we have to use Cluster-Window instead of the preconfigured node TiledClusterWindow (see section *Multiple display cluster*

). Each wall shows a different view of the scene, i.e. the camera looks at different directions. So we have to configure each view manually with a ProjectionViewModifier which modifies the camera orientation. Another attribute of a CAVE is it's immersive character, due to stereo projection.

Figure 5.6: CAVE with 5 walls

5.2.2 Assumptions

In this section we assume to have a CAVE with 3 walls. A bottom plane, a front plane and one side wall. Each wall has the size of 2.4 x 2.4 meters. We want the camera of our scene to be in the middle of the CAVE. Let's also assume, our scene is modeled in meters, so the side wall is -1.2 to the left of the camera, the bottom plane -1.2 below and the front plane -1.2 to the front (negative z-axis). Each projection should also have a square resolution of 1024 x 1024 pixels, generated by PCs with a standard framebuffer resolution of 1280 x 1024 pixels.

5.2.3 Setting up the views

Now we want to configure a different view for each wall. Therefore a concept exists which allows to define a plane in space with four points. This plane acts as a projection plane for the scene relative to the camera.

The projection plane is configured via the node ProjectionViewModifier , a modifier for the Viewarea node like ShearedStereoViewModifier . It also inherits the fields *leftEye* , *rightEye* and *eyeSeparation* from the stereo modifier.

The projection view modifier for the left wall and the left eye in our CAVE setup will look like this:

```
...
modifier [
  DEF mod_front_left ProjectionViewModifier {
    surface [
      -1.2 -1.2 -1.2,
       1.2 -1.2 -1.2,
       1.2  1.2 -1.2,
      -1.2  1.2 -1.2
    ]
    leftEye  TRUE
    rightEye FALSE
    eyeSeparation 0.08
```

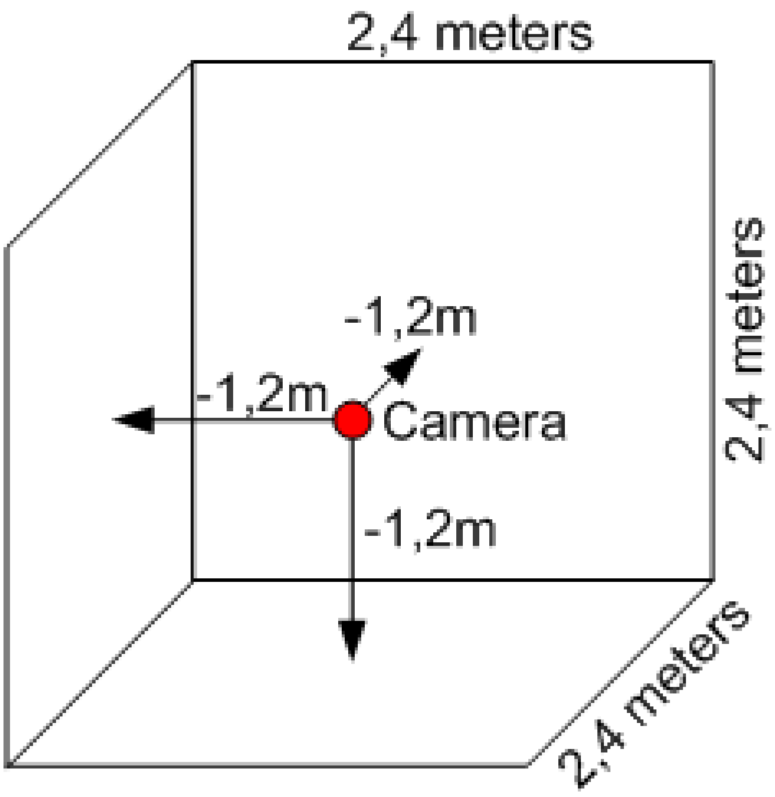


Figure 5.7: CAVE scheme with 3 walls

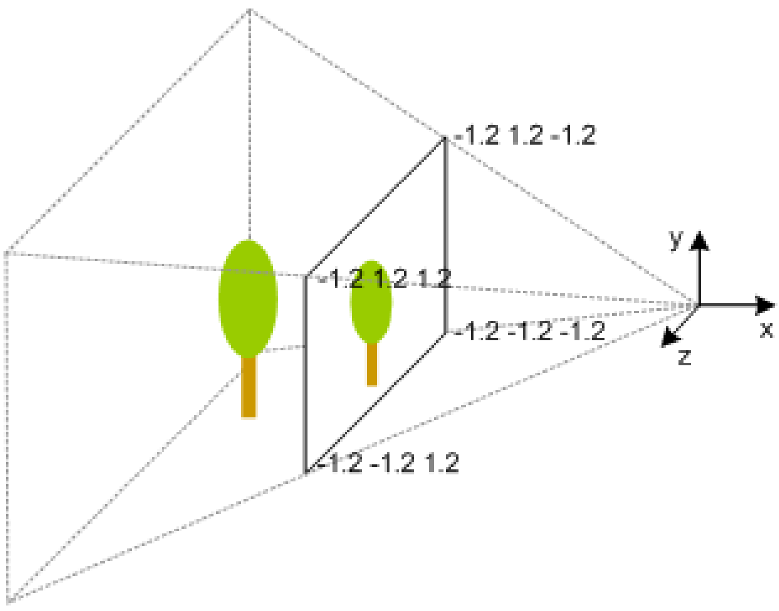


Figure 5.8: Projection plane for the left CAVE wall (proportions are not authentic)

```
    }
  ]
  ...
```

The surface points have to be counterclockwise, starting with the lower left corner. For the right eye on the same wall you just have to set *leftEye* to *FALSE* and *rightEye* to *TRUE* .

Respectively the front wall has to be set up like this:

```
  ...
  modifier [
    DEF mod_left_left ProjectionViewModifier {
      surface [
        -1.2 -1.2  1.2,
        -1.2 -1.2 -1.2,
        -1.2  1.2 -1.2,
        -1.2  1.2  1.2
      ]
      leftEye  TRUE
      rightEye FALSE
      eyeSeparation 0.08
    }
  ]
  ...
```

And finally for the floor:

```
  ...
  modifier [
    DEF mod_bottom_left ProjectionViewModifier {
      surface [
        -1.2 -1.2  1.2,
         1.2 -1.2  1.2,
         1.2 -1.2 -1.2,
        -1.2 -1.2 -1.2
      ]
      leftEye  TRUE
      rightEye FALSE
      eyeSeparation 0.08
    }
  ]
  ...
```

Important

Important: Use the same unit (e.g. metres, millimetres) for the projection surfaces like your scene is modeled in. Otherwise you will get interesting field of views.

With stereo configuration we now have 6 different views. Each view should be rendered by one PC. Let's call them *front_leftEye* , *front_rightEye* , *left_leftEye* , *left_rightEye* , *bottom_leftEye* and *bottom_rightEye* . As mentioned in the *Assumptions* section, each PC has a resolution of 1280 x 1024 pixels. So we will need a window with enough space for each PC (better said framebuffer of PC), which results in a ClusterWindow of a size of 7680 x 1024 pixels. The ClusterWindow configuration now looks like this:

```
  ...
  ClusterWindow {
    servers [
      "front_leftEye"
      "front_rightEye"
      "left_leftEye"
      "left_rightEye"
      "bottom_leftEye"
      "bottom_rightEye"
    ]
    size 7680 1024
    hServers 6
    vServers 1
    ...
  }
  ...
```

This configuration results in the following partitioning of the window:

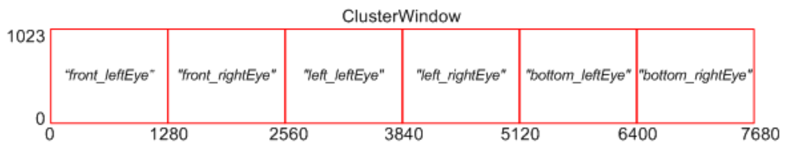


Figure 5.9: Partitioning of the cluster window

The last missing issue is the setup of view areas on the cluster window, because CAVE walls are square (1024 x 1024) but framebuffers of the PCs are not (1280 x 1024). We will define a square Viewarea per non-square PC region on the cluster window and put one of the above projection view modifiers into each.

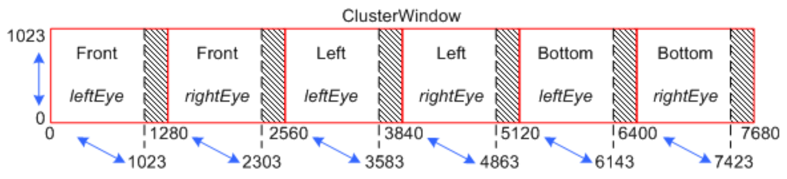


Figure 5.10: CAVE view areas in a cluster window

Then we will obtain the final configuration:

```
  ...
  DEF render RenderJob {
    windowGroups [
      WindowGroup {
        windows [
          LocalWindow {
            #This window is just for interaction
            enabled FALSE
          }
          ClusterWindow {
            servers [
              "front_leftEye"
              "front_rightEye"
              "left_leftEye"
              "left_rightEye"
              "bottom_leftEye"
              "bottom_rightEye"
            ]
            size 7680 1024
            hServers 6
            vServers 1
            views [
              #Front wall, left eye
```

```
Viewarea {
  lowerLeft  0 0
  upperRight 1023 1023
  modifier [
    DEF mod_front_left ProjectionViewModifier {
      surface [
        -1.2 -1.2 -1.2,
         1.2 -1.2 -1.2,
         1.2  1.2 -1.2,
        -1.2  1.2 -1.2
      ]
      leftEye  TRUE
      rightEye FALSE
      eyeSeparation 0.08
    }
  ]
}
#Front wall, right eye
Viewarea {
  lowerLeft  1280 0
  upperRight 2303 1023
  modifier [
    DEF mod_front_right ProjectionViewModifier {
      surface [
        -1.2 -1.2 -1.2,
         1.2 -1.2 -1.2,
         1.2  1.2 -1.2,
        -1.2  1.2 -1.2
      ]
      leftEye  FALSE
      rightEye TRUE
      eyeSeparation 0.08
    }
  ]
}
#Left wall, left eye
Viewarea {
  lowerLeft  2560 0
  upperRight 3583 1023
  modifier [
    DEF mod_left_left ProjectionViewModifier {
      surface [
        -1.2 -1.2  1.2,
        -1.2 -1.2 -1.2,
        -1.2  1.2 -1.2,
        -1.2  1.2  1.2
      ]
      leftEye  TRUE
      rightEye FALSE
      eyeSeparation 0.08
    }
  ]
}
#Left wall, right eye
Viewarea {
  lowerLeft  3840 0
  upperRight 4863 1023
  modifier [
                DEF mod_left_right ProjectionViewModifier {
                  surface [
                    -1.2 -1.2  1.2,
                    -1.2 -1.2 -1.2,
                    -1.2  1.2 -1.2,
                    -1.2  1.2  1.2
                  ]
                  leftEye  FALSE
                  rightEye TRUE
                  eyeSeparation 0.08
                }
              ]
            }
            #Bottom, left eye
            Viewarea {
              lowerLeft  5120 0
              upperRight 6143 1023
              modifier [
                DEF mod_bottom_left ProjectionViewModifier {
                  surface [
                    -1.2 -1.2  1.2,
                     1.2 -1.2  1.2,
                     1.2 -1.2 -1.2,
                    -1.2 -1.2 -1.2
                  ]
                  leftEye  TRUE
                  rightEye FALSE
                  eyeSeparation 0.08
                }
              ]
            }
            #Bottom, right eye
            Viewarea {
              lowerLeft  6400 0
              upperRight 7423 1023
              modifier [
                DEF mod_bottom_right ProjectionViewModifier {
                  surface [
                    -1.2 -1.2  1.2,
                     1.2 -1.2  1.2,
                     1.2 -1.2 -1.2,
                    -1.2 -1.2 -1.2
                  ]
                  leftEye  FALSE
                  rightEye TRUE
                  eyeSeparation 0.08
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
...
```

View areas are set in pixels instead of relative window coordinates for following reasons:

- **Readability:** Pixel coordinates are much better to associate to a region than something like

0.633205...

- **Accuracy:** Relative coordinates like 0.633333 are not as accurate than defined pixel coordinates
- **Calibration:** You don't have to calibrate projectors with pixel accuracy, just calibrate the view areas

Finally the result are 6 views you need for the CAVE. The image below shows only three views because it doesn't take stereo into account. It shows the *Dome of Siena* with front, left and bottom views. For a better visualization in the top left corner these views are texturing a virtual 3-sided CAVE.

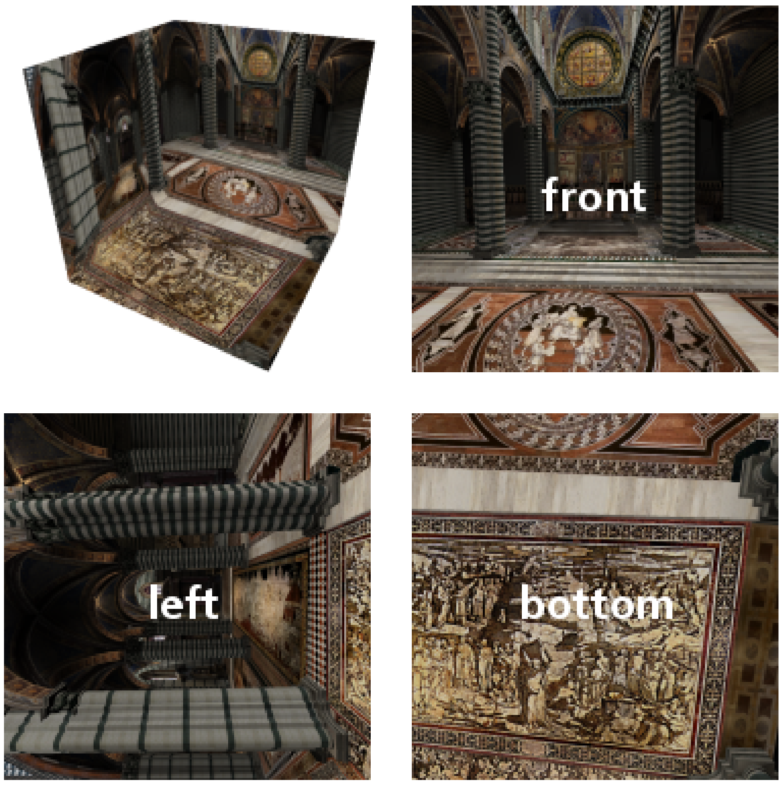


Figure 5.11: CAVE views (3 walls, mono)

5.2.4 Load balancing

If you've read the other *Clustering* sections, you will know how to switch load balancing on. Just add these two lines into the ClusterWindow node. The second one is just for debugging purpose to see how load balancing works. Be sure to use Gigabit LAN to obtain an effective balancing.

```
...
balance TRUE
showBalancing TRUE
...
```

5.2.5 Head tracking

One important issue has not been taken into account yet. As a user is moving around in a CAVE, the eye position is not the same as the camera position which is in the middle due to our setup of the projection planes. So the viewing frustums for each wall are not correct for the users field of view. They have to be adapted to the users position like the image below illustrates.

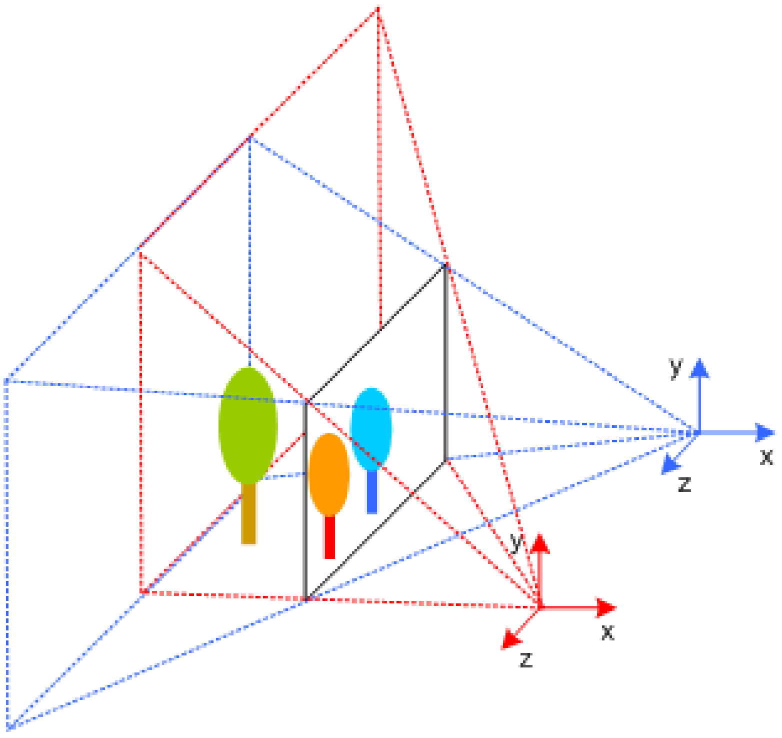


Figure 5.12: Different head positions and resulting view frustums

You see a projection plane with a red and a blue tree. The tree images belong to the red and blue viewing frustums. A users head is represented by coordinate systems which lie at the end of the frustums. When the head moves from blue to red position the accordant frustum is significant to show the correct view on the projection plane.

This additional modification of the camera is also done in the ProjectionViewModifier . A head tracking device is needed which returns a 4 x 4 transformation matrix for position and orientation of a head in a CAVE. The hardware is mostly an infrared or a magnetic device attached to stereo glasses. The device handling is not in scope of this section, but finally if you have a node which produces a transformation matrix from the device, you have to route it to the *set_eyeTransform* field of all ProjectionViewModifier nodes like this:

```
...
DEF render RenderJob {
  ...
}

ROUTE headSensor.value_changed TO mod_front_left.set_eyeTransform
ROUTE headSensor.value_changed TO mod_front_right.set_eyeTransform
ROUTE headSensor.value_changed TO mod_left_left.set_eyeTransform
ROUTE headSensor.value_changed TO mod_left_right.set_eyeTransform
ROUTE headSensor.value_changed TO mod_bottom_left.set_eyeTransform
ROUTE headSensor.value_changed TO mod_bottom_right.set_eyeTransform
...
```

Files:

- CaveStereo.wrl
- tie.wrl (test model)

5.3 Single display cluster

This section shows how to setup a cluster for a single display configuration. It also describes how to use real load balancing with this setup.

Please read the section *Cluster Basics* in the *Clustering* category to get an overview of how to configure a PC cluster using InstantCluster.

5.3.1 Single Display cluster

In this section we want to use three PCs of our cluster to render an image on a single display. That means one PC is designated to display the final scene while the others support it by rendering parts of the scene on their local framebuffer and sending the results (pixels of the image region) back. The appropriate RenderJob section will look like this:

```
...
DEF render RenderJob {
   windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
          #This window is just for interaction
          enabled FALSE
        }
        ClusterWindow {
          servers [ "displaypc" "clusterpc1" "clusterpc2" ]
          hServers 1
          vServers 1
          size 1024 768
          balance TRUE
          showBalancing TRUE
        }
      ]
    }
   ]
}
...
```

We see two windows here, the first one is the LocalWindow , which only exists for user interaction. We disable rendering here to gain a real speedup for the cluster window. Otherwise the PC with the local window would have to render the whole scene itself.

The important part of the configuration is the ClusterWindow . This line

```
servers [ "displaypc" "clusterpc1" "clusterpc2" ]
```

lists the hostnames of PCs, which should take part in the cluster. It is followed by the specification of the display area by setting the number of horizontal and vertical displays as well as its resolution:

```
hServers 1
vServers 1
size 1024 768
```

In this example we just use a single display with a resolution of 1024 x 768 pixels. The server which is responsible for displaying is “displaypc”, because it is the first server in the servers list.

The next two lines set up the load balancing, where the field *showBalancing* is just for debugging purpose and pigments those areas of the rendered image which are generated by other servers and copied over the network.

```
balance TRUE
showBalancing TRUE
```

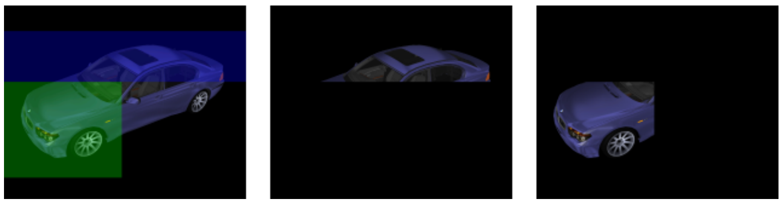


Figure 5.13: Single display cluster setup with 3 computers

The image shows framebuffers of three PCs, where the left one is dedicated to display the whole scene (single display constellation). The other ones generate rectangular parts of the scene and copy the pixel data to the first PC over a fast network. You can see the copied parts on the left image as coloured rectangles if *showBalancing* is set to TRUE.

It is very important for an effective load balancing to use a Gigabit network. Let's say we have a display PC and one additional server with a resolution of 1280 x 1024 pixels each. In the worst case the server has to deliver half or more of the the screen to the display PC. This is 1280 x 1024 / 2 = 655.360 pixels and for each pixel three color components (RGB), which results in almost 2 MB of data per frame. In a 100Mbit network we can send about 10 MB per second, so we would get a framerate of 5 fps!

Files:

- SingleDisplayLoadBalancing.wrl
- SingleDisplayLoadBalancing.x3d (same as above but in X3D syntax)
- tie.wrl (test model)

5.4 Multiple display cluster

This section shows how to setup a cluster for a multi display configuration. It will also implement real load balancing with this setup and extend it to a stereo configuration with five PCs.

Please read the *Cluster Basics* section in the *Clustering* category to get an overview of how to configure a PC cluster using InstantCluster. For information about stereo configurations you should take a look into the *Multiple Views and Stereo* section category (especially *Multiple Windows and Views* , *Stereo Basics* and *Passive Stereo* sections).

5.4.1 Multi display cluster

In this chapter we want to use three PCs of our cluster to render a scene over two displays. On one PC, just the application will run in a local window to provide interaction. Two other PCs (displaypc1 and displaypc2) are designated to display the scene over two screens.

5.4.1.1 Different concepts

There are two concepts of doing this. The first one is by using the known ClusterWindow node and the second is the TiledClusterWindow which is especially created for *n * m* displays arrangements and provides overlapping features. The latter is based on the first one internally and just simplifies the usage on some setups.

So the question is, when to use which node. The following list should get you on the right way. You should use a TiledClusterWindow if:

- you use multiple homogeneous displays to act as one big display
- all displays are in one plane
- above points apply and you want to use stereo
- you don't want to see "borders" between your displays (overlapping projections) ClusterWindow

- you use a single display cluster
- you setup a CAVE, i.e. multiple displays, but not in the same plane
- you want to configure view areas manually (e.g. for a CAVE), otherwise TiledClusterWindow is better to use

The main difference is the reduction of work when you have to configure stereo setups in a multi display cluster, because TiledClusterWindow configures view areas and different projection parameters for each area by itself. Another difference is the ability of this node to take overlapping into account. For CAVE setups you have to configure view areas manually, so you are free to arrange CAVE walls as you want. In this section both approaches will be explained and you will soon realize the advantage of the TiledClusterWindow .

5.4.1.2 Using ClusterWindow

The appropriate RenderJob section will look like this:

```
...
DEF render RenderJob {
   windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
          #This window is just for interaction
          enabled FALSE
        }
        ClusterWindow {
          servers [ "displaypc1" "displaypc2" ]
          hServers 2
          vServers 1
        }
      ]
    }
   ]
}
...
```

We see two windows here, the first one is the LocalWindow , which only exists for user interaction. We disable rendering here, otherwise the PC with the local window would have to render the whole scene itself. This can be a problem with large models, especially when using load balancing for the cluster later.

The important part of the configuration is the ClusterWindow . This line

```
servers [ "displaypc1" "displaypc2" ]
```

lists the hostnames of PCs, which should take part in the cluster. It is followed by the specification of the display area by setting the number of horizontal and vertical displays:

```
hServers 2
vServers 1
```

By setting *hServers* to 2 and *vServers* to 1, the whole window consists of two horizontal aligned displays. As there are just two displays (hServers * vServers), the first two servers (displaypc1 and displaypc2) are responsible for displaying the window area which is as large as the accumulated native resolutions of the PCs displays. The local window is opened on the machine from where the VRML file is loaded.

5.4.1.3 Using TiledClusterWindow

The appropriate RenderJob section will look like this:

Figure 5.14: Multi display cluster setup with 2 display PCs and one application PC

```
...
DEF render RenderJob {
   windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
          #This window is just for interaction
          enabled FALSE
        }
        TiledClusterWindow {
          servers [ "displaypc1" "displaypc2" ]
          columns 2
          rows 1
        }
      ]
    }
   ]
}
...
```

In this configuration the tiled window has 2 columns and 1 row as in the ClusterWindow case. When using the TiledClusterWindow, there are a few additional options, like overlapping:

```
overlapX 20
overlapY 0
```

These lines result in a region between the two displays which is rendered twice. So when using two video beamers, you can adjust these by taking the overlapping into account and an intersection will not be as noticable as without overlaps.

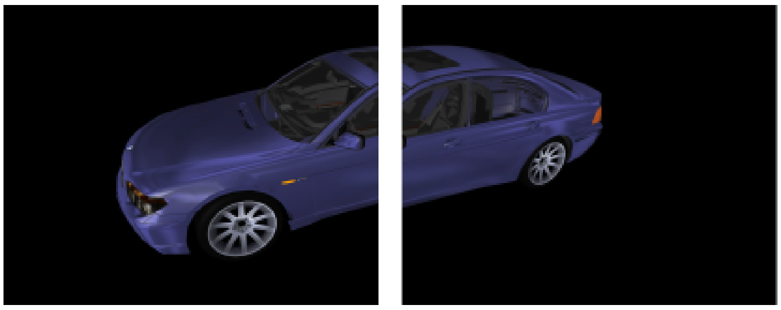


Figure 5.15: Two displays with X-overlap rendering

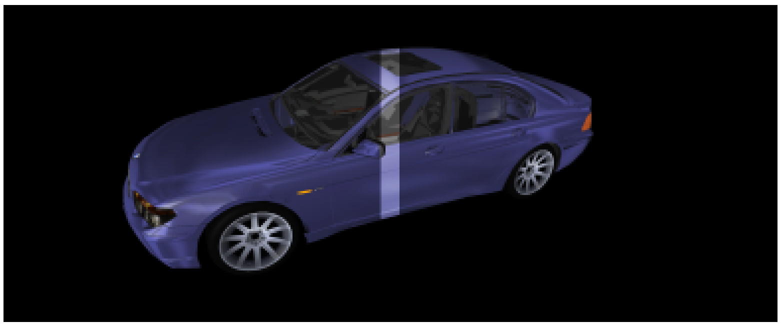


Figure 5.16: Overlapped displays (overlapping highlighted)

5.4.2 Load balancing

The next two lines set up the load balancing, where the field *showBalancing* is just for debugging purpose and pigments those areas of the rendered image which are generated by other servers and copied over the network. You just have to add these into the ClusterWindow node or TiledCluster-Window respectively.

```
balance TRUE
showBalancing TRUE
```

That means both cluster PCs support each other by rendering parts of the scene on their local framebuffer and sending the results (pixels of the image region) back. The role changes depending on which PC has the higher load. Additionally the resolution of the window has to be defined when using load balancing. In ClusterWindow you have to set the overall size of the window:

```
size 2048 768
```

You may adjust it to 2560 x 1024, if your single display resolution is 1280 x 1024. In TiledCluster-Window the size of the window is defined by setting the width and height of one tile. Together with *columns* and *rows* fields the window size is calculated internally:

```
tileWidth 1024
tileHeight 768
```

Tile sizes have usually to be adapted to the native resolution of one single display.

We will also attach an additional PC which is only used as support for the display PCs. We write the PC's name at the end of the servers list, because the first (defined through *hServer/vServers* or *rows/columns*) servers are automatically used as display.

```
servers [ "displaypc1" "displaypc2" "supportpc" ]
```

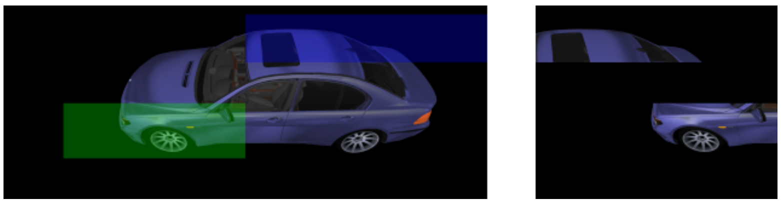


Figure 5.17: Multi display cluster setup with 3 computers and one application PC

5.4.3 Multi display stereo configuration

In a stereo configuration the existance of TiledClusterWindow will become clear. Imagine a stereo setup of two displays, i.e. two PCs for the displays of the left eye and two PCs for the displays of the right eye. One display (one tile) has the resolution of 1280 x 1024 pixels, so the whole window will have 2560 x 1024.

To foreclose the solution with a TiledClusterWindow here is the simple configuration:

```
...
DEF render RenderJob {
   windowGroups [
    WindowGroup {
      windows [
        LocalWindow {
          #This window is just for interaction
          enabled FALSE
        }
        TiledClusterWindow {
          servers [ "display_leftSide_leftEye"
                    "display_leftSide_rightEye"
                    "display_rightSide_leftEye"
                    "display_rightSide_rightEye" ]
          tileWidth 1280
          tileHeight 1024
          stereo TRUE
          eyeSeparation 0.08
          zeroParallaxDistance 1
          columns 2
          rows 1
        }
      ]
    }
   ]
}
...
```

The role of a server in the servers list is well defined here. If *stereo* is set to TRUE, the first server (display_leftSide_leftEye) will render the left eye camera of the first display, second server (display_leftSide_rightEye) will render the right eye camera of the first display. With third and fourth servers it's the same but for the right side. In a grid of *m* columns and *n* rows of displays the first one is always lower left and the last one upper right. Additional servers are only used by load balancing if it is switched on.

The ClusterWindow approach is more flexible as you can setup view areas which can be stereo or not. You can try to setup the above scenario to get the same result on the displays using Cluster-Window , but you might not want to. A hint: You'll need a window with four displays fitting in and four viewports each modified by a ProjectionViewModifier . How this is done and where you will need this is discussed in the next section for setting up a CAVE environment.

Files:

- MultiDisplayLoadBalancing.wrl
- TiledDisplayLoadBalancing.wrl
- TiledDisplayStereo.wrl
- tie.wrl (test model)

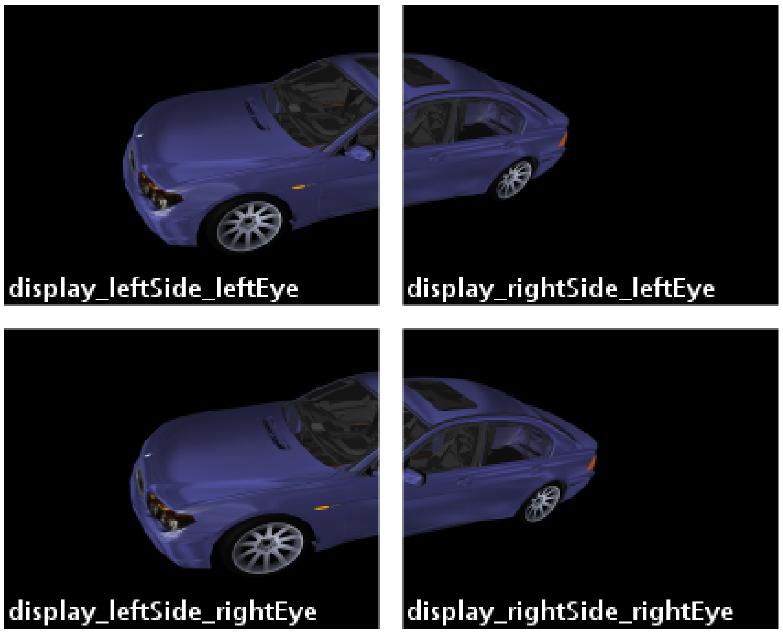

display_leftSide_leftEye
display_rightSide_leftEye
display_leftSide_rightEye
display_rightSide_rightEye

Figure 5.18: Tiled stereo setup with two displays and four PCs

Chapter 6

Scripting

6.1 Scripting: Java

This section shows you how to use Java in script nodes for making your scene dynamic.

6.1.1 Introduction

6.1.2 Setting up the Scene and the Script

Java script nodes are exactly looking like Ecmascript nodes in X3D. Only the “url” field is linking to a .class file instead of a .js file.

```
<Script DEF='javanode' directOutput='true' url='InstantJava.class'>
    <field name='set_touchtime' type='SFTime' accessType='inputOnly'/>
    <field name='get_newcolor' type='SFColor' accessType='outputOnly'/>
</Script>
```

The two fields of the script node are defining the incoming and outgoing values. *set_touchtime* will route an SFTime value into Java. *get_newcolor's* value will get filled by Java and routed on a *Material* node in the Scene.

6.1.2.1 Setting up a Java Class

Java Classes extend *vrml.node.Script* . Just like Ecmascripts Java Scripts have the common *initialize()* , *shutdown()* and *processEvent()* functions. This is an example how a basic Java Class looks like:

```
import vrml.*;
import vrml.field.*;
import vrml.node.*;
import vrml.Event;

public class InstantJava extends Script
{
    public void initialize()
    {
            System.out.println("initializing java..");
    }

    public void processEvent( Event e )
    {

    }

    public void shutdown()
    {
            System.out.println("bye!");
    }
}
```

6.1.2.2 Getting Values from the Scene

Incoming events are processed in the *processEvent()* function. In this example the *touchTime* field of a TouchSensor is routed on the script and catched by an if condition. The event has to be casted into the right type.

```
public void processEvent( Event e )
{
    if (e.getName().equals("set_touchtime"))
    {
            ConstSFTime time = (ConstSFTime)e.getValue();
            System.out.println( "touched at " + time.getValue());
    }
}
```

6.1.2.3 Writing back Values to the Scene

In order to send values to the scene we have to get the eventOut in the initialize() function and cast it into the right type. With the function *setValue(value)* we are sending the values to the script node's field in the scene.

```
public SFColor get_newcolor;

public void initialize()
{
    get_newcolor = (SFColor)getEventOut("get_newcolor");
    get_newcolor.setValue(1, 0.5, 0);
}
```

Files:

- InstantJava.x3d
- InstantJava.java
- InstantJava.class

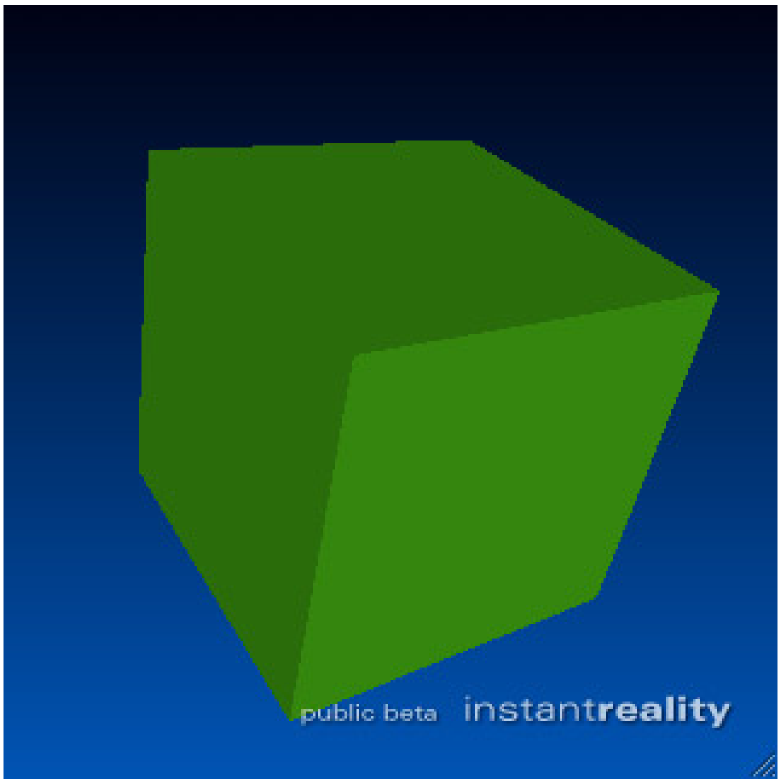


Figure 6.1: random color box

Chapter 7

Animation

7.1 Followers

This section shows you how to use damper and chaser nodes for animation.

7.1.1 Introduction

Followers divide in Dampers and Chasers. They are an easy to use alternative for common X3D interpolators. You only have to define a destination value and the duration of the interpolation in order to create a tween. Instant Reality provides the following dampers and chasers:

- ColorChaser
- ColorDamper
- CoordinateChaser
- CoordinateDamper
- OrientationChaser
- OrientationDamper
- PositionChaser2D
- PositionChaser3D
- PositionDamper2D
- PositionDamper3D
- ScalarChaser
- ScalarDamper
- TexCoordChaser
- TexCoordDamper

7.1.2 PositionChaser3D

This is an example about a PositionChaser3D that lets an object follow the mouse pointer. Other damper and chaser nodes follow the same logic.

Initially only the PositionChaser's *duration* - the time it takes to get to the *destination* value - has to be defined.

```
<PositionChaser3D DEF='pc' duration='5' />
```

The mouse pointer's position on a plane is recognized by a TouchSensor and routed to the PositionChaser3D's *destination* value. By routing the PositionChaser3D's *value_changed* field to a Transform's *set_translation* the objects seems to follow the mouse.

```
<PositionChaser DEF='pc' duration='5' />

<Transform DEF='trans_box'>
    <Shape>
            <Appearance>
                    <Material diffuseColor='0 0.329 0.706' />
            </Appearance>
            <Sphere radius='0.25'/>
    </Shape>
</Transform>

<Transform DEF='trans_plane' translation='0 -0.25 0'>
    <TouchSensor DEF='ts' />
    <Shape>
            <Appearance>
                    <Material diffuseColor='1 1 1' emissiveColor='1 1 1'/>
            </Appearance>
            <Box size='10 0.1 10'/>
    </Shape>
</Transform>

<ROUTE fromNode='ts' fromField='hitPoint_changed' toNode='pc' toField='set_destination'/>
<ROUTE fromNode='pc' fromField='value_changed' toNode='trans_box' toField='set_translation'/>
```

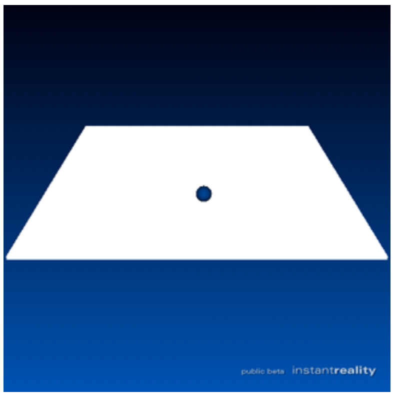


Figure 7.1: PositionChaser3D

Files:

- PositionChaser3D.x3d

7.2 Steering behaviour basics

This section shows you how to use steering behaviours to add some life to your world.

7.2.1 What are steering behaviours?

Citing Craig Reynolds GDC 1999 paper:

> Steering behaviours are a solution for one requirement of autonomous characters in animation and games: the ability to navigate around their world in a life-like and improvisational manner.

By combining predefined *behaviours* a variety of autonomous systems can be simulated. The basics of steering behaviours are described in Craig Reynolds paper and there are plenty of other resources out on the web (e.g. www.steeringbehaviors.de) - just google for “steering behaviours”. Make sure you have read and understood the basic principles (vehicles with behaviours) as the rest of this section focuses on how to use them with Avalon.

7.2.2 The steering sytem

We start with creating a SteeringSystem node and giving it a nice name:

```
<?xml version="1.0" encoding="UTF-8"?>
<X3D profile='Immersive'>
    <Scene>
            <SteeringSystem DEF='steerSystem'>
            </SteeringSystem>
    </Scene>
</X3D>
```

The parameters of the steering system will be discussed later. First we'll insert some vehicles into our system.

7.2.3 Adding vehicles

Within a steering system one or more vehicles represent autonomous agent(s) parameterized with behaviours. The vehicle class used in Avalon is based on a point-mass approximation which allows for a simple physically-based model (for example, a point mass has velocity (linear momentum) but no moment of inertia (rotational momentum)).

Adding a vehicle to the steering system looks like this:

```
<SteeringSystem DEF='steerSystem'>
    <SteeringVehicle DEF='vehicle1' maxSpeed='2' maxForce='4' />
    <SteeringVehicle DEF='vehicle2' maxSpeed='6' maxForce='12' />
</SteeringSystem>
```

Warning

A SteeringVehicle can only be a child of exactly one SteeringSystem at a time. Re- USEing a vehicle in another system is not supported and will lead to undefined results.

7.2.3.1 Parameterizing the vehicle

A vehicle has a few attributes, which can be read and set at any time:

mass the mass of the vehicle

radius the radius of the vehicle (used for obstacle and neighbour avoidance)

maxSpeed the maximum speed of the vehicle

maxForce the maximum force of the vehicle

There two additional inputOutput fields named `useFixedY` and `fixedY` . If `useFixedY` is `true` the value of `fixedY` is used as the y-component of the vehicles position. By setting a fixed value the vehicle can be constrained to stay on a fixed plane. If you dynamically route values to the field more interesting effects are possible.(e.g. terrain following). If `useFixedY` is `false` the value of `fixedY` is ignored.

7.2.4 Adding behaviours to the vehicles

After our steering system is equipped with some vehicles we need to add behaviour(s) to them, otherwise they won't do anything (which isn't very interesting). The available behaviours are:

SeekBehaviour A Seek behaviour acts to steer the character towards a specified position in global space.

FleeBehaviour Flee is the inverse of seek and acts to steer the character away from the target.

AvoidNeighborBehaviour Tries to keep characters which are moving in arbitrary directions from running into each other.

AvoidObstaclesBehaviour Gives a character the ability to avoid obstacles.

EvasionBehaviour Evasion is similar to Flee except that the menace (target) is another moving character.

PursuitBehaviour Pursuit is similar to Seek except that the quarry (target) is another moving character.

WanderBehaviour Wander is a type of random steering.

```
<SteeringSystem DEF='steerSystem'>

    <SteeringVehicle DEF='vehicle1' maxSpeed='2' maxForce='4' mass='1.4'>
        <SeekBehaviour DEF='seekBehaviour' factor='1.0' containerField='behaviours'  />
        <WanderBehaviour DEF='wanderBehaviour' factor='0.2' containerField='behaviours' />
    </SteeringVehicle>

    <SteeringVehicle DEF='vehicle2' maxSpeed='2' maxForce='4' mass='1.1' >
        <PursuitBehaviour DEF='pursuitBehaviour' containerField='behaviours' >
            <SteeringVehicle USE='vehicle1' containerField='quarry' />
        </PursuitBehaviour>
    </SteeringVehicle>

</SteeringSystem>
```

The example above shows a SteeringSystem that contains two vehicles. The vehicles contain different behaviours.

The first vehicle is called *vehicle1* and contains two behaviours: a SeekBehaviour and a WanderBehaviour . That way *vehicle1* is seeking the target while wandering a little (seek has a factor of 1.0 while wander has 0.1). The *target* of the SeekBehaviour is not specified explicitly so the default value (0,0,0) will be used. This results in *vehicle1* seeking around the origin.

The *vehicle2* only contains a PursuitBehaviour which is parameterized to pursue *vehicle1* . This results in *vehicle2* following *vehicle1* .

7.2.5 Updating the vehicles

In order to run the steering behaviour simulation the vehicle's `update` field has to be called contınously. This could be achieved by connecting a TimeSensor .time field to the `update` field. In practice the SteeringSystem does this job for you. Instead of connecting the TimeSensor to every single vehicle, you simply connect it to the SteeringSystem .time field which calls the update field on all it's child vehicles.

```
<TimeSensor DEF='timeSensor' loop='true' />
<ROUTE fromNode='timeSensor' fromField='time' toNode='steerSystem' toField='time' />
```

7.2.6 I don't see anything!?

Right. Until now we have setup a SteeringSystem , inserted some vehicles and added behaviours to them. But there is nothing to see!

That's because the SteeringSystem is a simulation node, that is: it has no visual output. All it does is simulating an autonomous behaviour by calculating a new position and orientation for every vehicle. So after each simulation step triggered by the TimeSensor connected to a SteeringSystem's *time* field the vehicles of the system contain a new position and orientation in their *translation* and *rotation* fields.

It's up to you what to do with these values. The most common practice is connecting the simulated position and rotation to a ComponentTransform which is the parent of a subgraph containing the geometry which visually represents the vehicle.

7.2.7 Moving some boxes

As an example we will add two boxes representing our vehicles.

```
<ComponentTransform DEF='trans1'>
    <Shape>
        <Box containerField='geometry' size='0.1 0.1 0.1' />
        <Appearance><Material diffuseColor='1 0 0' /></Appearance>
    </Shape>
</ComponentTransform>

<ComponentTransform DEF='trans2'>
    <Shape>
        <Box containerField='geometry' size='0.1 0.1 0.1' />
        <Appearance><Material diffuseColor='0 0 1' /></Appearance>
    </Shape>
</ComponentTransform>

<ROUTE fromNode='vehicle1' fromField='translation_changed' toNode='trans1' toField='translati
<ROUTE fromNode='vehicle2' fromField='translation_changed' toNode='trans2' toField='translati
<ROUTE fromNode='vehicle1' fromField='rotation_changed' toNode='trans1' toField='rotation' />
<ROUTE fromNode='vehicle2' fromField='rotation_changed' toNode='trans2' toField='rotation' />
```

7.2.8 Debugging vehicles and behaviours

A vehicle offers more attributes which are read-only and can be used for displaying the internal state of the vehicle in the case of debugging.

7.2.8.1 Vehicle's outputOnly fields

speed The current speed of the vehicle (which is the length of the velocity vector).

velocity The current velocity vector of the vehicle.

forward The current forward vector of the vehicle.

seekForce The current seek force of the vehicle (a null vector if no SeekBehaviour is used).

avoidObstaclesForce The current avoid obstacle force of the vehicle (a null vector if no AvoidObstacles behaviour is used).

Files:

- steeringBasics.x3d

7.3 Humanoid animation

This section shows how to animate virtual characters with H-Anim.

7.3.1 Overview

H-Anim figures are articulated 3D representations that depict animated characters. A single H-Anim figure is called a humanoid. While H-Anim figures are intended to represent human-like characters, they are a general concept that is not limited to human beings. Below two links on H-Anim are listed. The first one holds a good introducery overview on the concepts of H-Anim in general, and the second one contains the X3D specification for H-Anim nodes.

- Description of humanoid animation component
- X3D H-Anim component specification

Currently there exist two types of H-Anim figures: Skeletal body geometry describes the body as separate geometric pieces and therefore can lead to artifacts. Skinned body geometry in contrast regards the body as a continuous piece of geometry. Therefore all point and normal vector data sets are defined in one place, in the 'skinCoord' and 'skinNormal' fields of the HAnimHumanoid , for allowing smooth mesh animations. In this section only the latter, more natural looking type is described.

The 'skin' field of the HAnimHumanoid node contains the real mesh information, i.e. the Shape nodes, which define appearance and geometry of certain body parts like face or legs. As can be seen in the next code fragment, the Geometry's 'coord' and 'normal' fields only hold references to the Coordinates and Normals already defined in the 'skinCoord' and 'skinNormal' fields of the HAnimHumanoid. This way a seemless animation is achieved both for the vertices and the normals without the need to recalculate the latter.

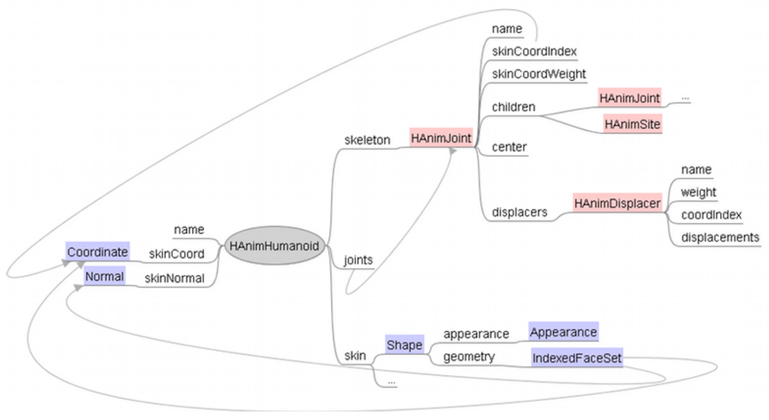


Figure 7.2: Overview of the HAnim component.

7.3.2 Animation

The HAnimJoint node is used to describe the articulations of the humanoid figure. Each articulation is represented by an HAnimJoint node. These joints are organized into a hierarchy of transformations that describes the parent-child relationship of joints of the skeleton and provides a container for information that is specific to each joint. This transformation hierarchy is listed in the 'skeleton' field of the HAnimHumanoid node. An additional field 'joints' holds references to all used HAnimJoint nodes.

An HAnimJoint has two fields that allow it to manipulate individual vertices defined within the skinCoord field of the HAnimHumanoid node. Incoming rotation or translation events of the joint affect the vertices indicated by the 'skinCoordIndex' field by a factor that is described by the corresponding values within the 'skinCoordWeight' field. The MFFloat field 'skinCoordWeight' contains a list of values that describe the amount of weighting to be used to affect the appropriate vertices, as indicated by the skinCoordIndex field, of the humanoid's 'skinCoord' and 'skinNormal' fields.

```
DEF HUMANOID HAnimHumanoid {
    name "Charles"
```

```
skeleton [
    DEF hanim_HumanoidRoot HAnimJoint {
        name "HumanoidRoot"
        center 0 .9723 -.0728
        skinCoordIndex [
            0 1 2 3 4 5 6 7 8 9 10 11
        ]
        skinCoordWeight [
            1 1 1 1 1 1 1 1 1 1 1 1
        ]
        children [
            DEF hanim_l_hip HAnimJoint {
                name "l_hip"
                center .0956 .9364 0
                skinCoordIndex [
                    #...
                ]
                skinCoordWeight [
                    #...
                ]
                children [
                    #...
                ]
            }
            DEF hanim_r_hip HAnimJoint {
                name "r_hip"
                #...
            }
            #...
        ]
    }
]
joints [
    USE hanim_HumanoidRoot
    USE hanim_r_hip
    USE hanim_l_hip
    #...
]
skinCoord DEF hanim_skin_coord Coordinate {
    point [
        #...
    ]
}
skinNormal DEF hanim_skin_normal Normal {
    vector [
        #...
    ]
}
skin [
    DEF faceShape Shape {
      appearance Appearance {
        texture ImageTexture {
          url "headTexture.jpg"
        }
      }
      geometry IndexedFaceSet {
        coord USE hanim_skin_coord
        normal USE hanim_skin_normal
```

```
                normalUpdateMode "none"
                coordIndex [
                    #...
                ]
                normalIndex [
                    #...
                ]
              }
            }
            #...
        ]
    }

    DEF TIMER TimeSensor {
        loop TRUE
        cycleInterval 5
    }

    DEF HUMANOIDROOT_POS_ANIMATOR PositionInterpolator {
        key []
        keyValue []
    }
    DEF HUMANOIDROOT_ANIMATOR OrientationInterpolator {
        key []
        keyValue []
    }
    DEF L_HIP_ANIMATOR OrientationInterpolator {
        key []
        keyValue []
    }
    DEF R_HIP_ANIMATOR OrientationInterpolator {
        key []
        keyValue []
    }
    #...

    ROUTE TIMER.fraction_changed TO HUMANOIDROOT_POSITION_ANIMATOR.set_fraction
    ROUTE TIMER.fraction_changed TO HUMANOIDROOT_ANIMATOR.set_fraction
    ROUTE TIMER.fraction_changed TO L_HIP_ANIMATOR.set_fraction
    ROUTE TIMER.fraction_changed TO R_HIP_ANIMATOR.set_fraction
    #...
    ROUTE HUMANOIDROOT_POS_ANIMATOR.value_changed TO hanim_HumanoidRoot.set_translation
    ROUTE HUMANOIDROOT_ANIMATOR.value_changed TO hanim_HumanoidRoot.set_rotation
    ROUTE L_HIP_ANIMATOR.value_changed TO hanim_l_hip.set_rotation
    ROUTE R_HIP_ANIMATOR.value_changed TO hanim_r_hip.set_rotation
    #...
```

The HAnimSegment node is a specialized grouping node that can only be defined as a child of an HAnimJoint node. It represents body parts of the humanoid figure and is organized in the skeletal hierarchy of the humanoid. The HAnimSite node can be used to define an attachment point for accessories such as jewelry and clothing on the one hand and an end effecter location for an inverse kinematics system on the other hand. Both nodes usually are not needed for skinned body animation.

The HAnimDisplacer nodes are usually used to control the shape of the face. Each HAnimDisplacer node specifies a location, called a morph target, that can be used to modify the displacement properties of the corresponding vertices defined by the 'coordIndex' field. The scalar magnitude of the displacement is given by the 'weight' field and can be dynamically driven by an interpolator or a script. The next code fragment shows an example. The mesh therefore can be morphed smoothly using the base mesh and a linear combination of all sets of displacement vectors, given by the MFVec3f

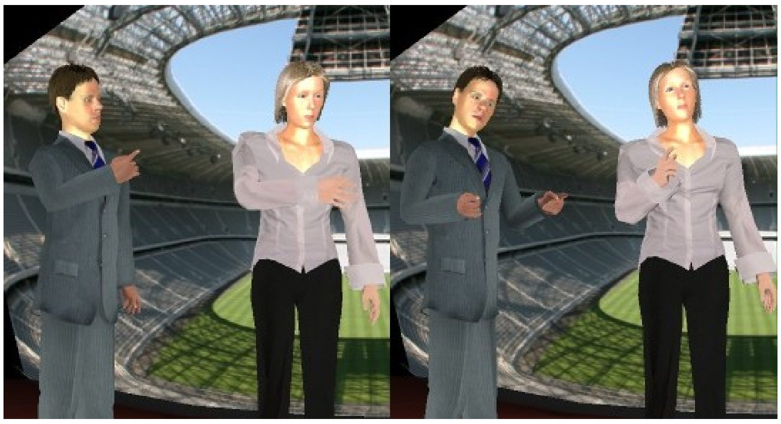


Figure 7.3: Talking and gesticulating virtual characters.

'displacements' field of the HAnimDisplacer nodes.

```
DEF Head HAnimJoint {
    name "Head"
    center 0 1.58 0.03
    skinCoordIndex [
        0  1  2  3  4  5  6  7  8  9  10 #...
    ]
    skinCoordWeight [
        1 1 1 1 1 1 1 1 1 1 #...
    ]
    displacers [
        DEF Phon_AShape HAnimDisplacer {
            name "Phon_AShape"
            weight 0.0
            coordIndex [
                0 1 2 3 4 5 6 7 8 9 10 #...
            ]
            displacements [
                0.000000 0.000000 0.000500,
                -0.002130 -0.002270 0.006110,
                #...
            ]
        }
        DEF Idle_Blink_bothShape HAnimDisplacer {
            #...
        }
    ]
}

DEF Timer TimeSensor {
    loop TRUE
    cycleInterval 5
}

DEF Interpol ScalarInterpolator {
    key [ 0.0, 0.25, 0.5, 0.75, 1.0 ]
    keyValue [ 0.0, 0.25, 0.5, 0.25, 0.0 ]
}

ROUTE Timer.fraction_changed TO Interpol.set_fraction
ROUTE Interpol.value_changed TO Idle_Blink_bothShape.weight
ROUTE Interpol.value_changed TO Phon_AShape.weight
```

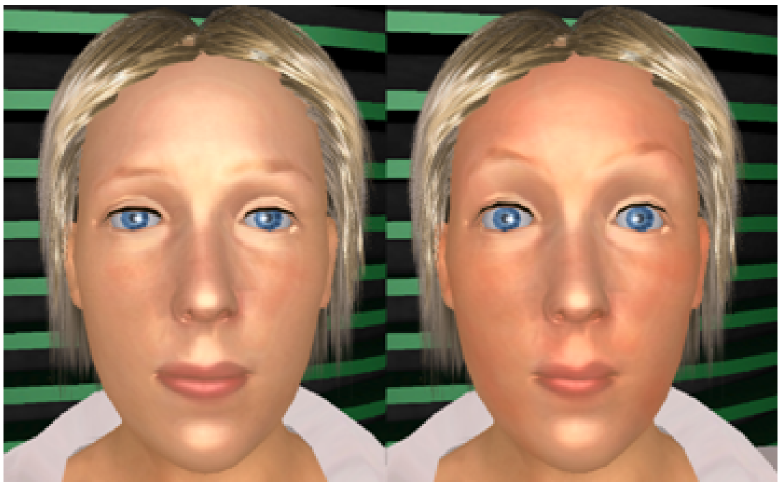


Figure 7.4: A woman getting a bit astonished...

7.3.3 Morphing

Quite similar to the already described Displacer node is the CoordinateMorpher node. Assume you want to animate a face, and you have given, say *n* , target states of your modelled face, a neutral one, and *n-1* other ones, e.g. a smiling one, one with open eyes, one with closed eyes, one with raised eyebrows, one saying 'a', and so on.

The Morpher node regards each of these states as a base vector of an n dimensional space spanning all possible combinations of point sets. In order to get valid linear combinations be careful that the coefficients (weights) of your data points (i.e. sets of expressions, which are also called morph targets) sum up to 1 (which is called a convex combination).

In the code fragment shown below we want to interpolate between a neutral state (the first one or 'keyValue' No 0 respectively) and state No 10. Therefore additionally a VectorInterpolator is needed. For each key time a vector of *n* keyValues is needed, defining the maximum weight for all morph targets (please note, that all lines sum up to 1). Another important thing to keep in mind, is that the sequence of points must not change, because they all belong to the same index field.

```
Shape {
    appearance Appearance {}
    geometry IndexedFaceSet {
        coord DEF coords Coordinate {
            point [
                0.086, 0.050, 0.431, 0.089, 0.044, 0.434,
                #...
            ]
        }
        coordIndex [
            0, 1, 2, -1, 3, 4, 5, -1,
            #...
        ]
    }
}

DEF morph CoordinateMorpher {
    keyValue [
        # 15 sets of coordinates; one set for each state:
            0.086, 0.050, 0.431, 0.089, 0.044, 0.434,
            #...
        ]
    }

    DEF vipol VectorInterpolator {
        key [ 0.0, 0.1, 0.75, 1.0 ]
        keyValue [
            1.0  0    0    0    0    0    0    0    0    0    0    0    0    0    0,
            0.4  0    0    0    0    0    0    0    0    0    0.6  0    0    0    0,
            0.6  0    0    0    0    0    0    0    0    0    0.4  0    0    0    0,
            1.0  0    0    0    0    0    0    0    0    0    0    0    0    0    0,
        ]
    }

    DEF ts TimeSensor {
        loop TRUE
        cycleInterval 5
    }

    ROUTE ts.fraction_changed TO vipol.set_fraction
    ROUTE vipol.value_changed TO morph.set_weights
    ROUTE morph.value_changed TO coords.set_point
```

The first of the attached files shows a simple but skinned walking character whereas the second file shows the morpher in action for doing simple facial animation.

Files:

- The famous boxman
- A morphed face

Chapter 8

Conclusion

The goal of this course was to demonstarte howeasy it is to go beyond simple Windows and Mouse/Pointer interaction. Using freely available tools and COTS hardware it is possible to create and interact with compelling Virtual and Augmented Environments without blowing the budget of a small lab or even an interested individual. So don't be a WIMP!

Go to the http://www.not-for-wimps.org to find the latest updates and links to further tutorals and code. Check out the http://www.instantreality.org and http://www.opensg.org projects to get more information and help to start your project.

Bibliography

[1] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. *IEEE Visualization 2001*, pages 21–28, October 2001. ISBN 0-7803-7200-x.

[2] Marc Alexa. Linear geometry interpolation in opensg, 2002.

[3] Marc Alexa and Johannes Behr. Cooperative VR enviroment. *brasil*, 2000.

[4] Marc Alexa and Johannes Behr. Volume Rendering in VRML. *Web3D - VRML 2001 Proceedings*, 2001.

[5] Marc Alexa and Johannes Behr. Fast and Effective Striping. *1. OpenSG Symposium OpenSG, 2002, Darmstadt*, 2002.

[6] Marc Alexa and Johannes Behr. Linear Geometry Interpolation in OpenSG. *1. OpenSG Symposium OpenSG*, 2002.

[7] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Computing and rendering point set surfaces.

[8] Marc Alexa, Johannes Behr, and Wolfgang Müller. The morph node. *Web3D - VRML 2000 Proceedings*, pages 29–34, 2000. ISBN 1-58113-211-5.

[9] Althoff, Stocker, and McGlaun. A Generic Approach for Interfacing VRML Browsers to Various Input Devices and Creating Customizable 3D Applications. *Web3D*, 2002.

[10] Y. Araki. A High-level Multi-user Extension Library For Interactive VRML Worlds. *VRML*, 1998.

[11] Avalon. Avalon. http://www.zgdv.de/avalon, 1998.

[12] J. Behr, SM. Choi, S. Großkopf, MH, and G. Sakas. Modelling, visualization, and interaction techniques for diagnosis and treatment planning in cardiology. *Computers & Graphics*, Vol 24.5:741–753, 2000. ISSN 0097-8493.

[13] Johannes Behr and Marc Alexa. Fast and effective striping. 1. OpenSG Symposium, Darmstadt, 2002.

[14] Johannes Behr and Patrik Dähne. AVALON: Ein komponentenorientiertes Rahmensystem für dynamische Mixed-Reality Anwendungen. *TUD thema Forschung*, 2003.

[15] Johannes Behr, Jorge A. Diz, and Marcelo G. Malheiros. An Extensible Interactive Image Synthesis Environment. *echnical report DCA-006/97 - DCA, FEEC, Unicamp*, 1997.

[16] Johannes Behr and Andreas Froehlich. AVALON, an Open VRML VR/AR system for Dynamic Application. *Topics*, 1(1):28, 1998.

[17] Johannes Behr, Torsten Froehlich, Christian Knoepfle, Bernd Lutz, Dirk Reiners, Frank Schoeffel, and Wolfram Kresse. The Digital Cathedral of Siena - Innovative Concepts for Interactive and Immersive Presentation of Cultural Heritage Sites. *ICCHIM Conference Proceedings, Milan*, 2001.

[18] Johannes Behr and Axel Hildebrand. Sanare – VR Med enviroment. *Topics*, 1998.

[19] Johannes Behr and Marc Niemann. Interactive Volume Data Rendering for Medical VR Applications. 1998.

[20] Johannes Behr, Choi Soo-Mi, and Stefan Großkopf. 3D Modellierung zur Diagnose und Behandlungsplanung in der Kardiologie. *Der Radiologe*, 40(3):256–261, 2000.

[21] Allan Bierbaum, Albert Baker, Carolina Cruz-Neira, Patrick Hartling, Christopher Just, and Kevin Meinert. VR Juggler: A Virtual Platform for Virtual Reality Application Development. Master's thesis, Iowa State University, 2000.

[22] Roland Blach, Juergen Landauer, Angela Roesch, and Andreas Simon. A flexible prototyping tool for 3d realtime user-interaction. 1998.

[23] Roland Blach, Jürgen Landauer, Angela Rösch, and Andreas Simon. A Highly FlexibleVirtual Reality System. *Future Generation Computer Systems*, 14(3–4):167–178, 1998.

[24] X3D Consortium. X3d standard. http://www.web3d.org/x3d/, 2008.

[25] Matthew Conway, Randy Pausch, Rich Gossweiler, and Tommy Burnette. Alice: A rapid prototyping system for building virtual environments. 2:295–296, April 1994.

[26] Carolina Cruz-Neira and Daniel J. Sandin. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. *ACM Computer Graphics, SIGGRAPH 93*, 1993.

[27] Paul J. Diefenbach, Daniel Hunt, and Prakash Mahesh. Building openworlds. *Web3D - VRML 1998 Proceedings*, 1998.

[28] N.I. Durlach and .A.S. Mavor. Virtual Reality: Scientific and Technological Challenges. *National Academy Press.*, 1995.

[29] Thorsten Fröhlich, Johannes Behr, and Peter Eschler. Cybernarium Days 2002 - A Public Experience of Virtual and Augmented Worlds. *First International Symposium on Cyber Worlds 2002*, 2002.

[30] Philippe Coiffet Grigore C. Burdea. *Designing Virtual Reality Systems: The Structured Approach*. Wiley-IEEE Press, 2003.

[31] H-Anim. ISO/IEC FCD 19774; Humanoid animation Specification. http://www.h-anim.org, 2001.

[32] Roger Hubbold, Jon Cook, Martin Keates, Simon Gibson, Toby Howard, Alan Murta, and Adrian West. Gnu/maverik a micro-kernel for large-scale virtual environments, 1999.

[33] John Kelso, Lance E. Arsenault, Ronald D. Kriz, and Steven G. Satterfield. DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments. *IEEE Virtual Reality Conference*, 2002.

[34] Gerard Kim. *Designing Virtual Reality Systems: The Structured Approach*. SpringerVerlag, 2005.

[35] Blair MacIntyre. A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment. *ISWC*, 1997.

[36] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. Technical Report TR94-023, 8, 1994.

[37] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 1994.

[38] OpenAL. OpenAL Specification and Documentation. *http://www.openal.org/*, 1999.

[39] OpenSceneGraph. OpenSceneGraph documenten. http://www.openscenegraph.org, 2003.

[40] Wayne Piekarski, Bruce Thomas, David Hepworth, Bernard Gunther, and Victor Demczuk. An architecture for outdoor wearable computers to support augmented reality and multimedia applications. 2000.

[41] Dirk Reiners. Opensg: Basic concepts.

[42] Dirk Reiners, Gerrit Voss, and Johannes Behr. A Multi-thread Safe Foundation for Scene Graphs and its Extension to Clusters. *Eurographics Workshop on Parallel Graphics and Visualisation 2002. Proceedings*, 2002.

[43] Dirk Reiners, Gerrit Voss, and Johannes Behr. OpenSG - Basic Concepts. *First OpenSG Symposium OpenSG, 2002, Darmstadt*, 2002.

[44] Patrick Reuter, Johannes Behr, and Marc Alexa. An improved adjacency data structure for efficient trianglestripping. *accepted for publication in the Journal of Graphics Tools*, To appear.

[45] J. Rohlf and J. Helman. IRIS Performer: A high performance toolkid for real-time 3D graphics. *ACM Computer Graphics, SIGGRAPH 94*, 1994.

[46] Mark Segal, Akeley Kurt, Chris Frazier, and Jon Leech. Opengl Specification. http://www.opengl.org/, 2003.

[47] William R. Sherman and Alan B. Craig. *Understanding Virtual Reality: Interface, Application, and Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[48] R. Stiles, S. Tewari, and M. Mehta. Adapting VRML For Free-form Immersed Manipulation. 1998.

[49] R. Stiles, S. Tewari, and M. Metha. Adapting VRML 2.0 for Immersive Use. *VRML 97, Second Symposium on the Virtual Reality Modeling language*, 1997.

[50] P.S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. *ACM Computer Graphics*, 1992.

[51] C. Szyperski. Component Software, Beyond Objekt-Oriented Programming. *ACM Press.*, 1998.

[52] H. Tramberend. Avocado – a distributed virtual environment framework. http://www.ercim.org/publication/Ercim_News/enw38/tramberen d.htm, 1999.

[53] John Vince. *Introduction to Virtual Reality*. SpringerVerlag, 2004.

[54] W3C. Xml Protocol Working Group, sOAP Version 1.2 Specification. *http://www.w3.org/2000/xp/Group/*, 2000.

[55] Wikipedia. Virtual reality. http://en.wikipedia.org/wiki/Virtual_reality, 2008.

[56] Wikipedia. Virtual reality. http://en.wikipedia.org/wiki/Augmented_reality, 2008.

[57] WorldToolKit. Sense8 Corporation; WorldToolKit: Virtual Reality Support Software. 4000 Bridgeway Suite 101, Sausalito, CA 94965, telephone : (415) 331-6318., 1994.

[58] S. Ting Wu and Johannes Behr. An Extensible Interactive Image Synthesis Environment. *XXIV Semish proceedings*, 1997.