H3D API Manual

For version 2.4

CONTENTS

CHAPTER 1

INTRODUCTION

First we would like to thank you for taking the time to download and open up this manual - most people would just skip right past it! The goal of this manual is to cover everything you need to know to download, compile and use H3D API to start developing hapto-visual applications as quickly as possible.

1.1 What is H3D API?

H3D API is an open-source, cross-platform, scene-graph API . H3D is written entirely in C++ and uses OpenGL for graphics rendering and HAPI for haptics rendering. HAPI is an open-source haptic API developed by the team behind H3D API. For more information about HAPI see the HAPI manual.

1.2 What makes H3D unique?

There are a lot of scene-graph APIs available today, and many of them are even open-source - so why should you use H3D API? Here are some of the features that make H3D a unique and powerful development tool for building 3D applications.

Standards

H3D is built using many industry standards including -

- X3D - http://www.web3d.org - the Extensible 3D file format that is the successor to the now outdated VRML standard. X3D, however, is more than just a file format - it is an ISO open standard scene-graph design that is easily extended to offer new functionality in a modular way.
- XML - http://www.w3.org/XML - Extensible Markup Language, XML is the standard markup language used in a wide variety of applications. The X3D file format is based on XML, and H3D comes with a full XML parser for loading scene-graph definitions.
- OpenGL - http://www.opengl.org - Open Graphics Library, the cross-language, cross-platform standard for 3D graphics. Today, all commercial graphics processors support OpenGL accelerated rendering and OpenGL rendering is available on nearly every known operating system.

- STL - The Standard Template Library is a large collection of C++ templates that support rapid development of highly efficient applications.

These industry standards mean that chances are you are already familiar with much of the design of H3D. It also means that much of what you will learn while starting to use H3D will apply to other development tools.

Cross Platform

H3D has been carefully designed to be a cross-platform API. The currently supported operating systems are Windows 7 and later, Linux and Mac OS X, though the open-source nature of H3D means that it can be easily ported to other operating systems.

Rapid Development

Unlike most other scene-graph APIs, H3D is designed chiefly to support a special rapid development process. By combining X3D, C++ and the scripting language Python, H3D offers you three ways of programming applications that gives you the best of both worlds - execution speed where performance is critical, and development speed where performance is less critical.

Using our unique blend of X3D, C++ and Python, you can cut development time by more than half when compared to using only C++.

Haptics

Reproducing the sense of touch in a computer simulation is still a relatively new technology and there are very few scene-graph based APIs that offer touch rendering. With our haptic extensions to X3D, H3D API is the ideal tool to begin writing hapto-visual applications that combine the sense of touch with vision. The binary release of H3D supports the following devices:

- Phantom Devices - Devices supported by OpenHaptics from 3D Systems. Visit https://www.3dsystems.com/haptics-devices/openhaptics for more information. Devices by this manufacturer are the only devices that will have haptic surface rendering when using OpenHaptics for haptic rendering.
- Force Dimension devices - Devices by Force Dimension. Visit http://www.forcedimension.com for more information.
- Novint Falcon - Low cost device targeted towards the gaming community. For more information about the device visit https://en.wikipedia.org/wiki/Novint_Technologies. Note that due to the manufacturer no longer existing making this device work on newer operating systems could be a hassle.

If your device is missing from this list then consider building a version of H3DAPI yourself. The source code might very well already support the device you desire with only minor tweaking or the correct library installed.

VR

H3D makes it easy to write applications that support stereo graphics rendering and require alignment between input devices and virtual environments. H3D is easily customised to work with a wide variety of VR display systems.

Open Source

The last but, we feel, the most important! Because H3D is open source, nothing is hidden from the developer. If you don't like the way we have implemented something - change it! If you don't understand how something works - look at the source!

1.3 Licensing

There are two versions of H3D API currently available, distributed under separate licensing schemes:

GNU GPL

The GNU General Public License version of the API is freely available for download at the h3dapi.org community website. The GPL license means that you can do whatever you want with the source code - as long as you also make your own work available under the GPL license.

Commercial

If you like the idea of having the source-code, but want to keep your own work private then you can use the commercial version of H3D. For a small licensing fee you can avoid the viral open-sourcing of the GPL license and develop closed-source applications using H3D. Please contact sales@sensegraphics.com for more information.

1.4 About the manual

It is the nature of open-source projects that documentation is usually produced as an after-thought and is often inadequate. We hope that this manual will be an exception! Rather than attempt to produce a work-

of-art when it comes to layout, design and use of the English language, we have prioritised usability and quality of information in the manual.

If you have any comments, feedback or complains regarding this manual, please visit the manual discussion forum on h3dapi.org.

It is for this reason that the H3D manual is itself a kind of open source development project - we will be releasing regular updates to the manual as it is developed and encouraging discussion (read: complaints) of the manual on our community web-site. Our primary goal is to give you the information you need to use H3D!

Layout

The manual is divided into three sections; Getting Started, Design and Development. The first section is aimed at getting you up and running with a usable H3D development environment as quickly as possible - we want you to be able to run our demos so you can see just how cool H3D really is! The Design section, we feel, is critical. Since H3D gives you a new way to develop it is important that we start you off on the right footing. Our design section gives you the right approach to really take advantage of the API. Finally the development section contains detailed information on how to use the various components of the API, from C++ through to X3D and Python, how to use what is there and how to create your own extensions to the API.

Good Luck!
- The SenseGraphics Development Team.

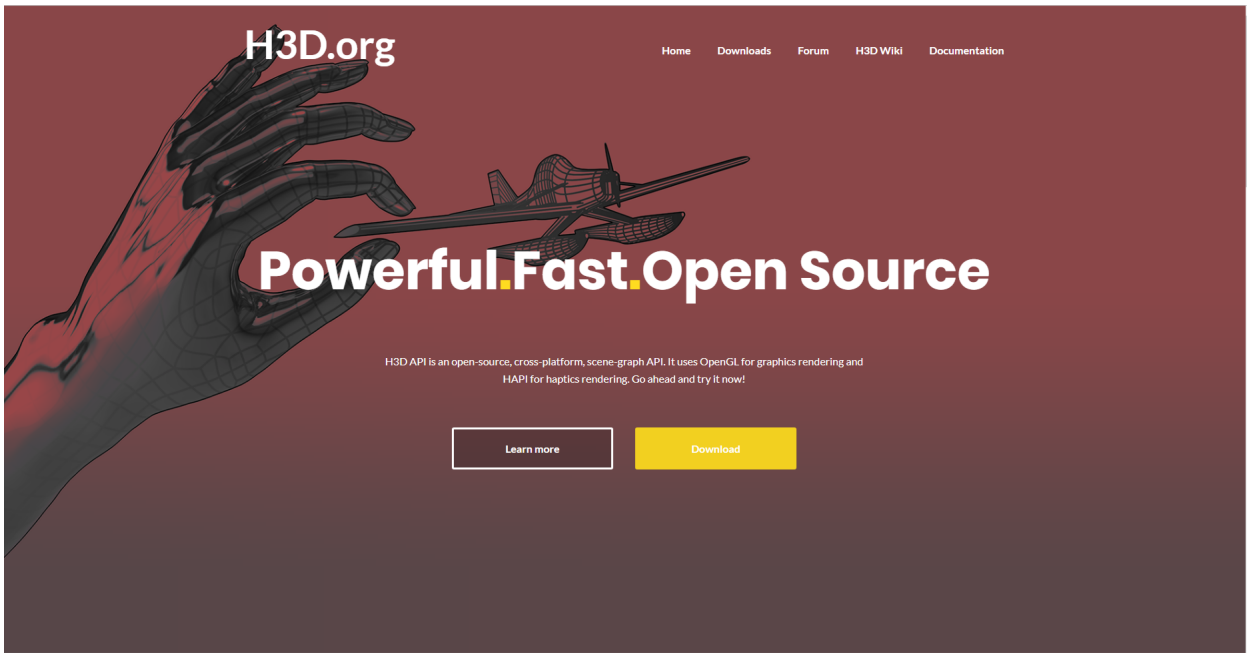
CHAPTER 2

GETTING STARTED

Because H3D is open-source, you won't be able to buy it from your local software retailer in a shiny box. To get started using H3D you will need to visit our community web-site, h3dapi.org, and download either the source code or a pre-compiled binary package. This section will take you through all options and steps involved in obtaining, building and using a H3D development environment.

2.1 H3DAPI.org

This is the H3D API community web-site, your first stop when you want to download the latest version or ask for help. Before you can post in the forum or edit the h3d wiki you need to create an account and login to h3dapi.org. Start by opening up your favorite web browser and going to the following url: https://h3dapi.org. You should see a page that looks roughly like this:



To register simply go to the forum page and click "Register" in the login section on the right side.

2.2 Downloading H3D

The downloads section on h3dapi.org will give you a list of all official release versions of H3D that are available for download as well as any toolkits distributed via the web-site. For each official release you can choose to download either the source code archive (compatible with all supported platforms) or a set of binary distributions files.

The official release versions of H3D are considered stable builds of the API, and are recommended for most users. If you want the "bleeding edge" development version of the API then you can download any of the nightly builds for the public ftp (ftp://www.h3dapi.org/pub/nightlybuilds/). In this location there are binary distribution files and source code archives that are updated nightly. Alternatively you can use Subversion to obtain the most up-to-date source code. Using the latest source or the nightly builds is only recommended for users who are comfortable building their own API binaries and/or can accept the occasional bug or two.

2.3 Subversion

Subversion is a source code revision management system - in short, it keeps track of every change you make to a project, when the change was made and why. This allows you, amongst other things, to backtrack various parts of the development or see who made a particular change and why.

To check out H3D using Subversion you will need a Subversion client. On Windows there is a Subversion GUI that integrates closely with the Explorer interface, Tortoise SVN. On other systems there are a number of alternatives, but we will assume you are using the command-line tools. To obtain Subversion, go to https://subversion.apache.org. Note: you will need a version of Subversion that supports SSL.

For Linux/ Mac OS X users, to check out the H3D source repository you will need to install Subversion and then run the following from a shell:

```
mkdir H3D
cd H3D
svn checkout https://svn.h3dapi.org/H3DAPI/metarepos/H3DCore/trunk/H3D
```

For Windows users, after installing TortoiseSVN, create a folder named H3D and checkout the source code with the following URL:

```
https://svn.h3dapi.org/H3DAPI/metarepos/H3DCoreWin/trunk/H3D/
```

This will check out the latest version from the subversion trunk using the WebDAV protocol. This is a read-only checkout and you will not be able to check in any changes. If you have any changes you would like to see applied to the H3D trunk you can contact us through the contact form on https://h3dapi.org.

The checkout includes H3DAPI, H3DUtil and HAPI (For both Linux and Windows) plus External (only for Windows). You therefore have a complete solution to build H3DAPI. From now on, whenever you update the checkout directory, all of them will be updated together.

2.4 Installing H3D API

2.4.1 CMake

This information is very important only when building H3DAPI from source. To generate projects for a system and compiler of choice a program called CMake need to be used. CMake can be found at http://www.cmake.org. After downloading and installing this program it should be used together with the CMakeLists.txt that exists in the build directory of H3D. CMake Version 2.8.7 or higher should be used.

To generate project files on Windows using the CMake GUI when H3D API is installed in its default directory simply put the path C:\H3D\build in the first textbox and the path C:\H3D\build\mysubdir in the second textbox. Click the Configure button and select your compiler. Wait until the configuration has finished then click the Generate button. The project files will now exist in "mysubdir". When configuring for Microsoft Visual Studio the file of interest in "mysubdir" will be a solution file (.sln). The build can now be started for the chosen compiler, how to do this is of course compiler dependent. If something seems to be wrong with the build during compilation then open up the CMake GUI again and check if any of the variables point to the wrong path. If H3D API is installed in any other directory than the default then change the above paths accordingly.

The CMake GUI can be used in a similar way for other operating systems.

To generate makefiles on Unix systems without a window environment simply navigate to the folder H3D/build. Create a new directory, that could for example be named "mybuild", navigate to the new folder and type "cmake ../" in the command line. All files automatically generated by CMake will be located in this new folder. If some changes to the variables used by CMake are desired type "ccmake ../" instead. This will start a terminal based text application.

See https://cmake.org/runningcmake/ for more instructions on how to run CMake.

2.4.2 Windows

The easiest way to install H3D API in Windows is to use the installer. It will install what you need and set all required environment variables for you. If you are using the source distribution directly you will have to manually set the following environment variables:

- H3D_ROOT - should be set to the path you have installed H3D API into, e.g. c:\H3D\H3DAPI

- H3D_EXTERNAL_ROOT - should be set to the directory containing the external third party libraries, e.g. c:\H3D\External

H3D required Visual Studio 2010 or a later version. The project files are generated by a program called CMake, see section 2.4.1 for more instructions.

2.4.3 Linux

H3D can be built from source on Linux using a Makefile and the make command. The makefile is generated by CMake. For more information about CMake see section 2.4.1. To build H3D API, go into the directory of the generated projects and type:

```
make
sudo make install
```

You also have to set the following environment variables in order for H3D API to work properly:

- H3D_ROOT - should be set to the path you have installed H3D API into, e.g. /home/daniel/H3D/H3DAPI

Without H3D_ROOT the settings file will not be found which will cause H3D API to run with default settings each time. If a user want to be able to choose which haptics device to use through the Settings-GUI.py program this environmental variable needs to be set.

For more step-by-step installation instructions for Ubuntu see the ReadMe file distributed with the H3D API source or the wiki on https://h3dapi.org.

2.4.4 Mac OS X

H3D currently needs to be built from source for Mac OS X. CMake is used to generate build files, see section 2.4.1 for more instructions. You will need to manually install the required and (if desired) optional external libraries yourself by downloading from the appropiate location and build from source. See section 2.4.5 for information about external libraries.

To test the binaries you should open a shell window and type:

```
cd H3DAPI/examples/CubeMap
../../H3DLoad CubeMap.x3d
```

If you built H3D with xerces support this should launch a H3D graphics window showing an animation of a cube mapped cylinder.

You also have to set the following environment variables in order for H3D API to work properly:

- H3D_ROOT - should be set to the path you have installed H3D API into.

Without H3D_ROOT the settings file will not be found which will cause H3D API to run with default settings each time. If a user want to be able to choose which haptics device to use through the Settings-GUI.py program this environmental variable needs to be set.

There is a project called macports (https://www.macports.org) which basically is a package handler that can be used on mac. H3D has distribution in macports which can be used to install H3DViewer. Note that macports is still building h3dapi from source but it does takes care of installing required libraries and downloading the needed source for h3dapi. Note that the public ftp at h3dapi.org contains macports files that can be used to obtain a nightly build of H3D.

2.4.5 Dependencies

H3D is by default dependent on the following third party libraries (with desired version number in parentheses), all of which are available under an open-source license model. Most of the following features can be easily disabled by editing H3DApi.h and undefining the appropriate setting. This will compile the API without that particular feature:

- Python (v2.7) - http://www.python.org

  Allow for the script language python to be used in conjunction with X3D.

- FreeImage (v3.17.0) - http://freeimage.sourceforge.net

  Used for reading image files in ImageTexture nodes.

- Xerces-c (v3.2.0) - http://xml.apache.org/xerces-c

  Used for parsing xml-files or strings containing xml-code.

- GLEW (v2.1.0) - http://glew.sourceforge.net

  This feature can not be disabled. Utility library for OpenGL.

- FreeType (v2.8.1) - http://www.freetype.org

  Used to render text. Needed by Text nodes.

- FTGL (v2.1.3_rc5) - https://sourceforge.net/projects/ftgl

  Used to render text. Needed by Text nodes.

- 3dxware(v2.0.4) http://www.3dconnexion.com

  Used to communicate with 3dxware devices, used by SpaceWareSensorNode.

- Cg (v3.1) - http://developer.nvidia.com/object/cg_toolkit.html

  Shading language for NVIDIA. Used for CG shading in shader nodes.

- OpenAL (v1.1) - http://www.openal.org

  OpenAL is used for sound in H3D API.

- libogg (v1.3.2) - http://xiph.org/downloads/

  Needed to support ogg vorbis sound files.

- libvorbis (v1.3.5) - http://xiph.org/downloads/

  Needed to support ogg vorbis sound files.

- libaudiofile (v0.3.6) - http://68k.org/ michael/audiofile/

  Needed to support a number of different sound files.

- libcurl (v7.55.1) - http://sourceforge.net/projects/curl/

  Used to support ftp and http protocols for finding files. Disabling this feature forces the user to only use local filenames.

- freeglut (v3.0.0) - http://freeglut.sourceforge.net/

  Used for window handling. Required if H3DLoad should be built.

- fparser (v4.5.2) - http://iki.fi/warp/FunctionParser/

  Used for classes in H3DAPI that need a simple way of evaluating functions. Edit HAPI.h to enable or disable support for this feature. Note: The files needed for this feature are distributed with HAPI.

- zlib (v1.2.11) - http://www.zlib.net/

  Required to parse zipped files.

- Teem (v1.11.0) - http://teem.sourceforge.net/

  Required for reading files in the Nrrd file format.

- DICOM Toolkit (v3.6.2) - http://dicom.offis.de/dcmtk

  Required for reading dicom files.

- libnifalcon - http://sourceforge.net/projects/libnifalcon/

  Required for Falcon support on linux.

- FFMpeg - http://ffmpeg.org/

  Required for the FFmpegDecoder class which is used for MovieTextures.

- CHAI3D (v2.0.0) - http://www.chai3d.org/

  Let Chai3D take care of haptics rendering (Chai3DRenderer class). Edit HAPI.h to enable or disable support for this feature.

- wxWidgets (v3.0.3) - http://wxwidgets.org/

  Needed by some example applications as well as H3DViewer. Not needed by H3DAPI itself.

These dependencies can be downloaded from the respective project websites given here. If using the Windows installer, all these libraries are included. Please note that, in most cases, downloading a newer version than that specified above should work, but is untested.

Libraries from the manufacturer needs to be obtained, in order to compile with support for haptics. The libraries are generally not open source and are as such distributed only with header files and library files. All features related to haptics can be found in HAPI.h.

- OpenHaptics - https://www.3dsystems.com/haptics-devices/openhaptics

  Needed to compile with support for devices from 3D Systems.

- DHD-API - http://www.forcedimension.com/

  Needed to compile with support from ForceDimension. Contact ForceDimension to get DHD-API. Note that an older version of DHD-API is included in the Windows distribution of H3D API.

Mac OS X: OpenAL is pre-installed on OS X so you will not need to install it manually. We highly recommend compiling the above dependencies as static libraries where possible. Python and Xerces-c must be compiled and installed as frameworks.

2.4.6 GLUT or Freeglut

The GLUT or Freeglut library might have to be manually compiled in order to allow C++ exception throwing from within GLUT. If this is the case on your system then Download the GLUT/Freeglut source code and modify the project settings to specify "-fexceptions" in the compiler flags. Note: this should note be necessary on Windows.

2.5 H3DLoad

H3DLoad is a simple loader for displaying X3D-files. To test your installation you can try to run any of the examples in the examples directory, e.g. if in the H3DAPI folder

```
H3DLoad.exe examples/x3dmodels/plane/bobcat.x3d
```

This should start a rendering of a simple toy airplane. If you get an error message saying "DLL not found" it means that you have specified the directories in PATH in a wrong way or have forgotten to add a directory to it. If you for example get a message about not finding hd.dll or hl.dll it means that you do not have OpenHaptics installed.

2.5.1 Configuring H3DLoad

Most configuration of H3DLoad can be done in the settings/h3dload.ini file. There is a GUI for changing this file that can be found in settings/SettingsGUI.py. You have to run this in Python with wxPython(www.wxPython.org) installed. In Windows you can use the settings/SettingsGUI.exe file. This allows you to select rendering options and default parameters for rendering.

You can override these settings by using environment variables if you want to. The following environment variables affects H3DLoad:

- H3D_ROOT - this variable has to be set to the directory of the H3D API installation in order for H3DLoad to find the h3dload.ini file. If changes in the h3dload.ini file have no effect you have probably forgotten to set this.
- H3D_FULLSCREEN - set to TRUE for fullscreen rendering, FALSE for window mode (default FALSE)
- H3D_MIRRORED - set to TRUE to flip the y-axis (default FALSE)
- H3D_RENDERMODE - determines stereo options for rendering. Most common choices are:
  - MONO - No stereo
  - QUAD_BUFFERED_STEREO - If you are using shutter-glasses.
  - VERTICAL_SPLIT_KEEP_RATIO - If you want to span two monitors using a side by side configuration.

  For all possible choices see the documentation for H3DWindowNode::RenderMode::Mode (default MONO)
- H3D_DEFAULT_DEVICEINFO - can be set to an x3d-file containing a DeviceInfo node specifying which haptics devices to use by default.
- H3D_DEFAULT_VIEWPOINT - can be set to an x3d-file containing a Viewpoint node specifying the viewpoint to use by default.
- H3D_DISPLAY - controls the directory in which to search for display setup files.
- H3D_STYLUS - can be set to an x3d-file which defines the look of the haptic stylus.
- H3D_GAMEMODE - set to TRUE to enable GLUT game mode.
- H3D_CONSOLE_OSTREAM - controls which output stream the Console is using. Valid values are cerr or cout.

2.5.2 Examples

A number of example programs are included in the H3D API distribution. Most of these examples use X3D and Python to define and manipulate the H3D scene-graph. Some of the examples show how to define and manipulate the H3D scene-graph using only C++.

2.6 H3DViewer

H3DViewer is an application for displaying x3d-files with haptic content using H3D API. When loading a file it uses the same default files as H3DLoad but gives the added feature of being able to change settings for haptic and graphic rendering while the scene is being rendered. At the same time H3DViewer works as any normal X3D-viewer which lets the user navigate the scene, change navigation type etc. Following is a description of H3DViewers menu system. H3DViewer is a much more mature viewer than H3DLoad and is extremely useful for developers when developing and debugging an H3D application.

- File - Let the user open and close files and also exit the viewer.
- Rendering - Contain menu choices for graphic and haptic rendering. Choosing "Settings" in this menu will open a dialog box which lets the user set a number of different properties of haptic and graphic rendering which might be used for debugging of a program or to speed up (or slow down) rendering.
- Viewpoints - Switch between viewpoints.
- Navigation - Choose between the allowed navigation types in the Scene.
- Advanced - Through this menu you can show the console, framerate of the application or a tree view. In the console warning messages and debug printouts will be displayed. The tree view is a very powerful tool and let the user see a complete scene graph tree and allows for modification of field values at run time. There are other features such as writing the current scene graph to file and more.
- Help - Currently there is no help section in H3DViewer. Copyright message is displayed when choosing About.

2.6.1 Configuring H3DViewer

Most configuration of H3DViewer can be done exactly as for H3DLoad by modifying the settings/h3dload.ini file see section 2.5.1. Additionally the menu options are persistent between runs although some of the options are only applied if the loaded x3d file does not explicitly set them (such as GlobalSettings).

The following environment variables affects H3DViewer in addition to the ones listed in 2.5.1:

- H3D_CONSOLE_LOGFILE - if TRUE and there is a log directory in the current directory at startup then the log from the Console will be written to files in this directory.
- H3D_CREATE_DBGDUMPFILE - if TRUE and there is an unhandled exception or crash there will be dump files written to a log directory.

Note that the environment variables H3D_GAMEMODE and H3D_CONSOLE_OSTREAM are not supported by H3DViewer.

CHAPTER 3

DESIGNING FOR H3D

H3D has a unique design that is powerful when used the right way. For this reason we have dedicated a section of the manual to design of H3D applications.

3.1 Introduction

X3D introduces a structural division in the scene-graph concept - the use of Nodes and Fields. Fields are data containers that know how to store and manipulate data properties. Nodes are essentially containers and managers of fields - all node functionality is implemented in terms of fields. Understanding when and how to use Nodes and Fields is essential in building an X3D based application.

There are three levels of programming for H3D - using C++, X3D or Python. X3D and Python are high level interfaces to the API, whilst C++ gives you somewhat raw access to the API. Each has appropriate strengths and weaknesses that are discussed below.

The SenseGraphics recommended design approach is to break the application up into generic functional units that can be encapsulated as fields and scene-graph nodes - this maximizes reusability and encourages good application design. Geometry and scene-graph structure for a particular application is then defined using X3D. Finally application / user-interface behavior is described using Python.

Spatial units in H3DAPI are by default meters, angles are given in radians and force is given in Newton.

3.2 Fields

Fields are the fundamental building blocks of X3D and thus H3D. In its simplest form, a Field is simply an event handling mechanism. Fields are arranged into a directed graph (called the field network) where events are passed from field to field according to how the field network has been constructed. Field connections are defined as routes and can be either a one-one, many-to-one or many-to-many connection. The field layer in H3D takes care of event handling and performs optimized lazy-evaluation of the field network where possible.

Fields have also been extended to be data containers that can store and pass data along the field network. Fields that are data containers can enforce strict typing to ensure to help minimize bugs in the field network. The default behavior of a data container field is to simply propagate the input data forward

to all outgoing connections, but a special update() function can be customized to perform any arbitrary operation on the data as it flows through the network.

Some fields use the update() function without being a data container, often to perform operations such as OpenGL rendering and display list cache management. In this manner all logic operations of a Node can be encapsulated as a series of fields.

3.3 Nodes

Nodes are the traditional building blocks of scene-graph APIs. In H3D nodes should be viewed as containers and managers for fields and field networks. In theory an entire application could be written in H3D using only the field network. However, the field network becomes rapidly unmanageable in anything other than a trivial application.

3.4 C++

The strength of C++ is that it is a compiled language that can be used to create highly efficient code. When writing haptic rendering algorithms or using OpenGL, C++ is the obvious choice. The main weakness of C++ is the relatively slow development time, stemming chiefly from the compile-test-debug cycle that means finding and fixing bugs can often take a long time.

Any function that can be considered time critical should be encapsulated as a C++ node or field in the API.

3.5 X3D

The X3D file format is used by H3D as an easy way to define geometry and arrange scene-graph elements such as a user interface. H3D has a full XML enabled X3D parser and automatically supports predefined X3D nodes as well as any new nodes developed for your application / toolkit.

Using X3D you can avoid long lines of unmanageable C++ scene-graph descriptions that are hard to read, maintain and debug.

3.6 Python

Python is an interpreted language meaning that you do not need to compile your python scripts to run them. One of the obvious benefits of Python is that you avoid the compile phase of the compile-test-debug cycle which allows for rapid prototyping of application functionality.

Python is ideal for defining fields that control simple non-time-critical behavioral properties such as managing a user-interface. Python can also be effective for prototyping complex time-critical fields that can be ported to C++ after prototyping.

Finally Python has an extensive library of extensions to simplify tasks such as file management, network communication and database access.

CHAPTER 4

PROGRAMMING WITH H3D

4.1 Fields

4.1.1 Introduction

Fields are the most fundamental building blocks of X3D and H3D API. Fields have several functions:

- Data storage - A field can contain data of some type (and most of the time does)
- Data dependency - Fields can be set up to be dependent on each other so that a change of value of one field triggers an update of another.
- Functional behavior - A field can calculate its value in any way it wants, depending on the fields routed to it.

Dependencies between fields in H3D API are specified by connecting them with something called routing. A route between field A and field B means that if something changes in field A an event message is sent to field B to let it know that A has changed and B can take appropriate actions. The most common fields are SFields and MFields. An SField is a field that contains a single value of some type and the field type is named depending on the type of the contained value. E.g. if it contains a bool value, it is named SFBool and if it contains a Vec3f it is named SFVec3f. MFields are the same but instead of just one value, it contains a vector of values of the same type. A route between two fields A and B of the same type will, in the default behavior, mean that if the value of field A changes the value of B will also change to the same value as in A. An example of this can be seen in:

```
<Group>
  <Transform translation="-0.1 0 0">
    <Shape>
      <Appearance>
        <Material DEF="MAT1" diffuseColor="1 0 0" />
      </Appearance>
      <Box size="0.1 0.1 0.1" />
    </Shape>
  </Transform>
  <Transform translation="0.1 0 0">
    <Shape>
```

```
      <Appearance>
        <Material DEF="MAT2" diffuseColor="0 1 0"/>
      </Appearance>
      <Sphere radius="0.05"/>
      </Shape>
  </Transform>
  <ROUTE fromNode="MAT1" fromField="diffuseColor"
         toNode="MAT2" toField="diffuseColor" />
</Group>
```

This is a simple x3d-scene with a box and a sphere. The initial values of the diffuseColor of the box is red and the sphere is green. However a route is set up from the diffuseColor of the box to the diffuseColor of the sphere. Therefore the resulting color of the sphere will be red. After this route has been set up any changes to the color of the box will also change the color of the sphere.

By default, routes can only be set up between fields of the same type.

4.1.2 Event propagation and lazy evaluation

The values of the fields are updated using lazy evaluation. This means that the value of the field will not be updated unless some part of the code asks for its value (with e.g. the getValue() function). When the value of the field A changes an event will be generated and send to all the fields it is routed to, to let it know that it has changed. The field (B) receiving an event will set a flag indicating that it has pending events and then pass on the event to all fields that it is routed to and so on. So events are propagated through the field network until it finds a field that has no routes out from it or a loop in the routing is detected. Nothing else will happen though. The value of B will remain unchanged, but the field now knows that its value is not up to date and will have to be updated if the value is asked for. This means that the value of A might change any number of times between two calls to getValue of B and all those values will be ignored (which is ok since at no time while field A had that value did we care about the value of B).

4.1.3 Specializing the update function

When the getValue() function is called (or any another function that requires the value to be up-to-date) the following happens:

- If the field is up to date, just use the current value.
- If not, an event has been received and we have to update the value.

The update() member function of the field takes care of updating of the value. By default it just copies the value of the incoming event, but it can be changed to do any arbitrary calculation by specializing the

update function yourself. This can be done from both Python and C++. The default update function for an SField looks something like:

```
class SFFloat: public Field {
  virtual void update() {
    value = static_cast< SFFloat* >(event.ptr)->getValue();
  }
};
```

The event member is a structure that contains a pointer to the field that caused the event and a time stamp with the time the event occurred. As you can see the value of the field is just copied. If we would like it to do something more complicated, like that the value of the field should be the sum of all SFFloat fields routed to it. This would be written as:

```
class FloatSum: public SFFloat {
  virtual void update() {
    value = 0;
    for( FieldVector::iterator i = routes_in.begin();
         i != routes_in.end();
         i++ ) {
      value += static_cast<SFFloat*>(*i)->getValue();
    }
  }
};
```

The routes_in member is a vector of all the fields that is routed to the field. There is also a similar member for the routes going out from the field, called routes_out.

This can also be done in Python, and would look like this:

```
class FloatSum( SFFloat ):
  def update( self, event ):
    routes_in = self.getRoutesIn()
    sum = 0
    for field in routes_in:
      sum += field.getValue()
    return sum
```

Note that the difference between C++ and Python is that in Python the event is given as an argument and in C++ it is a member variable. Also the update function in C++ sets the member variable “value” to the new value, while in Python the new value is returned.

4.1.4 Field template modifiers

There are some template modifiers that can be used to change the behavior of a field type in some way. See the "Online Reference Guide" for the available field template modifiers. Below follows a description of some of them.

AutoUpdate

Sometimes you do not want lazy evaluation, but instead want the update function to be called as soon as an event is received in order e.g. to perform an action based on a button press or if you want to print some field value. This can be done by specifying the field to be an AutoUpdate field. In C++ this would be done as:

```
class PrintInt32: public AutoUpdate< SFInt32 > {
  virtual void update() {
    SFInt32::update();
    cerr << value << endl;
  }
};
```

And in Python:

```
class PrintInt32( AutoUpdate( SFInt32 ) ):
  def update( self, event ):
    v = event.getValue()
    print v
    return v
```

So if you route to an instance of this class it would print the value of the field routed to it as soon as it changes. Be very careful when using the AutoUpdate field though since it might cause many unwanted update calls.

TypedField

In all the examples above the routes have to be set up between fields of the same types. Most of the time though you would probably want to route in several different types of fields and decide the value depending on them. This can be done by using the TypedField template modifier. With the TypedField modifier you can specify for each route which type it must have. If e.g. you have a field which value you

want to be the value of one of two different fields. Which one to choose depends on an SFBool field. In C++ you would specify the class as follows:

```
class MyField: public TypedField< SFFloat,
                                  Types< SFBool, SFFloat, SFFloat > >
                                  {
  virtual void update() {
    bool b = static_cast< SFBool * >( routes_in[0] )->getValue();
    H3DFloat f1 = static_cast< SFFloat * >( routes_in[1] )->getValue();
    H3DFloat f2 = static_cast< SFFloat * >( routes_in[2] )->getValue();

    if( b ) value = f1;
    else value = f2;
  }
};
```

The first argument given to the TypedField template is the base class that we want to use, i.e. if we specify SFFloat here the field will be an SFFloat and the value will be a H3DFloat. The second argument specifies the routes that are required in order for the field to work. There is also a third argument where you can specify optional fields but is is not used in this example. The second argument says that the first route must be of type SFBool and the second and third must be of type SFFloat.

In Python:

```
class MyField( TypedField( SFFloat,
                           (SFBool, SFFloat, SFFloat ) ) ):
  def update( self, event ):
    routes_in = getRoutesIn(self)
    b = routes_in[0].getValue()
    f1 = routes_in[1].getValue()
    f2 = routes_in[2].getValue()
    if b: return f1
    else: return f2
```

Example: The Sphere examples (section 5.1) controls the color of a sphere with the left mouse button using Python. If the button is pressed the sphere will be red, if it is not pressed it will be blue.

4.1.5 Field member functions

The following is not an extensive list of the functions that are available for fields, but lists the most useful ones. For the extensive list see the "Online Reference Guide". These functions are available in both

Python and C++.

For all fields:

- route(Field) - sets up a route from a field to another. An event is generated.
- routeNoEvent(Field) - same as route, but no event is generated.
- unroute(Field) - remove a route.
- touch() - manually generate an event from this field.

For SFields and MFields:

- getValue() - get the value of the field.
- setValue(value) - set the value of the field.

4.2 Nodes

Even though an entire application can be built using just fields and routes connecting them in one large field network, it would be very hard to get an overview over the application and manage it. That is where nodes come in. A Node is essentially a container for fields that groups fields together to create a larger reusable entities. They are also what is used to build the X3D scene-graph. A node collects all fields that are needed to control the behavior of what it is supposed to represent, e.g. if it is a sphere it has a field specifying the radius of it. The node specifies an interface to itself, determining what fields the user of the node can access and then hides away all the internal functionality from the user. A field in a node can be made to be one of four types:

- inputOnly - These fields are input to the node and can only be routed to and set. It is not possible to get the value or route from it.
- outputOnly - Output from the node. You can only route from it and get the value, not route to it and set.
- inititalizeOnly - The value can only be initialized. After the node has been initialized the value cannot be changed and works as an outputOnly field.
- inputOuput - No restrictions, can both get and set the value as well as route to and from.

and the node decides what type each of its fields will have.

4.2.1 Creating your own nodes

The first thing you have to do when designing a new node is to determine what fields should be available in the interface to the node. You then have to determine the dependencies between these fields and create an internal field network that does all the things you want the node to do.

Most nodes that do some kind of graphical rendering inherit from the interface class H3DDisplayListObject. This means that they will have an internal field called displayList. This field is responsible for creating an OpenGL display list for the graphic rendering of the node in order to speed up the rendering. This display list will only be created though if the fields routed to it have not generated an event. The display list will be created and used as long as no event is received, and when a new event is received the old display list will be removed and a new one will be created. This means that if you want a field to affect the graphics rendering, you will have to route it to the displayList field in order to make it aware of a change in the field. Otherwise it would just continue to use its old display list. Since display lists is a quite old OpenGL feature it is sometimes beneficial to simply shut off this behaviour. To do this use the GraphicsOptions node for a geometry. To do this for an entire application use the GlobalSettings node. It is however recommended that nodes that are used for graphical rendering should inherit from the interface class H3DDisplayObject regardless of the settings in the final application.

Important functions

There are two functions that are used for traversing the scene-graph. The scene-graph is traversed twice each loop. The first traverse is used to update various states and perform non graphical operations such as collecting haptic primitives for the haptic rendering. The next traverse is to do graphical rendering. The functions that will be called in the traversal are :

- traverseSG(TraverseInfo &ti) - called upon traversal of the scene-graph. The TraverseInfo object contains information about the coordinate space, current surfaces, haptics devices that are active etc.
- render() - makes all the OpenGL calls in order to graphically render the node.

Example: The Box node

In order to get a feeling for how to write a node we will go through the implementation of the Box node and explain the different parts. The first thing that is defined in the Box class in the Box.h is an internal field definition,

```
class SFBound: public X3DGeometryNode::SFBound {
  /// Update the bound from the size field.
  virtual void update() {
    Vec3f b = static_cast< SFVec3f * >( routes_in[0] )->getValue();
```

```
    BoxBound *bb = new BoxBound;
    bb->size->setValue( b );
    value = bb;
  }
};
```

This is a specialization of the SFBound class from X3DGeometryNode. All X3DGeometryNode objects inherit from H3DBoundedObject, which means that they have a bound field of type H3DBoundedObject::SFBound. This field contains a pointer to a Bound object, which specifies a bounding box for the node. In the case of the Box the bounding box is dependent on the size of the box so we create a new SFBound class that sets its value to a bounding box that encapsulates the box by using the size field to determine the size used for the bounding box. All new X3DGeometryNode classes need to define an SFBound field that calculates a bounding box for the geometry.

Further down you will find the definitions of the fields that are used in this node.

```
auto_ptr< SFVec3f >  size;
auto_ptr< SFBool  >  solid;
```

These are only fields that are available in C++. They are not by default part of the interface to the node. The interface to the node is determined by the H3DNodeDatabase object that comes next

```
static H3DNodeDatabase database;
```

The H3DNodeDatabase is a database over all the nodes that are available to use via an X3D-file. Each node has to register itself and the fields it wants to be accessible to the database. This is done in the initialization of the static database member in the Box.cpp file.

```
H3DNodeDatabase Box::database( "Box",
                               &(newInstance<Box>),
                               typeid( Box ),
                               &X3DGeometryNode::database );

namespace BoxInternals {
  FIELDDB_ELEMENT( Box, size, INPUT_OUTPUT );
  FIELDDB_ELEMENT( Box, solid, INPUT_OUTPUT );
}
```

The first argument to the constructor of the H3DNodeDatabase object is a string that determines what name the node should go by when using it in an X3D-file. The second argument is a pointer to a function that creates a new instance of the node in question, the third argument is the typeid of the C++ node class and the forth is an optional argument where the field interface from another H3DNodeDatabase object

can be inherited. Now we have to define which fields to be included in the interface to the node. This is easiest done using the FIELDDB_ELEMENT macro. Each FIELDDB_ELEMENT call adds a new field to the interface. The first argument is the Node class, the second is the field name, and the third is the access type for the field. So in this case we will add two fields to the interface of Box. The size field and the solid field. After doing this these fields are available from X3D. Now you just have to initialize the fields in the Box constructor:

```
Box::Box( Inst< SFNode  >  _metadata,
          Inst< SFBound >  _bound,
          Inst< SFVec3f >  _size,
          Inst< SFBool  >  _solid ):
  X3DGeometryNode( _metadata, _bound ),
  size     ( _size     ),
  solid    ( _solid    ) {

  type_name = "Box";
  database.initFields( this );

  size->setValue( Vec3f( 2, 2, 2 ) );
  solid->setValue( true );

  size->route( bound );

  size->route( displayList );
  solid->route( displayList );

}
```

In the constructor there are four things that you need to do:

- Set the type_name member in order to get better error messages.
- Initialize the fields of the nodes using the database.
- Set the default values of your fields.
- Set up internal routes.

This is all done in the example as you can see. The size field is routed to the bound field since the bound is dependent on the size and both the size and solid field is routed to the displayList field since they both affect the graphics rendering.

The render() function is not all that interesting. It just renders a box.

The traverseSG() function adds a new haptic shape to render if a surface has been specified and if haptics is enabled. The traverseSG() function of our Box example calls X3DGeometryNode::traverseSG() since the traverseSG() function of X3DGeometryNode has an implementation that is general for all geometries. X3DGeometryNode::traverseSG() always use a set of triangles and/or lines and/or points to specify the haptics shape to render.

```
void Box::traverseSG( TraverseInfo &ti ) {
  if( solid->getValue() ) {
    useBackFaceCulling( true );
  } else {
    useBackFaceCulling( false );
  }
  X3DGeometryNode::traverseSG( ti );
}
```

If the geometry is built up by many different triangles but the haptic shape used for the geometry is always the same and very simple the traverseSG() function could be specialized to get faster rendering. One example of this is the traverseSG function of Sphere. If you are interested in how this is done see Sphere.cpp in the source.

4.2.2 How to use the new nodes

When you have written your own nodes you would probably like to be able to use them in X3D-files. There are two ways of doing this. The first is to include your nodes in your executable that you use to run your program. E.g. if you are using the H3DLoad program that is distributed with the H3D API, you can add your new files to the H3DLoad project file (or Makefile). This will make them available when running H3DLoad. A more flexible way is to compile your nodes into a DLL or shared library. This is very useful if you have made a toolkit with nodes for a specific purpose, e.g. user interface widgets. Your nodes can be easily used by anyone that has H3D API installed. They just need to import your dll in the X3D-file and have the new widgets available.

```
<ImportLibrary library="c:\H3D\bin\UI.dll" />
```

4.3 Python

The Python scripting language can be used in H3D API by using the PythonScript node. This node has a url field where the path to the python script file you want to execute is specified. By adding a PythonScript node to the scene-graph you can get your python code to execute.

The H3DInterface Python module

All H3D specific types, like fields and types like Vec3f, Matrix3f, etc, are included in a Python module called H3DInterface. In order to get access to these types and functions from your python script you will have to import the module. The H3DInterface module is imported like any other python module.

```
# import the H3DInterface module. Types can then be accessed with the
# module name as prefix
import H3DInterface
v = H3DInterface.Vec3f()
```

or

```
# import all objects from the H3DInterface module into the current
# namespace. Types can be accessed directly without the module name prefix.
from H3DInterface import *
v = Vec3f()
```

Some of the functions and classes provided by the H3DInterface Python module are listed below. The list is not complete, for a full list see the doxygen documentation for the H3DInterface module.

Global fields

- time - Python access to Scene::time, which is the current time updated each scene-graph loop.
- eventSink - A Python field representation of Scene's eventSink field. Any field can be routed to eventSink and will thus have its value queried exactly once per scene-graph loop. This will cause the fields update function to be called once per scene-graph loop.

Functions

- **X3D creation functions** - each of the functions below returns a tuple where the first element is the node that has been created and the second argument is a dictionary with all the DEFined nodes in the string/url provided.
  - X3D/XML functions
    - createNode(node_name, name, kwargs)
    - createX3DFromURL(url)
    - createX3DFromString(string)
    - createX3DNodeFromURL(url)
    - createX3DNodeFromString(string)

  - X3D/VRML functions
    * createVRMLFromURL(url)
    * createVRMLFromString(string)
    * createVRMLNodeFromURL(url)
    * createVRMLNodeFromString(string)

- **X3D write functions** - each of the functions below returns a string containing the X3D syntax for the argument.
  - writeNodeAsX3D(node)
- **Bindable nodes access** - the following functions return the currently bound node for different bindable node stacks
  - getActiveDeviceInfo()
  - getActiveViewpoint()
  - getActiveNavigationInfo()
  - getActiveStereoInfo()
  - getActiveBackground()
  - getActiveFog()
  - getActiveGlobalSettings()
  - getActiveBindableNode(bindable) - The string argument *bindable* specifies the type of bindable node to return.
- **Scene instance access**
  - getCurrentScenes() - returns a list of all currently instantiated Scene instances.
- **Resource resolver** - access to the H3D API ResourceResolver
  - resolveURLAsFile(file)
- **Node access** - access to different nodes
  - getNrHapticsDevices()
  - getHapticsDevice(index)
  - getNamedNode(node_DEF_name)

Types

- **X3D field types** - The field types included are:
  - SFields - SFFloat, SFDouble, SFTime, SFInt32, SFVec2f, SFVec2d, SFVec3f, SFVec3d, SFVec4f, SFVec4d, SFBool, SFString, SFColor, SFColorRGBA, SFRotation, SFMatrix3f, SFMatrix3d, SFMatrix4f, SFMatrix4d, SFNode
  - MFields - MFFloat, MFDouble, MFTime, MFInt32, MFVec2f, MFVec2d, MFVec3f, MFVec3d, MFVec4f, MFVec4d, MFBool, MFString, MFColor, MFColorRGBA, MFRotation, MFMatrix3f, MFMatrix3d, MFMatrix4f, MFNode
- **Field functions** - the member functions available in all fields are:
  - route(field) - set up a new route
  - routeNoEvent(field) - set up a new route without generating an event
  - unroute(field) - remove a route
  - touch() - generate an event from the field
  - getRoutesIn() - returns a list of all fields routed to the field
  - getRoutesOut() - returns a list of all fields the field is routed to
- **SField specific functions** - the member functions for all SFields(i.e. SFFloat, SFDouble, etc) are:
  - setValue(value) - set the field to the new value
  - getValue() - returns the current value of the field
- **MField specific functions** - the member functions for all MFields(i.e. MFFloat, MFDouble, etc) are:
  - setValue(list) - set the field to the new value
  - getValue() - get the value of the field as a list
  - push_back(element) - add a new element to the end of the list of values
  - pop_back() - removes the last element
  - empty() - returns 1 if MField is empty, 0 otherwise
  - front() - returns the first element
  - back() - returns the last element
  - clear() - removes all elements
  - erase(element) - removes the first occurence of element
  - size() - returns the size of the MField
- **TypedField** - used for fields where you define the type of all the routes to it

- **AutoUpdate** - used if you want to add AutoUpdate capabilities to a field, i.e. make it update its value as soon as it receives an event
- **PeriodicUpdate** - used if you want to add PeriodicUpdate capabilities to a field, i.e. make it update its value once per scene-graph loop.

The H3DUtils Python module

H3DUtils is a Python module which contains helpful functions and classes to easily create specialized field classes which are useful when debugging. In order to get access to these classes from your python script you will have to import the module. The H3DUtils module is imported like any other python module.

The functions provided by the H3DUtils Python module are:

- **PrintFieldValue(base_class)** - Creates an instance of a field which prints the value of the fields routed to it whenever the value of those fields changes. The argument *base_class* should be any of the H3D field classes in python.
- **FieldValue2String(base_class)** - Creates an instance of a field which returns the value of fields routed to it as a string. The output type of the created field is SFString. The argument *base_class* should be any of the H3D field classes in python.
- **FieldValue2StringList(base_class)** - Creates an instance of a field which returns the value of fields routed to it as a list of strings. The output type of the created field is MFString. The argument *base_class* should be any of the H3D field classes in python.
- **FieldValue2Int(base_class)** - Creates an instance of a field which converts from the value of the fields routed to it to an integer. The output type of the created field is SFInt32. The argument *base_class* should be any of the H3D field classes in python.
- **SField2MField(sfield, mfield)** - Creates an instance of a field with input type sfield and output type mfield. The field classes should be of the same type, for example if *sfield = SFInt32* then *mfield = MFInt32*.

The classes provided by the H3DUtils Python module are:

- **TimerCallback** - The TimerCallback class is a field class in which you can set callback functions to be called at a later time that you specify. This is done through the function:
  - **addCallback(self, time, func, args)** - Where *time* is the time at which to call the callback function *func* that takes the arguments in *args*.

4.3.1 Specializing fields in Python

You can easily specialize the behaviour of any field type in Python, by inheriting from the base field type that you want to change, and define an update()-function to get the value you want. Following is an example of a specialized SFFloat field, where the value of the field is the average value of all fields routed to it.

```
class SFFloatAverage( SFFloat ):
  def update( self, event ):
    routes_in = self.getRoutesIn()
    sum = 0
    for field in routes_in:
      sum += field.getValue()
    return sum / len( routes_in )
```

The event argument given to the update-function is the field that was the last one to generate an event. This means that you can get the value of the field that generated the event by

```
v = event.getValue()
```

TypedField

In the previous example only fields of type SFFloat can be routed to the SFFloatAverage field. If you want the field to be dependent on values of different field types than the base type you will have to use TypedField. TypedField lets you specify the types of the input routes to a field. Here is an example that lets you route two SFVec3f fields to it and calculates the dot product between them as its value.

```
class DotProduct( TypedField( SFFloat, (SFVec3f, SFVec3f) ) ):
  def update( self, event ):
    routes_in = self.getRoutesIn()
    return( routes_in[0].getValue() * routes_in[1].getValue() )
```

The first argument to TypedField is the base field class to use. In this case we want the field to be an SFFloat field. The second argument is a tuple with field types specifying what type each of the fields routed to it should have. In this case we want two fields where both are of the type SFVec3f.

AutoUpdate

AutoUpdate can be used to disable lazy evaluation and use greedy evaluation for a field instead, i.e. the update()-function will always execute as soon as an event is received. This can e.g. be useful if you just

want to print the value of a field each time it changes. The following example prints an SFFloat value.

```
class PrintSFFloat( AutoUpdate( SFFloat ) ):
  def update( self, event ):
    v = event.getValue()
    print v
    return v
```

PeriodicUpdate

PeriodicUpdate is something in between lazy and greedy evaluation. Periodic update will update all fields routed to it once per scene-graph loop. This can be used for example to print frame rate of the scene.

References to X3D nodes

The PythonScript node has a field called references, which can be used to make nodes specified in an X3D-file available to the PythonScript. In the example below the Group node called "GROUP" is specified in the references field and can then be accessed directly in the python file.

X3D code:

```
<Group>
  <Group DEF="GROUP" />
  <PythonScript url="spheres.py">
    <Group USE="GROUP" containerField="references"/>
  </PythonScript>
</Group>
```

Python code:

```
group = references.getValue()[0]
```

The reference in the python script can then be used to change the fields of the group or anything else you might want to do with it.

Creating X3D nodes from Python

There are four functions available to create X3D-nodes using X3D syntax in python. They are:

- g, dn = createX3DFromString(string)

- n, dn = createX3DNodeFromString(string)
- g, dn = createX3DFromURL(url)
- n. dn = createX3DNodeFromURL(url)

The *FromString functions take a string in X3D/XML format while the FromURL versions accepts a url/filename to a x3d-file.

The difference between the createX3DFrom* and createX3DNode* functions is that the createX3DFrom* version will put everything in a group and return that while the createX3DNode* version will give you the node directly. E.g. createX3DFromString(“”) will give you a Group node containing a Shape node (if “” defines a Shape node), while createX3DNodeFromString(“”) will give you the Shape node itself.

All of the functions return a tuple where the first element is the node created as described above. The second element is a dictionary over the DEF-names and their corresponding nodes as specified in the X3D input to the functions. E.g. after creating a Shape node specifying a sphere the components of the Shape can be accessed as shown below:

```
shape, dn = createX3DNodeFromString( \
"""<Shape>
       <Appearance>
          <Material DEF="MATERIAL" />
       </Appearance>
       <Sphere DEF="SPHERE" />
     </Shape> """ )

material = dn["MATERIAL"]
sphere = dn["SPHERE"]
```

The triple quotes(""") used to specify the string is a handy way to specify a string in in Python when the string itself contains quotes and it spans over several lines. Everything between the """ will be interpreted as a string and you do not need to use escape codes for quotation marks.

Apart from using the createX3D* functions to create a node hierarchy it is also possible to use the function createNode to create only one node complete with a name and initial field values.

Special functions in PythonScript python files

There are two special functions that can be defined in a python file used by a PythonScript node that has a special meaning. They are:

- initialize()
- traverseSG()

These functions (if defined) work just as the C++ implementation of the initialize() and traverseSG()-functions in a Node and are actually called from the PythonScript nodes implementation of these function. The initialize() function is called once upon the first reference of the PythonScript node. It will never be called again. The traverseSG() function will be called during the traversal of the scene-graph when encountering the PythonScript node and will hence be called once each scene-graph loop if the PythonScript node is part of the current scene-graph. E.g. if you specify the following in you python file, "Init" will be printed first and then "Traverse" will be printed each loop.

```
def initialize():
  print "Init"

def traverseSG():
  print "Traverse"
```

Accessing Python fields from X3D

Fields that are specified in python code in a PythonScript node can be accessed directly from X3D in order to set up routes from and to them. E.g. if you specify the following in a file named python.py

```
position = SFVec3f()
```

you can access it as

```
<Group>
  <Transform DEF="TRANSFORM" />
  <PythonScript DEF="PS" url="python.py" />
  <ROUTE fromNode="PS" fromField="position"
         toNode="TRANSFORM" toField="translation" />
</Group>
```

Access between PythonScript nodes

Each PythonScript node will create its own Python module in which all its code will be executed. Therefore classes and variables can be imported from other PythonScript instances and be used there. The moduleName field can be used to name the python module. E.g. if you have two PythonScript nodes as below:

```
<Group>
  <PythonScript moduleName="PS1" url="f.py" />
  <PythonScript moduleName="PS2" url="functions.py" />
```

```
</Group>
```

you can access classes in PS1 from PS2 with the following:

```
import PS1
i = PS1.ClassName()
```

4.4 Haptics in H3D API

This section describes how to use haptics in H3D API. In order to be able to touch an object, you will have to do two things:

- Specify the haptics devices to use.
- For each shape that should be touchable, set its haptic properties.

This section also contains information about customizing the haptics in H3DAPI. Generally this means implementing a class in HAPI and then write a node interface for this class in H3DAPI. Take care so that classes inheriting from classes in HAPI never access the field interface of H3DAPI directly since this will most likely cause thread problems.

4.4.1 Specifying a haptics device

If you are using H3DLoad, you can specify the haptics device to use via the "H3DLoad settings" GUI and do not have to make any changes to your x3d-files. The GUI will let you easily choose what kind of haptics device and properties of the haptics device you have. However if you do not use H3DLoad or want to specify the haptics device manually you will have to create an instance of the bindable node DeviceInfo and add the haptics devices to use to it. E.g.

```
<DeviceInfo>
  <PhantomDevice positionCalibration="2 0 0 0
                                      0 2 0 0
                                      0 0 2 0
                                      0 0 0 1" >
    <OpenHapticsRenderer/>
    <Shape containerField="stylus">
      <Appearance> <Material/> </Appearance>
      <Sphere radius="0.005" />
    </Shape>
  </PhantomDevice>
```

```
</DeviceInfo>
```

The PhantomDevice is the node to use when having a device from 3D Systems. In the example above we also specify the "stylus" of the haptics device, which is the graphical representation of the device. This can be changed to any representation you want. In this case we have chosen a sphere. Another thing worth noticing is the positionCalibration field. It specifies a transformation matrix from the haptics device local space to world coordinates in H3D API space. By changing this you can specify how a position of the haptics device should translate to a position in your world. There is also a similar field for the calibration of the orientation of the pen of the haptics device called orientationCalibration. To get haptics rendering with this device a haptic renderer is specified. For full documentation of all the fields available in PhantomDevice, see the "Online Reference Guide".

4.4.2 Multiple haptics devices

In order to use multiple haptics devices you will have to specify several H3DHapticsDevice instances in the DeviceInfo node. In this example we only use PhantomDevice but you could as well combine PhantomDevice with ForceDimensionDevice if that is how your system looks.

```
<DeviceInfo>
  <PhantomDevice deviceName="Phantom1"
                 positionCalibration="1 0 0 0
                                      0 1 0 0
                                      0 0 1 0
                                      0 0 0 1" >
    <OpenHapticsRenderer/>
    <Shape DEF="STYLUS" containerField="stylus">
      <Appearance> <Material/> </Appearance>
      <Sphere radius="0.005" />
    </Shape>
  </PhantomDevice>

  <PhantomDevice deviceName="Phantom2"
                 positionCalibration="1 0 0 0
                                      0 1 0 0
                                      0 0 1 0
                                      0 0 0 1" >
    <RuspiniRenderer/>
    <Shape USE="STYLUS" containerField="stylus"/>
  </PhantomDevice>
</DeviceInfo>
```

As you can see we use the deviceName field to specify which Phantom device to use. The name specified should match the name set up for the device in the "Phantom Configuration" tool distributed with the Phantom device drivers. Also notice that each device has its own renderer specified in the setup.

4.4.3 Accessing the haptics device

You might want to access fields in the currently active haptics device in order to, for example, route from its fields. It is a little different depending on at what programming level you want to access it.

X3D

In X3D you can get a reference to the currently active haptics device by importing it from the H3D_EXPORTS definitions. H3D_EXPORTS contains the current haptics device as the name "HDEV". To import the name into your X3D-file you can write

```
<IMPORT inlineDEF='H3D_EXPORTS' exportedDEF='HDEV' AS='HDEV' />
```

After this statement the "HDEV" name can be used to refer to the haptics device. In case of several devices in the scene "HDEV0", "HDEV1" and "HDEV2" can be used instead of "HDEV" to access the first, second and third device according to how they are specified in the DeviceInfo node.

Python

Since the DeviceInfo node is a X3DBindableNode you can get the currently active node using the getActiveDeviceInfo function. All current devices is in the "device" field of the DeviceInfo node.

```
di = getActiveDeviceInfo()
if( di ):
  devices = di.device.getValue()
  if( len( devices ) > 0 ):
    hdev = devices[0]
```

If only one haptics device is of interest the getHapticsDevice function can be used instead.

```
hdev = getHapticsDevice(0) # hdev will be None if there is no connected haptics device
```

C++

Similar to Python, you can get the currently active DeviceInfo node.

```
DeviceInfo *di = DeviceInfo::getActive();
H3DHapticsDevice *hdev = NULL;

if( di && di->device->size() > 0 ) {
  hdev = di->device->getValueByIndex(0)
}
```

4.4.4 Implementing custom haptics device nodes

To add support for a new haptics device first subclass HAPIHapticsDevice. For information about how to do this see the HAPI manual. When this new HAPI device class exists it is time to subclass H3DHapticsDevice. Start by creating a constructor, fields and the database interface for this new device node. To make this new node a functional device node add an instance of the class created in HAPI to the variable "hapi_device". This is usually done by overriding the *initialize* function of H3DHapticsDevice. Lets look at how this is done for the FalconDevice node:

```
void FalconDevice::initialize() {
  H3DHapticsDevice::initialize();
  hapi_device.reset( new HAPI::FalconHapticsDevice( deviceName->getValue() ) );
}
```

Depending on how the haptics device works it might be useful to override other virtual functions of H3DHapticsDevice. If for example any of the fields of the new device node needs information about the initialized device the *initDevice* function could be overridden. If any of the fields need to set properties of the contained HAPI device then specialize the fields. Examples of overriding some virtual functions and specializing fields for haptics device nodes can be seen by examining the source code for the ForceDimensionDevice node.

4.4.5 Surface properties

In order to be able to touch a Shape, you will have to add a surface node with haptic properties to its' Appearance. At the moment there are five kinds of surface nodes to choose from:

- SmoothSurface - a surface without friction
- FrictionalSurface - a surface with friction
- MagneticSurface - makes the shape magnetic, only works with OpenHapticsRenderer.
- OpenHapticsSurface - specify properties as when using OpenHaptics, only works with OpenHapticsRenderer.

- DepthMapSurface - a surface with friction in which a texture decides how deep the surface feels.
- HapticTexturesSurface - a surface with friction in which a texture modifies the value of some of the parameters depending on where the shape is touched and how the texture coordinates are specified for the shape.
- MultiDeviceSurface - a surface which allows you to specify a different surface for each haptics device in a multi device setup.

For full documentation of all fields available see the "Online Reference Guide".

An example to get a touchable sphere:

```
<Shape>
  <Appearance>
    <Material/>
    <SmoothSurface stiffness="0.3" />
  </Appearance>
  <Sphere radius="0.05" />
</Shape>
```

4.4.6 Implementing custom surfaces

Sometimes the existing surfaces are not enough to get the desired haptic behaviour and a custom made surface node has to be implemented. Provided that a custom made class inheriting from HAPISurfaceObject exists all that has to be done is to subclass H3DSurfaceNode. For information on how to subclass HAPISurfaceObject see the manual for HAPI. To subclass H3DSurfaceNode start by creating a constructor, fields and the database interface just as for any other node. To make this new node a functional surface node add an instance of the class created in HAPI to the variable "hapi_surface". This is usually done by overriding the *initialize* function of H3DSurfaceNode. Lets look at how this is done for the SmoothSurface node:

```
void SmoothSurface::initialize() {
  H3DSurfaceNode::initialize();
  hapi_surface.reset(
    new HAPI::FrictionSurface( stiffness->getValue(),
                               damping->getValue() ) );
}
```

If it is desired to be possible to change the properties of the surface at run time from the X3D/python level an interface from the field corresponding to a value in the subclassed HAPISurfaceObject has to be implemented. This could be done by specializing the field so that it changes the properties in

the subclassed HAPISurfaceObject when its own value changes. As an example lets look at how the "stiffness" field in SmoothSurface is implemented. First comes the declaration in the header file:

```
// Specialized field class
class H3DAPI_API UpdateStiffness:
      public AutoUpdate< OnValueChangeSField< SFFloat > > {
    protected:
      virtual void onValueChange( const H3DFloat &v );
};

// Field declaration
auto_ptr< UpdateStiffness > stiffness;
```

Then the definition in the source file:

```
void SmoothSurface::UpdateStiffness::onValueChange( const H3DFloat &v ) {
  SmoothSurface *ss =
    static_cast< SmoothSurface * >( getOwner() );
  if( ss->hapi_surface.get() ) {
    static_cast< HAPI::FrictionSurface * >( ss->hapi_surface.get() )
      ->stiffness = v;
  }
}
```

4.4.7 Force effects

In addition to having touchable shapes, you can specify free space force effects. A force effect is more or less a function from the position/orientation of the haptics device to a force/torque. The force effects available in H3D API are:

- ForceField - specify a constant force to send to the haptics device
- MagneticGeometryEffect - is similar in behaviour to magnetic surface. It is a haptic spring which can be used to hold the haptics device at a given surface.
- PositionFunctionEffect - three functions decides the force output with input being the position of the haptics device.
- RotationalSpringEffect - a torque spring that can be used to hold the device at a certain axis. Only for devices that support torque.
- SpringEffect - a haptic spring that can be used to hold the haptics device at a given position.

- TimeFunctionEffect - three functions decides the force oupput with input being system time.
- ViscosityEffect - specifies a force in the opposite direction of the movement of the haptics device.

For full documentation of all fields available see the "Online Reference Guide".

An example to get a constant force on the haptics device:

```
<ForceField force="0 1 0" />
```

4.4.8 Implementing custom force effects

It is also possible to implement custom force effects easily. To do this one has to create a class that inherits from HAPIForceEffect and implement the calculateForces method. Lets look at a simplified version of HapticSpring

```
class HapticSpring: public HAPIForceEffect {
public:
  /// Constructor
  HapticSpring::HapticSpring( const Vec3 &_position,
                              HAPIFloat _spring_constant ):
                              position( _position ),
                              spring_constant( _spring_constant ) {}

  /// The force of the EffectOutput will be a force from the position of
  /// the haptics device to the position of the HapticSpring.
  EffectOutput virtual calculateForces( const EffectInput &input ) {
    force = ( position - input.hd->getPosition() ) * spring_constant;
    return EffectOutput( force );
  }

protected:
  Vec3 force;
  Vec3 position;
  HAPIFloat spring_constant;
};
```

It implements a simple spring force centered at "position".

The inputs to the calculateForces functions is a struct called EffectInput which contains:

- deltaT - The change in time since the last haptics loop.

- hd - The haptics device for the device that created the EffectInput. This is a pointer to a HAPIHapticsDevice and through this pointer functions can be called to get properties of the haptics device, such as position and velocity.

The EffectOutput structure to return contain:

- force - the force to render
- torque - the torque to render

All values are in global coordinates.

The calculateForces function will be called from the haptic loop at 1000 Hz and hence you should try to keep them as simple as possible. Also you should not access the field network at all.

Once you have created your new force effect and want to use it, you will have to add an instance of it to the TraverseInfo object in the traverseSG function in a Node. Here is an example from ForceField,

```
/// Adds a HapticForceField effect to the TraverseInfo.
virtual void traverseSG( TraverseInfo &ti ) {
  if( ti.hapticsEnabled() ) {
    ti.addForceEffectToAll( new HAPI::HapticForceField(
                                ti.getAccForwardMatrix().getRotationPart() *
                                force->getValue() ) );
  }
}
```

The addForceEffectsToAll function is used to add the force effect to all available haptics devices. You can also use the addForceEffect function to just add it to a specified haptics device if you have several.

Take a look at the source code for ForceField and SpringEffect for a full example of how to specify a force effect.

4.5 Tips and Tricks

This section contains general programming tips and tricks when developing with H3DAPI.

4.5.1 Useful nodes

This section contains information about nodes which might be useful when building applications.

Options nodes

H3DAPI has a way of setting global option parameters that affects rendering (graphics and haptics) and more at run-time. These options can be changed at any time. Here follows a list of the existing options nodes:

- CollisionOptions
- DebugOptions
- DefaultAppearance
- GeometryBoundTreeOptions
- GraphicsOptions
- HapticsOptions
- OpenHapticsOptions

All of the options node can be put in a GlobalSettings node to affect the entire scene. In X3D the syntax is:

```
<GlobalSettings>
  <HapticsOptions maxDistance="0.5" />
</GlobalSettings>
```

Some of the options node can be used to set properties local for geometries. In X3D the syntax is:

```
<Shape>
  <Appearance>
    <Material/>
    <SmoothSurface/>
  </Appearance>
  <Box size="0.2 0.2 0.2">
    <HapticsOptions maxDistance="0.5" />
  </Box>
</Shape>
```

4.5.2 C++

Reference counting

When programming in C++ it is important to know that instances of nodes and other classes inheriting from RefCountedClass are reference counted. This means that no explicit deletion of a class instance is needed if, and only if, the class instance is put in an AutoRef, AutoRefVector, SFNode or MFNode. The instance of the class will be deleted when there are no more references to it. In practice this creates code parts like below:

```
{
  AutoRef< Group > group( new Group ); // reference
  Sphere *sphere = new Sphere;
  group->children->push_back( sphere );  // reference
}
// Out of scope. Reference counting on the instance of Group and therefore
// also the instance of Sphere reaches zero and the allocated memory is freed.
```

Note that there is no “delete sphere;” statement in this code.

DependentSFNode and DependentMFNode

DependentSFNode are modified TypedSFNode fields where the field dirty status is dependent on fields in the node it contains. E.g. if we have a Coordinate node, a TypedSFNode< Coordinate > would only generate an event to fields it is routed to if the Coordinate node itself is changed. If the points in the Coordinate nodes point field are changed however no event is generated. With a DependentSFNode you can specify that the field is dependent on e.g. the ’point’ field of the Node and then the field will generate an event when the point field in the Coordinate node generates an event.

DependentMFNode are the multi valued field equivalent of DependenSFNode. For information about syntax of these classes see the “Online Reference Guide”.

4.5.3 Other useful features

This section contains information about certain useful features not covered in previous sections.

Program Setting

When creating scene graphs the creator might want to expose fields for a user to experiment with the scene at runtime. Using H3DViewer the tree viewer window (press F9 or used menu "Advanced"->"Show Tree View") can be used for this purpose. For large scene graphs this view is very hard to use because

of constant scrolling and manually search for fields. This is where the Program setting feature comes in. The syntax is:

```
<PROGRAM_SETTING node="DEFName" field="FieldName" name="Display name" section="Display section" />
```

Where the *name* and *section* arguments can be omitted. For an example x3d file in your H3D API installation see H3DAPI/examples/SuperShape/SuperShape.x3d

CHAPTER 5

EXAMPLES

The examples that follows can be found in the examples/manualExamples folder in your H3D API installation.

5.1 Sphere

This example controls the color of a sphere with the left mouse button using Python. If the button is pressed the sphere will be red, if it is not pressed it will be blue.

5.1.1 Python

Sphere.x3d

```
<Group>
  <Viewpoint position="0 0 1" />
  <Shape>
    <Appearance>
      <Material DEF="MATERIAL" />
    </Appearance>
    <Sphere radius="0.1" />
  </Shape>
  <PythonScript DEF="PS" url="Sphere.py" />
  <MouseSensor DEF="MS" />

  <ROUTE fromNode="MS" fromField="leftButton"
         toNode="PS" toField="color" />
  <ROUTE fromNode="PS" fromField="color"
         toNode="MATERIAL" toField="diffuseColor" />
</Group>
```

Sphere.py

```
#import the H3D fields and types
from H3DInterface import *

# The Color class is of type SFColor and its value is determined by
# the SFBool field that is routed to it. If its value is 1 the color
# is red, otherwise it is blue.
class Color( TypedField( SFColor, SFBool ) ):
  def update( self, event ):
    if( event.getValue() ):
      return RGB( 1, 0, 0 )
    else:
      return RGB( 0, 0, 1 )

# create an instance of the Color class.
color = Color()
```

5.1.2 C++

The same scene can be created using only C++. When using C++ you will have to set up the scene and windows to use as well, while this is already taken care of in the H3DLoad program, when using H3DLoad to load your files.

Sphere_X3D.cpp

```
// This file can be found in the examples/manualExamples/C++ directory in
// your H3D API installation.

// Files included that are needed for setting up the scene graph
#include <H3D/X3D.h>
#include <H3D/Scene.h>
#include <H3D/MouseSensor.h>
#include <H3D/Material.h>
#include <H3D/GLUTWindow.h>

using namespace H3D;

// The Color class is of type SFColor and its value is determined by
// the SFBool field that is routed to it. If its value is true the color
```

```
// is red, otherwise it is blue.
class Color : public TypedField< SFColor, SFBool > {
protected:
  virtual void update() {
    if( static_cast< SFBool * >(event.ptr)->getValue() )
      value = RGB( 1, 0, 0 );
    else
      value = RGB( 0, 0, 1 );
  }
};

int main(int argc, char* argv[]) {
  // Set up the scene graph by specifying a string
  // and using createX3DNodeFromString
  string scene_graph_string = "<Group>"
                              "<Viewpoint position=\"0 0 1\" />"
                              "  <Shape>"
                              "    <Appearance>"
                              "      <Material DEF=\"MATERIAL\" />"
                              "    </Appearance>"
                              "    <Sphere radius=\"0.1\" />"
                              "  </Shape>"
                              "  <MouseSensor DEF=\"MS\" />"
                              "</Group>";

  // myDefNodes contains functions for getting a Node from the scenegraph
  // by giving the DEF name of the node as a string.
  X3D::DEFNodes def_nodes = X3D::DEFNodes();

  // createX3DNodeFromString returns an AutoRef containing a pointer
  // to the top most node in the given string
  AutoRef< Node > group( X3D::createX3DNodeFromString(
                           scene_graph_string, &def_nodes ) );

  // Getting the nodes needed for routes
  MouseSensor *mouse_sensor =
    static_cast< MouseSensor * > (def_nodes.getNode("MS"));
  Material *material =
    static_cast< Material * > (def_nodes.getNode("MATERIAL"));

  // Creating and instance of the Color class needed for routes.
  auto_ptr< Color > myColor( new Color() );
```

```
  // Setting up routes
  mouse_sensor->leftButton->route( myColor );
  myColor->route( material->diffuseColor );

  /// Create a new scene
  AutoRef< Scene > scene( new Scene );

  // create a window to display
  GLUTWindow *gl_window = new GLUTWindow;

  // add the window to the scene.
  scene->window->push_back( gl_window );

  // add our group node that is to be displayed to the scene.
  scene->sceneRoot->setValue( group.get() );

  // start the main loop
  Scene::mainLoop();
}
```

5.2 Spheres

This example adds a new sphere to be rendered at a random position each time the left mouse button is pressed.

5.2.1 Python

Spheres.x3d

```
<Group>
  <Viewpoint position="0 0 1" />
  <Group DEF="GROUP" />
  <PythonScript DEF="PS" url="Spheres.py">
    <Group USE="GROUP" containerField="references"/>
  </PythonScript>
  <MouseSensor DEF="MS" />
  <ROUTE fromNode="MS" fromField="leftButton"
         toNode="PS" toField="add_sphere" />
```

```
</Group>
```

Spheres.py

```
import random
#import the H3D fields and types
from H3DInterface import *

# get the reference to the group to put the spheres in
group, = references.getValue()

# create the sphere geometry
sphere = createNode( "Sphere" )
sphere.radius.setValue( 0.02 )

# The AddSphere class adds a new sphere to the group nodes children
# field each time a field routed to it generates an 1 event.
class AddSphere( AutoUpdate( SFBool ) ):
  def update( self, event ):
    if( event.getValue() ):
      t, dn = createX3DNodeFromString( """\
<Transform>
  <Shape DEF="SHAPE" >
    <Appearance>
      <Material DEF="MATERIAL" />
    </Appearance>
    </Shape>
  </Transform>""" )
      c = RGB( random.random(), random.random(), random.random() )
      dn["MATERIAL"].diffuseColor.setValue( c )
      dn["SHAPE"].geometry.setValue( sphere )
      t.translation.setValue( Vec3f( c.r * 0.5 - 0.25,
                                     c.g * 0.5 - 0.25,
                                     c.b * 0.5 - 0.25 ) )

      group.children.push_back( t )
    return event.getValue()

# create an instance of the AddSphere class
add_sphere = AddSphere()
```

5.2.2 C++

Spheres_X3D.cpp

```
// This file can be found in the examples/manualExamples/C++ directory in
// your H3D API installation.

// Files included that are needed for setting up the scene graph
#include <H3D/X3D.h>
#include <H3D/Scene.h>
#include <H3D/MouseSensor.h>
#include <H3D/Material.h>
#include <H3D/Shape.h>
#include <H3D/Transform.h>
#include <H3D/GLUTWindow.h>

using namespace H3D;

// Global variables.
// The geometry to add at each click.
AutoRef< Node > sphere(
  X3D::createX3DNodeFromString( "<Sphere radius=\"0.02\" /> " ) );

X3D::DEFNodes def_nodes;
AutoRef< Node > group(
  X3D::createX3DNodeFromString( "<Group>\n"
                                "<Viewpoint position=\"0 0 1\" />\n"
                                "  <Group DEF=\"GROUP\" />\n"
                                "  <MouseSensor DEF=\"MS\" />\n"
                                "</Group>",
                                &def_nodes ) );

// The AddSphere class adds a new sphere to the group nodes children
// field each time a field routed to it generates an event.
class AddSphere : public AutoUpdate< SFBool > {
protected:
  virtual void update() {
    AutoUpdate< SFBool >::update();
```

```
    if( value ) {
      X3D::DEFNodes temp_def_nodes;
      AutoRef< Transform > transform(
                      static_cast< Transform * >(  X3D::createX3DNodeFromString(
"<Transform>\n"
"  <Shape DEF=\"SHAPE\" >\n"
"    <Appearance>\n"
"      <Material DEF=\"MATERIAL\" />\n"
"    </Appearance>\n"
"  </Shape> \n"
"</Transform>", &temp_def_nodes ).get() ) );

      RGB c = RGB( (H3DFloat)rand()/RAND_MAX,
                   (H3DFloat)rand()/RAND_MAX,
                   (H3DFloat)rand()/RAND_MAX );
      Material *m;
      temp_def_nodes.getNode( "MATERIAL", m );
      m->diffuseColor->setValue( c );

      Shape *s;
      temp_def_nodes.getNode( "SHAPE", s );
      s->geometry->setValue( sphere );

      transform->translation->setValue( Vec3f( c.r * 0.5f - 0.25f,
                                               c.g * 0.5f - 0.25f,
                                               c.b * 0.5f - 0.25f ) );
      Group *g;
      def_nodes.getNode( "GROUP", g );
      g->children->push_back( transform.get() );
    }
  }
};

int main(int argc, char* argv[]) {
  // Creating and instance of the AddSphere class needed for routes.
  auto_ptr< AddSphere > addSphere( new AddSphere() );
  // Setting up routes
  MouseSensor *ms;
  def_nodes.getNode( "MS", ms );
  ms->leftButton->route( addSphere );

  /// Create a new scene
```

```
  AutoRef< Scene > scene( new Scene );

  // create a window to display
  GLUTWindow *gl_window = new GLUTWindow;

  // add the window to the scene.
  scene->window->push_back( gl_window );

  // add our group node that is to be displayed to the scene.
  scene->sceneRoot->setValue( group.get() );

  // start the main loop
  Scene::mainLoop();

}
```