OpenSG Starter Guide

1.2.0

Generated by Doxygen 1.3-rc2

Wed Mar 19 06:18:02 2003

Contents

1 Introduction 3

1.1 What is OpenSG? 3
1.2 What is OpenSG not? 4
1.3 Compilation 4
1.4 System Structure 5
1.5 Installation 6
1.6 Making and executing the test programs 6
1.7 Making and executing the tutorials 6
1.8 Where to get it 6
1.9 Scene Graphs 6

2 Base 9

2.1 Base Types 9
2.2 Log 9
2.3 Time & Date 10
2.4 Math 10
2.5 System 11
2.6 Fields 11
2.7 Base Functors 12
2.8 Socket 12
2.9 StringConversion 15

3 Fields & Field Containers 17

3.1 Creating a FieldContainer instance 17
3.2 Reference counting 17
3.3 Manipulation 18
3.4 FieldContainer attachments 18
3.5 Data separation & Thread safety 18

4 Image 19

5 Nodes & NodeCores 21

6 Groups 23

6.1 Group 23

6.2 Switch 23

6.3 Transform 23

6.4 ComponentTransform 23

6.5 DistanceLOD 24

6.6 Lights 24

7 Drawables 27

7.1 Base Drawables 27

7.2 Geometry 27

7.3 Slices 34

7.4 Particles 35

8 State Handling 37

8.1 BlendChunk 38

8.2 ClipPlaneChunk 38

8.3 CubeTextureChunk 39

8.4 LightChunk 39

8.5 LineChunk 39

8.6 PointChunk 39

8.7 MaterialChunk 39

8.8 PolygonChunk 40

8.9 RegisterCombinersChunk 40

8.10 TexGenChunk 40

8.11 TextureChunk 40

8.12 TextureTransformChunk 40

8.13 TransformChunk 40

8.14 ProgramChunk 41

8.15 VertexProgramChunk 41

8.16 FragmentProgramChunk 41

9 Materials 43

9.1 Material types 43

10 Action 45

10.1 Usage . . . 45

10.2 RenderAction . . . 45

10.3 IntersectAction . . . 46

10.4 Simple Traversal . . . 46

10.5 Write your own action handler . . . 46

11 Window 47

11.1 Window . . . 47

11.2 Viewport . . . 48

11.3 Camera . . . 49

11.4 Background . . . 51

11.5 Foreground . . . 52

11.6 Navigators . . . 54

11.7 SimpleSceneManager . . . 55

11.8 OpenGL Objects & Extension Handling . . . 55

12 Window System Libraries 57

12.1 GLUT Window System Library . . . 57

12.2 QT Window System Library . . . 57

12.3 X Window System Library . . . 58

12.4 Win32 Window System Library . . . 58

13 File Input/Output 59

13.1 Usage . . . 59

13.2 Cluster . . . 60

13.3 Rendering Backend . . . 62

14 Statistics 63

14.1 Frequently Asked Questions . . . 64

Chapter 1

Introduction

Welcome to the OpenSG starter guide. This document will help you understand the structure of the OpenSG system and the main classes that you need to know to write graphics programs using OpenSG. It started as a quick-start, but got a little big for that. ;) If you're not into reading text, take a look at the tutorial programs and come back when you need a little background for specific topics.

It is not meant to be an introduction to computer graphics or real-time rendering. It would actually be useful to be somewhat familiar with another scene graph system, but that's not mandatory. For a short introduction to the scene graph concept, see Scene Graphs .

It does not explain every single function and its parameters, take a look at the Code documentation that is generated by doxygen for that. It will also not motivate the decisions taken and alternatives that were rejected, see the design document for that. This is just for jump-starting OpenSG usage.

If you want to send feedback or have further questions you can send them either to the OpenSG user list (opensg-users@lists.sourceforge.net) or to us directly at feedback@opensg.org.

This starter guide was written by the core team, but it wouldn't be what it is without the helpful reviews and comments by Marc Alexa and Patrick Dähne.

This is for version 1.2.0, which is a stable release, meaning that we put quite a bit of work into writing it. That doesn't mean it's perfect or without problems, which we would like to hear about. But it's a lot better than what we had before.

1.1 What is OpenSG?

OpenSG is a real-time rendering system based on a scene graph metaphor. It works along the lines of OpenInventor, Performer or Java3D, although it is probably closest to Performer. It supports parallel processing, albeit in a more general way, and will drive multiple displays for multi-screen stereo projection systems, possible distributed across a cluster of machines. The goal is to have something that handles multi-threaded data structures as simply as possible without compromising performance too much. It should also support heterogeneous multi-pipe applications, i.e. multiple different graphics cards running one application. Many things are quite easy to do with a little program, but are sometimes hard to fit into an existing system. Thus accessibility is an important goal, and we're striving to make OpenSG very extensible.

It works on different Unix systems and Windows. It compiles with the Microsoft Visual Studio compiler, starting with version 7.

1.2 What is OpenSG not?

OpenSG is not a complete VR system. Things like device access and interaction are left out on purpose, there are other systems for that (like VRJuggler or Open Tracker).

1.3 Compilation

1.3.1 Applications Using OpenSG

1.3.1.1 Unix / Cygwin

You either need to compile the library (see below) or install a binary version. OpenSG comes with a `osg-config` script that can output the necessary option for compiling/linking applications suing OpenSG. As OpenSG is split into several optional libraries (see System Structure), the script needs to know which libraries you want to use. For example to find the options necessary to compile a source file for an application that uses the base, system and GLUT window-system libaries you would put the output of `osg-config --cflags Base System GLUT` into your `Makefile.` For an example see the `Makefile` of the tutorials, it works on every system with a usable command line.

1.3.1.2 Visual Studio

If you want to compile an application using OpenSG using Visual Studio we **strongly** recommend to not start a new project and add the OpenSG libraries, but start with a project for a tutorial example from the binary distribution.

The reason is that OpenSG needs a bunch of command line options for the Visual Studio compiler to use it, and using different options for compiling applications than the ones used for compiling the library can and will lead to mysterious crashes, typically in inline methods that reallocate memory like adding children to nodes, or creating Geometry. The reason is usually the braindead memory management of Windows, which does not allow freeing memory that was allocated in another DLL, unless all of them have been linked with the same runtime lib. If you didn't understand that: never mind, just make sure you use the right compiler options.

1.3.2 Compiling OpenSG Itself

1.3.2.1 Unix / Cygwin

Short version: ./configure ; make should work.

Long version: OpenSG uses a relatively standard configure script to adapt the options it needs to the system it's compiled on. configure is also used to set up optional libraries that are available. For Windows you can use the cygwin environment (`http://sources.redhat.com/cygwin/`) to get the needed shell tools.

Note that there are some constraints. You should install the Visual Studio and Intel compiler in their default locations (...:\Program Files\...), otherwise the configure script will not find them. Install the GLUT in UNIX-style directories, i.e. .../include/GL/glut.h and .../lib/ ∗dll+∗lib, otherwise it won't be found. Note that you need to add the directory where the libs are created to your PATH or WINPATH (or to the Windows environment variable PATH), or copy all the dlls to the directory where your programs are. Some of these will be lifted in the future, but that's the way it is right now.

All options are optional. The most useful ones are `--with-jpg[=<dir>]`, `--with-glut[=<dir>]`, `--with-qt[=<dir>]`, `--with-tif[=<dir>]` and `--with-png[=<dir>]` which specify the directories where the specific libraries/header can be found. If you need to download these libraries, check `http://www.opensg.org/prerequisites.EN.html` for locations. All examples and tutorials use GLUT, so it's highly recommended to configure it. There are some other options that are less often needed, run configure –help to get a full list.

So for a standard Linux distribution you should call ./configure –with-jpg –with-tif –with-png –with-qt=$QTDIR to get a useful configuration.

Configure creates a directory in Builds with a name specific for the current system, e.g. Builds/i686-pc-linux-gnu-g++/. Go into this directory and call make to create the libraries.

In general you can call make help in any directory with a makefile to get a list of supported targets in this directory.

Not that OpenSG needs the GNU version of make. This is the default on Linux and Cygwin systems, so a simple make is fine. On IRIX it's usually called gmake, if you don't have it you need to install it yourself.

1.3.2.2 Visual Studio

Project files for Visual Studio 6 and 7 are in the `VSBuild` directory. They can be created from the Makefiles on a Cygwin system automatically (make dsp or make dsp7). The project for VS7 is actually a VS6 project that will be converted to VS7 format the first time it is used.

Note that you need the Intel compiler for VS6, the Microsoft VS6 compiler is just too far away from the C++ standard to use OpenSG.

1.4 System Structure

The full OpenSG system consistes of several libraries.

The Base library contains the low-level functions and basic classes like vectors and volumes. It also contains the operating system abstractions and the basic multi-threading wrappers.

The System library is the main library. It contains all the higher-level objects starting at the FieldContainer level. That includes all the scenegraph nodes and the actions to work on them. It includes the loaders for images and scene files, and the general window handling classes. In general, it includes everything the other libraries don't.

For each window system that is supported there's a separate little library which contains the interface objects for that window system. Right now we have interfaces to X, WIN32, GLUT and QT.

The sources for all of these are in the `Source` subdirectory.

1.5 Installation

Call make install to install the libraries in the place specified to configure, or /usr/local per default. You can change the installation directory at install time by specifying it as make INSTALL_DIR=<dir> install. make install will copy all include files into a directory OpenSG in $INSTALL_DIR/include and the libs into $INSTALL_DIR/lib/dbg or $INSTALL_DIR/lib/opt, depending on the optimization used.

For a local install it's useful to use BUILD as the prefix for configure, i.e. call configure –prefix=BUILD. This will install the system into the correct Builds/ * directory.

1.6 Making and executing the test programs

To build the test programs, go into the BaseTest, SystemTest etc. directories and call make or make Tests. make list lists the available programs in the current directory. You can make a specific test program by running the listed command.

To execute the test programs you need to have the OpenSG dynamic libraries in your library search path. For that it's easiest to locally install them as described before and call (csh/tcsh) setenv LD_LIBRARY_-PATH ${LD_LIBRARY_PATH}:../lib/dbg or (sh/bash) export LD_LIBRARY_PATH=${LD_LIBRARY_-PATH}:../lib/dbg. Then you can just run the test programs in the *Test directory.

1.7 Making and executing the tutorials

There are a number of tutorial examples in the tutorials directory. They are available in compiled form as a part of the binary distributions, but it is possible and probably the best way to start using OpenSG to change and recompile them.

You need to make and install the libraries first. To actually create the tutorials on Unix just call make. For Windows there's a Visual Studio Workspace file in the binary distribution.

1.8 Where to get it

If you read this you probably have it already, otherwise take a look at http://www.opensg.org/ for the latest released version. You can also get it from SourceForge, which we use for project management, at http://www.sf.net/project/showfiles.php?group_id=5283.

If you want to get access to the current development version you can get it from CVS, see http://sourceforge.net/cvs/?group_id=5283 for details. Nightly snapshots of the CVS source and libs are available at at www.opensg.org/dailybuild_logs.

1.9 Scene Graphs

The most prominent standard library used for writing interactive 3D graphics applications is OpenGL. OpenGL is cleanyl and well designed and has a very specific goal: be a layer on top of the graphics hardware, as thin as possible, as thick as necessary to abstract away the differences between different graphics hardware variants.

A consequence of this design is that OpenGL is very fine-grained and flexible. Every object has to be specified vertex by vertex including all its data. Newer OpenGL functions (e.g. Vertex Arrays) simplify

this somewhat, but only in a limited sense. The closest thing to an object in OpenGL is a display list, which is just like a macro recording of the executed OpenGL commands, and don't feature any object-specific interface.

The idea of a scene graph is to provide a higher level of abstraction. A scene graph actually stores the whole scene in the form of a graph of connected objects.

Figure 1.1: Example Scene

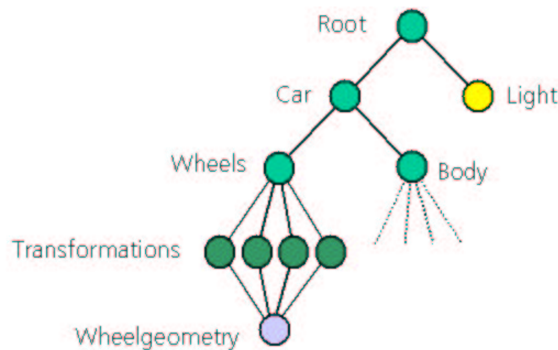

Figure 1.2: Example Scene's Graph

Graphs are used in many places in computer science, scene graphs are a subset of general graphs. They are acyclic, i.e. there can be no loops in the scene graph, as they would lead to infinite loops when trying to walk the graph. They are directed, i.e. there is a defined direction of the connections between the nodes. Thus nodes can be grouped together by attaching them to a parent node. This conceptually allows manipulation of a large number of nodes by manipulating this parent node. Of course this is recursive, i.e. these parent nodes themselves can also be grouped and attached to another parent. In the end this leads to a single root node, which defines the whole scene to be rendered. In general there can be more than one graph in an application, but whatever is displayed in a single window is usually defined by a single root node.

One aspect that scene graphs make more explicit than OpenGL is the sharing of data. It is desirable to use the data of a node in multiple places in the graph, e.g. store only one copy of the wheel's geometry and use it in multiple places. The disadvantage of the simple multi-use is that it is not possible any more to identify a single wheel, as they are all the same object. Thus OpenSG uses a different method to do sharing, the Node-Core split, see Nodes & NodeCores .

Scene graphs, in contrast to many other graphs in computer science, are also heterogeneous, i.e. there are many different types of nodes with different functions (see Nodes & NodeCores for the different types OpenSG supports). There are two general types of nodes: groups and drawables.

Groups (see Groups) are used to structure the graph into parts and to control either attributes of their subgraph (e.g. Transform) or to select only a selection of their children for traversal (e.g. Switch).

Drawables (see Drawables) are usually put in the leaves of the tree (i.e. they have no children) and contain the actual elements to be rendered. In most cases this will be polygonal geometry (see Geometry), but other types are possible (e.g. Particles or Slices). The parameters that define the surface charaterictics of the rendered objects e.g. surface color, shininess or texture (or rendering paramaters in general, as some Drawables don't have a surface) are usually associated with the Drawable wrapped in a separate object, in OpenGL it is called Materials . This Material can and should be used by as many object as possible, to get good performance.

I hope this page can give you a first rough idea of what scene graphs are about.

Chapter 2

Base

All OpenSG symbols are part of the OSG name space, and they have no prefix. The actual files, including headers, all use the OSG prefix.

2.1 Base Types

As one goal of OpenSG is the ability to run programs on a lot of different platforms, especially Unix and Windows, we have our own types which are guaranteed to have the same size on all platforms.

We have our own signed and unsigned integers in all useful sizes: Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64, UInt64 as well as the two usual float sizes Real32 and Real64. Some useful constant are available: Eps, Pi, Inf and NegInf. A useful construct for template programming is the TypeConstants<type> structure, which defines some standard functions/values for the given type, see OSGBaseTypes.h for details.

2.2 Log

All output that OpenSG generates is channeled through the Log class, which is defined in OSGLog.h. OpenSG supplies a global Log object that is used by the library, but the application can create its own logs, if needed.

Every log message has one specific level. Available levels are FATAL, WARNING, NOTICE, INFO and DEBUG. They are also numbered from 0 to 5. The verbosity of the system can be controlled by ignoring messages of specific levels. This can be achieved by calling osgLog().setLogLevel(<enum>); or by setting the environment variable OSG_LOG_LEVEL.

The system log has two different interfaces. One is based on C++ streams, one is based on C printf semantics.

The stream interface can be used by using SFATAL, SWARNING, SNOTICE or SINFO instead of cout or cerr. Note that there is no SDEBUG for efficiency reasons, as FDEBUG can be compiled out. These print the position in the code where the log is executed. For multi-line outputs you'll only want that on the first line, for the other lines use PFATAL, PWARNING, PNOTICE or PINFO.

To synchronize multiple outputs from various threads all S* commands lock the stream. You have to use 'osg::endLog' (e.g. SFATAL << "Message" << endLog;) to unlock the stream output.

Example: SINFO << "Line 1 of message 1" << endl; PINFO << "line2 of message 1" << endLog;

SINFO << "Message 2" << endLog;.

The C interface tries to mimic the printf semantics. The following functions can be used for that: FFATAL, FWARNING, FNOTICE, FINFO and FDEBUG. The only difference to printf is that they have to be called with double parentheses, i.e. FWARNING(("What do you mean by s?", s));. The nice thing about the C style interface is that the whole output can be compiled out. Actually, the FDEBUG (( )) are only compiled in when OSG_DEBUG is set. The OSG_DEBUG define is automatically set while compiling the system in debug (default) mode.

The user can activate/deactivate various elements per log message at runtime by changing the LogHeaderElem mask. The following elements are supported right now:

LOG_BEGIN_NEWLINE_HEADER (creates an extra newline in front of every output), LOG_TYPE_HEADER (writes the Level (e.g. WARNING) as first element), LOG_TIMESTAMP_HEADER (writes a timestamp), LOG_MODULE_HEADER (writes the name of the current module), LOG_FILE_HEADER (writes the source file name), LOG_LINE_HEADER (writes the source line number) and LOG_END_NEWLINE_HEADER (creates an extra newline at the end)

When unchanged, the time stamp will be the time in seconds since the program started. The user can set/reset the time stamp at any time (e.g. osgLog().resetRefTime()).

2.3 Time & Date

To wrap time and date handling we have a little abstraction for them.

getSystemTime() returns the current time since system has been started in seconds, using the highest resolution timer available.

The Date class provides a second resolution time stamp, factored into second, minute, hour, day, month and year. Date::setSystemDate() can be used to set it to the current date/time.

2.4 Math

Of course every scene-graph needs the basic math objects like Vectors, Points, Matrices, Quaternions etc., and OpenSG is no exception.

2.4.1 Matrices

OpenSG matrices are similar to the OpenGL matrices in their storage structure and conventions, i.e. a matrix is per default a 4x4 Real32 matrix, and the multiplication convention is just like OpenGL's: v'=M*v.

The matrix is stored column major and access methods respect the storage format, i.e. matrix[0] yields the first column. This is also true for the vector-based constructor. However, the constructor taking 16 single elements expects its parameters row-major like the matrix is written on paper.

The positive side effect of this setup is the ability to access the base vectors of the matrix' coordinate space by accessing the vectors, i.e. matrix[3] is the translation to the origin of the local coordinate space. This is useful if you want to create your matrices from vectors, if you don't want to do that, don't worry about it.

Setting the contents of a matrix is done by the setValues() methods, accessing the values via the [] operator for access to single columns or by using getValues() to get a pointer to the first element. In general most classes in OpenSG that keep an array of elements allow access to them via getValues().

If you need to create a matrix for a specific transformation, use the setTransform() methods, which create a matrix that executes the given transformation.

Matrices also supply the standard set of matrix operations like det(), det3(), invert(), transpose(), mult() and multLeft(). There are some variants that change the matrix in place, return their results in a different matrix or get their source data from a different matrix, see the class docs for details.

The default vector/point multiplication methods multMatrixVec() and multMatrixPnt() assume that the matrix only uses the standard 3x4 elements. To use the full 4x4 matrix use multFullMatrixPnt(). As Vectors have a w coordinate of 0, compared to points which have w = 1, they don't need a full transform.

2.4.2 Vectors/Points/Colors

OpenSG is different from most other systems in differentiating between vectors, points and colors.

Vectors are the most common class, and they should behave like every other vector library on the planet. They are templated to simplify having variants, and the standard ones that are available are Vec4ub, Vec2s, Vec2f, Vec3f and Vec4f. They have operators for the scalar operations, and methods for everything else, see the doxygen docs for osg::VectorInterface for details. Conceptually, the 3 element vector has a w coordinate of 0, thus there is no full matrix multiplication for vectors.

Points represent positions in space, and as such they are more restricted than vectors. The available variants are Pnt2f, Pnt3f and Pnt4f. Some vector operations (dot, cross, etc.) don't make sense for points. Points can be subtracted (creating a vector), scaled and a vector can be added to or subtracted from them. If you want to represent a position, use a point. It helps keeping the concepts in order and not mix up everything just because it has the same data. When multiplied with a matrix, the w coordinate is set as 1 for 3 element points.

If you really need to get from a point to a vector or vice versa, you can use

- Vector &osg::Point.subZero()
- Point &osg::Vector.addToZero()

to cast a point to a vector and back.

Colors are RGB vectors, which also have access functions to the named components. They also allow access via the HSV color model and scalar multiplication, but no other operations.

2.4.3 Quaternions

Quaternions are the standard way to represent rotations. OpenSG quaternions feature the standard set of methods to get and set the rotations, in variants for radians and degrees. The standard order of the components is x,y,z,w. The standard operations (length, normalize, mult) are available, as well as slerp and multVec.

2.5 System

SystemLib: everything not base and winsys

2.6 Fields

All data in FieldContainers is organized in fields. There are two general types of fields, fields for single values (SFields) and fields for multiple values (MFields). For the standard types and most pointer and ptr types there are predefined instances of both types of fields.

2.6.1 Single Fields

Single fields hold, as the name says, a single value. Their content can be accessed directly using getValue(); and setValue();. It can also be copied from another field of the same type by setValue(); (for fields of the same type) or by setAbstrValue(); (for fields which have the same type, but are given as an abstract field).

2.6.2 Multi Fields

Multi fields hold multiple values. They are realized as STL vectors and offer a similar interface. The field defines types for iterators and references, and the standard begin(), end(), front(), back(), push_back(), insert(), erase(), clear(), size(), resize(), reserve() and other functions.

In addition, Multi fields have an interface reminiscent of single fields. It features the setValue() variants mentioned above and indexed variants like getValue(const UInt32 index) and setValue(const FieldTypeT &value, const UInt32 index) methods. It also features an OpenSG-style getSize() method.

2.6.3 FieldContainer Fields

Each attribute has a name, e.g. someValue, and every field container has a set of standard access functions to access its fields. The field itself can be accessed via getSFSomeValue() or getMFSomeValue() for single or multiple value fields respectively.

For SFields containers features getSomeValue() and setSomeValue() direct access methods. The MField getSomeValue() method returns the whole field, just like the getMFSomeValue() method. Some field containers have more access functions, often something like an addSomeValue() method to simplify adding data to multi fields. See the field container docs for details.

2.7 Base Functors

2.8 Socket

Socket baseclass. The Socket class wraps a socket descriptor. This class has no additional state variables. It is only h handle to the underlaying descriptor. Class createion and destruction has no influence to any descriptor. Use open to assign a descriptor and close to remove it from the system. If this class is copied, then there are to classes which uses the same descriptor. This is ok until you call close for one of this classes. One purpose of this implementation is to hide the differences between Windows and Unix sockets. Calls to this class should behave equally on all systems. As a result, some methods will not work as an experienced Windows ore Unix programmer maight expect. Please refere to the function docu to get details about this.

Stream sockets

A stream socket is an endpoint for a reliable point to point communication. The following example code shows, how to establish a stream socket connection.

```
// server code
StreamSocket socket,client;
socket.open();
socket.bind(SocketAddress(SocketAddress::ANY,12345));
```

```
client=socket.accept();
client.send(buffer,100);
client.close();
socket.close();

// client code
StreamSocket server;
server.open();
server.connect(SocketAddress("localhost",12345));
server.recv(buffer,100);
server.close();
```

The method bind() assigns the socket to a given port and network device. If the network device is ANY, then the socket will accept connections from all network interfaces. If the port number is zero, then a free port is assigned. In the above example, the socket will accept incoming connections at each interface on the port 12345. StreamSocket::accept() waits for a incoming connection. For each incoming connection a new socket object will be created. With send and recv data can be transferred over the connection. By default all calls will block until the operation has finished.

Datagram sockets

Datagram sockets are not connection orientated. There is no connect or accept methods for datagram sockets. You have to provide a destination address for each send and will get a source address for each recv call. There is no guarantee that packages will arrive and there is no guarantee for the order in which the package will arrive. The followin code shows how to wait for incoming packages at port 22222.

```
SocketAddress client;
DgramSocket socket;
socket.open();
socket.bind(SocketAddress(SocketAddress::ANY,22222))
socket.recvFrom(buffer,100,client);
socket.close();
```

This is the code to send the package. This is a simple example. If the package gets lost for example because the server is not started, then this code wont work. If your application relays on reliable data transmission, then StreamSockets should be used.

```
DgramSocket sock;
SocketAddress server;
socket.open();
socket.sendTo("hallo",100,SocketAddress("localhost",22222));
socket.close();
```

Exceptions:

All socket methods will throw exceptions if the desired function could not be finished. All socket related exceptions are derived from the SocketException class. All socket calls should be enclosed in try/catch blocks. For example, if you want to check if a connect operation was successful then you should use the following code.

```
try
{
    server..connect(SocketAddress("localhost",12345));
}
catch(SocketException &e)
```

```
{
    SFATAL $<$$<$ "Unable to connect to server:" $<$$<$ e.what() $<$$<$ endl;
}
```

Each exceptions contains the system error text that causes the error situation. This text can be accessed by SocketException::what(). The text is given as an std:string object.

Broadcast Messages

Broadcast packages are a special case of datagram packages. Broadcast packages are send to each host in a network. For broadcasting packages a special address type is used. For example with the address SocketAddress(SocketAddress::BROADCAST,2345) packages will be send to each host on port number 2345. Equal to normal datagram sockets, broadcast packages are transmitted not reliable.

Multicast Messages

With multicast it is possible to send packages to more then one destination. In contrast to broadcast the package is not send to all hosts but only to those hosts that have joined a multicast group. A multicast group is specified with special IP addresses (224.xxx.xxx.xxx). To write a package to the multicast group 224.22.33.33 at port 4444 you could use

```
socket.sendTo(buffer,100,SocketAddress("224.22.33.44",4444)
```

The receive has to join this group. The following examples shows how to join and leave multicast groups. It is possible to join more then one group.

```
socket.bind(SocketAddress(SocketAddress::ANY,4444));
socket.join(SocketAddress("224.22.33.44"));
socket.join(SocketAddress("224.0.0.52"));
socket.join(SocketAddress("224.0.0.53"));
socket.leave(SocketAddress("224.0.0.52"));
```

With multicast groups you also have to bind your socket to a network device and port. In the example above only those packages are received that are send to port 4444. With multicast it often happens, that you have more then one member of a multicast group on a single host. If you try to call bind for the same port multiple times, then you will get socket exceptions. To be able to assign more then one process to the same port you have to call socket.setReusePort(true) before the call to bind.

Nonblocking IO

In many applications it is not possible or not wanted to block execution until an accept() or recv() call has finished. Each socket has methods to ask if it is possible to read or write data without blocking. The followin examples shows how to wait for data without blocking.

- socket.waitReadable(0): Don't wait, returns true if data is available otherwise it returns false.
- socket.waitReadable(0.5): If data arrives in the next .5 seconds then true is returned. Otherwise false is returned.
- socket.waitReadable(-1): Wait until data is available and then return true. This function call is equal to a blocking read.
- socket.waitWritable(10): If the socket is able to send data in the next 10 seconds, then true is returned. Otherwise false.

With this simple interface it is possible to wait a specified time for incoming or outgoing data. But it is only possible to wait for exactly one socket. If it is necessary to wait for more then one socket then the Selection class could be used. A selection specifies for each socket for what kind of event the system should wait.

```
SocketSelection s;
s.setRead(socket1);
s.setRead(socket2);
s.select(.1);
if(s.isSetRead(sock1))
   socket 1 is readable
if(s.isSetRead(sock2))
   socket 1 is readable
```

A call of selection.select(seconds) waits the given number of seconds for the events set with setRead() or serWrite(). It returns after the first occurrence of an event. If no events occur in the given periode then 0 is returned. Otherwise the number of readable and writable sockets is returned. The method select() modifies the selection object. You have to call setRead() and setWrite() for each time, you want to call select(). If the SocketSelection is constant, then you can use an other version of select. With s.select(seconds,rs) s will not be modified. The result will be set into the SocketSelection rs. You have to call rs.isReadable(sock) instead of s.isReadable(sock).

IO-Buffers

If data should be read from more then one socket, then the performance could be improved by not reading from the first socket that provides data but from that socket with the most data in the input buffer. With socket.getAvailable() the number of bytes in the input buffer could be retrieved. In addition the Socket class provides the functions setWriteBufferSize and setReadBufferSize set the size of input and output buffers. Current sizes can be retrieved with getReadBufferSize and getWriteBufferSize.

2.9 StringConversion

TODO

Chapter 3

Fields & Field Containers

One central goal in OpenSG's design is easy to use thread-safe data. To do that right, you need to replicate the data so that every thread can have its private copy (called aspect) to work on. At some point these different copies will have to be synchronized, and then the parts that actually changed need to be copied from one aspect to another. To do that, the system needs to know what actually changed. As C++ is not reflective, i.e. the classes cannot tell the system which members they have, OpenSG needs to keep track of the changes. That's what Fields and FieldContainers are for.

3.1 Creating a FieldContainer instance

FieldContainer can be created in two ways: By using the FieldContainerFactory or from the class's prototype. You cannot create instances of FieldContainers neither by creating automatic or static variables nor by calling new. You have to use the mentioned two ways.

For generic loaders it is useful to create an object by name, and this is what the factory is for. The factory is a singleton, the single instance can be accessed via FieldContainerFactory::the(), which has functions to create arbitrary field containers, with some special versions to directly create different subsets of field containers (Nodes, NodeCores, Attachments).

For reasons connected to multi-threading (s. [threadsafety]) specific kinds of pointers have to be used. For every FieldContainer type fc there is a specific pointer type fcPtr. It has all the features of a standard pointer, i.e. it can be dereferenced via -> and it can be downcasted to a derived type by DerivedPtr.dcast(ParentPtr);.

Creating a new instance of a specific class is done by calling fcPtr var=fcPtr::create().

3.2 Reference counting

FieldContainers are reference-counted. They are created with a reference count of 0, and the reference count can be manipulated through addRefCP() and subRefCP().

The system increases the reference count only when it stores a reference to an object in the system, e.g. when a node is attached to another node. It does not increase the reference counter for every parameter that is passed around, the pointers mentioned in [fcinstance] are not smart pointers.

The reference count is decreased when an object is removed from the system, e.g. when a node is detached from another node, or explicitly using subRefCP(). If the reference count goes to or below 0, the object is removed. Note that objects are created with a reference count of zero, so if a new object (refCnt: 0) is

attached to a node (increasing the refCnt to 1) and removed later on (decreasing it to 0), it will be destroyed. Increasing the reference count before removing it is needed to prevent the destruction.

3.3 Manipulation

The FieldContainer is the basic unit for multi-thread safety. To synchronize changes between different copies of the data the system needs to know when and what changed.

This has to be done explicitly by the program. Thus, before changing a FieldContainer beginEditCP(fcPtr, fieldMask); has to be called. After the changes to the FieldContainer are done this also has to be communicated by calling endEditCP(fcPtr, fieldMask);. Here, fcPtr is the pointer to the FieldContainer being changed, fieldMask is a bit mask describing the fields that are changed.

Every FieldContainer defines constants for all its fields that can be used to set up this mask. The naming convention is [FieldContainer]::[FieldName]FieldMask, e.g. Geometry::PositionsFieldMask. These masks can be or-ed together to create the full mask of fields that are changed.

3.4 FieldContainer attachments

OpenSG field containers and nodes do not feature an unused pointer to attach data, usually called user data in other systems. Instead, many field containers feature a map to attach specific kinds of field containers called attachments. The most important ones are Nodes and NodeCores, but many other like Window, Viewport, Camera, etc. are derived from AttachmentContainer and, therefore, can carry attachments.

Attachments have to be derived from Attachment (see for details on how to do that). There are also predefined attachments, right now the only one is NameAttachment, which allows assigning a name to the field containers.

Every AttachmentContainer can hold an arbitrary number of attachments. Attachments are divided into separate groups, and there can be only one attachment of every group attached to an AC. Most attachments are a group, but if needed new ones can be used as replacements for their parents.

3.5 Data separation & Thread safety

One of the primary design goals of OpenSG is supporting multi-threaded applications. For asynchronous threads that means that every thread might need its private copy of the data. To combine that with easy usability and efficient access we decided to replicate at the field container level.

When a field container is created not only one instance is created but multiple, per default 2. These are called aspects, and every running thread is associated with one of them. Whenever data is changed in a thread, only the aspect that's associated with it is changed, the rest is left as is.

Chapter 4

Image

Defines and holds a 1D/2D/3D image and optionally a mipmap pyramid and/or a list of equally sized frames with a single frameDelay. Various pixelTypes are supported to handle gray and RGB color images with or without alpha channel.

An Image is only a container for the pixel data and image description. It does not create or handle any OpenGL state elements. However, image objects are utilized to handle the data for texture (e.g. osg::SimpleTextureMaterial or osg::TextureChunk) or bitmap objects (e.g. osg::ImageForeground).

The image object holds some describing fields (e.g. width/height/depth), and a single block of memory for all the raster data.

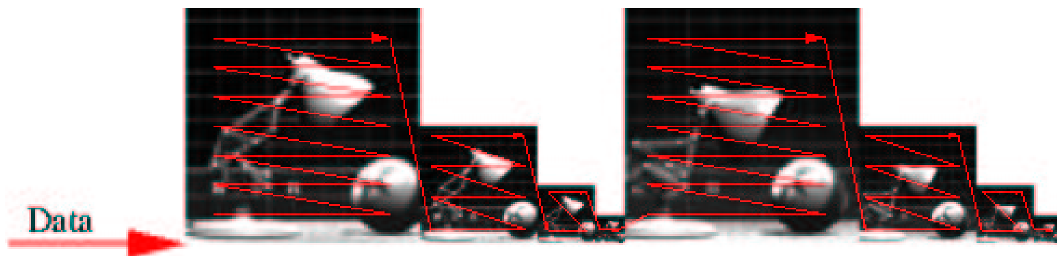

Figure 4.1: Mem layout for Multi-Frame mip-map image data

The image data starts in the lower left (front) corner and all bytes for a single pixel (e.g. RGB) are stored sequentially in memory. They are not organized in separate layers or channels.

The optional mipmap-level are stored directly after the Image data. The dimensions (width/height/depth) are always reduced to 50% of the previous level but stay at least 1. The Image object can hold just a number of levels and must not be complete (end with a 1x1x1 level).

If there are any additional mipmap-levels defined, OpenSG will use the data turing the texture-upload process. If the image does not contain mipmap levels, which is true in most cases, OpenSG will use OpenGL functions to create the level direclty as OpenGL objects. The System will not change the image object for rendering. Therefore, the ability to define the mipmap levels is rarely needed but very usefull (e.g. Real-Time Hatching).

The pyramid (at least one Level) defines a frame. All frames are stored after each other separately starting with frame 0.

If the user loads a multi-frame image and assigns the object to a osg::TextureChunk it is not played auto-

matically as movie. The Application has to set and change the 'current' frame in the TextureChunk.

The Image Class implementation does not include Image-Processing functionality, but provides some simple funktions to scale and crop and to set subregions of the data.

The system provides loaders and writers for various formats including png, jpeg, tiff, gif and sgi. In addition a specific mtd (multi texture data) reader/writer is included. The mtd file format is a simple platform independent header and binary dump of the Image object data. It is the only format which can hold all field and data properties of the OpenSG Image.

The graph loaders (e.g. OSGLoader, VRMLLoader) use the image loaders to fetch the raster data.

Chapter 5

Nodes & NodeCores

Of course the most important structures in a scene-graph are the actual nodes that make up the graph.

OpenSG uses a somewhat different approach than many other systems. A node is split into two parts: the Node and a NodeCore (s. fig [singleParentFig]), both of which are FieldContainers, so all that has been said before applies to them.

A Node keeps the general information: a children list, a parent pointer, a bounding volume and a core pointer. Note that the node itself contains no information about its type (e.g. transform, group, etc.). A Node cannot be shared, every node can only be at one place in the graph, thus a single parent pointer is enough. All nodes together define the topology of the graph, without defining any content. Actions that depend on a position in the graph, like accessing the accumulated matrix to the world coordinate system or the world bounding volume, have to be done on the node, as it uniquely defines and identifies the position in the graph.

A NodeCore carries the differentiating information for a node. There are NodeCores for all the different functions needed in the tree: groups, transformations, geometry and many more. NodeCores can be shared between different nodes, thus they keep an array or actually a MultiField of Node pointers.

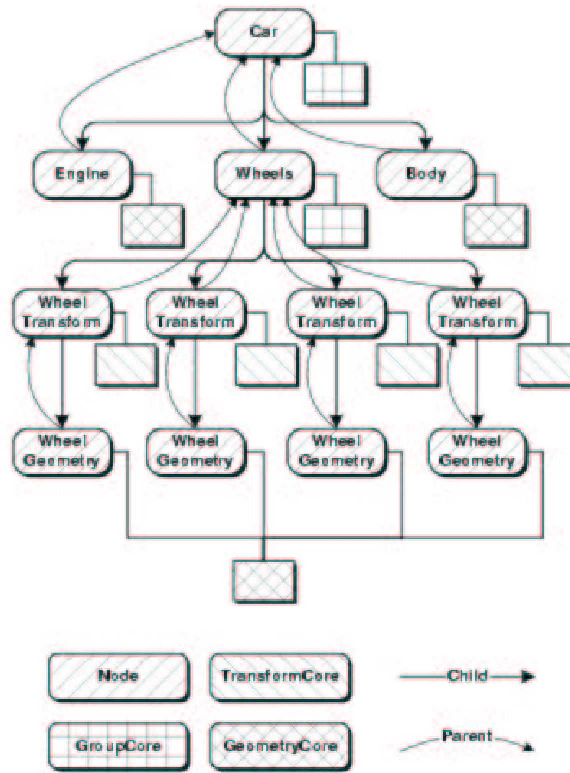

Figure 5.1: Node & Node Core Sharing

The types of NodeCores using in OpenSG are divided into two large groups: Groups (Groups) and Drawables (Drawables).

Chapter 6

Groups

Groups are all the NodeCores who can be used as interior nodes in the graph. These nodes can have and actually use their children.

6.1 Group

A Group is the simplest NodeCore, it doesn't do much. If asked to do something it calls its children to do the same thing, if asked for information it gathers it from the children. It does not intoduce a new transformation.

6.2 Switch

A Switch node allows to select one of its children for traversal instead of all of them (as for the other nodes). The Switch grouping node traverses zero, all or one of the children.

6.3 Transform

The Transform node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its ancestors. A Transform Core is the basic means of moving objects around the scene. It keeps a single Matrix that is applied to all its children.

6.4 ComponentTransform

A ComponentTransform is close to a Transform, but the transformation is defined in an easier to use way, the same way it is done in systems like OpenInventor or VRML:

The center field specifies a translation offset from the origin of the local coordinate system (0,0,0). The rotation field specifies a rotation of the coordinate system. The scale field specifies a non-uniform scale of the coordinate system. scale values shall be greater than zero. The scaleOrientation specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The scaleOrientation applies only to the scale operation. The translation field specifies a translation to the coordinate system.

Given a 3-dimensional point P and Transform node, P is transformed into point P’ in its parent’s coordinate system by a series of intermediate transformations. In matrix transformation notation, where C (center), SR (scaleOrientation), T (translation), R (rotation), and S (scale) are the equivalent transformation matrices,

$P' = T \times C \times R \times SR \times S \times -SR \times -C \times P$

6.5 DistanceLOD

Levels of Detail are a simple way of increasing rendering performance. The basic idea is to have a number of differently detailed versions of an object and use low-res versions for objects that are far away.

A DistanceLOD is the simplest version, which switches versions based on distance to the viewer. The children is selected based on the center and range settings.

The center field is a translation offset in the local coordinate system that specifies the centre of the LOD node for distance calculations.

The number of children shall exceed the number of values in the range field by one (i.e., N+1 children for N range values). The range field contains monotonic increasing values that shall be greater than 0. In order to calculate which level to display, the distance is calculated from the viewer’s location, transformed into the local coordinate system of the LOD node (including any scaling transformations), to the center point of the LOD node.

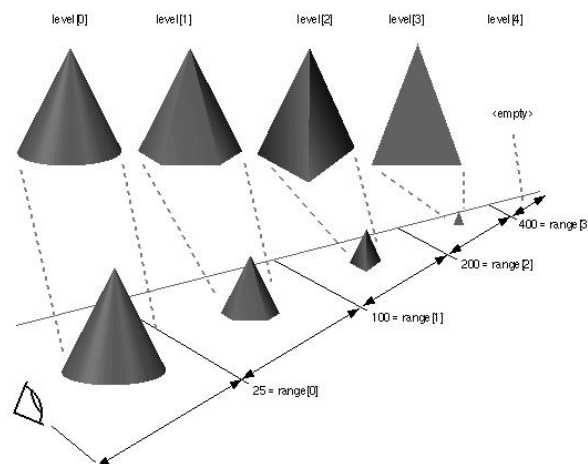

Figure 6.1: DistanceLOD example

6.6 Lights

A Light defines a source of light in the scene. Generally, two types of information are of interest: The position of the light source (geometry), and what elements of the scene are lit (semantics).

Using the position of the light in the graph for geometry allows moving the Light just like any other node, by putting it below a osg::Transform Node and changing the transformation. This consistency also simplifies attaching Lights to moving parts in the scene: just put them below the same Transform and they will move with the object.

The semantic interpretation also makes sense, it lets you restrict the influence area of the light to a subgraph of the scene. This can be used for efficiency, as every active light increases the amount of calculations necessary per vertex, even if the light doesn't influence the vertex, because it is too far away. It can also be used to overcome the restrictions on the number of lights. OpenSG currently only allows 8 concurrently active lights.

It is also not difficult to imagine situations where both interpretations are necessary at the same time. Take for example a car driving through a night scene. You'd want to headlights to be fixed to the car and move together with it. But at the same time they should light the houses you're driving by, and not the mountains in the distance.

Thus there should be a way to do both at the same time. OpenSG solves this by splitting the two tasks to two Nodes. The Light's Node is for the sematntic part, it defines which object are lit by the Light. FOr the geometrc part the Light keeps a SFNodePtr to a different Node, the so called beacon. The local coordinate system of the beacon provides the reference coordinate system for the light's position.

Thus the typical setup of an OpenSG scenegraph starts with a set of lights, which light the whole scene, followed by the actual geometry.

Tip: Using the beacon of the camera (see Camera) as the beacon of a light source creates a headlight.

NOTE: Currently OpenSG does not implement the restricted influence area. All Light sources are global and light the whole scene. Expect that to change soon!

Every light is closely related to OpenGL's light specification. It has a diffuse, specular and ambient color. Additionally it can be switched on and off using the on field.

6.6.1 DirectionalLight

The DirectionalLight just has a direction.

To use it as a headlight use (0,0,-1) as a direction. it is the computationally cheapest light source. Thus for the fastest lit rendering, just a single directional light source. The osg::SimpleSceneManager's headlight is a directional light source.

6.6.2 PointLight

The PointLight has a position to define its location. In addition, as it really is located in the scene, it has attenuation parameters to change the light's intensity depending on the distance to the light.

Point lights are more expesinve to compute than directional lights, but not quite as expesive as spot lights. If you need to see the localized effects of the light, a point light is a good compromise between speed and quality.

6.6.3 SpotLight

The SpotLight adds a direction to the PointLight and a spotCutOff angle to define the area that's lit. To define the light intensity fallof within that area the spotExponent field is used.

Spot lights are very expensive to compute, use them sparingly.

Chapter 7

Drawables

Drawables are the leaf nodes of the graph. They do not use their children, even if they have some. Drawables contain the actual rendered geometry in the different forms it can be specified in.

7.1 Base Drawables

To simplify implementation there are two base Drawables that can be used as parent classes for derivation: osg::Drawable and osg::MaterialDrawable.

7.1.1 Drawable

The simple Drawable is the root class of all renderable NodeCores. It contains the Statistics element descriptors that can be used to collect information about renderable geometry.

7.1.2 MaterialDrawable

The MaterialDrawable contains a pointer to a osg::Material that is used to render the node. It is abstract, but the base for most renderable classes.

7.2 Geometry

Geometries in most cases define what's being rendered. Note that Geometries don't necessarily have to be leaves of the tree, as due the Node/Core divison every Node keeping a osg::Geometry Corehas children anyway, which are used just as all other osg::Node's children. Geometry has to be flexible, to accommodate the needs of the application. Different data types for the data that defines the geometry are useful, as well as different indexing capabilities to reuse data as much as possible. On the other hand, it also has to be efficient to render. Flexibility and performance don't always go well together, thus, there are some simplifications to make.

7.2.1 Properties

OpenSG geometry is modeled closely following OpenGL. The data that make up the geometry are stored in separate arrays. Positions, Colors, Normals and Texture Coordinates all have their own arrays, (or

osg::MField, to stay in OpenSG terminology). As OpenGL can handle a lot of different formats for the data, some of which might be more appropriate due to speed and memory consumption than others, depending on the application, OpenSG features different versions of this data, allowing pretty much all the variants that OpenGL can handle. To allow that with type safety and without having a separate geometry class for every possible combination the data fields are stored in separate field containers, a so called osg::GeoProperty. There are separate osg::GeoProperty for different attributes, and variants for different data types for each kind of osg::GeoProperty. The most prominent types are probably osg::GeoPositions3f for osg::Pnt3f positions, osg::GeoNormals3f for osg::Vec3f normals, osg::GeoColors3f for osg::Color3f colors and osg::GeoTexCoords2f for osg::Vec2f texture coordinates, but other variants are possible.

As properties only have a single field they can mimic that field by exposing parts of the standard osg::MField interface for their contents, so you can use a GeoProperty pretty much just like an osg::MField. One problem with the type variety is that writing functions that work on every type of property can become tedious, as you have to have a big switch for every kind of data that could arrive. To make that easier for every property there is defined generic format, e.g. for Positions the format is osg::Pnt3f. A property has a getValue()/setValue() interface for these generic types, i.e. every property, no matter in what format it stores the data, can be used as if it used the generic format. Of course, this is not as efficient as directly accessing the data, but if speed is not the highest priority or as a fall-back it's quite useful. And, finally, osg::GeoProperty features an interface for OpenGL vertex arrays, giving access to the data and the types involved, which is used for rendering.

In addition to the above-mentioned data there are some other osg::GeoProperty. OpenSG allows multiple primitive types per geometry, i.e. you can freely mix triangles, triangle strips and polygons in a single geometry node. The osg::GeoPTypes property defines the type of the primitives used. Right now, it only exists as a osg::GeoPTypesUI32 variant, but others may follow. The number of vertices per primitive is defined by another property, the osg::GeoPLengths property. This, too, only exists in a osg::GeoPLengths-UI32 variant right now.

7.2.1.1 Class Structure

The basic idea of the GeoProperty class structure is very simple. The actual implementation looks complicated, but is primarily so to simplify reuse of code and simplify extensions. If the ideas behind the structure are understood, the actual code doesn't look so bad any more.

All properties are derived from osg::Attachment, so they can directly be used as attachments and attached to an osg::AttachmentContainer.

There are two primary types of GeoProperties: osg::AbstractGeoProperty and osg::GeoProperty.

osg::AbstractGeoProperty describes the different kinds of attributes a Geometry can have: Positions, Colors, Normals, TexCoords, Indices, Types and Lengths. These are abstract, they have no specified data type and thus cannot be instantiated. They are the kinds of properties, not the actual properties. Use them in cases where you want to be able to use any actual type of data for a specific kind of property.

osg::GeoProperty describes the concrete versions of the properties. These are typed, can be instantiated, and these are the versions normally used by an application to set up their osg::Geometry node cores. All the concrete classes like osg::GeoPositions3f and osg::GeoTexCoords2f are typedefs of this class.

The actual interface to access the Properties has been factored out into separate classes, osg::GeoProperty-ArrayInterface and osg::GeoPropertyInterface. Both of these are supported by all kinds and types of properties, the distinction is made for future extensions.

osg::GeoPropertyArrayInterface defines a general, type-independent interface to all kinds of properties. It is very similar to the interface used by OpenGL's Vertex Arrays, i.e. it is possible to access the type, dimensionality and stride of the data, as well as getting access to the base pointer. This interface is primarily used for rendering.

osg::GeoPropertyInterface is the typed interface. Thus there are specific classes for every kind of property,

which use the GenericType of the specific kind in their interface. This interface is the one that models the osg::MField interface, it has functions to add values to and change values of the property.

Hint!

Some rules of thumb:

- Everywhere you actually want to create a new property you have to use the typed versions like osg::GeoPositions3f, as they are the only ones that actually contain data.

- To write functions that can handle arbitrary types of data, use abstract property pointers and the generic interface to access the data.

```
GeoColorsPtr col = GeoColorsPtr::dcast(geo->getColors());
if(col == NullFC)
{
    FWARNING(("Downcast failed!\n"));
    return;
}
beginEditCP(col, Geometry::ColorsFieldMask);
col->push_back(Color3f(1, 0, 1));
...
endEditCP(col, Geometry::ColorsFieldMask);
```

  This may incur some overhead, as the data might have to be converted to the actual data type of the property.

- If you know that all the geometry your function has to work on has been created yourself using a single type of property you can just downcast to that type and use the interface of the osg::MField that holds the data directly. For safety reasons you should make sure the downcast suceeded.

```
GeoColors3ubPtr col = GeoColors3ubPtr::dcast(geo->getColors());
if(col == NullFC)
{
    FWARNING(("Downcast failed!\n"));
    return;
}
MFColor3ub *c = col->getFieldPtr();
beginEditCP(col, Geometry::ColorsFieldMask);
c->push_back(Color3ub(255, 0, 255));
...
endEditCP(col, Geometry::ColorsFieldMask);
```

  This is the most efficient way to access the data.

- The most general is the osg::GeoPropertyArrayInterface. If you want to be able to support all kinds of osg::Geometry properties this is the way to go. It doesn't differentiate between property types, so this is as general as it gets. There are probably not many situations that need this kind of generality, the renderer and some very generic osg::Geometry optimizations like the osg::createSharedIndex and osg::createSingleIndex are the only places where it's used right now.

7.2.2 Indexing

Using these properties it is possible to define geometry. Note that OpenSG inherits the constraints and specifications that concern geometry from OpenGL. Vertex orientation is counterclockwise when seen from the outside, and concave polygons are not supported.

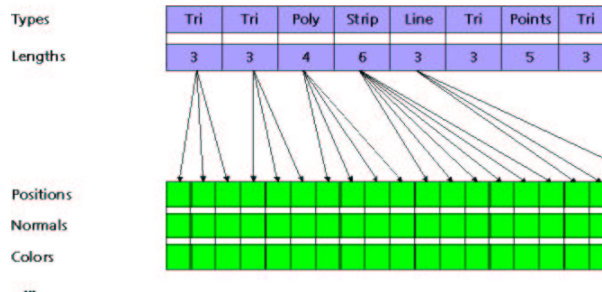

Figure 7.1: Non-Indexed Geometry

One additional advantage of separating properties from Geometry is the ability to share properties between geometry osg::NodeCore s. As geometries can only have one material right now that's useful for simplifying the handling of objects with multiple materials.

This simple geometry has one problem: there is no way to reuse vertex data. When a vertex is to be used multiple times, it has to be replicated, which can increase the amount of memory needed significantly. Thus, some sort of indexing to reuse vertices is needed. You can guess what's coming? Right, another property.

Indices are stored in the osg::GeoIndices property, which only exists in the osg::GeoIndicesUI32 variant right now. When indices are present the given lengths define how many indices are used to define the primitive, while that actual data is indexed by the indices.

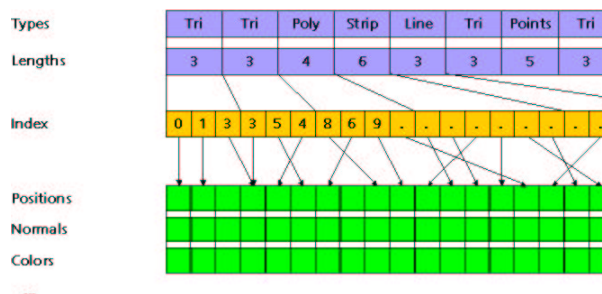

Figure 7.2: Indexed Geometry

Indexed geometry is very close to OpenGL, and probably the most often used type of geometry. It doesn't handle all the cases, though.

Sometimes vertices need different additional attributes, even though they have the same position. One example are discontinuities in texture coordinates, e.g. when texturing a simple cube. The edges of the cube don't necessarily use the same texture coordinate. To support that a single indexed geometry has to replicate the vertices.

To get around that you need multiple indices per vertex to index the different attributes. Adding an index for every attribute would blow up the geometry significantly and not necessarily make it easier to use. We

decided to use another way: interleaved indices.

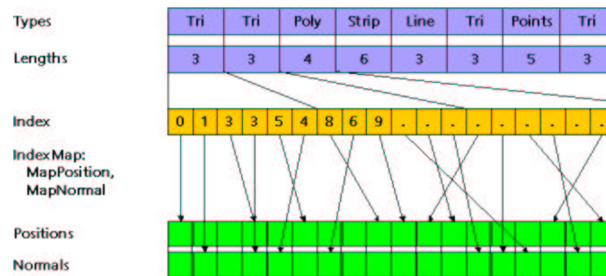

Figure 7.3: Multi-Indexed Geometry

Interleaved indices require every vertex to hold multiple indices. Which index is used for what attribute is defined by a separate indexMapping field. The indexMapping field is a osg::UInt32 osg::MField. The possible values are bitwise combinations of the available attribute masks: osg::Geometry::MapPosition, osg::Geometry::MapNormal etc. The length of the indexMapping defines how many indices are used per vertex. If it's not set a single index for all available properties is used (or none at all).

In addition to the properties geometry keeps a osg::MaterialPtr to define the material that's used for rendering the geometry (see Materials) and a flag that activates caching the geometry in OpenGL display lists. As geometry rendering is not optimized very much right now that's the best way to get decent performance. Display lists are turned on by default.

7.2.3 Geometry Iterators

The osg::Geometry setup is very nice and flexible to define: you can mix different kinds of primitives in an object, you can have properties and different kinds and the indexing allows the reuse of some or all of the data.

From the other side of the fence things look different: if you want to walk over all triangles of a geometry to calculate the average triangle size or the surface area, or for calculating face normals or for whatever reason you have to take care of all the flexibility and be prepared for lots of different ways to define geometry.

To simplify that the concept of a geometry iterator has been introduced. A geometry iterator allows to iterate over a given geometry primitive by primitive, face by face (a face being a triangle or quad), or triangle by triangle.

All of them are used like STL iterators: the osg::Geometry has methods to return the first or last+1th iterator, and to step from one element to the next. They can also unify the different indexing variants: when using an iterator you can access the index value for each attribute of each vertex of the iterator separately. Or you can directly access the data that's behind the index in its generic form, which is probably the easiest way of accessing the data of the osg::Geometry.

Example: The following loop prints all the vertices and normals of all the triangles of a geometry:

```
for(it = geo->beginTriangles(); it != geo->endTriangles(); ++it)
{
    std::cout << "Triangle " << it.getIndex() << ":" << std::endl;
    std::cout << it.getPosition(0) << " " << it.getNormal(0) << std::endl;
    std::cout << it.getPosition(1) << " " << it.getNormal(1) << std::endl;
    std::cout << it.getPosition(2) << " " << it.getNormal(2) << std::endl;
}
```

If you're used to having a separate Face object that keeps all the data for a face, the Iterators pretty much

mimic that behavior. The one thing you can't do using iterators is changing the data. To do that you have to use the Indices the Iterators give you and access the Properties directly. Be aware that the Iterators hide all data sharing, so manipulating data for a face the iterator gives you can influence an arbitrary set of other faces.

7.2.3.1 Primitive Iterator

The osg::PrimitiveIterator is the basic iterator that just iterates through the osg::GeoPTypes property and gives access to the primitive's data. It is useful to solve the index mapping complications and to get access to the generic data, but it's primarily a base class for the following two iterator types.

7.2.3.2 Face Iterator

The osg::FaceIterator only iterates over polygonal geometry and ignores points, lines and polygonal primitives with less than three vertices. It also splits the geometry into triangles or quads.

7.2.3.3 Triangle Iterator

The osg::TriangleIterator behaves like the FaceIterator, but it also splits Quads into two triangles, thus it does an implicit triangulation. As OpenSG just like OpenGL doesn't support concave geometry that's not as hard as it sounds.

The iterators can also be used to indicate a specific primitive/face/triangle. Each of these has an associated index that the iterator keeps and that can be accessed using getIndex(). A new iterator can be used to seek() a given primitive/face/triangle again and work on it. This is used for example in the osg::IntersectAction.

7.2.3.4 Simple Geometry

OpenSG does not have NodeCores for geometric primitives like spheres, cones, cylinders etc. Instead there are a number of utility functions that can create these objects. They can be created as a ready-to-use node and as a naked node core. In most cases we tried to mimic the VRML primitive semantics, so if you're familiar with VRML you will feel right at home.

Plane osg::makePlane creates a single subdivided quad.

Box osg::makeBox creates a box around the origin with subdivided sides.

Cone osg::makeCone create a cone at the origin.

Cylinder osg::makeCylinder create a cylinder at the origin.

Torus osg::makeTorus create a torus at the origin.

ConicalFrustum osg::makeConicalFrustum creates a truncated cone at the origin.

Sphere There are two ways to create a sphere. osg::makeSphere uses a icosahedron as a base and subdivides it. This gives a sphere with equilateral triangles, but they do not correspond to latitude or longitude, which makes it hard to get good texture mapping on it. As every subdivision step quadruples the number of triangles, it is also hard to control the complexity of these kinds of spheres.

osg::makeLatLongSphere on the other hand creates a sphere by simply usign a regular subdivision of latitude and longitude. This creates very small polygons near the poles, but is more amendable to texture mapping and gives finer control of the resoltuion of the sphere.

Extrusion Geometry osg::makeExtrusion creates a pretty general extruded geometry. It works by sweeping a given cross section, which can be given clockwise or counterclockwise, across a spine. For every spine point an orientation and a scale factor are specified. The beginning and the end of the object can be closed by caps, but for the capping to work the cross section has to be convex. The resulting geometry can be refined as a subdivision surface (no idea which subdivision scheme is applied, anyone care who knows care to take a look?). Optionally normals and texture coordinates can be generated

7.2.3.5 Helper Functions

A number of helper functions can be used in conjunction with manipulating and optimizing geometry.

Normal Calculation A common problem for self-created geometry or for geometry loaded from simple file formats are missing normals. Normals are needed for proper lighting, without them objects will either be black or uniformly colored.

Normals can be calculated either for every face or for every vertex.

Face normals, as calculated by osg::calcFaceNormals, are only unique for a given triangle or quad. The resulting object will look faceted, whcih may or may not be the desired effect. This will also work for striped or fanned models, as OpenSG doesn't have a per-face binding and uses multi-indexed per-vertex normals for this.

Vertex normals are calculated for every vertex and allow a shape to look smooth, as the lighting calculation is done using the vertex normals and interpolated across the surface. They can be calculated using two different methods.

osg::calcVertexNormals(GeometryPtr geo) will just average all the normals of the faces touching a vertex. It does not unify the vertices, i.e. it does not check if a vertex with given coordinates appears in the position property multiple times, the geometry has to be created correctly or be run thrugh osg::createSharedIndex.

The disadvantage of osg::calcVertexNormals(GeometryPtr geo) is its indiscriminative nature, it will average out all the edges in the object. The alternative is osg::calcVertexNormals(GeometryPtr geo, Real32 creaseAngle), which uses a crease angle criterion to define which edges to keep. Edges that have an angle larger than *creaseAngle* will not be averaged out. It won't always work for striped geometry. It will process it, but if a stripe point needs to be split because it has two normals, that won't be done. The same sharing caveat as given above applies.

Calculating vertex normals with a crease angle sounds simpler than it is, especially if the calculation should be independent of the triangulation of the object. Thus the algorithm is relatively expensive and should be avoided in a per-frame loop. There are some ideas to do the expesive calculations once and quickly reaverage the normals when needed. These have not been realized, if you need this or even better want to implement it, notify us at `info@opensg.org`.

Geometry Creation Setting up all the objects needed to fully specify an OpenSG Geometry can be a bit tedious. So to simplify the process there are some functions that take data in other formats and create the corresponding OpenSG Geometry data.

Right now there is only one function to help with this, osg::setIndexFromVRMLData. It takes separate indices for the different attributes, as given in the VRML97 specification, together with the flags that influence the interpretation of these indices, and sets up the indices, lengths and types properties of the given geometry.

Geometry Optimization OpenSG's Geometry structure is very flexible and pretty closed modeled on OpenGL. But not all of the Geometry data specification variants are similarly efficient to render. The functions in this group help optimizie different aspects of the Geometry.

osg::createOptimizedPrimitives takes a Geometry and tries to change it so that it can be rendered using the minimum number of vertex transformations. To do that it connects triangles to strips and fans (optionally). It does not change the actual property values, it just creates new indices, types and lengths. The algorithm realized here does not try a high-level optimization, instead it is optimized for speed. Due to its pseudo-random nature it can be run multiple times in the same time a more complex algorithm needs, allowing it to try different variants and keeping the best one found. Or, if execution time is a problem, it can be run only once and create a very quick result that is good, but not optimal.

osg::createSharedIndex tries to find identical elements in the Geometries Properties and remove the copies. It will not actually change the Property data, it will just change the indexing to only use one version of the data. This is a necessary preparation step to allow osg::createOptimizedPrimitives to identify the triangles it can connect to form stripes and fans.

osg::createSingleIndex resorts the Geometry's Property values to allow using a single index (in contrast to interleaved multi-indices) to represent the Geometry. To do that it might have to remap and copy Property values, as well as index values. While multi-indexing can be very efficient datawise, as as much of possible is shared, for rendering it is problematic. OpenGL doesn't know multi-indexing, thus for multi-indexed Geometry the more efficient OpenGL geometry specifiers like VertexArrays can't be used, which can have a significant impact on performance, especially for dynamic objects.

UInt32 calcPrimitiveCount (GeometryPtr geo, UInt32 &triangle, UInt32 &line, UInt32 &point);

Normal Visualisation For debugging it can be useful to actually see the normals of an object, as that allows making sure that the normals point in the expected direction and that normals are really identical and not just pretty close. Every normal is represented by a line of a user-defined length.

As OpenSG doesn't have an explicit face/vertex binding mode there are two different functions to create an object representing the vertex or face normals. The application should know whether face or vertex normals are used in the given geometry. In general it is safe to assume vertex normals are used.

osg::calcVertexNormalsGeo creates an object that shows the vertex normals of the given geometry, while osg::calcFaceNormalsGeo creates an object that shows the face normals.

7.3 Slices

Slices is a simple volume rendering core. The Slicer just creates and renders viewer-oriented slices from back to front through a rectangular volume, which is centered around the origin and specified using the size in the three dimensions. The numberOfSlices parameter defines the number of slices which should be utilizes along the diagonal. Therefore it is more the max number of slices rendered and should be set to match the texture sampling rate. The slice distance is uniform for different camera positions which is necessary to minimize the re-sampling artifacts.

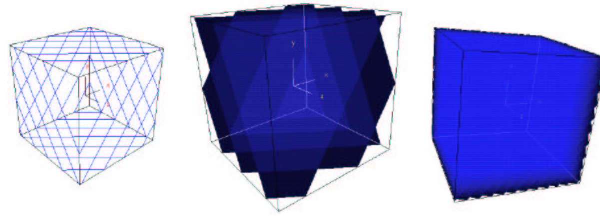

Figure 7.4: Polygon slices using various slice numbers

The core depends on the material to setup a 3D texture and BlendMode. You have to provide the voxelgrid-data as 3D-Texture. Check out the testSlicer programm as example.

7.4 Particles

The main idea of particles is to give a way of easily rendering large numbers of simple geometric objects. Particles are mostly used together with partially transparent textures to simulate fuzzy objects, but other uses are possible, e.g. molecules as stick-and-sphere models, stars or arrows for flow-field visualisations.

The Particles core can display different styles of particles and keeps all the data needed to do all of them. Not every style uses all the data, and using simpler setups can result in vastly improved performance.

The supported styles are:

- Points
- Lines
- ViewDirQuads
- Arrows
- ViewerArrows

The available attributes are:

- Positions
- secPositions
- Colors
- Normals
- Sizes
- textureZs
- Indices

The first 4 are realized similarly to the Geometry Properties, to allow unified manipulation of data and sharing with geometry nodes. They can handle the same data formats the geometry can handle.

The only attribute every particle needs is the position. Some styles need other attributes, too, but most are optional. Colors, Normals, Sizes and textureZs can either be left empty, have a single element or have as

many elements as there are positions. If only a single element is given, it is used by all particles, otherwise every particle has its own element.

The different types interpret the data as follows:

osg::Particles::ModeE::Points are rendered as simple GL_POINTS. Points use the positions, colors and textureZs attributes. A single size can (and should!) be given, but size per particle is not supported (yet).

osg::Particles::ModeE::Lines are two-point lines between a position and the corresponding secPosition, i.e. there have to be as many positions as secPositions. Colors and texture coordinates are used, too. A single size can (and should!) be given, but width per line is not supported (yet).

osg::Particles::ModeE::ViewDirQuads draw a single quad at the given position with the given size, color and texture Z. Only the X component of the size is used, and the given texture is used to fill the quad totally. The orientation of the quad is chosen to be parallel to the viewing direction, i.e. particles that are to the side of the viewer will be seen as a line.

osg::Particles::ModeE::ViewerQuads use just the same parameters as ViewDirQuads, but the particles are oriented to turn towards the viewer position, i.e. they always face the viewer, no matter where they are.

osg::Particles::ModeE::Arrows draws simple flat arrows. The coordinate system of the arrows is defined by the position, the secondary position and the normal. The tip of the arrow is at the position, the axis of the arrow points to the secondary position and the it is perpendicular to the normal. There need to be positions and secondary positions for every arrow, a single normal is possible, or a separate normal for every arrow.

There is also a single material that is used to draw the particles, and an attribute to change the draw order of the particles.

Chapter 8

State Handling

One of the main tasks of a scene graph system is efficiently managing the OpenGL state.

Every primitive is rendered using the currently active OpenGL state, which includes things like material parameters, textures, transformation matrices etc. Changing OpenGL state can be very expensive and should be minimized as much as possible. Primitives using the same OpenGL state set should be grouped together, so that changing the state is not needed.

But on top of that it is necessary to sort the different state sets so that the changed when changing from one to the other are minimized, too. For example all objects using the same texture should be rendered close together, as loading a texture into the graphics board can take a long time.

Thus the different state sets used by the visible objects should be sorted into a sequence that minimizes the state changes necessary. Unfortuneately that problem is NP-complete (it's equivalent to the Traveling Salesman Problem). To simplify it the rather large OpenGL state is divided into a small number of parts. Elements of the state that are usually changed together are grouped into a osg::StateChunk.

Every type of osg::StateChunk has an associated osg::StateChunkClass. The osg::StateChunkClass is used to identify the osg::StateChunk and associate it with a name and a low integer numeric Id. Some types of chunks can be used simulatenously in multiple instances in OpenGL, e.g. light sources or textures (in a multi-texture environment). The maximum number of concurrently active slots of the osg::StateChunk type is also stored in the osg::StateChunkClass.

The complete OpenGL state wrapper is the osg::State. It primarily contains a vector of osg::StateChunks. For every type of chunk there are as many slots as possibly concurrently active copies of the chunk. A chunk can be Normal Visualisation ::activate "activated", which sets the OpenGL state it covers to its settings, if the settings differ from an internally defined default state. It is possible to Normal Visualisation ::change-From "switch" from one instance of a state chunk to another instance of the same type. This tries to optimize and minimize the changes, to speed up switching. The cost of switching can be Normal Visualisation ::switchCost "estimated", currently that estimation will always be 0, though. Finally a chunk can be Normal Visualisation ::deactivate "deactivated", which resets the OpenGL state it covers to the default value. All the chunks in a osg::State can be managed at the same time by using the equivalent methods of the osg::State.

Hint! If you want to use state chunks in the scene graph, you can attach them to a osg::ChunkMaterial or one of its descendents.

Every chunk has a number of parameter which are pretty directly mapped to OpenGL parameters. Thus in many cases OpenGL constants are used to define different parameter values and enumerations. This allows usage of some OpenGL extensions directly by supplying the correct constants.

OpenGL uses explicit enabling in most situations. To get around having to keep extra variables for enabling the value GL_NONE is used in many places to indicate a disabled feature. The documentation of the chunk

notes where this is possible.

The different types of state chunks are:

- BlendChunk
- ClipPlaneChunk
- CubeTextureChunk
- LineChunk
- PointChunk
- LightChunk
- MaterialChunk
- PolygonChunk
- RegisterCombinersChunk
- TexGenChunk
- TextureChunk
- TextureTransformChunk
- TransformChunk
- ProgramChunk
  - VertexProgramChunk
  - FragmentProgramChunk

8.1 BlendChunk

The osg::BlendChunk handles OpenGL blending, i.e. the definition how incoming fragments are combined with the pixel already in the frame buffer, including alpha culling.

The wrapped OpenGL functions are glBlendFunc and glAlphaFunc, see their documentation for details. It also handles the common blending-related OpenGL extensions EXT_blend_color, ARB_imaging, EXT_blend_subtract, EXT_blend_minmax and EXT_blend_logic_op, when they are supported by the hardware.

8.2 ClipPlaneChunk

The osg::ClipPlaneChunk controls user-defined clipping. It uses a beacon osg::Node reference to control the cordinate system where the clipping plane is defined, this allows attaching the ClipPlane to another object to move it around. The clipping plane itself is defined in the standard (a,b,c,d) form used by OpenGL. 6 user defined clipping planes are possible.

8.3 CubeTextureChunk

The osg::CubeTextureChunk is similar to TextureChunk and uses the same osg::StateChunkClass (i.e. can be used instead of a osg::TextureChunk, but not both at the same time), but has 5 more texture images. Note that all textures have to be square and have to have the same resolution. The textures are accessed using 3D texture coordinates, which are usually created using a osg::TexGenChunk.

Cube textures are an extension that is only available in newer hardware. They can not be emulated, thus they are ignored when they are not supported by the active window.

8.4 LightChunk

The osg::LightChunk contains the parameter set for a single light source. It's parameters are taken straight from the glLight() manpage. The maximum number of concurrently active lights is currently set to 8.

Note that these chunks are created by the system internally from the osg::Light sources and shouldn't be directly used by an application.

8.5 LineChunk

The osg::LineChunk contains the parameters that are specificly set for lines. This includes line width, stippling and antialiasing.

8.6 PointChunk

The osg::PointChunk contains the parameters that are specific set for points. This includes point size and point antialiasing.

It also wraps the ARB_point_parameters and NV_point_sprite extensions.

ARB_point_parameters allows the specification of points whose size changes depending on the distance to the viewer, including the mapping of sizes smaller than a pixel to alpha. This is useful for objects rendered by points, e.g. particle systems or the so-called light point for runway lights in flight simulation.

The actual size of the point is derived from its given size, which is devided by a the reciprocal of a quadratic expression of the distance to the point in eye coordinates (sepcified by constantAttenuation, linearAttenuation and quadraticAttenuation). This size is then clamped into the minSize, maxSize range and possible clamped against OpenGL's internal size constraints. If the calculated size was smaller than the minSize the point's alpha is reduced proportionally.

8.7 MaterialChunk

The osg::MaterialChunk controls the material parameters, i.e. the parameters for phong lighting as used by glMaterial(). It also covers the commonly used parameters to enable/disable lighting and switching the influence of geometry colors on lighting (glColorMaterial). When lighting is enabled external colors like osg::Geometry colors are only used for the diffuse lighting component. In the unlit case they are always used.

8.8 PolygonChunk

The osg::PolygonChunk contains the parameters that are specific set for filled surfaces, i.e. polygons. This includes face culling and front face definition as well as front and back face rendering modes, polygon antialiasing, offset and stippling. As there is only one set of offset parameters for all the primitives, the offsetting for points and lines is also handled in this chunk, which admittedly is a bit awkward.

8.9 RegisterCombinersChunk

The osg::RegisterCombinersChunk chunk is a direct mapping of the nVidia RegisterCombiners extension. It is based on the GL_NV_register_combiners2 extension, i.e. it also supports per-stage constants.

8.10 TexGenChunk

The osg::TexGenChunk wraps texture coordinate generation functions. The texture coordinate generation for all 4 coordinates is wrapped in a single chunk, including the optional plane parameters.

8.11 TextureChunk

The osg::TextureChunk contains a single texture image and it's related parameters, which include the filters and texture environment mode and color. 1,2 and 3 dimensional textures are handled by this chunk uniformly and textures can also automatically be scaled to the next power of two (for 2D textures). It can also handle a multi-frame osg::Image by selecting one of the frames. If necessary (i.e. if mipmappibng filters are used) mipmaps will be calculated automatically. *Hint!* Don't use mipmaps for fast changing textures (i.e. movies), as mipmap generation takes a lot of time.

Multiple texture chunks (right now 4) can be used simultaneously for multi-texturing.

Hint! To do multi-texturing in the absense of a specific multi-texture material you can just append additional texture chunks to a osg::ChunkMaterial or one of its descendents.

8.12 TextureTransformChunk

Chunk for texture coordinate transformations, uses a simple matrix to transform texture coordinates.

Multiple texture transform chunks (right now 4) can be used simultaneously for multi-texturing.

8.13 TransformChunk

Chunk for transformations, uses a simple matrix to transform coordinates.

Note that these chunks are created by the system internally from the osg::Transform and osg::ComponentTransform cores and shouldn't be directly used by an application.

8.14 ProgramChunk

The OpenGL ARB has added two programmability extensions to the OpenGL core, which allows the users to replace the built-in vertex or fragment pipeline with their own programs. These extensions are extremely similar, and thus the common features have been implemented in an abstract base chunk, the osg::ProgramChunk.

The program is just a string, which can be directly set by the application, as a convenience it can be read from a file. It will only be compiled (and consequently checked for error) when it used for rendering the first time.

Depending on the type of program it has a differing set of specific parameters to work on, but they all have the ability to pass specific parameters from the outside. These have osg::Vec4f values and are just identified by an index. Usually they will be assigned to a named variable in the program to signify their meaning. To make their use easier to understand they can also be named in OpenSG, and can be accessed by their name. OpenSG does not (yet) try to find the name to index mapping automatically, the application is responsible for maintaing the correspondence here. The parametrs set in OpenSG are passed to the porgam as the ProgramLocalParameters. There is currently no way to change the ProgramEnvironmentParameters in OpenSG, as there is no central management instance.

8.15 VertexProgramChunk

The osg::VertexProgramChunk wraps the GL_ARB_vertex_program extension. The extension is much too big to be reproduced here (∼100 pages), see the `specification` for details.

8.16 FragmentProgramChunk

The osg::FragmentProgramChunk wraps the GL_ARB_fragment_program extension. This extension is also much too big to be reproduced here (∼100 pages), see the `specification` for details.

Chapter 9

Materials

Materials define the surface properties of the geometry. For the standard Phong lighting model that OpenGL uses these are ambient, diffuse and specular color as well as shininess. However, this is an area where extensions are added at an amazing pace. The purpose of materials is to add a level of abstraction and give the user an easy to use interface to define surface properties without having to worry about the details of realizing them (e.g. how to use multiple passes, when and how to calculate derived information like tangents and binormals etc.).

This area is quickly expanding, so what we have right now is just the beginning.

9.1 Material types

9.1.1 ChunkMaterial

The osg::ChunkMaterial is a material that is just a collection of osg::StateChunks . This allows adding and using new extensions in the form of state chunks pretty easily.

It is also the base class for the simple materials given below.

9.1.2 SimpleMaterial

The osg::SimpleMaterial is a pretty direct mapping of the OpenGL light model. It has colors for ambient, diffuse, specular and emission properties, and a shininess value. In addition to that it has a transparency setting, ranging from 0 for opaque to 1 for fully transparent.

There are two other attributes in a SimpleMaterial that control the appearance of an object. One is the lit attribute, which defines if the material is influenced by light sources at all. If it isn't, the color is directly taken from the diffuseColor component and other color attributes are ignored.

The other attribute is the colorMaterial field, which defines how colors that are given in the geometry influence the lighting calculation. By default they replace the diffuse color only. Possible values are taken from the glColorMaterial() call, the most useful being GL_DIFFUSE_AND_SPECULAR. One possible value that is not used by glColorMaterial() is GL_NONE, which switches off the color material handling and thus ignores colors that are given in the geometry.

As SimpleMaterial is derived from osg::ChunkMaterial, other attributes can be added in the form of StateChunks .

9.1.3 TexturedSimpleMaterial

osg::SimpleTexturedMaterial is derived from osg::SimpleMaterial and adds a texture. The texture is defined by an image (see Image for details on how to define or load an image).

Additionally there are some parameters to define the behavior of a texture. magFilter and minFilter define how to scale the texture image up or down, legal values taken from glTexParameter(). The most useful ones are GL_NEAREST or GL_LINEAR for magFilter, and additionally GL_LINEAR_MIPMAP_LINEAR for minFilter.

envMode defines how a color from the texture is combined with a color from the lighting calculation. The default is GL_REPLACE which completely ignores the lighting color. Other useful values are GL_-MODULATE, which just multiplies the two, and GL_DECAL, which interpolates between lighting and texture based on the texture's alpha channel.

Finally, a texture can be used as a spherical environment map to simulate a reflective object by setting the envMap field to true. Spherical environment maps need to display the image of a reflective sphere in the middle of the environment that is being reflected.

As TexturedSimpleMaterial is derived from osg::ChunkMaterial, other attributes can be added in the form of StateChunks .

Chapter 10

Action

Creating the scene-graph is just the first step, and not really useful in itself. Something needs to be done with it. Actions on the graph usually take the form of a traversal which goes through the nodes one by one and calls an appropriate action for each one on the way.

These are called Actions in OpenSG, and there are a number of predefined actions:

- Render
- Intersect

10.1 Usage

Actions use the same syntax for creating as actions that FieldContainers use, as they also use a prototype for that. Thus you need to call ActionType::create to get a new one. They are not FieldContainers, though, so simple pointers are OK.

To execute an action on a graph you apply it to the graph (action->apply(graph);) or to a list of nodes (action->apply(vector<NodePtr>::iterator begin, vector<NodePtr>::iterator end);).

10.2 RenderAction

RenderAction is the primary means of transforming the scene-graph into an image. It does view volume culling and state sorting by building a draw tree. It also handles transparent objects by rendering them last and back to front sorted. Put simple it does what a decent scene-graph needs to do.

To use it just create one and pass it to the OpenSG Window object (see Window).

It is possible to turn the view volume culling off using the setFrustumCulling() method. For debugging it is possible to turn the frustum update off (setAutoFrustum()) and to make the system render the tested bounding volumes (setVolumeDraw()).

10.3 IntersectAction

IntersectAction is used for sending rays into the scene and retrieving the first object hit. Right now, intersection testing is not very optimized, which is OK for selecting an object, but probably too slow for programmatic use.

A ray is defined by a Line (see ::Line) and optionally a maximum distance. It can either be set at construction time or by setLine(). To test the ray for intersection, apply the action to the root of the possible intersection objects.

If the ray hits an object didHit() will return true. In that case, detailed info about what was hit and where can be accessed through getHitT(), getHitPoint(), getHitObject() and getHitTriangle().

10.4 Simple Traversal

Actions are somewhat complicated to derive and, furthermore, they manage callback functors on a NodeCore basis. Sometimes it's easier to just define a function that is called for every node in a graph. That's what traverse() is for.

traverse() takes a NodePtr to define the graph and a functor to define the function to be called for every node as parameters. The functor just gets the traversed node as a parameter

10.5 Write your own action handler

Don't. Actions are being completely redesigned for 1.1 to become more flexible and clean. Use the available actions and try to stay with the traverse() function for now.

If you really need to do your own action take a look at IntersectAction, it shows what you need to implement. Talk to us before you do it, though, maybe the redesign is already usable so you can base your new stuff on that.

Chapter 11

Window

Windows are the general connection between OpenSG and the windowing system used by the application/platform. OpenSG itself does not open its own windows, that has to be done by the application. Using GLUT it's pretty trivial, take a look at the tutorial examples on how to do that. For other window systems its a little more work, but the goal is to have wrapper classes for the usual GUI toolkits like QT, Motif etc. that simplify the task. We have one for QT, OSGQGLManagedWidget, and are interested in similar ones for other Window systems.

11.1 Window

A osg::Window is the connection between OpenSG and the window system used. There are variants for different supported window systems like X, WIN32, GLUT and QT. The OpenSG Window object handles OpenGL context creation and activation/deactivation, and needs to be informed about resizes. It manages OpenGL objects like display lists and texture objects and is also responsible for OpenGL extension detection and functions.

It doesn't do any input event handling or similar things, it's only for output and keeping the osg::Viewport instances that fill the window and keep all the rendering parameters. See SimpleSceneManager for an easy-to-use wrapper for setting these up.

There should be a 1:1 relation between an osg::Window and a window-system window. As such the osg::Window is responsible for handlng the OpenGL related tasks such as OpenGL object and extensions management (see OpenGL Objects & Extension Handling for details on that).

11.1.1 Passive Window

The osg::PassiveWindow is a helper class for integration into other OpenGL programs. It does not manage its own OpenGL context, it expects the context to be active whenever it is called. It also ignores swap commands, as these are window system-specific. Thus the osg::PassiveWindow can be used in any window system and GUI system to simplify integration. If the same OpenGL context is active whenever it is initialised/called it will work nicely.

11.2 Viewport

A osg::Viewport is a part of the window that is being rendered into. It can cover the whole window, but doesn't have to. Every Window can handle an arbitrary number of viewports, e.g. to allow the typical front/left/right/perspective views in a single window. Every osg::Viewport can only be attached to one osg::Window.

The size of the viewport is defined by its left, right, bottom and top coordinates, given in OpenGL conventions, i.e. the bottom of the screen has the vertical coordinate 0. If the value is bigger than 1, it's a position in pixel. That's independent of the window size, if the window is smaller, parts of the viewport will be cut, if it's bigger parts of the window will not be covered. If they are between 0 and 1 (inclusively) they are relative to the window and are rescaled when the window is resized. If they are -1 they use the extreme applicable value, i.e. 0 for left and bottom, 1 for right and top. For relative sizes the actual value used for right and top is value $*$ size - 1. This allows abutting viewports by using the same relative values for right and left of the viewports that should fit. See the image for an example.

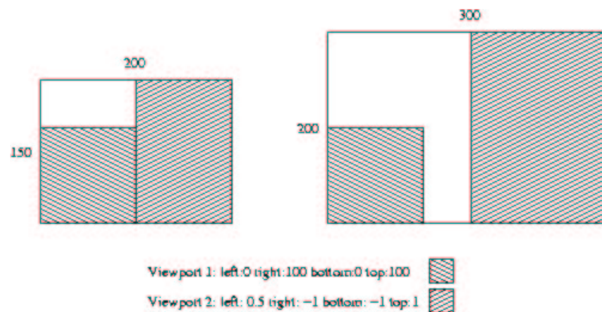

Figure 11.1: Viewports in differently sized windows

An exception to this size definition is the osg::PassiveViewport. It is meant to be used for integrating OpenSG into other applications and takes the size from the currently active OpenGL viewport and thus ignores its settings. This is pretty extreme, in most cases it will probably be enough to use a osg::PassiveWindow and set the Viewport parameters to not interfere with whatever other OpenGL rendering is taking place.

To define what is being rendered a viewport stores the root osg::Node of the scene graph to be displayed, the osg::Camera, the osg::Background and osg::Foreground instances to use. Of these, only the osg::Foreground is optional, all others are needed to draw or render the osg::Viewport.

There are some special kinds of viewports for specific purposes. The default osg::Viewport is for single buffered rendering, the osg::StereoBufferViewport is used for quad-buffer stereo rendering and the osg::ColorBufferViewport allows rendering to specific color channels only.

Hint!

The osg::StereoBufferViewport only works if the selected window actually has the four buffers. So make sure that your graphics card supports quad-buffer rendering (i.e. has Visuals that are stereo) and that the selected Visual is one of those.

The osg::PassiveViewport is totally passive. It ignores its set size and position and instead takes the current OpenGL settings. This allows pretty simple integratino into other OpenGL programs, but the OpenGL context must be active and the viewport set correctly for it to work.

11.3 Camera

A osg::Camera defines the parts of the scene that are actually being rendered. The definition can be split in two parts: position and orientation, and internal parameters.

Position and orientation of the camera are defined by a node in the scene graph, a beacon, similarly to the definition used by light sources. The camera uses the OpenGL defaults for specifying the used coordinate system, i.e. the camera looks along the negative Z coordinate, X points to the right and Y is up. Thus, to use a camera you need a beacon node in the scene to define its position. This can be an object you want to attach the camera to, but in general you'll probably have a Transform node somewhere close to the root to handle it.

This gives full flexibility to use a simple matrix to define camera position and orientation, but can be tedious to specify. Many systems use a from-at-up convention to define camera parameters, i.e. you specify a viewer position, a point that should be in the center of the screen and the direction that should be up on the screen. The osg::MatrixLookAt functions from OSGMatrixUtility.h can convert these settings into a matrix that can directly be used to specify the camera.

The internal parameters of the camera can vary between different kinds of cameras. The only constant thing that a camera for OpenGL needs are the near and far clip distances, which are defined in the general Camera class. The others are defined in the specific camera classes.

The camera parameters needed for rendering are split into three matrices. The first is for viewing, representing teh viewing part of the OpenGL GL_MODELVIEW matrix. The GL_PROJECTION matrix is split into two. The first is the real projection matrix, the second is a projection transformation matrix. The projection transformation is used to add a coordinate system for non-coplanar multi-screen projection setups like CAVEs. See "High-Quality High-Performance Rendering for Multi-Screen Projection Systems" by Dirk Reiners, in Proceedings of the 3 rd International Immersive Projection Technology Workshop for a motivation of this additional coordinate system.

The camera has a bunch of functions to access its parameters and derived values like its viewing frustum, and also to calculate rays through viewport pixel for picking etc.

11.3.1 Perspective Camera

osg::PerspectiveCamera is the standard camera used for OpenGL rendering. The only additional attribute it has is the vertical field of view, in radians. The horizontal field of view is automatically adjusted to the window size to create a square aspect ratio.

11.3.2 Matrix Camera

The osg::MatrixCamera is a very low-level camera class that keeps the ModelView and Projection matrices directly. It ignores all other parameters it has and just uses the given matrices.

11.3.3 Camera Decorators

A osg::CameraDecorator is object that can be used instead of a osg::Camera and replaces/enhances some of its functionality. It is an implementation of the standard Decorator pattern (see the Design Patterns book), in which a special object, the decorator, is placed between the original object and a user of the original object.

The main idea is to allow enhancing/manipulating arbitrary cameras for new features with minimal impact on applications. The osg::ShearedStereoDecorator for example allows using any camera, whether it's a perspective, orthographic or whatever camera, to control a stereo projection. Todo that it is only necessary

to create two viewports, one for the left and right eye, and use two decorators to decorate the single camera for the left and right eye. Changes to the camera's parameters can be done just like in a single view application and automatically influence all affected viewports, and the difference neccessary to create the different images for the left and right eye are managed by the decorators.

11.3.3.1 Tiled Rendering

The osg::TileCameraDecorator is used to select a rectangular part of the image and scale it to the full viewport. The primary application area is distrbuting the rendering of a large image over several screens. The viewing parameters can be defined as if it was a single large viewport, and the osg::TileCamera-Decorator will pick the part it needs to render. This is different from having multiple independent cameras positioned side by side, as the center of projection needs to be the same for all parts of the image calculated.

The part of the image displayed is defined by its left/right/bottom top coordinates and the width/height of the full image.

Hint!

Pixel-based osg::Background instances like Gradient Background will not work with the osg::TileCamera-Decorator, as they directly access the physical viewport right now.

11.3.3.2 Stereo

An interesting applications of CameraDecorators is the use to generate the left/right image pairs that are needed to display three-dimensional stereo images.

The osg::StereoCameraDecorator is the base class for all the Decorators that are able to do this. It keeps the generic parameters that are needed for every stereo image: the eye separation and the left/right eye distinction.

The eye separation defines the distance between the two eyes, given in the units used in the models, not some global unit like centimeters.

The left/right eye distinction is handled by a simple bool, which is true for the left eye decorators, and false for the right eye's.

The actual algorithm to generate the images is defined by the specific decorator.

Sheared Stereo The osg::ShearedStereoCameraDecorator can be used to generate the left/right image pair needed for stereoscopic displays. It uses the sheared frustum stereo model, which is most appropriate for situations with a non-headtracked user or multiple users, for head-tracked users see Projection Screen .

The only parameters it needs is the zero parallax plane distance, which defines the distance to a plane where left and right eye images perfectly overlap (i.e. they have zero parallax). This is the plane that the eye will associate with the projection screen or monitor, and the distance should be accurate, as far as possible, to create the most realistic result. It is defined in model units, just like the eye separation.

Projection Screen The osg::ProjectionStereoCameraDecorator is used to calculate the viewing and projection matrices needed for head-tracked stereo setups like some power-walls or, mostly, CAVEs.

The main difference to standard stereo stems from the fact that the user can now move in front of the screen. Thus the projection matrix needs to be created so that the center of projection is at the user's eye position and the frustum encloses the projection screen. As OpenGL viewports can only be rectangular, this only works right for rectangular projection screens conceptually. This rectangular projection screen is defined by its four corners, which have to be given to the osg::ProjectionStereoCameraDecorator. The order for the four corners is lower-left, lower-right, upper-right and finally upper-left.

The other information needed to calculate the matrices is the position of the user's head. This is usually delivered by a mechanical/magnetic/optical tracking system, which has its own coordinate system. This is coordinate system to be used for the user's head position as well as the corners of the projection screen.

The is an additional coordinate system that needs to be taken into account, that's the relation between the projection system and the world. As most projection system setups are smaller than the scenes they depict there is a way to move the whole system in the world, in addition to the user moving inside the projection system. This is conceptually equivalent to the standard user navigation in a scene. To hide the specifics of the osg::ProjectionStereoCameraDecorator from an application that should also be sued for standard screens, those two coordinate systems are defined by the coordinate systems of two osg::Node instances.

One of them is the osg::Camera's beacon Node. This is used to move the projection system in the scene, just as if it was a simple user.

The other is the osg::ProjectionStereoCameraDecorator's user Node, which should define the world position of the user's head. As the tracker generates relative coordinates, this node should be a descendant of the beacon node.

Thus the sub-graph for a head-tracked projected stereo should best look like this:

beacon->user (need a real picture here)

This allows simple switching between standard and head-tracked mode, in which the eye position in standard mode is the origin of the coordinate system in head-tracked mode.

11.4 Background

A background defines how the window is cleared before anything is rendered. There are a couple of different backgrounds.There can be only one background per viewport.

11.4.1 Solid Background

osg::SolidBackground is the simplest variant, it just fills the background with a single color.

11.4.2 Gradient Background

osg::GradientBackground fills the background with a color gradient. To specify the gradient a color has to be associated with a vertical position in the viewport (0 being at the bottom, 1 being at the top). An arbitrary number of gradient steps can be used, if only one is given it is used for the whole screen, if none is given black is used. Areas outside the secified gradient borders are filled with black, too.

11.4.3 Image Background

osg::ImageBackground draws an osg::Image to clear the background. The image can be scaled to fill the viewport, or it can be kept in the lower left corner of the viewport. The area not filled by the image can be cleared to a simple color. The image is really used as an image and not as a texture, thus clearing, especially with scaling, is not blindingly fast.

11.4.4 Passive Background

osg::PassiveBackground does nothing within the clear call, thus it also has no Fields at all. It is mainly used to stack viewports on top of each other, i.e. it makes it possible to have two viewports on top of each

other to generate a single combined image.

11.4.5 Sky Background

osg::SkyBackground is a sky/ground and skybox background, inspired by the VRML Background node. See http://www.vrml.org/technicalinfo/specifications/vrml97/part1/nodesRef.html#Backgr for a description on the parameter restrictions.

In general it has a set of increasing sky angles and a set of sky colors (one more than angles, for the apex). It has a similar set of parameters for the ground. The sky box is defined by 6 (optional) textures, one for each side of the cube.

11.5 Foreground

A foreground can be used to render something on top of the scene-graph image, or manipulate the image if necessary. There can be an arbitrary number of active foregrounds, which are evaluated in the order they are given in the osg::Viewport.

11.5.1 Image Foreground

ImageForeground renders images on top of the scene-graph image. The typical use is adding a logo to the image. The can be RGB or RGBA images, correct alpha blending is performed.

Images have to be loaded as osg::Image instances, their position has to be defined as a 2D position in the [0,1]x[0,1] range.

11.5.2 Grab Foreground

The osg::GrabForeground can be used to grab the rendered viewport into an osg::Image. The osg::Image has to be given to the foreground, if it is not set nothing is grabbed.

The size of the osg::Image is used to define the size of the grabbed area. If the image is set, but its size is 1 pixel in one dimension it is resized to the size of the viewport (The 1 is needed as it's impossible to create 0x0 pixel images).

11.5.3 File Grab Foreground

The osg::FileGrabForeground is used to grab rendered viewports into a single file or into a sequence of files.

It can be activated/deactivated using a flag, per default it is active. There's no need to set the image on the osg::FileGrabForeground, it is created automatically on first use.

To actually write the image the target file name needs to be set. To write a sequence of frames it also keeps a frame counter, which can be automatically incremented after each grabbed image. The name is used for a printf-style command, thus "%d" in the name is replaced by the frame number.

Hint!

Use "%04d" to create file names with leading zeros.

11.5.4 Statistics Foregrounds

The descendents of osg::StatisticsForeground can be used to print or draw Statistics elements on the rendered image.

The osg::StatisticsCollector that is used to collect the elements needs to be set in the foreground, as well as the list of osg::StatElemDesc IDs that should be displayed.

11.5.5 Simple Statistics Foreground

NOTE: The osg::SimpleStatisticsForeground is still considered experimental and can and probably will change!

osg::SimpleStatisticsForeground displays the statistics info as simple text lines. They are displayed using a compiled-in font that can use an arbitrary color and that can be arbitrarily resized, with the size per line given in pixel.

The format of every element is given by a format string for every element that is directly passed to osg::StatElem::putToString(), so go there to see the possible options.

If no elementIDs are given all elements in the osg::StatCollector are display, using the default format.

11.5.6 Graphic Statistics Foreground

NOTE: The osg::GraphicStatisticsForeground is still considered experimental and can and probably will change!

osg::GraphicStatisticsForeground displays the statistics info as one of a set of graphical elements. The possible elements are:

- an analog rotary display
- a single-bar display
- a block chart
- a line chart
- a simple text line

Every display can be put at an arbitrary position on screen, defined by a position in X-Window style, i.e. negative positive are taken to be relative to the right edge of the screen. The size of every element can be given either in pixel or relative to the viewport size, similar to the viewport itself, i.e. sizes <=1 are taken to be relative. Every display can have up to three independent colors. The min and max displayable values can either be set directly, or they can be adapted dynamically. To control that adaption there are two flags (Overflow Resize and Underflow Resize), which can be set for every display.

Other flags allow the display the min and max values as text, the display of the reciprocal value instead of the actual value, the display of dots at the positions of data points and the smoothing of the display. The smoothing is done using a running average filter of selectable length. If filtering is used the displays show two indicators, one for the current, one for the filtered value.

Some general attributes for all displays can be set: the line width used, the background color used and if the displays should have a drawn background and/or drawn border at all.

11.6 Navigators

The Navigators are utility classes to manage the transformation of a camera. There are different navigation models, which are handled by specific navigator classes. The general Navigator features a default mouse and keyboard button binding and a simple to use interface and can switch between these specific navigators.

11.6.1 General Navigator

The osg::Navigator is a helper class that wraps an instance of all the different low-level navigators and provides functions to map mouse/key event interaction to the lower-level navigator interfaces.

As a consequence the osg::Navigator is the most useful of the navigators and should be used in applications.

The default active navigator is the osg::TrackballNavigator. For this navigator the osg::Navigator will map a left mouse click with motion to a rotation, a left mouse click without motion will set the center of rotation to the hit point. A middle mouse click or move will translate in screen x/y, while keeping the hit point under the mouse. A right mouse click with vertical motion as well as using a mouse wheel (if configured to generate button 4 & 5 events) will move the viewer in the z direction.

The osg::FlyNavigator uses fly forward on left mouse button, backwards on right mouse button and stay still on middle mouse button. In all modes mouse motion is mapped to rotation.

11.6.2 Trackball Navigator

The osg::TrackballNavigator mimics a model enclosed in a glass sphere and manipulated by dragging the sphere. The primary application area of the osg::TrackballNavigator is examining a given object by looking at it from all directions, it's an outside-in exploration.

Rotation is done by dragging a point on the sphere, translation by focussing the sphere on a different position in space or by dragging the whole sphere. The size of the sphere can be changed, by default it is 80% of the screen size. It is also possible to click and drag outside the sphere. A useful special case is clicking and dragging at the extreme left and right edges of the screen, which can be used to rotate the model around the screen z-axis.

The distance between the center of rotation and the viewer can also be changed by dragging, resulting in a changing zoom.

11.6.3 Fly Navigator

The osg::FlyNavigator mimics a simple flying model. The user can move forwards and backwards along the viewing direction and he can turn left/right and up/down. The primary application area for the osg::FlyNavigator is models with rather large extends like buildings, cities and landscape, that should be explored from the inside rather than from the outside.

The primary parametrisation is a from/at/up point/vector triple, which can be used to initialize and read from the navigator.

11.6.4 Walk Navigator

The osg::WalkNavigator is an extension of the osg::FlyNavigator. It uses the same interaction model, but in addition constrains the viewer to keep a specific distance from a specified ground. It can also prevent him from walking through walls and objects as well as preventing him from passing through too narrow openings.

The main application area are architecture walkthroughs, where the user walks on a ground through one or more buildings.

11.7 SimpleSceneManager

The SimpleSceneManager (SSM) is a utility class to simplify the creation of simple applications. It manages a single window with a single viewport and a minimal scene-graph with a beacon for the camera and a headlight. It keeps a Trackball to interactively manipulate the camera.

It does not open a window itself, that is left to the user to keep the SSM useful for arbitrary window systems. The window has to be passed to the SSM by using setWindow(). That's one half of the necessary initialization. It can't handle input itself, the application has to pass it user input events. It's a lot simpler than it sounds, take a look at the tutorials to see how it works.

The other half of the necessary initialization is telling SSM what to draw by calling setRoot(). That's it. It might be useful to call showAll() to position the camera at a reasonable position, but that's not mandatory.

The SSM can be used in conjunction with any window system, it has been integrated into an easy-to-use QT widget called OSGQGLWidget. See testManagedWindowQT_qt.cpp for an example on how to use it.

As a little bonus, the SSM can display the "Powered by OpenSG" logo. Just call useOpenSGLogo() and you're done. ;)

11.8 OpenGL Objects & Extension Handling

11.8.1 OpenGL Objects

OpenGL objects are an important way to manage data and speed up repetitive use. OpenGL objects in OpenSG include everything that can be stored inside OpenGL, most prominently display lists and texture objects.

Handling OpenGL objects in a multi-window and possibly multi-pipe environment becomes an interesting problem. As the different windows may show different parts of a scene or different scenes alltogether the actually used and defined set of OpenGL objects should include only what's necessary to reduce the consumed ressources.

To do that OpenGL objects are managed by the OpenSG Windows. Before they are used they have to be registered with the osg::Window class. This is a static operation on the Window class, as it affects all exisiting Windows. Multiple objects can be registered in one call, and they will receive consecutive object ids. The ids are assigned by the object manager. It can not be queried from OpenGL, as the thread which creates the objects usually doesn't have a valid OpenGL context. As a consequence you should not use OpenGL-assigned ids, as they might interfere with OpenSGs handling of ids.

Part of the registration is to provide an update osg::Functor, which is called whenever the object needs to be updated. This functor gets passed the id and status of the object and has to execute the correct function. There are a number of stati that the functor has to handle.

The first time it is called the status be osg::Window::GLObjectE::initialize. The functor has to create the necessary OpenGL ressources and initialize the OpenGL object. For a texture object this is the definition of the image via glTexImage().

When the object changes there are two cases to distinguish. In the simple case the object has changed sig-

nificantly, needing a osg::Window::GLObjectE::reinitialize. For textures this would be changing the filter or changing the image size. Both of these actions necessitate a recreation of the actual texture object. If only the data of the image changes this can be handledmore efficiently via glTexSubImage() calls, which is an example for a osg::Window::GLObjectE::refresh. The osg::Window is responsible for keeping track of the current of the objects, and thus it has to be notified whenever the state of the OpenSG object underlying an OpenGL has changed, necessitating either a refresh or a reinitialize. This can be done by calling the static osg::Window::refreshGLObject or osg::Window::reinitializeGLObject methods. The object will be flagged as changed in all Windows and at the next validate time it will be refreshed/recreated.

Before an object can be used it has to be validated. This has to be done when the OpenGL context is valid and should usually be done just before the object is used. If the object is still valid, nothing happens. The osg::Window::validateObject method is inline and thus the overhead of calling it before every use is minimal.

When an object is not needed any more is needs to be destroyed. The destruction can be started via osg::Window::destroyGLObject. It will actually be executed the next time a Window has finished rendering (i.e. its osg::Window::frameExit() function is called). The object's functor will be called for the osg::Window::GLObjectE::destroy state, and it should free context-specific resources. After this has happened for all Windows it will be called one final time with osg::Window::GLObjectE::finaldestroy. Here context-independent resources can be freed.

11.8.2 OpenGL Extensions

The situation with OpenGL extensions is similar to the one with OpenGL objects: as the thread that initializes things probably has no OpenGl context, it cannot call the necessary OpenGL functions directly. Further complicating matters is the fact that in systems with multiple graohics cards they may not all be of the same type, and thus might support different extensions.

To handle these situations the extensions themselves and the extension functions need to be registered and accessed using the osg::Window. The registration (osg::Window::registerExtension, osg::Window::registerFunction) just needs the names and returns a handle that has to be be used to access the extensions/functions. This registration can be done from any thread.

When using the extension/function it is necessary to check if it supported on the currently active OpenGL context. To speed this up the Window caches the test results and provides the osg::Window::hasExtension method to check it. To access the functions osg::Window::getFunction method can be used. It is not advisable to store the received extension functions, as there is no guarantee that the pointer will be the same for different contexts.

Chapter 12

Window System Libraries

OpenSG doesn't have its own windowing system, it depends on the client-specific window systems. In many cases the rendering window is going to be integrated into another GUI anyway, and for the full-screen cases there's always GLUT. It is also possible to use the osg::PassiveWindow, which leaves the responsibility for the OpenGL context activation/deactivation and buffer swaps with the application. This allows easy integration into systems that have an OpenGL rendering context anyway. In general these systems do not support asynchronous rendering, i.e. having a separate thread that does OpenGL rendering, separate from the user interface thread. For that it is necessary to use the specific Window types for Win32, X or QT.

There is a specific class derived from osg::Window for every window system, which needs to be initialized with the window system specific parameters, and which wraps the specifics for context activation, deactivation and buffer swaps for the specific window system. For some systems (e.g. QT) there is also a Widget that can be integrated more directly.

12.1 GLUT Window System Library

GLUT is the OpenGL Utility Toolkit developed by Mark Kilgard for his columns in the X journal, but it has sort of established itself as the toolkit of choice for a large number of programs that only need OpenGL. It is avilable on Unix and Windows and contains some wrappers to simplify writing software that works on both platforms. See http://www.opengl.org/developers/documentation/glut/index.html for general information and documentation about GLUT.

The GLUT window is the simplest of the OpenSG windows, as GLUT takes care of almost everything. The only parameter is the integer Window ID that GLUT uses, but most GLUT applications only use the default window anyway and thus don't have to setup anything except for calling init() after creating the Window.

activate() calls glutSetWindow, eactivate() is a nop, as GLUT can't be witched off, and swap() directly calls glutSwapBuffers(). The osg::GLUTWindow is very simple, but very useful nonetheless. Most of the test and tutorial programs are written using GLUT.

12.2 QT Window System Library

QT is a portable GUI toolkit written by Trolltech (www.trolltech.com), which is used for a large number of Linux applications, especially the KDE project. Current versions do much more than windowing, but

OpenSG is only concerned about that.

The osg::QTWindow itself is only the OpenSG half of using OpenSG in QT, it has to be accompanied by an OSGQGLWidget, which is the QT half. Usually, in standard QT style, an application would derive its own widget from the OSGQGLWidget and add its data to the widget. The QTWindow needs to know which OSGQGLWidget it is associated with.

The main purpose of the special QGLWidget is to prevent OT from taking over the OpenGL context and let the osg::QTWindow handle that. The standard QT methods for OpenGL handling (makeCurrent, swapBuffers, initializeGL, paintGL, resizeGL) need to be redirected to the QTWindow, see testWindowQT_-qt.cpp for an example.

As a convenience class there is a osg::OSGQGLManagedWidget, which is derived from osg::QGLWidget and implements all the necessary functions. Furthermore it has an osg::SimpleSceneManager built in to provide some basic interaction features.

12.3 X Window System Library

The X Window System (www.x.org) is the base for the graphical user interfaces of nearly all Unix systems.

The OpenSG X Window needs the X Display and the X Window it should render into. It creates the OpenGL context itself during init(), but the X Window needs to have been created with a Visual that is OpenGL-capable. activate(), deactivate() and swap() do just that. ;)

12.4 Win32 Window System Library

Win32 are the native Microsoft Windows windows.

It needs the HWND before init() can be called. It gets the HDC itself and creates the HGLRC.

Note: we don't really know much about the intricacies of Windows programming. Thus the way we do things might be suboptimal, or maybe even illegal. Comments welcome.

Chapter 13

File Input/Output

As OpenSG user/developer you can always instantiate all your nodes, cores or whatever objects you need one by one in your application code.

In addition OpenSG also provides - as most scene-graph libraries - a set of loaders, which create a scene tree or image from a given file. The system creates singleton Handlers for both types (SceneFileHandler and ImageFileHandler), which handle all abstract requests. The concrete loaders are coded in mime/file type handlers (e.g. OSGTIFImageFileType.h/OSGTIFImageFileType.cpp) and are automatically registered.

The system architecture is designed to handle built-in types and to fetch loaders for a specific type on request. In the current version (1.0) only built-in types are provided since the meta interfaces may not be final yet. However the 1.0 version comes with a sufficient set of built-in loaders and you can always extend the library to handle you own file format.

13.1 Usage

You can always use a specific loader directly, but in most cases you would ask the Handler (e.g. SceneFileHandler) to load a file independent of the file type. There is always just one handler (it is a singleton object) you can access the object using the static the() method (e.g. osg::SceneFileType::the()). The Handler knows all the valid suffixes for every mime/file type and can pick the correct loader automatically.

If you would like to get a loader for a specific type or suffix you can just ask the handler to find it (e.g. SceneFileHandler::the().getFileType(”wrl”)).

13.1.1 Scene

The Scene Handler provides two interfaces to load a scene: read (returns a single root node or NullFC on failure) or readTopNodes (does not create an extra root but returns all top nodes).

Just use the Handler to find and use the correct type for the given suffix: osg::NodePtr rootPtr = osg::SceneFileTypeSceneFileHandler::the().read(”test.wrl”) for example would pick the VRML loader (suffix is ”wrl”), parse the file and return the result.

13.1.2 Image

The image loader works more or less the same way as the scene loader. Let the singleton handler pick the mime type and just check the return value: osg::ImageFileHandler::the().read(”test.tif”) for example would

pick the tif loader (suffix "tif"), start loading the file, and return the new Image or Null.

13.2 Cluster

Clustering

The clustering part of OpenSG is used to transfer FieldContainers over a network and to render an Image on an arbitrary number of computers. The replication of FieldContaienrs is implemented as an extension of the multithread aspect model of OpenSG.

13.2.1 RemoteAspect

There are distinct field values for each thread. Changes between these fields are propagated when the changes stored in the changelist are applied to another aspect. The RemoteAspect class is used to transfer changes over a network connection to a number of remote computers. The RemoteAspect class has two methods for change distribution. Changes are sent with sendSync and received with receiveSync. It is important, that changes are transferred only once. The current changelist must be cleared before the next call of sendSync. In some situations it is useful to have a callback when a FieldContainer of a given type is created, changed ore destroyed. The methods registerCreated, registerChanged and registerDestroyed are used to register a functor. The ClusterServer uses this feature to be informed of new or changed Cluster-Windows

13.2.2 ClusterWindow

As there are many approaches to do rendering in a cluster (e.g. Sort-first, sort-middle, sort-last), Open-SG provides a base framework to implement special algorithms. Each algorithm is implemented in a class, derived from ClusterWindow. The idea of this window is, that a window describes the context, in which rendering is done. There is a GLUT-Window to redner with GLUT, X-Winow to render in an X11 environment and a ClusterWindow to render in a cluster. When ClusterWindow::init is called, the a connection is established to all servers stored in the servers field. This connection is used to transfer changes to all servers. As all fieldcontainers are synced with the RemoteAspect to all cluster node, there is an instance of the ClusterWindow on all cluster-nodes. The ClusterWindow has a couple of virtual methods that are called for the client and for the servers. In this context a server is a renderer and the client is the application.

On the client side the following methods are called by the ClusterWindow, when ClusterWindow::render is called.

- ClientInit: Is called when ClusterWindow::render is called for the first time
- clientPreSync: Is called before the changes in the current changelist are transferred to the servers
- clientRender: Is called after chanelist sync.
- ClientSwap: Called after clientRender and serverRender was called

On the server side the following methods are called when ClusterServer::render is called. The Cluster-Server is described in the following section.

- ServerInit: Called after an initial sync.

- ServerRender: Called after the changelist sync.
- ServerSwap: Called after clientRender and serverRender

13.2.3 ClusterServer

A ClusterServer is responsible for server side rendering. Each server has its unique name. If a server request with this name is received, a connection to the client is established. After this all changes to the scenegraph from the client are received through the RemoteAspect. When a new ClusterWindow is found in the data stream, server side rendering is initialized. As a cluster window has no GL context, you have to initialize a GLUT, Qt, X or WIN32 window with a valid GL context. This window is passed to ClusterWindow::serverRender and ClusterWindow::serverSwap. The second parameter of the ClusterServer constructor is a name. With this name the client finds the servers. The third parameter is the type of connection you want to use. e.g. StreamSocket or Multicast. The last parameter is an address string. If address is an empty string, then a default address is used. With the method server->start, the server starts waiting for incoming connections. In your rendering callback (for GLUT glutIdelFunc or glutDisplayFunc) you have to call server->render with a valid render action.

13.2.4 MultiDisplayWindow

The MultiDisplayWindow is derived from ClusterWindow. It is used to render one virtual window on a number of cluster servers. The whole window is split in equal sized regions. Each region is assigned to one server. With the fields hServers and vServers it is possible to set the number of rows and columns to which the whole window is split. For example if you want to drive a wall with 9 projectors you have to assign the name of 9 rendering servers to the servers field and set hServers and vServers to 3. The MultiDisplayWindow can have an arbitrary number of viewports. If a viewport is invisible on a server, this viewport is ignored for this server. With this it is possible to use the MultiDisplayWindow for more complicated setups then a tiled wall.

If you want to drive a cave, you could set the servers to six cave rendering servers. Then you have to set hServers to 6 and vServers to 1. For each cave side, assign one viewport with the following size settings:

```
vp0-$>$setSize(     0,0, 1.0/6 ,1)
vp1-$>$setSize( 1.0/6,0, 2.0/6 ,1)
vp2-$>$setSize( 2.0/6,0, 3.0/6 ,1)
vp3-$>$setSize( 3.0/6,0, 4.0/6 ,1)
vp4-$>$setSize( 4.0/6,0, 5.0/6 ,1)
vp5-$>$setSize( 5.0/6,0, 6.0/6 ,1)
```

If you have a resolution of 1024x1024 on each cave wall, you have to set the multiDisplayWindow->setSize(1024∗6,1024). The MultiDisplayWindow assigns each viewport to one rendering server. After this, you have to setup the camera parameters for each cave wall. This is done with camera decorators e.g. the ShearedStereoCameraDecorator.

Example client and server

There are two test programs that can be used to setup a test cluster environment. testClusterServer hast to be started on each rendering server.

```
testClusterServer -m MyServer1
```

This starts a server with the name MyServer1 that uses Multicast for communication. A simple scene viewer is started with:

```
testClusterServer -m MyServer1 MyServer2 -fscene.wrl
```

The client tries to connect to the servers MyServer1 and MyServer2. testClusterClient opens a local window for navigation.

13.3 Rendering Backend

The Rendering Backend manages the Draw Tree, which is used to render the visible geometry. It is about to change significantly in Version 1.3 and in general is not visible for a user anyway, thus it is not documented here.

Chapter 14

Statistics

Statistics contains general facilities for recording and collecting statistical data like counts and times. It is mainly used for collecting rendering statistics like the amount of time need for a frame or the number of visible polygons, but it can be used for any kind of statistics.

The statistics data is collected in the form of single osg::StatElem structures which are bundled in a osg::StatCollector. Every StatElem element can have one of a number of different types, the predefined ones are Int (osg::StatIntElem), Real32 (osg::StatRealElem), Time (osg::StatTimeElem) and String (osg::StatStringElem). Every StatElement also has a description in the form of a StatElemDesc that describes the actual use of the element. The StatCollector can hold an arbitrary subset of all known elements. The StatElemDesc obj holding the element type and ID must exist as long as any collector holds a corresponding elem object. The StatElemDesc constructor creates a unique ID which can be used to create/access an correspondent Elem (which holds the value) in any collector.

The contents of the StatCollector can be accessed either for every StatElem separately or can be output as a whole in the form of a string. For displaying the contents of a StatCollector on screen the osg::SimpleStatisticsForground and osg::GraphicStatisticsForeground can be used. The various StatElem objects provide methods to access or change the value (e.g get/set increase/decrease) In addition, every StatElem holds a on-flag to activate/deactivate the statistic element.

The main idea of the Statistics structure lies in high flexibility. Thus the set of variables that are statistically relevant is not fixed, new ones can be added by the application if necessary.

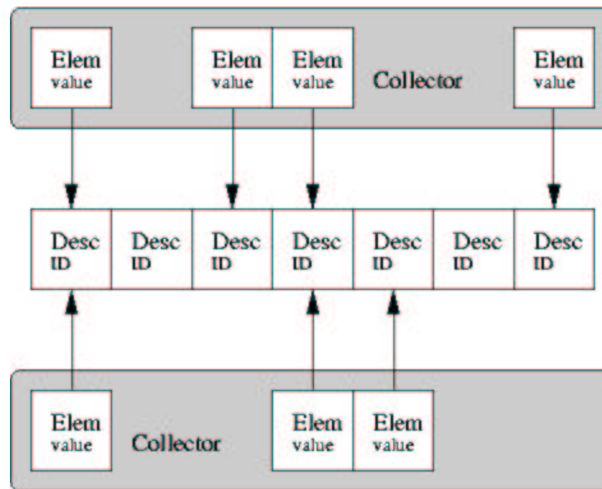

Figure 14.1: Stat Elem/Desc/Collector relation

14.1 Frequently Asked Questions

Questions:

1. How do I make the test programs?
2. Why do my 1.0 programs crash on 1.1 and later?
3. How do I set the name of an object?

Answers:

1. How do I make the test programs?

   Configure will create a directory in the Builds hierarchy for your system. In that directory you will find different directories for building the libraries (BaseLib, SystemLib etc.) and the test programs (BaseTest, SystemTest etc.). Go into the ...Test directory and say make to create all tests, make list gives you a list of the possible tests.

2. Why do my 1.0 programs crash on 1.1 and later?

   One of the major changes that we made was a revamp of the reference counting system. As a consequence the UnlinkSubTree and friends methods are not needed any more, subRefCP() is fine and will destroy the tree if it's not used anywhere else. But as a consequence you will have to make sure to addRefCP() objects that you want to be kept around, especially if you move them in and out of the tree. A simple subChild() will decrement the refcount and delete the object if it's not refed anywhere else. A rule of thumb would be to addRefCP all the objects that you keep a pointer to in your own code and be careful when moving objects around in the tree. The easy way is to leave the burden on the system and just add the node to the new parent. As we only have single parents it will automatically be removed from the original parent and moved to the new one, taking care of the refcounts.

3. How do I set the name of an object?

   Names in OpenSG are done via osg::Attachment classes. As names are a quite common problem, there are convenience functions osg::getName(void) and osg::setName(const Char8 *) in OSGSimpleAttachments.h. See the attachment tutorial for more info on using names and attachments in general.