

RAVE: Developer Guide

Ian J. Grimstead

May 31, 2006

1 Design Background

The infrastructure of Web and Grid Services is constantly evolving, so RAVE must be flexible enough to survive such changes with minimum effort. It should also be noted that Web and Grid Services are complex to debug, as they are running inside a third party container (rather than from within a debugger).

With this in mind, RAVE is designed in several layers; this enables us to test lower layers within a debugger, and wrap the trusted code inside different presentation layers (such as Web/Grid Services). Our layers consist of:

1. Generic Scenegraph API
2. Stand-Alone Scenegraph
3. Shared Scenegraph Render Engine (client/server model)
4. Generic Services Wrapper (Data Engine, Render Engine)
5. Multi-Threaded Serviced Wrapper
6. Web Services Wrapper

Each of the layers has similar abstractions, to enable rapid change of technology (be it rendering support or network transport).

1.1 Generic Scenegraph API

This is used to define a generic method of accessing the scenegraph, hiding the underlying implementation (be it Java3D or otherwise). This enabling import of graphical data and subsequent display, with user interactions also supported. (graphical objects can support user interaction such as move, rotate, etc.). This common API is used by the stand-alone scenegraph and the shared scenegraph.

1.2 Stand-Alone Scenegraph

Our bare API is implemented as Java3D or Nil; Java3D permits rendering, whilst Nil can just store data (but forms a useful test and framework for alternative implementations of the API).

1.3 Shared Scenegraph

This uses the same API as the stand-alone scenegraph, but reflects changes made to any scene graph to all other subscribers to this scene graph. A server variant of the API is initiated, this stores the central copy of the scene graph. When new clients are created (using the client variant of the API), they are directed to the server node which distributes the scene to the client (forming a bootstrap). Any updates made to the scenegraph (by the clients or the server) are reflected to all clients via the server. This forms a shared scenegraph between clients.

The client/server APIs are identical to the initial stand-alone API, apart from the constructor of the root node (as this must be informed of the location of the server node for each client to be constructed). This enables rapid testing, as we can run the server and multiple clients on a single machine, which can hence be run through a debugger.

The network communication between clients and server is mapped to an abstracted API, enabling network technology to be easily switched. At present a simple TCP/IP socket implementation is available, but alternatives could be used (such as Narada Brokering).

1.4 Generic Services Wrapper

This wraps the client/server scenegraph behind a generic discovery and interaction API. Our API supports multiple instances of the scene graph, so multiple sessions can be hosted by a single service (permitted sharing of resources). The API has various “housekeeping” calls to discover how many instances are available, number of users, current machine load, available memory, etc. The API also supports new clients subscribing to a service.

With the generic services API, we now make a distinction between two types of engine:

1. Data Engine

2. Render Engine

The Data Engine stores data, and reflects to multiple users. This maps onto the server node with the shared scenegraph.

The Render Engine is a client node for the shared scenegraph, but in turn supports multiple clients. Each render client of the Render Engine has their own camera in the scene, but the actual rendering of the scene for all of the render client cameras takes place at the Render Engine. The Render Engine then must compress the rendered images and transmit them to its render clients. Any interactions by render clients is processed by the Render Engine and forwarded to the Data Engine.

1.5 Multi-Threaded Services Wrapper

To assist with testing, we have an implementation of the Data/Render Services that works on a single machine, but uses multi-threading to facsimilate multiple machines. Each service and client runs in its own thread, which in turn may generate more threads (depending on the service).

Testing multiple services is complex, and made more difficult if a debugger cannot easily be used to investigate problems. Hence the multi-threaded implementation; this enables most issues to be verified before deploying as a Web/Grid Service.

1.6 Web Services Wrapper

This exposes the Data/Render Services as a Web Service. We make use of various utilities to reduce the maintenance overhead (such as `Java2WSDL`, `WSDL2Java`), Apache Axis to interpret SOAP¹ messages and Jakarta Tomcat as our web services container.

We now discuss each major module in detail, and how it maps onto implementation code.

2 Generic Scenegraph API

We break down the API for the scenegraph, starting with the basic nodes and then examining the component parts. The following documentation is shared with the JavaDocs, and it is recommended that you use the JavaDocs to examine the documented interfaces and classes.

¹SOAP is an XML format message used to support remote function calls as Web Services; Apache Axis interprets the message and calls the underlying method/function.

2.1 Generic SceneGraph support

The API for this is defined in `uk.ac.cardiff.cs.rave.graphics.scenegraph` and contains interfaces for the generic high-level API, exceptions thrown by scenegraphs and support for interaction inquiries.

2.1.1 High-Level API

The initial design used the Java3D API directly, but this has since moved to use an indirect API (to permit us to change from Java3D to another rendering API as required, such as JOGL). The initial usage of Java3D is reflected in the API we present, in that it could be considered a cut-down variant of Java3D.

IBranchGroup A branch group, a container for child nodes.

ICamera Camera for a user; can be on- or off-screen rendering. On-screen rendering is fastest, and used for a local client. Off-screen rendering is used to serve images to remote users.

IDirectionalLight A light source that has a direction, but not a position.

IPointLight A light that has a position (equivalent of a simple light bulb hanging in space)

IShape3D Represents a piece of geometry, be it implemented as tristrips, fans, triangle arrays or whatever.

ISpotLight A light that has a position and a direction, with inner and outer cones to define the light falloff.

ITransformGroup A group that also has a position in space. This enables new children to be added at an arbitrary position in space.

IVirtualUniverse The special node that contains the entire scene graph, and cannot be used as a child node. The virtual universe marshals all data, maintaining internal structures for the scenegraph (such as allocating and registering identifiers).

The interfaces extend common interfaces:

INode Basic access shared by all scene graph nodes (apart from **IVirtualUniverse**)

IGroup Group access, shared by **IBranchGroup** and **ITransformGroup**

ILight Basic light info for **IPointLight**, **ISpotLight** and **IDirectionalLight**

Note that all nodes can be added and removed at will from the scene graph; in Java3D, only the **BranchGroup** node has that facility.

All of the interfaces in turn inherit from multiple field access interfaces (see the `fieldaccess` package in Section 2.1.5). Overall, note that each object has a unique ID (this is how the object is referenced), and forms a hierarchy below the Virtual Universe, which forms the root node of the scenegraph.

2.1.2 Data Import Support

For any class to import external data, it should extend the **AImport** class. This contains methods to send an initialisation or update message for any scenegraph node, using the field access interfaces. Introspection is used to discover which interface are implemented, which are in turn used to examine the state of the scenegraph node, transmitting over a given stream as required. The stream is usually a network connection, but could be used to send to any stream, such as a disc file.

The **AImport#importGeometry** method is implemented and uses the **ImportDelegator** class to delegate importing to any applicable RAVE importers; these include Diffusion Tensor Imaging, ETOPO, FRO (used with the GECEM project), PLY and VTK formats.

2.1.3 Interaction Inquiries

For any item to discover available interactions in the scene graph, the **IVirtualUniverse** contains methods that use callbacks to enumerate the available interactions. Each method is given an instance of an **IRegisterInteractionCallback** object, which supports a single method call: **addInteraction**. This call lets the callee make a note of an available interaction, and is used by implementations of **IVirtualUniverse** and the interaction GUI **AInteractionGUI**.

2.1.4 Exceptions Thrown

`OffScreenNotSupportedException` Thrown when a camera is requested to render off-screen, but it cannot support off-screen rendering (occurs when a generic API is not fully supported by an underlying implementation)

`UnknownAccessTypeException` Thrown with the client/server shared scene graph, when an unknown type of interface is attempted to be transmitted or updated

`UnknownNodeClassException` Thrown with the client/server shared scene graph, when an unknown node type is requested to be sent over the network

2.1.5 Outline of Usage

We now present a short outline of how to use the scene graph API to create a scene, and view it. To contain a scene graph, we need a root node; this is the `IVirtualUniverse` node. This should be created viz:

```
IVirtualUniverse virtualUniverse = new VirtualUniverseImplementation();
```

An example of `VirtualUniverseImplementation` would be `J3DVirtualUniverse`. You then need an `IBranchGroup` object to contain any scene graph nodes you wish to create:

```
IBranchGroup bg = (IBranchGroup)virtualUniverse.getImporter().factory(IBranchGroup.class);  
// ...or...  
IBranchGroup bg = new BranchGroupImplementation(virtualUniverse);
```

The first approach is longer, but hides the underlying implementation; hence the code can rapidly switch to use an alternative implementation by only changing the `VirtualUniverse` constructor.

We now need to create something to see; let's try a cube:

```
IShape3D testCube = (IShape3D)virtualUniverse.getImporter().factory(IShape3D.class);  
testCube.setNumVertices(24);  
testCube.setFormat(IShape3D.IAVF_FORMAT_COORDINATES | IShape3D.IAVF_FORMAT_COLOUR_3);
```

Note—we have not caught any exceptions; the creation of geometry can throw many exceptions (such as `GeometryAssignedException`, `VertexCountsException`, etc.).

We have the shape initialised, ready for the vertex data; let's define the vertices (note—rather than type any of this in, the code is available in `DisplayCube`).

```
Tuple3f[] colourArray  
    = testCube.allocateTuple3fArray(IShape3D.IS3D_AT3F_HANDLE_VERTEX_COLOUR, numVerts);  
Tuple3f[] positionArray  
    = testCube.allocateTuple3fArray(IShape3D.IS3D_AT3F_HANDLE_VERTEX_POSITION, numVerts);
```

```
// Quad #0  
Color3f orange = new Color3f(Color.orange);  
colourArray[0] = orange;  
colourArray[1] = orange;  
colourArray[2] = orange;  
colourArray[3] = orange;  
positionArray[0] = new Point3f(+2, +2, +2);  
positionArray[1] = new Point3f(-2, +2, +2);  
positionArray[2] = new Point3f(-2, -2, +2);  
positionArray[3] = new Point3f(+2, -2, +2);
```

```
// Quad #1  
Color3f magenta = new Color3f(Color.magenta);  
colourArray[4] = magenta;  
colourArray[5] = magenta;  
colourArray[6] = magenta;
```

```

colourArray[7] = magenta;
positionArray[4] = new Point3f(-2, +2, -2);
positionArray[5] = new Point3f(+2, +2, -2);
positionArray[7] = new Point3f(-2, -2, -2);
positionArray[6] = new Point3f(+2, -2, -2);

// Quad #2
Color3f green = new Color3f(Color.green);
colourArray[8] = green;
colourArray[9] = green;
colourArray[10] = green;
colourArray[11] = green;
positionArray[8] = new Point3f(+2, -2, +2);
positionArray[9] = new Point3f(-2, -2, +2);
positionArray[10] = new Point3f(-2, -2, -2);
positionArray[11] = new Point3f(+2, -2, -2);

// Quad #3
Color3f pink = new Color3f(Color.pink);
colourArray[12] = pink;
colourArray[13] = pink;
colourArray[14] = pink;
colourArray[15] = pink;
positionArray[12] = new Point3f(-2, +2, +2);
positionArray[13] = new Point3f(+2, +2, +2);
positionArray[14] = new Point3f(+2, +2, -2);
positionArray[15] = new Point3f(-2, +2, -2);

// Quad #4
Color3f cyan = new Color3f(Color.cyan);
colourArray[16] = cyan;
colourArray[17] = cyan;
colourArray[18] = cyan;
colourArray[19] = cyan;
positionArray[16] = new Point3f(-2, +2, +2);
positionArray[17] = new Point3f(-2, +2, -2);
positionArray[19] = new Point3f(-2, -2, +2);
positionArray[18] = new Point3f(-2, -2, -2);

// Quad #5
Color3f yellow = new Color3f(Color.yellow);
colourArray[20] = yellow;
colourArray[21] = yellow;
colourArray[22] = yellow;
colourArray[23] = yellow;
positionArray[20] = new Point3f(+2, +2, -2);
positionArray[21] = new Point3f(+2, +2, +2);
positionArray[22] = new Point3f(+2, -2, +2);
positionArray[23] = new Point3f(+2, -2, -2);

```

We now pass the vertices to the cube, and tell the `IShape3D` object how to interpret the vertices:

```

testCube.setTuple3fArrayParam(IShape3D.IS3D_AT3F_HANDLE_VERTEX_COLOUR, colourArray);
testCube.setTuple3fArrayParam(IShape3D.IS3D_AT3F_HANDLE_VERTEX_POSITION, positionArray);

testCube.setType(IShape3D.S3D_SUBTYPE_QUADS);

```

The call to `setType` sets how the vertices we've just provided will be interpreted; this must be called last of all, as the vertices are needed to construct the geometry.

Now we have constructed our cube, we need to link it into the scene graph:

```
bg.addChildNode(testCube);
```

We also need to add a light to the scene; let's create a point light:

```
IPointLight plight = virtualUniverse.getImporter().factory(IPointLight.class);
plight.setPosition(new Point3f(300.0f, 300.0f, 300.0f));
bg.addChildNode(plight);
```

Let's add the new scene to the main scene graph:

```
virtualUniverse.addBranchGraph(bg);
```

Now let's finally render out scene graph; this requires a camera:

```
ICamera camera = (ICamera)virtualUniverse.getImporter().factory(ICamera.class);
camera.setOnScreen(400, 400);
camera.assignAvatar("Test Camera", null);
```

```
Point3f pos = new Point3f();
camera.getPosition(pos);
pos.z += 15.0f;
camera.setPosition(pos);
```

```
IBranchGroup cameraBg = (IBranchGroup) virtualUniverse.getImporter().factory(IBranchGroup.class);
cameraBg.addChildNode(camera);
virtualUniverse.addBranchGraph(cameraBg);
```

We now need to make the scene "live" so it will become visible:

```
virtualUniverse.setLive(true);
```

Finally, we need to display the camera's output on the screen; we will use a utility class `ActiveInteractionGUI` to do this (interaction GUIs are discussed in Section 2.5.3):

```
new ActiveInteractionGUI("ActiveInteraction Example",
                        camera.getCanvasContainer(), camera,
                        true, false);
```

This will produce a window on-screen, containing the yellow end-face of the cube.

2.2 Generic SceneGraph Component Support

The API for this is defined in `uk.ac.cardiff.cs.rave.graphics.scenegraph.fieldaccess` and contains component interfaces for the generic high-level API, assistance classes (for copying and comparison) and exception/errors that can be thrown.

2.2.1 Component Part API

Each of the high-level interfaces uses the common exposed interfaces defined in `uk.ac.cardiff.cs.rave.graphics.scenegraph.fieldaccess`. We are at present in the stages of moving away from any interfaces that are specific to a particular node type (such as `IAccessVerticesFormat` is only used with `IShape3D`). These are basic interfaces that are components of all scene graph nodes:

`IAccessChildren` Supports adding and obtaining of child nodes

`IAccessColour` Supports set/get for single colour associated with node; will eventually be mapped to use `IAccessTuple3f`

`IAccessDirection` Supports set/get for single direction vector associated with node; will eventually be mapped to use `IAccessTuple3f`

`IAccessFloat` Supports set/get of single float value parameters (accessed on a handle basis)

IAccessInt Supports set/get of single integer value parameters (accessed on a handle basis)

IAccessIntArray Supports set/get of arrays of integer value parameters (accessed on a handle basis)

IAccessInteraction Supports discovery and activation of node interactions related to this node without being selected

IAccessInteractionWhenSelected Supports discovery and activation of object interactions related to this node when it is selected

IAccessInteractionWithObject Supports discovery and activation of object interactions related to this node when another node is selected

IAccessNodeGenerics Generic operations supported by all nodes; these include reporting its parent Virtual Universe, if the node is attached to the virtual universe, its ID, its position and orientation in world space, how much memory it is using and outputting its contents to the given print stream.

IAccessOrientation Supports set/get for single 3×3 orientation matrix associated with node

IAccessParentId Supports set/get for the ID of the parent of this node

IAccessPosition Supports set/get for the position vector of this node; will eventually be mapped to use **IAccessTuple3f**

IAccessString Supports set/get of single string value parameters (accessed on a handle basis)

IAccessSubType Supports set/get of the sub type of an **IShape3D** node (such as tristrip, triangle fan, triangle array, etc.); will eventually be mapped to use

IAccessTuple2f Supports set/get of pairs of float value parameters (accessed on a handle basis)

IAccessTuple2fArray Supports set/get of arrays of pairs of float value parameters (accessed on a handle basis)

IAccessTuple3f Supports set/get of tuples of float value parameters (accessed on a handle basis)

IAccessTuple3fArray Supports set/get of arrays of tuples of float value parameters (accessed on a handle basis); used to access vertex position, normal, colour, etc. (with the appropriate handle)

IAccessTwoSidedRendering Supports set/get of two-sided rendering of **IShape3D** nodes; will eventually be mapped to **IAccessInt** (or perhaps a **IAccessBoolean** flag)

IAccessVerticesFormat Supports set/get of vertex format of **IShape3D** nodes (such as co-ordinates only, or co-ordinates with normals, or co-ordinates with colours and normals, etc.); will eventually be mapped to **IAccessInt**

IAccessVerticesNumber Supports set/get of number of vertices present in an **IShape3D** node; will eventually be mapped to **IAccessInt**

2.2.2 Access Assistance Classes for Component API

These classes support two common methods: `copy(destination object, source object)` and `equals(object 1, object 2)`. The `copy` method is used to copy component parts of objects, whilst the `equals` method returns `true` if the component parts of 2 objects match.

AssistAccessColour Supports the **IAccessColour** interface

AssistAccessDirection Supports the **IAccessDirection** interface

AssistAccessIntArray Supports the **IAccessIntArray** interface

AssistAccessNodeGenerics Supports the **IAccessNodeGenerics** interface

AssistAccessOrientation Supports the **IAccessOrientation** interface

AssistAccessParentId Supports the **IAccessParentId** interface

`AssistAccessPosition` Supports the `IAccessPosition` interface

`AssistAccessString` Supports the `IAccessString` interface

`AssistAccessSubType` Supports the `IAccessSubType` interface

`AssistAccessTuple3fArray` Supports the `IAccessTuple3fArray` interface

`AssistAccessTwoSidedRendering` Supports the `IAccessTwoSidedRendering` interface

`AssistAccessVerticesFormat` Supports the `IAccessVerticesFormat` interface

`AssistAccessVerticesNumber` Supports the `IAccessVerticesNumber` interface

2.2.3 Error and Exception Classes

`GeometryAssignedException` Thrown when geometry is already assigned to an `IShape3D` node, but a call has been made to adjust the number of vertices/vertex format/etc.

`InteractionIdError` Thrown when an unknown interaction ID is used with a method from `IAccessInteraction`, `IAccessInteractionWhenSelected` or `IAccessInteractionWithObject` interface.

`VertexCountsException` Thrown if a vertex count is not set before the geometry type is set for an `IShape3D` node; the number of counts (or count array) is needed by some formats (e.g. triangle strips)

`VertexFormatException` Thrown by `IShape3D` nodes when an illegal format is requested (e.g. both RGB and RGBA colour format, or an unknown format flag)

2.2.4 Example Usage of Component Interfaces

A scenegraph node is defined by implementing multiple component interfaces, such as `IPointLight`; this implements:

`IAccessPosition` For the light's position

`ILight` Which in turn implements:

`IAccessColour` For the light's colour

`INode` Which in turn implements:

`IAccessNodeGenerics` For the light's standard node support

`IAccessParentId` For the light's parent

In summary, an `IPointLight` implements:

`IAccessPosition` For the light's position

`IAccessColour` For the light's colour

`IAccessNodeGenerics` For the light's standard node support

`IAccessParentId` For the light's parent

However, compare this to `IDirectionalLight`; this implements:

`IAccessDirection` For the light's direction

`ILight` As with `IPointLight`

The two lights implement the same interfaces, apart from one stores a direction, the other a position. The interfaces implemented by a light can be detected by introspection in Java; this enables (e.g.) the network send/receive to automatically determine what information a class implements (by way of which fieldaccess interfaces are implemented). The implemented classes are then used to retrieve/set information, without any knowledge of what the underlying class actually is.

This enables us to add new classes that use existing fieldaccess interfaces without updating any code to read/write the class over the network.

2.2.5 Future Work

Eventually, the aim is to move away from any “specific” interfaces such as `IAccessColour`, and move towards “generic” interfaces such as `IAccessTuple3f`. This reduces the API, and hence simplifies changes; especially the introduction of new fields, as these will be implemented as new IDs that can be passed to a pre-existing interface.

2.3 Java3D Implementation of Generic SceneGraph Support

The API for this is defined in `uk.ac.cardiff.cs.rave.graphics.scenegraph.j3d` and contains a Java3D implementation of the generic scenegraph support.

2.3.1 AJ3DNode

This directly extends the Java3D `BranchGroup` class, so all of our J3D nodes are `BranchGroup` objects. This enables us to add or remove any RAVE scene graph nodes at any time (removing the restriction of Java3D, where only `BranchGroup` objects may do this).

This base class implements the node and parent ID support, attachment to the parent virtual universe and base memory usage.

2.3.2 AJ3DGroup

This extends `AJ3DNode` to support child access.

2.3.3 J3DBranchGroup

This simply extends `AJ3DGroup`; as the root class (`AJ3DNode`) extends Java3D’s `BranchGroup`, there is very little for this class to do! It mainly forwards RAVE calls onto their equivalent in Java3D.

The `compile` method in Java3D appears to have very little impact (if any) on the render speed; this may be caused by setting lots of access privileges (so we still retain sufficient access to the geometry), or through the setting of the JVM environment variable:

```
j3d.optimizeForSpace=false
```

This requests that Java3D should use as much memory as it wants in order to speed up rendering—such as display list usage. This may already introduce peak performance.

2.3.4 J3DCamera

This extends the `AJ3DGroup` class, storing the users’s avatar as a child of the camera object. The camera is actually Java3D `ViewPlatform` and `View` objects.

For ease of tiled rendering support, we use compatibility mode to give complete control over the viewing frustum. When tiles mode is entered, we use a `GridBag` to hold the various images in the correct form (one image is rendered locally, being a `Canvas3D`; the remainder are `JLabel` objects).

2.3.5 J3DDirectionalLight

This extends the `AJ3DNode` class, containing a Java3D `DirectionalLight` object. Method calls are mapped through to the underlying Java3D object, which is a child of the wrapped `BranchGroup` object (see `AJ3DNode`).

2.3.6 J3DPointLight

This extends the `AJ3DNode` class, containing a Java3D `PointLight` object. Method calls are mapped through to the underlying Java3D object, which is a child of the wrapped `BranchGroup` object (see `AJ3DNode`).

2.3.7 J3DShape3D

This extends the `AJ3DNode` class, containing a Java3D `Shape3D` object. Method calls are mapped through to the underlying Java3D object, which is a child of the wrapped `BranchGroup` object (see `AJ3DNode`).

To use, the full data must be defined (vertex positions, vertex indices, etc.) and the data format (namely—what data is actually supplied), before setting the type of the geometry (such as `S3D_SUBTYPE_TRIANGLES_STRIP`).

The generic data API enables us to reuse a simple API, rather than continuously extend the interfaces supported. Hence we could create a new datatype (say “Normal Pins”) and reuse an existing interface (such as `ITuple3fArray`).

2.3.8 J3DSpotLight

This extends the `AJ3DNode` class, containing a Java3D `SpotLight` object. Method calls are mapped through to the underlying Java3D object, which is a child of the wrapped `BranchGroup` object (see `AJ3DNode`).

2.3.9 J3DTransformGroup

This extends the `AJ3DGroup` class, containing a Java3D `TransformGroup` object. Method calls are mapped through to the underlying Java3D object, which is a child of the wrapped `BranchGroup` object (see `AJ3DGroup`).

2.3.10 J3DVirtualUniverse

This implements the `IVirtualUniverse` interface, extending the Java3D `VirtualUniverse` class. We use two `BranchGroup` objects at the top of the scene graph; one for all user objects, one for (hidden) system objects—namely, background and ambient colour.

2.3.11 J3DImport

This provides the Java3D implementation of data import, which extends `AImport`. It provides factory methods to create new instances of scene graph nodes. This is used to hide the underlying implementation; if you have an `AImport` instance, you can pass scene graph interfaces (such as `IBranchGroup`) to the factory and receive the correct object, without knowing if you had a Java3D based underlying implementation or other.

The `J3DImport#importGeometry` method first calls the parent implementation (`AImport#importGeometry`) to import the geometry. If this fails (i.e. returns `null`), then it uses Java3D’s importers. Additional formats made available are:

j3d Java3D scene graph stream (native to Java3D)

wrl VRML format, uses Xj3D’s VRML support

lws LightWave Scene format, provided by Sun with Java3D

obj WaveFront object format, provided by Sun with Java3D

stl Stereo Lithography format, provided by Sun with Java3D

2.4 Nil Implementation of Generic SceneGraph Support

The API for this is defined in `uk.ac.cardiff.cs.rave.graphics.scenegraph.nil` and contains a Nil implementation of the generic scenegraph support. It hence has an identical API to the Java3D implementation (see Section 2.2.5).

2.4.1 ANilNode

This base class implements the node and parent ID support, attachment to the parent virtual universe and base memory usage.

2.4.2 ANilGroup

This extends `ANilNode` to support child access. Children are simply stored as a vector of node IDs.

2.4.3 NilBranchGroup

This simply extends `ANilGroup`; the only specifics are the `compile` method, which has no side-effect as the `Nil` implementation can't render.

2.4.4 NilCamera

This extends the `ANilGroup` class, storing the users's avatar as a child of the camera object. As we can't render, most of the methods return a "Not Implemented" error when called. The interaction methods, however, are implemented (so objects can be moved, etc.).

2.4.5 NilDirectionalLight

This extends the `ANilNode` class, storing the state for a directional light.

2.4.6 NilPointLight

This extends the `ANilNode` class, storing the state for a point light.

2.4.7 NilShape3D

This extends the `ANilNode` class, storing the geometry of a `Shape3D` object.

2.4.8 NilSpotLight

This extends the `ANilNode` class, storing the state for a spotlight.

2.4.9 NilTransformGroup

This extends the `ANilGroup` class, storing the state for a transform group object.

2.4.10 NilVirtualUniverse

This implements the `IVirtualUniverse` interface, storing the state of the virtual universe object.

2.4.11 NilImport

This provides the `Nil` implementation of data import, which extends `AImport`. It provides factory methods to create new instances of scene graph nodes, returning the `Nil` implementations.

The factories hide the underlying implementation; if you have an `AImport` instance, you can pass scene graph interfaces (such as `IBranchGroup`) to the factory and receive the correct object, without knowing if you had a `Nil` based underlying implementation or other.

The `Nil` implementation does not provide any additional formats; the `NilImport#importGeometry` method just calls the parent implementation (`AImport#importGeometry`) to import geometry. If this fails, then the `Nil` implementation also gives up and returns `null`.

2.5 Generic SceneGraph Utilities

The API for this is defined in `uk.ac.cardiff.cs.rave.graphics.scenegraph.utils` and contains generic, non-implementation specific utilities (i.e. the utilities work with any implementation). The contents are grouped into generic utilities (such as bounding boxes), data importing, support for interaction GUIs, statistics handling and benchmarking.

2.5.1 Generic Utilities

We now review the available utility classes.

BoundingBox implements an axis-aligned bounding box, with methods to construct a bounding box to contain a given **INode**, **IVirtualUniverse**, point in space or another **BoundingBox**.

Also has support for focusing a given **ICamera** such that the camera shows the entire virtual universe from a given axis-aligned position.

CreateCube a simple class that creates a multi-coloured cube.

DisplayCube

Source code as used in Section 2.1.5 ; creates a simple scene: 1 coloured cube, 1 point light, 1 camera. This is then displayed on-screen.

GenerateShape

Creates general shapes; this is deprecated, as it isn't fully implemented. To create cubes, use the **CreateCube** class.

2.5.2 Data Importers

Various implementation-independant importers have been created for use with RAVE; this are presented below.

ImportDelegator This is used by **AImport**; this class has the known generic importers registered in it, and will attempt to match file extensions against known types. **AImport#importGeometry(URL inputUrl)** is overridden by an implementation (e.g. Java3D's **J3DImport**). The implementation-specific version first attempts to import by calling the overridden method (which in turn invokes **ImportDelegator**). If this fails, then the implementation-specific importers are tried instead.

For the implementation-independant importers, we have:

.curveint Runs **ImportDTI**

.fro Runs **ImportFRO**

.hdr Runs **ImportETOPO**

.ply Runs **ImportPLY**

.vtk Runs **ImportVTK**

ImportDTI This imports Diffusion Tensor Imaging datasets

ImportFRO Imports datasets used by the GECM project (electromagnetic simulation output). The file format consists of lists of triangles, with material IDs per triangles.

ImportETOPO This imports ETOPO2 datasets, as provided by the National Geophysical Data Centre (part of the National Oceanic and Atmospheric Administration) at <http://www.ngdc.noaa.gov/mgg/global/relief/ETOPO2>

ImportPLY Imports PLY (Polygon File Format) files (creates tristrips as required). Also known as the Stanford Triangle Format.

ImportVTK Imports both binary and ASCII output from VTK.

2.5.3 Interaction GUIs

AInteractionGUI This provides common support, used by both **ActiveInteractionGUI** and **ThinInteractionGUI**. The common core helps maintain lists of available interactions, repeated camera movement triggers, etc.

ActiveInteractionGUI This presents an Active RAVE client, where the displayed scene is rendered locally (rather than from a remote Render Service). This GUI can be used both as an Active Client and as a stand-alone viewer for a RAVE scenegraph.

The GUI presents an interface similar to a VR/Inventor environment, where if the mouse is dragged and then held steady, the interaction repeats until the mouse button is released. This enables simple flying around the environment without having to repeatedly start and stop dragging the mouse to trigger movement.

The scenegraph is interrogated via `IVirtualUniverse#callbackInteractionsWhenSelectedForNode(INode node, IRegisterInteractionCallback intCallback)` to populate the interaction menus, whilst `ICamera#selectObject(xpercent, float ypercent)` is used to select an object from the on-screen mouse button click.

`IAccessInteraction#doInteraction(ICamera camera, int intNum, float xpercent, float ypercent)`, `IAccessInteractionWhenSelected#doInteractionWhenSelected(int intNum, float xpercent, float ypercent)`, and `IAccessInteractionWithObject#doInteractionWithObject(int intNum, int objectId, float xpercent, float ypercent)` are used to perform the actual interactions. The related methods `isInteractionWithObjectContinuous(int intNum)` etc. are used to see if the interaction is to be continuous—i.e. repeated on a timed interval whilst the mouse button is still depressed. Creation of a new light source should not be repeated, but moving the camera should repeat until the mouse button is released.

ThinInteractionGUI Presents a Thin RAVE Client, and as such requires a Render Service to supply the frame buffer to be interacted with. Uses function calls to `IRenderServiceInstance` to discover available interactions, which in turn may be a Grid or Web service client.

Hence the same thin client GUI can be used with a stand-alone Thread implementation of Render Services, or a Web Services implementation, or a Grid Services implementation, etc.

2.5.4 Interaction Assistance

ObjectMovement Contains utility methods to rotate and move objects, either in local space or relative to other objects. Used to carry out the user interactions from `ICamera` and `IShape3D` implementations.

2.5.5 Statistics Handling

UniverseStats counts the triangles, point lights, directional lights, spotlights and cameras present in a given `IVirtualUniverse`, using generic access methods (so it will work with Java3D, Nil, etc. implementations).

2.5.6 Benchmark Support

RendererBenchmark benchmarks the time take to render a given `IShape3D` object or a predefined untextured sphere. Can render on- or off-screen, and is used to generate the benchmark results of polygons/sec for different machines.

3 Shared Scenegraph

3.1 Common Network Support for SceneGraph

This package contains common code that is reused (through class inheritance) with both server and client implementations. As `set` methods require reflection over the network, these differ between client and server; however, `get` methods are common, and these are the shared portions of code.

3.1.1 High-Level API

The common code supports the generic scenegraph API as defined in Section 2.1.5. The nodes wrap an underlying implementation, with all function calls being forwarded to the actual implementation for each node (stored as a member variable in the client/server class).

`CommonBranchGroup` Common code to implement `IBranchGroup`, extends `CommonGroup`

`CommonCamera` Common code to implement `ICamera`, extends `CommonGroup`

`CommonDirectionalLight` Common code to implement `IDirectionalLight`, extends `CommonNode`

`CommonPointLight` Common code to implement `IPointLight`, extends `CommonNode`

`CommonShape3D` Common code to implement `IShape3D`, extends `CommonNode`

`CommonTransformGroup` Common code to implement `ITransformGroup`, extends `CommonGroup`

`CommonVirtualUniverse` Common code to implement `IVirtualUniverse`

Note that `ISpotLight` is not currently supported in the networked mode.

All `set` methods call method `sendUpdateMessage` in `CommonVirtualUniverse`; this is implemented differently in `ClientVirtualUniverse` and `ServerVirtualUniverse`. The client forwards the changes to the server, whilst the server reflects changes to all clients.

3.1.2 Support Classes

We now review the other common support classes in the `scenegraph.net` package.

`CommonGroup` Common group support code, used by `CommonBranchGroup` and `CommonTransformGroup`

`CommonImport` Defines common interfaces, implemented differently by `ClientImport` and `ServerImport`.

`CommonNode` Common node support, used as a base by all `Common` node classes

`CommonReceiveUpdate` Common support code for receiving updates to existing nodes from the network; used by `CommonVirtualUniverse`.

`CommonReceiveNew` Common support code for receiving a new node from the network; implemented by each node type (client and server), to receive a new instance of their class.

`MessageHandlerSupport` Message handler used to identify network packets, defining updates to nodes or creation of new nodes

`MessageOriginator` Means of identifying who has sent a message; uses a session id, as IP addresses are insufficient in case we have multiple clients on the same machine

`DisplayClientServerCube` Example usage of shared scenegraph, as used below.

3.2 Example Usage of Common Network Support for SceneGraph

The source code for this example is presented in the `DisplayClientServerCube` class. First of all, we need to allocate the variables to be used by the server; these are the underlying `IVirtualUniverse` wrapped by the server, the `AMessageHandler` to bootstrap and maintain updates.

Note that `SocketMessageHandler` implementations of the `AMessageHandler` are used, although any implementation would suffice.

```

final int milliSecondWaitBetweenBootstrapCompletedChecks = 200;

IVirtualUniverse serverUnderlyingVU = new J3DVirtualUniverse();

SocketMessageHandler serverBootstrapMH = null;
SocketMessageHandler serverUpdateMH = null;

int hashCode = 0;
int bootstrapPortNum = 10000;
int updatePortNum = 20000;
String serverName = "localhost";

// - //

// Create server virtual universe
ServerVirtualUniverse serverVU = new ServerVirtualUniverse(serverUnderlyingVU);

serverVU.setVuName("Server");

    We now create a scene on the server for the client to view:

try
{
    // Server specific code - create something for us to look at
    IBranchGroup bg = (IBranchGroup) serverVU.getImporter().factory(IBranchGroup.class);

    IShape3D testCube = (IShape3D) serverVU.getImporter().factory(IShape3D.class);
    CreateCube.setCube(testCube, false);    // No normals

    bg.addChildNode(testCube);

    IPointLight plight;
    plight = (IPointLight) serverVU.getImporter().factory(IPointLight.class);
    plight.setPosition(new Point3f(300.0f, 300.0f, 300.0f));
    bg.addChildNode(plight);

    serverVU.addBranchGraph(bg);
}
catch (GeometryAssignedException e)
{
    // We're lazy and not handling this, so just panic...
    throw new Error(e.getMessage());
}
catch (Exception e)
{
    // We're lazy and not handling this, so just panic...
    throw new Error(e.getMessage());
}

// - //

// Make Server live...
serverVU.setLive(true);

```

We now create the server sockets on the scene graph server; this enables clients to attach at will to the server.

```
int numTriesAvailable = 10;
```

```

int numTriesRemaining = numTriesAvailable;
boolean retry = false;
do
{
    retry = false;

    try
    {
        serverBootstrapMH = SocketMessageHandler.createServer(hashCode, bootstrapPortNum);
    }
    catch (IOException e)
    {
        if (e instanceof BindException)
        {
            if (e.getMessage().equalsIgnoreCase("Address already in use"))
            {
                if (numTriesRemaining > 0)
                {
                    retry = true;
                    numTriesRemaining--;
                    bootstrapPortNum++;

                    System.err.println("Bootstrap port in use - trying alternative port");
                }
            }
        }
    }

    if (!retry)
    {
        // We're lazy and not handling this, so just panic...
        throw new Error(e.getMessage());
    }
}
while ((serverBootstrapMH == null) && (retry));

numTriesRemaining = numTriesAvailable;
do
{
    retry = false;

    try
    {
        serverUpdateMH = SocketMessageHandler.createServer(hashCode, updatePortNum);
    }
    catch (IOException e)
    {
        if (e instanceof BindException)
        {
            if (e.getMessage().equalsIgnoreCase("Address already in use"))
            {
                if (numTriesRemaining > 0)
                {
                    retry = true;
                    numTriesRemaining--;
                    updatePortNum++;

                    System.err.println("Update port in use - trying alternative port");
                }
            }
        }
    }
}

```

```

        }
    }
}

if (!retry)
{
    // We're lazy and not handling this, so just panic...
    throw new Error(e.getMessage());
}
}
}
while ((serverUpdateMH == null) && (retry));

```

Now the server is created and listening to network sockets, we create the client. We'll firstly create a `MessageOriginator`, this uniquely identifies the client to the server.

```

// Let's connect the client to the server

IVirtualUniverse clientUnderlyingVU = new J3DVirtualUniverse();

MessageOriginator clientMO = null;

try
{
    clientMO = new MessageOriginator(hashCode);
}
catch (UnknownHostException e)
{
    // We're lazy and not handling this, so just panic...
    throw new Error(e.getMessage());
}

```

The client is now ready to create bootstrap and update `AMessageHandler` objects; these connect the client to the server.

```

SocketMessageHandler clientUpdateMH
    = SocketMessageHandler.createClient(hashCode, updatePortNum,
        serverName, "client update");
SocketMessageHandler clientBootstrapMH
    = SocketMessageHandler.createClient(hashCode, bootstrapPortNum,
        serverName, "client bootstrap");

ClientVirtualUniverse clientVU
    = new ClientVirtualUniverse(clientUnderlyingVU, clientUpdateMH, clientBootstrapMH);
clientVU.setVuName("Client");

// Server will now bootstrap client as soon as client is connected...
// ...will bootstrap as soon the bootstrap MH is connected
// => ensure it is linked to a clientVU!!!

serverVU.registerClient(clientMO, serverUpdateMH, serverBootstrapMH);

```

The client will now be bootstrapped from the server; note that during a full test, the server and other clients may be receiving updates whilst this bootstrap is in progress.

We'll wait until the client has bootstrapped:

```

// Wait until client has same number of nodes as server...
int numberOfTimesToWait = 200;
int numberOfTimesLeftToWait = numberOfTimesToWait;
do

```

```

{
    synchronized (clientVU)
    {
        try
        {
            clientVU.wait(milliSecondWaitBetweenBootstrapCompletedChecks);
        }
        catch (InterruptedException e)
        {
            // Don't mind if we're interrupted, we're only waiting to avoid chewing the CPU
        }

        numberOfTimesLeftToWait--;
    }
}
while ((!clientVU.isInitialised()) && (numberOfTimesLeftToWait > 0));

```

For tidyness, we'll wait until the client's bootstrap has shut down (which implies the bootstrap must be complete).

```

// Now wait for the client bootstrap Message Handlers to shut down (as they're no longer needed)
int numBootstrapShutdownWaits = 10 * (2000 / milliSecondWaitBetweenBootstrapCompletedChecks);
int numBootstrapShutdownWaitsLeft = numBootstrapShutdownWaits;
while ((clientBootstrapMH.isReady()) && (numBootstrapShutdownWaitsLeft > 0))
{
    // Wait for server to shutdown - to make sure it does!
    synchronized (clientBootstrapMH)
    {
        System.out.println("Waiting for client bootstrap MH to shutdown... "
            + numBootstrapShutdownWaitsLeft + " tries left");
    }
    try
    {
        clientBootstrapMH.wait(milliSecondWaitBetweenBootstrapCompletedChecks);
    }
    catch (InterruptedException e)
    {
        // We're lazy and not handling this, so just panic...
        throw new Error(e.getMessage());
    }

    numBootstrapShutdownWaitsLeft--;
}
}

```

The client is now bootstrapped from the server, and is maintained in synchrony with it. Hence any updates sent to the server will be reflected to this client.

We'll now add a camera to the client (which will be reflected back to the server), and display it on-screen.

```

ActiveInteractionGUI clientGUI = null;

try
{
    ICamera camera;
    camera = (ICamera) clientVU.getImporter().factory(ICamera.class);
    camera.setOnScreen(400, 400);
    camera.assignAvatar("Test Camera", null);

    Point3f pos = new Point3f();
}

```

```

camera.getPosition(pos);
pos.z += 15.0f;
camera.setPosition(pos);

IBranchGroup cameraBg
    = (IBranchGroup) clientVU.getImporter().factory(IBranchGroup.class);
cameraBg.addChildNode(camera);
clientVU.addBranchGraph(cameraBg);

clientVU.setLive(true);

clientGUI = new ActiveInteractionGUI("ActiveInteraction test",
                                    camera.getCanvasContainer(),
                                    camera, true, false);
}
catch (Exception e)
{
    // We're lazy and not handling this, so just panic...
    throw new Error(e.getMessage());
}

// Let's go live!
clientVU.setLive(true);

```

Finally, we'll add shutdown code so that when the on-screen window is closed, we'll shut down the client and servers.

```

// Shutdown code - call when the window closes...
JFrame windowFrame = clientGUI.getWindowFrame();
final AMessageHandler finalServerUpdateMH = serverUpdateMH;
final AMessageHandler finalClientUpdateMH = clientUpdateMH;
final AMessageHandler finalServerBootstrapMH = serverBootstrapMH;

WindowListener wl = new WindowAdapter()
{
    public void windowClosing(WindowEvent we)
    {
        // Close sockets, etc.
        try
        {
            finalServerUpdateMH.sendShutdown();
        }
        catch (IOException e)
        {
            // We're lazy and not handling this, so just panic...
            throw new Error(e.getMessage());
        }

        do
        {
            System.out.println("Waiting for client update to shutdown cleanly...");
            finalClientUpdateMH.waitForShutdown(1000);
        }
        while (finalClientUpdateMH.isReady());

        do
        {
            System.out.println("Waiting for server to shutdown cleanly...");
            finalServerBootstrapMH.waitForShutdown(1000);

```

```
    }
    while (finalServerBootstrapMH.isReady());

    // Forcibly shut down - otherwise Java3D will not release its threads,
    // so the VM will not close.
    System.exit(0);
    }
};
windowFrame.addWindowListener(wl);
```

3.3 Network Update Support for Generic SceneGraph

Network (stream) bootstrap and update is achieved through the generic interfaces defined in `scenegraph`. For each interface, there is an equivalent support class to transmit that interface's data over a stream.

3.3.1 Supported Interfaces

The support classes can read or write the following related interface's specific state:

`UpdateColour` Supports read/write access to the state of an `IAccessColour` interface

`UpdateDirection` Supports read/write access to the state of an `IAccessDirection` interface

`UpdateIntArray` Supports read/write access to the state of an `IAccessIntArray` interface

`UpdateNodeGenerics` Supports read/write access to the state of an `IAccessNodeGenerics` interface

`UpdateOrientation` Supports read/write access to the state of an `IAccessOrientation` interface

`UpdateParentId` Supports read/write access to the state of an `IAccessParentId` interface

`UpdatePosition` Supports read/write access to the state of an `IAccessPosition` interface

`UpdateString` Supports read/write access to the state of an `IAccessString` interface

`UpdateSubType` Supports read/write access to the state of an `IAccessSubType` interface

`UpdateTuple3fArray` Supports read/write access to the state of an `IAccessTuple3fArray` interface

`UpdateTwoSidedRendering` Supports read/write access to the state of an `IAccessTwoSidedRendering` interface

`UpdateVerticesFormat` Supports read/write access to the state of an `IAccessVerticesFormat` interface

`UpdateVerticesNumber` Supports read/write access to the state of an `IAccessVerticesNumber` interface

3.3.2 Unsupported Interfaces

The following interfaces defined in `scenegraph` are not supported with network (stream) access:

`IAccessChildren` Supported instead through defining new nodes and then setting `IAccessParentId` with `UpdateParentId`

`IAccessFloat` Not yet used by any nodes in the `scenegraph` API

`IAccessInt` Not yet used by any nodes in the `scenegraph` API

`IAccessInteraction` Interaction support is pre-defined and unchangeable; hence they do not need to be transmitted

`IAccessInteractionWhenSelected` Interaction support is pre-defined and unchangeable; hence they do not need to be transmitted

`IAccessInteractionWithObject` Interaction support is pre-defined and unchangeable; hence they do not need to be transmitted

`IAccessTuple2f` Not yet used by any nodes in the `scenegraph` API

`IAccessTuple2fArray` Not yet used by any nodes in the `scenegraph` API

`IAccessTuple3f` Not yet used by any nodes in the `scenegraph` API

3.4 Client Specific Network Support for SceneGraph

This extends the common code in the `net` package.

ClientBranchGroup Simply creates a new client variant of the `IBranchGroup`; all implementation is in `CommonBranchGroup`, this just supplies specific constructors

ClientCamera Extends `CommonCamera` and provides tiled rendering support through management of tiles and related cameras

ClientDirectionalLight Simply creates a new client variant of the `IDirectionalLight`; all implementation is in `CommonDirectionalLight`, this just supplies specific constructors

ClientImport Extends `CommonImport`, providing constructors for the client nodes, but does not import any external data.

ClientPointLight Simply creates a new client variant of the `IPointLight`; all implementation is in `CommonPointLight`, this just supplies specific constructors

ClientShape3D Simply creates a new client variant of the `IShape3D`; all implementation is in `CommonShape3D`, this just supplies specific constructors

ClientTransformGroup Simply creates a new client variant of the `ITransformGroup`; all implementation is in `CommonTransformGroup`, this just supplies specific constructors

ClientVirtualUniverse Extends `CommonVirtualUniverse`, supplying custom node ID generation (the server is requested to provide all IDs, to ensure IDs are unique). Also a factory to produce client nodes.

3.5 Server Specific Network Support for SceneGraph

ISubscription Defines an API to enable a client to register interest in a subset of the scene graph; implemented by all the server scene graph nodes. Not fully implemented as yet.

Subscription Records if clients are interested in whatever item this **Subscription** instance is associated with. Work in progress.

ServerBranchGroup Simply creates a new client variant of the **IBranchGroup**; all implementation is in **CommonBranchGroup**, this just supplies specific constructors. Also implements **ISubscription** by encapsulating an instance of **Subscription**.

ServerCamera Simply creates a new client variant of the **ICamera**; all implementation is in **CommonCamera**, this just supplies specific constructors. Also implements **ISubscription** by encapsulating an instance of **Subscription**.

ServerDirectionalLight Simply creates a new client variant of the **IDirectionalLight**; all implementation is in **CommonDirectionalLight**, this just supplies specific constructors. Also implements **ISubscription** by encapsulating an instance of **Subscription**.

ServerImport Extends **CommonImport**, providing constructors for the server nodes, and imports external data.

ServerPointLight Simply creates a new client variant of the **IPointLight**; all implementation is in **CommonPointLight**, this just supplies specific constructors. Also implements **ISubscription** by encapsulating an instance of **Subscription**.

ServerShape3D Simply creates a new client variant of the **IShape3D**; all implementation is in **CommonShape3D**, this just supplies specific constructors. Also implements **ISubscription** by encapsulating an instance of **Subscription**.

ServerTransformGroup Simply creates a new client variant of the **ITransformGroup**; all implementation is in **CommonTransformGroup**, this just supplies specific constructors. Also implements **ISubscription** by encapsulating an instance of **Subscription**.

ServerVirtualUniverse Extends **CommonVirtualUniverse**, supplying node IDs to clients on request, keeping a note of attached clients and reflecting updates to all attached clients.

4 Generic Network Services Wrapper

4.1 Generic Network Services

This package does not contain any code directly, but has child packages:

data Definition of a RAVE data service, along with support for data service client recording

messaging A generic API for message handling (i.e. packets of data sent over streams)

render Definition of a RAVE render service, along with support for render service client recording

servicefactory Common code for data and render service factories

utils Utility classes such as host information and system status/load

These packages are encapsulated or extended to produce useable implementations, as found in the packages:

sockets TCP/IP socket support (implements **messaging**)

thread Multi-threaded test implementation of RAVE

ws Web-Services implementation of RAVE

4.2 Generic Data Service Support—Code Shared by All Data Service Implementations

This code is extended to form the the Web Service and other implementations. Classes are:

ADataServiceFactory Generic interface to create a factory, that can create new data service instances on demand

DataClient Extends **HostInfo** to record the host info of a data service client

DataClients A wrapper for an underlying Vector of **DataClient** elements

DataEngine A generic data engine that can be exposed by a specific service wrapper (such as Web Services or Grid Services). Contains generic methods, such as set which TCP/IP ports are available to be used, subscribe a new client, get number of clients, etc.

IDataEngine Defines the common methods used by **IDataServiceInstance** and **DataEngine**; this enables a common API and single documentation point.

IDataServiceInstance Extends **IServiceInstance** and **IDataEngine**, to define the additional service methods supported by a RAVE data service. This is a generic interface that can be used to refer to a Web Service or other implementation of the RAVE data service

4.2.1 Generic Data Engine

As defined in **DataEngine**, this takes an optional URL to import a remote dataset and creates a new scenegraph (or an empty scenegraph, if a null URL was provided). At present, it defaults to using the Java3D implementation.

A default set of ports are then scanned for use (i.e. are they available to RAVE?); if a port (range 30000-30099) is unused, it is added to a pool of known available ports for later allocation for client communication.

The user can assign the underlying Data Service endpoint to the service, such as the URL where its containing Web Service is hosted; this is used if the service needs to expose how it should be contacted by external processes.

Clients may subscribe to the Data Engine; this allocates two new sockets on the server, one for update and one for immediate bootstrap. Once the server is ready, the **subscribe()** method returns a hash code uniquely identifying the subscribing client. This code is used by the client to identify itself when interacting with the server.

4.3 Generic Message Handling Support—Code Used by All RAVE Server and Client Implementations

This contains classes for message handling and for remote access to log output (useful for remote debugging); classes are:

AMessageHandler A generic message handling API, that needs to be implemented to use a given underlying network type (such as TCP/IP sockets). User registers message (network packet) IDs with a callback (of type **IReceiveData**). When a packet is received with a certain ID, its related callback is triggered. This enables the user to rapidly register new messages by having custom receipt methods for each message type.

ALogFactoryClient Enables client to connect to a remote machine and receive a copy of its local Message Log. Needs to be implemented to be used, such as **SocketLogFactoryClient**.

ALogFactoryServer Enables server to supply its local Message Log to a remote machine. Needs to be implemented to be used, such as **SocketLogFactoryServer**.

ExtendedByteArrayOutputStream Extends **ByteArrayOutputStream**, but permits direct access to the underlying byte array for speed (rather than undergoing conversion of any kind)

IReceiveData Implementations of this are registered as callbacks with the **AMessageHandler** class against message (network packet) IDs. Hence, when a message of a known ID is received, the appropriate (registered) callback is triggered.

UnknownPacketIdException Thrown when an **AMessageHandler** implementation receives a message (packet) id which hasn't been registered and hence hasn't got a related callback—the message handler doesn't know how to process this message.

4.4 Generic Render Service Support—Code Shared by All Render Service Implementations

This code is extended to form the the Web Service and other implementations. Classes are:

ARenderServiceFactory Generic interface to create a factory, that can create new render service instances on demand

IRenderEngine Defines the common methods used by **IRenderServiceInstance** and **RenderEngine**; this enables a common API and single documentation point.

IRenderServiceInstance Extends **IServiceInstance** and **IRenderEngine**, to define the additional service methods supported by a RAVE render service. This is a generic interface that can be used to refer to a Web Service or other implementation of the RAVE render service

RegisterAllInteractions Implements **IRegisterInteractionCallback** to convert the available interactions to/from a String, which simplifies transmission of the interactions via an API. Namely, Web Services supply support for base types (integer, float, string), so we can use a simple API that does not require custom types to transmit the available interactions to a client.

RenderClient Extends **HostInfo** to record the host info of a render service client, whilst also supplying a **Runnable** implementation so the render client can be maintained by its own thread. This enables multiple clients to be supported by a single render service, as each client has its own thread. This also boosts speed on multi-processor systems.

RenderClients A wrapper for an underlying Vector of **RenderClient** elements

RenderEngine A generic render engine that can be exposed by a specific service wrapper (such as Web Services or Grid Services). Contains generic methods, such as set which TCP/IP ports are available to be used, subscribe a new client, get number of clients, retrieve available interactions, etc. It uses **CreateSharedScenegraph** to create a shared scene graph in the **RenderEngine** that is synchronized with the **DataService**.

4.4.1 Generic Render Engine

As defined in **RenderEngine**, this takes an **IDataServiceInstance** object which it handshakes with to obtain the shared scenegraph.

Current implementation uses Java3D by default, but this is only a single use of a constructor that determines this—hence it is simple to change to an alternative implementation.

A default set of ports are used, in the range 40000-40099. One port per client is required, but if another instance of the Render Engine is created, it will allocate a new batch of port numbers (chaining from the last batch, this would allocate ports 40100-40199). This enables multiple render engines to run on the same machine without clashing with port allocations.

RenderEngine#subscribe allocates a port number for the client, creates a new camera and prepares the render stream. Once **RenderEngine#startRenderStream** is called, the client's camera on the Render Service will start to render and images are streamed over the allocated port.

4.5 Generic Service Factory Support—Code Shared by All Render and Data Service Implementations

This code is extended to form the the Web Service and other implementations. Classes are:

AServiceFactory All RAVE services extend this class; it has a simple constructor—noting its base URL, i.e. the URL that exposes this service to the outside world. It defines a basic API, containing methods such as get number of subscribers, current CPU speed (measured in number of 3×3 matrices multiplied per second), implementation version of RAVE, etc.

IServiceInstance Generic interface used by all instances of RAVE services; when used, should actually forward a call to the underlying service as created by the **AServiceFactory** implementation.

4.6 Generic Service Utilities—Code Used by All RAVE Service Implementations

This code is used to assist all RAVE services, rather than render or data services in particular. The classes are:

CreateSharedScenegraph Connects to a given `IDataServiceInstance`, handshakes and receives a copy of the shared scenegraph (and maintains the shared link with the data service). Exposes this as an `IVirtualUniverse`, with the code reused in the `RenderEngine` and `RenderClientCreationPanel` to generate an Active Client.

HostInfo Stores the generic information used by all host machines; namely, ports used to contact host (update and bootstrap port) and the host's name

HostInfos Convenience wrapper to provide a `Vector` of `HostInfo` objects

SystemStats Object that returns system performance statistics, used by RAVE services to give some status of the local environment

5 Network Web-Services Wrapper

5.1 Web-Service Implementation of RAVE

No specific code in this package; it contains sub-packages that support the client (`client`), data service (`data`), render service (`render`) and utilities (`utils`).

5.2 Web-Service Implementation of RAVE—Client Support

No specific code in this package; it contains sub-packages that support the client, namely as GUI.

5.3 RAVE Management GUI, Web Service Client with UDDI Support for Service Discovery

Provides the RAVE Management GUI, as used in the Installation/User Guide to view available RAVE services, launch new service instances and to connect to them using Active and Thin Clients.

ColumnHeaderToolTips Produces tool tips for each column header (in the GUI tables)

CommonTableModel Generic table model used as a basis for all tables used in the GUI (namely, Data and Render Services, services and service instances)

CommonTableRow Generic table rows used as a basis for all table rows used in the GUI (namely, Data and Render Services, services and service instances)

RenderClientCreationPanel On-screen panel, lists available render clients and permits user to create an instance, connecting to previously selected render or data service instance as appropriate. Currently has Java3D Active Client and generic Java 2D Thin Client

RenderServiceTableRow Extends **ServiceTableRow** to present a row in the render services table. Each render service is listed with generic service info (such as number of subscribers, memory available, etc.), but adds the million polygons per second (to reflect available speed from render server).

ServiceDebugPanel Small debug panel displayed next to each "Available [data/render] Services" table; lets user force update of table from remote hosts, can show additional debug info and show Log output from remote service host

ServiceFactoryPanel Produces a table to contain the given type of row (hence can produce render service or data service table), to show available service hosts, with the debug window next to the table (for table refresh, debug info display, etc.). Table is wrapped inside a scrollpane to cope with tables wider than the available display. Used by **ServicePanelJ2SE** to produce a render or data service host table

ServiceInstanceCreation Takes a **CommonTableRow** object and examines this to create the correct form of service instance creation (data or render service instance creation). Used by **ServicePanelJ2SE** to produce the instance creation box next to the list of available instances.

ServiceInstanceTableModel Extends **CommonTableModel** to present a table that contains a list of service instances available from a given service

ServiceInstanceTableRow Extends **CommonTableRow** to create a single table row for a generic service instance; shows fields such as instance name, number of subscribers, description of instance, etc.

ServicePanelJ2SE Produces the panes required to represent a given service type (data or render); uses a provided **CommonTableRow** object to determine which type is required, producing a debug panel with associated table of available service hosts, a table of available service instances for a selected host and a pane for creation of new service instances.

Has a **ServicePanelJ2SE#main** method that spawns a complete GUI; used in the Installation/User Guide.

ServiceTableModel Extends **CommonTableModel** to present a table of RAVE service hosts; uses UDDI to discover the services (data or render services). Uses the supplied **CommonTableModel** object to populate the table rows (hence supports data and render services—**CommonTableModel** takes care of this)

ServiceTableRow Extends **CommonTableRow** to present a row in the data services table. Each data service is listed with generic service info (such as number of subscribers, memory available, etc.).

5.4 RAVE Data Service—Web-Service Implementation

No specific code in this package; it contains sub-packages that implement the data service, namely as client (`client`), interface defining the Web Service's API (`impl`), generated Web Service skeleton and stubs (`Data`), Web Service related utilities (`utils`).

The single file in this package is `Data.wsdl`; this is produced by running `Java2WSDL` on `Data`.

5.5 RAVE Data Service—Client Exposure of Web-Service Implementation

Used to present a common interface to the user by encapsulating the Web Services client implementation. Exposes a web service instance of a data service as `IDataServiceInstance`, and the factory

`DataServiceFactory` Extends `ADataServiceFactory`, using `DataServiceLocator` to get access to the remote Web Service.

`DataServiceInstance` Implements `IDataServiceInstance` to permit access to the wrapped Web Service implementation (as spawned by `DataServiceFactory`)

5.6 RAVE Data Service—Auto-Generated Web Service Support and Implementation

Produced by WSDL2Java (via the `Data.wsdl` file, in turn produced from `Data`). All files are repeatedly overwritten by running this tool, except for `RaveDataSoapBindingImpl`.

`Data` Interface auto-generated from WSDL file; should match API in `Data`

`DataService` Interface auto-generated, defines how to locate a RAVE Data Service

`DataServiceLocator` Auto-generated, used to locate the RAVE Data Service in the Web Service

`RaveDataSoapBindingImpl` Originally auto-generated with empty method bodies; completed to forward Web Service calls to the wrapped `DataEngine`

`RaveDataSoapBindingSkeleton` Auto-generated, used to forward SOAP messages from Apache Axis to `RaveDataSoapBindingImpl`

`RaveDataSoapBindingStub` Auto-generated, used by the client to access the remote RAVE Data Service. Presents the interface `Data`, and is returned from the `DataServiceLocator`

`deploy.wsdd` Auto-generated Web Service Deployment Descriptor; use to auto-deploy the Web Service implementation to Apache Axis

`undeploy.wsdd` Auto-generated Web Service Deployment Descriptor; use to auto-undeploy the Web Service implementation from Apache Axis

5.7 RAVE Data Service—Interface Used to Define Web Service Implementation

Defines a single interface: `Data`. This is used by `Java2WSDL` to produce a WSDL file; this is in turn used by `WSDL2Java` to generate the skeleton and stubs for Web Service support.

This API is in turn implemented, which then wraps calls to an underlying `DataEngine`.

5.8 RAVE Support/Utility Code for Data Service Web-Service Implementation

At present, has utility code for deploying and undeploying RAVE Data Web Services via UDDI.

`DataServiceDeployToUDDI` Deploys a given machine's RAVE data Web Service to UDDI, using `DeployWS2UDDI`

`DataServiceUndeployToUDDI` Undeploys a given machine's RAVE data Web Service to UDDI, using `UndeployWS2UDDI`

5.9 RAVE Render Service—Web-Service Implementation

No specific code in this package; it contains sub-packages that implement the render service, namely as client (`client`), interface defining the Web Service's API (`impl`), generated Web Service skeleton and stubs (`Render`), Web Service related utilities (`utils`).

The single file in this package is `Render.wsdl`; this is produced by running `Java2WSDL` on `Render`.

5.10 RAVE Render Service—Client Exposure of Web-Service Implementation

Used to present a common interface to the user by encapsulating the Web Services client implementation. Exposes a web service instance of a render service as `IRenderServiceInstance`, and the factory

`RenderServiceFactory` Extends `ARenderServiceFactory`, using `RenderServiceLocator` to get access to the remote Web Service.

`RenderServiceInstance` Implements `IRenderServiceInstance` to permit access to the wrapped Web Service implementation (as spawned by `RenderServiceFactory`)

5.11 RAVE Render Service—Auto-Generated Web Service Support and Implementation

Produced by WSDL2Java (via the `Render.wsdl` file, in turn produced from `Render`). All files are repeatedly overwritten by running this tool, except for `RaveRenderSoapBindingImpl`.

`Render` Interface auto-generated from WSDL file; should match API in `Render`

`RenderService` Interface auto-generated, defines how to locate a RAVE Render Service

`RenderServiceLocator` Auto-generated, used to locate the RAVE Render Service in the Web Service

`RaveRenderSoapBindingImpl` Originally auto-generated with empty method bodies; completed to forward Web Service calls to the wrapped `RenderEngine`

`RaveRenderSoapBindingSkeleton` Auto-generated, used to forward SOAP messages from Apache Axis to `RaveRenderSoapBindingImpl`

`RaveRenderSoapBindingStub` Auto-generated, used by the client to access the remote RAVE Render Service. Presents the interface `Render`, and is returned from the `RenderServiceLocator`

`deploy.wsdd` Auto-generated Web Service Deployment Descriptor; use to auto-deploy the Web Service implementation to Apache Axis

`undeploy.wsdd` Auto-generated Web Service Deployment Descriptor; use to auto-undeploy the Web Service implementation from Apache Axis

5.12 RAVE Render Service—Interface Used to Define Web Service Implementation

Defines a single interface: `Render`. This is used by `Java2WSDL` to produce a WSDL file; this is in turn used by `WSDL2Java` to generate the skeleton and stubs for Web Service support.

This API is in turn implemented, which then wraps calls to an underlying `RenderEngine`.

5.13 RAVE Support/Utility Code for Render Service Web-Service Implementation

At present, has utility code for deploying and undeploying RAVE Render Web Services via UDDI.

RenderServiceDeployToUDDI Deploys a given machine's RAVE render Web Service to UDDI, using `DeployWS2UDDI`

RenderServiceUndeployToUDDI Undeploys a given machine's RAVE render Web Service to UDDI, using `UndeployWS2UDDI`

RecruitRenderServiceAssistance Used by the tiled rendering support; discovers available RAVE Render Services, checks which one will deliver the best FPS when assisting the current host, and forwards the appropriate render service instance to the camera that requested tiled rendering assistance. Also creates new render service instance if necessary.

5.14 RAVE Support/Utility Code for Web-Service Implementation

At present, has utility code for discovering, deploying and undeploying RAVE Web Services via UDDI.

DeployWS2UDDI Supports deployment of a given web service to the UDDI server. Contains a `main()` method to enable the user to deploy at will; see the User and Installation Guide for example usage.

UndeployWS2UDDI Supports undeployment of a given web service to the UDDI server. Contains a `main()` method to enable the user to deploy at will; see the User and Installation Guide for example usage.

ServicesFromUDDI Searches the UDDI server for any access points for RAVE Web Services. Used to populate the RAVE Management GUI (`ServiceTableModel`) and by the tiled rendering support (`RecruitRenderServiceAssistance`).

6 Single Machine, Multi-Thread Services Test Wrapper

6.1 Multi-Threaded (Single Machine) Specific RAVE implementation

This package does not contain any code directly, but has child packages for data service support (**data**), render service support (**render**), and utilities (**utils**).

This implementation serves as test (to show non-Web Service implementations can be built using the interfaces defined in RAVE), also is quick to debug as it can run data & render services from a single root process, making it easy to debug compared to remote processes (such as spawned from a Web Services container).

6.2 Multi-Threaded (Single Machine) Specific RAVE Data Service

This package does not contain any code directly, but has child packages for data service client support (`client`).

The hierarchy is arranged to match that of the Web Services implementation (`data` package and sub-packages), which requires various sub-packages rather than just the client we need here.

6.3 Multi-Threaded (Single Machine) RAVE Data Service Client

Implements the basic classes required to present a data service, without using remote services.

DataServiceFactory Implements **ADataServiceFactory**, supports multiple instances of **DataServiceInstance** by recording a map from instance name to instance object. Supports basic functionality, such as network timing, number of subscribers, number of instances, etc.

DataServiceInstance Implements **IDataServiceInstance**, passing method calls directly to the wrapped **DataEngine**. Unlike the Web Services implementation, the data engine is stored directly inside the implementation, so there are no network messages sent when invoking methods.

Note that the scenegraph is still maintained via network bootstrap/update messages, but the packets are sent via loopback rather than external transport.

6.4 Multi-Threaded (Single Machine) Specific RAVE Render Service

This package does not contain any code directly, but has child packages for data service client support (`client`).

The hierarchy is arranged to match that of the Web Services implementation (`render` package and sub-packages), which requires various sub-packages rather than just the client we need here.

6.5 Multi-Threaded (Single Machine) RAVE Render Service Client

Implements the basic classes required to present a render service, without using remote services.

RenderServiceFactory Implements **ARenderServiceFactory**, supports multiple instances of **RenderServiceInstance** by recording a map from instance name to instance object. Supports basic functionality, such as network timing, number of subscribers, number of instances, etc.

RenderServiceInstance Implements **IRenderServiceInstance**, passing method calls directly to the wrapped **RenderEngine**. Unlike the Web Services implementation, the render engine is stored directly inside the implementation, so there are no network messages sent when invoking methods.

Note that the scenegraph is still maintained via network bootstrap/update messages, but the packets are sent via loopback rather than external transport.

6.6 Multi-Threaded (Single Machine) RAVE Service Utilities

Support utilities, just used for merging data into and forming Java3D stream files.

MergeData A static `main` method, called with N URLs. The URLs are imported into a single scene graph, and exported again to a single Java3D stream file, effectively merging the data sources.

ReformatData A static `main` method, called with 1 URL, which is exported to a single file, effectively reformatting the data into Java3D stream format.

7 UDDI Support

7.1 UDDI Support for RAVE Service Discovery and Advertisement

This code is used to both find and advertise RAVE services, and uses the `uddi4j` package (defines classes in `org.uddi4j` namespace). Various properties files are used to reduce the amount of hard-coded data:

business.properties Defines `businessName` (published name of the business, set as “RAVE Project”), `businessDescription` (description of the business, set as “RAVE: Resource-Aware Visualization Environment, School of Computer Science, Cardiff University”), `contactPersonName` (contact for business, set as “Dr.I.J.Grimstead”), finally `contactUseType` (set to define the contact’s job, namely “Senior Research Associate”)

common.properties Defines `inquiryURL`, the UDDI service that supports inquiries, and the related `publishURL`, `username` and `password` for UDDI publication. If `username` and `password` are not defined (they aren’t as standard!), then you must supply `rave.uddi.username` and `rave.uddi.password` to the Java Virtual Machine at run-time, viz:

```
java -Drave.uddi.username=fred -Drave.uddi.password=secret
```

dataservice-ws.properties Defines `serviceName` (name registered against the RAVE data service, hence set to “RAVE Data Service”), `serviceDescription` (description of data service, set to “Data service for RAVE, used to transmit to Render service”). The Technical Model details are also defined: `tModelName`, the name of the technical model (set to “RAVE Data Web Service”), `tModelDescription` description of the technical model (set to “Data web service 0.4 beta”), `tModelWsdUrl` where the technical model WSDL can be downloaded (set to “<http://safleprofi.grid.cf.ac.uk:28080/ax>” although this is a temporary home for the WSDL—it should not be relied upon!), `tModelWsdUrlDesc` the description of the service access point we’ve published (set to “Beta release of RAVE; do not rely on for stable service”)

renderservice-ws.properties Similar settings to `dataservice-ws.properties`, but relate to the Render Service instead (hence “RAVE Render Web Service” rather than “RAVE Data Web Service” etc.)

The classes are:

AccessPoints Convenience class to store a `Vector` of `AccessPoint` classes (from `org.uddi4j` package)

Business Uses the fields from a given properties file (such as `business.properties` to publish a business in UDDI. Also has support for undeployment.

Configurator Used by `Proxy` to load properties from file

Report Used by other UDDI methods to output errors in human-legible form

Service Used to store a UDDI service and any discovered access points to the service.

TechModel Used to store a UDDI technical model; also supports deployment and undeployment of technical models

The UDDI classes are used in the Web Services package (`ws`) to deploy, undeploy and discover RAVE Web Services.

8 TCP/IP Sockets Support

8.1 Socket Implementations of Network Support

This code is used to assist all RAVE services, rather than render or data services in particular. The classes are:

SocketLogFactoryClient A socket specific implementation of **ALogFactoryClient**, where a local user can connect to a remote process and views its **Log** output.

SocketLogFactoryServer Supported in the Web Services implementation of both data and render services, interacted via the method **AServiceFactory#requestSocketMHLog**. This serves the **Log** output to remote machines (exposed locally as a **SocketLogFactoryClient**).

SocketMessageHandler Provides a socket implementation of **AMessageHandler**, used in the Data and Render Services, Active and Thin Client implementations to support passing of messages (network packets)

SocketTimingSupportClient Connects to a **SocketTimingSupportServer**, and initially sends 32-byte packets (setting no TCP delay in Java, disabling the Nagle algorithm—otherwise, these small test packets will be buffered and sent when the buffer is full, distorting the results). The small packets are used to determine network latency. It then starts sending larger packets, sending increasingly larger packets up to a maximum size (**SocketTimingSupportServer#ms_maxBufferSize**) a maximum time spend testing the network (**SocketTimingSupportServer#ms_maximumTimeToWait**—this ensures we don't spend hours sending huge packets over a modem line by mistake).

SocketTimingSupportServer Provides a server to the **SocketTimingSupportClient**; instantiated on demand by a service implementation, for a local client to bounce packets on, to produce network timings.

9 Revision Log

The CVS revision log is now presented, to reveal changes made to this Chapter.

```
$Log: Guide-Developer.tex,v $
Revision 1.16  2005/03/15 22:18:04  scmijg
Slowly getting there... adding Web Services info

Revision 1.15  2005/03/15 21:47:19  scmijg
Completed

Revision 1.14  2005/03/15 16:11:45  scmijg
Slowly getting there...

Revision 1.13  2005/03/15 15:41:12  scmijg
Sockets written up...

Revision 1.12  2005/03/14 15:57:59  scmijg
Finished.

Revision 1.11  2005/03/14 13:19:22  scmijg
Dumb client removed from generic package...

Revision 1.10  2005/03/14 12:05:37  scmijg
Root package docs...

Revision 1.9   2005/03/11 10:16:43  scmijg
Reorganisation - use subsection as highest level in javadoc, can then group
in latex manuals by section (can't use chapters to break, as we may not
be a book!)

Breaking by subsection is OK with html, as we don't show the section numbers...
```

Revision 1.8 2005/03/10 09:46:10 scmijg
Finally, now works OK with HTML and PDF, uses common subdir (javadoc) to pull
in javadoc macros...

Revision 1.7 2005/03/07 17:37:44 scmijg
More macros being added...

Revision 1.6 2005/03/07 17:07:48 scmijg
Slowly tidying up - nearly there!

Revision 1.5 2005/03/07 16:05:39 scmijg
Slowly merging JavaDoc into the main developer guide...

Revision 1.4 2005/03/07 12:19:03 scmijg
Removed refs to javadocs that are removed... namely scenegraph.java3d
More work on developer guide...

Revision 1.3 2005/03/07 10:52:51 scmijg
First sections complete, now onto the nitty gritty...

Revision 1.2 2005/03/03 13:33:30 scmijg
Working on Tomcat/Axis installation...

Revision 1.1 2005/02/28 10:01:33 scmijg
Renamed files to make life easier! Now all consistent...
Also added new "part" file for Architecture

Revision 1.1 2005/02/25 10:59:25 scmijg
Ok, now working with developer/install/user guides