Chapter 21. Networked Tour3D

NetTour3D is a *very* simple networked virtual environment (NVE) which allows sprites representing users (clients) on different machines to move about in a shared world. The world is the familiar checkboard, with scenery and obstacles (red poles).

NetTour3D is essentially a combination of two earlier examples: Tour3D from chapter 10 which introduced the Sprite3D class, and networking code from the multi-threaded chat application in chapter 19. It would be a good idea to (re-)read those chapters before starting on this one.

NetTour3D is a client/server application, with each client running a copy of the shared world, with messages passing between the clients via the server.

Figure 1 shows NetTour3D being run by two clients. Each window is a particular client's view of the shared world (a third person camera which follows the client's sprite). Each client can see their own sprite and, when in the right location, the sprites of the other visitors. A client can move his sprite around using the usual keyboard controls. All the visitors use the same robot image but their own names are floating above the robots. The world includes a castle and several obstacles.

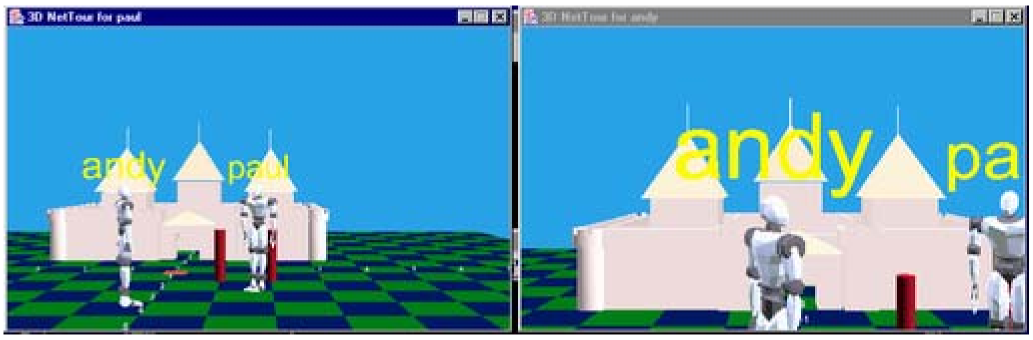


Figure 1. Two Visitors to the NetTour3D world.

Key features:

- Each user (client) is represented by two kinds of sprite: a sprite *local* to the user's machine, and a *distributed* sprite utilized on other machines. Both types have the same appearance (a robot) but are updated in different ways;
- The local and distributed sprites are subclasses of a common Sprite3D class, so are functionally very similar;
- Local sprites are updated directly, without the client first communicating with the server, so reducing latency;
- The world scenery is created by each client, not transmitted from the server, so conserving bandwidth. The scenery is specified in a 'tour' file and loaded by a PropManager object (first seen in chapter 9), so different worlds can be easily specified.

A major weakness of this example is that the users cannot interact, apart from gliding about the checkboard. However, it wouldn't be difficult to graft a multi-user chat component onto NetTour3D. This would involve integrating more of the multi-threaded chat example of chapter 19 into NetTour3D.

In this chapter, we begin by discussing the features of a fully fledged Networked Virtual Environment (NVE), and then consider the considerably simpler NetTour3D example.

1. Background on NVEs

An NVE is a computer-based artificial world of 3D spaces, visited by geographically-dispersed users who interact and collaborate with each other, and with object/entities local to the world. The world's 3D spaces and their objects may be maintained/hosted by numerous computers spread around the network

The NVE is a descendant of the MUD (Multiple-User Dungeons) – text-based role-playing adventure games that achieved enormous popularity from the mid 1970's onwards. In the 1990's, MOOs (MUDs object oriented) started to use object oriented programming techniques to implement their worlds, 2D and 3D chat environments appeared, and the first multi-user games were released.

The currently in-vogue gaming acronym is MMORPGs (Massively Multiplayer Online Role-Playing Games), typified by EverQuest, Asheron's Call, Ultima Online, and a growing list of others. Some sites that maintain lists, news, FAQs, reviews, and so on:

- MPOGD.com, the Multiplayer Online Games Directory
  http://www.mpogd.com/
- OMGN, the Online Multiplayer Gaming Network
  http://www.omgn.com/
- MMORPG.com, http://www.mmorpg.com/
- The Google directory on massive multiplayer online games
  http://directory.google.com/Top/Games/Video_Games/Roleplaying/Massive_Multiplayer_Online/

Aside from the game playing potential for NVEs, they are also the subject of much academic research. In the 1990's, DARPA's SIMNET project developed the Distributed Interactive Simulation (DIS) protocol for modeling real-world scenarios (usually military-related, but also complex, distributed applications such as Air Traffic Control systems). DIS has greatly influenced the communication protocols utilized in NVEs, and the utilization of real-time within the worlds.

A follow-on from DIS is the HLA (High Level Architecture), focussing on support for simulations composed from multiple distributed components (see https://www.dmso.mil/public/transition/hla/). The HLA offers federation rules to govern the interactions between components, and numerous management tools, called

Run-Time Infrastructure (RTI) services. These include time management (e.g. federated clocks), data distribution management (e.g. to filter user messages), and object ownership tools.

Another source of ideas comes from Collaborative Virtual Environments (CVEs) which emphasize human interaction in collaborative working frameworks, when the users are at different physical locations.

2. The Elements of an NVE

The most immediately noticeable elements of an NVE are spaces, users, objects, and views. Less evident are the notions of consistency, real-time, dead reckoning, security, and scalability. We briefly consider each of these.

NVEs are network applications and so must also deal with the network challenges described in chapter 18: latency, bandwidth, reliability, protocol, and topology.

2.1. Spaces

The 3D spaces in an NVE define the world's topology. A space may be a large common area (a landscape, playground, street) or a smaller private space for select groups (e.g. a conference room, gym, hall), or a place for individual interactions (office, kitchen). Spaces may be unchanging, or privileged users may be able to reconfigure them, delete them, or create new ones. Each space has a set of attributes, privileges, and/or security features (e.g. passwords) which govern who can use it, and in what ways.

The largest granularity of spaces are often known as *zones*, and play an important role in the underlying implementation of the NVE. In Ultima Online and EverQuest, zones are supported by different servers so a that a user who moves between zones will also move between servers. This approach lends itself to load balancing, although a very popular zone will still cause overloading. In Asheron's Call, *portal storming* is a mechanism for 'teleporting' users away from a high traffic zone to a randomly selected destination. Zones also make message filtering easier since a user need only receive information related to his/her own current zone.

2.2. Users

Users in an NVE are visually represented by *avatars*, created by the user when he/she first joins the world. At the implementation level, a user may be denoted by two kinds of avatar – a *local avatar* which is present on the player's own machine, and a *ghost avatar* which is employed on all the other client machines connected to the world. The avatars look the same on-screen, the differences lie at the communication layer. A local avatar may be controlled directly by the user without the overhead of the communication passing through a server first. However, a ghost avatar will require its state and behaviour updates to be delivered over the network, introducing the issue of latency.

NVEs frequently distinguish between different groups of users (e.g. novices, gurus, farmers), with corresponding differences in their abilities to affect spaces, objects, and other users.

Differing abilities lead to users forming groups to collaborate on common tasks. A task may be a CVE-style activity such as report writing, or a gaming-style objective such as treasure seeking. Collaboration usually requires a much richer communications protocol to support forms of interaction such as negotiation, brokering, bargaining, contracts, task division, and result combination.

2.3. Objects

Objects in a space can be classified in various ways. Some objects may never change, such as buildings, sign posts, and street fittings, while others may be mobile but still essentially passive (e.g. coins, maps). Movement in the world may require a corresponding implementation level movement of the object's representation between machines. Objects may react when a user 'triggers' them, for example a door opens when a user touches it. A dynamic object will have its own behaviour, often AI-based, allowing richer interactions with users, which may be initiated by the object.

Objects are one of the ways that users communicate, by giving objects to each other, by making copies for others, or by dividing a single object into smaller pieces.

2.4. Views

Views govern how a client sees a space, objects, or other users. Most multiplayer games are first person oriented so the player see very little of their own avatar. However, each user will be able to employ several views into the space, which can be dynamically adjusted. Views may be abstractions, as with maps showing player activities, or list of objects currently being carried by the player.

Interest management permits a player to subscribe and unsubscribe to the reception of messages concerning other users, objects, or spaces. For example, when a user changes position, all the subscribed players will be notified. IP multicasting is often utilized to implement this mechanism, although the large number of users and objects in a world may mean that the number of available multicast groups is exhausted. Zone-based notification schemes are more viable, due to the relative small number of zones in a world. Only users currently in the zone will receive updates when something changes in that zone.

2.5. Consistency

Consistency states that all users should see the same sequence of events in the same order. For instance, if user X walks through a door and then user Y shoots a gun, then X, Y, and all the other users in the vicinity should see that same sequence. The problem lies in the fact that the events may have occurred on geographically-separated machines, and that event details must be sent between the player's machines by message passing. The presentation of these events to every user in the same order implies that they can be temporally ordered. This means time stamping the events on different machines with clocks that are synchronized.

Fortunately, not all users require the same level of consistency. In the above example, only users close to X and Y really require complete consistency. Other users must receive the events eventually, but their ordering may not be so critical.

One approach to consistency, used in the MiMaze application, is called *bucket synchronization* (http://www-sop.inria.fr/rodeo/MiMaze/). First, an estimate is made of the communication latency between the users (e.g. 100ms), which becomes the *playout delay*. Then all event messages are time stamped when they occur (e.g. at time t), and stored in a 't' *bucket* until time t+100ms before being processed.

This delay gives time for messages generated at time t to travel over the network to the player. They should arrive before the 100ms delay is finished, and so be present in the t bucket when the player evaluates the world state for time t.

The underlying communication in MiMaze is UDP multicasting, with time stamps added to the messages so they can be ordered (placed in the right buckets). MiMaze also employs dead reckoning (explained below) to compensate for message which don't arrive before the global state is calculated.

2.6. Real-time

Real-time requirements mean that when an event occurs at time t for one user, other users should see that event at time t as well. As mentioned above, this assumes a globally consistent logical clock, usually implemented using local clocks on each machine which are kept synchronized.

It also requires assumptions about typical network latency and reliability. For example, bucket synchronization relies on the setting of a suitable playback delay, derived from the network latency. If the latency increases above 100ms, as it will over larger networks, the delay will need to be increased. This will further retard event processing, including events initiated by the user on his own machine. The increased delay will degrade the application's apparent response time, and will eventually become unacceptable to the user.

Another aspect of the problem is the likelihood of packet loss when utilizing UDP transmission. Moving to TCP is often ruled out since its guaranteed delivery can affect latency time too severely.

2.7. Dead Reckoning

A popular solution to the problems with real-time support is to combine UDP, time stamping and related algorithms with dead reckoning (also known as *predictive modeling*). The basic idea is that each client runs simulations of the other clients in the NVE. When it comes time to update the global state on the client's machine, any missing data from other clients will be replaced by extrapolations taken from the simulations of those clients. This means that delays caused by latency and lost data are hidden.

The client may also run a simulation of itself, and regularly compare its actual state with the one generated by the simulation. When the differences between them become too great, it can send a state update message to its peers, asking them to correct their details for the client. Consequently, state messages will probably need to be sent out less often, reducing network congestion.

Dead reckoning was first introduced in DARPA's SIMNET project, and was mostly concerned with updating the position of entities. For instance, the current position of an object (e.g. a tank) could be extrapolated by using its last known position and velocity, or by using position, velocity, and acceleration information.

Other dead reckoning algorithms take the orientation of the entity (i.e. its roll. pitch, heading) into account, and handle moving subparts. A recent DARPA initiative, the Advanced Distributed Simulation (ADS) architecture, introduced *predictive contracts*, which encourage extrapolations using non-physics based equations, and a wider range of object attributes.

The drawback with mathematical complex algorithms is the increased cost of their calculation.

A fundamental problem with dead reckoning is its assumption that an entity's state change (e.g. its movement) is predictable. This is often not true for user avatars in an NVE.

Another issue is deciding when a simulated state and actual state are sufficiently different to warrant the sending of update messages. If the difference threshold is too large, then the client's ghost avatar may undergo a very noticeable change in position (or other attribute) when the update message is processed. If the threshold is too small, then unnecessary messages will be sent out, contributing to network congestion.

The means by which a simulated state is changed to the correct value is called *convergence*. The simplest technique is to change the state in a single step, causing the avatar to jump, jitter, or snap to a new state. Other approaches rest on the idea of gradually interpolating between the simulated state and new one.

2.8. Security

NVE security takes many forms. Within the world, security determines how users behave with each other, and how they interact with spaces and objects. Active objects must be monitored since they have their own behaviour, and may be able to move between client and server machines.

Security is usually distributed. Typically, there is a connection manager which new users must deal with before they enter the world. Internal world security is handled by the servers responsible for each zone, and/or by the client-side software. The drawback with delegating security to the client-side is the possibility that it may be circumvented by hackers.

2.9. Scalability

NVE scalability is a complex problem since an increase in world size often makes implementations based upon a single server inadequate. The typical solution used by pay-for-play models, such as EverQuest and Ultima Online, is to utilize multiple servers, each managing a zone, and a connection manager to supervise user admission.

The zoning metaphor allows issues like consistency management, message volumes, and sharing to be kept under control. Zones may be duplicated across several servers to improve load balancing, dissuade some forms of hacker attack, and to act as backups if a server fails. Multiple servers can be geographically dispersed to help reduce latency, since international packets transfer times can easily extend beyond 200ms. Systems may utilize peer-to-peer communication for aspects of the game that do not need monitoring (e.g. chatting).

3. NVE Examples

There are three excellent commercial NVE applications using Java 3D:

- Magicosm, a fantasy role playing game
  http://www.magicosm.net/
  For screenshots, see http://www.magicosm.net/screenshots.php
- Pernica, also a fantasy role playing game
  http://www.starfireresearch.com/pernica/pernica.html
  For screenshots, see http://www.starfireresearch.com/pernica/graphics.html
- City of Nights BBS
  http://citynight.com/vc
  A long running chat service with a 3D interface.

Java 3D source code for NVE construction can be found at:

- eXtensible MUD (xmud)
  http://xmud.sourceforge.net/index.html
  The code handles avatar animation, terrain following, collision detection, and the creation of new objects for the world. The client/server network communication is sockets-based but uses object serialization. The server-side employs MySQL for data storage. There is a security manager for client authentication

  The results look very similar to the commercial products mentioned above (see http://xmud.sourceforge.net/screenshots.html)
- Java 3D Community MMORPG Project
  http://starfireresearch.com/services/java3d/mmorpg/mmorpg.html
  A project hosted at Starfire Research, the creators of Pernica. Unfortunately, it's development seems to be on hold, but there is some code available, written by David Yazel and Kevin Dulig.
- Tag3D
  http://www.croftsoft.com/portfolio/tag3d/
  A prototype multi-user online virtual reality using Java 3D and RMI, dating from 1999. Written by David Wallace Croft.
- Bang
  Last known address: http://www.in-orbit.com
  This site has disappeared, but use to host an open source 3D MOO game engine utilizing IP multicasting and Jini. Multiuser chat was supported, RTP real-time video, voice recognition, and object authoring and exchange.

Sites with Java source code, but not Java 3D:

- WolfMUD
  http://www.wolfmud.org/
  Supports multiplayer, networked adventure games, including a GUI-based world builder consisting of zones with objects. Unlike a large number of MUD development sites, this one is actively supported, and even has good online documentation.

- Millport
  http://millport.sourceforge.net/
  A MUD with a graphical interface.

- The Mars Simulation Project
  http://mars-sim.sourceforge.net/
  The aim is human settlement on Mars, represented by a multi-agent simulation. The emphasis is on setting various parameters to encourage the society to grow and develop. See screenshots at http://mars-sim.sourceforge.net/

- MADViWorld
  http://diuf.unifr.ch/sde/projects/madviworld/
  MADViWorld supports massively distributed virtual worlds. A world can consist of many subspaces (called rooms), filled with active objects which can be readily extended. Multiple servers are linked using RMI, but there are plans to utilize Jini Discovery and Lookup services.

- MiMaze
  http://www-sop.inria.fr/rodeo/MiMaze/
  MiMaze3D is a 3D maze game utilizing Java and VRML. It is totally distributed, using RTP/UDP/IP multicast communication between the players. MiMaze utilizes a bucket synchronization algorithm and dead reckoning for its real-time and consistency requirements.

- Two relevant Google directories
  Java MUDs:
  http://directory.google.com/Top/Games/Internet/MUDs/Servers/Java/

  Java Virtual Worlds:
  http://directory.google.com/Top/Computers/Programming/Languages/
  Java/Virtual_Worlds/

An academic text about NVEs:

Networked Virtual Environments: Design and Implementation
Sandeep Singhal and Michael Zyda
ACM Press, Addison-Wesley, 1999

4. NetTour3D Components

Figure 2 shows the main objects involved in a NetTour3D world.

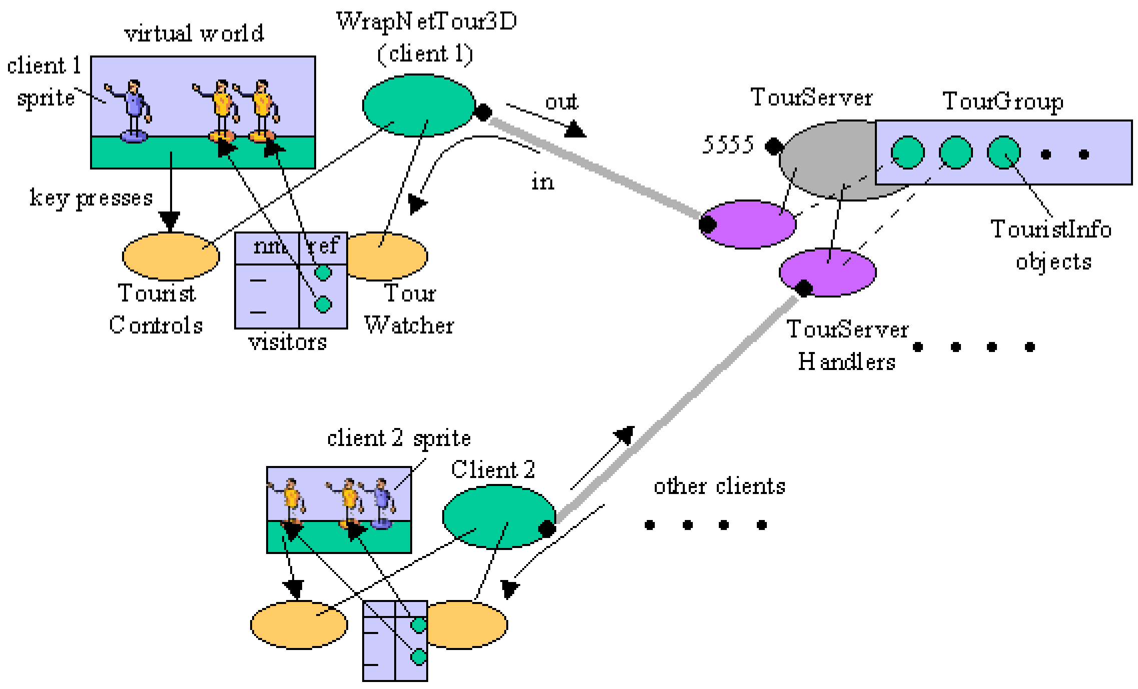


Figure 2. Objects in a NetTour3D World.

The NetTour3D application is the client-side of the system. It creates a WrapNetTour3D object to build a copy of the world and to handle communication with TourServer, the server-side.

The world consists of a checkboard, background, lighting, and scenery loaded from a 'tour' file with the help of a PropManager object. Consequently, no 3D objects or images need to be transferred over the network; all the necessary 3D models are already present on the client-side.

WrapNetTour3D starts a TouristControls object to monitor the client's key presses. They either move the client's local sprite (the ones coloured blue in Figure 1), or adjust the third person camera.

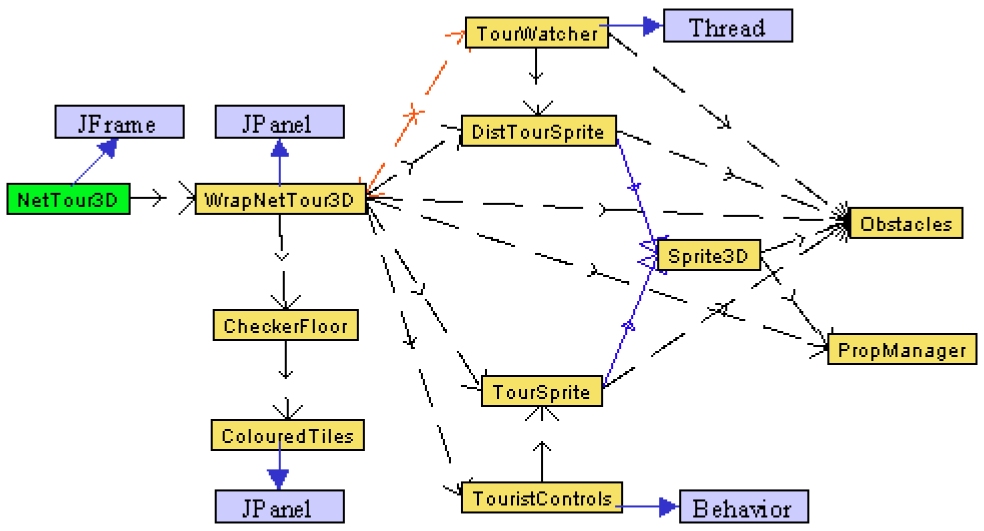
WrapNetTour3D sends message to the server-side itself, but the monitoring of messages coming from the server is delegated to a TourWatcher object. Most of these messages will be related to the creation and movement of distributed sprites (the sprites representing other clients, coloured orange in Figure 1). TourWatcher manages these sprites, and updates them in response to the server's messages.

TourServer creates a TourServerHandler thread for each client who connects to it, and stores information about the connections in a shared TourGroup object, in an ArrayList of TouristInfo objects. The main task of the server-side is to accept a message from one client and broadcast it to the others.

The networking model is a multi-threaded client/server, very like the multi-threaded chat application in chapter 19.

5. UML Diagrams for NetTour3D

Figure 3 shows UML diagrams for the classes in the NetTour3D client-side application and the server-side TourServer. Only the class names are shown.



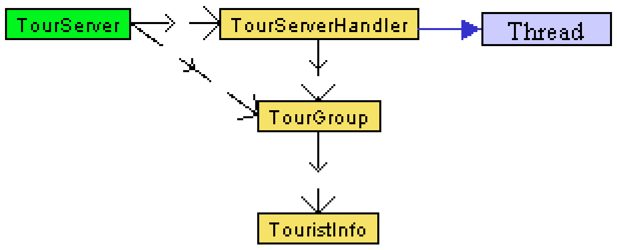


Figure 3. UML Class Diagrams for NetTour3D and TourServer.

NetTour3D is the top-level JFrame for the application, and very similar to our earlier examples.

CheckerFloor and ColouredTiles are unchanged from other applications: they create the checkboard floor.

PropManager loads the scenery, and Obstacles sets up the obstacles, specified in a 'tour' file. The same approach is used in Tour3D in chapter 10.

TouristControls in also unchanged from the Tour3D example.

TourServer's role is to spawn threaded handlers for clients.

6. The WrapNetTour3D Class

createSceneGraph() starts the main tasks of WrapNetTour3D: the 3D scene is created, contact is made with the server, and a local sprite is initialized:

```
void createSceneGraph(String userName, String tourFnm,
                              double xPosn, double zPosn)
{ sceneBG = new BranchGroup();
  bounds = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

  // allow clients to be added/removed from the world at run time
  sceneBG.setCapability(Group.ALLOW_CHILDREN_READ);
  sceneBG.setCapability(Group.ALLOW_CHILDREN_WRITE);
  sceneBG.setCapability(Group.ALLOW_CHILDREN_EXTEND);

  lightScene();         // add the lights
  addBackground();      // add the sky
  sceneBG.addChild( new CheckerFloor().getBG() );  // add the floor

  makeScenery(tourFnm);      // add scenery and obstacles

  makeContact(); // contact server (after Obstacles object created)

  addTourist(userName, xPosn, zPosn);
                        // add the user-controlled 3D sprite
  sceneBG.compile();   // fix the scene
}
```

Capability bits are set in order to allow distributed sprites to be added to the scene (and removed) during execution.

makeContact() sets up an input and output stream to the server, and passes the input stream to TourWatcher to monitor. TourWatcher creates distributed sprites when requested by the server, and so must know about the obstacles present in the world.

```
private void makeContact()
{ try {
    sock = new Socket(HOST, PORT);
    in  = new BufferedReader(
                new InputStreamReader( sock.getInputStream() ) );
    out = new PrintWriter( sock.getOutputStream(), true );

    new TourWatcher(this, in, obs).start();   // watch server msgs
  }
  catch(Exception e)
  {  System.out.println("No contact with server");
     System.exit(0);
  }
}
```

addTourist() creates a TourSprite object and connects a TouristControls object to it so that key presses can make it move and rotate.

```
private void addTourist(String userName,double xPosn,double zPosn)
{
  bob = new TourSprite(userName, "Coolrobo.3ds", obs,
                      xPosn, zPosn, out);   // local sprite
  sceneBG.addChild( bob.getBG() );

  ViewingPlatform vp = su.getViewingPlatform();
  TransformGroup viewerTG = vp.getViewPlatformTransform();
```

```
    TouristControls tcs = new TouristControls(bob, viewerTG);
    tcs.setSchedulingBounds( bounds ); // sprite's controls

    sceneBG.addChild( tcs );
  }
```

The TourSprite object is passed a reference to the output stream going to the server. The object can then notify the server of its creation, and when it moves or rotates. The server will tell the other clients, which can then affect the distributed sprite representing the user.

7. The Sprite3D Class

Sprite3D is the superclass for the local sprite class, TourSprite, and the distributed sprite class, DistTourSprite. Figure 4 shows the UML classes in this hierarchy, and all the public methods.

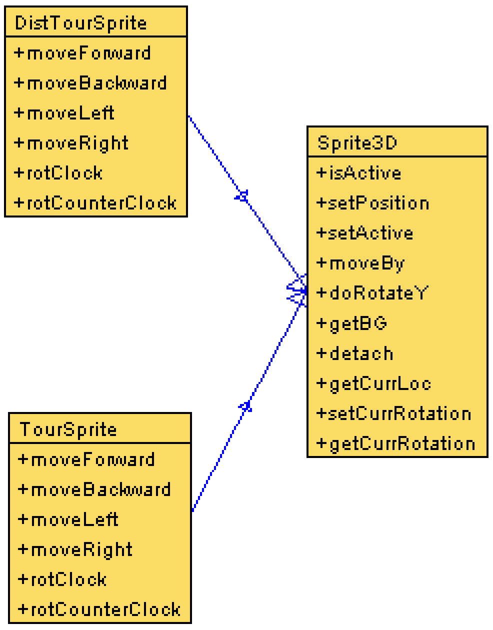


Figure 4. The Sprite Classes.

TourSprite and DistTourSprite offer a simplified interface for Sprite3D, fixing the sprite's rate of movement, and rotation increment. TourSprite also contains networking code to send its details to the server.

The version of Sprite3D in NetTour3D is very similar to the one in Tour3D. The main differences are in the subgraph created for a sprite, which looks like the one in Figure 5.

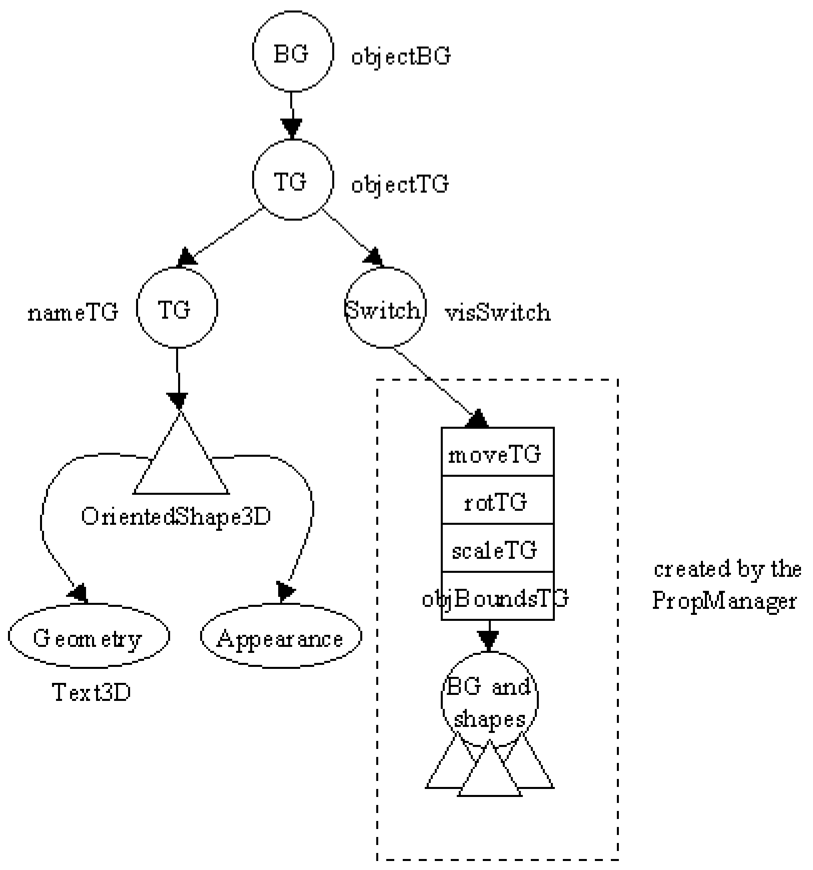


Figure 5. The Subgraph for the Sprite3D Sprite.

The subgraph has a BranchGroup node at its top, with capabilities set to make the branch detachable. This permits a sprite to be removed from the scene when the client leaves the world.

The other change is the addition of an OrientedShape3D shape holding the client's name, and set to rotate around the y-axis to follow the client's viewpoint. The shape is added to the TransformGroup above the Switch node, which means that if the sprite is made inactive (invisible), its name will remain on-screen.

The subgraph for the 3D model, together with its adjustments in size, position, and orientation, are handled by a PropManager object. Each sprite on the client-side uses a PropManager object to load a copy of the model.

The subgraph shown in Figure 5 is built in Sprite3D's constructor and makeName().

The movement and rotation code in Sprite3D works in the same way as in Tour3D: changes are made to the top-level TransformGroup, objectTG. Moves are first checked with the Obstacles object, before being carried out.

7.1. The TourSprite Class

TourSprite offers a simplified interface for moving and rotating the local sprite. It also communicates movements, rotations, and its initial creation to the server-side.

```
public class TourSprite extends Sprite3D
{
  private final static double MOVERATE = 0.3;
  private final static double ROTATE_AMT = Math.PI / 16.0;

  PrintWriter out;    // for sending commands to the server

  public TourSprite(String userName, String fnm, Obstacles obs,
                      double xPosn, double zPosn, PrintWriter o)
  { super(userName, fnm, obs);
    setPosition(xPosn, zPosn);
    out = o;
    out.println("create " + userName + " " + xPosn + " " + zPosn);
  }

  // moves
  public boolean moveForward()
  { out.println("forward");
    return moveBy(0.0, MOVERATE);
  }

  public boolean moveBackward()
  { out.println("back");
    return moveBy(0.0, -MOVERATE);
  }

  public boolean moveLeft()
  { out.println("left");
    return moveBy(-MOVERATE,0.0);
  }

  public boolean moveRight()
  { out.println("right");
    return moveBy(MOVERATE,0.0);
  }

  // rotations in Y-axis only
  public void rotClock()
  { out.println("rotClock");
    doRotateY(-ROTATE_AMT); // clockwise
  }

  public void rotCounterClock()
  { out.println("rotCClock");
    doRotateY(ROTATE_AMT);  // counter-clockwise
  }

}  // end of TourSprite
```

8. Creating a Local Sprite

WrapNetTour3D creates a local sprite by invoking a TourSprite object, and adding it to the scene graph. As part of TourSprite's construction, a "create n x z" message is sent to the server (n is its client's name, and (x,z) is its position on the XZ plane).

The pattern of communication following on from the sending of the "create" message is shown by the activity diagram in Figure 6.

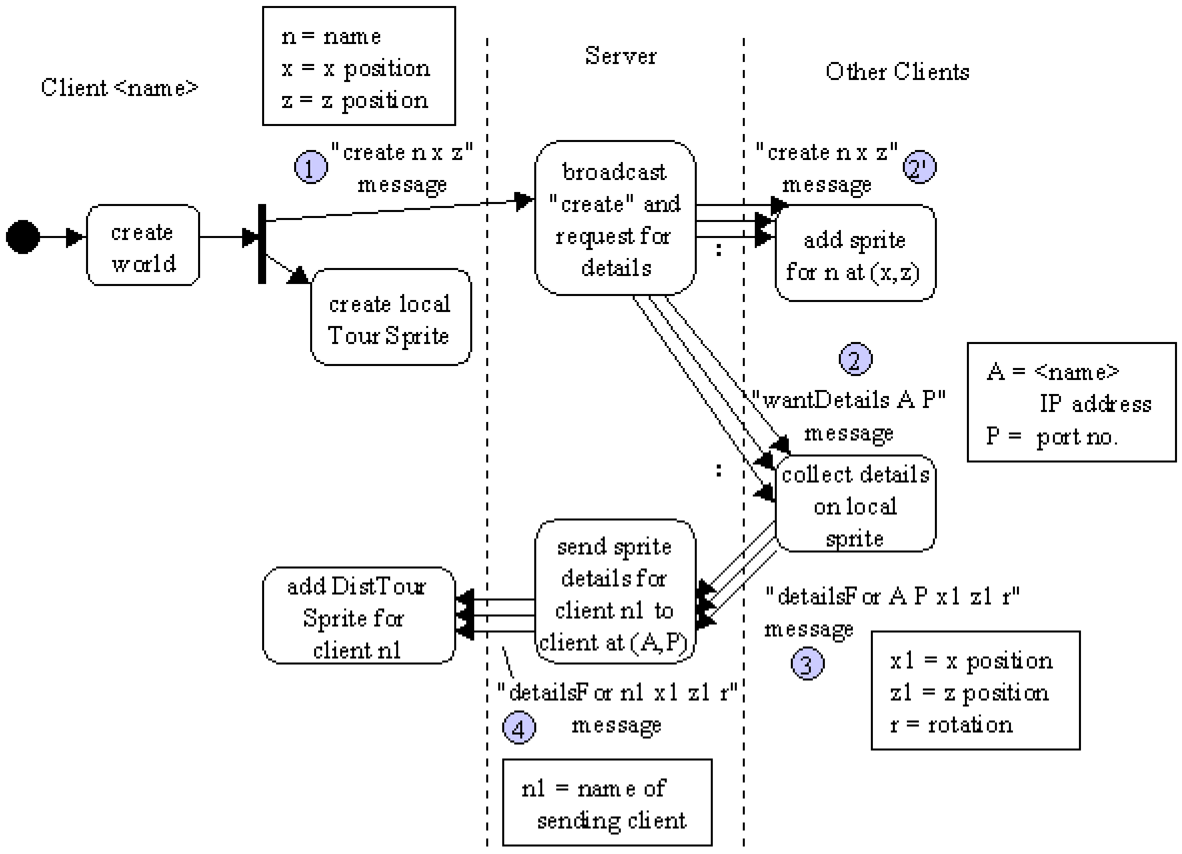

n = name
x = x position
z = z position
Client <name>
Server
Other Clients
1
"create n x z" message
create world
create local Tour Sprite
broadcast "create" and request for details
"create n x z" message
2'
add sprite for n at (x,z)
2
"wantDetails A P" message
A = <name> IP address
P = port no.
collect details on local sprite
send sprite details for client n1 to client at (A,P)
add DistTour Sprite for client n1
"detailsFor A P x1 z1 r" message
3
x1 = x position
z1 = z position
r = rotation
"detailsFor n1 x1 z1 r" message
4
n1 = name of sending client

Figure 6. Creating a Local Sprite.

The "create" message must be broadcast by the server to all the other current clients, as represented by the message labeled (2') in Figure 6.

Also, the new client must populate its copy of the world with distributed sprites representing the other users. This task is started by the server sending a "wantDetails" message to all the other clients (message (2)), which triggers a series of "detailsFor" replies (message (3)), which are passed back to the new client as messages of type (4).

The reception of the “create” and “wantDetails” messages in the other clients are handled by their TourWatcher threads, and TourWatcher in the new client deals with the “detailsFor” replies. Each “detailsFor” message causes a distributed sprite (an object of the DistTourSprite class) to be added to the client’s world.

9. Moving and Rotating a Local Sprite

Each movement and rotation of the local sprite has the side-effect of sending a message to the server. This is illustrated by the activity diagram in Figure 7.

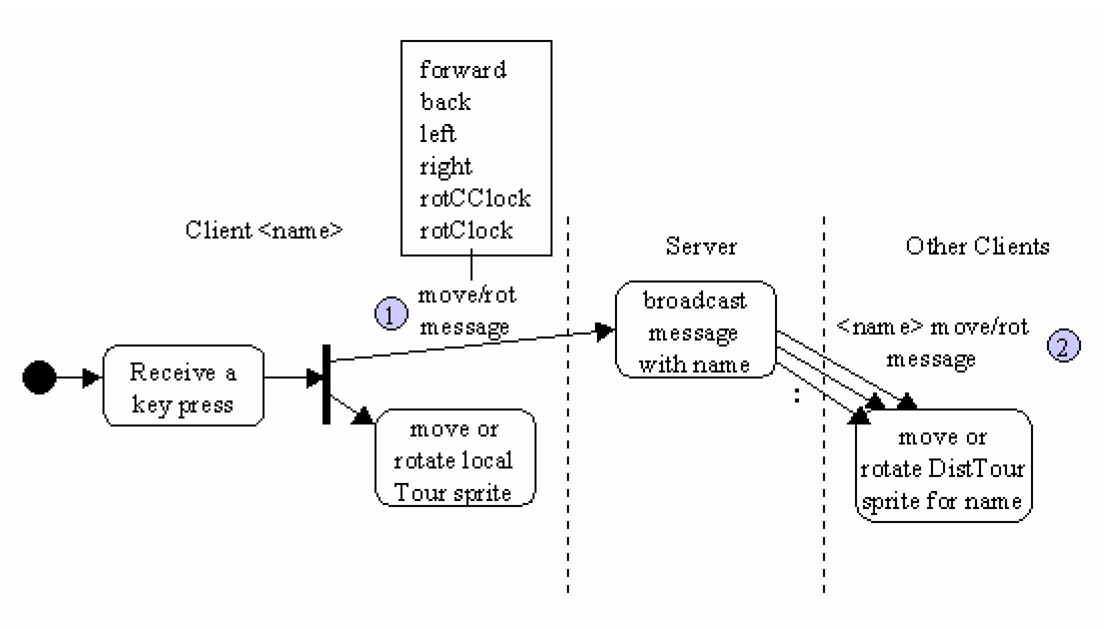


Figure 7. Moving/Rotating a Local Sprite.

The various messages are listed in the box in Figure 7; their transmission can be seen in the code for TourSprite given earlier in section 7.1.

The server must broadcast the messages to all the other clients in order that the distributed sprites representing the user can be moved or rotated. The message is prefixed with the user's name before being delivered to the TourWatcher threads of the other clients. This permits the TourWatchers to determine which distributed sprite to affect.

10. The Departure of a Local Sprite

When the user wants to leave the world, he/she will click the close box of the NetTour3D JFrame. This triggers a call to closeLink() in the WrapNetTour3D object:

```
public void closeLink()
{ try {
    out.println("bye");    // say bye to server
    sock.close();
  }
  catch(Exception e)
  {  System.out.println("Link terminated");  }

  System.exit( 0 );
}
```

The “bye” message causes the server to notify all the other clients of the user’s departure, as shown by the activity diagram in Figure 8.

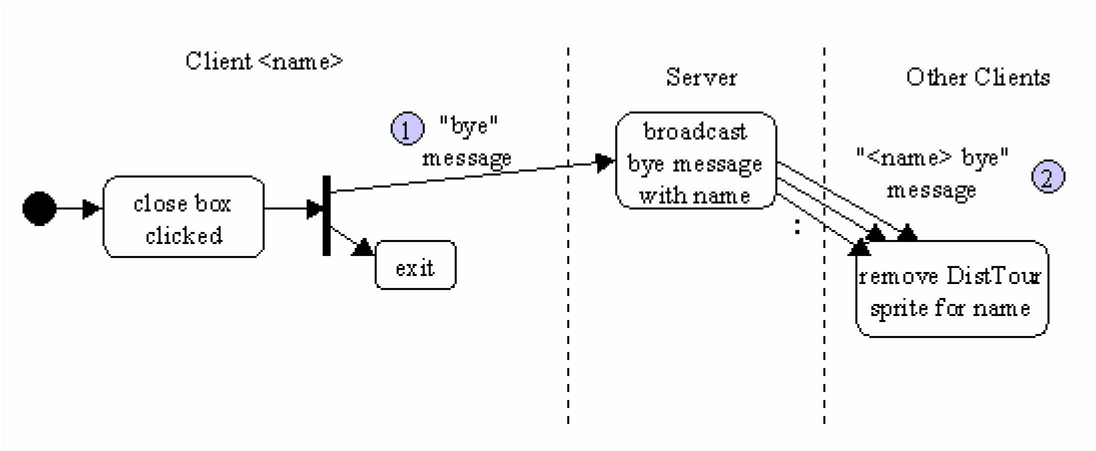


Figure 8. The Departure of a Local Sprite.

The TourWatcher threads for each of the clients receives a “bye” message, and uses the name prefix to decide which of the distributed sprites should be detached from the scene graph.

11. The TourWatcher Class

A TourWatcher thread monitors the server's output, waiting for messages. The message types are listed below, together with a brief description of what the TourWatcher does in response.

- create n x z
  Create a distributed sprite in the local world with name n, at position (x,0,z). By default, the sprite will face forwards along the positive z-axis.
- wantDetails A P
  The client at IP address A and port P is requesting information about the local sprite on this machine. Gather the data and send it back in a "detailsFor" message.
- detailsFor n1 x1 z1 r
  Create a distributed sprite with the name n1 at location (x1,0,z1), rotated r radians away from the positive z-axis.
- n <move or rotation command>
  <command> can be one of: forward, back, left, right, rotCClock, rotClock.
  The distributed sprite with name n is moved or rotated. rotCClock is a counter-clockwise rotation, rotClock is clockwise.
- n bye
  A client has left the world, so the distributed sprite with name n is detached (deleted).

The activities using these messages are shown in Figures 6, 7, and 8.

Almost all the messages are related to distributed sprites in the local world: creation, movement, rotation, and deletion. Therefore, these tasks are handled by TourWatcher itself, which maintains its sprites in a HashMap, mapping sprite name to DistTourSprite object:

```
private HashMap visitors;   // stores (name, sprite object) pairs
```

The run() method in TourWatcher accepts a message from the server, and tests the first word in the message to decide what to do.

```
public void run()
{ String line;
  try {
    while ((line = in.readLine()) != null) {
      if (line.startsWith("create"))
        createVisitor( line.trim() );
      else if (line.startsWith("wantDetails"))
        sendDetails( line.trim() );
     else if (line.startsWith("detailsFor"))
        receiveDetails( line.trim() );
      else
        doCommand( line.trim() );
    }
  }
  catch(Exception e) // socket closure causes termination of while
  { System.out.println("Link to Server Lost");
    System.exit( 0 );
  }
}
```

11.1. Creating a Distributed Sprite

A distributed sprite is made in response to a “create n x z” message, by creating a DistTourSprite object with name n at location (x,0,z), oriented along the positive z-axis. The name and sprite object are stored in the visitors HashMap for future reference.

```
private void createVisitor(String line)
{
  StringTokenizer st = new StringTokenizer(line);

  st.nextToken();   // skip 'create' word
  String userName = st.nextToken();
  double xPosn = Double.parseDouble( st.nextToken() );
  double zPosn = Double.parseDouble( st.nextToken() );

  if (visitors.containsKey(userName))
    System.out.println("Duplicate name -- ignoring it");
  else {
    DistTourSprite dtSprite =
                w3d.addVisitor(userName, xPosn, zPosn, 0);
    visitors.put( userName, dtSprite);
  }
}
```

A potential problem is if the proposed name has already been used for another sprite. TourWatcher only prints an error message on stdout; it would be better if a message was also sent back to the originating client.

11.2. The DistTourSprite Class

DistTourSprite is a simplified version of TourSprite: it's sprite movement and rotation interface is the same as TourSprite's, but DistTourSprite doesn't send messages to the server.

```
public class DistTourSprite extends Sprite3D
{
  private final static double MOVERATE = 0.3;
  private final static double ROTATE_AMT = Math.PI / 16.0;

  public DistTourSprite(String userName, String fnm, Obstacles obs,
                              double xPosn, double zPosn)
  { super(userName, fnm, obs);
    setPosition(xPosn, zPosn);
  }

  // moves
  public boolean moveForward()
  { return moveBy(0.0, MOVERATE); }

  public boolean moveBackward()
  { return moveBy(0.0, -MOVERATE); }

  public boolean moveLeft()
  { return moveBy(-MOVERATE,0.0); }

  public boolean moveRight()
  { return moveBy(MOVERATE,0.0); }

  // rotations in Y-axis only
  public void rotClock()
  { doRotateY(-ROTATE_AMT); }   // clockwise

  public void rotCounterClock()
  { doRotateY(ROTATE_AMT); }  // counter-clockwise

}
```

11.3. Moving and Rotating a Distributed Sprite

doCommand() in TourWatcher distinguishes between the various move and rotation messages, and also detects “bye”.

```
  private void doCommand(String line)
  {
    StringTokenizer st = new StringTokenizer(line);
    String userName = st.nextToken();
    String command = st.nextToken();
    DistTourSprite dtSprite =
                 (DistTourSprite) visitors.get(userName);
    if (dtSprite == null)
      System.out.println(userName + " is not here");
    else {
      if (command.equals("forward"))
        dtSprite.moveForward();
      else if (command.equals("back"))
        dtSprite.moveBackward();
      else if (command.equals("left"))
        dtSprite.moveLeft();
      else if (command.equals("right"))
        dtSprite.moveRight();
      else if (command.equals("rotCClock"))
        dtSprite.rotCounterClock();
      else if (command.equals("rotClock"))
        dtSprite.rotClock();
      else if (command.equals("bye")) {
        System.out.println("Removing info on " + userName);
        dtSprite.detach();
        visitors.remove(userName);
      }
      else
        System.out.println("Do not recognise the command");
    }
  }  // end of doCommand()
```

All of the commands start with the sprite's name, which is used to lookup the DistTourSprite object in the visitors HashMap. If the object cannot be found then TourWatcher only notifies the local machine, it should really also send an error message back to the original client.

The various moves and rotations are mapped to calls to methods in the DistTourSprite object.

The "bye" message causes the sprite to be detached from the local world's scene graph, and removed from the HashMap.

11.4. Responding to Sprite Detail Requests

Figure 6 shows that a "wantDetailsA P" message causes TourWatcher to collect information about the sprite local to this machine.

The details are sent back as a "detailsFor A P x1 z1 r" message to the client at IP address A and port P. The information states that the sprite is currently positioned at (x1,0,z1), and rotated r radians away from the positive z-axis.

TourWatcher doesn't manage the local sprite, and so passes the "wantDetails" request to the WrapNetTour3D object for processing.

```
  private void sendDetails(String line)
  { StringTokenizer st = new StringTokenizer(line);
    st.nextToken(); // skip 'wantDetails' word
    String cliAddr = st.nextToken();
    String strPort = st.nextToken();   // don't parse
    w3d.sendDetails(cliAddr, strPort);
  }
```

sendDetails() in WrapNetTour3D can easily access the local sprite (referred to as bob), and construct the necessary reply.

```
  public void sendDetails(String cliAddr, String strPort)
  {  Point3d currLoc = bob.getCurrLoc();
     double currRotation = bob.getCurrRotation();
     String msg = new String("detailsFor " + cliAddr + " " +
                        strPort + " " +
                        df.format(currLoc.x) + " " +
                        df.format(currLoc.z) + " " +
                        df.format(currRotation) );
     out.println(msg);
  }
```

The (x,z) location is formatted to 4 decimal places to reduce the length of the string sent over the network.

11.5. Receiving Other Client's Sprite Details

Figure 6 shows that when a user joins the world, it will be sent "detailsFor" messages by the existing clients. Each of these messages is received by TourWatcher, and leads to the creation of a distributed sprite.

TourWatcher's receiveDetails() method pulls apart a "detailsFor n1 x1 z1 r" message and creates a DistTourSprite with name n1 at (x1,0,z1), rotation r.

```
  private void receiveDetails(String line)
  {
    StringTokenizer st = new StringTokenizer(line);

    st.nextToken(); // skip 'detailsFor' word
    String userName = st.nextToken();
    double xPosn = Double.parseDouble( st.nextToken() );
    double zPosn = Double.parseDouble( st.nextToken() );
    double rotRadians = Double.parseDouble( st.nextToken() );

    if (visitors.containsKey(userName))
      System.out.println("Duplicate name -- ignoring it");
    else {
      DistTourSprite dtSprite =
                w3d.addVisitor(userName, xPosn, zPosn, rotRadians);
      visitors.put( userName, dtSprite);
    }
  }
```

The new sprite must be added to the local world's scene graph, and so is created in WrapNetTour3D by addVisitor().

11.6. Dynamically Adding to a Scene Graph

addVisitor() creates the distributed sprite and adds it to the scene.

```
public DistTourSprite addVisitor(String userName,
          double xPosn, double zPosn, double rotRadians)
{
   DistTourSprite dtSprite =
    new DistTourSprite(userName,"Coolrobo.3ds",obs,xPosn,zPosn);
   if (rotRadians != 0)
     dtSprite.setCurrRotation(rotRadians);

   BranchGroup sBG = dtSprite.getBG();
   sBG.compile();    // generally a good idea
   try {
     Thread.sleep(200);    // delay a little, so world is finished
   }
   catch(InterruptedException e) {}
   sceneBG.addChild( sBG );

   if (!sBG.isLive())    // just in case, but problem seems solved
     System.out.println("Visitor Sprite is NOT live");
   else
     System.out.println("Visitor Sprite is now live");

   return dtSprite;
 }
```

Two important elements of this code are that the subbranch for the distributed sprite is compiled, and that the method delays for 200ms before adding it to the scene. Without these extras, the new BranchGroup, sBG, *sometimes* fails to become live, which means that it subsequently cannot be manipulated (e.g. its TransformGroup cannot be adjusted to move/rotate the sprite).

The problem appears to be due to the threaded nature of the client-side: WrapNetTour3D may be building the world's scene graph at the same time that TourWatcher is receiving "detailsFor" messages, and so adding new branches to the self same graph. It is (just about) possible that addVisitor() is called before the scene graph has been compiled (and made live) in createSceneGraph(). This means that Java 3D will be asked to add a branch (sBG) to a node (sceneBG) which is not yet live, so causing the attachment to fail.

Our solution is to delay the attachment by 200ms, which solves the problem, at least in the many, many tests we've carried out.

Another thread-related problem of this type is when two (or more) threads attempt to add branches to the same live node at the same time. This may cause one or more of the attachments to fail to become live. The solution is to add synchronization code to the method which is doing the attachment, preventing multiple threads from executing it concurrently.

That problem does not arise in NetTour3D since new branches are only added to a client by a single TourWatcher thread.

12. Server-Side Activities

The processing done by the server is illustrated in Figures 6, 7, and 8, and is of two types:

- a message arrives and is *broadcast* to all the other clients;
- a "detailsFor" message arrives for a specified client, and is routed to that client. This is a *client-to-client* message.

12.1. Broadcasting

The most complex broadcasting is triggered by the arrival of a "create" message at the server.

Figure 6 shows how "create" fits into the overall activity of creating a new sprite. Figure 9 expands the "broadcast 'create' and request for details" box in the server swimlane of Figure 6.

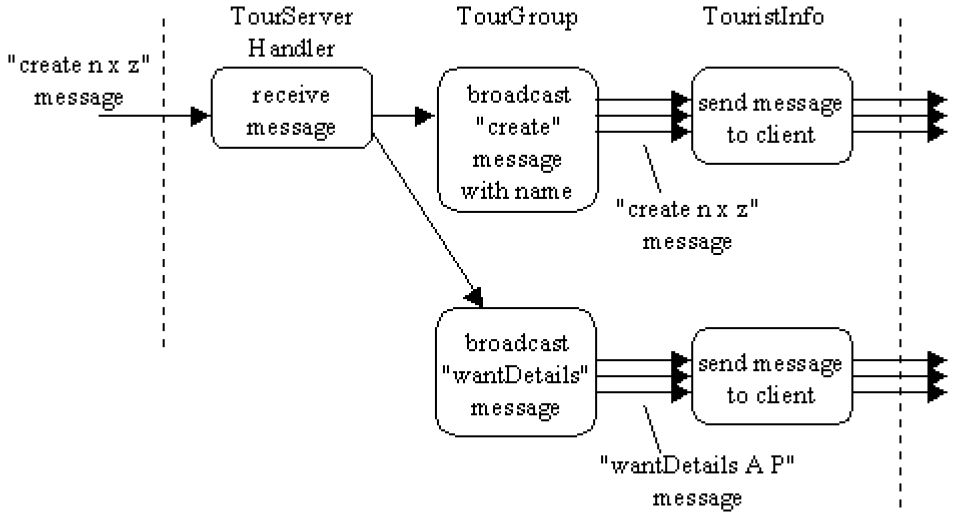


Figure 9. Server-side Activities for a "create" Message.

TourServer is not involved at all: it's role is to create TourServerHandlers.

TourServerHandler is principally concerned with differentiating between the messages it receives.

TourGroup handles the two modes of client communication: broadcasting or client-to-client. TourGroup also maintains an ArrayList of TouristInfo objects, which contain the output streams going to the clients.

When a "create n x z" message arrives at the TourServerHandler, it is passed to doRequest() which decides how to process it (by calling sendCreate()).

```
private void doRequest(String line, PrintWriter out)
{
  if (line.startsWith("create"))
    sendCreate(line);
  else if (line.startsWith("detailsFor"))
    sendDetails(line);
  else  // use TourGroup object to broadcast the message
    tg.broadcast(cliAddr, port, userName + " " + line);
}
```

sendCreate() extracts the sprite's name from the message, and stores it for later use. It then uses the TourGroup object to broadcast "wantDetails" and "create" messages to the other clients.

```
private void sendCreate(String line)
{
  StringTokenizer st = new StringTokenizer(line);
  st.nextToken(); // skip 'create' word
  userName = st.nextToken();       // userName is a global
  String xPosn = st.nextToken();   // don't parse
  String zPosn = st.nextToken();   // don't parse

  // request details from other clients
  tg.broadcast(cliAddr, port, "wantDetails " + cliAddr + " " + port);

  // tell other clients about the new one
  tg.broadcast(cliAddr,port,"create "+userName +" "+xPosn+" "+zPosn);
}
```

The broadcast() method in TourGroup iterates through its TouristInfo objects, and sends the message to all of them, except the client that transmitted the message originally.

```
  synchronized public void broadcast(String cliAddr, int port,
                                          String msg)
  { TouristInfo c;
    for(int i=0; i < tourPeople.size(); i++) {
      c = (TouristInfo) tourPeople.get(i);
      if (!c.matches(cliAddr, port))
        c.sendMessage(msg);
    }
  }
```

All the methods in TourGroup are synchronized, since the same TourGroup object is shared between all the TourServerHandler threads. The synchronization prevents a TouristInfo object being affected by more than one thread at a time.

sendMessage() in TouristInfo places the message on the output stream going to its client.

```
  public void sendMessage(String msg)
  {  out.println(msg);  }
```

12.2. Client-to-Client Message Passing

Client-to-client message passing is only used to pass a "detailsFor" message to a client.

Figure 6 shows how "detailsFor" fits into the overall activity of creating a new sprite. Figure 10 expands the "send sprite details for client n1 to client at (A,P)" box in the server swimlane of Figure 6.

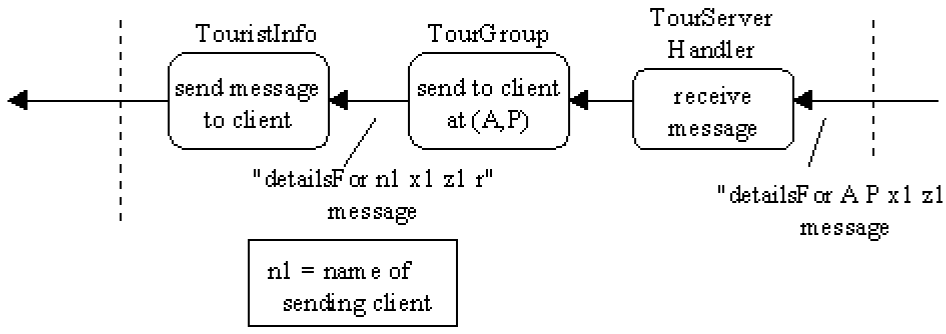


Figure 10. Server-side Activities for a "detailsFor" Message.

The TourServerHandler processes the message in doRequest(), as shown in the last section, and calls sendDetails().

```
private void sendDetails(String line)
{
  StringTokenizer st = new StringTokenizer(line);

  st.nextToken(); // skip 'detailsFor' word
  String toAddr = st.nextToken();
  int toPort = Integer.parseInt( st.nextToken() );
  String xPosn = st.nextToken();        // don't parse
  String zPosn = st.nextToken();        // don't parse
  String rotRadians = st.nextToken();   // don't parse

  tg.sendTo(toAddr, toPort,  "detailsFor " + userName + " " +
               xPosn + " " + zPosn +  " " + rotRadians);
}
```

sendDetails() passes the message onto TourGroup's sendTo() method. However, the client IP address and port number are extracted first, to 'aim' the message at the right recipient. Also, the name of the sprite (userName) is added to the message, obtained from a global in TourServerHandler.

TourGroup's sendTo() cycles through its ArrayList of TouristInfo objects until it finds the right client, and sends the message.

```
synchronized public void sendTo(String cliAddr,int port,String msg)
{
  TouristInfo c;
  for(int i=0; i < tourPeople.size(); i++) {
    c = (TouristInfo) tourPeople.get(i);
    if (c.matches(cliAddr, port)) {
      c.sendMessage(msg);
      break;
    }
  }
}  // end of sendTo()
```

13. NetTour3D and NVEs

The key elements that make up a fully featured NVE are: spaces, users, objects, views, and the notions of consistency, real-time, dead reckoning, security, and scalability. How does NetTour3D measure up against these?

NetTour3D only utilizes a single space (the checkboard), and users all have the same appearance (but with different names!). However, the local and ghost avatars idea is present, as represented by the TourSprite and DistTourSprite classes. The only view is a third person camera. Objects, in the form of scenery and obstacles, can be easily added to the world, but they are static; not mobile, reactive, or intelligent.

There is actually no way for NetTour3D users to interact. However, it is not difficult to add a multi-user chat component to NetTour3D. The multi-threaded chat example in chapter 19 is a good source.

None of the NVE ideas of consistency, real-time, dead reckoning, security, and scalability are addressed in this example.

A TCP/IP client/server communication model is employed in NetTour3D, whereas a more realistic approach would include UDP multicasting and peer-to-peer elements, and deploy some kind of connection manager.

These are just some of the reasons why I am not planning to release NetTour3D as a commercial product ☺.