Chapter 20. A Networked Two-Person Game

This chapter describes a networked two-person 3D game. The initial implementation lacks networking components, so we can concentrate on issues such as game logic, 3D modeling, and user interface design. In the second stage, networking elements are added, with fairly minimal changes to the basic game.

FourByFour is a 3D version of tic-tac-toe, as shown in Figure 1. Player 1's markers are red spheres, player 2 has blue cubes. The aim is to create a line of four markers, all of the same type, along the x-, y-, or z- axes, or the diagonals across the XY, XZ, or YZ planes, or from corner-to-corner.

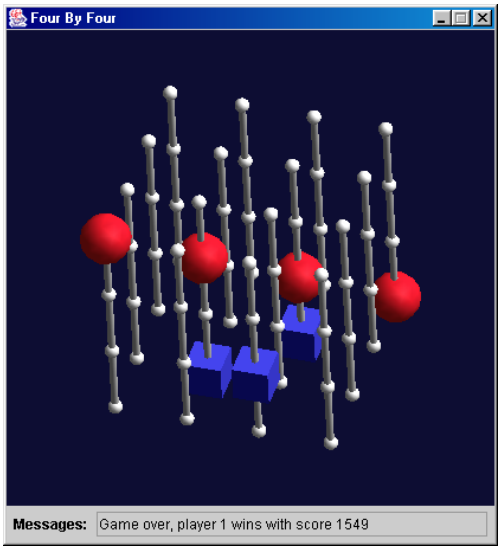

Figure 1. FourByFour in Action.

The non-networked FourByFour game described in this chapter is a simplified version of a game available in the Java 3D distribution.

The main programming features of FourByFour are:

- the use of *parallel projection* so that the poles and markers at the back of the scene do not appear smaller than those at the front;
- the use of Switch nodes to hide the red and blue markers on screen until they are required by a player;
- a PickDragBehavior class that allows a player to select a position by clicking with the mouse, and to rotate the game board via mouse dragging;
- another example of picking, somewhat simpler than those in the Shooting and Fractal landscape examples (chapters 14 and 17).

The networked version of FourByFour, called NetFourByFour, is an instance of a *threaded client and server*, similar in style to the threaded Chat application of chapter

19. However, there are additional requirements for a networked two-person game, including the need to order the players' turns, and to control the number of players involved.

The main contribution of NetFourByFour is to illustrate strategies for designing and implementing networked two-person games:

- the development of the application in two stages, as mentioned above;
- the use of network protocol diagrams to specify the interactions between the clients and server;
- the use of UML activity and sequence diagrams to trace turn-taking from one player, through the server, to the other player.

The NetFourByFour GUI also extends the Canvas3D class to place *overlays* on-screen (see Figure 11).

1. UML Class Diagram for FourByFour

Figure 2 gives the UML class diagram for all the classes in the FourByFour application. The class names and public methods are shown.

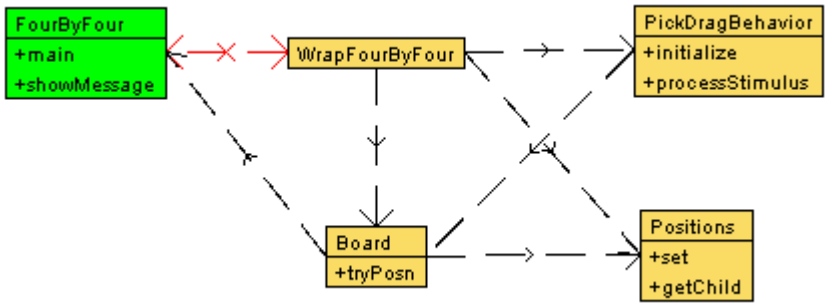

Figure 2. UML Class Diagram for FourByFour.

The FourByFour class is the usual JFrame, which contains a GUI made up of a WrapFourByFour object for the 3D canvas, and a text field for messages. The public showMessage() method allows other objects to write into that field.

WrapFourByFour constructs the scene: the lights, background, the parallel projection, and 16 poles, but leaves the initialization of the markers to a Positions object. WrapFourByFour also creates a PickDragBehavior object to handle mouse picking and dragging.

The Board object contains the game logic, consisting of data structures representing the current state of the board, and methods for making a move and reporting a winner.

2. The Original FourByFour Application

The original FourByFour demo has been part of the Java 3D distribution for several years. It has a more extensive GUI than the version described in this chapter, supporting repeated games, varying skill levels, and a high scores list. A crucial

difference is that the demo pits *the machine* against a single player, rather than player versus player. The requires a much more complicated Board class, weighing in at 2300 lines (compared to 300 in our version of Board).

Increasing the machine's skill level translates into Board using increasingly comprehensive analyses of the player's moves and the current game state. Board also renders the game to a 2D window in addition to the 3D canvas.

The FourByFour demo utilizes its own versions of the Box and Cylinder utilities, and builds its GUI with AWT classes rather than Swing. It uses an ID class to number shapes; our code employs the userData field of the Shape3D class.

3. The WrapFourByFour class

The WrapFourByFour class uses the same coding style as earlier 3D examples: it creates the 3D canvas and adds the scene in createSceneGraph():

```
private void createSceneGraph(Canvas3D canvas3D, FourByFour fbf)
{
  sceneBG = new BranchGroup();
  bounds = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

  // Create the transform group which moves the game
  TransformGroup gameTG = new TransformGroup();
  gameTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  gameTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  sceneBG.addChild(gameTG);

  lightScene();          // add the lights
  addBackground();       // add the background

  gameTG.addChild( makePoles() );       // add poles

  // posns holds the spheres/boxes which mark a player's turn.
  // Initially posns displays a series of small white spheres.
  Positions posns = new Positions();

  // board tracks the players' moves on the game board
  Board board = new Board(posns, fbf);

  gameTG.addChild( posns.getChild() );  // add markers

  mouseControls(canvas3D, board, gameTG);

  sceneBG.compile();   // fix the scene
}
```


A TransformGroup, gameTG, is added below the scene's BranchGroup – it is used by the PickDragBehavior object to rotate the game when the mouse is dragged. Consequently, all the visible game objects (e.g. poles, markers) are linked to gameTG, while static entities (e.g. the lights, background) are connected to sceneBG.

makePoles() creates the 16 poles where the game markers appear. The initial positions of the poles are shown in Figure 3.

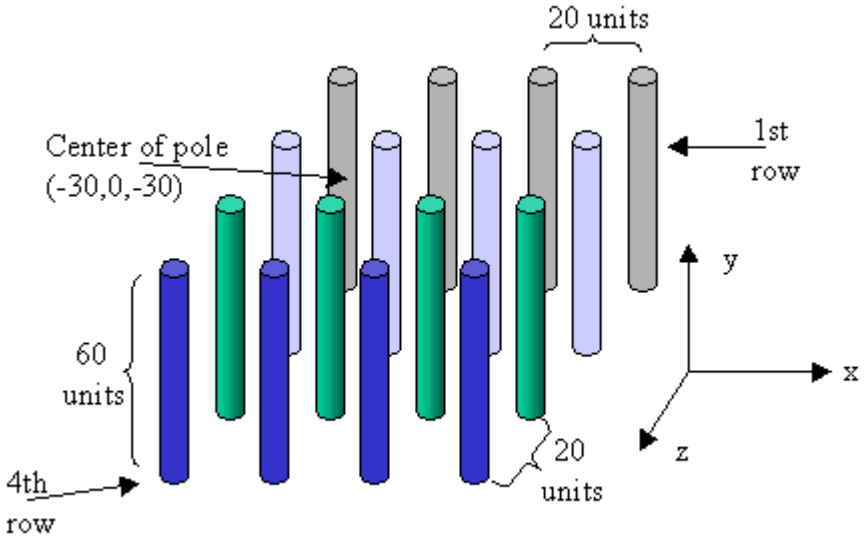

Figure 3. The Game Poles.

The first pole (the left most pole of the first row) is centered at (-30,0,30), and has length 60 units. The other three poles in the row are spaced at 20 unit intervals to the right, and the next row begins, 20 units along the z axis.

```
private BranchGroup makePoles()
{
   Color3f grey = new Color3f(0.25f, 0.25f, 0.25f);
   Color3f black = new Color3f(0.0f, 0.0f, 0.0f);
   Color3f diffuseWhite = new Color3f(0.7f, 0.7f, 0.7f);
   Color3f specularWhite = new Color3f(0.9f, 0.9f, 0.9f);

   // Create the pole appearance
   Material poleMaterial =
      new Material(grey, black, diffuseWhite,specularWhite,110.f);
   poleMaterial.setLightingEnable(true);
   Appearance poleApp = new Appearance();
   poleApp.setMaterial(poleMaterial);

   BranchGroup bg = new BranchGroup();
   float x = -30.0f;
   float z = -30.0f;

   for(int i=0; i<4; i++) {
     for(int j=0; j<4; j++) {
       Transform3D t3d = new Transform3D();
       t3d.set( new Vector3f(x, 0.0f, z) );
       TransformGroup tg = new TransformGroup(t3d);
       Cylinder cyl = new Cylinder(1.0f, 60.0f, poleApp);
```



```
        cyl.setPickable(false);  // user cannot select the poles
        tg.addChild( cyl );
        bg.addChild(tg);
        x += 20.0f;
      }
      x = -30.0f;
      z += 20.0f;
    }
    return bg;
  }  // end of makePoles()
```

A pole is represented by a Cylinder object, below a TransformGroup which positions it. The poles (and transforms) are grouped under a BranchGroup.

The cylinders are made unpickable, which simplifies the picking task in PickDragBehavior.

mouseControls() creates a PickDragBehavior object, attaching it to the scene.

```
private void mouseControls(Canvas3D c,Board board,
                                   TransformGroup gameTG)
{ PickDragBehavior mouseBeh =
      new PickDragBehavior(c, board, sceneBG, gameTG);
  mouseBeh.setSchedulingBounds(bounds);
  sceneBG.addChild(mouseBeh);
}
```

initUserPosition() modifies the view to use parallel projection, and moves the viewpoint along the +z axis so that the entire game board is visible and centered in the canvas.

```
private void initUserPosition()
{
  View view = su.getViewer().getView();
  view.setProjectionPolicy(View.PARALLEL_PROJECTION);

  TransformGroup steerTG =
       su.getViewingPlatform().getViewPlatformTransform();
  Transform3D t3d = new Transform3D();
  t3d.set(65.0f, new Vector3f(0.0f, 0.0f, 400.0f));
  steerTG.setTransform(t3d);
}
```


4. The Positions Class

The Positions object creates three sets of markers: 64 small white balls, 64 larger red balls, and 64 blue cubes. The white balls are visible, the other shapes invisible, when the game starts. When a player makes a move, the selected white ball is replaced by a red one (if it was player 1's turn) or a blue cube (for player 2).

This functionality is achieved by creating three Switch nodes, one for each set of markers, linked to the scene with a Group node. The scene graph branch representing this is shown in Figure 4.

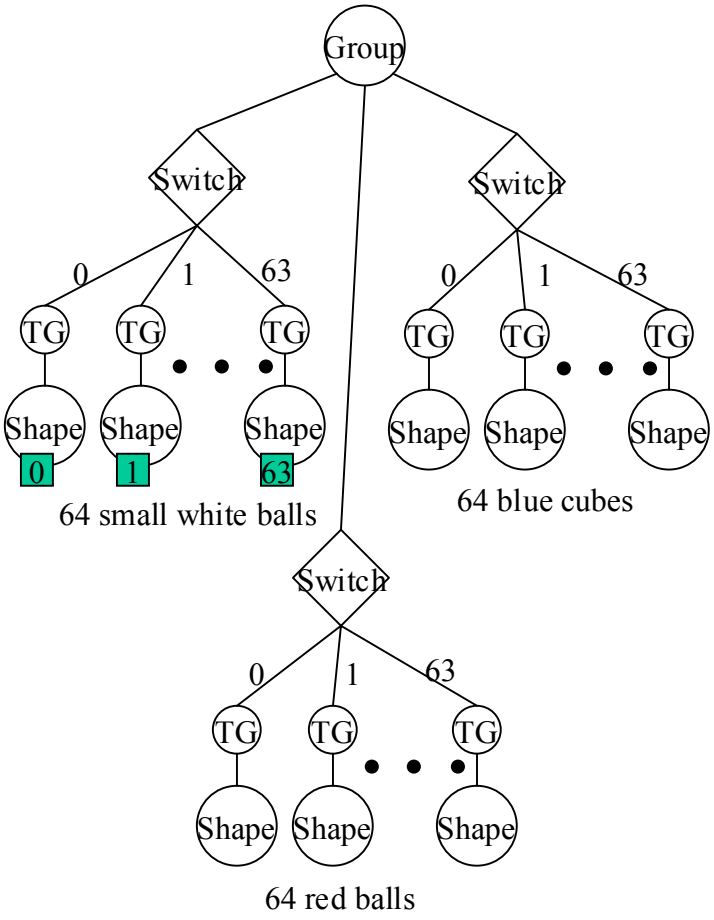

Figure 4. Scene Graph Branch for the Game Markers.

Each Shape3D is positioned with a TransformGroup and allocated a bit in a BitSet corresponding to its position in the game (positions are numbered 0 to 63). The BitSet is used as a mask in the Switch node to specify what shapes are visible/invisible.

The three Switch branches are created with calls to makeWhiteSpheres(), makeRedSpheres() and makeBlueCubes(), which are functionally almost the same. The code for makeWhiteSpheres():

```
private void makeWhiteSpheres()
{
   // Create the switch nodes
   posSwitch = new Switch(Switch.CHILD_MASK);
```



```
    // Set the capability bits
    posSwitch.setCapability(Switch.ALLOW_SWITCH_READ);
    posSwitch.setCapability(Switch.ALLOW_SWITCH_WRITE);
    posMask = new BitSet();   // create the bit mask

    Sphere posSphere;
    for (int i=0; i<NUM_SPOTS; i++) {
       Transform3D t3d = new Transform3D();
       t3d.set( points[i] );    // set position
       TransformGroup tg = new TransformGroup(t3d);
       posSphere = new Sphere(2.0f, whiteApp);
       Shape3D shape = posSphere.getShape();
       shape.setUserData( new Integer(i) );
              // add board position ID to each shape
       tg.addChild( posSphere );
       posSwitch.addChild(tg);
       posMask.set(i);    // make visible
    }
    // Set the positions mask
    posSwitch.setChildMask(posMask);

    group.addChild( posSwitch );
  } // end of makeWhiteSpheres()
```

All Shape3D objects (i.e. the game markers) are pickable by default, which means that the PickDragBehavior object will be able to select them.

An important feature of makeWhiteSpheres() is that each ball is assigned user data – an Integer object holding its position index. makeRedSpheres() and makeBlueCubes() do not set user data for their markers. (This difference is denoted by little numbered boxes in Figure 4.)

The integer field enables PickDragBehavior to determine the position of the white ball which a user has selected. Also if a red ball or blue cube is chosen, the absence of the integer field means that the selection should be ignored, since that position is already taken on the board.

Another difference between makeWhiteSpheres() and the other two methods is that the white balls are all set to be visible initially, while the red balls and blue cubes are invisible. This is changed during the course of game play by calls to set().

```
public void set(int pos, int player)
// called by Board to update the 3D scene
{
  // turn off the white marker for the given position
  posMask.clear(pos);
  posSwitch.setChildMask(posMask);

  // turn on one of the player markers
  if (player == PLAYER1) {
     player1Mask.set(pos);
     player1Switch.setChildMask(player1Mask);   // red for p1
  }
  else if (player == PLAYER2) {
     player2Mask.set(pos);
     player2Switch.setChildMask(player2Mask);   // blue for p2
  }
  else  // should not happen
     System.out.println("Illegal player value: " + player);
```



```
}
```

The pos argument is the position index (a number between 0 and 63), extracted from the user data field of the selected white marker. The player value represents the first or second player.

The main design choice in the Positions class is to create all the possible markers at scene creation time. This makes the scene's initialization a little slower, but the rendering speed for displaying a player's marker is improved because Shape3D nodes do not need to be attached or detached from the scene graph at run time.

The position indices for the markers (0-63) are tied to locations in space by the initLocations() method, which creates a points[] array of markers' coordinates. The positions correspond to the indices of the points[] array.

```
private void initLocations()
{
  points = new Vector3f[NUM_SPOTS];
  int count = 0;
  for (int z=-30; z<40; z+=20)
    for (int y=-30; y<40; y+=20)
      for (int x=-30; x<40; x+=20) {
        points[count] = new Vector3f((float)x,
                                     (float)y, (float)z);
        count++;
      }
}
```

The point[] array is used to initialize the TransformGroups for the markers, positioning them in space.

The coordinates were chosen so that the markers appear embedded in the poles. Figure 5 shows the first row of poles (the back row in Figure 3), and its 16 markers which have position indices 0-15.

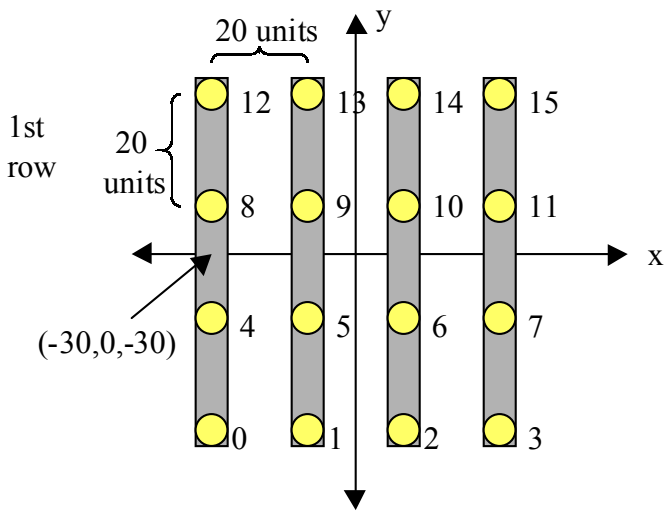

Figure 5. Marker Positions in the First Row of Poles.

5. PickDragBehavior

The PickDragBehavior object allows the user to click on a white marker to signal that the next move should be in that position, and to utilize mouse dragging to rotate the scene.

The behaviour actually monitors mouse drags, presses, *and* releases. A mouse release is employed as a simple way of detecting that a new mouse drag may be initiated during the next user interaction.

processStimulus() responds to the three mouse operations:

```
public void processStimulus(Enumeration criteria)
{
  WakeupCriterion wakeup;
  AWTEvent[] event;
  int id;
  int xPos, yPos;
  while (criteria.hasMoreElements()) {
      wakeup = (WakeupCriterion) criteria.nextElement();
      if (wakeup instanceof WakeupOnAWTEvent) {
         event = ((WakeupOnAWTEvent)wakeup).getAWTEvent();
         for (int i=0; i<event.length; i++) {
            xPos = ((MouseEvent)event[i]).getX();
            yPos = ((MouseEvent)event[i]).getY();
            id = event[i].getID();
            if (id == MouseEvent.MOUSE_DRAGGED)
              processDrag(xPos, yPos);
            else if (id == MouseEvent.MOUSE_PRESSED)
              processPress(xPos, yPos);
            else if (id == MouseEvent.MOUSE_RELEASED)
              isStartDrag = true;   // may be end of a drag
          }
      }
   }
   wakeupOn (mouseCriterion);
}
```

processDrag() handles mouse dragging, and is passed the current (x,y) position of the cursor on screen. processPress() deals with a mouse press, and the global boolean isStartDrag is set to true when the mouse is released.

5.1. Dragging the Board

When the user drags the mouse, a sequence of MOUSE_DRAGGED events are generated, each one including the current (x,y) position of the cursor.

processDrag() obtains the movement covered by a single MOUSE_DRAGGED event by calculating the offset relative to the (x,y) coordinate from the previous drag event (stored in xPrev and yPrev). The x and y components of the move are converted into x- and y- axis rotations and applied to the TransformGroup for the board.

However, this approach only works *after* the first event, so that the second event (and subsequent ones) have a previous coordinate to consider. The first event in a drag sequence is distinguished by the isStartDrag boolean.


```
private void processDrag(int xPos, int yPos)
{
   if (isStartDrag)
     isStartDrag = false;
   else {  // not the start of a drag, so can calculate offset
     int dx = xPos - xPrev;              // get dists dragged
     int dy = yPos - yPrev;
     transformX.rotX( dy * YFACTOR );    // convert to rotations
     transformY.rotY( dx * XFACTOR );
     modelTrans.mul(transformX, modelTrans);
     modelTrans.mul(transformY, modelTrans);
                    // add to existing x- and y- rotations
     boardTG.setTransform(modelTrans);
   }
   xPrev = xPos;     // save locs so can work out drag next time
   yPrev = yPos;
}
```

modelTrans is a global Transform3D object which stores the ongoing, total rotational effect on the board. transformX and transformY are also globals. boardTG is the TransformGroup for the board, passed in from WrapFourByFour when the PickDragBehavior object is created.

5.2. Picking a Marker

processPress() sends a pick ray along the z-axis into the world, starting from the current mouse press position. The closest intersecting node is retrieved and if it's a Shape3D containing a position index, then that position is used as the player's desired move.

One problem is translating the (x,y) position supplied by the MOUSE_PRESSED event into world coordinates. This is done in two stages: first the screen coordinate is mapped to the canvas' image plate, and then to world coordinates.

The picking code is considerably simplified by the judicious use of setPickable(false) when the scene is set up. The poles are made unpickable when created in makePoles() in WrapFourByFour, which means that only the markers can be selected.

```
// global
private final static Vector3d IN_VEC = new Vector3d(0.f,0.f,-1.f);
          // direction for picking -- into the scene

private Point3d mousePos;
private Transform3D imWorldT3d;
private PickRay pickRay = new PickRay();
private SceneGraphPath nodePath;
   :

private void processPress(int xPos, int yPos)
{
  canvas3D.getPixelLocationInImagePlate(xPos, yPos, mousePos);
               // get the mouse position on the image plate
  canvas3D.getImagePlateToVworld(imWorldT3d);
                 // get image plate --> world transform
  imWorldT3d.transform(mousePos);   // convert to world coords
```



```
    pickRay.set(mousePos, IN_VEC);
                // ray starts at mouse pos, and goes straight in

    nodePath = bg.pickClosest(pickRay);
            // get 1st node along pickray (and its path)
    if (nodePath != null)
      selectedPosn(nodePath);
  }
```

The image plate to world coordinates transform is obtained from the canvas, and applied to mousePos, which changes it in place. A ray is sent into the scene starting from that position, and the closest SceneGraphPath object is retrieved. This should be a branch ending in a game marker or null (i.e. the user clicked on a pole or the background).

selectedPosn() gets the terminal node of the path, and checks that it is a Shape3D containing user data (only the white markers hold data: their position index).

```
  private void selectedPosn(SceneGraphPath np)
  {
    Node node = np.getObject();       // get terminal node of path
    if (node instanceof Shape3D) {    // check for shape3D
      Integer posID = (Integer) node.getUserData();  //get posn index
      if (posID != null)
        board.tryPosn( posID.intValue() );
    }
  }
```

The position index (as an int) is passed to the Board object where the game logic is located.

If a red or blue marker is selected, the lack of user data will stop any further processing – it is not possible for a player to make a move in a spot which has already been used.

5.3. Picking Comparisons

This is the third example of Java 3D picking in the book, and it is worth comparing the three approaches.

In chapter 14, picking was used to select a point in the scene, and a gun rotated and shot at it. The picking was coded using a subclass of PickMouseBehavior, and details about the intersection coordinate were required.

In chapter 17, a ray was shot straight down from the user's position in a landscape, to get the floor height of the spot where they were standing. The picking was implemented with PickTool, and an intersection coordinate was again necessary.

The picking employed in this chapter does not use any of the picking utilities (i.e. PickMouseBehavior, PickTool), and only requires the shape that is first touched by the ray. Consequently, the task is simple enough to code directly, although the conversion from screen to world coordinates is somewhat tricky.

6. The Board Class

The Board object initializes two arrays when it is first created: winLines[][], and posToLines[][].

winLines[][] lists all the possible winning lines in the game, in terms of the four positions which make up a line. For example, referring to Figure 5, {0,1,2,3}, {3,6,9,12}, and {0,4,8,12} are winning lines; there are a total of 76 winning lines in the game.

For each line, winLines[][] also records the number of positions currently occupied by a player. If the total reaches four for a particular line, then the player has completed the line, and won.

posToLines[] specifies all the lines which utilize a given position. Thus, when a player selects a given position, all of those lines can be updated at once.

The main entry point into Board is tryPosn(), called by PickDragBehavior to pass the player's selected position into the Board object for processing.

```
public void tryPosn(int pos)
{
  if (gameOver)   // don't process position when game is over
    return;

  positions.set(pos, player);  // change the 3D marker shown at pos
  playMove(pos);               // play the move on the board

  // switch players, if the game isn't over
  if (!gameOver) {
    player = ((player == PLAYER1) ? PLAYER2 : PLAYER1 );
    if (player == PLAYER1)
      fbf.showMessage("Player 1's turn (red spheres)");
    else
      fbf.showMessage("Player 2's turn (blue cubes)");
  }
}  // end of tryPosn()
```

Board uses a global boolean, gameOver, to record when the game is over. The test of gameOver at the start of tryPosn() means that selecting a marker will have no effect once the game is over.

The player's marker is made visible by a call to set() in the Positions object, and playMove() updates the winLines[][] array. After the move, the current player is switched – the player variable holds the current player's ID. However, the move may have been a winning one, and so gameOver is checked before the switch.

The calls to showMessage() cause the text field in the GUI to be updated.

playMove() uses the supplied position index to modify the various lines in which it appears. If the number of used positions in any of those lines reaches 4, then the player has won, and reportWinner() is called.

```
private void playMove(int pos)
{
   nmoves++;                     // update the number of moves
```



```
    // get number of lines that this position is involved in
    int numWinLines = posToLines[pos][0];

    /* Go through each line associated with this position
       and update its status. If we have a winner, stop game. */

    int line;
    for (int j=0; j<numWinLines; j++) {
      line = posToLines[pos][j+1];
      if (winLines[line][1] != player &&
          winLines[line][1] != UNOCCUPIED)
        winLines[line][0] = -1;
        /* The other player has already made a move in this line
           so this line is now useless to both players. */
      else {
        winLines[line][1] = player;  //this line belongs to player
        winLines[line][0]++;         // one more posn used in line
        if (winLines[line][0] == 4) {  // all positions used,
          gameOver = true;             // so this player has won
          reportWinner();
        }
      }
  }
} // end of playMove()
```

The winLines[x][1] field for line x states whether a player has already made a move in the line or not. If a player selects a position in a line already used by another player, then the line becomes useless, which is signaled by setting winLines[x][0] == -1.

reportWinner() does some numerical 'hand waving' to obtain a score, based on the running time of the game and the number of moves made. The score is reported in the text field of the GUI.

```
private void reportWinner()
{
  long end_time = System.currentTimeMillis();
  long time = (end_time - startTime)/1000;

  int score = (NUM_SPOTS + 2 - nmoves) * 111 -
                 (int) Math.min(time*1000, 5000);

  if (player == PLAYER1)
    fbf.showMessage("Game over, player 1 wins with score "+score);
  else   // PLAYER2
    fbf.showMessage("Game over, player 2 wins with score "+score);
}  // end of reportWinner()
```


7. NetFourByFour

NetFourByFour is based on the FourByFour game, retaining most of its game logic, 3D modeling, and GUI interface, and adding a threaded client/server communications layer. This development sequence was quite deliberate, as it allowed most of the game-specific and user interface issues to be addressed before networking complexity was introduced.

Figure 6 shows the main functional components of NetFourByFour.

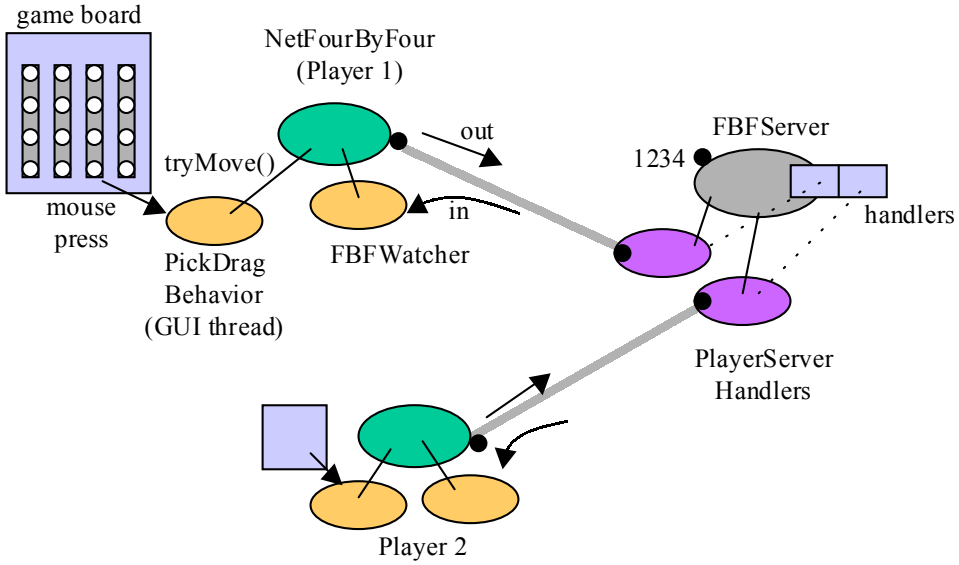

Figure 6. NetFourByFour Clients and Server.

The top-level server, FBFServer, creates two PlayServerHandler threads to manage communication between the players. The server and its threads are both 'thin': they carry out very little processing, acting mainly as a switch board for messages passing between the players.

An advantage of this approach is that most of the client's functionality can be borrowed from the standalone FourByFour, and the server-side is kept simple. Also, processing is carried out locally in the client, whereas server-side processing would introduce networking delays between the user's selection and the resulting changes in the game window. A drawback is the need to duplicate processing across the clients.

Each NetFourByFour client utilizes the standard GUI thread (where PickDragBehavior executes), the application thread for game processing, *and* a FBFWatcher thread to handle messages coming from the server. This threaded model was last seen in the chat application of chapter 19, but there are some differences when building two-person networked games.

One change is the restriction on the number of participants: a chat system allows any number of users, who may join and leave at any time. A two-person game can begin only when both players are present, and stops if one of them leaves. There cannot be more than two players, and we prohibit the mid-game change of players (i.e. substitutes aren't allowed).

Another complication is the need to impose an order on the game play: first player 1, then player 2, back to player 1, and so on. A chat system doesn't enforce any sequencing on its users.

The ordering criteria seems to suggest that a player (e.g. player 2) should wait until the other (e.g. player 1) has finished their move. The problem is that when player 2 is eventually notified of the finished move, it will have to duplicate most of player 1's processing in order to keep its own game state and display current. In other words, player 2's waiting time before its turn will be almost doubled.

Although latency is less of an issue in turn-based games, a doubling in wait time should be avoided. One solution is for player 2 to be notified as soon as player 1 has *selected* a move, before it has been executed and rendered.

This coding style is illustrated by the UML activity diagram in Figure 7, where the game engines for player 1 and 2 concurrently evaluate player 1's selected move.

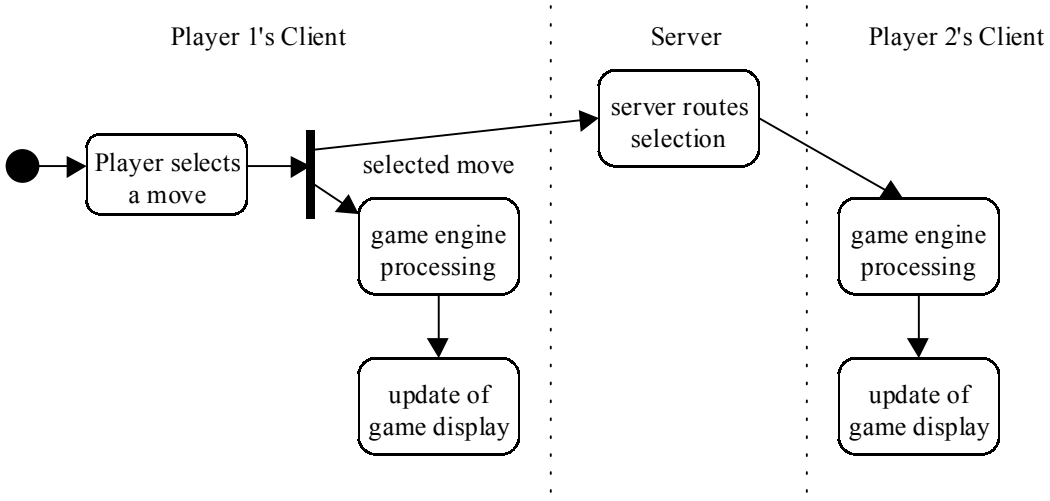

Figure 7. Concurrent Processing of Player 1's Move.

There are issues that need to be considered with this technique. One is whether the selection will result in too much overhead in the server and second player's client. This is especially true if the move is rejected by player 1's game engine, which makes the overhead of the network communication and processing by player 2 superfluous. For this reason, it is useful to do some preliminary, fast testing of the selection before it is sent out over the network.

Another concern is whether the two game engines will stay in step with each other. For example, the processing of player 1's move by player 2's game engine may be much quicker than in player 1's client. Consequently, player 2 may send his turn to player 1 before player 1 has finished the processing of his own move. Player 1's client must be prepared to handle 'early' move messages. Of course, early moves may also arrive at player 2 from player 1.

This is one reason why it is useful to have a separate 'watcher' thread as part of the client. Another reason is to handle server messages, such as the announcement that the other player has disconnected, which may arrive at any time.

8. Communication Protocols

A good starting point for changing a two-person game into a networked application is to consider the communication protocols (e.g. the kinds of messages) required during the various stages of the game. It is useful to consider three stages: initialization, termination, and game play.

Initialization in a two-person game is a little problematic due to the need to have two participants before the game can start, and to restrict more than two players from joining.

The *termination* stage is entered when a player decides to stop participating in the game, which may be due to many reasons. When a player leaves the game, the other player must be notified.

Game play is usually the simplest to specify since it often only involves the transmission of a move from one player to another.

In the following diagrams, we usually only consider the cases when player 1 starts communication (e.g. when player 1 sends a new move to player 2). However, the communication patterns apply equally to the cases when player 2 initiates matters.

The initialization stage in NetFourByFour uses the protocol shown in Figure 8.

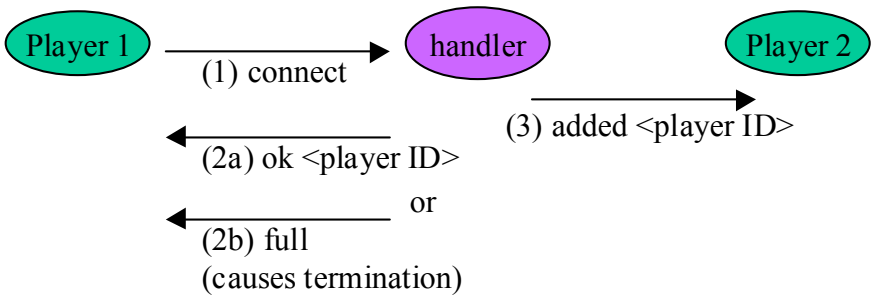

Figure 8. Initialization Stage Protocol in NetFourByFour.

The 'connect' message is implicit in the connection created when player 1's client opens a socket link to the server. The handler can send back an "ok" message containing an assigned ID, or reject the link with a "full" reply. If the connection was accepted then an "added" message is sent to the other player (if there is one).

The game will commence when the player ID in the "ok" and "added" messages is the number 2, meaning that there are now two players ready to compete.

The termination stage in NetFourByFour uses the protocol given in Figure 9.

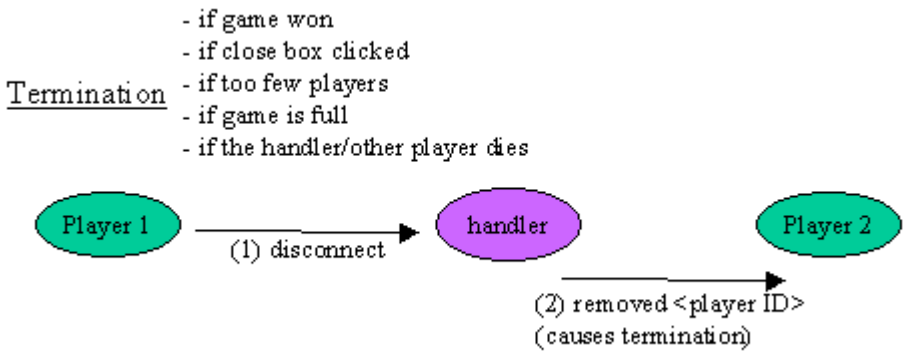

Figure 9. Termination Stage Protocol in NetFourByFour.

The five conditions that cause a player to leave the game are listed at the top of Figure 9. The player sends a "disconnect" message and breaks its link with the server. In NetFourByFour, this does not cause the player's client to exit (although that is a design possibility). The server sends a "removed" message to the other player (if one exists), which causes it to break its link, since there are now too few players.

The server will also send a "removed" message if its socket link to player 1 closes without a preceding "disconnect" warning. This behaviour is required to deal with network or machine failure.

If the handler dies then the players will detect it by noticing that their socket links have closed prematurely.

The game play stage is shown in Figure 10.

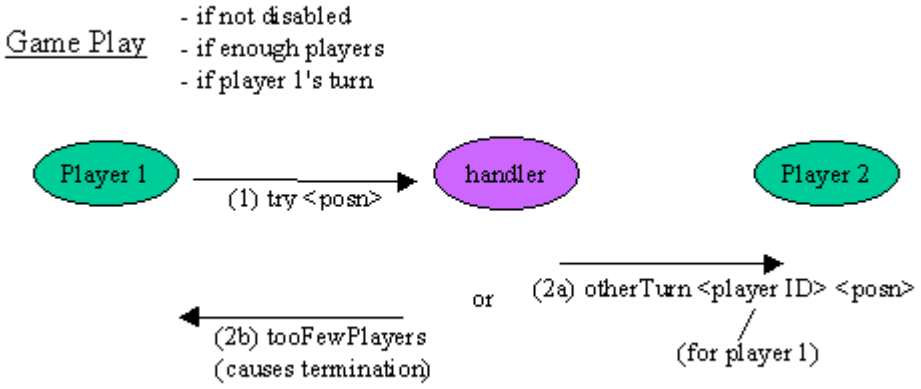

Figure 10. The Game Play Stage Protocol in NetFourByFour.

The three conditions necessary for game play to continue are shown at the top of Figure 10. The selected move is sent as a "try" message via the server, arriving as an "otherTurn" message. The "otherTurn" message may arrive at the player while it is still processing the previous move, for reasons described above. Also, if player 2 has

suddenly departed, perhaps due to a network failure, then the server may send a "tooFewPlayers" message back to player 1.

9. NetFourByFour Illustrated

Figure 11 shows two NetFourByFour players in fierce competition. It looks like player 1 is about to win.

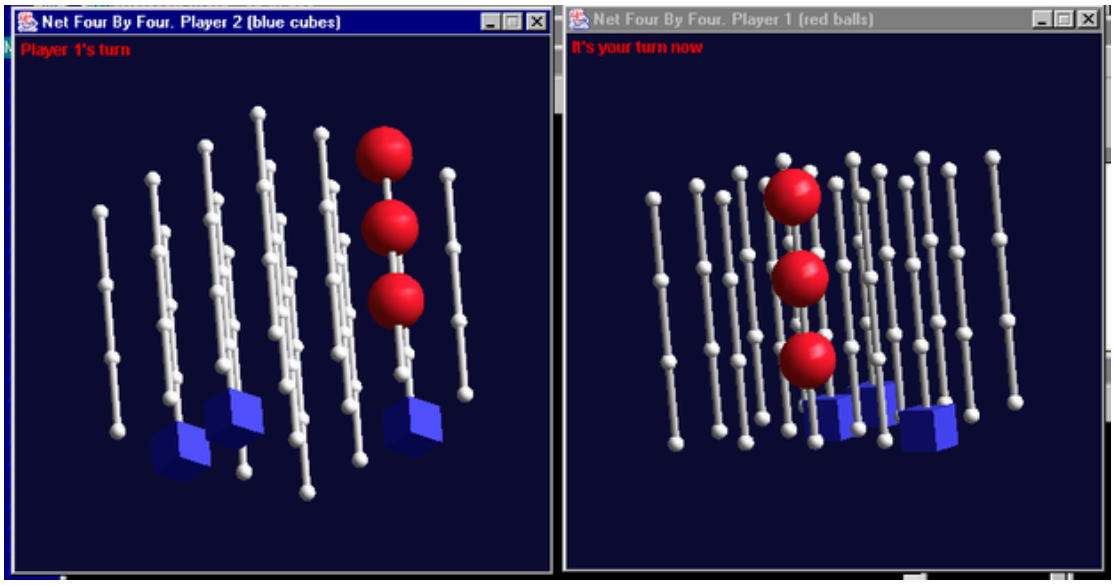

Figure 11. Two NetFourByFour Players.

The players have rotated their game boards in different ways, but the markers are in the same positions in both windows.

The GUI in NetFourByFour is changed from the FourByFour game: the messages text field has been replaced by a message string which appears as an overlay at the top-left corner of the Canvas3D window. This functionality is achieved by subclassing the Canvas3D class to implement the mixed-mode rendering method postSwap(), resulting in the OverlayCanvas class.

The class diagrams for the NetFourByFour client are given in Figure 12, while the server side classes appear in Figure 13. Only public methods are listed, and methods which are synchronized are prefixed with a 'S'.

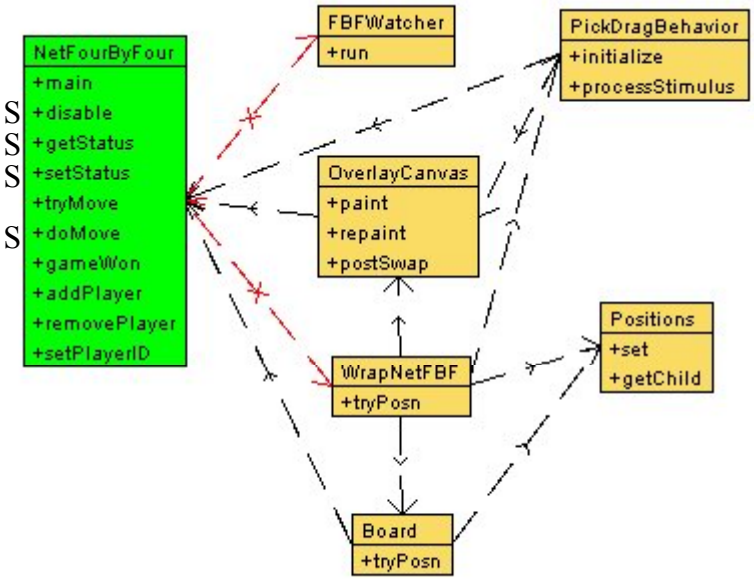

Figure 12. UML Class Diagrams for the NetFourByFour Client.

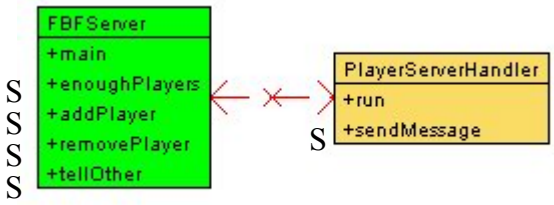

Figure 13. UML Class Diagrams for the NetFourByFour Server.

FBFServer is the top-level class on the server-side.

As with previous networked examples, the connections between the client and server are unclear since they are in terms of message passing rather than method calls.

10. The Relationship between NetFourByFour and FourByFour

Many of the classes in NetFourByFour are very similar to those in FourByFour; indeed, this was one of the reasons for keeping the game logic inside the client.

The Positions class, which manages the on-screen markers is unchanged from FourByFour.

PickDragBehavior still handles user picking and dragging, but reports a selected position to NetFourByFour rather than to Board.

The game data structures in Board are as before, but the tryMove() method for processing a move, and reportWinner() are a little different.

WrapNetFourByFour is virtually the same as WrapFourByFour but utilizes the OverlayCanvas class rather than Canvas3D.

NetFourByFour is considerable changed, since the networking code for the client-side is located there. FBFWatcher is used to monitor messages coming from the server, so is completely new.

11. The FBFServer Class

As indicated in Figure 6, the FBFServer class manages a small amount of shared data used by its two handlers: an array of PlayServerHandler references, and the current number of players.

```
private PlayerServerHandler[] handlers;     // handlers for players
private int numPlayers;
```

The references are to allow a message to be sent to a handler by calling its sendMessage() method:

```
synchronized public void tellOther(int playerID, String msg)
// send mesg to the other player
{ int otherID = ((playerID == PLAYER1) ? PLAYER2 : PLAYER1 );
  if (handlers[otherID-1] != null)     // index is ID-1
    handlers[otherID-1].sendMessage(msg);
}
```

tellOther() is called from the handler for one player to send a message to the other player.

The numPlayers global is modified as a side-effect of adding and removing a player, and is used to decide whether there are enough players to start a game:

```
synchronized public boolean enoughPlayers()
{ return (numPlayers == MAX_PLAYERS); }
```

12. The PlayerServerHandler Class

The PlayServerHandler thread for a client deals with the various 'handler' messages defined in the initialization, termination and game play stages shown in Figures 8, 9, and 10.

When the thread is first created, it is passed the socket link to the client, and I/O streams are layered on top of it.

```
// globals
private FBFServer server;
private Socket clientSock;
```



```
  private BufferedReader in;
  private PrintWriter out;

  private int playerID;     // player id assigned by FBFServer


  public PlayerServerHandler(Socket s, FBFServer serv)
  {
    clientSock = s;
    server = serv;
    System.out.println("Player connection request");
    try {
      in  = new BufferedReader(  new InputStreamReader(
                                   clientSock.getInputStream() ) );
      out = new PrintWriter( clientSock.getOutputStream(), true );
    }
    catch(Exception e)
    {  System.out.println(e);  }
  }
```

run() starts by carrying out the messages specified in the initialization stage of the network communication. It calls addPlayer() in the server to add the new player. This may fail if there are already two players, and a "full" message is sent back to the client. If the joining is successful then an "ok" message is sent to the new player, and an "added" message to the other player (if one exists).

```
  public void run()
  {
    playerID = server.addPlayer(this);
    if (playerID != -1) {    // -1 means player was rejected
      sendMessage("ok " + playerID); // tell player his/her ID
      server.tellOther(playerID, "added " + playerID);

      processPlayerInput();

      server.removePlayer(playerID);   // goodbye
      server.tellOther(playerID, "removed " +
                                    playerID); // tell others
    }
    else    // game is full
      sendMessage("full");

    try {     // close socket from player
      clientSock.close();
      System.out.println("Player "+playerID+" connection closed\n");
    }
    catch(Exception e)
    {  System.out.println(e);  }
  }
```

When processPlayer() returns it means that the player has broken the network link, and so the server must be updated, and the other player notified with a "removed" message. Thus, run() finishes by carrying out the termination stage.

processPlayer() monitors the input stream for its closure of a "disconnect" message. Otherwise, messages are sent on to doRequest(), which deals with the game play stage of the communication.


```
private void doRequest(String line)
{
  if (line.startsWith("try")) {
    try {
      int posn = Integer.parseInt( line.substring(4).trim() );

      if (server.enoughPlayers())
        server.tellOther(playerID, "otherTurn " + playerID +
                    " " + posn);  // pass turn to others
      else
        sendMessage("tooFewPlayers");
    }
    catch(NumberFormatException e)
    { System.out.println(e); }
  }
}
```

A "try" message is sent on to the other player as an "otherTurn" message.

sendMessage() simply writes a string onto the PrintWriter stream going to the player. However, the method must be synchronized since it is possible that the handler and top-level server may try to call it a the same time.

```
synchronized public void sendMessage(String msg)
{ try {
      out.println(msg);
  }
  catch(Exception e)
  {  System.out.println("Handler for player "+playerID+"\n"+e); }
}
```

13. Game Initialization in the Client

The network initialization done in NetFourByFour consists of opening a connection to the server, and creating a FBFWatcher thread to wait for a response.

```
// globals in NetFourByFour
private Socket sock;
private PrintWriter out;
    :

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

    new FBFWatcher(this, in).start();  // start watching server
  }
  catch(Exception e)
  {  // System.out.println(e);
     System.out.println("Cannot contact the NetFourByFour Server");
```



```
      System.exit(0);
    }
  }  // end of makeContact()
```

A consideration of Figure 8 shows that an "ok" or "full" message may be sent back. These responses, and the other possible client-directed messages, are caught by FBFWatcher in its run() method:

```
public class FBFWatcher extends Thread
{
  private NetFourByFour fbf;    // ref back to client
  private BufferedReader in;

  public FBFWatcher(NetFourByFour fbf, BufferedReader i)
  {  this.fbf = fbf;
     in = i;
  }

  public void run()
  { String line;
    try {
      while ((line = in.readLine()) != null) {
        if (line.startsWith("ok"))
          extractID(line.substring(3));
        else if (line.startsWith("full"))
          fbf.disable("full game");
        else if (line.startsWith("tooFewPlayers"))
          fbf.disable("other player has left");
        else if (line.startsWith("otherTurn"))
          extractOther(line.substring(10));
        else if (line.startsWith("added"))    // don't use ID
          fbf.addPlayer();        // client adds other player
        else if (line.startsWith("removed"))  // don't use ID
          fbf.removePlayer();     // client removes other player
        else   // anything else
          System.out.println("ERR: " + line + "\n");
      }
    }
    catch(Exception e)
    {. . .}
  }  // end of run()

  :
} // end of FBFWatcher class
```

The messages considered inside run() match the kinds of communication which a player may receive, as given in Figures 8, 9, and 10.

An "ok" message causes extractID() to extract the playerID, and call NetFourByFour's setPlayerID() method. This binds the playerID value used throughout the client's execution.

A "full" message triggers a call to NetFourByFour's disable() method. This is called from various places to initiate the client's departure from the game.

The handler for the player also sends an "added" message to the FBFWatcher of the other player, leading to a call of it's NetFourByFour addPlayer() method. This

increments the client's numPlayers counter, which permits game play to commence when equal to 2.

14. Game Termination in the Client

Figure 9 lists five ways in which game play may stop:

1. the player has won;
2. the close box was clicked;
3. there are too few players to continue (i.e. the other player has departed);
4. the game already has enough participants;
5. The handler or other player dies.

Each case is considered below.

14.1. The Player has Won

The Board object detects whether a game has been won in a same way as the FourByFour version, and then calls reportWinner()

```
private void reportWinner(int playerID)  // in Board
{
  long end_time = System.currentTimeMillis();
  long time = (end_time - startTime)/1000;

  int score = (NUM_SPOTS + 2 - nmoves) * 111 -
           (int) Math.min(time * 1000, 5000);

  fbf.gameWon(playerID, score);
}
```

There are two changes to reportWinner(): it is passed the player ID of the winner, and it calls gameWon() in NetFourByFour rather than write to a text field.

gameWon() checks the player ID against the client's own ID, and passes a suitable string to disable():

```
public void gameWon(int pid, int score)  // in NetFourByFour
{
  if (pid == playerID)   // this client has won
    disable("You've won with score " + score);
  else
    disable("Player " + pid + " has won with score " + score);
}
```

disable() is the core method for terminating game play for the client. It sends a "disconnect" method to the server (see Figure 9), sets a global boolean isDisabled to true, and updates the status string.

```
synchronized public void disable(String msg)  // in NetFourByFour
{ if (!isDisabled) {    // client can only be disabled once
    try {
```



```
        isDisabled = true;
        out.println("disconnect");  // tell server
        sock.close();
        setStatus("Game Over: " + msg);
        // System.out.println("Disabled: " + msg);
      }
      catch(Exception e)
      {  System.out.println( e );  }
    }
  }
```

disable() may be called by the close box, FBFWatcher, or Board (via gameWon()), and so must be synchronized. The isDisabled flag means that the client can only be disabled once. Disabling breaks the network connection, and makes further selections have no effect on the board. However, the application is still left running, and the player can rotate the game board.

14.2. The Close Box was Clicked

The constructor for NetFourByFour sets up a call to disable() and exit() in a window listener.

```
public NetFourByFour()
{        :
  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { disable("exiting");
      System.exit( 0 );
    }
  });
       :
} // end of NetFourByFour()
```

14.3. Too Few Players, and the Game is Full

If FBFWatcher receives a "tooFewPlayer" or a "full" message from its handler, it calls disable().

```
public void run()   // in FBFWatcher
{ String line;
  try {
    while ((line = in.readLine()) != null) {
            :
      else if (line.startsWith("full"))
        fbf.disable("full game");
      else if (line.startsWith("tooFewPlayers"))
        fbf.disable("other player has left");
            :
    }
  }
  catch(Exception e)
  {. . .}
}
```


14.4. The Handler or Other Player has Died

If the other player's client suddenly terminates, then its server-side handler will detect the closure of its socket, and sent a "removed" message to the other player.

```
public void run()    // in FBFWatcher
{ String line;
  try {
    while ((line = in.readLine()) != null) {
        :
      else if (line.startsWith("removed"))  // don't use ID
        fbf.removePlayer();     // client removes other player
        :
    }
  }
  catch(Exception e)    // socket closure will end while
  {  fbf.disable("server link lost");   // end game as well
  }
}
```

FBFWatcher will see the "removed" message and call removePlayer() in NetFourByFour. This will decrement its number of players counter which will prevent any further selected moves from being carried out.

If the server dies then FBFWatcher will raise an exception when it tries to read from the socket. This triggers a call to disable() which will end the game.

15. Game Play in the Client

Figure 7 presents an overview of typical game play using a UML activity diagram. Player 1 selects a move which is processed locally while also being set via the server to the other player to be processed.

A closer examination of this turn-taking operation is quite complex because it involves both the clients and the server. We will break it into three parts, corresponding to the *swimlanes* in the activity diagram. Each part will be expanded into its own UML sequence diagram, which allows more detail to be exposed.

15.1. Player 1's Client

The sequence diagram for the left hand of Figure 7 (player 1's client) is shown in Figure 14.

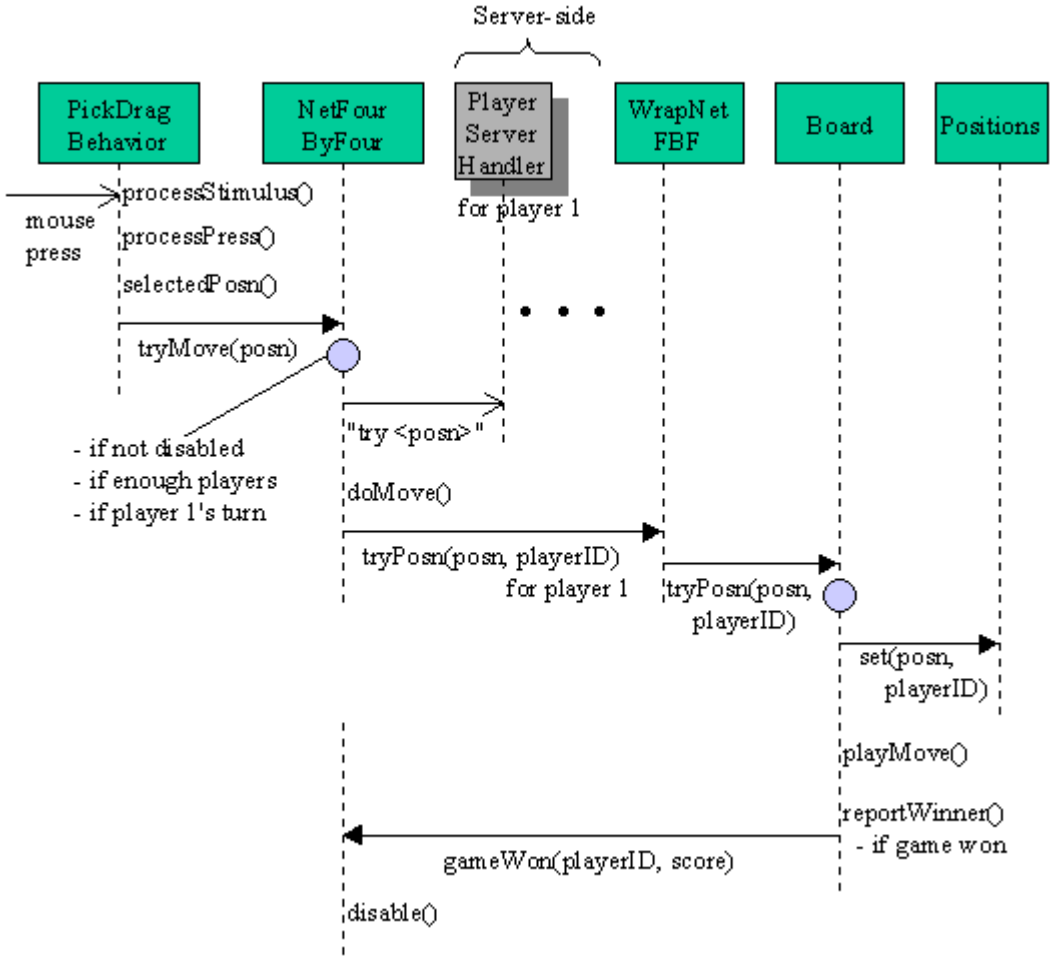

Figure 14. Sequence Diagram for Player 1's Client.

The mouse press is dealt with by PickDragBehavior in a similar way to in FourByFour, except that tryMove() is called in NetFourByFour, and passed the selected position index.

tryMove() carries out a few simple tests before sending a message off to its server and calling doMove() to execute the game logic.


```
public void tryMove(int posn)  // in NetFourByFour
{
  if (!isDisabled) {
    if (numPlayers < MAX_PLAYERS)
      setStatus("Waiting for player " + otherPlayer(playerID) );
    else if (playerID != currPlayer)
      setStatus("Sorry, it is Player " + currPlayer + "'s turn");
    else if (numPlayers == MAX_PLAYERS) {
      out.println( "try " + posn );   // tell the server
      doMove(posn, playerID);     // do it, don't wait for response
    }
    else
     System.out.println("Error on processing position");
  }
}  // end of tryMove()
```

tryPosn() in Board is simpler than the version in FourByFour.

```
public void tryPosn(int pos, int playerID)  // in Board
{ positions.set(pos, playerID);  // change 3D marker shown at pos
  playMove(pos, playerID);       // play the move on the board
}
```

The gameOver boolean is gone, since the isDisabled boolean has taken its place back in NetFourByFour. Also, tryPosn() no longer changes the player ID since a client is dedicated to a single player.

The playerID input argument of tryPosn() is a new requirement since this code may be called to process moves by either of the two players.

set() in Positions is unchanged from FourByFour, and playMove() utilizes the same game logic to update the game and test for a winning move. reportWinner() is a little altered, as explained when we considering the termination cases.

15.2. Server-Side Processing

The sequence diagram on the server side (Figure 15) shows how a "try" message from player 1 is passed on to player 2 as an "otherTurn" message.

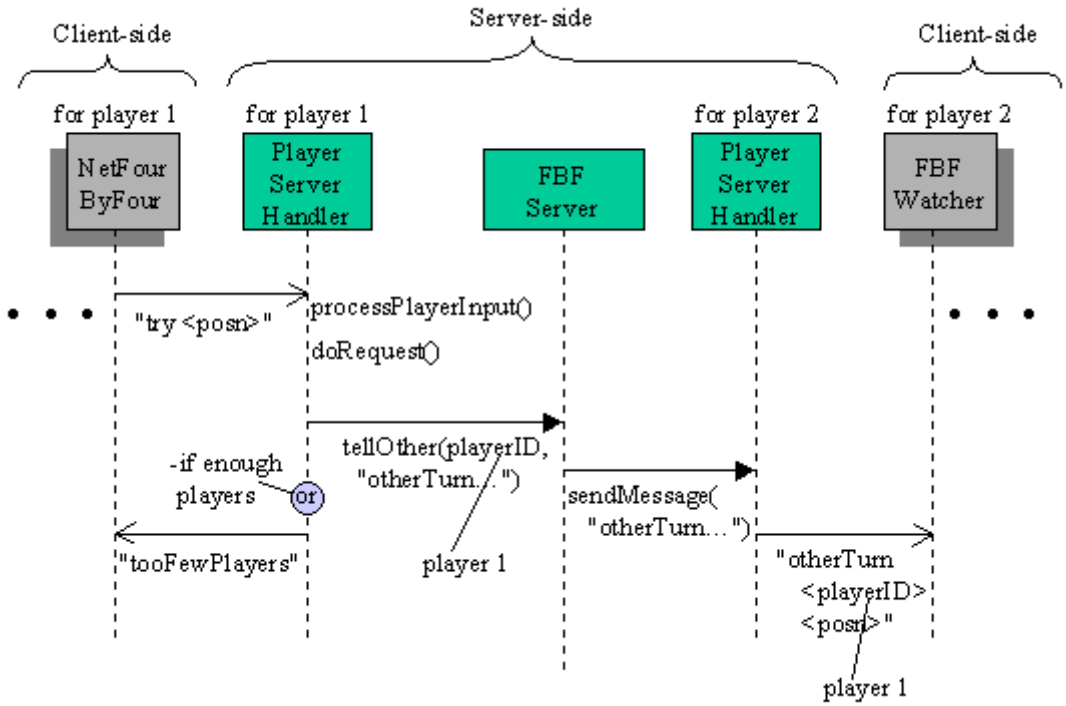

Figure 15. Sequence Diagram for the Server.

Much of the coding behind these diagrams has already been considered when we looked at the server-side classes earlier.

The diagram shows that the "otherTurn" message is received by the FBFWatcher of player 2.

```
public void run()   // in FBFWatcher
{ String line;
  try {
    while ((line = in.readLine()) != null) {
            :
      else if (line.startsWith("otherTurn"))
        extractOther(line.substring(10));
            :
    }
  }
  catch(Exception e)
  { }
}
```


15.3. Player 2's Client

The sequence diagram for the right hand side of Figure 7 (player 2's client) is shown in Figure 16.

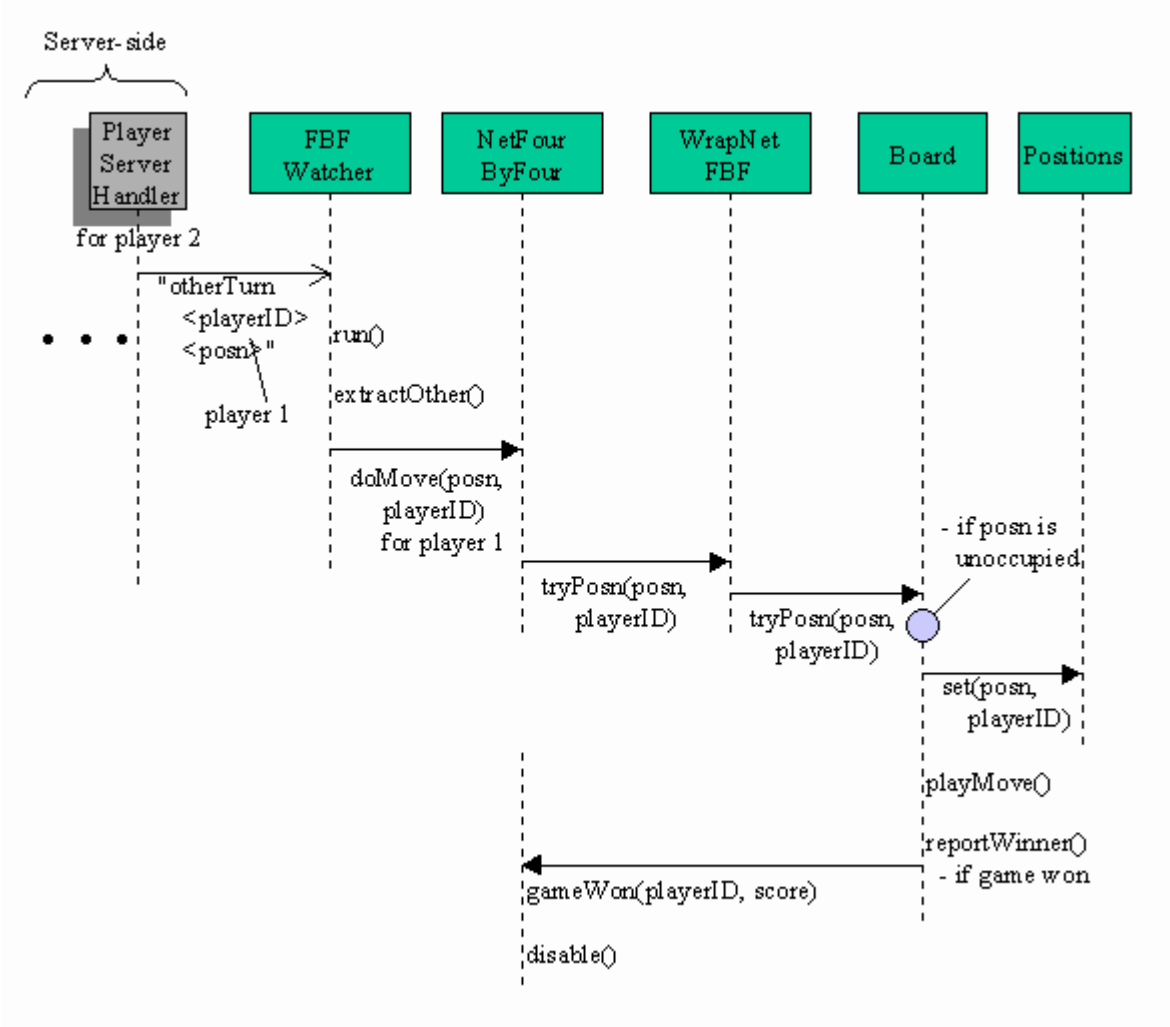

Figure 16. Sequence Diagram for Player 2's Client.

The call to extractOther in FBFWatcher extracts the player's ID and the position index from the "otherTurn" message. It then calls doMove() in NetFourByFour.

```
synchronized public void doMove(int posn,int pid) //in NetFourByFour
 {
   wrapFBF.tryPosn(posn, pid);   // and so to Board
   if (!isDisabled) {
     currPlayer = otherPlayer( currPlayer );  // player's turn over
     if (currPlayer == playerID)  // this player's turn now
       setStatus("It's your turn now");
     else    // the other player's turn
       setStatus("Player " + currPlayer + "'s turn");
   }
 }
```

doMove() and the methods it calls (e.g. tryPosn() in WrapNetFBF and Board) are used by the client to execute its moves, *and* to execute the moves of the other player.

This is why the methods all take the player ID as an argument – so the 'owner' of the move is clear.

As mentioned before, it is possible for the client to still be processing its move when a request to process the opponent's move comes in. This situation is handled by the use of the synchronized keyword with doMove(): a new call to doMove() must wait until the current call has finished.

16. The OverlayCanvas Class

OverlayCanvas is a subclass of Canvas3D which draws a status string onto the canvas in its top-left hand corner (see Figure 11). The string reports on the current state of play. It is implemented as an *overlay*, meaning that the string is not part of the 3D scene, instead it is resting on 'top' of the canvas.

This technique utilizes Java 3D's *mixed mode rendering*, which gives the programmer access to the rendering loop at different stages in its execution.

The client makes regular calls to setStatus() in NetFourByFour to update a global status string:

```
synchronized public void setStatus(String msg) //in NetFourByFour
{  status = msg;  }
```

This string is periodically accessed from the OverlayCanvas object by calling getStatus():

```
synchronized public String getStatus()  // in NetFourByFour
{  return status; }
```

The get and set methods are synchronized since the calls from OverlayCanvas may come at any time, but should not access the string while it is being updated.

An OverlayCanvas object is created in the constructor of WrapNetFBF, in the usual way that a Canvas3D object is created.

```
public WrapNetFBF(NetFourByFour fbf)
{
     :
  GraphicsConfiguration config =
          SimpleUniverse.getPreferredConfiguration();

  OverlayCanvas canvas3D = new OverlayCanvas(config, fbf);
  add("Center", canvas3D);
  canvas3D.setFocusable(true);     // give focus to the canvas
  canvas3D.requestFocus();
     :
}
```


The only visible difference is the passing of a reference to the NetFourByFour object into the OverlayCanvas constructor (the fbf variable). This is utilized by OverlayCanvas to call the getStatus() method in NetFourByFour.

16.1. Mixed Mode Rendering

Canvas3D provides four methods for accessing Java 3D's rendering loop: preRender (), postRender(), postSwap(), and renderField(). By default, these methods have empty implementations, and are called automatically at various stages in each cycle of the rendering loop. They are utilized by the programmer subclassing Canvas3D, and providing implementations for the required methods.

The four methods in more detail:

- preRender(). It is called after the canvas has been cleared and *before any rendering* is carried out, at the very start of the current rendering cycle.
- postRender(). It is called after all the rendering is completing, but *before the buffer swap*. This means that the current rendering has not yet been placed on screen.
- postSwap(). It is called after the current rendering is on-screen (i.e. after the buffer containing it has been swapped out to the frame). This occurs at the very end of the current rendering cycle.
- renderField() is most useful in stereo rendering: it is called after the left eye's visible objects are rendered, and again after the right eye's visible objects are rendered.

16.2. postSwap() in OverlayCanvas

postSwap() is used to draw the updated status string on screen:

```
// globals
private final static int XPOS = 5;
private final static int YPOS = 15;
private final static Font MSGFONT =
         new Font( "SansSerif", Font.BOLD, 12);

private NetFourByFour fbf;
private String status;
    :


public void postSwap()
{
  Graphics2D g = (Graphics2D) getGraphics();
  g.setColor(Color.red);
  g.setFont( MSGFONT );

  if ((status = fbf.getStatus()) != null)  // it has a value
    g.drawString(status, XPOS, YPOS);

  // this call is made to compensate for the javaw repaint bug,
  Toolkit.getDefaultToolkit().sync();
```



```
  }  // end of postSwap()
```

The call to getStatus() in NetFourByFour may return null at the very start of the client's execution if the canvas is rendered before status gets a value.

The repaint() and paint() methods are overridden:

```
public void repaint()
// Overriding repaint() makes the worst flickering disappear
{ Graphics2D g = (Graphics2D) getGraphics();
  paint(g);
}

public void paint(Graphics g)
// paint() is overridden to compensate for the javaw repaint bug
{ super.paint(g);
  Toolkit.getDefaultToolkit().sync();
}
```

repaint() is overridden to stop the canvas being cleared before being repainted, which would cause a nasty flicker.

The calls to sync() in postSwap() and paint() are bug fixes to avoid painting problems when using javaw to execute Java 3D mixed-mode applications.

