Chapter 6.4. An Isometric Tile Game

AlienTiles is a basic *isometric tile* game consisting of one player who must pick up a cup, flower pot, and a watch before four nasty aliens catch and hit him three times (see Figure 1).

Figure 1. AlienTiles in Action.

The player is represented by a little blue man with a red cap in the center of the screen. Three of the four orange aliens are visible in Figure 1, although one of them is mostly hidden behind the black and white column just to the right of the player.

1. Background

Isometric tiles are the basis of many real-time strategy (RTS) games, war games, and simulations (e.g. *Civilization II*, *Age of Empires*, and *SimCity* variants), although the tiling of the game surface is usually hidden.

Isometric tiles give an artificial sense of depth, as if the player's viewpoint is somewhere up in the sky, looking down over the playing area. It's artificial since no perspective effects are applied: the tiles in the row 'nearest' the viewer are the same size and shape as the tiles in the most 'distant' row at the top of the screen. This is where the term 'isometric' comes from: an *isometric projection* is a 3D projection that doesn't correct for distance.

The illusion that each row of tiles is further back inside the game is supported by the z-ordering of things (sprites, objects) drawn in the rows. An object on a row nearer the front is drawn after those on rows further back, hiding anything behind it. This is the case in Figure 1 with the black and white column hiding the alien standing two rows behind it.

There are various ways of labeling the x- and y- axes of a isometric tile map. We use the fairly standard *staggered map* approach illustrated in Figure 2.

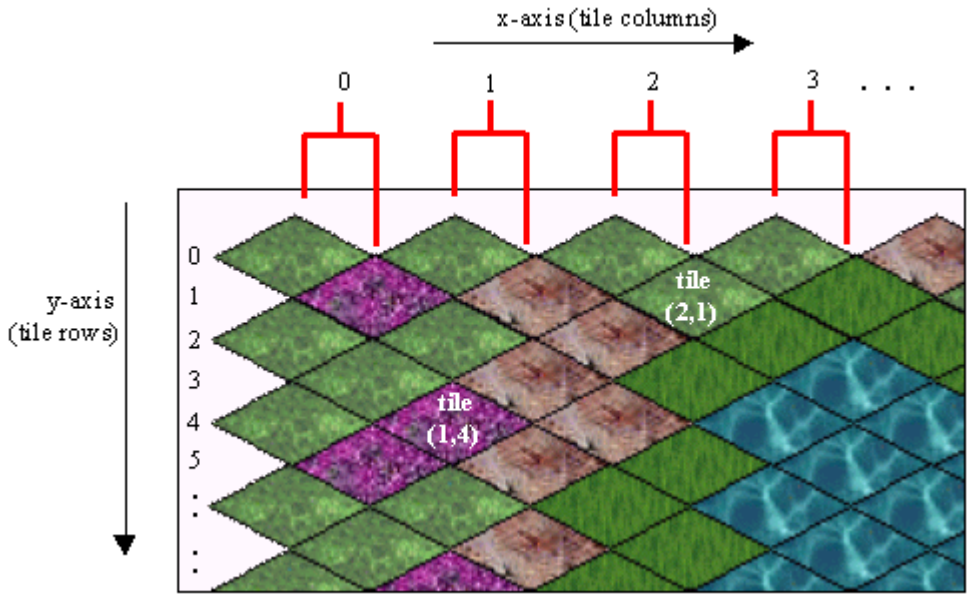

Figure 2. A Staggered Isometric Tile Map.

Odd and even rows are offset from each other, which means that the tile coordinates can be a little tricky to work out as a sprite moves between rows.

AlienTiles uses tile coordinates to position sprites and other objects on the surface. However, the surface itself isn't made from tiles, instead it's a single medium size GIF (216K), as shown in Figure 3.

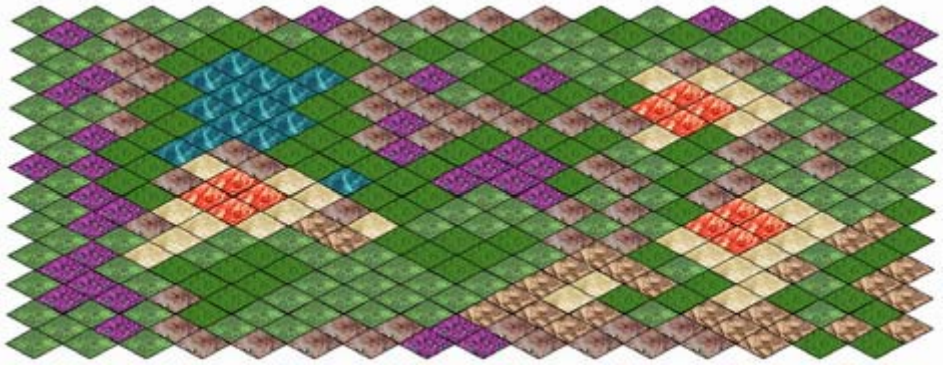

Figure 3. The surface.gif Image.

Most isometric games construct the surface from individual tiles, which allows the floor space to be incrementally rendered, and to dynamically change over time. The drawback is the increased complexity (and time) in drawing the tiles to the screen. It is necessary to draw them in back-to-front row order, with each diamond represented by a rectangular GIF with transparent corners.

Often the surface will be a composite of several layers of tile of different sizes. For example, there may be several large green tiles for the terrain, layered over with smaller grass, dirt, and sand tiles to create variety. *Fringe* tiles are employed to break up the regularity of the edges between two large areas, such as the land and the sea.

1.1. Movement

AlienTiles offers four directions for a sprite to follow: north east, south east, south west, and north west, as in Figure 4.

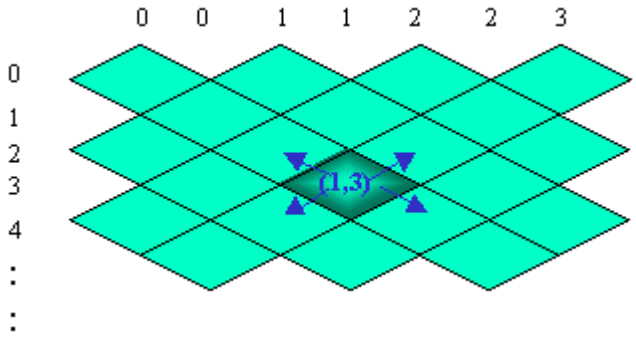

Figure 4. Directions of Movement for a Sprite.

The user interface maps these directions to the four corners of the numbers keypad: to the keys '9', '3', '1', and '7'. It's also possible to not move at all by pressing '5'. An obvious extension is to also offer north, east, south, and west movement.

The range of directions is dictated by the tile shape to a large extent, and diamonds aren't the only possibility. For instance, a number of strategy games use hexagons to form a *Hex* map (Figure 5), which allows six compass directions out of a tile.

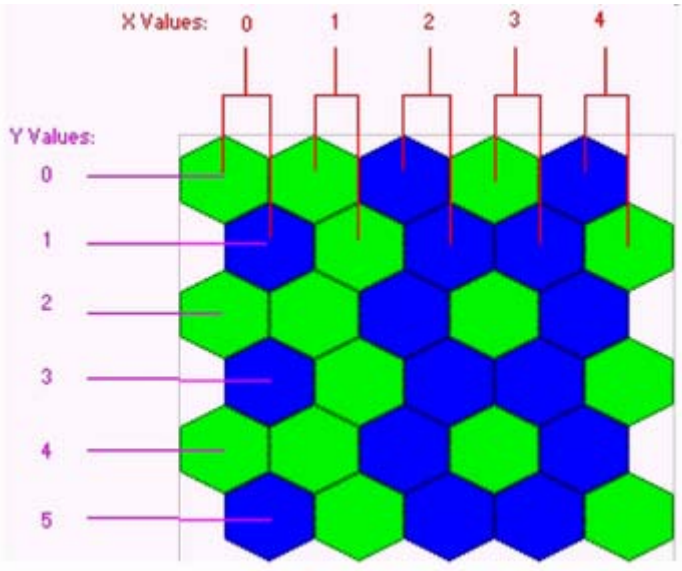

Figure 5. A Hex Map.

Movement around an isometric tile surface is often based on single steps between tiles. It's not possible for a sprite to move about inside a tile, only stand on a tile or make a single step to an adjacent tile. In AlienTiles, a key press causes a single step; the user must hold down the key to make the sprite sprint across several tiles.

Although we talk about a player moving around the surface, the truth is that the sprite doesn't move at all. Instead the surface moves in the opposite direction, together with the other objects and sprites. For instance, when the player moves to the north east, the sprite stays still but the ground underneath it shifts to the south west.

This non-movement is only true for the player sprite, the alien sprites do move from one tile to another.

1.2. Placing a Sprite/Object

Care must be taken with object placement so the illusion that it's standing on the tile is maintained. Figure 6 shows that the positioning of a sprite's top-left corner so that its 'feet' are planted on the tile's surface can be a little tricky:

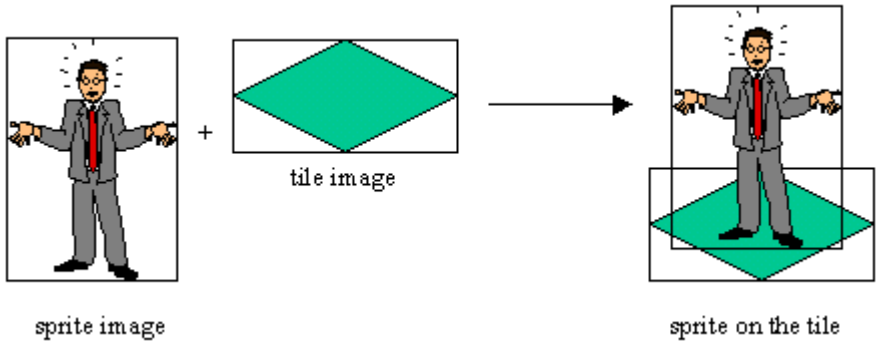

Figure 6. Placing a Sprite onto a Tile.

The sprite can occupy screen space above the tile, but should not overlap the bottom left and right edges of the diamond. If it does then the image will seem to be partly in the next row, weakening the row ordering effect.

1.3. The Tile Map Surface

The AlienTiles surface contains no-go areas which the sprites cannot enter. These include the 'ocean' around the edges of the tiled surface, a lake, a pond, and four red squares (all visible in Figure 1). The no-go areas are defined in a configuration file read in by AlienTiles at start-up.

The game surface has two kinds of objects resting on it: *blocks* and *pickups*. A block fully occupies a tile, preventing a sprite from moving onto it. The block image can be anything – we employ various columns and geometric shapes. A player can remove a pickup from the surface when it's standing on the same tile; the user presses '2' on the numbers keypad.

More sophisticated games have a much greater variety of surface objects. Two common types are *walls* and *portals* (doors). A wall between two tiles prevents a sprite from moving between the tiles. A portal is often used as a way of moving between tile maps, for example when moving to the next game level or entering a building with its own floor plan.

1.4. The Aliens

AlienTiles offers two types of aliens: those that actively chase after the player (AlienAStarSprite objects) and those that congregate around the pickup that the player is heading towards (AlienQuadSprite objects). The AlienAStarSprite class uses A* (pronounced A *star*) pathfinding to chase the player, which will be explained later.

In general, alien design opens the door to 'intelligent' behaviour code, often based on Artificial Intelligence (AI) techniques. Surprisingly though, quite believable sprite behaviour can often be hacked together with the use of a few random numbers and conventional loops and branches: AlienQuadSprite illustrates that point.

2. UML Diagrams for AlienTiles

Figure 7 shows a simplified set of UML class diagrams for AlienTiles. The audio and image classes (e.g. MidisLoader, ClipsLoader, and ImagesLoader) have been edited away, and the less important links between the remaining classes have been pruned back.

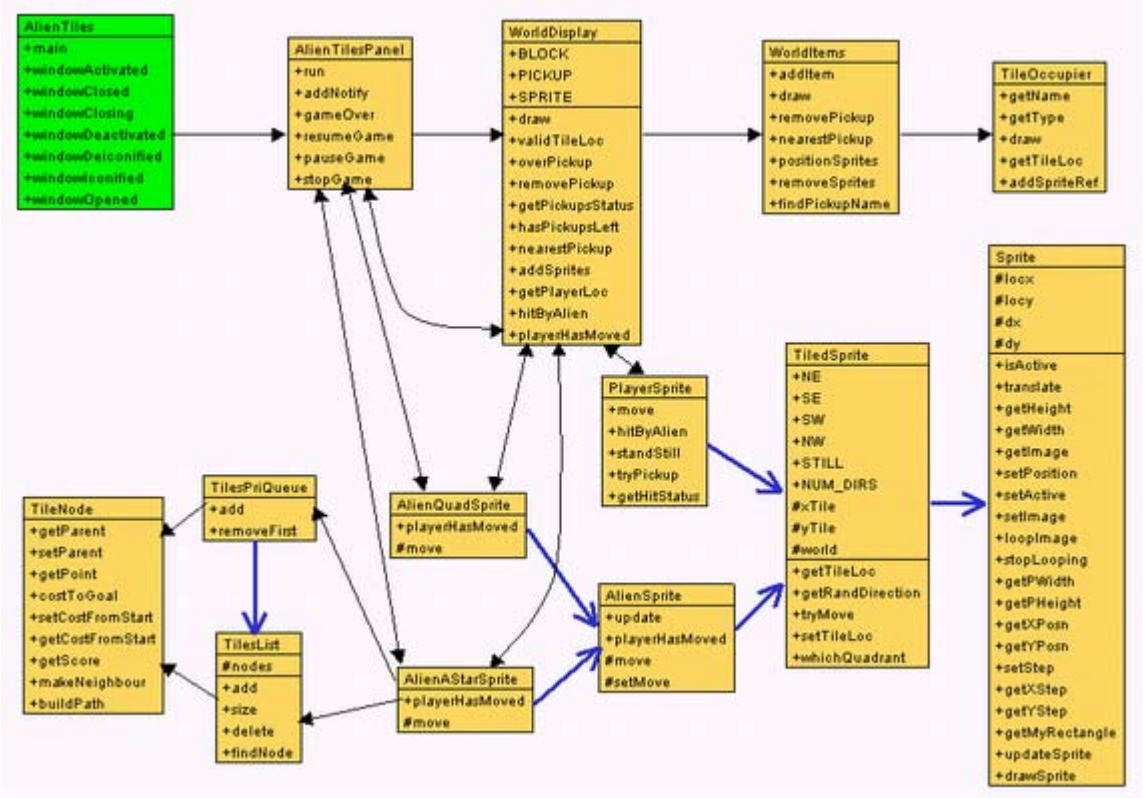

Figure 7. AlienTiles Classes Diagram (Simplified).

The AlienTiles JFrame and the AlienTilesPanel JPanel implement the windowed animation framework introduced in chapters 1 and 2; BugRunner of chapter 6 and JumpingJack of chapter 6.2 use the same technique.

Pausing, resuming. and quitting are controlled via AlienTiles' window listener methods.

The frame rate is set to 40 FPS, which is still too fast for the alien sprites; they are slowed down further by code in AlienQuadSprite and AlienAStarSprite.

WorldDisplay displays the surface image and the blocks, pickups, and sprites resting on the surface. The tile coordinates for the entities are stored in a WorldItems object, using a TileOccupier object for each one. WorldDisplay also acts as a communication layer between the player and the aliens.

Figure 7 includes a small sprite inheritance hierarchy, rooted at Sprite, which is shown on its own in Figure 8.

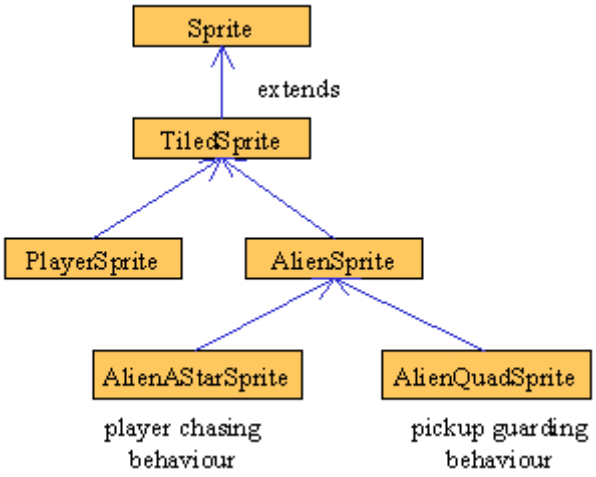

Figure 8. The Sprite Hierarchy in AlienTiles.

The methods in Sprite are barely used, except when the sprite needs to be drawn at a given pixel location on screen. Tile coordinates are utilized most of the time, supported by methods in TiledSprite. The player is represented by a PlayerSprite object.

AlienAStarSprite uses the A* pathfinding algorithm, which necessitates the TilesPriQueue and TilesList data structure classes; they maintain sequences of TileNode objects.

3. The AlienTilesPanel Class

AlienTilesPanel is similar to JackPanel in chapter 6.2: it uses an active rendering animation loop driven by Java 3D's timer. It displays a simple introductory image when the game starts, which doubles as a help screen during the course of play. While the help is being shown, the game pauses.

3.1. Managing the Game World

AlienTilesPanel creates the various game elements in createWorld():

```
// globals game entities
private WorldDisplay world;
private PlayerSprite player;
private AlienSprite aliens[];
         :


private void createWorld(ImagesLoader imsLoader)
// create the world display, the player, and aliens
{
  world = new WorldDisplay(imsLoader, this);

  player = new PlayerSprite(7,12, PWIDTH, PHEIGHT,
                 clipsLoader, imsLoader, world, this);
                 // sprite starts on tile (7,12)

  aliens = new AlienSprite[4];
  aliens[0] = new AlienAStarSprite(10, 11, PWIDTH, PHEIGHT,
                                          imsLoader, world);
  aliens[1] = new AlienQuadSprite(6, 21, PWIDTH, PHEIGHT,
                                          imsLoader, world);
  aliens[2] = new AlienQuadSprite(14, 20, PWIDTH, PHEIGHT,
                                          imsLoader, world);
  aliens[3] = new AlienAStarSprite(34, 34, PWIDTH, PHEIGHT,
                                          imsLoader, world);
   // use 2 AStar and 2 quad alien sprites
   // the 4th alien is placed at an illegal tile location (34,34)

  world.addSprites(player, aliens);
                  // tell the world about the sprites
}  // end of createWorld()
```

Tile coordinates are passed to the sprites rather than pixel locations in the JPanel.

The two AStar and two quad sprites are stored in an aliens[] array to make it easier to send messages to all of them as a group.

The player and aliens do not communicate directly, instead they call methods in the WorldDisplay object, world, which passes the messages on. This requires that sprite references be passed to world via a call to addSprites().

3.2. Dealing with Input

The game is controlled from the keyboard only, no mouse events are caught. As in previous applications, the key presses are handled by processKey(), which deals with termination keys (e.g. ctrl-C), toggling the help screen, and player controls. The code related to the player keys is shown in the fragment below:

```
private void processKey(KeyEvent e)
// handles termination, help, and game-play keys
{
  int keyCode = e.getKeyCode();
       :
```



```
    // game-play keys
    if (!isPaused && !gameOver) {
      // move the player based on the numpad key pressed
      if (keyCode == KeyEvent.VK_NUMPAD7)
        player.move(TiledSprite.NW);   // move north west
      else if (keyCode == KeyEvent.VK_NUMPAD9)
        player.move(TiledSprite.NE);   // north east
      else if (keyCode == KeyEvent.VK_NUMPAD3)
        player.move(TiledSprite.SE);   // south east
      else if (keyCode == KeyEvent.VK_NUMPAD1)
        player.move(TiledSprite.SW);   // south west
      else if (keyCode == KeyEvent.VK_NUMPAD5)
        player.standStill();           // stand still
      else if (keyCode == KeyEvent.VK_NUMPAD2)
        player.tryPickup();      // try to pick up from this tile
    }
  }  // end of processKey()
```

Three PlayerSprite methods are called: move(), standStill(), and tryPickup().

3.3. The Animation Loop

The animation loop is located in run(), and unchanged from earlier examples. In essence, it is:

```
public void run()
{ ...
  while (running) {
    gameUpdate();
    gameRender();
    paintScreen();
    // timing correction code
  }
  System.exit(0);
}
```

gameUpdate() updates the changing game entities: the four mobile aliens.

```
private void gameUpdate()
{ if (!isPaused && !gameOver) {
    for(int i=0; i < aliens.length; i++)
      aliens[i].update();
  }
}
```

gameRender() relies on the WorldDisplay object to draw the surface and its contents:

```
private void gameRender()
{
        :
  // a light blue background
  dbg.setColor(lightBlue);
  dbg.fillRect(0, 0, PWIDTH, PHEIGHT);

  // draw the game elements: order is important
  world.draw(dbg);
```



```
    /* WorldDisplay draws the game world: the tile floor, blocks,
       pickups, and the sprites. */

    reportStats(dbg);
      // report time spent playing, number of hits, pickups left

          :
}  // end of gameRender()
```

3.4. Ending the Game

The game finishes (gameOver is set to true) either when the player has been hit enough times, or when all the pickups (a cup, flower pot, and watch) have been gathered. The first condition is detected by the PlayerSprite object, the second by the WorldDisplay object; both of them call gameOver() to inform AlienTilesPanel.

```
public void gameOver()
{  if (!gameOver) {
     gameOver = true;
     score = (int) ((J3DTimer.getValue() -
                              gameStartTime)/1000000000L);
     clipsLoader.play("applause", false);
   }
}
```

4. The WorldDisplay Class

WorldDisplay manages:

- the moving tile floor, represented by a single GIF;
- no-go areas on the floor;
- blocks occupying certain tiles;
- pickups occupying certain tiles;
- communication between the player and aliens sprites.

The communication layer permits WorldDisplay to monitor and control the interactions between the sprites.

WorldDisplay utilizes three main data structures:

- an obstacles[][] boolean array specifying which tiles are no-gos or contain blocks;
- a WorldItems objects which stores details on blocks, pickups, and sprites in tile row order to make them easier to draw with the correct z-ordering;
- a numPickups counter to record how many pickups are still left to be picked up.

```
private boolean obstacles[][];
private WorldItems wItems;
private int numPickups;
```

WorldDisplay's methods fall into five main groups:

- the loading of floor information, which describes where the tiles rows and columns are located on the floor;
- the loading of world entity information, which gives the tile coordinates of the no-gos areas, blocks, and pickups;
- pickup-related methods;
- sprite-related methods;
- others, e.g. draw().

4.1. Loading Floor Information

The floor image is a single GIF, so additional information must state where the odd and even tile rows are located, and give the dimensions for a tile (a diamond). These details are shown in Figure 9.

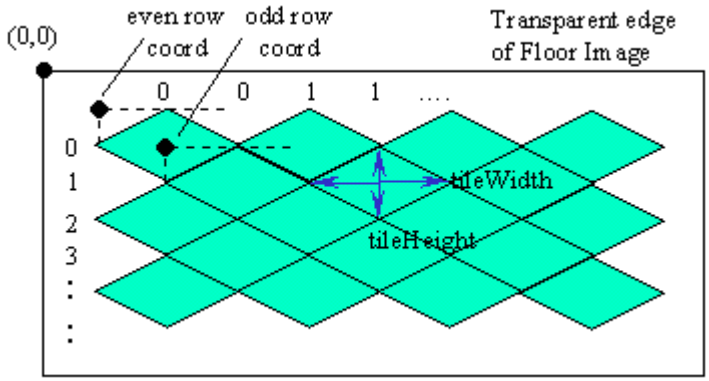

Figure 9. Floor Information.

The relevant information is stored in worldInfo.txt in the World/ subdirectory, and read in by loadWorldInfo(). The file contains:

```
// name of the GIF (surface.gif) holding the floor image
image surface

// number of tiles (x,y)
numTiles  16 23

// pixel dimensions of a single tile (width, height)
dimTile 56 29

// 'start of first even row' (x,y) coordinate
evenRow 12 8

// 'start of first odd row' (x,y) coordinate
oddRow 40 23
```

Lines beginning with "//" are comments.

The image used is surface.gif, which should be in the Images/ subdirectory below the AlienTiles directory.

There are 16 columns of tiles, and 23 rows. Each tile is 56 pixels wide at its widest point, and 29 pixels high. The first even row (row 0) starts at pixel coordinate (12,8), the first odd row (row 1) at (40,23). The starting point is taken to be the top-left corner of the rectangle that surrounds the diamond.

With this information it is possible to translate any tile coordinate into a pixel location in the floor image.

The data read in by loadFloorInfo() and its secondary methods is stored in a series of globals in WorldDisplay:

```
// world size in number of tiles
private int numXTiles, numYTiles;

// max pixel width/height of a tile
private int tileWidth, tileHeight;

// 'start of first even row' coordinate
private int evenRowX, evenRowY;

// 'start of first odd row' coordinate
private int oddRowX, oddRowY;
```

Most of them are used only to initialize the WorldItems object:

```
WorldItems wItems = new WorldItems(tileWidth, tileHeight,
                          evenRowX, evenRowY, oddRowX, oddRowY);
```

The WorldItems object organizes details about the surface entities (blocks, pickups, and sprites) by tile row so they are drawn to the JPanel with the correct z-ordering. This requires the floor information so that an entity's tile coordinates can be translated to pixel locations.

The number of tiles on the surface is used to initialize the obstacles[][] array:

```
private void initObstacles()
// initially there are no obstacles in the world
{
  obstacles = new boolean[numXTiles][numYTiles];
  for(int i=0; i < numXTiles; i++)
    for(int j=0; j < numYTiles; j++)
      obstacles[i][j] = false;
}
```

Obstacles are registered (i.e. particular cells are set to true) as WorldDisplay loads entity information (see below).

Sprites utilizes validTileLoc() to check if a particular tile (x,y) can be entered.

```
public boolean validTileLoc(int x, int y)
// Is tile coord (x,y) on the tile map and not contain an obstacle?
```



```
{
  if ((x < 0) || (x >= numXTiles) || (y < 0) || (y >= numYTiles))
    return false;
  if (obstacles[x][y])
    return false;
  return true;
}
```

4.2. Loading World Entity Information

Rather than hardwire entity positions into the code, the information is read in by loadWorldObjects() from the file worldObjs.txt in the subdirectory World/.

The data come in three flavours: no-go areas, blocks, and pickups, placed at a given tile coordinate, and unable to move. Sprites aren't included since their position can change during game play.

Consequently, worldObjs.txt supports three data formats:

```
// no-go coordinates
n <x1>-<y1> <x2>-<y2> .....
  .... #

// block coordinates for blockName
b <blockName>
  <x1>-<y1> <x2>-<y2> .....
  .... #

// pickup coordinate for pickupName
p <pickupName> <x>-<y>
```

A 'n' is for no-go, followed by multiple lines of (x,y) coordinates defining which tiles are inaccessible. The sequence of coordinates is terminated with a '#'.

A 'b' line starts with a block name, which corresponds to the name of the GIF file for the block, then a sequence of tile coordinates where the block appears.

The name on a 'p' line is also mapped to a GIF file name, but is followed only by a single coordinate. A pickup is assumed to only appear once on the floor.

The GIFs should be in the subdirectory Images/ below the AlienTiles directory.

A fragment of worldObjs.txt:

```
// bottom right danger zone (red in the GIF)
n 12-13 12-14 13-14 12-15 #

// blocks
b column1
9-3 7-7 7-18 #

b pyramid
1-12 5-16 #

b statue
14-13 #
```

```
// pickups
p cup 1-8
```

A quick examination of the Images/ subdirectory will show the presence of column1.gif, pyramid.gif, statue.gif, and cup.gif.

As the information is parsed by loadWorldObjects() and its helper methods, the obstacles[][] array and the worldItems object are passed the entity details. For instance, in getsBlocksLine() the following code fragment is executed when a (x,y) coordinate for a block has been found:

```
wItems.addItem( blockName+blocksCounter, BLOCK,
                                    coord.x, coord.y, im);
obstacles[coord.x][coord.y] = true;
```

addItem() adds information about the block to the WorldItems object. The relevant obstacles[][] cell is also set to true.

Similar code is executed for a pickup in getPickup():

```
wItems.addItem( pickupName, PICKUP, coord.x, coord.y, pickupIm);
numPickups++;
```

The obstacles[][] array is not modified since a sprite can move to a tile occupied by a pickup (it must do so before the item can be picked up).

BLOCK, PICKUP, and SPRITE are constants used by WorldItems to distinguish between tile entities.

4.3. Pickup Methods

WorldDisplay offers a range of pickup-related methods used by the sprites. For example, the PlayerSprite object calls removePickup() to pick up a named item.

```
public void removePickup(String name)
{ if (wItems.removePickup(name)) {  // try to remove it
    numPickups--;
    if (numPickups == 0)   // player has picked up everything
      atPanel.gameOver();
  }
  else
    System.out.println("Cannot delete unknown pickup: " + name);
}
```

WorldDisplay communicates with its WorldItems object to attempt the removal, and decrements its numPickups counter. If the counter reaches 0, then the player has collected all the pickups, and AlienTilesPanel (atPanel) can be told that the game is over.

4.4. Player Methods

The player sprite and the aliens don't communicate directly, instead their interaction is mediated by WorldDisplay. One of the more complicated player methods is playerHasMoved(), called by the PlayerSprite object when it moves to a new tile.

```
public void playerHasMoved(Point newPt, int moveQuad)
{
  for(int i=0; i < aliens.length; i++)
    aliens[i].playerHasMoved(newPt);   // tell the aliens
  updateOffsets(moveQuad);   // update world's offset
}
```

The player passes in a Point object holding its new tile coordinate, and the quadrant direction which brought the sprite to the tile. The moveQuad value can be the constant NE, SE, SW, NW, or STILL, which correspond to the four possible compass directions that a sprite can use (plus no movement).

The new tile location is passed to the aliens, which can use it to modify their intended destination. The quadrant direction is passed to updateOffsets() to change the surface image's offset from the enclosing JPanel.

As mentioned earlier, the player sprite doesn't move at all. A careful examination of AlienTiles during execution shows that the sprite always stays at the center of the game's JPanel. The floor image, and its contents (blocks, pickups, aliens), move instead. For instance, when the player sprite is instructed to move north west (the quadrant direction NW), the sprite does nothing but the floor and its contents shifts to the south east.

The floor offset is maintained in two globals:

```
private int xOffset = 0;
private int yOffset = 0;
```


xOffset and yOffset hold the pixel offsets for drawing the top-left corner of the floor image (and its contents) relative to the top-left corner (0,0) of the JPanel, as shown in Figure 10. The offsets may have negative values.

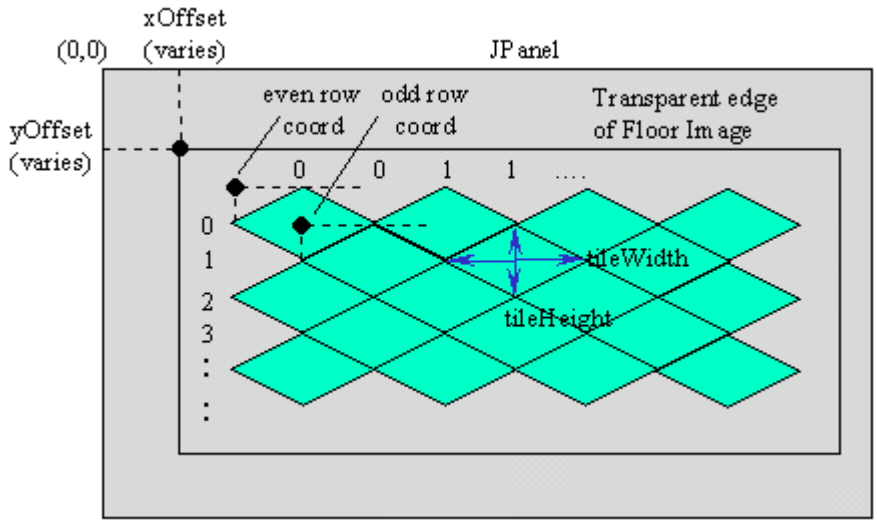

Figure 10. The Floor Offset from the JPanel.

The offsets are the final part of the mapping required to translate a tile coordinate into an on-screen pixel location.

This approach means that a stationary block or pickup, always positioned on the same tile, will be drawn at different places inside the JPanel as the xOffset and yOffset values change.

The offsets are adjusted by updateOffsets():

```
private void updateOffsets(int moveQuad)
{
  if (moveQuad == TiledSprite.SW) {   // offset to NE
    xOffset += tileWidth/2;
    yOffset -= tileHeight/2;
  }
  else if (moveQuad == TiledSprite.NW) {  // offset to SE
    xOffset += tileWidth/2;
    yOffset += tileHeight/2;
  }
  else if (moveQuad == TiledSprite.NE) {  // offset to SW
    xOffset -= tileWidth/2;
    yOffset += tileHeight/2;
  }
  else if (moveQuad == TiledSprite.SE) {  // offset to NW
    xOffset -= tileWidth/2;
    yOffset -= tileHeight/2;
  }
  else if (moveQuad == TiledSprite.STILL) {  // do nothing
  }
  else
    System.out.println("moveQuad error detected");
}
```

4.5. Drawing the World

AlienTilesPanel delegates the world drawing task to draw() in WorldDisplay:

```
public void draw(Graphics g)
{
  g.drawImage(floorIm, xOffset, yOffset, null); // draw floor image
  wItems.positionSprites(player, aliens);       // add the sprites
  wItems.draw(g, xOffset, yOffset);             // draw entities
  wItems.removeSprites();                       // remove sprites
}
```

WorldDisplay draws the floor GIF, suitably offset, but the entities resting on the floor (the blocks, pickups, and sprites) are left to WorldItems to render.

During WorldDisplay's loading phase, the WorldItems object is initialized with the locations of the blocks and pickups, but *not* sprites. The reason is that sprites move about at run time, so would have to be repeatedly reordered in WorldItem's internal data structures.

Instead, whenever the game surface needs to be drawn, the sprites' current positions are recorded *temporarily* in WorldItems by calling positionSprites(). After the drawing is completed, the sprite data is deleted with removeSprites().

This approach greatly simplifies the house-keeping carried out by WorldItems, as we will see. The drawback is the need for repeated insertions and deletions of sprite information. However, they are only five sprites in AlienTiles, so the overhead isn't excessive.

5. The WorldItems Class

WorldItems maintains an ArrayList of TileOccupier objects (called items) ordered by increasing tile row. Figure 10 shows that row 0 is the row 'furthest' back in the game, while the last row is nearest the front. When the ArrayList objects are drawn, the ones in the rows further back will be drawn first, matching the intended z-ordering of the rows.

A TileOccupier object can represent a block, pickup, or sprite.

The ArrayList changes over time. The most frequent change is to temporarily add sprites, so they can be drawn in their correct positions relative to the blocks and pickups. Also, pickups are deleted as they are collected by the player.

The WorldItems constructor stores floor information. This is used to translate the tile coordinates of the TileOccupiers into pixel locations on the floor.

```
// max pixel width/height of a tile
private int tileWidth, tileHeight;

// 'start of first even row' coordinate
private int evenRowX, evenRowY;

// 'start of first odd row' coordinate
```



```
  private int oddRowX, oddRowY;

  private ArrayList items;
         // a row-ordered list of TileOccupier objects


  public WorldItems(int w, int h, int erX, int erY,
                                  int orX, int orY)
  { tileWidth = w; tileHeight = h;
    evenRowX = erX; evenRowY = erY;
    oddRowX = orX; oddRowY = orY;
    items = new ArrayList();
  }
```

5.1. Adding an Entity

Adding an entity (either a pickup or a block) requires the creation of a TileOccupier object, and its placement in the items ArrayList sorted by its row/column position.

```
  public void addItem(String name, int type, int x, int y,
                                          BufferedImage im)
  { TileOccupier toc;
    if (y%2 == 0) // even row
      toc = new TileOccupier(name, type, x, y, im,
                                   evenRowX, evenRowY,
                                   tileWidth, tileHeight);
    else
      toc = new TileOccupier(name, type, x, y, im,
                                   oddRowX, oddRowY,
                                   tileWidth, tileHeight);
    rowInsert(toc, x, y);
  }
```

Each TileOccupier object must calculate its pixel location on the floor, which requires the tile coordinate of the occupier (x,y), the dimensions of a tile (tileWidth and tileHeight), and the start coordinate of the first even or odd row. If the TileOccupier is positioned on an even row (i.e. y%2 == 0) then it is passed the even row coordinate, otherwise the odd coordinate.

addItem() only deals with blocks or pickups, and so the type argument will be BLOCK or PICKUP. The creation of a SPRITE entity is handled by a separate method, posnSprite(), which is similar to addItem(). posnSprite() adds a sprite reference to the information in the TileOccupier object.

rowInsert() inserts the TileOccupier object into the ArrayList in increasing row order. Within a row, the objects are ordered by increasing column position.

5.2. Drawing Entities

WorldDisplay's draw() displays all the entities using a z-ordering that draws the rows further back first. Since the TileOccupier objects are stored in the ArrayList in increasing row order, this is achieved by cycling through them from start to finish.


```
public void draw(Graphics g, int xOffset, int yOffset)
{
  TileOccupier item;
  for(int i = 0; i < items.size(); i++) {
    item = (TileOccupier) items.get(i);
    item.draw(g, xOffset, yOffset);   // draw the item
  }
}
```

The TileOccupier draw() call is passed the x- and y- offsets of the floor image from the JPanel's top-left corner. They are used to draw the entity offset by the same amount as the floor.

5.3. Pickup Methods

WorldItems contains several pickup-related methods. They all have a similar structure, involving a loop through the items list looking for a specified pickup. Then a method is called in the found TileOccupier object.

One of the more involved methods is nearestPickup(). It is supplied with a tile coordinate, and returns the coordinate of the nearest pickup.

```
public Point nearestPickup(Point pt)
{
  double minDist = 1000000;   // dummy large value (a hack)
  Point minPoint = null;
  double dist;
  TileOccupier item;
  for(int i=0; i < items.size(); i++) {
    item = (TileOccupier) items.get(i);
    if (item.getType() == WorldDisplay.PICKUP) {
      dist = pt.distanceSq( item.getTileLoc() );
                                // get squared dist. to pickup
      if (dist < minDist) {
        minDist = dist;               // store smallest dist
        minPoint = item.getTileLoc();  // store associated pt
      }
    }
  }
  return minPoint;
}  // end of nearestPickup()
```

The pickups are found by searching for the PICKUP type. The square of the distance between the input point and a pickup is calculated, thereby avoiding negative lengths, and the current minimum distance and the associated pickup point is stored.

6. The TileOccupier Class

A tile occupier has a unique name, a type value (BLOCK, PICKUP, or SPRITE), a tile coordinate (xTile, yTile), and a coordinate relative to the top-left corner of the floor image (xDraw, yDraw) where the occupier's image should be drawn. The relationship between these coordinates is shown in Figure 11.

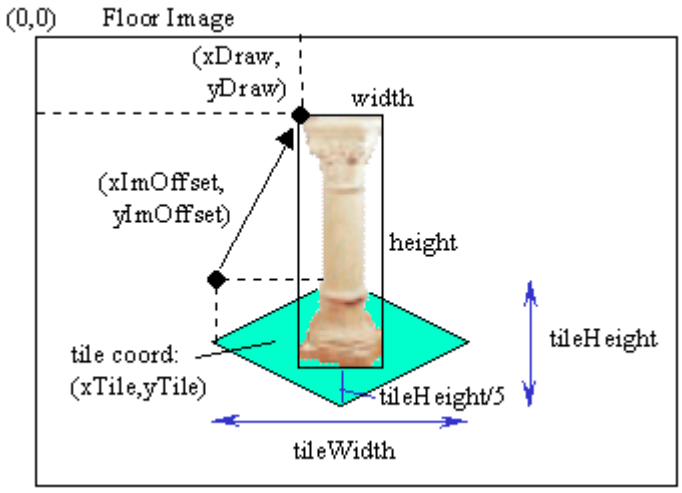

Figure 11. Positioning a Tile Occupier in a Tile.

xDraw and yDraw are relative to the floor image, and so floor offsets must be added to them before the image is drawn into the JPanel.

The constructor initializes the coordinate details, and calls calcPosition() to calculate xDraw and yDraw.

```
// globals
private String name;
private int type;      // BLOCK, PICKUP, or SPRITE
private BufferedImage image;
private int xTile, yTile;    // tile coordinate
private int xDraw, yDraw;
         // coordinate relative to the floor image where the tile
         // occupier should be drawn

private TiledSprite sprite = null;
  // used when the TileOccupier is a sprite


public TileOccupier(String nm, int ty, int x, int y,
          BufferedImage im, int xRowStart, int yRowStart,
                            int xTileWidth, int yTileHeight)
{ name = nm;
  type = ty;
  xTile = x; yTile = y;
  image = im;
  calcPosition(xRowStart, yRowStart, xTileWidth, yTileHeight);
}
```

If this object is in an even row, then xRowStart and yRowStart will hold the pixel location of the first even row, otherwise the location of the first odd row. The (x,y) arguments give the tile's location.

calcPosition() calculates the (xDraw,yDraw) coordinate relative to the floor image.

```
private void calcPosition(int xRowStart, int yRowStart,
                         int xTileWidth, int yTileHeight)
{
  // top-left corner of image relative to its tile
  int xImOffset = xTileWidth/2 - image.getWidth()/2;   // in middle
  int yImOffset = yTileHeight - image.getHeight() - yTileHeight/5;
                 // up a little from bottom point of the diamond

  // top-left corner of image relative to floor image
  xDraw = xRowStart + (xTile * xTileWidth) + xImOffset;
  if (yTile%2 == 0)    // on an even row
    yDraw = yRowStart + (yTile/2 * yTileHeight) + yImOffset;
  else       // on an odd row
    yDraw = yRowStart + ((yTile-1)/2 * yTileHeight) + yImOffset;
}
```

The (xDraw,yDraw) coordinate will cause the TileOccupier's image to be rendered so that its base appears to be resting on the tile, centered in the x-direction, and a little forward of the middle in the y-direction.

6.1. Additional Sprite Information

When a TileOccupier object is created for a sprite, the addSpriteRef() method is called to store a reference to the sprite. This is used by the draw() method, as explained below.

```
public void addSpriteRef(TiledSprite s)
{ if (type == WorldDisplay.SPRITE)
    sprite = s;
}
```

6.2. Drawing a Tile Occupier

When the draw() method is called, the (xDraw, yDraw) coordinate relative to the floor image is already known. Now the x- and y- offsets of the floor image relative to the JPanel must be added to get the image's position in the JPanel.

There is one complication: drawing a sprite. A sprite may be animated, and will almost certainly be represented by several images, so which one should be drawn? This task is delegated to the sprite, by calling its draw() method. Prior to the draw, the sprite's pixel position must be set.

```
public void draw(Graphics g, int xOffset, int yOffset)
{
  if (type == WorldDisplay.SPRITE) {
    sprite.setPosition( xDraw+xOffset, yDraw+yOffset);
                            // set its position in the JPanel
```



```
      sprite.drawSprite(g);   // let the sprite do the drawing
    }
    else    // the entity is a PICKUP or BLOCK
      g.drawImage( image, xDraw+xOffset, yDraw+yOffset, null);
  }
```

draw() in TileOccupier is the only place where the pixel coordinates maintained by the Sprite class are manipulated. Tile coordinates, held in the TiledSprite subclass, are utilized in the rest of AlienTiles.

7. The TiledSprite Class

A TiledSprite represents a sprite's position using tile coordinates (xTile, yTile); its most important method allows a sprite to move from its current tile to an adjacent one using a compass direction (quadrant): NE, SE, SW, NW.

An implicit assumption of TiledSprite is that a sprite cannot move around inside a tile, only step from one tile to another.

The constructor initializes a sprite's tile position, but only after checking its validity with WorldDisplay.

```
protected int xTile, yTile;    // tile coordinate for the sprite
protected WorldDisplay world;


public TiledSprite(int x, int y, int w, int h,
                        ImagesLoader imsLd, String name,
                        WorldDisplay wd)
{ super(0, 0, w, h, imsLd, name);
  setStep(0, 0);      // no movement
  world = wd;

  if (!world.validTileLoc(x, y)) {  // is tile (x,y) valid
    System.out.println("Alien tile location (" + x + "," + y +
                                      ") not valid; using (0,0)");
    x = 0; y = 0;
  }
  xTile = x; yTile = y;
}  // end of TiledSprite()
```

7.1. Moving to Another Tile

AlienTiles' staggered tiles layout means that the coordinates of the four tiles adjacent to the current one are calculated in a slightly different way depending on if the current tile is on an even or odd row.

The current tile in Figure 12 is in row 3 (odd), the one in Figure 13 on row 2 (even). The coordinates of the adjacent tiles are calculated slightly differently in the two cases.

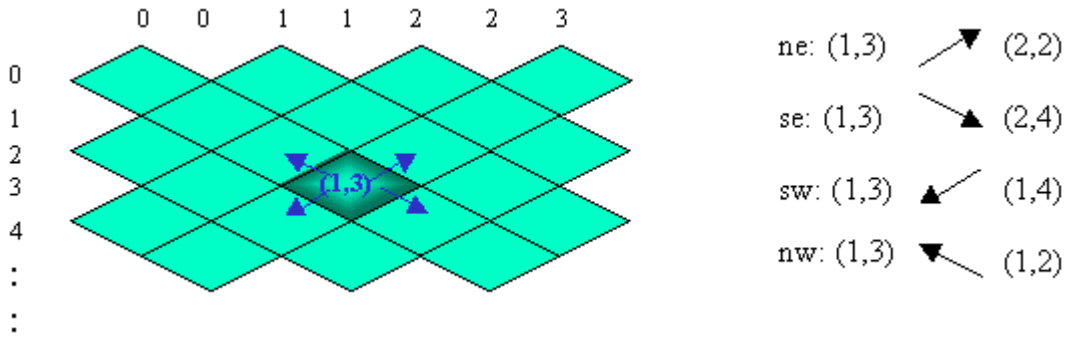

Figure 12. Moving from Tile (1,3).

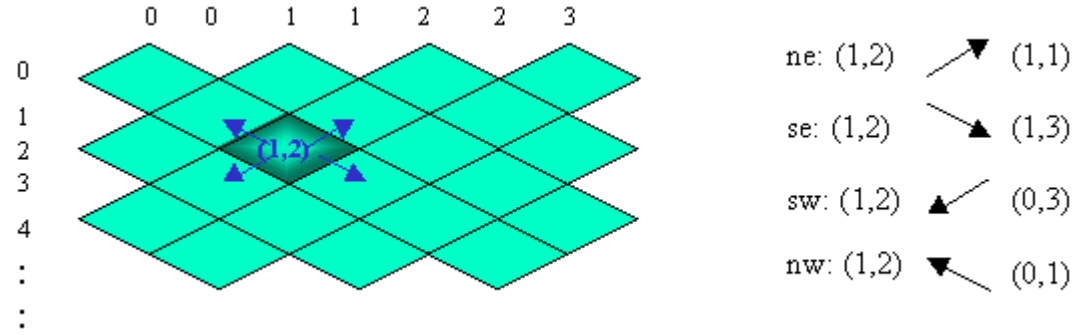

Figure 13. Moving from Tile (1,2).

tryMove() calculates a new tile coordinate based on the current location and the supplied quadrant. A four-way branch deals with the four possible directions, and each branch considers whether the starting point is on an even or odd row.

```
public Point tryMove(int quad)
{
  Point nextPt;
  if (quad == NE)
    nextPt = (yTile%2 == 0)? new Point(xTile,yTile-1) :
                             new Point(xTile+1,yTile-1);
  else if (quad == SE)
    nextPt = (yTile%2 == 0)? new Point(xTile,yTile+1) :
                           new Point(xTile+1,yTile+1);
  else if (quad == SW)
    nextPt = (yTile%2 == 0)? new Point(xTile-1,yTile+1) :
                             new Point(xTile,yTile+1);
  else if (quad == NW)
    nextPt = (yTile%2 == 0)? new Point(xTile-1,yTile-1) :
                             new Point(xTile,yTile-1);
  else
    return null;

  if (world.validTileLoc(nextPt.x, nextPt.y))
    // ask WorldDisplay if proposed tile is valid
    return nextPt;
  else
```



```
    return null;
} // end of tryMove()
```

The method is called tryMove() since there is a possibility that the desired quadrant direction is invalid, either because the new tile is a no-go area, it is occupied by a block, or the coordinate lies off the surface. These cases are checked by called validTileLoc() in WorldDisplay.

8. The PlayerSprite Class

PlayerSprite represents the player, and is a subclass of TiledSprite.

The player's aim is to move over the surface to collect the pickups before being beaten to death by the aliens.

The statechart for PlayerSprite in Figure 14 shows that the sprite has three main concurrent activities.

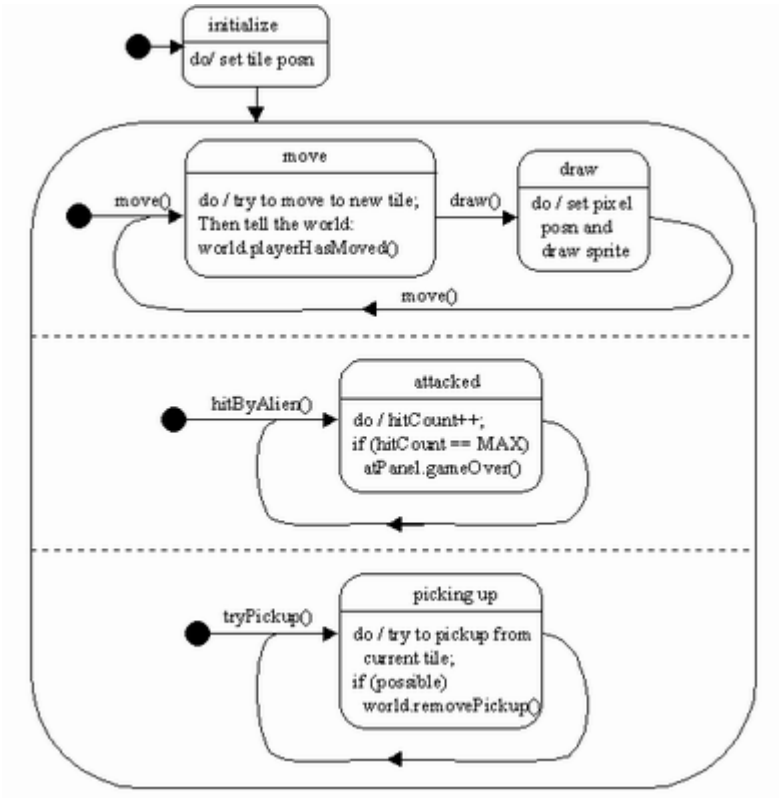

Figure 14. PlayerSprite Statechart.

The move() and tryPickup() transitions are triggered by the user from the keyboard. The hitByAlien() transition is initiated by the WorldDisplay object when an alien tells it that it has hit the player.

During various activities, the player must communicate with the WorldDisplay object (called world) and the AlienTilePanel game panel (called atPanel).

PlayerSprite attaches sound effects to different tasks, including: picking something up, failing to pick something up, being hit by an alien, and failing to move onto a no-go tile or a tile holding a block.

8.1. Moving (and Standing Still)

A PlayerSprite tries to move when the user presses one of the quadrant keys ('9', '3', '1', or '7').

```
public void move(int quad)
{
  Point newPt = tryMove(quad);
  if (newPt == null) {   // move not possible
    clipsLoader.play("slap", false);
    standStill();
  }
  else {    // move is possible
    setTileLoc(newPt);    // update the sprite's tile location
    if (quad == NE)
      setImage("ne");
    else if (quad == SE)
      setImage("se");
    else if (quad == SW)
      setImage("sw");
    else // quad == NW
      setImage("nw");
    world.playerHasMoved(newPt, quad);
  }
}  // end of move()
```

The attempt is handled by TiledSprite's inherited tryMove(), and the sprite's tile location is updated if it's successful.

The move is dressed up with an image change for the sprite, and the playing of a sound effect if the move is blocked.

A special (lazy) case of moving is 'standing still', which only requires an image change. This is triggered by the user pressing '5' on the numbers keypad.

```
public void standStill()
{ setImage("still"); }
```

8.2. Drawing the Player

The statechart includes a draw state, triggered by a draw() transition. The draw activity is implemented by using the setPosition() and draw() methods inherited from Sprite. However, the drawing is not initiated by code in PlayerSprite, but by WorldDisplay's draw() method:

```
public void draw(Graphics g)
// in WorldDisplay
{ g.drawImage(floorIm, xOffset, yOffset, null); // draw floor image
  wItems.positionSprites(player, aliens);       // add sprites
  wItems.draw(g, xOffset, yOffset);             // draw things
```



```
    wItems.removeSprites();                          // remove sprites
  }
```

As explained earlier, all the sprites, including the player, are added to WorldItems temporarily so they can be drawn in the correct order. Each sprite is stored as a TileOccupier object, and setPosition() and draw() are called from there.

8.3. Being Hit by an Alien

PlayerSprite maintains a hit counter, which is incremented by a call to hitByAlien() from the WorldDisplay object.

```
public void hitByAlien()
{ clipsLoader.play("hit", false);
  hitCount++;
  if (hitCount == MAX_HITS)    // player is dead
    atPanel.gameOver();
}
```

When hitCount reaches a certain value (MAX_HITS), it's all over. The sprite doesn't actually terminate at this point, it only notifies AlienTilePanel.

8.4. Trying to Pick up a Pickup

The user tries to pick up an item by pressing '2' on the numbers keypad. The hard work is to determine if the sprite's current tile location contains a pickup, and then to remove that item from the scene. The two operations are handled by WorldDisplay methods.

```
public boolean tryPickup()
{
  String pickupName;
  if ((pickupName = world.overPickup( getTileLoc())) == null) {
    clipsLoader.play("noPickup", false);     // nothing to pickup
    return false;
  }
  else {     // found a pickup
    clipsLoader.play("gotPickup", false);
    world.removePickup(pickupName);    // tell WorldDisplay
    return true;
  }
}
```

The name of the pickup on the current tile is obtained (it may be null if they isn't a pickup there), then the name is used in the deletion request.

9. The AlienSprite Class

AlienSprite implements the basic behaviour of an alien sprite, and is subclassed to create the AlienAStarSprite and AlenQuadSprite classes. AlienSprite is a subclass of TiledSprite.

Alien behaviour can be understood by considering the statechart in Figure 15.

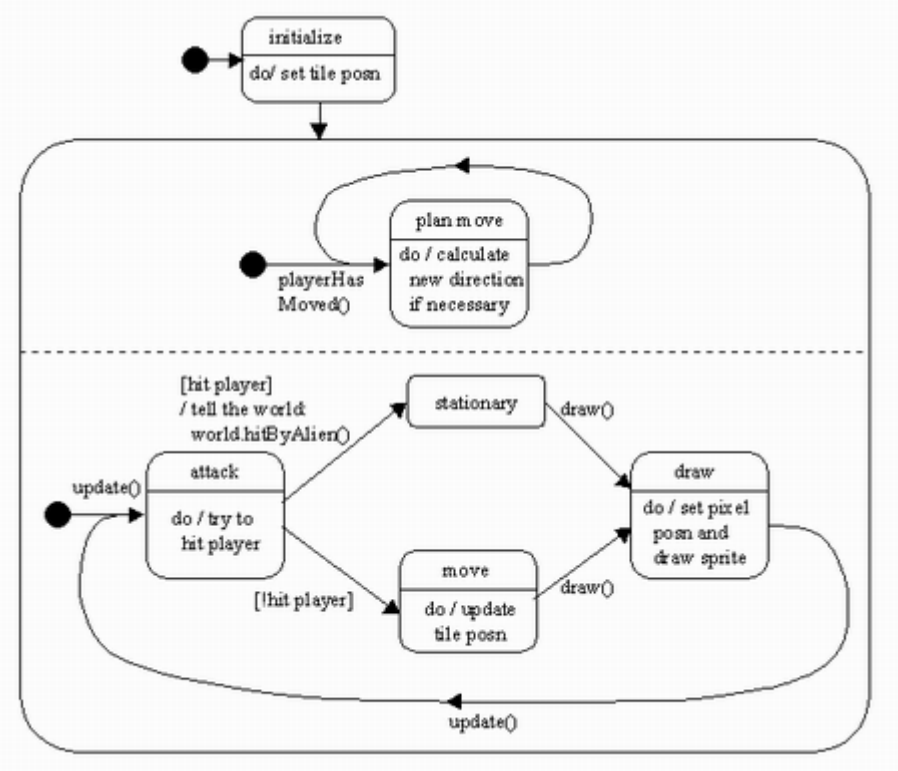

Figure 15. Alien Statechart.

The 'plan move' state is entered by the WorldDisplay object notifying the alien that the player has moved. This will probably cause it to recalculate its current direction or destination.

The other activity is the usual update/draw cycle driven by the animation loop in AlienTilesPanel. The alien tries to hit the player while in the 'attack' state . A successful hit is reported to the WorldDisplay object, and the alien stays where it is. Otherwise, the alien updates its position, in the hope of getting closer to the player. In the draw state, the sprite's tile coordinates are mapped to a pixel location and the sprite's image is rendered.

Responding to a player's movement is very sprite-specific, so playerHasMoved() is empty in AlienSprite:

```
public void playerHasMoved(Point playerLoc)
{ }
```

PlayerLoc contains the current tile coordinates for the PlayerSprite object.

9.1. Updating the AlienSprite

The 'attack', 'stationary', and 'move' states are encapsulated in update().

```
// globals
private final static int UPDATE_FREQ = 30;
private int updateCounter = 0;
    :

public void update()
{
  updateCounter = (updateCounter+1)%UPDATE_FREQ;
  if (updateCounter == 0) {   // reduced update frequency
    if (!hitPlayer())
      move();
  }
}
```

update() is called from AlienTilesPanel's animation loop, which executes at 40 FPS. Although this is a slow-ish frame rate, it still makes the aliens respond too quickly. Our solution is to use a counter to further reduce the update frequency.

hitPlayer() checks if the alien is on the same tile as the player. If it is then the WorldDisplay object is informed of a hit.

```
private boolean hitPlayer()
{
  Point playerLoc = world.getPlayerLoc();
  if (playerLoc.equals( getTileLoc() )) {
    world.hitByAlien();  // whack!
    return true;
  }
  return false;
}
```

The details of the 'move' state will vary from one alien to another, which translates to the alien subclasses overriding the move() method.

AlienSprite's move() carries out a random walk. getRandDirection() (a method inherited from TiledSprite) returns a quadrant, and this is tried out with TiledSprite's tryMove().

```
protected void move()
{
  int quad = getRandDirection();
  Point newPt;
  while ((newPt = tryMove(quad)) == null)
    quad = getRandDirection();
    // the loop could repeat for a while,
    // but it should eventually find a direction
  setMove(newPt, quad);
}
```

The new tile coordinate is use to update the sprite's position in setMove():


```
protected void setMove(Point newPt, int quad)
{
  if (world.validTileLoc(newPt.x, newPt.y)) {   // should be ok
    setTileLoc(newPt);
    if ((quad == NE) || (quad == SE))
      setImage("baddieRight");
    else if ((quad == SW) || (quad == NW))
      setImage("baddieLeft");
    else
      System.out.println("Unknown alien quadrant: " + quad);
  }
  else
    System.out.println("Cannot move alien to (" + newPt.x +
                                          ", " + newPt.y + ")");
}  // end of doMove()
```

setMove() double-checks the validity of the new tile, and also changes the sprite's appearance. The method is protected since only subclasses of AlienSprite will use it, as part of their versions of move().

As mentioned at the start of this section, update() handles the 'attack', 'stationary', and 'move' states of the alien statechart. Where is the 'draw' state processed? As with the PlayerSprite class, this task is part of the drawing operation carried out by WorldDisplay through its WorldItems object.

10. The AlienQuadSprite Class

AlienQuadSprite is a subclass of AlienSprite, so overrides that superclass' playerHasMoved() and move() methods. Another way of thinking about the specialization is that the 'plan move' and 'move' states in Figure 15 are being modified.

In the 'plan move' state, the alien calculates a quadrant direction (i.e. one of NE, SE, SW, or NW). The direction is chosen by finding the nearest pickup point to the player, then calculating that pickup's quadrant direction relative to the alien. This gives the alien a 'pickup guarding' behaviour, since the alien moves towards the pickup that the player (probably) wants to collect.

10.1. Planning a Move

playerHasMoved() calculates a quadrant direction for the sprite.

```
// global
private int currentQuad;
   :

public void playerHasMoved(Point playerLoc)
{
  if (world.hasPickupsLeft()) {
    Point nearPickup = world.nearestPickup(playerLoc);
           // return coord of nearest pickup to the player
```



```
      currentQuad = calcQuadrant(nearPickup);
    }
  }

  private int calcQuadrant(Point pickupPt)
  /* Roughly calculate a quadrant by comparing the
     pickup's point with the alien's position. */
  {
    if ((pickupPt.x > xTile) && (pickupPt.y > yTile))
      return SE;
    else if ((pickupPt.x > xTile) && (pickupPt.y < yTile))
      return NE;
    else if ((pickupPt.x < xTile) && (pickupPt.y > yTile))
      return SW;
    else
      return NW;
   }  // end of calcQuadrant()
```

calcQuadrant() could be more complex, but the emphasis is on speed. playerHasMoved() and calcQuadrant() will be called frequently, whenever the player moves, so there is no need to spend a large amount of time processing a single move.

10.2. Moving the AlienQuadSprite

The sprite tries to move in the currentQuad direction. If that way is blocked then it randomly tries another direction. This approach may lead to the sprite getting stuck for a while.

```
protected void move()
{ int quad = currentQuad;
  Point newPt;
  while ((newPt = tryMove(quad)) == null)
    quad = getRandDirection();
    // the loop could repeat for a while,
    // but it should eventually find a way
  setMove(newPt, quad);
}
```

11. The AlienAStarSprite Class

In a similar manner to AlienQuadSprite, AlienAStarSprite is a subclass of AlienSprite, so overrides that superclass' playerHasMoved() and move() methods.

The alien calculates a path to the player using the A* pathfinding algorithm. The path is stored as a sequence of tile coordinates that need to be visited in order to reach the player. In each call to move(), the sprite moves to the next coordinate in the sequence, giving it a 'player chasing' behaviour.

11.1. Planning a Move

The player keeps moving, which is reflected in the repeated calls to playerHasMoved (). We don't want to recalculate a path after every player move, since the change will

be minimal but still expensive to generate. Instead, the path is generated only when the player has moved MAX_MOVES steps. This saves on computation, and makes things a bit easier for the player.

```
// globals
private final static int MAX_MOVES = 5;

private int numPlayerMoves = 0;
private ArrayList path;    // tile coords going to the player
private int pathIndex = 0;
             :

public void playerHasMoved(Point playerLoc)
{ if (numPlayerMoves == 0)
    calcNewPath(playerLoc);
  else
    numPlayerMoves = (numPlayerMoves+1)%MAX_MOVES;
}


private void calcNewPath(Point playerLoc)
{ path = aStarSearch( getTileLoc(), playerLoc );
  pathIndex = 0;   // reset the index for the new path
}
```

11.2. The A* Algorithm

A* search finds a path from a start node to a goal node; in AlienTiles the starting point is the alien's current tile position, and the goal is the player's tile.

The algorithm repeatedly examines the most promising (highest scoring) tile position it has seen. When a tile is considered, the search is finished if that tile is the player's location, otherwise it stores the locations of the adjacent tiles for future exploration.

It scores a tile (let's call it *node*) by estimating the cost of the best path that starts at the alien's position, goes through *node*, and finishes at the player's tile.

The scoring formula is expressed using two functions, often called g() and h(); we'll break with tradition and call them getCostFromStart() and costToGoal():

score(node) = node.getCostFromStart() + node.costToGoal()

getCostFromStart() is the smallest cost of arriving at *node* from the starting tile (the alien's current position). costToGoal() is a *heuristic* estimate (an educated guess) of the cost of reaching the goal tile (the player's location) from *node*.

A* search is popular because it's guaranteed to find the shortest path from the start to the goal, as long as the heuristic estimate, costToGoal(), is *admissible*. Admissibility means that the node.costToGoal() value is always less than (or equal to) the actual cost of getting to the goal from *node*.

The A* algorithm has been proven to make the most efficient use of costToGoal(), in the sense that other search techniques cannot find an optimal path by checking fewer nodes.

If costToGoal() is inaccurate, returning too large a value, then the search will become unfocussed, examining nodes which will not contribute to the final path. Also, the

generated path may not be the shortest possible. However, a less accurate costToGoal () function may be easier (and faster) to calculate, so path generation may be quicker. Speed might be preferable, as long as the resulting path is not excessively meandering.

Both scoring functions, getCostFromStart() and costToGoal(), rely on being able to calculate a 'cost' of moving from one tile to another. Various costing approaches are possible, including the distance between the tiles, the cost in time, the cost of 'fuel', or weights based on the terrain type.

A* employs two list data structures, usually called open and closed. Open is a list of tiles which have not yet been examined (i.e. their adjacent tiles have not been scored). Closed contains the tiles which have been examined. The tiles in open are sorted by decreasing score, so the most promising tile is always the first one.

The following pseudo-code shows how the A* search progresses.

```
add the start tile to open;
create an empty closed list;

while (open isn't empty) {
  get the highest scoring tile x from open;
  if (x is the goal tile)
    return a path to x;   // we're done
  else {
    for (each adjacent tile y to x) {
      calculate the costFromStart() value for y;
      if ((y is already in open or closed) and
          (value is no improvement))
        continue;   // ignore y
      else {
        delete old y from open or close (if present);
        calculate costToGoal() and the total score for y;
        store y in open;
      }
    }
  }
  put x into closed;  // since we're finished with it
}
report no path found;
```

The pseudo-code is based on code in the article:

> "The Basics of A* for Path Planning", Bryan Stout
> In *Game Programming Gems*, Mike DeLoura (ed.)
> Charles River Media, 2000, part 3.3, pp. 254-263

The translation of the pseudo-code to the aStarSearch() method is quite direct:

```
  private ArrayList aStarSearch(Point startLoc, Point goalLoc)
  {
    double newCost;
    TileNode bestNode, newNode;

    TileNode startNode = new TileNode(startLoc);  // set start node
```



```
    startNode.costToGoal(goalLoc);

    // create the open queue and closed list
    TilesPriQueue open = new TilesPriQueue(startNode);
    TilesList closed = new TilesList();

    while (open.size() != 0) {  // while some node still left
      bestNode = open.removeFirst();
      if (goalLoc.equals( bestNode.getPoint() ))   // reached goal
        return bestNode.buildPath();   // return a path to that goal
      else {
        for (int i=0; i < NUM_DIRS; i++) {    // try every direction
          if ((newNode = bestNode.makeNeighbour(i, world)) != null) {
            newCost = newNode.getCostFromStart();
            TileNode oldVer;
            // if this tile already has a cheaper open or closed node
            // then ignore the new node
            if (((oldVer=open.findNode(newNode.getPoint())) !=null)&&
                (oldVer.getCostFromStart() <= newCost))
              continue;
            else if (((oldVer = closed.findNode( newNode.getPoint()))
                                                          != null) &&
                (oldVer.getCostFromStart() <= newCost))
              continue;
            else {   // store new/improved node, removing old one
              newNode.costToGoal(goalLoc);
              // delete the old details (if they exist)
              closed.delete( newNode.getPoint());  // may do nothing
              open.delete( newNode.getPoint());   // may do nothing
              open.add(newNode);
            }
          }
        } // end of for block
      } // end of if-else
      closed.add(bestNode);
    }
    return null;   // no path found
  }  // end of aStarSearch()
```

The code is simplified by being able to rely on the TilesList and TilesPriQueue classes to represent the closed and open lists. They store tile information as TileNode objects.

TilesList is essentially a wrapper for an ArrayList of TileNode objects, with additional methods for finding and deleting a node based on a supplied coordinate.

TilesPriQueue is a subclass of TilesList which stores TileNodes sorted by decreasing node score (i.e. the highest scoring node comes first).

The TileNode class has some interesting properties, which will be explained shortly.

11.3. Moving the AlienAStarSprite

AlienAStarSprite overrides AlienSprite's move() method, so that the next move is to the next tile in the path calculated by the A* algorithm.

```
  protected void move()
  {
```



```
    if (pathIndex == path.size())  // current path is used up
      calcNewPath( world.getPlayerLoc() );
    Point nextPt = (Point) path.get(pathIndex);
    pathIndex++;
    int quad = whichQuadrant(nextPt);
    setMove(nextPt, quad);
  }
```

If move() finds that the current path has been all used, then it initiates the calculation of a new one by calling calcNewPath().

12. The TileNode Class

A TileNode object stores tile details used by the A* algorithm. The most important are values for the costFromStart() and costToGoal() functions so that the overall score for the node can be worked out.

The costFromStart() function is the cost of the path that leads to this node from the starting tile. We use the simplest measure, the length of the path, with the step between adjacent tiles assigned a value of 1.

costToGoal() estimates the cost of going from the current node to the goal. This is a little harder to calculate in AlienTiles due to the staggered layout of the tiles.

Each TileNode also stores a reference to its parent, the node that was visited before it. The sequence of nodes back to the starting tile defines the path to this node (when reversed).

12.1. Calculating costToGoal()

costToGoal() calculates the *floor* of the straight line distance from the current tile to the goal.

```
public void costToGoal(Point goal)
{ double dist = coord.distance(goal.x, goal.y);
  costToGoal = Math.floor(dist);
}
```

This value is *not* always less than or equal to the actual cheapest path, so the path found by A* may not be optimal. However, the calculation is simple and fast, and the path is sufficient for AlienTiles.

The reason for using Math.floor() can be justified by considering an example. Figure 16 shows the four adjacent tiles to tile (1,3).

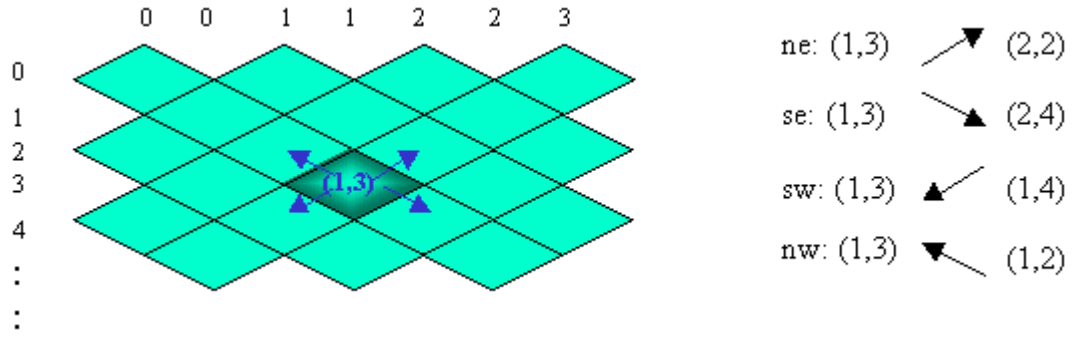

Figure 16. Tiles Adjacent to (1,3).

Figure 17 maps the five tile points to a rectangular grid, and shows the straight line distances between them.

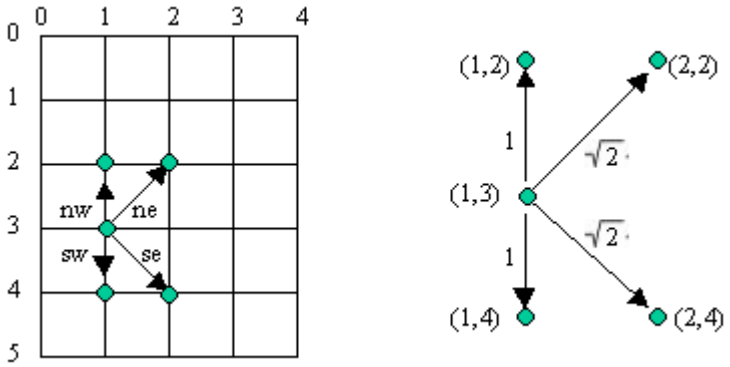

Figure 17. Straight Line Distances Between the Tiles in Figure 16.

The actual cost of moving to a neighbour is 1 in all cases. However, the straight line distances to two of the tiles (the north east and south east ones) are root(2). Fortunately, the floor of all the distances is 1 (1.414 is rounded down to 1), which makes the cost function optimal.

The Math.floor() solution works for adjacent tiles, but is less successful when the straight lines distances span multiple tiles.

13. More Information

The Isometrix Project (http://www.isometrix.org/) concentrates on isometric tile games. The articles section covers topics such as map formats, tile layout, placing objects, and lighting. The engines section lists code sources, tools, and demos.

We mentioned GameDev.net's Isometric and Tile-based Games section at the end of the last chapter (http://www.gamedev.net/reference/list.asp?categoryid=44). It currently lists over 30 articles.

An introductory book:

> *Isometric Game Programming with DirectX 7.0*
> Ernest Pazera, Premier Press, March, 2001

The first 230 or so pages are about MS Windows programming, and the examples use C. However, there's good stuff on the basics of rectangular and isometric games tile plotting, drawing, world and map coordinate systems, and moving about a map.

Some modern Java isometric or tile games examples, which come with source code:

- Javagaming.org: Scroller (http://sourceforge.net/projects/jgo-scroller/)
  A full-screen isometric scrolling game intended to illustrate how to write high-performance 2D games in J2SE 1.4.
- haphazard (http://haphazard.sourceforge.net/)
  A role-playing game set in an isometric world.
- CivQuest (http://civquest.sourceforge.net/)
  A strategy game inspired by *Civilization*, including game play against AI opponents. The coding is at an earlier stage.
- IsometricEngine (http://sourceforge.net/projects/jisoman/)
  An isometric game engine, with support for line-of-sight calculations, entity and terrain objects, a tile map and wall map. It also has a graphically mode for designing maps.
- JTBRPG (http://jtbrpg.sourceforge.net/)
  JTBRPG includes tools for creating role-playing isometric game content, and an engine for making it playable.
- YARTS (http://www.btinternet.com/~duncan.jauncey/old/javagame/)
  YARTS (Yet Another Real Time Strategy game) is a 2D rectangular tile-based real-time strategy game. The source code for the first version is available.
- Hephaestus (http://kuoi.asui.uidaho.edu/~kamikaze/Hephaestus/)
  A role-playing game construction kit based around 2D rectangular tiles.

Mappy for PC (http://www.tilemap.co.uk/mappy.php) can create isometric and hexagonal tile maps, and there are several Java-based playback libraries, including JavaMappy (http://www.alienfactory.co.uk/javamappy/).

The surface image created for AlienTiles (shown in Figure 3) was hacked together using MS PowerPoint and Paint – a reasonable approach for demos but not recommended for real maps.

The workings of the A* algorithm can be hard to visualize. The A* Demo page (http://www.ccg.leeds.ac.uk/james/aStar/) by James Macgill, lets the user create a search map, and watch the scoring process in action. The applet source code can be downloaded.

The pseudo-code I used is based on code from:

> "The Basics of A* for Path Planning", Bryan Stout
> In *Game Programming Gems*, Mike DeLoura (ed.)
> Charles River Media, 2000, part 3.3, pp. 254-263

There are two other articles in *Game Programming Gem* related to A* optimization.

An online version of another A* article by Bryan Stout:

> "Smart Moves: Intelligent Pathfinding"
> Bryan Stout, August 1997
> http://www.gamasutra.com/features/19970801/pathfinding.htm

It includes a PathDemo application which graphically illustrates several search algorithms, including A*.

The A* algorithm tutor (http://www.geocities.com/SiliconValley/Lakes/4929/astar.html) by Justin Heyes-Jones offers a detailed account of the algorithm.

Amit J. Patel's Web site on games programming (http://www-cs-students.stanford.edu/~amitp/gameprog.html) covers several relevant topics, including pathfinding (with a bias towards A*), tile games, and the use of hexagonal grids.

Information on A* can be found at game AI sites, usually under the pathfinding heading. Two excellent sources:

- Game AI Site (http://www.gameai.com/)
- GameDev's AI section (http://www.gamedev.net/reference/list.asp?categoryid=18)

A modern AI textbook, with plentiful discussion of search algorithms, including A*:

> *Artificial Intelligence: A Modern Approach*
> Stuart Russell and Peter Norvig
> Prentice Hall, 2nd edition, 2002
> http://aima.cs.berkeley.edu/

Many of the pseudo-code examples from the book have been rewritten in Java (including those for doing search); they're available from the Web site.

