Chapter 10. 3D Sprites

In this chapter, we develop a Sprite3D class, very similar in nature to the 2D sprite class of chapter ??. Subclasses of Sprite3D are used to create different kinds of sprites, which are placed in a world filled with scenery and obstacles.

The user's sprite (the robot), and the chasing sprite (the hand) are shown in action in Figure 1.

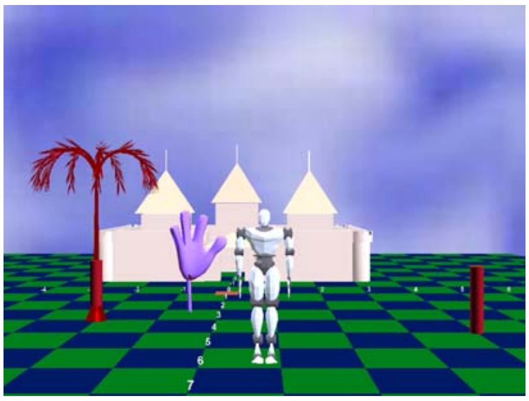


Figure 1. 3D Sprites in Action.

Other features illustrated by the Tour3D application:

- Sprite control using Behavior subclasses;
- A simple third person camera that adjusts the user's viewpoint as the user's sprite moves;
- Obstacles which a sprite cannot pass through (represented by cylinders in Figure 1). A sprite is also prevented from moving off the checkered floor;
- A "tour" text file to load obstacle and scenery information. The scenery models (e.g. the castle and the palm tree) are loaded with PropManager objects;
- The world's background is drawn using a scaled JPEG;
- The application is configured to be full-screen.

UML Diagrams for Tour3D

Figure 2 shows the UML diagrams for all the classes in the Tour3D application. Only the class names are shown.

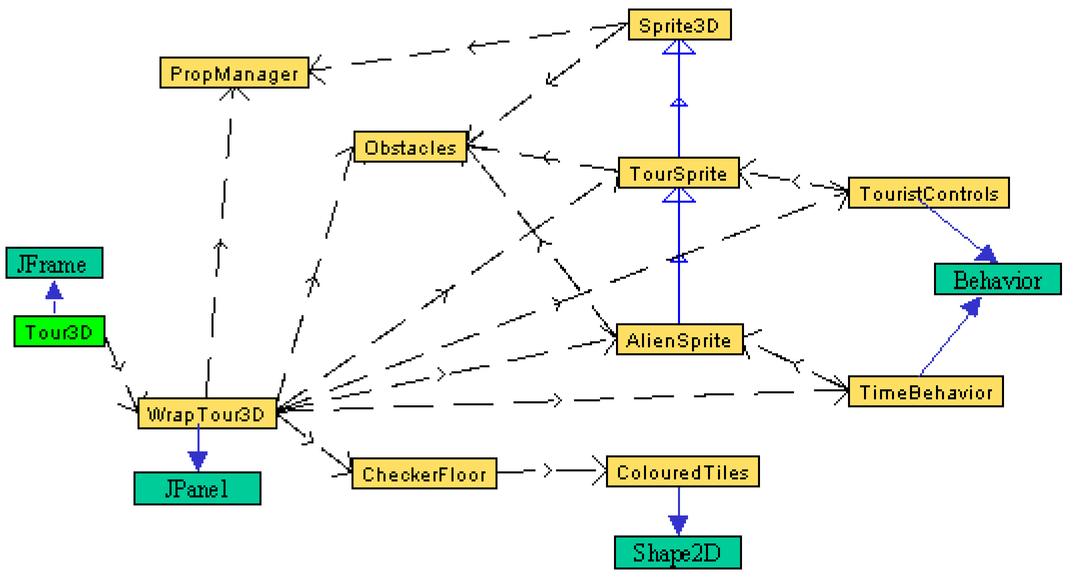


Figure 2. UML Class Diagrams for Tour3D.

Tour3D is the top-level JFrame for the application.

WrapTour3D creates the 3D world, and is similar in many respects to the earlier 'Wrap' classes in that it creates the checkered floor, and sets up the lighting. However, this version also loads the scenery, obstacles, and creates the sprites.

PropManager is unchanged from the code in the Loader3D application of chapter 10.

CheckerFloor and ColouredTiles are the same classes as in previous examples.

An Obstacles object stores information about the scene's obstacles.

The sprites are subclasses of Sprite3D. The robot is an instance of TourSprite, while the hand is an AlienSprite object. TourSprite is controlled by TouristControls, and TimeBehavior updates AlienSprite. Both TouristControls and TimeBehavior are subclasses of Java 3D's Behavior class.

WrapTour3D: Creating the World

Figure 3 shows the methods defined in WrapTour3D.

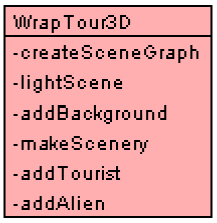


Figure3. WrapTour3D Methods.

WrapTour3D sets up the checkered floor and lights as before. However, addBackground() now uses a scaled image, and there are three new methods: makeScenery(), addTourist(), and addAlien(). These methods are called by createSceneGraph() to add scenery, obstacles, and the sprites to the scene.

The scene graph for the world in Figure 1 is shown in Figure 4. Its details will be explained in subsequent sections.

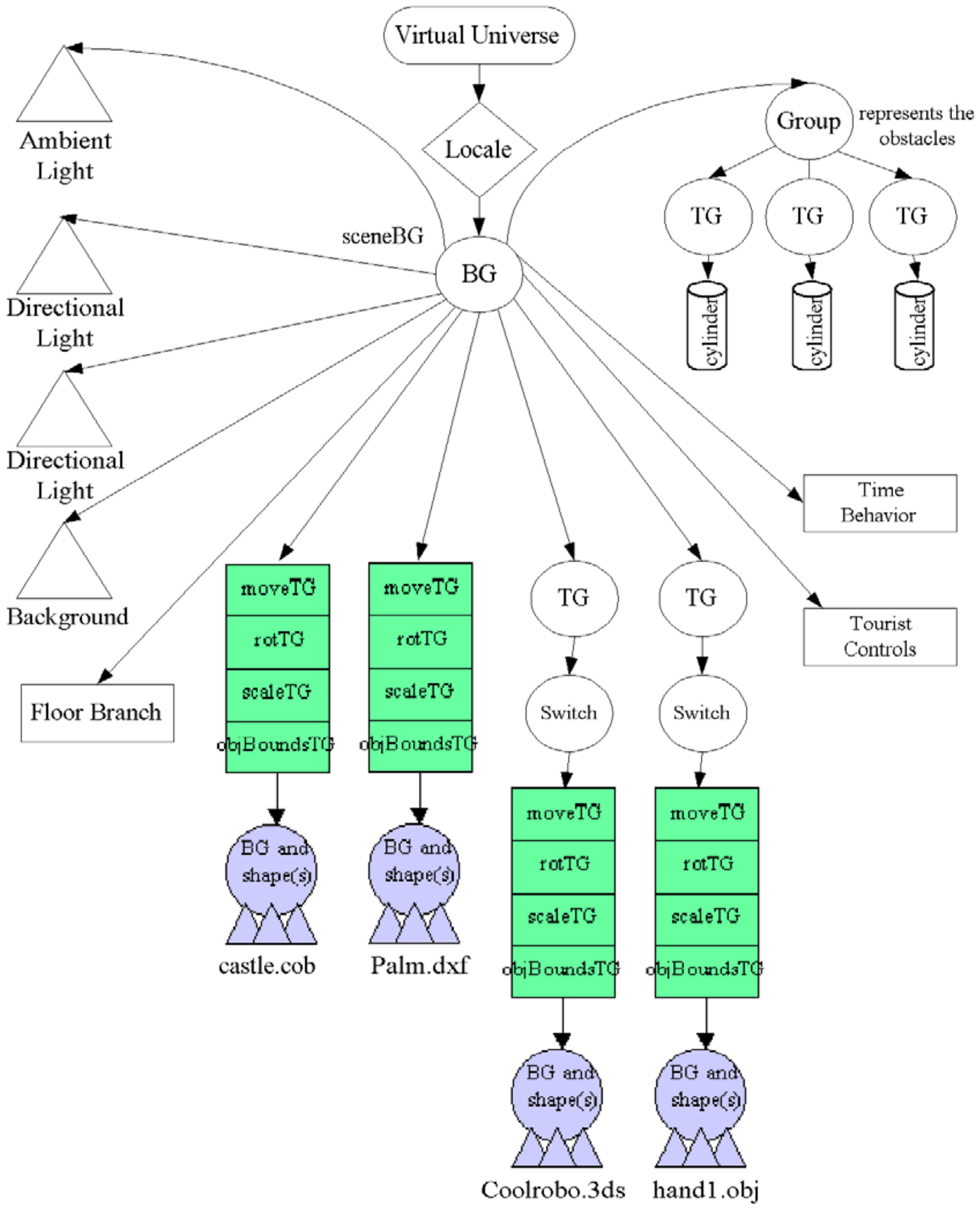


Figure 4. Scene Graph for the Figure 1 World.

Two familiar methods have disappeared from WrapTour3D: initUserPosition() and orbitControls(). The positioning (and adjustment) of the user's viewpoint is now dealt with by TouristControls.

A Background Image

As mentioned in chapter 8, a Background node can use a solid color, an image, or a geometric shape (e.g. a sphere, a box) with an image rendered onto it. In Tour3D, we utilise a picture of a hazy sky, 400*400 pixels in size, stored in bigSky.jpg.

```
TextureLoader bgTexture =
              new TextureLoader("models/bigSky.jpg", null);
Background back = new Background(bgTexture.getImage());
back.setImageScaleMode(Background.SCALE_FIT_MAX);
back.setApplicationBounds( bounds );
sceneBG.addChild( back );
```

The image is loaded as a texture, then converted to ImageComponent2D form for the Background object. Java 3D v.1.3 added several scaling modes to Background; the one we employ scales the image to fit the display window. This can cause significant distortions to the original picture.

Another scaling mode is Background.SCALE_REPEAT, which tiles the image over the display area.

A drawback of using a background image is that it remains stationary in the background, even when the viewpoint moves or rotates.

Full-Screen Display

There are two approaches to making a full-screen application in Java 3D, corresponding to the two approaches for Java: either the display window's dimensions can be set to match those of the monitor, or Full Screen Exclusive Mode (FSEM) can be deployed. Both of these techniques were explained in chapter ??, in the context of 2D Java games.

When writing a Java 3D application, which technique is to be preferred? In terms of speed, there seems little difference between them, probably because Java 3D already has fairly direct access to the graphics hardware through OpenGL or DirectX.

One advantage of using FSEM is control over screen resolution. A disadvantage is that FSEM interacts poorly with Swing components, but these are unlikely to be needed in most full-screen games. A limitation on FSEM is that GraphicsDevice.isFullScreenSupported() must return true. It usually does with Windows machines, but fails with many flavours of Unix. We will be using FSEM with the Java 3D application in the next chapter.

In Tour3D, we resize the display window, which requires three pieces of code. In the Tour3D class, the menu bars and other JFrame decoration must be turned off:

```
setUndecorated(true);
```

In WrapTour3D, the panel must be resized to fill the monitor:

```
setPreferredSize( Toolkit.getDefaultToolkit().getScreenSize() );
```

A full screen application with no menu bar raises the question of how to terminate the program. The usual approach is to add a KeyAdapter anonymous class to the window that has keyboard focus, which is the Canvas3D object in this application:

```
canvas3D.setFocusable(true);
canvas3D.requestFocus();

canvas3D.addKeyListener( new KeyAdapter() {
// listen for esc, q, end, ctrl-c on the canvas to
// allow a convenient exit from the full screen configuration
   public void keyPressed(KeyEvent e)
   { int keyCode = e.getKeyCode();
     if ((keyCode == KeyEvent.VK_ESCAPE) ||
         (keyCode == KeyEvent.VK_Q) ||
         (keyCode == KeyEvent.VK_END) ||
         ((keyCode == KeyEvent.VK_C) && e.isControlDown()) ) {
       win.dispose();
       System.exit(0);    // exit() isn't sufficient usually
     }
   }
 });
```

Catching KeyEvents in WrapTour3D does not preclude their use in other parts of the application. As we will see, the TouristControls class also utilises KeyEvents to govern the movement of the robot sprite and to adjust the user's viewpoint.

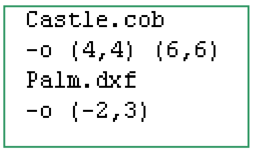
The listener responds to the typical "kill" keys used in application. The unusual aspect of the code is the dispose() call, applied to win, which is a reference to the top-level JFrame created in Tour3D. I found that a call to exit() alone would kill the application but often not clear the screen of the world image.

Adding Scenery and Obstacles

We make a distinction between scenery and obstacles in Tour3D: scenery comes from external models (e.g. the castle, palm tree) and are loaded with PropManager objects. A crucial attribute of scenery is it's intangibility: the robot and hand sprites can move right through it if they wish. In contrast, a sprite is disallowed from passing through an obstacle.

Scenery and obstacle data is read from a text file whose name is supplied on the command line when Tour3D is started. The format of a 'tour' file is simple: each line contains the filename of a model or a –o sequence of coordinates.

The ctour.txt file used to decorate the world in Figure 1 is:



```
Castle.cob
-o (4,4) (6,6)
Palm.dxf
-o (-2,3)
```

The –o sequences are integer coordinates on the XZ plane where obstacles will be placed. There can be any number of coordinates in the sequence; we have two –o lines in ctour.txt simply as an example; the 3 points could be listed on a single –o line.

The obstacle coordinates are passed to an Obstacle object which creates the necessary data structures, including the on-screen cylinders.

The model filenames are assumed to be located in the /models subdirectory, and to come with "coord" data files for positioning them in the scene. The hard work of actually loading a model is passed to a PropManager object.

The loading of the 'tour' file is done by makeScenery() in WrapTour3D. A code fragment from that method follows:

```
obs = new Obstacles();    // initialise Obstacle object
PropManager propMan;
   :
BufferedReader br =
        new BufferedReader( new FileReader(tourFile));
String line;
while((line = br.readLine()) != null) {
  if (line.startsWith("-o"))   // save obstacle info
    obs.store( line.substring(2).trim() );
  else {    // load scenery
    propMan = new PropManager(line.trim(),true);
    sceneBG.addChild( propMan.getTG() );    // add to world
  }
}
br.close();
sceneBG.addChild( obs.getObsGroup() );  // add obs to scene
```

A PropManager object creates a scene graph branch containing a chain of TransformGroups. In Figure 4, the chains above the BranchGroups for the castle and palm tree are drawn as rectangles. A chain of TransformGroups may be considered too much of an overhead for loading a model, but it can be removed fairly easily: PropManager must be extended with a method which switches off the capability bits in the TransformGroups. For example:

```
moveTG.clearCapability(TransformGroup.ALLOW_TRANSFORM_READ);
moveTG.clearCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

This should be done before the branch is added to the main scene and compiled. Compilation will optimise the chain away to a single TransformGroup, since Java 3D will notice that none of the chain's nodes can be transformed at run time.

Obstacles

The Obstacle object created by makeScenery() maintain three types of information:

- a 2D array of booleans called obs, which indicates if a particular (x,z) location is occupied by an obstacle;
- a 2D array of BoundingSphere objects called obsBounds, which specifies the influence of an obstacle at a given (x,z) location;
- a Group node called obsGroup, which holds the cylinders representing the obstacles.

The full UML class diagram for Obstacles is given in Figure 5.

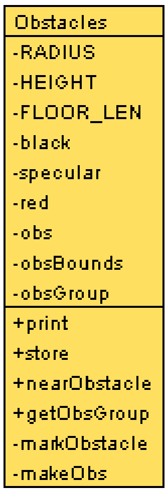


Figure 5. UML Class Diagram for Obstacles.

The coding of the class is simplified by restricting the obstacles to integer positions on the checkboard, which permits array-based data structures and algorithms to be employed.

A given (x,z) coordinate is checked for obstacles with nearObstacle() (called by the sprites from the Sprite3D class). It returns false if the supplied position is outside the floor area, or too near an obstacle. Nearness testing is done by determining if a bounding sphere centered at the coordinate intersects with any of the bounding spheres in obsBounds:

```
BoundingSphere bs = new BoundingSphere( pos, radius);
for (int z=0; z <= FLOOR_LEN; z++)
  for(int x=0; x <= FLOOR_LEN; x++)
    if (obs[z][x]) {   // does (x,z) have an obstacle?
      if (obsBounds[z][x].intersect(bs))
        return true;
    }
return false;
```

The algorithm is exhaustive in that it tests every obstacle against the supplied position (pos); a more efficient approach would use the pos value to limit the number of obstacles considered.

Each obstacle is displayed as a cylinder, placed below a TransformGroup to orient it on screen, as shown in the scene graph in Figure 4. The position uses the (x,z) coordinate of the obstacle, and moves the cylinder upwards by HEIGHT/2 so that it's base is resting on the floor.

Sprite3D

Sprite3D is the main class for creating 3D sprites; the TourSprite subclass is used to create the user's robot sprite, and AlienSprite is a subclass of TourSprite for the alien hand.

TourSprite is controlled by the TouristControls Behavior class, which monitors user key presses and can adjust the sprite's position or the user's viewpoint. AlienSprite chases the user's robot, and is periodically updated by the TimeBehavior class.

Figure 6 shows the visible methods of the sprite and Behavior classes, and the relationships between them.

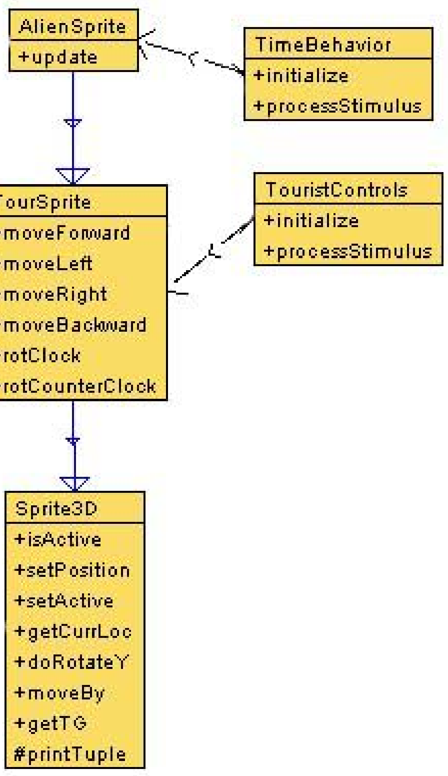


Figure 6. Sprite and Behavior Classes.

Sprite3D bears many similarities with our first 2D sprite class, ImageSprite, discussed in chapter ??. It represents a single model which can move about the XZ plane, rotate only about the y-axis, and can detect obstacles and floor boundaries. The sprite can be made inactive, which will cause it to disappear from the scene.

The constructor for Sprite3D utilises PropManager to load the model representing the sprite. It then adds a Switch node and TransformGroup above the model's graph; the result can be seen in the branches for the robot (Coolrobo.3ds) and alien hand (hand1.obj) in Figure 4. The relevant code:

```
PropManager propMan = new PropManager(fnm, true);

visSwitch = new Switch();   // for sprite visibility
visSwitch.setCapability(Switch.ALLOW_SWITCH_WRITE);
visSwitch.addChild( propMan.getTG() );     // add obj to switch
visSwitch.setWhichChild( Switch.CHILD_ALL );   // make visible

objectTG = new TransformGroup();  // for sprite moves
objectTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
objectTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
objectTG.addChild( visSwitch );   // add switch to TG
```

The objectTG node is made available to the application's scene via getTG(), which is called by WrapLoader3D.

Visibility

The Switch node is used to control the image branch's visibility. This is done in the setActive() method of Sprite3D:

```
   public void setActive(boolean b)
  { isActive = b;
    if (!isActive)
       visSwitch.setWhichChild( Switch.CHILD_NONE ); //make invisible
    else if (isActive)
      visSwitch.setWhichChild( Switch.CHILD_ALL );   // make visible
  } // end of setActive()
```

In Java 3D, there are at least three ways of controlling the visibility of a model:

- use setWhichChild() on a Switch above the model (as here);
- use setVisible() on the model's rendering attributes;
- use TransparencyAttributes, as detailed in the LoaderInfo3D application of chapter 9.

In older versions of Java 3D, there were some problems with using TransparencyAttributes, especially with multiple models in a scene, but these bugs have been addressed in recent versions.

The overhead of manipulating rendering or transparency attributes can be quite high, and will continue producing an overhead during rendering. A Switch node is above the model in the scene graph, which means that rendering does not need to visit the model at all when the Switch node is set to CHILD_NONE – a clear gain in efficiency.

Another advantage of Switch is that it can be placed above Group nodes to control the visibility of subgraphs in the scene. Attribute approaches only apply to individual Shape3D nodes.

Movement and Rotation

The addition of yet another TransformGroup to the model's scene branch is for coding simplicity. It means that Sprite3D does not need to know about the graph structure returned by PropManager's getTG().

A sprite can be moved with setPosition() and moveBy(), and rotated with doRotateY (). The reader may think this means adding two TransformGroup's to the branch – one to handle movement, one for the rotation, which is the approach taken in PropManager. The answer is 'no', because of the movement behaviour we want for the sprite.

In PropManager, the use of a separate moveTG node above rotTG has the effect that rotations do not affect the move directions. For instance, the model can be rotated 90 degrees around the y-axis, changing its local coordinate system so that the positive x-axis is pointing away from the viewer (along the old –z axis). But moveTG is above the rotTG node, so it will be unaffected by the change to the coordinate system: a move of 2 units along the positive x-axis will still move the model 2 units to the right from the user's viewpoint.

This kind of behaviour is suitable for the Loader3D application, where the user wants to position a model on-screen in an intuitive way, without requiring a mental model of rotated coordinate spaces.

The required behaviour for a sprite moving around the world is quite different. When a sprite moves 'forward', it *should* move forward according to the direction it is currently facing. In other words, we **do** want rotation to affect movement. This is achieved by **not** separating the movement and rotational transforms into two nodes; instead they are all applied to the same objectTG TransformGroup.

doMove() makes the sprite move by the distance specified in a vector:

```
private void doMove(Vector3d theMove)
// move the sprite by the amount in theMove
{
  objectTG.getTransform( t3d );
  toMove.setTranslation(theMove);    // overwrite previous trans
  t3d.mul(toMove);
  objectTG.setTransform(t3d);
}
```

The coding is straightforward, but where are the Transform3D objects t3d and toMove created? They are declared globally, and created in the constructor of Sprite3D, for reasons of speed. The alternative would be to create new Transform3D objects each time that doMove() was called, which is quite inefficient, due to the overheads of object creation and garbage collection. Reusing 'temporary' objects is a good way of improving Java's speed.

doRotateY() is very similar, and uses another global 'temporary' Transform3D object called toRot:

```
public void doRotateY(double radians)
// rotate the sprite by radians around its y-axis
{
  objectTG.getTransform( t3d );
  toRot.rotY(radians);   // overwrite previous rotation
  t3d.mul(toRot);
  objectTG.setTransform(t3d);
}
```

Obstacle and Boundary Detection

The sprite should not pass through obstacles or move off the floor. This behaviour is obtained by utilising the Obstacles object. A reference is passed into the sprite at creation time, and then used in moveBy(). moveBy() is the public movement method for the sprite, and accepts an (x,z) step:

```
public boolean moveBy(double x, double z)
// Move the sprite by offsets x and z, but only if within
// the floor and there is no obstacle nearby.
{
  if (isActive()) {
    Point3d nextLoc = tryMove(new Vector3d(x, 0, z));
    if (obs.nearObstacle(nextLoc, radius*OBS_FACTOR))
      return false;
    else {
      doMove( new Vector3d(x,0,z) );
      return true;
    }
  }
  else  // not active
    return false;
}  // end of moveBy()
```

moveBy() calculates its next position by calling tryMove(), which is almost the same as doMove() except that it does not adjust the position of objectTG. The possible new location, nextLoc, is passed to nearObstacle() of the Obstacles object for testing. If the new location is acceptable, the step is actually made, by calling doMove().

This approach nicely separates the issues of obstacle and boundary detection from the sprite, placing them in the Obstacles class. The other design aim was to implement this form of collision detection without utilising features in Java 3D.

Java 3D can be employed for collision detection in two main ways:

1. Java 3D can generate an event when one shape intersects with another, which is then processed by a Behavior object. The drawback is that such events only occur once the shapes have intersected. What is really required is an event *just before* the shapes intersect.
2. Java 3D picking can query whether moving the user's viewpoint will result in a collision with an object in the scene. This approach is suitable for first person games where the viewpoint represents the player. Tour3D is the beginnings of a third person game, where the viewer is distinct from the player (the robot). We return to this picky question when we look at first person games in chapter ??.

Updating the Sprite

A comparison of ImageSprite (the 2D sprite class) and Sprite3D highlights an important difference between our 2D and 3D games programming styles. The 2D games all use an update-redraw cycle, with a Timer object to control the cycle's frequency. Sprite3D has no redraw method, and there is no Timer object controlling its redraw rate.

The difference is due to the high level nature of Java 3D's scene graph. Java 3D controls graph rendering, so it handles redraws, and their frequency. At the programming level, we only have to change the scene (e.g. by adjusting the objectTG node), and sit back. Of course, if we want direct control we can switch from the default *retained mode* to *immediate mode*, as explained in chapter 8.

TourSprite

TourSprite is a simple class which subclasses Sprite3D in order to fix the movement step and rotation amounts of the sprite. A code fragment:

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

  public TourSprite(String fnm, Obstacles obs)
  { super(fnm, obs);  }
      :
  public boolean moveForward()
  { return moveBy(0.0, MOVERATE); }
      :
  public void rotClock()
  { doRotateY(-ROTATE_AMT); }  // clockwise
}
```

The reason for TourSprite's simplicity is that it does not contain any 'behaviour' code to specify when the move and rotation methods should be called. Java 3D encourages this kind of programming to be placed in a separate Behavior class (TouristControls for TourSprite). Behavior classes are explained below.

AlienSprite

The chasing behaviour of the alien makes the coding of AlienSprite more interesting. The alien is driven by a TimeBehavior object, which calls its update() method periodically.

update() uses the alien's and robot's current positions to calculate a rotation which makes the alien turn to face the robot. Then the alien moves towards the robot. A complication is dealing with any obstacles between the alien and the robot. Once the alien is sufficiently close to the robot, an exciting message is printed to standard output (this is, after all, just a demo ☺).

In many ways, the behavior in AlienSprite is akin to the ?? sprite in the Tiles game of chapter ??.

update()'s definition:

```
  public void update()
  // called by TimeBehaviour to update the alien
  { if (isActive()) {
      headTowardsTourist();
      if (closeTogether(getCurrLoc(), ts.getCurrLoc()))
        System.out.println("Alien and Tourist are close together");
    }
  }
```

headTowardsTourist() rotates the sprite then attempts to move it forward:

```
  private void headTowardsTourist()
  {
    double rotAngle = calcTurn( getCurrLoc(), ts.getCurrLoc());
    double angleChg = rotAngle-currAngle;
    doRotateY(angleChg);   // rotate to face tourist
    currAngle = rotAngle;  // store new angle for next time
    if (moveForward())
      ;
    else if (moveLeft())
      ;
    else if (moveRight())
```

```
      ;
    else if (moveBackward())
      ;
    else
      System.out.println("Alien stuck!");
  }
```

AlienSprite extends TourSprite, so can use the movement and rotation methods defined there.

The obstacles problem is dealt with by simply trying to move in every direction until one succeeds. This may lead to the sprite moving in a rather inefficient manner (and even into livelock), due to the lack of any path planning, but it is quite satisfactory (and fast) in a world with few obstacles.

calcTurn() deals with seven possible positional relationships between the alien and the robot, which can be understood by referring to Figure 7.

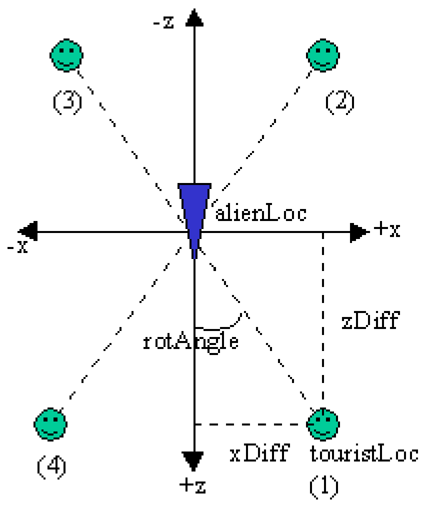


Figure 7. Possible Angles Between the Alien and Robot.

The alien begins by facing along the positive z-axis, towards the user's viewpoint. The rotation (rotAngle) is calculated relative to that starting angle so that the rotation change from the previous orientation can be obtained by subtraction (see the start of headTowardsTourist() for the code which does this).

The tourist may be in any of the four different quadrants marked in figure 7, or on the positive or negative x-axes (i.e. with a zero z value), or at the same spot as the alien; altogether seven possibilities. Note that a positive rotation around the y-axis is counter-clockwise.

The possibilities for rotAngle are shown in Table 1.

Quadrant	x loc	z loc	rotAngle
(1)	+ve	+ve	arctan x/z
(2)	+ve	-ve	pi + arctan x/-z
(3)	-ve	-ve	pi + arctan –x/-z
(4)	-ve	+ve	arctan –x/z
On the +x axis	+ve	0	pi/2
On the –x axis	-ve	0	-pi/2
Same spot	0	0	0

Table 1. Positions for the Robot Relative to the Alien.

These choices are encoded in calcTurn() as a series of if-tests after calculating xDiff and zDiff (the x-axis and z-axis distances between the two sprites).

In fact, the calculations for quadrants (1) and (4) and quadrants (2) and (3) could be combined since the signs of x and z are implicit in the values for xDiff and zDiff.

Behaviours in Java 3D

A Behavior object is used to monitor events occurring in the application, such as key presses, the rendering of frames, the passage of time, the movement of the user's viewpoint, Transform3D changes, and collisions. These events, called *wakeup criteria*, activate the Behavior object so it can carry out specified tasks.

A typical Behavior subclass has the following format:

```
public class FooBehavior extends Behavior
{ private WakeupCondition wc;   // what will wake the object
      :
  public FooBehavior(…)
  {  // initialise globals
      wc = new ....  //  create the wakeup criteria
  }

  public void initialize()
  // register interest in the wakeup criteria
  {  wakeupOn(wc);  }

  public void processStimulus(Enumeration criteria)
  {  WakeupCriterion wakeup;
     while (criteria.hasMoreElements() ) {
       wakeup = (WakeupCriterion) criteria.nextElement();
       // determine the type of criterion assigned to wakeup
       // carry out the relevant task
     }
     wakeupOn(wc);  // re-register interest
```

```
   } // end of processStimulus()
}  // end of FooBehavior class
```

A subclass of Behavior must implement initialize() and processStimulus().

initialize() should register the behaviour's wakeup criteria, but other initialization code can be placed in the constructor for the class.

processStimulus() is called by Java 3D when an event (or events) of interest to the behaviour is received. Often, the simple matter of processStimulus() being called is enough to decide what task should be carried out (e.g. as in TimeBehavior). In more complex classes, the events passed to the object must be analyzed.

A common error when implementing processStimulus() is to forget to re-register the wakeup criteria at the end of the method. If this is not done, the behaviour will not be triggered again.

A WakeupCondition object can be a combination of one or more WakeupCriterion. There are many subclasses of WakeupCriterion, including:

- WakeupOnAWTEvent – for AWT events such as key presses and mouse movements. WakeupOnAWTEvent is used in TouristControls.
- WakeupOnElapsedFrames – an event can be generated after a specified number of renderings. This criterion should be used with care since it may result in the object being triggered many times per second.
- WakeupOnElapsedTime – an event can be generated after a specified time interval. WakeupOnElapsedTime is used in TimeBehavior.

Another common mistake when using Behaviors is to forget to specify a scheduling volume with setSchedulingBounds(), in a similar way to lights and other environmental nodes. If no volume is set then the Behavior will never be triggered.

TouristControls

The TouristControl object responds to key presses, either by moving the robot sprite, or by changing the user's viewpoint. As the sprite moves, the viewpoint is automatically adjusted so they stay a fixed distance apart. This is a simple form of third person camera.

The TourSprite and TouristControls are created and linked inside addTourist() in WrapTour3D:

```
private void addTourist()
{ bob = new TourSprite("Coolrobo.3ds", obs);   // sprite
  bob.setPosition(2.0, 1.0);
  sceneBG.addChild( bob.getTG() );

  ViewingPlatform vp = su.getViewingPlatform();
  TransformGroup viewerTG = vp.getViewPlatformTransform();
                  // TransformGroup for the user's viewpoint

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

```
    tcs.setSchedulingBounds( bounds );
    sceneBG.addChild( tcs );
  }
```

The TouristControls objects requires a reference to the TourSprite in order to monitor and change its position, and a reference to the user's viewpoint TransformGroup, to move the viewpoint in line with the TourSprite's position. Inside the TouristControl object, these arguments are stored as the globals bob and viewerTG.

The WakeupCondition for TouristControls is a AWT key press, which is specified in the constructor:

```
    keyPress = new WakeupOnAWTEvent( KeyEvent.KEY_PRESSED );
```

It is registered in initialize():

```
    wakeupOn( keyPress );
```

processStimulus() contains code to check that the criterion is indeed a AWT event, to deal with multiple AWT events, and to respond only to key presses:

```
  public void processStimulus(Enumeration criteria)
  { WakeupCriterion wakeup;
    AWTEvent[] event;

    while( criteria.hasMoreElements() ) {
      wakeup = (WakeupCriterion) criteria.nextElement();
      if( wakeup instanceof WakeupOnAWTEvent ) {  // is it AWT?
        event = ((WakeupOnAWTEvent)wakeup).getAWTEvent();
        for( int i = 0; i < event.length; i++ ) { // many events
          if( event[i].getID() == KeyEvent.KEY_PRESSED )
            processKeyEvent((KeyEvent)event[i]);
        }
      }
    }
    wakeupOn( keyPress );  // re-register
  } // end of processStimulus()
```

All the testing and iteration through the event[] array leads to a call to processKeyEvent() which actually does something in response to the key press.

Keys Understood by TouristControls

Our sprite can move in four directions: forwards, backwards, left, and right, and can rotate left or right around the y-axis. The down, up, left, and right arrow keys cover forwards, backwards, rotate left, and rotate right. The <alt> key combined with the left and right arrows support left and right movement.

The viewpoint can be 'zoomed' in and out along the z-axis; those two operations are represented by the 'i' and 'o' keys.

One subtlety here is the choice of keys to denote direction. The 'down' arrow key is most natural for representing 'forward' when the sprite is facing out of the world, along the +z axis, but is less appealing when the sprite has been rotated by 180 degrees and is facing into the scene. For this reason, it may be better to use letter keys such as 'f', 'b', 'l', and 'r'.

processKeyEvent() 's definition:

```
private void processKeyEvent(KeyEvent eventKey)
{ int keyCode = eventKey.getKeyCode();
  if( eventKey.isAltDown() )
    altMove(keyCode);
  else
    standardMove(keyCode);

  viewerMove();
}
```

Every key has a unique key code constant -- they are listed at length in the documentation for the KeyEvent class. Checking for modifier keys, such as <alt> and <shift> can be done by testing the KeyEvent object.

standardMove() calls the relevant methods in the TourSprite (bob) depending on the key pressed. For instance:

```
  if(keycode == forwardKey )
    bob.moveForward();
  else if(keycode == backKey)
    bob.moveBackward();
```

forwardKey and backKey (and others) are constants defined in TouristControls:

```
private final static int forwardKey = KeyEvent.VK_DOWN;
private final static int backKey = KeyEvent.VK_UP;
```

A Third Person Camera

A third person camera is a viewpoint which (semi-)automatically tracks the user's sprite as it moves through a game. This is fairly difficult to automate since the best vantage point for a camera depends not only on the sprite's position and orientation, but also on the location of nearby scenery and other sprites, as well as the focal point for the current action. A common solution is to offer the player a selection of several cameras, which can be switched between as necessary.

Tour3D is considerably simpler: the camera stays at a certain distance from the sprite, offset along the positive z-axis. This distance is maintained as the sprite moves forwards, backwards, left, and right. The only permitted adjustment to the camera is a 'zoom' capability which reduces or increases the offset.

Although this approach is simple, it is quite effective. Also the coding can be readily extended to support more complex changes in the camera' position and orientation.

By coding our own camera, we no longer need the OrbitBehavior class.

Viewpoint Initialisation

The initial positioning of the user's viewpoint is done in TouristControls in the setViewer() method:

```
private void setViewer()
{ bobPosn = bob.getCurrLoc();   // start location for bob
  viewerTG.getTransform( t3d );
  t3d.lookAt( new Point3d(bobPosn.x, HEIGHT, bobPosn.z + ZOFFSET),
              new Point3d(bobPosn.x, HEIGHT, bobPosn.z),
              new Vector3d(0,1,0));
```

```
    t3d.invert();
    viewerTG.setTransform(t3d);
  }
```

lookAt() specifies the viewer's position, the point being looked at, and the up direction. The coordinates are obtained from the TourSprite's original position. The viewpoint is raised HEIGHT units up the y-axis, and ZOFFSET units away down the positive z-axis, to give an overview of the robot.

It is important that the vector between the user's viewpoint and the sprite is at right angles to the XY plane. This means that a translation applied to the sprite will have the same effect when applied to the viewpoint.

This issue is a consequence of both the translation and rotation components of the viewer being applied to a single TransformGroup. We will talk about this more in the "Rotating the Camera" section below.

Moving the Camera

The camera is moved by viewerMove(), which is called at the end of processKeyEvent(), after the sprite's position or orientation have been altered.

viewerMove() obtains the new position of the sprite, and calculates the translation compared to the previous position. This translation is applied to the viewer.

```
  private void viewerMove()
  { Point3d newLoc = bob.getCurrLoc();
    Vector3d trans = new Vector3d( newLoc.x - bobPosn.x,
                                 0, newLoc.z - bobPosn.z);
    viewerTG.getTransform( t3d );
    toMove.setTranslation(trans);
    t3d.mul(toMove);
    viewerTG.setTransform(t3d);
    bobPosn = newLoc;   // save for next time
  }
```

Figure 8 shows two screenshots of Tour3D with the sprite in different locations and orientations, but the viewpoint is in the same relative position in both pictures.

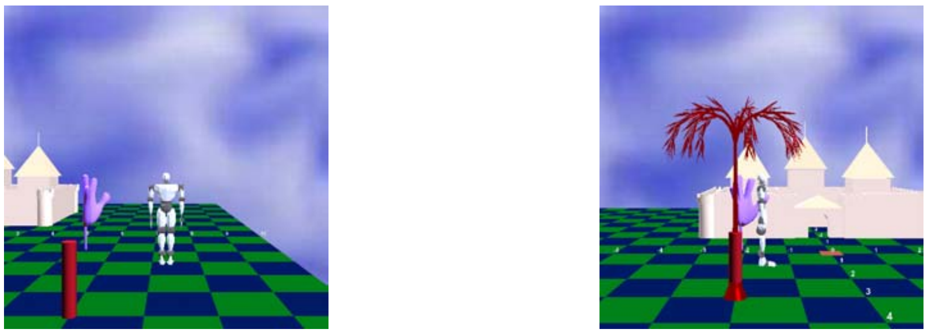


Figure 8. Sprite Moves Leave the Viewpoint Unchanged.

Zooming the Camera

Camera zooming is achieved by adjusting the z-axis distance between the viewpoint and the sprite. When the user types ‘i’ or ‘o’, eventually shiftViewer() is called inside standardMove():

```
       :
    else if(keycode == inKey)   // letter 'i'
      shiftViewer(-ZSTEP);
    else if(keycode == outKey)  // letter 'o'
      shiftViewer(ZSTEP);
```

ZSTEP is set to be 1.0. shiftViewer() moves the TransformGroup by the required amount along the z-axis:

```
  private void shiftViewer(double zDist)
  { Vector3d trans = new Vector3d(0,0,zDist);
    viewerTG.getTransform( t3d );
    toMove.setTranslation(trans);
    t3d.mul(toMove);
    viewerTG.setTransform(t3d);
  }
```

Figure 9 shows the result of pressing 'i' 5 times. Compare the viewpoint's position with the images in Figure 8.

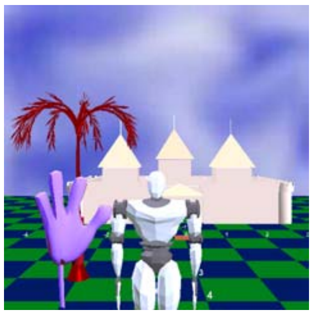


Figure 9. A Closer Viewpoint.

Rotating the Camera

The TouristControls class does not support viewpoint rotation, but it is still interesting to discuss the various issues involved in implementing it.

The first problem is to make sure that the rotations and translations applied to the sprite are echoed by the same rotations and translations of the viewpoint. If the viewpoint rotates by a different amount, then the sprite's translations will have a different effect on the viewpoint since it will be facing in a different direction.

This echoing is best implemented by duplicating the Sprite3D methods for translation and rotation inside TouristControls. The coding will require some modifications since the viewpoint is facing towards the sprite, so its notions of forwards, backwards, left, and right are different.

Even if the rotations of the sprite and viewpoint are always aligned, there are still problems. For instance, a 180 degree rotation of the sprite will cause a 180 degree rotation of the viewpoint, and the viewpoint will now be facing away from the sprite!

This is caused by rotating both the sprite and the viewpoint around their own centers. In fact, the rotation of the viewpoint must not use its own coordinate system, but be relative to the sprite. In other words, a viewpoint rotation must use the sprite's position as its center of rotation.

Figure 10 shows the desired viewpoint rotation after the sprite has just rotated 30 degrees.

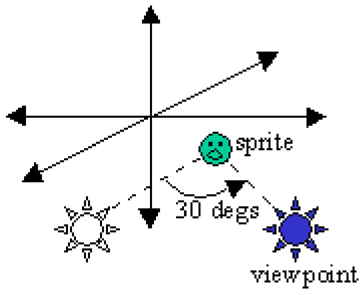


Figure 10. Viewpoint Rotation.

In coding terms, this requires the viewpoint TransformGroup to be translated to the sprite's position, rotated, and then translated back. The translation back will be the negative of the first translation since the viewpoint's coordinate system will have been changed by the rotation.

A more fundamental question still remains – does rotation give the user a better view of the sprite? Unfortunately, the answer is only 'maybe'. The problem is that the rotation may move the viewpoint inside a piece of scenery, which will block the view.

One solution is to offer the user several viewpoints at the same time, in the hope that at least one of them will be useful. We will be implementing multiple viewpoints in the Maze3D application of chapter ??.

TimeBehavior

TimeBehavior acts as a Timer-like object for calling the update() method in AlienSprite every 500 milliseconds. The two are linked together by WrapTour3D in its addAlien() method:

```
  private void addAlien()
  { AlienSprite al =
              new AlienSprite("hand1.obj", obs, bob);   // alien
    al.setPosition(-6.0, -6.0);
    sceneBG.addChild( al.getTG() );
    TimeBehavior alienTimer =
              new TimeBehavior(500, al);  // alien's controls
    alienTimer.setSchedulingBounds( bounds );
    sceneBG.addChild( alienTimer );
  }
```

The TimeBehavior class is significantly simpler than TouristControls since the mere fact of it's processStimulus() method being called is enough to trigger the call to update().

```
public class TimeBehavior extends Behavior
{ private WakeupCondition timeOut;
  private AlienSprite alien;

  public TimeBehavior(int timeDelay, AlienSprite as)
  { alien = as;
```

```
    timeOut = new WakeupOnElapsedTime(timeDelay);
  }

  public void initialize()
  { wakeupOn( timeOut );  }

  public void processStimulus( Enumeration criteria )
  { alien.update();      // ignore criteria
    wakeupOn( timeOut ); // re-register
  }
}
```

The wakeup criterion is an instance of WakeupOnElapsedTime.