Chapter 11. Animated 3D Sprites

In this chapter, we modify the Sprite3D class of chapter 10, correcting one of its main weaknesses: the AnimSprite3D class loads several models, which represent different *poses* for the sprite depending on the action it is carrying out. For instance, as the sprite moves forward, the 'standing' pose is replaced by a 'walking' pose.

The poses are further enhanced by being animated, so that walking is actually represented by a series of walking poses which are displayed one after another.

We call this succession of poses an *animation sequence*. Typically, a user's key press (e.g. down arrow to move forward) is translated into an animation sequence of several sprite poses.

Initially, we describe the application running in a medium-size window but, at the end of the chapter, we show how to use Full-Screen Exclusive Mode (FSEM). We also explain how to modify the display mode for the monitor with FSEM.

Figure 1 shows the sprite (a "stick child") walking. Figure 2 shows the sprite punching.

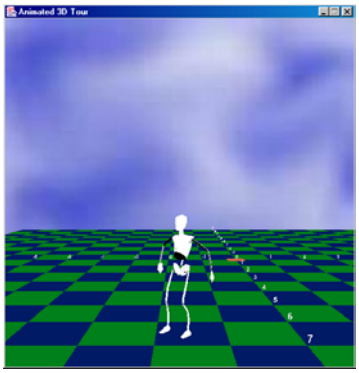

Figure 1. Walking Sprite.

Figure 2. Punching Sprite.

The application does not include a chasing alien sprite, scenery, or obstacles. However, these could be added by using the techniques described in chapter 10.

UML Diagrams

Figure 3 shows the UML diagrams for the classes in the AnimTour3D application. Only the class names are shown.

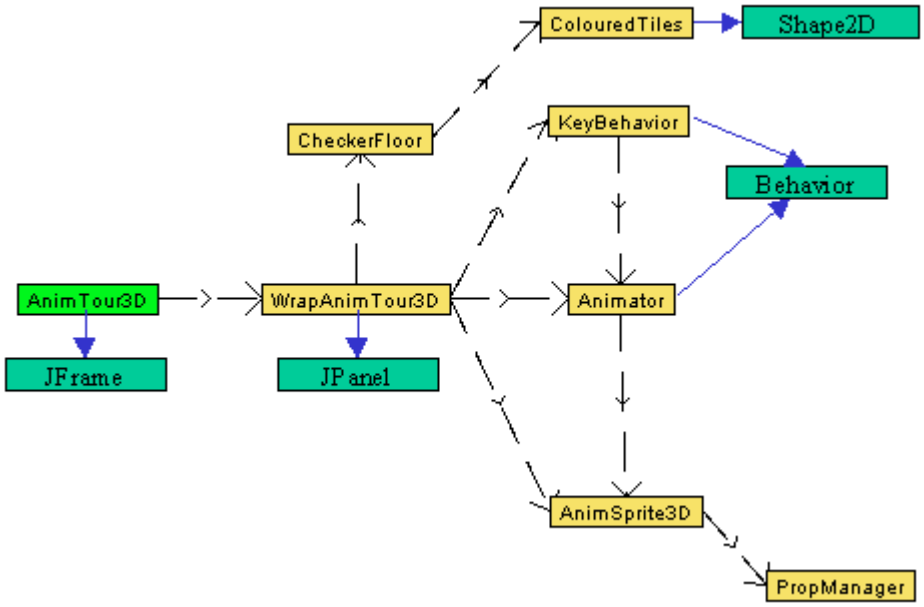

Figure 3. UML Class Diagrams for AnimTour3D.

AnimTour3D is the top-level JFrame for the application.

WrapAnimTour3D creates the 3D world, and is similar in many respects to the earlier 'Wrap' classes in that it creates the checkered floor and lighting. It also loads the "stick child" sprite and sets up its controlling behaviours.

PropManager is unchanged from the code in the Loader3D application of chapter 9.

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

KeyBehavior and Animator are subclasses of Behavior. KeyBehavior is triggered by key presses, like those used in Tour3D in chapter 10. It responds by requesting that the Animator object adds animation sequences to its animation schedule. Animator wakes up periodically and processes the next animation in its schedule, thereby altering the sprite.

The code for this application is in /Code/AnimTour3D.

WrapAnimTour3D: Creating the World

Figure 4 shows all the methods in WrapAnimTour3D.

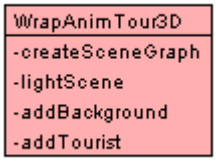

Figure 4. WrapAnimTour3D Methods.

The code is virtually identical to earlier 'wrap' classes apart from the contents of addTourist(), which sets up the sprite and the KeyBehavior and Animator objects, and links them as shown in the UML diagrams of Figure 3.

```
private void addTourist()
{ // sprite
  AnimSprite3D bob = new AnimSprite3D();
  bob.setPosition(2.0, 1.0);
  sceneBG.addChild( bob.getTG() );

  // viewpoint TG
  ViewingPlatform vp = su.getViewingPlatform();
  TransformGroup viewerTG = vp.getViewPlatformTransform();

  // sprite's animator
  Animator animBeh = new Animator(20, bob, viewerTG);
  animBeh.setSchedulingBounds( bounds );
  sceneBG.addChild( animBeh );

  // sprite's input keys
  KeyBehavior kb = new KeyBehavior(animBeh);
  kb.setSchedulingBounds( bounds );
  sceneBG.addChild( kb );
} // end of addTourist()
```

The AnimSprite3D object is responsible for loading the multiple models that represent the sprite's various poses.

Animator is in charge of adjusting the user's viewpoint, and so is passed the ViewingPlatform's TransformGroup node. We set the Animator to wake up every 20ms, through the first argument of its constructor.

The scene graph for AnimTour3D is shown in Figure 5. Since we have not bothered with an alien, scenery, or obstacles, it is simpler than the scene graph for Tour3D. The subgraph for the user's sprite, and the two behaviours will be explained in subsequent sections.

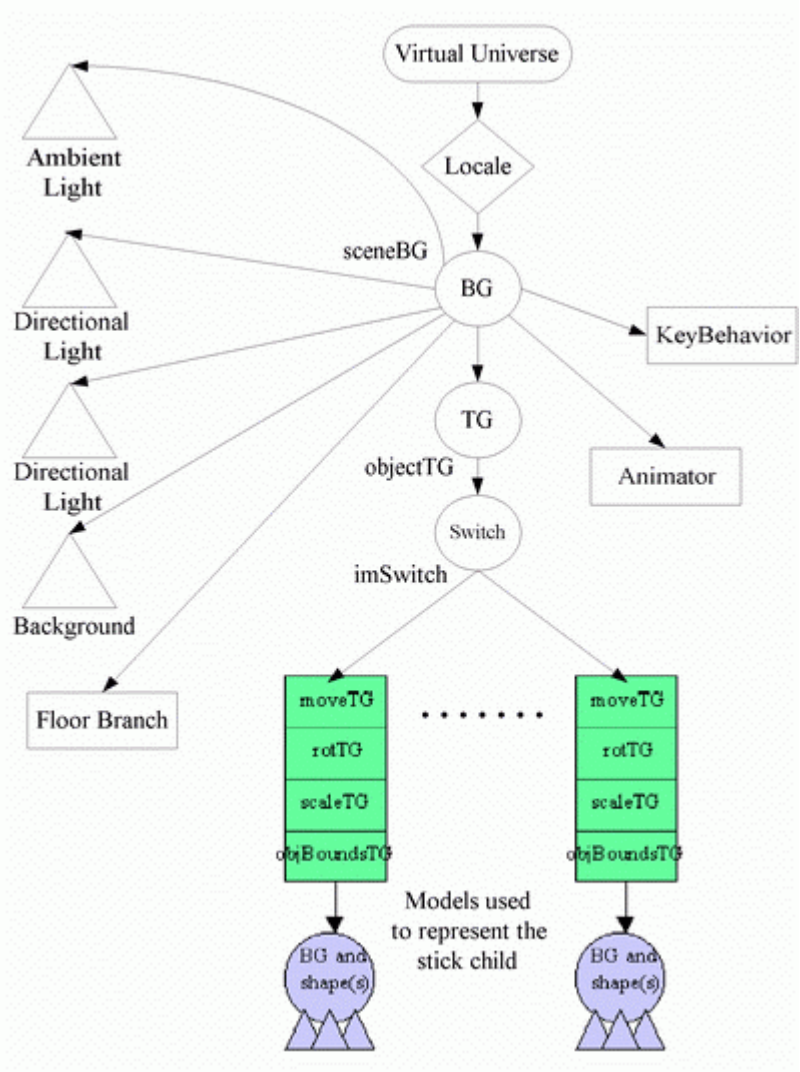

Figure 5. Scene Graph for the AnimTour3D World.

AnimSprite3D

Figure 6 shows the visible methods of AnimSprite3D.

Figure 6. The Visible Methods of AnimSprite3D.

The interface is almost identical to Sprite3D in Tour3D. The setPosition(), moveBy(), and doRotateY() operations adjust the position and orientation of the sprite, isActive() and setActive() relate to its activity (i.e. whether it is visible on-screen or not), getCurrLoc() returns the sprite's position, and getTG() returns its top-level TransformGroup.

The only new method is setPose() which takes a pose name as an argument and changes the displayed model accordingly.

Loading the Pose Models

We have simplified matters a little by hardwiring the choice of models inside AnimSprite3D. The models which will be loaded are predefined in the poses[] array:

```
private final static String poses[] =
       {"stand", "walk1", "walk2", "rev1", "rev2", "rotClock",
        "rotCC", "mleft", "mright", "punch1", "punch2"};
```

poses[] is used by loadPoses() which loads each 3ds file using PropManager, and attaches them below a Switch node:

```
private void loadPoses()
{ PropManager propMan;

  imSwitch = new Switch(Switch.CHILD_MASK);
  imSwitch.setCapability(Switch.ALLOW_SWITCH_WRITE);

  maxPoses = poses.length;
  for (int i=0; i < maxPoses; i++) {
    propMan = new PropManager(poses[i] + ".3ds", true);
    imSwitch.addChild( propMan.getTG() );   // add obj to switch
  }

  visIms = new BitSet( maxPoses );    // bitset used for switching
  currPoseNo = STAND_NUM;  // sprite standing still
  setPoseNum( currPoseNo );
}
```


The resulting Switch node (imSwitch) is shown in the scene graph of Figure 5. It is a child of the objectTG TransformGroup, which is the used for positioning and rotating the sprite.

imSwitch is created with the CHILD_MASK value, which means that its children can be switched between by using a BitSet object (visIms). The bits of the BitSet are mapped to the children of the Switch: bit 0 corresponds to child 0, bit 1 to child 1, and so on. BitSet offers various methods for clearing bits and setting them.

When the visIms BitSet object has been suitably manipulated, the Switch is adjusted by a call to setChildMask():

```
imSwitch.setChildMask(visIms);
```

The bit manipulation is hidden inside the setPoseNum() method, which takes as input the bit index which should be turned on in imSwitch:

```
private void setPoseNum(int idx)
{ visIms.clear();
  visIms.set( idx );       // switch on new pose
  imSwitch.setChildMask( visIms );
  currPoseNo = idx;
}
```

The runtime adjustment of the Switch requires its write capability to be turned on.

Where do the Models come from?

The models were created using the Poser application (http://www.curiouslabs.com). Poser specialises in 3D figure creation and animation, and includes a range of predefined models, poses, and animation sequences. Poser fans should also check out the collection of links in the Google directory http://directory.google.com/Top/Computers/Software/Graphics/3D/Animation_and_Design_Tools/Poser/.

I used one of Poser's existing figures, the "stick child", and exported different versions of it in various standard poses to 3ds formatted files. I did not use animation sequences: each file only contains a single figure.

I loaded the models into the Loader3D application of chapter 9 to adjust their position and orientation. I discovered that Poser exports 3ds models orientated with the XZ plane as their base, pointing up the positive y-axis. In other words, the default rotation carried out by PropManager actually rotates a model into a prone position on its back. I had to undo that rotation in the "coord" data files. This suggests that PropManager should be modified to make the 3ds rotation an option.

Some of the models required slight repositioning, which was due to my inexperienced usage of Poser. With more care, it should be unnecessary to associate "coords" data files with the models.

Each of the 3ds files is about 20K in size, due to my choice of a very simple model.

Setting a Pose

The visible method for adjusting a pose is setPose() which takes a pose name as its input argument. It then determines the index position of that name in the poses[] array, and calls setPoseNum():

```
public boolean setPose(String name)
 { if (isActive()) {
    int idx = getPoseIndex(name);
    if ((idx < 0) || (idx > maxPoses-1))
      return false;
    setPoseNum( idx );
    return true;
  }
  else
    return false;
}
```

The code is complicated slightly by the need to check for sprite activity. If the sprite is inactive then no changes should be made. The same test is also present in moveBy() and doRotateY() which affect the sprite.

The use of a name as the setPose() argument means that the caller must know the pose names used in poses[].

Sprite Activity

Sprite activity can be toggled on and off by calls to setActive() with a suitable boolean argument:

```
public void setActive(boolean b)
{ isActive = b;
  if (!isActive) {
    visIms.clear();
    imSwitch.setChildMask( visIms );   // display nothing
  }
  else if (isActive)
    setPoseNum( currPoseNo );   // make visible
}
```

This approach requires a global integer, currPoseNo, which records the index of the current pose. It is used to make the sprite visible again after a period of inactivity.

Floor Boundary Detection

The movement and rotation methods are essentially unchanged from Sprite3D, except in the case of moveBy() which must try a move before carrying it out. Our decision not to use obstacles, means that there is no Obstacle object available for checking if the sprite is about to move off the floor.

This is remedied by a beyondEdge() method which determines if the sprite's x or z coordinate is outside the limits of the floor:

```
public boolean moveBy(double x, double z)
// move the sprite by an (x,z) offset
{ if (isActive()) {
```



```
      Point3d nextLoc = tryMove( new Vector3d(x, 0, z));
      if (beyondEdge(nextLoc.x) || beyondEdge(nextLoc.z))
        return false;
      else {
         :
  }  // end of moveBy()

  private boolean beyondEdge(double pos)
  { if ((pos < -FLOOR_LEN/2) || (pos > FLOOR_LEN/2))
      return true;
    return false;
  }
```

KeyBehavior

KeyBehavior can be considered a version of the TouristControls class of chapter 10 with all the key *processing* code removed. All that remains is the testing of the key press to decide which method of the Animator class to call. This can be seen by looking at the standardMove() and altMove() methods. For example:

```
private void standardMove(int keycode)
  { if(keycode == forwardKey )
      animBeh.moveForward();
    else if...  // more lines
      :
    else if(keycode == activeKey)
      animBeh.toggleActive();
    else if(keycode == punchKey)
      animBeh.punch();
    else if(keycode == inKey)
      animBeh.shiftInViewer();
    else if(keycode == outKey)
      animBeh.shiftOutViewer();
  }
```

The activeKey is new – it is the letter 'a' for toggling the sprite's activity. The punchKey is also new – 'p' sets the sprite into punching pose, as illustrated in Figure 2. Also note that viewer manipulation has been moved to the Animator object.

Animator

Figure 7 shows the visible methods in Animator.

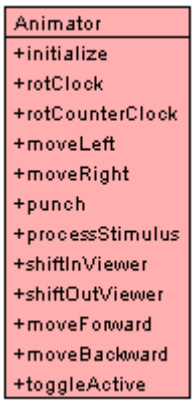

Figure 7. Animator Visible Methods.

Animator performs three main tasks:

1. The addition of animation sequences to its schedule in response to calls from the KeyBehavior object.
2. The removal of an animation operation from the schedule and its execution. The execution typically changes the sprite's position and pose. The removal is carried out periodically, triggered by the arrival of a WakeupOnElapsedTime event.
3. The updating of the user's viewpoint in response to calls from KeyBehavior.

Adding an Animation Sequence

The majority of the visible methods (rotClock(), rotCounterClock(), moveLeft(), moveRight(), punch(), moveForward(), moveBackwards(), and toggleActive()) are all processed in the same way when called by the KeyBehavior object. The response is to add a predefined animation sequence to a schedule (an ArrayList called animSchedule). For example:

```
public void moveForward()
{ addAnims(forwards); }
```

forwards is an array of strings, which represents the animation sequence for moving the sprite forwards:

```
private final static String forwards[] = {"walk1", "walk2", "stand"};
```

"walk1", "walk2", and so on , are also the names of the 3ds files holding the sprite's poses. Another requirement of an animation sequence is that it ends with "stand"; this property is used by the code to detect the end of a sequence.

addAnims() adds the sequence to the end of animSchedule, and increments seqCount, which stores the number of sequences currently in the schedule.

```
synchronized private void addAnims(String ims[])
{ if (seqCount < MAX_SEQS) {  // not too many animation sequences
    for(int i=0; i < ims.length; i++)
      animSchedule.add(ims[i]);
    seqCount++;
  }
}
```

The maximum number of sequences in the schedule at any time is restricted to MAX_SEQS. This ensures that a key press (or, equivalently, an animation sequence) is not kept waiting too long in the schedule before being processed. This would happen if the user pressed a key continuously, causing a very long schedule to form. By limiting the number of sequences, we essentially ignore repeated key presses after a certain number.

addAnims() is synchronized so that it is impossible for the animSchedule to be accessed while being extended. This problem could occur when Animator is triggered by a WakeupOnElapsedTime event into removing an animation operation from animSchedule.

Processing an Animation Operation

The Animator constructor creates a WakeupCondition object based on the time delay passed to it from outside:

```
public Animator(int td, AnimSprite3D b, TransformGroup vTG)
{    timeDelay = new WakeupOnElapsedTime(td);
          :
```

This condition is registered in initialize() so that processStimulus() will be called every td ms:

```
public void processStimulus( Enumeration criteria )
{ // ignore criteria
  String anim = getNextAnim();
  if (anim != null)
    doAnimation(anim);
  wakeupOn( timeDelay );
}
```

getNextAnim() tries to remove an animation operation from animSchedule. However, the ArrayList may be empty, so the method may return null.

```
synchronized private String getNextAnim()
{ if (animSchedule.isEmpty())
    return null;
  else {
    String anim = (String) animSchedule.remove(0);
    if (anim.equals("stand"))   // end of a sequence
       seqCount--;
    return anim;
  }
}
```

getNextAnim() is synchronized to enforce mutual exclusion on animSchedule. Also, if the retrieved operation is “stand” than the end of an animation sequence has been reached, and seqCount is decremented.

doAnimation() can process an animation operation (a String) in two ways: the operation may trigger a transformation in the user’s sprite (called bob), and/or it could cause a change to the sprite’s pose. In addition, it may be necessary to update the user’s viewpoint if the sprite has moved in the world.

```
private void doAnimation(String anim)
{ /* Carry out a transformation on the sprite.
     Note: "stand", "punch1", "punch2" have no transforms
  */
  if ( anim.equals("walk1") || anim.equals("walk2"))   // forward
    bob.moveBy(0.0, MOVERATE/2);      // half a step
  else if ( anim.equals("rev1") || anim.equals("rev2"))  // back
    bob.moveBy(0.0, -MOVERATE/2);     // half a step
  else if (anim.equals("rotClock"))
    bob.doRotateY(-ROTATE_AMT);       // clockwise rot
  else if (anim.equals("rotCC"))
    bob.doRotateY(ROTATE_AMT);        // counter-clockwise rot
  else if (anim.equals("mleft"))      // move left
    bob.moveBy(-MOVERATE,0.0);
  else if (anim.equals("mright"))     // move right
    bob.moveBy(MOVERATE,0.0);
  else if (anim.equals("toggle")) {
```



```
      isActive = !isActive;    // toggle activity
      bob.setActive(isActive);
    }

    // update the sprite's pose, except for "toggle"
    if (!anim.equals("toggle"))
      bob.setPose(anim);

    viewerMove();    // update the user's viewpoint
  }  // end of doAnimation()
```

The first part of doAnimation() specifies how an animation operation is translated into a sprite transformation. One trick is shown in the processing of forward and backwards sequences. These sequences are:

```
private final static String forwards[] = {"walk1","walk2","stand"};
private final static String backwards[] = {"rev1","rev2","stand"};
```

The forwards sequence is carried out in response to the user pressing the down arrow. What exactly happens? The sequences is made up of three poses ("walk1", "walk2", and "stand"), and so the sequence will be spread over 3 activations of processStimulus (). This means that the total sequence will take 3*<time delay> to be evaluated, which is about 60ms. However, the last animation operation is "stand", so the walking is completed after 40ms.

In chapter 10, a sprite was moved by MOVERATE units on each key press. Since walking is split into two parts in Animator, doAnimation() requests that each walk operation be represented by a move of MOVERATE/2 units.

The same reasoning applies to the processing of the backwards[] animation sequence.

The punching animation sequence is:

```
private final static String punch[] =
          {"punch1", "punch1", "punch2", "punch2", "stand"};
```

Since "punch1" and "punch2" both appear twice, they will be processed twice by processStimulus(), which means their effects will last for 2*<time delay>. Consequently, the poses will be on-screen for double the normal time, suggesting that the sprite is holding its stance.

Animator is hardwired in the sense that we assume that the animation operations will be understood as poses by the sprite. This linkage is apparent when doAnimation() calls setPose() in the sprite:

```
    if (!anim.equals("toggle"))
      bob.setPose(anim);
```

Updating the User's Viewpoint

Animator reuses the viewpoint manipulation code which we first developed in TouristControls in chapter 10.

The initial viewpoint is set up in Animator's constructor via a call to setViewer(), which is the same method as found in TouristControls.

The viewpoint is updated by a call to viewerMove() at the end of doAnimation(). The call is located there since this is *when* an animation is carried out. It is not possible to do the update in KeyBehavior since the triggered animation will probably be delayed for a (short) time inside animSchedule. The viewpoint change must be delayed until the animation is actually executed.

The final aspect of viewpoint adjustment are the keys 'i' and 'o' which are used to directly change the viewpoint. The subtle issue here is the relative timing of these operations compared to other user operations, such as moving the sprite with the down arrow key.

Our approach is to immediately process the keys by calls to shiftViewer() which changes the viewpoint based on the sprite's current position.

```
public void shiftInViewer()
{  shiftViewer(-ZSTEP); }   // move viewer negatively on z-axis

public void shiftOutViewer()
{  shiftViewer(ZSTEP); }

private void shiftViewer(double zDist)
// move the viewer inwards or outwards
{ Vector3d trans = new Vector3d(0,0,zDist);
  viewerTG.getTransform( t3d );
  toMove.setTranslation(trans);
  t3d.mul(toMove);
  viewerTG.setTransform(t3d);
}
```

The key phrase is "immediately process", which means that the shift operations are not scheduled like the sprite movement commands. This has the effect that a viewpoint changes immediately even if a large number of movement key presses precede the 'i' or 'o'.

Pros and Cons of Our Animation Approach

AnimTour3D uses two animation techniques: switching between preloaded models (poses), and the translation/rotation of the model's top-level TransformGroup. The use of the TransformGroup is quite standard, so the real question is about the advantages and disadvantages of switching poses.

The advantage of using models is that they can be designed and created outside of Java 3D with software specific to the task (I used Poser). In fact, the model creation process can be quite separate from the implementation of the game, and can be assigned to someone skilled in 3D modeling.

A major drawback is the potential size of each model, and the number of models required to cover the necessary poses. I chose the simplest figure I could find (size 20K), and a small number of poses (13), coming to a very reasonable 260K, which must be loaded at start-up. These numbers could quickly become unmanageable with larger models and more poses, but that depends on the application.

One means of reducing the memory requirements is to share shapes between sprites (by using SharedGroup nodes), which would be useful for groups of related figures such as soldiers.

Another inherent inefficiency is that the models all contain much the same geometry, which may hardly ever be altered. For example, the head and torso of the stick figure barely change from one pose to another, but each model requires its own head and body.

Other Animation Techniques

The principal coding alternative is to create an articulated figure. The figure would contain TransformGroup nodes at all the main 'joint's (e.g. at the shoulders, wrists, knees). These joints can be moved using Java 3D transforms in the same way that the entire figure is moved and rotated in AnimTour3D.

Advantages include the increase in control over the figure, and the reduction in memory requirements since only a single model is being manipulated.

A disadvantage is that the model will probably be a Java 3D creature of cylinders, spheres, and blocks, which must be 'dressed' in some way. This will usually necessitate the loading of 3D models, which may bring back the problems we were trying to avoid.

Another issue is the increased complexity of the control code, which will require some mechanism for coordinating the movement of numerous TransformGroups; for instance, a single step forward will affect many joints, and the exact changes needed are far from obvious.

This technique is widespread in 3D computer graphics, and known as *forward kinematics*.

There are alternatives to articulated cylinder beings. Another approach is to create a complex shape using a GeometryArray, but store the relevant data in an array maintained by the program, accessed from Java 3D by reference. The advantage is that the data array can be modified inside your code, and the on-screen representation can then be updated by a call to GeometryArray's updateData() method.

There is an example of this technique in the Java 3D demos, in the directory /GeometryByReference, showing how the shape and colour of an object can be changed. We discuss geometry referencing in the next chapter, in relation to particle systems.

Another animation possibility is *morphing*, which allows a smooth transition between different GeometryArrays. This is done using a Morph node, set up in a similar way to a Switch to access its child geometries, but with weights assigned to the shapes. The Morph node combines the geometries into a single aggregate shape based on each GeometryArray's corresponding weight. Typically, Behavior nodes modify the weights to achieve various morphing effects.

The drawback is that the various objects must have very similar structures (e.g. the same number of coordinates). Also, it is debatable whether morphing is required for

rapidly changing poses, since our eyes tend to deceive us by filling in the gaps themselves.

There is a morphing example in the Java 3D demos, in the directory /Morphing, illustrating how a hand can be made to slowly wave by morphing between three obj files. Morphing is also discussed in chapter 5 of the Java 3D tutorial where a stick figure is animated.

Full Screen Exclusive Mode

Full Screen Exclusive Mode (FSEM) was introduced in J2SDK 1.4, as a way of switching off Java's windowing system and allowing direct drawing to the screen. The principal reason for this is speed, an obvious advantage for games. Secondary benefits include the ability to control the bit depth and size of the screen (its display mode). Also, advanced graphics techniques, such as page flipping and stereo buffering, are often only supported by display cards when FSEM is enabled.

We discussed FSEM in the context of 2D games in chapter ??, when we looked at active rendering and page flipping. The code described here uses passive rendering, which means that Java 3D is left in control of when to render to the screen. Active rendering is possible, by switching to Java 3D's immediate mode.

There is an excellent Java tutorial on FSEM, in <JAVAHOME>\tutorial\extra\fullscreen\ (assuming you've downloaded the tutorial), which includes good overviews of topics like passive/active rendering, page flipping, and display modes. There are also a few example, including the useful CapabilitiesTest.java program which allows you to test your machine for FSEM support.

The AnimTourFS Application

The AnimTourFS application is essentially AnimTour3D, but with the original AnimTour3D and WrapAnimTour3D classes replaced by a single new class, AnimTourFS. The rest of the classes unchanged, as Figure 8 suggests.

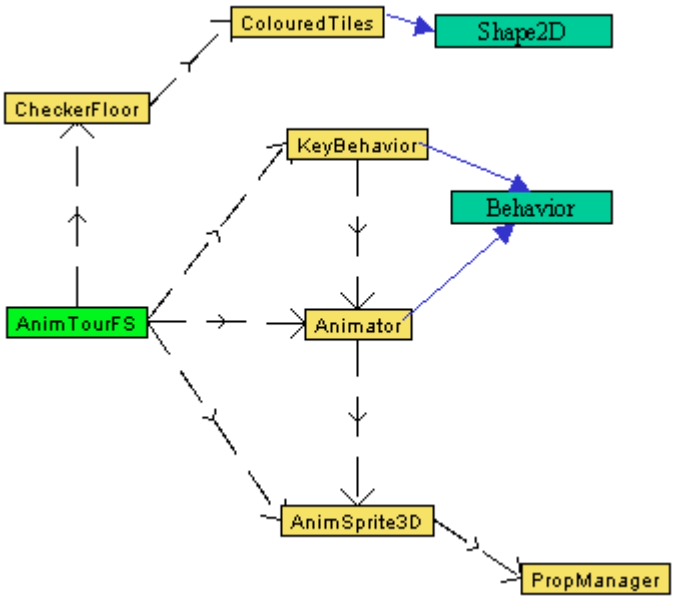

Figure 8. UML Class Diagrams for AnimTourFS.

Also, the invocation of the application must include the option `-Dsun.java2d.noddraw=true`:

```
java -cp %CLASSPATH%;ncsa\portfolio.jar
                        -Dsun.java2d.noddraw=true AnimTourFS
```

The "nodraw" property switches off the use of Window's DirectDraw for drawing AWT elements and offscreen surfaces.

The AnimTourFS Class

Figure 9 shows all the methods in the AnimTourFS class, and should be compared with Figure 4 which lists the methods in the old WrapAnimTour3D class.

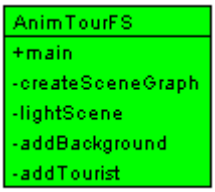

Figure 9. Methods in AnimTourFS.

The only apparent change is the main() method; all the other changes are in the AnimTourFS() constructor (with some new private global variables).

FSEM doesn't work well with Swing components, and so AnimTourFS uses a Frame object as the top-level window, and embeds the Canvas3D GUI inside it:

```
private Frame win;   // required at quit time
      :
public AnimTourFS()
{
  GraphicsConfiguration config =
          SimpleUniverse.getPreferredConfiguration();

  win = new Frame("AnimTourFS", config);   // use SU's preference
  win.setUndecorated(true) ;
            // no menu bar, borders, etc. or Swing components
  win.setResizable(false);    // fixed size display
            :
  Canvas3D canvas3D = new Canvas3D(config);    // setup canvas3D
  win.add(canvas3D);
            :
```

We take the opportunity to specify the graphics configuration of the Frame to be the one preferred by Java 3D. FSEM also likes a fixed size window with no decoration.

FSEM is handled through a GraphicsDevice object representing the screen, which is accessed via a GraphicsEnvironment reference. There may be several GraphicsDevice objects for a machine if, for example, it uses dual monitors. However, for the normal single-monitor system, the getDefaultScreenDevice() method is sufficient:

```
private GraphicsDevice gd;
   :
GraphicsEnvironment ge =
    GraphicsEnvironment.getLocalGraphicsEnvironment();
gd = ge.getDefaultScreenDevice();
```

Once the GraphicsDevice object (gd) has been obtained, it is a good idea to check if FSEM is supported by the OS before attempting to use it:

```
if (!gd.isFullScreenSupported()) {
  System.out.println("FSEM not supported.") ;
  System.out.println("Device = " + gd) ;
  System.exit(0) ;
}
```



```
        :
gd.setFullScreenWindow(win); // set FSEM
if (gd.getFullScreenWindow() == null)
  System.out.println("Did not get FSEM");
else
  System.out.println("Got FSEM") ;
```

If setFullScreenWindow() fails then the window is positioned at (0,0) and resized to fit the screen, so that the application can continue without being overly affected.

The final point is to switch off FSEM when the program terminates, which we do as part of the response to a 'quit' key being pressed:

```
canvas3D.addKeyListener( new KeyAdapter() {
public void keyPressed(KeyEvent e)
{ int keyCode = e.getKeyCode();
  if ((keyCode == KeyEvent.VK_ESCAPE) || ...) {
    gd.setFullScreenWindow(null);  // exit FSEM
    win.dispose();
    System.exit(0);
  }
 }
});
```

Figure 10 shows the FSEM in operation (note the absence of a window frame or other decoration).

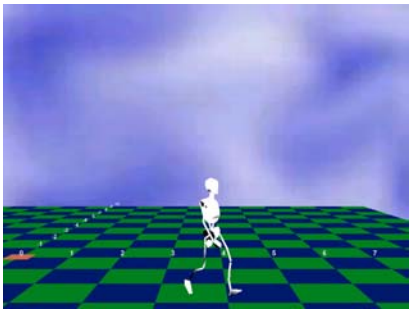

Figure 10. AnimTourFS in Action.

Changing the Display Mode

Once an application is in FSEM, further performance gains may be available by adjusting the screen's display mode, typically the bit depth (number of bits per pixel), and refresh rate (how frequently the monitor updates itself). Reducing the bit depth will increase rendering speed, but there may be an impact on textures and other images in the scene.

A key thing to do when changing the display mode is to store the original version so that it can be restored at the end of execution. A related issue is for the program to recover if the change to the display mode fails.

The Java FSEM tutorial includes a section on display modes, and an example showing how to use them.

In AnimTourFS, we change the display mode after FSEM has been initiated:

```
private DisplayMode origDM = null;
   :
```



```
if (gd.isDisplayChangeSupported()) {
    origDM = gd.getDisplayMode();
    gd.setDisplayMode(
         new DisplayMode( origDM.getWidth(), origDM.getHeight(),
                  origDM.getBitDepth()/2,
                  origDM.getRefreshRate()  ));
  }
```

The code checks if display changing is supported via a call to isDisplayChangeSupported(), stores the original display mode, and updates the mode using the original screen dimensions and refresh rate, but halving the bit depth. This reduces the bit depth of my machine from 32 to 16 bits.

The original display mode is restored at the end of the program by two extra lines in the 'quit' key listener:

```
if ((keyCode == KeyEvent.VK_ESCAPE) || ....) {
   if (origDM != null)  // original was saved
     gd.setDisplayMode(origDM);
   gd.setFullScreenWindow(null);
   win.dispose();
   System.exit(0);
}
```

Figure 11 shows AnimTourFS with its reduced bit depth.

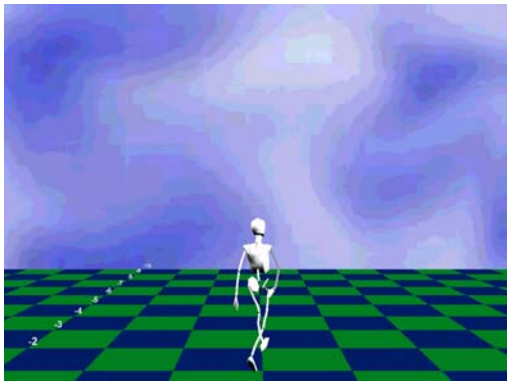

Figure 11. Reduced Bit Depth AnimTourFS.

The 24-bit JPEG used for the background has become rather pixilated, as you would expect with only 16 bits per pixel.

Speed Gains?

One of the main selling points of FSEM, and display mode adjustment, is to improve performance. I used FRAPS (http://www.fraps.com) to measure the frames/second rendering speed of three versions of the AnimTour3D application: the results are shown in Table 1.

	Frames/Second Rendering
AnimTour3D (in a window)	10
AnimTourFS (using FSEM; 32 bit depth)	11
AnimTourFS (16 bit depth)	13

Table 1. Frames/sec Rendering for Different Versions of AnimTour3D.

The rendering speed of the application increases, which is especially pleasing when moving to full screen rendering, but I actually expected better overall performance.

The poor rendering rate is probably due to the keyboard-based nature of the application. The rates shown in the table were only achieved when I continuously held down a key to get rapid updates to the sprite. The rendering slowness is due to the inherent slowness of passing key presses from the keyboard into Java 3D.

Animator is running with a time-interval of 20ms (potentially 50 executions per second), so is not the bottleneck.

