Chapter 14. Shooting a Gun

The application in this chapter, Shooter3D, contains a gun (actually a cone mounted on a cylinder) which fires a laser beam at the point on the checkered floor clicked on by the user. The flight of the laser beam (actually a red cylinder) is accompanied by a suitable sound, and followed by an explosion (an animated series of images and another sound).

Figure 1 shows three screenshots of Shooter3D. The first one has the laser beam in mid-flight, the second captures the explosion, and the third is another explosion after the user has clicked on a different part of the floor, from a different viewpoint.

Note how the cone head rotates to aim at the target point. Also, the animated explosion always faces the user's viewpoint.

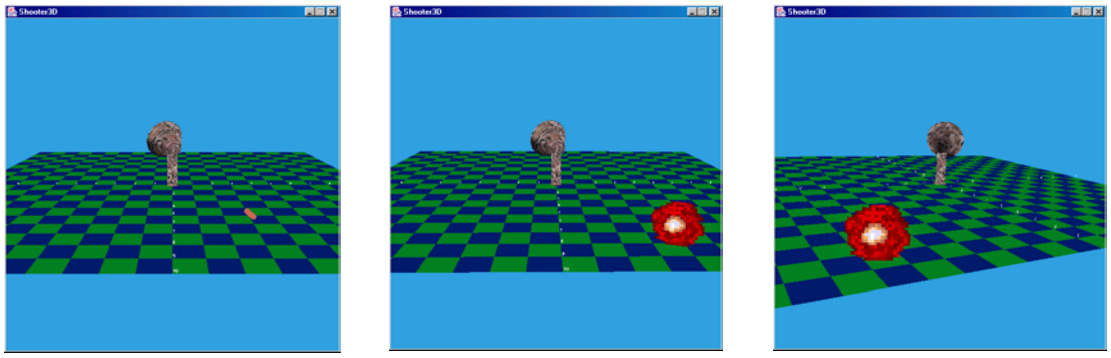

Figure 1. The Deadly Shooter3D.

Java 3D and Java features illustrated by this example:

- the user's clicking on the floor is dealt with by Java 3D picking;
- the laser beam and explosion sounds are PointSound objects;
- the rotations of the cone and the laser beam are handled by AxisAngle4d objects;
- the explosion visual is created with our ImagesSeries class, which simplifies to the loading and displaying of a sequence of transparent GIFs as an animation;
- the delivery of the laser beam, and subsequent explosion, are managed by a FireBeam thread, showing how Java threads and the built-in threading of Java 3D can co-exist;
- the overall complexity of this application is greatly reduced by using OO design principles – each of the main entities (the gun, the laser beam, the explosion) are represented by its own class.

1. UML Diagrams for Shooter3D

Figure 2 gives the UML diagrams for all the classes in the Shooter3D application. The class names and public methods are shown.

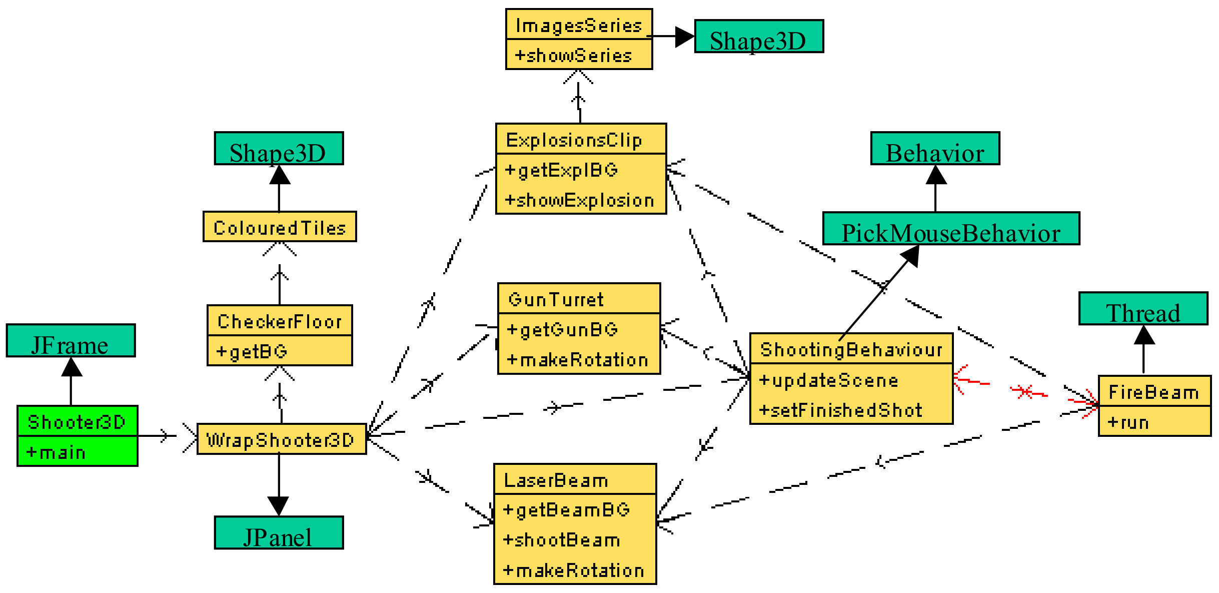

Figure 2. UML Class Diagrams for Shooter3D.

Shooter3D is the top-level JFrame for the application, and very similar to our earlier JFrame classes.

WrapShooter3D creates the 3D world as usual, and adds the gun, laser beam and explosion, and their controlling behavior.

CheckerFloor and ColouredTiles manage the checkboard, but are slightly changed to deal with floor picking.

ExplosionsClip represents the explosion, GunTurret represents the gun, and LaserBeam the laser beam. ExplosionsClip uses ImagesSeries to animate the explosion.

ShootingBehaviour contains the behavior triggered when the user clicks on the floor. Its picking capabilities come from being a subclass of PickMouseBehavior, a utility class in Java 3D, which is itself a subclass of Behavior.

The tasks of firing the laser beam and triggering the explosion are delegated to the FireBeam thread.

2. WrapShooter3D

WrapShooter3D's createSceneGraph() calls makeGun() to initialise the various elements of the application.

```
private void makeGun(Canvas3D canvas3D)
{ // starting vector for the gun cone and beam
  Vector3d startVec = new Vector3d(0, 2, 0);

  // the gun
  GunTurret gun = new GunTurret(startVec);
  sceneBG.addChild( gun.getGunBG() );

  // explosion and sound
  PointSound explPS = initSound("Explo1.wav");
  ExplosionsClip expl = new ExplosionsClip( startVec, explPS);
  sceneBG.addChild( expl.getExplBG() );

  // laser beam and sound
  PointSound beamPS = initSound("laser2.wav");
  LaserBeam laser = new LaserBeam( startVec, beamPS);
  sceneBG.addChild( laser.getBeamBG() );

  // the behaviour that controls the shooting
  ShootingBehaviour shootBeh =
      new ShootingBehaviour(canvas3D, sceneBG, bounds,
                           new Point3d(0,2,0), expl, laser, gun );
  sceneBG.addChild(shootBeh);
} // end of makeGun()
```

The position vector of the gun cone is hard-wired to be (0,2,0). The same vector is also used to place the laser beam (a red cylinder) inside the cone.

3. The Sound of Shooting

There are three kinds of sound nodes in Java 3D: a BackgroundSound object allows a sound to permeate the entire scene, located at no particular place, a PointSound object has a location, and so its volume varies as the user moves away from it (or the sound node moves away from the user), and a ConeSound can be focused in a particular direction. All three are subclasses of the Sound class.

Before sound nodes can be added to a scene, an audio device must be created and linked to the Viewer object. This is quite simple if the SimpleUniverse class is being used (as in Shooter3D):

```
AudioDevice audioDev = su.getViewer().createAudioDevice();
```

This line of code appears in the WrapShooter3D constructor.

WrapShooter3D uses initSound() to load a WAV sound file and create a PointSound object.

```
  private PointSound initSound(String filename)
  { MediaContainer soundMC = null;
    try {
      soundMC = new MediaContainer("file:sounds/" + filename);
      soundMC.setCacheEnable(true);   // load sound into container
    }
    catch (Exception ex)
    {  System.out.println(ex); }

    // create a point sound
    PointSound ps = new PointSound();
    ps.setSchedulingBounds( bounds );
    ps.setSoundData( soundMC );

    ps.setInitialGain(1.0f);  // full on sound from the start

    // allow sound to be switched on/off & its position to be moved
    ps.setCapability(PointSound.ALLOW_ENABLE_WRITE);
    ps.setCapability(PointSound.ALLOW_POSITION_WRITE);

    System.out.println("PointSound created from sounds/" + filename);
    return ps;
  } // end of initSound()
```

A Sound node needs a sound source, which is loaded with a MediaContainer object. Loading can be done from a URL, local file, or input stream; possible failure means that the loading must be wrapped in a try-catch block. initSound() loads its sound from a local file in the subdirectory sounds/.

All Sound nodes must be given a bounding region, and assigned a sound source:

```
   PointSound ps = new PointSound();
   ps.setSchedulingBounds( bounds );
   ps.setSoundData( soundMC );
```

In order to play, the sound node must be enabled with setEnable(). However, initSound() does not call setEnable(), and so the sound isn't played when first loaded. Instead the node's capability bits are set to allow it to be enabled and disabled during execution:

```
    ps.setCapability(PointSound.ALLOW_ENABLE_WRITE);
```

The explosion sound will be positioned at runtime, requiring another capability bit:

```
    ps.setCapability(PointSound.ALLOW_POSITION_WRITE);
```

Other sound elements include setting the volume, and saying whether the sound should loop (and if so, how many times). The relevant methods are:

```
  void setInitialGain(float volume);
  void setLoop(int loopTimes);
```

initSound() sets the volume to 1.0f (full on), and uses the default looping behaviour (play once, finish).

PointSound nodes have a location in space, given by setPosition(). They also emit sound in all directions, so attenuation factors can be specified in a similar way to PointLight nodes.

Problems with Sound

The Sound classes in the current version of Java 3D (v.1.3.1) contain some severe bugs, including poor volume adjustment when the user moves away from a PointSound or ConeSound, strange interactions between multiple sounds at different locations, and anomalies between left and right ear sounds. Hopefully, these will be fixed in future versions of the API.

4. What to Pick and Not to Pick

Picking is the selection of a shape (or shapes) in the scene, usually by having the user click the mouse while the pointer is 'over' a particular shape.

This is implemented by projecting a line (a ray) into the scene from the user's viewpoint, through the mouse pointer position on screen so it intersects with things in the scene (see Figure 3).

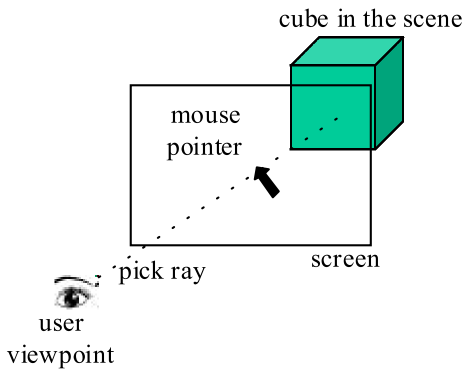

Figure 3. Picking using a Ray.

There are many variations of this idea, such as using parallel projection instead of a perspective view. Another is to use a different projection geometry instead of a line -- a cone or cylinder, for example. A third variation is to return a list of all the intersected objects rather than just the one nearest to the viewer.

Picking returns the visual object selected from in the scene, which may not contain sufficient information for the application. However, visual objects are leaf nodes of scene graph branches containing such things as TransfromGroups, Switches, and so on. Java 3D allows the path from the Locale node to the selected node to be accessed.

By default, leaf nodes, such as Shape3D's and OrientedShape3D's, are pickable, and so programmers try to switch *off* picking (with setPickable(false)) in as many nodes as possible, to reduce the cost of intersection testing.

Nodes which are internal to a scene graph (typically subclasses of Group) are not normally added to the scene graph path generated at picking time. Often a path will be empty due to this. If a Group node needs to be added to the path then its ENABLE_PICK_REPORTING capability must be set prior to picking.

The picking of a leaf node can return different amounts of information, but more information means increased processing times, and requires the setting of a bewildering range of capability bits. Fortunately, the PickTool class offers a setCapabilities() method for setting the necessary bits in a Shape3D node at three different levels of picking:

```
static void setCapabilities(Node node, int level);
```

The levels are: `INTERSECT_TEST`, `INTERSECT_COORD`, and `INTERSECT_FULL`. `INTERSECT_TEST` is used to only test if a given shape intersects a ray, `INTERSECT_COORD` goes further by returning intersection coordinates, while `INTERSECT_FULL` returns details of the geometry's colour, normals, and texture coordinates.

A consideration of the scene in Shooter3D (Figure 1) reveals a range of shapes:

- two Shape3D nodes holding the QuadArrays for the blue and green floor tiles;
- 42 Text2D axis labels;
- the Shape3D red tile at the origin;
- the gun cylinder and cone;
- the laser beam cylinder;
- the explosion Shape3D.

Shooter3D switches off picking for all of these apart from the floor tiles and the red tile at the origin. This means that small changes must be made to our familiar CheckerFloor and ColouredTiles classes.

In CheckerFloor, makeText() is employed to create an axis label, and now includes a call to setPickable():

```
private TransformGroup makeText(Vector3d vertex, String text)
// Create a Text2D object at the specified vertex
{
  Text2D message = new Text2D(text,white,"SansSerif",36,Font.BOLD);
  message.setPickable(false);   // cannot be picked
      :
}
```

In ColouredTile, picking is left on, but the amount of detail to return must be set with a call to setCapabilities(). We only require intersection coordinates, not information about the shape's colour, normals, etc., and so the `INTERSECT_COORD` picking mode is sufficient:

```
public ColouredTiles(ArrayList coords, Color3f col)
{ plane = new QuadArray(coords.size(),
          GeometryArray.COORDINATES | GeometryArray.COLOR_3 );
  createGeometry(coords, col);
  createAppearance();
  // set the picking capabilities so that intersection
  // coords can be extracted after the shape is picked
  PickTool.setCapabilities(this, PickTool.INTERSECT_COORD);
  }
```

The other objects in the scene: the gun, laser beam, and explosion, all contain calls to setPickable(false).

ShootingBehaviour extracts and utilises the intersection information, so a detailed discussion of that side of picking will be delayed until later. Essentially, it uses a ray to find the intersection point on a tile nearest to the viewer. There is no need to obtain path information about the Group nodes above it in the graph, so no need to set any ENABLE_PICK_REPORTING capability bits for Group nodes.

5. The Gun

The GunTurret class hides the creation of a scene graph branch for the cylinder and cone, and has two public methods: getGunBG() and makeRotation(). getGunBG() is used by WrapShooter3D to retrieve a reference to the gun's top-level BranchGroup, gunBG, so it can be added to the scene. makeRotation() is called by ShootingBehaviour to rotate the cone to point at the clicked position.

The scene graph branch built inside GunTurret is shown in Figure 4.

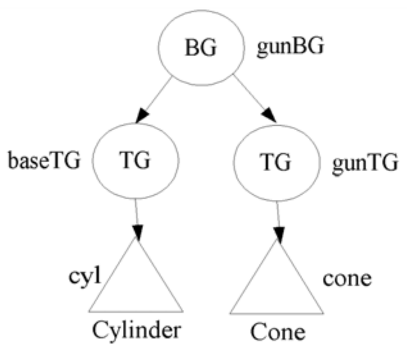

Figure 4. Scene Graph Branch for GunTurret.

The GunTurret constructor:

```
  public GunTurret(Vector3d svec)
  { startVec = svec;
    gunBG = new BranchGroup();
    Appearance apStone = stoneApp();
    placeGunBase(apStone);
    placeGun(apStone);
  }
```

StartVec contains the position of the gun cone (0,2,0).

apStone is a blending of a stone texture and white material, with lighting enabled, which allows lighting effects to be seen on the gun's surfaces. The blending is done using the TextureAttribute.MODULATE setting for the texture mode; a similar approach was used in chapter 12 for the texture applied to the particle system of Quads.

placeGunBase() creates the left hand side of the sub-graph shown in Figure 4, and placeGun() the right side.

The cylinder and cone are both made unpickable:

```
    cyl.setPickable(false);         cone.setPickable(false);
```

The TransformGroup for the cone (gunTG) will have its rotation details changed at run-time, so its capability bits are set accordingly:

```
    gunTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    gunTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
```

makeRotation() is called with an AxisAngle4d object which, as the name suggests, is a combination of an axis (vector) and an angle to rotate around that vector. The vector can specify any direction, and so the resulting rotation is a generalisation of the rotations we have used so far, which have been around the x-, y-, or z- axes only.

```
  public void makeRotation(AxisAngle4d rotAxis)
  // rotate the cone of the gun turret
  { gunTG.getTransform( gunT3d );        // get current transform
    gunT3d.get( currTrans );             // get current translation
    gunT3d.setTranslation( ORIGIN );     // translate to origin

    rotT3d.setRotation( rotAxis );       // apply rotation
    gunT3d.mul(rotT3d);

    gunT3d.setTranslation( currTrans );  // translate back
    gunTG.setTransform( gunT3d );
  }
```

The rotation is applied to gunTG. Since the cone is located away from the origin, it is first translated to the origin, rotated, then moved back to it's original position.

A general optimization used here, and in many other places in the application, is to employ global variables for repeated calculations instead of creating new, temporary objects. This is why the Transform3D and Vector3d objects are globals (gunT3d, rotT3d, currTrans). A further optimization would be to hardwire the translation value, since the cone never moves, only rotates.

6. The Laser Beam

The LaserBeam object is a red cylinder, which is hidden inside the gun cone when not in use, so there is no need for a Switch or visibility-controlling code.

ShootingBehaviour rotates the cylinder (and gun cone) to point at the location picked by the user on the checkboard. It then lets FireBeam handle the shooting of the beam and subsequent explosion.

The laser beam is accompanied by a PointSound, which moves along with it. This means that its volume increases (or diminishes) as the beam travels towards (or away from) the user's viewpoint.

The class diagram for LaserBeam in Figure 5 shows its public methods.

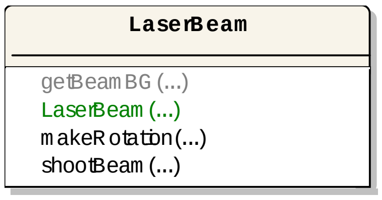

Figure 5. LaserBeam's Public Methods.

WrapShooter3D uses getBeamBG() to retrieve the beam's BranchGroup for addition to the scene. ShootingBehaviour rotates the beam with makeRotation(), which is identical to the method in GunTurret, except that it applies the rotation to the beam's TransformGroup. shootBeam() is called from FireBeam to deliver the beam to the position on the floor clicked on by the user.

The scene graph branch built inside LaserBeam (by makeBeam()) is shown in Figure 6.

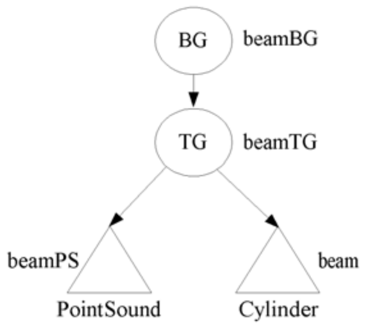

Figure 6. Scene Graph Branch for LaserBeam.

The cylinder is made unpickable:

```
    beam.setPickable(false);
```

The capability bits of beamTG are set to allow it to be rotated and translated.

shootBeam() moves the beam towards a point (the intercept), in incremental steps defined by stepVec, with a brief delay between each move of SLEEP_TIME ms. While the beam is in flight, a sound is played.

```
public void shootBeam(Point3d intercept)
{ double travelDist = startPt.distance(intercept);
  calcStepVec(intercept, travelDist);

  beamPS.setEnable(true);         // switch on laser beam sound

  double currDist = 0.0;
  currVec.set(startVec);
  beamTG.getTransform(beamT3d);   // get current beam transform

  while (currDist <= travelDist) {  // not at destination yet
    beamT3d.setTranslation(currVec);  // move the laser beam
    beamTG.setTransform(beamT3d);
    currVec.add(stepVec);
    currDist += STEP_SIZE;
    try {
      Thread.sleep(SLEEP_TIME);    // wait a while
    }
    catch (Exception ex) {}
  }

  // reset beam to its original coordinates
  beamT3d.setTranslation(startVec);
  beamTG.setTransform(beamT3d);

  beamPS.setEnable(false);   // switch off laser beam sound
}  // end of shootBeam()
```

shootBeam() first calculates the distance to be traveled (travelDist) from the starting point to the intercept, and a translation increment (stepVec) based on a hardwired step size constant. These values are shown graphically in Figure 7.

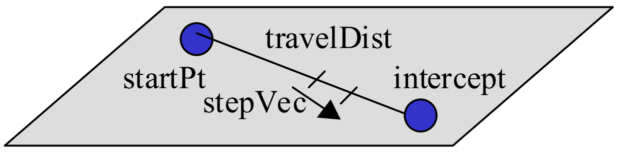

Figure 7. Moving the Laser Beam.

The playing of the sound is controlled by setEnable() which requires the `WRITE` capability bit to be set in initSound(), as explained earlier.

The beam's current position is stored in currVec, and its current distance along the path to the intercept in currDist. currVec is used to update the beam's position by modifying its TransformGroup, beamTG. The while loop continues this process until the required distance has been traveled.

When the beam has reached the intercept, it is reset to its original position at startVec, which hides it from the user back inside the cone.

7. The Explosion

The explosion is best explained by considering the subgraph created inside ExplosionsClip (see Figure 8).

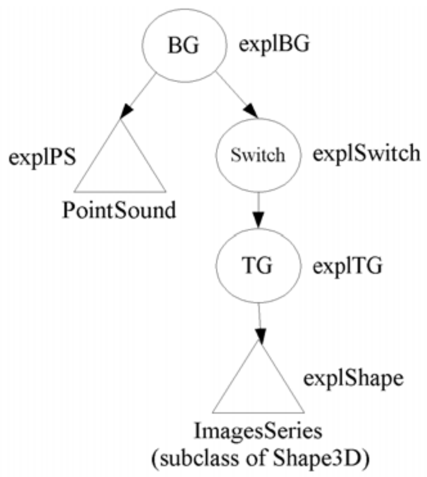

Figure 8. Scene Graph Branch for ExplosionsClip.

The visual component of the explosion is implemented as a series of transparent GIF images, drawn one after another onto the surface of a QuadArray inside explShape. The details will be explained below when we consider the ImagesSeries class.

explTG is utilised to position the explosion shape at the point where the user clicked the mouse, and rotate it around the y-axis to face the user's viewpoint. The Switch node is used to hide the explosion until needed.

The original design for the explosion had the PointSound attached to the TransformGroup, in a similar way to the laser beam subgraph (Figure 6). However, a runtime exception was always raised as the sound was enabled, apparently because of its presence below a Switch node. Consequently, it was moved to a separate branch but now requires explicit positioning.

The subgraph is created in the constructor for ExplosionsClip, and a reference to explBG is retrieved by WrapShooter3D calling getExplBG().

The explosion is displayed by showExplosion() called from FireBeam, after the laser beam has reached the click point.

```
public void showExplosion(double turnAngle, Point3d intercept)
// turn to face eye and move to click point
{
  endVec.set(intercept.x, intercept.y, intercept.z);
  rotateMove(turnAngle, endVec);

  explSwitch.setWhichChild( Switch.CHILD_ALL );   // make visible
  explPS.setPosition((float)intercept.x,
                     (float)intercept.y, (float)intercept.z);
                     // move sound to click point
  explPS.setEnable(true);         // switch on explosion sound
  explShape.showSeries();         // show the explosion
  explPS.setEnable(false);        // switch off sound
  explSwitch.setWhichChild( Switch.CHILD_NONE ); // invisible
```

```
    // face front again, and reset position
    rotateMove(-turnAngle, startVec);
  }  // end of showExplosion()
```

FireBeam passes the user's click point (intercept) and the turning angle for the explosion (turnAngle) to showExplosion(). The rotation is handled by rotateMove(), explained below, and the animation is triggered by a call to showSeries() in the ImagesSeries object.

Note that the positioning and enabling of the PointSound, explPS, require capabilities to have been set previously, which was done in initSound().

After the explosion has finished, it is hidden, and rotated back to its original orientation.

rotateMove() uses the turning angle to rotate around the y-axis and so can employ rotY() rather than an AxisAngle4d object. As usual, the object must be translated to the origin before the rotation, then translated to its new position afterwards.

```
  private void rotateMove(double turn, Vector3d vec)
  // rotate the explosion around the Y-axis, and move to vec
  {
    explTG.getTransform(explT3d);       // get transform info
    explT3d.setTranslation(ORIGIN);     // move to origin

    rotT3d.rotY(turn);         // rotate around the y-axis
    explT3d.mul(rotT3d);

    explT3d.setTranslation(vec);       // move to vector
    explTG.setTransform(explT3d);      // update transform
  }
```

The ImagesSeries Class

The constructor for the ImagesSeries class takes a partial filename (e.g. "images/explo"), and a number (e.g. 6), and attempts to loads GIF files which use that name and numbering scheme (e.g. "images/explo0.gif", …, "images/explo5.gif"). The images are stores as ImageComponent2D objects in an ims[] array.

ImagesSeries is a Shape3D subclass, containing a QuadArray placed on the XZ plane centered at (0,0). The quad is a single square, of size screenSize, with its front face oriented along the positive z-axis, as in Figure 9.

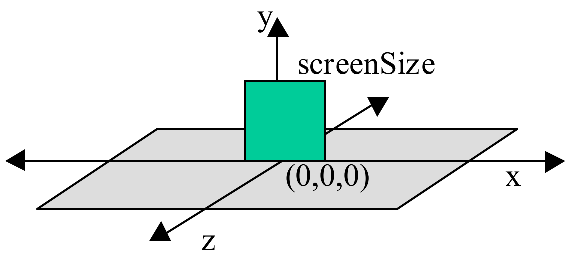

Figure 9. The ImagesSeries QuadArray.

Implicit in the square shape is the assumption that the GIFs will be square, otherwise they will be distorted as they are laid over the face.

The texture coordinates are assigned anti-clockwise from the bottom left coordinate of the quad, so the texture will be the right way up, and facing out along the positive z-axis, towards the viewer.

The important component of the shape is its appearance, which uses blended transparency so that the any transparent parts of a GIF will remain transparent when the image is applied as a texture.

```
    Appearance app = new Appearance();
    // blended transparency so texture can be irregular
    TransparencyAttributes tra = new TransparencyAttributes();
    tra.setTransparencyMode( TransparencyAttributes.BLENDED );
    app.setTransparencyAttributes( tra );
```

No Material node component is assigned to the shape which means that lighting cannot be enabled, and so the shape is unaffected by the lighting in the scene. The code to do this would be:

```
    // mix the texture and the material colour
    TextureAttributes ta = new TextureAttributes();
    ta.setTextureMode(TextureAttributes.MODULATE);
    app.setTextureAttributes(ta);

    Material mat = new Material();    // set material and lighting
    mat.setLightingEnable(true);
    app.setMaterial(mat);
```

The Texture2D object that holds the texture is based on the size of the first image in ims[], and the code assumes that all the subsequent GIFs are the same size:

```
    // Set the texture from the first loaded image
    texture = new Texture2D(Texture2D.BASE_LEVEL, Texture.RGBA,
                       ims[0].getWidth(), ims[0].getHeight());
    texture.setImage(0, ims[0]);
    texture.setCapability(Texture.ALLOW_IMAGE_WRITE);
                              // texture can change
    app.setTexture(texture);

    setAppearance(app);
```

The capability bit allows the texture to be changed by showSeries(), which is called from ExplosionsClip:

```
  public void showSeries()
  { for (int i=0; i < ims.length; i++) {
      texture.setImage(0, ims[i]);
      try {
        Thread.sleep(DELAY);      // wait a while
      }
      catch (Exception ex) {}
```

```
    }
  }
```

There are obvious variants of this idea, such as allowing the animation to cycle, or to be played from some arbitrary point in the sequence.

8. Picking with a Mouse Click

As mentioned earlier, picking is usually implemented by projecting a line (a ray) into the scene from the user's viewpoint, through the mouse pointer position on screen, until it intersects with a shape in the scene (see Figure 3).

The PickCanvas class is used to turn a mouse click into a ray (a PickShape object). Java 3D finds the pickable shapes which intersect with the PickShape object, returning them as a list of PickResult objects. It is also possible to return just the PickResult object closest to the viewer.

A single PickResult may contain many PickIntersection objects, which hold the data for each intersection of the shape (e.g. the ray may go through the front and back face of the shape, leading to two intersection points for the single shape).

The complexity of the picking coding is somewhat alleviated by using the PickMouseBehavior utility class, a subclass of Behaviour, which hides much of the picking mechanism. The general format for a subclass of PickMouseBehavior:

```
import javax.media.j3d.*;
import com.sun.j3d.utils.picking.PickTool;
import com.sun.j3d.utils.picking.PickResult;
import com.sun.j3d.utils.picking.behaviors.PickMouseBehavior;
      :  // other imports as necessary

public class ExamplePickBehavior extends PickMouseBehavior
{
  public PickHighlightBehavior(Canvas3D canvas, BranchGroup bg,
                                                     Bounds bounds)
  { super(canvas, bg, bounds);
    setSchedulingBounds(bounds);

    pickCanvas.setMode(PickTool.GEOMETRY_INTERSECT_INFO);
       // allows PickIntersection objects to be returned
         :
  }

  public void updateScene(int xpos, int ypos)
  {
    pickCanvas.setShapeLocation(xpos, ypos);
        // register mouse pointer location on the screen (canvas)

    Point3d eyePos = pickCanvas.getStartPosition();
        // get the viewer's eye location

    PickResult pickResult = null;
    pickResult = pickCanvas.pickClosest();
        // get the intersected shape closest to the viewer

    if (pickResult != null) {
      PickIntersection pi =
        pickResult.getClosestIntersection(eyePos);
            // get the closest intersect to the eyePos point
      Point3d intercept = pi.getPointCoordinatesVW();
           // extract the intersection pt in scene coords space
              :
    }
  } // end of updateScene()

} // end of ExamplePickBehavior class
```

The constructor must pass the Canvas3D object, a BranchGroup, and bounds information to the superclass in order for it to create a PickCanvas object and a PickShape. The PickCanvas object, pickCanvas, is available, and can be used to configure the PickShape, such as changing it from a ray to a cone, or adjusting the tolerance for how close a shape needs to be to the PickShape to be picked.

There are many subclasses of PickShape (e.g. PickRay, PickCone, PickCylinder) which specify different kinds of ray geometries; the default one employed by PickMouseBehavior is a line (PickRay).

PickCanvas is intended to make picking based on mouse events easier. It is a subclass of PickTool which has many further features/methods.

The call to pickCanvas.setMode() in ExamplePickBehavior's constructor specifies the level of detail for the returned pick and intersection data. The various modes are: `BOUNDS`, `GEOMETRY`, and `GEOMETRY_INTERSECT_INFO`. The `BOUNDS` mode tests for intersection using the bounds of the shapes rather than the shapes themselves, so is quicker. `GEOMETRY` uses the actual shapes, so it more accurate. Both modes return the intersected shapes, but nothing more detailed.

The `GEOMETRY_INTERSECT_INFO` mode tests for intersection using the shapes, and returns details about the intersections, stored in PickIntersection objects. The level of detail is controlled by the capabilities set in the shapes using PickTool's setCapabilities() method.

Although PickMouseBehavior is a Behavior subclass, no use should be made of the initialize() or processStimulus() methods. Instead the programmer should implement the updateScene() method which is called whenever the user clicks the mouse button – the method is passed the (x,y) coordinate of the mouse click on the screen (the Canvas3D).

The first step in updateScene() is to call setShapeLocation() to inform pickCanvas of the mouse position so that the PickShape (the ray) can be cast into the scene.

The intersecting shapes can be obtained in a variety of ways: pickClosest() gets the PickResult object closest to the viewer. Other methods include:

```
      PickResult[] pickAll();
      PickResult[] pickAllSorted();
      PickResult pickAny();
```

The first two return all the intersecting shapes, with the second method sorting them into increasing distance from the viewer. pickAny() returns any shape from the ones found, which should be quicker than finding the closest.

Finding Intersections

A PickResult will usually refer to a Shape3D containing a GeometryArray subclass made up of many surfaces. All the intersections between the shape and the ray can be obtained in the following way:

```
PickIntersection pi;
for (int i = 0; i < pickResult.numIntersections(); i++) {
  pi = pickResult.getIntersection(i);
    :
}
```

More commonly, the intersection closest to some point in the scene is obtained:

```
PickIntersection pi = pickResult.getClosestIntersection(pt);
```

In ExamplePickBehavior, the point is the viewer's position, which is extracted from pickCanvas with getStartPosition().

A PickIntersection object can hold a wide range of information about the GeometryArray, such as the point, line, triangle, or quad that was intersected, and the intersection point both in the local coordinates of the shape or the coordinate space of the scene. If the picking level for the shape is `INTERSECT_FULL`, then there will also be details about the closest vertex to the intersection point, and the color, normal and texture coordinates at the intersection point.

The call to getPointCoordinatesVW() obtains the intercept point in the scene's coordinate space:

```
Point3d intercept = pi.getPointCoordinatesVW();
```

9. ShootingBehaviour

ShootingBehaviour is a subclass of PickMouseBehavior, and controls the various shooting-related entities when the user clicks the mouse. The gun cone and laser beam are rotated to point at the placed clicked on the checkboard. Then a FireBeam thread is created to move ('fire') the beam to the location, and display the explosion.

ShootingBehaviour's central role in the application means that it is passed references to the GunTurret, LaserBeam, and ExplosionsClip objects. In the first version of this class, the code was extremely complex since it dealt directly with the TransformGroups and Shape3Ds of the shooting elements. Good OO design of the application entities (e.g. hiding subgraph details and computation) meant a halving of ShootingBehaviour's code length, making it much easier to understand, maintain, and modify.

The ShootingBehaviour constructor is quite similar to the constructor in ExamplePickBehavior:

```
public ShootingBehaviour(Canvas3D canvas, BranchGroup root,
          Bounds bounds, Point3d sp, ExplosionsClip ec,
          LaserBeam lb, GunTurret g)
{ super(canvas, root, bounds);
```

```
    setSchedulingBounds(bounds);

    pickCanvas.setMode(PickCanvas.GEOMETRY_INTERSECT_INFO);
    // allows PickIntersection objects to be returned

    startPt = sp; // location of the gun cone
    explsClip = ec;
    laser = lb;
    gun = g;
       :
  }
```

updateScene() is also similar to the one in ExamplePickBehavior since it requires intersection information. updateScene() rotates the gun cone and beam to point at the intercept and starts an FireBeam thread to fire the beam and display an explosion.

```
public void updateScene(int xpos, int ypos)
  {
    if (finishedShot) {   // previous shot has finished
      pickCanvas.setShapeLocation(xpos, ypos);

      Point3d eyePos = pickCanvas.getStartPosition();  // viewer loc

      PickResult pickResult = null;
      pickResult = pickCanvas.pickClosest();

      if (pickResult != null) {
        pickResultInfo(pickResult);  // for debugging

        PickIntersection pi =
            pickResult.getClosestIntersection(startPt);
                  // get intersection closest to the gun cone
        Point3d intercept = pi.getPointCoordinatesVW();

        rotateToPoint(intercept);    // rotate the cone and beam
        double turnAngle = calcTurn(eyePos, intercept);

        finishedShot = false;
        new FireBeam(intercept, this, laser,
                              explsClip, turnAngle).start();
            // fire the beam and show explosion
      }
    }
  } // end of updateScene()
```

The finishedShot flag has an important effect on the behaviour of the application – it only allows a single laser beam to be 'in the air' at a time. As FireBeam is started, finishedShot is set to false, and will remain so until the thread has moved the beam to the intercept point. This means that if the user clicks on the checkboard while a beam is still travelling, nothing will happen since the if-test in updateScene() will return false.

The reason for this design is to limit the number of laser beam objects required by the application to just one! Otherwise, the coding would have to deal with a user that could quickly click multiple times, each requiring its own laser beam.

The call to getClosestIntersection() uses startPt, which is set in the constructor to be the cone's location. The resulting intercept will be the point nearest to the cone.

Debugging Picking

The call to pickResultInfo() plays no part in the shooting process; it is used to print extra information about the PickResult object (pr), to check that the picking code is selecting the correct shape.

getNode() is called to return a reference to the shape that the PickResult object represents:

```
Shape3D shape = (Shape3D) pr.getNode(PickResult.SHAPE3D);
```

The code must deal with a potential null result which would occur if the selected node was not a Shape3D object.

The PickResult object, pr, also contains the scene graph path between the Locale and picked node, which can be used to access an object above the picked node, such as a TransformGroup. The path is obtained by calling getSceneGraph():

```
SceneGraphPath path = pr.getSceneGraphPath();
```

This path may often be empty, since internal nodes are not added to it unless their `ENABLE_PICK_REPORTING` capability bit is set. (In fact, the matter is a little more complex than this if there are SharedGroup nodes in the scene graph.)

The path can be printed with a for-loop:

```
int pathLen = path.nodeCount();
for (int i=0; i < pathLen; i++) {
  Node node = path.getNode(i);
  System.out.println(i + ". Node: " + node);
}
```

If the sceneBG BranchGroup node created in WrapShooter3D has the necessary capability bit set:

```
sceneBG.setCapability(BranchGroup.ENABLE_PICK_REPORTING);
```

Then the output from the for-loop in pickResultInfo() is:

```
0.  Node: javax.media.j3d.BranchGroup@2bcd4b
```

This is not particularly informative. A typical way of improving the labeling of scene graph nodes is to use the setUserData() method from SceneGraphObject, which allows arbitrary objects to be assigned to a node (e.g. a String object):

```
sceneBG.setUserData("the sceneBG node");
```

After a reference to the node has been retrieved, getUserData() can be employed:

```
String name = (String)node.getUserData();
System.out.println(i + ". Node name: " + name);
```

Rotating the Cone

rotateToPoint() rotates the gun cone and laser beam cylinder to point at the intercept. The problem is that a simple rotation about the x-, y-, or z- axis is insufficient, since the intercept can be anywhere on the floor. Instead, an AxisAngle4d rotation is utilised, which allows a rotation about any vector.

The essential algorithm is illustrated in Figure 10.

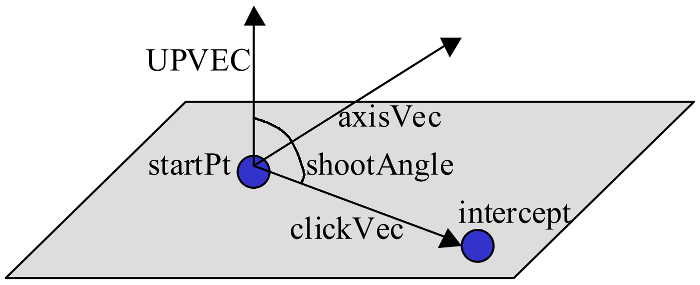

Figure 10. Rotating to face the Intercept.

The cone (and beam) start by pointing in the UPVEC direction at the StartPt location, and they must be rotated to point in the clickVec direction, a rotation of shootAngle radians. The rotation is around the axisVec vector, which is normal to the plane defined by the two vectors UPVEC and clickVec.

startPt and UPVEC values are predefined, and intercept is supplied by updateScene() when it calls rotateToPoint(). clickVec is readily calculated from the startPt and intercept points:

```
clickVec.set( intercept.x-startPt.x, intercept.y-startPt.y,
                                  intercept.z-startPt.z);
```

axisVec is known as the *cross product*, and the Vector3d class contains a cross() method which calculates it, given normalized values for UPVEC and clickVec:

```
clickVec.normalize();
axisVec.cross( UPVEC, clickVec);
```

The rotation angle, shootAngle, between UPVEC and clickVec can also be easily calculated with Vector3d's angle() method:

```
shootAngle = UPVEC.angle(clickVec);
```

shootAngle is related to the *dot product*: the dot product of vectors a and b (often written as a . b) gives the length of the projection of b onto a. For example, a . b = | x | in Figure 11.

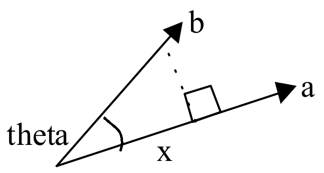

Figure 11. The Dot Product of Vectors a and b.

The angle between a and b, theta, can be expressed as:

$$\cos \text{theta} = \frac{a \cdot b}{|a|\ |b|}$$

If a and b are unit vectors, as in our code, then:

$$\cos \text{theta} = a \cdot b$$

This leads to another way of calculating shootAngle in Java 3D:

```
    shootAngle = Math.acos( UPVEC.dot(clickVec) );
```

dot() is the dot product operation, acos() the arc cosine.

An AxisAngle4d object requires a vector and rotation, which can now be supplied:

```
    rotAxisAngle.set(axisVec, shootAngle);
```

This is used to rotate both the cone and laser beam:

```
    gun.makeRotation(rotAxisAngle);
    laser.makeRotation(rotAxisAngle);
```

An extra complication is that rotateToPoint() assumes that the cone and beam start in the UPVEC direction, which is only true at the start of the application. For rotations after the first, the objects must be rotated back to the vertical first. This is achieved by rotating by shootAngle around the *negative* of the axisVec vector:

```
    if (!firstRotation) {    // undo previous rotations
      axisVec.negate();
      rotAxisAngle.set( axisVec, shootAngle);
      gun.makeRotation(rotAxisAngle);
      laser.makeRotation(rotAxisAngle);
    }
```

Making the Explosion Face the Viewer

updateScene() also calls calcTurn() to calculate the angle which the explosion shape should rotate in order to face the viewer:

```
    double turnAngle = calcTurn(eyePos, intercept);
```

The algorithm is illustrated by Figure 12.

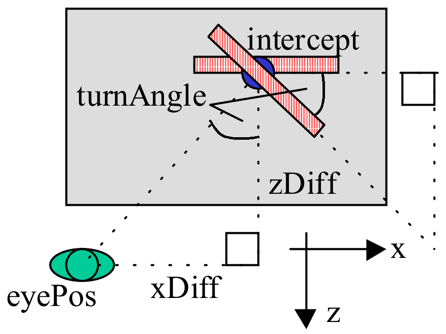

Figure 12. Turning to Face the Viewer.

The eyePos and intercept points are supplied by updateScene(). turnAngle is readily calculated as the arc tangent of xDiff and zDiff:

```
    double zDiff = eyePos.z - intercept.z;
    double xDiff = eyePos.x - intercept.x;
    double turnAngle = Math.atan2(xDiff, zDiff);
```

Figure 12 shows the explosion shape as a striped rectangle facing the positive z- axis, which must be rotated to face eyePos. A little geometry confirms that its required rotation is the same as turnAngle calculated above.

10. Firing the Beam

The FireBeam thread is started from updateScene() like so:

```
    new FireBeam(intercept, this, laser, explsClip,
                                       turnAngle).start();
```

A basic question is why use a thread? One answer is that by passing the job of beam delivery and explosion to a new thread, the ShootingBehaviour object is free to do other tasks. Actually, this benefit is not much exploited in our code since updateScene () will only process new user picks once the finishedShot boolean has been set to true *near* the end of run() in the FireBeam thread.

```
  public void run()
  {
    laser.shootBeam(intercept);
    shooter.setFinishedShot();    // beam has reached its target
    explsClip.showExplosion(turnAngle, intercept);   // boom!
  }
```

The call to setFinishedShot() sets finishedShot to true, which permits updateScene() to respond to user clicks and, *at the same time*, the explosion for the current beam will be initiated from FireBeam. This improves the responsiveness of the application since the explosion animation lasts 1-2 seconds.

However, there is a problem – what if the explosion animation for the beam (i.e. the current call to showExplosion()) has not finished before the FireBeam thread for the next beam calls showExplosion()?

The worst that happens is an interruption to the explosion animation, and the truncation of the playing of the sound. However, in the vast majority of situations, the travel time of the laser beam and the explosion animation speed means that the explosion has finished before it is required again.

From a practical point of view, this may be sufficient, but in the next chapter we look at a better coding approach that allows multiple beams and multiple explosions to co-exist safely on screen at the same time.

11. More on Picking

The Java 3D tutorial has a long section on picking in chapter 4 “Interaction and Animation”, and there are two picking examples: MousePickApp.java and PickCallbackApp.java.

The former is explained in the tutorial and shows how to use the PickRotateBehavior subclass of PickMouseBehavior to select and rotate shapes. The other predefined subclasses are PickTranslateBehavior and PickZoomBehavior.

There is not much information on how to create your own subclasses of MousePickBehavior, but it is possible to look at the source code for these utilities, which is in java3d-utils-src.jar (if you downloaded it). A potential source of confusion is that you will find two copies of each of these classes: the deprecated ones located in com.sun.j3dutils.behaviors.picking, and the current ones in com.sun.j3dutils.picking.behaviors.

The PickCallbackApp.java example shows how to attach a callback method to the PickRotateBehavior object, which is called automatically when a pick operation takes place. The code is derived from the MousePickApp.java example.

Another source of examples are the Java 3D demos, located in demo/java3d/ below JAVA_HOME. There are several relevant subdirectories: PickTest/, PickText3D/, and TickTockPicking/.

The TickTockPicking example involves the picking of cubes and tetrahedrons to change their appearance, and utilises a easy to understand subclass of PickMouseBehavior called PickHighlightBehavior.

We return to picking in chapter 17 ?? when we use it to determine the position of terrain below the viewer as he/she moves around a landscape. This is a common approach to handling movement over irregular surfaces. The code utilises a PickTool object to fire a ray straight down beneath the viewer to intersect with the shape representing the ground.