Development of 3D graphics software using Java 3D

By Martin Adams

20044845

Due: 9th May 2003

Marking Tutor: Gary Hill

Table of contents

Table of figures

Table of examples

1.0 Introduction

This report will explore the use of the Java programming language and the Java 3D API to develop a 3D enabled application. The application will demonstrate the use of creating 3D geometry, applying materials and textures, user navigation and collision detection. The requirement for the application is to create a room with a table positioned inside. On the table will be a 2-dimensional maze that the user will be able to navigate around.

The technologies available within Java will be explored to create such an environment and allow user interaction around the 3D room and maze.

2.0 Analysis

2.1 Analysis of the application

The requirements of the application are as follows:

- Create a VirtualUniverse to contain the room.
- The room must be 5 metres wide, 7 metres long and 3 metres high.
- Use 100mm block walls and 150mm floor and roof slabs.
- Walls, floors and ceilings should all 'look' different.
- A table should be place centrally in the room. Its top should be circular.
- On the table is to be a maze.
- Horizontal navigation throughout the room.
- Coloured or textured surfaces.
- Incorporation of lighting.
- The maze should be navigable.
- Collision detection on walls and objects.
- Use of perspective.

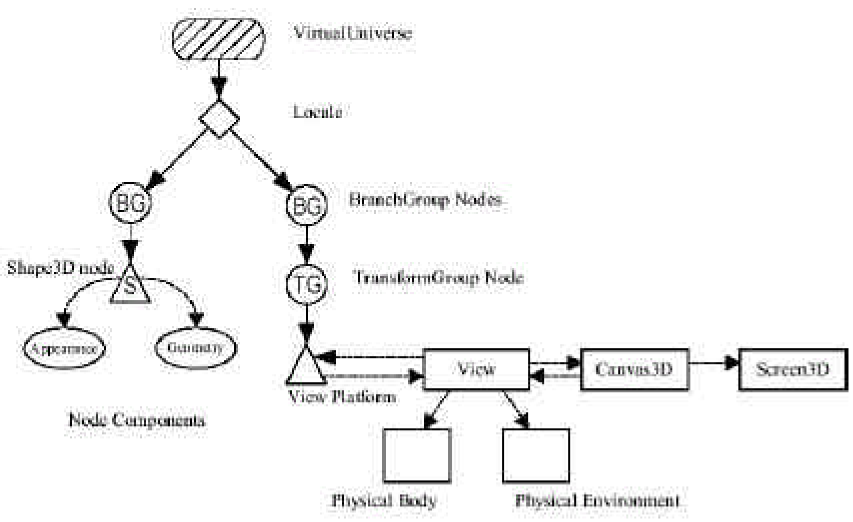
2.2 Analysis of Java 3D

Java 3D is an Application Programming Interface (API) to create a virtual world utilising 3D hardware and software technologies provided either by DirectX or OpenGL. The Java 3D Software Development Kit (SDK), an extension on the Java SDK is freely available from the Sun Microsystems web site *(http://java.sun.com)*. The Java SDK and runtime packages are also freely available.

Java 3D allows a Java developer to implement 3D graphics into their application without excessive code dominating the application. The Java 3D API has been designed to take an object-oriented approach to OpenGL or DirectX programming unlike other languages such as C++. This means that it is possible to program 3D applications using 100% pure Java.

2.2.1 The scene graph

Java 3D uses a class hierarchy to create a *virtual universe* that contains all elements to be rendered. This is called the *scene graph*. Within this hierarchy there are two *Branch Groups* as shown in *figure 2-1*. One branch group defines all the content within the scene that will be used to contain all geometry and lights. The other branch group will contain the *View Platform* that ultimately represents the virtual camera the scene is being viewed from.



Bouvier (2002)

Figure 2-1 – Scene graph example

Each branch group can contain any number of *Transform Groups* that allow the object or view to be manipulated by moving, rotating or scaling the content of view relative to the universe. Transform groups can contain any number of child elements including additional transform groups. This allows content to be transformed relative to a parent content item. An example of this would be to rotate a character's elbow relative to the character's shoulder rotation and position.

2.2.2 3D geometry

A *Shape3D* object defines a piece of geometry that can visually be rendered. This shape can range from being a single triangle polygon to being a complex mesh used to represent real-life objects such as a person or a vehicle. The shape will contain it's geometry information such as its physical points in space and also its appearance such as colour, secularity, textures, etc.

2.2.3 Lights

Lights are somewhat different to a *Shape3D* object because it affects the appearance of other objects but is not rendered on screen. Lights in Java 3D are designed to simulate the effect of real-life lights but are designed for fast performance. The use of ambient and directional lighting is a quick way to simulate natural lighting that does not give a decrease in brightness the further away the object is from the source. Point and spotlights are designed to illuminate a specified region where the strength decreases when the object is further away. These are designed to mimic lights such as a candle or a torch but are computationally expensive compared to ambient and directional lighting.

3.0 Design

3.1 Design of the room

When using a computer system to create objects in 2D or 3D, the computer does not have an understanding of measurement. It is irrelevant to a computer if it is working in millimetres, inches, metres or miles. By definition, according to the Java 3D API documentation, 1 unit in 3D space is equivalent to 1 metre. For the sake of this application, it shall be treated that 1 unit in 3D space is equivalent to 1 cm. This will result in an easier understanding of the application when dealing with small-scale objects such as the 3D maze. *Figure 3-1* shows the desired dimensions of the scene.

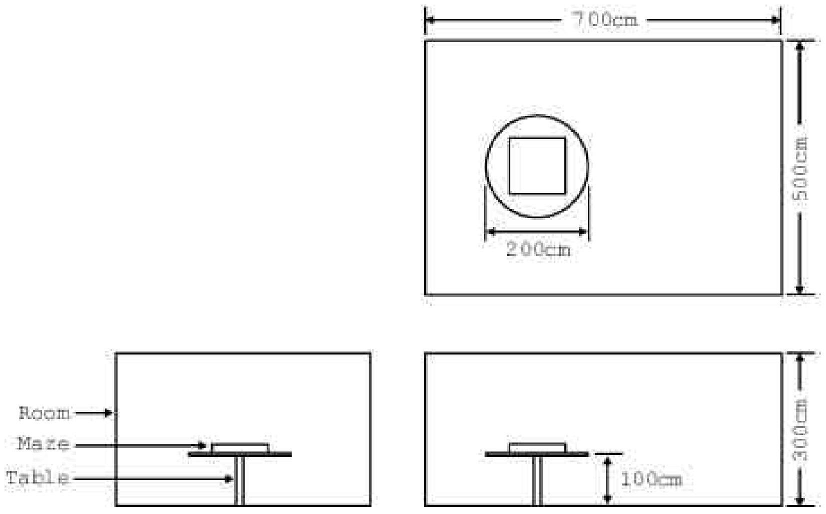


Figure 3-1 – Dimensions of 3D Room

3.2 Approach to developing the application

It is important to identify the main elements to the application and build a class hierarchy that will build up the application. Following is a definition of the classes required for effective implementation of the application:

- **Application class**

  The application class will initialise the 3D context, create the world class, listen for user navigation, update the view port and check for movement collisions when the user moves. This class will mostly act as a joining interface between the child classes.

  - **Viewport class**

    This creates the view and will be used to manipulate the view transformations based on the user navigations, e.g. when the user presses the 'Up' key the view port will move forwards.

  - **World class**

    This will contain all the objects within the world. It will be used to create the room, table, maze and lights and also manage any collision detection based on these objects.

    - **Maze class**

      This will generate the maze geometry and manage any collision detection for the maze.

3.2.1 Creating the room

The room will be created using an *IndexedQuadArray* to specify the geometry. The room will have the following properties:

- Surfaces facing inwards.
- Separate textures for the walls, ceiling and floor.
- Collision detection so the user does not exit the room.

3.2.2 Creating the table

The table will be generated from a series of *Cylinder* objects to create the tabletop, the table leg and the table base. A wood texture will be loaded and applied to the table to add realism. Once the table has been created it must be moved into position within the room.

3.2.3 Creating the maze

The maze will be generated from a local array of integers. The array will be used to define each maze block with the value ‘1’ representing a wall or a ‘0’ representing an open space. *Figure 3-2* shows an example array of integers used to build up a simple maze.

1	1	1	0	1
1	0	0	0	1
1	1	1	0	1
1	0	0	0	1
1	0	1	1	1

$\longrightarrow$

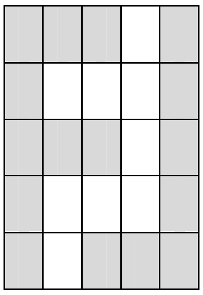


Figure 3-2 – Maze definition by array list

The maze will be generated form an *IndexedQuadArray* by defining all the possible coordinates in 3D space that could be used. Based on the array each surface will be created by indexing the already specified coordinates. When generating a block, it is important to identify any walls based on the neighbouring blocks. For example, two blocks together would not require a joining wall between them, as it would not be visible to the user.

3.2.4 Navigation

All navigation will be activated when the user presses specific keys on the keyboard. When the user presses the ‘Up’ key, it will result in the view *Transform Group* moving forward. Pressing the ‘Left’ or ‘Right’ keys will rotate the view *Transform Group* about the Y axis to simulate turning. By rotating the view matrix, it will allow the user to continue to move in that new direction based on any previous transformations.

3.2.5 Collision detection

Collision detection is where the user tried to pass through what is considered, a physical object. This will include trying to exit the room by walking through a wall or to walk through a wall in the maze.

An important consideration with testing for a collision from the view *Transform Group* is that depending on any transformations from previous movements, it will not directly relate to where the view is in the world. Fortunately, Java 3D provides a method to calculate a world coordinate from the view matrix.

Taking this position coordinate of the view, it can be tested against the various objects within the scene to test for collision. Collision will be tested for the following objects:

- The room

  This will ensure the user does not exit the room.
- The table

  This will be used to detect if the user is currently on the table. If the user enters the table, the view port and navigation will be scaled to match the scale for the maze. When the user exits the table, the view port and navigation will be restored to their default values.
- The maze

  When the user is within the bounds of the maze, the user's position will be tested against the array definition of the maze. If the user has collided with a wall, they will be forced away from the wall so they cannot pass through it.

4.0 Implementation

The application has successfully been created using Java and the Java 3D API. Following is a detailed explanation of the application structure and elements.

4.1 Application structure overview

A series of classes have been created to delegate the responsibility of certain processes for the application. *Figure 4-1* shows the class hierarchy of the *G3DMazeRoom* application.

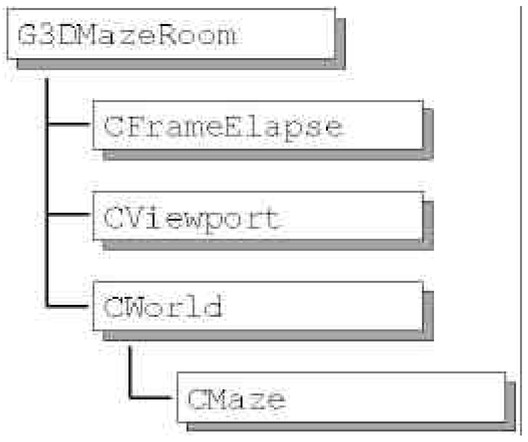


Figure 4-1 – Application class hierarchy

G3DMazeRoom
This is the main application that creates the application window or initialises the applet if run from a web page. This class creates the frame monitor class (*CFrameElapse*), view port and the 3D world. It is also responsible for monitoring user interaction via the keyboard and passes the necessary view port movements and collision detection to the respective sub classes.

CFrameElapse
This class creates a behaviour that monitors every frame that has been rendered on screen. When a frame has been rendered it calls a function within the *G3DMazeRoom* class informing that any animation requires updating. Animation in this context will include movement of the view port, for example, moving the view port forward or changing its field of view.

CViewport

This class creates the necessary transform groups and view platform required to move the view around the 3D scene. It is responsible for moving the view port's position, rotation of and animation when the user enters or exits the table.

CWorld

This class creates all geometry and lights within the scene. It will create an instance of the 3D maze and position it correctly. Collision detection will be checked in this class for the room and table. The maze collision detection will be delegated to the *CMaze* class.

CMaze

This class creates the maze geometry based on its creation position and size. It will also be responsible for managing and correcting collision detection when the view port is within the maze boundaries.

4.2 Creating the world

The world is considered to be all objects within the scene. This will include the room the user is inside, the contents of the room and any lighting.

4.2.1 Creating the room

The room has been created using an *IndexedQuadArray*, which is a series of 4-point flat surfaces (a quad) used to build up each wall surface. There will be four wall surfaces, one floor surface and one ceiling surface making a total of 6 surfaces.

Example 4-1 shows a simplified example on how to create the room surfaces using an *IndexedQuadArray*. Notice the walls, ceiling and floor are created as separate objects. This is to make it easier to apply different textures to the surfaces later.

Example 4-1 – Creation of a Room using an IndexedQuadArray

```
public TransformGroup createRoom(float x, float y, float z, float width,
                                                    float height, float depth)
{
  IndexedQuadArray wallsShape = new IndexedQuadArray(8,
                                  IndexedQuadArray.COORDINATES, 16);
  IndexedQuadArray floorShape = new IndexedQuadArray(8,
                                  IndexedQuadArray.COORDINATES, 4);
  IndexedQuadArray roofShape  = new IndexedQuadArray(8,
                                  IndexedQuadArray.COORDINATES, 4);

  // Specify the room coordinates
  Point3f[] shapeCoordinates = {
    new Point3f(x, y, z),
    new Point3f(x + width, y, z),
    new Point3f(x + width, y + height, z),
    new Point3f(x, y + height, z),
    new Point3f(x, y, z + depth),
    new Point3f(x + width, y, z + depth),
    new Point3f(x + width, y + height, z + depth),
    new Point3f(x, y + height, z + depth)
  };

  // Define the walls surface
  int wallsCoordIndices[] = {
    0, 1, 2, 3,     // back wall
    1, 5, 6, 2, // right wall
    5, 4, 7, 6, // front wall
    4, 0, 3, 7 // left wall
  };

  // Define the ceiling surface
  int roofCoordIndices[] = {
    2, 6, 7, 3 // roof
  };

  // Define the floor surface
  int floorCoordIndices[] = {
    0, 4, 5, 1  // floor
  };

  // Set the wall coordinates and surfaces
  wallsShape.setCoordinates(0, shapeCoordinates);
  wallsShape.setCoordinateIndices(0, wallsCoordIndices);

  // Set the floor coordinates and surfaces
  floorShape.setCoordinates(0, shapeCoordinates);
  floorShape.setCoordinateIndices(0, floorCoordIndices);

  // Set the ceiling coordinates and surfaces
  roofShape.setCoordinates(0, shapeCoordinates);
  roofShape.setCoordinateIndices(0, roofCoordIndices);

  // Move the room into position
  TransformGroup tg = new TransformGroup();
  tg.addChild(new Shape3D(wallsShape, new Appearance()));
  tg.addChild(new Shape3D(floorShape, new Appearance()));
  tg.addChild(new Shape3D(roofShape, new Appearance()));

  return tg;
}
```

4.2.2 Room normals

When creating the above room, it will look rather unattractive from inside due to the wall colour taking a default value of white and there would be no differentiation between each wall due to lack of lighting. When lights are added to the scene, they require a 3D object to have their normals set in order to calculate its appearance in relation to the light and the view port. A normal is a vector that defines how the light reacts to the surface. The next step is to define the room's normals ready for adding a light. *Example 4-2* shows how to add normals the existing room.

Example 4-2 – Adding Normals to the 3D room

```
// Specify the wall normals
Vector3f[] wallsNormals = {
  new Vector3f( 0.0f, 0.0f, -1.0f), // z negative - back wall
  new Vector3f( 1.0f, 0.0f,  0.0f), // x positive - right wall
  new Vector3f( 0.0f, 0.0f,  1.0f), // z positive - front wall
  new Vector3f(-1.0f, 0.0f,  0.0f)  // x negative - left wall
};

// Specify the ceiling normals
Vector3f[] roofNormals = {
  new Vector3f(0.0f, 1.0f, 0.0f) // y positive - roof
};

// Specify the floor normals
Vector3f[] floorNormals = {
  new Vector3f(0.0f, -1.0f, 0.0f) // y negative - floor
};

// Attach the wall normals to each wall point
int wallsNormalIndices[] =
{
  0,0,0,0,
  1,1,1,1,
  2,2,2,2,
  3,3,3,3
};

// Attach the ceiling normals to each ceiling point
int roofNormalIndices[] =
{
  0,0,0,0
};

// Attach the floor normals to each floor point
int floorNormalIndices[] =
{
  0,0,0,0
};

// Set the above noramls to the shapes
wallsShape.setNormals(0, wallsNormals);
wallsShape.setNormalIndices(0, wallsNormalIndices);

floorShape.setNormals(0, floorNormals);
floorShape.setNormalIndices(0, floorNormalIndices);
```

```
roofShape.setNormals(0, roofNormals);
roofShape.setNormalIndices(0, roofNormalIndices);
```

Please note that when creating the *IndexedQuadArray* it is important to include that you are specifying the normals for the object. *Example 4-3* shows how to achieve this.

Example 4-3 – Specifying that normals will be used for an IndexedQuadArray

```
IndexedQuadArray shape = new IndexedQuadArray(8, IndexedQuadArray.COORDINATES |
                                            IndexedQuadArray.NORMALS, 16);
```

4.2.3 Texture mapping the room

While having a nicely shaded room, the scene will obviously lack realism due to the environment around us having further detail than a simple primitive object. It could be possible to add detail by modifying the geometry to a more complex shape but this becomes inefficient for a computer to render in real-time. The solution is to use textures, which are 2-dimensional images that contain a pattern to give the effect of realism in a 3-dimensional world.

One of the important things to bear in mind with Java 3D is that a texture's dimensions must be a power of two. This means it is possible to use a texture dimensions such as 64x64, 128x128 and 256x256 but dimension such as 513x243 are invalid as it breaks the 'power of two' rule. The reason for this is to optimise the performance of rendering the 3D scene.

A texture is mapped (hence the term texture mapping) onto each surface of a triangle or quad. The texture position is defined between 0 and 1 so position (0,0) would be the bottom left corner and (1,1) would be the top right corner. *Figure 4-2* shows examples of a texture mapped onto a variety of surfaces.

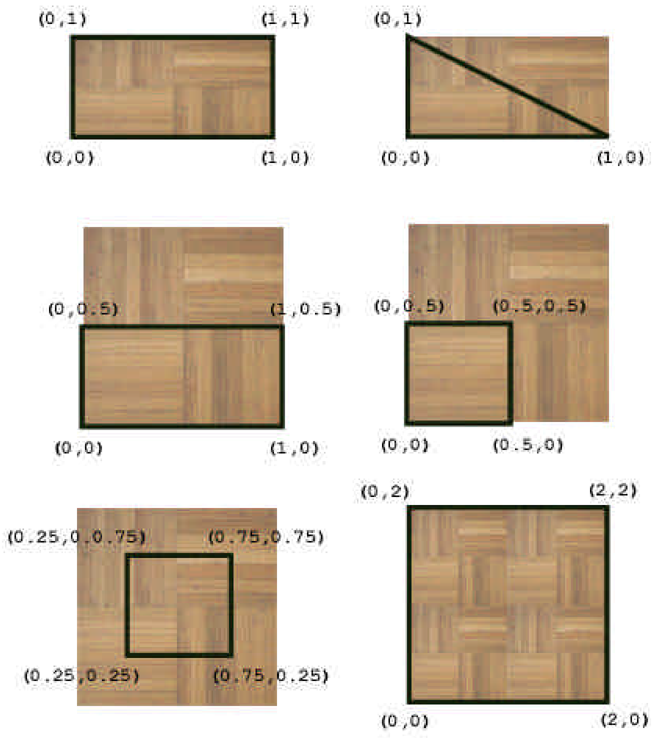


Figure 4-2 – Examples of texture mapping a surface

To texture each surface of the room, the coordinates of where the texture is placed on the surface must be defined. For each surface index, an index to the texture coordinate will be created as well. *Example 4-4* shows a section of code that generates the texture mapping for the existing room.

Example 4-4 – Texture mapping the 3D room

```
// Specify the wall texture coordiantes
TexCoord2f wallsTexCoords[] = {
  new TexCoord2f(0.0f, 0.0f),    // bottom left
  new TexCoord2f(7.0f, 0.0f), // bottom right, 7 along the x
  new TexCoord2f(7.0f, 5.0f), // top right, 7 along the x, 5 in the y
  new TexCoord2f(0.0f, 5.0f)  // top left, 5 in the y
};

// Specify the texture coordinates for the floor and ceiling
TexCoord2f vertTexCoords[] = {
  new TexCoord2f(0.0f, 0.0f), // bottom left
  new TexCoord2f(5.0f, 0.0f), // bottom right x5
  new TexCoord2f(5.0f, 5.0f), // top right x5
  new TexCoord2f(0.0f, 5.0f), // top left x5
};

// Attach the wall coordinates to the wall points
int texWallCoordIndices[] =
{
  0, 1, 2, 3, // back wall
  0, 1, 2, 3, // right wall
  0, 1, 2, 3, // front wall
  0, 1, 2, 3  // left wall
};
```

```
// Attach the floor and ceiling coordinates to the floor and ceiling points
int texVertCoordIndices[] =
{
  0, 1, 2, 3
};

// Set the texture coordinates for all surfaces
wallsShape.setTextureCoordinates(0, 0, wallsTexCoords);
wallsShape.setTextureCoordinateIndices(0, 0, texWallCoordIndices);

floorShape.setTextureCoordinates(0, 0, vertTexCoords);
floorShape.setTextureCoordinateIndices(0, 0, texVertCoordIndices);

// Create the roof based on the above points, normals and texture coordinates
roofShape.setTextureCoordinates(0, 0, vertTexCoords);
roofShape.setTextureCoordinateIndices(0, 0, texVertCoordIndices);

// Generate the surface material for the room
Color3f ambientColour = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
Color3f diffuseColour = new Color3f(1.0f, 1.0f, 1.0f); // white
Color3f specularColour = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
Color3f emissiveColour = new Color3f(0.0f, 0.0f, 0.0f); // black

Material material = new Material(ambientColour, emissiveColour,
                                  diffuseColour, specularColour, 5.0f);

// Set the walls appearance loading the brick texture
Appearance wallsAppearance = new Appearance();
wallsAppearance.setMaterial(material);
wallsAppearance.setTexture(loadTexture("textures/brick_128.jpg"));

// Set the floor appearance loading the floor texture
Appearance floorAppearance = new Appearance();
floorAppearance.setMaterial(material);
floorAppearance.setTexture(loadTexture("textures/wood_floor_128.jpg"));

// Set the ceiling appearance loading the roof texture
Appearance roofAppearance = new Appearance();
roofAppearance.setMaterial(material);
roofAppearance.setTexture(loadTexture("textures/roof_128.jpg"));

// Set the texture attributes to work with the surface material
TextureAttributes texAttributes = new TextureAttributes();
texAttributes.setTextureMode(TextureAttributes.MODULATE);
wallsAppearance.setTextureAttributes(texAttributes);
floorAppearance.setTextureAttributes(texAttributes);
roofAppearance.setTextureAttributes(texAttributes);

// Make sure the walls are flat shaded to react to lights
ColoringAttributes colAttrib = new ColoringAttributes(0.0f, 0.0f, 1.0f,
                                        ColoringAttributes.SHADE_FLAT);
wallsAppearance.setColoringAttributes(colAttrib);
floorAppearance.setColoringAttributes(colAttrib);
roofAppearance.setColoringAttributes(colAttrib);
```

Please note that when creating the *IndexedQuadArray* it is important to include that you are specifying the texture coordinates for the object. *Example 4-5* shows how to achieve this.

Example 4-5 – Specifying texture coordinates will be used for an IndexedQuadArray

```
IndexedQuadArray shape = new IndexedQuadArray(8, IndexedQuadArray.COORDINATES |
           IndexedQuadArray.NORMALS | IndexedQuadArray.TEXTURE_COORDINATE_2, 16);
```

4.2.4 Creating the table

To create the table, three cylinders will be used to create the tabletop, the table leg and the table base. *Figure 4-3* shows what the finished table looks like.

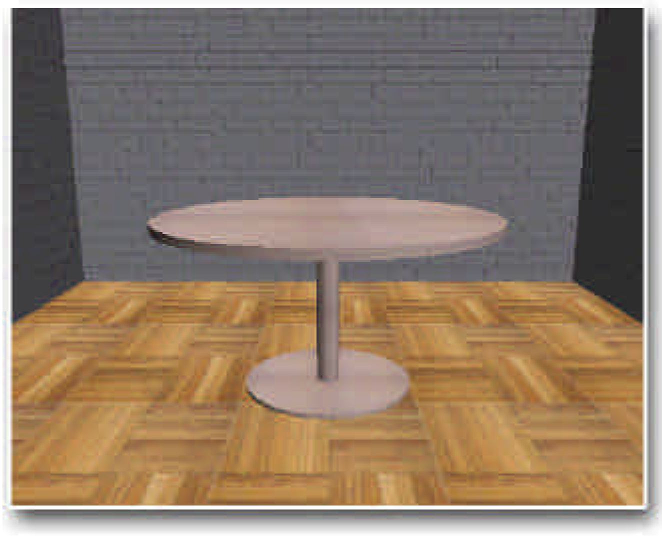


Figure 4-3 – 3D table generated from cylinders

The table will be created using the *Cylinder* object and a wood texture will be applied to it. Since the table is being created using standard primitive objects it is not necessary to specify the normals or texture coordinates as this will be taken care of automatically. *Example 4-6* shows how to create the 3D table.

Example 4-6 – Creating a 3D table using cylinders

```
private TransformGroup createTable(float x, float y, float z,
                                   float diameter, float height)
{
  // Create the TGs for the table sections
  TransformGroup tableTop = new TransformGroup();
  TransformGroup tableLeg = new TransformGroup();
  TransformGroup tableBase = new TransformGroup();

  // Generate an appearance for this shape
  Color3f ambientColour  = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
  Color3f diffuseColour  = new Color3f(1.0f, 1.0f, 1.0f); // white
  Color3f specularColour = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
  Color3f emissiveColour = new Color3f(0.0f, 0.0f, 0.0f); // black
```

```
    // Create a material for the appearance
    Material material = new Material(ambientColour, emissiveColour,
                                  diffuseColour, specularColour, 5.0f);

    // Create the object's appearance and load the wood texture
    Appearance appearance = new Appearance();
    appearance.setMaterial(material);
    appearance.setTexture(loadTexture("textures/pinewood_128.jpg"));

    // Set the texture's attributes to blend with the material properties
    TextureAttributes texAttributes = new TextureAttributes();
    texAttributes.setTextureMode(TextureAttributes.MODULATE);

    // Scale the texture to fit on the surface of the table 2x2
    Transform3D scaleTex = new Transform3D();
    scaleTex.setScale(2);
    texAttributes.setTextureTransform(scaleTex);

    // Set the textures attributes to the appearance
    appearance.setTextureAttributes(texAttributes);

    // Add a cylinder as the table top
    tableTop.addChild(new Cylinder(diameter / 2, 5.0f, Cylinder.GENERATE_NORMALS |
                            Cylinder.GENERATE_TEXTURE_COORDS, 30, 30, appearance));

    // Move the table top to the top of the leg
    Transform3D change = new Transform3D();
    change.set(new Vector3f(0.0f, height / 2, 0.0f));
    tableTop.setTransform(change);

    // Create the leg
    tableLeg.addChild(new Cylinder(7.0f, height, Cylinder.GENERATE_NORMALS |
                            Cylinder.GENERATE_TEXTURE_COORDS, 10, 10, appearance));
    tableLeg.addChild(tableTop);

    // Move the table leg and top into position
    change.set(new Vector3f(0.0f, height / 2, 0.0f));
    tableLeg.setTransform(change);

    // Create the base
    tableBase.addChild(new Cylinder(50.0f, 5.0f, Cylinder.GENERATE_NORMALS |
                            Cylinder.GENERATE_TEXTURE_COORDS, 20, 20, appearance));
    tableBase.addChild(tableLeg);

    // Move the entire table into position
    change.set(new Vector3f(x, y, z));
    tableBase.setTransform(change);

    return tableBase;
}
```

4.3 Lighting the scene

Lights are used in the 3D scene to illuminate the objects within the scene. As discussed previously, a light is not a physical object but it does affect other objects with the scene. For the sake of this application the use of ambient lights and directional lights will be used.

4.3.1 Ambient lights

An ambient light gives an even distribution of light in every direction and does not attenuate as the objects get further away from the light source. An example of an ambient light would be on a sunny day where the sun's light gives an even distribution in all directions. *Example 4-7* shows how to create an ambient light in the scene.

Example 4-7 – Creating an ambient light

```
TransformGroup lights = new TransformGroup();

// Set the lights bounding area to everywhere
BoundingSphere boundingSphere = new BoundingSphere(
             new Point3d(0.0,0.0,0.0), Double.POSITIVE_INFINITY);

// Create an ambient light at 40%
Color3f ambientColour = new Color3f(0.4f, 0.4f, 0.4f);
AmbientLight ambientLight = new AmbientLight(ambientColour);
ambientLight.setInfluencingBounds(boundingSphere);
lights.addChild(ambientLight);
```

4.3.2 Directional lights

Directional lights are similar to ambient lights except they have different strengths depending on the surface's direction from the light source. A directional light does not attenuate as the objects get further away from the light source. *Example 4-8* shows how to create a directional light in the scene.

Example 4-8 – Creating a directional light

```
TransformGroup lights = new TransformGroup();

// Set the lights bounding area to everywhere
BoundingSphere boundingSphere = new BoundingSphere(
             new Point3d(0.0,0.0,0.0), Double.POSITIVE_INFINITY);

// Setup a white directional light
Color3f directionalColour = new Color3f(1.0f, 1.0f, 1.0f);
Vector3f direction = new Vector3f(0.2f, 0.6f, 0.4f);
DirectionalLight directionalLight =
                new DirectionalLight(directionalColour, direction);
directionalLight.setInfluencingBounds(boundingSphere);
lights.addChild(directionalLight);
```

4.4 Managing the view port

The view port is the point of view the user will see the 3D scene from. This could be considered to be the user's position within the 3D world or character's position should the application be a game. The view port will allow movement across the x and z directions along with rotation about the y-axis to turn, and rotation about the x-axis to look up and down. The view has two transform groups, the first to manipulate movement and turning and the second to manipulate the pitch without necessarily effecting how the user moves.

4.4.1 Moving the view port

To move the view in the x and z directions, it's original position must be read back from the *TransformGroup*. It is possible to create a new vector with the change that the view port must move to. When the change vector is multiplied with the original transformation, the view will move in the current direction along the changed axis. *Example 4-9* shows how to move the view forward by 10 units.

Example 4-9 – Moving the view forward by 10 units

```
// Create 2 transformation that will be multiplied together
Transform3D change   = new Transform3D();
Transform3D movement = new Transform3D();

// Create a new transform value by moving forward in the negative Z direction
movement.set(new Vector3d(0,0,-10));

// Read the current view port’s transformation
viewFromGroup.getTransform(change);

// Multiply the movement with the current transformation
change.mul(movement);

// Apply the new position back to the view transform group
viewFromGroup.setTransform(change);
```

Based on this method it is possible to move the view port in the x and y direction as well to give full 3D movement around the scene.

4.4.2 Turning left and right

Similar to moving along the x and z axis, it is possible to rotate the view by multiplying the current view transformation with the amount to rotate. *Example 4-10* shows how to rotate the view port left by 0.1 radians.

Example 4-10 – Rotating the view port left by 0.1 radians

```
// Create 2 transformation that will be multiplied together
Transform3D change   = new Transform3D();
Transform3D movement = new Transform3D();

// Create a new rotation by 0.1 radians about the Y-axis to add to the view’s
// current transformation
movement.set(new AxisAngle4d(0, 1, 0, 0.1));

// Read the current view port’s transformation
viewFromGroup.getTransform(change);

// Multiply the rotation with the current transformation
change.mul(movement);

// Apply the new rotation back to the view transform group
viewFromGroup.setTransform(change);
```

The important thing to note is that after this transformation, moving along the z-axis will move in the direction of the rotation. The view port matrix has been rotated to generate a new x, y and z direction.

4.4.3 Looking up and down

Changing the pitch to look up and down will work the same as rotating about the y-axis to turn left and right. Instead of rotating about the y, to change the pitch it is necessary to rotate about the x-axis. One thing to note, it may be desirable to move along the new pitch, for example, in a flight simulator, changing the airplane's pitch results in changing the height of the view when moving forwards or backwards. In the case of this application, changing the pitch will only look up and down, but moving forwards or backwards will not change the height. An example of this would be a person lookup up at the sky, but when they walk forward they do not get closer to the sky. To achieve this we do not want to rotate the *viewFromGroup* about the x-axis but to rotate a child transform group called *viewPitchGroup* instead. This results in any movement transformations remaining unaffected by the pitch. *Example 4-11* shows how to rotate the pitch.

Example 4-11 – Rotating the view's pitch

```
// Create 2 transformation that will be multiplied together
Transform3D change   = new Transform3D();
Transform3D movement = new Transform3D();

// Create a new rotation by 0.1 radians about the X-axis and add to the view's
// current transformation
movement.set(new AxisAngle4d(1, 0, 0, 0.1));

// Get the current pitch transformation
viewPitchGroup.getTransform(change);

// Multiply the rotation with the current transformation
change.mul(movement);

// Apply the new rotation back to the pitch transform group
viewPitchGroup.setTransform(movement);
```

4.4.4 Animating the view port scale change

When the user enters or exists the radius of the table, the view port animates to emphasise the change in scale. Moving the view's y position to the correct height and animating the field of view to a wider angle and back again over the space of one second achieves this. When each frame is rendered, the view port makes the necessary transformation to the view to ensure the animation matches when computers run at different speeds.

The animation routine simply stores the start height and field of view, along with the destination values and processes a linear animation over time. This means that at 50% of the time the height of the view port will be 50% between the start and end values. The field of view change is similar but actually animates twice. When the animation is at 50%, the field of view is at its maximum before it returns to the start point. *Example 4-12* shows how to modify the view's field of view to a desired value.

Example 4-12 – Modifying the view's field of view

```
// Change to a wide angle view
view.setFieldOfView(2.0);

// Change to a telephoto view
view.setFieldOfView(0.5);

// Change to a suitable field of view
view.setFieldOfView(1.0f);
```

4.5 Reacting when every frame is rendered

One of the most important aspects of a 3D application that incorporates a form of animation is animating when the user does not interact with the program. This could be as complex as a 3D character walking around the scene to a subtle effect where the movement decelerates when the user releases a key from the keyboard. Either way, the application needs to react when a frame has been rendered on screen to know when to animate the next frame. To do this the use of a *Behavior* object is required to monitor every frame change. The *CFrameElapse* class has been given the role of doing this as it extends the functionality of the *Behavior* class.

The class creates a *WakeupCriterion* object and assigns this to *WakeupOnElapsedFrames(0)*. The class function *processStimulus* will get executed when a frame has elapsed. It would be possible to change the parameter of *WakeupOnElapsedFrames* to a value other than zero to wake up at a different interval. For example, it could be set to wake up after 10 frames.

When the *processStimulus* function is called, the application can then process any animation such as view port movement and thus calculate any collision detection. An important note is that once a wake up criterion has been executed it would need to be reset to run again. In this case it would be necessary to call *this.wakeupOn(wakeupNextFrame)* at the end of every *processStimulus* call.

The *CFrameElapse* class has been designed to monitor the time taken between the render of two frames. This can be used to smooth out any animation should one computer work slower than another. This can mean than an animation designed to run for 10 seconds will run at that duration regardless of the client computer's speed.

4.6 Reacting to the user

The application has been designed to react to the user via a keyboard input. While it is possible to react to mouse or joystick input, the keyboard is the easiest method for the user and for developers to implement. The *Canvas3D* object extends the functionality of the java *Component* object that allows a key listener to be implemented. *Example 4-13* shows how to create a key listener to react to the keyboard.

Example 4-13 – Creating a key listener for the Canvas3D object

```
public void load3DWorld()
{
  ...

  // Add a keyboard listener to the Canvas3D to respond to user input
  canvas3D.addKeyListener(this);

  ...
}

public void keyPressed(KeyEvent e)
{
  switch(e.getKeyCode())
  {
    case KeyEvent.VK_UP:
       // Up key pressed
       break;
    case KeyEvent.VK_DOWN:
        // Down key pressed
       break;
  }
}

// This is not used but must be implemented
public void keyTyped(KeyEvent e) {}

public void keyReleased(KeyEvent e)
{
  switch(e.getKeyCode())
  {
    case KeyEvent.VK_UP:
      // Up key released
      break;
    case KeyEvent.VK_DOWN:
      // Down key released
      break;
  }
}
```

It is important to note that in order to implement a key listener to the *Canvas3D* object, the class that implements the *keyPressed*, *keyTyped* and *keyReleased* functions must extend the *KeyListener* class.

4.7 Building the maze

The maze has been generated using similar techniques to building the room. The maze is a single object generated from an *IndexedQuadArray*.

4.7.1 Defining the maze

The definition of the maze has been created from a static array to define where the walls and pathways are. *Example 4-14* shows how the maze was defined using an array of integers.

Example 4-14 – Defining the maze using an array

```
private static final int mazeDef[] = {
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,0,
  0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,1,0,
  0,1,0,1,1,1,1,1,1,0,1,1,1,1,0,1,0,1,1,0,
  0,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,1,0,
  0,1,1,1,0,1,1,1,1,1,1,1,0,0,1,0,0,0,1,0,
  0,1,0,1,0,1,0,1,0,0,0,1,0,0,1,0,1,0,1,0,
  0,1,0,1,1,1,0,1,0,1,0,1,1,0,1,1,1,0,1,0,
  0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,
  0,1,1,0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,0,
  0,1,0,0,1,0,0,0,0,0,0,1,0,0,0,0,1,0,1,0,
  0,1,0,1,1,0,1,0,1,0,1,1,1,1,0,1,1,0,1,0,
  0,1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,1,0,
  0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,
  0,1,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,
  0,1,0,0,1,0,0,0,1,1,1,1,1,0,1,1,1,0,1,0,
  0,1,0,1,1,1,1,0,1,0,0,0,1,0,0,0,1,0,1,0,
  0,1,0,1,0,1,1,0,1,0,1,0,1,1,1,1,1,0,1,0,
  0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,1,0,
  0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
};

// Set the maze dimensions
private static final int width = 20;
private static final int height = 21;
```

Notice that the maze definition contains a boundary of empty spaces around the edge. This is to allow the generation and collision detection routines to have space to work with when working at the edge of the maze.

4.7.2 Creating the maze surfaces

As the maze is generated from an *IndexedQuadArray* the two most important elements that need to be generated are the physical coordinates in 3D space, and the indexes to these coordinates to generate the required maze surfaces.

Creating the coordinates is relatively straight forward as the maze is a two level grid. The base level defines the coordinates of the grid when flat along the table surface and the second level defines the coordinates of the grid when raised above the table surface. *Figure 4-4* shows an example of this two level grid comprised of coordinates where a cube can be generated by indexing those points. *Example 4-15* shows how to create the coordinates for each possible point in the maze.

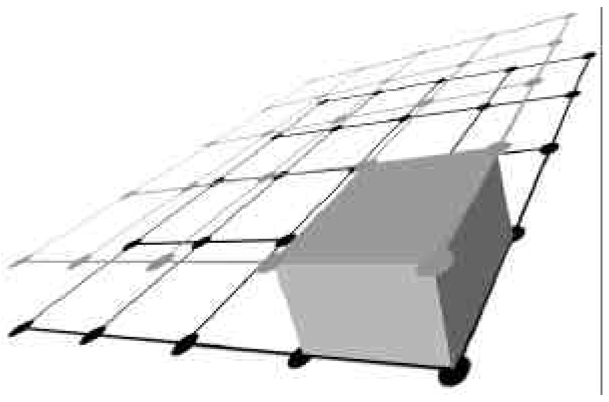


Figure 4-4 – An index array used to define the maze surfaces

Example 4-15 – Generating the maze coordinates

```
// Generate all the possible coordinates that could be used to
// create the maze.  These coordinates will then be indexed to define the
// required surfaces.

int x          = 0;
int y          = 0;
int indexPoint = 0;

// Loop through each edge in the row
for(y = 0; y <= height; y++)
{
  // Loop through each edge in the column for this row
  for(x = 0; x <= width; x++)
  {
    // Define the coordinate for this point in the x,y maze
    maze.setCoordinate(indexPoint, new
                 Point3f(x * this.blockWidth, 0.0f, y * this.blockDepth));
    indexPoint++;
  }
}
```

```
// Repeat the above routine except position the coordinates higher in the Y axis
for(y = 0; y <= height; y++)
{
  for(x = 0; x <= width; x++)
  {
    maze.setCoordinate(indexPoint, new
           Point3f(x * this.blockWidth, this.scaleHeight, y * this.blockDepth));
    indexPoint++;
  }
}
```

Once each point that could be used by the maze has been defined, it then necessary to only index the points required for the maze walls. To do this, the application works along the maze array and determines whether or not to generate a roof surface for the block and if there is a surface required for the right and bottom direction. It does not test for the top and left surfaces because the previous block will have checked these. *Example 4-16* shows how to determine which surfaces to generate based on the maze array definition.

Example 4-16 – Generating the maze surfaces

```
// Loop through each maze block and generates the necessary top, right and
// bottom surfaces

int x = 0;
int y = 0;

// Loop through the maze rows
for(y = 0; y < height; y++)
{
  // Loop through the maze columns for this row
  for(x = 0; x < width; x++)
  {
    // Make sure we are not at the right edge
    if (x + 1 < width && y + 1 < height)
    {
      if(mazeDef[(y*width)+x] == 0) // currently no block
      {
        if(mazeDef[(y*width) + (x+1)] == 1) // the one to the right is a block
       {
          drawWestFacingWall(x+1,y);
       }
       if(mazeDef[((y+1)*width) + x] == 1) // the one to the bottom is a block
        {
          drawNorthFacingWall(x,y+1);
        }
      }
      else // there is a block here
      {
        if(mazeDef[(y*width) + (x+1)] == 0) // the one to the right is no block
        {
          drawEastFacingWall(x,y);
        }
        if(mazeDef[(((y+1)*width) + x)] == 0) // there is none to the bottom
        {
          drawSouthFacingWall(x,y);
        }
        // Draw the top of this block
        drawBlockTopSurface(x,y);
      }
    }
  }
}
```

The functions *drawNorthFacingWall*, *drawSouthFacingWall*, *drawWestFacingWall*, *drawEastFacingWall* and *drawBlockTopSurface* simply join the coordinates together to build a quad surface. These functions determine the four points required to join and their order to determine which side the surface faces. Once these have been determined they call the *drawSurface* function to tell the *IndexedQuadArray* to build a surface for those coordinates. *Example 4-17* shows how to add the coordinate indices to the *IndexedQuadArray*.

Example 4-17 – Create a surface by defining the coordinate indices

```
private void drawSurface(int p1, int p2, int p3, int p4)
{
  // This function joins the 4 points together to create a quad surface
  maze.setCoordinateIndex(indexCount, p1);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p2);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p3);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p4);
  indexCount++;
}
```

4.7.3 Creating the maze normals

As mentioned previously, to enable a surface to react to lighting and give a more realistic appearance it is necessary to add the surface normals. As the blocks that we are creating is ultimately a cube with six sides, there are only six normals that require defining. *Example 4-18* shows the definition of the normals for the maze blocks.

Example 4-18 – Defining the maze normals

```
// Store the normals for each surface for the maze
private static final Vector3f normals[] = {
  new Vector3f( 0.0f,  0.0f,  1.0f), // North facing
  new Vector3f( 0.0f,  0.0f, -1.0f), // South facing
  new Vector3f( 1.0f,  0.0f,  0.0f), // West facing
  new Vector3f(-1.0f,  0.0f,  0.0f), // East facing
  new Vector3f( 0.0f,  1.0f,  0.0f), // Down facing
  new Vector3f( 0.0f, -1.0f,  0.0f), // Up facing
};
```

For every surface that gets created it is important to state which normal is being used when the points are added. *Example 4-19* shows how to index the normals for each point added when a surface is created.

Example 4-19 – Indexing the normal for each surface point

```
private void addNormals(int index)
{
  // This function adds the correct normal for each point in this surface
  for(int count = 0; count < 4; count++)
  {
    maze.setNormalIndex(normalCount, index);
    normalCount++;
  }
}
```

The *index* parameter is the index in the *normals* vector array. For example setting with the value zero results in a North-facing vector along the z-axis whereas value 3 would result in an East-facing vector along the x-axis.

4.7.4 Texturing the maze

Texturing the maze blocks are somewhat simpler when compared to the room. Firstly all the textures will be the same so it will not be necessary to break the shape into smaller shapes and texture those separately. Secondly the texture will fix exactly once for each block meaning there is no need to scale the texture or repeat it across any one surface. As before, it is necessary to define the texture coordinates to state how the texture will be applied to each point along the surface. *Example 4-20* shows how to define the texture coordinates for the maze.

Example 4-20 – Defining the maze texture coordinates

```
private static final TexCoord2f textureCoords[] = {
  new TexCoord2f(0.0f, 0.0f), // bottom left
  new TexCoord2f(1.0f, 0.0f), // bottom right
  new TexCoord2f(1.0f, 1.0f), // top right
  new TexCoord2f(0.0f, 1.0f)  // top left
};
```

It is now necessary to index these coordinates for every surface that is created. *Example 4-21* shows a modified example of *drawSurface* that now includes how to index the texture coordinates.

Example 4-21 – Indexing the texture coordinates when generating a surface

```
private void drawSurface(int p1, int p2, int p3, int p4)
{
  // This function joins the 4 points together to create a quad surface
  maze.setCoordinateIndex(indexCount, p1);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p2);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p3);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p4);
  indexCount++;

  // Add the texture coordinate for each point in this surface
  for(int coordIndex = 0; coordIndex < 4; coordIndex++)
  {
    maze.setTextureCoordinateIndex(0, textureCount, coordIndex);
    textureCount++;
  }
}
```

4.8 Collision detection

One of the most challenging aspects of a 3D application is the use of collision detection. Within this application there are three objects that will react to collision. The first will be the room to ensure that the user does not simply walk through the wall in empty space. The second will be to detect when the user enters or exits the radius of the table area. Lastly the maze will include collision detection to ensure that the user does not walk through the maze walls.

There are two main methods to react to a collision when it has been detected. The first method would be to reject the transformation and stop the user from moving. This can be somewhat limited in usability, as the movement would freeze should the user collide with an object. The second method would be to correct the movement to ensure the user continues to move but within the bounding area that they are allowed to move in.

Once the *CViewport* class has made a transformation along the x or z axis, this is considered as a movement and thus any collisions will need to be detected. It is not necessary to test for collision should the user turn or look up or down since they are not changing their position, only orientation.

Before a test for collision can be made, the view port’s world position needs to be identified. Fortunately Java 3D contains the ability to take a *TransformGroup* and identity its position in relation to the world matrix. *Example 4-22* shows how to retrieve the view port's position coordinates wherever it is within the 3D scene.

Example 4-22 – Retrieving the view port's position

```
// Create a Transform3D to read the view's transform from the world origin
Transform3D xform = new Transform3D();

// Get the transformation from the origin to the viewport
viewPlatform.getLocalToVworld(xform);

// Get the x,y,z coordinates for this position
xform.get(currentPosition);

currentX = currentPosition.x;
currentY = currentPosition.y;
currentZ = currentPosition.z;
```

Based on these coordinates, it is possible to test for collisions for the room, table and maze. If the position is modified to correct the view's location it is possible to set this back to the view transform group. One important thing to bear in mind is that it is only the position that requires updating, as all the existing rotations and scale (if necessary) would need to remain as they are. *Example 4-23* shows how to set the view's position while retaining all other transformation information.

Example 4-23 – Setting the view's position without affecting the rotation or scale

```
// Create a transform 3D that will be a compilation of the new position
// and the existing rotation and scale
Transform3D restore = new Transform3D();

// Get the current transformation of the view
viewFromGroup.getTransform(restore);

// Set the new position where newPosition is a vector
restore.setTranslation(newPosition);

// Apply the new transformation back to the view
viewFromGroup.setTransform(restore);
```

4.8.1 Colliding with the room walls

To test to see if the user has collided with the room walls, a test is made between the view port's position against the boundaries of the room. Firstly the x-axis is tested and corrected if necessary and then the z-axis is tested. It is not necessary to test the y-axis since the user cannot control the y position of the view. *Example 4-24* shows a function that checks to see if the user's position is outside the room boundaries.

Example 4-24 – Testing for room collisions and correcting if necessary

```
private boolean outsideRoom(Vector3f position)
{
  // This function checks to see if the user's position is outside the
  //room boundaries
  boolean bCollide = false;

  // Test to see if the user has tried to exit the room along the x axis
  if(position.x < roomX + LARGE_WALL_DISTANCE)
  {
    position.x = roomX + LARGE_WALL_DISTANCE;
    bCollide = true;
  }
  else if(position.x > roomX + roomWidth - LARGE_WALL_DISTANCE)
  {
    position.x = roomX + roomWidth - LARGE_WALL_DISTANCE;
    bCollide = true;
  }

  // Test to see if the user has tried to exit the room along the z axis
  if(position.z < roomZ + LARGE_WALL_DISTANCE)
  {
    position.z = roomZ + LARGE_WALL_DISTANCE;
    bCollide = true;
  }
  else if(position.z > roomZ + roomDepth - LARGE_WALL_DISTANCE)
  {
    position.z = roomZ + roomDepth - LARGE_WALL_DISTANCE;
    bCollide = true;
  }

  return bCollide;
}
```

Notice that the constant *LARGE_WALL_DISTANCE* is used to give a small gap between the view port and the surface of the wall. This is used to eliminate any clipping through the wall, as the wall surface may be closer than the minimum clipping distance of the view.

4.8.2 Entering and exiting the table

To identify whether the user is on the table or off the table, the position is compared against the centre point of the round table. Using Pythagoras' theorem it is possible to determine the distance between the current x and z points against the table centre x and z points. If this distance is less than the table radius then the user is on the table surface. If it is greater then the user is outside the table area. By creating a Boolean variable it is possible to determine whether a user has just entered or exited the table area and then animate the view accordingly to shrink or enlarge the view port.

4.8.3 Maze collision detection

The maze collision detection is broken down into parts. The first part is to determine whether a user has walked into a wall of the maze. This is achieved by converting the view port's position into a grid value of the maze definition array. If the value of this grid is a '1' then the user is positioned in a wall and thus, a collision has been detected. *Example 4-25* shows how to detect whether a user has collided with a wall.

Example 4-25 – Identifying a collision within the maze

```
public boolean checkCollisionDetection(Vector3f position)
{
  // This function tests the specified vector and sees if it collides with any
  // of the maze walls
  boolean bXCollision = false;
  boolean bZCollision = false;

  // First make sure the user is within the maze grid
  if(position.x > posX && position.x < posX + scaleWidth &&
     position.z > posZ && position.z < posZ + scaleDepth)
  {
    // Check if there is a collision in the X direction
    bXCollision = hitXWall(position.x + SMALL_WALL_DISTANCE, position);
    if(!bXCollision)
      bXCollision = hitXWall(position.x - SMALL_WALL_DISTANCE, position);
    // Check if there is a collision in the Z direction
    bZCollision = hitYWall(position.z + SMALL_WALL_DISTANCE, position);
    if(!bZCollision)
      bZCollision = hitYWall(position.z - SMALL_WALL_DISTANCE, position);

  }

  // Store this position as it may be used to place the viewport into
  // if the user collides with a wall on the next check
  int lastValidX = (int)(((position.x - posX) / scaleWidth) * width);
  int lastValidY = (int)(((position.z - posZ) / scaleDepth) * height);

  // Return whether there was a collision
  return bXCollision | bZCollision;
}

private boolean hitXWall(float x, Vector3f position)
{
  // This function tests to see if the position intersects with a block
  // along the X axis.  If it does, it corrects the position to force
  // the user back into a valid block

  boolean bCollision = false;
  // Convert the coordinate x into a grid value
  int gridX = (int)(((x - posX) / scaleWidth) * width);
  int gridY = (int)(((position.z - posZ) / scaleDepth) * height);

  // Make sure that these coordinates are within the grid
  if(gridX >= width || gridY >= height)
    return false;

  // See if we have collided with a wall
  if(mazeDef[(gridY * width) + gridX] == 1)
  {
    // There has been a collision so the best thing to do is
    // move the viewport to the nearest valid edge
    moveXTo(lastValidX, gridX, position);
    bCollision = true;
  }
  return bCollision;
}

private boolean hitYWall(float z, Vector3f position)
{
  // This function tests to see if the position intersects with a block
  // along the Z axis.  If it does, it corrects the position to force
  // the user back into a valid block

  boolean bCollision = false;
  // Convert the coordinate x into a grid value
  int gridX = (int)(((position.x - posX) / scaleWidth) * width);
  int gridY = (int)(((z - posZ) / scaleDepth) * height);

  // Make sure that these coordinates are within the grid
  if(gridX >= width || gridY >= height)
    return false;

  // See if we have collided with a wall
  if(mazeDef[(gridY * width) + gridX] == 1)
  {
    // There has been a collision so the best thing to do is
    // move the viewport to the nearest valid edge
    moveYTo(lastValidY, gridY, position);
    bCollision = true;
  }
  return bCollision;
}
```

Upon detecting a collision the position needs to be corrected to ‘slide’ the user back into a valid block in the maze. This is achieved using the two functions *moveXTo* and *moveYTo* that moves the position to the closest point in the valid wall. *Example 4-26* shows how to correct the maze collision.

Example 4-26 – Correcting the maze collision

```
private void moveXTo(int toX, int fromX, Vector3f position)
{
  // This moves the X position into a valid block
  if(toX > fromX) // LHS of toX
  {
    position.x = (((float)toX / width) * scaleWidth) + posX +
                                           SMALL_WALL_DISTANCE;
  }
  else // RHS of toX
  {
    position.x = (((float)(toX + 1) / width) * scaleWidth) + posX –
                                           SMALL_WALL_DISTANCE;
  }
}

private void moveYTo(int toY, int fromY, Vector3f position)
{
  // This moves the Y position into a valid block
  if(toY > fromY) // top of toY
  {
    position.z = (((float)toY / height) * scaleDepth) + posZ +
                                           SMALL_WALL_DISTANCE;
  }
  else // bottom of to Y
  {
    position.z = (((float)(toY + 1) / height) * scaleDepth) + posZ –
                                                SMALL_WALL_DISTANCE;
  }
}
```

This method of collision detection may not produce perfect results in certain situations. For example, if the user tried to move diagonally across an invalid block the view port may ‘jolt’ across giving a somewhat sharp movement. For most common collisions the user may make, this method does produce very acceptable results and performance.

5.0 Testing

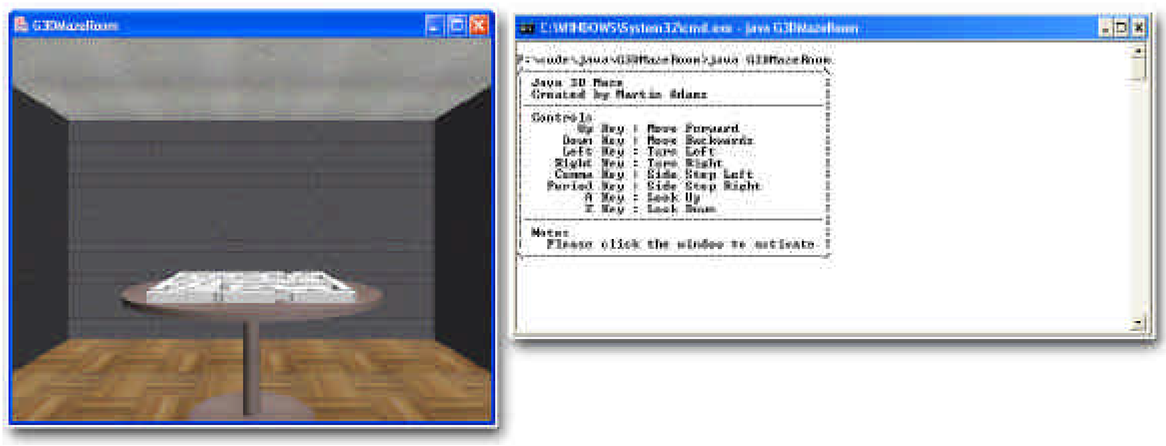
In order to run the application the client computer must have the Java Runtime Environment (JRE) 1.4.1 installed and the Java 3D OpenGL Runtime Environment 1.3 installed. Both these are freely available from Sun Microsystem's website.

5.1 Running the application

To run the application from the command prompt it is important that the java runtime files can be found in the path. After locating the directory of the *G3DMazeRoom.class* application file simply type:

```
java G3DMazeRoom
```

In the console window, information text should appear and the application should load. *Figure 5-1* shows the application loading for the first time on a Window XP operating system.



```
:\code\java\G3DMazeRoom>java G3DMazeRoom
 Java 3D Maze
 Created by Martin Adams
 ------------------------------------
 Controls
        Up Key : Move Forward
      Down Key : Move Backwards
      Left Key : Turn Left
     Right Key : Turn Right
     Comma Key : Side Step Left
    Period Key : Side Step Right
         A Key : Look Up
         Z Key : Look Down
 ------------------------------------
 Notes
   Please click the window to activate
```

Figure 5-1 – Running the G3DMazeRoom application

5.2 Testing the application

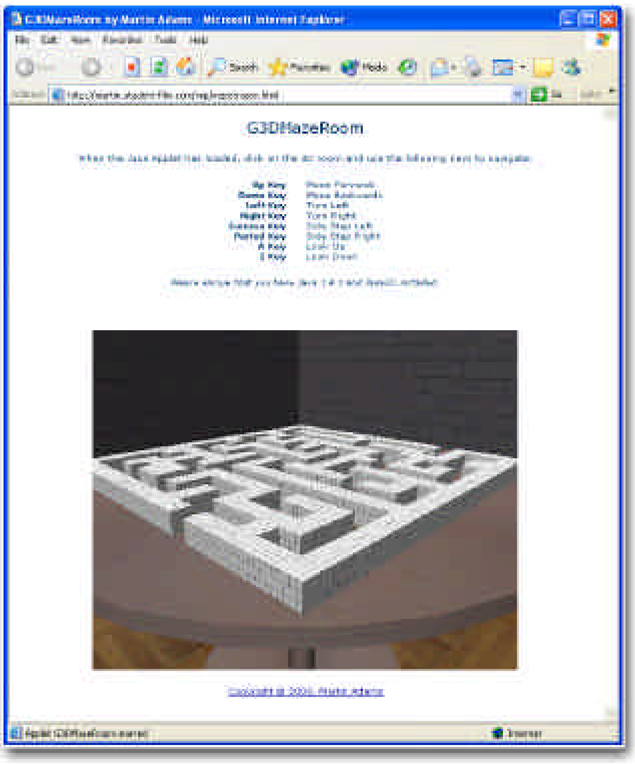
The application has been tested to check for movement errors, collision errors and visual errors. While there have been no major problems with the application is has been evident that the movement controls may be somewhat sensitive when run on different computers.

Occasionally the application may slow down when running but this may be due to other processes running on the client computer or the Java garbage collector cleaning up resulting in a slight pause.

Nevertheless, the application functioned as expected and does not suffer from any unexpected problems. The only problem may occur if the textures are not located in the '\textures' sub directory from the class files.

5.3 Cross platform testing

The application has been tested on a variety of computer systems and has shown to function correctly. While the application has been designed to run as an applet as well as a program run from the command prompt, there have been a few unexpected results where the applet will fail to locate all the necessary class files on certain clients. It seems that this may be caused by a difference in web browsers and is not the fault of the application. *Figure 5-2* show the application running from a web browser.



[illegible]

Up Key	Move Forwards
Down Key	Move Backwards
Left Key	Turn Left
Right Key	Turn Right
Delete Key	Side Step Left
Page Down Key	Side Step Right
A Key	Look Up
Z Key	Look Down

[illegible]

Figure 5-2 – Running G3DMazeRoom from a web browser

6.0 Conclusion and recommendations

This has been a great project to explore the functionality of Java 3D and OpenGL. The project has been a great success in the ability to generate geometry in a 3D scene, respond the user interaction and provide advanced functionality such as collision detection. The application was originally designed to react in a natural way with smooth navigation through the scene, and while this involved further complexity in collision detection, the results were extremely rewarding.

The major problems that occurred in the application were related to collision detection of the maze. The original plan was to detect if the user had collided with a wall and simply reject that move. This proved to be somewhat undesirable as it would become difficult to move through the maze and the user would not be allowed to touch a wall without the navigation freezing. The solution was to reject this idea and correct any movement instead. This added complexity to the application but the results allowed the user to naturally navigate their way through the maze at their own pace.

Improvements to this application would be to explore the use of adding 3D objects from file. These can be objects created from party applications such as 3D Studio Max or LightWave 3D. Animation of 3D objects would be extremely useful as the application could be turned into a game with an animated character navigating their way through the maze.

Java also supports the use of sound, which can add an extra level of entertainment by having sound effects in relation to what the user is doing along with background music.

While the maze is generated from an array that can be modified in the source code, it would be useful to generate the maze from a separate file to incorporate multiple mazes at different difficulty levels to the user. The results of such modifications would head towards a fully functional and enjoyable 3D game.

References

Dennis J Bouvier, *(2000)*. *Getting Started with Java 3D.* Sun Microsystems, Inc.

Bibliography

Dennis J Bouvier. *(2001) Getting Started with the Java 3D API* [online]
Available from: http://java.sun.com/products/java-media/3D/collateral [27/3/2003]

Jon Barrilleaux. *(2001) 3D User Interfaces with Java 3D*. Manning Publications Co.

Mason Woo, *et al. (1999) OpenGL Programming Guide Third Edition*.
Silicon Graphics, Inc.

Sun Microsystems, Inc. *(2002) Java 2 Platform, Standard Edition, v1.4.0 API Specification.* Sun Microsystems, Inc.

Sun Microsystems, Inc. *Java 3D API* [online]
Available from: http://java.sun.com/products/java-media/3D/forDevelopers/
J3D_1_3_API/j3dapi/ [27/3/2003]

Appendices

Appendix A – G3DMazeRoom Source Code

G3DMazeRoom Source Code

```
/****************************************
* G3DMazeRoom
* Author: Martin Adams
****************************************
* Main application to create 3D world
* and monitor keyboard input
****************************************/

import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.GraphicsConfiguration;

import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;

import javax.media.j3d.*;
import javax.vecmath.*;
import java.awt.event.*;

public class G3DMazeRoom extends Applet
implements KeyListener, CFrameElapseListener, CViewportListener, CWorldListener
{
  private Canvas3D      canvas3D = null;
  private CWorld        world = null;
  private CViewport     view = null;
  private CFrameElapse  frameElapse = null;
  private String        appPath = "LOCAL";

  public G3DMazeRoom(String appPath)
  {
    // Load the program using this new path
    this.appPath = appPath;
    load();
  }

  public G3DMazeRoom()
  {
    // The applet has been loaded from the web so set the location to the images
    this.appPath = "http://martin.student-film.com/wip/maze/";
    load();
  }

  public void load()
  {
    // Create a virtual universe to contain our 3D world
    VirtualUniverse universe = new VirtualUniverse();
    Locale loc = new Locale(universe);
    GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();

    // Add the Canvas3D to the window
    setLayout(new BorderLayout());
    canvas3D = new Canvas3D(config);
    add("Center", canvas3D);

    // Add a keyboard listener to the Canvas3D to respond to user input
    canvas3D.addKeyListener(this);

    // Create our viewport
    BranchGroup viewport = new BranchGroup();
    view = new CViewport(canvas3D, viewport, this);
    loc.addBranchGraph(viewport);

    // Create our world
    BranchGroup scene = new BranchGroup();
    world = new CWorld(scene, canvas3D, this, appPath);

    // Monitor every frame elapse
    frameElapse = new CFrameElapse(this);
    BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0),
                                                Double.POSITIVE_INFINITY);
    frameElapse.setSchedulingBounds(bounds);
    scene.addChild(frameElapse);

    // Compile and add the scene
    scene.compile();
    loc.addBranchGraph(scene);

    // Move the viewport to a default position
    view.moveTo(250.0f, 150.0f, 600.0f);
  }

  public void keyReleased(KeyEvent e)
  {
    switch(e.getKeyCode())
    {
      case KeyEvent.VK_UP:
        view.moveForward();
        break;
      case KeyEvent.VK_DOWN:
        view.moveForward();
        break;
      case KeyEvent.VK_LEFT:
        view.moveYaw();
        break;
      case KeyEvent.VK_RIGHT:
        view.moveYaw();
        break;
      case KeyEvent.VK_COMMA:
        view.moveSideways();
        break;
      case KeyEvent.VK_PERIOD:
        view.moveSideways();
        break;
      case KeyEvent.VK_A:
        view.movePitch();
        break;
      case KeyEvent.VK_Z:
        view.movePitch();
        break;
    }
  }

  public void keyTyped(KeyEvent e) {}

  public void keyPressed(KeyEvent e)
  {
    switch(e.getKeyCode())
    {
      case KeyEvent.VK_UP:
        view.moveForward(-1f);
        break;
      case KeyEvent.VK_DOWN:
        view.moveForward(1f);
        break;
      case KeyEvent.VK_LEFT:
        view.moveYaw(0.05f);
        break;
      case KeyEvent.VK_RIGHT:
        view.moveYaw(-0.05f);
        break;
      case KeyEvent.VK_COMMA:
        view.moveSideways(-1f);
        break;
      case KeyEvent.VK_PERIOD:
        view.moveSideways(1f);
        break;
      case KeyEvent.VK_A:
        view.movePitch(0.05f);
        break;
      case KeyEvent.VK_Z:
        view.movePitch(-0.05f);
        break;
    }
  }

  public void onFrameElapse()
  {
    // A frame has advanced so get the view to update
    // any animation
    view.updateViewport(frameElapse.mElapsedTime);
  }

  public void onEnterTable()
  {
    // The user has entered the table area so get the
    // viewport to shrink to the smaller scale
    view.shrinkToHeight(105.0f);
  }

  public void onExitTable()
  {
    // The user has left the table area so get the
    // viewport to enlarge to the normal scale
    view.enlargeToHeight(150.0f);
  }

  public void onCollisionDetect(Vector3f newPosition)
  {
    // There was a collision with an object so set the
    // viewport's position to the corrected value
    view.boundMove(newPosition);
  }

  public void onViewportMove(Vector3f position)
  {
    // The viewport's position was moved so check for
    // any collision with any objects in the world
    world.processMovement(position);
  }

  public static void main(String[] args)
  {
    // The program has been loaded from the command prompt so assume
    // the textures are relative to the current path

    // Load the applet in a window
    Frame frame = new MainFrame(new G3DMazeRoom(""), 500, 400);

    // Show the welcome message
    System.out.println("/---------------------------------------\\");
    System.out.println("| Java 3D Maze                          |");
    System.out.println("| Created by Martin Adams               |");
    System.out.println("|---------------------------------------|");
    System.out.println("| Controls                              |");
    System.out.println("|       Up Key : Move Forward           |");
    System.out.println("|     Down Key : Move Backwards         |");
    System.out.println("|     Left Key : Turn Left              |");
    System.out.println("|    Right Key : Turn Right             |");
    System.out.println("|    Comma Key : Side Step Left         |");
    System.out.println("|   Period Key : Side Step Right        |");
    System.out.println("|        A Key : Look Up                |");
    System.out.println("|        Z Key : Look Down              |");
    System.out.println("|---------------------------------------|");
    System.out.println("| Notes                                 |");
    System.out.println("|   Please click the window to activate |");
    System.out.println("\\---------------------------------------/");
  }
}
```

CFrameElapse Source Code

```
/****************************************
* CFrameElapse
* Author: Martin Adams
****************************************
* This class sets a behaviour to monitor
* every time a frame has been rendered
```

```
****************************************/

import javax.media.j3d.*;
import java.util.Enumeration;
import java.util.Calendar;

public class CFrameElapse extends Behavior
{
  // Create a wakeup criterion that will be used to
  // wake up every time a frame has been rendered
  private WakeupCriterion       wakeupNextFrame;

  // Store the link to the parent class
  private CFrameElapseListener   listener = null;

  // Store the time variables to montior how long it
  // took to render the frames.  This is designed to
  // help make any animation take the same amount of
  // time on slower and faster computers
  private long     mCurrentTime   = 0;
  private long     mLastFrameTime = 0;
  public  float    mElapsedTime   = 0;
  private Calendar cal;

  CFrameElapse(CFrameElapseListener l)
  {
    // Store the parent class link
    listener = l;
    // Set the behaviour to wakeup every time a frame is rendered
    wakeupNextFrame = new WakeupOnElapsedFrames(0);
  }

  public void initialize()
  {
    // Activate the wakeup
    this.wakeupOn(wakeupNextFrame);
  }

  public void processStimulus(Enumeration criteria)
  {
    // This function is called every time a frame elapses

    // Calculate the time difference
    cal            = Calendar.getInstance();
    mCurrentTime   = cal.getTimeInMillis();
    mElapsedTime   = (mCurrentTime - mLastFrameTime) / 10;
    mLastFrameTime = mCurrentTime;

    if(criteria.nextElement().equals(wakeupNextFrame))
    {
      // Call the onFrameElapse function to inform the parent
      // class that a frame has elapsed
      listener.onFrameElapse();
    }
    // Tell the behaviour to wake up again on the next frame
    this.wakeupOn(wakeupNextFrame);
  }
}
```

CFrameElapseListener Source Code

```
/****************************************
* CFrameElapseListener
* Author: Martin Adams
****************************************
* Defines the frame elapse callback
* procedures
****************************************/

public interface CFrameElapseListener
{
  public void onFrameElapse();
}
```

CViewport Source Code

```
/***************************************
* CViewport
* Author: Martin Adams
****************************************
* Moves the viewport to a correct
* locatoin
****************************************/

import javax.media.j3d.*;
import javax.vecmath.*;

public class CViewport
{
  // Store the amount the strength of movement decelerates when
  // the user releases the key from the keyboard
  private static final float MOVE_DECELERATION     = 0.6f;
  private static final float ROTATION_DECELERATION = 0.5f;

  // Store the amount the user moves when in a large scale mode
  // or when shrunk to the scale of the maze
  public static final float LARGE_SCALE_MOVEMENT = 1.0f;
  public static final float SMALL_SCALE_MOVEMENT = 0.15f;

  // Store the maximum field of view angle and the normal field
  // of view angle used when animating the scale change
  public static final float MAX_FOV    = 2.0f;
  public static final float NORMAL_FOV = 1.0f;

  // Store the necessary objects required to manipulate the view
  private View              view           = null;
  private BranchGroup       parent         = null;
  private TransformGroup     viewFromGroup  = new TransformGroup();
  private TransformGroup    viewPitchGroup = new TransformGroup();
  private ViewPlatform      viewPlatform   = new ViewPlatform();
  private CViewportListener listener       = null;

  // Store the default scale of movement
  private float scaleMovement = LARGE_SCALE_MOVEMENT;

  // Forward movement
  private float   mForwardMovement = 0;
  private boolean mMovingForward   = false;

  // Sideways movement
  private float   mSidewaysMovement = 0;
  private boolean mMovingSideways   = false;

  // Turn movement
  private float   mYawMovement = 0;
  private boolean mMovingYaw   = false;

  // Pitch movement
  private float   mPitchMovement = 0;
  private boolean mMovingPitch   = false;

  // Store the current position values of where the viewport is
  // relative the centre of teh world
  public float currentX = 0.0f;
  public float currentY = 0.0f;
  public float currentZ = 0.0f;
  public Vector3f currentPosition = new Vector3f();

  // Scale values when the user enters/exits the table
  private boolean bScaleViewport   = false;
  private float   scaleFromHeight  = 0.0f;
  private float   scaleToHeight    = 0.0f;
  private long    currentAnimFrame = 0;
  private static final long ANIMATION_LENGTH = 100; // 1 seconds

  public CViewport(Canvas3D canvas, BranchGroup parent, CViewportListener l)
  {
    this.parent = parent;
    this.listener = l;

    // Setup the vew platform
    viewPlatform.setCapability(ViewPlatform.ALLOW_LOCAL_TO_VWORLD_READ);
    PhysicalBody body = new PhysicalBody();
    PhysicalEnvironment env = new PhysicalEnvironment();

    // Set the view from and pitch transform groups to read/write
    viewFromGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    viewFromGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    viewPitchGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    viewPitchGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    // Add the view platform the the pitch TG
    viewPitchGroup.addChild(viewPlatform);
    // Add teh pitch TG to the view from group
    viewFromGroup.addChild(viewPitchGroup);
    // Add the view from TG to the world
    parent.addChild(viewFromGroup);

    // Create a enw view
    view = new View();
    view.addCanvas3D(canvas);
    view.attachViewPlatform(viewPlatform);
    view.setPhysicalBody(body);
    view.setPhysicalEnvironment(env);

    // Set the view's field of view
    view.setFieldOfView(NORMAL_FOV);

    // Set the back clip distance to suit the scale we are working in
    view.setBackClipDistance(300.0);
  }

  public void moveForward(float strength)
  {
    // Set the mode to moving forward and the strength
    mForwardMovement = scaleMovement * strength;
    mMovingForward   = true;
  }

  public void moveForward()
  {
    // Stop moving forward
    mMovingForward = false;
  }

  public void moveSideways(float strength)
  {
    // Set the mode to moving sideways and the strength
    mSidewaysMovement = scaleMovement * strength;
    mMovingSideways   = true;
  }

  public void moveSideways()
  {
    // Cancel the side stepping
    mMovingSideways = false;
  }

  public void moveYaw(float strength)
  {
    // Set the mode to turning left or right and the strength
    mYawMovement = strength;
    mMovingYaw   = true;
  }

  public void moveYaw()
  {
    // Cancel the turning
    mMovingYaw = false;
  }

  public void movePitch(float strength)
  {
    // Set the mode to looking up or down and the strength
    mPitchMovement = strength;
    mMovingPitch   = true;
  }

  public void movePitch()
  {
    // Cancel the looking up or down
    mMovingPitch = false;
  }

  public void moveTo(float x, float y, float z)
  {
    // This moves the viewport to a specified position
    Transform3D change = new Transform3D();
    change.set(new Vector3d(x, y, z));
    viewFromGroup.setTransform(change);
  }

  public void updateViewport(float elapsedTime)
  {
    // This function updates any change to thew viewport when
    // the each frame is rendered

    boolean bMove        = false;
    Transform3D change   = new Transform3D();
    Transform3D movement = new Transform3D();

    // Store the current position before moving
    //viewFromGroup.getTransform(restorePosition);

    // See if there is a scale animation in progress.
    // If so process it and ignore any movment the user
    // is trying to make.
    if(bScaleViewport)
    {
      processScaleAnimation(elapsedTime);
    }
    else
    {
      // See if the viewport needs moving forward
      if(mForwardMovement > 0.05f || mForwardMovement < -0.05f)
      {
        // Decelerate if the user released the key
        if(!mMovingForward) mForwardMovement *= MOVE_DECELERATION;

        // Create a new transform value
        movement.set(new Vector3d(0,0,elapsedTime * mForwardMovement));

        // Add it to the existing matrix
        viewFromGroup.getTransform(change);
        change.mul(movement);
        viewFromGroup.setTransform(change);

        bMove = true;
      }

      // See if the viewport needs moving sideways
      if(mSidewaysMovement > 0.05f || mSidewaysMovement < -0.05f)
      {
        // Decelerate if the user released the key
        if(!mMovingSideways) mSidewaysMovement *= MOVE_DECELERATION;

        // Create a new transform value
        movement.set(new Vector3d(elapsedTime * mSidewaysMovement, 0, 0));

        // Add it to the existing matrix
        viewFromGroup.getTransform(change);
        change.mul(movement);
        viewFromGroup.setTransform(change);

        bMove = true;
      }

      // See if the viewport needs rotating about the Y
      if(mYawMovement > 0.001f || mYawMovement < -0.001f)
      {
        // Decelerate if the user released the key
        if(!mMovingYaw) mYawMovement *= ROTATION_DECELERATION;

        // Get the current matrix transformation
        viewFromGroup.getTransform(movement);

        // Add the new rotation to this matrix
        change.set(new AxisAngle4d(0, 1, 0, elapsedTime * mYawMovement));
        movement.mul(change);
        viewFromGroup.setTransform(movement);
      }

      // See if the view port needs rotating about the X
      if(mPitchMovement > 0.001f || mPitchMovement < -0.001f)
      {
        // Decelerate if the user released the key
        if(!mMovingPitch) mPitchMovement *= ROTATION_DECELERATION;

        // Get the current matrix transform
        viewPitchGroup.getTransform(movement);
        //viewFromGroup.getTransform(movement);

        // Add the new rotaton to this matrix
        change.set(new AxisAngle4d(1, 0, 0, elapsedTime * mPitchMovement));
        movement.mul(change);
        //viewFromGroup.setTransform(movement);
        viewPitchGroup.setTransform(movement);
      }
    }

    // If the viewport's position changed then call the parent class's onViewportMove
    // procedture to check for any collisions with world objects
    if(bMove)
    {
      // Get the camera's position
      getPosition();
      listener.onViewportMove(currentPosition);
    }
  }

  private void processScaleAnimation(float elapsedTime)
  {
    // This changes the viewport's position for each animation frame when the
    // user enters/exits the table area

    boolean bEnlarge = false;

    // See if we are enlarging the viewport
    if(scaleFromHeight < scaleToHeight)
    {
      bEnlarge = true;
    }

    // Add the animation time and test for the last frame of the animation
    currentAnimFrame += (long)elapsedTime;
    if(currentAnimFrame > ANIMATION_LENGTH)
    {
      currentAnimFrame = ANIMATION_LENGTH;
      bScaleViewport = false;
    }

    // Move the height of the viewport
    float percent = (float)currentAnimFrame / (float)ANIMATION_LENGTH;
    float moveTo;
    if(bEnlarge)
    {
      moveTo = scaleFromHeight + (percent * (scaleToHeight - scaleFromHeight));
    }
    else
    {
      // Reverse the percentage as we are shrinking
      percent = 1 - percent;
      moveTo = scaleToHeight + (percent * (scaleFromHeight - scaleToHeight));
    }

    // Move to the new height
    Transform3D change = new Transform3D();
    Vector3f currentPos = new Vector3f();

    viewFromGroup.getTransform(change);
    change.get(currentPos);

    currentPos.y = moveTo;
    change.setTranslation(currentPos);
    viewFromGroup.setTransform(change);

    // Change the FOV through the animation
    float fov;
    if(percent > 0.5)  // keep expanding fov
    {
      // get the percent between 0 and 1
      fov = (float)(1 - ((percent - 0.5) * 2));
    }
    else // reset fov
    {
      // get the percent between 0 and 1
      fov = (percent * 2);
    }
    fov = NORMAL_FOV + (fov * (MAX_FOV - NORMAL_FOV));

    view.setFieldOfView(fov);
  }

  public void boundMove(Vector3f newPosition)
  {
    Transform3D restore = new Transform3D();

    // Get the current transformation and only reset the position and not the
    // rotation and scale
    viewFromGroup.getTransform(restore);
    restore.setTranslation(newPosition);
    viewFromGroup.setTransform(restore);
  }

  public void getPosition()
  {
    Transform3D xform = new Transform3D();

    // Get the transformation from the origin to the viewport
    viewPlatform.getLocalToVworld(xform);

    // Get the x,y,z coordinates for this position
    xform.get(currentPosition);

    currentX = currentPosition.x;
    currentY = currentPosition.y;
    currentZ = currentPosition.z;
  }

  public void shrinkToHeight(float y)
  {
    // This function sets the values to start the shink animation
    bScaleViewport   = true;
    currentAnimFrame = 0;
    scaleToHeight    = y;
    scaleFromHeight  = this.currentY;
    scaleMovement    = SMALL_SCALE_MOVEMENT;

    // Adjust the movment scale if the user has the key currenly pressed down
    mForwardMovement *= SMALL_SCALE_MOVEMENT;
  }

  public void enlargeToHeight(float y)
  {
    // This function sets the values to start the enlarge animation
    bScaleViewport   = true;
    currentAnimFrame = 0;
    scaleToHeight    = y;
    scaleFromHeight  = this.currentY;
    scaleMovement    = LARGE_SCALE_MOVEMENT;

    // Adjust the movment scale if the user has the key currenly pressed down
    mForwardMovement /= SMALL_SCALE_MOVEMENT;
  }
}
```

CViewportListener Source Code

```
/****************************************
* CViewportListener
* Author: Martin Adams
****************************************
* Defines the viewport callback
* procedures
****************************************/

import javax.vecmath.*;

public interface CViewportListener
{
  public void onViewportMove(Vector3f position);
}
```

CWorld Source Code

```
/***************************************
* CWorld
* Author: Martin Adams
****************************************
* Create all world objects and
* test for collisions
****************************************/

import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.TextureLoader;

import javax.media.j3d.*;
import javax.vecmath.*;
import java.net.URL;
import java.lang.Math;

public class CWorld
{
  private static final float LARGE_WALL_DISTANCE = 10.0f; // 10 cm

  private BranchGroup     parent       = null;
  private TransformGroup  worldTG      = new TransformGroup();
  private Canvas3D        parentCanvas = null;
  private CMaze           maze         = null;
  private CWorldListener  listener     = null;
  private String          appPath      = "";

  // Collision detection for the room
  private float roomX     = 0.0f;
  private float roomZ     = 0.0f;
  private float roomWidth = 0.0f;
  private float roomDepth = 0.0f;

  // Collision detection for the table
  private float   tableCentrePointX = 0.0f;
  private float   tableCentrePointZ = 0.0f;
  private float   tableRadius       = 0.0f;
  private boolean bOnTable          = false;

  public CWorld(BranchGroup parent, Canvas3D canvas, CWorldListener l, String appPath)
  {
    // Store the parameters into their respective global variables
    this.parent = parent;
    this.parentCanvas = canvas;
    this.listener = l;
    this.appPath = appPath;

    // Allow the world TG's transform to read and write
    worldTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    worldTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

    // Add the world TG to the universe
    parent.addChild(worldTG);

    // Create a new instance of the 3D maze object (this does not create the maze itself)
    maze = new CMaze(175.0f, 100.0f, 175.0f, 150.0f, 10.0f, 150.0f, canvas, appPath);

    // Create the world objects
    createWorld();
  }
```

```
public void processMovement(Vector3f position)
{
  // This function returns whether the specified position has activated any
  // collision detection or event such as enter/exit the table area

  // Check to see if the user is outside the room area
  if(outsideRoom(position))
  {
    // Call the onCollisionDetect function with the updated position
    listener.onCollisionDetect(position);
    return;
  }

  // Check to see if the user has moved on or off the table
  checkTableTransition(position);

  // Only check for maze collision when on the table
  if(bOnTable)
  {
    // Check the maze collision detection
    if(maze.checkCollisionDetection(position))
    {
    // Call the onCollisionDetect function with the updated position
      listener.onCollisionDetect(position);
      return;
    }
  }
}

private void checkTableTransition(Vector3f position)
{
  if(bOnTable)
  {
    // Check to see if the user is outside the table radius
    if (distanceBetween2Points(position.x, position.z, tableCentrePointX,
                                           tableCentrePointZ) > tableRadius)
    {
      bOnTable = false;
      listener.onExitTable();
    }
  }
  else
  {
    // Check to see if the user is inside the table radius
    if (distanceBetween2Points(position.x, position.z, tableCentrePointX,
                                           tableCentrePointZ) < tableRadius)
    {
      bOnTable = true;
      listener.onEnterTable();
    }
  }
}

private float distanceBetween2Points(float x1, float y1, float x2, float y2)
{
  // Return the distance between the two points using pythagoras
  return (float)(Math.sqrt(((x2-x1)*(x2-x1)) + ((y2-y1)*(y2-y1))));
}


private boolean outsideRoom(Vector3f position)
{
  // This function checks to see if the user's position is outside the room boundaries
  boolean bCollide = false;

  // Test to see if the user has tried to exit the room along the x axis
  if(position.x < roomX + LARGE_WALL_DISTANCE)
  {
    position.x = roomX + LARGE_WALL_DISTANCE;
    bCollide = true;
  }
  else if(position.x > roomX + roomWidth - LARGE_WALL_DISTANCE)
  {
    position.x = roomX + roomWidth - LARGE_WALL_DISTANCE;
    bCollide = true;
  }
```

```
    // Test to see if the user has tried to exit the room along the z axis
    if(position.z < roomZ + LARGE_WALL_DISTANCE)
    {
      position.z = roomZ + LARGE_WALL_DISTANCE;
      bCollide = true;
    }
    else if(position.z > roomZ + roomDepth - LARGE_WALL_DISTANCE)
    {
      position.z = roomZ + roomDepth - LARGE_WALL_DISTANCE;
      bCollide = true;
    }

    return bCollide;
  }

  public void createWorld()
  {
    // Create a 3D cube for the time being
    worldTG.addChild(createRoom(0.0f,0.0f,0.0f,500.0f,300.0f,700.0f));
    worldTG.addChild(createTable(250.0f, 0.0f, 250.0f, 200.0f, 100.0f));
    worldTG.addChild(createLights());
    worldTG.addChild(maze.generateMaze());
  }

  private TransformGroup createTable(float x, float y, float z, float diameter,
                                                                  float height)
  {
    // Create the TGs for the table sections
    TransformGroup tableTop = new TransformGroup();
    TransformGroup tableLeg = new TransformGroup();
    TransformGroup tableBase = new TransformGroup();

    // Store the table properties for detection for when user enters/exits the table
    this.tableCentrePointX = x;
    this.tableCentrePointZ = z;
    this.tableRadius       = diameter / 2;

    // Generate an appearance for this shape
    Color3f ambientColour  = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
    Color3f diffuseColour  = new Color3f(1.0f, 1.0f, 1.0f); // white
    Color3f specularColour = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
    Color3f emissiveColour = new Color3f(0.0f, 0.0f, 0.0f); // black

    // Create a material for the appearance
    Material material = new Material(ambientColour, emissiveColour,
                            diffuseColour, specularColour, 5.0f);

    // Create the object's appearance and load the wood texture
    Appearance appearance = new Appearance();
    appearance.setMaterial(material);
    appearance.setTexture(loadTexture("textures/pinewood_128.jpg"));

    // Set the texture's attributes to blend with the material properties
    TextureAttributes texAttributes = new TextureAttributes();
    texAttributes.setTextureMode(TextureAttributes.MODULATE);

    // Scale the texture to fit on the surface of the table 2x2
    Transform3D scaleTex = new Transform3D();
    scaleTex.setScale(2);
    texAttributes.setTextureTransform(scaleTex);

    // Set the textures attributes to the appearance
    appearance.setTextureAttributes(texAttributes);

    // Add a cylinder as the table top
    tableTop.addChild(new Cylinder(diameter / 2, 5.0f, Cylinder.GENERATE_NORMALS |
                            Cylinder.GENERATE_TEXTURE_COORDS, 30, 30, appearance));

    // Move the table top to the top of the leg
    Transform3D change = new Transform3D();
    change.set(new Vector3f(0.0f, height / 2, 0.0f));
    tableTop.setTransform(change);

    // Create the leg
    tableLeg.addChild(new Cylinder(7.0f, height, Cylinder.GENERATE_NORMALS |
                            Cylinder.GENERATE_TEXTURE_COORDS, 10, 10, appearance));
    tableLeg.addChild(tableTop);
    // Move the table leg and top into position
    change.set(new Vector3f(0.0f, height / 2, 0.0f));
    tableLeg.setTransform(change);

    // Create the base
    tableBase.addChild(new Cylinder(50.0f, 5.0f, Cylinder.GENERATE_NORMALS |
                              Cylinder.GENERATE_TEXTURE_COORDS, 20, 20, appearance));
    tableBase.addChild(tableLeg);

    // Move the entire table into position
    change.set(new Vector3f(x, y, z));
    tableBase.setTransform(change);

    return tableBase;
  }

  public TransformGroup createLights()
  {
    // Create a light TG to store all the lights into
    TransformGroup lights = new TransformGroup();

    // Set the lights bounding area to everywhere
    BoundingSphere boundingSphere = new BoundingSphere(
        new Point3d(0.0,0.0,0.0), Double.POSITIVE_INFINITY);

    // Create an ambient light at 40%
    Color3f ambientColour = new Color3f(0.4f, 0.4f, 0.4f);
    AmbientLight ambientLight = new AmbientLight(ambientColour);
    ambientLight.setInfluencingBounds(boundingSphere);
    lights.addChild(ambientLight);

    // Setup a first directional light
    Color3f directionalColour = new Color3f(1.0f, 1.0f, 1.0f);
    Vector3f direction = new Vector3f(0.2f, 0.6f, 0.4f);
    DirectionalLight directionalLight = new DirectionalLight(directionalColour, direction);
    directionalLight.setInfluencingBounds(boundingSphere);
    lights.addChild(directionalLight);

    // Setup a second directional light
    Color3f directionalColour2 = new Color3f(0.6f, 0.6f, 0.6f);
    Vector3f direction2 = new Vector3f(-0.2f, -0.6f, -0.4f);
    DirectionalLight directionalLight2 = new DirectionalLight(directionalColour2,
                                                                       direction2);
    directionalLight2.setInfluencingBounds(boundingSphere);
    lights.addChild(directionalLight2);

    // Return the lights TG
    return lights;
  }

  public TransformGroup createRoom(float x, float y, float z, float width, float height,
                                                                       float depth)
  {
    // Store the collision detection details globally about the room
    this.roomX     = x;
    this.roomZ     = z;
    this.roomWidth = width;
    this.roomDepth = depth;

    // Create 3 indexed arrays for the walls, floor and ceiling
    IndexedQuadArray wallsShape = new IndexedQuadArray(8, IndexedQuadArray.COORDINATES |
               IndexedQuadArray.NORMALS | IndexedQuadArray.TEXTURE_COORDINATE_2, 16);
    IndexedQuadArray floorShape = new IndexedQuadArray(8, IndexedQuadArray.COORDINATES |
               IndexedQuadArray.NORMALS | IndexedQuadArray.TEXTURE_COORDINATE_2, 4);
    IndexedQuadArray roofShape = new IndexedQuadArray(8, IndexedQuadArray.COORDINATES |
               IndexedQuadArray.NORMALS | IndexedQuadArray.TEXTURE_COORDINATE_2, 4);

    // Specify the room coordinates for the 8 points
    Point3f[] shapeCoordinates = {
      new Point3f(x, y, z),
      new Point3f(x + width, y, z),
      new Point3f(x + width, y + height, z),
      new Point3f(x, y + height, z),
      new Point3f(x, y, z + depth),
      new Point3f(x + width, y, z + depth),
      new Point3f(x + width, y + height, z + depth),
      new Point3f(x, y + height, z + depth)
    };

    // Specify the wall surfaces
    int wallsCoordIndices[] = {
      0, 1, 2, 3,  // back wall
      1, 5, 6, 2, // right wall
      5, 4, 7, 6, // front wall
      4, 0, 3, 7  // left wall
    };

    // Specify the ceiling surfaces
    int roofCoordIndices[] = {
      2, 6, 7, 3 // roof
    };

    // Specify the floor surfaces
    int floorCoordIndices[] = {
      0, 4, 5, 1  // floor
    };

    // Specify the wall normals
    Vector3f[] wallsNormals = {
      new Vector3f( 0.0f, 0.0f, -1.0f), // z netagive - back wall
      new Vector3f( 1.0f, 0.0f,  0.0f), // x positive - right wall
      new Vector3f( 0.0f, 0.0f,  1.0f), // z positive - front wall
      new Vector3f(-1.0f, 0.0f,  0.0f)  // x negative - left wall
    };

    // Specify the ceiling normals
    Vector3f[] roofNormals = {
      new Vector3f(0.0f, 1.0f, 0.0f) // y positive - roof
    };

    // Specify the floor normals
    Vector3f[] floorNormals = {
      new Vector3f(0.0f, -1.0f, 0.0f) // y negative - floor
    };

    // Attach the wall normals to each wall point
    int wallsNormalIndices[] =
    {
      0,0,0,0,
      1,1,1,1,
      2,2,2,2,
      3,3,3,3
    };

    // Attach the ceiling normals to each ceiling point
    int roofNormalIndices[] =
    {
      0,0,0,0
    };

    // Attach the floor normals to each floor point
    int floorNormalIndices[] =
    {
      0,0,0,0
    };

    // Specify the wall texture coordiantes
    TexCoord2f wallsTexCoords[] = {
      new TexCoord2f(0.0f, 0.0f),  // bottom left
      new TexCoord2f(7.0f, 0.0f), // bottom right, 7 along the x
      new TexCoord2f(7.0f, 5.0f), // top right, 7 along the x, 5 in the y
      new TexCoord2f(0.0f, 5.0f)  // top left, 5 in the y
    };

    // Specify the texture coordinates for the floor and ceiling
    TexCoord2f vertTexCoords[] = {
      new TexCoord2f(0.0f, 0.0f), // bottom left
      new TexCoord2f(5.0f, 0.0f), // bottom right x5
      new TexCoord2f(5.0f, 5.0f), // top right x5
      new TexCoord2f(0.0f, 5.0f), // top left x5
    };

    // Attach the wall coordinates to the wall points
```

```
int texWallCoordIndices[] =
{
  0, 1, 2, 3, // back wall
  0, 1, 2, 3, // right wall
  0, 1, 2, 3, // front wall
  0, 1, 2, 3  // left wall
};

// Attach the floor and ceiling coordinates to the floor and ceiling points
int texVertCoordIndices[] =
{
  0, 1, 2, 3
};

// Create the walls based on the above points, normals and texture coordinates
wallsShape.setCoordinates(0, shapeCoordinates);
wallsShape.setCoordinateIndices(0, wallsCoordIndices);
wallsShape.setNormals(0, wallsNormals);
wallsShape.setNormalIndices(0, wallsNormalIndices);
wallsShape.setTextureCoordinates(0, 0, wallsTexCoords);
wallsShape.setTextureCoordinateIndices(0, 0, texWallCoordIndices);

// Create the floor based on the above points, normals and texture coordinates
floorShape.setCoordinates(0, shapeCoordinates);
floorShape.setCoordinateIndices(0, floorCoordIndices);
floorShape.setNormals(0, floorNormals);
floorShape.setNormalIndices(0, floorNormalIndices);
floorShape.setTextureCoordinates(0, 0, vertTexCoords);
floorShape.setTextureCoordinateIndices(0, 0, texVertCoordIndices);

// Create the roof based on the above points, normals and texture coordinates
roofShape.setCoordinates(0, shapeCoordinates);
roofShape.setCoordinateIndices(0, roofCoordIndices);
roofShape.setNormals(0, roofNormals);
roofShape.setNormalIndices(0, roofNormalIndices);
roofShape.setTextureCoordinates(0, 0, vertTexCoords);
roofShape.setTextureCoordinateIndices(0, 0, texVertCoordIndices);

// Generate the surface material for the room
Color3f ambientColour = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
Color3f diffuseColour = new Color3f(1.0f, 1.0f, 1.0f); // white
Color3f specularColour = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
Color3f emissiveColour = new Color3f(0.0f, 0.0f, 0.0f); // black

Material material = new Material(ambientColour, emissiveColour,
                       diffuseColour, specularColour, 5.0f);

// Set the walls appearance loading the brick texture
Appearance wallsAppearance = new Appearance();
wallsAppearance.setMaterial(material);
wallsAppearance.setTexture(loadTexture("textures/brick_128.jpg"));

// Set the floor appearance loading the floor texture
Appearance floorAppearance = new Appearance();
floorAppearance.setMaterial(material);
floorAppearance.setTexture(loadTexture("textures/wood_floor_128.jpg"));

// Set the ceiling appearance loading the roof texture
Appearance roofAppearance = new Appearance();
roofAppearance.setMaterial(material);
roofAppearance.setTexture(loadTexture("textures/roof_128.jpg"));

// Set the texture attributes to work with the surface material
TextureAttributes texAttributes = new TextureAttributes();
texAttributes.setTextureMode(TextureAttributes.MODULATE);
wallsAppearance.setTextureAttributes(texAttributes);
floorAppearance.setTextureAttributes(texAttributes);

// Make sure the walls are flat shaded to react to lights
ColoringAttributes colAttrib = new
           ColoringAttributes(0.0f, 0.0f, 1.0f, ColoringAttributes.SHADE_FLAT);
wallsAppearance.setColoringAttributes(colAttrib);
floorAppearance.setColoringAttributes(colAttrib);

// Create the actual shapes and add them to the room TG
TransformGroup tg = new TransformGroup();
tg.addChild(new Shape3D(wallsShape, wallsAppearance));
```

```
    tg.addChild(new Shape3D(floorShape, floorAppearance));
    tg.addChild(new Shape3D(roofShape, roofAppearance));

    return tg;
  }

  private Texture2D loadTexture(String filename)
  {
    TextureLoader loader = null;

    // See whether to get the file from the web or from the local path
    if(appPath.length() > 0)
    {
      // Load the texture from a URL
      URL path = null;

      try
      {
        path = new URL(appPath + filename);
      }
      catch(Exception e) {}

      // Download from the web
      loader = new TextureLoader(path, parentCanvas);
    }
    else
    {
      // Load from the local drive
      loader = new TextureLoader(filename, parentCanvas);
    }

    // Get the image from the TextureLoader
    ImageComponent2D image = loader.getImage();

    // Set the texture's properties
    Texture2D texture = new Texture2D(Texture.BASE_LEVEL, Texture.RGB,
                                          image.getWidth(), image.getHeight());
    texture.setImage(0, image);

    // Return the texture
    return texture;
  }
}
```

CWorldListener Source Code

```
/***************************************
* CWorldListener
* Author: Martin Adams
****************************************
* Defines the worlds callback
* procedures
***************************************/

import javax.vecmath.*;

public interface CWorldListener
{
  public void onEnterTable();
  public void onExitTable();
  public void onCollisionDetect(Vector3f newPosition);
}
```

CMaze Source Code

```
/***************************************
* CMaze
* Author: Martin Adams
****************************************
* Create a 3D maze and test for
* collisions
***************************************/

import com.sun.j3d.utils.image.TextureLoader;

import javax.media.j3d.*;
import javax.vecmath.*;
import java.net.URL;

class CMaze
{
  private static final float SMALL_WALL_DISTANCE = 2.0f; // 2cm

  // Store the application path to load the texutre
  private String   appPath      = "";
  // Store the canvas object to assist loading the texture
  private Canvas3D parentCanvas = null;

  // Store the maze's position and dimensions
  private float posX = 0;
  private float posY = 0;
  private float posZ = 0;
  private float scaleWidth = 0;
  private float scaleHeight = 0;
  private float scaleDepth = 0;

  // Define the maze
  private static final int mazeDef[] = {
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,0,
    0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,1,0,
    0,1,0,1,1,1,1,1,1,0,1,1,1,1,0,1,0,1,1,0,
    0,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,1,0,
    0,1,1,1,0,1,1,1,1,1,1,1,0,0,1,0,0,0,1,0,
    0,1,0,1,0,1,0,1,0,0,0,1,0,0,1,0,1,0,1,0,
    0,1,0,1,1,1,0,1,0,1,0,1,1,0,1,1,1,0,1,0,
    0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,
    0,1,1,0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,0,
    0,1,0,0,1,0,0,0,0,0,0,1,0,0,0,0,1,0,1,0,
    0,1,0,1,1,0,1,0,1,0,1,1,1,1,0,1,1,0,1,0,
    0,1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,1,0,
    0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,
    0,1,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,
    0,1,0,0,1,0,0,0,1,1,1,1,1,0,1,1,1,0,1,0,
    0,1,0,1,1,1,1,0,1,0,0,0,1,0,0,0,1,0,1,0,
    0,1,0,1,0,1,1,0,1,0,1,0,1,1,1,1,1,0,1,0,
    0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,1,0,
    0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  };

  // Set the maze dimensions
  private static final int width = 20;
  private static final int height = 21;

  // Store the distance between each maze block
  private float blockWidth = 1.0f;
  private float blockDepth = 1.0f;

  // Store the maze object
  private IndexedQuadArray maze = null;

  // Store the last valid maze block that the user was in
  private int lastValidX = 0;
  private int lastValidY = 0;

  // Store the coordinate index where the top coordinates start
  private int topStart     = 0;
  private int indexCount   = 0;
  private int normalCount  = 0;
  private int textureCount = 0;

  // Store the normals for each surface for the maze
  private static final Vector3f normals[] = {
    new Vector3f( 0.0f,  0.0f,  1.0f), // North facing
    new Vector3f( 0.0f,  0.0f, -1.0f), // South facing
    new Vector3f( 1.0f,  0.0f,  0.0f), // West facing
    new Vector3f(-1.0f,  0.0f,  0.0f), // East facing
    new Vector3f( 0.0f,  1.0f,  0.0f), // Down facing
    new Vector3f( 0.0f, -1.0f,  0.0f), // Up facing
  };

  // Store the texture coordinates for each surface
  private static final TexCoord2f textureCoords[] = {
    new TexCoord2f(0.0f, 0.0f), // bottom left
    new TexCoord2f(1.0f, 0.0f), // bottom right
    new TexCoord2f(1.0f, 1.0f), // top right
    new TexCoord2f(0.0f, 1.0f)  // top left
  };

  public CMaze(float x, float y, float z, float width, float height, float depth,
                                                  Canvas3D canvas, String appPath)
  {
    // Set the global values to the parameters
    this.posX         = x;
    this.posY         = y;
    this.posZ         = z;
    this.scaleWidth   = width;
    this.scaleHeight  = height;
    this.scaleDepth   = depth;
    this.parentCanvas = canvas;
    this.appPath      = appPath;

    // Calculate the block width (the distance between points)
    this.blockWidth = (this.scaleWidth / this.width);
    this.blockDepth = (this.scaleDepth / this.height);

    // Store the array starting point where coordinate indexes for the higher point start
    this.topStart = (this.height * (this.width + 1)) + this.width + 1;
  }

  public TransformGroup generateMaze()
  {
    TransformGroup tg = new TransformGroup();

    // Create the IndexedQuadArray object ready for defining the points and surfaces
    maze = new IndexedQuadArray((width + 1)*(height + 1)*2, IndexedQuadArray.COORDINATES |
       IndexedQuadArray.NORMALS | IndexedQuadArray.TEXTURE_COORDINATE_2,
       (getSurfaceCount() * 4));

    // Generate all the possible coordinates that could be used in this maze
    generateCoordinates();

    // Set the normals for the maze
    maze.setNormals(0, normals);

    // Set the texture coordinates for the maze
    maze.setTextureCoordinates(0, 0, textureCoords);

    // Generate all the surfaces that build up the maze walls
    generateSurfaces();

    // Create an appearance for this maze
    Appearance mazeAppearance = new Appearance();

    // Set the material properties
    Color3f ambientColour = new Color3f(0.2f, 0.2f, 0.2f); // dark grey
    Color3f diffuseColour = new Color3f(1.0f, 1.0f, 1.0f); // white
    Color3f specularColour = new Color3f(0.0f, 0.0f, 0.0f); // white
    Color3f emissiveColour = new Color3f(0.0f, 0.0f, 0.0f); // black

    Material material = new Material(ambientColour, emissiveColour,
                           diffuseColour, specularColour, 0.0f);

    // Set the maze material and load the grid texture
    mazeAppearance.setMaterial(material);
    mazeAppearance.setTexture(loadTexture("textures/grid_128.jpg"));

    // Set the texture properties to work with the material properties
    TextureAttributes texAttributes = new TextureAttributes();
    texAttributes.setTextureMode(TextureAttributes.MODULATE);
    mazeAppearance.setTextureAttributes(texAttributes);

    // Make sure the surfaces are flat shaded to work with the lights
    ColoringAttributes colAttrib = new
              ColoringAttributes(0.0f, 0.0f, 1.0f, ColoringAttributes.SHADE_FLAT);
    mazeAppearance.setColoringAttributes(colAttrib);

    // Add the newly generated maze the the transform group
    tg.addChild(new Shape3D(maze, mazeAppearance));
    // Move the maze into position
    Transform3D move = new Transform3D();
    move.set(new Vector3f(this.posX, this.posY, this.posZ));
    tg.setTransform(move);

    // Return the maze in the TG
    return tg;
  }

  private void generateSurfaces()
  {
    // This function loops through each maze block and generates the necessary top,
    // right and bottom surfaces

    int x = 0;
    int y = 0;

    // Loop through the maze rows
    for(y = 0; y < height; y++)
    {
      // Loop through the maze columns for this row
      for(x = 0; x < width; x++)
      {
        // Make sure we are not at the right edge
        if (x + 1 < width && y + 1 < height)
        {
          if(mazeDef[(y*width)+x] == 0) // currently no block
          {
            if(mazeDef[(y*width) + (x+1)] == 1) // the one to the right is a block
            {
              drawWestFacingWall(x+1,y);
            }
            if(mazeDef[((y+1)*width) + x] == 1) // the one to the bottom is a block
            {
              drawNorthFacingWall(x,y+1);
            }
          }
          else // there is a block here
          {
            if(mazeDef[(y*width) + (x+1)] == 0) // the one to the right is no block
            {
              drawEastFacingWall(x,y);
            }
            if(mazeDef[(((y+1)*width) + x)] == 0) // there is none to the bottom
            {
              drawSouthFacingWall(x,y);
            }
            // Draw the top of this block
            drawBlockTopSurface(x,y);
          }
        }
      }
    }
  }

  private int getSurfaceCount()
  {
    // This funciton is similar to generateSurfaces() except it only counts
    // how many surfaces will be required to create the completed maze

    int count = 0;
    int x     = 0;
    int y     = 0;

    // Loop through the maze rows
    for(y = 0; y < height; y++)
    {
      // Loop through the maze columns for this row
      for(x = 0; x < width; x++)
      {
        // Make sure we are not at the right edge
        if (x + 1 < width && y + 1 < height)
        {
          if(mazeDef[(y*width)+x] == 0) // currently no block
          {
            if(mazeDef[(y*width) + (x+1)] == 1) // the one to the right is a block
            {
              count++;
            }
            if(mazeDef[((y+1)*width) + x] == 1) // the one to the bottom is a block
            {
              count++;
            }
          }
          else // there is a block here
          {
            if(mazeDef[(y*width) + (x+1)] == 0) // the none to the right is no block
            {
              count++;
            }
            if(mazeDef[(((y+1)*width) + x)] == 0) // there is none to the bottom
            {
              count++;
            }
            // Draw the top of this block
            count++;
          }
        }
      }
    }

    // Return the number of surfaces there were
    return count;
  }

  private void generateCoordinates()
  {
    // This function generates all the possible coordinates that could be used to
    // create the maze.  These coordinates will then be indexed to define the
    // required surfaces.

    int x          = 0;
    int y          = 0;
    int indexPoint = 0;

    // Loop through each edge in the row
    for(y = 0; y <= height; y++)
    {
      // Loop through each edge in the column for this row
      for(x = 0; x <= width; x++)
      {
        // Define the coordinate for this point in the x,y maze
        maze.setCoordinate(indexPoint, new
                     Point3f(x * this.blockWidth, 0.0f, y * this.blockDepth));
        indexPoint++;
      }
    }

    // Repeat the above routine except position the coordinates higher in the Y axis
    for(y = 0; y <= height; y++)
    {
      for(x = 0; x <= width; x++)
      {
        maze.setCoordinate(indexPoint, new
                     Point3f(x * this.blockWidth, this.scaleHeight, y * this.blockDepth));
        indexPoint++;
      }
    }
  }

  private void drawBlockTopSurface(int x, int y)
  {
    // This function creates a horizontal surface
    int p1 = topStart + ((y + 1) * (width + 1)) + x;
    int p2 = p1 + 1;
    int p3 = topStart + (y * (width + 1)) + x + 1;
    int p4 = p3 - 1;

    drawSurface(p1, p2, p3, p4);
    addNormals(5); // Up facing
  }

  private void drawNorthFacingWall(int x, int y)
```

```
{
  // This function creates a surface facing up
  int p1 = (y * (width + 1)) + x + 1;
  int p2 = p1 - 1;
  int p3 = p2 + topStart;
  int p4 = p3 + 1;

  drawSurface(p1, p2, p3, p4);
  addNormals(0); // North facing
}

private void drawSouthFacingWall(int x, int y)
{
  // This function creates a surface facing down
  int p1 = ((y + 1) * (width + 1)) + x;
  int p2 = p1 + 1;
  int p3 = p2 + topStart;
  int p4 = p3 - 1;

      drawSurface(p1, p2, p3, p4);
  addNormals(1); // South facing
}

private void drawEastFacingWall(int x, int y)
{
  // This function creates a surface facing right
  int p1 = ((y + 1) * (width + 1)) + x + 1;
  int p2 = (y * (width + 1)) + x + 1;
  int p3 = p2 + topStart;
  int p4 = p1 + topStart;

  drawSurface(p1, p2, p3, p4);
  addNormals(3); // East facing
}

private void drawWestFacingWall(int x, int y)
{
  // This function creates a surface facing left
  int p1 = (y * (width + 1)) + x;
  int p2 = ((y + 1) * (width + 1)) + x;
  int p3 = p2 + topStart;
  int p4 = p1 + topStart;

  drawSurface(p1, p2, p3, p4);
  addNormals(2); // West facing
}

private void drawSurface(int p1, int p2, int p3, int p4)
{
  // This function joins the 4 points together to create a quad surface
  maze.setCoordinateIndex(indexCount, p1);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p2);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p3);
  indexCount++;
  maze.setCoordinateIndex(indexCount, p4);
  indexCount++;

  // Add the texture coordinate for each point in this surface
  for(int coordIndex = 0; coordIndex < 4; coordIndex++)
  {
    maze.setTextureCoordinateIndex(0, textureCount, coordIndex);
    textureCount++;
  }
}

private void addNormals(int index)
{
  // This function adds the correct normal for each point in this surface
  for(int count = 0; count < 4; count++)
  {
    maze.setNormalIndex(normalCount, index);
    normalCount++;
  }
}
```

```
private Texture2D loadTexture(String filename)
{
  TextureLoader loader = null;

  // See whether to get the file from the web or from the local path
  if(appPath.length() > 0)
  {
    // Load the texture from a URL
    URL path = null;

    try
    {
      path = new URL(appPath + filename);
    }
    catch(Exception e) {}

    // Download from the web
    loader = new TextureLoader(path, parentCanvas);
  }
  else
  {
    // Load from the local drive
    loader = new TextureLoader(filename, parentCanvas);
  }

  // Get the image from the TextureLoader
  ImageComponent2D image = loader.getImage();

  // Set the texture's properties
  Texture2D texture = new Texture2D(Texture.BASE_LEVEL, Texture.RGB,
                                    image.getWidth(), image.getHeight());
  texture.setImage(0, image);

  // Return th etexture
  return texture;
}

public boolean checkCollisionDetection(Vector3f position)
{
  // This function tests the specified vector and sees if it collides with
  // any of the maze walls
  boolean bXCollision = false;
  boolean bZCollision = false;

  // First make sure the user is within the maze grid
  if(position.x > posX && position.x < posX + scaleWidth &&
     position.z > posZ && position.z < posZ + scaleDepth)
  {
    // Check if there is a collision in the X direction
    bXCollision = hitXWall(position.x + SMALL_WALL_DISTANCE, position);
    if(!bXCollision)
      bXCollision = hitXWall(position.x - SMALL_WALL_DISTANCE, position);
    // Check if there is a collision in the Z direction
    bZCollision = hitYWall(position.z + SMALL_WALL_DISTANCE, position);
    if(!bZCollision)
      bZCollision = hitYWall(position.z - SMALL_WALL_DISTANCE, position);
  }

  // Store this position as it may be used to place the viewport into
  // if the user collides with a wall on the next check
  int gridX = (int)(((position.x - posX) / scaleWidth) * width);
  int gridY = (int)(((position.z - posZ) / scaleDepth) * height);

  lastValidX = gridX;
  lastValidY = gridY;

  // Return whether there was a collision
  return bXCollision | bZCollision;
}

private boolean hitXWall(float x, Vector3f position)
{
  // This function tests to see if the position intersects with a block
  // along the X axis.  If it does, it corrects the position to force
  // the user back into a valid block

  boolean bCollision = false;
```

```
    // Convert the coordinate x into a grid value
    int gridX = (int)(((x - posX) / scaleWidth) * width);
    int gridY = (int)(((position.z - posZ) / scaleDepth) * height);

    // Make sure that these coordinates are within the grid
    if(gridX >= width || gridY >= height)
      return false;

    // See if we have collided with a wall
    if(mazeDef[(gridY * width) + gridX] == 1)
    {
      // There has been a collision so the best thing to do is
      // move the viewport to the nearest valid edge
      moveXTo(lastValidX, gridX, position);
      bCollision = true;
    }
    return bCollision;
  }

  private boolean hitYWall(float z, Vector3f position)
  {
    // This function tests to see if the position intersects with a block
    // along the Z axis.  If it does, it corrects the position to force
    // the user back into a valid block

    boolean bCollision = false;
    // Convert the coordinate x into a grid value
    int gridX = (int)(((position.x - posX) / scaleWidth) * width);
    int gridY = (int)(((z - posZ) / scaleDepth) * height);

    // Make sure that these coordinates are within the grid
    if(gridX >= width || gridY >= height)
      return false;

    // See if we have collided with a wall
    if(mazeDef[(gridY * width) + gridX] == 1)
    {
      // There has been a collision so the best thing to do is
      // move the viewport to the nearest valid edge
      moveYTo(lastValidY, gridY, position);
      bCollision = true;
    }
    return bCollision;
  }

  private void moveXTo(int toX, int fromX, Vector3f position)
  {
    // This moves the X position into a valid block
    if(toX > fromX) // LHS of toX
    {
      position.x = (((float)toX / width) * scaleWidth) + posX + SMALL_WALL_DISTANCE;
    }
    else // RHS of toX
    {
      position.x = (((float)(toX + 1) / width) * scaleWidth) + posX - SMALL_WALL_DISTANCE;
    }
  }

  private void moveYTo(int toY, int fromY, Vector3f position)
  {
    // This moves the Y position into a valid block
    if(toY > fromY) // top of toY
    {
      position.z = (((float)toY / height) * scaleDepth) + posZ + SMALL_WALL_DISTANCE;
    }
    else // bottom of to Y
    {
      position.z = (((float)(toY + 1) / height) * scaleDepth) + posZ - SMALL_WALL_DISTANCE;
    }
  }
}
```