Games Programming

Philip Hanna 110CSC207

Tutorial 5: ‘Isometric Map’

Tutorial Overview:

This tutorial will explore how to load and construct an isometric tile map. In terms of getting the most out of this tutorial, you will want to follow the outlined steps and develop your own version – i.e. please don’t just read the material, but rather try it out and play about with the code by introducing some additional changes.

As with any tutorial, I’m particularly interested to receive feedback – when writing this tutorial I will try to include enough detail for you to carry out the steps, however, I depend upon your feedback to better understand the correct level at which to pitch the tutorial. Please send comments to P.Hanna@qub.ac.uk

What I assume you will have done prior to this tutorial:

Before attempting this tutorial you should have installed NetBeans, Java 1.6 and the CSC207 Code Repository. I will also assume that you have already completed the first four tutorials and are happy with creating new classes with NetBeans, etc.

Phase 0: What are we going to do?

In this tutorial we are going to construct an isometric tile based game. In particular, we will look at how we can build a suitable class for storing the tile map. In turn, we will consider how we can construct an appropriate tile object that can store the graphical tile representation. Finally, we will also consider how we can load in the tile map from a disk file and how it will be drawn to the screen

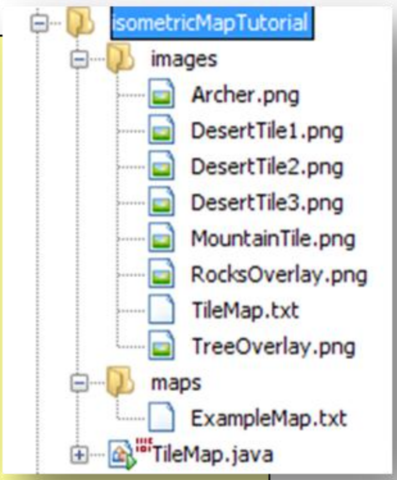
Once we can load, construct and draw the tile map we will also consider how we can have an object travel across the tile map (making use of the notion of passable and impassable tiles). We will construct the game object so that it moves across the map a tile at a time, although, we could equally build it so that it moves freely over the tiles. Finally, we will also consider the notion of constructing a viewport around the object that is moving across the tile map.

Phase 1: Getting ready for developing this project.

This tutorial will take a bit of a running start towards developing the isometric tile map. In particular, we will put in place a number of classes and images that we will make use of later - the tutorial does not reflect the normal development approach (i.e. where we start with a problem and then incrementally develop a solution).

As a means of setting up this project, complete the following steps:

1. Within NetBeans create a new package entitled **isometricMapTutorial** within the TutorialsDemosAndExtras.tutorials package
2. Within isometricMapTutorial create a new class **TileMap** with the code shown below.
3. Within the isometricMapTutorial package create a folder entitled **images**. Once done, copy across the **Archer.png**, **DesertTile1.png**, **DesertTile2.png**, **DesertTile3.png**, **MountainTile.png**, **RocksOverlay.png** and **TreeOverlay.png** images that you will have downloaded along with this project. Also copy across the TileMap.txt asset loader file.
4. Within the isometricMapTutorial package create another folder entitled **maps**. Once done, copy across the **ExampleMap.txt** file that you also will have downloaded along with this project.



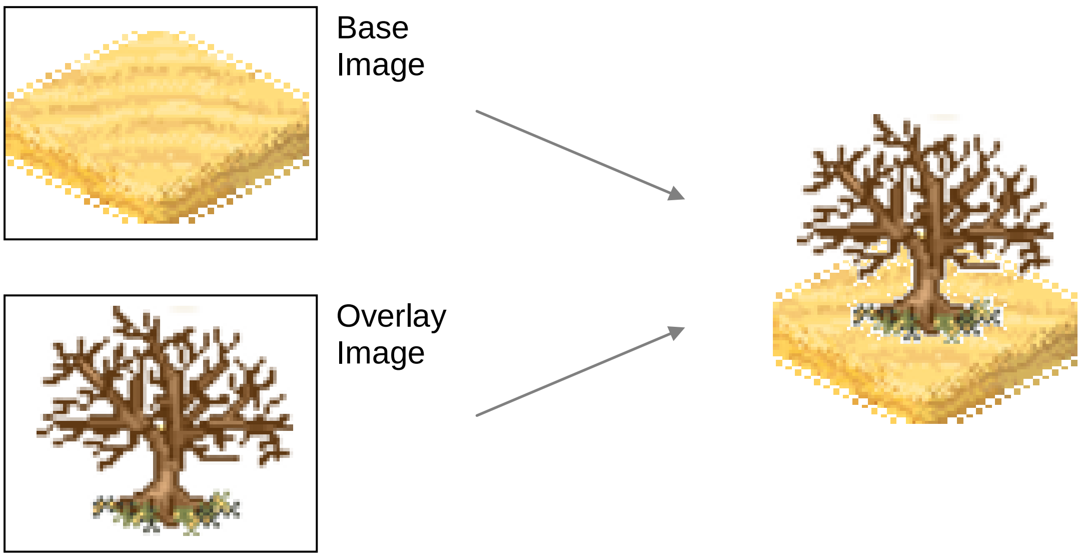
```
package tutorials.isometricMapTutorial;

import game.engine.*;

public class TileMap extends GameEngine {

    private static final int SCREEN_WIDTH = 1024;
    private static final int SCREEN_HEIGHT = 768;

    public TileMap() {
        gameStart( SCREEN_WIDTH, SCREEN_HEIGHT, 32 );
    }

    @Override
    public boolean buildAssetManager() {
        assetManager.loadAssetsFromFile(
                getClass().getResource("images/TileMap.txt"));

        return true;
    }

    @Override
    protected boolean buildInitialGameLayers() {
        TileMapLayer isometricTileMapLayer
                = new TileMapLayer( this, "maps/ExampleMap.txt" );
        addGameLayer( isometricTileMapLayer );

        return true;
    }

    public static void main(String[] args) {
        TileMap instance = new TileMap();
    }
}
```

Phase 2: Thinking about tiles

Unsurprisingly a tile based map consists of tiles. But how should we construct or represent a tile? There is no one correct means of doing this. Within this particular tutorial we will assume that a tile is going to be comprised of a number of different layers (i.e. each tile can have a number of different graphics layered on top of it).

To make this a little bit more formal, this tutorial will assume that each tile is comprised of a single base image (representing the ground) and zero, one or more overlay images, representing things on the ground.

An example is shown below:



The outline for this tutorial has already mentioned that we will construct the map from a loaded map tile file (you've already added this, i.e. ExampleMap.txt, to the project). If you look within this file you will see that the map is described in terms of a textual description, e.g. "MountainTile DesertTile2 DesertTile1". This then raises the question of where and how do we map the textual description onto a suitable graphical object. This could also be done in a number of different ways, in this particular tutorial we have assumed that the Tile class will take, as a parameter, a textual tile description and from this select a suitable graphical realisation.

Ok, let's add in our tile class. To do this you should:

1. Within the isometricTileMap package create a new class **Tile** and update the class to reflect the code shown on the next page.

As can be seen, the class really only consists of two key methods – one to setup the base graphical tile and another to add in an additional overlay graphical realisation. Both of these methods call the addTileGraphic() method which constructs a suitable array of graphical assets and (x, y) offsets for each graphical image. Note that for the base graphic an (x,y) offset of (0,0) is always used. However, for the tile overlays the offset can be different. The offset is selected to ensure that the graphical overlay is situated at an appropriate location within the tile – for example, ensuring that the tree image is positioned so that the bottom of the tree is around the centre of the tile.

```
package tutorials.isometricMapTutorial;

import game.assets.*;

public class Tile {

    public boolean isPassable = true;

    public double tileHeight = 0.0, tileWidth = 0.0;

    public GraphicalAsset[] tileImages = new GraphicalAsset[0];

    private IsometricTileMap tileMap;


    public Tile( String baseTileType, IsometricTileMap tileMap  ) {
        this.tileMap = tileMap;
        setBaseTile( baseTileType );
    }

    public void setBaseTile( String baseTileType ) {

        if( baseTileType.equals( "DesertTile1" ) ) {
            isPassable = true;
            addTileGraphic( "DesertTile1", 0, 0 );
        }
        else if( baseTileType.equals( "DesertTile2" ) ) {
            isPassable = true;
            addTileGraphic( "DesertTile2", 0, 0 );
        }
        else if( baseTileType.equals( "DesertTile3" ) ) {
            isPassable = true;
            addTileGraphic( "DesertTile3", 0, 0 );
        }
        else if( baseTileType.equals( "MountainTile" ) ) {
            isPassable = false;
            addTileGraphic( "MountainTile", 0, 0 );
        }

        tileWidth = tileImages[0].width;
        tileHeight = tileImages[0].height;

    }

    public void addTileOverlay( String tileOverlayType ) {

        if( tileOverlayType.equals( "TreeOverlay" ) ) {
            isPassable = false;
            addTileGraphic( "TreeOverlay", 5, -20 );
        }

        else if( tileOverlayType.equals( "RocksOverlay" ) ) {
            addTileGraphic( "RocksOverlay", 0, 0 );
        }
    }

    private void addTileGraphic( String assetName, int assetXOffset, int assetYOffset )
{
        GraphicalAsset[] newRealisation = new GraphicalAsset[tileImages.length+1];
        System.arraycopy( tileImages, 0, newRealisation, 0, tileImages.length );

        newRealisation[tileImages.length]
                = tileMap.gameLayer.assetManager.retrieveGraphicalAsset(assetName);
        newRealisation[tileImages.length].offsetX = assetXOffset;
        newRealisation[tileImages.length].offsetY = assetYOffset;

        tileImages = newRealisation;
    }
}
```

Within the Tile class you will also notice that we define an instance variable:

```
    protected boolean isPassable;
```

This variable will be used to determine if the tile is passable, i.e. other objects can pass over the tile. For example, we setup the mountain tile to be impassable. As another example, should we add a tree graphical overlay to a tile, then we also set it to be impassable.

In terms of your own use of this class, you can added whatever instance variables make sense for your own type of game.

Phase 3: Building up the map

Note: In this phase we'll look at a number of pieces of code before I'll ask you to start putting them together into a Java class.

The next class we should create will be one that can load, hold and draw the isometric tile map, additionally it will also check to see if a specified object is on top of any impassable tiles (i.e. it will do all the core functions needed of the map),

First question: how should we represent the map? For this particular tutorial we define the following parameters:

```
    protected int mapWidth;
    protected int mapHeight;

    protected int tileWidth;
    protected int tileHeight;

    protected Tile tiles[][];
```

Each map has a width and height in terms of tiles. In turn, each tile has a default width and height in terms of pixels. Finally, we define a 2D array of Tile objects that will comprise our isometric tile map.

Ok, so how should we load in the map from a disk file? Having defined the parameters above we know that we should load in the map width and height and also the default tile width and height. Given this, we can then go on to load in each individual tile description.

Consider the code shown on the next page:

We pass in a String containing the file name which is duly opened with a call to:

```
        BufferedReader bufferedReader
           = new BufferedReader( new InputStreamReader(
                this.getClass().getResource( mapName ).openStream() ) );
```

Once opened, we then load in the next four lines (which we assume contain a single number relating to the tile width, tile height, map width and map height).

```
public void constructMap( String mapName ) {
        try {
            BufferedReader bufferedReader
                    = new BufferedReader( new InputStreamReader(
                    this.getClass().getResource( mapName ).openStream() ) );

            tileWidth = Integer.parseInt( bufferedReader.readLine() );
            tileHeight = Integer.parseInt( bufferedReader.readLine() );

            mapWidth = Integer.parseInt( bufferedReader.readLine() );
            mapHeight = Integer.parseInt( bufferedReader.readLine() );

            tiles = new Tile[mapWidth][mapHeight];

            for( int rowIdx = 0; rowIdx < mapHeight; rowIdx++ ) {
                String rowData = bufferedReader.readLine();
                StringTokenizer tokenizer = new StringTokenizer( rowData );

                for( int colIdx = 0; colIdx < mapWidth; colIdx++ )
                    tiles[colIdx][rowIdx]
                        = new Tile( tokenizer.nextToken(), this );
            }

            while( bufferedReader.ready() ) {
                StringTokenizer tokenizer
                        = new StringTokenizer( bufferedReader.readLine() );

                int colIdx = Integer.parseInt( tokenizer.nextToken() );
                int rowIdx = Integer.parseInt( tokenizer.nextToken() );

                String tileOverlayType = tokenizer.nextToken();

                tiles[colIdx][rowIdx].addTileOverlay( tileOverlayType );
            }
        } catch( IOException e ) {
            System.out.println( "Error: Can't read from: " + mapName );
        }

        setGeometry( new Box( 0, 0,
                mapWidth * tileWidth + tileWidth/2,
                mapHeight * tileHeight/2 + tileHeight/2 ) );
    }
```

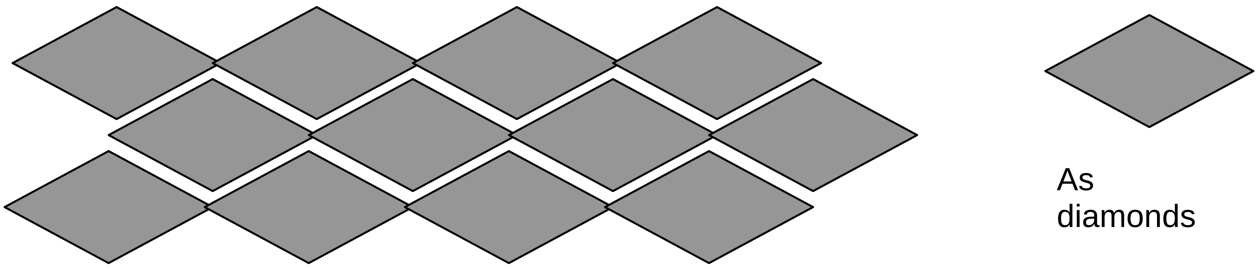
We then have two for loops to load in the base image tile descriptions. In order to make the parsing easier, we make use of Java's StringTokenizer class (try a Google on this if you've not used it before – it provides a very easy means of parsing strings). Once we load in the base tile graphic for each tile, we finally (have a look at the ExampleMap.txt file) load in a number of lines which provide tile x and y offsets and whatever forms of graphical overlay we wish to add.

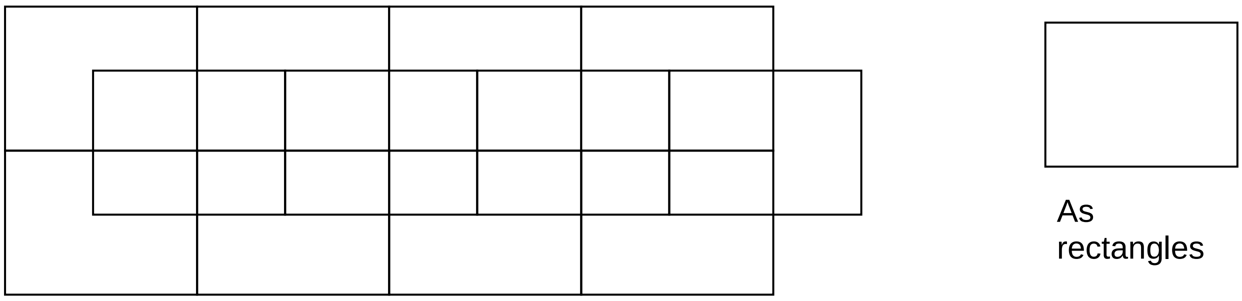
Finally, if the map is to be a game object we can give it a geometry based on the width and height of everything we have loaded.

Note: This is a somewhat arbitrary load file format – you can change it to whatever makes best sense for your own development.

Ok, so we've defined a suitable structure, how should we draw the map out to the screen? As this is an isometric tile map it's not as simple as the case of a nice 2-D array of rectangular shaped tiles (i.e. each isometric tile is diamond shaped).

The particular pattern we need to draw is shown below:





In other words when drawing out each tile, every odd row (i.e. 1, 3, 5, etc.) should be moved ½ tile width to the right.

Additionally, the spacing of the tiles is such that each row of tiles is separated by ½ tile height (i.e. they overlap). We can do this as follow:

```
public void draw( Graphics2D graphics2D, int drawX, int drawY ) {

   for( int rowIdx = 0; rowIdx < mapHeight; rowIdx++ )
      for( int colIdx = 0; colIdx < mapWidth; colIdx++ ) {

         int tileDrawX = drawX + colIdx * tileWidth - (int)width/2;
         if( rowIdx % 2 == 1 ) tileDrawX += tileWidth/2;

         int tileDrawY = drawY + rowIdx * tileHeight/2 - (int)height/2;

         int tileDrawYDrop = 0;
         if( tiles[colIdx][rowIdx].tileHeight > tileHeight )
            tileDrawYDrop
                    = (int)((tiles[colIdx][rowIdx].tileHeight - tileHeight));

         GraphicalAsset[] tileImages = tiles[colIdx][rowIdx].tileImages;
         for( int imageIdx = 0; imageIdx < tileImages.length; imageIdx++ ) {
            if( (tileDrawX + tileImages[imageIdx].width/2) >=
                && (tileDrawX - tileImages[imageIdx].width < gameEngine.screenWidth)
               && (tileDrawY + tileImages[imageIdx].height/2 >= 0)
               && (tileDrawY - tileImages[imageIdx].height/2 < gameEngine.screenWidth) )
               {
                    tileImages[imageIdx].draw(graphics2D,
                       tileDrawX + (int)tileImages[imageIdx].offsetX,
                       tileDrawY + (int)tileImages[imageIdx].offsetY - tileDrawYDrop );
               }
         }
      }
   }
}
```

Now… the code shown above contains some additional complications. Sometimes we might want to draw a tile which is larger than the default defined tile height (maybe it contains high mountains or a high tree). In this case, we check to see if the height is greater and if this is the case, ensure the draw will place the bottom of the tile at the same position as other tiles in the row. We also do a quick check to see if the tile is within the bounds of the screen (there is little point drawing tiles that fall outside of the screen within large maps!)

Phase 4: Adding units

So we can draw the tiles out - what about adding some units to our map? There are a number of different ways this could be done – depending upon the type of game that is to be developed. For example, we could restrict units to individual tiles, alternatively we could allow units free roam of the map (i.e. they could overlap two, or more, tiles). We could also have our units as something separate to the map, i.e. the map is drawn first and then units are drawn on top of it. Alternatively, we could have the units as part of the map. We will opt for the later approach as it will entail we can correctly draw units that fall behind other tiles (e.g. a tall tree).

Hence, within our tile map class we will define a suitable container for our units, namely:

```
public HashMap<Integer, Unit> units = new HashMap<Integer, Unit>();
```

Why a HashMap? And why does it take an Integer key? I'm getting ahead of myself here. Whenever we add units to the map we must ask what type of processing we are likely to want to perform on each unit. We will certainly want to update the units, and as part of the update we are likely to want to know the location of the unit on the map. It is also likely that we will want to know if a tile contains any units (e.g. for update checks or when rendering the map to the screen). In other words, within the game it is important that we have ready access to units and a fast means of identifying units on a particular tile.

In order to provide this fast access we will use a hash map where the key to the hash map is effectively a map coordinate, e.g. we can very quickly determine if any units are on a particular location. How do we build up a map coordinate key? Have a look at the following methods:

```
public int convertPositionToHash( int mapX, int mapY ) {
    return mapX * 65536 + mapY;
}

public int getMapXFromHash( int hash ) {
    return hash / 65536;
}

public int getMapYFromHash( int hash ) {
    return hash % 65536;
}
```

By using these methods we can take an (x,y) tile location and turn it into an combined integer (and vice versa). **Aside:** This assumes at most one unit will occupy a tile. If multiple units can be stacked on a tile then an extended approach would need to be taken. It also assumes that the map won't be larger than 64Kx64K tiles!

Ok, we’re just about at the point where we can bring the entire map class together. Create a new class **IsometricTileMap** and add the following code:

```
package tutorials.isometricMapTutorial;
import game.assets.*; import game.engine.*; import game.geometry.*;
import java.io.*; import java.util.*; import java.awt.*;

public class IsometricTileMap extends GameObject {

    public int mapWidth, mapHeight;
    public int tileWidth, tileHeight;

    public Tile tiles[][];
    public HashMap<Integer, Unit> units = new HashMap<Integer, Unit>();
    public Unit focalUnit = null;
    public GameLayer gameLayer;

    public IsometricTileMap( GameLayer gameLayer, String mapFile ) {
        super( gameLayer );
        this.gameLayer = gameLayer;
        constructMap( mapFile );
        addUnits();
    }

    public int convertPositionToHash( int mapX, int mapY ) {
        return mapX * 65536 + mapY;
    }

    public int getMapXFromHash( int hash ) {
        return hash / 65536;
    }

    public int getMapYFromHash( int hash ) {
        return hash % 65536;
    }

    public void constructMap( String mapName ) {
        try {
            BufferedReader bufferedReader
                    = new BufferedReader( new InputStreamReader(
                    this.getClass().getResource( mapName ).openStream() ) );

            tileWidth = Integer.parseInt( bufferedReader.readLine() );
            tileHeight = Integer.parseInt( bufferedReader.readLine() );

            mapWidth = Integer.parseInt( bufferedReader.readLine() );
            mapHeight = Integer.parseInt( bufferedReader.readLine() );

            tiles = new Tile[mapWidth][mapHeight];

            for( int rowIdx = 0; rowIdx < mapHeight; rowIdx++ ) {
                String rowData = bufferedReader.readLine();
                StringTokenizer tokenizer = new StringTokenizer( rowData );

                for( int colIdx = 0; colIdx < mapWidth; colIdx++ )
                    tiles[colIdx][rowIdx] = new Tile( tokenizer.nextToken(), this );
            }

            while( bufferedReader.ready() ) {
                StringTokenizer tokenizer
                        = new StringTokenizer( bufferedReader.readLine() );

                int colIdx = Integer.parseInt( tokenizer.nextToken() );
                int rowIdx = Integer.parseInt( tokenizer.nextToken() );

                String tileOverlayType = tokenizer.nextToken();

                tiles[colIdx][rowIdx].addTileOverlay( tileOverlayType );
            }
        } catch( IOException e ) {
            System.out.println( "Error: Can't read from: " + mapName );
        }

        setGeometry( new Box( 0, 0,
                mapWidth * tileWidth + tileWidth/2,
                mapHeight * tileHeight/2 + tileHeight/2 ) );
    }
    private void addUnits() {
        Unit archer = new Unit( "Archer", 5, 5, this );

        int positionHash = convertPositionToHash( 5, 5 );
        units.put(positionHash, archer);

        focalUnit = archer;
    }

    @Override
    public void update() {
        for( Integer positionHash : units.keySet() )
            units.get(positionHash).update();

        if( focalUnit != null ) {
            gameLayer.centerViewportOnPosition(
                    (this.x - this.width/2) + focalUnit.mapX * tileWidth,
                    (this.y - this.height/2) + focalUnit.mapY * tileHeight/2,
                    gameLayer.gameEngine.screenWidth/4, gameLayer.gameEngine.screenHeight/4  );
        }
    }

    public void moveUnit( Unit unit, int oldMapX, int oldMapY, int newMapX, int newMapY ) {
        units.remove(convertPositionToHash(oldMapX,oldMapY));
        units.put(convertPositionToHash(newMapX,newMapY), unit);
    }

    @Override
    public void draw( Graphics2D graphics2D, int drawX, int drawY ) {
        for( int rowIdx = 0; rowIdx < mapHeight; rowIdx++ )
            for( int colIdx = 0; colIdx < mapWidth; colIdx++ ) {

            int tileDrawX = drawX + colIdx * tileWidth - (int)width/2;
            if( rowIdx % 2 == 1 )
                tileDrawX += tileWidth/2;

            int tileDrawY = drawY + rowIdx * tileHeight/2 - (int)height/2;

            int tileDrawYDrop = 0;
            if( tiles[colIdx][rowIdx].tileHeight > tileHeight )
                tileDrawYDrop = (int)((tiles[colIdx][rowIdx].tileHeight - tileHeight));

            GraphicalAsset[] tileImages = tiles[colIdx][rowIdx].tileImages;
            for( int imageIdx = 0; imageIdx < tileImages.length; imageIdx++ ) {
                if( (tileDrawX + tileImages[imageIdx].width/2) >= 0
                        && (tileDrawX - tileImages[imageIdx].width < gameEngine.screenWidth)
                        && (tileDrawY + tileImages[imageIdx].height/2 >= 0)
                        && (tileDrawY - tileImages[imageIdx].height/2 < gameEngine.screenWidth) )
                    tileImages[imageIdx].draw(graphics2D,
                            tileDrawX + (int)tileImages[imageIdx].offsetX,
                            tileDrawY + (int)tileImages[imageIdx].offsetY - tileDrawYDrop );
            }

            int positionHash = convertPositionToHash(colIdx, rowIdx);
            if( units.containsKey( positionHash ) ) {
                Unit unit = units.get(positionHash);
                if( (tileDrawX + unit.image.width/2) >= 0
                        && (tileDrawX - unit.image.width < gameEngine.screenWidth)
                        && (tileDrawY + unit.image.height/2 >= 0)
                        && (tileDrawY - unit.image.height/2 < gameEngine.screenWidth) )
                    unit.image.draw(graphics2D,
                        tileDrawX+tileWidth/2-(int)unit.width/2, tileDrawY-tileHeight/3 );
            }
        }
    }
}
```

Note that within this class we have extended the previous introduced draw method to draw units. As before, if we detect that a unit is on a particular tile we also check to see if the unit will actually be drawn on-screen (as opposed to off-screen). The class also introduces the notion of a focal unit, e.g. whenever we update the map we update all objects and also centre the game layer on the location of the focal game unit, e.g. as the focal unit moves around the screen we will keep the layer viewport focussed on it. **Aside:** We also explicitly store a GameLayer reference as we will use it later the unit class to get assets.

There is an awful lot of code to this class – you might also wish to explore the IsometricTileMap.java file available as part of this download as it provides additional comments on the code.

Phase 5: Creating a unit

Add the code shown below:

```
package tutorials.isometricMapTutorial;

import game.assets.*;
import game.engine.*;
import java.awt.event.*;

public class Unit {

    public String type;
    public int mapX, mapY;
    public double width = 0.0, height = 0.0;

    public GraphicalAsset image;
    private IsometricTileMap tileMap;

    public Unit( String type, int mapX, int mapY, IsometricTileMap tileMap ) {
        this.tileMap = tileMap;

        this.mapX = mapX;
        this.mapY = mapY;

        setType( type );
    }

    public void setType( String type ) {
        if( type.equals( "Archer" ) ) {
            image = tileMap.gameLayer.assetManager.retrieveGraphicalAsset( "Archer");
        }

        width = image.width;
        height = image.height;
    }

    public void update() {
        GameInputEventManager inputEvent = tileMap.gameLayer.inputEvent;

        int newMapX = mapX, newMapY = mapY;

        if( inputEvent.keyTyped(KeyEvent.VK_UP) ) {
            if( tileMap.tiles[mapX+(mapY%2==1?1:0)][mapY-1].isPassable ) {
                newMapX += (mapY%2==1 ? 1 : 0);
                newMapY--;
            }
        } else if( inputEvent.keyTyped(KeyEvent.VK_DOWN)) {
            if( tileMap.tiles[mapX-(mapY%2==0?1:0)][mapY+1].isPassable ) {
                newMapX -=(mapY%2==0?1:0);
                newMapY++;
            }
        } if( inputEvent.keyTyped(KeyEvent.VK_LEFT) ) {
            if( tileMap.tiles[mapX-(mapY%2==0?1:0)][mapY-1].isPassable ) {
                newMapX -= (mapY%2==0?1:0);
                newMapY--;
            }
        } else if( inputEvent.keyTyped(KeyEvent.VK_RIGHT)) {
            if( tileMap.tiles[mapX+(mapY%2==1?1:0)][mapY+1].isPassable ) {
                newMapX += (mapY%2==1?1:0);
                newMapY++;
            }
        }

        if( newMapX != mapX || newMapY != mapY ) {
            tileMap.moveUnit(this, mapX, mapY, newMapX, newMapY);
            mapX = newMapX;
            mapY = newMapY;
        }
    }
}
```

Each unit has a defined type and graphical realisation (we could have used an animated asset if we had wanted to).

The update method within the class shows how you can map up, down, left and right key presses onto map movement. The code is a little bit complex as it needs to take into account the row offset between successive map rows. **Aside:** The code also assumes that the map will be surrounded by impassable tiles, e.g. it does not include an edge error checks.

Phase 6: Putting it all together

The last part of the puzzle that we need is a suitable game layer that will tie the various created classes together. To do this you should:

1. Within the isometricMapTutorial package create a new class **TileMapLayer** and update the class to reflect the code shown below.

This game layer is somewhat similar to those explored within earlier tutorials. We create suitable objects and add them to the game layer. Finally, build and run the game.

```
package tutorials.isometricMapTutorial;

import game.engine.*;
import java.awt.*;

public class TileMapLayer extends GameLayer {

    public TileMapLayer( GameEngine gameEngine, String mapFile ) {
        super( "TileMapLayer", gameEngine );
        createMap( mapFile );
    }

    private void createMap( String mapFile ) {
        IsometricTileMap isometricTileMap = new IsometricTileMap( this, mapFile );
        isometricTileMap.setName( "TileMap" );

        double mapWidth = isometricTileMap.width;
        double mapHeight = isometricTileMap.height;

        if( width <  mapWidth ) width = mapWidth;
        if( height  <  mapHeight ) height = mapHeight;

        isometricTileMap.x = mapWidth/2;
        isometricTileMap.y = mapHeight/2;

        addGameObject( isometricTileMap );
    }

    @Override
    public void update() {
        IsometricTileMap isometricTileMap = (IsometricTileMap)getGameObject("TileMap");
        isometricTileMap.update();
    }

    @Override
    public void draw( Graphics2D graphics2D ) {
        Color originalColour = graphics2D.getColor();
        graphics2D.setColor( Color.black );
        graphics2D.fillRect( 0, 0, gameEngine.screenWidth, gameEngine.screenHeight );
        graphics2D.setColor( originalColour );

        super.draw( graphics2D );
    }
}
```