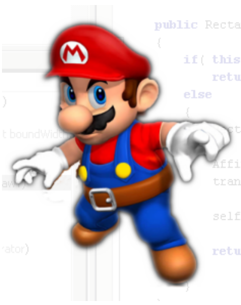


Games Programming

Philip Hanna 110CSC207

# Tutorial 6: ‘Top Down Scroller’

## Tutorial Overview:

This tutorial will explore how to load and construct a top-down scrolling game. 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 five tutorials.

## Phase 0: What are we going to do?

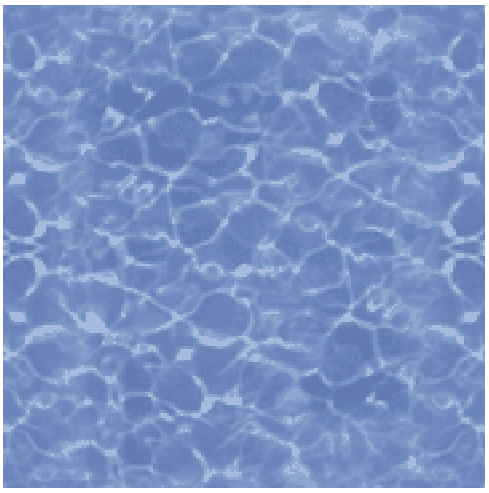
In this tutorial we are going to construct a top-down perspective scrolling game. The game will consist of a number of different layers that scroll at different speeds, thereby giving the impression that of depth and movement.

This tutorial will also differ from other tutorial in that it will simply present the high-level concepts and then provide a full coded solution. The coded solution will be heavily commented, however, and should be read as a means of understanding of the game operates. The coded solution is exactly the same as that found within the TutorialsDemosAndExtras.tutorials.topDownScroller package.

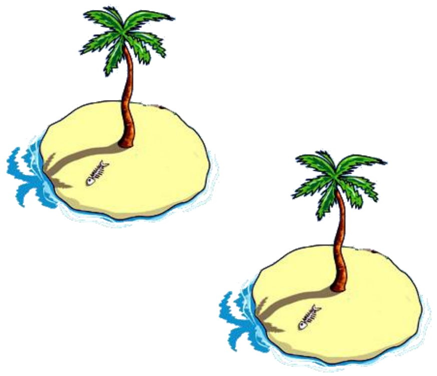
Does this mean I’ve taken the easy way out of this tutorial and avoided the need to write large amounts of descriptive material? In part yes (for which I’m grateful!), however, more importantly, this tutorial represents a jumping off point into the rest of the code repository, e.g. in mastering this tutorial will be further build up your skills in understanding ‘foreign’ code that will enable you to easily make use of other classes contained within the code repository.

As an aside, it might be worthwhile thinking back to when you tackled the first tutorial and reviewing how much you’ve learnt – hopefully by now the early tutorials appear nice and simple (which they may not have done first time around!)

## Phase 1: The high level game…

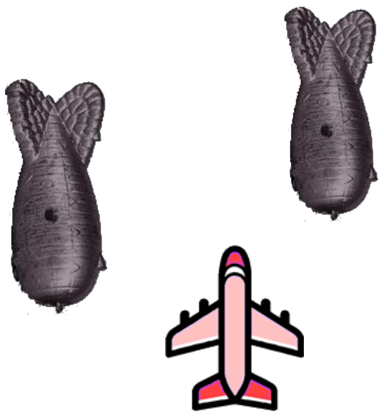


The bottom layer will consist of a single game object that contains a scrolling image asset of water.



The next layer will contain a number of island graphics that will scroll slowly (i.e. they are far away from the player's plane and have low visual movement).

The next layer up will contain a number of cloud objects that scroll faster than the islands but slower than the player's plane)



The topmost layer will contain all the 'proper', i.e. interacting game objects. The player will be in control of a plane that can move about the screen (within a defined movement range). The plane can fire projectiles and 'bounce' off the barrage balloons which will scroll down the screen.

## Phase 1: Extending the game engine…

```
package tutorials.topDownScroller;

import game.engine.GameEngine;

import java.awt.*;
import java.awt.event.KeyEvent;

public class TopDownScroller extends GameEngine {

    /***
     * Define the default screen width and height the game will run at
     */
    private static final int SCREEN_WIDTH = 1024;
    private static final int SCREEN_HEIGHT = 768;

    private boolean displayHelpOverlay = false;

    /**
     * Create and start new instance of space box
     */
    public TopDownScroller() {
        this.gameStart( SCREEN_WIDTH, SCREEN_HEIGHT, 32 );
    }

    /**
     * Attempt to load all the graphical and sound assets required by the game.
     * Loaded assets are added to the game engine's asset manager.
     *
     * @return boolean true if all assets could be loaded, otherwise false
     */
    public boolean buildAssetManager() {
        assetManager.loadAssetsFromFile(
            this.getClass().getResource("images/TopDownScrollerAssets.txt"));
        return true;
    }

    /**
     * Build initial game layers
     */
    @Override
    protected boolean buildInitialGameLayers() {
        TopDownScollerLayer spaceLayer = new TopDownScollerLayer( this );
        addGameLayer( spaceLayer );

        return true;
    }

    /**
     * Define a global input event manager to handle game wide input events
     */
    @Override
    protected void considerInput() {
        // Invoke the default game engine consider input handler
        super.considerInput();

        // Toggle the display of the help overlay
        if( this.inputEvent.keyTyped(KeyEvent.VK_H) )
            this.displayHelpOverlay = !this.displayHelpOverlay;
    }
    /**
     * Extend the default game render to include a game statistics report
     * and/or help overlay if requested
     *
     * @param  graphics2D Graphics context on which to render the game
     */
    @Override
    protected void gameRender( Graphics2D graphics2D ) {
        Color originalColour = graphics2D.getColor();
        graphics2D.setColor( Color.black );
        graphics2D.fillRect( 0, 0, screenWidth, screenHeight );
        graphics2D.setColor( originalColour );

        // Call the game engine gameRender to render all visible game layers
        super.gameRender( graphics2D );

        graphics2D.setFont( new Font( "MONOSPACED", Font.BOLD, 12 ) );
        graphics2D.setColor( Color.YELLOW );

        // Display game exit information
        graphics2D.drawString( "ESC - Quit : H - help", 10, 10 );

        if( this.displayHelpOverlay == true ) {
            graphics2D.drawString( "Right Arrow - Move right", 10, 30 );
            graphics2D.drawString( "Left Arrow - Move left", 10, 45 );
            graphics2D.drawString( "Up Arrow - Move forward", 10, 60 );
            graphics2D.drawString( "Down Arrow - Move backwards", 10, 75 );
            graphics2D.drawString( "Space - Fire", 10, 90 );
        }
    }


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

## Phase 2: Adding the game layer(s)…

```
package tutorials.topDownScroller;

import game.assets.ImageAssetTile;

import game.physics.*;
import game.engine.*;
import game.geometry.*;

/**
 * Core game layer within the top-down scroller example (i.e. the layer
 * which contains the scroller itself). This layer contains the core game
 * update loop and game logic.
*/

public class TopDownScollerLayer extends CollisionSpace {

    ///////////////////////////////////////////////////////////////////////////
    // Class data: Class variables                                           //
    ///////////////////////////////////////////////////////////////////////////

    /**
     * Objects at different heights appear to move at different speeds
     * in relation to one another, i.e. cloud scroll past more slowly than
     * the plane and the water/islands scroll past most slowly than the
     * clouds (this gives a better feeling of depth).
     * <P>
     * The following values can be used to determine the strength of parallax
     * effect, i.e. a value of 0.5 means that for every two pixels that a game
     * object moves (parallax value of 1.0) the object with a value of 0.5 will
     * only move 1 pixels.
     */
    private static final double PARALLAX_VALUE_ISLANDS = 0.4;
    private static final double PARALLAX_VALUE_CLOUDS = 0.8;
    private static final double PARALLAX_VALUE_GAMEOBJECTS = 1.0;

    /**
     * Height of the game layer (the width is assumed to be equal to
     * the screen width) and the speed (in pixels/update) that the
     * screen scrolls.
     */
    private static final double GAME_HEIGHT = 20000;
    private static final double GAME_SCROLLSPEED = 4.0;

    /**
     * Current position within the game (i.e. how far have we scrolled
     * through the layer.
     */
    private double gamePosition = 0;

    /**
     * Region (widthxheight) relative and offset from the centre of the
     * screen within which the player plane will be forced to remain
     * (using this you can control the region of the screen that the
     * player is permitted to fly over).
     */
    private double flyRegionOffsetX = 0;
    private double flyRegionOffsetY = 0;
    private double flyRegionWidth = gameEngine.screenWidth - 400;
    private double flyRegionHeight = gameEngine.screenHeight - 300;
```

```
////////////////////////////////////////////////////////////////////////////
// Constructors                                                           //
////////////////////////////////////////////////////////////////////////////

public TopDownScollerLayer( GameEngine gameEngine ) {
    super( "TopDownScollerLayer", gameEngine );

    // Set the layer width to the width of the screen and the layer
    // height to that specified in the class variable
    width = gameEngine.screenWidth;
    height = GAME_HEIGHT;

    // Set the current layer viewport to the 'bottom' starting point
    // for the scroller
    viewPortLayerX = width/2;
    viewPortLayerY = height - gameEngine.screenHeight/2;

    // Add in game object sets for each of the different types of
    // game object (this makes it easier to update the objects
    // during the update phase).
    addGameObjectCollection( "Projectiles" );
    addGameObjectCollection( "Obstacles" );

    // Create the various background, etc. objects and the player plane
    createBackground();
    createObstacles();
    createPlayerPlane();
}

////////////////////////////////////////////////////////////////////////////
// Methods: Layer initialisation                                          //
////////////////////////////////////////////////////////////////////////////

/**
 * Create the background objects, i.e. water, islands, clouds, obstacles.
 * <P>
 * Note: Islands, clouds and obstacles are created at random throughout the
 * scrollable region. A more structure approach would load the placement
 * positions from a data file to provide each level with a defined layout
 */
private void createBackground() {
    int NUM_ISLANDS = 20;
    int NUM_CLOUDS = 150;

    // Create the background water layer and object object (with correct
    // layer draw order, etc. that will hold a game object displaying a
    // scrolling background. This layer will be sized to be equal to the
    // screen dimensions as the game won't move, instead the graphical
    // image displayed by the water object will scoll.
    GameLayer waterLayer = new GameLayer( "WaterLayer", this.gameEngine );
    waterLayer.width = gameEngine.screenWidth;
    waterLayer.height = gameEngine.screenHeight;

    GameObject backgroundWater = new GameObject( waterLayer );
    backgroundWater.setName( "BackgroundWater" );
    backgroundWater.setRealisation( "BackgroundWater" );
    backgroundWater.setGeometry(
        new Box( 0, 0, gameEngine.screenWidth, gameEngine.screenHeight));
    backgroundWater.setPosition( waterLayer.width/2, waterLayer.height/2 );
    waterLayer.addGameObject( backgroundWater );

    gameEngine.addGameLayer(waterLayer);
    // Create a layer and set of island objects that will be drawn just
    // after the water layer is drawn. The layer is sized to reflect the
    // size of the top most layer scalled by the island parallax value
    GameLayer islandLayer
        = new GameLayer( "IslandLayer", this.gameEngine );

    islandLayer.width = gameEngine.screenWidth;
    islandLayer.height = GAME_HEIGHT * PARALLAX_VALUE_ISLANDS;
    islandLayer.viewPortLayerY
        = islandLayer.height - gameEngine.screenHeight/2;

    // Randomly create island objects.
    for( int idx = 0; idx < NUM_ISLANDS; idx++ ) {
        GameObject island = new GameObject( islandLayer );
        island.setRealisationAndGeometry( "Island" );

        island.setPosition(
              island.width/2 + gameEngine.randomiser.nextInt(
                    (int)(islandLayer.width - island.width/2)),
              island.height/2 + gameEngine.randomiser.nextInt(
                    (int)(islandLayer.height - island.height/2)));
        islandLayer.addGameObject( island );
    }

    gameEngine.addGameLayer(islandLayer);
    islandLayer.setDrawOrder(waterLayer.getDrawOrder()+1);

    // Create a layer and set of cloud objects that will be drawn just
    // after the island layer. As with the island layer, the cloud layer
    // is sized to reflect the size of the topmost layer scalled by the
    // cloud parallax value
    GameLayer cloudLayer = new GameLayer( "CloudLayer", this.gameEngine );
    cloudLayer.width = gameEngine.screenWidth;
    cloudLayer.height = GAME_HEIGHT * PARALLAX_VALUE_CLOUDS;
    cloudLayer.viewPortLayerY
        = cloudLayer.height - gameEngine.screenHeight/2;

    // Randomly create cloud objects
    for( int idx = 0; idx < NUM_CLOUDS; idx++ ) {
        GameObject cloud = new GameObject( cloudLayer );
        cloud.setRealisationAndGeometry(
              "Cloud" + (gameEngine.randomiser.nextInt(2)+1) );

        cloud.setPosition(
                cloud.width/2 + gameEngine.randomiser.nextInt(
                    (int)(cloudLayer.width - cloud.width/2)),
                cloud.height/2 + gameEngine.randomiser.nextInt(
                    (int)(cloudLayer.height - cloud.height/2)));
        cloudLayer.addGameObject( cloud );
    }

    gameEngine.addGameLayer(cloudLayer);
    cloudLayer.setDrawOrder(islandLayer.getDrawOrder()+1);

    // Set the draw order of this layer (holding the player's plane to
    // be the topmost layer
    this.setDrawOrder(cloudLayer.getDrawOrder()+1);
}
```

```
/**
 * Create a set of barrage balloons within the layer to provide obstacles
 * for the player to avoid
 */
private void createObstacles() {
    int NUM_OBSTACLES = 40;

    for( int idx = 0; idx < NUM_OBSTACLES; idx++ ) {
        Body obstacle = new Body( this );
        obstacle.setRealisationAndGeometry( "BarrageBalloon" );
        obstacle.restitution = 0.25;

        obstacle.setPosition(
                obstacle.width/2 + gameEngine.randomiser.nextInt(
                    (int)(this.width - obstacle.width/2)),
                obstacle.height/2 + gameEngine.randomiser.nextInt(
                    (int)(this.height - obstacle.height/2)));
        this.addGameObject( obstacle );
    }
}

/**
 * Create the player plane
 */
protected void createPlayerPlane() {
    Plane playerPlane = new Plane( this );
    playerPlane.setName("PlayerPlane");
    playerPlane.setPosition(gameEngine.screenWidth/2,
            GAME_HEIGHT -gameEngine.screenHeight/2 );
    addGameObject( playerPlane );
}


////////////////////////////////////////////////////////////////////////////
// Methods: Game update                                                   //
////////////////////////////////////////////////////////////////////////////

/**
 * Central update loop called by the game engine
 */
public void update() {

    updateViewPort();
    updatePlayerPlane();
    updateProjectiles();

    // It is important that the super.update method is called - for two
    // reasons: The CollisionSpace update will update game object positions
    // and handle any collisions between objects, whilst the GameLayer
    // update will add and/or remove any queued game objects.
    super.update();
}
```

```
    /**
     * Update the viewport, i.e. scroll the level forward.
     */
    private void updateViewPort() {
        // Don't scroll if we've reached the top of the level
        if( gamePosition + gameEngine.screenHeight > GAME_HEIGHT )
            return;

        // Move this layer forward by the set scroll speed
        gamePosition += GAME_SCROLLSPEED;

        // Update the viewport for this layer and the other layers
        // by the correct amount
        viewPortLayerY -= GAME_SCROLLSPEED;
        gameEngine.getGameLayer("IslandLayer").viewPortLayerY
                -= GAME_SCROLLSPEED *  PARALLAX_VALUE_ISLANDS;
        gameEngine.getGameLayer("CloudLayer").viewPortLayerY
                -= GAME_SCROLLSPEED *  PARALLAX_VALUE_CLOUDS;

        // Update the bottom layer (which contains the water object)
        // so that it scrolls it's graphical realisation
        GameObject backgroundWater =
                gameEngine.getGameObjectFromLayer(
                  "BackgroundWater", "WaterLayer");
        ((ImageAssetTile)backgroundWater.getRealisation(0)).setViewPort(
                gameEngine.screenWidth/2,
                (int)(viewPortLayerY*PARALLAX_VALUE_ISLANDS) );
        backgroundWater.getRealisation(0).update();
    }

    /**
     * Update the player's plane
     */
    private void updatePlayerPlane() {
        // Update the plane movement
        Plane playerPlane = (Plane)getGameObject( "PlayerPlane" );
        if( playerPlane == null )
            return;

        playerPlane.update();

        // Ensure the plane remains within the defined movement region
        if( playerPlane.x
                  < viewPortLayerX + flyRegionOffsetX - flyRegionWidth/2 )
            playerPlane.velocityx += 50.0;
        else if( playerPlane.x
                  > viewPortLayerX + flyRegionOffsetX + flyRegionWidth/2 )
            playerPlane.velocityx -= 50.0;

        if( playerPlane.y
                  < viewPortLayerY + flyRegionOffsetY - flyRegionHeight/2 ) {
            playerPlane.velocityy += 50.0;
        } else if( playerPlane.y
                  > viewPortLayerY + flyRegionOffsetY + flyRegionHeight/2 ) {
            playerPlane.y = viewPortLayerY
                  + flyRegionOffsetY + flyRegionHeight/2;
        }
    }
    /**
     * Update all projectiles
     */
    private void updateProjectiles() {
        GameObjectCollection projectiles
            = getGameObjectCollection( "Projectiles" );
        for( int idx = 0; idx < projectiles.size; idx++ ) {
            projectiles.gameObjects[idx].update();
        }
    }
}
```

## Phase 3: Adding the player controlled plane…

```
package tutorials.topDownScroller;

import game.engine.*;
import game.physics.*;
import game.geometry.*;

import java.awt.event.KeyEvent;
import java.util.*;

/**
 * Base plane object.
 * <P>
 * This object provide core plane functionality, including defining how
 * each plane will look (and the corresponding bounding regions),
 * plane characteristics, and core methods (e.g. the flight model, projectile
 * fire, etc.).
*/

public class Plane extends Body {

    ///////////////////////////////////////////////////////////////////////////
    // Class data: Plane movement                                           //
    ///////////////////////////////////////////////////////////////////////////

    /**
     * Enumerated type defining the possible types of forward acceleration
     */
    public enum ForwardAcceleration { Positive, Negative, None }
    protected ForwardAcceleration
      forwardAccelerationType = ForwardAcceleration.None;

    /**
     * Enumerated type defining the possible types of sideways acceleration
     */
    public enum SidewayAcceleration { Left, Right, None }
    protected SidewayAcceleration
      sidewayAccelerationType = SidewayAcceleration.None;

    /**
     * Plane forward acceleration force, i.e. determining how quickly the ship
     * can accelerate forwards and the maximum forward velocity of the plane
     */
    protected double forwardAcceleration;
    protected double maxFowardVelocity;
     /**
      * Plane backward acceleration force, i.e. determining how quickly the ship
      * can accelerate backwards and the maximum backward velocity of the plane
      */
     protected double backwardAcceleration;
     protected double maxBackwardVelocity;

     /**
      * Plane sideways acceleration force, i.e. determining how quickly the ship
      * can accelerate sideways and the maximum sideways velocity of the plane
      */
     protected double sidewayAcceleration;
     protected double maxSidewaysVelocity;

     /**
      * The plane should gradually reduce its velocity when the player is not
      * instructing it to move in a particular direction. The magnitude of this
      * value determines how quickly the velocity returns to zero.
      */
     protected double dampeningVelocity;


     ///////////////////////////////////////////////////////////////////////////
     // Class data: Weapons, hitpoints                                        //
     ///////////////////////////////////////////////////////////////////////////

     /**
      * An inner class is define to hold the various weapon characteristics.
      * In particular the x, y offset relative to the centre of the plane of
      * the weapon is stored, alongside the weapon's rotation, i.e. the
      * direction that projectiles will be fired in. A fire frequency and last
      * fire time is also maintained to keep track of how often the weapon can
      * fire. Finally the type of projectile is stored, which will be used
      * create a projectile of the appropriate type.
      */
     public class Weapon {
         double xOffset, yOffset;
         double rotation;
         long fireDelay = 1000, lastFireTime = 0;
         String projectileType;

         public Weapon( String type,
             double xOffset, double yOffset, double rotation ) {
             this.projectileType = type;
             this.xOffset = xOffset;
             this.yOffset = yOffset;
             this.rotation = rotation;
         }
     }

     /**
      * Array list of weapons that are attached to this plane
      */
     protected ArrayList<Weapon> planeWeapons = new ArrayList<Weapon>();

     /**
      * Initial number and current number of plane hitpoints. These values
      * are not used within this example, although they would have
      * obvious uses if a more complete game is to be developed
      */
     protected int initialHitPoints;
     protected int currentHitPoints;
     ////////////////////////////////////////////////////////////////////////////
     // Constructors                                                           //
     ////////////////////////////////////////////////////////////////////////////

     public Plane( GameLayer gameLayer ) {
         super( gameLayer );

         // If adding support for multiple planes then the type of plane
         // to be created could be passed into the constructor
         definePlane();
     }

     ////////////////////////////////////////////////////////////////////////////
     // Methods: Plane Definition                                              //
     ////////////////////////////////////////////////////////////////////////////

     /**
      * Define the plane based on the specified plane type
      * Each plane is defined in terms of its graphical realisation,
      * bounding region, maximum directional velocities, accelerative values,
      * initial hitpoints, etc.
      * <P>
      * Only one type of plane is defined at the moment, although this could
      * be extended to incorporate any number of different plane types.
      * <P>
      * Note: Ideally this particular method should load the various plane
      * parameters from a file, as opposed to having hard-coded values. The
      * obvious advantage of being able to load the values from a file is the
      * ability to change plane parameters without a need to
      */
     protected void definePlane( ) {
         // Define the graphical realisation and plane geometry
         setRealisationAndGeometry( "Plane" );
         setGeometry( new Shape[] {
             new Box( 0, 0, 38, 140), new Box(0, 0, 117, 31) } );

         // Setup the movement characteristics of the plane
         maxFowardVelocity = 500.0;
         maxBackwardVelocity  = 500.0;
         maxSidewaysVelocity = 500.0;

         forwardAcceleration = 30.0;
         backwardAcceleration = 30.0;
         sidewayAcceleration = 30.0;

         dampeningVelocity = 10.0;

         // Define the plane hit points and weapon s
         currentHitPoints = initialHitPoints = 1000;

         planeWeapons.add( new Weapon( "Bullet", -50, -90, -0.3 ) );
         planeWeapons.add( new Weapon( "Bullet", -40, -95, 0.0 ) );
         planeWeapons.add( new Weapon( "Bullet", 40, -95, 0.0 ) );
         planeWeapons.add( new Weapon( "Bullet", 50, -90, 0.3 ) );

         // Give the plane an arbitrary mass - the plane should have a
         // defined mass as, by default, all Body objects are assumed
         // to have infinite mass, i.e. they don't move. The actual
         // value that is assigned would make more sense if there are
         // a number of different planes/objects that this plane can
         // bump into.
         setMass(100.0);
     }
     ///////////////////////////////////////////////////////////////////////////
     // Methods: Plane Acceleration                                           //
     ///////////////////////////////////////////////////////////////////////////

     /**
      * Set the plane forward acceleration type to that specified
      */
     public void setForwardAcceleration(
             ForwardAcceleration forwardAcclerationType ) {
         this.forwardAccelerationType = forwardAcclerationType;
     }

     /**
      * Set the plane sideway acceleration type to that specified
      */
     public void setSidewayAcceleration(
             SidewayAcceleration sidewayAccelerationType ) {
         this.sidewayAccelerationType = sidewayAccelerationType;
     }

     ///////////////////////////////////////////////////////////////////////////
     // Methods: Update and render methods                                    //
     ///////////////////////////////////////////////////////////////////////////

     /**
      * Update the plane's acceleration, velocity and location
      */
     @Override
     public void update() {
         // If an AI controlled then we could call the AI update method here.
         updatePlayerInput();
         updatePlaneMovemenet();
     }

     /**
      * Update the forward acceleration, sideway acceleration, etc.
      */
     public void updatePlayerInput() {

         // Set the type of forward acceleration based upon current key state
         if( inputEvent.keyPressed[ KeyEvent.VK_UP ]
                 && !inputEvent.keyPressed[ KeyEvent.VK_DOWN ] )
             setForwardAcceleration( Plane.ForwardAcceleration.Positive );
         else if( inputEvent.keyPressed[ KeyEvent.VK_DOWN ]
                 && !inputEvent.keyPressed[ KeyEvent.VK_UP ] )
             setForwardAcceleration( Plane.ForwardAcceleration.Negative );
         else
             setForwardAcceleration( Plane.ForwardAcceleration.None );

         // Set the type of sideway acceleration based upon current key state
         if( inputEvent.keyPressed[ KeyEvent.VK_RIGHT ]
                 && !inputEvent.keyPressed[ KeyEvent.VK_LEFT ] )
             setSidewayAcceleration( Plane.SidewayAcceleration.Right );
         else if( inputEvent.keyPressed[ KeyEvent.VK_LEFT ]
                 && !inputEvent.keyPressed[ KeyEvent.VK_RIGHT ] )
             setSidewayAcceleration( Plane.SidewayAcceleration.Left );
         else
             setSidewayAcceleration( Plane.SidewayAcceleration.None );

         // Consider if we should fire
         if( inputEvent.keyPressed[ KeyEvent.VK_SPACE ] == true )
             considerFiring();
    }
```

```java
/**
 * Update the plane's acceleration, velocity and location
 */
private void updatePlaneMovemenet() {
    // Determine the plane's current forward acceleration
    switch( forwardAccelerationType ) {
        case Positive:
            velocityy -= forwardAcceleration; // Negative as moving 'up'
            if( velocityy < -maxFowardVelocity )
               velocityy = -maxFowardVelocity;
            break;
        case Negative:
            velocityy += backwardAcceleration; // Positive as moving 'down'
            if( velocityy > backwardAcceleration )
              velocityy = backwardAcceleration;
            break;
        case None:
            if( Math.abs(velocityy) < dampeningVelocity )
                velocityy = 0.0;
            else
                velocityy += (velocityy > 0.0 ?
                    -dampeningVelocity : dampeningVelocity );
            break;
    }

    // Determine the plane's current sideway acceleration
    switch( sidewayAccelerationType ) {
        case Right:
            velocityx += sidewayAcceleration;
            if( velocityx > maxSidewaysVelocity )
              velocityx = maxSidewaysVelocity;
            break;
        case Left:
            velocityx -= sidewayAcceleration;
            if( velocityx < -maxSidewaysVelocity )
              velocityx = -maxSidewaysVelocity;
            break;
        case None:
            if( Math.abs(velocityx) < dampeningVelocity )
                velocityx = 0.0;
            else
                velocityx += (velocityx > 0.0 ?
                    -dampeningVelocity : dampeningVelocity );
            break;
    }

    // By default all game objects can have an arbitrary rotation which
    // will be updated following collisions, etc. To keep the plane
    // pointing upwards, the rotation is continually corrected if needed.
    if( rotation != 0.0 ) {
        angularVelocity = 0.0;
        if( Math.abs( rotation ) < 0.01 )
            rotation = 0.0;
        else
            rotation -= 0.01 * Math.signum(rotation);
    }
}
```

```
    /**
     * Consider if this plane can fire, and if so created projectiles
     */
    protected void considerFiring() {
        long currentTime = System.nanoTime() / 1000000;
        for( Weapon weapon : planeWeapons ) {
            // If the weapon is enabled and the fire delay has elapsed then
            // create a new projectile of the appropriate type
            if( weapon.lastFireTime + weapon.fireDelay < currentTime ) {
                // Set the last fire time to the current time
                weapon.lastFireTime = currentTime;

                // Add a new projectile to the determined weapon fire location.
                // The projectile is defined in terms of the relevant weapon
                // parameters.
                gameLayer.queueGameObjectToAdd(
                        new Projectile( weapon.projectileType,
                        x + weapon.xOffset, y + weapon.yOffset,
                        weapon.rotation, gameLayer ), "Projectiles" );
            }
        }
    }
}
```

## Phase 4: Finishing the game by adding projectiles…

```
package tutorials.topDownScroller;

import game.assets.ImageAssetSequence;
import game.engine.GameLayer;

import game.physics.*;

/**
 * Projectile
 * <P>
 * A number of different types of projectile can be supported within this
 * class. Each projectile will travel and disappear once the travel
 * range has been exceeded. It is assumed that hit projectile detection
 * will be dealt with within the parent GameLayer.
*/

public class Projectile extends Body {

    ///////////////////////////////////////////////////////////////////////////
    // Class data                                                            //
    ///////////////////////////////////////////////////////////////////////////

    /**
     * Define the current state of the projectile, i.e. is it moving,
     * has hit something or extended its maximum travel time and is
     * disappearing.
     */
    protected enum ProjectileState { Move, Hit, Disappear }
    protected ProjectileState projectileState = ProjectileState.Move;

    /**
     * Amount of damage which will result should this projectile hit
     */
    protected double projectileDamage;
    /**
     * Define the projectiles acceleration, maximum travel speed and
     * maximum travel time (in ms)
     */
    private double projectileAcceleration;
    private double projectileMaxSpeed;
    private long projectileMaxTime;

    /**
     * Define the current projectile trigger time (this can be multifaced,
     * i.e. the time can be since the projectile was created, or since the
     * hit animation started playing, etc.
     */
    private long projectileTriggerTime;

    /**
     * Define the graphicals assets to be used whenever this projectile hits
     * something or exceeds its travel time
     */
    private String hitAsset;
    private String disappearAsset;

    ///////////////////////////////////////////////////////////////////////////
    // Constructors                                                          //
    ///////////////////////////////////////////////////////////////////////////

    /**
     * Create a new projectile of the specified type with the specified
     * parameters.
     *
     * @param  projectileType Type of projectile to create
     * @param  x x location at which the projectile should be created
     * @param  y y location at which the projectile should be created
     * @param  rotation Rotation of the projectile
     * @param gameLayer GameLayer instance to which the projectile belongs
     */
    public Projectile( String projectileType,
            double x, double y, double rotation, GameLayer gameLayer ) {
        super( gameLayer, x, y );

        this.rotation = rotation;

        defineWeaponType( projectileType );
    }


    ///////////////////////////////////////////////////////////////////////////
    // Methods: Projectile setup and state change                            //
    ///////////////////////////////////////////////////////////////////////////

    /**
     * Define the characteristics of this projectile. Note, only one type
     * of projectile is currently defined within this class, although
     * others could be easily added.
     */
    private void defineWeaponType( String weaponType ) {
        if( weaponType.equals("Bullet")) {
            // Define the projectile speed, acceleration and damage
            projectileMaxSpeed = 1000.0;
            projectileAcceleration = 100.0;
            projectileDamage = 100.0;
            // Define the projectile travel time and record the current time
            projectileMaxTime = 2000;
            projectileTriggerTime = System.nanoTime();

            // Set the projectile's realisation and record the assets to
            // be used for hit and disappear
            setRealisationAndGeometry( "EnergyBall" );
            hitAsset = "EnergyBallExplode";
            disappearAsset = "EnergyBallDisappear";

            // Give this projectile a mass and restiution. This does not
            // make a lot of sense in the current setup, although it would
            // if enemy planes, etc. were added, e.g. enable the projectile
            // to 'thump' into whatever is it, or deflect, etc.
            setMass( 5.0 );
            restitution = 1.0;
        }
    }

    /**
     * Display the projectile hit animation.
     */
    public void triggerProjectileHit() {
        projectileState = ProjectileState.Hit;

        setRealisation( hitAsset );

        // It is assumed that once hit a projectile stops (i.e. explodes
        // and cannot interact other objects). This is not a necessary
        // assumption and could be extended to support projectiles that
        // can bounce off a number of different objects, etc.
        velocityx = 0.0;
        velocityy = 0.0;
        canIntersectOtherGraphicalObjects = false;

        projectileTriggerTime = System.nanoTime();
    }

    /**
     * Display the projectile disappearing animation, triggered
     * once the projectile goes beyond the maximum travel distance
     */
    private void triggerProjectDisappear() {
        projectileState = ProjectileState.Disappear;

        setRealisation( disappearAsset );

        canIntersectOtherGraphicalObjects = false;

        projectileTriggerTime = System.nanoTime();
    }
    ///////////////////////////////////////////////////////////////////////////
    // Methods: Update methods                                               //
    ///////////////////////////////////////////////////////////////////////////

    /**
     * Update the projectile, triggering the disappearing animation if needed
     */
    public void update() {
        // If the current projectile realisation is an image animation, e.g.
        // hit/disappear animations, then update the animation frame
        if( getRealisation(0) instanceof ImageAssetSequence )
            getRealisation(0).update();

        // If the projectile is in the hit or disappear state, then remove
        // the object once the hit/disppear animation has finished playing
        if( projectileState != ProjectileState.Move ) {
            if( (System.nanoTime() - projectileTriggerTime)/1000000L >
             ((ImageAssetSequence)getRealisation(0)).getAnimationPeriod() )
                gameLayer.queueGameObjectToRemove(this);
        }

        // If the projectile has not hit, then update velocity based upon
        // the current projectile rotation and maximum speed
        if( projectileState != ProjectileState.Hit ) {
            if( velocityx*velocityx + velocityy*velocityy
                    < projectileMaxSpeed*projectileMaxSpeed ) {
                velocityx += projectileAcceleration * Math.sin(rotation);
                velocityy -= projectileAcceleration * Math.cos(rotation);
            }

            // If the maximum movemement time has been exceeded, then
            // trigger the project disappear
            if( projectileState == ProjectileState.Move )
                if( (System.nanoTime() - projectileTriggerTime)/1000000L >
                        projectileMaxTime )
                    triggerProjectDisappear();
        }
    }
}
```